前言

大家好，我是博哥，近4年的时候时间我坚持在微信公众号“**数据结构和算法**”中写算法题，截止到目前已经在公众号上发布了600多道算法题。现在我把部分算法题整理成PDF的格式，这样方便大家阅读，有的可能时间比较久了，这里没有全部列出来。

当然学习的脚步不能停止，我还会继续在公众号上不断的输出各种算法题。如果大家不小心把这个文档搞丢了也没关系，可以到我微信公众号中回复“pdf”即可获取下载地址，我把文档放到了百度网盘上了。这个文档也会不停的更新，因为我公众号会不停的写算法题。关于我微信公众号大家可以扫描下方二维码关注。如果觉得不错的，可以**相互转发**，发给自己的小伙伴们，我们大家一起学习。感谢大家的支持。

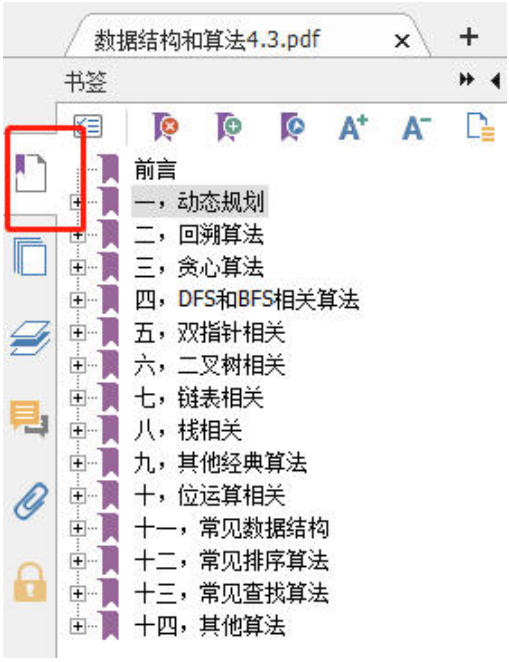


V4.7

注意：文章之前都是图文结合的，后来有了视频，视频没法在pdf中播放，如果想看视频可以直接点击，会跳转到到我公众号中，然后就可以观看了。

牛客网
NOWCODER

点击右边可以查看目录



目录

前言 10

目录

一，动态规划 11

598，动态规划解目标和 11

588，动态规划解分割等和子集 15

587，最大的以1为边界的正方形 22

576，动态规划解最长公共子串 28

573，动态规划解单词拆分 31

572，动态规划解分割回文串 III 36

570，动态规划解回文串分割 IV 43

568，动态规划解最后一块石头的重量 II 46

559，动态规划解不相交的线 50

557，动态规划解戳气球 54

553，动态规划解分割回文串 II 60

552，动态规划解统计全为1的正方形子矩阵 65

548，动态规划解最长的斐波那契子序列的长度 68

543，剑指 Offer-动态规划解礼物的最大价值 73

540，动态规划和中心扩散法解回文子串 78

530，动态规划解最大正方形 86

529，动态规划解最长回文子序列 92

522，俄罗斯套娃信封问题 96

517，最长回文子串的3种解决方式 101

515，动态规划解买卖股票的最佳时机含手续费 107

493，动态规划解打家劫舍III 111

492，动态规划和贪心算法解买卖股票的最佳时机II 114

490，动态规划和双指针解买卖股票的最佳时机 119

486，动态规划解最大子序和 127

477，动态规划解按摩师的最长预约时间 130

465. 递归和动态规划解三角形最小路径和 133

430，剑指 Offer-动态规划求正则表达式匹配 138

423，动态规划和递归解最小路径和 148

413，动态规划求最长上升子序列 152

411，动态规划和递归求不同路径 II 156
409，动态规划求不同路径 161
407，动态规划和滑动窗口解决最长重复子数组 167
395，动态规划解通配符匹配问题 173
376，动态规划之编辑距离 179
370，最长公共子串和子序列 185
二，回溯算法 192
594，回溯算法解含有重复数字的全排列 II 192
593，经典回溯算法题-全排列 197
590，回溯算法解正方形数组的数目 203
575，回溯算法和DFS解单词拆分 II 207
551，回溯算法解分割回文串 213
537，剑指 Offer-字符串的排列 219
520，回溯算法解火柴拼正方形 222
498，回溯算法解活字印刷 226
491，回溯算法解将数组拆分成斐波那契序列 230
478，回溯算法解单词搜索 234
451，回溯和位运算解子集 238
450，什么叫回溯算法，一看就会，一写就废 244
448，组合的几种解决方式 255
446，回溯算法解黄金矿工问题 259
442，剑指 Offer-回溯算法解二叉树中和为某一值的路径 263
420，剑指 Offer-回溯算法解矩阵中的路径 268
391，回溯算法求组合问题 273
三，贪心算法 279
600，贪心算法解救生艇问题 279
516，贪心算法解按要求补齐数组 281
505，分发糖果（贪心算法解决） 284
501，贪心算法解分发饼干 289
489，柠檬水找零 292
四，DFS和BFS相关算法 295
589，DFS和BFS解从根到叶的二进制数之和 295
586，BFS和DFS解层数最深叶子节点的和 299

580，BFS和DFS解二叉树的堂兄弟节点 302
574，DFS和BFS解单词拆分 306
566，DFS解目标和问题 313
532，BFS解打开转盘锁 316
531，BFS和动态规划解完全平方数 321
507，BFS和DFS解二叉树的层序遍历 II 326
473，BFS解单词接龙 329
470，DFS和BFS解合并二叉树 334
455，DFS和BFS解被围绕的区域 338
453，DFS和BFS解求根到叶子节点数字之和 344
445，BFS和DFS两种方式解岛屿数量 349
422，剑指 Offer-使用DFS和BFS解机器人的运动范围 354
417，BFS和DFS两种方式求岛屿的最大面积 358
五，双指针相关 362
597，双指针解验证回文字符串 II 362
549，滑动窗口解可获得的最大点数 364
542，滑动窗口解最小覆盖子串 367
539，双指针解删除有序数组中的重复项 371
538，剑指 Offer-和为s的连续正数序列 374
527，两个数组的交集 II 378
514，双指针解替换后的最长重复字符 381
497，双指针验证回文串 388
466. 使用快慢指针把有序链表转换二叉搜索树 391
449，快慢指针解决环形链表 394
447，双指针解旋转链表 400
398，双指针求无重复字符的最长子串 404
397，双指针求接雨水问题 410
396，双指针求盛最多水的容器 417
六，二叉树相关 422
591，二叉树的垂序遍历 422
582，DFS解二叉树剪枝 427
564，二叉树最大宽度 432
563，N叉树的最大深度 438

561，二叉搜索树中第K小的元素 443
547，叶子相似的树 447
545，二叉搜索树的范围和 453
544，剑指 Offer-平衡二叉树 457
510，将有序数组转换为二叉搜索树 462
503，二叉搜索树中的众数 465
488，二叉树的Morris中序和前序遍历 470
485，递归和非递归两种方式解相同的树 482
483，完全二叉树的节点个数 486
474，翻转二叉树的多种解决方式 491
471，二叉搜索树中的插入操作 496
464. BFS和DFS解二叉树的所有路径 501
458，填充每个节点的下一个右侧节点指针 II 508
457，二叉搜索树的最近公共祖先 514
456，解二叉树的右视图的两种方式 518
444，二叉树的序列化与反序列化 523
441，剑指 Offer-二叉搜索树的后序遍历序列 527
440，剑指 Offer-从上到下打印二叉树 II 532
439，剑指 Offer-从上到下打印二叉树 535
435，剑指 Offer-对称的二叉树 538
434，剑指 Offer-二叉树的镜像 542
433，剑指 Offer-树的子结构 548
414，剑指 Offer-重建二叉树 552
403，验证二叉搜索树 555
401，删除二叉搜索树中的节点 559
400，二叉树的锯齿形层次遍历 563
399，从前序与中序遍历序列构造二叉树 567
388，先序遍历构造二叉树 574
387，二叉树中的最大路径和 584
375，在每个树行中找最大值 590
374，二叉树的最小深度 595
372，二叉树的最近公共祖先 598
367，二叉树的最大深度 602

七，链表相关 608
596，删除排序链表中的重复元素 II 608
595，删除排序链表中的重复元素 611
554，反转链表 II 614
502，分隔链表的解决方式 618
463. 判断回文链表的3种方式 621
462. 找出两个链表的第一个公共节点 626
461. 两两交换链表中的节点 634
460. 快慢指针解环形链表 II 639
459. 删除链表的倒数第N个节点的3种方式 644
432，剑指 Offer-反转链表的3种方式 648
431，剑指 Offer-链表中倒数第k个节点 654
429，剑指 Offer-删除链表的节点 659
410，剑指 Offer-从尾到头打印链表 664
386，链表中的下一个更大节点 668
381，合并两个有序链表（易） 677
八，栈相关 681
528，使用栈解基本计算器 II 681
526，删除字符串中的所有相邻重复项 684
523，单调栈解下一个更大元素 II 688
519，单调栈解下一个更大元素 I 690
508，使用栈来判断有效的括号 693
500，验证栈序列 697
438，剑指 Offer-栈的压入、弹出序列 702
437，剑指 Offer-包含min函数的栈 704
416，剑指 Offer-用两个栈实现队列 712
九，其他经典算法 714
Manacher(马拉车)算法 714
426，什么是递归，通过这篇文章，让你彻底搞懂递归 723
394，经典的八皇后问题和N皇后问题 737
389，两个超级大数相加 749
371，背包问题系列之-基础背包问题 753
366，约瑟夫环 764

362，汉诺塔 775
356，青蛙跳台阶相关问题 780
十，位运算相关 784
592，位运算解颠倒二进制位 784
565，多种方式解2的幂 788
560，位运算解只出现一次的数字 II 792
556，位运算解形成两个异或相等数组的三元组数目 799
534，剑指 Offer-0～n-1中缺失的数字 802
513，汉明距离 805
512，反转二进制位 807
499，位运算解只出现一次的数字III 811
495，位运算等多种方式解找不同 815
494，位运算解只出现一次的数字 819
476，根据数字二进制下1的数目排序 822
469，位运算求最小的2的n次方 826
425，剑指 Offer-二进制中1的个数 830
383，不使用“+”,“-”,“×”,“÷”实现四则运算 836
361，交替位二进制数 845
364，位1的个数系列（一） 849
385，位1的个数系列（二） 854
402，位1的个数系列（三） 860
357，交换两个数字的值 865
十一，常见数据结构 868
348，数据结构-1,数组 868
352，数据结构-2,链表 871
359，数据结构-3,队列 879
363，数据结构-4,栈 884
368，数据结构-5,散列表 888
373，数据结构-6,树 896
378，数据结构-7,堆 905
十二，常见排序算法 915
101，排序-冒泡排序 915
102，排序-选择排序 918

103，排序-插入排序 920
104，排序-快速排序 923
105，排序-归并排序 929
106，排序-堆排序 933
107，排序-桶排序 937
108，排序-基数排序 939
109，排序-希尔排序 944
110，排序-计数排序 947
111，排序-位图排序 949
112，排序-其他排序 951
十三，常见查找算法 954
201，查找-顺序查找 954
202，查找-二分法查找 955
203，查找-插值查找 956
204，查找-斐波那契查找 957
205，查找-分块查找 960
206，查找-哈希查找 961
207，查找-其他查找 963
十四，其他算法 964
601，下一个排列 964
599，统计全 1 子矩形 968
585，最大升序子数组和 974
584，前缀和解和为K的子数组 977
583，字符串中的最大奇数 981
581，所有蚂蚁掉下来前的最后一刻 984
579，摩尔投票算法解主要元素 989
578，计数质数 992
577，数组中的最长连续子序列 994
571，山脉数组的峰顶索引 997
569，多种方式解4的幂 1000
567，最后一块石头的重量 1004
562，数组中的最长山脉 1007
558，最长回文串 1011

550，旋转图像 1014
546，砖墙，哈希表解决 1017
541，字符串压缩，视频演示 1021
536，剑指 Offer-构建乘积数组 1024
535，剑指 Offer-扑克牌中的顺子 1027
533，剑指 Offer-最小的k个数 1030
525，最富有客户的资产总量 1035
524，爱生气的书店老板 1037
521，滑动窗口解最大连续1的个数 III 1040
518，托普利茨矩阵 1045
511，独一无二的出现次数 1048
509，数组中的第K个最大元素 1050
506，无重叠区间 1053
504，旋转数组的3种解决方式 1056
496，字符串中的第一个唯一字符 1064
487，重构字符串 1067
484，打家劫舍 II 1070
482，上升下降字符串 1074
481，用最少数量的箭引爆气球 1078
480，移动零 1083
479，递归方式解打家劫舍 1088
475，有效的山脉数组 1092
472，插入区间 1096
468，提莫攻击的两种解决方式 1099
467. 递归和非递归解路径总和问题 1102
454，字母异位词分组 1107
452，跳跃游戏 1110
443，滑动窗口最大值 1113
436，剑指 Offer-顺时针打印矩阵 1120
428，剑指 Offer-打印从1到最大的n位数 1124
427，剑指 Offer-数值的整数次方 1126
424，剑指 Offer-剪绳子 1129
421，在排序数组中查找元素的第一个和最后一个位置 1135

419，剑指 Offer-旋转数组的最小数字 1139
418，剑指 Offer-斐波那契数列 1142
415，最佳观光组合 1145
412，判断子序列 1147
408，剑指 Offer-替换空格 1153
406，剑指 Offer-二维数组中的查找 1155
405，换酒问题 1159
404，剑指 Offer-数组中重复的数字 1163
393，括号生成 1167
392，检查数组对是否可以被 k 整除 1173
390，长度最小的子数组 1178
384，整数反转 1186
382，每日温度的5种解题思路 1189
380，缺失的第一个正数（中） 1198
379，柱状图中最大的矩形（难） 1205
377，调整数组顺序使奇数位于偶数前面 1218
369，整数替换 1223
365，消除游戏 1226
360，等差数列划分 1231
358，移掉K位数字 1234
355，两数相加 II 1237
354，字典序排数 1242
353，打乱数组 1244
351，最少移动次数使数组元素相等 II 1246
350，有序矩阵中第K小的元素 1248
349，组合总和 IV 1250
347，猜数字大小 II 1252
346，查找和最小的K对数字 1256
345，超级次方 1258
344，最大整除子集 1260
343，水壶问题 1262
342，计算各个位数不同的数字个数 1264

598，动态规划解目标和

原创 博哥 数据结构和算法 1周前

问题描述

来源：LeetCode第494题
难度：中等

给你一个整数数组nums和一个整数target 。

向数组中的每个整数前添加'+'或'-'，然后串联起所有整数，可以构造一个表达式：

- 例如，nums=[2,1]，可以在2之前添加'+'，在1之前添加'-'，然后串联起来得到表达式"+2-1"。

返回可以通过上述方法构造的、运算结果等于target的不同表达式的数目。

示例 1：

> **输入**：nums = [1,1,1,1,1], target = 3
> **输出**：5
> **解释**：一共有 5 种方法让最终目标和为 3 。
> -1 + 1 + 1 + 1 + 1 = 3
> +1 - 1 + 1 + 1 + 1 = 3
> +1 + 1 - 1 + 1 + 1 = 3
> +1 + 1 + 1 - 1 + 1 = 3
> +1 + 1 + 1 + 1 - 1 = 3

示例 2：

> **输入**：nums = [1], target = 1
> **输出**：1

提示：

- 1 <= nums.length <= 20
- 0 <= nums[i] <= 1000
- 0 <= sum(nums[i]) <= 1000
- -1000 <= target <= 1000

动态规划解决

这题之前讲过，具体可以看下566，DFS解目标和问题，由于当时时间仓促，只介绍了DFS的解决方式，其实这道题还有另外一种解决方式，就是使用动态规划来解决。

我们假设在一些数字前添加“+”，这些数字的和是plusSum。剩下的数字前添加“-”，这些数字的和是minusSum。我们要求的是
plusSum-minusSum=target ①
的方案数目。

假设数组中所有元素的和是sum。那么我们可以得到
plusSum+minusSum=sum ②

由公式①和公式②我们可以得到
minusSum*2=sum-target;

我们可以看到如果要让上面等式成立，sum-target必须是偶数。

- 也就是说如果sum-target不是偶数，无论怎么添加符号，表达式的值都不可能是target，直接返回0。
- 如果sum-target是偶数，我们只需要找出一些数字让他们的和等于minusSum，也就是(sum-target)/2的方案数。

通过上面的分析，这题就变成了从数组中选择一些元素，让他们的和等于(sum-target)/2的方案数。这和0-1背包非常像，具体可以看下371，背包问题系列之-基础背包问题

我们定义dp[i][j]表示从数组前i个元素中选取一些数字，让他们的和等于j的方案数。很明显我们最终只需要返回dp[length][(sum-target)/2]即可。

其中dp[0][0]=1，表示选择0个元素让他们的和等于0，只有一种方案。

遍历到当前数字num的时候，如果当前数字num大于j，那么我们是不能选择的，所以dp[i][j]=dp[i-1][j]。他表示的意思就是前i个元素中不选择第i个元素，而选择前i个元素中其他的一些数字，让他们的和等于j的方案数。

如果当前数字num小于或等于j，我们可以选择也可以不选择。如果不选择就是dp[i][j]=dp[i-1][j]，如果选择就是dp[i][j]=dp[i-1][j-num]；那么总的方案数就是

dp[i][j]=dp[i-1][j]+dp[i-1][j-num]

递推公式如下

```
if (j >= num) {//不选num和选num
    dp[i][j] = dp[i - 1][j] + dp[i - 1][j - num];
} else {//不能选择num
    dp[i][j] = dp[i - 1][j];
}
```

通过上面的分析，我们再来看下最终代码

```
public int findTargetSumWays(int[] nums, int target) {
  int length = nums.length;
  //求数组中所有数字的和
  int sum = 0;
  for (int num : nums)
    sum += num;

  //如果所有数字的和小于target，或者sum - target是奇数，
  //说明无论怎么添加符号，表达式的值都不可能是target
  if (sum < target || ((sum - target) & 1) != 0) {
    return 0;
  }
  //我们要找到一些元素让他们的和等于capacity的方案数即可。
  int capacity = (sum - target) >> 1;
  //dp[i][j]表示在数组nums的前i元素中选择一些元素，
  //使得选择的元素之和等于j的方案数
  int dp[][]= new int[length + 1][capacity + 1];
  //边界条件
  dp[0][0] = 1;
  for (int i = 1; i <= length; i++) {
    for (int j = 0; j <= capacity; j++) {
      //下面是地推公式
      if (j >= nums[i - 1]) {//不选第i个和选第i个元素
        dp[i][j] = dp[i - 1][j] + dp[i-1][j - nums[i - 1]];
      } else {//不能选择第i个元素
        dp[i][j] = dp[i - 1][j];
      }
    }
  }
  //从数组前length个（也就是全部）元素中选择一些元素，让他们的
  //和等于capacity的方案数。
  return dp[length][capacity];
}
```

时间复杂度：O（n*capacity），n是数组的长度。

空间复杂度：O（n*capacity），capacity是(sum-target)/2。

代码优化

我们看到上面二维数组计算的时候，当前那一行的值只和上一行的有关，所以我们可以改成一维数组，这里要注意嵌套中的第二个for循环要倒叙遍历。因为改成一维数组之后，数组后面的值要依赖前面的（改变之前的），如果从前往后遍历，前面的值被修改了，会

导致后面的运行结果错误。如果倒叙，也就是先计算数组后面的值，因为前面的还没有计算，也就是还没有被修改，所以不会导致结果错误。来看下代码

```
public int findTargetSumWays(int[] nums, int target) {
    int length = nums.length;
    //求数组中所有数字的和
    int sum = 0;
    for (int num : nums)
        sum += num;

    //如果所有数字的和小于target，或者sum - target是奇数，
    //说明无论怎么添加符号，表达式的值都不可能是target
    if (sum < target || ((sum - target) & 1) != 0) {
        return 0;
    }
    //我们要找到一些元素让他们的和等于capacity的方案数即可。
    int capacity = (sum - target) >> 1;
    int dp[] = new int[capacity + 1];
    //边界条件
    dp[0] = 1;
    for (int i = 1; i <= length; i++) {
        //注意，这里要倒叙
        for (int j = capacity; j >= 0; j--) {
            /*
            地推公式
            if (j >= nums[i - 1]) {
                dp[j] = dp[j] + dp[j - nums[i - 1]];
            } else {
                dp[j] = dp[j];
            }
             */
            //上面的代码合并之后的
            if (j >= nums[i - 1]) {
                dp[j] += dp[j - nums[i - 1]];
            }
        }
    }
    return dp[capacity];
}
```

时间复杂度：O（n*capacity）。

空间复杂度：O（capacity）

往期推荐

- 588，动态规划解分割等和子集
- 573，动态规划解单词拆分
- 568，动态规划解最后一块石头的重量 II
- 557，动态规划解戳气球

588，动态规划解分割等和子集

原创 博哥 数据结构和算法 今天

问题描述

来源：LeetCode第416题
难度：中等

给你一个只包含正整数的非空数组nums。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

示例 1：

> **输入**：nums = [1,5,11,5]
> **输出**：true
> **解释**：数组可以分割成 [1, 5, 5] 和 [11] 。

示例 2：

> **输入**：nums = [1,2,3,5]
> **输出**：false
> **解释**：数组不能分割成两个元素和相等的子集。

提示：

- 1 <= nums.length <= 200
- 1 <= nums[i] <= 100

动态规划解决

前面我们讲过520，回溯算法解火柴拼正方形，和这题类似，具体可以看下。第520题可以认为是把数组分隔成4个元素和相等的子集，而这题是把数组分隔成2个元素和相等的子集。如果数据量比较少的话，第520题的答案稍微修改一下就是这题的答案了，但如果数据量大的话就会超时。所以这题不能使用回溯算法来解决，我们可以使用动态规划。

这题判断把数组分成两份，这两份的元素和是否相等。首先我们需要计算数组中所有元素的和sum，然后判断sum是否是偶数：
如果不是偶数，说明不可能分割成完全相等的两份，直接返回false。

如果是偶数，我们只需要判断是否存在一些元素的和等于sum/2，如果等于sum/2，那么剩下的肯定也等于sum/2，说明我们可以把数组分为元素和相等的两部分。

那么这个时候问题就很明朗了，假设sum/2是一个背包的容量，我们只需要找出一些元素把他放到背包中，如果背包中元素的最大和等于sum/2，说明我们可以把数组分成完成相等的两份。这不就是经典的0-1背包问题吗。之前也讲过371，背包问题系列之-基础背包问题，具体可以看下，这里就不在重复介绍。我们在来找一下他的递推公式，定义dp[i][j]表示把第i个物品放到容量为j的背包中所获得的的最大值。

第i个物品的值是nums[i-1]：
如果nums[i-1]>j，说明背包容量不够，第i件物品放不进去，所以我们不能选择第i个物品，那么
dp[i][j]=dp[i-1][j];

如果nums[i-1]<=j，说明可以把第j个物品放到背包中，我们可以选择放也可以选择不放，取最大值即可，如果放就会占用一部分背包容量，最大价值是
dp[i][j]=dp[i-1][j-nums[i-1]]+nums[i-1]

如果不放
dp[i][j]=dp[i-1][j];
取两者的最大值

最终递推公式如下

```
if (j >= nums[i - 1]) {
   dp[i][j] = Math.max(dp[i - 1][j], dp[i-1][j - nums[i - 1]] + nums[i - 1]);
} else {
   dp[i][j] = dp[i - 1][j];
}
```

我们来看下最终代码

```
public boolean canPartition(int[] nums) {
   //计算数组中所有元素的和
   int sum = 0;
   for (int num : nums)
      sum += num;
   //如果sum是奇数，说明数组不可能分成完全相等的两份
   if ((sum & 1) == 1)
      return false;
   //sum除以2
   int target = sum >> 1;
```

```
    int length = nums.length;
    int[][] dp = new int[length + 1][target+1];
    for (int i = 1; i <= length; i++) {
      for (int j = 1; j <= target; j++) {
        //下面是递推公式
        if (j >= nums[i - 1]) {
          dp[i][j] = Math.max(dp[i - 1][j], dp[i-1][j - nums[i - 1]] + nums[i - 1]);
        } else {
          dp[i][j] = dp[i - 1][j];
        }
      }
    }
    //判断背包最大是否能存放和为target的元素
    return dp[length][target] == target;
}
```

我们还可以这样写，二维数组dp是boolean类型，dp[i][j]表示数组中前i个元素的和是否可以组成和为j，很明显dp[0][0]=true，表示前0个元素（也就是没有元素）可以组成和为0。代码如下

```
public boolean canPartition(int[] nums) {
    //计算数组中所有元素的和
    int sum = 0;
    for (int num : nums)
      sum += num;
    //如果sum是奇数，说明数组不可能分成完全相等的两份
    if ((sum & 1) == 1)
      return false;
    //sum除以2
    int target = sum >> 1;
    int length = nums.length;
    boolean[][]dp = new boolean[length + 1][target+1];
    dp[0][0] = true;//base case
    for (int i = 1; i <= length; i++) {
      for (int j = 1; j <= target; j++) {
        //递推公式
        if (j >= nums[i - 1]) {
          dp[i][j] = (dp[i - 1][j] || dp[i-1][j - nums[i - 1]]);
        } else {
          dp[i][j] = dp[i - 1][j];
        }
      }
    }
    return dp[length][target];
}
```

我们看到上面二维数组计算的时候当前值只和上面一行有关，所以我们可以把它改成一维的，注意第二个for循环要倒叙，否则会把前面的值给覆盖掉导致结果错误，仔细看一下

dp[j] = (dp[j] || dp[j - nums[i - 1]]);

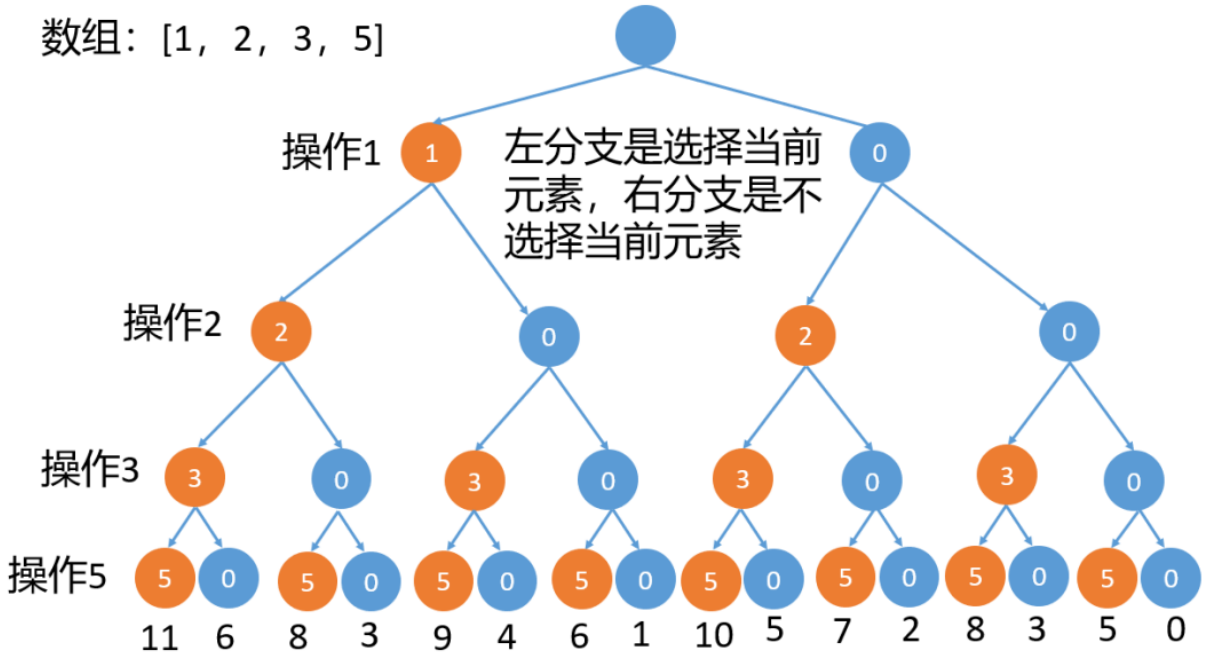
就明白了，相当于同一行后面的值依赖前面的，如果不是倒叙，前面的值被修改了，在计算后面的就会导致错误。我们来看下代码。

```
public boolean canPartition(int[] nums) {
    //计算数组中所有元素的和
    int sum = 0;
    for (int num : nums)
      sum += num;
    //如果sum是奇数，说明数组不可能分成完全相等的两份
    if ((sum & 1) == 1)
      return false;
    //sum除以2
    int target = sum >> 1;
    int length = nums.length;
    boolean[] dp = new boolean[target + 1];
```

```
    dp[0] = true;//base case
    for (int i = 1; i <= length; i++) {
        //注意这里j要倒叙
        for (int j = target; j >= 1; j--) {
            //递推公式
            if (j >= nums[i - 1]) {
                dp[j] = (dp[j] || dp[j - nums[i - 1]]);
            }
            //else {//这里省略
            //  dp[j] = dp[j];
            //}
        }
    }
    return dp[target];
}
```

DFS解决

每种元素可以选择也可以不选择，只需要判断他所有的可能组合中，元素和是否有等于sum/2的，我们可以把它看做是一棵二叉树，左子节点表示选择当前元素，右子节点表示不选择当前元素，如下图所示，橙色节点表示选择当前元素，蓝色表示不选择。



我们来看下代码

```
public boolean canPartition(int[] nums) {
    //计算数组中所有元素的和
    int sum = 0;
    for (int num : nums)
        sum += num;
    //如果sum是奇数，说明数组不可能分成完全相等的两份
    if ((sum & 1) == 1)
        return false;
    return dfs(nums, sum >> 1, 0);
}


private boolean dfs(int[] nums, int target, int index) {
    //targe等于0，说明存在一些元素的和等于sum/2，直接返回true
    if (target == 0)
        return true;
    //如果数组元素都找完了，或者target小于0，直接返回false
```

```
    if (index == nums.length || target < 0)
        return false;
    //选择当前元素和不选择当前元素两种情况
    return dfs(nums, target - nums[index], index + 1)
            || dfs(nums, target, index + 1);
}
```

但很遗憾的是，因为计算量太大，会导致运行超时，我们可以优化一下，来看下代码

```
public boolean canPartition(int[] nums) {
    //计算数组中所有元素的和
    int sum = 0;
    for (int num : nums)
        sum += num;
    //如果sum是奇数，说明数组不可能分成完全相等的两份
    if ((sum & 1) == 1)
        return false;
    //sum除以2
    int target = sum >> 1;
    Boolean[][] map = new Boolean[nums.length][target + 1];
    return dfs(nums, 0, target, map);
}

private boolean dfs(int[] nums, int index, int target, Boolean[][] map) {
    //targe等于0，说明存在一些元素的和等于sum/2，直接返回true
    if (target == 0)
        return true;
    //如果数组元素都找完了，或者target小于0，直接返回false
    if (index == nums.length || target < 0)
        return false;
    //从map中取
    if (map[index][target] != null)
        return map[index][target];
    //选择当前元素
    boolean select = dfs(nums, index + 1, target - nums[index], map);
    //不择当前元素
    boolean unSelect = dfs(nums, index + 1, target, map);
    //只要有一个为true，就返回true，否则返回false
    if (select || unSelect) {
        map[index][target] = true;
        return true;
    }
    map[index][target] = false;
    return false;
}
```

位运算解决

这里能使用位运算，关键在于题中的一些限制条件，比如

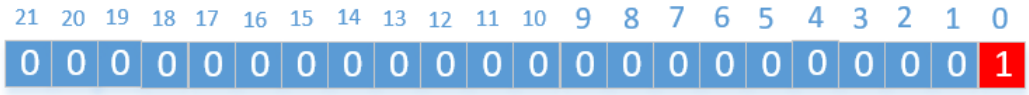
- 正整数的非空数组，
- 每个数组中的元素不会超过 100，
- 数组的大小不会超过 200等。

原理很简单，我们只需要申请一个大小为sum+1的数组bits[sum+1]，数组中的数字只能是0和1，我们可以把它想象为一个**很长的二进制位**，因为int和long类型太短了，我们这里使用的是数组。然后每遍历数组中的一个元素比如m，就把二进

制位往左移m位然后在和原来的二进制位进行**或运算**。最后判断bits[sum/2]是否是1，如果是1就返回true。文字叙述不是很直接，我们就以示例1为例来画个图看一下

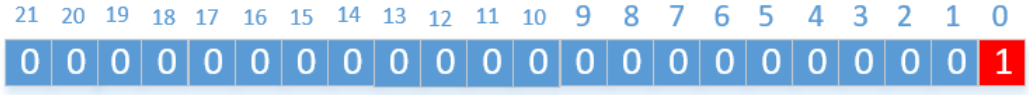
[1，5，11，5]

初始状态

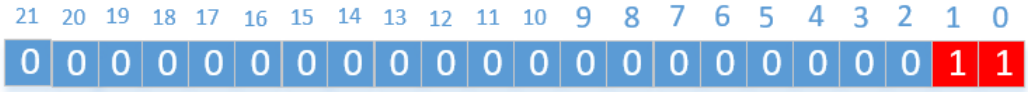


[1，5，11，5]

第一步

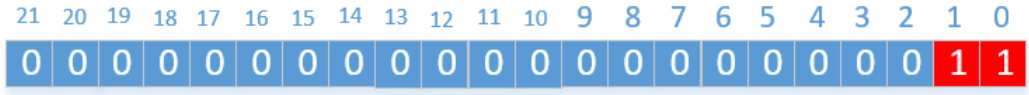


访问数组元素1，二进制位往左移动1位，然后在和原来二进制位进行或运算，结果如下

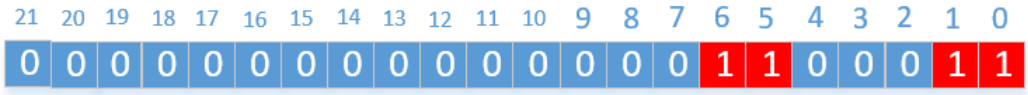


[1，5，11，5]

第二步

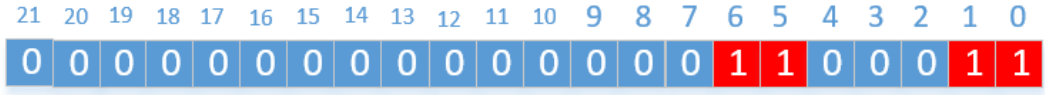


访问数组元素5，二进制位往左移动5位，然后在和原来二进制位进行或运算，结果如下

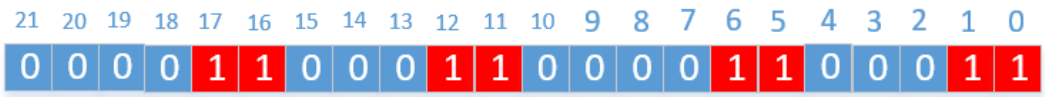


[1，5，11，5]

第三步

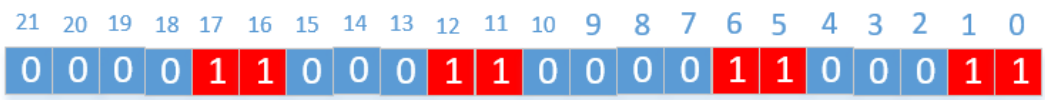


访问数组元素11，二进制位往左移动11位，然后在和原来二进制位进行或运算，结果如下



[1，5，11，5]

第四步



访问数组元素5，二进制位往左移动5位，然后在和原来二进制位进行或运算，结果如下



最后你会发现一个规律，就是最后运算的结果**只要是1的位置，都可以使用数组中的元素组合而成，只要是0的都不能使用数组中的元素组合而成**，搞懂了上面的过程，代码就很容易写了。因为我们只需要判断二进制位中中间的那个值是否为1，所以我们只需要计算低位，高位完全不用计算，因为是往左移动的，高位不会对中间的值产生任何影响，所以这里能做一点优化，最后再来看下代码

```
public boolean canPartition(int[] nums) {
  //计算数组中所有数字的和
  int sum = 0;
  for (int n : nums)
    sum += n;
  //如果sum是奇数，直接返回false
  if ((sum & 1) == 1)
    return false;
  int len = sum >> 1;
  //这里bits的长度是len+1，因为我们只需要计算
  //低位就行了，没必要计算所有的
  byte[] bits = new byte[len + 1];
  bits[0] = 1;
  for (int i = 0; i < nums.length; i++) {
    int num = nums[i];
    int size = len - num;
    for (int j = size; j >= 0; j--) {
      bits[j + num] |= bits[j];
    }
    //判断中位数如果是1，说明可以分成两种相等的
    //子集，直接返回true，不需要再计算了
    if ((bits[len] & 1) != 0)
      return true;
  }
  return false;
}
```

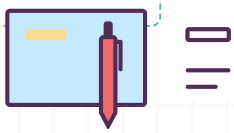
往期推荐

- 573，动态规划解单词拆分
- 572，动态规划解分割回文串 III
- 570，动态规划解回文串分割 IV
- 557，动态规划解戳气球

587，最大的以1为边界的正方形

原创 博哥 数据结构和算法 4天前

> The rest of the world may follow the rules, but I must follow my heart.
>
> 世人也许循规蹈矩，但我必须遵从内心。



问题描述

来源：LeetCode第1139题
难度：中等

给你一个由若干0和1组成的二维网格grid，请你找出**边界**全部由1组成的最大正方形子网格，并返回该子网格中的元素数量。如果不存在，则返回0。

示例 1：

> **输入**：grid = [
> [1,1,1],
> [1,0,1],
> [1,1,1]]
>
> **输出**：9

示例 2：

> **输入**：grid = [[1,1,0,0]]
> **输出**：1

提示：

- 1<=grid.length<=100
- 1<=grid[0].length<=100

- grid[i][j]为0或1

问题分析

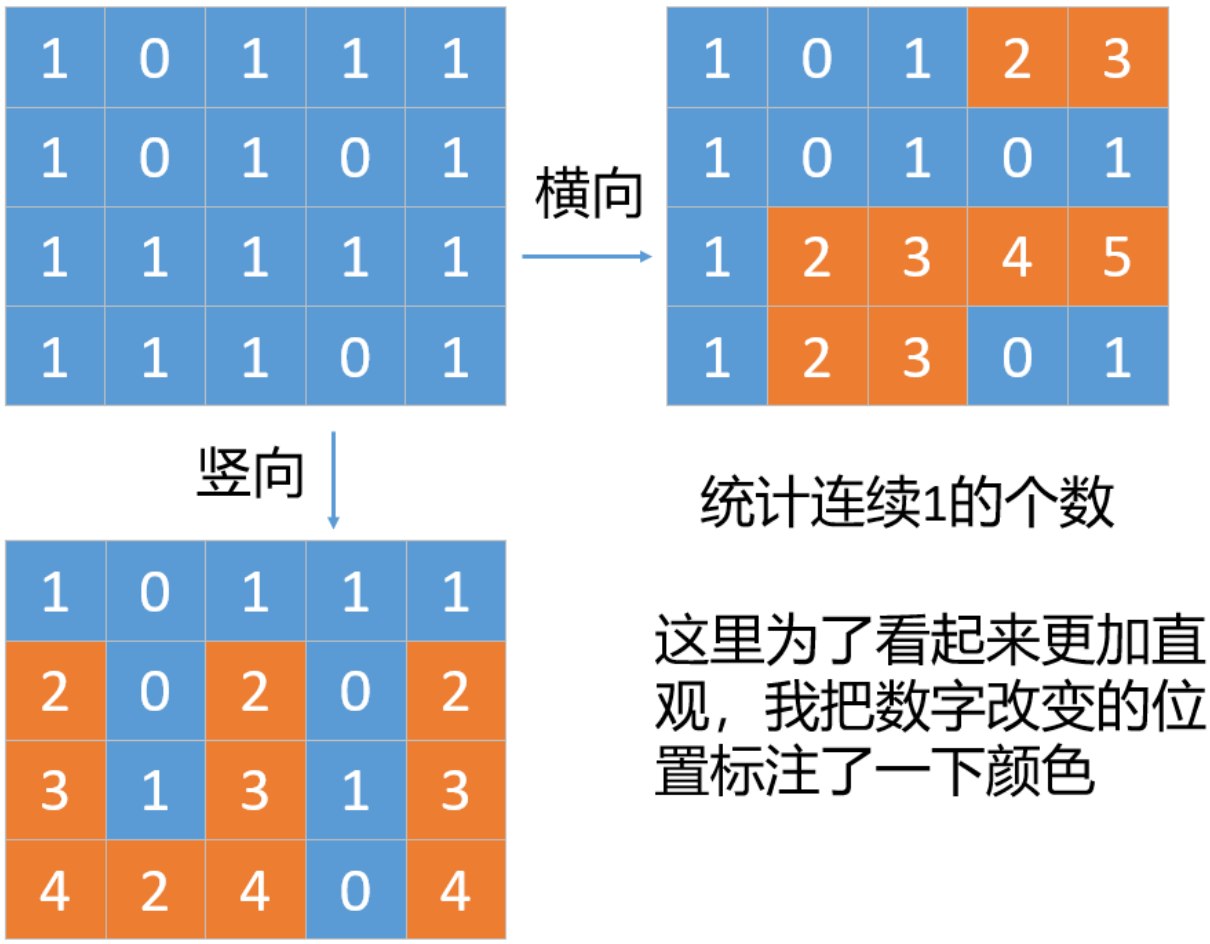
前面我们讲过530，动态规划解最大正方形。第530题需要正方形所有网格中的数字都是1，只要搞懂动态规划的原理，代码就非常简洁。而这题只要正方形4条边的网格都是1即可，中间是什么数字不用管，相对来说这题难度要比第530题稍微大一些。

这题解题思路是这样的

- 第一步先计算每个网格中横向和竖向连续1的个数。
- 第二步遍历二维网格，以每一个格子为正方形的右下角，分别找出上边和左边连续1的个数，取最小值作为正方形的边长，然后判断正方形的左边和上边长度是否都大于等于正方形边长，如果都大于等于正方形边长就更新正方形的最大边长，否则缩小正方形的边长，继续判断……。

如果看不懂也没关系，我们一步一步来，等分析完之后回过头来看，你会恍然大悟，原来这么简单。

1，第一步，计算横向和竖向连续1的个数，举个例子。



代码比较简单，我们定义一个三维数组，其中

- dp[i][j][0]: (i,j)横向连续1的个数
- dp[i][j][1]: (i,j)竖向连续1的个数

我们计算的时候，如果当前位置是0就跳过，只有是1的时候才计算，分别统计左边和上边（也就是横向和竖向）连续1的个数。代码比较简单，我们来看下（这里为了减少一些边界条件的判断，把dp的宽和高都增加了1）。

```
int m = grid.length;
int n = grid[0].length;
//dp[i][j][0]: (i,j)横向连续1的个数
//dp[i][j][1]: (i,j)竖向连续1的个数
int[][][] dp = new int[m + 1][n+1][2];
for (int i = 1; i <= m; i++) {
  for (int j = 1; j <= n; j++) {
    //如果当前位置是0，就跳过
    if (grid[i - 1][j - 1] == 0)
      continue;
    //如果是1，我们就计算横向和竖向连续1的个数
    dp[i][j][0] = dp[i][j - 1][0] + 1;
    dp[i][j][1] = dp[i - 1][j][1] + 1;
  }
}
```

2，**第二步，找出正方形的最大边长**

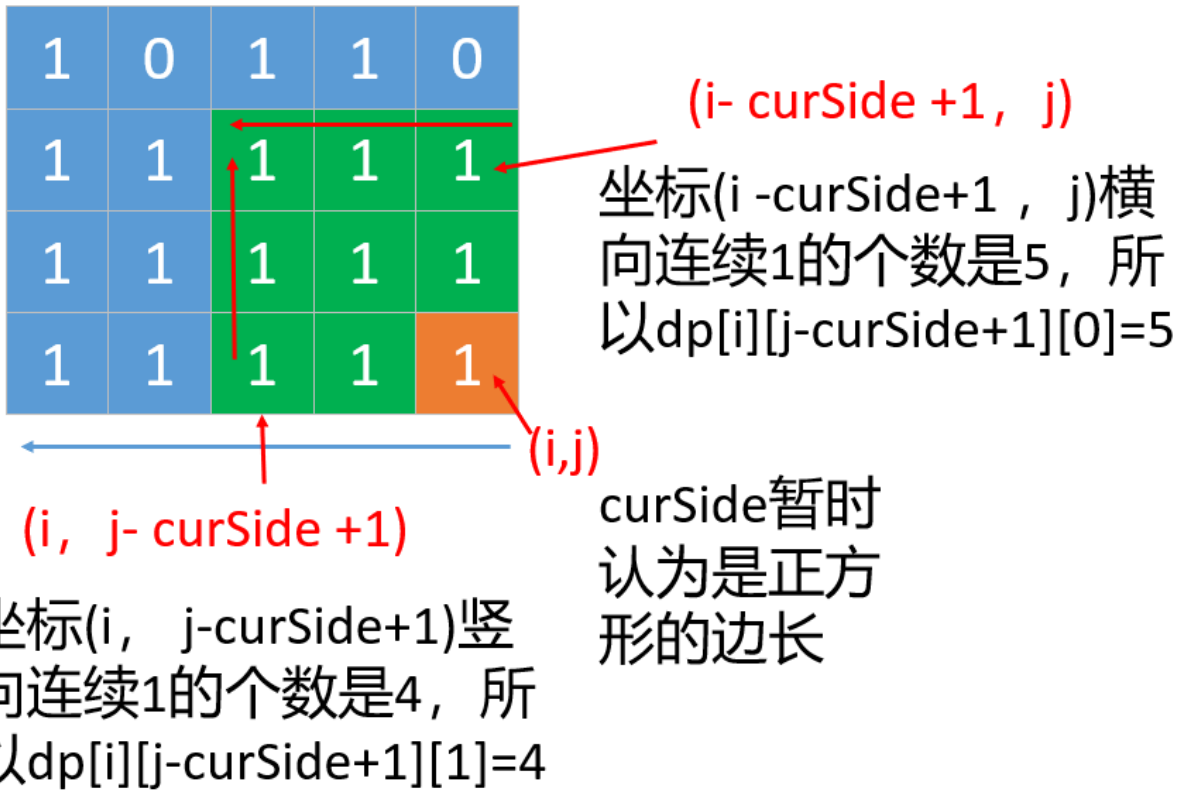
我们会以网格中的每一个位置为正方形的右下角，来找出正方形的边长。如下图所示，我们以橙色的位置1为正方形的右下角，分别沿着左边和上边找出他们连续1的个数，**最小**的作为正方形的边长。因为左边和上边连续1的个数我们在第一步的时候已经计算过，分别是dp[i][j][0]和dp[i][j][1]，也就是正方形的边长我们**暂时**可以认为是，

```
1  int curSide = Math.min(dp[i][j][0], dp[i][j][1]);
```

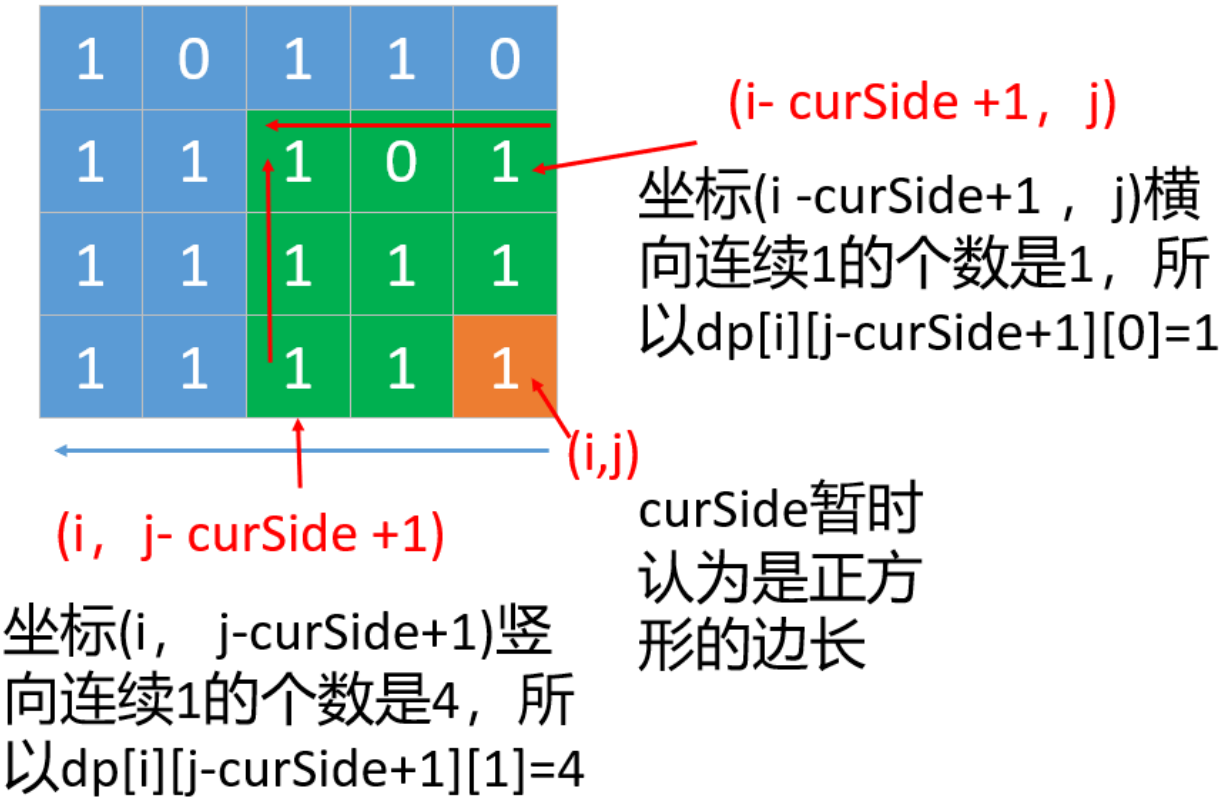


其实大家已经看到了这个边长就是正方形下边和右边的长度，但是正方形的上边和左边我们还没确定，我们继续确定正方形左边和上边的长度。会有两种情况

一种如下图所示，就是正方形左边和上边的长度都大于curSide，我们可以认为以坐标(i,j)为右下角的正方形的最大长度就是curSide。



另一种如下图所示，正方形上边的长度是1，小于curSide。



这种情况下是构不成正方形的，所以我们要缩小curSide的值，然后再继续判断……

搞懂了上面的过程，代码就简单多了，我们来直接看下代码。

```
public int largest1BorderedSquare(int[][] grid) {
  int m = grid.length;
  int n = grid[0].length;
  //dp[i][j][0]: (i,j)横向连续1的个数
  //dp[i][j][1]: (i,j)竖向连续1的个数
  int[][][] dp = new int[m + 1][n + 1][2];
  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
      //如果当前位置是0，就跳过
      if (grid[i - 1][j - 1] == 0)
        continue;
      //如果是1，我们就计算横向和竖向连续1的个数
      dp[i][j][0] = dp[i][j - 1][0] + 1;
      dp[i][j][1] = dp[i - 1][j][1] + 1;
    }
  }
  int maxSide = 0;//记录正方形的最大长度
  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
      //沿着当前坐标往上和往左找出最短的距离，暂时看做是正方形的边长(正方形的具体边长
      //还要看上边和左边的长度，所以这里要判断一下)
      int curSide = Math.min(dp[i][j][0], dp[i][j][1]);
      //如果边长小于maxSide，即使找到了也不可能再比maxSide大，所以我们没必要再找，直接跳过，
      if (curSide <= maxSide)
        continue;
      //curSide可以认为是正方形下边和右边的长度，我们还需要根据正方形上边和左边的长度
      //来确认是否满足正方形的条件
      for (; curSide > maxSide; curSide--) {
        //判断正方形的左边和上边的长度是否大于curSide，如果不大于，我们就缩小正方形
        //的长度curSide，然后继续判断
        if (dp[i][j - curSide + 1][1] >= curSide && dp[i - curSide + 1][j][0] >= curSide) {
          maxSide = curSide;
          //更短的就没必要考虑了，这里直接中断
```

```
                break;
            }
        }
    }
  }
  //返回正方形的边长
  return maxSide * maxSide;
}
```

时间复杂度：O(m*n*min(m,n))，m和n分别是矩阵的宽和高
空间复杂度：O(m*n)，使用了一个三维数组(m*n*2)

往期推荐

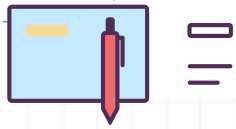
- 573，动态规划解单词拆分
- 572，动态规划解分割回文串 III
- 570，动态规划解回文串分割 IV
- 568，动态规划解最后一块石头的重量 II

576，动态规划解最长公共子串

原创 博哥 数据结构和算法 昨天

We are so excited about the future.

未来可期。



问题描述

来源：牛客题霸第127题

难度：中等

给定两个字符串str1和str2,输出两个字符串的最长公共子串
题目保证str1和str2的最长公共子串存在且唯一。

示例1

```
"1AB2345CD","12345EF"
"2345"
```

备注：

- $1 \le |str1|, |str2| \le 5000$

动态规划解决

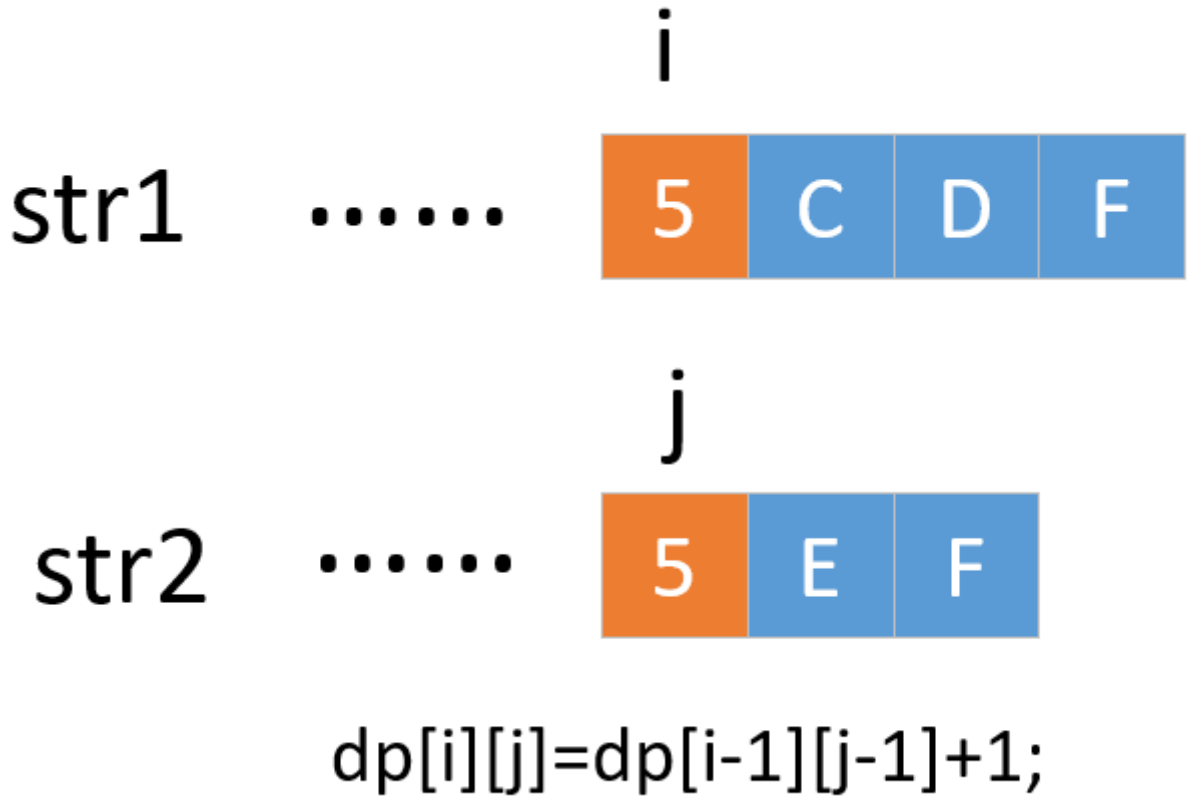
注意这题求的是最长公共子串，不是最长公共子序列，子序列可以是不连续的，但子串一定是连续的。

定义dp[i][j]表示字符串str1中第i个字符和str2种第j个字符为最后一个元素所构成的最长公共子串。如果要求dp[i][j]，也就是str1的第i个字符和str2的第j个字符为最后一个元素所构成的最长公共子串，我们首先需要判断这两个字符是否相等。

如果不相等，那么他们就不能构成公共子串，也就是

dp[i][j]=0;

如果相等，我们还需要计算前面相等字符的个数，其实就是dp[i-1][j-1]，所以dp[i][j]=dp[i-1][j-1]+1;



有了递推公式，代码就比较简单了，我们使用两个变量，一个记录最长的公共子串，一个记录最长公共子串的结束位置，最后再对字符串进行截取即可，来看下代码

```
public String LCS(String str1, String str2) {
  int maxLenth = 0;//记录最长公共子串的长度
  //记录最长公共子串最后一个元素在字符串str1中的位置
  int maxLastIndex = 0;
  int[][] dp = new int[str1.length() + 1][str2.length() + 1];
  for (int i = 0; i < str1.length(); i++) {
    for (int j = 0; j < str2.length(); j++) {
      //递推公式，两个字符相等的情况
      if (str1.charAt(i) == str2.charAt(j)) {
        dp[i + 1][j + 1] = dp[i][j] + 1;
        //如果遇到了更长的子串，要更新，记录最长子串的长度，
        //以及最长子串最后一个元素的位置
        if (dp[i + 1][j + 1] > maxLenth) {
          maxLenth = dp[i + 1][j+1];
          maxLastIndex = i;
        }
      } else {
        //递推公式，两个字符不相等的情况
        dp[i + 1][j+1] = 0;
      }
    }
  }
  //最字符串进行截取，substring(a,b)中a和b分别表示截取的开始和结束位置
  return str1.substring(maxLastIndex - maxLenth + 1, maxLastIndex + 1);
}
```

时间复杂度：O（m*n），m和n分别表示两个字符串的长度

空间复杂度：O（m*n）

代码优化，把二维数组变为一维数组

```java
public String LCS(String str1, String str2) {
    int maxLenth = 0;//记录最长公共子串的长度
    //记录最长公共子串最后一个元素在字符串str1中的位置
    int maxLastIndex = 0;
    int[] dp = new int[str2.length() + 1];
    for (int i = 0; i < str1.length(); i++) {
        //注意这里是倒叙
        for (int j = str2.length() - 1; j >= 0; j--) {
            //递推公式，两个字符相等的情况
            if (str1.charAt(i) == str2.charAt(j)) {
                dp[j + 1] = dp[j] + 1;
                //如果遇到了更长的子串，要更新，记录最长子串的长度，
                //以及最长子串最后一个元素的位置
                if (dp[j + 1] > maxLenth) {
                    maxLenth = dp[j + 1];
                    maxLastIndex = i;
                }
            } else {
                //递推公式，两个字符不相等的情况
                dp[j + 1] = 0;
            }
        }
    }
    //最字符串进行截取，substring(a,b)中a和b分别表示截取的开始和结束位置
    return str1.substring(maxLastIndex - maxLenth + 1, maxLastIndex + 1);
}
```

时间复杂度：O（m*n），m和n分别表示两个字符串的长度

空间复杂度：O（n），只需要一个一维数组即可

总结

这题比较简单，很久以前也讲过这题[370，最长公共子串和子序列]，之前只是返回一个具体的数字即可，而这题需要返回具体的公共子串，多了一步。

往期推荐

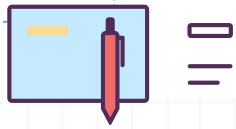
- 407，动态规划和滑动窗口解决最长重复子数组
- 568，动态规划解最后一块石头的重量 II
- 557，动态规划解戳气球
- 543，剑指 Offer-动态规划解礼物的最大价值

573，动态规划解单词拆分

原创 博哥 数据结构和算法 1周前

It dose not do to dwell on dreams and forget to live.

人不能因为依赖梦想而忘记生活。



问题描述

给定一个非空字符串s和一个包含非空单词的列表wordDict，判定s是否可以被空格拆分为一个或多个在字典中出现的单词。

说明：

1. **拆分时可以重复使用字典中的单词。**
2. **你可以假设字典中没有重复的单词。**

示例 1：

> **输入**:
> s = "leetcode",
> wordDict = ["leet", "code"]
>
> **输出**: true
> **解释**: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。

示例 2：

> **输入**:
> s = "applepenapple",
> wordDict = ["apple", "pen"]
>
> **输出**: true
> **解释**: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
> 注意你可以重复使用字典中的单词。

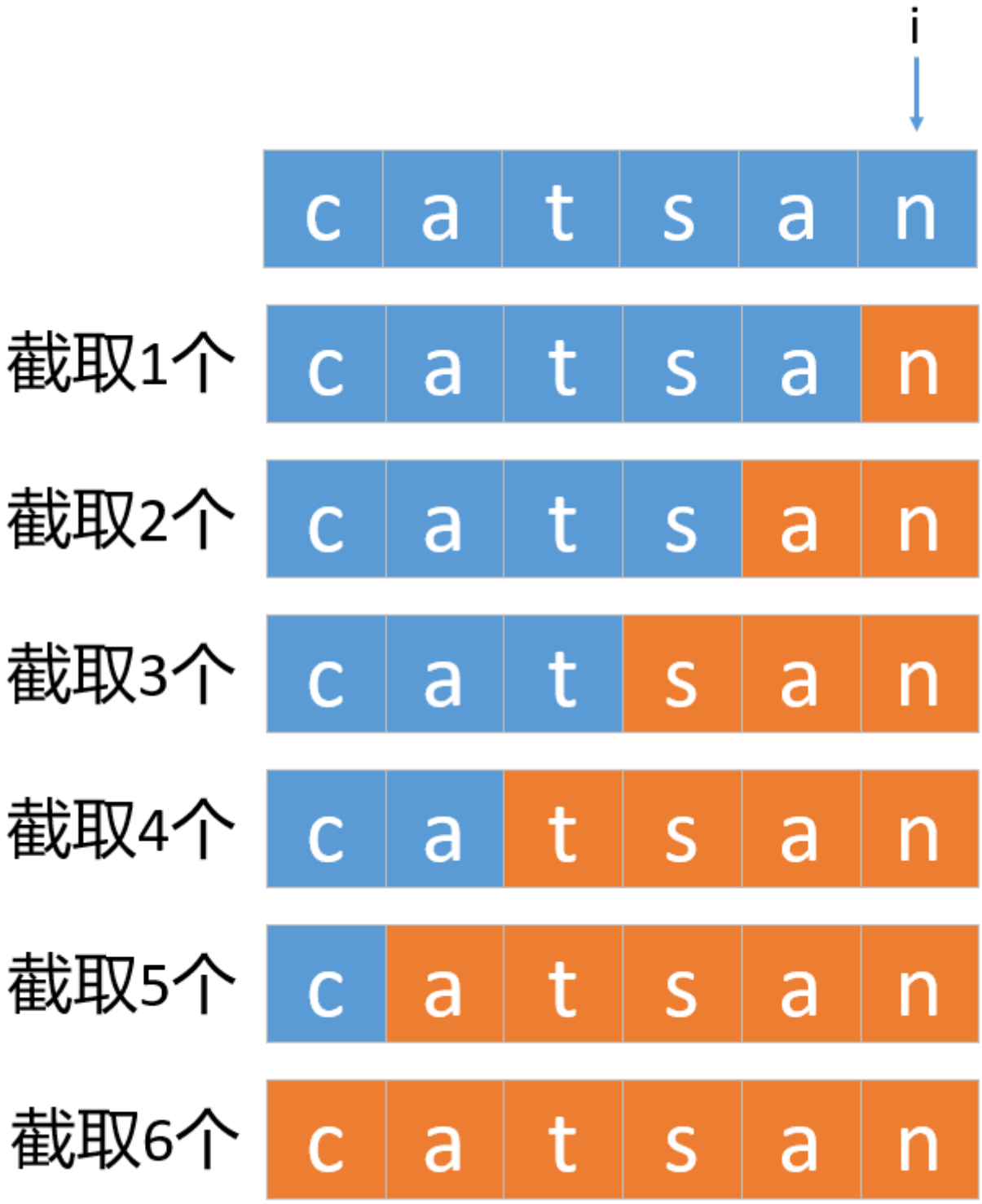
示例 3：

> **输入**:
> s = "catsandog",
> wordDict = ["cats", "dog", "sand", "and", "cat"]
>
> **输出**: false

动态规划解决

这题是让把字符串s拆分成一些子串，并且判断这些子串是否都存在字典wordDict中。

我们定义dp[i]表示字符串的前i个字符经过拆分是否都存在于字典wordDict中。如果要求dp[i]，我们需要往前截取k个字符，判断子串[i-k+1，i]是否存在于字典wordDict中，并且前面[0，i-k]子串拆分的子串也是否都存在于wordDict中，如下图所示



前面[0，i-k]子串拆分的子串是否都存在于wordDict中只需要判断dp[i-k]即可，而子串[i-k+1，i]是否存在于字典wordDict中需要查找，所以动态规划的递推公式很容易列出来

dp[i]=dp[i-k]&&dict.contains(s.substring(i-k, i));

这个k我们需要一个个枚举，我们来看下最终代码

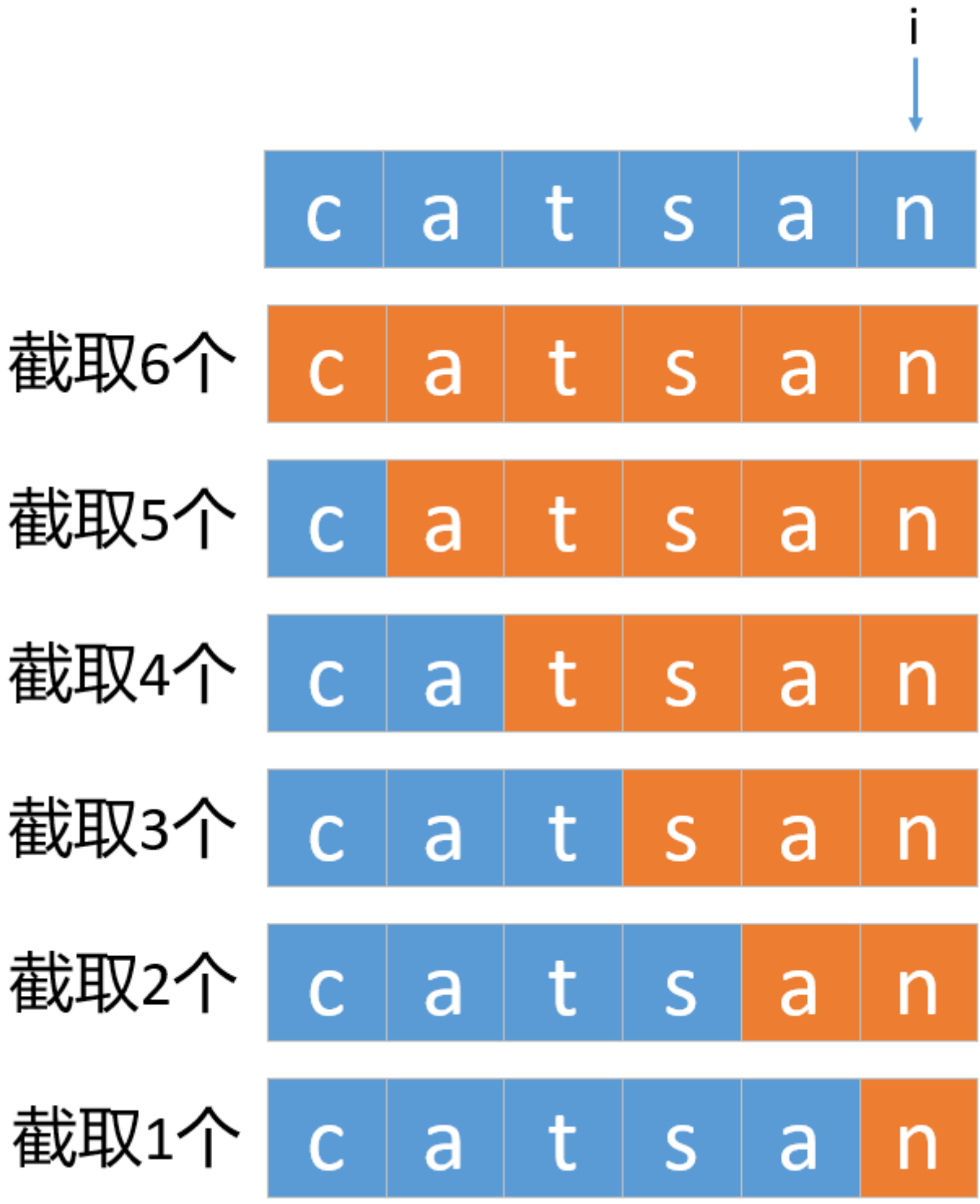
```
public boolean wordBreak(String s, List<String> dict) {
  boolean[] dp = new boolean[s.length() + 1];
  for (int i = 1; i <= s.length(); i++) {
    //枚举k的值
    for (int k = 0; k <= i; k++) {
      //如果往前截取全部字符串，我们直接判断子串[0,i-1]
      //是否存在于字典wordDict中即可
      if (k == i) {
        if (dict.contains(s.substring(0, i))) {
```

```
            dp[i] = true;
            continue;
          }
        }
        //递推公式
        dp[i] = dp[i - k] && dict.contains(s.substring(i - k, i));
        //如果dp[i]为true，说明前i个字符串结果拆解可以让他的所有子串
        //都存在于字典wordDict中，直接终止内层循环，不用再计算dp[i]了。
        if (dp[i]) {
          break;
        }
      }
    }
    return dp[s.length()];
  }
```

上面代码有一个判断，就是截取的是前面全部字符串的时候要单独判断，其实当截取全部的时候我们只需要判断这个字符串是否存在于字典wordDict中即可，可以让dp[0]为true，dp[0]表示的是空字符串。这样代码会简洁很多，我们来看下

```
  public boolean wordBreak(String s, List<String> dict) {
    boolean[] dp = new boolean[s.length() + 1];
    dp[0] = true;//边界条件
    for (int i = 1; i <= s.length(); i++) {
      for (int j = 0; j < i; j++) {
        dp[i] = dp[j] && dict.contains(s.substring(j, i));
        if (dp[i]) {
          break;
        }
      }
    }
    return dp[s.length()];
  }
```

这个和第一种写法不太一样，这个每次截取的方式如下图所示。



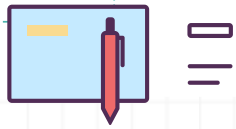
总结

这题我们也可以把它看作是一个完全背包问题，背包就是字符串s，而所选择的商品就是字典中的字符串，因为字典中的字符串可以选择多次并且没有限制，所以我们可以认为他是一个完全背包问题，完全背包问题我们以后会讲。其实这题还可以使用dfs来解决，由于这题写完的时候已经比较晚了，就留在明天讲吧。

572，动态规划解分割回文串 III

原创 博哥 数据结构和算法 1周前

> Nothing in this world that is worth having comes easy.
>
> 这世界上凡是值得拥有的东西，都不易获得。



问题描述

来源：LeetCode第1278题
难度：困难

给你一个由小写字母组成的字符串s，和一个整数k。

请你按下面的要求分割字符串：

- **首先，你可以将s中的部分字符修改为其他的小写英文字母。**
- **接着，你需要把s分割成k个非空且不相交的子串，并且每个子串都是回文串。**

请返回以这种方式分割字符串所需修改的最少字符数。

示例 1：

> **输入**：s = "abc", k = 2
> **输出**：1
> **解释**：你可以把字符串分割成 "ab" 和 "c"，并修改 "ab" 中的 1 个字符，将它变成回文串。

示例 2：

> **输入**：s = "aabbc", k = 3
> **输出**：0
> **解释**：你可以把字符串分割成 "aa"、"bb" 和 "c"，它们都是回文串。

示例 3：

> **输入**：s = "leetcode", k = 8
> **输出**：0

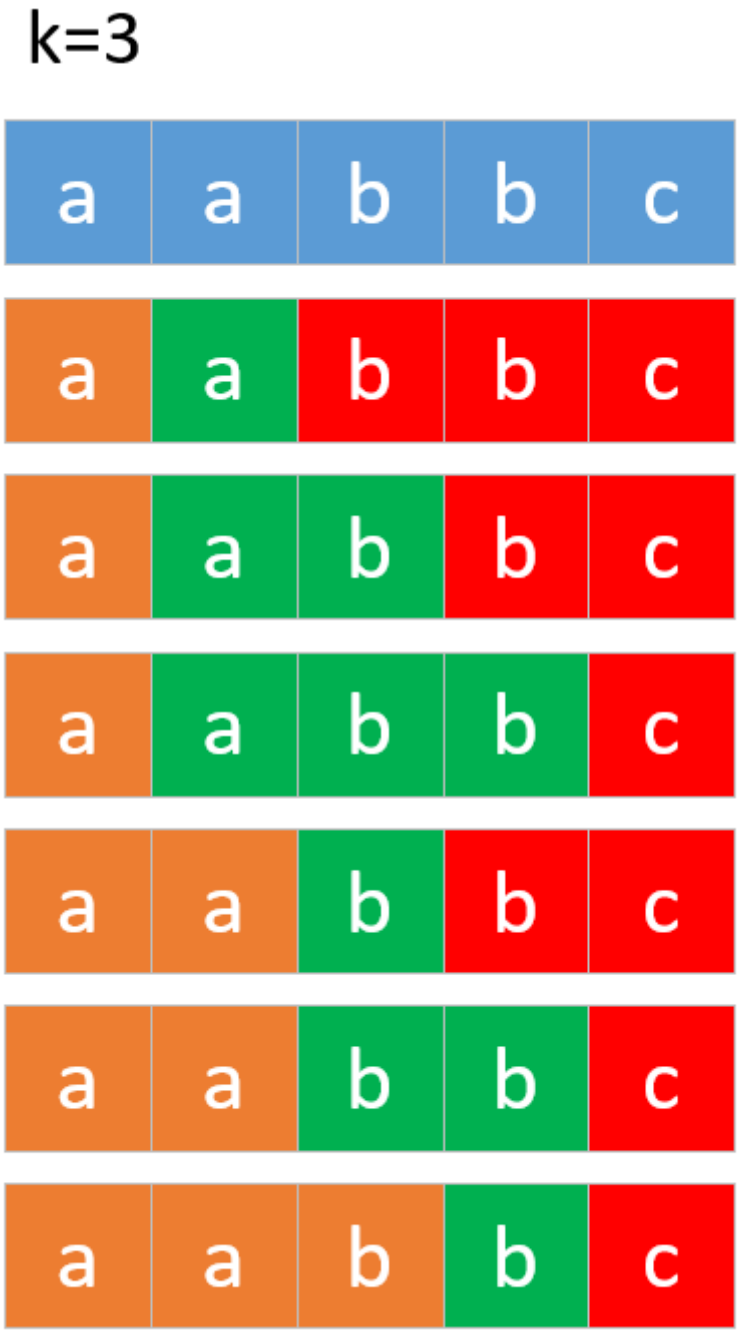
提示：

- 1 <= k <= s.length <= 100
- s 中只含有小写英文字母。

动态规划解决

这题是让通过修改一些字符把字符串分割为k个子串，并且每个子串都是回文的，问最少修改的字符数。

最容易想到的就是把字符串s分割k个子串的所有可能组合，然后计算每个组合中子串变成回文串修改字符的数量，最后返回最小的即可。比如示例二中我们可以分割为



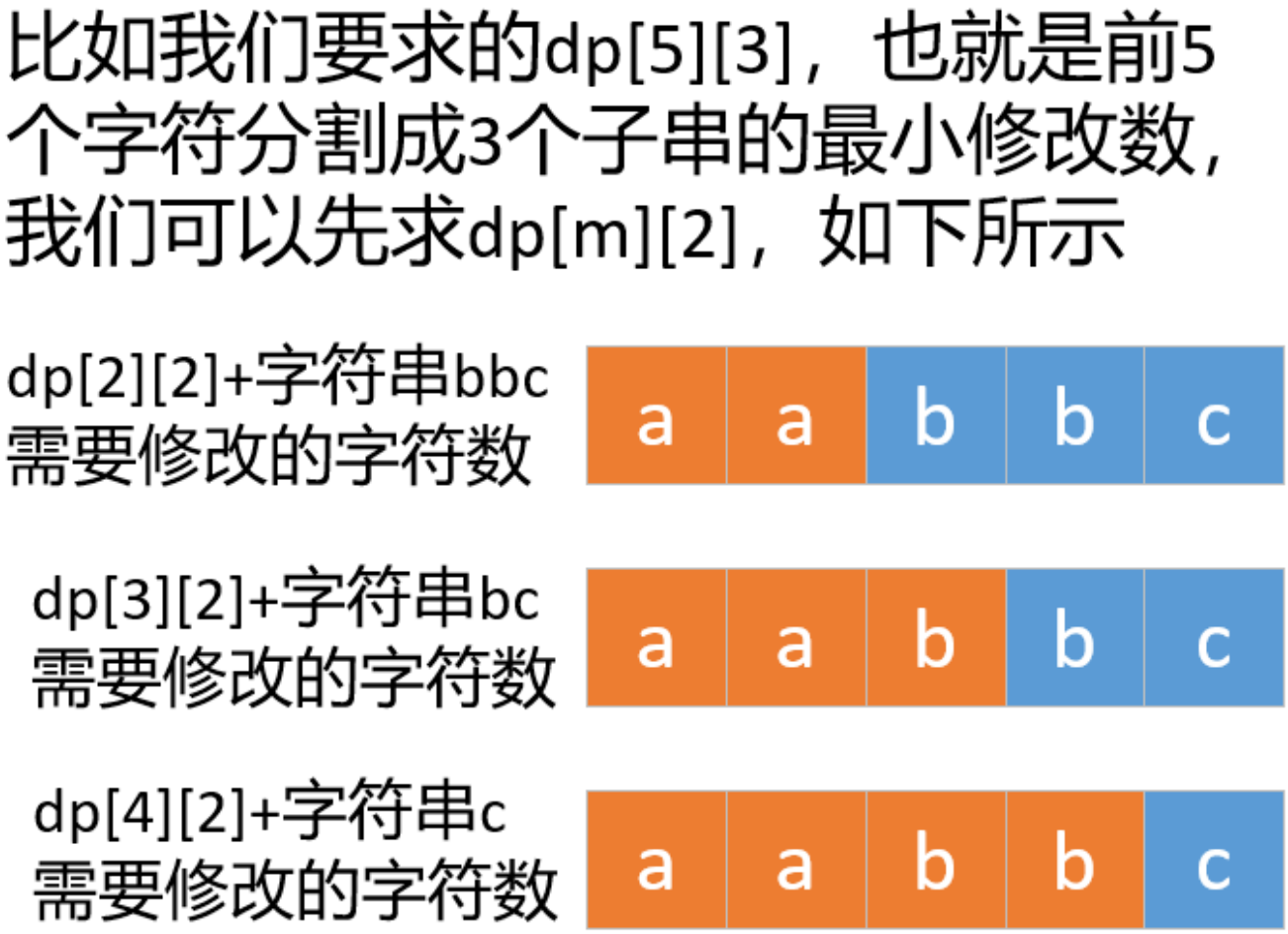
但是当字符串s比较长的时候，运行效率是很差的。这里我们可以使用动态规划来解决，定义dp[i][j]表示表示字符串s的前i个字符分割为j个子串的修改的最小字符数。很明显i必须大于等于j，要不然不可能分割为j个子串。

动态规划最关键的是找出递推公式，当我们要求dp[i][j]的时候，只需要找出前m个字符分割成j-1个子串所修改的最小字符数+最后一个子串所需要修改字符数。（最后一个子串就是前i个字符-前m个字符），看到这里大家是不是很容易想到比较经典的基础背包问题，前面也有讲过371，背包问题系列之-基础背包问题，其中还有多重背包和完全背包这些我们以后会再讲。

所以递推公式我们很容易想到

```
dp[i][j]=dp[m][j-1]+change(s,m,i-1);
```

这里我们需要枚举m的值，但要注意m必须大于等于j-1，如下图所示



这题要求的是返回最小的字符修改数，所以我们记录最小的即可

```
dp[i][j] = Math.min(dp[i][j], dp[m][j - 1] + change(s, m, i - 1));
```

其中change(s,m,i-1)表示字符串s[m,i-1]变成回文串所需要修改的字符数。他的代码比较简单，我们来看下

```
//字符串的子串[left,right]变成回文串所需要修改的字符数
private int change(String s, int left, int right) {
  int count = 0;
  while (left < right) {
    //如果两个指针指向的字符相同，我们不需要修改。
    //如果不相同，只需要修改其中的一个即可，所以
    // 修改数要加1
    if (s.charAt(left++) != s.charAt(right--))
      count++;
  }
  return count;
}
```

分析到这里，这题的代码基本上就呼之欲出了，我们来看下完整代码

```
public int palindromePartition(String s, int k) {
  int length = s.length();
  //dp[i][j]表示s的前i个字符分割成k个子串所修改的最少字符数。
  int[][] dp = new int[length + 1][k + 1];
  //因为这题要求的是所需要修改的最少字符数，初始值我们赋值尽可能大
  for (int i = 0; i < dp.length; i++) {
    Arrays.fill(dp[i], length);
  }
  //前i个字符，分割成j个回文子串
```

```
    for (int i = 1; i <= length; i++) {
      //前i个字符最大只能分割成i个子串，所以不能超过i，
      //我们取i和k的最小值
      int len = Math.min(i, k);
      for (int j = 1; j <= len; j++) {
        if (j == 1) {
          //如果j等于1，则表示没有分割，我们直接计算
          dp[i][j] = change(s, j - 1, i - 1);
        } else {
          //如果j不等于1，我们计算分割所需要修改的最小字符数，因为m的值要
          //大于等于j-1，我们就从最小的开始枚举
          for (int m = j - 1; m < i; m++) {
            //递推公式
            dp[i][j] = Math.min(dp[i][j], dp[m][j - 1] + change(s, m, i - 1));
          }
        }
      }
    }
    //返回前length个字符分割成k个子串所需要修改的最少字符数
    return dp[length][k];
  }

  //字符串的子串[left,right]变成回文串所需要修改的字符数
  private int change(String s, int left, int right) {
    int count = 0;
    while (left < right) {
      //如果两个指针指向的字符相同，我们不需要修改。
      //如果不相同，只需要修改其中的一个即可，所以
      // 修改数要加1
      if (s.charAt(left++) != s.charAt(right--))
        count++;
    }
    return count;
  }
```

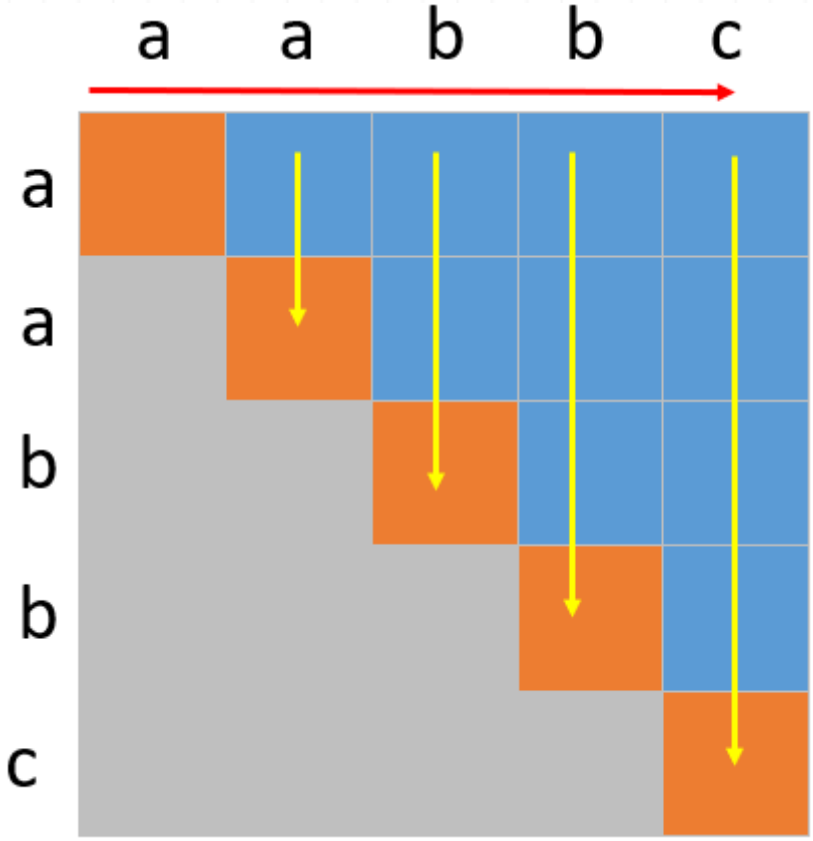
动态规划代码优化

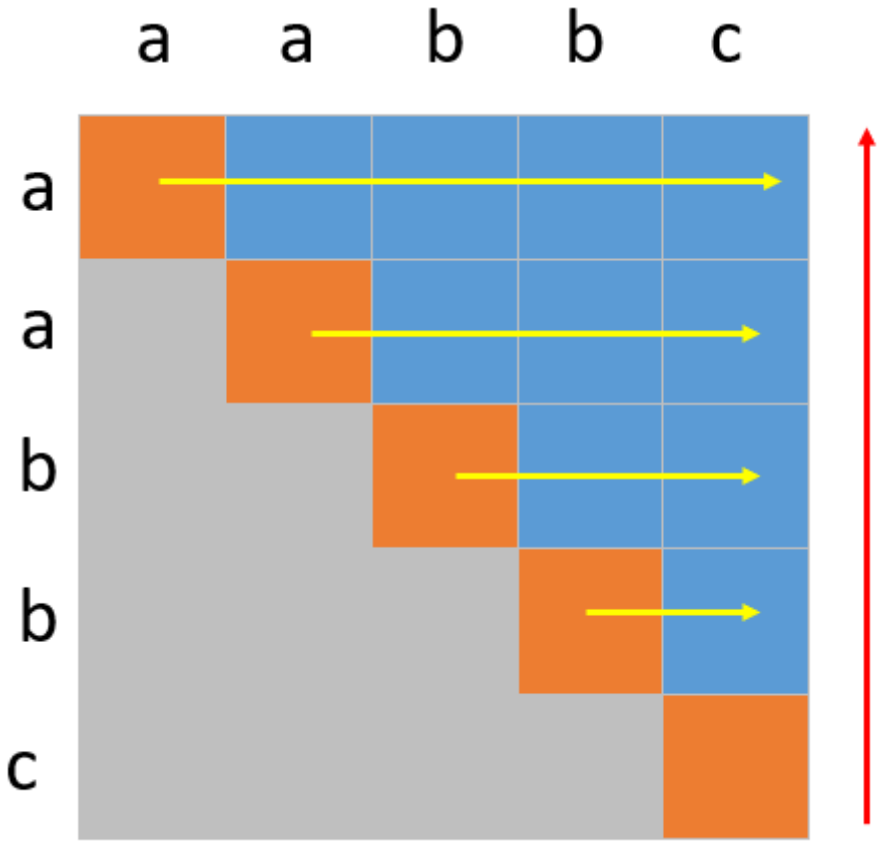
上面代码中都有注释，这里就不在过多介绍，我们来看一下change方法，因为这里会涉及到字符的重复计算，导致效率不高，我们可以先计算所有的子串变成回文串需要修改的字符数，然后再使用，来看下代码

```
  public int palindromePartition(String s, int k) {
    int length = s.length();

    //palindrome[i][j]表示子串[i,j]转化为回文串所需要的修改的字符数
    int[][] palindrome = new int[length][length];
    //2种实现方式
    //    //一列一列的从左往右（只遍历右上部分）
    //    for (int j = 1; j < length; j++) {
    //      for (int i = 0; i < j; i++) {
    //        palindrome[i][j] = palindrome[i + 1][j - 1] + (s.charAt(i) == s.charAt(j) ? 0 : 1);
    //      }
    //    }

    //一行一行的从下往上（只遍历右上部分）
    for (int i = length - 2; i >= 0; i--) {
      for (int j = i + 1; j < length; j++) {
        palindrome[i][j] = palindrome[i + 1][j - 1] + (s.charAt(i) == s.charAt(j) ? 0 : 1);
      }
    }

    //dp[i][j]表示s的前i个字符分割成k个回文子串的最少次数，
    //第一行和第一列应该都是0。
    int[][] dp = new int[length + 1][k + 1];
    for (int i = 1; i < dp.length; i++) {
```

```
25          Arrays.fill(dp[i], Integer.MAX_VALUE);
26      }
27      //前i个字符，分割成j个回文子串
28      for (int i = 1; i <= length; i++) {
29          int len = Math.min(i, k);
30          for (int j = 1; j <= len; j++) {
31              if (j == 1)//字符串的下标是从0开始的，所以这里要减1
32                  dp[i][j] = palindrome[j - 1][i - 1];
33              else
34                  for (int m = j - 1; m < i; m++) {
35                      dp[i][j] = Math.min(dp[i][j], dp[m][j - 1] + palindrome[m][i - 1]);
36                  }
37          }
38      }
39      return dp[length][k];
40  }
```

因为dp[i][j]中i必须大于等于j，所以这里遍历的时候可以有两种方式，如下图所示





总结

LeetCode上分隔回文串总共有4题，其中只有一道中等，其他3道都是困难，这里把这4道都讲完了，也可以一起看下前面的3道题。

551，回溯算法解分割回文串

553，动态规划解分割回文串 II

570，动态规划解回文串分割 IV

往期推荐

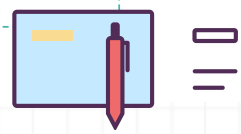
- 558，最长回文串
- 540，动态规划和中心扩散法解回文子串
- 529，动态规划解最长回文子序列
- 517，最长回文子串的3种解决方式

570，动态规划解回文串分割 IV

原创 博哥 数据结构和算法 昨天

I am not afraid of storms,for I am learning how to sail my ship.

我不害怕风暴，因为我正在学习如何乘风破浪。



问题描述

来源：LeetCode第1745题
难度：困难

给你一个字符串s，如果可以将它分割成三个非空**回文**子字符串，那么返回true，否则返回false。

当一个字符串正着读和反着读是一模一样的，就称其为回文字符串。

示例 1：

> **输入**：s = "abcbdd"
> **输出**：true
> **解释**："abcbdd" = "a" + "bcb" + "dd"，三个子字符串都是回文的。

示例 2：

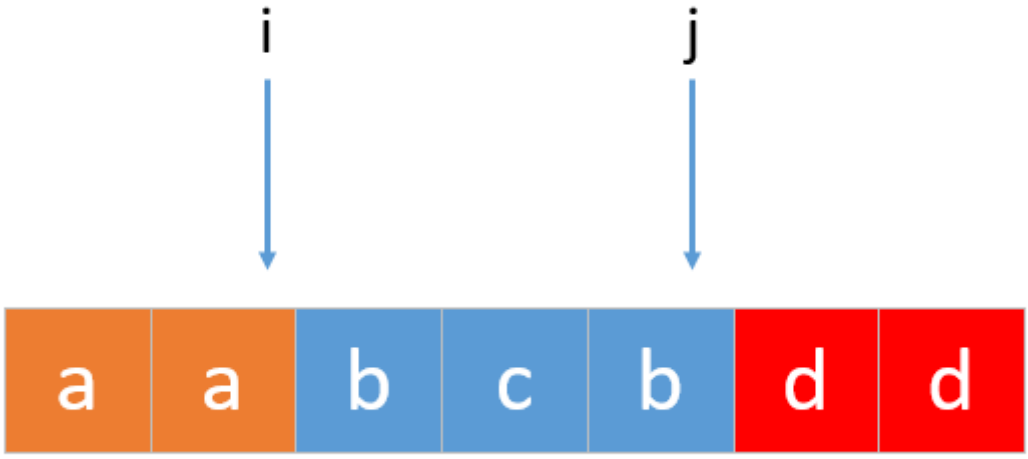
> **输入**：s = "bcbddxy"
> **输出**：false
> **解释**：s 没办法被分割成 3 个回文子字符串。

提示：

- 3 <= s.length <= 2000
- s 只包含小写英文字母。

动态规划解决

这题要求能不能把字符串s分隔成3个非空的回文子串。最简单的一种方式就是把字符串分隔成3个子串，然后再判他们是否都是回文的，如果是则返回true，如果不是则继续分隔然后接着在判断……直到不能分隔为止，我们随便举个例子画个图看一下。



大致代码如下

```
public boolean checkPartitioning(String s) {
  for (int i = 0; i < s.length(); i++) {
    for (int j = i + 1; j < s.length() - 1; j++) {
      //截取字符串[0,i]
      String str1 = s.substring(0, i + 1);
      //截取字符串[i+1,j]
      String str2 = s.substring(i + 1, j + 1);
      //截取字符串[j+1,s.length()-1]
      String str3 = s.substring(j + 1, s.length());
      //把字符串s截取3段，判断每段是否都是回文的
      if (isPalindrome(str1) && isPalindrome(str2) && isPalindrome(str3))
        return true;
    }
  }
  return false;
}
```

如果字符串s比较短的话，上面代码是没有问题的，但如果字符串s比较长，很容易超时。

我们可以先计算字符串s的所有子串中哪些是回文的，哪些不是，然后再判断。关于计算方式在《540，动态规划和中心扩散法解回文子串》中有详细介绍，除此之外还可有看下

553，动态规划解分割回文串 II

551，回溯算法解分割回文串

517，最长回文子串的3种解决方式

这里就不在重复介绍。我们来直接看下代码

```
public boolean checkPartitioning(String s) {
  int length = s.length();
  //dp[i][j]：表示字符串s从下标i到j是否是回文串
  boolean[][] dp = new boolean[length][length];
  for (int i = length - 1; i >= 0; i--) {
    for (int j = i; j < length; j++) {
      //如果i和j指向的字符不一样，那么dp[i][j]就
      //不能构成回文字符串
      if (s.charAt(i) != s.charAt(j))
```

```
            continue;
          dp[i][j] = j - i <= 2 || dp[i + 1][j - 1];
        }
      }

      //然后再截取3段，判断这3段是否都是回文的
      for (int i = 0; i < length; i++) {
        for (int j = i + 1; j < length - 1; j++) {
          if (dp[0][i] && dp[i + 1][j] && dp[j + 1][length - 1])
            return true;
        }
      }
      return false;
    }

```

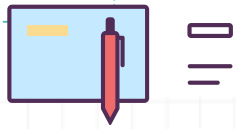
往期推荐

- 558，最长回文串
- 553，动态规划解分割回文串 II
- 551，回溯算法解分割回文串
- 497，双指针验证回文串

568，动态规划解最后一块石头的重量 II

原创 博哥 数据结构和算法 今天

> Victory won't come to me unless I go to it.
>
> 胜利是不会向我走来的，我必须自己走向胜利。



问题描述

来源：LeetCode第1049题
难度：中等

有一堆石头，用整数数组stones表示。其中stones[i]表示第i块石头的重量。

每一回合，从中选出任意两块石头，然后将它们一起粉碎。假设石头的重量分别为x和y，且x<=y。那么粉碎的可能结果如下：

如果x==y，那么两块石头都会被完全粉碎；
如果x!=y，那么重量为x的石头将会完全粉碎，而重量为y的石头新重量为y-x。
最后，最多只会剩下一块石头。返回此石头最小的可能重量。如果没有石头剩下，就返回0。

示例 1：

> **输入**：stones = [2,7,4,1,8,1]
> **输出**：1
> **解释**：
> 组合 2 和 4，得到 2，所以数组转化为 [2,7,1,8,1]，
> 组合 7 和 8，得到 1，所以数组转化为 [2,1,1,1]，
> 组合 2 和 1，得到 1，所以数组转化为 [1,1,1]，
> 组合 1 和 1，得到 0，所以数组转化为 [1]，这就是最优值。

示例 2：

> **输入**：stones = [31,26,33,21,40]

> **输出**：5

示例 3：

> **输入**：stones = [1,2]
> **输出**：1

提示：

- 1<=stones.length<=30
- 1<=stones[i]<=100

动态规划解决

这题是让每次从数组中任意选择两个石头，让他们相互销毁，求最终剩下的一块石头最小可能重量，如果没有剩下，则返回0。直接计算可能不太好计算，我们这样来思考一下

这题相当于一群战斗力各不相同的人相互厮杀，如果战斗力相同他们就同归于尽，如果战斗力不同，战斗力小的死去，战斗力高的活下来，但战斗力高的战斗值要减去战斗力小的值。最后如果他们都能同归于尽最好不过了，如果不能都同归于尽，最后会剩下一个人，他的战斗值越小越好。

所以这题相当于把一群战斗力各不相同的人分为两组，每组战斗力总和相差越小越好，也就是把数组分为两部分，这两部分的差值尽可能小。

解题思路如下
我们首先计算数组中所有元素的和，比如是sum，然后把它分为两部分，如果sum是**偶数**，那么这两部分的值都是sum/2，如果sum是**奇数**，则一部分是sum/2，另一部分是sum/2+1；我们取较小的sum/2。

解题思路就变成了从数组中选择一些元素，让他们的和尽可能的接近sum/2，经过我们的不断分析，所以这题就变成了经典的基础背包问题，前面我们也讲过《371，背包问题系列之-基础背包问题》。也就是说背包的容量是sum/2，让我们从数组选选择一些元素放到背包中，求背包所能容纳的最大价值。

定义dp[i][j]表示前i个元素放到背包容量为j的背包中，所能获取的最大价值，当我们选择第i个元素stones[i-1]的时候。
如果stones[i-1]<=j，说明第i个元素能放到背包中，我们可以选择放也可以选择不放，如果选择放那么

dp[i][j]=dp[i-1][j-stones[i-1]]+stones[i-1]

如果选择不放，那么

dp[i][j]=dp[i-1][j]

这两种情况我们取最大值即可，也就是

```
dp[i][j]=Math.max(dp[i-1][j],dp[i-1][j-stones[i-1]]+stones[i-1]);
```

如果stones[i-1]>j，说明背包容量不够，我们没法把第i个元素放到背包中，所以

dp[i][j]=dp[i-1][j]

递推公式如下

```
if (j >= stones[i - 1]) {
  dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - stones[i - 1]] + stones[i - 1]);
} else {
  //背包容量已经放不下了
  dp[i][j] = dp[i - 1][j];
}
```

但这道题并不是让求背包的最大容量，前面我们分析的是把数组分为两组，如果一组的值是x，那么另一组的值就是sum-x，他们的差值就是(sum-x)-x；这个就是我们要求的值，而x就是上面我们所求的背包的最大容量。

我们来看下最终代码

```
public int lastStoneWeightII(int[] stones) {
  int length = stones.length;
  int sum = 0;
  for (int num : stones) {
    sum += num;
  }
  //背包的容量
  int capacity = sum >> 1;
  //dp[i][j]表示前i个石头放进容量为j的背包所能获取的最大重量
  int dp[][] = new int[length + 1][capacity + 1];

  for (int i = 1; i <= length; i++) {
    for (int j = 1; j <= capacity; j++) {
      //如果背包剩余容量能放下石头stones[i - 1]，取把石头stones[i - 1]放进
      //背包和不放进背包的最大值
      if (j >= stones[i - 1]) {
        dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - stones[i - 1]] + stones[i - 1]);
      } else {
        //背包容量已经放不下石头了
        dp[i][j] = dp[i - 1][j];
      }
    }
  }
  //sum - dp[length][capacity]是一部分，dp[length][capacity]是另一部分，上面
  //capacity的取值是sum >> 1，往下取整，所以前面的肯定不小于后面的，不需要取绝对值
  return (sum - dp[length][capacity]) - dp[length][capacity];
}
```

动态规划代码优化

上面代码计算dp[i][j]的时候只和二维数组的上面一行有关，所以我们可以改为一位数组，注意计算dp[i][j]的时候dp[i - 1][j - stones[i - 1]]的值可能已经被重新赋值，我们可以从后往前遍历，这样使用前面值的时候可以保证前面的还没有计算，也就是还没有被重新赋值。代码如下

```
public int lastStoneWeightII(int[] stones) {
    int length = stones.length;
    int sum = 0;
    for (int num : stones) {
        sum += num;
    }
    //背包的容量
    int capacity = sum >> 1;

    int dp[] = new int[capacity + 1];

    for (int i = 1; i <= length; i++) {
        //这里要从大到小开始遍历
        for (int j = capacity; j >= 1; j--) {
            //如果背包剩余容量能放下石头stones[i - 1]，取把石头stones[i - 1]放进
            //背包和不放进背包的最大值
            if (j >= stones[i - 1]) {
                dp[j] = Math.max(dp[j], dp[j - stones[i - 1]] + stones[i - 1]);
            }
        }
    }
    return (sum - dp[capacity]) - dp[capacity];
}
```

往期推荐

- 567，最后一块石头的重量
- 557，动态规划解戳气球
- 531，BFS和动态规划解完全平方数
- 530，动态规划解最大正方形

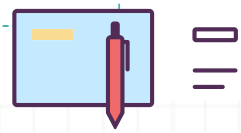
559，动态规划解不相交的线

原创 博哥 数据结构和算法 5月27日

收录于话题
#算法图文分析

161个 >

> Victory belongs to those who believe in it the most and believe in it the longest.
>
> 胜利属于那些意志坚强、持之以恒的人。



问题描述

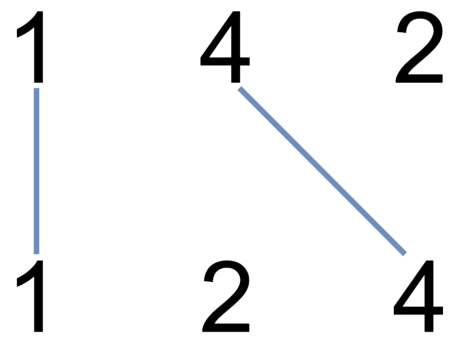
来源：LeetCode第1035题
难度：中等

在两条独立的水平线上按给定的顺序写下nums1和nums2中的整数。现在，可以绘制一些连接两个数字nums1[i]和nums2[j]的直线，这些直线需要同时满足满足：

- **nums1[i] == nums2[j]**
- **且绘制的直线不与任何其他连线（非水平线）相交。**

请注意，连线即使在端点也不能相交：每个数字只能属于一条连线。
以这种方法绘制线条，并返回可以绘制的最大连线数。

示例 1：



> **输入**：nums1 = [1,4,2], nums2 = [1,2,4]
> **输出**：2
> **解释**：可以画出两条不交叉的线，如上图所示。
> 但无法画出第三条不相交的直线，因为从 nums1[1]=4 到 nums2[2]=4 的直线将与从 nums1[2]=2 到 nums2[1]=2 的直线相交。

示例 2：

> **输入**：nums1 = [2,5,1,2,5],
> nums2 = [10,5,2,1,5,2]
> **输出**：3

示例 3：

> **输入**：nums1 = [1,3,7,1,7,5],
> nums2 = [1,9,2,5,1]
> **输出**：2

提示：

- 1 <= nums1.length <= 500
- 1 <= nums2.length <= 500
- 1 <= nums1[i], nums2[i] <= 2000

动态规划解决

定义**dp[i][j]**表示数组nums1的前i个元素和nums2的前j个元素所能绘制的**最大连接数**，如果要求dp[i][j]，我们首先判断nums1的第i个元素和nums2的第j个元素是否相等

如果相等，说明nums1的第i个元素可以和nums2的第j个元素可以连成一条线，这个时候nums1的前i个元素和nums2的前j个元素所能绘制的最大连接数就是nums1的前i-1个元素和nums2的前j-1个元素所能绘制的最大连接数**加1**，也就是**dp[i][j]=dp[i-1][j-1]+1**;

如果不相等，我们就把nums1去掉一个元素，计算nums1的前i-1个元素和nums2的前j个元素能绘制的最大连接数，也就是**dp[i-1][j]**，或者把nums2去掉一个元素，计算nums2的前j-1个元素和nums1的前i个元素能绘制的最大连接数，也就是**dp[i][j-1]**，这两种情况我们取最大的即可，所以我们可以找出递推公式

```
if(nums[i] == nums[j])
    dp[i][j] = dp[i-1][j-1] + 1;
else
    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
```

来看下最终代码

```
public int maxUncrossedLines(int[] nums1, int[] nums2) {
  int m = nums1.length, n = nums2.length;
  //这里为了方便计算，减少一些边界条件的判断，把dp的宽高都增加1
  int dp[][] = new int[m + 1][n+1];
  for (int i = 1; i <= m; ++i)
    for (int j = 1; j <= n; ++j)
      //下面是递推公式
      if (nums1[i - 1] == nums2[j - 1])
        dp[i][j] = dp[i - 1][j - 1] + 1;
      else
        dp[i][j] = Math.max(dp[i][j - 1], dp[i - 1][j]);
  return dp[m][n];
}
```

动态规划代码优化

上面代码中计算**当前值的时候只会和左边，上边，左上边**这3个位置的值有关，和其他的值无关，所以没必要使用二维数组，我们可以改为一维数组，怎么改我们画个图看一下

如果只是

`dp[i][j] = Math.max(dp[i][j - 1], dp[i - 1][j]),`

我们直接把前面的一维去掉即可，其他的不需要修改，即

`dp[j] = Math.max(dp[j - 1], dp[j]);`

如下图所示，



dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);　　dp[j] = Math.max(dp[j], dp[j - 1]);

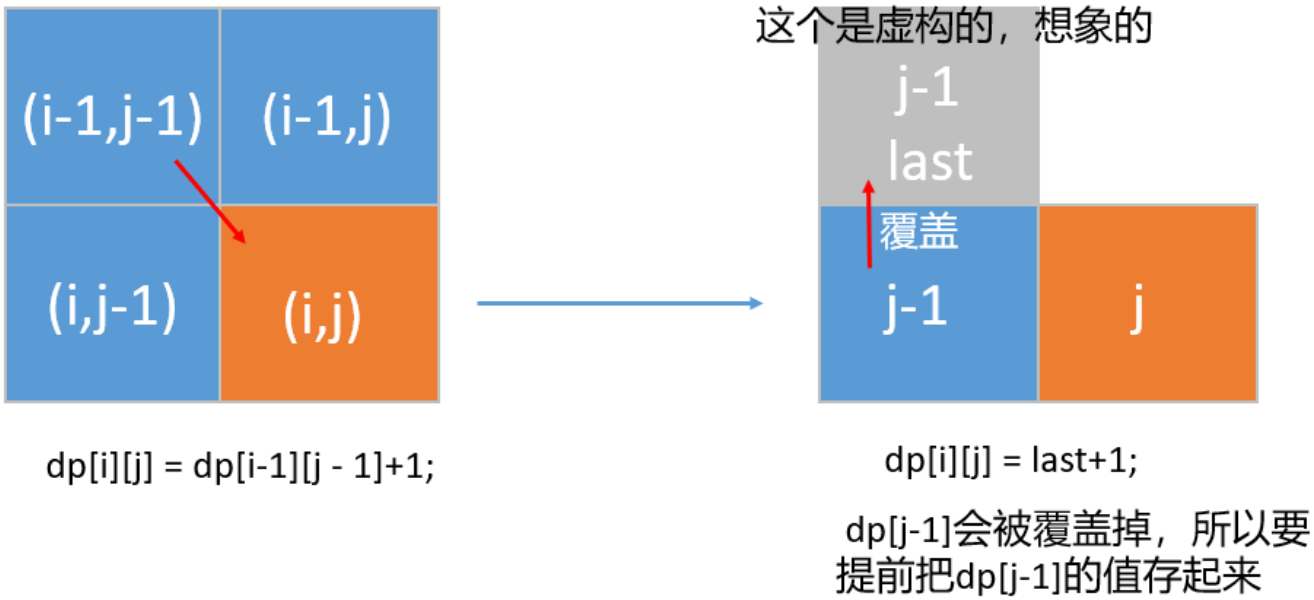
如果

`dp[i][j] = dp[i - 1][j - 1] + 1`

上面方式就行不通了，如果改为

`dp[j] = dp[j - 1] + 1,`

我们发现这个dp[j - 1] 并不是之前的那个dp[j - 1]，在前一步计算的时候已经被覆盖掉了，所以我们需要一个变量在计算dp[j - 1]的值之前先要把dp[j - 1]的值给存起来，如下图所示



来看下代码。

```
1   public int maxUncrossedLines(int[] nums1, int[] nums2) {
2     int m = nums1.length, n = nums2.length;
3     int dp[] = new int[n + 1];
4     for (int i = 1; i <= m; ++i) {
5       int last = dp[0];
6       for (int j = 1; j <= n; ++j) {
7         //dp[j]计算过会被覆盖，这里先把他存储起来
8         int temp = dp[j];
9         //下面是递推公式
10        if (nums1[i - 1] == nums2[j - 1])
11          dp[j] = last + 1;
12        else
13          dp[j] = Math.max(dp[j - 1], dp[j]);
14        last = temp;
15      }
16    }
17    return dp[n];
18  }
```

总结

这题其实就是求最长公共子序列问题，只不过换了一种说法。

往期推荐

- 557，动态规划解戳气球
- 553，动态规划解分割回文串 II
- 552，动态规划解统计全为1的正方形子矩阵
- 370，最长公共子串和子序列

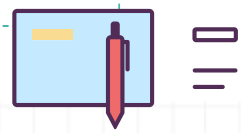
557，动态规划解戳气球

原创 博哥 数据结构和算法 5月24日

收录于话题
#算法图文分析

161个 >

> Any mind that is capable of a real sorrow is capable of good.
>
> 真正悲伤过的人都是心存善念的。



问题描述

来源：LeetCode第312题
难度：困难

有n个气球，编号为0到n-1，每个气球上都标有一个数字，这些数字存在数组nums中。

现在要求你戳破所有的气球。戳破第i个气球，你可以获得nums[i-1]*nums[i]*nums[i+1]枚硬币。这里的i-1和i+1代表和i相邻的两个气球的序号。如果i-1或i+1超出了数组的边界，那么就当它是一个数字为1的气球。

求所能获得硬币的最大数量。

示例 1：

> **输入**：nums = [3,1,5,8]
> **输出**：167
> **解释**：
> nums=[3,1,5,8]-->[3,5,8]-->[3,8]-->[8]-->[]
> coins=3*1*5+3*5*8+1*3*8+1*8*1=167

示例 2：

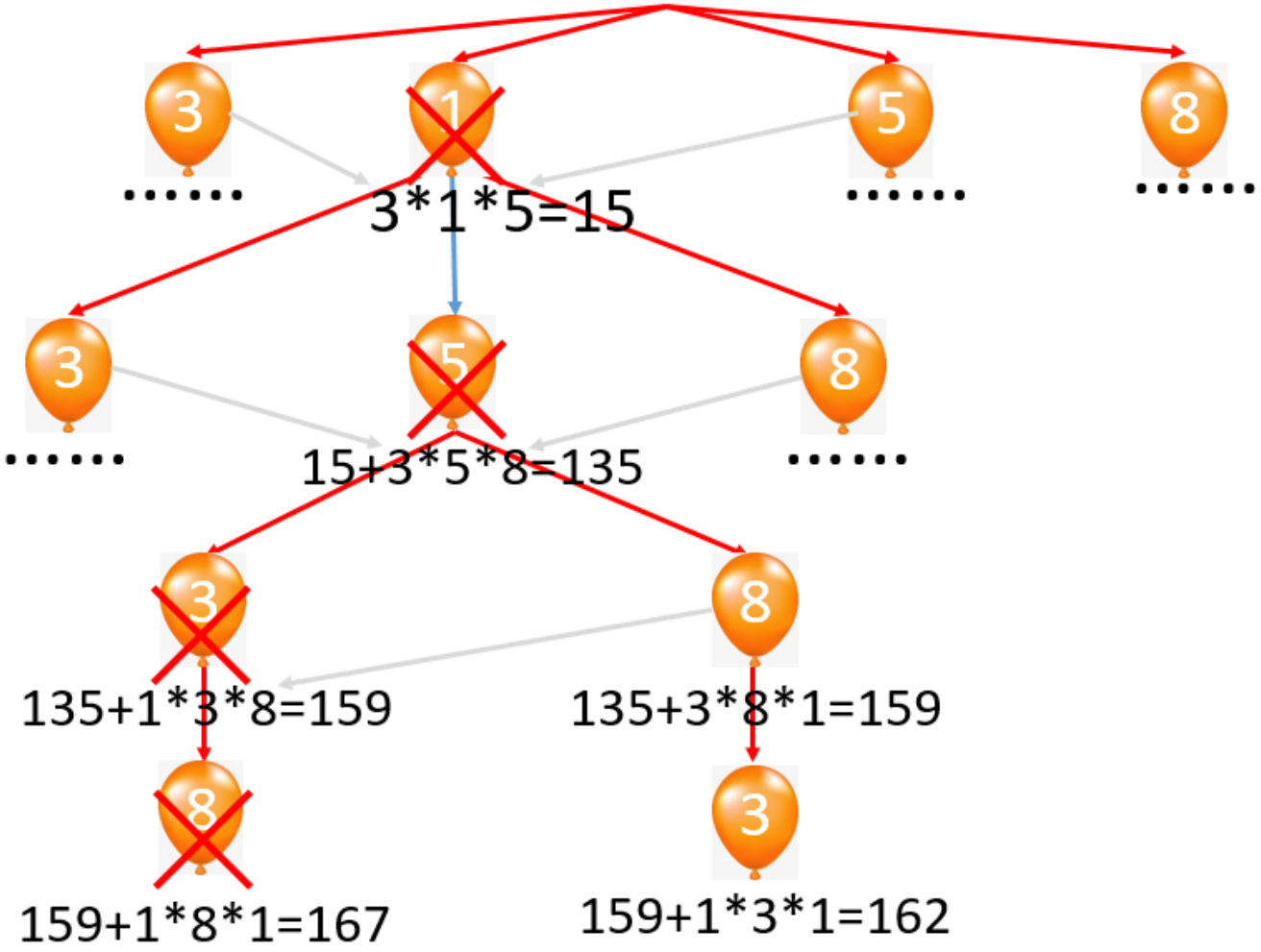
> **输入**：nums = [1,5]
> **输出**：10

提示：

- n == nums.length
- 1 <= n <= 500
- 0 <= nums[i] <= 100

递归解决

一般情况下我们会考虑到每一种可能的结果，然后计算他们的值，最后保留最大的即可，因为可选择的结果很多，下图只画出了最大的那个分支，来看下



比如上图中3和5本来是不相邻的，但当我们戳破1的时候，3和5变成了相邻，所以我们可以使用一个list把每个气球的值都存起来，戳破的时候就把他从list中给删除。来看下代码

```
1   public int maxCoins(int[] nums) {
2     List<Integer> list = new LinkedList<>();
3     //先把nums数组中的元素放到list中
4     for (int num : nums) {
5       list.add(num);
```

```
    }
    return helper(list);
  }

  public int helper(List<Integer> list) {
    //如果是空，则表示没有气球了，直接返回0
    if (list.isEmpty())
      return 0;
    int max = 0;
    int size = list.size();
    for (int i = 0; i < size; ++i) {
      //戳破当前气球所获得的硬币数量
      int sum = 0;
      if (size == 1) {
        //如果只有一个气球，直接戳破他获取的硬币就是
        //当前气球上的数字
        sum = list.get(i);
      } else if (size == 2) {
        //如果有两个气球，戳破任何一个气球都是他俩的乘积
        sum = list.get(0) * list.get(1);
      } else {
        //如果是3个和3个以上的气球要注意戳破第1个和最后一个气球
        //的边界条件
        if (i == 0) {
          sum = list.get(i) * list.get(i + 1);
        } else if (i == size - 1) {
          sum = list.get(i) * list.get(i - 1);
        } else {
          sum = list.get(i - 1) * list.get(i) * list.get(i + 1);
        }
      }
      //把当前气球戳破了，就把他从list中移除
      Integer num = list.remove(i);
      //记录最大的值，sum表示戳破当前气球所得到的硬币数，
      //helper(list)表示戳破剩下的气球所获得的硬币数
      max = Math.max(max, sum + helper(list));
      //把当前气球添加进list，尝试戳破下一个气球
      list.add(i, num);
    }
    return max;
  }
```

当数据量比较大的时候，上面代码很容易超时。我们来换一种解法，定义函数helper(left,right)表示戳破**开区间**(left,right)中的所有气球所获得的最大硬币数。这里之所以定义开区间是因为我们戳破区间(left,right)中的任何气球都能找到两边的气球。假如是闭区间[left,right]，当我们戳破两边的气球的时候我们是不知道前面一个或者后面一个气球的值。

因为是开区间，当left+1>=right的时候，区间内是没有气球的。这里为了方便计算，我们来申请一个比原来数组长度大2的临时数组，临时数组的第1个和最后一个元素都是1。来看下代码

```
  public int maxCoins(int[] nums) {
    int length = nums.length;
    //申请一个比原来长度大2的临时数组
    int[] temp = new int[length + 2];
    //临时数组的首尾值赋为1，中间的值和nums的值一样
    for (int i = 1; i <= length; i++)
      temp[i] = nums[i - 1];
    temp[0] = temp[length + 1] = 1;
    return helper(temp, 0, length + 1);
  }

```

```
//戳破开区间(left,right)中所有气球所获得的最大硬币
public int helper(int[] temp, int left, int right) {
  //如果(left，right)之间没有气球，返回0，表示没有任何气球可戳破
  if (left + 1 == right)
    return 0;
  int res = 0;
  //在(left，right)中我们尝试戳破每一个位置的气球，取最大值即可
  for (int i = left + 1; i < right; ++i) {
    int sum = temp[left] * temp[i] * temp[right] + helper(temp, left, i) + helper(temp, i, right);
    res = Math.max(res, sum);
  }
  return res;
}
```

这样当数据量比较大的时候还是会超时，因为这里包含了大量的重复计算，我们还可以使用一个map，把计算的结果存到map中，下次使用的时候如果计算过了，就直接从map中取，来看下代码

```
public int maxCoins(int[] nums) {
  int length = nums.length;
  //申请一个比原来长度大2的临时数组
  int[] temp = new int[length + 2];
  //临时数组的首尾值赋为1，中间的值和nums的值一样
  for (int i = 1; i <= length; i++)
    temp[i] = nums[i - 1];
  temp[0] = temp[length + 1] = 1;
  int[][] map = new int[length + 2][length + 2];
  return helper(map, temp, 0, length + 1);
}

public int helper(int[][] map, int[] temp, int left, int right) {
  //如果(left，right)之间没有气球，返回0，表示没有任何气球可戳破
  if (left + 1 == right)
    return 0;
  //如果已经计算过了，就直接从map中取
  if (map[left][right] > 0)
    return map[left][right];
  int res = 0;
  for (int i = left + 1; i < right; ++i) {
    int sum = temp[left] * temp[i] * temp[right] + helper(map, temp, left, i) + helper(map, temp, i, r
    res = Math.max(res, sum);
  }
  //把计算的结果在存储到map中
  map[left][right] = res;
  return res;
}
```

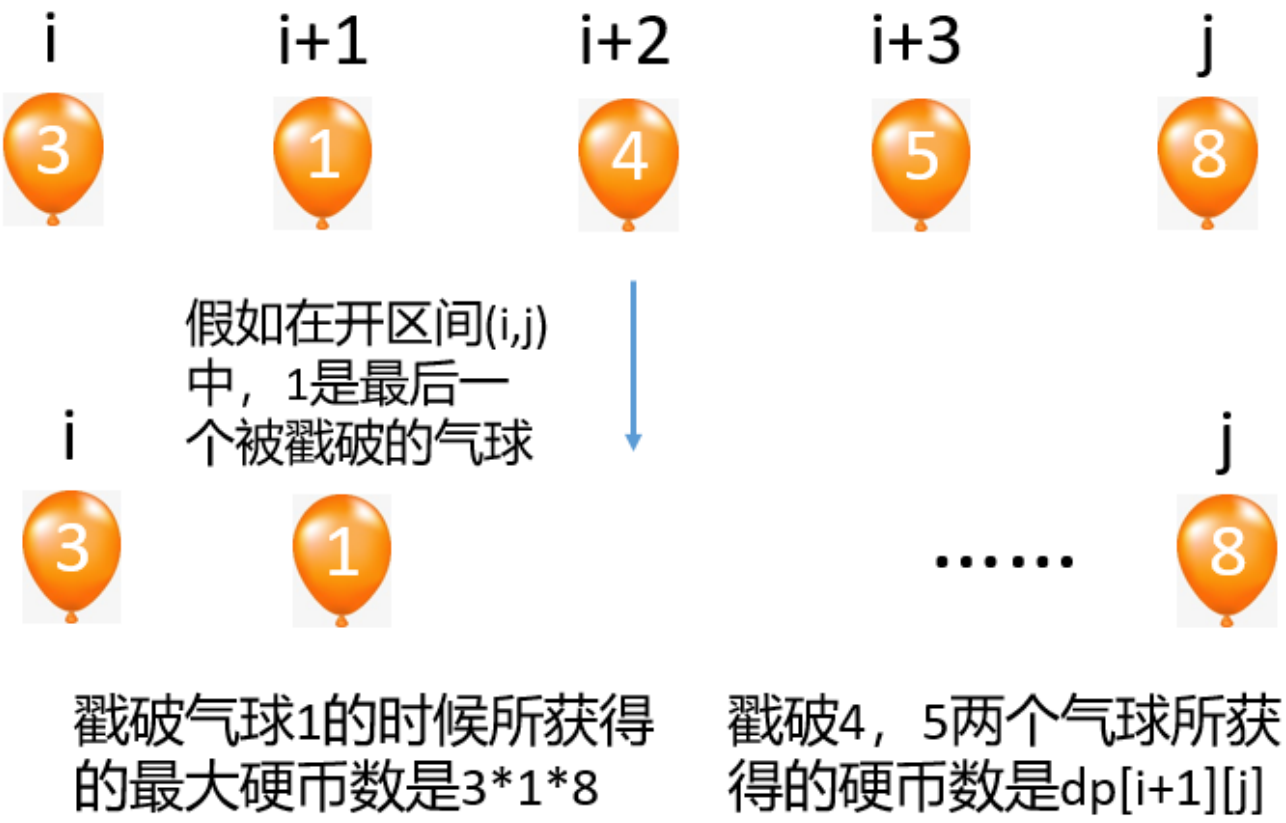
动态规划解决

我们定义dp[i][j]表示戳破开区间(i,j)中所有气球所获得的最大硬币，为了方便计算我们还是来申请一个临时数组，他的长度比原数组长度大2，那么这题就可以变成求戳破开区间(0,length+1)中所有气球所获得的最大硬币数，像下面这样

```
 for (int i = length - 1; i >= 0; i--) {
   for (int j = i + 2; j <= length + 1; j++) {
     //计算戳破开区间(i,j)中间的气球所获取的最大硬币数
     ……
   }
 }
```

如果一个气球被戳破后，本来不挨着的两个气球可能变成挨着的了，这样还是不好计算。我们可以这样来计算，在开区间(i,j)中随便找一个气球，假如是k，我们让k成为最后一个被戳破的气球，那么这样就简单了，当我们戳破气球k的时候所获得的最大硬币是**nums[i]*nums[k]*nums[j];**

如下图所示



这个k可以是开区间(i,j)中的任何一个气球，取最大的即可，所以我们可以找出递推公式，在开区间(i,j)中戳破第k个气球所获得硬币数

sum=temp[i]*temp[k]*temp[j]+dp[i][k]+dp[k][j];

其中dp[i][k]表示戳破开区间(i,k)中的气球所获得的最大硬币数量，同理dp[k][j]，来看下代码

```
public int maxCoins(int[] nums) {
  int length = nums.length;
  //申请一个比原来长度大2的临时数组
  int[] temp = new int[length + 2];
  //临时数组的首尾值赋为1，中间的值和nums的值一样
  for (int i = 1; i <= length; i++)
    temp[i] = nums[i - 1];
  temp[0] = temp[length + 1] = 1;
  //dp[i][j]表示戳破开区间(i,j)之间的气球所获得的最大硬币数
  int[][] dp = new int[length + 2][length + 2];
  for (int i = length - 1; i >= 0; i--) {
    for (int j = i + 2; j <= length + 1; j++) {
      //遍历开区间(i,j)中的每一个气球，尝试戳破他所获得的最大硬币
      for (int k = i + 1; k < j; k++) {
        //递推公式，戳破气球k可以获得temp[i] * temp[k] * temp[j]个硬币，
        //dp[i][k]表示戳破开区间(i,k)之间的气球所获得的硬币数
        //dp[k][j]表示戳破开区间(k,j)之间的气球所获得的硬币数
        int sum = temp[i] * temp[k] * temp[j] + dp[i][k] + dp[k][j];
        //保留最大的
        dp[i][j] = Math.max(dp[i][j], sum);
      }
```

```
22          }
23      }
24      return dp[0][length + 1];
25  }
```

往期推荐

- 553，动态规划解分割回文串 II
- 552，动态规划解统计全为1的正方形子矩阵
- 540，动态规划和中心扩散法解回文子串
- 530，动态规划解最大正方形

553，动态规划解分割回文串 II

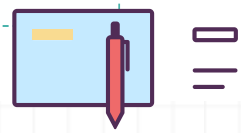
原创 博哥 数据结构和算法 5月16日

收录于话题
#算法图文分析

161个 >

When a man's stories are remembered, then he is immortal.

一个人的故事被记住了，他就千古不朽。



问题描述

来源：LeetCode第132题
难度：困难

给你一个字符串s，请你将s分割成一些子串，使每个子串都是回文。
返回符合要求的最少分割次数 。

示例 1：

> **输入**：s = "aab"
> **输出**：1
> **解释**：只需一次分割就可将 s 分割成 ["aa","b"] 这样两个回文子串。

示例 2：

> **输入**：s = "a"
> **输出**：0

示例 3：

> **输入**：s = "ab"
> **输出**：1

提示：

- 1 <= s.length <= 2000
- s 仅由小写英文字母组成

动态规划解决

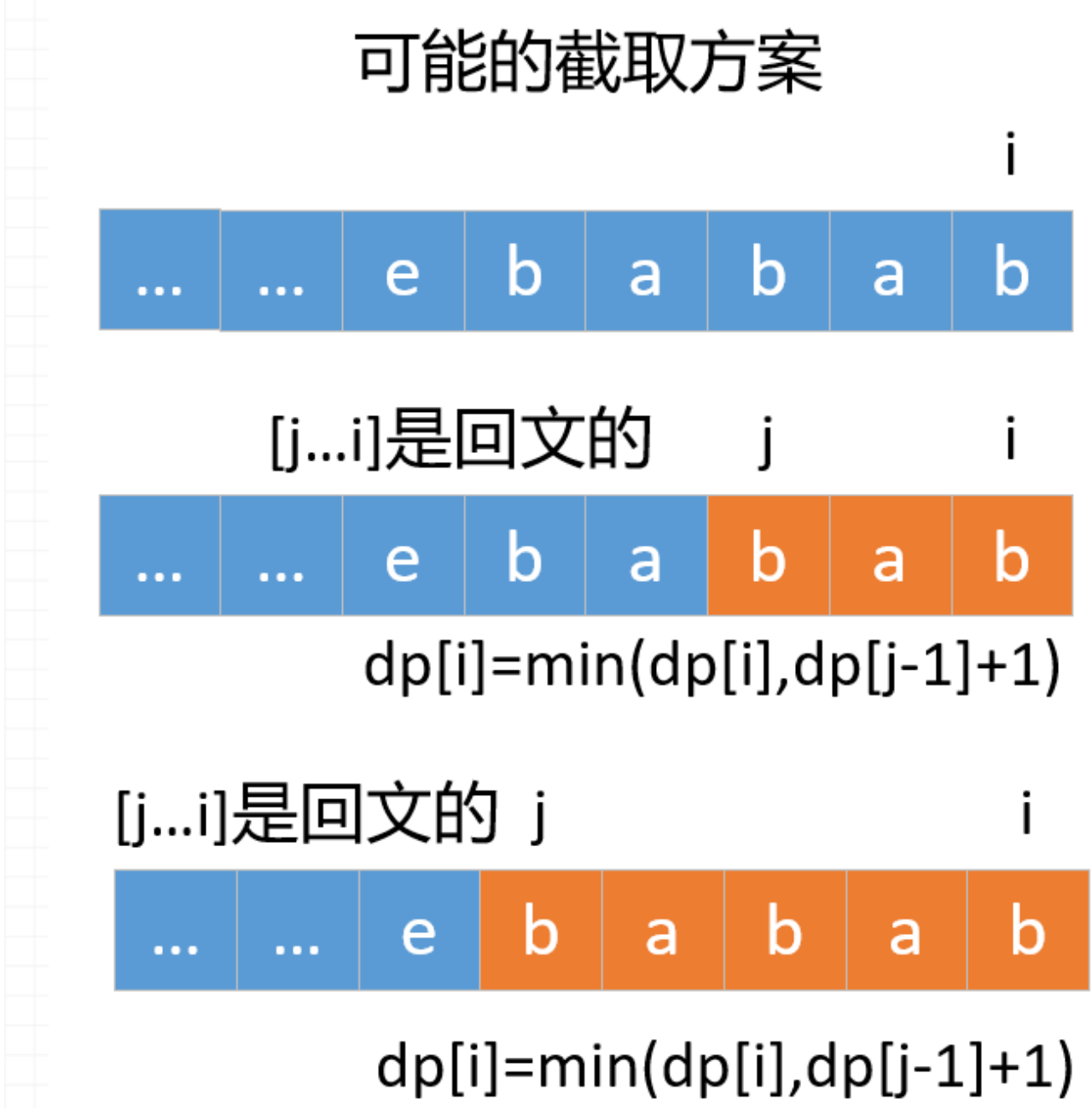
前面刚讲过《551，回溯算法解分割回文串》，第551要求返回所有可能分隔的结果，而这题要求返回最小的分隔次数。如果数据量不大的话我们是可以使用第551题的答案的，找出第551题所有可能分隔的方案中最小的分隔次数就是我们这题的答案。

但这题提示中明显数量比较大，所以使用回溯算法是行不通的，我们可以改成动态规划来解决。

定义dp[i]表示字符串前i个字符构成的子串能分隔的最少分隔次数，如果要求s的最少分隔次数，只需要返回dp[length-1]即可（length是字符串s的长度）。

1，如果子串从[0...i]是回文是，就不需要分隔，即
dp[i]=0;

2，如果子串从[0...i]不是回文的，我们就需要从子串[0...i]后面截取一个回文的子串[j...i]，所以子串[0...i]可以分隔为[0...j-1]和[j...i]，而[0...j-1]的最小分隔次数就是dp[j-1]，[j...i]是一个回文串，不需要分隔，所以dp[i]=dp[j-1]+1，因为要求的是截取的最小次数，所以我们只需要保留最小的即可，即
dp[i]=min(dp[i],dp[j-1]+1);
如下图所示



因为这里求最小值，默认值我们给他每个都初始化最大的，来看下代码

```
public int minCut(String s) {
  int length = s.length();
  int[] dp = new int[length];
  //字符串s的回文子串最大也只能是字符串的长度length，
  //所以这里都默认初始化为最大值
  Arrays.fill(dp, length);
  for (int i = 0; i < length; ++i) {
    //如果字符串从0到i本身就是一个回文的，就不需要分隔，
    //直接返回0
    if (palindrome(s, 0, i)) {
      dp[i] = 0;
    } else {
      //否则就要分隔，找出最小的分隔方案
      for (int j = 1; j <= i; ++j) {
        if (palindrome(s, j, i)) {
          dp[i] = Math.min(dp[i], dp[j - 1] + 1);
        }
      }
    }
  }
  return dp[length - 1];
}

```

```
  //判断从left到right之间的子串是否是回文的（使用双指针判断）
  private boolean palindrome(String s, int left, int right) {
    while (left < right) {
      if (s.charAt(left++) != s.charAt(right--))
        return false;
    }
    return true;
  }
```

动态规划代码优化

上面代码虽然也能给解决，但每次截取的时候都要判断一遍是否是回文的，明显效率不是很高，我们还可以先计算他的每个子串是否是回文的，在截取的时候直接用即可，回文串的判断前面刚介绍过，《540，动态规划和中心扩散法解回文子串》，这里就不在重复介绍。我们可以先把代码复制过来

```
    int length = s.length();
    //判断子串[i…j]是否是回文串
    boolean[][] dp = new boolean[length][length];
    for (int j = 0; j < length; j++) {
      for (int i = 0; i <= j; i++) {
        //如果i和j指向的字符不一样，那么dp[i][j]就
        //不能构成回文字符串
        if (s.charAt(i) != s.charAt(j))
          continue;
        dp[i][j] = j - i <= 2 || dp[i + 1][j - 1];
      }
    }

```

再来看下最终代码

```
  public int minCut(String s) {
    int length = s.length();
    int[] dp = new int[length];

    //判断子串[i…j]是否是回文串
    boolean[][] palindrome = new boolean[length][length];
    for (int j = 0; j < length; j++) {
      for (int i = 0; i <= j; i++) {
        //如果i和j指向的字符不一样，那么dp[i][j]就
        //不能构成回文字符串
        if (s.charAt(i) != s.charAt(j))
          continue;
        palindrome[i][j] = j - i <= 2 || palindrome[i + 1][j - 1];
      }
    }

    //字符串s的回文子串最大也只能是字符串的长度length，
    //所以这里都默认初始化为最大值
    Arrays.fill(dp, length);
    for (int i = 0; i < length; ++i) {
      //如果字符串从0到i本身就是一个回文的，就不需要分隔，
      //直接返回0
      if (palindrome[0][i]) {
        dp[i] = 0;
      } else {
        //否则就要分隔，找出最小的分隔方案
        for (int j = 1; j <= i; ++j) {
          if (palindrome[j][i]) {
            dp[i] = Math.min(dp[i], dp[j - 1] + 1);
          }
        }
      }
    }
```

```
34      return dp[length - 1];
35  }
```

往期推荐

- 540，动态规划和中心扩散法解回文子串
- 529，动态规划解最长回文子序列
- 517，最长回文子串的3种解决方式
- 497，双指针验证回文串

552，动态规划解统计全为1的正方形子矩阵

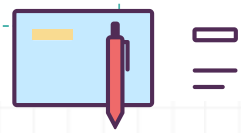
原创 博哥 数据结构和算法 5月14日

收录于话题
#算法图文分析

161个 >

If well used, books are the best of all things; if abused, among the worst.

如果利用得当，书籍就是最好的朋友；反之，如果滥用，它就会变成最坏的东西了。



问题描述

来源：LeetCode第1277题
难度：中等

给你一个m*n的矩阵，矩阵中的元素不是0就是1，请你统计并返回其中完全由1组成的**正方形**子矩阵的个数。

示例 1：

```
输入：matrix =
[
  [0,1,1,1],
  [1,1,1,1],
  [0,1,1,1]
]
输出：15
解释：
边长为 1 的正方形有 10 个。
边长为 2 的正方形有 4 个。
边长为 3 的正方形有 1 个。
正方形的总数 = 10 + 4 + 1 = 15.
```

示例 2：

```
输入：matrix =
[
  [1,0,1],
  [1,1,0],
  [1,1,0]
]
输出：7
解释：
边长为 1 的正方形有 6 个。
边长为 2 的正方形有 1 个。
正方形的总数 = 6 + 1 = 7.
```

提示：

- 1 <= arr.length <= 300
- 1 <= arr[0].length <= 300
- 0 <= arr[i][j] <= 1

动态规划解决

这题和《530，动态规划解最大正方形》解法类似，不过不同的是第530题让求的是最大正方形的面积，而这题要求的是正方形的个数。我们还按照第530题的方式来解

定义二维数组dp[i][j]，其中dp[i][j]表示的是在矩阵中以坐标(i,j)为右下角的最大正方形边长。

那么递推公式就是（不明白的可以看下第530题，这里就不在重复介绍）
dp[i][j]=min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1;

并且以坐标(i,j)为右下角的正方形个数就是dp[i][j]，画个图来看一下



所以这题就简单了，我们只需要把530题的答案拿过来，然后把所有dp[i][j]值不等于0的累加即可，来看下代码

```
public int countSquares(int[][] matrix) {
  int count = 0;//正方形的个数
  int[][] dp = new int[matrix.length + 1][matrix[0].length + 1];
  for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[0].length; j++) {
      //如果当前坐标是0，就不可能构成正方形，直接跳过
      if (matrix[i][j] == 0)
        continue;
      //递推公式
      dp[i + 1][j + 1] = Math.min(Math.min(dp[i + 1][j], dp[i][j + 1]), dp[i][j]) + 1;
      //累加所有的dp值
      count += dp[i + 1][j + 1];
    }
  }
  return count;
}

```

总结

搞懂了《530，动态规划解最大正方形》，这题就非常简单了，这里需要理解的是以坐标(i,j)为右下角的最大正方形边长就是以(i,j)为右下角正方形的个数。

548，动态规划解最长的斐波那契子序列的长度

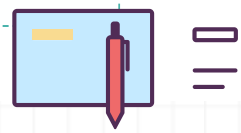
原创 博哥 数据结构和算法 5月8日

收录于话题
#算法图文分析

161个 >

This is the last game so make it count, it's now or never.

不要辜负这最后的比赛，时不我待。



问题描述

如果序列X_1,X_2,...,X_n满足下列条件，就说它是斐波那契式的：

- n>=3
- 对于所有i+2<=n，都有X_i+X_{i+1}=X_{i+2}

给定一个**严格递增的正整数数组**形成序列，找到A中最长的斐波那契式的子序列的长度。如果一个不存在，返回0。

（回想一下，子序列是从原序列A中派生出来的，它从A中删掉任意数量的元素（也可以不删），而不改变其余元素的顺序。例如，[3,5,8]是[3,4,5,6,7,8]的一个子序列）

示例 1：

> **输入**：[1,2,3,4,5,6,7,8]
> **输出**：5
> **解释**：
> 最长的斐波那契式子序列为：[1,2,3,5,8] 。

示例 2：

> **输入**：[1,3,7,11,12,14,18]
> **输出**：3

> **解释**:
> 最长的斐波那契式子序列有:
> [1,11,12]，[3,11,14] 以及 [7,11,18] 。

提示:

- 3<=A.length<=1000
- 1<=A[0]<A[1]<...<A[A.length-1]<=10^9
- （对于以Java，C，C++，以及C#的提交，时间限制被减少了50%）

暴力解决

斐波那契数列满足的条件是前两项的和等于第3项的值。暴力求解的方式就是先固定第1项和第2项的值，然后来查找第3项，如果数组中存在第3项的值，说明这3项可以构成一个斐波那契数列，然后在更新第1项和第2项的值，更新结果是
(first, second) -> (second, first+second)
接着重复上面的判断……记录最长的即可。

因为题中说了是一个递增的正整数数组，所以我们可以先把数组中的元素全部存放到集合Set中，查找第3项的时候直接从集合Set中查找即可，来看下代码。

```
1   public int lenLongestFibSubseq(int[] A) {
2     int size = A.length;
3     //先把数组A中的所有元素都存储在Set中
4     Set<Integer> set = new HashSet<>();
5     for (int num : A)
6       set.add(num);
7     //记录构成的最长斐波那契数列的长度
8     int maxLength = 0;
9     for (int i = 0; i < size; ++i)
10      for (int j = i + 1; j < size; ++j) {
11        //斐波那契数列的第一项
12        int first = A[i];
13        //斐波那契数列的第二项
14        int second = A[j];
15        //当前斐波那契数列的长度
16        int len = 2;
17        //斐波那契数列前两项和等于第三项的值，这里
18        //判断数组A中是否存在前两项的和
19        while (set.contains(first + second)) {
20          //如果能构成斐波那契数列，我们需要更新后面的值，
21          //(first, second) -> (second, first+second)
22          second = first + second;
23          first = second - first;
24          //记录当前能构成的斐波那契数列的长度
25          len++;
26        }
27        //更新最大的斐波那契数列长度
28        if (len > maxLength)
29          maxLength = len;
30      }
31    //能构成斐波那契数列，长度必须大于等于3，如果小于3，
32    //说明不能构成斐波那契数列，直接返回0，否则返回maxLength
33    return maxLength >= 3 ? maxLength : 0;
34  }
```

动态规划解决

动态规划首先要找出**状态的定义**和**状态转移公式**。定义dp[i][j]表示以A[i]和A[j]结尾的斐波那契数列的最大长度。也就是[……，A[i]，A[j]]构成的斐波那契数列的最大长度。

状态转移公式就是，如果以A[i]和A[j]结尾能构成斐波那契数列，那么在这个数列A[i]之前必定有一个值A[k]满足A[k]+A[i]=A[j]；所以如果确定了A[i]和A[j]的值之后，我们可以往前来找A[k]，那么转移公式就是

dp[i][j]=dp[k][i]+1;

所以大致代码我们就很容易写出来了。

```
for (int j = 2; j < length; j++) {//确定A[j]
  for (int i = j - 1; i > 0; i--) {//确定A[i]
    for (int k = i - 1; k >= 0; k--) {//往前查找A[k]
      if (A[k] + A[i] == A[j]) {
        dp[i][j] = dp[k][i] + 1;
        //如果找到就终止内层循环，不在往前查找了
        break;
      } else if (A[k] + A[i] < A[j]) {
        //题中说了是递增的正整数数组，如果当前A[k]
        //小了，那么前面的就更小，没有合适的，没必要
        //在往前找了，直接终止内层循环
        break;
      }
    }
    maxLength = Math.max(maxLength, dp[i][j]);
  }
}
```

我们来看下最终代码

```
public int lenLongestFibSubseq(int[] A) {
  //记录最大的斐波那契数列的长度
  int maxLength = 0;
  int length = A.length;
  int[][] dp = new int[length][length];
  for (int j = 2; j < length; j++) {//确定A[j]
    for (int i = j - 1; i > 0; i--) {//确定A[i]
      for (int k = i - 1; k >= 0; k--) {//往前查找A[k]
        if (A[k] + A[i] == A[j]) {
          dp[i][j] = dp[k][i] + 1;
          //如果找到就终止内层循环，不在往前查找了
          break;
        } else if (A[k] + A[i] < A[j]) {
          //题中说了是递增的正整数数组，如果当前A[k]
          //小了，那么前面的就更小，没有合适的，没必要
          //在往前找了，直接终止内层循环
          break;
        }
      }
      maxLength = Math.max(maxLength, dp[i][j]);
    }
  }
  //注意数列长度统计的时候没有统计前面两项，如果能构成
  //斐波那契数列最后还需要加上
  return maxLength > 0 ? maxLength + 2 : 0;
}
```

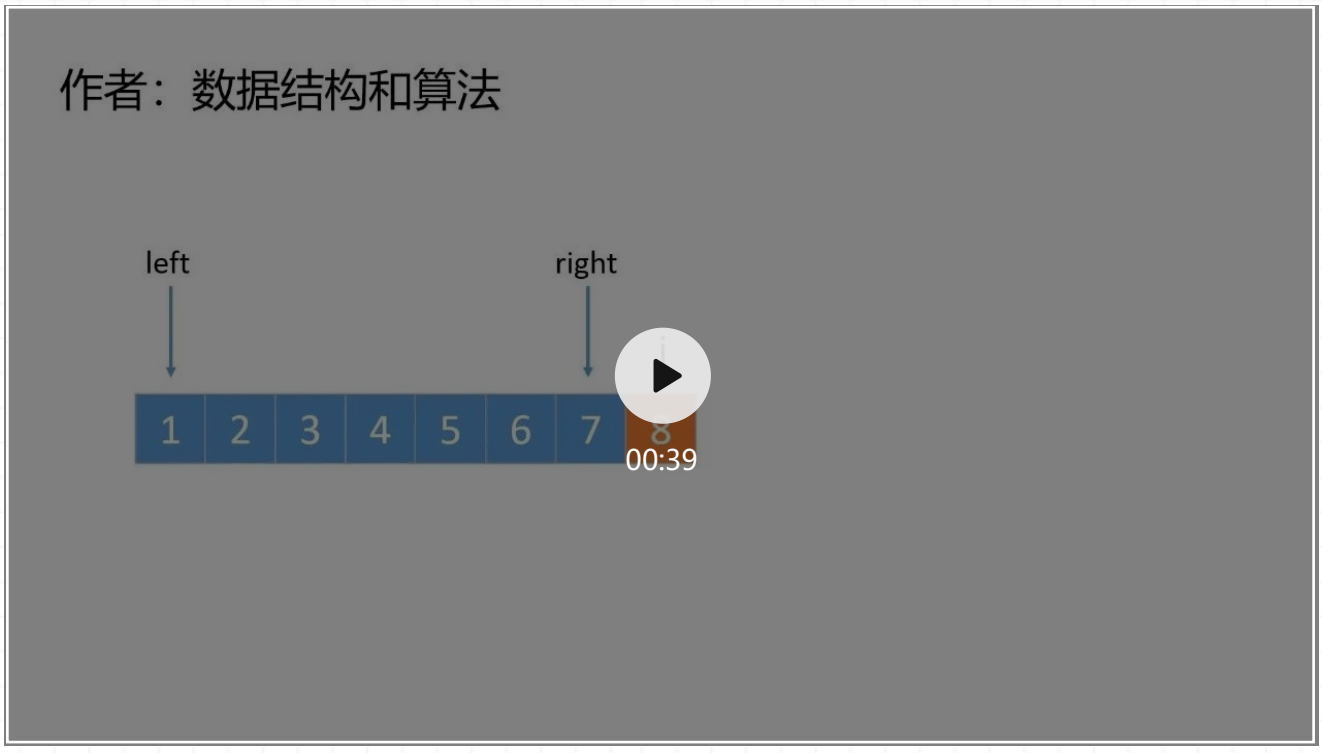
动态规划代码优化

上面代码确定A[j]和A[i]，查找A[k]的时候是一个个往前查，整个计算时间复杂度比较高，我们还可以把遍历过的A[j]和他对应的下标存放到map中，在查找的时候直接从map中查找，这样时间复杂度就会从O(n^3)降到O(n^2)，来看下代码。

```
public int lenLongestFibSubseq(int[] A) {
  //记录最大的斐波那契数列的长度
  int maxLength = 0;
  int length = A.length;
  int[][] dp = new int[length][length];
  Map<Integer, Integer> map = new HashMap<>();
  for (int j = 0; j < length; j++) {//确定A[j]
    map.put(A[j], j);
    for (int i = j - 1; i > 0; i--) {//确定A[i]
      //因为是递增的，如果A[j]和A[i]之间相差比较大，
      //就不需要再往前查找了
      if (A[j] - A[i] >= A[i])
        continue;
      //通过map往前查找A[k]
      int k = map.getOrDefault(A[j] - A[i], -1);
      //如果k不等于-1，说明在数组中找到了A[k]这个值
      if (k >= 0) {
        dp[i][j] = dp[k][i] + 1;
        maxLength = Math.max(maxLength, dp[i][j]);
      }
    }
  }
  return maxLength > 0 ? maxLength + 2 : 0;
}
```

动态规划+双指针代码优化

对于题中的条件是递增的数量，也就是有序的，所以我们还可以使用双指针来解决，当确定A[j]之后，我们在A[j]的前面来使用两个指针来找和等于A[j]的两个值，这里以示例一为例看下视频



最后再来看下代码

```
public int lenLongestFibSubseq(int[] A) {
```

```
    //记录最大的斐波那契数列的长度
    int maxLength = 0;
    int length = A.length;
    int[][] dp = new int[length][length];
    for (int j = 2; j < length; j++) {//确定A[j]
      //左右两个指针
      int left = 0;
      int right = j - 1;
      while (left < right) {
        //两个指针指向值的和
        int sum = A[left] + A[right];
        if (sum > A[j]) {
          //因为数组是递增的，如果两个指针指向的值
          //大了，那么右指针往左移一步来减小他俩的和
          right--;
        } else if (sum < A[j]) {
          //如果两个指针指向的值小了，那么左指针往
          //右移一步来增大他俩的和
          left++;
        } else {
          //如果两个指针指向的和等于A[j]，说明这两个指针
          //指向的值和A[j]可以构成斐波那契数列
          dp[right][j] = dp[left][right] + 1;
          maxLength = Math.max(maxLength, dp[right][j]);
          right--;
          left++;
        }
      }
    }
    return maxLength > 0 ? maxLength + 2 : 0;
  }
```

往期推荐

- 543，剑指 Offer-动态规划解礼物的最大价值
- 540，动态规划和中心扩散法解回文子串
- 531，BFS和动态规划解完全平方数
- 529，动态规划解最长回文子序列

543，剑指 Offer-动态规划解礼物的最大价值

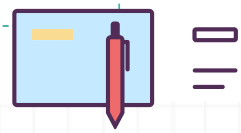
原创 博哥 数据结构和算法 今天

收录于话题
#剑指offer

33个 >

A person who won't read has no advantage over one who can't read.

识字却不愿阅读的人，比文盲也好不到哪去。



问题描述

在一个m*n的棋盘的每一格都放有一个礼物，每个礼物都有一定的价值（价值大于0）。你可以从棋盘的**左上角**开始拿格子里的礼物，并每次向右或者向下移动一格、直到到达棋盘的**右下角**。给定一个棋盘及其上面的礼物的价值，请计算你最多能拿到多少价值的礼物？

示例 1：

> **输入**:
> [
> [1,3,1],
> [1,5,1],
> [4,2,1]
>]
> **输出**: 12
> **解释**: 路径 1→3→5→2→1 可以拿到最多价值的礼物

提示：

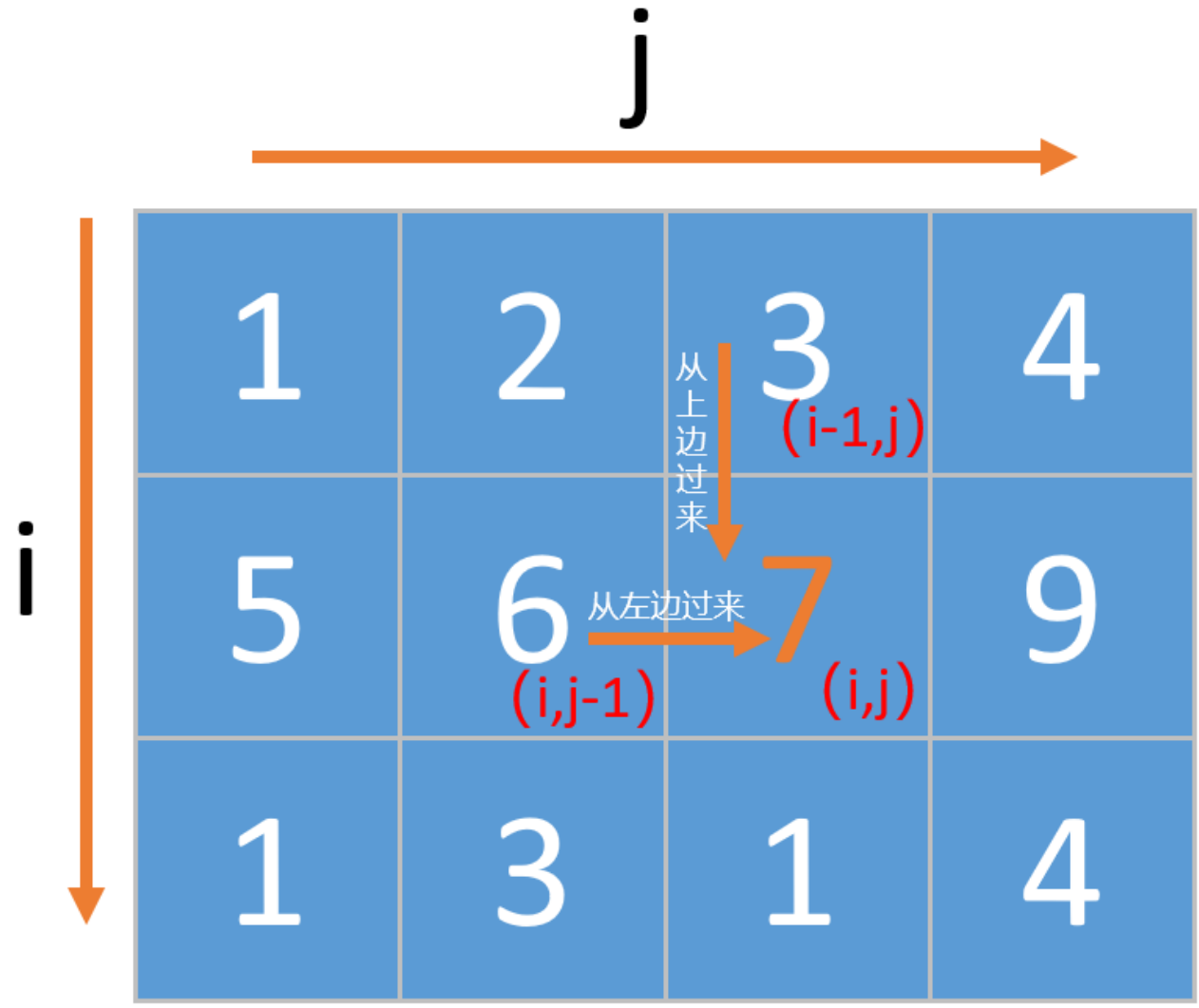
- 0 < grid.length <= 200
- 0 < grid[0].length <= 200

动态规划解决

这题可以参照409，动态规划求不同路径，第409题让求的是有多少种路径，而这题让求的是所有路径中数字和最大的值。这题很容易想到的解决方式就是动态规划。

根据题意要求我们定义dp[i][j]表示从矩阵的左上角走到坐标（i，j）所能拿到的最大礼物。

如果要走到坐标（i，j），我们可以从坐标（i-1，j）往下走一步，或者从坐标（i，j-1）往右走一步，到底应该从哪里，我们应该取这两个方向的最大值，所以
dp[i][j]=max(dp[i-1][j],dp[i][j-1])+grid[i][j];



那么边界条件是什么呢

- 如果在左上角，dp[0][0]=grid[0][0]，
- 如果在最上边一行，因为不能从上面走过来，只能从左边走过来，所以当前值是他左边元素的累加。
- 如果在最左边一列，因为不能从左边走过来，只能从上边走过来，所以当前值是他上边元素的累加。

来看下代码

```
1   public int maxValue(int[][] grid) {
2     //边界条件判断
3     if (grid == null || grid.length == 0)
```

```
        return 0;
    int m = grid.length;
    int n = grid[0].length;
    int[][] dp = new int[m][n];
    dp[0][0] = grid[0][0];
    //初始化dp的最上面一行，从左到右累加
    for (int i = 1; i < n; i++) {
      dp[0][i] = dp[0][i - 1] + grid[0][i];
    }
    //初始化dp的最左边一列，从上到下累加
    for (int i = 1; i < m; i++) {
      dp[i][0] = dp[i - 1][0] + grid[i][0];
    }

    //下面是递推公式的计算
    for (int i = 1; i < m; i++) {
      for (int j = 1; j < n; j++) {
        dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]) + grid[i][j];
      }
    }
    return dp[m - 1][n - 1];
  }
```

为了方便计算我们还可以把dp的宽和高增加1，也就是dp的最上面一行和最左边一列不存储任何数值，他们都是0，这样是为了减少一些判断

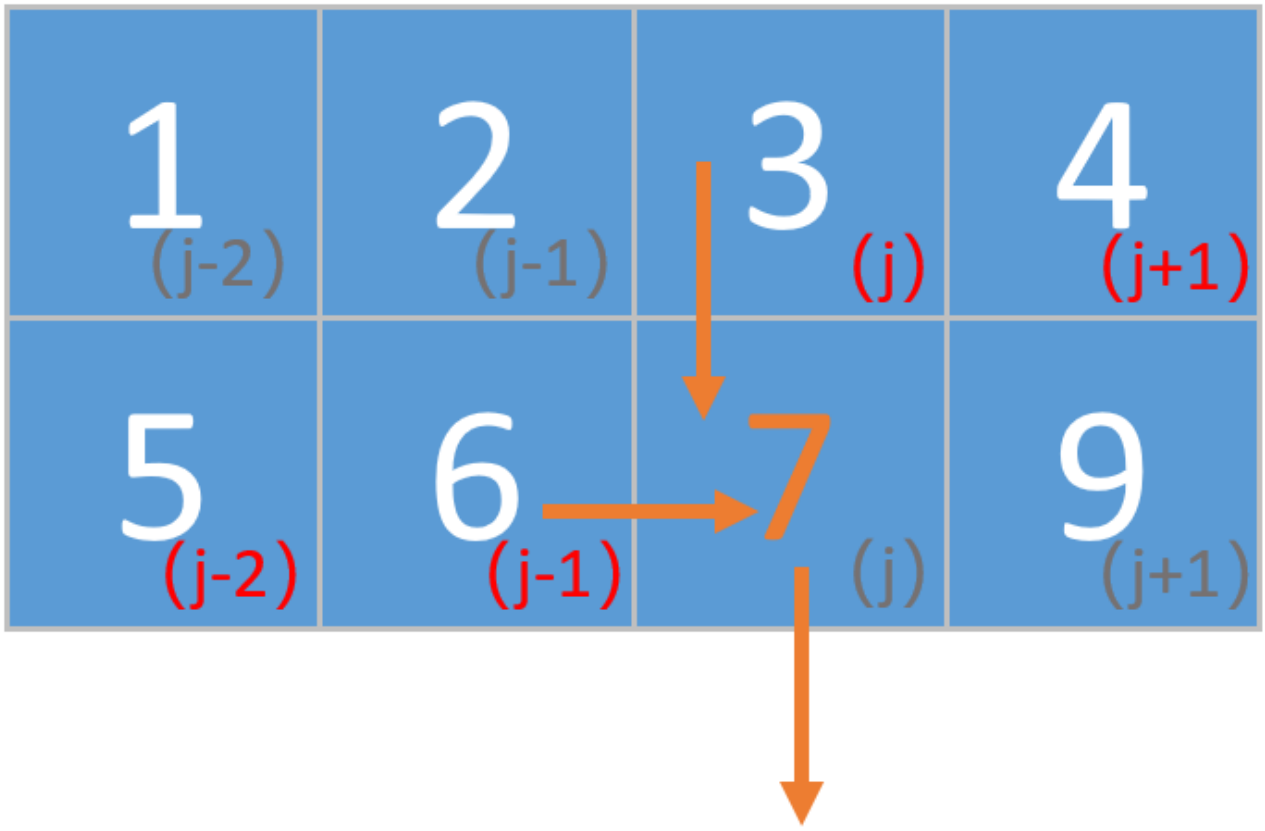
```
  public int maxValue(int[][] grid) {
    if (grid == null || grid.length == 0)
      return 0;
    int m = grid.length;
    int n = grid[0].length;
    //为了方便计算，dp的宽和高都增加了1
    int[][] dp = new int[m + 1][n+1];
    //下面是递推公式的计算
    for (int i = 0; i < m; i++) {
      for (int j = 0; j < n; j++) {
        dp[i + 1][j + 1] = Math.max(dp[i + 1][j], dp[i][j + 1]) + grid[i][j];
      }
    }
    return dp[m][n];
  }
```

动态规划代码优化

我们来看一下这个公式

dp[i][j]=max(dp[i-1][j],dp[i][j-1])+grid[i][j];

我们发现当前值只和左边和上边的值有关，和其他的无关，比如我们在计算第5行的时候，那么很明显第1，2，3行的对我们来说都是无用的，所以我们这里可以把二维dp改成一维的，其中dp[i][j-1]可以用dp[j-1]来表示，就是当前元素前面的，dp[i-1][j]可以用dp[j]来表示，就是上边的元素。



来看下代码

```
public int maxValue(int[][] grid) {
  if (grid == null || grid.length == 0)
    return 0;
  int m = grid.length;
  int n = grid[0].length;
  //数组改成一维的
  int[] dp = new int[n + 1];
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      dp[j + 1] = Math.max(dp[j], dp[j + 1]) + grid[i][j];
    }
  }
  return dp[n];
}
```

我们再来仔细看这道题，题中没说不可以修改原来数组的值，所以我还可以使用题中的二维数组来代替二维dp数组

```
public int maxValue(int[][] grid) {
  //边界条件判断
  if (grid == null || grid.length == 0)
    return 0;
  int m = grid.length;
  int n = grid[0].length;
  //初始化dp的最上面一行，从左到右累加
  for (int i = 1; i < n; i++) {
    grid[0][i] += grid[0][i - 1];
  }
  //初始化dp的最左边一列，从上到下累加
  for (int i = 1; i < m; i++) {
    grid[i][0] += grid[i - 1][0];
  }

  //下面是递推公式的计算
  for (int i = 1; i < m; i++) {
    for (int j = 1; j < n; j++) {
      grid[i][j] = Math.max(grid[i - 1][j], grid[i][j - 1]) + grid[i][j];
```

```
20          }
21      }
22      return grid[m - 1][n - 1];
23  }
```

这样最终我们把空间复杂度从O(MN)降到O(1)。

往期推荐

- 538，剑指 Offer-和为s的连续正数序列
- 537，剑指 Offer-字符串的排列
- 535，剑指 Offer-扑克牌中的顺子
- 533，剑指 Offer-最小的k个数

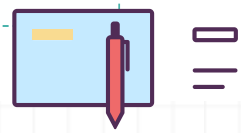
540，动态规划和中心扩散法解回文子串

原创 博哥 数据结构和算法 今天

收录于话题
#算法图文分析

145个 >

> Listen to the pain. It's both history teacher and fortune teller.
>
> 倾听你的痛苦。痛苦是历史老师，也是预言家。



问题描述

给定一个字符串，你的任务是计算这个字符串中有多少个**回文子串**。

具有不同开始位置或结束位置的子串，即使是由相同的字符组成，也会被视作不同的子串。

示例 1：

> **输入**：`"abc"`
> **输出**：3
> **解释**：三个回文子串：`"a"`, `"b"`, `"c"`

示例 2：

> **输入**：`"aaa"`
> **输出**：6
> **解释**：6个回文子串：`"a"`, `"a"`, `"a"`, `"aa"`, `"aa"`, `"aaa"`

提示：

- 输入的字符串长度不会超过 1000 。

动态规划解决

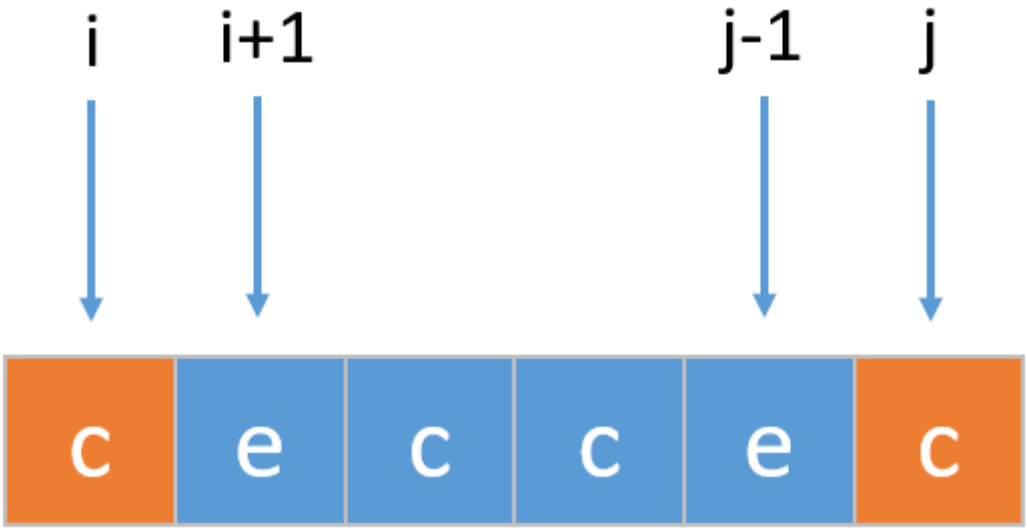
这题让求一个字符串中有多少个回文子串，子串必须是连续的，子序列可以不连续，这题可以使用动态规划来解决。

定义dp[i][j]：表示字符串s从下标i到j是否是回文串，如果dp[i][j]是true，则表示是回文串，否则不是回文串。

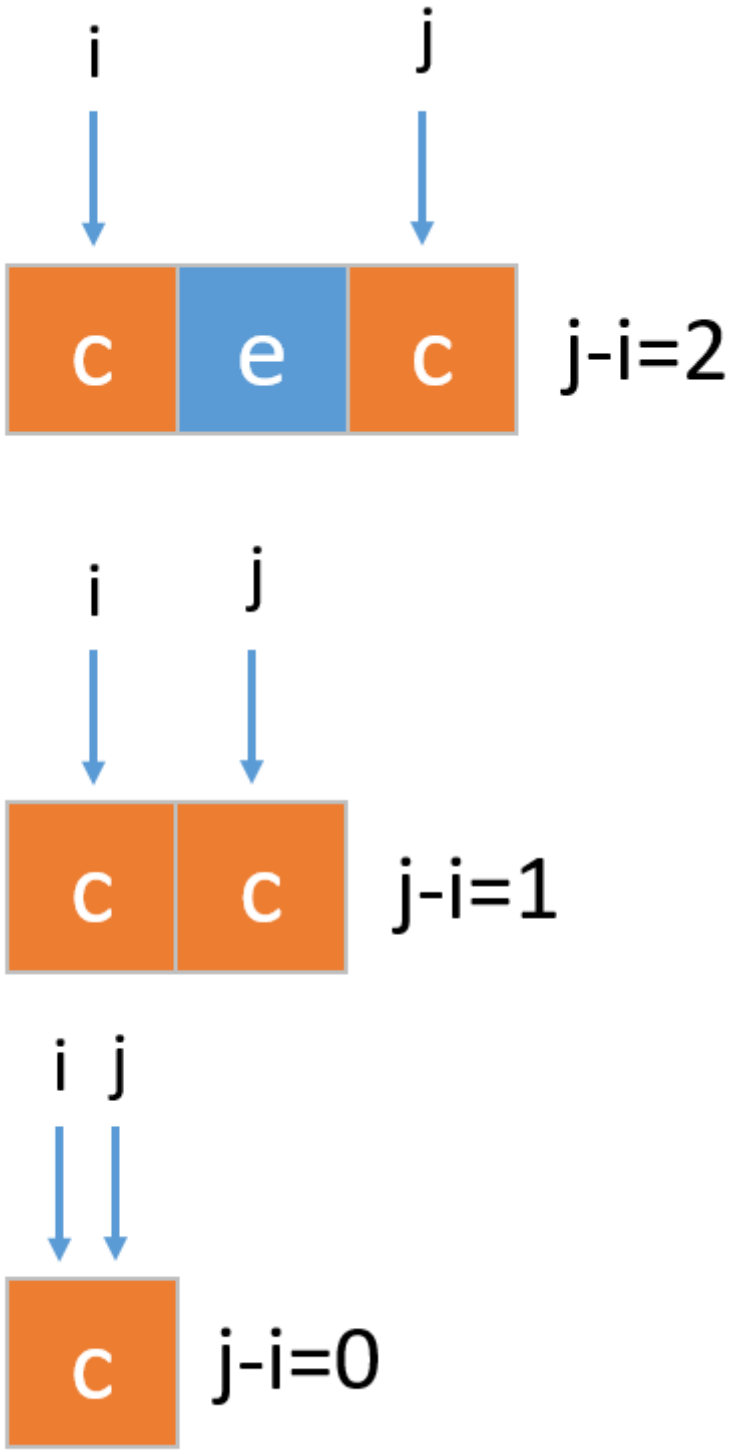
如果要计算dp[i][j]，首先要判断s.charAt(i)==s.charAt(j)是否成立。
1，如果s.charAt(i)!=s.charAt(j)，那么dp[i][j]肯定不能构成回文串。如下图所示



2，如果s.charAt(i)==s.charAt(j)，我们还需要判断dp[i+1][j-1]是否是回文串，如下图所示。



实际上如果i和j离的非常近的时候，比如j-i<=2，我们也可以认为dp[i][j]是回文子串，如下图所示



搞懂了上面的分析过程，递推公式就呼之欲出了。

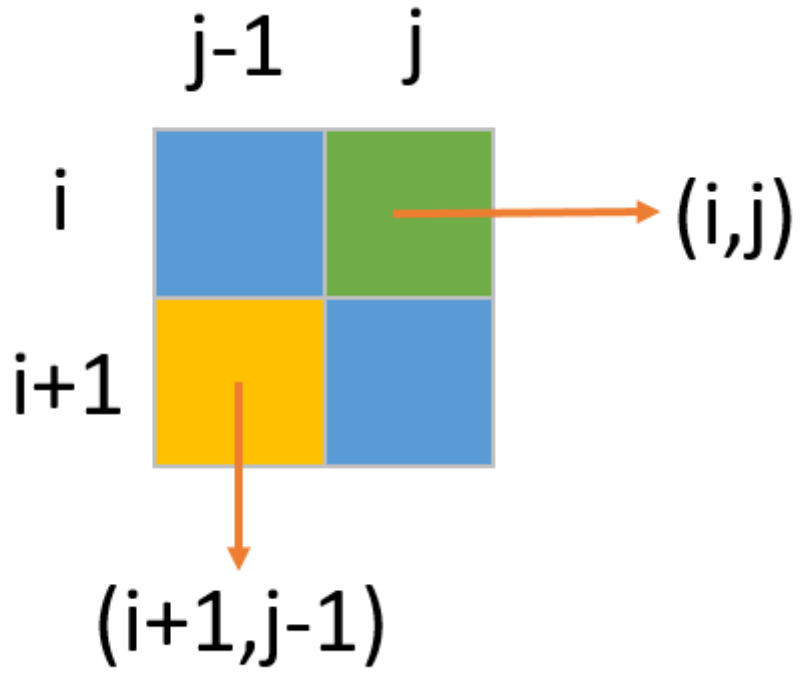
```
if (s.charAt(i) != s.charAt(j))
    continue;
dp[i][j] = j - i <= 2 || dp[i + 1][j - 1];
```

代码我们大致也能写出来了，因为是从i到j，所以j不能小于i。

```
for (int i = 0; i < length; i--) {
  for (int j = i; j < length; j++) {
    if (s.charAt(i) != s.charAt(j))
      continue;
    dp[i][j] = j - i <= 2 || dp[i + 1][j - 1];
```

```
6         }
7       }
```

但是这里有个问题，如果我们想要求dp[i][j]还需要知道dp[i+1][j-1]，如下图所示



对于这个二维数组，如果从上往下遍历当计算dp[i][j]的时候，我们还没有计算dp[i+1][j-1]的值，所以这个时候是没法计算dp[i][j]的，但我们可以从下往上计算，如下所示

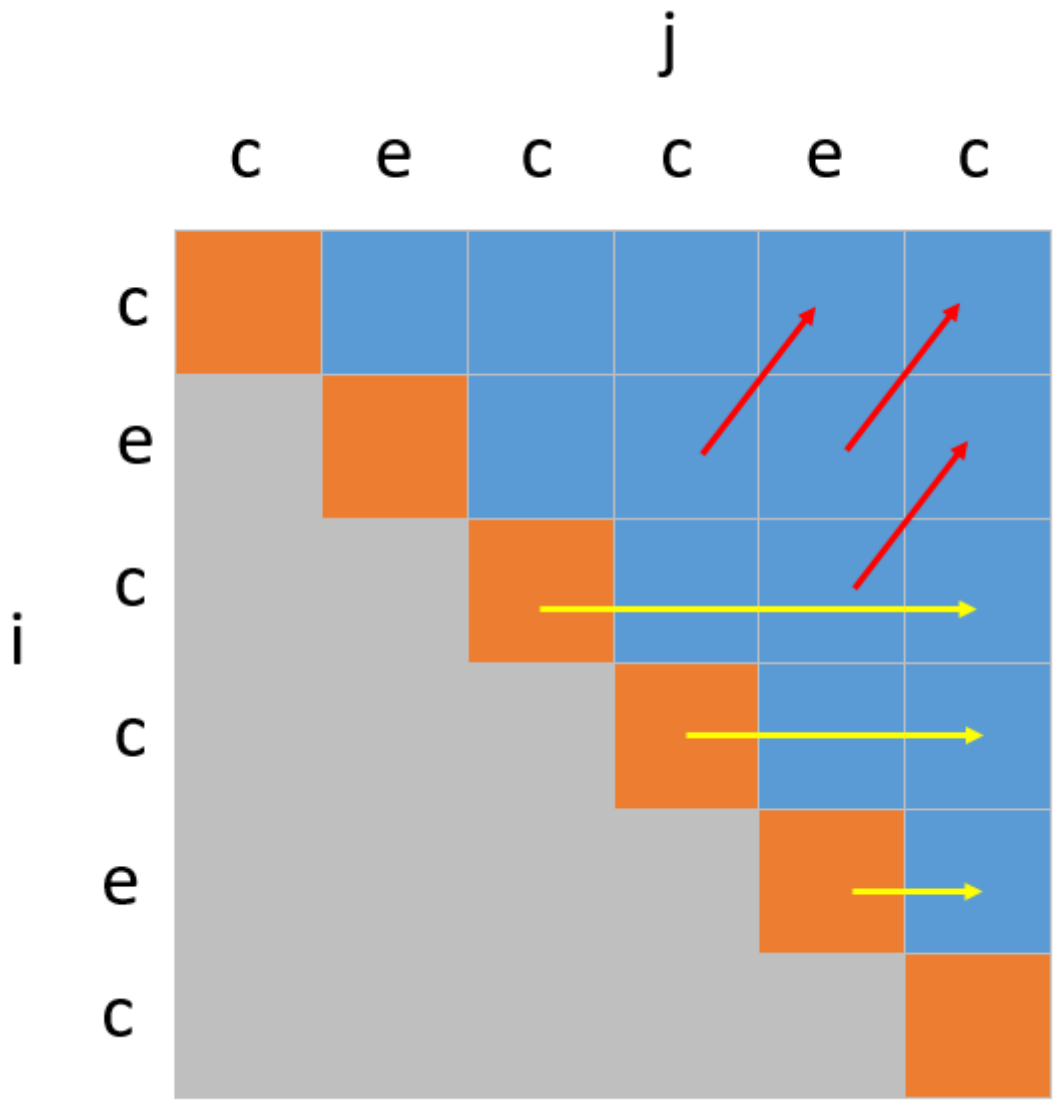
```
1       //从下往上计算
2       for (int i = length - 1; i >= 0; i--) {
3         for (int j = i; j < length; j++) {
4           if (s.charAt(i) != s.charAt(j))
5             continue;
6           dp[i][j] = j - i <= 2 || dp[i + 1][j - 1];
7         }
8       }
```

我们来看下最终代码

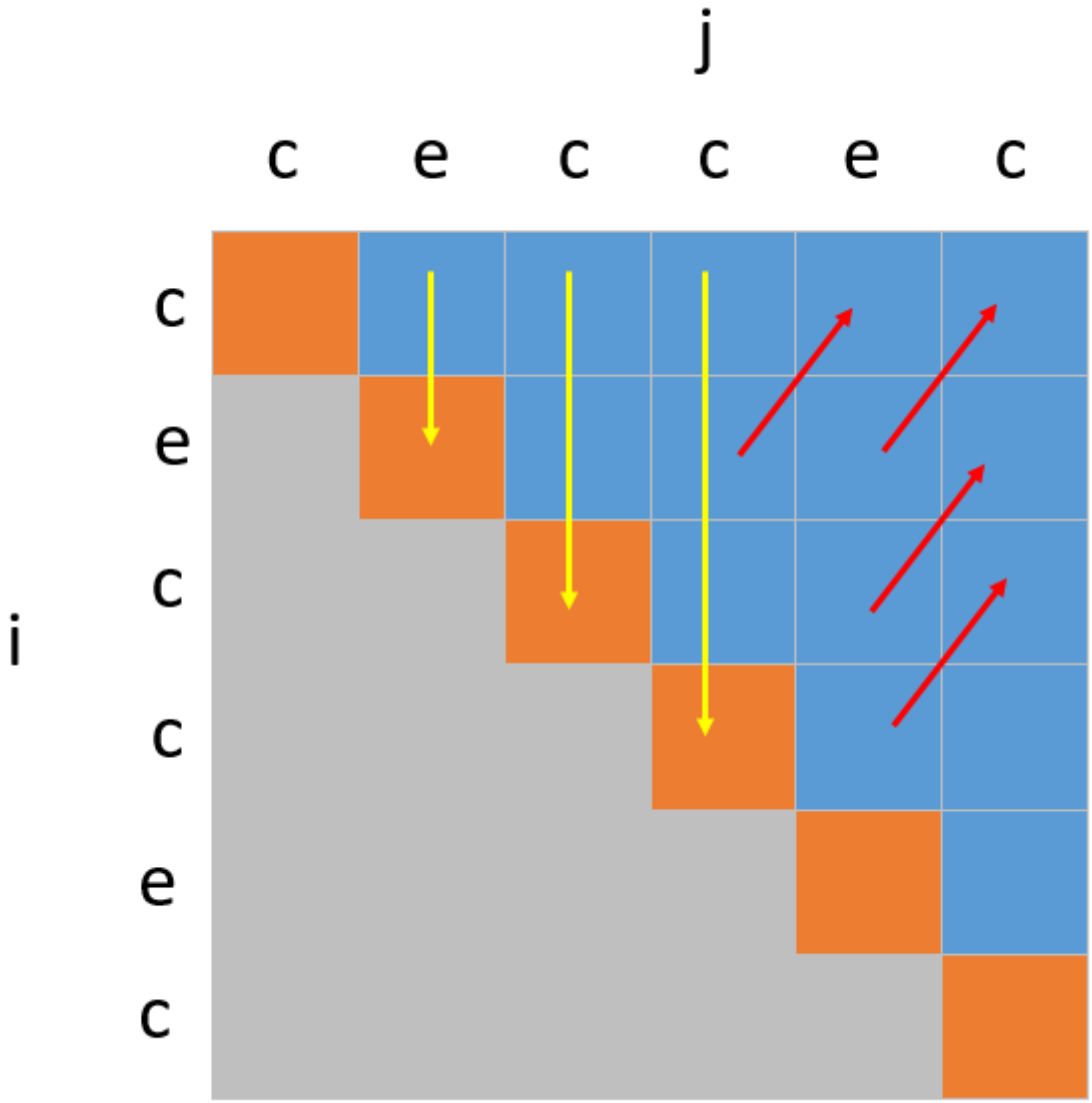
```
1   public int countSubstrings(String s) {
2     int length = s.length();
3     boolean[][] dp = new boolean[length][length];
4     int count = 0;//回文串的数量
5     //字符串从后往前判断
6     for (int i = length - 1; i >= 0; i--) {
7       for (int j = i; j < length; j++) {
8         //如果i和j指向的字符不一样，那么dp[i][j]就
9         //不能构成回文字符串
10        if (s.charAt(i) != s.charAt(j))
11          continue;
12        dp[i][j] = j - i <= 2 || dp[i + 1][j - 1];
13        //如果从i到j能构成回文串，count就加1
14        if (dp[i][j])
15          count++;
16      }
17    }
18    return count;
19  }
```

我们来看一下他的计算过程，如下图所示，红色箭头表示的是右上角的值依赖左下角的值，这里只画了部分，没画完。黄色表示的是计算的顺序，他是从右下角开始，从下往

上，从左往右开始计算的，所以当计算dp[i][j]的时候，dp[i+1][j-1]已经计算过了（图中灰色部分是i>j，属于无效的）



除了上面这种方式以外，我们还可以从左往右，从上往下开始计算，这样也能保证在计算dp[i][j]的时候，dp[i+1][j-1]已经计算过了，如下图所示



来看下代码

```
1   public int countSubstrings(String s) {
2     int length = s.length();
3     boolean[][] dp = new boolean[length][length];
4     int count = 0;//回文串的数量
5     for (int j = 0; j < length; j++) {
6       for (int i = 0; i <= j; i++) {
7         //如果i和j指向的字符不一样，那么dp[i][j]就
8         //不能构成回文字符串
9         if (s.charAt(i) != s.charAt(j))
10          continue;
11        dp[i][j] = j - i <= 2 || dp[i + 1][j - 1];
12        //如果从i到j能构成回文串，count就加1
13        if (dp[i][j])
14          count++;
15      }
16    }
17    return count;
18  }
```

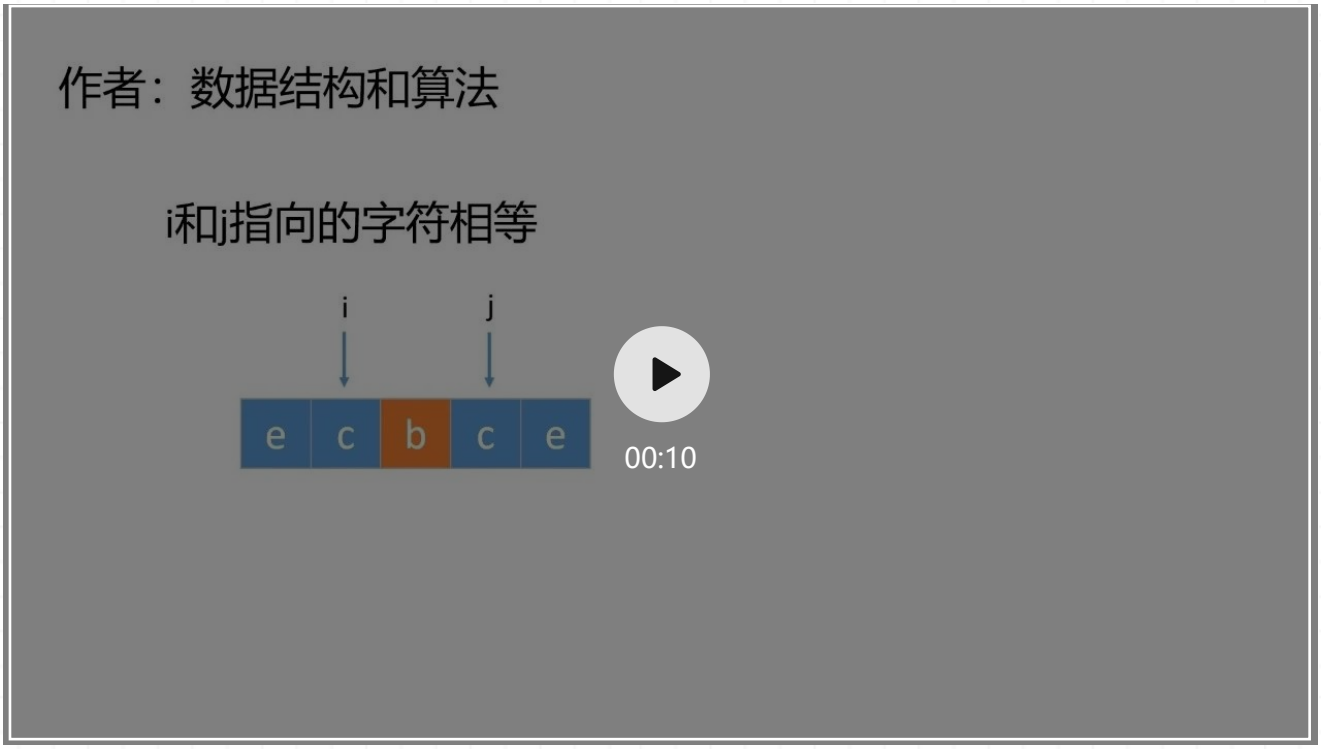
上面代码中dp是二维数组，但实际上在计算当前列的时候（如上图所示），我们只会用到上一列的结果，再往之前的就用不到了，所以我们还可以在优化一下，把二维数组改为一维。

```
1   public int countSubstrings(String s) {
2     int length = s.length();
3     boolean[] dp = new boolean[length];
```

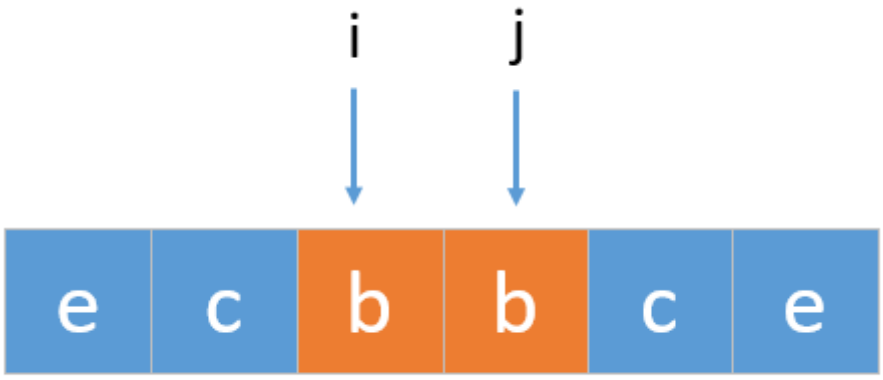
```
    int count = 0;//回文串的数量
    for (int j = 0; j < length; j++) {
      for (int i = 0; i <= j; i++) {
        if (s.charAt(i) == s.charAt(j) && (j - i <= 2 || dp[i + 1])) {
          dp[i] = true;
          count++;
        } else {
          dp[i] = false;
        }
      }
    }
    return count;
  }
```

中心扩散法解决

中心扩散的思想，是找到一个字符作为回文字符串的中心，往两边扩散，来看个视频



实际上回文串的字符不一定都是奇数个，还有可能是下面这样，所以我们计算的时候不光要考虑奇数的情况，还要考虑偶数的情况



来看下代码。

```
  //回文串的数量
```

```
int count = 0;

public int countSubstrings(String s) {
  //边界条件判断
  if (s == null || s.length() == 0)
    return 0;

  for (int i = 0; i < s.length(); i++) {
    //回文的长度是奇数
    extendPalindrome(s, i, i);
    //回文是长度是偶数
    extendPalindrome(s, i, i + 1);
  }
  return count;
}

//以left和right为中心点，求回文字符串的数量
private void extendPalindrome(String s, int left, int right) {
  while (left >= 0 && right < s.length() && s.charAt(left--) == s.charAt(right++)) {
    count++;
  }
}
```

还可以把上面的代码合并一下，如果回文串是奇数，我们把回文串中心的那个字符叫做中心点，如果回文串是偶数我们就把中间的那两个字符叫做中心点。

对于一个长度为n的字符串，我们可以用它的任意一个字符当做中心点，所以中心点的个数是n。我们还可以用它任意挨着的两个字符当做中心点，所以中心点是n-1，总的中心点就是2*n-1。

然后以这2*n-1个中心点为起点，往左右两边查找回文串，来看下代码。

```
//中心点的个数
public int countSubstrings(String s) {
  //字符串的长度
  int length = s.length();
  //中心点的个数
  int size = 2 * length - 1;
  //回文串的个数
  int count = 0;
  for (int i = 0; i < size; ++i) {
    //要么left等于right，要么left+1等于right。也就是说如果left等于
    //right，那么中心点就是一个字符，如果left+1等于right，中心点就是
    //2个字符
    int left = i / 2;
    int right = left + i % 2;

    while (left >= 0 && right < length && s.charAt(left--) == s.charAt(right++))
      ++count;
  }
  return count;
}
```

总结

回文字符串前面也讲过很多了，中心扩散法应该是最容易理解的，也是最常见的解回文字符串的一种方式。而对于动态规划要注意dp之间计算的先后顺序。

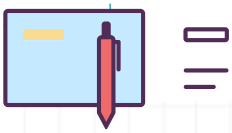
530，动态规划解最大正方形

原创 博哥 数据结构和算法 今天

收录于话题
#算法图文分析

140个 >

> The goal is not always meant to be reached, but to serve as a mark for our aim.
>
> 目标不一定永远都会达到，但可以当我们瞄准的方向。



问题描述

在一个由'0'和'1'组成的二维矩阵内，找到只包含'1'的最大正方形，并返回其面积。

示例 1：

1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

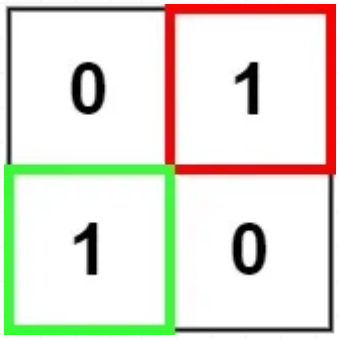
输入：

```
matrix =
[["1","0","1","0","0"],
 ["1","0","1","1","1"],
 ["1","1","1","1","1"],
```

```
["1","0","0","1","0"]]
```

输出：4

示例 2：




```
输入：matrix = [["0","1"],["1","0"]]
输出：1
```

示例 3：

```
输入：matrix = [["0"]]
输出：0
```

提示：

- m == matrix.length
- n == matrix[i].length
- 1 <= m, n <= 300
- matrix[i][j] 为 '0' 或 '1'

动态规划解决

这题让求的是由1围成的最大正方形，最容易想到的一种方式就是暴力求解。解决方式就是如果某个位置是1，就以他为正方形左上角，然后沿着右边和下边找最大的正方形，并且还要保证围成的正方形中所有的数字都是1。

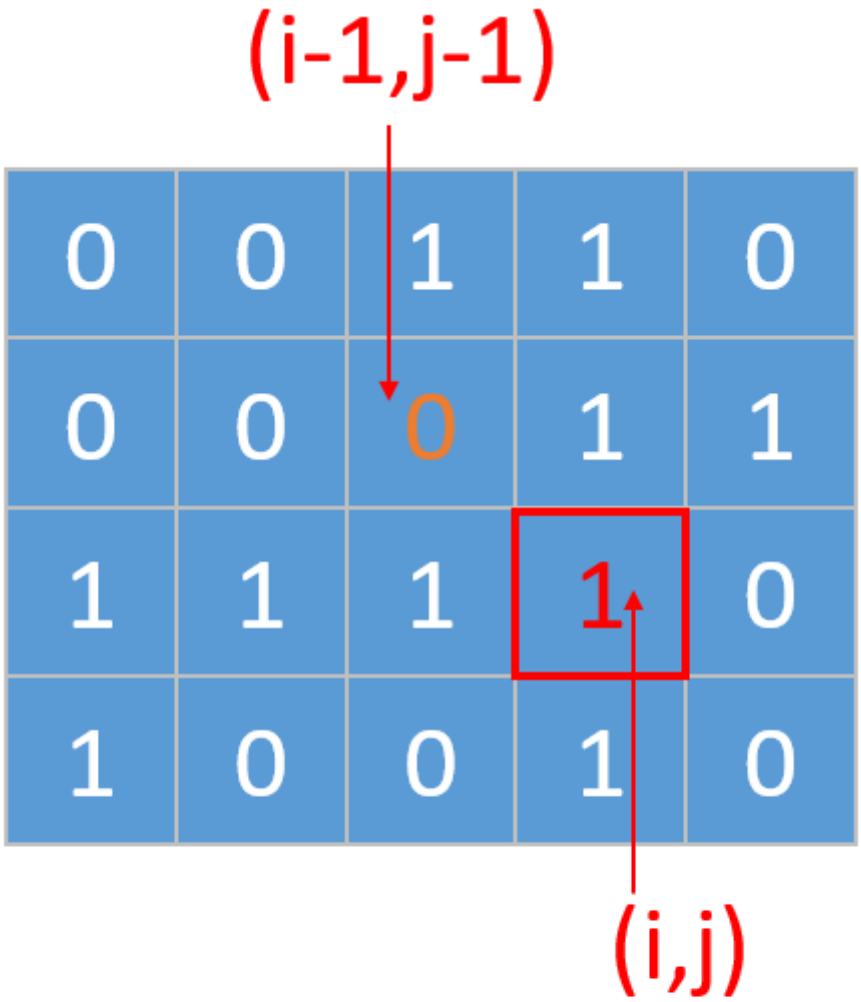
这种虽然容易想到但代码不太好写，并且时间复杂度也比较高。下面我们来看另一种实现方式，使用动态规划来解决。

定义二维数组dp[m][n]，其中dp[i][j]表示的是在矩阵中以坐标(i,j)为右下角的最大正方形边长。如果我们想求dp[i][j]，需要判断矩阵中matrix[i][j]的值。

如果matrix[i][j]是0就没法构成正方形，所以dp[i][j]=0。

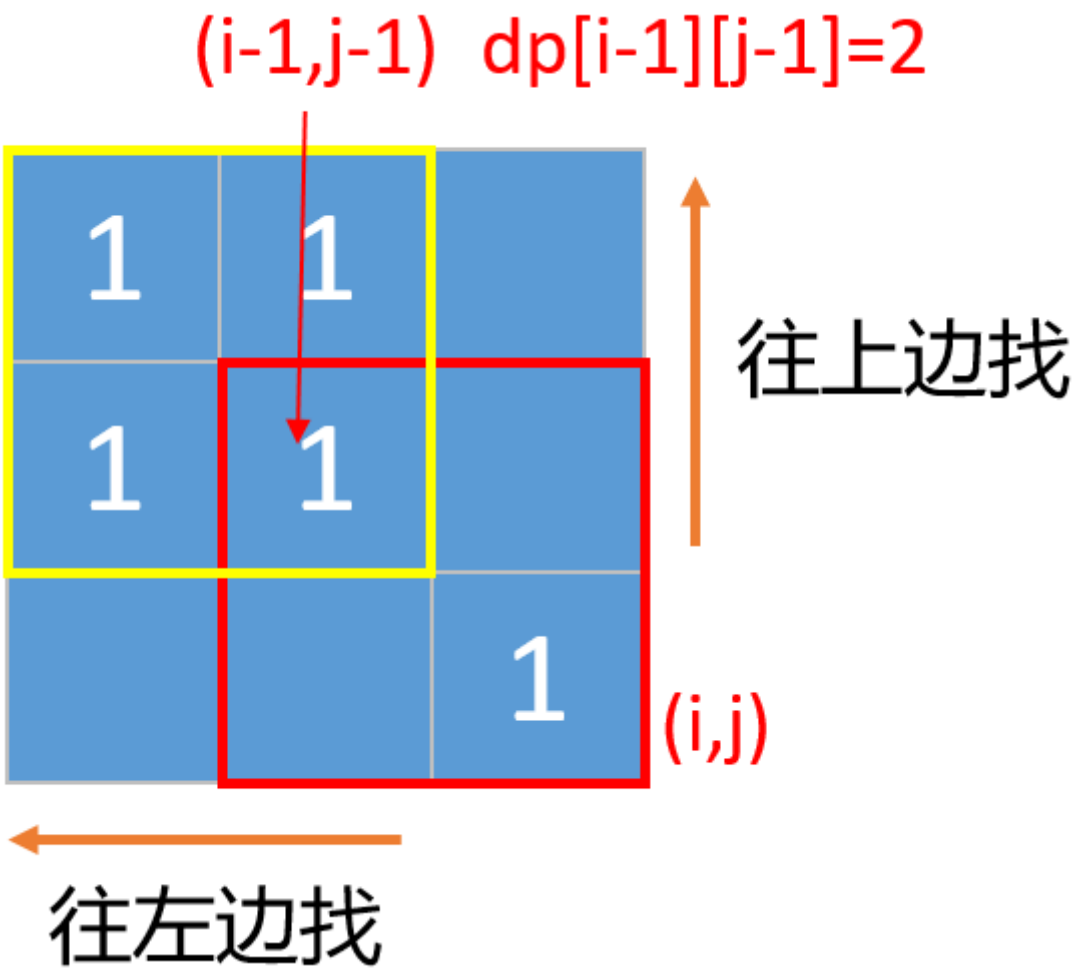
如果matrix[i][j]是1，说明他可以构成一个正方形，并且这个正方形的边长最小是1。

- 如果我们想求最大值，还需要判断他**左上角**的值dp[i-1][j-1]，如果dp[i-1][j-1]是0，那么以坐标(i,j)为右下角的最大正方形边长就是1，如下图所示

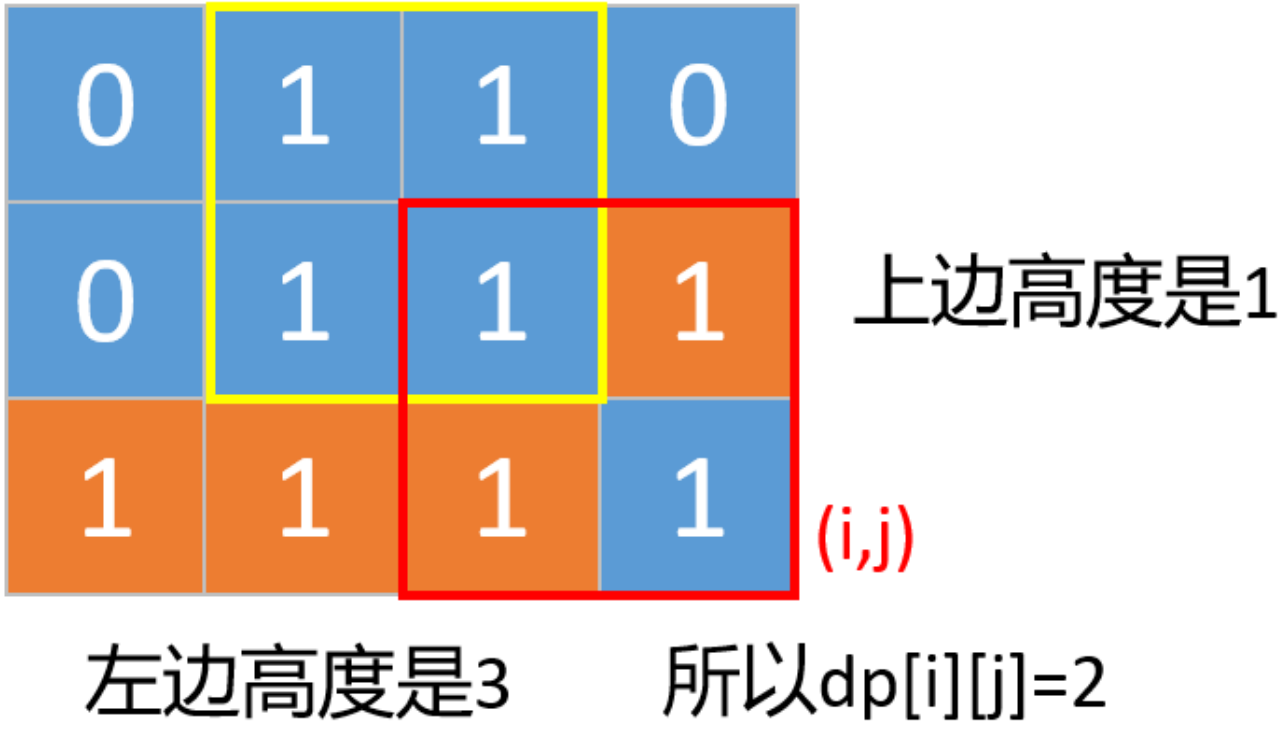


上图中，因为dp[i-1][j-1]是0，所以以(i,j)为右下角的最大正方形边长是1，即dp[i][j]=1

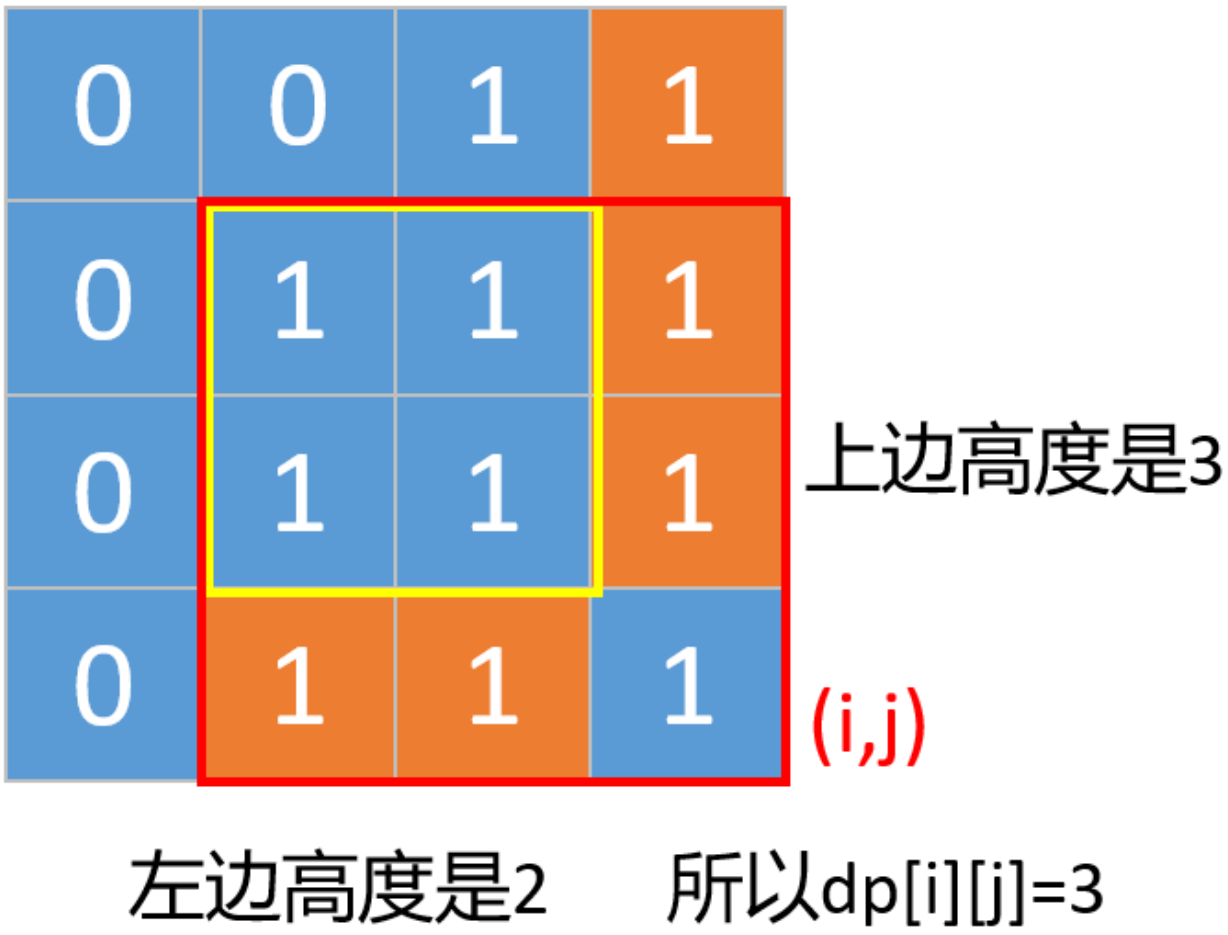
如果左上角的值dp[i-1][j-1]不是0，也就是说他也可以构成正方形，那么以坐标(i,j)为右下角有可能可以构成一个更大的正方形。为啥说是有可能，因为如果我们要确定他能不能构成一个更大的正方形，还要往他的上边和左边找，看下下面的图。



有可能是下面这种情况，就是左边或者上边的某一个高度小于dp[i-1][j-1]的值，要想构成最大的正方形我们只能取最小的。



也有可能是下面这种，就是左边和上边的高度都不小于dp[i-1][j-1]的值。



所以我们可以得出结论，如果(i,j)是1，那么以他为右下角的最大正方形边长是**{dp[i-1][j-1]，上边的高度，左边的高度}这3个中最小的+1。**

这里有个问题就是，如果(i,j)是1，我们有必要往他的上边和左边查找吗，很明显没必要，上边可以用dp[i-1][j]来代表，左边可以用dp[i][j-1]来代表。（注意这里dp[i-1][j]并不是上边为1的高度，dp[i-1][j]只会小，不会大，可以想一下为什么可以代表）

所以我们可以找出递推公式

如果坐标(i,j)为0，

则dp[i][j]=0;

如果坐标(i,j)为1，

则dp[i][j]=min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1;

为了防止一些不必要的边界条件判断，我把dp数组的长和宽都增加了1，来看下代码

```
1   public int maximalSquare(char[][] matrix) {
2     //二维矩阵的宽和高
3     int height = matrix.length;
4     int width = matrix[0].length;
5     int[][] dp = new int[height + 1][width + 1];
6     int maxSide = 0;//最大正方形的宽
7     for (int i = 1; i <= height; i++) {
```

```
        for (int j = 1; j <= width; j++) {
          if (matrix[i - 1][j - 1] == '1') {
            //递推公式
            dp[i][j] = Math.min(dp[i - 1][j], Math.min(dp[i - 1][j - 1], dp[i][j - 1])) + 1;
            //记录最大的边长
            maxSide = Math.max(maxSide, dp[i][j]);
          }
        }
      }
      //返回正方形的面积
      return maxSide * maxSide;
    }
```

总结

这题代码没什么难度，关键点在于怎么找出动态规划的递推公式，上面画了那么多图就是为了找出这个公式，有了公式代码就很容易写出来了。

往期推荐

- 529，动态规划解最长回文子序列
- 490，动态规划和双指针解买卖股票的最佳时机
- 430，剑指 Offer-动态规划求正则表达式匹配
- 395，动态规划解通配符匹配问题

529，动态规划解最长回文子序列

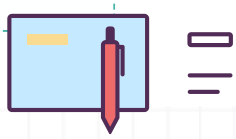
原创 博哥 数据结构和算法 前天

收录于话题
#算法图文分析

140个 >

You may be out, but you never lose the attitude.

人可以受挫，但精神不可以倒下。



问题描述

给定一个字符串s，找到其中最长的回文子序列，并返回该序列的长度。可以假设s的最大长度为 1000 。

示例 1:

输入:

```
"bbbab"
```

输出:

```
4
```

一个可能的最长回文子序列为 "bbbb"。

示例 2:

输入:

```
"cbbd"
```

输出:

```
2
```

一个可能的最长回文子序列为 "bb"。

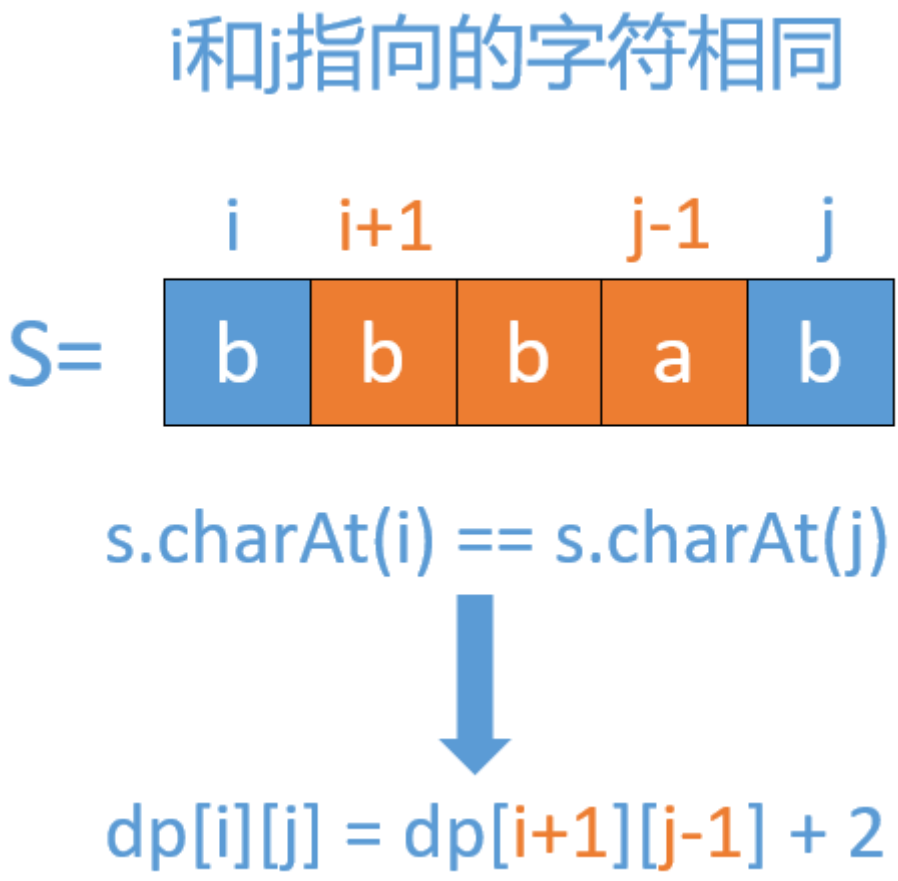
提示：

- 1 <= s.length <= 1000
- s 只包含小写英文字母

动态规划解决

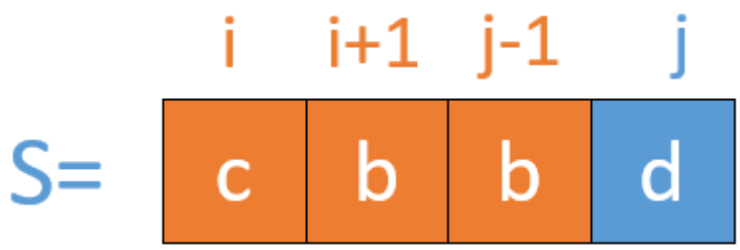
注意这里是求最长回文子序列不是回文子串，子串必须是连续的，但子序列不一定是连续的。

我们定义dp[i][j]表示字符串中从i到j之间的最长回文子序列。
1，如果s.charAt(i) == s.charAt(j)，也就是说两头的字符是一样的，他们可以和中间的最长回文子序列构成一个更长的回文子序列，即
dp[i][j]=dp[i+1][j-1]+2
如下图所示

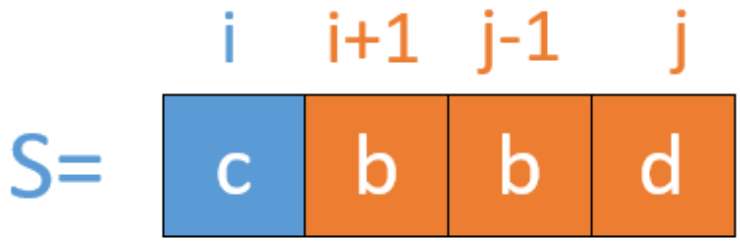


2，如果s.charAt(i) != s.charAt(j)，说明i和j指向的字符是不相等的，我们可以截取，要么去掉i指向的字符，要么去掉j指向的字符，然后取最大值，即
dp[i][j] = Math.max(dp[i][j-1], dp[i+1][j])
如下图所示

i和j指向的字符不相同

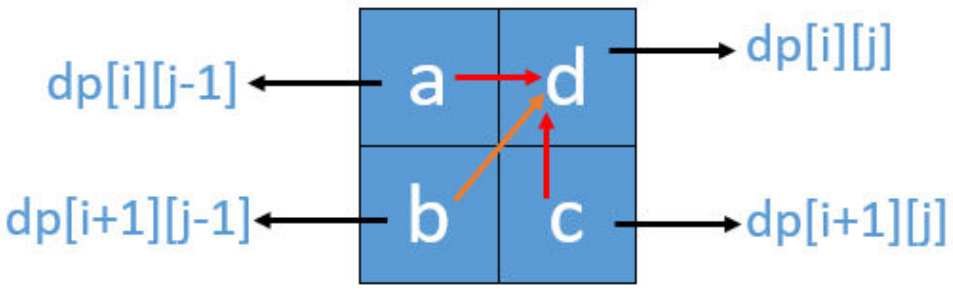


s.charAt(i) != s.charAt(j)



dp[i][j] = Math.max(dp[i][j-1], dp[i+1][j])

有了递推公式，我们再来看下Base case，就是一个字符也是回文串，即dp[i][i]=1；
注意二维数组dp[i][j]的定义，如果i>j是没有意义的。再来看一下他们之间的关系



dp[i][j]=dp[i+1][j-1]+2

dp[i][j] = Math.max(dp[i+1][j], dp[i][j-1])

从上面图中可以看出如果我们想求dp[i][j]，那么其他3个必须都是已知的，很明显从上往下遍历是不行的，我们只能让i从最后一个字符往前遍历，j从i的下一个开始遍历，最后只需要返回dp[0][length - 1]即可。来看下代码

```
public int longestPalindromeSubseq(String s) {
    int length = s.length();
    int[][] dp = new int[length][length];
    //这里i要从最后一个开始遍历
    for (int i = length - 1; i >= 0; i--) {
        //单个字符也是一个回文串
        dp[i][i] = 1;
        //j从i的下一个开始
        for (int j = i + 1; j < length; j++) {
            //下面是递推公式
            if (s.charAt(i) == s.charAt(j)) {
                dp[i][j] = dp[i + 1][j - 1] + 2;
            } else {
                dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[0][length - 1];
}
```

总结

这里递推公式比较好找，关键点在于字符串遍历的顺序。

往期推荐

- 515，动态规划解买卖股票的最佳时机含手续费
- 493，动态规划解打家劫舍 III
- 486，动态规划解最大子序和
- 413，动态规划求最长上升子序列

522，俄罗斯套娃信封问题

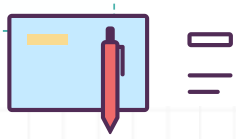
原创 博哥 数据结构和算法 1周前

收录于话题
#算法图文分析

137个 >

The time you enjoy wasting is not wasted time.

你乐在其中所度过的时间，就不算浪费掉的时间。



问题描述

给你一个二维整数数组envelopes，其中envelopes[i]=[wi,hi]，表示第i个信封的宽度和高度。

当另一个信封的宽度和高度都比这个信封大的时候，这个信封就可以放进另一个信封里，如同俄罗斯套娃一样。

请计算最多能有多少个信封能组成一组“俄罗斯套娃”信封（即可以把一个信封放到另一个信封里面）。

注意：不允许旋转信封。

示例 1：

> **输入**：envelopes = [[5,4],[6,4],[6,7],[2,3]]
> **输出**：3
> **解释**：最多信封的个数为 3，组合为：[2,3] => [5,4] => [6,7]。

示例 2：

> **输入**：envelopes = [[1,1],[1,1],[1,1]]
> **输出**：1

提示：

- 1<=envelopes.length<=5000
- envelopes[i].length==2
- 1<=wi,hi<=10^4

动态规划解决

题中说了信封不能旋转，但没说位置不能变。我们可先对信封进行排序，只要后面信封的宽和高都比前面的大，那么后面的信封就能套住前面的信封。

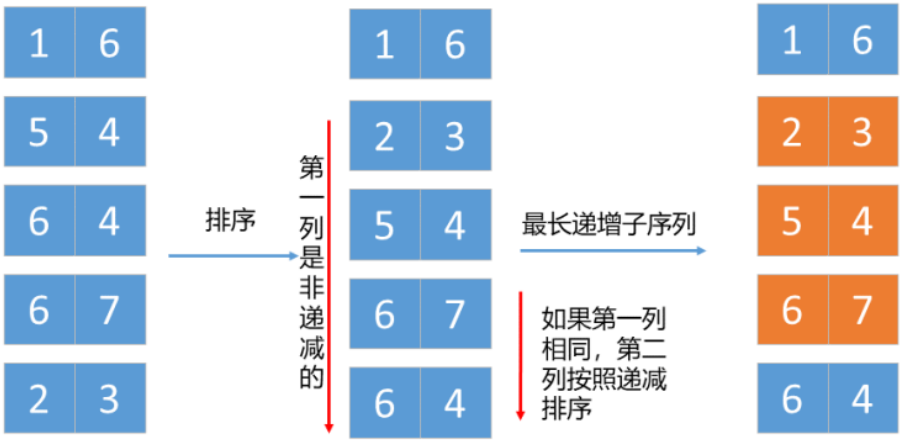
注意：子序列不是子数组，子序列不要求是连续的。

那么关键问题是怎么对信封进行排序，假如按照面积排序肯定是不行的，比如两个信封分别是[1，9]和[3，4]，后面的信封面积比前面的大，但很明显是不能把前面信封套住的。

既然不能按面积排序，我们可以先按照宽度进行**升序**排序，因为宽度是升序的，当我们从前往后选择的时候就可以忽略宽度，然后只需要比较高度即可，这个时候就可以转化为求最长上升子序列的问题了，也就是413，动态规划求最长上升子序列，只不过第413题是一维数组，这里是二维数组，但我们比较的时候只比较高度，其实原理都一样。

如果宽度相同，我们在按照高度**降序**排序。为什么要降序，因为如果宽度相同的时候，在这些宽度相同的信封中我们最多只能选择一个，如果是升序的话有可能会选择多个，只有降序才能满足这个条件。

作者：数据结构和算法



最长上升子序列的计算这里就不在赘述，具体可以看下[413，动态规划求最长上升子序列]，我们来看下这题的最终代码。

```
public int maxEnvelopes(int[][] envelopes) {
    //边界条件判断
    if (envelopes == null || envelopes.length == 0)
        return 0;
    //先对信封进行排序
    Arrays.sort(envelopes, (int[] arr1, int[] arr2) -> {
        if (arr1[0] == arr2[0])
            return arr2[1] - arr1[1];
        else
            return arr1[0] - arr2[0];
    });
    return lengthOfLIS(envelopes);
}

//求最长上升子序列
public int lengthOfLIS(int[][] nums) {
    int[] dp = new int[nums.length];
    //初始化数组dp的每个值为1
    Arrays.fill(dp, 1);
    int max = 1;
    for (int i = 1; i < nums.length; i++) {
        for (int j = 0; j < i; j++) {
            //如果当前值nums[i]大于nums[j]，说明nums[i]可以和
            //nums[j]结尾的上升序列构成一个新的上升子序列
            if (nums[i][1] > nums[j][1]) {
                dp[i] = Math.max(dp[i], dp[j] + 1);
            }
        }
        //记录构成的最大值
        max = Math.max(max, dp[i]);
    }
    return max;
}
```

二分法查找

二分法的查找原理很简单，我们使用一个临时变量list，并且list中的元素是递增的。我们每次遍历数组的时候如果list不为空，都会拿当前值nums[i]和list的最后一个元素比较

1，如果nums[i]比list最后一个元素大，说明nums[i]加入到list的末尾可以构成一个更长的上升子序列，我们就把nums[i]加入到list的末尾。

2，如果nums[i]不大于list的最后一个元素，说明nums[i]和list不能构成一个更长的上升子序列，但我们可以用nums[i]把list中第一个大于他的给替换掉。我们要保证list中元素不变的情况下，值越小越好，这样当我们加入一个新值的时候，构成上升子序列的可能性就越大。

代码中有详细注释，具体可以看下

```
public int maxEnvelopes(int[][] envelopes) {
  //边界条件判断
  if (envelopes == null || envelopes.length == 0)
    return 0;
  //先对信封进行排序
  Arrays.sort(envelopes, (int[] arr1, int[] arr2) -> {
    if (arr1[0] == arr2[0])
      return arr2[1] - arr1[1];
    else
      return arr1[0] - arr2[0];
  });
  return lengthOfLIS(envelopes);
}

//最长上升子序列
public int lengthOfLIS(int[][] nums) {
  //list中保存的是构成的上升子序列
  ArrayList<Integer> list = new ArrayList<>(nums.length);
  for (int[] num : nums) {
    //如果list为空，我们直接把num加进去。如果list的最后一个元素小于num，
    //说明num加入到list的末尾可以构成一个更长的上升子序列，我们就把num
    //加入到list的末尾
    if (list.size() == 0 || list.get(list.size() - 1) < num[1])
      list.add(num[1]);
    else {
      //如果num不小于list的最后一个元素，我们就用num把list中第一
      //个大于他的值给替换掉,这样我们才能保证list中的元素在长度不变
      //的情况下，元素值尽可能的小
      int i = Collections.binarySearch(list, num[1]);
      //因为list是从小到大排序的，并且上面使用的是二分法查找。当i大
      //于0的时候，说明出现了重复的，我们直接把他替换即可，如果i小于
      //0，我们对i取反，他就是list中第一个大于num值的位置，我们把它
      //替换即可
      list.set((i < 0) ? -i - 1 : i, num[1]);
    }
  }
  return list.size();
}
```

对于二分法查找的代码可以看下202，查找-二分法查找，也可以看下官方提供的代码

```
/**
 * Searches the specified list for the specified object using the binary
 * search algorithm.  The list must be sorted into ascending order
 * according to the {@linkplain Comparable natural ordering} of its
 * elements (as by the {@link #sort(List)} method) prior to making this
```

```
 * call.  If it is not sorted, the results are undefined.  If the list
 * contains multiple elements equal to the specified object, there is no
 * guarantee which one will be found.
 *
 * <p>This method runs in log(n) time for a "random access" list (which
 * provides near-constant-time positional access).  If the specified list
 * does not implement the {@link RandomAccess} interface and is large,
 * this method will do an iterator-based binary search that performs
 * O(n) link traversals and O(log n) element comparisons.
 *
 * @param  <T> the class of the objects in the list
 * @param  list the list to be searched.
 * @param  key the key to be searched for.
 * @return the index of the search key, if it is contained in the list;
 *         otherwise, <tt>(-(<i>insertion point</i>) - 1)</tt>.  The
 *         <i>insertion point</i> is defined as the point at which the
 *         key would be inserted into the list: the index of the first
 *         element greater than the key, or <tt>list.size()</tt> if all
 *         elements in the list are less than the specified key.  Note
 *         that this guarantees that the return value will be &gt;= 0 if
 *         and only if the key is found.
 * @throws ClassCastException if the list contains elements that are not
 *         <i>mutually comparable</i> (for example, strings and
 *         integers), or the search key is not mutually comparable
 *         with the elements of the list.
 */
public static <T>
int binarySearch(List<? extends Comparable<? super T>> list, T key) {
  if (list instanceof RandomAccess || list.size()<BINARYSEARCH_THRESHOLD)
    return Collections.indexedBinarySearch(list, key);
  else
    return Collections.iteratorBinarySearch(list, key);
}

private static <T>
int indexedBinarySearch(List<? extends Comparable<? super T>> list, T key) {
  int low = 0;
  int high = list.size()-1;

  while (low <= high) {
    int mid = (low + high) >>> 1;
    Comparable<? super T> midVal = list.get(mid);
    int cmp = midVal.compareTo(key);

    if (cmp < 0)
      low = mid + 1;
    else if (cmp > 0)
      high = mid - 1;
    else
      return mid; // key found
  }
  return -(low + 1);  // key not found
}
```

总结

如果直接计算，不太好算，如果我们把它转化为求最长上升子序列问题，就简单多了。

往期推荐

- 515，动态规划解买卖股票的最佳时机含手续费

517，最长回文子串的3种解决方式

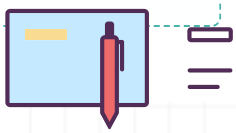
原创 博哥 数据结构和算法 2月22日

收录于话题
#算法图文分析

137个 >

The real opportunity for success lies within the person and not in the job.

成功的真正机会在于人而非工作。



问题描述

给你一个字符串s，找到s中最长的回文子串。

示例 1：

```
输入：s = "babad"
输出："bab"
解释："aba" 同样是符合题意的答案。
```

示例 2：

```
输入：s = "cbbd"
输出："bb"
```

示例 3：

```
输入：s = "a"
输出："a"
```

示例 4：

```
输入：s = "ac"
```

> 输出：“a”

提示：

- 1 <= s.length <= 1000
- s 仅由数字和英文字母（大写和/或小写）组成

暴力求解

暴力求解是最容易想到的，要截取字符串的所有子串，然后再判断这些子串中哪些是回文的，最后返回回文子串中最长的即可。

这里我们可以使用两个变量，一个记录最长回文子串开始的位置，一个记录最长回文子串的长度，最后再截取。代码如下

```
public String longestPalindrome(String s) {
  if (s.length() < 2)
    return s;
  int start = 0;
  int maxLen = 0;
  for (int i = 0; i < s.length() - 1; i++) {
    for (int j = i; j < s.length(); j++) {
      //截取所有子串，然后在逐个判断是否是回文的
      if (isPalindrome(s, i, j)) {
        if (maxLen < j - i + 1) {
          start = i;
          maxLen = j - i + 1;
        }
      }
    }
  }
  return s.substring(start, start + maxLen);
}

//判断是否是回文串
private boolean isPalindrome(String s, int start, int end) {
  while (start < end) {
    if (s.charAt(start++) != s.charAt(end--))
      return false;
  }
  return true;
}
```

暴力求解毕竟效率很差，上面代码其实还可以优化一下，在截取的时候，如果截取的长度小于等于目前查找到的最长回文子串，我们可以直接跳过，不需要再判断了，因为即使他是回文子串，也不可能是最长的。

```
public String longestPalindrome(String s) {
  if (s.length() < 2)
    return s;
  int start = 0;
  int maxLen = 0;
  for (int i = 0; i < s.length() - 1; i++) {
    for (int j = i; j < s.length(); j++) {
      //截取所有子串，如果截取的子串小于等于之前
      //遍历过的最大回文串，直接跳过。因为截取
```

```
            //的子串即使是回文串也不可能是最大的，所以
            //不需要判断
            if (j - i < maxLen)
              continue;
            if (isPalindrome(s, i, j)) {
              if (maxLen < j - i + 1) {
                start = i;
                maxLen = j - i + 1;
              }
            }
          }
        }
        return s.substring(start, start + maxLen);
    }

    //判断是否是回文串
    private boolean isPalindrome(String s, int start, int end) {
        while (start < end) {
            if (s.charAt(start++) != s.charAt(end--))
                return false;
        }
        return true;
    }
```

中心扩散法

中心扩散法也很好理解，我们遍历字符串的每一个字符，然后以当前字符为中心往两边扩散，查找最长的回文子串，下面随便举个例子，看一下视频演示（如果觉得视频播放过快，可以点击暂停）



视频中我们是以每一个字符为中心，往两边扩散，来求最长的回文子串。但忽略了一个问题，回文串的长度不一定都是奇数，也可能是偶数，比如字符串"abba"，如果使用上面的方式判断肯定是不对的。

我们来思考这样一个问题，如果是单个字符，我们可以认为他是回文子串，如果是多个字符，并且他们都是相同的，那么他们也是回文串。

所以对于上面的问题，我们以当前字符为中心往两边扩散的时候，先要判断和他挨着的有没有相同的字符，如果有，则直接跳过，来看下代码

```
String longestPalindrome(String s) {
  //边界条件判断
  if (s.length() < 2)
    return s;
  //start表示最长回文串开始的位置，
  //maxLen表示最长回文串的长度
  int start = 0, maxLen = 0;
  int length = s.length();
  for (int i = 0; i < length; ) {
    //如果剩余子串长度小于目前查找到的最长回文子串的长度，直接终止循环
    //（因为即使他是回文子串，也不是最长的，所以直接终止循环，不再判断）
    if (length - i <= maxLen / 2)
      break;
    int left = i, right = i;
    while (right < length - 1 && s.charAt(right + 1) == s.charAt(right))
      ++right; //过滤掉重复的
    //下次在判断的时候从重复的下一个字符开始判断
    i = right + 1;
    //然后往两边判断，找出回文子串的长度
    while (right < length - 1 && left > 0 && s.charAt(right + 1) == s.charAt(left - 1)) {
      ++right;
      --left;
    }
    //保留最长的
    if (right - left + 1 > maxLen) {
      start = left;
      maxLen = right - left + 1;
    }
  }
  //截取回文子串
  return s.substring(start, start + maxLen);
}
```

动态规划

定义二维数组dp[length][length]，如果dp[left][right]为true，则表示字符串从left到right是回文子串，如果dp[left][right]为false，则表示字符串从left到right不是回文子串。

如果dp[left+1][right-1]为true，我们判断s.charAt(left)和s.charAt(right)是否相等，如果相等，那么dp[left][right]肯定也是回文子串，否则dp[left][right]一定不是回文子串。

所以我们可以找出递推公式

```
dp[left][right]=s.charAt(left)==s.charAt(right)&&dp[left+1][right-1]
```

有了递推公式，还要确定边界条件：

如果s.charAt(left)！=s.charAt(right)，那么字符串从left到right是不可能构成子串的，直接跳过即可。

如果s.charAt(left)==s.charAt(right)，字符串从left到right能不能构成回文子串还需要进一步判断

- 如果left==right，也就是说只有一个字符，我们认为他是回文子串。即dp[left][right]=true（left==right）
- 如果right-left<=2，类似于"aa"，或者"aba"，我们认为他是回文子串。即dp[left][right]=true（right-left<=2）
- 如果right-left>2，我们只需要判断dp[left+1][right-1]是否是回文子串，才能确定dp[left][right]是否为true还是false。即dp[left][right]=dp[left+1][right-1]

有了递推公式和边界条件，代码就很容易写了，来看下

```java
public static String longestPalindrome(String s) {
  //边界条件判断
  if (s.length() < 2)
    return s;
  //start表示最长回文串开始的位置，
  //maxLen表示最长回文串的长度
  int start = 0, maxLen = 1;
  int length = s.length();
  boolean[][] dp = new boolean[length][length];
  for (int right = 1; right < length; right++) {
    for (int left = 0; left < right; left++) {
      //如果两种字符不相同，肯定不能构成回文子串
      if (s.charAt(left) != s.charAt(right))
        continue;

      //下面是s.charAt(left)和s.charAt(right)两个
      //字符相同情况下的判断
      //如果只有一个字符，肯定是回文子串
      if (right == left) {
        dp[left][right] = true;
      } else if (right - left <= 2) {
        //类似于"aa"和"aba"，也是回文子串
        dp[left][right] = true;
      } else {
        //类似于"a******a"，要判断他是否是回文子串，只需要
        //判断"******"是否是回文子串即可
        dp[left][right] = dp[left + 1][right - 1];
      }
      //如果字符串从left到right是回文子串，只需要保存最长的即可
      if (dp[left][right] && right - left + 1 > maxLen) {
        maxLen = right - left + 1;
        start = left;
      }
    }
  }
  //截取最长的回文子串
  return s.substring(start, start + maxLen);
}
```

总结

这题不是很难，上面几种应该都很容易想到。其实还有一种更高效的解决方式就是Manacher算法，很多人习惯把它称为马拉车算法，这种解法比上面几种难度稍微大一些，后续有时间会单独拿出来讲。

往期推荐

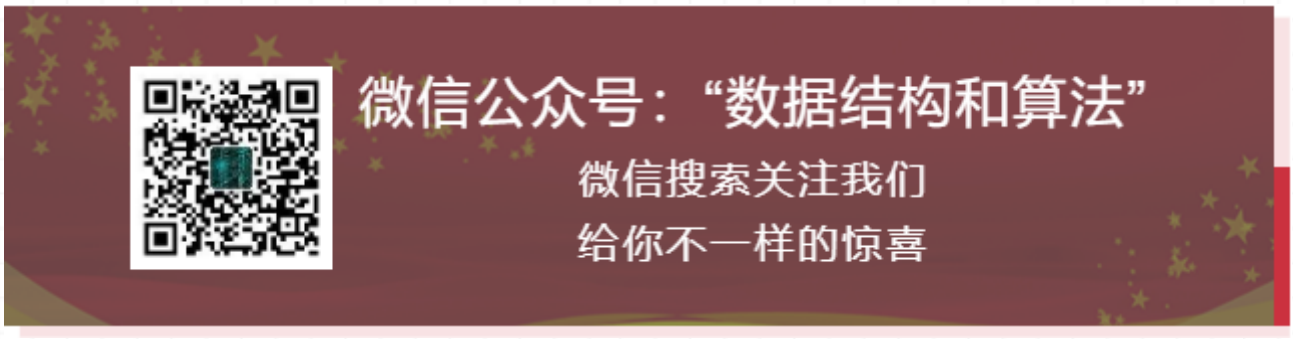
- 497，双指针验证回文串
- 463. 判断回文链表的3种方式
- 486，动态规划解最大子序和
- 413，动态规划求最长上升子序列

515，动态规划解买卖股票的最佳时机含手续费

原创 山大王wld 数据结构和算法 2月7日

收录于话题
#算法图文分析

137个 >



Since you are like no other being ever created since the beginning of time, you are incomparable.

因为你和有史以来任何人类都不相同，所以你是无可比拟的。

问题描述

给定一个整数数组prices，其中第i个元素代表了第i天的股票价格；非负整数fee代表了交易股票的手续费用。

你可以无限次地完成交易，但是你每笔交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

返回获得利润的最大值。

注意：这里的一笔交易指买入持有并卖出股票的整个过程，每笔交易你只需要为支付一次手续费。

示例 1:

> 输入: prices = [1, 3, 2, 8, 4, 9], fee = 2
> 输出: 8
> 解释: 能够达到的最大利润:
> 在此处买入 prices[0] = 1
> 在此处卖出 prices[3] = 8
> 在此处买入 prices[4] = 4
> 在此处卖出 prices[5] = 9
> 总利润: ((8 - 1) - 2) + ((9 - 4) - 2) = 8.

注意:

- 0 < prices.length <= 50000.
- 0 < prices[i] < 50000.
- 0 <= fee < 50000.

动态规划解决

这题和492，动态规划和贪心算法解买卖股票的最佳时机 II非常类似，不同的是第492题不需要手续费，而这个题需要手续费。参照第492题我们来看下这题使用动态规划该怎么解决。

定义

dp[i][0]表示第i天交易完之后手里**没有**股票的最大利润。
dp[i][1]表示第i天交易完之后手里**持有**股票的最大利润。

当天交易完之后手里**没有**股票可能有两种情况:

- 一种是当天没有进行任何交易，又因为当天手里没有股票，所以当天没有股票的利润只能取前一天手里没有股票的利润。
- 一种是把当天手里的股票给卖了，既然能卖，说明手里是有股票的，所以这个时候当天没有股票的利润要取前一天手里有股票的利润加上当天股票能卖的价格再减去手续费。

这两种情况我们取利润最大的即可，所以可以得到
dp[i][0]=max(dp[i-1][0],dp[i-1][1]+prices[i]-fee);

当天交易完之后手里**持有**股票也有两种情况:

- 一种是当天没有任何交易，又因为当天手里持有股票，所以当天手里持有的股票其实前一天就已经持有了。
- 还一种是当天买入了股票，当天能卖股票，说明前一天手里肯定是没有股票的。

我们取这两者的最大值，所以可以得到

dp[i][1]=max(dp[i-1][1],dp[i-1][0]-prices[i]);

动态规划的递推公式有了，那么初始条件是什么呢，就是第0天

如果买入：dp[0][1]=-prices[0];

如果没买：dp[0][0]=0;

有了递推公式和边界条件，代码很容易就写出来了。

```
public int maxProfit(int[] prices, int fee) {
    //边界条件判断
    if (prices == null || prices.length < 2)
        return 0;
    int length = prices.length;
    int[][] dp = new int[length][2];
    //初始条件
    dp[0][1] = -prices[0];
    dp[0][0] = 0;
    for (int i = 1; i < length; i++) {
        //递推公式
        dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i] - fee);
        dp[i][1] = Math.max(dp[i - 1][1], dp[i - 1][0] - prices[i]);
    }
    //最后一天肯定是手里没有股票的时候，利润才会最大，
    //只需要返回dp[length - 1][0]即可
    return dp[length - 1][0];
}
```

代码优化

上面计算的时候我们看到当天的利润只和前一天的记录有关，没必要使用一个二维数组，只需要使用两个变量：

一个记录当天交易完之后手里持有股票的最大利润。

一个记录当天交易完之后手里没有股票的最大利润。

来看下代码

```
public int maxProfit(int[] prices, int fee) {
    //边界条件判断
    if (prices == null || prices.length < 2)
        return 0;
    int length = prices.length;
    //初始条件
    int hold = -prices[0];//持有股票
    int noHold = 0;//没持有股票
    for (int i = 1; i < length; i++) {
        //递推公式转化的
        noHold = Math.max(noHold, hold + prices[i] - fee);
        hold = Math.max(hold, noHold - prices[i]);
    }
    //最后一天肯定是手里没有股票的时候利润才会最大，
    //所以这里返回的是noHold
    return noHold;
}
```

总结

所有的动态规划，只要找到递推公式和边界条件以及初始值，然后在套用公式就很容易解决。

往期推荐

- 493，动态规划解打家劫舍 III
- 492，动态规划和贪心算法解买卖股票的最佳时机 II
- 490，动态规划和双指针解买卖股票的最佳时机
- 486，动态规划解最大子序和

493，动态规划解打家劫舍 III

原创 山大王wld 数据结构和算法 2020-12-18

收录于话题
#算法图文分析

111个 >



The world is a book, and those who do not travel read only a page.

世界是一本书，不旅行的人只读了其中的一页。

问题描述

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

示例 1:

```
输入: [3,2,3,null,3,null,1]

     3
    / \
```

```
    2   3
     \   \
      3   1

输出：7
解释：小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = 7.
```

示例 2：

```
输入：[3,4,5,1,3,null,1]

      3
     / \
    4   5
   / \   \
  1   3   1

输出：9
解释：小偷一晚能够盗取的最高金额 = 4 + 5 = 9.
```

动态规划解决

前面我们刚讲过484，打家劫舍 II和479，递归方式解打家劫舍，今天这题和之前讲的这两道题完全不同，因为前面两题所偷窃的房屋都是数组，而这题是一个二叉树。但这道题也可以使用动态规划来解决。

对于二叉树的每一个节点都有两种状态，一种是偷，一种是不偷。我们定义一个长度为2的数组dp，其中dp[0]表示不偷当前这个节点所能偷窃的最高金额，dp[1]表示偷当前节点所能偷窃的最高金额。

我们就从根节点开遍历这棵二叉树。
如果偷根节点，那么就不能偷根节点的两个子节点，所以
dp[1]=root.val+left.dp[0]+right.dp[0];
这里的伪代码left.dp[0]表示的是不能偷当前节点的左子节点

如果不偷根节点，那么我们可以偷子节点也可以不偷子节点，我们取最大值即可，所以
dp[0]=max(left.dp[0],left.dp[1])+max(right.dp[0],right.dp[1]);

那么边界条件是什么呢，就是节点为空的时候，直接返回0即可。

有了递推公式和边界条件，我们来看下最终代码

```java
public int rob(TreeNode root) {
    int[] robHelp = robHelper(root);
    //取偷根节点和不偷根节点的最大值
    return Math.max(robHelp[1], robHelp[0]);
}

public int[] robHelper(TreeNode root) {
    //边界条件
    if (root == null)
        return new int[2];
    //这里的left是个长度为2的一维数组，其中left[0]表示不偷root.left节点
    //所能偷窃的最大金额，left[1]表示偷root.left节点所能偷窃的最大金额。
    int[] left = robHelper(root.left);
    //right节点同left
    int[] right = robHelper(root.right);
    //Math.max(right[0], right[1]), root.val + left[0] + right[0]表示
    //的是不能偷当前节点，所以可以偷两个子节点，也可以不偷子节点，我们取最大的。
    //root.val + left[1] + right[1]表示的是偷当前节点，所以不能偷两个子节点。
    return new int[]{Math.max(left[0], left[1]) +
        Math.max(right[0], right[1]), root.val + left[0] + right[0]};
}
```

总结

如果对二叉树的遍历方式比较熟悉的话，上面代码一看就知道，他和二叉树的后续遍历非常相似，是完全从下到上的一种遍历方式，每一个节点都记录了两种状态，一种是偷，一种是不偷。每次从下往上走的时候这两种状态都会记录下来……一直到根节点，最后我们只需要返回偷根节点和不偷根节点的最大值即可。

往期推荐

- 465. 递归和动态规划解三角形最小路径和
- 470，DFS和BFS解合并二叉树
- 455，DFS和BFS解被围绕的区域
- 445，BFS和DFS两种方式解岛屿数量

492，动态规划和贪心算法解买卖股票的最佳时机 II

原创 山大王wld 数据结构和算法 2020-12-16

收录于话题
#算法图文分析

111个 >



The world is a fine place and worth fighting for.

这世界是值得为之奋斗的好地方。

问题描述

给定一个数组，它的第i个元素是一支给定股票第i天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

> **输入**: [7,1,5,3,6,4]
> **输出**: 7
> **解释**: 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5-1 = 4 。
> 随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 = 6-3 = 3 。

示例 2:

> **输入**: [1,2,3,4,5]
> **输出**: 4
> **解释**: 在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5-1 = 4 。
> 注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。
> 因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。

示例 3:

> **输入**: [7,6,4,3,1]
> **输出**: 0
> **解释**: 在这种情况下，没有交易完成，所以最大利润为 0。

提示:

- 1<=prices.length<=3*10^4
- 0<=prices[i]<=10^4

动态规划解决

前面刚讲过490，动态规划和双指针解买卖股票的最佳时机，其中也有动态规划解决的方式，这两题非常相似，不同的是第490题最多只能有一笔交易，而这题可以有多笔交易。所以这题我们也可以参照第490题来解。

定义dp[i][0]表示第i+1天交易完之后手里没有股票的最大利润，dp[i][1]表示第i+1天交易完之后手里持有股票的最大利润。

当天交易完之后手里没有股票可能有两种情况，一种是当天没有进行任何交易，又因为当天手里没有股票，所以当天没有股票的利润只能取前一天手里没有股票的利润。一种是把当天手里的股票给卖了，既然能卖，说明手里是有股票的，所以这个时候当天没有股票的利润要取前一天手里有股票的利润加上当天股票能卖的价格。这两种情况我们取利润最大的即可，所以可以得到
dp[i][0]=max(dp[i-1][0],dp[i-1][1]+prices[i]);

当天交易完之后手里持有股票也有两种情况，一种是当天没有任何交易，又因为当天手里持有股票，所以当天手里持有的股票其实前一天就已经持有了。还一种是当天买入了

股票，当天能卖股票，说明前一天手里肯定是没有股票的，我们取这两者的最大值，所以可以得到
dp[i][1]=max(dp[i-1][1],dp[i-1][0]-prices[i]);

动态规划的递推公式有了，那么边界条件是什么，就是第一天
如果买入：dp[0][1]=-prices[0];
如果没买：dp[0][0]=0;

有了递推公式和边界条件，代码很容易就写出来了。

```
public int maxProfit(int[] prices) {
    if (prices == null || prices.length < 2)
        return 0;
    int length = prices.length;
    int[][] dp = new int[length][2];
    //初始条件
    dp[0][1] = -prices[0];
    dp[0][0] = 0;
    for (int i = 1; i < length; i++) {
        //递推公式
        dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
        dp[i][1] = Math.max(dp[i - 1][1], dp[i - 1][0] - prices[i]);
    }
    //最后一天肯定是手里没有股票的时候，利润才会最大，
    //只需要返回dp[length - 1][0]即可
    return dp[length - 1][0];
}
```

代码优化

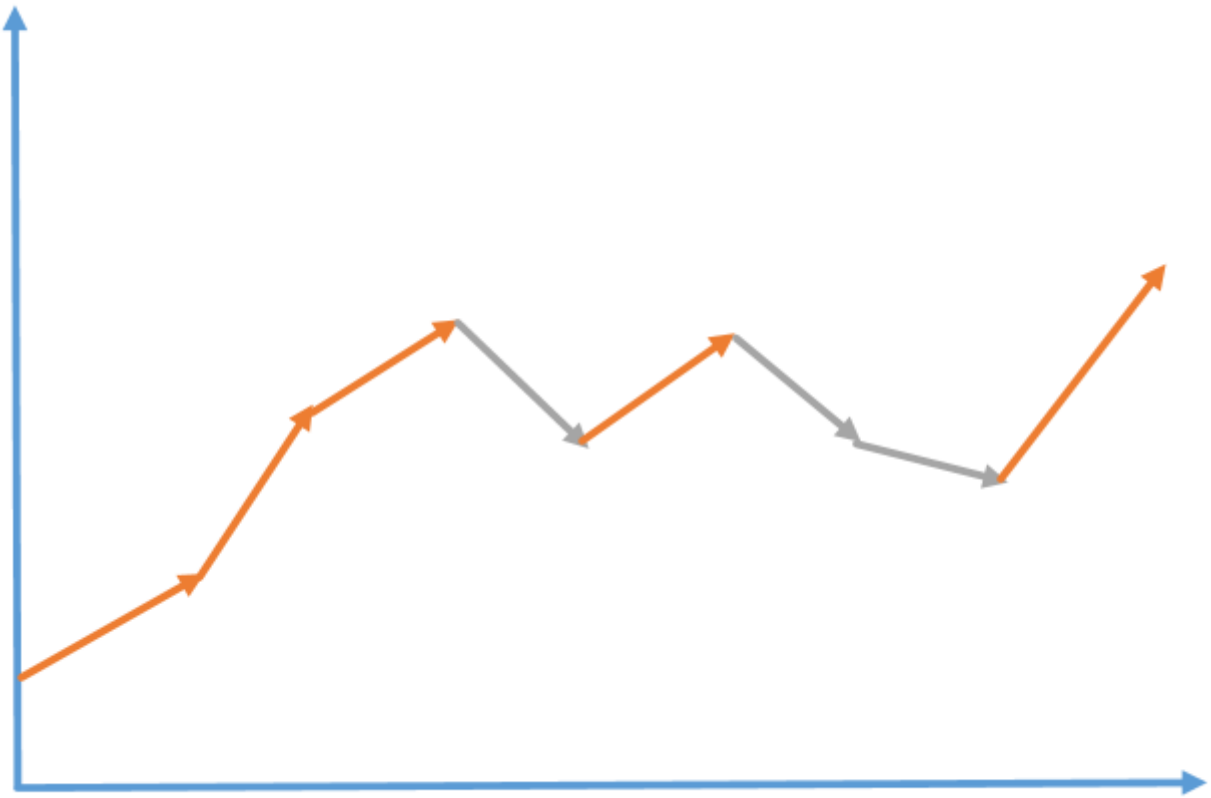
上面计算的时候我们看到当天的利润只和前一天有关，没必要使用一个二维数组，只需要使用两个变量，一个记录当天交易完之后手里持有股票的最大利润，一个记录当天交易完之后手里没有股票的最大利润，来看下代码

```
public int maxProfit(int[] prices) {
    if (prices == null || prices.length < 2)
        return 0;
    int length = prices.length;
    //初始条件
    int hold = -prices[0];//持有股票
    int noHold = 0;//没持有股票
    for (int i = 1; i < length; i++) {
        //递推公式转化的
        noHold = Math.max(noHold, hold + prices[i]);
        hold = Math.max(hold, noHold - prices[i]);
    }
    //最后一天肯定是手里没有股票的时候利润才会最大，
    //所以这里返回的是noHold
    return noHold;
}
```

贪心算法解决

下面我随便画了一个股票的曲线图，可以看到如果股票一直上涨，只需要找到股票上涨的最大值和股票开始上涨的最小值，计算他们的差就是这段时间内股票的最大利润。如

果股票下跌就不用计算，最终只需要把所有股票上涨的时间段内的利润累加就是我们所要求的结果



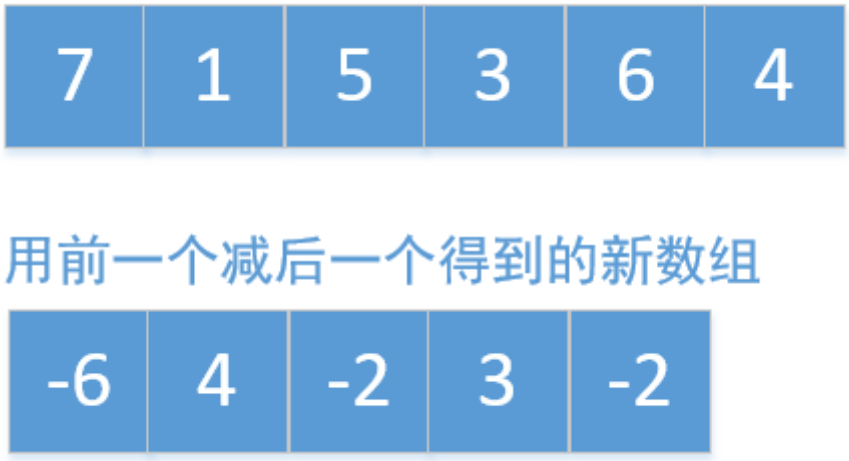
来看下代码

```
public int maxProfit(int[] prices) {
  if (prices == null || prices.length < 2)
    return 0;
  int total = 0, index = 0, length = prices.length;
  while (index < length) {
    //如果股票下跌就一直找，直到找到股票开始上涨为止
    while (index < length - 1 && prices[index] >= prices[index + 1])
      index++;
    //股票上涨开始的值，也就是这段时间上涨的最小值
    int min = prices[index];
    //一直找到股票上涨的最大值为止
    while (index < length - 1 && prices[index] <= prices[index + 1])
      index++;
    //计算这段上涨时间的差值，然后累加
    total += prices[index++] - min;
  }
  return total;
}
```

贪心算法是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，算法得到的是在某种意义上的局部最优解。

那么这道题使用贪心算法也是最容易解决的，只要是上涨的我们就要计算他们的差值进行累加，不需要再找开始上涨的最小值和最大值。为什么能这样计算，我举个例子。

比如a<b<c<d，因为从a到d一直是上涨的，那么最大值和最小值的差值就是d-a，也可以写成(b-a)+(c-b)+(d-c)，搞懂了这个公式所有的一切都明白了。如果还不明白，可以想象成数组中前一个值减去后一个值，构成一个新的数组，我们只需要计算这个新数组中正数的和即可，这里以示例1为例画个图看下



这里只需要计算新数组中正数的和，也就是4+3=7。这个时候代码就已经非常简化了，我们来看下

```
public int maxProfit(int[] prices) {
    int total = 0;
    for (int i = 0; i < prices.length - 1; i++) {
        //原数组中如果后一个减去前一个是正数，说明是上涨的，
        //我们就要累加，否则就不累加
        total += Math.max(prices[i + 1] - prices[i], 0);
    }
    return total;
}
```

总结

这题使用动态规划很容易理解，具体也可以看下[490，动态规划和双指针解买卖股票的最佳时机]。但这道题还可以使用贪心算法，只要是上涨的就要累加，但我们要明白累加的意义，他并不是说一定要当天买当天卖，比如从a到d一直是上涨的，我们一直累加其实就相当于在a的时候买，在d的时候卖。

往期推荐

- 490，动态规划和双指针解买卖股票的最佳时机
- 486，动态规划解最大子序和
- 407，动态规划和滑动窗口解决最长重复子数组
- 395，动态规划解通配符匹配问题

490，动态规划和双指针解买卖股票的最佳时机

原创 山大王wld 数据结构和算法 2020-12-14

收录于话题
#算法图文分析

111个 >



We do not remember days, we remember moments.

我们往往记住的不是某一天，而是某个时刻。

问题描述

给定一个数组，它的第i个元素是一支给定股票第i天的价格。
如果你最多只允许完成一笔交易（即买入和卖出一支股票一次），设计一个算法来计算你所能获取的最大利润。
注意：你不能在买入股票前卖出股票。

示例 1:

> **输入**: [7,1,5,3,6,4]
> **输出**: 5
> **解释**: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。
> 注意利润不能是 7-1 = 6，因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。

示例 2:

> **输入**: [7,6,4,3,1]
> **输出**: 0
> **解释**: 在这种情况下，没有交易完成，所以最大利润为 0。

动态规划解决

这题是让求完成一笔交易所获得的最大利润，首先我们来看一下使用动态规划该怎么解决，动态规划还是那常见的几个步骤

- 确定状态
- 找到转移公式
- 确定初始条件以及边界条件
- 计算结果

我们来定义一个二维数组dp[length][2]，其中dp[i][0]表示第i+1天（i是从0开始的）结束的时候没持有股票的最大利润，dp[i][1]表示第i+1天结束的时候持有股票的最大利润。

如果我们要求第i+1天结束的时候没持有股票的最大利润dp[i][0]，那么会有两种情况。
第一种情况就是第i+1天我们即没买也没卖，那么最大利润就是第i天没持有股票的最大利润dp[i-1][0]。
第二种情况就是第i+1天我们卖了一支股票，那么最大利润就是第i天持有股票的最大利润（这个是负的，并且也不一定是第i天开始持有的，有可能在第i天之前就已经持有了）加上第i+1天卖出股票的最大利润，dp[i-1][1]+prices[i]。

很明显我们可以得出
dp[i][0]=max(dp[i-1][0],dp[i-1][1]+prices[i]);

同理我们可以得出第i+1天结束的时候我们持有股票的最大利润
dp[i][1]=max(dp[i-1][1],-prices[i]);

边界条件就是第1天的时候，如果我们不持有股票，那么
dp[0][0]=0;
如果持有股票，那么
dp[0][1]=-prices[0];

有了边界条件和递推公式，代码就很容易写出来了，来看下代码

```
public int maxProfit(int[] prices) {
    if (prices == null || prices.length == 0)
        return 0;
    int length = prices.length;
    int[][] dp = new int[length][2];
    //边界条件
    dp[0][0]= 0;
    dp[0][1] = -prices[0];
    for (int i = 1; i < length; i++) {
        //递推公式
        dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
        dp[i][1] = Math.max(dp[i - 1][1], -prices[i]);
    }
    //毋庸置疑，最后肯定是手里没持有股票利润才会最大，也就是卖出去了
    return dp[length - 1][0];
}
```

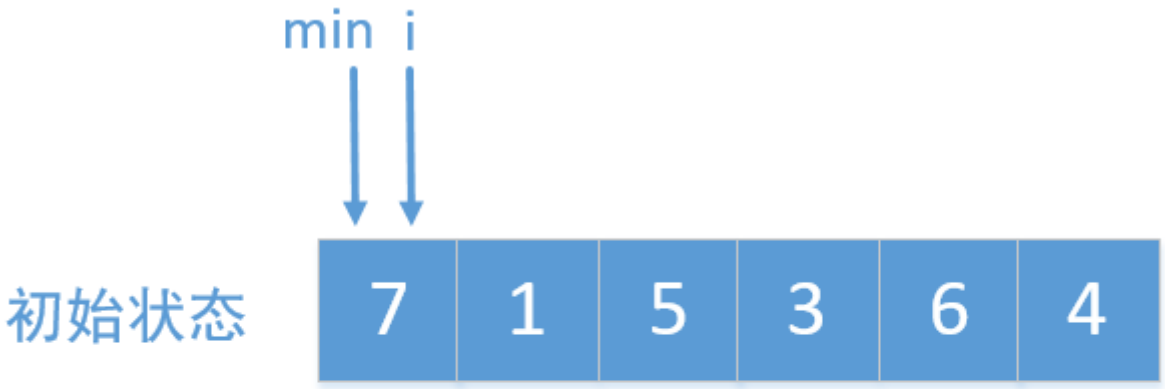
代码优化

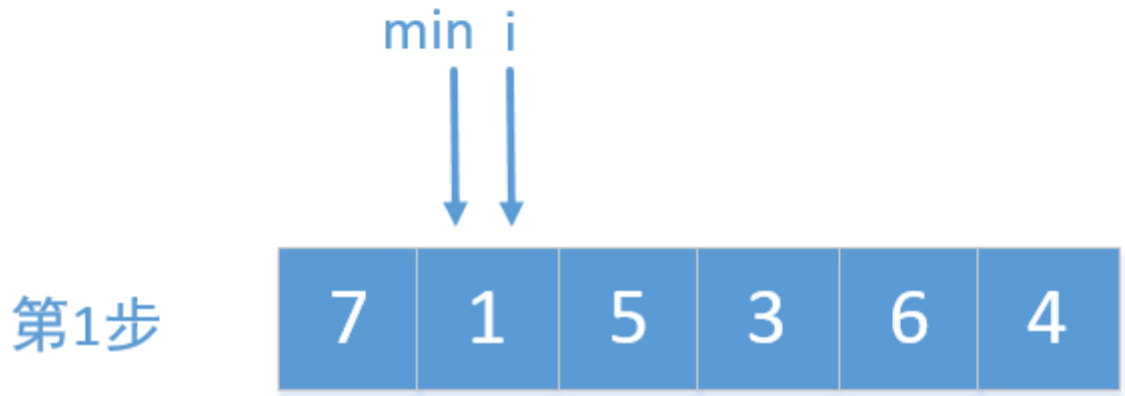
我们看到上面二维数组中计算当天的最大利润只和前一天的利润有关，所以没必要使用二维数组，只需要使用两个变量即可，一个表示当天持有股票的最大利润，一个表示当天没持有股票的最大利润，代码如下。

```
public int maxProfit(int[] prices) {
    if (prices == null || prices.length == 0)
        return 0;
    int length = prices.length;
    int hold = -prices[0];//持有股票
    int noHold = 0;//不持有股票
    for (int i = 1; i < length; i++) {
        //递推公式
        noHold = Math.max(noHold, hold + prices[i]);
        hold = Math.max(hold, -prices[i]);
    }
    //毋庸置疑，最后肯定是手里没持有股票利润才会最大，
    //也就是卖出去了
    return noHold;
}
```

双指针解决

我们还可以使用两个指针，一个指针记录访问过的最小值（注意这里是访问过的最小值），一个指针一直往后走，然后计算他们的差值，保存最大的即可，这里就以示例1为例来画个图看下

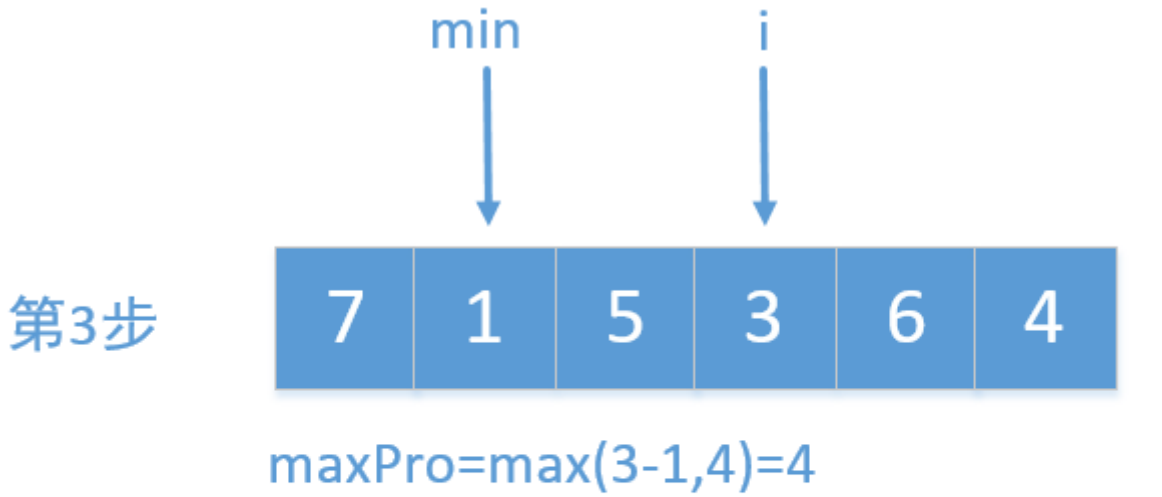


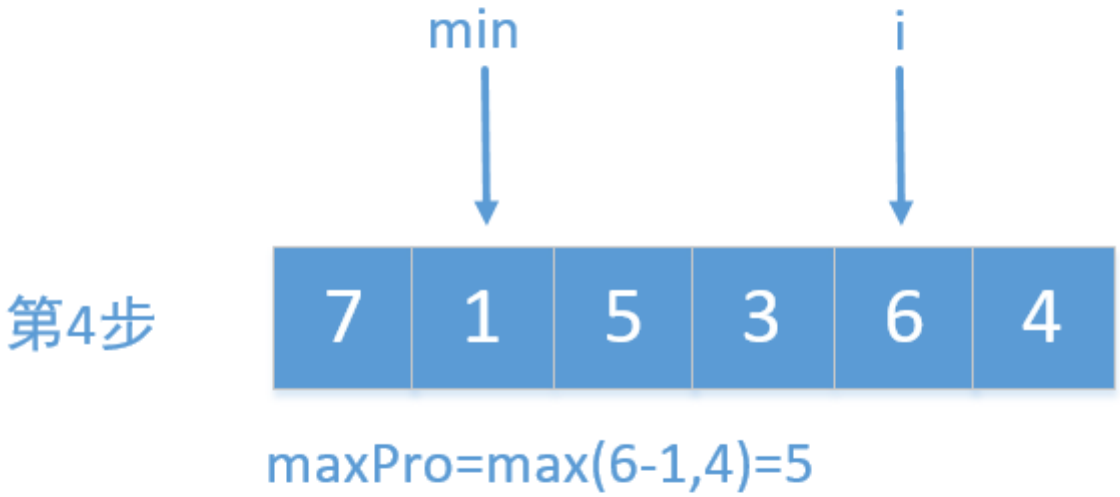


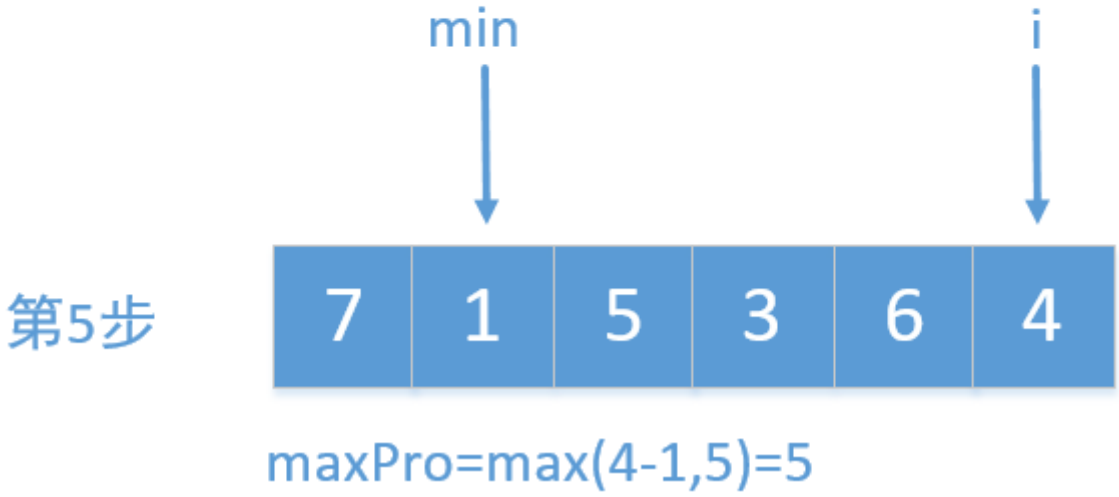
maxPro=max(1-1,0)=0



maxPro=max(5-1,0)=4







原理比较简单，来看下代码

```
public static int maxProfit(int[] prices) {
  if (prices == null || prices.length == 0)
    return 0;
  int maxPro = 0;//记录最大利润
  int min = prices[0];//记录数组中访问过的最小值
  for (int i = 1; i < prices.length; i++) {
    min = Math.min(min, prices[i]);
    maxPro = Math.max(prices[i] - min, maxPro);
  }
```

```
    return maxPro;
}
```

单调栈解决

单调栈解决的原理很简单，我们要始终保持栈顶元素是**所访问过的元素**中最小的，如果当前元素小于栈顶元素，就让栈顶元素出栈，让当前元素入栈。如果访问的元素大于栈顶元素，就要计算他和栈顶元素的差值，我们记录最大的即可，代码如下。

```
public int maxProfit(int[] prices) {
    if (prices == null || prices.length == 0)
        return 0;
    Stack<Integer> stack = new Stack<>();
    stack.push(prices[0]);
    int max = 0;
    for (int i = 1; i < prices.length; i++) {
        //如果栈顶元素大于prices[i]，那么栈顶元素出栈，
        //把prices[i]压栈，要始终保证栈顶元素是最小的
        if (stack.peek() > prices[i]) {
            stack.pop();
            stack.push(prices[i]);
        } else {
            //否则如果栈顶元素不大于prices[i]，就要计算
            //prices[i]和栈顶元素的差值
            max = Math.max(max, prices[i] - stack.peek());
        }
    }
    return max;
}
```

仔细看下就会明白这种解法其实就是双指针的另一种实现方式，只不过双指针使用的是一个变量记录访问过的最小值，而这里使用的是栈记录的。

参照最大子序和

在前面刚讲过最大子序和的问题，不明白的可以看下486，动态规划解最大子序和，今天这题完全可以参照第486的解题思路。

假设数组的值是[a,b,c,d,e,f]，我们用数组的前一个值减去后一个值，得到的新数组如下

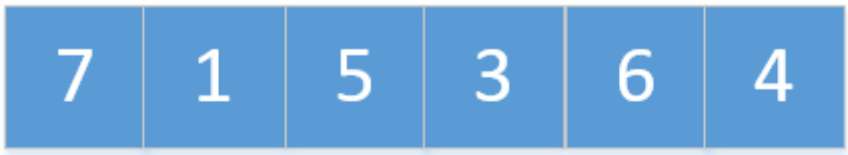
[b-a,c-b,d-c,e-d,f-e]

我们在新数组中随便找几个连续的数字相加就会发现一个规律，就是中间的数字都可以约掉，比如新数组中第1个到第4个数字的和是

b-a+c-b+d-c+e-d=e-a。

我们来看下示例1中得到的新数组，连续的最大值就是

4+(-2)+3=5。



用前一个减后一个得到的新数组



搞懂了上面的原理代码就简单多了，我们前面刚讲的最大子序和的代码如下

```
public int maxSubArray(int[] num) {
    int length = num.length;
    int cur = num[0];
    int max = cur;
    for (int i = 1; i < length; i++) {
        cur = Math.max(cur, 0) + num[i];
        //记录最大值
        max = Math.max(max, cur);
    }
    return max;
}
```

然后我们来对他进行修改一下，就是今天这题的答案了。

```
public int maxProfit(int[] prices) {
    if (prices == null || prices.length == 0)
        return 0;
    int length = prices.length;
    int cur = 0;
    int max = cur;
    for (int i = 1; i < length; i++) {
        //这地方把prices[i]改为 prices[i] - prices[i - 1]即可
        cur = Math.max(cur, 0) + prices[i] - prices[i - 1];
        //记录最大值
        max = Math.max(max, cur);
    }
    return max;
}
```

暴力解决

这种是两两比较，保存计算的最大值即可，没啥可说的，虽然简单，但效率很差，看下代码

```
public int maxProfit(int[] prices) {
    if (prices == null || prices.length == 0)
        return 0;
    int maxPro = 0;
    for (int i = 0; i < prices.length; i++) {
        for (int j = i + 1; j < prices.length; j++) {
            maxPro = Math.max(maxPro, prices[j] - prices[i]);
        }
    }
    return maxPro;
}
```

总结

这道题解法比较多，暴力求解就不说了，除了暴力求解，双指针应该是最容易想到的。其中参照最大子序和求解也算是比较巧妙的一种解题思路。

往期推荐

- 486，动态规划解最大子序和
- 465. 递归和动态规划解三角形最小路径和
- 423，动态规划和递归解最小路径和
- 413，动态规划求最长上升子序列

486，动态规划解最大子序和

原创 山大王wld 数据结构和算法 今天

收录于话题
#算法图文分析

96个 >



When a friendship is real, you can feel it.

交到真心朋友的时候，你是可以感觉到的。

问题描述

给定一个整数数组 `nums` ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例:

> **输入**: [-2,1,-3,4,-1,2,1,-5,4]
> **输出**: 6
> **解释**: 连续子数组[4,-1,2,1]的和最大，为6。

动态规划解决

这题是让求最大的连续子序和，如果不是连续的非常简单，只需要把所有的正数相加即可。但这里说的是连续的，中间可能掺杂负数，如果求出一个最大子序和在加上负数肯定要比原来小了。解这题最简单的一种方式就是使用动态规划。

我们先来了解一下动态规划的几个步骤

1，确定状态

2，找到转移公式

3，确定初始条件以及边界条件

4，计算结果。

最后一个不用看，只看前3个就行，因为前3个一旦确定，最后一个结果也就出来了。我们试着找一下

1，定义dp[i]表示数组中前i+1（注意这里的i是从0开始的）个元素构成的连续子数组的最大和。

2，如果要计算前i+1个元素构成的连续子数组的最大和，也就是计算dp[i]，只需要判断dp[i-1]是大于0还是小于0。如果dp[i-1]大于0，就继续累加，dp[i]=dp[i-1]+num[i]。如果dp[i-1]小于0，我们直接把前面的舍弃，也就是说重新开始计算，否则会越加越小的，直接让dp[i]=num[i]。所以转移公式如下

dp[i]=num[i]+max(dp[i-1],0);

3，边界条件判断，当i等于0的时候，也就是前1个元素，他能构成的最大和也就是他自己，所以

dp[0]=num[0];

找到了上面的转移公式，代码就简单多了，来看下

```
1   public int maxSubArray(int[] num) {
2     int length = num.length;
3     int[] dp = new int[length];
4     //边界条件
5     dp[0] = num[0];
6     int max = dp[0];
7     for (int i = 1; i < length; i++) {
8       //转移公式
9       dp[i] = Math.max(dp[i - 1], 0) + num[i];
10      //记录最大值
11      max = Math.max(max, dp[i]);
12    }
13    return max;
14  }
```

代码优化

仔细看下上面的代码会发现，我们申请了一个长度为length的数组，但在转移公式计算的时候，每次计算当前值的时候只会用到前面的那个值，再往前面就用不到了，这样实际上是造成了空间的浪费。这里不需要一个数组，只需要一个临时变量即可，看下代码

```
1   public int maxSubArray(int[] num) {
```

```
    int length = num.length;
    int cur = num[0];
    int max = cur;
    for (int i = 1; i < length; i++) {
      cur = Math.max(cur, 0) + num[i];
      //记录最大值
      max = Math.max(max, cur);
    }
    return max;
  }
```

总结

动态规划最重要的3步就是先确定状态，最关键的是找出转移公式，最后再确定边界条件，防止数组越界等问题，这3步确定以后基本上就能解决了。

往期推荐

- 477，动态规划解按摩师的最长预约时间
- 465. 递归和动态规划解三角形最小路径和
- 413，动态规划求最长上升子序列
- 371，背包问题系列之-基础背包问题

477，动态规划解按摩师的最长预约时间

原创 山大王wld 数据结构和算法 11月16日

收录于话题
#算法图文分析

95个 >



Life is a collection of moments. The idea is to have as many good ones as you can.

生命由一系列的瞬间组成。宗旨是尽可能地拥有快乐的瞬间。

问题描述

一个有名的按摩师会收到源源不断的预约请求，每个预约都可以选择接或不接。在每次预约服务之间要有休息时间，因此她不能接受相邻的预约。给定一个预约请求序列，替按摩师找到最优的预约集合（总预约时间最长），返回总的分钟数。

示例 1：

> **输入**：[1,2,3,1]
> **输出**：4
> **解释**：选择 1 号预约和 3 号预约，总时长 = 1 + 3 = 4。

示例 2：

> **输入**：[2,7,9,3,1]

> **输出**：12
> **解释**：选择 1 号预约、 3 号预约和 5 号预约，总时长 = 2 + 9 + 1 = 12。

示例 3：

> **输入**：[2,1,4,5,3,1,1,3]
> **输出**：12
> **解释**：选择 1 号预约、 3 号预约、 5 号预约和 8 号预约，总时长 = 2 + 4 + 3 + 3 = 12。

动态规划解决

数组中的值表示的是预约时间，按摩师可以选择接或者不接，如果前一个接了，那么下一个肯定是不能接的，因为按摩师不能接相邻的两次预约。如果上一个没接，那么下一个可以选择接也可以选择不接，视情况而定。

这里可以定义一个二维数组dp[length][2]，其中dp[i][0]表示第i+1（因为数组下标是从0开始的，所以这里是i+1）个预约没有接的最长总预约时间，dp[i][1]表示的是第i+1个预约接了的最长总预约时间。那么我们找出递推公式

1，dp[i][0]=max(dp[i-1][0],dp[i-1][1])
他表示如果第i+1个没有接，那么第i个有没有接都是可以的，我们取最大值即可。

2，dp[i][1]=dp[i-1][0]+nums[i]
他表示的是如果第i+1个接了，那么第i个必须要没接，这里nums[i]表示的是第i+1个预约的时间。

递推公式找出来之后我们再来看下边界条件，第一个预约可以选择接，也可以选择不接，所以

- dp[0][0]=0，第一个没接
- dp[0][1]=nums[0]，第一个接了。

最后再来看下代码

```
1   public int massage(int[] nums) {
2     //边界条件判断
3     if (nums == null || nums.length == 0)
4       return 0;
5     int length = nums.length;
6     int[][] dp = new int[length][2];
7     dp[0][0] = 0;//第1个没接
8     dp[0][1] = nums[0];//第1个接了
9     //从第2个开始判断
10    for (int i = 1; i < length; i++) {
```

```
        //下面两行是递推公式
        dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1]);
        dp[i][1] = dp[i - 1][0] + nums[i];
    }
    //最后取最大值即可
    return Math.max(dp[length - 1][0], dp[length - 1][1]);
}
```

动态规划优化

上面定义了一个二维数组，但每次计算的时候都只是用二维数组的前一对值，在往前面的就永远使用不到了，这样就会造成巨大的空间浪费，所以我们可以定义两个变量来解决，来看下代码

```
public int massage(int[] nums) {
    //边界条件判断
    if (nums == null || nums.length == 0)
        return 0;
    int length = nums.length;
    int dp0 = 0;//第1个没接
    int dp1 = nums[0];//第1个接了
    //从第2个开始判断
    for (int i = 1; i < length; i++) {
        //防止dp0被修改之后对下面运算造成影响，这里
        //使用一个临时变量temp先把结果存起来，计算完
        //之后再赋值给dp0.
        int temp = Math.max(dp0, dp1);
        dp1 = dp0 + nums[i];
        dp0 = temp;
    }
    //最后取最大值即可
    return Math.max(dp0, dp1);
}
```

总结

首先这里都是正规的按摩师😜，使用动态规划是最容易解决的，只要找准递推公式，基本上也没什么难度。

往期推荐

- 465. 递归和动态规划解三角形最小路径和
- 430，剑指 Offer-动态规划求正则表达式匹配
- 423，动态规划和递归解最小路径和
- 413，动态规划求最长上升子序列

465. 递归和动态规划解三角形最小路径和

原创 山大王wld 数据结构和算法 10月20日

收录于话题
#算法图文分析

95个 >



A smile is the best makeup any girl can wear.

微笑是每个女孩最好的化妆品。

问题描述

给定一个三角形，找出自顶向下的最小路径和。每一步只能移动到下一行中相邻的结点上。

相邻的结点在这里指的是下标与上一层结点下标相同或者等于一层结点下标+1的两个结点。

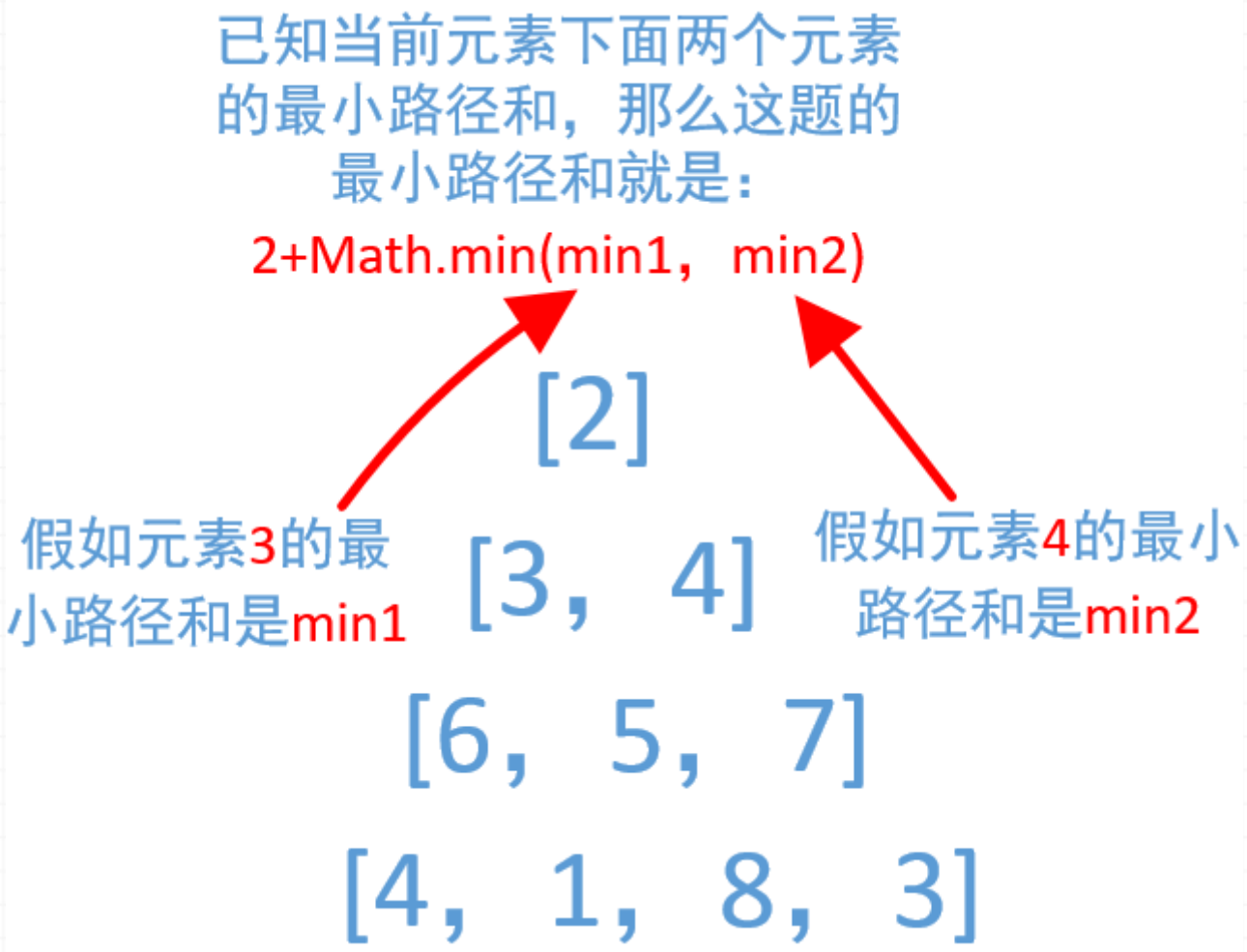
例如，给定三角形：

```
[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
```

自顶向下的最小路径和为 11（即，**2** + **3** + **5** + **1** = 11）。

递归求解

这题让求路径的最小值，如果知道了下面路径的最小值，只需要选择最小的即可，描述不是很明白，这里以示例为例画个图来看一下



对于上面的元素他们的最小路径和都要依赖下面的元素，但最后一行的最小路径和就是他自己了，所以看到这题我们很容易想到的一种解决方式就是递归，来看下，代码中有详细注释。

```
public int minimumTotal(List<List<Integer>> triangle) {
    return minimumTotal(triangle, 0, 0, triangle.size());
}

//line和row分别表示行和列，total表示总共多少行
public int minimumTotal(List<List<Integer>> triangle, int line, int row, int total) {
    //如果行或者列大于total，说明跑到三角形的外面去了，直接返回0。
    if (line >= total || row >= total)
        return 0;
    //left表示下一行左边的最小路径和
    int left = minimumTotal(triangle, line + 1, row, total);
    //left表示下一行右边的最小路径和
    int right = minimumTotal(triangle, line + 1, row + 1, total);
    //返回当前值加上下一行中左右两个最小的路径
    return triangle.get(line).get(row) + (left < right ? left : right);
}
```

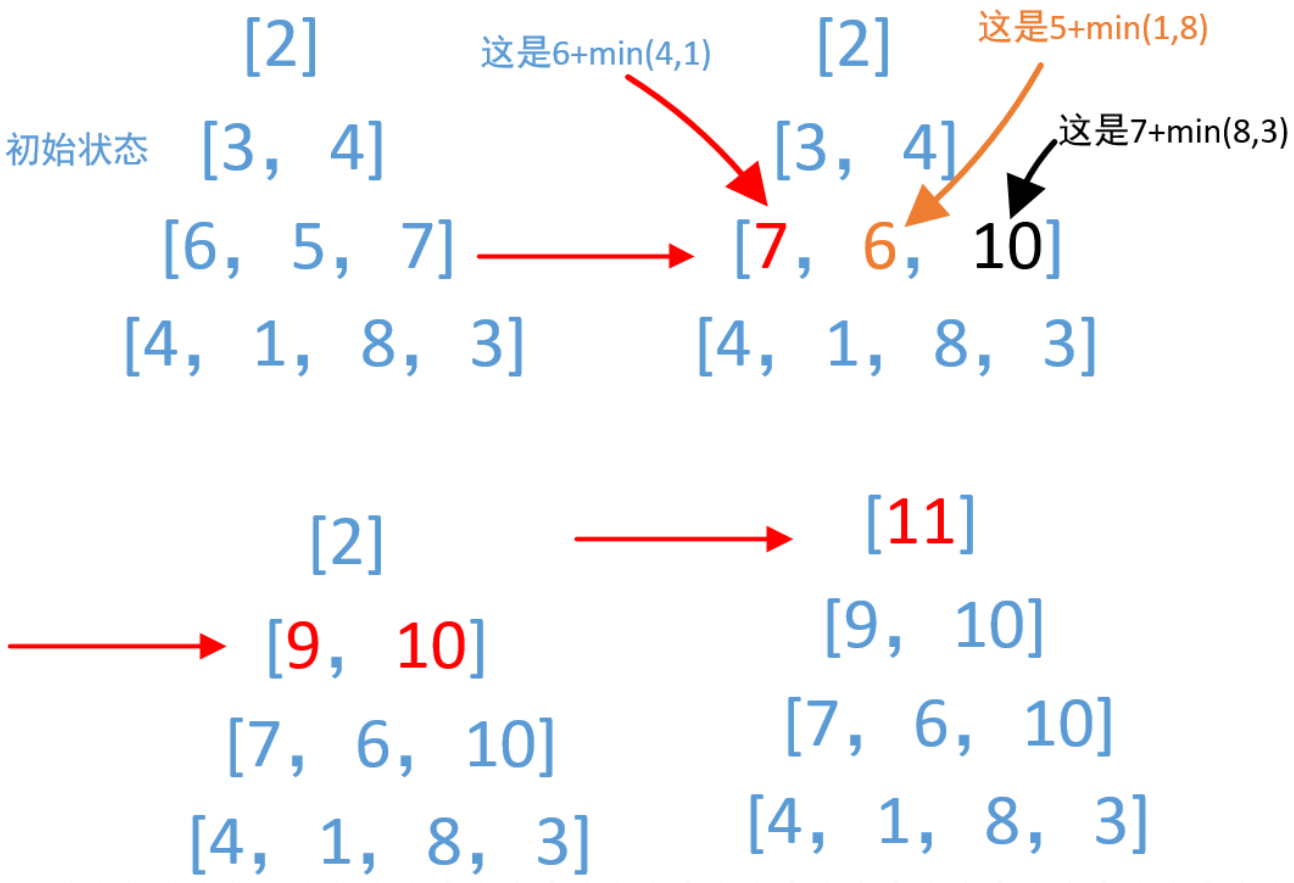
这种解法虽然没有逻辑上的错误，但是执行效率很差，因为他包含大量的重复计算，有点类似斐波那契数量一样，在之前讲过356，青蛙跳台阶相关问题的时候提到过斐波那契数列可以使用map把计算的结果存储起来，下次用的时候如果map中有就直接从map中取，如果map中没有再计算。

所以这题也可以使用一个map，把计算的结果存储起来，来看下代码

```
public int minimumTotal(List<List<Integer>> triangle) {
    return minimumTotal(triangle, 0, 0, triangle.size(), new HashMap());
}

public int minimumTotal(List<List<Integer>> triangle, int line, int row, int total, Map<String, In
    //如果行或者列大于total，说明跑到三角形的外面去了，直接返回0。
    if (line >= total || row >= total)
        return 0;
    String key = line + "-" + row;
    if (map.containsKey(key))
        return map.get(key);
    int left = minimumTotal(triangle, line + 1, row, total, map);
    int right = minimumTotal(triangle, line + 1, row + 1, total, map);
    int sum = triangle.get(line).get(row) + (left < right ? left : right);
    //把计算的结果存储到map中
    map.put(key, sum);
    return sum;
}
```

动态规划解决

递归使用的是从上到下的方式来计算（但实际上计算的时候由于递归的原因，他还是先从下面开始计算，然后把计算的结果返回给上一层调用递归的地方）。其实还可以不使用递归，倒数第一行的最小路径就是他自己，从倒数第二行开始，当前元素的最小路径就是当前元素加上他下一行左右两个元素的最小路径。画个图来看一下



我们来定义一个二维数组dp，他表示当前位置的最小路径和，我们可以得出递推公式

```
dp[i][j]=min(dp[i+1][j]+dp[i+1][j+1])
+triangle.get(i).get(j);
```

他表示当前位置的最小**路径和**是下一行左右两个最小的**路径和**加上当前的值。最后再来看下代码

```
1   public int minimumTotal(List<List<Integer>> triangle) {
2     //定义一个二维数组
3     int[][] dp = new int[triangle.size() + 1][triangle.size() + 1];
4     //从最后一行开始计算
5     for (int i = triangle.size() - 1; i >= 0; i--) {
6       for (int j = 0; j < triangle.get(i).size(); j++) {
7         //递归公式
8         dp[i][j] = Math.min(dp[i + 1][j], dp[i + 1][j + 1]) + triangle.get(i).get(j);
9       }
10    }
11    return dp[0][0];
12  }
13
```

上面代码使用的是二维数组，每一行使用之后就不会再使用了，造成了空间的浪费，其实我们还可以把它改成一维的，就像之前讲的370，最长公共子串和子序列中提到的代码优化那样，最后再来看下代码

```
1   public int minimumTotal(List<List<Integer>> triangle) {
2     int[] dp = new int[triangle.size() + 1];
3     for (int i = triangle.size() - 1; i >= 0; i--) {
4       for (int j = 0; j < triangle.get(i).size(); j++) {
5         //min函数中的dp[j]实际上是下一行的值这里还没有更新
6         dp[j] = Math.min(dp[j], dp[j + 1]) + triangle.get(i).get(j);
7       }
8     }
9     return dp[0];
10  }
```

总结

三角形的最小路径和也是很常见的一道题，使用动态规划从下往上递推应该说效率是最高的，也很容易理解。

往期推荐

- 442，剑指 Offer-回溯算法解二叉树中和为某一值的路径
- 423，动态规划和递归解最小路径和
- 411，动态规划和递归求不同路径 II
- 387，二叉树中的最大路径和

430，剑指 Offer-动态规划求正则表达式匹配

原创 山大王wld 数据结构和算法 8月14日

收录于话题
#剑指offer

27个 >



Hope is a good thing, maybe the best of things, and no good thing ever dies.

希望是件好事，可能也是这世间最美好的事物，而美好的事情永不磨灭。

问题描述

请实现一个函数用来匹配包含'. '和'*'的正则表达式。模式中的字符'.'表示任意一个字符，而'*'表示它前面的字符可以出现任意次（含0次）。在本题中，匹配是指字符串的所有字符匹配整个模式。

例如，字符串"aaa"与模式"a.a"和"ab*ac*a"匹配，但与"aa.a"和"ab*a"均不匹配。

示例 1:

```
输入:
s = "aa"
p = "a"
```

> 输出: false
> 解释: "a" 无法匹配 "aa" 整个字符串。

示例 2:

> 输入:
> s = "aa"
> p = "a*"
>
> 输出: true
> 解释: 因为 '*' 代表可以匹配零个或多个前面的那一个元素, 在这里前面的元素就是 'a'。因此, 字符串 "aa" 可被视为 'a' 重复了一次。

示例 3:

> s = "ab"
> p = ".*"
>
> 输出: true
> 解释: ".*" 表示可匹配零个或多个（'*'）任意字符（'.'）。

示例 4:

> 输入:
> s = "aab"
> p = "c*a*b"
>
> 输出: true
> 解释: 因为 '*' 表示零个或多个, 这里 'c' 为 0 个, 'a' 被重复一次。因此可以匹配字符串 "aab"。

示例 5:

> 输入:
> s = "mississippi"
> p = "mis*is*p*."
>
> 输出: false

- s 可能为空，且只包含从 a-z 的小写字母。
- p 可能为空，且只包含从 a-z 的小写字母以及字符 . 和 *，无连续的 '*'。

动态规划求解

这题是剑指offer的第19题，难度是困难。我们也可以看下之前写的一道和这题非常类似的一道题395，动态规划解通配符匹配问题，今天这题和第395题有一点不同的是，第395题的“*”可以匹配任意字符串，而这题的“*”表示他前面的字符可以出现任意次（包含0次）。

我们先定义一个二维数组dp，其中dp[i][j]表示的是p的前j个字符和s的前i个字符匹配的结果。

一，边界条件

我们默认dp[0][0]=true；也就是p的前0个字符和s的前0个字符是可以匹配的。因为字符“*”表示的是匹配他前面的字符0次或者多次，如果p的字符类似于“a*b*c”，那么字符“*”是可以消去前面的一个字符的。我们就以字符"a*b*c"为例来画个图看一下

默认dp[0][0]=true

第1步



dp[0][1]=false。因为p的第1个字符”a”不可能和s的前0个字符匹配成功

第2步



dp[0][2]=ture。因为p的第2个字符”*”是可以消去前面的字符”a”，dp[0][2]相当于p的前0个字符和s的前0个字符是否匹配，因为dp[0][0]=true，所以dp[0][2]=true

第3步



dp[0][3]=false。因为p的第3个字符”b”不可能和s的前0个字符匹配成功

第4步



dp[0][4]=ture。因为p的第4个字符”*”是可以消去前面的字符”b”，dp[0][4]相当于p的前2个字符和s的前0个字符是否匹配，因为dp[0][2]=true，所以dp[0][4]=true

第5步



dp[0][5]=false。因为p的第5个字符”c”不可能和s的前0个字符匹配成功

所以边界条件的代码如下

```
public boolean isMatch(String s, String p) {
    if (s == null || p == null)
        return false;
    int m = s.length();
    int n = p.length();
    boolean[][] dp = new boolean[m + 1][n+1];
    dp[0][0] = true;
    for (int i = 0; i < n; i++) {
        //如果p的第i+1个字符也就是p.charAt(i)是"*"的话，
        //那么他就可以把p的第i个字符给消掉（也就是匹配0次）。
        //我们只需要判断p的第i-1个字符和s的前0个字符是否匹
        //配即可。比如p是"a*b*"，如果要判断p的第4个字符
        //"*"和s的前0个字符是否匹配，因为字符"*"可以消去
        //前面的任意字符，只需要判断p的"a*"和s的前0个字
        //符是否匹配即可
        if (p.charAt(i) == '*' && dp[0][i - 1]) {
            dp[0][i + 1] = true;
        }
    }
    ……
}
```

边界条件我们已经找到了，下面再来看一下递推公式。

二，递推公式

1，如果p的第j+1个字符和s的第i+1个字符相同，或者p的第j+1个字符是“.”（"."可以匹配任意字符），我们只需要判断p的前j个字符和s的前i个字符是否匹配，这个还好理解，我们画个图看一下

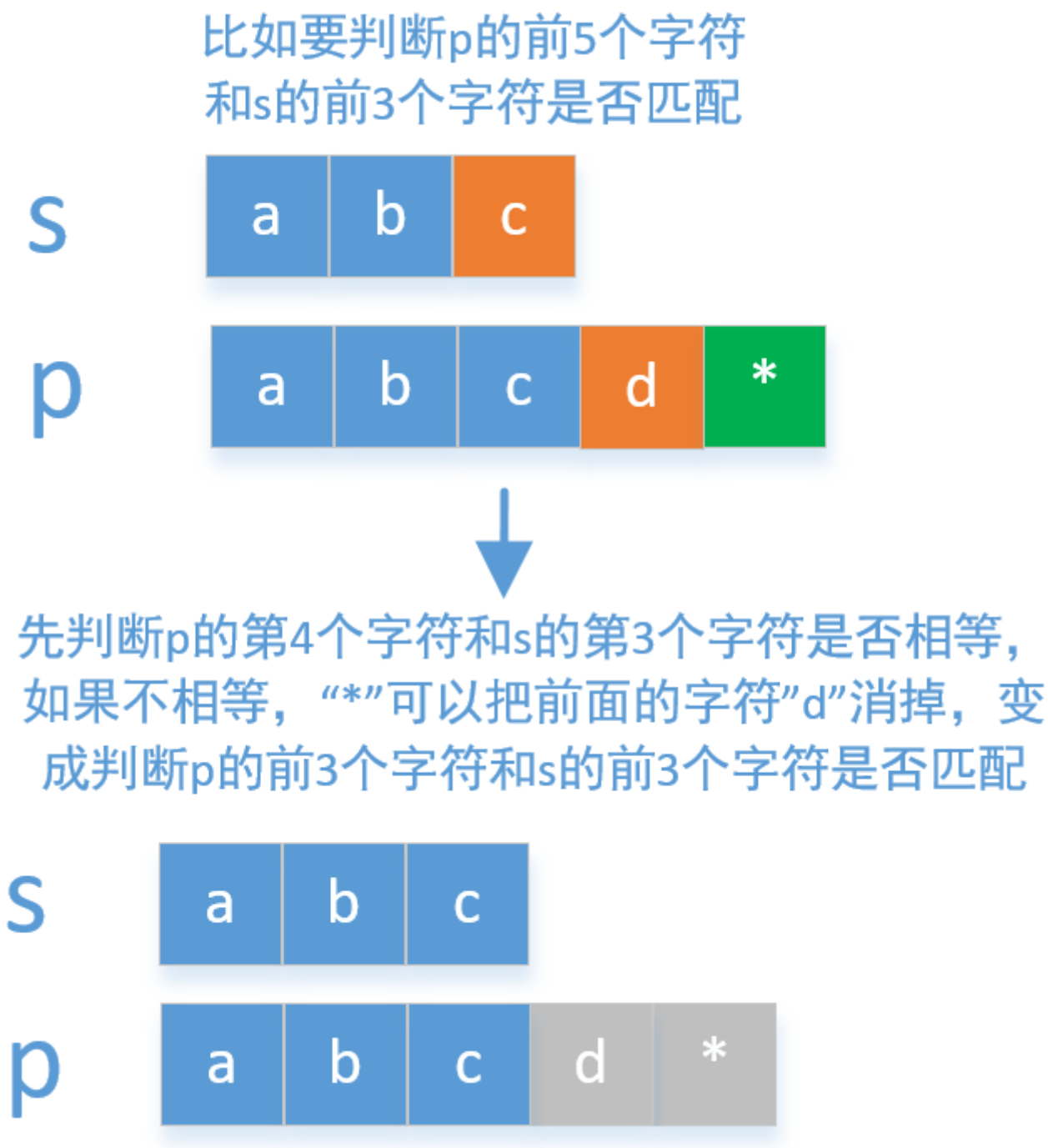




p的第3个字符"."是可以和s的第3个字符"f"匹配成功的，我们只需要判断p的前2个字符和s的前2个字符是否匹配成功即可。
代码如下

```
if (p.charAt(j) == s.charAt(i) || p.charAt(j) == '.') {
        dp[i + 1][j + 1] = dp[i][j];
}
```

2，如果p的第j+1个字符和s的第i+1个字符不能匹配，并且p的第j+1个字符是"*"，那么就要分两种情况
（1）p的第j个字符和s的第i+1个字符不能匹配，
比如：s="abc"，p="abcd*"
我们就让p的第j个和第j+1个字符同时消失，也就是让"d*"消失，只需要判断p的前j-1个字符和s的前i+1个字符是否匹配即可。
也就是下面这样



代码如下

```
if (p.charAt(j) == '*') {
    if (p.charAt(j - 1) != s.charAt(i) && p.charAt(j - 1) != '.') {
        dp[i + 1][j + 1] = dp[i + 1][j - 1];
    }
}
```

（2）p的第j个字符和s的第i+1个字符匹配成功，有3种情况

- 类似于s="abc"，p="abcc*"； 我们就让*匹配**0个**，把p的"c*"砍掉，判断s="abc"和p="abc"是否匹配

```
dp[i+1][j+1] = dp[i+1][j-1]
```

- 类似于s="abc"，p="abc*"； 我们就让*匹配**1个**，把p的字符"*"砍掉，判断s="abc"和p="abc"是否匹配

```
dp[i+1][j+1] = dp[i+1][j]
```

- 类似于s="abcc"(或者s="abccc"，s="abcccc"……)，p="abc*"；我们就让*匹配**多个**，把s的最后一个字符"c"砍掉，判断s="abc"(或者s="abcc"，s="abccc"……)和p="abc*"是否匹配

```
dp[i+1][j+1] = dp[i][j+1]
```

前面两个的递推公式很好理解，关键是第3个为什么要这样写。其实我们可以这样想，把"c*"看做是一个整体，比如"abccc"的最后一个字符"c"和p的倒数第二个字符匹配成功，因为"c*"可以匹配多个，我们就把"abccc"砍掉一个字符"c"，然后再判断"abcc"和"abc*"是否匹配。

上面三个递推公式只要有一个为true，就表示能够匹配成功

我们来看下完整的递推公式

```
if (p.charAt(j) == s.charAt(i) || p.charAt(j) == '.') {
  dp[i + 1][j+1] = dp[i][j];
} else if (p.charAt(j) == '*') {
  if (p.charAt(j - 1) != s.charAt(i) && p.charAt(j - 1) != '.') {
    dp[i + 1] [j+1]= dp[i + 1][j-1];
  } else {
    dp[i + 1][j+1] = (dp[i + 1][j] || dp[i][j+1]|| dp[i + 1][j-1]);
  }
}
```

其实上面代码有个重复的地方就是当p的第j+1个字符是"*"的时候，里面的两种判断方式都会有一个匹配0个的判断，我们可以把它提取出来，像下面这样

```
 if (p.charAt(j) == s.charAt(i) || p.charAt(j) == '.') {
   dp[i + 1][j+1] = dp[i][j];
 } else if (p.charAt(j) == '*') {
   //递归公式
   if (p.charAt(j - 1) == s.charAt(i) || p.charAt(j - 1) == '.') {
     dp[i + 1][j+1] = dp[i + 1] [j]|| dp[i][j+1];
   }
   dp[i + 1] [j+1]|= dp[i + 1][j-1];
 }

```

实际上匹配1个和匹配多个也可以合并，代码如下

```
if (p.charAt(j) == s.charAt(i) || p.charAt(j) == '.') {
  dp[i + 1][j + 1] = dp[i][j];
} else if (p.charAt(j) == '*') {
  //递归公式
  if (p.charAt(j - 1) == s.charAt(i) || p.charAt(j - 1) == '.') {
    dp[i + 1][j + 1] = dp[i][j];
```

```
    }
    dp[i + 1][j + 1] |= dp[i + 1][j - 1];
  }
```

边界条件和递推公式都有了，我们再来看下完整代码

```
public boolean isMatch(String s, String p) {
  if (s == null || p == null)
    return false;
  int m = s.length();
  int n = p.length();
  boolean[][] dp = new boolean[m + 1][n + 1];
  dp[0][0] = true;
  for (int i = 0; i < n; i++) {
    //如果p的第i+1个字符也就是p.charAt(i)是"*"的话，
    //那么他就可以把p的第i个字符给消掉（也就是匹配0次）。
    //我们只需要判断p的第i-1个字符和s的前0个字符是否匹
    //配即可。比如p是"a*b*"，如果要判断p的第4个字符
    //"*"和s的前0个字符是否匹配，因为字符"*"可以消去
    //前面的任意字符，只需要判断p的"a*"和s的前0个字
    //符是否匹配即可
    if (p.charAt(i) == '*' && dp[0][i - 1]) {
      dp[0][i + 1] = true;
    }
  }
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      if (p.charAt(j) == s.charAt(i) || p.charAt(j) == '.') {
        dp[i + 1][j + 1] = dp[i][j];
      } else if (p.charAt(j) == '*') {
        //递归公式
        if (p.charAt(j - 1) == s.charAt(i) || p.charAt(j - 1) == '.') {
          dp[i + 1][j + 1] = dp[i][j + 1];
        }
        dp[i + 1][j + 1] |= dp[i + 1][j - 1];
      }
    }
  }
  return dp[m][n];
}
```

如果觉得代码有点长，还可以看个更简洁的写法，不过原理都一样

```
public boolean isMatch(String s, String p) {
  int m = s.length(), n = p.length();
  boolean[][] dp = new boolean[m + 1][n + 1];
  dp[0][0] = true;
  for (int i = 0; i <= m; i++)
    for (int j = 1; j <= n; j++)
      if (p.charAt(j - 1) == '*')
        dp[i][j] = dp[i][j - 2] || (i > 0 && (s.charAt(i - 1) == p.charAt(j - 2) || p.charAt(j - 2) ==
      else
        dp[i][j] = i > 0 && dp[i - 1][j - 1] && (s.charAt(i - 1) == p.charAt(j - 1) || p.charAt(j - 1)
  return dp[m][n];
}
```

递归求解

先来定义一个函数，他表示的是s的首字符和p的首字符是否匹配。

```
//比较s的首字符和p的首字符是否匹配
private boolean comp(String s, String p) {
  return s.charAt(0) == p.charAt(0) || p.charAt(0) == '.';
}
```

如果要判断字符串s和p是否匹配，我们来看一下递归函数的大致框架

```
public boolean isMatch(String s, String p) {
    if (p.length() == 0) {
        return s.length() == 0;
    }
    if (p.length() > 1 && p.charAt(1) == '*') {
        // p的第二个字符是 '*'
        ……
    } else {
        // p的第二个字符不是 '*'
        ……
    }
}
```

因为字符"*"不能单独存在，他需要和他前面的字符搭配使用，成为一个组合。

1，当p的第二个字符不是"*"的时候，那么p的第一个字符就可以单独和s的第一个字符进行比较。

2，如果p的第二个字符是"*"，那么p的第二个字符和第一个字符必须成为一个组合来进行匹配，也就类收于"a*"。下面会分为两种情况

- 字符"*"匹配0次，让字符"*"和他前面的那个字符同时消失，然后判断字符串s和p.substring(2)是否匹配。
- 字符"*"匹配1次或多次，让字符串s砍掉首字符，然后继续和字符串p匹配。

搞懂了上面的过程，代码就比较简单了，来看下完整代码

```
public boolean isMatch(String s, String p) {
    if (p.length() == 0) {
        return s.length() == 0;
    }
    if (p.length() > 1 && p.charAt(1) == '*') {
        // p的第二个字符是 '*'
        //1,字符"*"把前面的字符消掉，也就是匹配0次
        //2,字符"*"匹配1次或多次
        return isMatch(s, p.substring(2)) || (s.length() > 0 && comp(s, p)) && isMatch(s.substring(1),
    } else {
        // p的第二个字符不是 '*'，判断首字符是否相同，如果相同再从第二位继续比较
        return s.length() > 0 && comp(s, p) && (isMatch(s.substring(1), p.substring(1)));
    }
}

//比较s的首字符和p的首字符是否匹配
private boolean comp(String s, String p) {
    return s.charAt(0) == p.charAt(0) || p.charAt(0) == '.';
}
```

总结

这题被标注为hard，确实是有一定的难度，使用动态规划应该是最容易理解的，关键难点是上面递推公式的推导

423，动态规划和递归解最小路径和

原创 山大王wld 数据结构和算法 8月6日

收录于话题
#算法图文分析

95个 >



When the world turns its back on you, you turn your back on the world! And only embrace what's next!

若世界与你背道而驰，你无需亦步亦趋，欣然走好接下来的每一步吧！

问题描述

给定一个包含非负整数的 m x n 网格，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明： 每次只能向下或者向右移动一步。

示例:

> **输入**:
> [
> [**1**,**3**,**1**],
> [1,5,**1**],
> [4,2,**1**]
>]
>
> **输出**: 7

> **解释**: 因为路径 1→3→1→1→1 的总和最小。

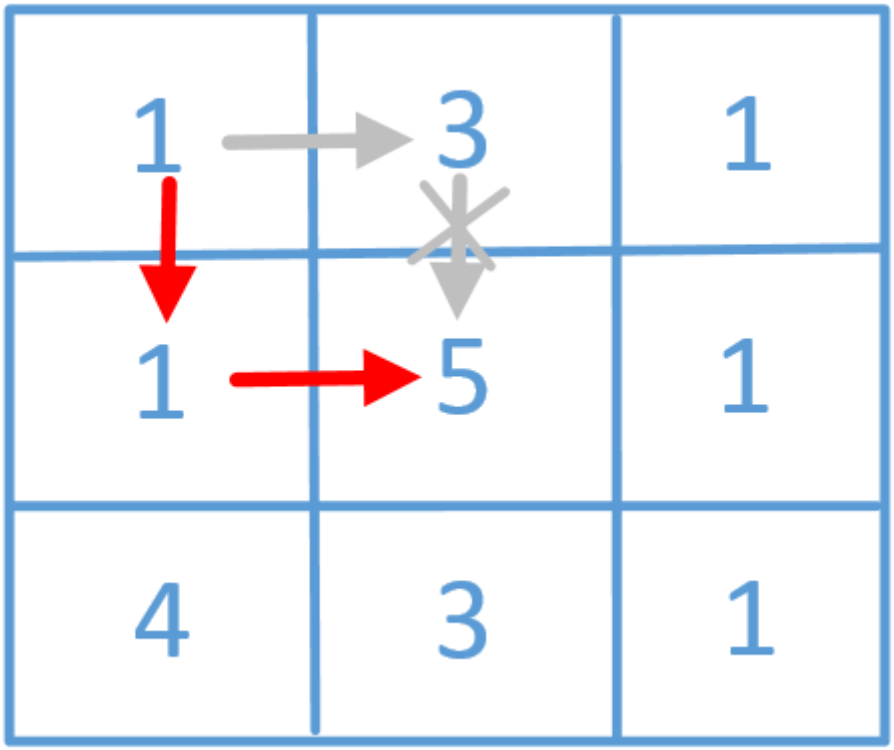
动态规划求解

这题求的是从左上角到右下角，路径上的数字和最小，并且每次只能向下或向右移动。所以上面很容易想到动态规划求解。我们可以使用一个二维数组dp，dp[i][j]表示的是从左上角到坐标(i，j)的最小路径和。那么走到坐标(i，j)的位置只有这两种可能，要么从上面(i-1，j)走下来，要么从左边（i，j-1）走过来，我们要选择路径和最小的再加上当前坐标的值就是到坐标（i，j）的最小路径。

所以递推公式就是

$$dp[i][j]=min(dp[i-1][j]+dp[i][j-1])+grid[i][j];$$

有了递推公式再来看一下边界条件，当在第一行的时候，因为不能从上面走下来，所以当前值就是前面的累加。同理第一列也一样，因为他不能从左边走过来，所以当前值只能是上面的累加。



比如上面图中，如果我们走到中间这一步的话，我们可以从上面1→3→5走过来，也可以从左边1→1→5，我们取最小的即可。我们来看下代码

```
1   public int minPathSum(int[][] grid) {
2       int m = grid.length, n = grid[0].length;
3       int[][] dp = new int[m][n];
4       dp[0][0] = grid[0][0];
```

```
    //第一列只能从上面走下来
    for (int i = 1; i < m; i++) {
      dp[i][0] = dp[i - 1][0] + grid[i][0];
    }
    //第一行只能从左边走过来
    for (int i = 1; i < n; i++) {
      dp[0][i] = dp[0][i - 1] + grid[0][i];
    }
    for (int i = 1; i < m; i++) {
      for (int j = 1; j < n; j++) {
        //递推公式，取从上面走下来和从左边走过来的最小值+当前坐标的值
        dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - 1]) + grid[i][j];
      }
    }
    return dp[m - 1][n - 1];
  }

```

我们看到二维数组dp和二维数组grid的长和宽都是一样的，没必要再申请一个dp数组，完全可以使用grid，来看下代码

```
  public int minPathSum(int[][] grid) {
    int m = grid.length, n = grid[0].length;
    for (int i = 0; i < m; i++) {
      for (int j = 0; j < n; j++) {
        if (i == 0 && j == 0)
          continue;
        if (i == 0) {
          //第一行只能从左边走过来
          grid[i][j] += grid[i][j - 1];
        } else if (j == 0) {
          //第一列只能从上面走下来
          grid[i][j] += grid[i - 1][j];
        } else {
          //递推公式，取从上面走下来和从左边走过来的最小值+当前坐标的值
          grid[i][j] += Math.min(grid[i - 1][j], grid[i][j - 1]);
        }
      }
    }
    return grid[m - 1][n - 1];
  }
```

递归求解

我们还可以把上面的动态规划改为递归，定义一个函数minPathSum(int[][] grid, int i, int j)表示从左上角到坐标(i，j)的最短路径和，那么同样道理，要走到坐标(i，j)只能从上面下来或者左边过来。所以代码轮廓我们大致能写出来

```
  public int minPathSum(int[][] grid, int i, int j) {
    if (边界条件的判断) {
      return
    }

    //一些逻辑处理

    //取从上面走下来和从左边走过来的最小值+当前坐标的值
    return grid[i][j] + Math.min(minPathSum(grid, i - 1, j), minPathSum(grid, i, j - 1));
  }
```

下面再来看下完整代码

```
  public int minPathSum(int[][] grid) {
```

```
    return minPathSum(grid, grid.length - 1, grid[0].length - 1);
}

public int minPathSum(int[][] grid, int i, int j) {
    if (i == 0 && j == 0)
        return grid[i][j];
    //第一行只能从左边走过来
    if (i == 0)
        return grid[i][j] + minPathSum(grid, i, j - 1);
    //第一列只能从上面走下来
    if (j == 0)
        return grid[i][j] + minPathSum(grid, i - 1, j);
    //取从上面走下来和从左边走过来的最小值+当前坐标的值
    return grid[i][j] + Math.min(minPathSum(grid, i - 1, j), minPathSum(grid, i, j - 1));
}
```

因为这里面的递归会导致大量的重复计算，所以还是老方法，就是把计算过的值存储到一个map中，下次计算的时候先看map中是否有，如果有就直接从map中取，如果没有再计算，计算之后再把结果放到map中，来看下代码

```
public int minPathSum(int[][] grid) {
    return minPathSum(grid, grid.length - 1, grid[0].length - 1, new HashMap<String, Integer>());
}

public int minPathSum(int[][] grid, int i, int j, Map<String, Integer> map) {
    if (i == 0 && j == 0)
        return grid[i][j];
    String key = i + "*" + j;
    if (map.containsKey(key))
        return map.get(key);
    int res = 0;
    //第一行只能从左边走过来
    if (i == 0)
        res = grid[i][j] + minPathSum(grid, i, j - 1, map);
        //第一列只能从上面走下来
    else if (j == 0)
        res = grid[i][j] + minPathSum(grid, i - 1, j, map);
        //取从上面走下来和从左边走过来的最小值+当前坐标的值
    else
        res = grid[i][j] + Math.min(minPathSum(grid, i - 1, j, map), minPathSum(grid, i, j - 1, map));
    map.put(key, res);
    return res;
}
```

总结

这题使用动态规划应该说是最容易理解的，也可以参照前面的409，动态规划求不同路径和411，动态规划和递归求不同路径 II，只不过递推公式会有点差别。

往期推荐

- 417，BFS和DFS两种方式求岛屿的最大面积
- 413，动态规划求最长上升子序列
- 411，动态规划和递归求不同路径 II

413，动态规划求最长上升子序列

原创 山大王wld 数据结构和算法 7月29日

收录于话题
#算法图文分析

95个 >



Hope is like the sun, as we journey toward it, casts the shadow of our burden behind us.

希望有如太阳，当我们向它行进时，便把我们负担的阴影投在身后。

问题描述

给定一个无序的整数数组，找到其中最长上升子序列的长度。

示例:

> **输入**: [10,9,2,5,3,7,101,18]
> **输出**: 4
> **解释**: 最长的上升子序列是 [2,3,7,101]，它的长度是 4。

动态规划

我们用dp[i]表示数组的前i个元素构成的最长上升子序列，如果要求dp[i]，我们需要用num[i]和前面的数字一个个比较，如果比前面的任何一个数字大，说明加入到他的后面可以构成一个上升子序列 ，就更新dp[i]。我们就以[8，2，3，1，4]为例来画个图看一下





1，4比8小，不更新dp[4],dp[4]=1。
2，4比2大，所以4可以和2构成一个上升的子序列，dp[4]=dp[1]+1=2。也就是[2,4]
3，4又比3大，所以4又可以和3结尾的子序列构成一个上升的子序列，dp[4]=dp[2]+1=2+1=3,也就是[2,3,4]。
4，4还比1大，也可以和1构成上升子序列[1,4]，但长度是2，小于和3构成的上升子序列，我们要取最大的，综上dp[4]=3。

我们再来看下代码

```
1   public int lengthOfLIS(int[] nums) {
2     //边界条件判断
3     if (nums == null || nums.length == 0) {
4       return 0;
5     }
6     int[] dp = new int[nums.length];
7     //初始化数组dp的每个值为1
8     Arrays.fill(dp, 1);
9     int max = 1;
10    for (int i = 1; i < nums.length; i++) {
11      for (int j = 0; j < i; j++) {
12        //如果当前值nums[i]大于nums[j]，说明nums[i]可以和
13        //nums[j]结尾的上升序列构成一个新的上升子序列
14        if (nums[i] > nums[j]) {
15          dp[i] = Math.max(dp[i], dp[j] + 1);
16          //记录构成的最大值
17          max = Math.max(max, dp[i]);
18        }
19      }
20    }
21    return max;
22  }
```

上升子序列加入到集合中

这题还有一种解决方式就是把找到的上升子序列放到一个集合list中，我们每次遍历的时候都会拿当前值nums[i]和list的最后一个元素比较

1，如果nums[i]比list最后一个元素大，说明nums[i]加入到list的末尾可以构成一个更长的上升子序列，我们就把nums[i]加入到list的末尾。

2，如果nums[i]不大于list的最后一个元素，说明nums[i]和list不能构成一个更长的上升子序列，但我们可以用nums[i]把list中第一个大于他的给替换掉。我们要保证list

中元素不变的情况下，值越小越好，这样当我们加入一个新值的时候，构成上升子序列的可能性就越大。

再来看下代码

```
public int lengthOfLIS(int[] nums) {
  //list中保存的是构成的上升子序列
  ArrayList<Integer> list = new ArrayList<>(nums.length);
  for (int num : nums) {
    //如果list为空，我们直接把num加进去。如果list的最后一个元素小于num，
    //说明num加入到list的末尾可以构成一个更长的上升子序列，我们就把num
    //加入到list的末尾
    if (list.size() == 0 || list.get(list.size() - 1) < num)
      list.add(num);
    else {
      //如果num不小于list的最后一个元素，我们就用num把list中第一
      //个大于他的值给替换掉,这样我们才能保证list中的元素在长度不变
      //的情况下，元素值尽可能的小
      int i = Collections.binarySearch(list, num);
      //因为list是从小到大排序的，所以上面使用的是二分法查找。当i大
      //于0的时候，说明出现了重复的，我们直接把他替换即可，如果i小于
      //0，我们对i取反，他就是list中第一个大于num值的位置，我们把它
      //替换即可
      list.set((i < 0) ? -i - 1 : i, num);
    }
  }
  return list.size();
}
```

总结

这题也是比较常见的一道题，动态规划应该说是最好理解的。如果完全搞懂的话，下面的那种解法其实也是比较经典的。

往期推荐

- 411，动态规划和递归求不同路径 II
- 409，动态规划求不同路径
- 407，动态规划和滑动窗口解决最长重复子数组
- 395，动态规划解通配符匹配问题

411，动态规划和递归求不同路径 II

原创 山大王wld 数据结构和算法 7月24日

收录于话题
#算法图文分析

95个 >



If you're not satisfied with the life you're living, don't just complain. Do something about it.

对于现况的不满，你不能只是抱怨，而是要有勇气作出改变。

问题描述

一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为“Start”）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

现在考虑网格中有障碍物。那么从左上角到右下角将会有多少条不同的路径？



网格中的障碍物和空位置分别用 1 和 0 来表示。

说明： m 和 n 的值均不超过 100。

示例 1:

> 输入:
> [
> [0,0,0],
> [0,1,0],
> [0,0,0]
>]
> 输出: 2
> 解释:
> 3x3 网格的正中间有一个障碍物。
> 从左上角到右下角一共有 2 条不同的路径:
> 1. 向右 -> 向右 -> 向下 -> 向下
> 2. 向下 -> 向下 -> 向右 -> 向右

动态规划解决

上一篇我们讲过和这非常类似的一道题，只不过上一篇没有障碍物，但并不影响我们解这道题，我们还用dp[i][j]表示到坐标(i，j)这个格内有多少条不同的路径，所以最终的答案就是求dp[m-1][n-1]。

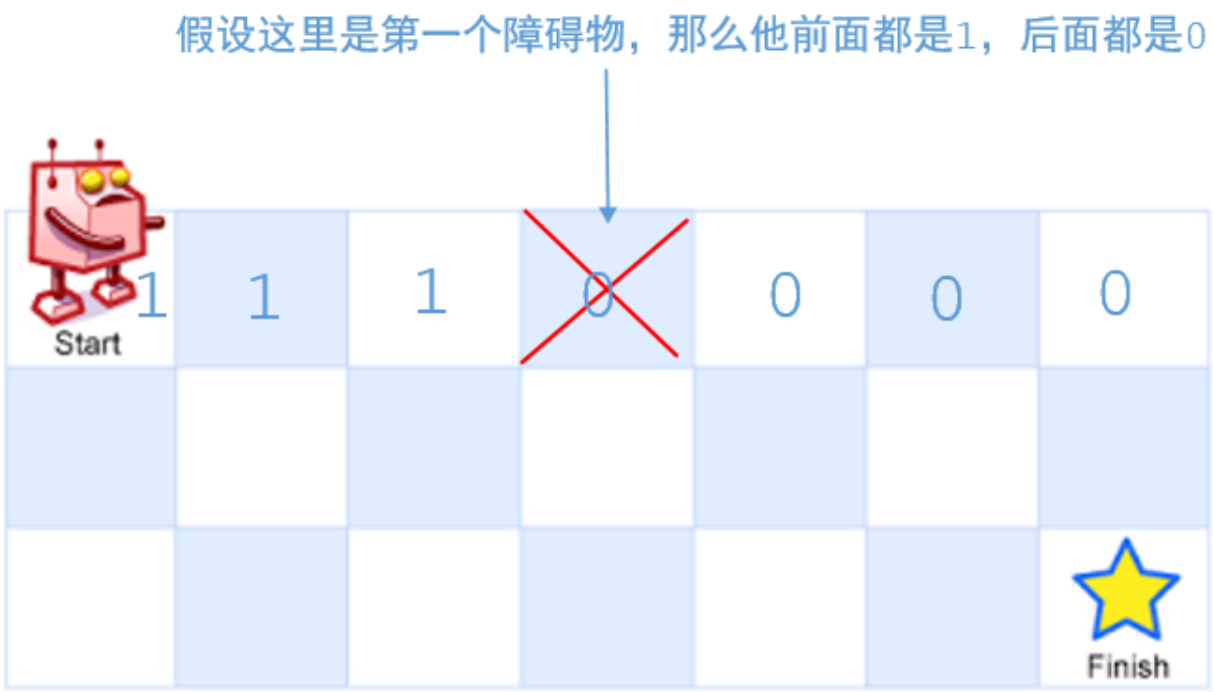
这里的递推分为两种情况，一种是当前网格没有障碍物，一种是当前网格有障碍物。

1，如果当前网格dp[i][j]有障碍物，那么这里肯定是走不过去的，所以dp[i][j]=0。

2，如果当前网格dp[i][j]没有障碍物，那么递推公式就和上一题409，动态规划求不同路径一样了。
因为只能从上面或左边走过来，所以递推公式是
dp[i][j]=dp[i-1][j]+dp[i][j-1]。

- dp[i-1][j]表示的是从上面走过来的路径条数。
- dp[i][j-1]表示的是从左边走过来的路径条数。

边界条件也好判断，如果当前行没有障碍物，那么当前行的值都是1，如果有障碍物，那么第一个障碍物前面都是1，其他的都是0。同理第一列也一样。



我们来看下代码

```
1   public int uniquePathsWithObstacles(int[][] obstacleGrid) {
2     int m = obstacleGrid.length;
3     int n = obstacleGrid[0].length;
4     int dp[][] = new int[m][n];
5     //第一列初始化
6     for (int i = 0; i < m; i++) {
7       if (obstacleGrid[i][0] == 0)
8         dp[i][0] = 1;
9       else
10        break;
11    }
12    //第一行初始化
13    for (int i = 0; i < n; i++) {
14      if (obstacleGrid[0][i] == 0)
15        dp[0][i] = 1;
16      else
17        break;
18    }
19    for (int i = 1; i < m; ++i)
20      for (int j = 1; j < n; ++j)
21        if (obstacleGrid[i][j] == 0)
22          dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
23    return dp[m - 1][n - 1];
24  }
```

代码和409，动态规划求不同路径差不多，只不过在第一行和第一列还有上面第21行多了一些判断。

动态规划代码量优化

上面代码虽然也能实现，但有那么多条件判断总感觉很繁琐，所以我们还有一种方式就是把二维数组的长和宽都放大一格，这样数组的第一行和第一列都不存储任何值，但初始条件要变了

- dp[1][1] = obstacleGrid[0][0] ^ 1;
- dp[0][1] = 1;
- dp[1][0] = 1;

上面3种初始条件都可以，我们来任选一个，看下代码

```
1   public int uniquePathsWithObstacles(int[][] obstacleGrid) {
2     int m = obstacleGrid.length;
3     int n = obstacleGrid[0].length;
4     int dp[][] = new int[m + 1][n + 1];
5     //初始条件，下面3个任选一个
6     //dp[1][1] = obstacleGrid[0][0] ^ 1;
7     //dp[0][1] = 1;
8     dp[1][0] = 1;
9     for (int i = 1; i <= m; ++i)
10      for (int j = 1; j <= n; ++j)
11        if (obstacleGrid[i - 1][j - 1] == 0)
12          dp[i][j] += dp[i - 1][j] + dp[i][j - 1];
13    return dp[m][n];
14  }
15
```

动态规划空间优化

我们可以参照上一题把二维空间改为一维的，原理很简单，我们来直接看代码

```
1   public int uniquePathsWithObstacles(int[][] obstacleGrid) {
2     int m = obstacleGrid.length;
3     int n = obstacleGrid[0].length;
4     int[] dp = new int[n + 1];
5     dp[1] = 1;
6     for (int i = 0; i < m; i++) {
7       for (int j = 1; j <= n; j++) {
8         if (obstacleGrid[i][j - 1] == 1) {
9           dp[j] = 0;//有障碍物
10        } else {//无障碍物
11          dp[j] += dp[j - 1];
12        }
13      }
14    }
15    return dp[n];
16  }
```

上一题有人问过一个问题说看不懂第11行，这里再说一下，因为是一行一行的遍历，在当前行遍历之前dp（这里是一维数组）表示的是上一行的值，然后遍历到当前行的时候，假如遍历当前行的第j列的时候，那么当前行第j列之前的数据都会被更新掉，当前行第j列之后的数据还是上一行的，所以dp[j]=dp[j]+dp[j-1]（为了区分，这里标成了不同的颜色），dp[j]表示的是当前列的上一行值，dp[j-1]表示的是当前行的前一个值。

递归方式

上一题我们提到过，使用递归的方式会造成大量的重复计算，所以为了减少重复计算，这里使用一个map把计算过的值存储起来，下次用的时候先从map中取，如果有就返回，如果没有再计算。

```
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
    return helper(obstacleGrid, 0, 0, new HashMap<String, Integer>());
}

public static int helper(int[][] obstacleGrid, int down, int right, Map<String, Integer> map) {
    String key = down + "and" + right;
    int result = 0;
    if (map.containsKey(key))
        return map.get(key);
    if (obstacleGrid[down][right] == 1) {
        result = 0;
        map.put(key, result);
        return result;
    }
    if (right == obstacleGrid[0].length - 1 && down == obstacleGrid.length - 1) {
        if (obstacleGrid[down][right] == 1) {
            result = 0;
        } else {
            result = 1;
        }
        map.put(key, result);
        return result;
    }
    if (right == obstacleGrid[0].length - 1 || down == obstacleGrid.length - 1) {
        if (right == obstacleGrid[0].length - 1) {
            result = helper(obstacleGrid, down + 1, right, map);
        } else {
            result = helper(obstacleGrid, down, right + 1, map);
        }
        map.put(key, result);
        return result;
    }
    result = helper(obstacleGrid, down, right + 1, map) + helper(obstacleGrid, down + 1, right, map);
    map.put(key, result);
    return result;
}
```

这种不看也可以，因为动态规划非常简单，没人会傻到会使用这种方式，但他也算是提供了一种思路，有时间看看也行。

总结

这题多了一个障碍物的判断，但难度其实并没有增加多少，如果当前位置出现了障碍物，说明不能从当前位置通过，所以当前位置的路径是0，如果当前位置不是0，那么计算就还和以前一样了。

往期推荐

- 409，动态规划求不同路径
- 407，动态规划和滑动窗口解决最长重复子数组

409，动态规划求不同路径

原创 山大王wld 数据结构和算法 7月22日

收录于话题
#算法图文分析

95个 >



Keep on going never give up!

勇往直前，决不放弃

问题描述

一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为“Start”）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

问总共有多少条不同的路径？



例如，上图是一个7 x 3 的网格。有多少可能的路径？

示例 1:

> 输入: m = 3, n = 2
> 输出: 3
> 解释:
> 从左上角开始，总共有 3 条路径可以到达右下角。
> 1. 向右 -> 向右 -> 向下
> 2. 向右 -> 向下 -> 向右
> 3. 向下 -> 向右 -> 向右

示例 2:

> 输入: m = 7, n = 3
> 输出: 28

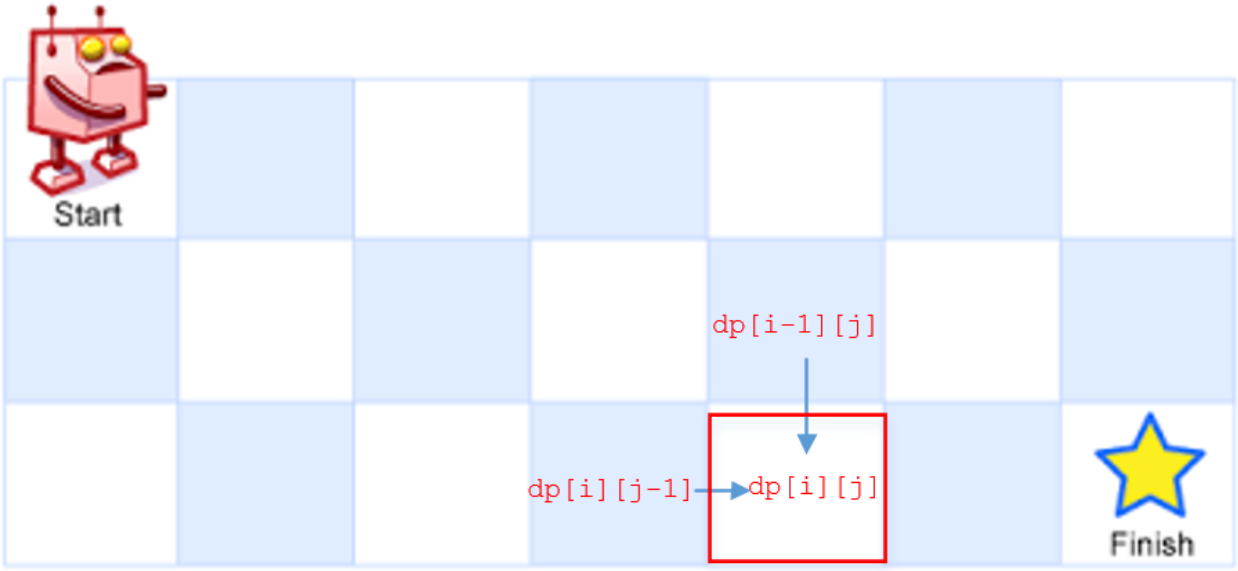
提示:

- 1 <= m, n <= 100
- 题目数据保证答案小于等于 2 * 10 ^ 9

动态规划解决

注意这里机器人只能向下和向右移动，不能往其他方向移动，我们用dp[i][j]表示到坐标(i，j)这个格内有多少条不同的路径，所以最终的答案就是求dp[m-1][n-1]。

因为只能从上面或左边走过来，所以递推公式是
dp[i][j]=dp[i-1][j]+dp[i][j-1]。
dp[i-1][j]表示的是从上面走过来的路径条数。
dp[i][j-1]表示的是从左边走过来的路径条数。



那么边界条件是什么呢，如果Finish在第一行的任何位置都只有一条路径，同理Finish在第一列的任何位置也都只有一条路径，所以边界条件是第一行和第一列都是1。我们已经找到了递推公式，又找到了边界条件，所以动态规划代码很容易写出来，我们来看下

```
public int uniquePaths(int m, int n) {
  int[][] dp = new int[m][n];
  //第一列都是1
  for (int i = 0; i < m; i++) {
    dp[i][0] = 1;
  }
  //第一行都是1
  for (int i = 0; i < n; i++) {
    dp[0][i] = 1;
  }

  //这里是递推公式
  for (int i = 1; i < m; i++)
    for (int j = 1; j < n; j++)
      dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
  return dp[m - 1][n - 1];
}

```

动态规划优化

我们看上面二维数组的递推公式，当前坐标的值只和左边与上面的值有关，和其他的无关，这样二维数组造成大量的空间浪费，所以我们可以把它改为一维数组。

```
public int uniquePaths(int m, int n) {
  int[] dp = new int[m];
  Arrays.fill(dp, 1);
  for (int j = 1; j < n; j++)
    for (int i = 1; i < m; i++)
      dp[i] += dp[i - 1];
  return dp[m - 1];
}
```

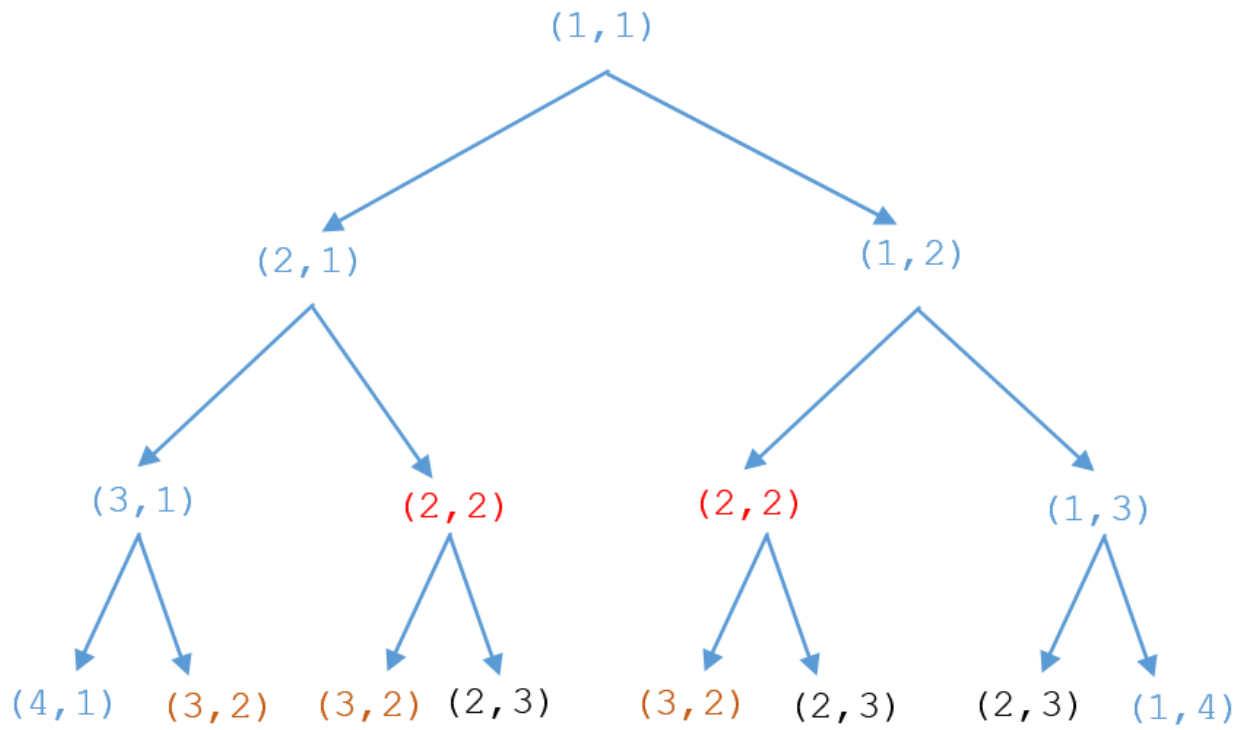
这里的dp[i]+=dp[i-1]；实际上就是dp[i]=dp[i]+dp[i-1]，我们可以这样理解，上面的网格我们是一行一行计算的，dp[i]也就是上面红色的表示的是当前位置上面的值，dp[i-1]表示的是当前位置左边的值。

递归方式

这题除了动态规划以外，还可以把上面的动态规划改为递归的方式

```
public int uniquePaths(int m, int n) {
    return uniquePathsHelper(1, 1, m, n);
}

//第i行第j列到第m行第n列共有多少种路径
public int uniquePathsHelper(int i, int j, int m, int n) {
    //边界条件的判断
    if (i > m || j > n)
        return 0;
    if ((i == m && j == n))
        return 1;
    //从右边走有多少条路径
    int right = uniquePathsHelper(i + 1, j, m, n);
    //从下边走有多少条路径
    int down = uniquePathsHelper(i, j + 1, m, n);
    //返回总的路径
    return right + down;
}
```

代码中有注释，很容易理解，但其实这种效率很差，因为他包含了大量的重复计算，我们画个图来看一下。



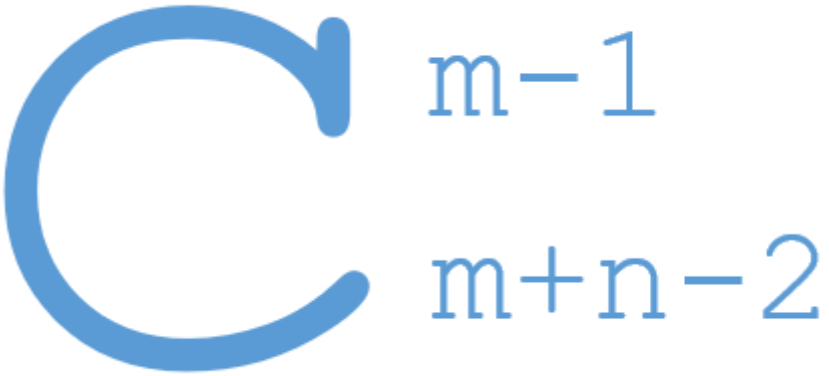
我们看到上面图中红色，黑色，还有那种什么颜色的都表示重复的计算，所以有一种方式就是把计算过的值使用一个map存储起来，用的时候先查看是否计算过，如果计算过就直接拿来用，看下代码

```
public int uniquePaths(int m, int n) {
    return uniquePathsHelper(1, 1, m, n, new HashMap<>());
}

public int uniquePathsHelper(int i, int j, int m, int n, Map<String, Integer> map) {
    if (i > m || j > n)
        return 0;
    if ((i == m && j == n))
        return 1;
    String key = i + "*" + j;
    if (map.containsKey(key))
```

```
        return map.get(key);
    int right = uniquePathsHelper(i + 1, j, m, n, map);
    int down = uniquePathsHelper(i, j + 1, m, n, map);
    int totla = right + down;
    map.put(key, totla);
    return totla;
}
```

使用公式计算

我们要想到达终点，需要往下走n-1步，往右走m-1步，总共需要走n+m-2步。他无论往右走还是往下走他的总的步数是不会变的。也就相当于总共要走n+m-2步，往右走m-1步总共有多少种走法，很明显这就是一个排列组合问题，公式如下



排列组合的计算公式如下

$$C_n^m = \frac{A_n^m}{m!} = \frac{n!}{m!\,(n-m)!}$$

公式为(m+n-2)! / [(m-1)! * (n-1)!]

代码如下

```
public int uniquePaths(int m, int n) {
    int N = n + m - 2;
    double res = 1;
    for (int i = 1; i < m; i++)
        res = res * (N - (m - 1) + i) / i;
    return (int) res;
}
```

总结

这题使用动态规划是最容易理解也是最容易解决的，当然后面的递归和公式计算也能解决。

往期推荐

- 407，动态规划和滑动窗口解决最长重复子数组
- 395，动态规划解通配符匹配问题
- 382，每日温度的5种解题思路
- 380，缺失的第一个正数（中）

407，动态规划和滑动窗口解决最长重复子数组

原创 山大王wld 数据结构和算法 7月21日

收录于话题
#算法图文分析

95个 >



Never be afraid to reach for the stars, because even if you fall, you'll always be wearing a parent-chute.

永远不要害怕去摘星星，因为就算你跌下来，你永远有“父母牌”降落伞防身。

问题描述

给两个整数数组 A 和 B ，返回两个数组中公共的、长度最长的子数组的长度。

示例：

输入：
A: [1,2,3,2,1]
B: [3,2,1,4,7]
输出： 3
解释：
长度最长的公共子数组是 [3，2，1] 。

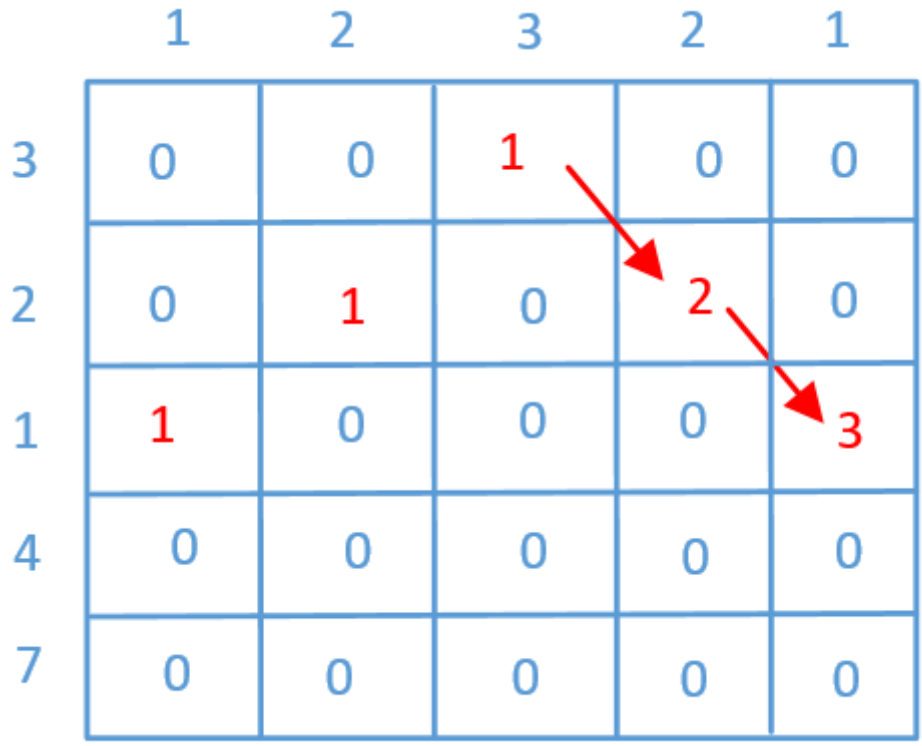
提示：

1 <= len(A), len(B) <= 1000

0 <= A[i], B[i] < 100

动态规划

这题一看就知道其实就是求最长公共子串问题，不懂的可以看下前面的370，最长公共子串和子序列，只不过换了种说法，换汤不换药，本质还是没变。我们就以题中的示例画个图来看一下



最长的公共子数组就是上面红色所对应的[3,2,1]，长度是3。

递推公式是

```
if(s1.charAt(i) == s2.charAr(j))
    dp[i][j] = dp[i-1][j-1] + 1;
else
    dp[i][j] = 0;
```

有了递推公式，代码就容易多了，我们来看下完整代码

```
public int findLength(int[] A, int[] B) {
  int max = 0;
  int[][] dp = new int[A.length + 1][B.length + 1];
  for (int i = 1; i <= A.length; i++) {
    for (int j = 1; j <= B.length; j++) {
      if (A[i - 1] == B[j - 1]) {
        dp[i][j] = dp[i - 1][j - 1] + 1;
        max = Math.max(max, dp[i][j]);
      }
    }
  }
  return max;
```

```
}

```

这里的二维数组dp长和宽都要加1是为了减少判断，当然也可以不加1，但这样会多了一些边界的判断，我们来看下不加1的代码

```
public int findLength(int[] A, int[] B) {
    int max = 0;
    int[][] dp = new int[A.length][B.length];
    for (int i = 0; i < A.length; i++) {
        for (int j = 0; j < B.length; j++) {
            if (A[i] == B[j]) {
                if (i > 0 && j > 0)
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                else
                    dp[i][j] = 1;
                max = Math.max(max, dp[i][j]);
            }
        }
    }
    return max;
}
```

如果看过之前写的370，最长公共子串和子序列，我们还可以把二维数组改为一维数组来减少空间复杂度

```
public int findLength(int[] A, int[] B) {
    int max = 0;
    int[] dp = new int[B.length + 1];
    for (int i = 1; i <= A.length; i++) {
        for (int j = B.length; j >= 1; j--) {
            if (A[i - 1] == B[j - 1])
                dp[j] = dp[j - 1] + 1;
            else
                dp[j] = 0;
            max = Math.max(max, dp[j]);
        }
    }
    return max;
}
```

注意这里第二个for循环是从后往前遍历的，这是因为dp后面的值会依赖前面的值，但前面的值不会依赖后面的值，如果我们改变后面的值对前面的值不会有影响，但改变前面的值会影响面的值，所以这里我们从后往前计算是最合适的。

滑动窗口

第2种方式是滑动窗口，文字叙述不好理解，我们就以[1，2，3，2，1]和[3,2,1,4]为例来举例说明，这两个数组我故意弄成两个长度不一样的，我们画个图来看一下

第一步

3, 2, 1, [1, 4] 2, 3, 2, 1

第二步

3, 2, [1, 2, 1, 4] 3, 2, 1

第三步

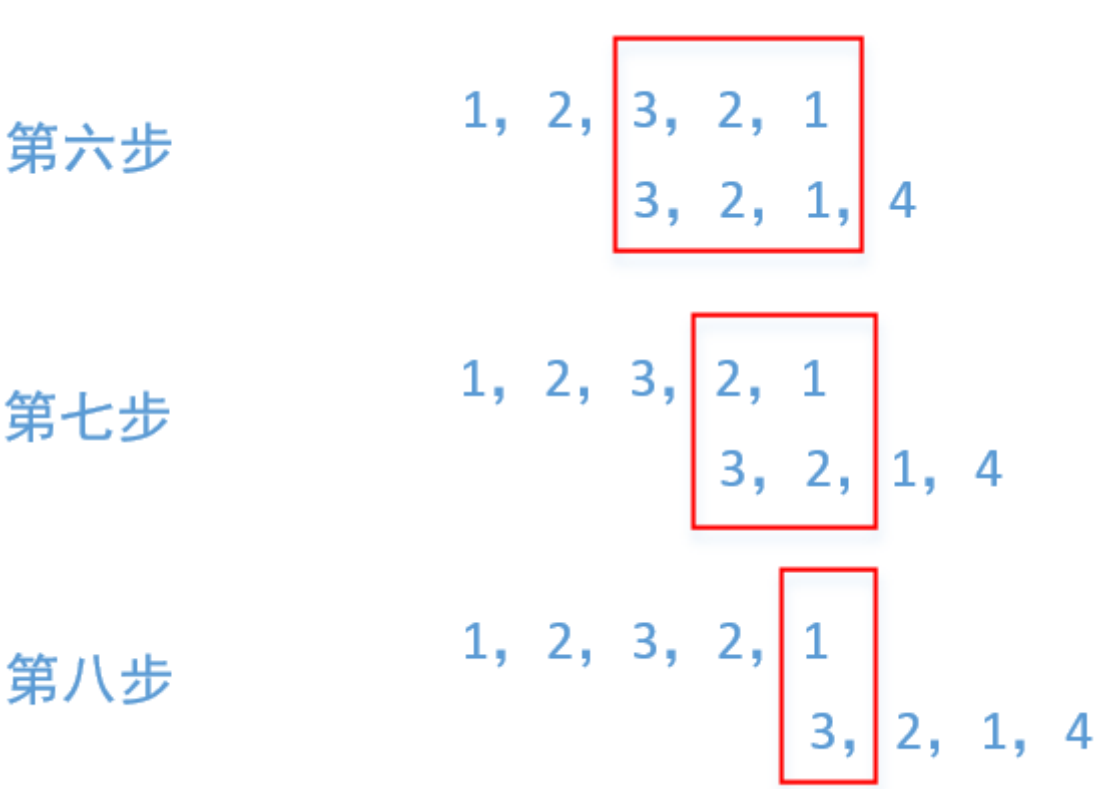
3, [1, 2, 3, 2, 1, 4] 2, 1

第四步

[1, 2, 3, 2, 3, 2, 1, 4] 1

第五步

1, [2, 3, 2, 1, 3, 2, 1, 4]



相当于说第一个数组位置不动，第二个数组每次往右移一位，搞懂了上的分析过程，代码就容易多了，我们来看下

```
public int findLength(int[] A, int[] B) {
  if (A.length < B.length)
    return findLengthHelper(A, B);
  return findLengthHelper(B, A);
}

public int findLengthHelper(int[] A, int[] B) {
  int aLength = A.length, bLength = B.length;
  //total是总共运行的次数
  int total = aLength + bLength - 1;
  int max = 0;
  for (int i = 0; i < total; i++) {
    //下面一大坨主要判断数组A和数组B比较的起始位置和比较的长度
    int aStart = 0;
    int bStart = 0;
    int len = 0;
    if (i < aLength) {
      aStart = aLength - i - 1;
      bStart = 0;
      len = i + 1;
    } else {
      aStart = 0;
      bStart = i - aLength + 1;
      len = Math.min(bLength - bStart, aLength);
    }
    int maxlen = maxLength(A, B, aStart, bStart, len);
    max = Math.max(max, maxlen);
  }
  return max;
}

//计算A和B在上面图中红色框内的最大长度
public int maxLength(int[] A, int[] B, int aStart, int bStart, int len) {
  int max = 0, count = 0;
  for (int i = 0; i < len; i++) {
    if (A[aStart + i] == B[bStart + i]) {
      count++;
      max = Math.max(max, count);
```

```
39          } else {
40              count = 0;
41          }
42      }
43      return max;
44  }
```

总结

其实这道题求的就是最长公共子串问题，通过上面的图分析，可以发现第一种方式和第二种方式都比较好理解，但第一种方式代码明显比第二种少了很多。

往期推荐

- 405，换酒问题
- 397，双指针求接雨水问题
- 394，经典的八皇后问题和N皇后问题
- 392，检查数组对是否可以被 k 整除

395，动态规划解通配符匹配问题

原创 山大王wld 数据结构和算法 7月6日

收录于话题
#算法图文分析

96个 >



If you really like something, you've got to put yourself out there...got to reach out and grab it.

如果你真的喜欢某样东西，你就得置身其中，伸手抓住它。

问题描述

给定一个字符串（s）和一个字符模式（p），实现一个支持 '?' 和 '*' 的通配符匹配。

> '?' 可以匹配任何单个字符。
> '*' 可以匹配任意字符串（包括空字符串）。

两个字符串完全匹配才算匹配成功。

说明:

- s 可能为空，且只包含从 a-z 的小写字母。
- p 可能为空，且只包含从 a-z 的小写字母，以及字符 ？和 *。

示例 1:

输入:
s = "aa"
p = "a"
输出: false
解释: "a" 无法匹配 "aa" 整个字符串。

示例 2:

输入:
s = "aa"
p = "*"
输出: true
解释: '*' 可以匹配任意字符串。

示例 3:

输入:
s = "cb"
p = "?a"
输出: false
解释: '?' 可以匹配 'c', 但第二个 'a' 无法匹配 'b'。

示例 4:

输入:
s = "adceb"
p = "*a*b"
输出: true
解释: 第一个 '*' 可以匹配空字符串, 第二个 '*' 可以匹配字符串 "dce".

示例 5:

输入:
s = "acdcb"
p = "a*c?b"
输出: false

问题分析

通配符匹配，如果p的某个位置是字母，那么他只能和s的字母匹配，如果p的某个位置是“?”字符，那么他可以匹配s的任何字母，如果p的某个位置是“*”字符，那么他可以匹配s的任意多个字母。

1，状态定义

dp[i][j]表示s的前i个字符和p的前j个字符是否匹配。我们最后只需要返回dp[s.length][p.length]即可

2，状态转移

这里分两种情况，一种是p的第j个字符不是*，一个是p的第j个字符是*。

- p的第j个字符不是*

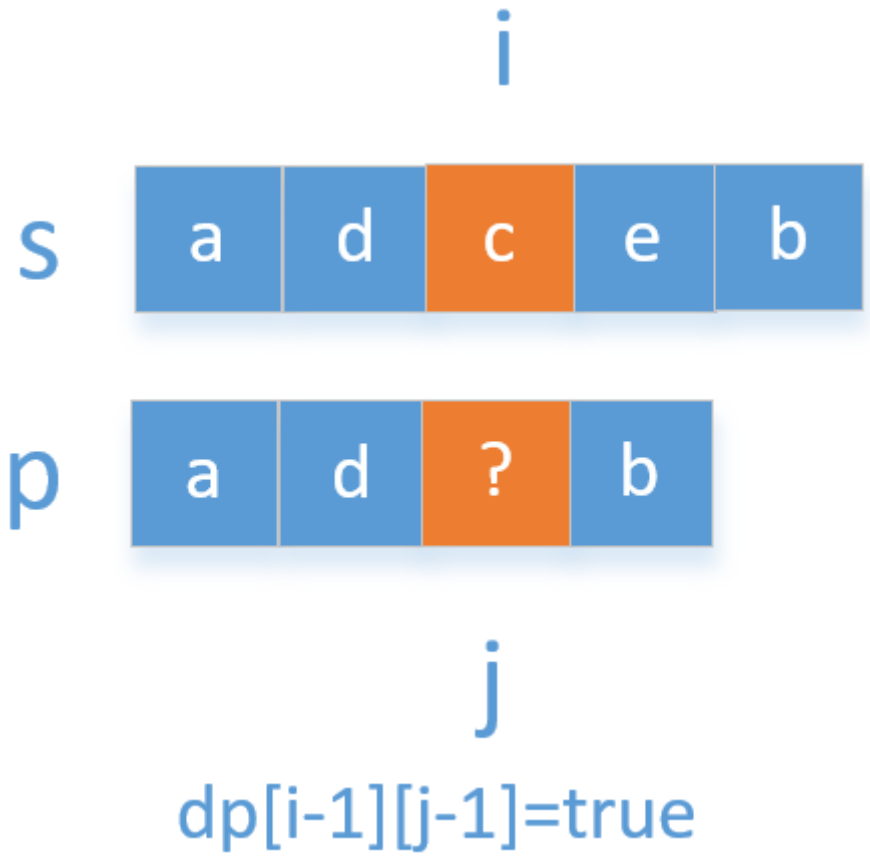
```
if (s.charAt(i - 1) == p.charAt(j - 1) || p.charAt(j - 1)== '?')
        dp[i][j] = dp[i - 1][j - 1];
```

- p的第j个字符是*

```
dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
```

第一种

第一种情况比较容易理解，我们可以根据下面的图来看下



比如上面的s和p的第3个字符匹配成功（要么这两个字符相等，要么p的第3个字符是“?”），我们要看他们前一个字符是否也匹配成功，所以dp[i][j]=dp[i-1][j-1]。

第二种

第二种情况就是p的第j个字符是*，这个*可以匹配0个，也可以匹配多个。

1，如果要匹配0个，也就是判断p的前j-1个字符和s的前i个字符是否匹配，因为匹配0个的话，也就相当于p的第j个字符是空的，所以

dp[i][j]=dp[i][j-1]

2，如果要匹配1个，只需要判断s的前i-1个和p的前j-1个是否匹配，也可以理解为p的第j个和s的第i个同时消失，只需要前面的匹配即可。

dp[i][j]=dp[i-1][j-1]

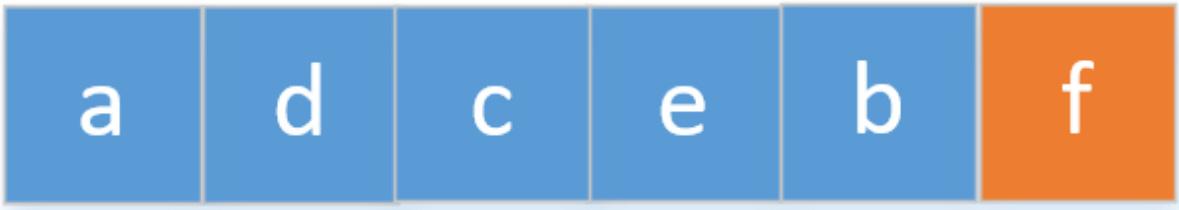
3，如果要匹配2个的话，只需要判断s的前i-2个和p的前j-1个是否匹配，也可以理解为p的第j个和s的第i和第i-1个同时消失（因为p的*匹配两个，相当于把这两个抵消了，我们只需要判断前面的），只需要前面的匹配即可。

dp[i][j]=dp[i-2][j-1]

4，如果要匹配n个的话，只需要判断s的前i-n个和p的前j-1个是否匹配即可

dp[i][j]=dp[i-n][j-1]

那要这样写下去估计永远写不完了，我们再仔细观察一下。假如p的*要匹配s的n个字符，也就是下面这样

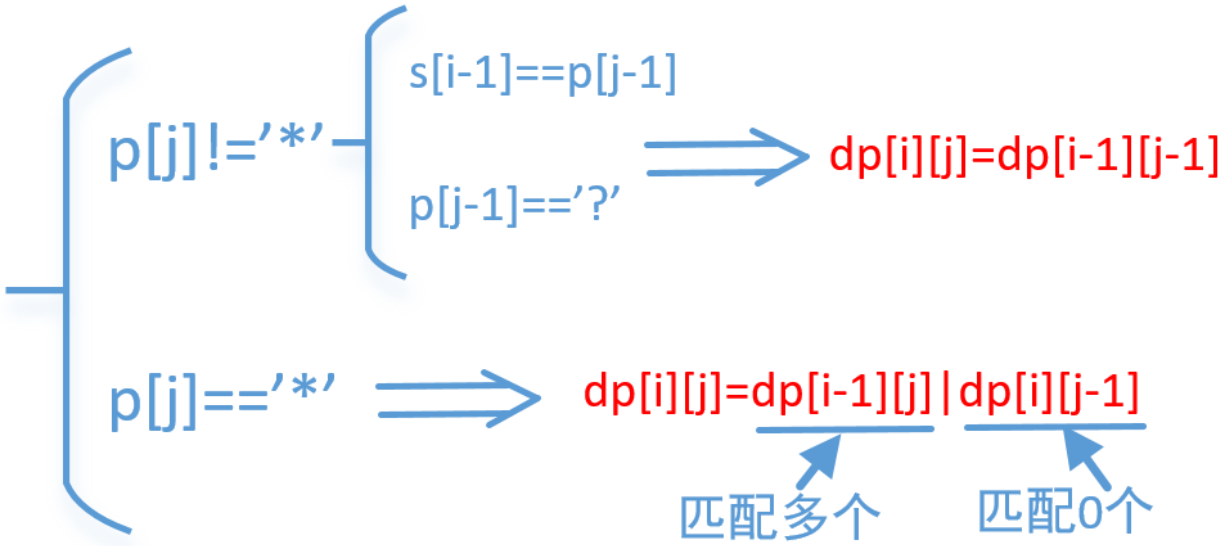




如果要判断p的字符*和s的字符f匹配的话，我们只需要判断p的字符*和s的字符b是否匹配即可。如果要判断这一步我们只需要p的字符*和s的字符e是否匹配即可……，所以我们可以把上面的4个公式简写成一个

dp[i][j]=dp[i-1][j]

综上我们把这题的递推公式找出来了，就是



3，边界条件

边界条件很容易发现，如果s和p都为空，我们默认是可以匹配的。否则p最前面有*也是可以匹配的，因为*可以匹配空串。所以边界条件是

```
boolean[][] dp = new boolean[slen + 1][plen + 1];
dp[0][0] = true;
for (int j = 1; j <= plen && dp[0][j - 1]; j++)
    dp[0][j] = p.charAt(j - 1) == '*';
```

最终代码

通过上面的分析我们来看下最终的完整代码

```
public static boolean isMatch(String s, String p) {
  //如果s不为空，p为空，是匹配不成功的，直接返回false
  if (s.length() != 0 && p.length() == 0)
    return false;

  int slen = s.length(), plen = p.length();
  boolean[][] dp = new boolean[slen + 1][plen + 1];
  dp[0][0] = true;
  //边界条件的初始化
  for (int j = 1; j <= plen && dp[0][j - 1]; j++)
    dp[0][j] = p.charAt(j - 1) == '*';

  for (int i = 1; i <= slen; i++) {
    for (int j = 1; j <= plen; j++) {
      char si = s.charAt(i - 1), pj = p.charAt(j - 1);
      //下面是动态规划的状态转移方程
      if (si == pj || pj == '?') {
        dp[i][j] = dp[i - 1][j - 1];
      } else if (pj == '*') {
        dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
      }
    }
  }
  return dp[slen][plen];
```

```
25    }
26
```

代码优化

如果看过之前讲的370，最长公共子串和子序列，我们发现上面的代码和第370题的最长公共子序列的代码有很类似的地方，所以我们也可以参照第370题的代码来优化一下，把上面的二维数组改为一维的，来看下代码

```
1   public static boolean isMatch(String s, String p) {
2     //如果s不为空，p为空，是匹配不成功的，直接返回false
3     if (s.length() != 0 && p.length() == 0)
4       return false;
5     int slen = s.length(), plen = p.length();
6     boolean[] dp = new boolean[plen + 1];
7     dp[0] = true;
8     //边界条件的初始化
9     for (int j = 1; j <= plen && dp[j - 1]; j++)
10      dp[j] = p.charAt(j - 1) == '*';
11
12    for (int i = 1; i <= slen; i++) {
13      //这里的last我们可以认为是上面代码没优化之前的
14      //dp[i - 1][j - 1]
15      boolean last = false;
16      if (i == 1)
17        last = true;
18      for (int j = 1; j <= plen; j++) {
19        //dp[j]使用之后值会被覆盖，所以我们这里在
20        //使用前把它先保留下来
21        boolean temp = dp[j];
22        char si = s.charAt(i - 1), pj = p.charAt(j - 1);
23        //下面是动态规划的状态转移方程
24        if (si == pj || pj == '?') {
25          dp[j] = last;
26        } else if (pj == '*') {
27          dp[j] = dp[j] || dp[j - 1];
28        } else {
29          dp[j] = false;
30        }
31        last = temp;
32      }
33    }
34    return dp[plen];
35  }
```

总结

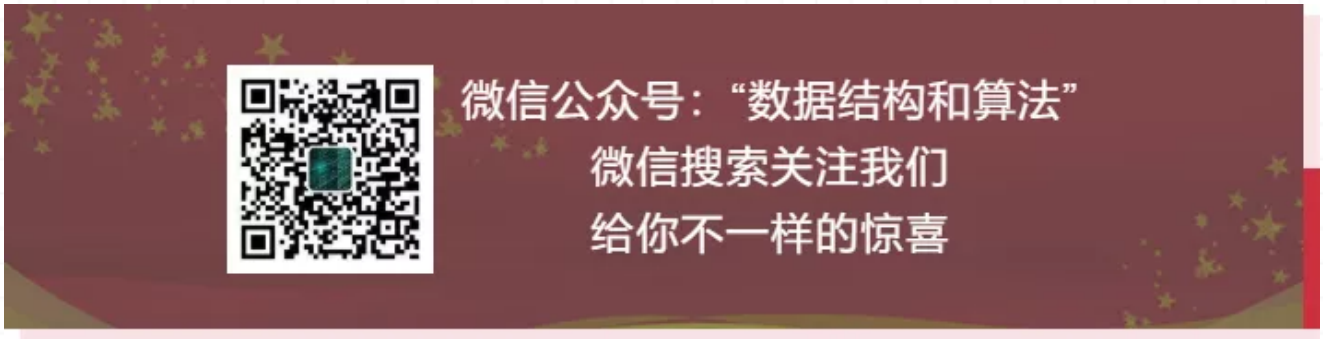
动态规划基本上都这同一套路，先定义状态表达式，在找转移方程，最后是边界条件。这题可能有一点难度的是当p的第j个字符是*的时候状态方程该怎么写。其实也很好理解，因为*可以匹配0个或多个字符。当匹配0个的时候，也就是相当于判断p的前j-1个字符和s的前i个字符是否匹配。当匹配多个的时候，*可以把s中的多个字符同时消掉，我们先移除掉s中的一个看是否匹配，如果不匹配我们再移除2个……，所以这样很容易找出状态转移方程。

376，动态规划之编辑距离

原创 山大王wld 数据结构和算法 6月4日

收录于话题
#算法图文分析

96个 >



Everything goes back to the way it was.

一切终将恢复如初。

问题描述

给你两个单词 word1 和 word2，请你计算出将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

插入一个字符
删除一个字符
替换一个字符

示例 1：

```
输入：word1 = "horse", word2 = "ros"
输出：3

解释：
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')

总有共3步
```

示例 2：

```
输入：word1 = "intention", word2 = "execution"
输出：5

解释：
intention -> inention (删除 't')
inention -> enention (将 'i' 替换为 'e')
enention -> exention (将 'n' 替换为 'x')
exention -> exection (将 'n' 替换为 'c')
exection -> execution (插入 'u')

总有共5步
```

问题分析：

1，如果想把word1变为word2，对于word1的操作我们有3种方式：

- 删除一个字符
- 添加一个字符
- 修改一个字符

这就好比对数据库的增删改查一样，不过这里没有查找。

我们用dp[i][j]表示把word1的前i个字符变为word2的前j个字符所需要的最少编辑距离，这里要分两种情况

1，当word1[i]==word2[j]：也就是说word1的第i个字符和word2的第j个字符相等，我们不需要修改word1的第i个字符，所以这时dp[i][j]=dp[i-1][j-1]。

2，当word1[i]!=word2[j]：也就是说word1的第i个字符和word2的第j个字符不相等。这时我们可以有3种操作来计算dp[i][j];

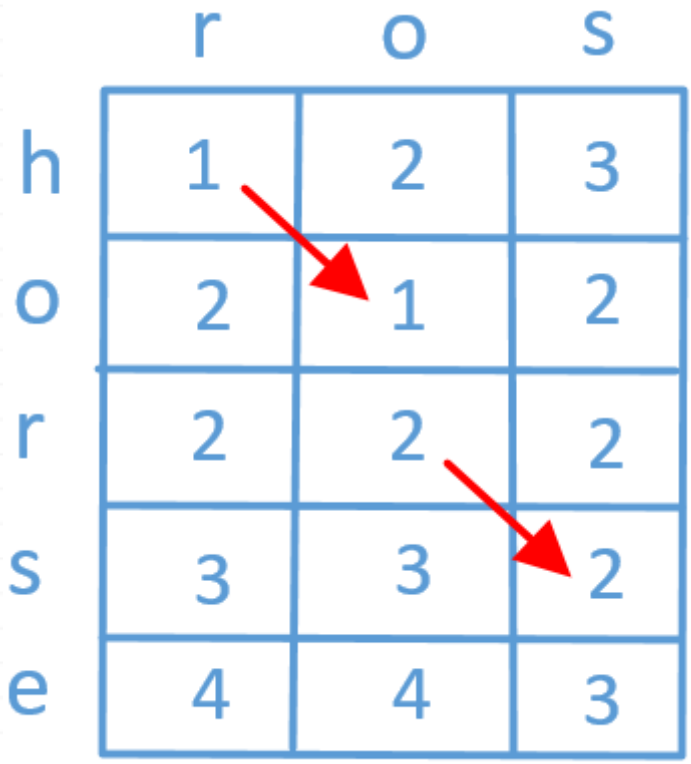
- **删**，dp[i-1][j]：表示的是word1的前i-1个字符和word2的前j个字符的最小编辑距离，在dp[i][j]中我们只需要把word1中第i个字符删除就是dp[i-1][j]，所以dp[i][j]=dp[i-1][j]+1。
- **增**，dp[i][j-1]：表示的是word1的前i个字符和word2的前j-1个字符的最小编辑距离，在dp[i][j]中我们只需要把word2中的第j个字符删除就是dp[i][j-1]，所以dp[i][j]=dp[i][j-1]+1。（注：我们这一步明明是增，但这里为什么是删，因为我们这里删的是word2的字符，增和删是相对的，word2字符的删除也可以认为是word1字符的添加，举个例子，比如word1="a"，word2="ab"，我们在word1中添加一个b或者在word2中删除一个b，最短编辑距离都是一样的）
- **改**，dp[i-1][j-1]：表示的是word1的前i-1个字符和word2的前j-1个字符的最小编辑距离，我们只需要把word1的第i个字符修改为word2的第j个字符就可以求出dp[i][j]，所以dp[i][j]=dp[i-1][j-1]+1。

上面三种情况我们要选最小的，所以递推公式
1，当word1[i]==word2[j]:
dp[i][j]=dp[i-1][j-1]

2，当word1[i]!=word2[j]:
dp[i][j]=min{dp[i-1][j-1]，dp[i-1][j]，dp[i][j-1]}+1

边界条件：
如果word1为空，我们要把word1变为word2就是不停的插入，
如果word2为空，我们要把word1变为word2就是不停的删除。

下面我们来画个图看一下



举个例子，
比如（0，0）格内，我们只需要把h变为r即可，所以需要1步。
比如（0，1）格内，我们只需要把h变为r，然后删除O，所以需要2步。
比如（1，0）格内，我们只需要把h变为r，然后在添加一个O，所以需要2步。
比如（1，1）格内，因为O==O，我们只需要把h变为r即可，所以需要1步。

看懂了上面的分析过程，代码就容易多了，我们来看下代码

01 代码部分

```java
public static int minDistance(String word1, String word2) {
    int length1 = word1.length();
    int length2 = word2.length();
    if (length1 * length2 == 0)
        return length1 + length2;//如果有一个为空，直接返回另一个的长度即可
    int dp[][] = new int[length1 + 1][length2 + 1];
    for (int i = 0; i <= length1; i++) {
        dp[i][0] = i;//边界条件，相当于word1的删除操作
    }
    for (int i = 0; i <= length2; i++) {
        dp[0][i] = i;//边界条件，相当于word1的添加操作
    }
    for (int i = 1; i <= word1.length(); i++) {
        for (int j = 1; j <= length2; j++) {//下面是上面分析的递推公式
            if (word1.charAt(i - 1) == word2.charAt(j - 1)) {//判断两个字符是否相等
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]) + 1;
            }
        }
    }
    Util.printTwoIntArrays(dp);//测试数据的打印，可去掉
```

```
23      return dp[length1][length2];
24    }
```

代码比较简单，核心代码也就15到19行，其他的也就是一些边界的判断。

我们还用上面的数据测试一下，看一下打印结果

```
1   public static void main(String args[]) {
2     System.out.println(minDistance("horse", "ros"));
3   }
```

结果如下

```
0   1   2   3
1   1   2   3
2   2   1   2
3   2   2   2
4   3   3   2
5   4   4   3
3
```

和我们上面分析的完全一致。

02 代码优化

我们看到虽然dp是二维数组，但我们计算的时候每个元素只和他的左边，上边，左上角的3个值有关，所以这里我们还可以优化一下，使用一维数组，我们看下代码

```
1   public static int minDistance2(String word1, String word2) {
2     int length1 = word1.length();
3     int length2 = word2.length();
4     if (length1 * length2 == 0)
5       return length1 + length2;
6     int dp[] = new int[length2 + 1];
7     for (int i = 1; i <= length2; i++) {
8       dp[i] = i;
9     }
```

```
    int last = 0;
    for (int i = 1; i <= word1.length(); i++) {
      last = dp[0];
      dp[0] = i;
      for (int j = 1; j <= length2; j++) {
        int temp = dp[j];
        if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
          dp[j] = last;
        } else {
          dp[j] = Math.min(Math.min(dp[j - 1], dp[j]), last) + 1;
        }
        last = temp;
      }
      Util.printIntArrays(dp);//这两行代码仅做测试打印数据使用，可删除
      System.out.println();
    }
    return dp[length2];
  }
```

代码中last记录的是左上角的值，因为这个值会被覆盖，所以我们提前记录了下来，我们还用上面的代码测试一下，再来看一下打印结果

```
1   1  2  3
2   2  1  2
3   2  2  2
4   3  3  2
5   4  4  3
3
```

结果和我们上面分析的完全一致。

总结：

这道题相对来说还是有一定的难度的，首先要了解什么是动态规划，然后再找出他的递推公式，还有一些边界条件的判断，最后是代码的优化。

往期推荐

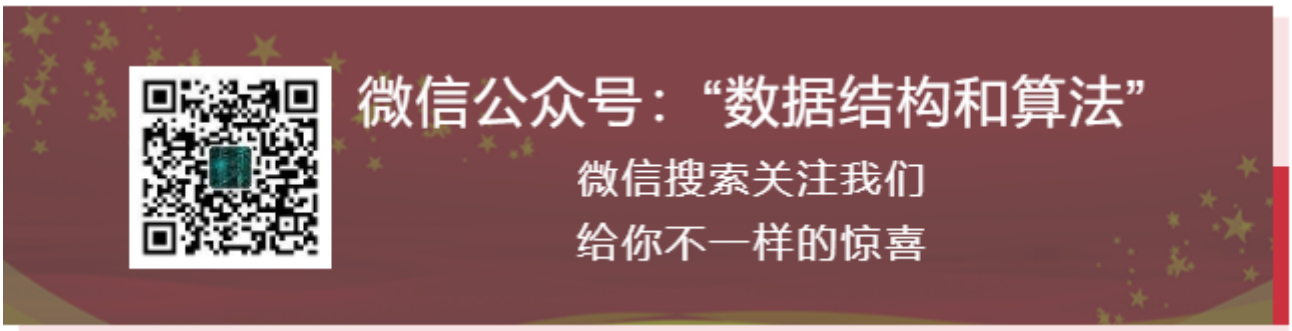
- 371，背包问题系列之-基础背包问题
- 370，最长公共子串和子序列
- 356，青蛙跳台阶相关问题

370，最长公共子串和子序列

原创 山大王wld 数据结构和算法 5月27日

收录于话题
#算法图文分析

96个 >



1，最长公共子串

假如有两个字符串，s1="people"和s2="eplm"，我们要求他俩最长的公共子串。我们一眼就能看出他们的最长公共子串是"pl"，长度是2。但如果字符串特别长的话就不容易那么观察了。

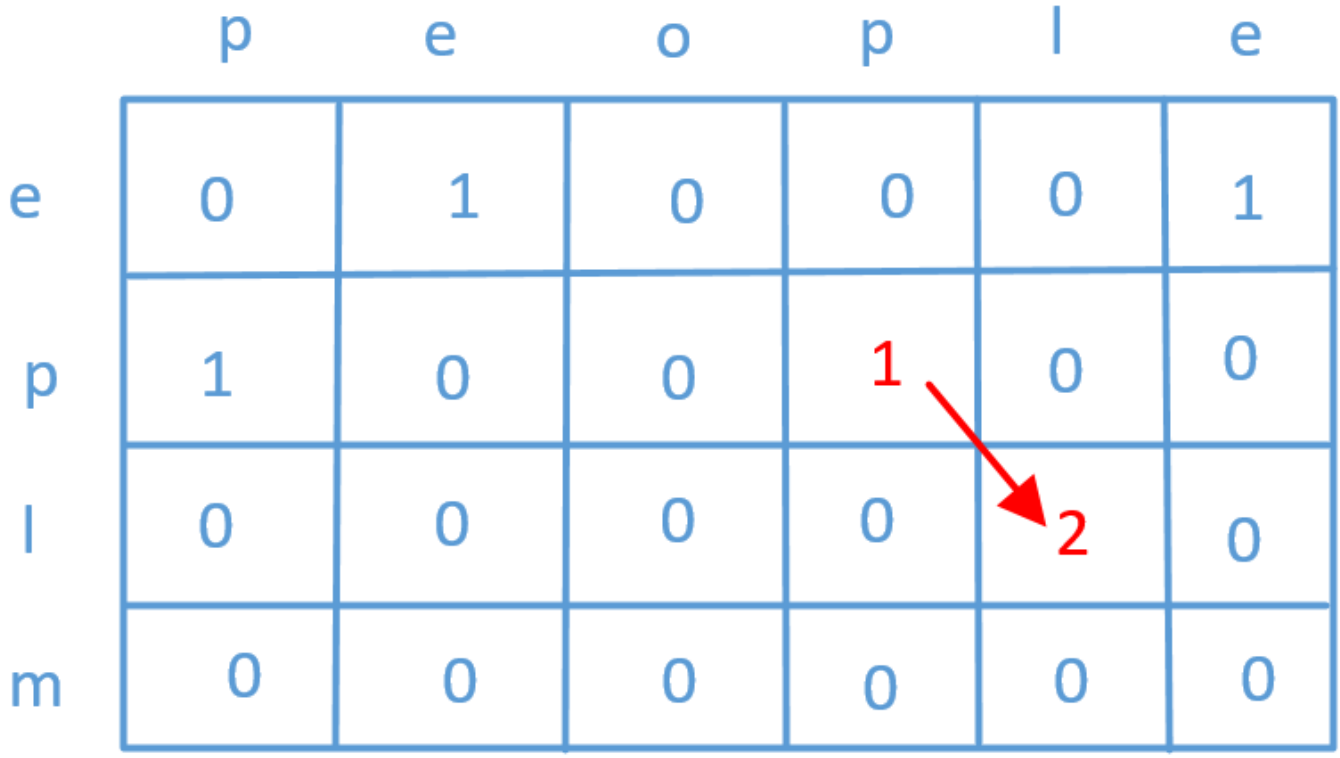
1，暴力求解：暴力求解对于字符串比较短的我们还可以接受，如果字符串太长实在是效率太低，所以这种我们就不再考虑

2，动态规划：我们用一个二维数组dp[i][j]表示第一个字符串前i个字符和第二个字符串前j个字符组成的最长公共字符串的长度。那么我们在计算dp[i][j]的时候，我们首先要判断s1.charAt(i)是否等于s2.charAt(j)，如果不相等，说明当前字符无法构成公共子串，所以dp[i][j]=0。如果相等，说明可以构成公共子串，我们还要加上他们前一个字符构成的最长公共子串长度，也就是dp[i-1][j-1]。所以我们很容易找到递推公式

01 最长公共子串的递推公式

```
1     if(s1.charAt(i) == s2.charAr(j))
2       dp[i][j] = dp[i-1][j-1] + 1;
3     else
4       dp[i][j] = 0;
```

02 最长公共子串画图分析



我们看到在动态规划中，最大值不一定是在最后一个空格内，所以我们要使用一个临时变量在遍历的时候记录下最大值。代码如下

03 最长公共子串代码

```
1  public static int maxLong(String str1, String str2) {
2    if (str1 == null || str2 == null || str1.length() == 0 || str2.length() == 0)
3      return 0;
4    int max = 0;
5    int[][] dp = new int[str1.length() + 1][str2.length() + 1];
6    for (int i = 1; i <= str1.length(); i++) {
7      for (int j = 1; j <= str2.length(); j++) {
8        if (str1.charAt(i - 1) == str2.charAt(j - 1))
9          dp[i][j] = dp[i - 1][j - 1] + 1;
10       else
11         dp[i][j] = 0;
12       max = Math.max(max, dp[i][j]);
13     }
14   }
15   Util.printTwoIntArrays(dp);//这一行是打印测试数据的，也可以去掉
16   return max;
17 }
18
```

2-3行是一些边界的判断。

重点是在8-11行，就是我们上面提到的递推公式。

第12行是记录最大值，因为这里最大值不一定出现在数组的最后一个位置，所以要用一个临时变量记录下来。

第15行主要用于数据的测试打印，也可以去掉。

我们还用上面的数据来测试一下，看一下结果

```
1   public static void main(String[] args) {
2     System.out.println(maxLong("eplm", "people"));
3   }
```

运行结果

```
"D:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ..
0   0   0   0   0   0   0
0   0   1   0   0   0   1
0   1   0   0   1   0   0
0   0   0   0   2   0
0   0   0   0   0   0
2

Process finished with exit code 0
```

结果和我们上面图中分析的完全一致。

我们发现上面的代码有个规律，就是在遍历的时候只使用了dp数组的上面一行，其他的都用不到，所以我们可以考虑把二维数组转化为一位数组，来看下代码

04 最长公共子串代码优化

```
1   public static int maxLong(String str1, String str2) {
2     if (str1 == null || str2 == null || str1.length() == 0 || str2.length() == 0)
3       return 0;
4     int max = 0;
5     int[] dp = new int[str2.length() + 1];
6     for (int i = 1; i <= str1.length(); i++) {
7       for (int j = str2.length(); j >= 1; j--) {
8         if (str1.charAt(i - 1) == str2.charAt(j - 1))
9           dp[j] = dp[j - 1] + 1;
10        else
11          dp[j] = 0;
12        max = Math.max(max, dp[j]);
13      }
14      Util.printIntArrays(dp);//这一行和下面一行是打印测试数据的，也可以去掉
15      System.out.println();
16    }
17    return max;
18  }
```

上面第7行的for循环我们使用的倒序的方式，这是因为dp数组后面的值会依赖前面的值，而前面的值不依赖后面的值，所以后面的值先修改对前面的没影响，但前面的值修改会对后面的值有影响，所以这里要使用倒序的方式。

我们还用上面的两个字符串来测试打印一下

```
"D:\Program Files\Java\jdk1.8.0_211\bin\java.exe" .
0    0    1    0    0    0    1
0    1    0    0    1    0    0
0    0    0    0    0    2    0
0    0    0    0    0    0    0
2

Process finished with exit code 0
```

我们看到结果和之前的完全一样。

2，最长公共子序列

上面我们讲了最长公共子串，子串是连续的。下面我们来讲一下最长公共子序列，而子序列不是连续的。我们还来看上面的两个字符串s1="people"，s2="eplm"，我们可以很明显看到他们的最长公共子序列是"epl"，我们先来画个图再来找一下他的递推公式。

05 最长公共子序列画图分析



我们通过上面的图分析发现，子序列不一定都是连续的，只要前面有相同的子序列，哪怕当前比较的字符不一样，那么当前字符串之前的子序列也不会为0。换句话说，如果当前字符不一样，我们只需要把第一个字符串往前退一个字符或者第二个字符串往前退一个字符然后记录最大值即可。

举个例子，比如图中第4行第4列（就是图中灰色部分），p和m不相等，如果字符串"eplm"退一步是"epl"再和"epop"对比我们发现有2个相同的子序列（也就是上面表格中数组（2，3）的位置）。如果字符串"peop"退一步是"peo"再和"eplm"对比我们发现只有1个相同的子序列（这里的pe和ep只能有一个相同，要么p相同，要么e相同，因为子序列的顺序不能变）（也就是上面表格中数组（3，2）的位置）。所以我们很容易找出递推公式

06 最长公共子序列的递推公式

```
1     if(s1.charAt(i) == s2.charAr(j))
2        dp[i][j] = dp[i-1][j-1] + 1;
3     else
4        dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
```

有了上面的递推公式，代码就很容易写出来了，我们来看下

07 最长公共子序列代码

```
1   public static int maxLong(String str1, String str2) {
2     if (str1 == null || str2 == null || str1.length() == 0 || str2.length() == 0)
3       return 0;
4     int[][] dp = new int[str1.length() + 1][str2.length() + 1];
5     for (int i = 1; i <= str1.length(); i++) {
```

```
6       for (int j = 1; j <= str2.length(); j++) {
7         if (str1.charAt(i - 1) == str2.charAt(j - 1))
8           dp[i][j] = dp[i - 1][j - 1] + 1;
9         else
10          dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
11      }
12    }
13    Util.printTwoIntArrays(dp);//这一行是打印测试数据的，也可以去掉
14    return dp[str1.length()][str2.length()];
15  }
16
```

我们发现他和最长公共子串的唯一区别就在第10行，我们还用图中分析的两个字符串测试一下，看一下结果

08 最长公共子序列测试结果

```
"D:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...
0   0   0   0   0   0   0
0   0   1   1   1   1   1
0   1   1   1   2   2   2
0   1   1   1   2   3   3
0   1   1   1   2   3   3
3

Process finished with exit code 0
```

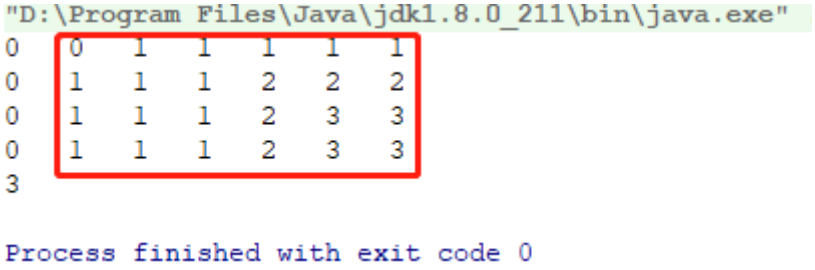
我们看到打印的结果和上面图中分析的完全一致。上面在讲到最长公共子串的时候我们可以把二维数组变为一维数组来实现对代码性能的优化，这里我们也可以参照上面的代码来优化一下，但这里和上面稍微有点不同，如果当前字符相同的时候，他会依赖左上角的值，但这个值有可能会被上一步计算的时候就被替换掉了，所以我们必须要先保存下来，我们来看下代码

09 最长公共子序列代码优化

```
1   public static int maxLong(String str1, String str2) {
2     if (str1 == null || str2 == null || str1.length() == 0 || str2.length() == 0)
3       return 0;
4     int[] dp = new int[str2.length() + 1];
5     for (int i = 1; i <= str1.length(); i++) {
6       int last = 0;
7       for (int j = 1; j <= str2.length(); j++) {
8         int temp = dp[j];//dp[j]这个值会被替换，所以替换之前要把他保存下来
9         if (str1.charAt(i - 1) == str2.charAt(j - 1))
10          dp[j] = last + 1;
11        else
12          dp[j] = Math.max(dp[j], dp[j - 1]);
13        last = temp;
14      }
15      Util.printIntArrays(dp);//这一行和下面一行是打印测试数据的，也可以去掉
16      System.out.println();
17    }
18    return dp[str2.length()];
19  }
```

代码在第8行的时候先把要被替换的值保存下来，我们还是用上面的数据来测试一下，看一下打印结果

10 最长公共子序列优化测试结果



我们看到结果和我们之前分析的完全一致。

594，回溯算法解含有重复数字的全排列 II

原创 博哥 数据结构和算法 8月13日

问题描述

来源：LeetCode第47题
难度：中等

给定一个可包含重复数字的序列nums，按任意顺序返回所有不重复的全排列。

示例 1：

```
输入：nums = [1,1,2]
输出：
[[1,1,2],
 [1,2,1],
 [2,1,1]]
```

示例 2：

```
输入：nums = [1,2,3]
输出：
[[1,2,3],
 [1,3,2],
 [2,1,3],
 [2,3,1],
 [3,1,2],
 [3,2,1]]
```

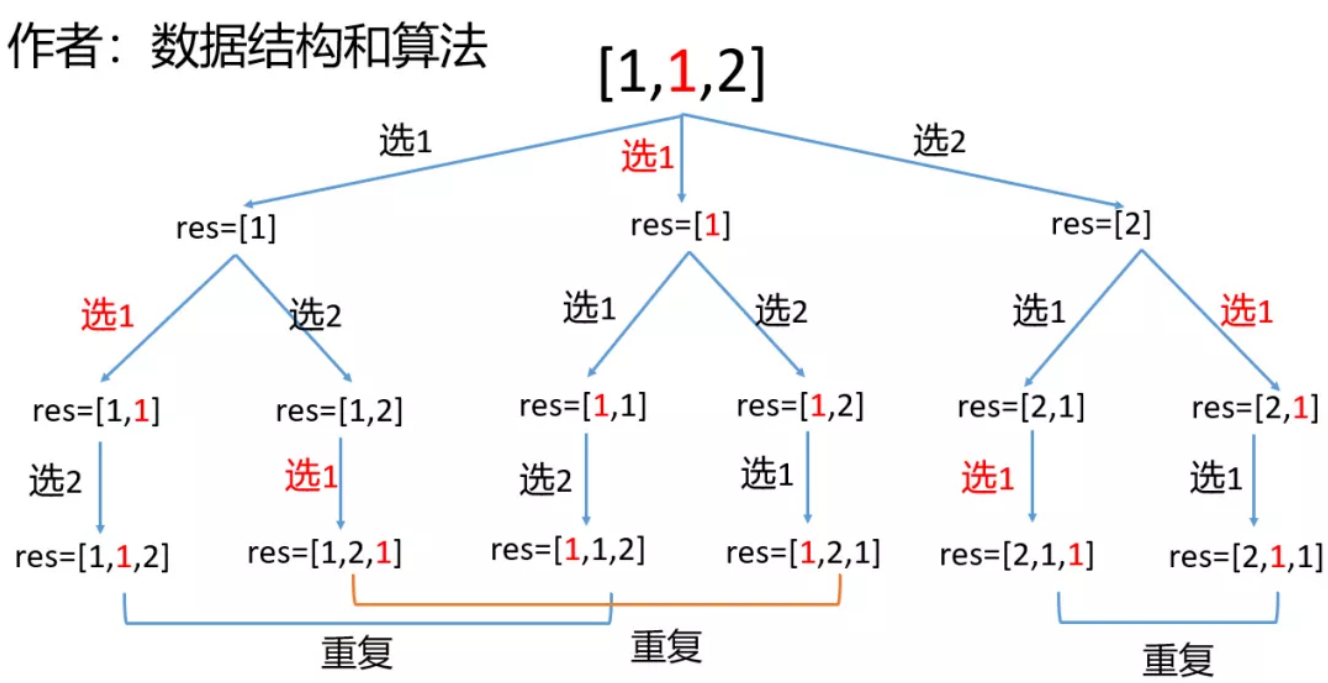
提示：

- 1 <= nums.length <= 8
- -10 <= nums[i] <= 10

回溯算法解决

这题和前面讲的593，经典回溯算法题-全排列差不多，不过这题有重复数字，但593题没有重复数字。有重复的数字肯定就会有重复的组合，所以这题需要过滤掉重复的组合。

如果不过滤会有什么结果，我们以示例一为例来个图来看一下（这里为了区分第一个1和第二个1，我分别用了黑色和红色标记，在公众号中如果看不清可以点击放大）。

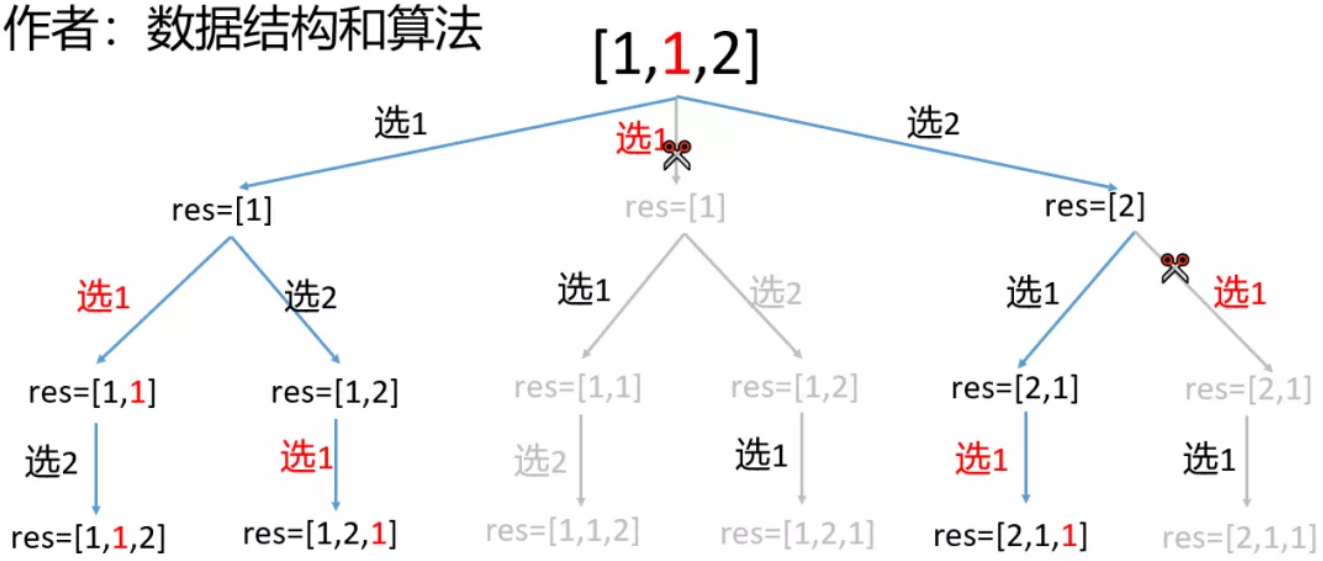


怎么样才能过滤掉重复的数字呢，一种方式就是找出所有的组合结果，然后在这个结果中过滤掉重复的组合。如果组合是字符串还好比较，但这里是个数组，所有数组两两比较复杂度太高，这种方式我们不考虑。

除了上面说的一种解法还有一种方式就是我们常说的剪枝，怎么剪呢？因为要过滤掉重复的，只有重复的数字才会造成重复的结果。所以第一步要做的就是对数组进行排序，排序之后相同的数字肯定是挨着的。

当遍历到当前数字的时候，如果数组中当前数字和前一个数字一样，并且前一个数字**没有被使用**，我们就跳过当前分支，也就是把当前分支给剪掉。如下图所示

作者：数据结构和算法

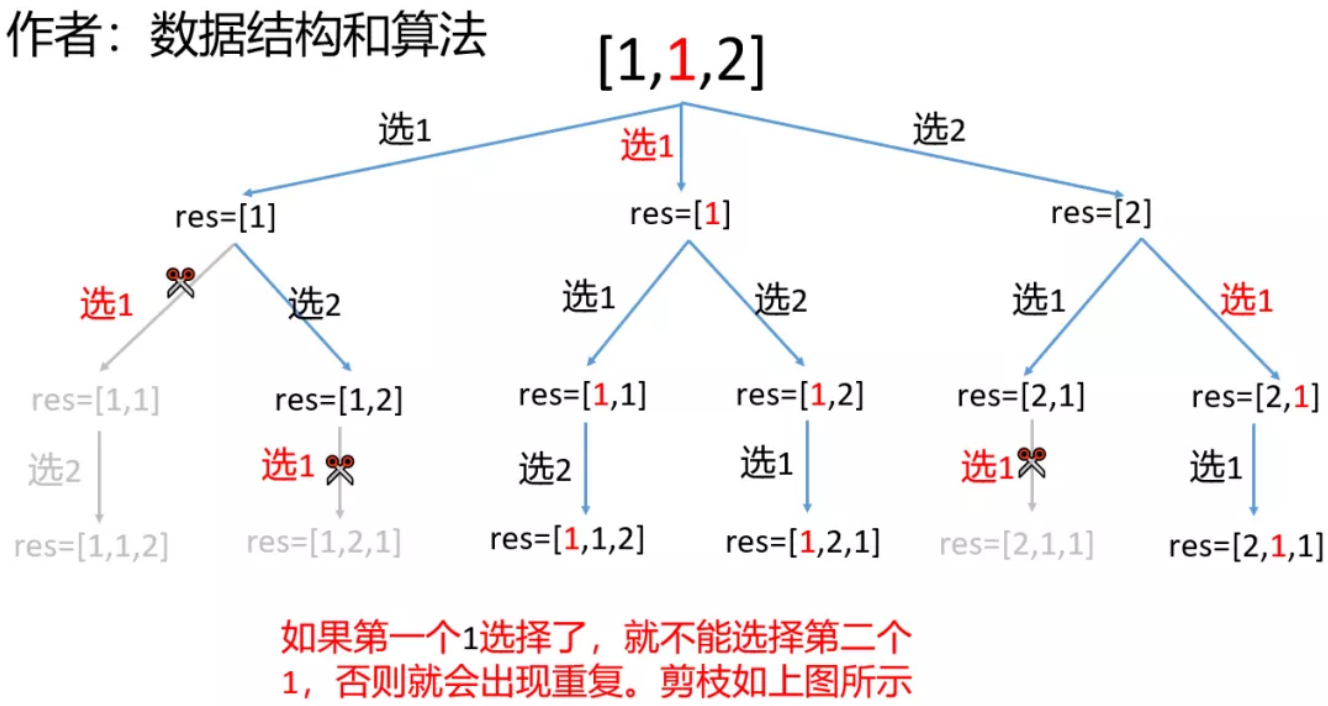


如果第一个1没有选择，我们就不能选择第二个1，否则就会出现重复。剪枝如上图所示

代码如下

```
public List<List<Integer>> permuteUnique(int[] nums) {
  //先对数组进行排序，这样做目的是相同的值在数组中肯定是挨着的，
  //方便过滤掉重复的结果
  Arrays.sort(nums);
  List<List<Integer>> res = new ArrayList<>();
  //boolean数组，used[i]表示元素nums[i]是否被访问过
  boolean[] used = new boolean[nums.length];
  //执行回溯算法
  backtrack(nums, used, new ArrayList<>(), res);
  return res;
}

public void backtrack(int[] nums, boolean[] used, List<Integer> tempList, List<List<Integer>> res) {
  //如果数组中的所有元素都使用完了，类似于到了叶子节点，
  //我们直接把从根节点到当前叶子节点这条路径的元素加入
  //到集合res中
  if (tempList.size() == nums.length) {
    res.add(new ArrayList<>(tempList));
    return;
  }
  //遍历数组中的元素
  for (int i = 0; i < nums.length; i++) {
    //如果已经被使用过，则直接跳过
    if (used[i])
      continue;
    //注意，这里要剪掉重复的组合
    //如果当前元素和前一个一样，并且前一个没有被使用过，我们也跳过
    if (i > 0 && nums[i - 1] == nums[i] && !used[i - 1])
      continue;
    //否则我们就使用当前元素，把他标记为已使用
    used[i] = true;
    //把当前元素nums[i]添加到tempList中
    tempList.add(nums[i]);
    //递归，类似于n叉树的遍历，继续往下走
    backtrack(nums, used, tempList, res);
    //递归完之后会往回走，往回走的时候要撤销选择
    used[i] = false;
    tempList.remove(tempList.size() - 1);
  }
}
```

除了上面说的剪枝方式，还有没有其他的剪枝方式呢，实际上是有的。就是当遍历到当前数字的时候，如果当前数字和数组中前一个数字一样，并且前一个数字**被使用了**，我们就跳过当前分支，也就是把当前分支给剪掉（和上面的相反）。如下图所示



这就是前面我们在讲590，回溯算法解正方形数组的数目中最后提到的，这两种剪枝方式都是可以的，一种是把整个大枝剪掉，一种是在每个大枝下面不停的剪小枝。很明显第一种剪枝效率更高一些，我们来看下代码

```
public List<List<Integer>> permuteUnique(int[] nums) {
    //先对数组进行排序，这样做目的是相同的值在数组中肯定是挨着的，
    //方便过滤掉重复的结果
    Arrays.sort(nums);
    List<List<Integer>> res = new ArrayList<>();
    //boolean数组，used[i]表示元素nums[i]是否被访问过
    boolean[] used = new boolean[nums.length];
    //执行回溯算法
    backtrack(nums, used, new ArrayList<>(), res);
    return res;
}

public void backtrack(int[] nums, boolean[] used, List<Integer> tempList, List<List<Integer>> res) {
    //如果数组中的所有元素都使用完了，类似于到了叶子节点，
    //我们直接把从根节点到当前叶子节点这条路径的元素加入
    //到集合res中
    if (tempList.size() == nums.length) {
        res.add(new ArrayList<>(tempList));
        return;
    }
    //遍历数组中的元素
    for (int i = 0; i < nums.length; i++) {
        //如果已经被使用过，则直接跳过
        if (used[i])
            continue;
        //注意，这里要剪掉重复的组合
        //如果当前元素和前一个一样，并且前一个被使用了，我们也跳过
        if (i > 0 && nums[i - 1] == nums[i] && used[i - 1])
            continue;
```

```
            //否则我们就使用当前元素，把他标记为已使用
            used[i] = true;
            //把当前元素nums[i]添加到tempList中
            tempList.add(nums[i]);
            //递归，类似于n叉树的遍历，继续往下走
            backtrack(nums, used, tempList, res);
            //递归完之后会往回走，往回走的时候要撤销选择
            used[i] = false;
            tempList.remove(tempList.size() - 1);
        }
    }
```

上面两种代码非常相似，唯一不同的就是下面这行，其他的都一样。

```
if (i > 0 && nums[i - 1] == nums[i] && used[i - 1])
```

如果让我们选择的话，我们肯定会选择第一种方式，把整个大的枝给剪掉。

往期推荐

- 593，经典回溯算法题-全排列
- 590，回溯算法解正方形数组的数目
- 551，回溯算法解分割回文串
- 450，什么叫回溯算法，一看就会，一写就废

593，经典回溯算法题-全排列

原创 博哥 数据结构和算法 8月11日

问题描述

来源：LeetCode第46题
难度：中等

给定一个不含重复数字的数组nums，返回其所有可能的全排列。你可以按任意顺序返回答案。

示例 1：

> **输入**：nums = [1,2,3]
> **输出**：[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

示例 2：

> **输入**：nums = [0,1]
> **输出**：[[0,1],[1,0]]

示例 3：

> **输入**：nums = [1]
> **输出**：[[1]]

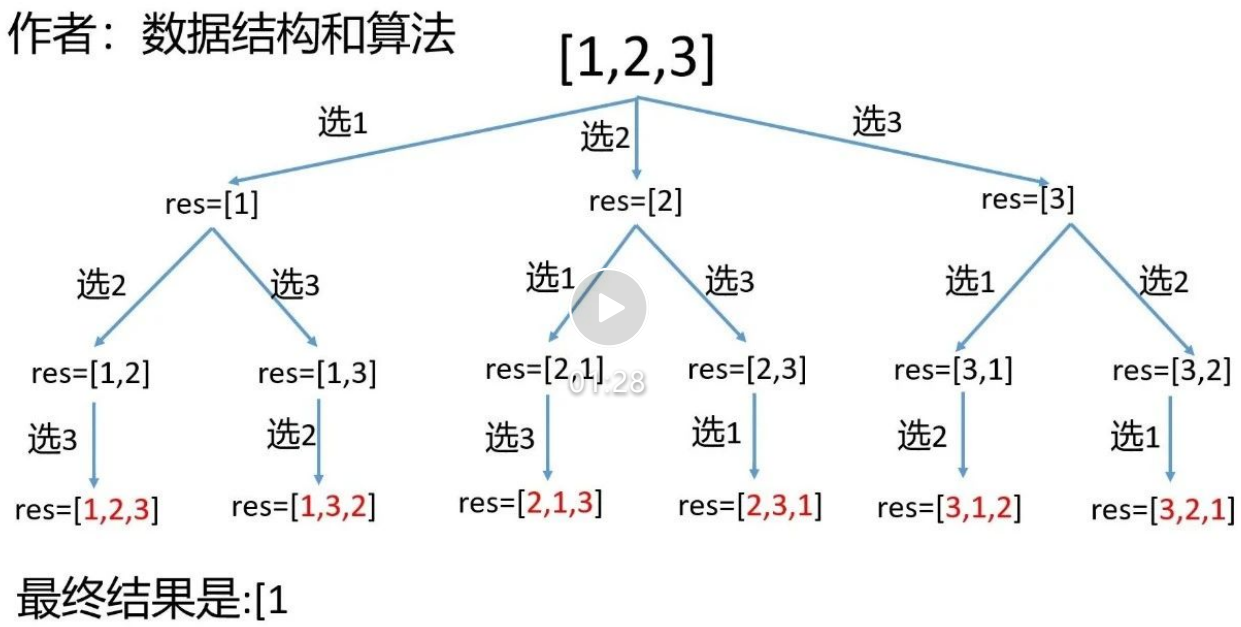
提示：

- 1 <= nums.length <= 6
- -10 <= nums[i] <= 10
- nums中的所有整数互不相同

回溯算法解决

全排列也是一道经典的题，之前在讲450，什么叫回溯算法，一看就会，一写就废的时候，也提到过使用回溯算法来解决，具体细节可以看下。假设数组长度是n，我们可以把

回溯算法看做是一颗n叉树的前序遍历，第一层有n个子节点，第二层有n-1个子节点……，来看个视频



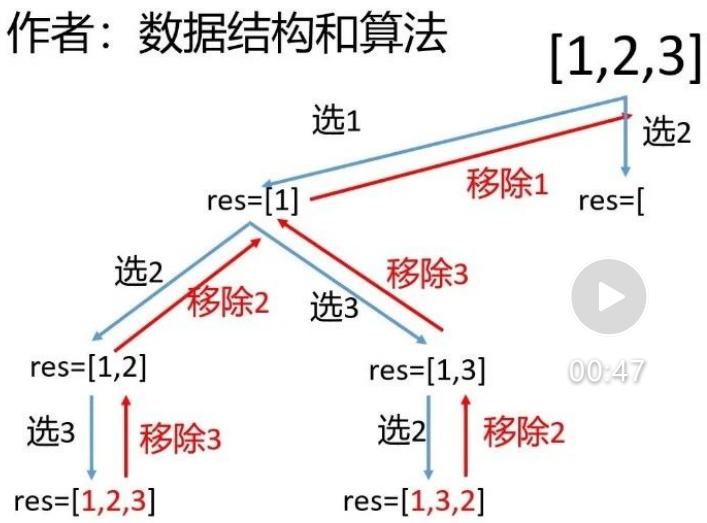
代码如下

```
public List<List<Integer>> permute(int[] nums) {
   List<List<Integer>> list = new ArrayList<>();
   backtrack(list, new ArrayList<>(), nums);
   return list;
}

private void backtrack(List<List<Integer>> list, List<Integer> tempList, int[] nums) {
   //终止条件，如果数字都被使用完了，说明找到了一个排列，（可以把它看做是n叉树到
   //叶子节点了，不能往下走了，所以要返回）
   if (tempList.size() == nums.length) {
      //因为list是引用传递，这里必须要重新new一个
      list.add(new ArrayList<>(tempList));
      return;
   }
   //（可以把它看做是遍历n叉树每个节点的子节点）
   for (int i = 0; i < nums.length; i++) {
      //因为不能有重复的，所以有重复的就跳过
      if (tempList.contains(nums[i]))
         continue;
      //选择当前值
      tempList.add(nums[i]);
      //递归（可以把它看做遍历子节点的子节点）
      backtrack(list, tempList, nums);
   }
}
```

我们就用数组[1,2,3]来测试一下，看一下打印结果

```
1  [1, 2, 3]
```

是不是很意外，示例1给出的是6个结果，这里打印的是1个，这是因为list是引用传递，当遍历到叶子节点以后要往回走，往回走的时候必须把之前添加的值给移除了，否则会越加越多，我们来看下视频



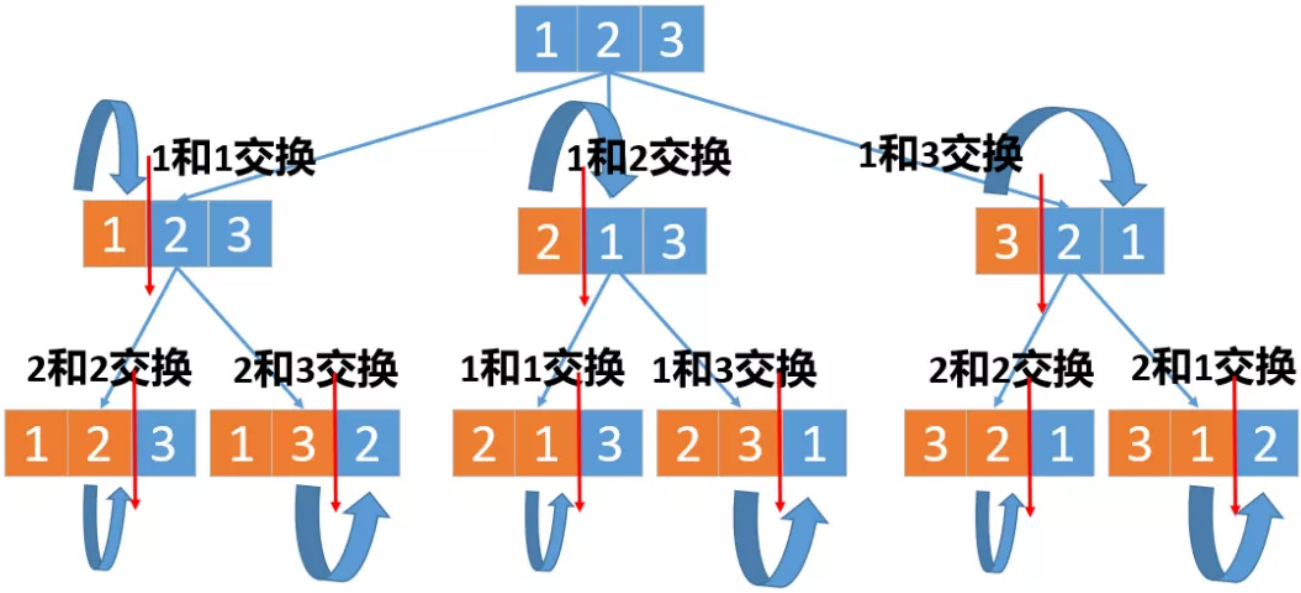
再来看下代码

```
public List<List<Integer>> permute(int[] nums) {
  List<List<Integer>> list = new ArrayList<>();
  backtrack(list, new ArrayList<>(), nums);
  return list;
}

private void backtrack(List<List<Integer>> list, List<Integer> tempList, int[] nums) {
  //终止条件，如果数字都被使用完了，说明找到了一个排列，（可以把它看做是n叉树到
  //叶子节点了，不能往下走了，所以要返回）
  if (tempList.size() == nums.length) {
    //因为list是引用传递，这里必须要重新new一个
    list.add(new ArrayList<>(tempList));
    return;
  }
  //（可以把它看做是遍历n叉树每个节点的子节点）
  for (int i = 0; i < nums.length; i++) {
    //因为不能有重复的，所以有重复的就跳过
    if (tempList.contains(nums[i]))
      continue;
    //选择当前值
    tempList.add(nums[i]);
    //递归（可以把它看做遍历子节点的子节点）
    backtrack(list, tempList, nums);
    //撤销选择，把最后一次添加的值给移除
    tempList.remove(tempList.size() - 1);
  }
}
```

我们来看一下运行结果

```
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

这题使用回溯算法的还一种解决方式就是交换，比如我们先选择第一个数字，然后和后面的所有数字都交换一遍，这样全排列的第一位就确定了。然后第二个数字在和后面的所有数字交换一遍，这样全排列的第二位数字也确定了……，一直继续下去，直到最后一个数字不能交换为止，这里画个图来看一下

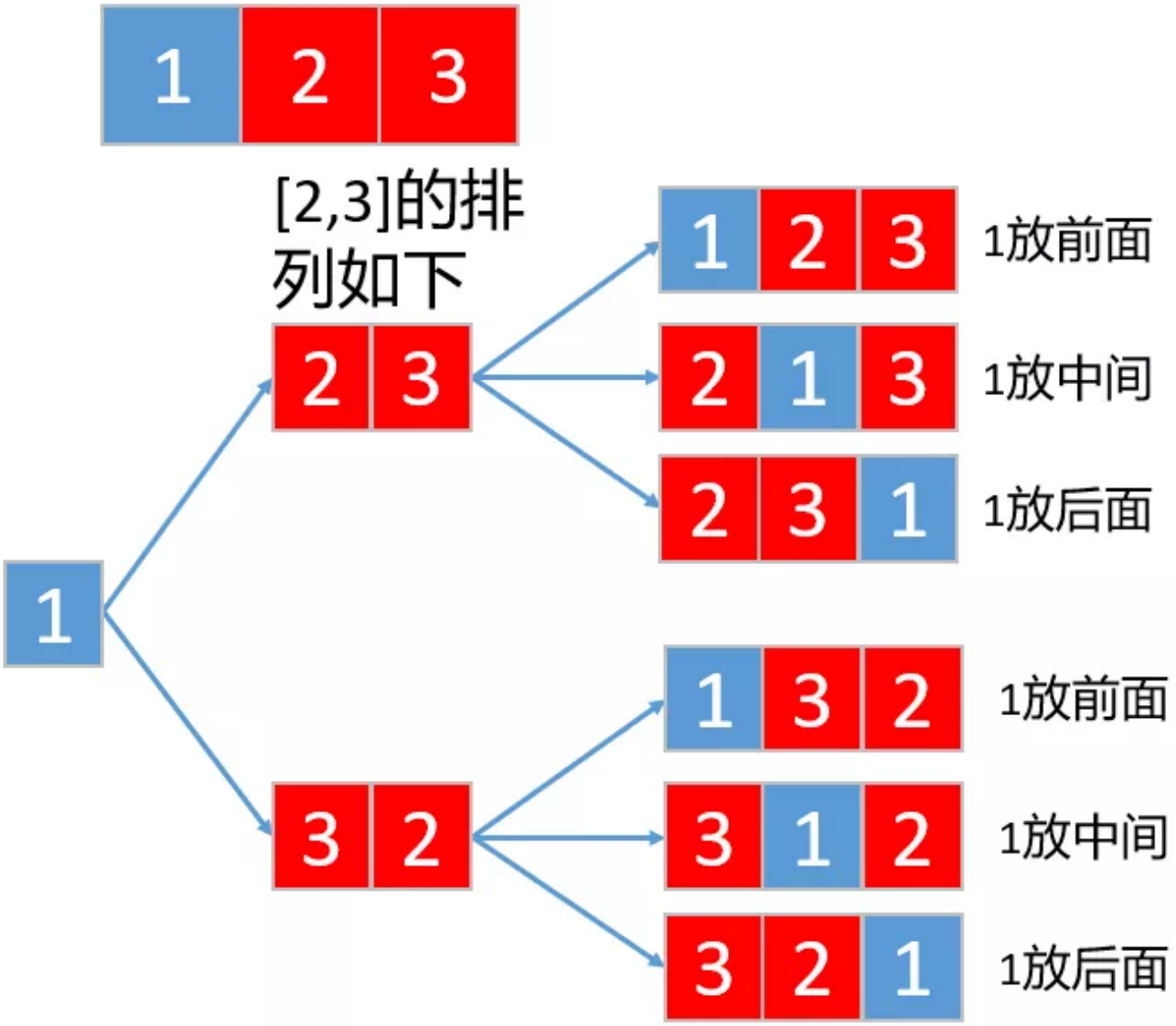


来看下代码

```
public List<List<Integer>> permute(int[] nums) {
   List<List<Integer>> res = new ArrayList<>();
   backtrack(nums, 0, res);
   return res;
}

public void backtrack(int[] nums, int index, List<List<Integer>> res) {
   //到最后一个数字，没法再交换了，直接把数组转化为list
   if (index == nums.length - 1) {
      //把数组转为list
      List<Integer> tempList = new ArrayList<>();
      for (int num : nums)
         tempList.add(num);
      //把list加入到res中
      res.add(tempList);
      return;
   }
   for (int i = index; i < nums.length; i++) {
      //但前数字nums[index]要和后面所有的数字都要交换一遍（包括
      // 他自己）
      swap(nums, index, i);
      //递归，数组[0,index]默认是已经排列好的，然后从index+1开始
      //后面元素的交换
      backtrack(nums, index + 1, res);
      //还原回来
      swap(nums, index, i);
   }
}

//交换两个数字的值
private void swap(int[] nums, int i, int j) {
   if (i != j) {
      nums[i] ^= nums[j];
      nums[j] ^= nums[i];
      nums[i] ^= nums[j];
   }
}
```

递归解决

我们来思考这样一个问题，假如数组[1,2,3]，我们知道了[2,3]的全排列结果，只需要把1放到这些全排列的所有位置，即是数组[1,2,3]的全排列，画个图来看一下




```
//递归解决
public List<List<Integer>> permute(int[] nums) {
   return helper(nums, 0);
}

/**
 * @param nums
 * @param index 递归当前数字的下标
 * @return
 */
private List<List<Integer>> helper(int[] nums, int index) {
   List<List<Integer>> res = new ArrayList<>();
   //递归的终止条件，如果到最后一个数组，直接把它放到res中
   if (index == nums.length - 1) {
      //创建一个临时数组
      List<Integer> temp = new ArrayList<>();
      //把数字nums[index]加入到临时数组中
      temp.add(nums[index]);
      res.add(temp);
```

```
        return res;
    }
    //计算后面数字的全排列
    List<List<Integer>> subList = helper(nums, index + 1);
    //集合中每个子集的长度
    int count = subList.get(0).size();
    //遍历集合subList的子集
    for (int i = 0, size = subList.size(); i < size; i++) {
        //把当前数字nums[index]添加到子集的每一个位置
        for (int j = 0; j <= count; j++) {
            List<Integer> temp = new ArrayList<>(subList.get(i));
            temp.add(j, nums[index]);
            res.add(temp);
        }
    }
    return res;
}
```

往期推荐

- 575，回溯算法和DFS解单词拆分 II
- 551，回溯算法解分割回文串
- 520，回溯算法解火柴拼正方形
- 450，什么叫回溯算法，一看就会，一写就废

590，回溯算法解正方形数组的数目

原创 博哥 数据结构和算法 8月5日

问题描述

来源：LeetCode第996题
难度：困难

给定一个非负整数数组A，如果该数组每对相邻元素之和是一个完全平方数，则称这一数组为正方形数组。

返回A的正方形排列的数目。两个排列A1和A2不同的充要条件是存在某个索引i，使得A1[i] != A2[i]。

示例 1：

> **输入**：[1,17,8]
> **输出**：2
> **解释**：
> [1,8,17] 和 [17,8,1] 都是有效的排列。

示例 2：

> **输入**：[2,2,2]
> **输出**：1

提示：

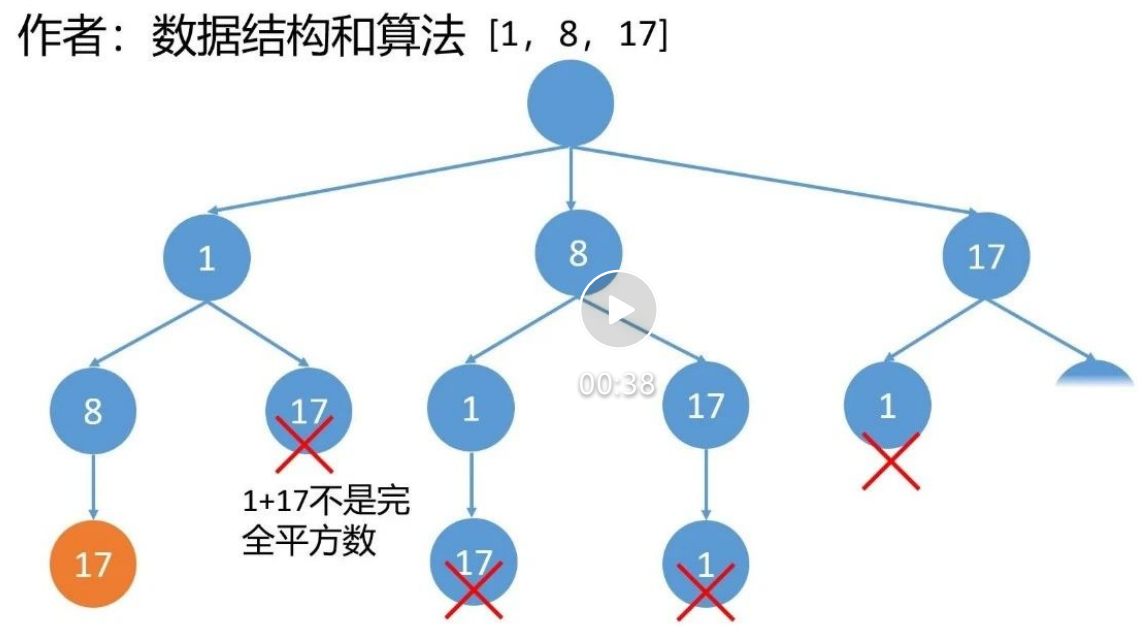
- 1 <= A.length <= 12
- 0 <= A[i] <= 1e9

回溯算法解决

这题是让打乱数组的顺序，打乱之后的数组如果所有的前后两个数字之和是一个完全平方数，那么这个打乱的数组就是正方形数组，这里让求的是正方形数组是个数。

这里我们可以使用回溯算法来解决，我之前专门讲过450，什么叫回溯算法，一看就会，

一写就废，回溯算法我们可以把它看做是一棵N叉树，也就是说每个节点最多有N个子节点，然后使用DFS进行遍历。对于这道题我们以示例1为例来看个视频



看懂了这个，这题就非常简单了，之前在讲450题回溯算法的时候，总结了一个模板

```
private void backtrack("原始参数") {
  //终止条件(递归必须要有终止条件)
  if ("终止条件") {
    //一些逻辑操作（可有可无，视情况而定）
    return;
  }

  for (int i = "for循环开始的参数"; i < "for循环结束的参数"; i++) {
    //一些逻辑操作（可有可无，视情况而定）

    做出选择

    //递归
    backtrack("新的参数");
    //一些逻辑操作（可有可无，视情况而定）

    撤销选择
  }
}
```

这道题完全可以使用这个模板，每次从数组中选择元素的时候，只能选择未使用的，使用过的则直接跳过，模板的终止条件就是所有元素都访问完了。我们来看下完整代码

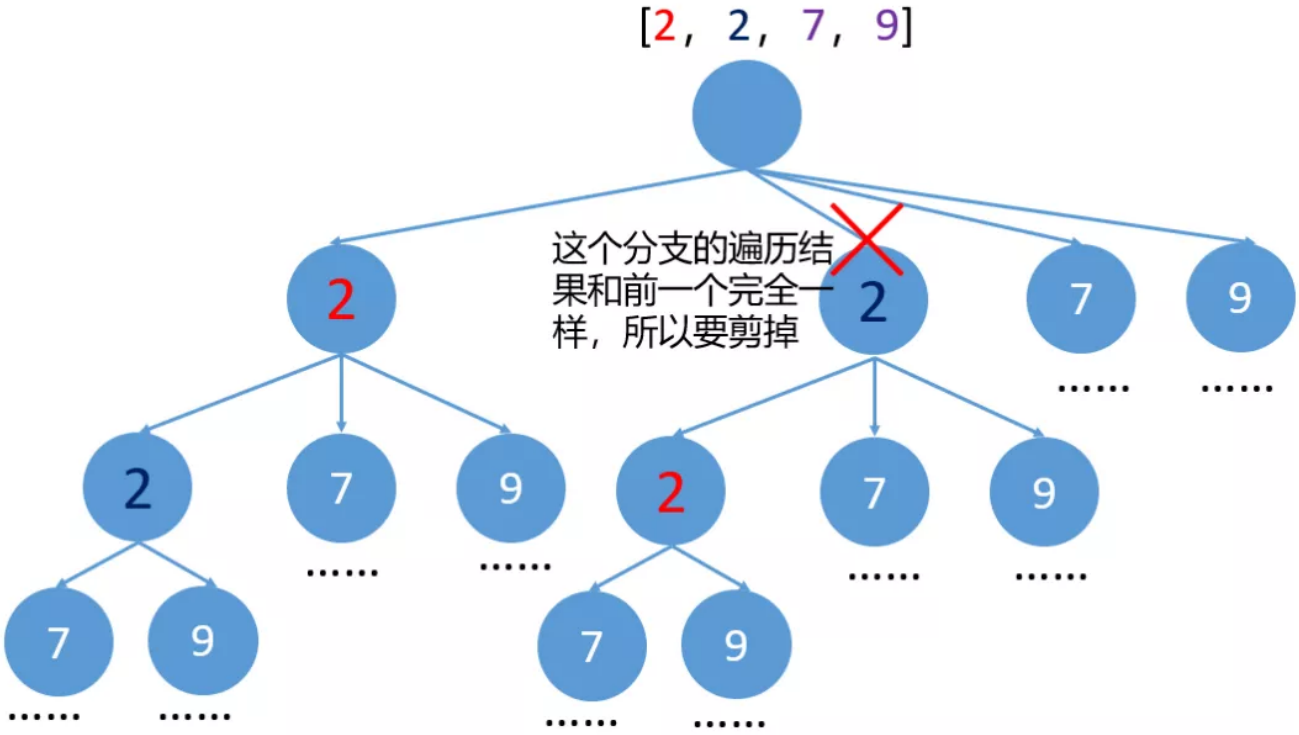
```
//统计正方形数组的个数
int count = 0;

public int numSquarefulPerms(int[] nums) {
  //先对数组进行排序，目的是过滤掉重复的
  Arrays.sort(nums);
  backtrack(nums, new boolean[nums.length], 0, -1);
  return count;
}

/**
 * @param nums
 * @param visited 标记某个数字是否被使用过
 * @param index  当前数字的下标
 * @param preNum  打乱数组的前一个数字
```

```
 */
private void backtrack(int[] nums, boolean[] visited, int index, int preNum) {
  //如果数字都访问完了，说明这个打乱的数组是一个正方形数组
  if (index == nums.length) {
    count++;
    return;
  }
  for (int i = 0; i < nums.length; i++) {
    //如果当前数字被访问过，直接跳过
    if (visited[i])
      continue;
    //如果当前数字和前一个数字不能构成完全平方数，直接跳过
    if (preNum > 0 && !isSquare(preNum + nums[i]))
      continue;
    //这里是为了过滤掉重复的，比如数组[2,2,7,9]，我们选择了
    //[2,7(第一个2)]和[2,7(第二个2)]，剩下的数字都是一样的，
    //所以他们的组合也是一样的，会出现重复，我们需要过滤掉
    if ((i > 0 && nums[i] == nums[i - 1] && !visited[i - 1]))
      continue;
    //选择当前数字，标记为已经被使用过
    visited[i] = true;
    //递归，进行到下一层
    backtrack(nums, visited, index + 1, nums[i]);
    //回到上一层，撤销选择
    visited[i] = false;
  }
}

//判断数字num是否是完全平方数
private boolean isSquare(int num) {
  int sqr = (int) Math.sqrt(num);
  return sqr * sqr == num;
}
```

上面有一行代码过滤掉重复的，其实就是剪枝，我们看一下如果相同的数字，在前一个分支遍历过，那么在当前分支就不要再次遍历了，否则会出现重复的，如下图所示。上面代码过滤的时候使用

!visited[i-1]和使用**visited[i-1]**都是可以的，只不过这两种剪枝方式是不一样的，一个是把整个大枝剪掉，一个是在每个大枝下面不停的剪小枝，这个后面讲有重复数据的全排列的时候在详细介绍。

作者：数据结构和算法



往期推荐

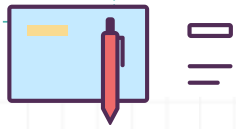
- 575，回溯算法和DFS解单词拆分 II
- 551，回溯算法解分割回文串
- 520，回溯算法解火柴拼正方形
- 450，什么叫回溯算法，一看就会，一写就废

575，回溯算法和DFS解单词拆分 II

原创 博哥 数据结构和算法 5天前

You reap what you sow.

一分耕耘，一分收获。



问题描述

来源：LeetCode第140题

难度：困难

给定一个非空字符串s和一个包含非空单词列表的字典wordDict，在字符串中增加空格来构建一个句子，使得句子中所有的单词都在词典中。返回所有这些可能的句子。

说明：

- **分隔时可以重复使用字典中的单词。**
- **你可以假设字典中没有重复的单词。**

示例 1：

> **输入：**
> s = "catsanddog"
> wordDict = ["cat", "cats", "and", "sand", "dog"]
>
> **输出：**
> [
> "cats and dog",
> "cat sand dog"
>]

示例 2：

> **输入：**
> s = "pineapplepenapple"
> wordDict = ["apple", "pen", "applepen", "pine", "pineapple"]

> **输出:**
> [
> "pine apple pen apple",
> "pineapple pen apple",
> "pine applepen apple"
>]
> **解释:** 注意你可以重复使用字典中的单词。

示例 3:

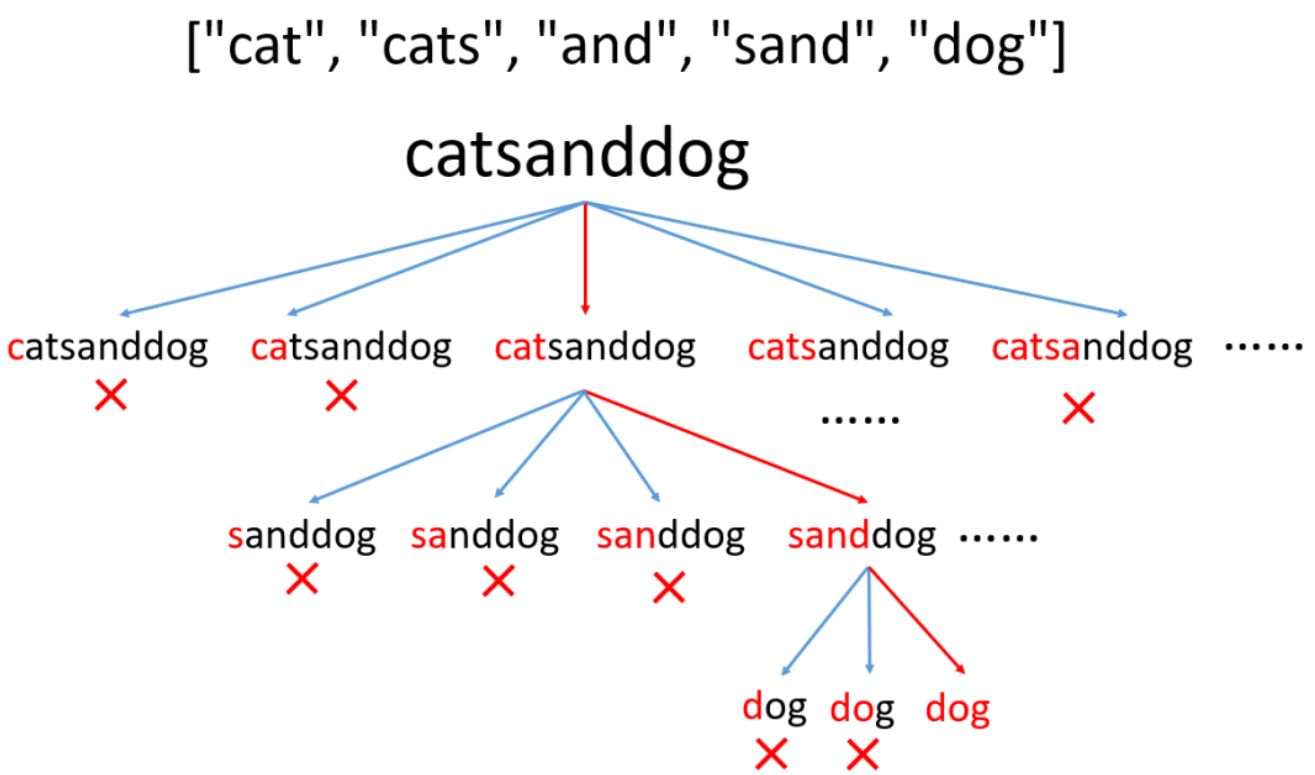
> **输入:**
> s = "catsandog"
> wordDict = ["cats", "dog", "sand", "and", "cat"]
>
> **输出:**
> []

回溯算法解决

前面我们分别通过动态规划，DFS以及BFS三种方式来判断字符串是否可以拆分，具体可以看下
573，动态规划解单词拆分
574，DFS和BFS解单词拆分

今天这题不光要判断字符串是否可以拆分，如果可以拆分还要打印所有可能的拆分结果。判断是否可拆分比较简单，我们直接使用第573题动态规划的方式来判断即可，如果不能拆分我们直接返回一个空的集合，如果可以拆分我们就拆分。

拆分的原理比较简单，我们逐渐截取子串，判断是否在字典中，如果不在字典中就继续截取更长的子串……，如果在字典中我们就沿着这个路径走下，如下图所示（这里是有两组结果，由于图画不下了，剩下的使用省略号没有画出来），我们可以把它看做是一棵n叉树



对于n叉树的路径我们可以使用回溯算法来解决，回溯算法我们之前也讲过很多次了，具体可以看下[450，什么叫回溯算法，一看就会，一写就废]，其实他有一个经典的模板

```
private void backtrack("原始参数") {
  //终止条件(递归必须要有终止条件)
  if ("终止条件") {
    //一些逻辑操作（可有可无，视情况而定）
    return;
  }

  for (int i = "for循环开始的参数"; i < "for循环结束的参数"; i++) {
    //一些逻辑操作（可有可无，视情况而定）

    //做出选择

    //递归
    backtrack("新的参数");
    //一些逻辑操作（可有可无，视情况而定）

    //撤销选择
  }
}
```

我们对这个模板稍微修改一下就是今天这题的答案，虽然这是一道hard题，但经过我们的逐步分析，发现也不是那么难，来看下代码

```
public List<String> wordBreak(String s, List<String> wordDict) {
  //集合Set的查找效率要比集合list高，这里为了提高效率，
  //先把list转化为集合set
  Set<String> set = new HashSet<>(wordDict);
  //下面是动态规划方式，判断字符串被拆解的子串是否都在字典中
  int length = s.length();
  boolean[] dp = new boolean[length + 1];
  dp[0] = true;//边界条件
  for (int i = 1; i <= length; i++) {
    for (int j = 0; j < i; j++) {
      dp[i] = dp[j] && set.contains(s.substring(j, i));
      if (dp[i]) {
        break;
      }
    }
```

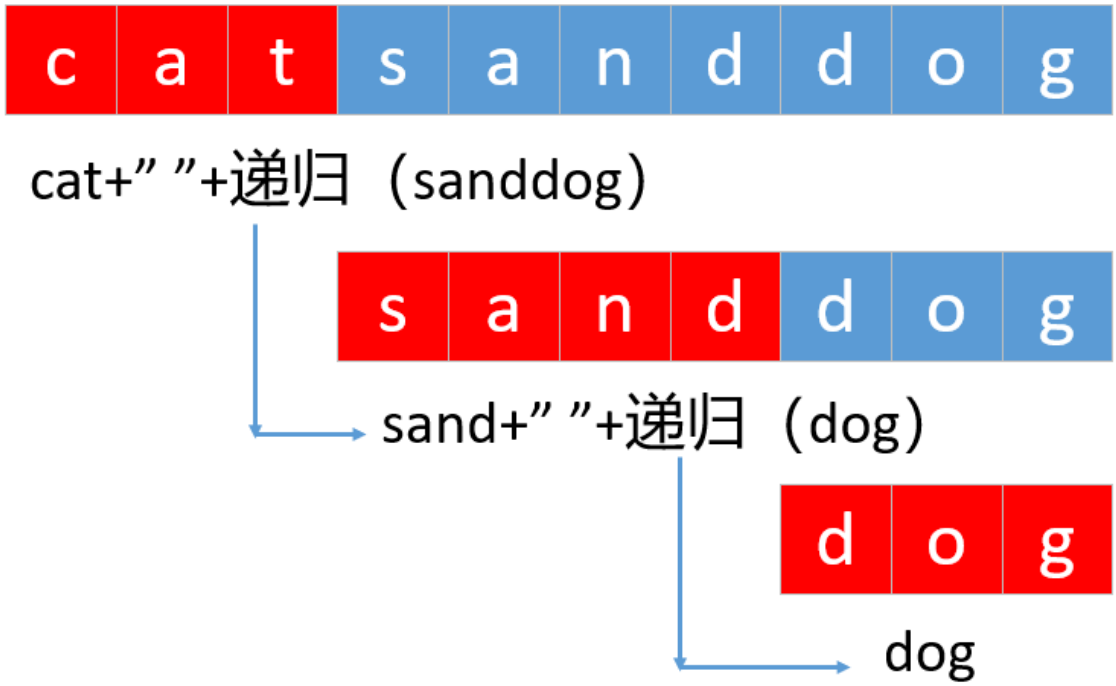
```
    }
    List<String> res = new ArrayList<>();
    //如果不能拆解，直接返回空的集合就行了
    if (!dp[length])
      return res;
    //如果能被拆解，我们通过回溯方式执行拆解
    traceback(s, set, 0, res, new ArrayList<>());
    return res;
  }

  //回溯方式拆解字符串
  private void traceback(String s, Set<String> wordDict, int start, List<String> res, List<String> pa
    if (start == s.length()) {
      //下面这两行代码都是字符串加空格然后拼接，一个是官方提供的方法，
      //一个是我自己写的，都可以实现
      //res.add(String.join(" ", path));
      res.add(join(path));
      return;
    }
    for (int i = start + 1; i <= s.length(); i++) {
      //拆解字符串
      String str = s.substring(start, i);
      //如果拆解的子串不存在于字典中，就继续扩大子串
      if (!wordDict.contains(str))
        continue;
      //如果拆解的子串存在于字典中，就把子串添加到path中
      path.add(str);
      //往下继续递归
      traceback(s, wordDict, i, res, path);
      //执行回溯，把之前添加的子串给移除
      path.remove(path.size() - 1);
    }
  }

  //字符串加空格拼接
  private String join(List<String> path) {
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < path.size(); i++) {
      stringBuilder.append(path.get(i));
      if (i != path.size() - 1)
        stringBuilder.append(" ");
    }
    return stringBuilder.toString();
  }
```

DFS解决

回溯算法比较容易理解，这题我们还可以使用DFS来解决。比如我们进行截取的时候，如果截取的子串存在于字典中，我们通过递归的方式拆分剩下的，如果剩下的能够拆分我们就把当前拆分的子串和剩下拆分的结果进行拼接，如果剩下的不能拆分，不会进行任何拼接，直接返回空的集合即可，其实这种思路还可以借鉴《464. BFS和DFS解二叉树的所有路径》的最后一种解法，实现原理如下所示

["cat", "cats", "and", "sand", "dog"]

catsanddog



来看下代码

```java
public List<String> wordBreak(String s, List<String> wordDict) {
  return backtrack(s, new HashSet<>(wordDict), 0);
}

public List<String> backtrack(String s, Set<String> wordDict, int start) {
  List<String> res = new ArrayList<>();
  for (int i = start + 1; i <= s.length(); i++) {
    String str = s.substring(start, i);
    //如果拆解的子串不存在于字典中，就继续拆
    if (!wordDict.contains(str))
      continue;
    //走到下面这个地方，说明拆分的子串str存在于字典中
    //如果正好拆完了，我们直接把最后一个子串添加到res中返回
    if (i == s.length())
      res.add(str);
    else {
      //如果没有拆完，我们对剩下的子串继续拆分
      List<String> remainRes = backtrack(s, wordDict, i);
      //然后用当前的子串str和剩下的子串进行拼接（注意如果剩下的子串不能
      //完全拆分，remainRes就会为空，就不会进行拼接）
      for (String remainStr : remainRes) {
        res.add(str + " " + remainStr);
      }
    }
  }
  return res;
}
```

我们知道递归必须要有终止条件的，但这题好像没有，其实是有的，当for循环不满足条件的时候，就不会再调用自己了。

往期推荐

- 551，回溯算法解分割回文串
- 520，回溯算法解火柴拼正方形
- 566，DFS解目标和问题
- 507，BFS和DFS解二叉树的层序遍历 II

551，回溯算法解分割回文串

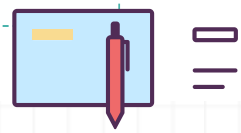
原创 博哥 数据结构和算法 5月12日

收录于话题
#算法图文分析

161个 >

I think a man does what he can until his destiny is revealed to him.

一个人应该竭尽所能，然后才听天由命。



问题描述

来源：LeetCode第131题
难度：中等

给你一个字符串s，请你将s分割成一些子串，使每个子串都是回文串。返回s所有可能的分割方案。

回文串是正着读和反着读都一样的字符串。

示例 1：

> **输入**：s = "aab"
> **输出**：[["a","a","b"],["aa","b"]]

示例 2：

> **输入**：s = "a"
> **输出**：[["a"]]

提示：

- 1 <= s.length <= 16

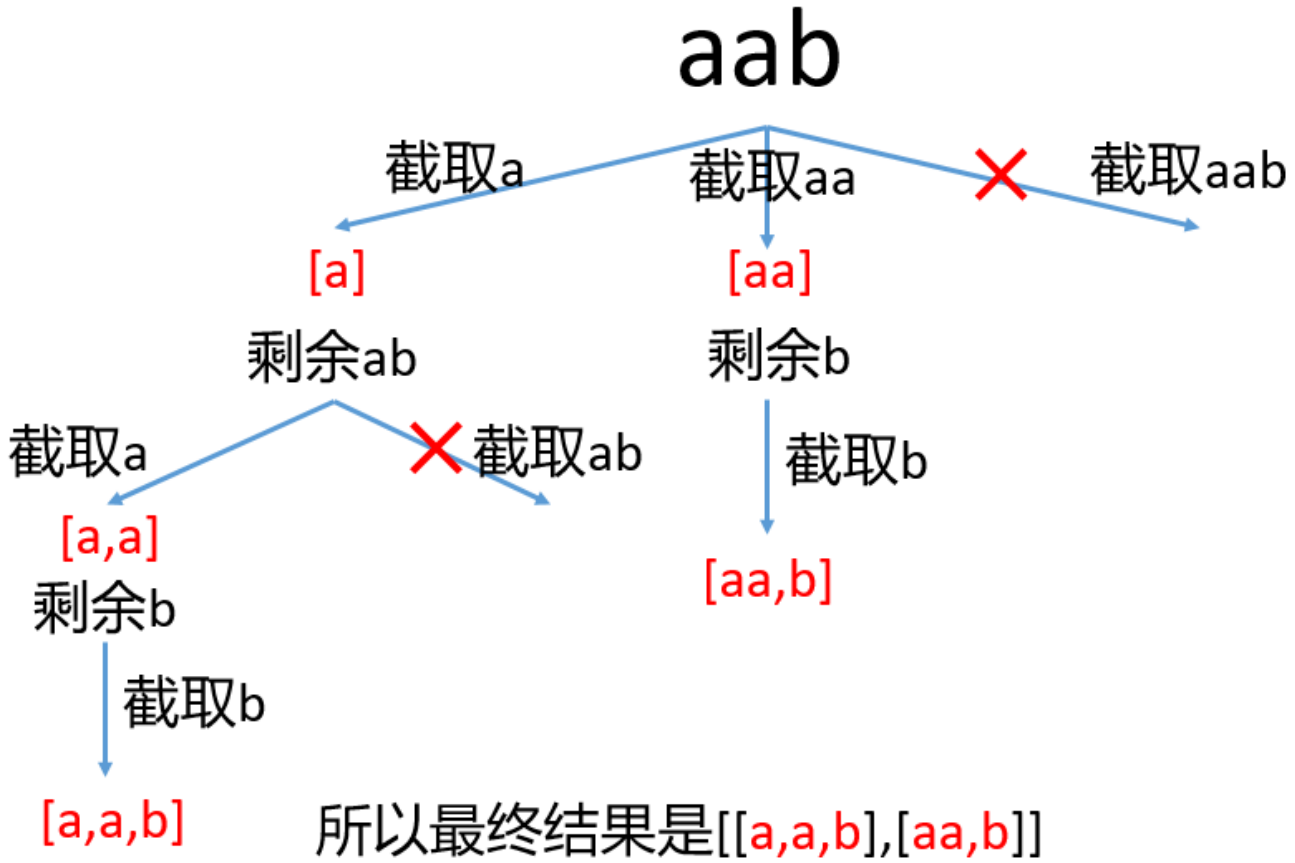
- s 仅由小写英文字母组成

回溯算法解决

这题让把字符串分隔成多个子串，并且每个子串都是回文的，问有多少种分隔方案。这里就以示例1为例来看下怎么分隔，字符串aab

- 截取1位a，把剩下的ab当做一个新的字符串再继续判断……
- 截取2位aa，把剩下的b当做一个新的字符串再继续判断……
- 截取3位aab，已经截取完了。

具体截取来看下图



来看下视频演示



一看就知道，其实就是个n叉树的DFS遍历，从根节点到叶子节点是字符串s截取的子串，我们只需要判断这些子串是否都是回文串即可，只要有一个不是就可以直接终止，如果从根节点到叶子节点的每个子串都是回文串，说明我们找到了一组截取方案。

但这题让把截取的结果返回，所以我们很容易想到的就是使用回溯算法解决，关于回溯算法不明白的可以看下《450，什么叫回溯算法，一看就会，一写就废》，大致代码如下

```
/**
 * @param s    就是需要截取的字符串
 * @param index 字符串开始截取的位置
 * @param res  最终的分隔方案结果
 * @param cur  沿着当前分支截取的子串
 */
public void backTrack(String s, int index, List<List<String>> res, List<String> cur) {
    //边界条件判断，如果字符串s中的字符都访问完了（类似于到叶子节点了），就停止查找，
    //然后这个分支的所有元素加入到集合res中
    if (index >= s.length()) {
        res.add(new ArrayList<>(cur));
        return;
    }

    for (int i = index; i < s.length(); i++) {
        //如果当前截取的子串不是回文的，就跳过
        if (!isPalindrome(s, index, i))
            continue;
        //做出选择，开始截取，把截取的子串放到集合cur中
        cur.add(s.substring(index, i + 1));
        //递归，到下一层（类似于到n叉树的子节点去遍历）
        backTrack(s, i + 1, res, cur);
        //撤销选择，就是把之前截取放到集合cur中的子串给移除掉
        cur.remove(cur.size() - 1);
    }
}
```

搞懂了上面的分析过程，我们再来看下最终代码

```
public List<List<String>> partition(String s) {
    //最终要返回的结果
    List<List<String>> res = new ArrayList<>();
    backTrack(s, 0, res, new ArrayList<>());
```

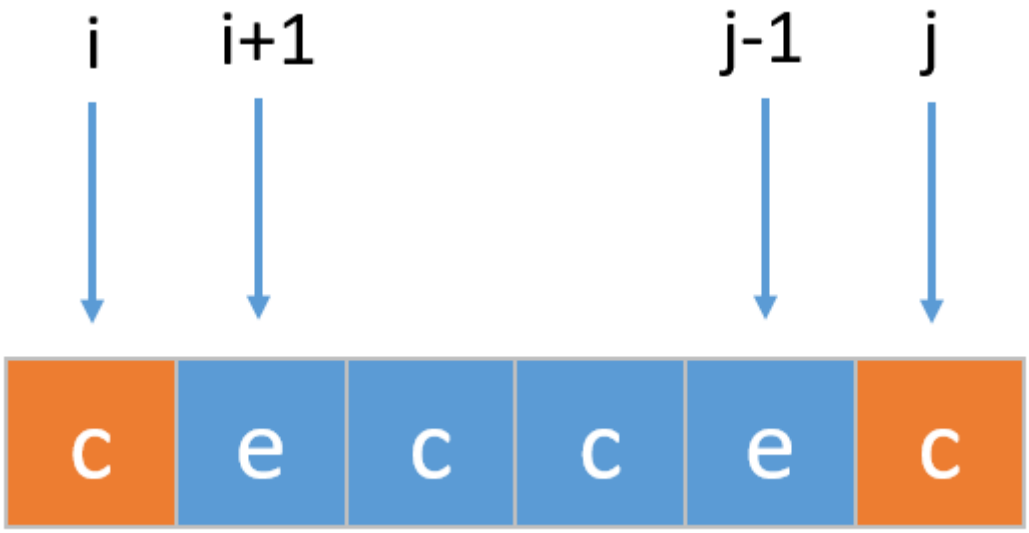
```
    return res;
}

public void backTrack(String s, int index, List<List<String>> res, List<String> cur) {
    //边界条件判断，如果字符串s中的字符都访问完了（类似于到叶子节点了），就停止查找，
    //然后这个分支的所有元素加入到集合res中
    if (index >= s.length()) {
        res.add(new ArrayList<>(cur));
        return;
    }
    for (int i = index; i < s.length(); i++) {
        //如果当前截取的子串不是回文的，就跳过
        if (!isPalindrome(s, index, i))
            continue;
        //做出选择
        cur.add(s.substring(index, i + 1));
        //递归
        backTrack(s, i + 1, res, cur);
        //撤销选择
        cur.remove(cur.size() - 1);
    }
}

//判断字符串从[left，right]的子串是否是回文的
public boolean isPalindrome(String str, int left, int right) {
    while (left < right) {
        if (str.charAt(left++) != str.charAt(right--))
            return false;
    }
    return true;
}
```

回溯算法代码优化

上面每次截取的时候都要判断截取的子串是否是回文的，实际上我们还可以先判断字符串s中所有的子串哪些是回文的哪些不是，然后计算的时候直接使用就可以了。这里使用一个二维数组dp，其中dp[i][j]表示字符串从下标i到j构成的子串是否是回文的，看下代码

```
    //数组dp[i][j]表示字符串s下标[i,j]的子串是否是回文的
    boolean[][] dp = new boolean[length][length];
    for (int i = 0; i < length; i++) {
        for (int j = 0; j <= i; j++) {
            if (s.charAt(i) == s.charAt(j) && (i - j <= 2 || dp[j + 1][i - 1])) {
                dp[j][i] = true;
            }
        }
    }

```

不懂的可以看下[《540，动态规划和中心扩散法解回文子串》]，这里就不在重复介绍



再来看下最终代码

```java
public List<List<String>> partition(String s) {
    //最终要返回的结果
    List<List<String>> res = new ArrayList<>();
    int length = s.length();

    //下面先计算子串中哪些是回文的，哪些不是
    //数组dp[i][j]表示字符串s下标[i,j]的子串是否是回文的
    boolean[][] dp = new boolean[length][length];
    for (int i = 0; i < length; i++) {
        for (int j = 0; j <= i; j++) {
            if (s.charAt(i) == s.charAt(j) && (i - j <= 2 || dp[j + 1][i - 1])) {
                dp[j][i] = true;
            }
        }
    }
    backTrack(s, dp, 0, res, new ArrayList<>());
    return res;
}

public void backTrack(String s, boolean[][] dp, int index, List<List<String>> res, List<String> cur
    //边界条件判断，如果字符串s中的字符都访问完了（类似于到叶子节点了），就停止查找，
    //然后这个分支的所有元素加入到集合res中
    if (index >= s.length()) {
        res.add(new ArrayList<>(cur));
        return;
    }
    for (int i = index; i < s.length(); i++) {
        //如果当前截取的子串不是回文的，就跳过
        if (!dp[index][i])
            continue;
        //做出选择
        cur.add(s.substring(index, i + 1));
        //递归
        backTrack(s, dp, i + 1, res, cur);
        //撤销选择
        cur.remove(cur.size() - 1);
    }
}
```

总结

回溯算法其实是有一个经典的模板的，掌握这个模板，很多关于回溯算法的题套用模板然后稍加修改基本上都能解决，关于模板可以看下《450，什么叫回溯算法，一看就会，一写就废》，总结一下就是3步

- 做出选择
- 递归到下一层
- 撤销选择

往期推荐

- 540，动态规划和中心扩散法解回文子串
- 529，动态规划解最长回文子序列
- 517，最长回文子串的3种解决方式
- 497，双指针验证回文串

537，剑指 Offer-字符串的排列

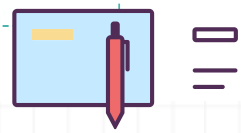
原创 博哥 数据结构和算法 5天前

收录于话题
#剑指offer

32个 >

Contentment is natural wealth, luxury is artificial poverty.

知足是天赋的财富，奢侈是人为的贫穷。



问题描述

输入一个字符串，打印出该字符串中字符的所有排列。

你可以以任意顺序返回这个字符串数组，但里面不能有重复元素。

示例:

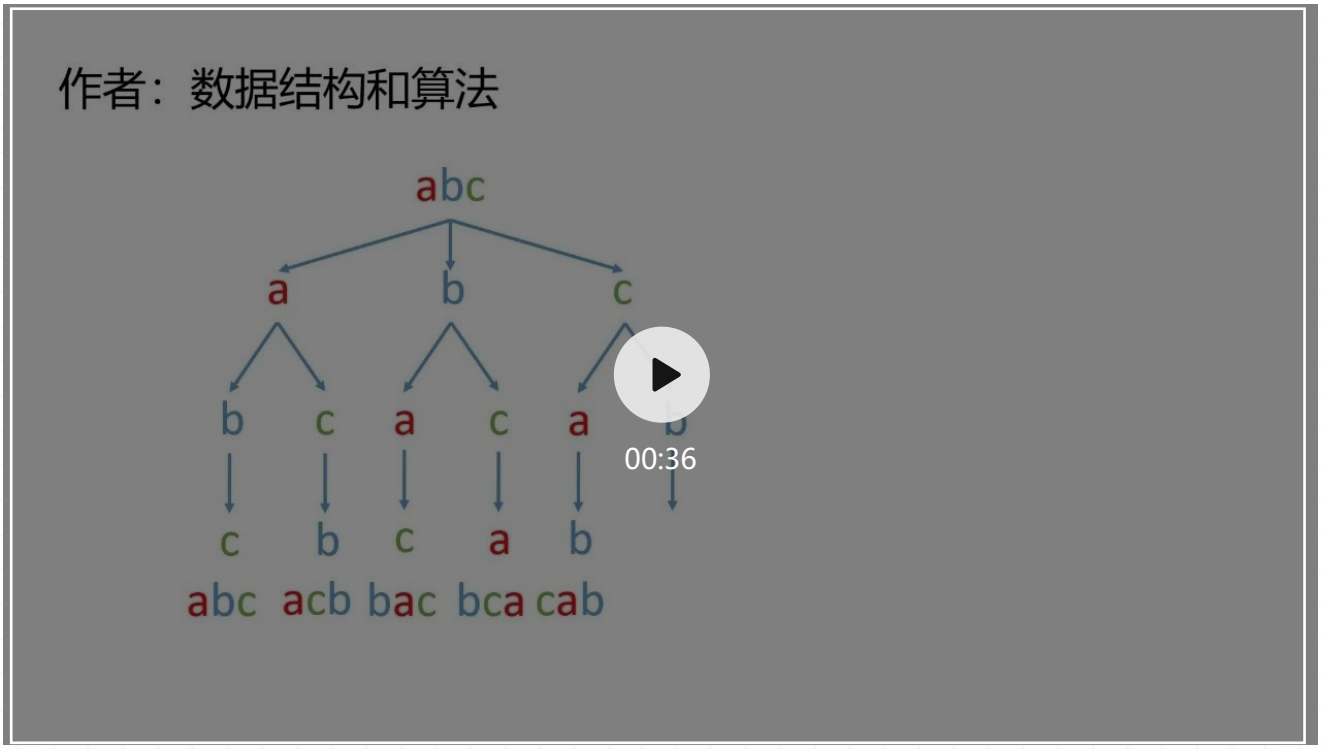
> **输入**：s = "abc"
> **输出**：
> ["abc","acb","bac","bca","cab","cba"]

限制：

- 1 <= s 的长度 <= 8

回溯算法解决

字符串的排列其实就是排列组合，我们可以把它想象成为一棵n叉树（n是s的长度），然后每一个节点都要从字符串中选择一个字符，但注意不能选择重复的，比如在一个节点选择了a，那么他的子孙节点都不能再选择a了，我们来看个视频



看到这里我们很容易想到的一种解决方式就是回溯，具体可以看下《450，什么叫回溯算法，一看就会，一写就废》，之前我们总结回溯算法的时候有一个经典的模板

```
private void backtrack("原始参数") {
  //终止条件(递归必须要有终止条件)
  if ("终止条件") {
    //一些逻辑操作（可有可无，视情况而定）
    return;
  }

  for (int i = "for循环开始的参数"; i < "for循环结束的参数"; i++) {
    //一些逻辑操作（可有可无，视情况而定）

    //做出选择

    //递归
    backtrack("新的参数");
    //一些逻辑操作（可有可无，视情况而定）

    //撤销选择
  }
}
```

这里只需要按照这个模板对他稍作修改即可，代码如下

```
public String[] permutation(String s) {
  Set<String> res = new HashSet<>();
  backtrack(s.toCharArray(), "", new boolean[s.length()], res);
  return res.toArray(new String[res.size()]);
}

private void backtrack(char[] chars, String temp, boolean[] visited, Set<String> res) {
  //边界条件判断，当选择的字符长度等于原字符串长度的时候，说明原字符串的字符都已经
  //选完了
  if (temp.length() == chars.length) {
    res.add(temp);
    return;
  }
  //每一个节点我们都要从头开始选
  for (int i = 0; i < chars.length; i++) {
    //已经选择过的就不能再选了
    if (visited[i])
      continue;
    //表示选择当前字符
```

```
        visited[i] = true;
        //把当前字符选择后，到树的下一层继续选
        backtrack(chars, temp + chars[i], visited, res);
        //递归往回走的时候要撤销选择
        visited[i] = false;
    }
}
```

总结

这里字符串中可能有重复的字符，所以结果中也会出现重复的数据，但上面使用了Set集合去重，保证了最终的排列不会出现重复的。其实还有一种方式就是先对字符串中的字符（先转化为字符数组）进行排序，然后在计算，这个就是常见的有重复数字的全排列，后续会在讲。

往期推荐

- 520，回溯算法解火柴拼正方形
- 498，回溯算法解活字印刷
- 491，回溯算法解将数组拆分成斐波那契序列
- 450，什么叫回溯算法，一看就会，一写就废

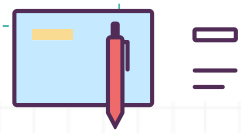
520，回溯算法解火柴拼正方形

原创 博哥 数据结构和算法 1周前

收录于话题
#算法图文分析

137个 >

> Only those that risk going too far can possibly know how far they can go.
>
> 只有勇于承担风险，敢于走出去的人，才会明白他们到底能走多远。



问题描述

还记得童话《卖火柴的小女孩》吗？现在，你知道小女孩有多少根火柴，请找出一种能使用所有火柴拼成一个正方形的方法。不能折断火柴，可以把火柴连接起来，并且每根火柴都要用到。

输入为小女孩拥有火柴的数目，每根火柴用其长度表示。输出即为是否能用所有的火柴拼成正方形。

示例 1:

> **输入**: [1,1,2,2,2]
> **输出**: true
>
> **解释**: 能拼成一个边长为2的正方形，每边两根火柴。

示例 2:

> **输入**: [3,3,3,3,4]
> **输出**: false
>
> **解释**: 不能用所有火柴拼成一个正方形。

注意:

1. 给定的火柴长度和在 0 到 10^9之间。
2. 火柴数组的长度不超过15。

回溯算法解决

这题是让所有的火柴能不能拼接成一个正方形，不能折断火柴，并且所有的火柴都要用到。

首先先求出所有火柴的长度，然后判断是否是4的倍数，如果不是，直接返回false，表示不能拼接成正方形，如果是4的倍数然后在往下走。

把每一根火柴都尝试往4条边上放，如果最终能构成正方形，直接返回true。看一下视频



这就是回溯算法，他不断的尝试，一旦成功也就成功了。如果不成功，在回到上一步继续尝试，其实他有一个经典的模板

```
private void backtrack("原始参数") {
  //终止条件(递归必须要有终止条件)
  if ("终止条件") {
    //一些逻辑操作（可有可无，视情况而定）
    return;
  }

  for (int i = "for循环开始的参数"; i < "for循环结束的参数"; i++) {
    //一些逻辑操作（可有可无，视情况而定）

    //做出选择

    //递归
    backtrack("新的参数");
    //一些逻辑操作（可有可无，视情况而定）
```

```
16
17          //撤销选择
18      }
19  }
```

我们来看下最终代码

```
1   public boolean makesquare(int[] nums) {
2       int total = 0;
3       //统计所有火柴的长度
4       for (int num : nums) {
5           total += num;
6       }
7       //如果所有火柴的长度不是4的倍数，直接返回false
8       if (total == 0 || (total & 3) != 0)
9           return false;
10      //回溯
11      return backtrack(nums, 0, total >> 2, new int[4]);
12  }
13
14  //index表示访问到当前火柴的位置，target表示正方形的边长，size是长度为4的数组，
15  //分别保存正方形4个边的长度
16  private boolean backtrack(int[] nums, int index, int target, int[] size) {
17      if (index == nums.length) {
18          //如果火柴都访问完了，并且size的4个边的长度都相等，说明是正方形，直接返回true，
19          //否则返回false
20          if (size[0] == size[1] && size[1] == size[2] && size[2] == size[3])
21              return true;
22          return false;
23      }
24      //到这一步说明火柴还没访问完
25      for (int i = 0; i < size.length; i++) {
26          //如果把当前火柴放到size[i]这个边上，他的长度大于target，我们直接跳过
27          if (size[i] + nums[index] > target)
28              continue;
29          //如果当前火柴放到size[i]这个边上，长度不大于target，我们就放上面
30          size[i] += nums[index];
31          //然后在放下一个火柴，如果最终能变成正方形，直接返回true
32          if (backtrack(nums, index + 1, target, size))
33              return true;
34          //如果当前火柴放到size[i]这个边上，最终不能构成正方形，我们就把他从
35          //size[i]这个边上给移除，然后在试其他的边
36          size[i] -= nums[index];
37      }
38      //如果不能构成正方形，直接返回false
39      return false;
40  }
```

代码优化

如果数组前面数组比较小，这会导致递归的比较深，所以我们可以先对数组进行排序，从大的开始递归，代码如下

```
1   public boolean makesquare(int[] nums) {
2       int total = 0;
3       //统计所有火柴的长度
4       for (int num : nums) {
5           total += num;
6       }
7       //如果所有火柴的长度不是4的倍数，直接返回false
8       if (total == 0 || (total & 3) != 0)
9           return false;
10      //先排序
11      Arrays.sort(nums);
12      //回溯，从最长的火柴开始
13      return backtrack(nums, nums.length - 1, total >> 2, new int[4]);
```

```
}

//index表示访问到当前火柴的位置，target表示正方形的边长，size是长度为4的数组，
//分别保存正方形4个边的长度
private boolean backtrack(int[] nums, int index, int target, int[] size) {
    if (index == -1) {
        //如果火柴都访问完了，并且size的4个边的长度都相等，说明是正方形，直接返回true，
        //否则返回false
        if (size[0] == size[1] && size[1] == size[2] && size[2] == size[3])
            return true;
        return false;
    }
    //到这一步说明火柴还没访问完
    for (int i = 0; i < size.length; i++) {
        //如果把当前火柴放到size[i]这个边上，他的长度大于target，我们直接跳过。或者
        // size[i] == size[i - 1]即上一个分支的值和当前分支的一样，上一个分支没有成功，
        //说明这个分支也不会成功，直接跳过即可。
        if (size[i] + nums[index] > target || (i > 0 && size[i] == size[i - 1]))
            continue;
        //如果当前火柴放到size[i]这个边上，长度不大于target，我们就放上面
        size[i] += nums[index];
        //然后在放下一个火柴，如果最终能变成正方形，直接返回true
        if (backtrack(nums, index - 1, target, size))
            return true;
        //如果当前火柴放到size[i]这个边上，最终不能构成正方形，我们就把他从
        //size[i]这个边上给移除，然后在试其他的边
        size[i] -= nums[index];
    }
    //如果不能构成正方形，直接返回false
    return false;
}
```

总结

回溯算法还是有一个经典模板的，他的原理就是不断尝试的过程，一旦尝试失败就会回到上一步继续尝试，上面代码中都有详细的注释，很容易理解。

往期推荐

- 498，回溯算法解活字印刷
- 491，回溯算法解将数组拆分成斐波那契序列
- 450，什么叫回溯算法，一看就会，一写就废
- 446，回溯算法解黄金矿工问题

498，回溯算法解活字印刷

原创 山大王wld 数据结构和算法 5天前

收录于话题
#算法图文分析

111个 >



If you lose your purpose, it's like you're broken.

如果你生活漫无目的，那就好像你坏了。

问题描述

你有一套活字字模tiles，其中每个字模上都刻有一个字母tiles[i]。返回你可以印出的非空字母序列的数目。

注意：本题中，每个活字字模只能使用一次。

示例 1：

> **输入**：“AAB”
> **输出**：8
> **解释**：可能的序列为 “A”, “B”, “AA”, “AB”, “BA”, “AAB”, “ABA”, “BAA”。

示例 2：

> **输入**：“AAABBC”
> **输出**：188

提示：

1. `1 <= tiles.length <= 7`
2. `tiles` 由大写英文字母组成

回溯算法解决

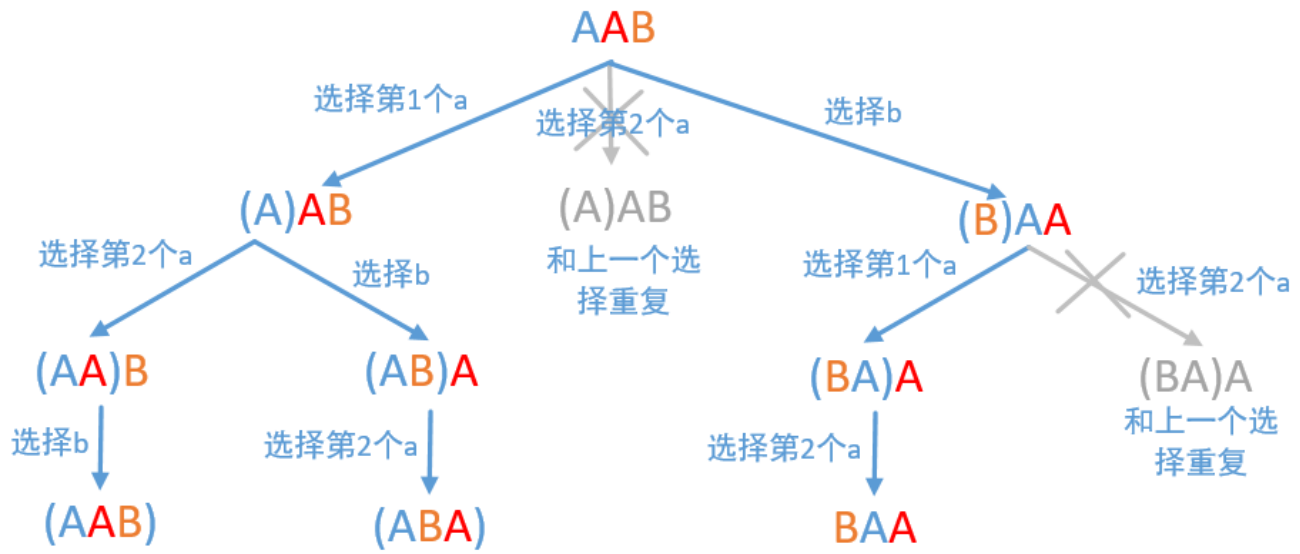
这题实际上是让求输入的字符可以组成多少种不同的组合。之前也讲过很多种类似这样的题

391，回溯算法求组合问题

448，组合的几种解决方式

451，回溯和位运算解子集

但不同的是前面几道题字符出现的顺序是不会变的，比如第451题，他的子集有123，但不能有321。但这题不一样，输入AAB，他的序列中，A可以在B的前面也可以在B的后面。那么这道题使用回溯算法该怎么解，我们就以示例1为例来画个图看一下



总共有[A，B，AA，AB，BA，AAB，ABA，BAA]共8种

前面介绍的几个组合的回溯算法，因为结果不能有重复的（比如[1，3]和[3，1]被认为是重复的结果），所以每次选择的时候都只能从前往后选。但这题中子集[A，B]和[B，A]被认为是两种不同的结果，所以每次都要从头开始选择，因为每个字符只能被使用一次，所以如果使用之后下次就不能再使用了，这里可以使用一个数组visit来标记有没有被使用。

但这里有个难点就是怎么过滤掉上面图中灰色的部分（也就是重复的部分）。举个例子，比如ABBCD，如果我们选择了第1个B，那么剩余的字符就变成了ABCD，这个时

候我们再选择第2个B是可以的。但如果我们没选择第1个B，直接选择第2个B，那么剩余的字符就是ABCD，和上面重复了。所以代码大致是这样的

```
if (i - 1 >= 0 && chars[i] == chars[i - 1] && !used[i - 1])
        continue;
```

在前面讲450，什么叫回溯算法，一看就会，一写就废的时候，回溯算法的模板也被多次提及

```
private void backtrack("原始参数") {
  //终止条件(递归必须要有终止条件)
  if ("终止条件") {
    //一些逻辑操作（可有可无，视情况而定）
    return;
  }

  for (int i = "for循环开始的参数"; i < "for循环结束的参数"; i++) {
    //一些逻辑操作（可有可无，视情况而定）

    //做出选择

    //递归
    backtrack("新的参数");
    //一些逻辑操作（可有可无，视情况而定）

    //撤销选择
  }
}
```

最后来看下这题的最终代码

```
public int numTilePossibilities(String tiles) {
  char[] chars = tiles.toCharArray();
  //先排序，目的是让相同的字符挨着，在下面计算的时候好过滤掉重复的
  Arrays.sort(chars);
  int[] res = new int[1];
  backtrack(res, chars, new boolean[tiles.length()], tiles.length(), 0);
  return res[0];
}

private void backtrack(int[] res, char[] chars, boolean[] used, int length, int index) {
  //如果没有可以选择的就返回
  if (index == length)
    return;
  //注意，这里的i每次都是从0开始的，不是从index开始
  for (int i = 0; i < length; i++) {
    //一个字符只能选择一次，如果当前字符已经选择了，就不能再选了。
    if (used[i])
      continue;
    //过滤掉重复的结果
    if (i - 1 >= 0 && chars[i] == chars[i - 1] && !used[i - 1])
      continue;
    //选择当前字符，并把它标记为已选择
    used[i] = true;
    res[0]++;//选择一个字符，就多了一种结果
    //下一分支继续递归
    backtrack(res, chars, used, length, index + 1);
    //使用完之后再把它给复原。
    used[i] = false;
  }
}
```

这里还可以换一种写法，先统计每个字符的数量，然后再使用，代码如下。原理都类似，但他不需要去重，因为这里根本不可能出现重复的。

```java
public int numTilePossibilities(String tiles) {
    char[] chars = tiles.toCharArray();
    //统计每个字符的数量
    int[] count = new int[26];
    for (char c : chars)
        count[c - 'A']++;
    int[] res = new int[1];
    backtrack(res, count);
    return res[0];
}

private void backtrack(int[] res, int[] arr) {
    //遍历所有的字符
    for (int i = 0; i < 26; i++) {
        //如果当前字符使用完了再查找下一个
        if (arr[i] == 0)
            continue;
        //如果没使用完就继续使用，然后把这个字符的数量减1
        arr[i]--;
        //使用一个字符，子集数量就会多一个
        res[0]++;
        backtrack(res, arr);
        //当前字符使用完之后，把它的数量还原
        arr[i]++;
    }
}
```

总结

前面也介绍过很多关于回溯算法的题，回溯算法有个大致的模板，只要掌握这个模板，然后对于不同的题在稍加修改，基本上都能做的出来。

往期推荐

- 450，什么叫回溯算法，一看就会，一写就废
- 446，回溯算法解黄金矿工问题
- 442，剑指 Offer-回溯算法解二叉树中和为某一值的路径
- 420，剑指 Offer-回溯算法解矩阵中的路径

491，回溯算法解将数组拆分成斐波那契序列

原创 山大王wld 数据结构和算法 2020-12-15

收录于话题
#算法图文分析

111个 >



Sometimes life hits you in the head with a brick. Don't lose faith.

有时生活给你当头痛击，但是别丧失信念。

问题描述

给定一个数字字符串S，比如S = "123456579"，我们可以将它分成斐波那契式的序列[123, 456, 579]。

形式上，斐波那契式序列是一个非负整数列表 F，且满足：

- 0 <= F[i] <= 2^31 - 1，（也就是说，每个整数都符合32位有符号整数类型）；
- F.length >= 3；
- 对于所有的0 <= i < F.length - 2，都有F[i] + F[i + 1] = F[i + 2]成立。

另外，请注意，将字符串拆分成小块时，每个块的数字一定不要以零开头，除非这个块是数字 0 本身。

返回从 S 拆分出来的任意一组斐波那契式的序列块，如果不能拆分则返回 []。

示例 1：

> **输入**: "123456579"
> **输出**: [123,456,579]

示例 2:

> **输入**: "11235813"
> **输出**: [1,1,2,3,5,8,13]

示例 3:

> **输入**: "112358130"
> **输出**: []
> **解释**: 这项任务无法完成。

示例 4:

> **输入**: "0123"
> **输出**: []
> **解释**: 每个块的数字不能以零开头，因此 "01"，"2"，"3" 不是有效答案。

示例 5:

> **输入**: "1101111"
> **输出**: [110, 1, 111]
> **解释**: 输出 [11,0,11,11] 也同样被接受。

提示:

1. 1<=S.length<= 200
2. 字符串S中只含有数字。

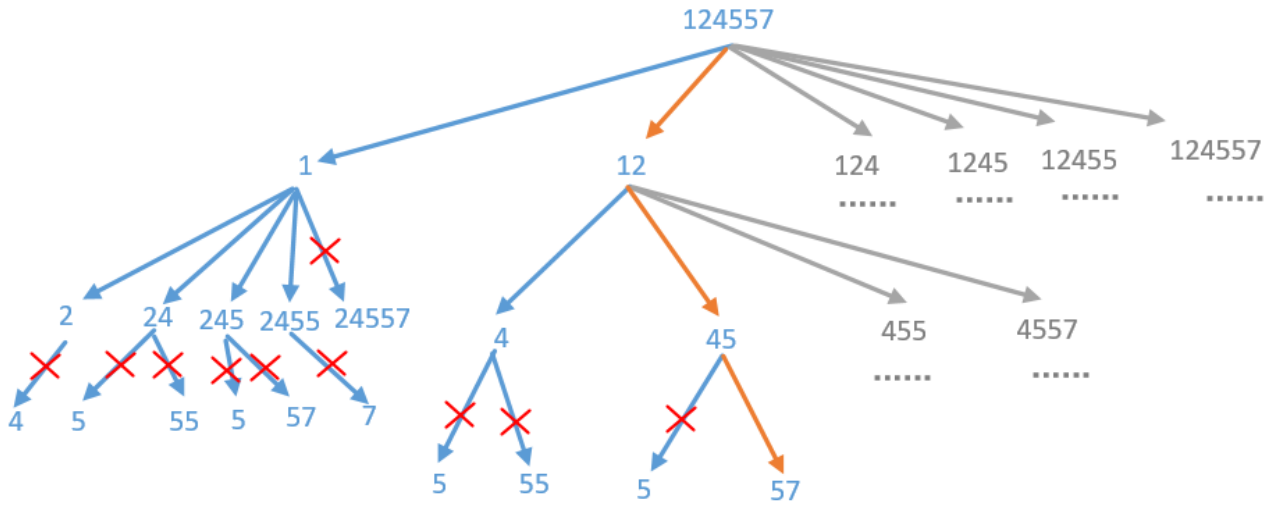
回溯算法解决

这题是让把字符串S拆成一些子串，并且这些子串满足斐波那契数列的关系式。对于这道题我们可以使用回溯算法来解决，回溯算法其实就是不断尝试的过程，一旦尝试成功了，就算成功了，如果尝试失败了还会回到上一步，注意回到上一步的时候还要把状态还原到上一步的状态。回溯算法这里就不在过多介绍，关于回溯算法的解题思路可以看下450，什么叫回溯算法，一看就会，一写就废。

回溯算法其实有一个经典的模板

```
private void backtrack("原始参数") {
    //终止条件(递归必须要有终止条件)
    if ("终止条件") {
        //一些逻辑操作（可有可无，视情况而定）
        return;
    }

    for (int i = "for循环开始的参数"; i < "for循环结束的参数"; i++) {
        //一些逻辑操作（可有可无，视情况而定）

        //做出选择

        //递归
        backtrack("新的参数");
        //一些逻辑操作（可有可无，视情况而定）

        //撤销选择
    }
}
```

对于这道题也一样，我们先把字符串不断的截取，看一下能不能构成斐波那契序列，如果不能就回到上一步，如果能就继续往下走，具体我们看下下面的图，这里是参照示例1为例画的一个图，只不过数字缩短了，只有124557，因为如果数字比较多的话，图太大，画不下。



搞懂了上面的原理，代码就简单多了，我们来看下代码

```
public List<Integer> splitIntoFibonacci(String S) {
    List<Integer> res = new ArrayList<>();
    backtrack(S.toCharArray(), res, 0);
    return res;
```

```
    }

    public boolean backtrack(char[] digit, List<Integer> res, int index) {
        //边界条件判断，如果截取完了，并且res长度大于等于3，表示找到了一个组合。
        if (index == digit.length && res.size() >= 3) {
            return true;
        }
        for (int i = index; i < digit.length; i++) {
            //两位以上的数字不能以0开头
            if (digit[index] == '0' && i > index) {
                break;
            }
            //截取字符串转化为数字
            long num = subDigit(digit, index, i + 1);
            //如果截取的数字大于int的最大值，则终止截取
            if (num > Integer.MAX_VALUE) {
                break;
            }
            int size = res.size();
            //如果截取的数字大于res中前两个数字的和，说明这次截取的太大，直接终止，因为后面越截取越大
            if (size >= 2 && num > res.get(size - 1) + res.get(size - 2)) {
                break;
            }
            if (size <= 1 || num == res.get(size - 1) + res.get(size - 2)) {
                //把数字num添加到集合res中
                res.add((int) num);
                //如果找到了就直接返回
                if (backtrack(digit, res, i + 1))
                    return true;
                //如果没找到，就会走回溯这一步，然后把上一步添加到集合res中的数字给移除掉
                res.remove(res.size() - 1);
            }
        }
        return false;
    }

    //相当于截取字符串S中的子串然后转换为十进制数字
    private long subDigit(char[] digit, int start, int end) {
        long res = 0;
        for (int i = start; i < end; i++) {
            res = res * 10 + digit[i] - '0';
        }
        return res;
    }
```

总结

真正有模板的算法题型其实不多，但回溯算法算是其中的一个，只不过对于不同的题型要做不同的修改，只要掌握了这个模板，对于很多回溯算法题型稍加修改，我们就很容易做出来。

往期推荐

- 478，回溯算法解单词搜索
- 451，回溯和位运算解子集
- 450，什么叫回溯算法，一看就会，一写就废
- 446，回溯算法解黄金矿工问题

478，回溯算法解单词搜索

原创 山大王wld 数据结构和算法 11月17日

收录于话题
#算法图文分析

95个 >



Memory is a wonderful thing if you don't have to deal with the past.

如果你不必纠缠过去，那么回忆是一件非常美好的事情。

问题描述

给定一个二维网格和一个单词，找出该单词是否存在于网格中。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

示例：

```
board =
[
  ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']
]
```

> 给定 word = "**ABCCED**", 返回 **true**
> 给定 word = "**SEE**", 返回 **true**
> 给定 word = "**ABCB**", 返回 **false**

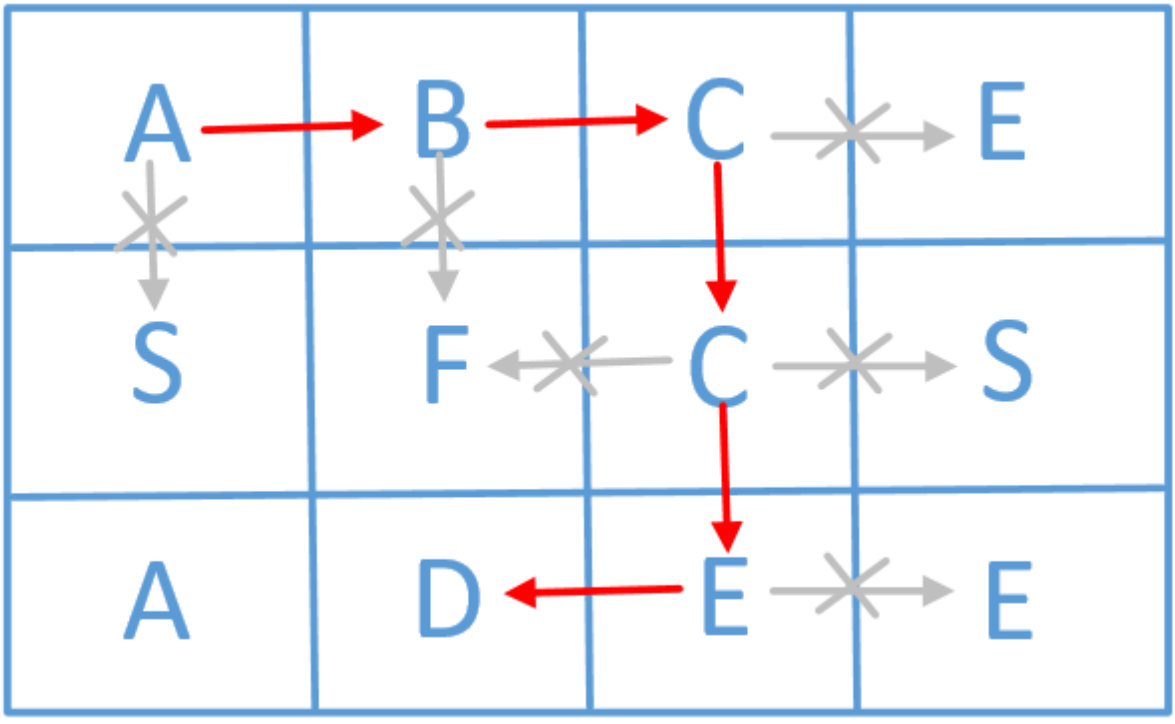
提示：

- board 和 word 中只包含大写和小写英文字母。
- 1 <= board.length <= 200
- 1 <= board[i].length <= 200
- 1 <= word.length <= 10^3

回溯算法解决

这题是让判断给定的单词是否在二维网格中，所以最简单的一种方式就是使用dfs，沿着一个点往他的4个方向判断，如果最后能找到给定的单词就返回true，否则就返回false。

回溯算法实际上就是一个类似枚举的搜索尝试过程，也就是一个个去试，我们解这道题也是通过一个个去试，下面就用示例1来画个图看一下



他是从矩形中的一个点开始往他的上下左右四个方向查找，这个点可以是矩形中的任何一个点，所以代码的大致轮廓我们应该能写出来，就是遍历矩形所有的点，然后从这个点开始往他的4个方向走，因为是二维数组，所以有两个for循环，代码如下

```
1   public boolean exist(char[][] board, String word) {
2     char[] words = word.toCharArray();
3
4     //下面两个for循环，来遍历数组的每一个值
```

```
    for (int i = 0; i < board.length; i++) {
      for (int j = 0; j < board[0].length; j++) {
        //从[i,j]这个坐标开始查找,如果能查找到，直接
        // 返回true，后面就不需要再查找了
        if (dfs(board, words, i, j, 0))
          return true;
      }
    }
    return false;
  }
```

这里关键代码是dfs这个函数，因为每一个点都可以往他的4个方向查找，所以我们可以把它想象为一棵4叉树，就是每个节点有4个子节点，而树的遍历我们最容易想到的就是递归，我们来大概看一下

```
  boolean dfs(char[][] board, char[] word, int i, int j, int index) {
    if (边界条件的判断) {
      return;
    }

    一些逻辑处理

    boolean res;
    //往右
    res = dfs(board, word, i + 1, j, index + 1)
    //往左
    res |= dfs(board, word, i - 1, j, index + 1)
    //往下
    res |= dfs(board, word, i, j + 1, index + 1)
    //往上
    res |= dfs(board, word, i, j - 1, index + 1)
    //上面4个方向，只要有一个能查找到，就返回true；
    return res;
  }
```

最终的完整代码如下

```
  public boolean exist(char[][] board, String word) {
    char[] words = word.toCharArray();
    for (int i = 0; i < board.length; i++) {
      for (int j = 0; j < board[0].length; j++) {
        //从[i,j]这个坐标开始查找
        if (dfs(board, words, i, j, 0))
          return true;
      }
    }
    return false;
  }

  boolean dfs(char[][] board, char[] word, int i, int j, int index) {
    //边界的判断，如果越界直接返回false。index表示的是查找到字符串word的第几个字符，
    //如果这个字符不等于board[i][j]，说明验证这个坐标路径是走不通的，直接返回false
    if (i >= board.length || i < 0 || j >= board[0].length || j < 0 || board[i][j] != word[index])
      return false;
    //如果word的每个字符都查找完了，直接返回true
    if (index == word.length - 1)
      return true;
    //把当前坐标的值保存下来，为了在最后复原
    char tmp = board[i][j];
    //然后修改当前坐标的值
    board[i][j] = '.';
    //走递归，沿着当前坐标的上下左右4个方向查找
    boolean res = dfs(board, word, i + 1, j, index + 1)
        || dfs(board, word, i - 1, j, index + 1)
        || dfs(board, word, i, j + 1, index + 1)
        || dfs(board, word, i, j - 1, index + 1);
    //递归之后再把当前的坐标复原
    board[i][j] = tmp;
```

```
32      return res;
33  }
```

总结

要想弄懂这题，首先要搞懂回溯算法，要想弄懂回溯算法，就要先要搞懂递归。关于递归和回溯算法之前有过详细介绍，可以看下

450，什么叫回溯算法，一看就会，一写就废

426，什么是递归，通过这篇文章，让你彻底搞懂递归

往期推荐

- 451，回溯和位运算解子集
- 446，回溯算法解黄金矿工问题
- 420，剑指 Offer-回溯算法解矩阵中的路径
- 391，回溯算法求组合问题

451，回溯和位运算解子集

原创 山大王wld 数据结构和算法 9月16日

收录于话题
#算法图文分析

95个 >



Whatever tomorrow brings, I'm grateful to see it.

不管明天将带来什么，我都会感激。

问题描述

给定一组**不含重复元素**的整数数组 *nums*，返回该数组所有可能的子集（幂集）。
说明： 解集不能包含重复的子集。

示例:

```
输入: nums = [1,2,3]

输出:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
```

```
]
```

回溯解决解决

在前面一道题[450，什么叫回溯算法，一看就会，一写就废](#)中提到过子集的问题，这里再来看一下，回溯的模板如下，就是先选择，最后再撤销

```
private void backtrack("原始参数") {
  //终止条件(递归必须要有终止条件)
  if ("终止条件") {
    //一些逻辑操作（可有可无，视情况而定）
    return;
  }

  for (int i = "for循环开始的参数"; i < "for循环结束的参数"; i++) {
    //一些逻辑操作（可有可无，视情况而定）

    //做出选择

    //递归
    backtrack("新的参数");
    //一些逻辑操作（可有可无，视情况而定）

    //撤销选择
  }
}
```

这道题也一样，可以把它想象成为一颗n叉树，通过DFS遍历这棵n叉树，他所走过的所有路径都是子集的一部分，看下代码

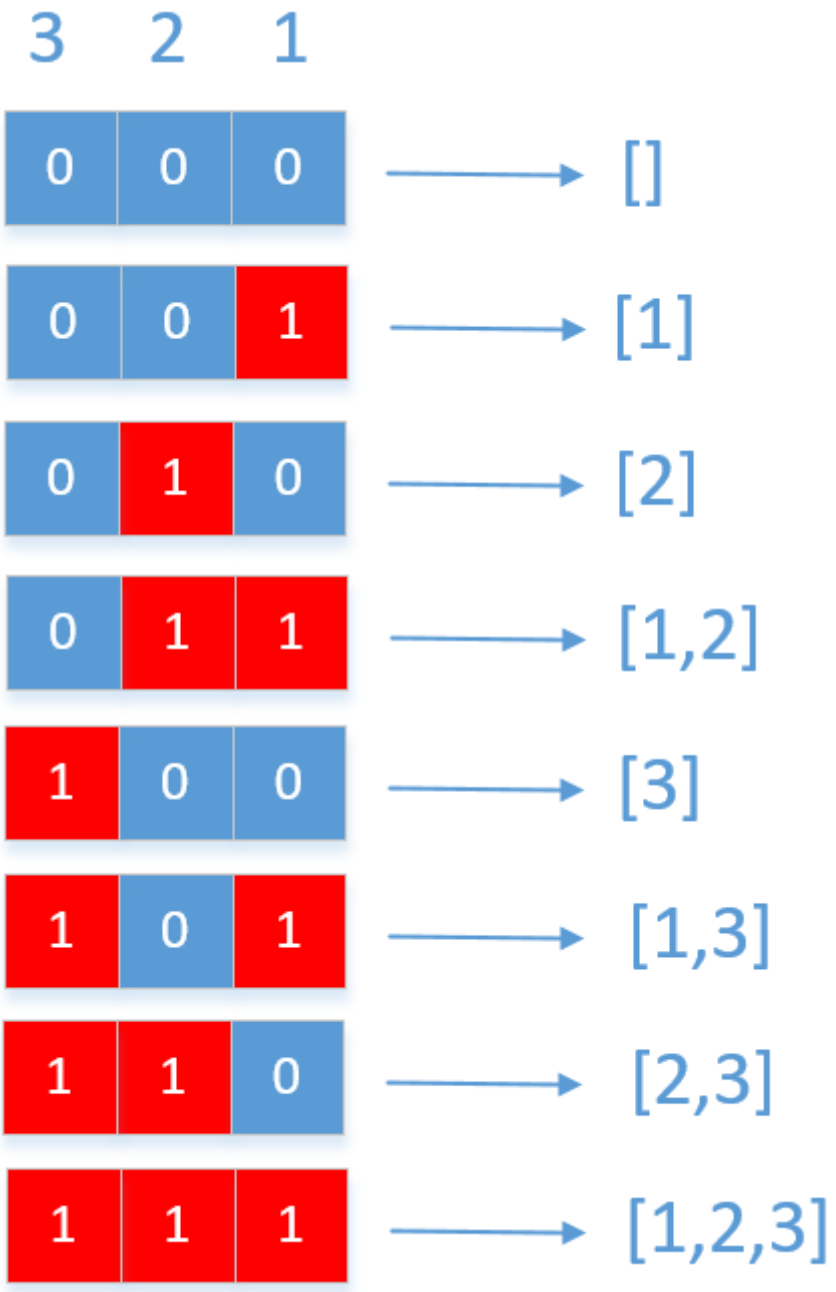
```
public List<List<Integer>> subsets(int[] nums) {
  List<List<Integer>> list = new ArrayList<>();
  backtrack(list, new ArrayList<>(), nums, 0);
  return list;
}

private void backtrack(List<List<Integer>> list, List<Integer> tempList, int[] nums, int start) {
  //走过的所有路径都是子集的一部分，所以都要加入到集合中
  list.add(new ArrayList<>(tempList));
  for (int i = start; i < nums.length; i++) {
    //做出选择
    tempList.add(nums[i]);
    //递归
    backtrack(list, tempList, nums, i + 1);
    //撤销选择
    tempList.remove(tempList.size() - 1);
  }
}
```

因为在第450题刚讲过这道题，所以基本上没什么难度，其实这道题还可以使用位运算解决，来看下

位运算解决

数组中的每一个数字都有选和不选两种状态，我们可以用0和1表示，0表示不选，1表示选择。如果数组的长度是n，那么子集的数量就是2^n。比如数组长度是3，就有8种可能，分别是

[0, 0, 0]
[0, 0, 1]
[0, 1, 0]
[0, 1, 1]
[1, 0, 0]
[1, 0, 1]
[1, 1, 0]
[1, 1, 1]
这里参照示例画个图来看下




```java
public static List<List<Integer>> subsets(int[] nums) {
    //子集的长度是2的nums.length次方，这里通过移位计算
    int length = 1 << nums.length;
    List<List<Integer>> res = new ArrayList<>(length);
    //遍历从0到length中间的所有数字，根据数字中1的位置来找子集
    for (int i = 0; i < length; i++) {
        List<Integer> list = new ArrayList<>();
```

```
        for (int j = 0; j < nums.length; j++) {
            //如果数字i的某一个位置是1，就把数组中对
            //应的数字添加到集合
            if (((i >> j) & 1) == 1)
                list.add(nums[j]);
        }
        res.add(list);
    }
    return res;
}
```

非递归解决

这题还有其他解题思路，比如先加入一个空集让他成为新的子集，然后每遍历一个元素就在原来的子集的后面追加这个值。还以示例来分析下

第1步：添加一个空集，结果是[]

第2步：访问数字1，对原来的子集追加1，结果是：[]，[1]

第3步：访问数字2，对原来的子集追加2，结果是：[]，[1]，[2]，[1，2]

第4步：访问数字3，对原来的子集追加3，结果是：
[]，[1]，[2]，[1，2]，[3]，[1，3]，[2，3]，[1，2，3]

```
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> res = new ArrayList<>(1 << nums.length);
    //先添加一个空的集合
    res.add(new ArrayList<>());
    for (int num : nums) {
        //每遍历一个元素就在之前子集中的每个集合追加这个元素，让他变成新的子集
        for (int i = 0, j = res.size(); i < j; i++) {
            //遍历之前的子集，重新封装成一个新的子集
            List<Integer> list = new ArrayList<>(res.get(i));
            //然后在新的子集后面追加这个元素
            list.add(num);
            //把这个新的子集添加到集合中
            res.add(list);
        }
    }
    return res;
}
```

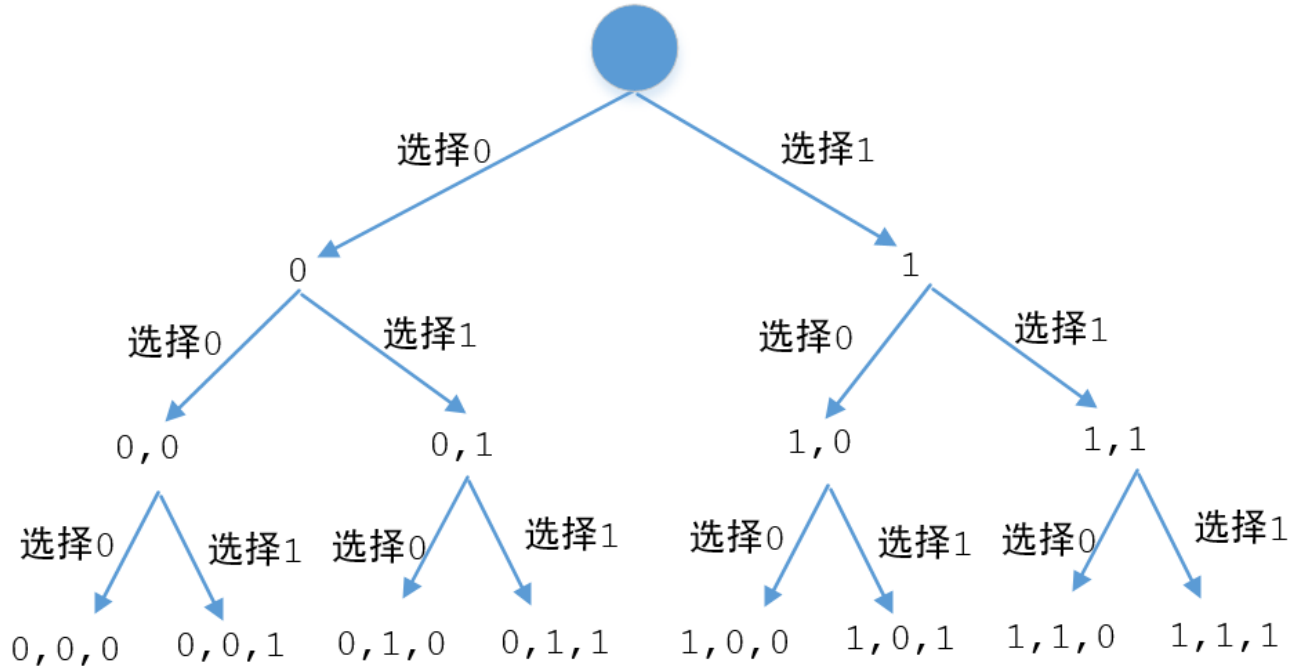
如果非要把它改为递归的也是可以的，仅仅提供了一种思路，有兴趣的也可以看下

```
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> res = new ArrayList<>(1 << nums.length);
    res.add(new ArrayList<>());
    recursion(nums, 0, res);
    return res;
}

public static void recursion(int[] nums, int index, List<List<Integer>> res) {
    //数组中的元素都访问完了，直接return
    if (index >= nums.length)
        return;
    int size = res.size();
    for (int j = 0; j < size; j++) {
        List<Integer> list = new ArrayList<>(res.get(j));
```

```
        //然后在新的子集后面追加一个值
        list.add(nums[index]);
        res.add(list);
    }
    //递归下一个元素
    recursion(nums, index + 1, res);
}
```

其他解决方式

在426，什么是递归，通过这篇文章，让你彻底搞懂递归中最后讲到分支污染的时候提到过这样一个问题：生成一个2^n长的数组，数组的值从0到(2^n)-1。我们可以把它想象成为一颗二叉树，每个节点的子树都是一个可选一个不可选



所以我们也可以参照这种方式来写，代码如下

```
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> res = new ArrayList<>();
    helper(res, nums, new ArrayList<>(), 0);
    return res;
}

private void helper(List<List<Integer>> res, int[] nums, List<Integer> list, int index) {
    //终止条件判断
    if (index == nums.length) {
        res.add(new ArrayList<>(list));
        return;
    }
    //每一个节点都有两个分支，一个选一个不选
    //走不选这个分支
    helper(res, nums, list, index + 1);
    //走选择这个分支
    list.add(nums[index]);
    helper(res, nums, list, index + 1);
    //撤销选择
    list.remove(list.size() - 1);
}
```

总结

这题难度不大，但解法比较多，上面介绍的每一种基本上都是一种新的解题思路，如果能全部掌握将会有很大收获。

往期推荐

- 450，什么叫回溯算法，一看就会，一写就废
- 446，回溯算法解黄金矿工问题
- 442，剑指 Offer-回溯算法解二叉树中和为某一值的路径
- 391，回溯算法求组合问题

450，什么叫回溯算法，一看就会，一写就废

原创 山大王wld 数据结构和算法 9月14日

收录于话题
#算法图文分析
95个 >



Don't ever let somebody tell you you can't do something.

别让他人告诉你你不行。

什么叫回溯算法

对于回溯算法的定义，百度百科上是这样描述的：回溯算法实际上一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。许多复杂的，规模较大的问题都可以使用回溯法，有“通用解题方法”的美称。

看明白没，回溯算法其实就是一个不断探索尝试的过程，探索成功了也就成功了，探索失败了就在退一步，继续尝试……，

组合总和

组合总和算是一道比较经典的回溯算法题，其中有一道题是这样的。

找出所有相加之和为 n 的 k 个数的组合。组合中只允许含有 1 - 9 的正整数，并且每种组合中不存在重复的数字。

说明：

- 所有数字都是正整数。
- 解集不能包含重复的组合。

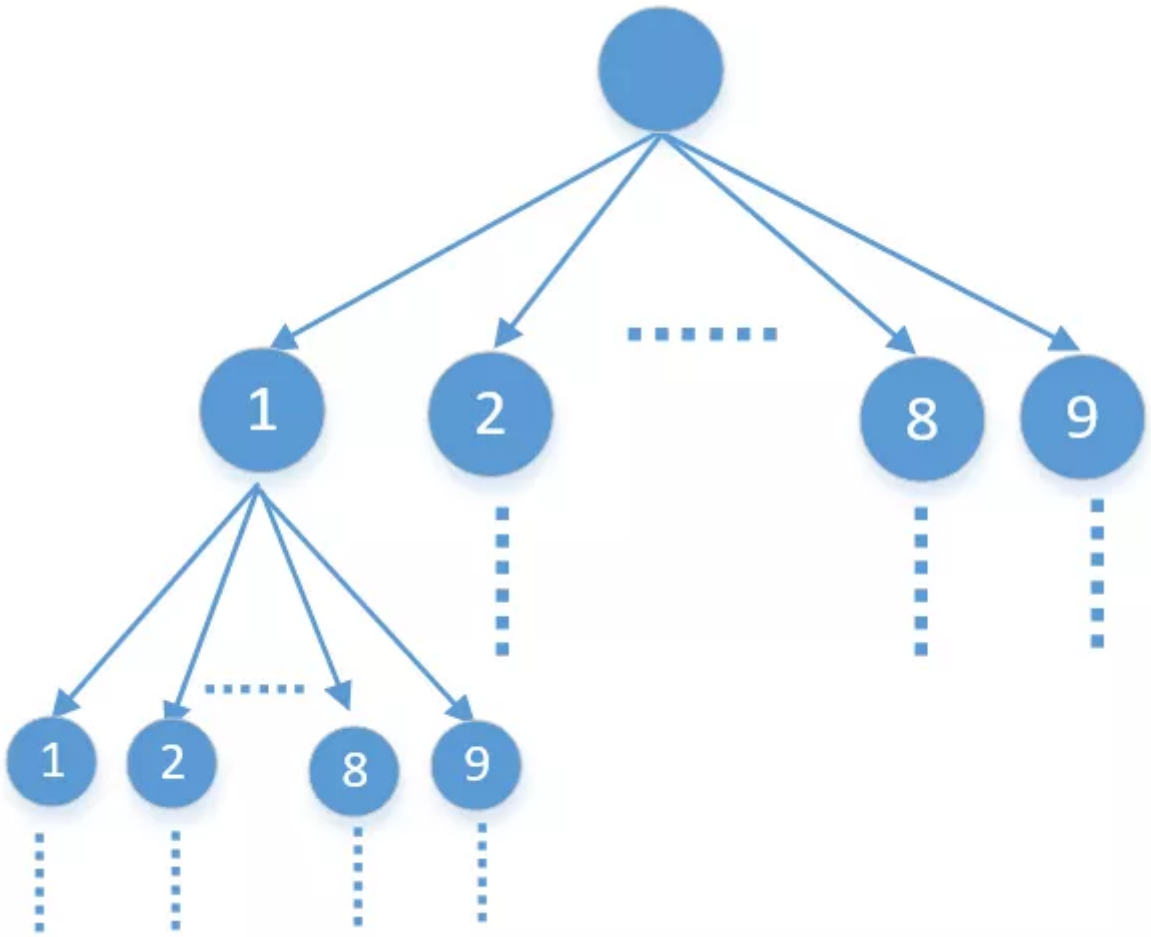
示例 1：

> **输入**: k = 3, n = 7
> **输出**: [[1,2,4]]

示例 2：

> **输入**: k = 3, n = 9
> **输出**: [[1,2,6], [1,3,5], [2,3,4]]

这题说的很明白，就是从1-9中选出k个数字，他们的和等于n，并且这k个数字不能有重复的。所以第一次选择的时候可以从这9个数字中任选一个，沿着这个分支走下去，第二次选择的时候还可以从这9个数字中任选一个，但因为不能有重复的，所以要排除第一次选择的，第二次选择的时候只能从剩下的8个数字中选一个……。这个选择的过程就比较明朗了，我们可以把它看做一棵9叉树，除叶子结点外每个节点都有9个子节点，也就是下面这样



从9个数字中任选一个，就是沿着他的任一个分支一直走下去，其实就是DFS（如果不懂DFS的可以看下373，数据结构-6,树），二叉树的DFS代码是下面这样的

```
public static void treeDFS(TreeNode root) {
  if (root == null)
    return;
  System.out.println(root.val);
  treeDFS(root.left);
  treeDFS(root.right);
}
```

这里可以仿照二叉树的DFS来写一下9叉树，9叉树的DFS就是通过递归遍历他的9个子节点，代码如下

```
public static void treeDFS(TreeNode root) {
  //递归必须要有终止条件
  if (root == null)
    return;
  System.out.println(root.val);

  //通过循环，分别遍历9个子树
  for (int i = 1; i <= 9; i++) {
    //2，一些操作，可有可无，视情况而定

    treeDFS("第i个子节点");

    //3，一些操作，可有可无，视情况而定
  }
}
```

DFS其实就相当于遍历他的所有分支，就是列出他所有的可能结果，只要判断结果等于n就是我们要找的，那么这棵9叉树最多有多少层呢，因为有k个数字，所以最多只能有k

层。来看下代码

```java
public List<List<Integer>> combinationSum3(int k, int n) {
    List<List<Integer>> res = new ArrayList<>();
    dfs(res, new ArrayList<>(), k, 1, n);
    return res;
}

private void dfs(List<List<Integer>> res, List<Integer> list, int k, int start, int n) {
    //终止条件，如果满足这个条件，再往下找也没什么意义了
    if (list.size() == k || n <= 0) {
        //如果找到一组合适的就把他加入到集合list中
        if (list.size() == k && n == 0)
            res.add(new ArrayList<>(list));
        return;
    }
    //通过循环，分别遍历9个子树
    for (int i = start; i <= 9; i++) {
        //选择当前值
        list.add(i);
        //递归
        dfs(res, list, k, i + 1, n - i);
        //撤销选择
        list.remove(list.size() - 1);
    }
}
```

代码相对来说还是比较简单的，我们来分析下（如果看懂了可以直接跳过）。

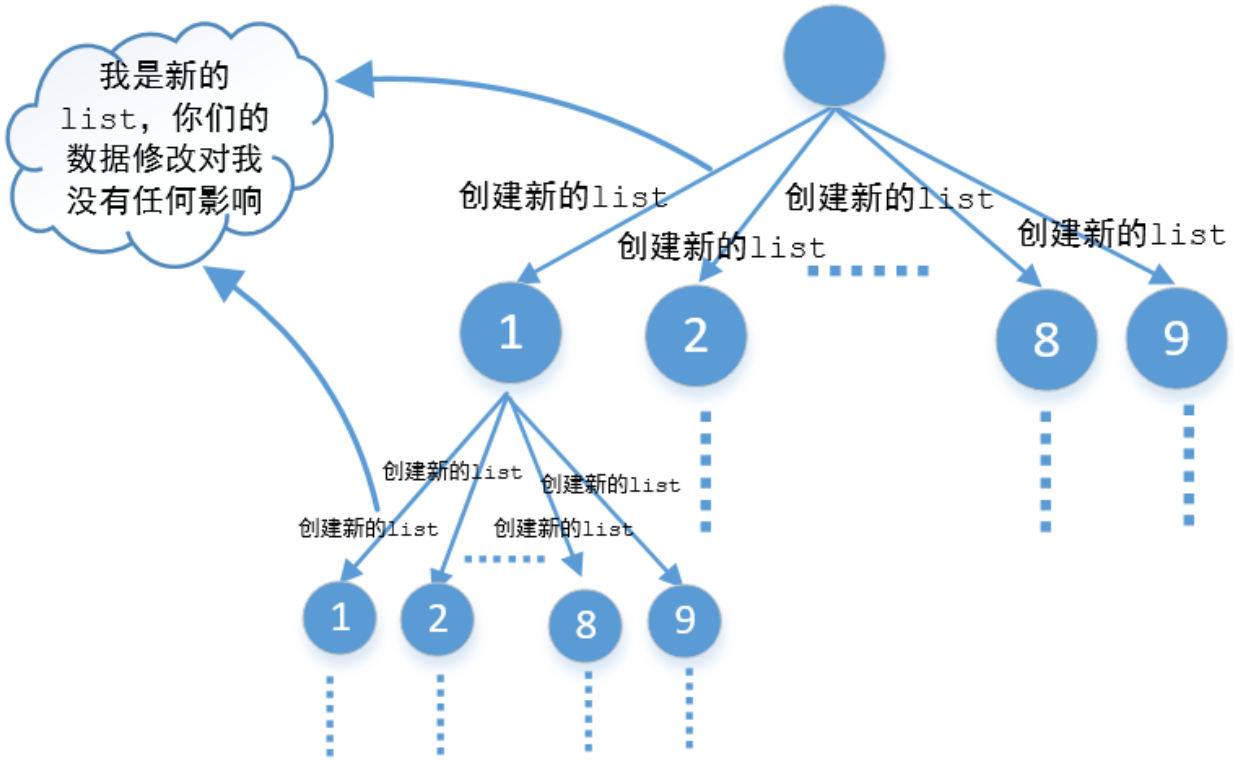
1，代码dfs中最开始的地方是终止条件的判断，之前在426，什么是递归，通过这篇文章，让你彻底搞懂递归中讲过，递归必须要有终止条件。

2，下面的for循环分别遍历他的9个子节点，注意这里的i是从start开始的，所以有可能还没有9个子树，前面说过，如果从9个数字中选择一个之后，第2次就只能从剩下的8个选择了，第3次就只能从剩下的7个中选择了……

3，在第20行dsf中的start是i+1，这里要说一下为什么是i+1。比如我选择了3，下次就应该从4开始选择，如果不加1，下次还从3开始就出现了数字重复，明显与题中的要求不符

4，for循环的i为什么不能每次都从1开始，如果每次都从1开始就会出现结果重复，比如选择了[1，2]，下次可能就会选择[2，1]。

5，如果对回溯算法不懂的，可能最不容易理解的就是最后一行，为什么要撤销选择。如果经常看我公众号的同学可能知道，也就是我经常提到的分支污染问题，因为list是引用传递，当从一个分支跳到另一个分支的时候，如果不把前一个分支的数据给移除掉，那么list就会把前一个分支的数据带到下一个分支去，造成结果错误（下面会讲）。那么除了把前一个分支的数据给移除以外还有一种方式就是每个分支都创建一个list，这样每个分支都是一个新的list，都不互相干扰，也就是下面图中这样



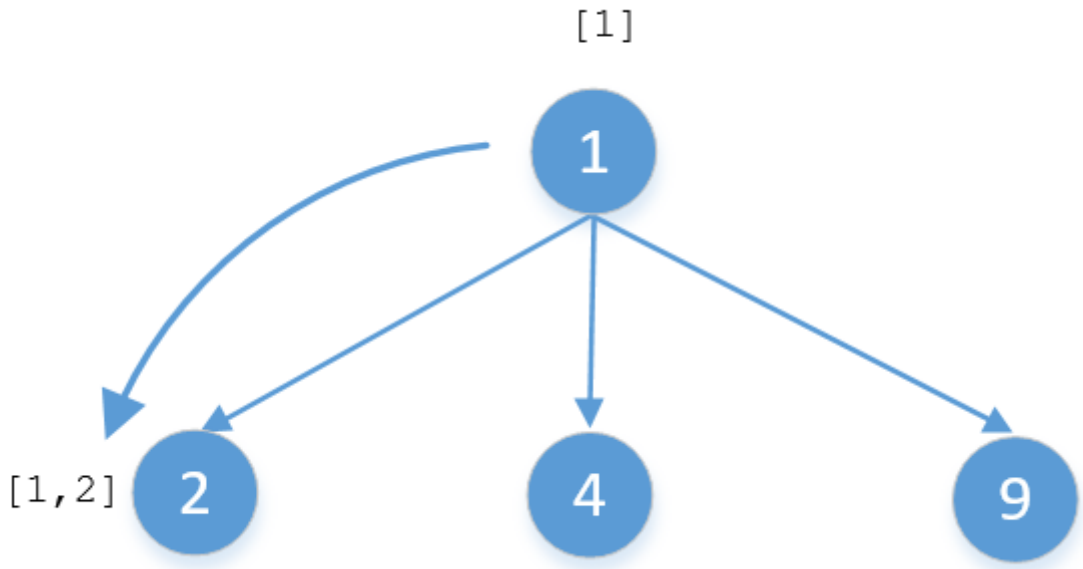
代码如下

```java
public List<List<Integer>> combinationSum3(int k, int n) {
  List<List<Integer>> res = new ArrayList<>();
  dfs(res, new ArrayList<>(), k, 1, n);
  return res;
}

private void dfs(List<List<Integer>> res, List<Integer> list, int k, int start, int n) {
  //终止条件，如果满足这个条件，再往下找也没什么意义了
  if (list.size() == k || n <= 0) {
    //如果找到一组合适的就把他加入到集合list中
    if (list.size() == k && n == 0)
      res.add(new ArrayList<>(list));
    return;
  }
  //通过循环，分别遍历9个子树
  for (int i = start; i <= 9; i++) {
    //创建一个新的list，和原来的list撇清关系，对当前list的修改并不会影响到之前的list
    List<Integer> subList = new LinkedList<>(list);
    subList.add(i);
    //递归
    dfs(res, subList, k, i + 1, n - i);
    //注意这里没有撤销的操作，因为是在一个新的list中的修改，原来的list并没有修改，
    //所以不需要撤销操作
  }
}
```

我们看到最后并没有撤销的操作，这是因为每个分支都是一个新的list，你对当前分支的修改并不会影响到其他分支，所以并不需要撤销操作。

注意：大家尽量不要写这样的代码，这种方式虽然也能解决，但每个分支都会重新创建list，效率很差。

要搞懂最后一行代码首先要明白什么是递归，递归分为递和归两部分，递就是往下传递，归就是往回走。递归你从什么地方调用最终还会回到什么地方去，我们来画个简单的图看一下



这是一棵非常简单的3叉树，假如要对他进行DFS遍历，当沿着1→2这条路径走下去的时候，list中应该是[1，2]。因为是递归调用最终还会回到节点1，如果不把2给移除掉，当沿着1→4这个分支走下去的时候就变成[1，2，4]，但实际上1→4这个分支的结果应该是[1，4]，这是因为我们把前一个分支的值给带过来了。当1，2这两个节点遍历完之后最终还是返回节点1，在回到节点1的时候就应该把结点2的值给移除掉，让list变为[1]，然后再沿着1→4这个分支走下去，结果就是[1，4]。

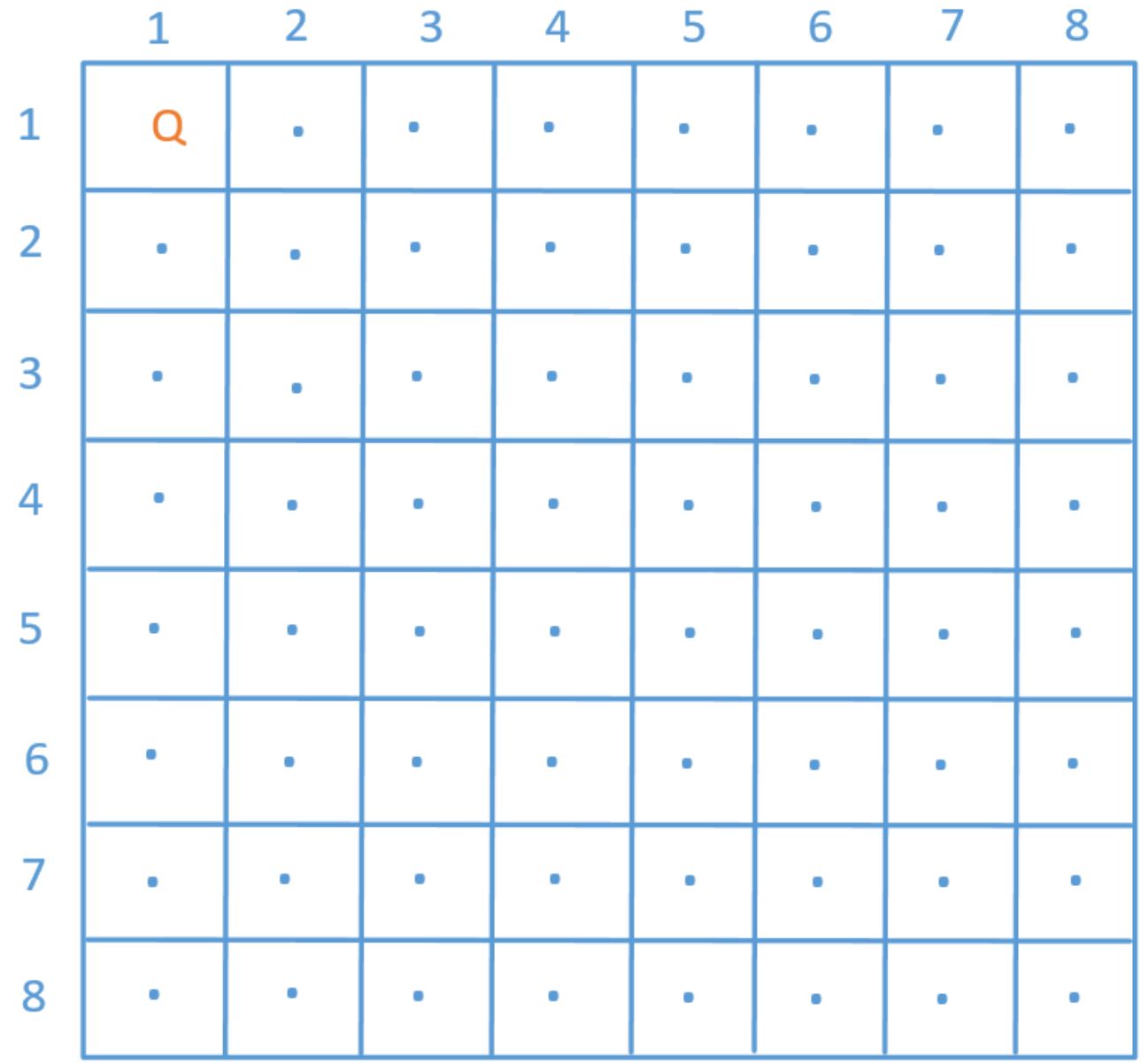
我们来总结一下回溯算法的代码模板吧

```
private void backtrack("原始参数") {
  //终止条件(递归必须要有终止条件)
  if ("终止条件") {
    //一些逻辑操作（可有可无，视情况而定）
    return;
  }

  for (int i = "for循环开始的参数"; i < "for循环结束的参数"; i++) {
    //一些逻辑操作（可有可无，视情况而定）

    //做出选择

    //递归
    backtrack("新的参数");
    //一些逻辑操作（可有可无，视情况而定）

    //撤销选择
  }
}
```

有了模板，回溯算法的代码就非常容易写了，下面就根据模板来写几个经典回溯案例的答案。

八皇后问题

八皇后问题也是一道非常经典的回溯算法题，前面也有对八皇后问题的专门介绍，有不明白的可以先看下394，经典的八皇后问题和N皇后问题。他就是不断的尝试，如果当前位置能放皇后就放，一直到最后一行如果也能放就表示成功了，如果某一个位置不能放，就回到上一步重新尝试。比如我们先在第1行第1列放一个皇后，如下图所示



然后第2行从第1列开始在不冲突的位置再放一个皇后，然后第3行……一直这样放下去，如果能放到第8行还不冲突说明成功了，如果没到第8行的时候出现了冲突就回到上一步继续查找合适的位置……来看下代码

```
//row表示的是第几行
public void solve(char[][] chess, int row) {
  //终止条件，row是从0开始的，最后一行都可以放，说明成功了
  if (row == chess.length) {
    //自己写的一个公共方法，打印二维数组的，
    //主要用于测试数据用的
    Util.printTwoCharArrays(chess);
    System.out.println();
    return;
  }
  for (int col = 0; col < chess.length; col++) {
    //验证当前位置是否可以放皇后
    if (valid(chess, row, col)) {
      //如果在当前位置放一个皇后不冲突，
      //就在当前位置放一个皇后
```

```
            chess[row][col] = 'Q';
            //递归，在下一行继续……
            solve(chess, row + 1);
            //撤销当前位置的皇后
            chess[row][col] = '.';
        }
    }
}
```

全排列问题

给定一个**没有重复**数字的序列，返回其所有可能的全排列。

示例:

> **输入**: [1,2,3]
> **输出**:
> [
> [1,2,3],
> [1,3,2],
> [2,1,3],
> [2,3,1],
> [3,1,2],
> [3,2,1]
>]

这道题我们可以把它当做一棵3叉树，所以每一步都会有3种选择，具体过程就不在分析了，直接根据上面的模板来写下代码

```
public List<List<Integer>> permute(int[] nums) {
    List<List<Integer>> list = new ArrayList<>();
    backtrack(list, new ArrayList<>(), nums);
    return list;
}

private void backtrack(List<List<Integer>> list, List<Integer> tempList, int[] nums) {
    //终止条件
    if (tempList.size() == nums.length) {
        //说明找到一一组组合
        list.add(new ArrayList<>(tempList));
        return;
    }
    for (int i = 0; i < nums.length; i++) {
        //因为不能有重复的，所以有重复的就不能选
        if (tempList.contains(nums[i]))
            continue;
        //选择当前值
        tempList.add(nums[i]);
        //递归
        backtrack(list, tempList, nums);
        //撤销选择
        tempList.remove(tempList.size() - 1);
    }
}
```

是不是感觉很简单。

子集问题

给定一组**不含重复元素**的整数数组 *nums*，返回该数组所有可能的子集（幂集）。

说明：解集不能包含重复的子集。

示例:

> **输入**: nums = [1,2,3]
> **输出**:
> [
> [3],
> [1],
> [2],
> [1,2,3],
> [1,3],
> [2,3],
> [1,2],
> []
>]

我们还是根据模板来修改吧

```java
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> list = new ArrayList<>();
    //先排序
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, 0);
    return list;
}

private void backtrack(List<List<Integer>> list, List<Integer> tempList, int[] nums, int start) {
    //注意这里没有写终止条件，不是说递归一定要有终止条件的吗，这里怎么没写，其实这里的终止条件
    //隐含在for循环中了，当然我们也可以写if(start>nums.length) rerurn;只不过这里省略了。
    list.add(new ArrayList<>(tempList));
    for (int i = start; i < nums.length; i++) {
        //做出选择
        tempList.add(nums[i]);
        //递归
        backtrack(list, tempList, nums, i + 1);
        //撤销选择
        tempList.remove(tempList.size() - 1);
    }
}
```

所以类似这种题基本上也就是这个套路，就是先做出选择，然后递归，最后再撤销选择。在讲第一道示例的时候提到过撤销的原因是防止把前一个分支的结果带到下一个分支造成结果错误。不过他们不同的是，一个是终止条件的判断，还一个就是for循环的起始值，这些都要具体问题具体分析。下面再来看最后一到题解数独。

解数独

数独大家都玩过吧，他的规则是这样的
一个数独的解法需遵循如下规则：

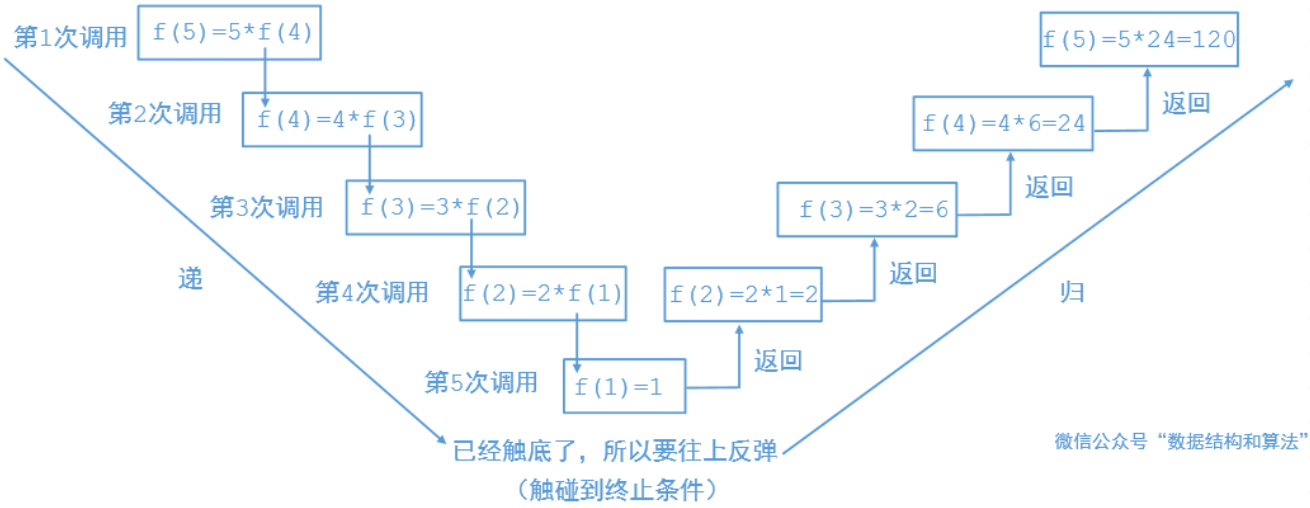
- 数字 1-9 在每一行只能出现一次。
- 数字 1-9 在每一列只能出现一次。
- 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。

过程就不在分析了，直接看代码，代码中也有详细的注释，这篇讲的是回溯算法，这里主要看一下backTrace方法即可，其他的可以先不用看

```
//回溯算法
public static boolean solveSudoku(char[][] board) {
    return backTrace(board, 0, 0);
}

//注意这里的参数，row表示第几行，col表示第几列。
private static boolean backTrace(char[][] board, int row, int col) {
    //注意row是从0开始的，当row等于board.length的时候表示数独的
    //最后一行全部读遍历完了，说明数独中的值是有效的，直接返回true
    if (row == board.length)
        return true;
    //如果当前行的最后一列也遍历完了，就从下一行的第一列开始。这里的遍历
    //顺序是从第1行的第1列一直到最后一列，然后第二行的第一列一直到最后
    //一列，然后第三行的……
    if (col == board.length)
        return backTrace(board, row + 1, 0);
    //如果当前位置已经有数字了，就不能再填了，直接到这一行的下一列
    if (board[row][col] != '.')
        return backTrace(board, row, col + 1);
    //如果上面条件都不满足，我们就从1到9中选择一个合适的数字填入到数独中
    for (char i = '1'; i <= '9'; i++) {
        //判断当前位置[row，col]是否可以放数字i，如果不能放再判断下
        //一个能不能放，直到找到能放的为止，如果从1-9都不能放，就会下面
        //直接return false
        if (!isValid(board, row, col, i))
            continue;
        //如果能放数字i，就把数字i放进去
        board[row][col] = i;
        //如果成功就直接返回，不需要再尝试了
        if (backTrace(board, row, col))
            return true;
        //否则就撤销重新选择
        board[row][col] = '.';
    }
    //如果当前位置[row，col]不能放任何数字，直接返回false
    return false;
}

//验证当前位置[row，col]是否可以存放字符c
private static boolean isValid(char[][] board, int row, int col, char c) {
    for (int i = 0; i < 9; i++) {
        if (board[i][col] != '.' && board[i][col] == c)
            return false;
        if (board[row][i] != '.' && board[row][i] == c)
            return false;
        if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] != '.' &&
                board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c)
            return false;
    }
    return true;
}
```

总结

回溯算法要和递归结合起来就很好理解了，递归分为两部分，第一部分是先往下传递，第二部分到达终止条件的时候然后再反弹往回走，我们来看一下阶乘的递归



其实回溯算法就是在往下传递的时候把某个值给改变，然后往回反弹的时候再把原来的值复原即可。比如八皇后的时候我们先假设一个位置可以放皇后，如果走不通就把当前位置给撤销，放其他的位置。如果是组合之类的问题，往下传递的时候我们把当前值加入到list中，然后往回反弹的时候在把它从list中给移除掉即可。

关于回溯算法前面也讲过一些，有兴趣的可以看下

446，回溯算法解黄金矿工问题

442，回溯算法解二叉树中和为某一值的路径

420，回溯算法解矩阵中的路径

391，回溯算法求组合问题

往期推荐

- 426，什么是递归，通过这篇文章，让你彻底搞懂递归
- 394，经典的八皇后问题和N皇后问题
- 371，背包问题系列之-基础背包问题
- 366，约瑟夫环
- 362，汉诺塔
- 356，青蛙跳台阶相关问题

448，组合的几种解决方式

原创 山大王wld 数据结构和算法 9月9日

收录于话题
#算法图文分析

95个 >



We are all in the gutter, but some of us are looking at the stars.

身在井隅，心向璀璨。

问题描述

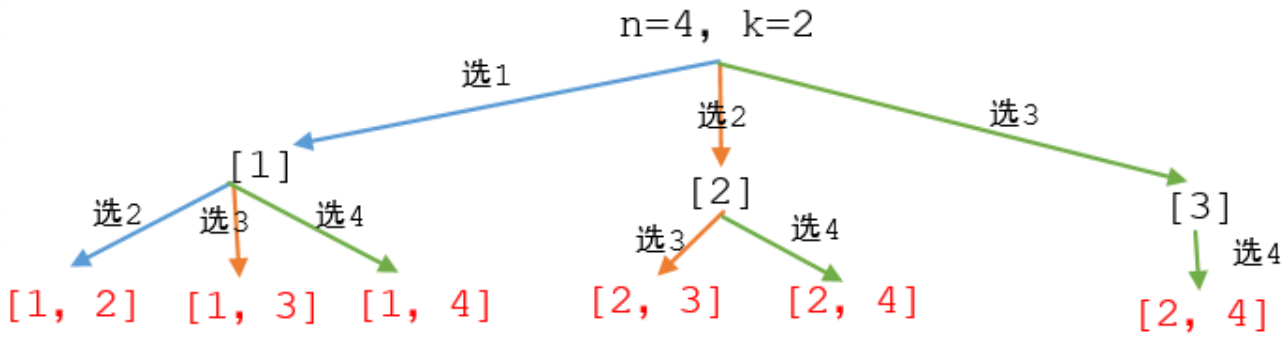
给定两个整数 *n* 和 *k*，返回 1 ... *n* 中所有可能的 *k* 个数的组合。

示例:

> **输入**: n = 4, k = 2
> **输出**:
> [
> [2,4],
> [3,4],
> [2,3],
> [1,2],
> [1,3],
> [1,4],
>]

回溯方式解决

这题要求的是从n个数字中选出k个弄成一组合，问最后总共有多少种组合，其实我们可以把它看做是一棵n-k+1叉树，比如n=4，k=2，那么就是一颗3叉树，这里以示例为例画个图看一下。



注意这里只能选择后面的数字不能选择前面的数字，比如我选择了2，就能在选择1了，否则就会出现[1，2]和[2，1]这种重复的组合，同理我选择了3，那么1和2都不能再选了因为如果再选就会出现重复。

n叉树的遍历还记得吗

```
private void backtrack() {
  if ("终止条件") {
    return;
  }

  for (int i = ?; i <= n - k + 1; i++) {
    //逻辑运算1，（可有可无，视情况而定）

    //递归调用，遍历每一个分支
    backtrack(list, n, k - 1, i + 1, tempList);

    //逻辑运算2，（可有可无，视情况而定）
  }
}
```

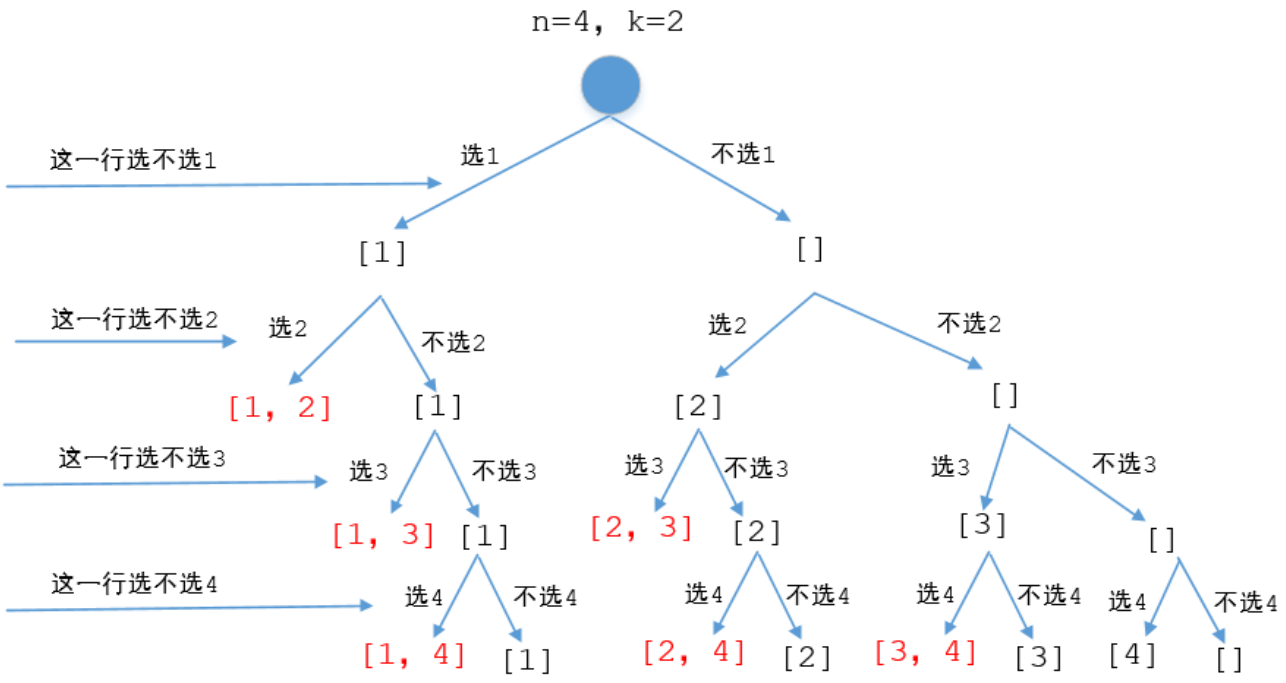
我们来看下上面的框架，逻辑运算1的时候，就表示沿着当前分支走下去，我们把当前选择的值添加到集合中。逻辑2表示这个分支走完了我们要跳到另一个分支，之前讲426，什么是递归，通过这篇文章，让你彻底搞懂递归的时候提到过，从一个分支跳到另一个分支的时候，要么之前分支在运算之前先复制一份，要么把当前分支添加的值给移除，否则会造成分支污染。这两种方式都可以，但后一种效率会更好，他不会大量的复制数据。那么终止条件是什么呢，就是集合中的数据大小为k的时候，就表示找到了一组组合。搞懂了这一点代码就比较容易写了

```
public List<List<Integer>> combine(int n, int k) {
  List<List<Integer>> list = new LinkedList<>();
  backtrack(list, n, k, 1, new ArrayList<>());
  return list;
}

```

```java
private void backtrack(List<List<Integer>> list, int n, int k, int start, List<Integer> tempList) {
    //终止条件，找到一组组合
    if (k == 0) {
        list.add(new LinkedList<>(tempList));
        return;
    }
    //注意这里的i不能从0开始，如果从0开始会出现重复的，比如[1，2]和[2，1]
    for (int i = start; i <= n - k + 1; i++) {
        //把当前值添加到集合中
        tempList.add(i);
        //递归调用
        backtrack(list, n, k - 1, i + 1, tempList);
        //从当前分支跳到下一个分支的时候要把之前添加的值给移除
        tempList.remove(tempList.size() - 1);
    }
}
```

参考二进制位

我们知道二进制位中，每个位置有两种状态，一种是0一种是1，这里也可以参照位运算的表示方式， 每个数字都有选和不选两种状态 ，具体画个图来看下




```java
public List<List<Integer>> combine(int n, int k) {
    List<List<Integer>> list = new ArrayList<>();
    backtrack(list, n, k, 1, new ArrayList<>());
    return list;
}

private void backtrack(List<List<Integer>> list, int n, int k, int start, List<Integer> tempList) {
    //终止条件，找到一对组合
    if (k == 0) {
        list.add(new ArrayList<>(tempList));
        return;
    }
    if (start <= n - k) {
        //不选当前值，k不变
        backtrack(list, n, k, start + 1, tempList);
    }
    //选择当前值，k要减1
    tempList.add(start);
```

```
        backtrack(list, n, k - 1, start + 1, tempList);
        //因为是递归调用，跳到下一个分支的时候，要把这个分支选的值给移除
        tempList.remove(tempList.size() - 1);
    }
```

递归方式解决

这题要求的是从n个数字中选出k个弄成一组合，问最后总共有多少种组合，如果用数学知识就是C(n,k)，这个公式又可以表示为C(n,k)=C(n-1,k-1)+C(n-1,k)，也就是说要么选第n个数字，要么不选第n个数字。

（1），选第n个数字

如果选第n个数字，我们需要从前面n-1个数字中选择k-1个，然后在和数字n组合

（2），不选第n个数字

如果不选第n个数字，我们可以直接从前面n-1个数字中选择k个即可

那么最终的结果就是上面两种的和，我们来直接看下代码

```
public List<List<Integer>> combine(int n, int k) {
    List<List<Integer>> res = new ArrayList<>();
    //边界条件判断
    if (n < k || k == 0)
        return res;
    //不选择第n个，从前面n-1个数字中选择k-1个构成一个集合
    res = combine(n - 1, k - 1);
    //如果res是空，添加一个空的集合
    if (res.isEmpty())
        res.add(new ArrayList<>());
    //然后在前面选择的集合res中的每个子集合的后面添加一个数字n
    for (List<Integer> list : res)
        list.add(n);
    //res中不光要包含选择第n个数字的集合，也要包含不选择第n
    //个数字的集合
    res.addAll(combine(n - 1, k));
    return res;
}
```

总结

这题也不是特别难，但解题思路很多，每一种都比较经典。关于组合的题型也比较多，前面还讲过391，回溯算法求组合问题，有兴趣的也可以看下

往期推荐

- 391，回溯算法求组合问题

446，回溯算法解黄金矿工问题

原创 山大王wld 数据结构和算法 9月4日

收录于话题
#算法图文分析

95个 >



TENET. It will open the right doors, some of the wrong ones too.

信条，它会打开正确的门，也会打开错误的门。

问题描述

你要开发一座金矿，地质勘测学家已经探明了这座金矿中的资源分布，并用大小为 m * n 的网格 grid 进行了标注。每个单元格中的整数就表示这一单元格中的黄金数量；如果该单元格是空的，那么就是 0。

为了使收益最大化，矿工需要按以下规则来开采黄金：

- 每当矿工进入一个单元，就会收集该单元格中的所有黄金。
- 矿工每次可以从当前位置向上下左右四个方向走。
- 每个单元格只能被开采（进入）一次。
- 不得开采（进入）黄金数目为 0 的单元格。
- 矿工可以从网格中 任意一个 有黄金的单元格出发或者是停止。

示例 1：

输入：
grid = [[0,6,0],[5,8,7],[0,9,0]]

输出：24
解释：
[[0,6,0],
 [5,8,7],
 [0,9,0]]

一种收集最多黄金的路线是：
9 -> 8 -> 7。

示例 2：

输入：
grid = [[1,0,7],[2,0,6],[3,4,5],[0,3,0],[9,0,20]]

输出：28
解释：
[[1,0,7],
 [2,0,6],
 [3,4,5],
 [0,3,0],
 [9,0,20]]

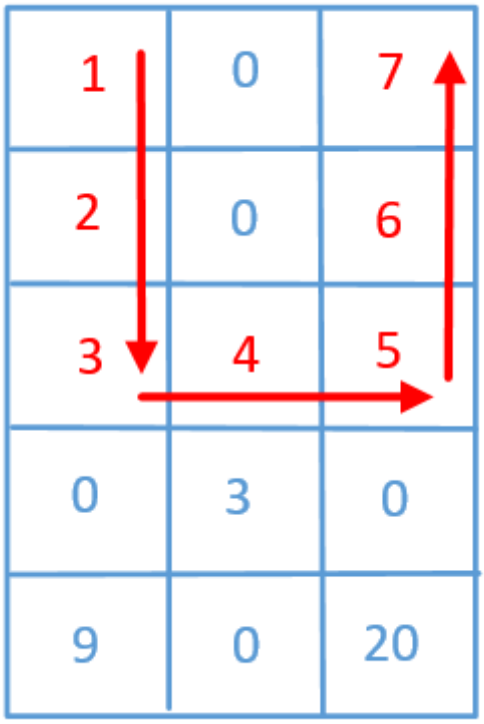
一种收集最多黄金的路线是：
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7。

提示：

- 1 <= grid.length, grid[i].length <= 15
- 0 <= grid[i][j] <= 100
- 最多 **25** 个单元格中有黄金。

问题分析

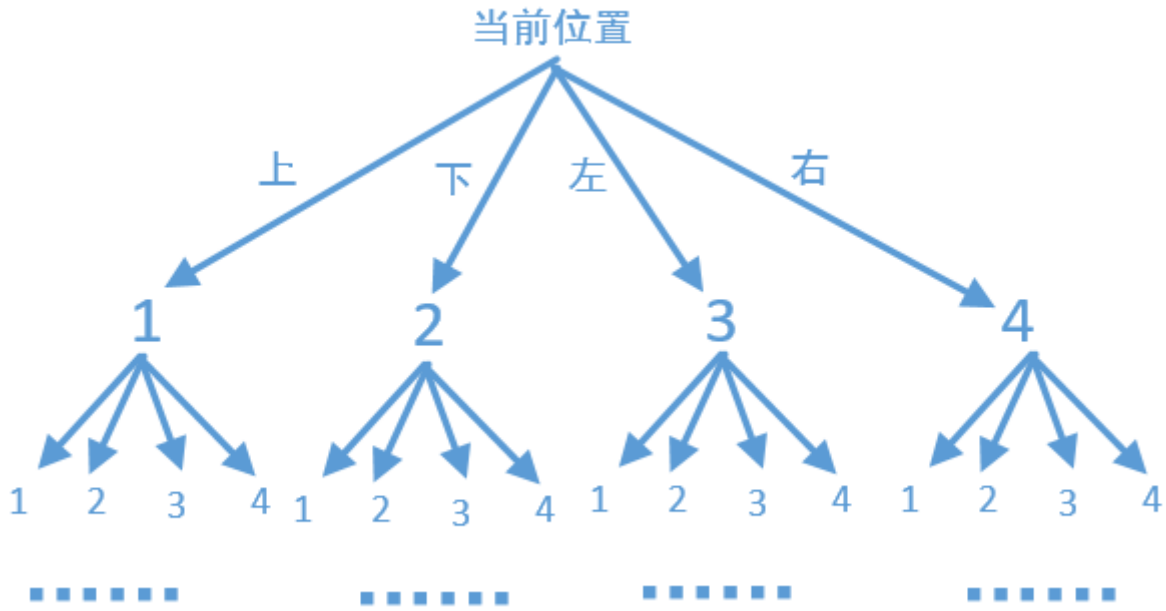
这题上面说了一大堆，其实就是在一个二维数组中从任一位置开始，可以往他的上下左右4个方向走，然后返回走过的路线中值最大的，0其实就相当于障碍物，不能往位置为0的地方走，画个简单的图看一下



我们需要遍历每一个位置，从任何一个位置开始找到最大路径，所以代码大致轮廓如下

```
public int getMaximumGold(int[][] grid) {
    //边界条件判断
    if (grid == null || grid.length == 0)
        return 0;
    //保存最大路径值
    int res = 0;
    //两个for循环，遍历每一个位置，让他们当做起点
    //查找最大路径值
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[0].length; j++) {
            //函数dfs是以坐标(i,j)为起点，查找最大路径值
            res = Math.max(res, dfs(grid, i, j));
        }
    }
    //返回最大路径值
    return res;
}
```

代码的大致轮廓写出来了，这里主要是dfs这个函数，他表示的是以(i，j)为坐标点，沿着他的上下左右4个方向查找最大路径，这里我们很容易把它想象成为一颗4叉树，就像下面这样



看到这个图，很容易想到之前讲的[426，什么是递归，通过这篇文章，让你彻底搞懂递归](#)。他会沿着每一个分支一直走下去，直到遇到终止条件，并且把走过的位置全部置为0，表示不能再走这个位置了。终止条件是什么呢，很明显，i和j都不能越界，并且当前位置不能是0，也就是下面这样

```
if (i < 0 || i >= grid.length || j < 0 || j >= grid[0].length || grid[i][j] == 0)
  return 0;
```

因为使用的是递归，往下走的时候把当前的值给置为0了，当递归往回走的时候我们当前位置的值给还原，所以上面dfs的最终代码如下

```
public int dfs(int[][] grid, int x, int y) {
  //边界条件的判断，x,y都不能越界，同时当前坐标的位置如果是0，表示有障碍物
  //或者遍历过了
  if (x < 0 || x >= grid.length || y < 0 || y >= grid[0].length || grid[x][y] == 0)
    return 0;
  //先把当前坐标的值保存下来，最后再还原
  int temp = grid[x][y];
  //当前坐标已经访问过了，要把他标记为0，防止再次访问
  grid[x][y] = 0;
  //然后沿着当前坐标的上下左右4个方向查找
  int up = dfs(grid, x, y - 1);//往上找
  int down = dfs(grid, x, y + 1);//往下找
  int left = dfs(grid, x - 1, y);//往左找
  int right = dfs(grid, x + 1, y);//往右找
  //这里只要4个方向的最大值即可
  int max = Math.max(left, Math.max(right, Math.max(up, down)));
  //然后再把当前位置的值还原
  grid[x][y] = temp;
  //返回最大路径值
  return grid[x][y] + max;
}
```

总结

这题需要遍历所有的位置，然后以他为中心点往他的上下左右4个方向查找，最后返回找到的最大值即可。如果遍历到某个位置的时候先要把它置为0，表示已经访问过了，当最后访问完之后还要把它复原。

442，剑指 Offer-回溯算法解二叉树中和为某一值的路径

原创 山大王wld 数据结构和算法 8月22日

收录于话题
#剑指offer

27个 >



There is no point me doing this if I can't be myself.

如果我不能做自己，那还有什么意义。

问题描述

输入一棵二叉树和一个整数，打印出二叉树中节点值的和为输入整数的所有路径。从树的根节点开始往下一直到叶节点所经过的节点形成一条路径。

示例:
给定如下二叉树，以及目标和 sum = 22，

```
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \    / \
        7    2  5   1
```

返回:

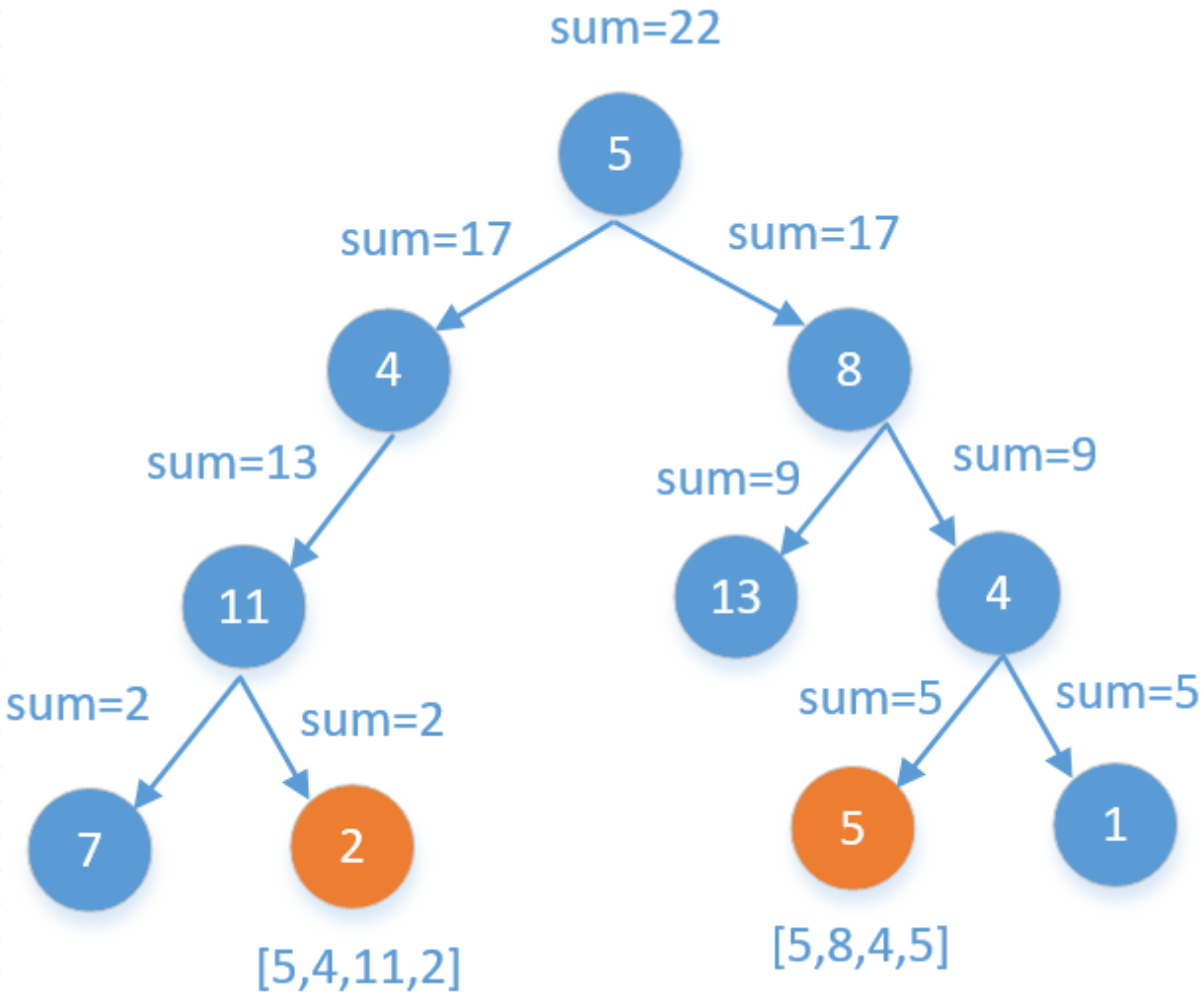
```
[
   [5,4,11,2],
   [5,8,4,5]
]
```

提示：

1. 节点总数 <= 10000

回溯算法

这题没说sum是正数还是负数，也没说树中节点的值有没有负数。我们要做的是从根节点到叶子节点遍历他所有的路径，返回他所有路径中和等于sum的节点，这里有两种实现方式，一种是减，一种是加。减就是从根节点开始，用sum不断的减去遍历到的每一个节点，一直到叶子节点，在减去叶子节点之前查看sum是否等于叶子节点，如果等于说明我们找到了一组，画个图看一下



来看下代码

```
1   public List<List<Integer>> pathSum(TreeNode root, int sum) {
```

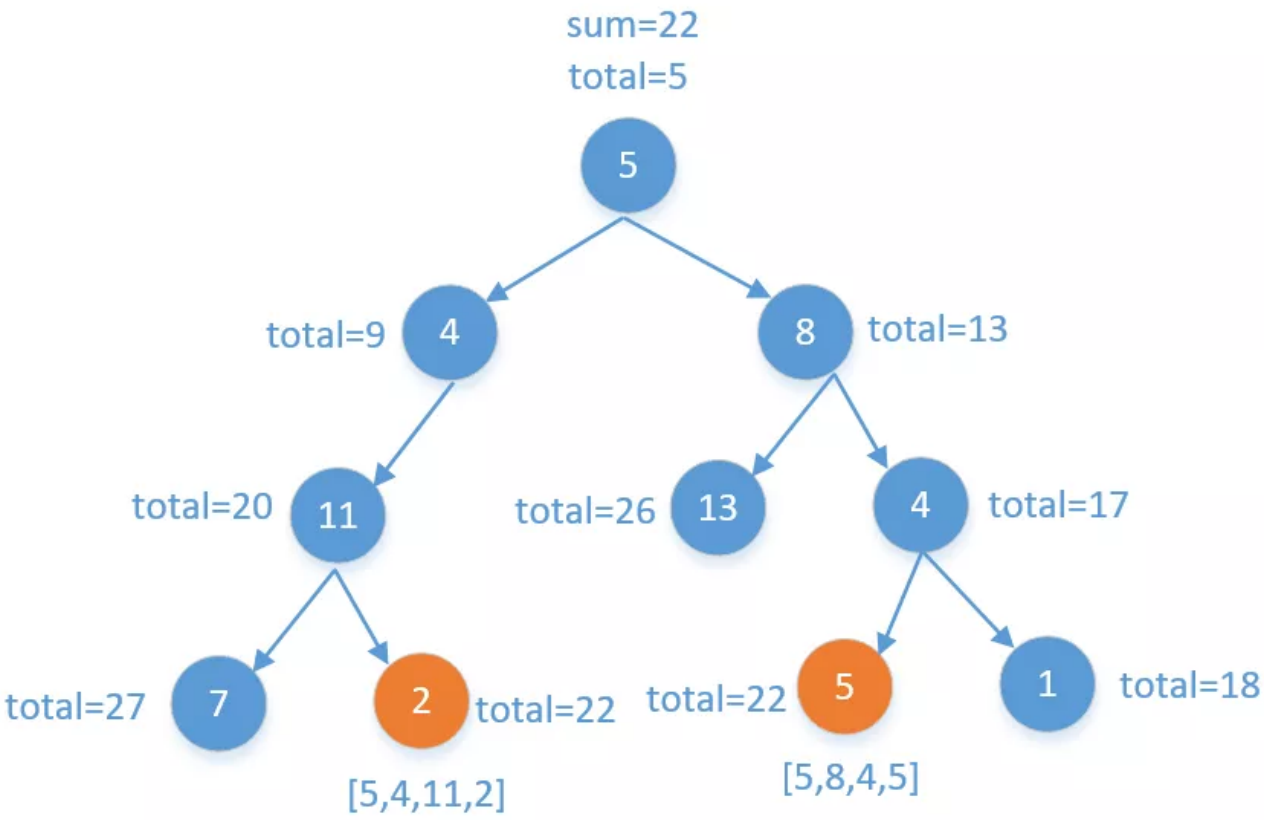
```
    List<List<Integer>> result = new ArrayList<>();
    dfs(root, sum, new ArrayList<>(), result);
    return result;
}

public void dfs(TreeNode root, int sum, List<Integer> list,
          List<List<Integer>> result) {
    //如果节点为空直接返回
    if (root == null)
        return;
    //因为list是引用传递，为了防止递归的时候分支污染，我们要在每个路径
    //中都要新建一个subList
    List<Integer> subList = new ArrayList<>(list);
    //把当前节点值加入到subList中
    subList.add(new Integer(root.val));
    //如果到达叶子节点，就不能往下走了，直接return
    if (root.left == null && root.right == null) {
        //如果到达叶子节点，并且sum等于叶子节点的值，说明我们找到了一组，
        //要把它放到result中
        if (sum == root.val)
            result.add(subList);
        //到叶子节点之后直接返回，因为在往下就走不动了
        return;
    }
    //如果没到达叶子节点，就继续从他的左右两个子节点往下找，注意到
    //下一步的时候，sum值要减去当前节点的值
    dfs(root.left, sum - root.val, subList, result);
    dfs(root.right, sum - root.val, subList, result);
}
```

上面只是对二叉树的深度优先搜索（DFS），并没有使用**回溯**，之前讲递归的时候提到过为了防止分支污染我们还可以把使用过的值在返回的时候把它给remove掉，这就是大家常提的**回溯算法**，也可以看下之前讲的[426，什么是递归，通过这篇文章，让你彻底搞懂递归]，看下代码

```
public List<List<Integer>> pathSum(TreeNode root, int sum) {
    List<List<Integer>> result = new ArrayList<>();
    dfs(root, sum, new ArrayList<>(), result);
    return result;
}

public void dfs(TreeNode root, int sum, List<Integer> list,
          List<List<Integer>> result) {
    //如果节点为空直接返回
    if (root == null)
        return;
    //把当前节点值加入到list中
    list.add(new Integer(root.val));
    //如果到达叶子节点，就不能往下走了，直接return
    if (root.left == null && root.right == null) {
        //如果到达叶子节点，并且sum等于叶子节点的值，说明我们找到了一组，
        //要把它放到result中
        if (sum == root.val)
            result.add(new ArrayList(list));
        //注意别忘了把最后加入的结点值给移除掉，因为下一步直接return了，
        //不会再走最后一行的remove了，所以这里在rerurn之前提前把最后
        //一个结点的值给remove掉。
        list.remove(list.size() - 1);
        //到叶子节点之后直接返回，因为在往下就走不动了
        return;
    }
    //如果没到达叶子节点，就继续从他的左右两个子节点往下找，注意到
    //下一步的时候，sum值要减去当前节点的值
    dfs(root.left, sum - root.val, list, result);
    dfs(root.right, sum - root.val, list, result);
    //我们要理解递归的本质，当递归往下传递的时候他最后还是会往回走，
    //我们把这个值使用完之后还要把它给移除，这就是回溯
```

```
33      list.remove(list.size() - 1);
34  }
```

上面是减的方式，我们再来看一个加的方式，其实他就是从根节点开始到叶子节点把这个路径上的所有节点都加起来，最后查看是否等于sum，画个图看一下



代码就很简单了，来看下

```
1   public List<List<Integer>> pathSum(TreeNode root, int sum) {
2     List<List<Integer>> result = new ArrayList<>();
3     dfs(root, sum, 0, new ArrayList<>(), result);
4     return result;
5   }
6
7   public void dfs(TreeNode root, int sum, int toal, List<Integer> list,
8           List<List<Integer>> result) {
9     //如果节点为空直接返回
10    if (root == null)
11      return;
12    //把当前节点值加入到list中
13    list.add(new Integer(root.val));
14    //没往下走一步就要计算走过的路径和
15    toal += root.val;
16    //如果到达叶子节点，就不能往下走了，直接return
17    if (root.left == null && root.right == null) {
18      //如果到达叶子节点，并且sum等于toal，说明我们找到了一组，
19      //要把它放到result中
20      if (sum == toal)
21        result.add(new ArrayList(list));
22      //注意别忘了把最后加入的结点值给移除掉，因为下一步直接return了，
23      //不会再走最后一行的remove了，所以这里在rerurn之前提前把最后
24      //一个结点的值给remove掉。
25      list.remove(list.size() - 1);
26      //到叶子节点之后直接返回，因为在往下就走不动了
27      return;
28    }
29    //如果没到达叶子节点，就继续从他的左右两个子节点往下找
30    dfs(root.left, sum, toal, list, result);
31    dfs(root.right, sum, toal, list, result);
```

```
32      //我们要理解递归的本质，当递归往下传递的时候他最后还是会往回走，
33      //我们把这个值使用完之后还要把它给移除，这就是回溯
34      list.remove(list.size() - 1);
35  }
```

总结

原理很简单，就是从根节点到所有叶子节点路径上的结点值，总和是不是等于sum，如果等于就说明找到了一组。这里要注意因为list是引用传递，在往下传递的过程中不能干扰其他路径，有两种方式可以解决，一种是每个路径都新建一个list，一种就是使用回溯算法，就是在当前路径把当前值使用过之后，跳到其他路径的时候要把那个值给移除掉，防止带到下一个分支，给下一个分支造成干扰。

往期推荐

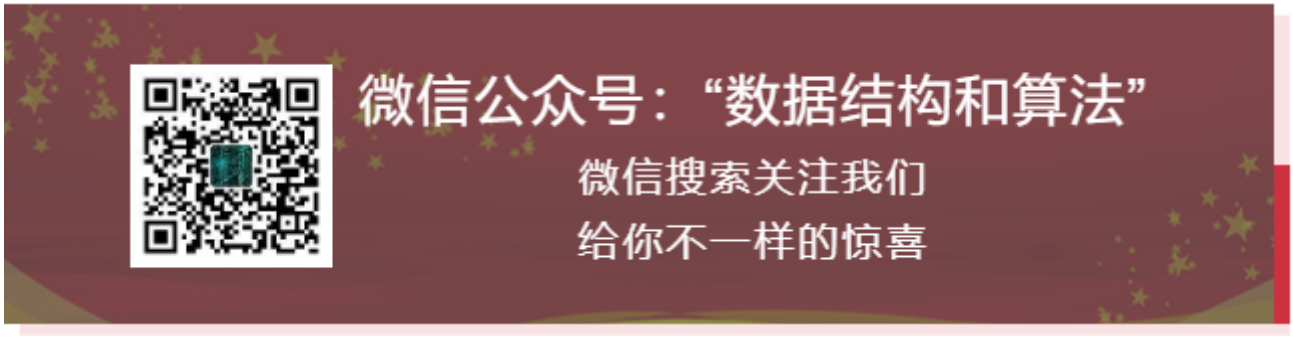
- 440，剑指 Offer-从上到下打印二叉树 II
- 434，剑指 Offer-二叉树的镜像
- 426，什么是递归，通过这篇文章，让你彻底搞懂递归
- 409，动态规划求不同路径

420，剑指 Offer-回溯算法解矩阵中的路径

原创 山大王wld 数据结构和算法 8月4日

收录于话题
#剑指offer

27个 >



I am simply I, and I cannot be labeled.

我就是我，无法被贴上任何标签。

问题描述

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一格开始，每一步可以在矩阵中向左、右、上、下移动一格。如果一条路径经过了矩阵的某一格，那么该路径不能再次进入该格子。

例如，在下面的3×4的矩阵中包含一条字符串“bfce”的路径（路径中的字母用加粗标出）。

[["a","**b**","c","e"],
["s","**f**","**c**","s"],
["a","d","**e**","e"]]

但矩阵中不包含字符串“abfb”的路径，因为字符串的第一个字符b占据了矩阵中的第一行第二个格子之后，路径不能再次进入这个格子。

示例 1：

```
输入：
board = [["A","B","C","E"],
         ["S","F","C","S"],
         ["A","D","E","E"]],
word = "ABCCED"

输出：true
```

示例 2：

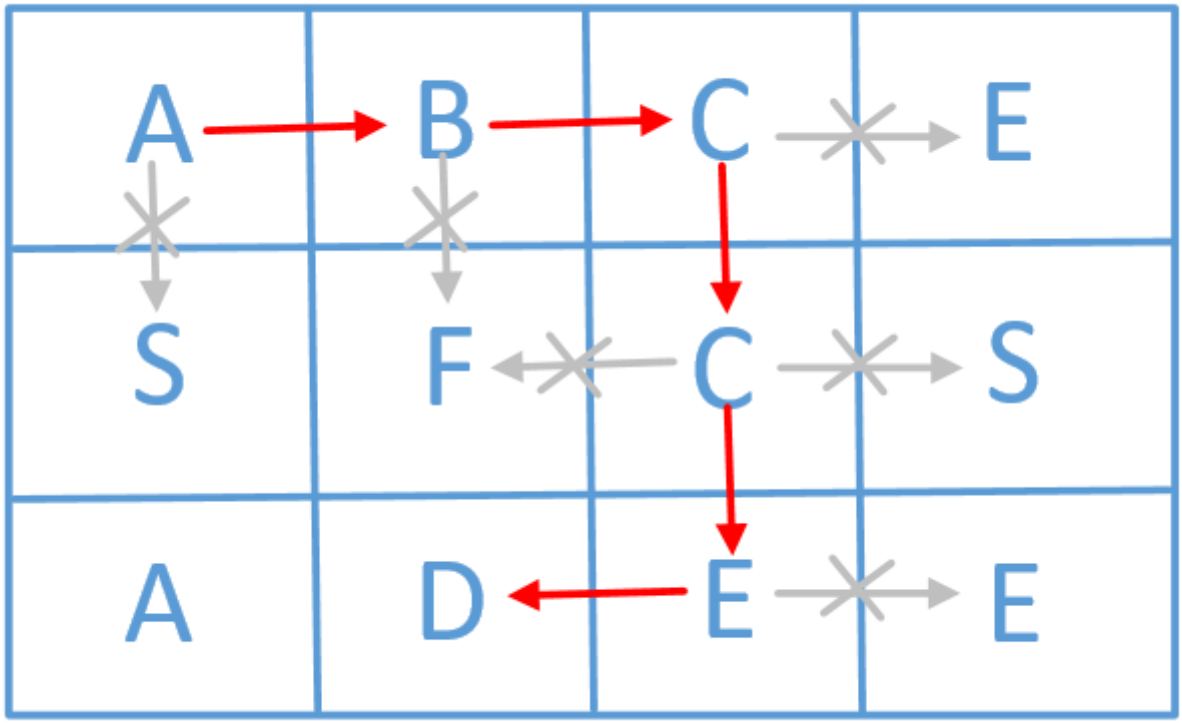
```
输入：board = [["a","b"],
               ["c","d"]],
word = "abcd"

输出：false
```

提示：

- 1 <= board.length <= 200
- 1 <= board[i].length <= 200

问题分析

回溯算法实际上一个类似枚举的搜索尝试过程，也就是一个个去试，我们解这道题也是通过一个个去试，下面就用示例1来画个图看一下



我们看到他是从矩形中的一个点开始往他的上下左右四个方向查找，这个点可以是矩形中的任何一个点，所以代码的大致轮廓我们应该能写出来，就是遍历矩形所有的点，然后从这个点开始往他的4个方向走，因为是二维数组，所以有两个for循环，代码如下

```
public boolean exist(char[][] board, String word) {
  char[] words = word.toCharArray();
  for (int i = 0; i < board.length; i++) {
    for (int j = 0; j < board[0].length; j++) {
      //从[i,j]这个坐标开始查找
      if (dfs(board, words, i, j, 0))
        return true;
    }
  }
  return false;
}
```

这里关键代码是dfs这个函数，因为每一个点我们都可以往他的4个方向查找，所以我们可以把它想象为一棵4叉树，就是每个节点有4个子节点，而树的遍历我们最容易想到的就是递归，我们来大概看一下

```
boolean dfs(char[][] board, char[] word, int i, int j, int index) {
  if (边界条件的判断) {
    return;
  }

  一些逻辑处理

  boolean res;
  //往右
  res = dfs(board, word, i + 1, j, index + 1)
  //往左
  res |= dfs(board, word, i - 1, j, index + 1)
  //往下
  res |= dfs(board, word, i, j + 1, index + 1)
  //往上
  res |= dfs(board, word, i, j - 1, index + 1)
  //上面4个方向，只要有一个能查找到，就返回true;
  return res;
}
```

最终的完整代码如下

```
1   boolean dfs(char[][] board, char[] word, int i, int j, int index) {
2     //边界的判断，如果越界直接返回false。index表示的是查找到字符串word的第几个字符，
3     //如果这个字符不等于board[i][j]，说明验证这个坐标路径是走不通的，直接返回false
4     if (i >= board.length || i < 0 || j >= board[0].length || j < 0 || board[i][j] != word[index])
5       return false;
6     //如果word的每个字符都查找完了，直接返回true
7     if (index == word.length - 1)
8       return true;
9     //为了防止分支污染，把board数组复制一份
10    char[][] newArra = copyArray(board);
11    //把newArra[i][j]置为特殊符号，表示已经被使用过了(注意：word中不能包含'.')
12    newArra[i][j] = '.';
13    //从当前坐标的上下左右四个方向查找
14    boolean res = dfs(newArra, word, i + 1, j, index + 1) || dfs(newArra, word, i - 1, j, index + 1) ||
15        dfs(newArra, word, i, j + 1, index + 1) || dfs(newArra, word, i, j - 1, index + 1);
16    return res;
17  }
18
19  //复制一份新的数组
20  private char[][] copyArray(char[][] word) {
21    char[][] newArray = new char[word.length][word[0].length];
22    for (int i = 0; i < word.length; i++) {
23      for (int j = 0; j < word[0].length; j++) {
24        newArray[i][j] = word[i][j];
25      }
26    }
27    return newArray;
28  }
```

这里在第10行是新建了一个数组，因为一般来说数组都是引用传递，当我们在一个分支修改了数组之后，其他分支上的数据也会改变，这也就造成了分支污染。所以在递归往下传递的时候我们都会新建一个数组，这样在当前分支的修改并不会影响到其他的分支，也就不会出错。

这样虽然也能解决问题，但每次递归传递的时候都要创建一个新的数组，会造成大量的空间浪费，并且每次都创建也非常耗时，所以一般我们都不会使用上面的方式。我们会使用另外一个方法，也就是**回溯**。那么回溯又是如何解决这个问题的呢，要想弄懂回溯我们首先要搞懂递归，递归分为两步，先是递，然后才是归。当我们沿着当前坐标往下传递的时候，我们可以把当前坐标的值修改，然后回归到当前坐标的时候再把当前坐标的值复原，这就是回溯的过程。我们来看下代码，比上面简洁了好多，运行效率也会有很大的提升。

```
1   boolean dfs(char[][] board, char[] word, int i, int j, int index) {
2     //边界的判断，如果越界直接返回false。index表示的是查找到字符串word的第几个字符，
3     //如果这个字符不等于board[i][j]，说明验证这个坐标路径是走不通的，直接返回false
4     if (i >= board.length || i < 0 || j >= board[0].length || j < 0 || board[i][j] != word[index])
5       return false;
6     //如果word的每个字符都查找完了，直接返回true
7     if (index == word.length - 1)
8       return true;
9     //把当前坐标的值保存下来，为了在最后复原
10    char tmp = board[i][j];
11    //然后修改当前坐标的值
12    board[i][j] = '.';
13    //走递归，沿着当前坐标的上下左右4个方向查找
14    boolean res = dfs(board, word, i + 1, j, index + 1) || dfs(board, word, i - 1, j, index + 1) ||
15        dfs(board, word, i, j + 1, index + 1) || dfs(board, word, i, j - 1, index + 1);
16    //递归之后再把当前的坐标复原
17    board[i][j] = tmp;
18    return res;
19  }
```

总结

回溯往往会伴随着递归，要想搞懂回溯，必须要搞懂递归，搞懂了递归，回溯就很容易理解了。其实递归我们可以把它抽象为一棵N叉树的遍历，递归的过程也就是沿着子节点走下去的过程，并且递归必须要有终止条件，不能无限制的递归下去。

往期推荐

- 419，剑指 Offer-旋转数组的最小数字
- 418，剑指 Offer-斐波那契数列
- 416，剑指 Offer-用两个栈实现队列
- 410，剑指 Offer-从尾到头打印链表

391，回溯算法求组合问题

原创 山大王wld 数据结构和算法 6月30日

收录于话题
#算法图文分析

96个 >



Faith can move mountains.

信念能战胜一切。

问题一

给定一个无重复元素的数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。

说明：

所有数字（包括 target）都是正整数。

解集不能包含重复的组合。

candidates 中的数字可以无限制重复被选取。

示例 1:

```
输入: candidates = [2,3,6,7]
target = 7
```

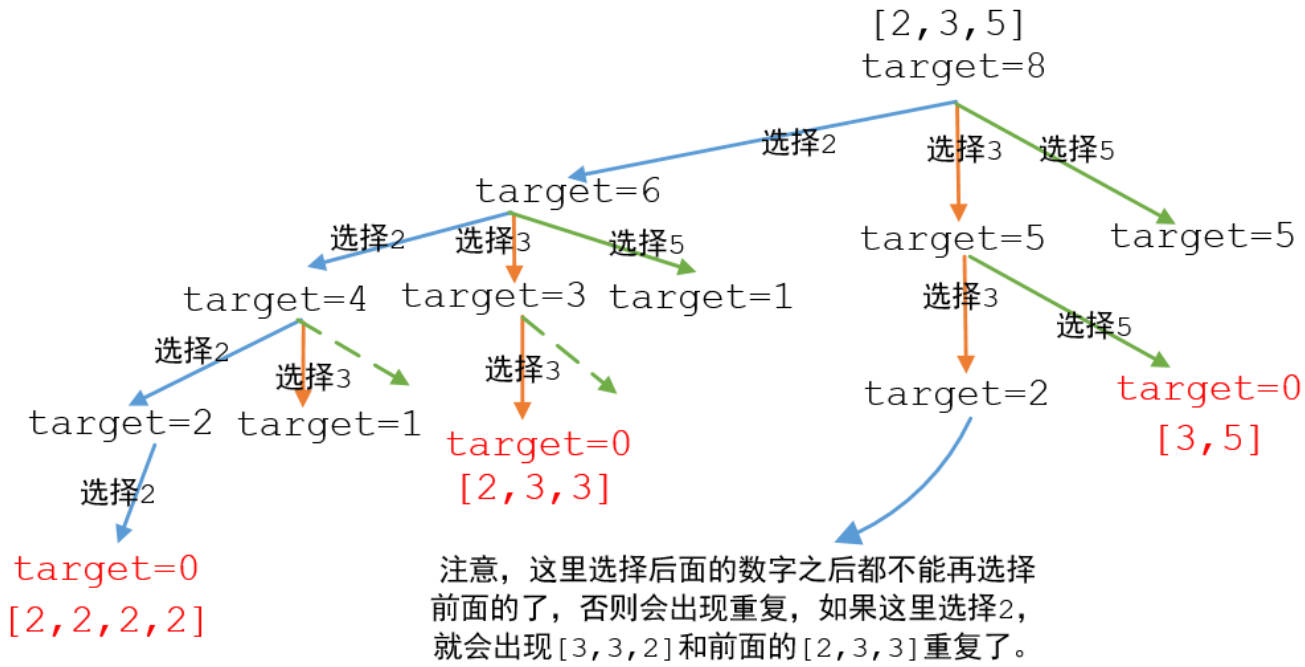
```
所求解集为:
[
  [7],
  [2,2,3]
]
```

示例 2:

```
输入: candidates = [2,3,5]
target = 8

所求解集为:
[
  [2,2,2,2],
  [2,3,3],
  [3,5]
]
```

问题分析

文中说的很明白，从candidates中找到一些数字让他们的和等于target，总共有多少种方式，并且candidates中的数字可以重复使用。我们可以先选择一个数字，用target减去他，然后再重复选择……，当target等于0的时候说明我们找到了一种组合。当target小于0的时候，说明没有找到合适的，我们回到上一步再重新选择数字……。看到这题我们首先想到的是N叉树，我们就以示例2为例画个图来看下



这和二叉树的前序遍历非常相似，他先从根节点一直往左走，直到走到叶子节点为止，然后再回到父节点按同样的方式走右节点的路径，不了解前序遍历的可以看一下373，数据结构-6,树，代码如下

```
public void preOrder(TreeNode tree) {
  if (tree == null)
    return;
  System.out.println(tree.val);
  preOrder(tree.left);
  preOrder(tree.right);
}
```

而N叉树的前序遍历和他类似

```
public void preOrder(TreeNode tree) {
  if (tree == null)
    return;
  System.out.println(tree.val);
  preOrder("第一个子节点");
  preOrder("第二个子节点");
  ……
  preOrder("第N个子节点");
}
```

这样写也是可以的，但不方便，我们一般会使用一个for循环来写

```
public void preOrder(TreeNode root) {
  if (root == null)
    return;
  System.out.println(root.val);
  //root.children获取root节点的所有子节点
  for (int i = 0; i < root.children.size(); i++) {
    preOrder(root.children.get(i));
  }
}
```

搞懂了上面的分析过程，代码就简单多了，我们来看下

```
public static List<List<Integer>> combinationSum(int[] candidates, int target) {
  List<List<Integer>> result = new ArrayList<>();
  backtrack(result, new ArrayList<>(), candidates, target);
  return result;
}

private static void backtrack(List<List<Integer>> result, List<Integer> cur, int candidates[], in
  if (target == 0) {
    result.add(new ArrayList<>(cur));
    return;
  }
  //相当于遍历N叉树的子节点
  for (int i = 0; i < candidates.length; i++) {
    //如果当前节点大于target我们就不要选了
    if (target < candidates[i])
      continue;
    //由于在java中List是引用传递，所以这里要重新创建一个
    List<Integer> list = new ArrayList<>(cur);
    list.add(candidates[i]);
    backtrack(result, list, candidates, target - candidates[i]);
  }
}
```

我们来看下运行结果

```
[2, 2, 2, 2]
[2, 3, 3]
```

```
[3, 2, 3]
[3, 3, 2]
[3, 5]
[5, 3]
```

完全出乎我们的意料之外，这是因为出现了重复的数据，[2,3,3],[3,2,3],[3,3,2]其实应该只算一个。在上面的图中我们分析过，如果选择了后面的数字就不能再选择前面的了，因为这样会出现重复，所以我们可以添加一个变量start表示访问的数组中元素的位置，我们只能访问start和start后面的数字，我们再来看下代码

```
1  public static List<List<Integer>> combinationSum(int[] candidates, int target) {
2    List<List<Integer>> result = new ArrayList<>();
3    backtrack(result, new ArrayList<>(), candidates, target, 0);
4    return result;
5  }
6
7  private static void backtrack(List<List<Integer>> result, List<Integer> cur, int candidates[], in
8    if (target == 0) {
9      result.add(new ArrayList<>(cur));
10     return;
11   }
12   //相当于遍历N叉树的子节点
13   for (int i = start; i < candidates.length; i++) {
14     //如果当前节点大于target我们就不要选了
15     if (target < candidates[i])
16       continue;
17     //由于在java中List是引用传递，所以这里要重新创建一个
18     List<Integer> list = new ArrayList<>(cur);
19     list.add(candidates[i]);
20     backtrack(result, list, candidates, target - candidates[i], i);
21   }
22 }
```

注意这里第13行的for循环不是从0开始了，再来看下运行结果

```
[2, 2, 2, 2]
[2, 3, 3]
[3, 5]
```

和我们图中分析的完全一致，并且也没有了重复的。在上面的18行我们是新建了一个list，其实我们还可以不用新建，在回溯的时候把它移除即可，可以这样写

```
1  public static List<List<Integer>> combinationSum(int[] candidates, int target) {
2    List<List<Integer>> result = new ArrayList<>();
3    getResult(result, new ArrayList<>(), candidates, target, 0);
4    return result;
5  }
6
7
8  private static void getResult(List<List<Integer>> result, List<Integer> cur, int candidates[], in
9    if (target == 0) {
10     result.add(new ArrayList<>(cur));
11     return;
12   }
13   for (int i = start; i < candidates.length; i++) {
14     if (target < candidates[i])
15       continue;
```

```
        //选择当前节点，类似于从当前节点开始往下遍历
        cur.add(candidates[i]);
        getResult(result, cur, candidates, target - candidates[i], i);
        //回到当前节点的时候我们把当前节点给移除，
        // 然后通过循环走同一层的其他节点。
        //举个例子，比如上面图中，最开始的时候
        // 我们先选择2，然后沿着这个分支走下去，
        //当回到当前分支的时候我们把2给移除，然后
        // 选择同一层的下一个3，沿着这个节点
        //分支走下去……
        cur.remove(cur.size() - 1);
    }
}
```

搞懂了上面的题，我们再来看一个非常相似的题

问题二

给定一个数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。

说明：

所有数字（包括目标数）都是正整数。

解集不能包含重复的组合。

candidates 中的每个数字在每个组合中只能使用一次。

示例 1:

```
输入: candidates = [10,1,2,7,6,1,5]
 target = 8

所求解集为:
[
  [1, 7],
  [1, 2, 5],
  [2, 6],
  [1, 1, 6]
]
```

示例 2:

```
输入: candidates = [2,5,2,1,2]
 target = 5

所求解集为:
[
```

```
  [1,2,2],
  [5]
]
```

问题分析

这道题和第一道题不同的是，这道题数组中的数字只能被选择一次，而第一道题数组中的数字可以被选中无数次。解题思想还是一样的，我们要做的是怎么过滤掉重复的，首先可以对原数组进行排序，排序之后相同的肯定是挨着的，if（candidates[i] == candidates[i - 1]）我们就过滤掉candidates[i]，我们就仿照第一道题来写下这道题的答案

```
1   public List<List<Integer>> combinationSum2(int[] candidates, int target) {
2     List<List<Integer>> list = new LinkedList<>();
3     Arrays.sort(candidates);//先排序
4     backtrack(list, new ArrayList<>(), candidates, target, 0);
5     return list;
6   }
7
8   private void backtrack(List<List<Integer>> list, List<Integer> cur, int[] candidates, int target,
9     if (target == 0) {
10      list.add(new ArrayList<>(cur));
11      return;
12    }
13    for (int i = start; i < candidates.length; i++) {
14      if (target < candidates[i])
15        continue;
16      if (i > start && candidates[i] == candidates[i - 1])
17        continue; //去掉重复的
18      cur.add(candidates[i]);
19      backtrack(list, cur, candidates, target - candidates[i], i + 1);
20      cur.remove(cur.size() - 1);
21    }
22  }
```

我们发现和第一道题很相似，只不过在第3行先做了排序，第16-17行做了去重，由于不能选择重复的，所以在第19行选择一个之后我们从当前元素的下一个进行选择

总结

类似这样的题我们可以把它想象为一棵N叉树，我们先选择一个，然后再递归选择（根据是否可以选择重复的有不同的选择，如果允许有重复的，我们递归的时候还可以再选择当前的，如果不允许有重复的，我们递归的时候就从当前的下一个开始选择），沿着这个分支走完之后我们会把当前节点删除，然后再从下一个分支走……

600，贪心算法解救生艇问题

原创 博哥 数据结构和算法 前天

问题描述

来源：LeetCode第881题
难度：中等

第i个人的体重为people[i]，每艘船可以承载的最大重量为limit。每艘船最多可同时载两人，但条件是这些人的重量之和最多为limit。

返回载到每一个人所需的最小船数。(保证每个人都能被船载)。

示例 1：

> **输入**：people = [1,2], limit = 3
> **输出**：1
> **解释**：1 艘船载 (1, 2)

示例 2：

> **输入**：people = [3,2,2,1], limit = 3
> **输出**：3
> **解释**：3 艘船分别载 (1, 2), (2) 和 (3)

示例 3：

> **输入**：people = [3,5,3,4], limit = 5
> **输出**：4
> **解释**：4 艘船分别载 (3), (3), (4), (5)

提示：

- 1 <= people.length <= 50000
- 1 <= people[i] <= limit <= 30000

贪心算法解决

题中说了每艘船最多可同时载两人，要让船最少，就应该让载两人的船最多。解题思路就是先对数组进行排序，我们优先考虑体重最大的。

- 如果体重最大的和体重最小的可以同时乘坐一艘船，就让他俩乘坐一条船。
- 如果体重最大的和体重最小的不能同时乘坐一艘船，那么体重最大的和其他任何人都不可能同时乘坐一条船，我们只能给体重最大的单独分配一条船。

原理比较简单，我们来看下代码。

```
public int numRescueBoats(int[] people, int limit) {
    //先对数组进行排序（人发重量按照从小到大的顺序排序）
    Arrays.sort(people);
    //统计船的个数
    int count = 0;
    //从小到大排序，左边的是最小的，右边的是最大的
    int left = 0;
    int right = people.length - 1;
    while (left <= right) {
        //如果体重最大的和体重最小的可以单独乘坐
        //一条船，就让他们同乘一条船
        if (people[right] + people[left] <= limit)
            left++;
        //体重最大的每次都要减1
        right--;
        //使用船的数量
        count++;
    }
    return count;
}
```

时间复杂度：O（nlog(n)），这个是排序使用的复杂度。

空间复杂度：O（log(n)），排序使用的空间复杂度，这个需要看Arrays.sort的源码，他调用的是DualPivotQuicksort类中的方法。

往期推荐

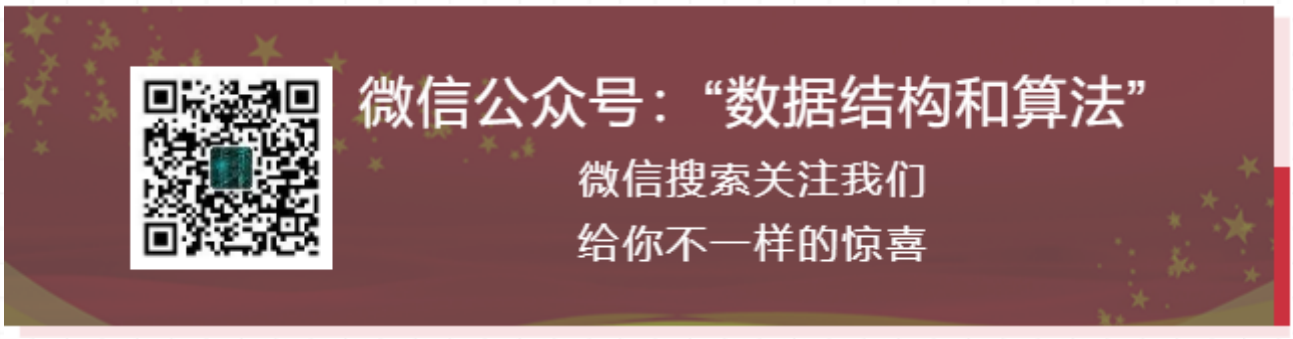
- 516，贪心算法解按要求补齐数组
- 505，分发糖果（贪心算法解决）
- 501，贪心算法解分发饼干
- 492，动态规划和贪心算法解买卖股票的最佳时机 II

516，贪心算法解按要求补齐数组

原创 山大王wld 数据结构和算法 2月8日

收录于话题
#算法图文分析

137个 >



When the lord closes a door, somewhere he opens a window.

当主关上了一扇门，就会在别处打开一扇窗。

问题描述

给定一个已排序的正整数数组nums，和一个正整数n。从[1，n]区间内选取任意个数字补充到nums中，使得[1,n]区间内的任何数字都可以用 nums中某几个数字的和来表示。请输出满足上述要求的最少需要补充的数字个数。

示例 1:

> **输入**: nums=[1,3],n=6
> **输出**:1
> **解释**:
> 根据nums里现有的组合[1],[3],[1,3]，可以得出1,3,4。
> 现在如果我们将 2 添加到 nums 中， 组合变为: [1], [2], [3], [1,3], [2,3], [1,2,3]。
> 其和可以表示数字1,2,3,4,5,6，能够覆盖[1,6]区间里所有的数。
> 所以我们最少需要添加一个数字。

示例 2:

> **输入**:nums=[1,5,10],n=20
> **输出**:2
> **解释**:我们需要添加[2,4]。

示例 3:

> **输入**:nums=[1,2,2],n=5
> **输出**:0

贪心算法解决

这题让从数组中找出任意数字都可以组成n，题中说了，数组是排序的。

假设数组中前k个数字能组成的数字范围是[1,total]，当我们添加数组中第k+1个数字nums[k](数组的下标是从0开始的)的时候，范围就变成了[1,total]U[nums[k],nums[k]]U[1+nums[k]，total+nums[k]]，这是个并集，可以合并成[1,total]U[nums[k],total+nums[k]]，我们仔细观察一下

1，如果左边的total<nums[k]-1，那么他们中间肯定会有空缺，不能构成完整的[1，total+nums[k]]。
举个例子
[1,5]U[7,10]，因为5<7-1，所以是没法构成[1,10]的

这个时候我们需要添加一个数字total+1。先构成一个更大的范围[1，total*2+1]。
这里为什么是添加total+1而不是添加total，我举个例子，比如可以构成的数字范围是[1,5]，如果需要添加一个构成更大范围的，我们应该选择6而不是选择5。

2，如果左边的total>=nums[k]-1，那么就可以构成完整的[1，total+nums[k]]，就不需要在添加数字了。

来看下代码

```
1   public int minPatches(int[] nums, int n) {
2       //累加的总和
3       long total = 0;
4       //需要补充的数字个数
5       int count = 0;
6       //访问的数组下标索引
7       int index = 0;
8       while (total < n) {
```

```
        if (index < nums.length && nums[index] <= total + 1) {
            //如果数组能组成的数字范围是[1,total]，那么加上nums[index]
            //就变成了[1,total]U[nums[index],total+nums[index]]
            //结果就是[1,total+nums[index]]
            total += nums[index++];
        } else {
            //添加一个新数字，并且count加1
            total = total + (total + 1);
            count++;
        }
    }
    return count;
}
```

另一种写法

上面组成数字的范围是闭区间，我们还可以改成开区间 [1, total) ，原理都一样，稍作修改即可，代码如下

```
public int minPatches(int[] nums, int n) {
    //累加的总和
    long total = 1;
    //需要补充的数字个数
    int count = 0;
    //访问的数组下标索引
    int index = 0;
    while (total <= n) {
        if (index < nums.length && nums[index] <= total) {
            //如果数组能组成的数字范围是[1,total)，那么加上nums[index]
            //就变成了[1,total)U[nums[index],total+nums[index])
            //结果就是[1,total+nums[index])
            total += nums[index++];
        } else {
            //添加一个新数字，并且count加1
            total <<= 1;
            count++;
        }
    }
    return count;
}
```


往期推荐

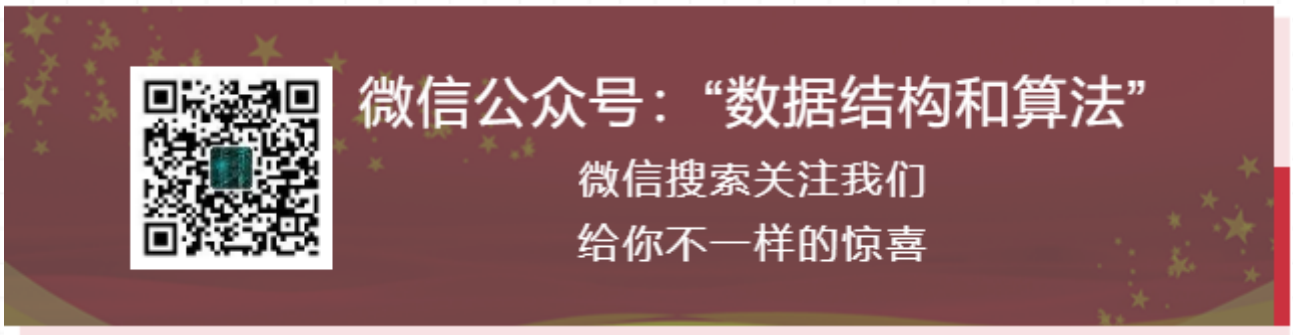
- 505，分发糖果（贪心算法解决）
- 501，贪心算法解分发饼干
- 492，动态规划和贪心算法解买卖股票的最佳时机 II
- 413，动态规划求最长上升子序列

505，分发糖果（贪心算法解决）

原创 山大王wld 数据结构和算法 1月11日

收录于话题
#算法图文分析

137个 >



For small creatures such as we the vastness is bearable only through love.

如此渺小的我们，只有通过爱，才能承受宇宙的广漠。

问题描述

老师想给孩子们分发糖果，有*N*个孩子站成了一条直线，老师会根据每个孩子的表现，预先给他们评分。

你需要按照以下要求，帮助老师给这些孩子分发糖果：

- 每个孩子至少分配到1个糖果。
- 评分更高的孩子必须比他两侧的邻位孩子获得更多的糖果。

那么这样下来，老师至少需要准备多少颗糖果呢？

示例 1：

> **输入**：[1,0,2]
> **输出**：5
> **解释**：你可以分别给这三个孩子分发2、1、2颗糖果。

示例 2：

> **输入**：[1,2,2]
> **输出**：4
> **解释**：你可以分别给这三个孩子分发1、2、1颗糖果。
> 第三个孩子只得到1颗糖果，这已满足上述两个条件。

左右两次遍历

这题有3个条件

1，每个孩子至少有一个糖果

2，相邻的孩子，得分高的会有更多的糖果

3，至少需要多少糖果

第一步：要满足第1个和第3个条件，只需要让所有的孩子都有1个糖果即可。

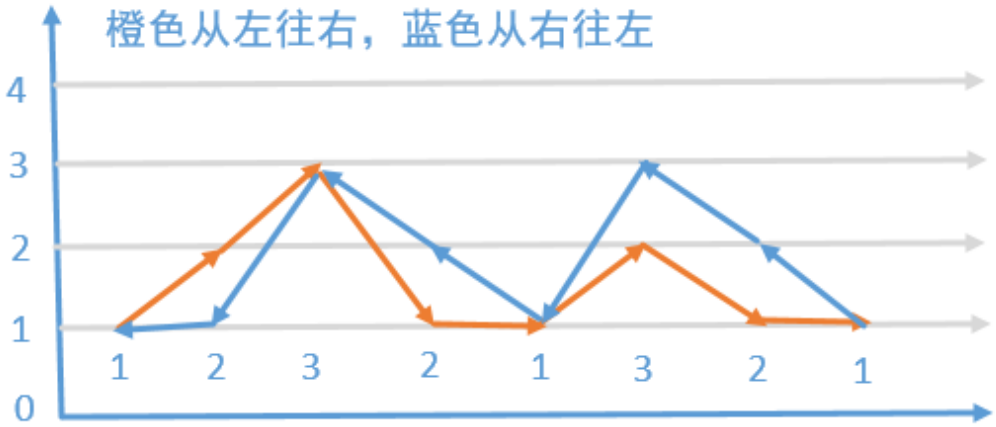
第二步：要满足第2个和第3个条件

- 如果只比左边的值大，那么当前孩子的糖果数量要比左边孩子的糖果多1。
- 如果只比右边的值大，那么当前孩子的糖果数量要比右边孩子的糖果多1。
- 如果比左右两边都大，那么当前孩子的糖果数量要比左右两边最多的糖果还要多1。

要满足上面的第二步，可以使用两次遍历，第一次从左往右，如果右边的孩子比左边得分高，那么右边的孩子糖果数量就要比左边的多1。这样每个孩子都能满足右边的条件，就是右边比他得分高的都会比他多一个。满足了右边的条件之后我们还要满足左边的条件，原理都一样，我们随便举个例子画个图来看一下

原始数据

1	3	6	4	2	7	2	1



1	3	6	4	2	7	2	1

原理比较简单，来看下代码

```java
public int candy(int[] ratings) {
    int length = ratings.length;
    //记录的是从左往右循环的结果
    int[] left = new int[length];
    //记录的是从右往左循环的结果
    int[] right = new int[length];
    //因为每个孩子至少有一个糖果，默认都给他们一个
    Arrays.fill(left, 1);
    Arrays.fill(right, 1);
    //统计最少的总共糖果数量
    int total = 0;
    //先从左往右遍历，如果当前孩子的得分比左边的高，
    //那么当前孩子的糖果要比左边孩子的多一个
    for (int i = 1; i < length; i++) {
        if (ratings[i] > ratings[i - 1])
            left[i] = left[i - 1] + 1;
    }
    //然后再从右往左遍历，如果当前孩子的得分比右边的高，
    //那么当前孩子的糖果要比右边边孩子的多一个
    for (int i = length - 1; i > 0; i--) {
        if (ratings[i - 1] > ratings[i])
            right[i - 1] = right[i] + 1;
    }
    //要满足左右两边的条件，那么当前孩子的糖果就要取
    //从左往右和从右往左的最大值。
    for (int i = 0; i < length; i++) {
        //累计每个孩子的糖果
        total += Math.max(left[i], right[i]);
    }
    return total;
}
```

其实还可以优化一下，在从右往左遍历的时候就可以统计total的值了，不需要再使用第3个for循环，代码如下

```java
public int candy(int[] ratings) {
    int length = ratings.length;
    int[] count = new int[length];
    //默认给每个孩子一个糖果
    Arrays.fill(count, 1);
    //先从左往右遍历
    for (int i = 1; i < length; i++) {
        if (ratings[i] > ratings[i - 1])
            count[i] = count[i - 1] + 1;
    }
    //因为下面的for循环中，total没有统计最后一个孩子的糖果，所以这里total
    //的默认值就是最后那个孩子的糖果数量
    int total = count[length - 1];
    //从右往左遍历，然后顺便累加total的值
    for (int i = length - 1; i > 0; i--) {
        if (ratings[i - 1] > ratings[i])
            count[i - 1] = Math.max(count[i - 1], count[i] + 1);
        total += count[i - 1];
    }
    return total;
}
```

从左到右一次遍历

我们还可以从左往右一次遍历来解决。如果从左往右一次遍历判断当前孩子糖果的数量，会有3种情况

1. 当前孩子的评分比左边的高，也就是递增的，那么当前孩子的糖果数量要比左边个多1。
2. 当前孩子的评分等于左边孩子的评分，我们让他降为1,也就是说当前孩子的糖果是1，最终是不是1,后面还需要在判断。
3. 当前孩子的评分低于左边孩子的评分，也就是递减的，这个我们就没法确定当前孩子的糖果了，但我们可以统计递减孩子的数量（我们可以反向思考，这个递减的序列中，最后一个肯定是1，并且从后往前都逐渐增加的）

叙述起来比较拗口，来举个例子，比如数组是[1,3,4,6,8,5,3,1]，8是得分的最高点，8前面的从1开始都是递增的，8后面的从5开始是递减的。也就是说8前面的从1到6，每个孩子的糖果数量分别是[1,2,3,4],那么8后面的从5到1每个孩子的糖果数量分别是[3,2,1]。那么8的糖果数量就是左右两边的最大值加1，也就是4+1=5。
所以8左边糖果的数量是(1+4)*4/2=10，右边的糖果数量是(1+3)*3/2=6，所以总的糖果数量是10+6+5=21。

上面的数组的单调性用图像来表示就是



但实际上数组不一定都是这样的，有可能是这样的，比如[2,5,7,4,3,1,5,7,8,6,4,3]



这种也没关系，我们可以把它拆成[2,5,7,4,3,1]和[1,5,7,8,6,4,3]两个子数组来分别计算。（注意这里的1被分到了两个子数组中，也就是说会被计算两次），我们来看下代码

```
public int candy(int[] ratings) {
  int length = ratings.length;
  if (length == 1)
    return 1;
  //记录访问到哪个孩子了
  int index = 0;
  //记录总共的糖果
  int total = 0;
  while (index < length - 1) {
    int left = 0;
    //统计递增的长度
    while (index < length - 1 && ratings[index + 1] > ratings[index]) {
      index++;
      left++;
    }
    int right = 0;
    //统计递减的长度
    while (index < length - 1 && ratings[index + 1] < ratings[index]) {
      index++;
      right++;
    }
    //记录顶点的值，也就是左右两边最大的值加1
    int peekCount = Math.max(left, right) + 1;
    //注意这里如果total不等于0要减1，是因为把数组拆分成子数组的时候，低谷的那
    //个会被拆到两个数组中，如果total不等于0，说明之前已经统计过，而下面会再次
    //统计，所以要提前减去。
    if (total != 0)
      total--;
    //当前这个子数组的糖果数量就是前面递增的加上后面递减的然后在加上顶点的。
    total += (1 + left) * left / 2 + (1 + right) * right / 2 + peekCount;
    //如果当前孩子的得分和前一个一样，我们让他降为1
    while (index < length - 1 && ratings[index + 1] == ratings[index]) {
      index++;
      total++;
    }
  }
  return total;
}
```

总结

贪心算法和动态规划不一样，贪心算法只需要在每步做出局部最优解即可。对于这道题来说，如果右边的得分比当前的高，那么右边的糖果只需要比当前的糖果多一个即可，否则就让右边的糖果变为1，这样能满足题目的要求，并且所需要的糖果最少，同理和右边比完之后还要和左边在比较。

501，贪心算法解分发饼干

原创 山大王wld 数据结构和算法 今天

收录于话题
#算法图文分析

111个 >



> But in times of crisis the wise build bridges, while the foolish build barriers.
>
> 危机四伏之时，智者筑桥，愚者设障。

问题描述

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。

对每个孩子i，都有一个胃口值g[i]，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干j，都有一个尺寸s[j]。如果s[j]>=g[i]，我们可以将这个饼干j分配给孩子i，这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

示例 1:

> **输入**: g = [1,2,3], s = [1,1]
> **输出**: 1
> **解释**:
> 你有三个孩子和两块小饼干，3个孩子的胃口值分别是：1,2,3。
> 虽然你有两块小饼干，由于他们的尺寸都是1，你只能让胃口值是1的孩子满足。

> 所以你应该输出1。

示例 2:

> **输入**: g = [1,2], s = [1,2,3]
> **输出**: 2
> **解释**:
> 你有两个孩子和三块小饼干，2个孩子的胃口值分别是1,2。
> 你拥有的饼干数量和尺寸都足以让所有孩子满足。
> 所以你应该输出2.

提示:

1<=g.length<=3*10^4

0<=s.length<=3*10^4

1<=g[i], s[j]<=2^31-1

贪心算法解决

贪心算法（又称贪婪算法）是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，算法得到的是在某种意义上的局部最优解。

《算法导论》中对贪心算法是这样描述的

> 贪心算法（greedy algorithm）就是这样的算法，它在每一步都做出当时看起来最佳的选择。也就是说，它总是做出局部最优的选择，寄希望这样的选择能导致全局最优解。

对于这道题来说是最适合使用贪心算法的，题中说了要尽可能满足更多的孩子，因为饼干不能掰开，所以这道题我们可以让胃口大的吃大块，胃口小的吃小块。一种最简单的方式就是先从胃口最小的孩子开始，拿最小的饼干试一下能不能满足他，如果能满足就更好，如果不能满足，在找稍微大一点的，如果还不能满足就再找更大一点的……

```
public int findContentChildren(int[] g, int[] s) {
  //先对胃口值和饼干尺寸进行排序
  Arrays.sort(g);
  Arrays.sort(s);
  int count = 0;
  for (int j = 0; count < g.length && j < s.length; j++) {
    //如果当前饼干能满足当前孩子的胃口值，count就加1，否则就继续查找更大的饼干
    if (g[count] <= s[j])
      count++;
  }
  return count;
}
```

还一种方式就是先从最大的饼干开始，看一下能不能满足胃口最大的，如果不能满足就找胃口稍微小一点是再试一下，如果还不能满足就一直找……

```java
public int findContentChildren(int[] g, int[] s) {
    //先对胃口值和饼干尺寸进行排序
    Arrays.sort(g);
    Arrays.sort(s);
    int count = 0;
    int i = s.length - 1;
    for (int j = g.length - 1; i >= 0 && j >= 0; j--) {
        //如果当前饼干能满足当前孩子的胃口值，count就加1，否则就继续查找胃口更小的孩子
        if (g[j] <= s[i]) {
            count++;
            i--;
        }
    }
    return count;
}
```

总结

贪心算法只需要满足局部最优解，他只能确定某些问题的可行性范围，不能保证解是最佳的。因为贪心算法总是从局部出发，并没从整体考虑，对于有些问题使用贪心算法是可以的，有些是不可以的，这些都要根据具体问题具体分析。

往期推荐

- 494，位运算解只出现一次的数字
- 491，回溯算法解将数组拆分成斐波那契序列
- 413，动态规划求最长上升子序列
- 397，双指针求接雨水问题

489，柠檬水找零

原创 山大王wld 数据结构和算法 2020-12-12

收录于话题
#算法图文分析

111个 >



Maybe things didn't work out, because there's something better out there for you.

或许有时候会失败，是因为有更好的事在等着你。

问题描述

在柠檬水摊上，每一杯柠檬水的售价为5美元。
顾客排队购买你的产品，（按账单bills支付的顺序）一次购买一杯。
每位顾客只买一杯柠檬水，然后向你付5美元、10美元或20美元。你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付5美元。

注意，一开始你手头没有任何零钱。
如果你能给每位顾客正确找零，返回true ，否则返回false 。

示例 1：

> 输入：[5,5,5,10,20]
> 输出：true
> 解释：

> 前 3 位顾客那里，我们按顺序收取 3 张 5 美元的钞票。
> 第 4 位顾客那里，我们收取一张 10 美元的钞票，并返还 5 美元。
> 第 5 位顾客那里，我们找还一张 10 美元的钞票和一张 5 美元的钞票。
> 由于所有客户都得到了正确的找零，所以我们输出 true。

示例 2：

> **输入**：[5,5,10]
> **输出**：true

示例 3：

> **输入**：[10,10]
> **输出**：false

示例 4：

> **输入**：[5,5,10,10,20]
> **输出**：false
> **解释**：
> 前 2 位顾客那里，我们按顺序收取 2 张 5 美元的钞票。
> 对于接下来的 2 位顾客，我们收取一张 10 美元的钞票，然后返还 5 美元。
> 对于最后一位顾客，我们无法退回 15 美元，因为我们现在只有两张 10 美元的钞票。
> 由于不是每位顾客都得到了正确的找零，所以答案是 false。

提示：

- `0 <= bills.length <= 10000`
- `bills[i]` 不是 5 就是 10 或是 20

问题分析

这道题算是生活中很常见的一道题，对于每一个顾客如果我们都有足够的零钱给他找零，那么就返回true，只要有一个顾客没有足够的零钱找给他就返回false。

顾客只能有3种纸币，5元，10元，20元。我们要统计5元和10元的数量，20元的不需要统计，因为20元没法找给别人。

- 顾客给5元，5元的数量加1

- 顾客给10元，5元的数量减1（减完之后再判断5元的数量，如果小于0，说明5元的不够了，没法给顾客找零了，直接返回false）
- 顾客给20元，根据生活常识，如果有10元的，应该先找他10元的，然后再找他一个5元的。如果没有10元的就找他3个5元的，然后再判断5元的数量，如果小于0直接返回false。

原理比较简单，我们来看下代码

```java
public boolean lemonadeChange(int[] bills) {
    //统计店员所拥有的5元和10元的数量（20元的不需要统计，
    //因为顾客只能使用5元，10元和20元，而20元是没法
    // 给顾客找零的）
    int five = 0, ten = 0;
    for (int bill : bills) {
        if (bill == 5) {
            //如果顾客使用的是5元，不用找零，5元数量加1
            five++;
        } else if (bill == 10) {
            //如果顾客使用的是10元，需要找他5元，所以
            //5元数量减1,10元数量加1
            five--;
            ten++;
        } else if (ten > 0) {
            //否则顾客使用的只能是20元，顾客使用20元的时候，
            //如果我们有10元的，要尽量先给他10元的，然后再
            //给他5元的，所以这里5元和10元数量都要减1
            ten--;
            five--;
        } else {
            //如果顾客使用的是20元，而店员没有10元的，
            //就只能给他找3个5元的，所以5元的数量要减3
            five -= 3;
        }

        //上面我们找零的时候并没有判断5元的数量，如果5元的
        //数量小于0，说明上面某一步找零的时候5元的不够了，
        //也就是说没法给顾客找零，直接返回false即可
        if (five < 0) {
            return false;
        }
    }
    return true;
}
```

总结

一道生活常识问题，找零的时候我们并没有先判断5元的数量，而找完之后再判断5元的数量是否是大于0还是小于0。如果在找零的时候先判断5元的数量也是可以的，只不过稍微有一点点麻烦，因为顾客只要给的不是5元的都要先判断。

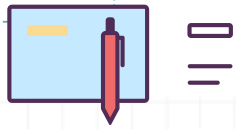
往期推荐

- 486，动态规划解最大子序和
- 481，用最少数量的箭引爆气球

589，DFS和BFS解从根到叶的二进制数之和

原创 博哥 数据结构和算法 8月2日

> Confidence grows with success.
>
> 越成功越自信。



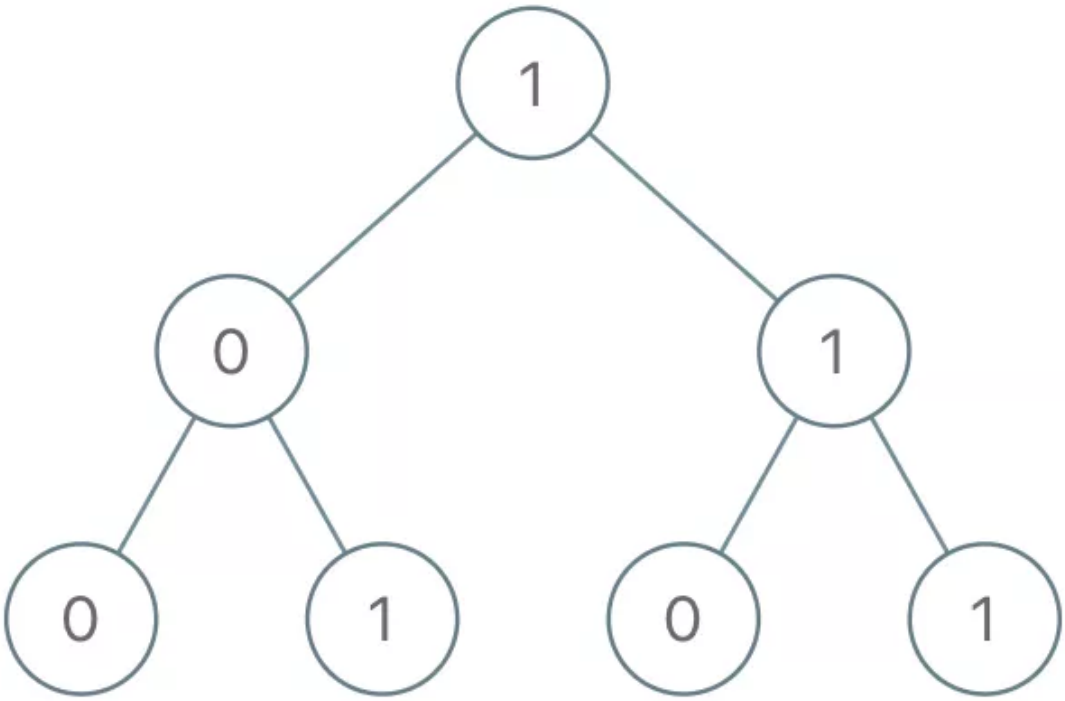
问题描述

来源：LeetCode第1022题
难度：简单

给出一棵二叉树，其上每个结点的值都是0或1。每一条从根到叶的路径都代表一个从最高有效位开始的二进制数。例如，如果路径为0->1->1->0->1，那么它表示二进制数01101，也就是13。对树上的每一片叶子，我们都要找出从根到该叶子的路径所表示的数字。

返回这些数字之和。题目数据保证答案是一个32位整数。

示例 1：



> **输入**：root = [1,0,1,0,1,0,1]
>
> **输出**：22
> **解释**：(100) + (101) + (110) + (111) = 4 + 5 + 6 + 7 = 22

示例 2：

> **输入**：root = [0]
>
> **输出**：0

示例 3：

> **输入**：root = [1]
> **输出**：1

示例 4：

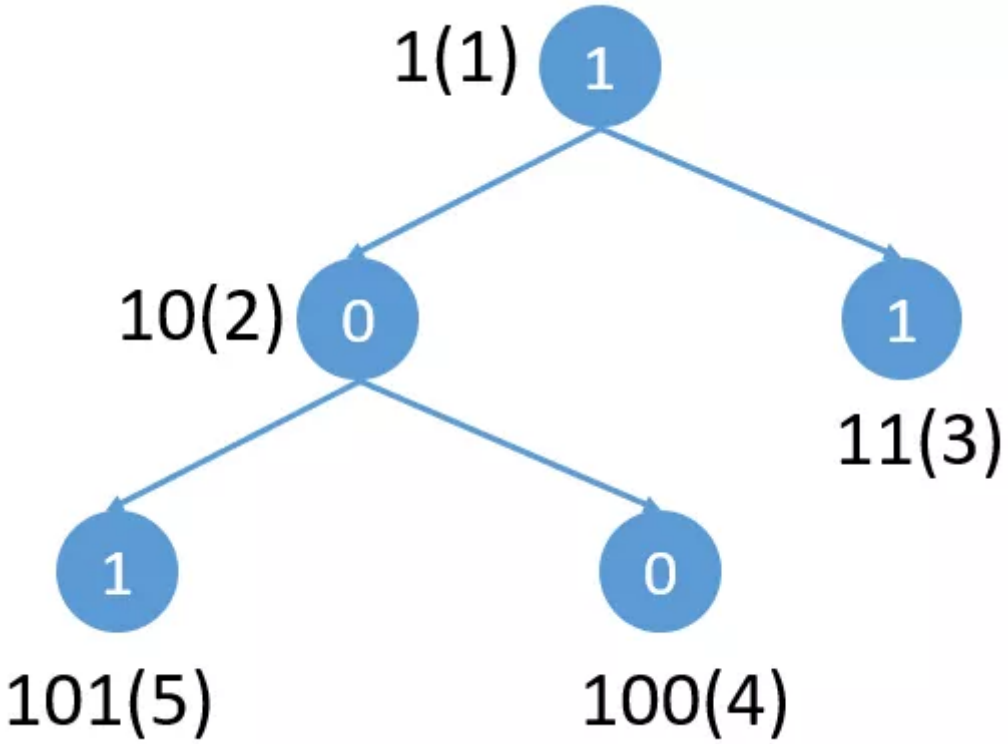
> **输入**：root = [1,1]
> **输出**：3

提示：

- 树中的结点数介于1和1000之间。
- Node.val为0或1。

DFS解决

这里我们先画个图来看一下



我们可以看到从根节点到当前节点这条路径的值就是**父节点的值*2加上当前节点的值。**

我们定义一个全局的变量res，他就是所有从根节点到叶子节点表示数字的和。

我们可以通过前序遍历来解这道题，当遇到叶子节点的时候就把从根节点到当前叶子节点表示的数字加到res中。直接把二叉树的前序遍历方式修改一下即可。

```
//最终返回的数字
int res = 0;

public int sumRootToLeaf(TreeNode root) {
  dfs(root, 0);
  return res;
}

//parentPathSum表示从根节点到当前父节点这条路径表示的数字
public void dfs(TreeNode root, int parentPathSum) {
  //如果节点为空，直接返回
  if (root == null)
    return;
  //父节点的值*2，在加上当前节点的值就是从根节点到
  //当前节点这条路径表示的数字
  int sum = parentPathSum * 2 + root.val;
  //如果到叶子节点，说明找到了一个从根节点到叶子
  //节点的完整路径，把这条路径的值加到res中
  if (root.left == null && root.right == null) {
    res += sum;
    return;
  }
```

```
    //如果没到叶子节点就继续遍历当前节点的左子节点和右子节点
    dfs(root.left, sum);
    dfs(root.right, sum);
}
```

时间复杂度：O(N)，N是节点的个数，所有节点都要访问一遍

空间复杂度：O(H)，H是树的最大高度，也是栈的深度

BFS解决

除了DFS，我们还可以使用BFS来解决，DFS就是深度优先搜索，BFS就是广度优先搜索，具体也可以看下373，数据结构-6,树。BFS就是一层一层的访问。这里需要使用两个队列：

- **一个存放节点**
- **一个存放从根节点到当前节点这条路径表示的数字**

如果访问到叶子节点的时候就把表示的数字加入到res中，最后返回res即可，我们来看下代码。

```
public int sumRootToLeaf(TreeNode root) {
    int res = 0;//结果值
    //两个队列，一个存放节点，一个存放从根节点到当前
    //节点的父节点这条路径所表示的数字
    Queue<TreeNode> queueNode = new LinkedList<>();
    Queue<Integer> queueParentSum = new LinkedList<>();
    queueNode.add(root);
    queueParentSum.add(0);
    while (!queueNode.isEmpty()) {
        TreeNode cur = queueNode.poll();
        int parentSum = queueParentSum.poll();
        //计算从根节点到当前节点这条路径表示的数字
        int sum = parentSum * 2 + cur.val;
        //如果当前节点是叶子节点，就把sum加到res中
        if (cur.left == null && cur.right == null) {
            res += sum;
            continue;
        }
        //如果左子节点不为空，就把他和他对应的值分别加入
        //到对应的队列中
        if (cur.left != null) {
            queueNode.add(cur.left);
            queueParentSum.add(sum);
        }
        //右子节点同上
        if (cur.right != null) {
            queueNode.add(cur.right);
            queueParentSum.add(sum);
        }
    }
    return res;
}
```

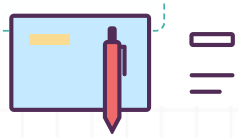
时间复杂度：O(N)，N是节点的个数，所有节点都要访问一遍。

空间复杂度：O(N)，这里使用了两个队列，因为队列中元素不停的进和出，最差情况下是满二叉树，到叶子节点的时候每个队列使用的空间是整颗树节点的一半(N/2)

586，BFS和DFS解层数最深叶子节点的和

原创 博哥 数据结构和算法 1周前

> Even with two eyes, you only see half of the picture.
>
> 即便亲眼所见，也无法窥得全貌。

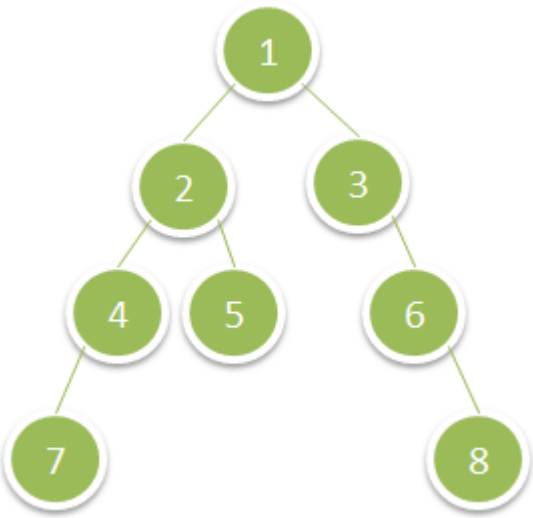


问题描述

来源：LeetCode第1302题
难度：中等

给你一棵二叉树的根节点root，请你返回层数最深的叶子节点的和。

示例 1：



> **输入**：root = [1,2,3,4,5,null,6,7,null,null,null,null,8]
> **输出**：15

示例 2：

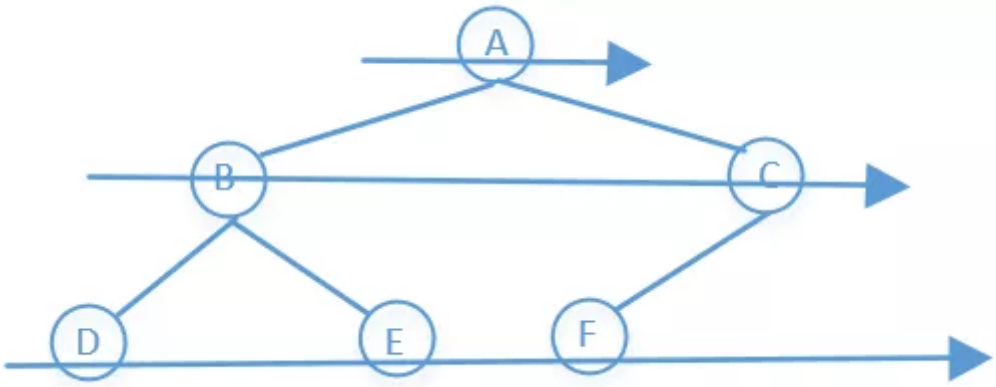
> **输入**：root = [6,7,8,2,7,1,3,9,null,1,4,null,null,null,5]
>
> **输出**：19

提示：

- 树中节点数目在范围 [1, 10^4] 之间。
- 1 <= Node.val <= 100

BFS解决

这题让求的是最深叶子节点的和，最容易想到的就BFS解决，也就是一层一层的从上往下遍历，BFS的遍历方式如下，也可以看下[373，数据结构-6,树]



到最后一层的时候我们再把节点的值相加即可。这里我们不知道哪一层是最深的层数，我们可以把每一层所有节点的值都相加，下一层会把上一次的给覆盖掉，最后一层下面没有了，所以没法覆盖，直接返回即可。来看下代码

```
public int deepestLeavesSum(TreeNode root) {
  //每层节点的和
  int res = 0;
  //队列
  Queue<TreeNode> queue = new LinkedList<>();
  queue.add(root);
  //队列不为空，继续访问队列的元素
  while (!queue.isEmpty()) {
    //当前层的节点数量
    int levelCount = queue.size();
    //当前层所有节点值的和
    res = 0;
    //遍历当前层的所有节点
    for (int j = 0; j < levelCount; j++) {
      TreeNode node = queue.poll();
      //累加当前层节点的值
      res += node.val;
      //如果左子节点不为空就把他加入到队列中
      if (node.left != null)
        queue.add(node.left);
      //如果右子节点不为空就把他加入到队列中
      if (node.right != null)
        queue.add(node.right);
    }
  }
  return res;
}
```

时间复杂度：O(N)，N是节点的个数，所有节点都要访问一遍

空间复杂度：O(N)，队列中存放的是每层节点的个数，满二叉树的时候最后一层节点的个数是（N+1）/2

DFS解决

这题除了BFS从上到下一层一层的访问以外，我们还可以使用DFS来解决，这里以前序遍历为例，往下遍历的时候需要计算节点的层数

- 如果当前节点的层数大于之前记录的最大层数，说明之前记录的层数还不是最大的，那之前累加的和sum肯定要作废，重新赋值为0，最大层数也要更新。
- 如果当前层数等于记录的最大层数，sum就累加。

来看下代码

```
//记录树的最大深度
private int maxLevel = 0;
//记录最后一层所有节点的和
private int sum = 0;

public int deepestLeavesSum(TreeNode root) {
  dfs(root, 0);
  return sum;
}

//level表示第几层，根节点是第0层
private void dfs(TreeNode root, int level) {
  //边界条件判断，如果是空直接返回
  if (root == null)
    return;
  //操作当前节点，如果没到最后一层，记录
  //当前访问过的最大层数，并且把sum重置为0，
  //也就是之前加的作废
  if (level > maxLevel) {
    maxLevel = level;
    sum = 0;
  }
  //如果到了最后一层，就把节点值相加
  if (level == maxLevel) {
    sum = sum + root.val;
  }
  //访问左子节点和右子节点，
  dfs(root.left, level + 1);
  dfs(root.right, level + 1);
}
```

时间复杂度：O(N)，N是树的所有节点个数

空间复杂度：O(H)，H是树的最大深度

往期推荐

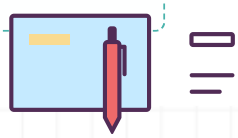
- 582，DFS解二叉树剪枝
- 574，DFS和BFS解单词拆分
- 507，BFS和DFS解二叉树的层序遍历 II
- 488，二叉树的Morris中序和前序遍历

580，BFS和DFS解二叉树的堂兄弟节点

原创 博哥 数据结构和算法 7月12日

You are never wrong to do the right thing.

坚持做对的事，你永远不会错。



问题描述

来源：LeetCode第993题
难度：简单

在二叉树中，根节点位于深度0处，每个深度为k的节点的子节点位于深度k+1处。

如果二叉树的两个节点深度相同，但父节点不同，则它们是一对堂兄弟节点。

我们给出了具有唯一值的二叉树的根节点root，以及树中两个不同节点的值x和y。

只有与值x和y对应的节点是堂兄弟节点时，才返回true。否则，返回false。

示例 1：

> **输入**：root = [1,2,3,4],
> x = 4, y = 3
>
> **输出**：false

示例 2：

> **输入**：root = [1,2,3,null,4,null,5],
> x = 5, y = 4
>
> **输出**：true

示例 3：

> **输入**：root = [1,2,3,null,4],
> x = 2, y = 3
>
> **输出**：false

提示：

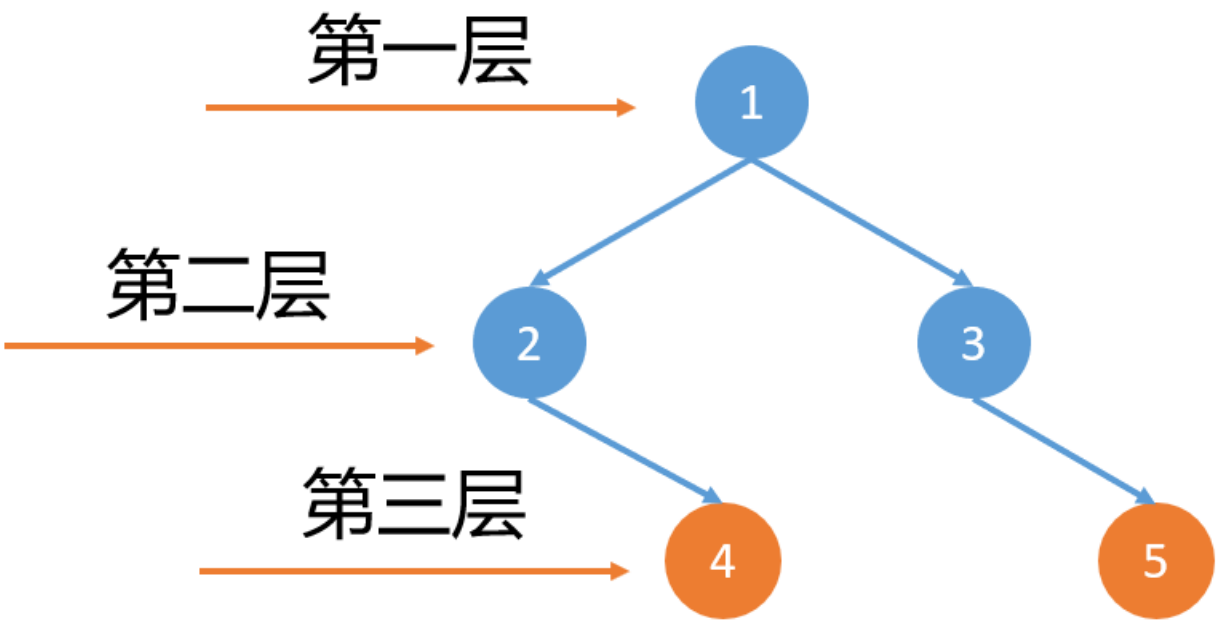
- 二叉树的节点数介于2到100之间。
- 每个节点的值都是唯一的、范围为1到100的整数。

BFS解决

BFS就是宽度优先搜索，一层一层的遍历。这题使用BFS是最容易想到的，BFS遍历首先会使用一个队列，把每一层的节点都加入到队列中，对于同一层的节点我们可以认为他们辈分是一样的，要么是亲兄弟节点，要么是堂兄弟节点。

首先判断两个节点值是否是亲兄弟：

- **如果是就返回false。**
- **如果不是，在判断他们是否都在同一层，如果在同一层，说明他们是堂兄弟节点，返回true。**
- **否则遍历完了，说明他们不在同一层，不是堂兄弟节点。**




```java
public boolean isCousins(TreeNode root, int x, int y) {
    //两个队列一个存放树的节点，一个存放节点对应的值
    Queue<TreeNode> queue = new LinkedList<>();
    Queue<Integer> value = new LinkedList<>();
    queue.add(root);
    value.add(root.val);
    //如果队列不为空，说明树的节点没有遍历完，就继续遍历
```

```
    while (!queue.isEmpty()) {
      //BFS是从上到下一层一层的打印，levelSize表示
      //当前层的节点个数
      int levelSize = queue.size();
      for (int i = 0; i < levelSize; i++) {
        //节点和节点值同时出队
        TreeNode poll = queue.poll();
        value.poll();
        //首先判断x和y是否是兄弟节点的值，也就是判断他们的父节点
        //是否是同一个
        if (poll.left != null && poll.right != null) {
          //如果是亲兄弟节点，直接返回false
          if ((poll.left.val == x && poll.right.val == y) ||
              (poll.left.val == y && poll.right.val == x)) {
            return false;
          }
        }
        //左子节点不为空加入到队列中
        if (poll.left != null) {
          queue.offer(poll.left);
          value.offer(poll.left.val);
        }
        //右子节点不为空加入到队列中
        if (poll.right != null) {
          queue.offer(poll.right);
          value.offer(poll.right.val);
        }
      }
      //判断当前层是否包含这两个节点的值，如果包含就是堂兄弟节点
      if (value.contains(x) && value.contains(y))
        return true;
    }
    return false;
  }
```

时间复杂度：O(n)，n是节点的个数，最差情况下遍历到最后一层。

空间复杂度：O(n)，使用两个队列，队列中一个存放的是节点，一个存放的是节点的值。

DFS解决

前序遍历，中序遍历，和后续遍历都可以认为是DFS。我们遍历的时候如果找到x节点或者y节点，就**记录下他们的父节点和深度**，最后再判断。

- **如果深度不一样，说明不在同一层，那么他俩的辈分就不一样，肯定不是堂兄弟节点。**
- **如果深度一样，说明在同一层，是同一辈的，但还要判断他们的父亲是否是同一个，如果是同一个说明他俩是亲兄弟，否则就是堂兄弟。**

代码如下

```
  private TreeNode xParent = null;//x的父节点
  private TreeNode yParent = null;//y的父节点
  private int xDepth = -1;//x的深度
  private int yDepth = -2;//y的深度

  public boolean isCousins(TreeNode root, int x, int y) {
    dfs(root, null, x, y, 0);
    //如果他俩的深度一样，也就是在同一层，又不是同一个父亲，那么他俩
```

```
    //就是堂兄弟节点，否则不是
    return xDepth == yDepth && xParent != yParent ? true : false;
  }

  public void dfs(TreeNode root, TreeNode parent, int x, int y, int depth) {
    if (root == null)
      return;
    if (root.val == x) {
      //如果找到了x节点，就把他的父节点和深度记录下来
      xParent = parent;
      xDepth = depth;
    } else if (root.val == y) {
      //如果找到了y节点，就把他的父节点和深度记录下来
      yParent = parent;
      yDepth = depth;
    }
    //如果确定他俩是堂兄弟节点了，直接返回，不用再往下遍历了
    if (xDepth == yDepth && xParent != yParent)
      return;
    dfs(root.left, root, x, y, depth + 1);
    dfs(root.right, root, x, y, depth + 1);
  }
```

时间复杂度：O(n)，n是节点的个数，最差情况下遍历所有节点。

空间复杂度：O(n)，栈的深度，最坏情况下二叉树退化为链表形状。

往期推荐

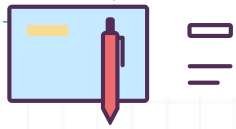
- 574，DFS和BFS解单词拆分
- 532，BFS解打开转盘锁
- 531，BFS和动态规划解完全平方数
- 473，BFS解单词接龙

574，DFS和BFS解单词拆分

原创 博哥 数据结构和算法 1周前

You will never age for me, nor fade, nor die.

你于我而言，无岁月之流失，无花容之褪色，无人生之离别。



问题描述

给定一个非空字符串s和一个包含非空单词的列表wordDict，判定s是否可以被空格拆分为一个或多个在字典中出现的单词。

说明：

- **拆分时可以重复使用字典中的单词。**
- **你可以假设字典中没有重复的单词。**

示例 1：

> **输入**:
> s = "leetcode",
> wordDict = ["leet", "code"]
>
> **输出**: true
> **解释**: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。

示例 2：

> **输入**:
> s = "applepenapple",
> wordDict = ["apple", "pen"]
>
> **输出**: true
> **解释**: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
> 注意你可以重复使用字典中的单词。

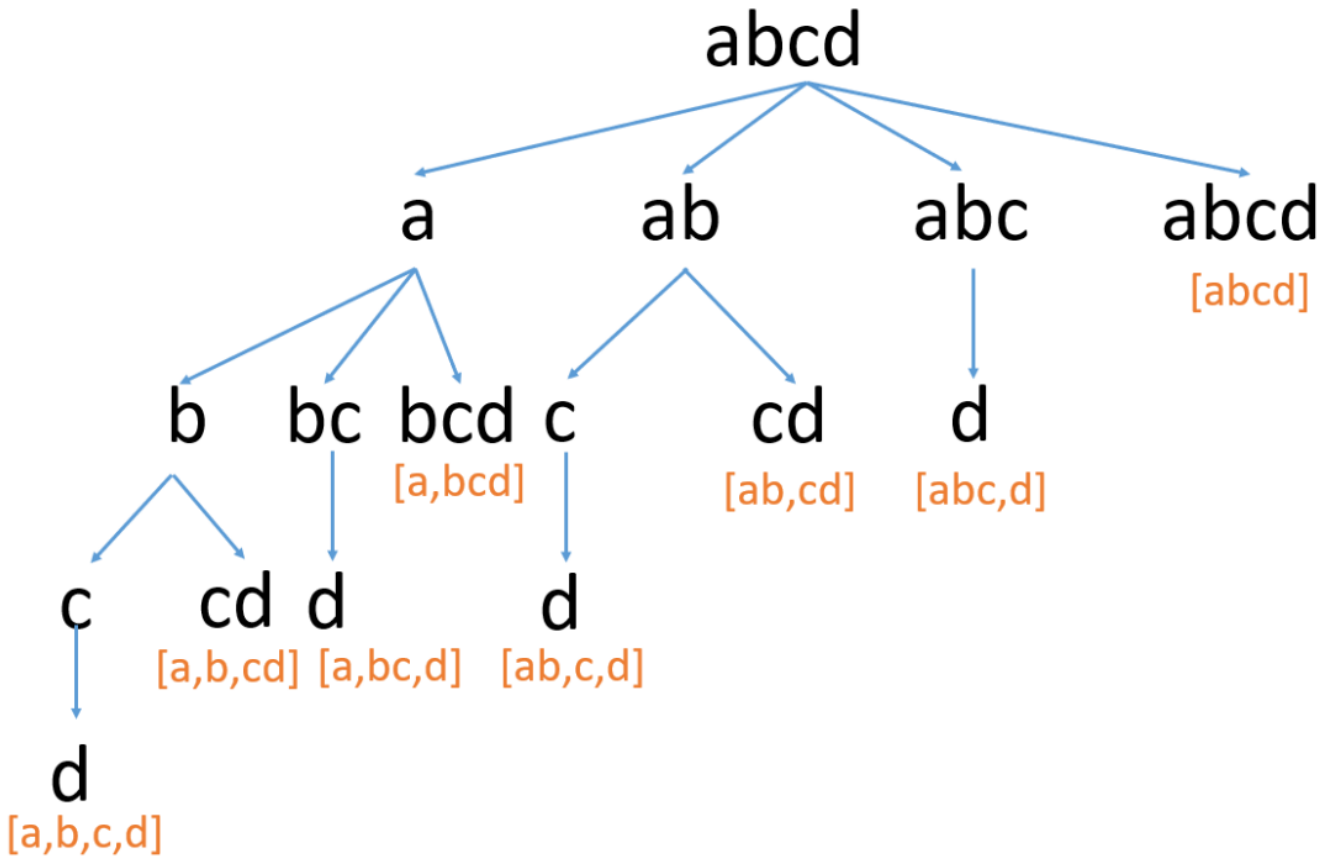
示例 3：

> **输入**:
> s = "catsandog",
> wordDict = ["cats", "dog", "sand", "and", "cat"]
>
> **输出**: false

DFS解决

前面刚讲过这题，使用的是动态规划，具体可以看下《573，动态规划解单词拆分》，今天我们分别使用DFS和BFS来解决这道题。

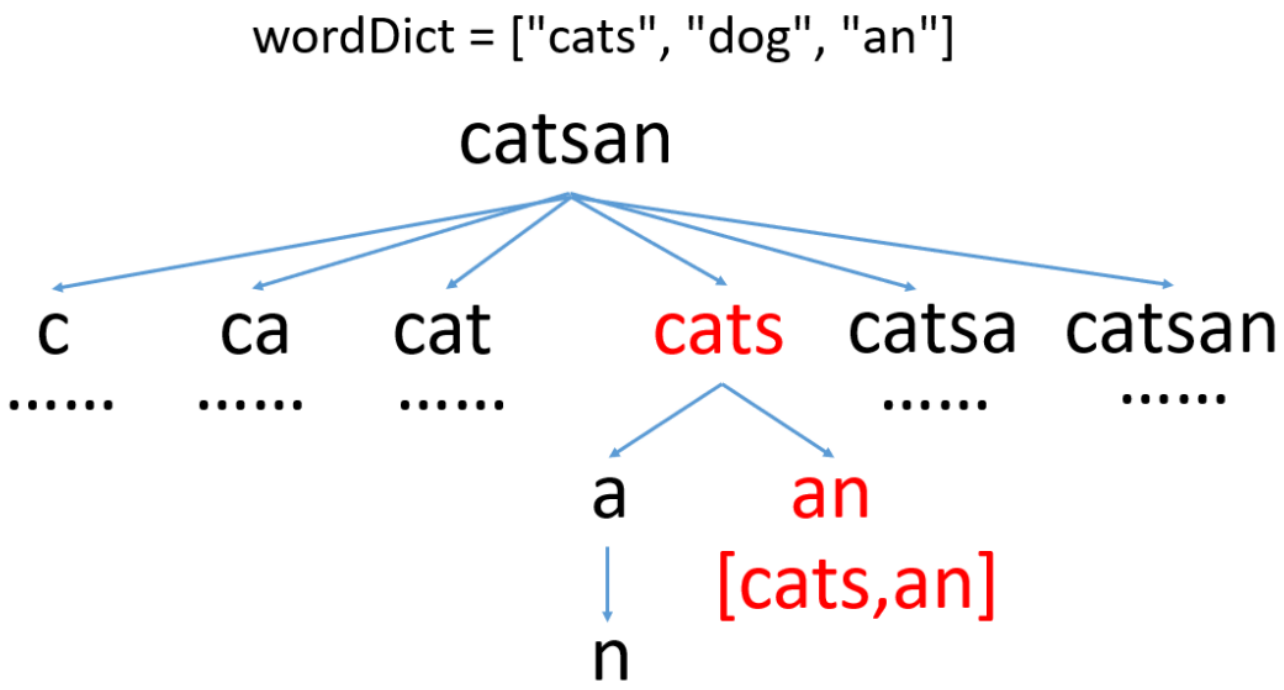
这题要求的是把字符串拆分，并且判断拆分的子串是否都存在于字典中，那么字符串怎么拆分呢，我们举个例子来看下，比如字符串[abcd]，我们可以拆分为
[a,b,c,d]
[a,b,cd]
[a,bc,d]
[a,bcd]
[ab,c,d]
[ab,cd]
[abc,d]
[abcd]
具体来看下图



每次截取一个子串，判断他是否存在于字典中，如果不存在于字典中，继续截取更长的子串……如果存在于字典中，然后递归拆分剩下的子串，这是一个递归的过程。上面的执行过程我们可以把它看做是一棵n叉树的DFS遍历，所以大致代码我们可以列出来

```
public boolean wordBreak(String s, List<String> wordDict) {
  return dfs(s, wordDict);
}

public boolean dfs(String s, List<String> wordDict) {
  if (最终条件，都截取完了，直接返回true)
  return true;
  //开始拆分字符串s
  for (int i = 开始截取的位置; i <= s.length(); i++) {
    //如果截取的子串不在字典中，继续截取更大的子串
    if (!wordDict.contains(截取子串))
      continue;
    //如果截取的子串在字典中，继续剩下的拆分，如果剩下的可以拆分成
    //在字典中出现的单词，直接返回true，如果不能则继续
    //截取更大的子串判断
    if (dfs(s, wordDict))
      return true;
  }
  //如果都不能正确拆分，直接返回false
  return false;
}
```

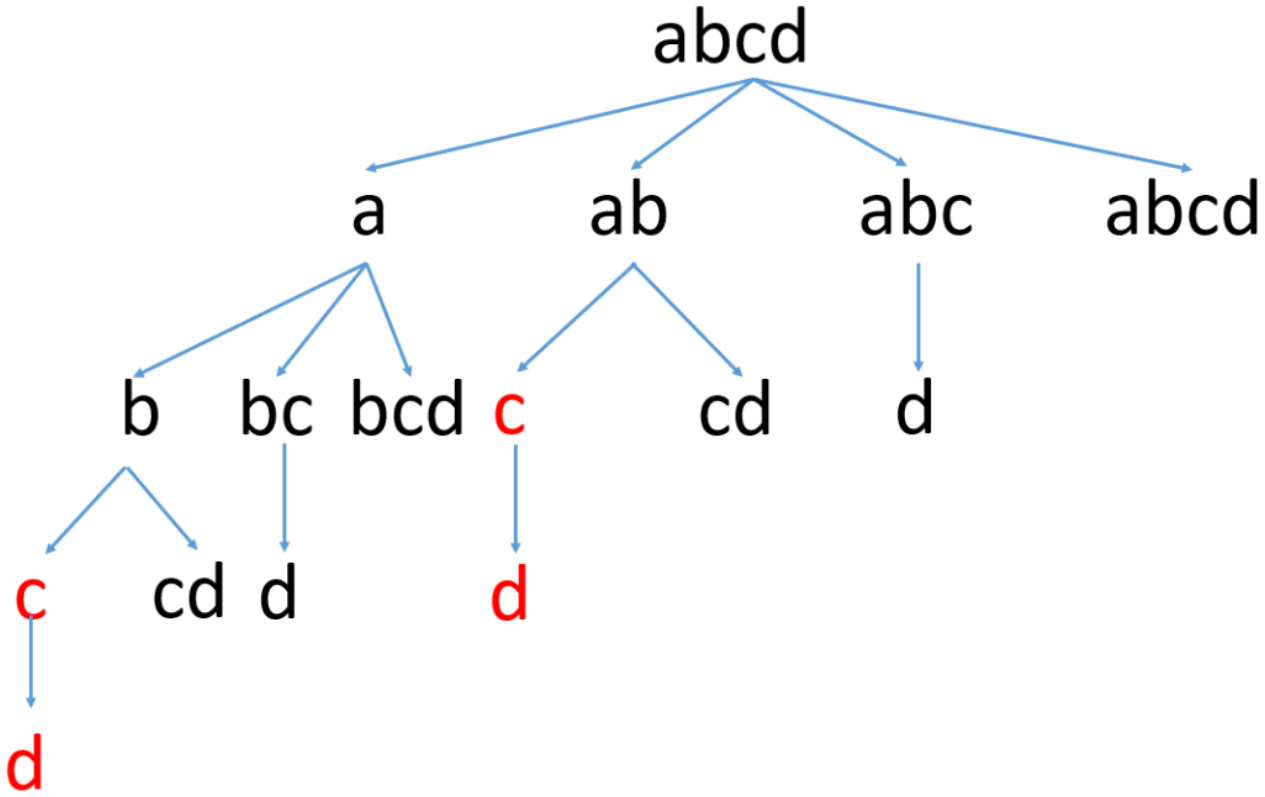
上面代码中因为递归必须要有终止条件，通过上面的图我们可以发现，终止条件就是把字符串s中的所有字符都遍历完了，这个时候说明字符串s可以拆分成一些子串，并且这些子串都存在于字典中。我们来看个图



因为是拆分，所以字符串截取的时候不能有重叠，那么[开始截取的位置]实际上就是上次截取位置的下一个，来看下代码。

```
public boolean wordBreak(String s, List<String> wordDict) {
  return dfs(s, wordDict, 0);
}

//start表示的是从字符串s的哪个位置开始
public boolean dfs(String s, List<String> wordDict, int start) {
  //字符串中的所有字符都遍历完了，也就是到叶子节点了，说明字符串s可以拆分成
  //在字典中出现的单词，直接返回true
  if (start == s.length())
    return true;
  //开始拆分字符串s，
  for (int i = start + 1; i <= s.length(); i++) {
    //如果截取的子串不在字典中，继续截取更大的子串
    if (!wordDict.contains(s.substring(start, i)))
      continue;
    //如果截取的子串在字典中，继续剩下的拆分，如果剩下的可以拆分成
    //在字典中出现的单词，直接返回true，如果不能则继续
    //截取更大的子串判断
    if (dfs(s, wordDict, i))
      return true;
  }
  return false;
}
```

实际上上面代码运行效率很差，这是因为如果字符串s比较长的话，这里会包含大量的重复计算，我们还用上面的图来看下

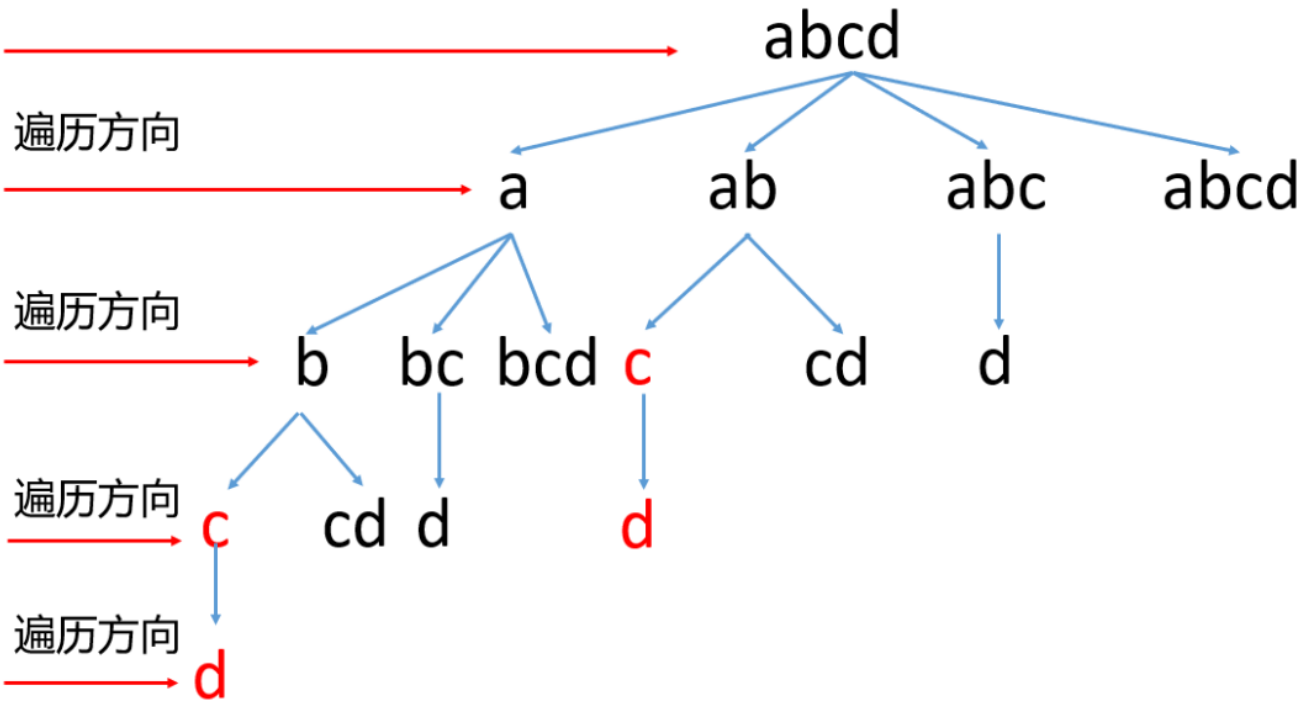


我们看到红色的就是重复计算，这里因为字符串比较短，不是很明显，当字符串比较长的时候，这里的重复计算非常多。我们可以使用一个变量，来记录计算过的位置，如果之前判断过，就不在重复判断，直接跳过即可，代码如下

```
1   public boolean wordBreak(String s, List<String> wordDict) {
2     return dfs(s, wordDict, new HashSet<>(), 0);
3   }
4
5   //start表示的是从字符串s的哪个位置开始
6   public boolean dfs(String s, List<String> wordDict, Set<Integer> indexSet, int start) {
7     //字符串都拆分完了，返回true
8     if (start == s.length())
9       return true;
10    for (int i = start + 1; i <= s.length(); i++) {
11      //如果已经判断过了，就直接跳过，防止重复判断
12      if (indexSet.contains(i))
13        continue;
14      //截取子串，判断是否是在字典中
15      if (wordDict.contains(s.substring(start, i))) {
16        if (dfs(s, wordDict, indexSet, i))
17          return true;
18        //标记为已判断过
19        indexSet.add(i);
20      }
21    }
22    return false;
23  }
```

BFS解决

这题除了DFS以外，还可以使用BFS，BFS就是一层一层的遍历，如下图所示



BFS一般不需要递归，只需要使用一个队列记录每一层需要记录的值即可。BFS中在截取的时候，如果截取的子串存在于字典中，我们就要记录截取的位置，到下一层的时候就从这个位置的下一个继续截取，来看下代码。

```
public boolean wordBreak(String s, List<String> wordDict) {
  //这里为了提高效率，把list转化为set，因为set的查找效率要比list高
  Set<String> setDict = new HashSet<>(wordDict);
  //记录当前层开始遍历字符串s的位置
  Queue<Integer> queue = new LinkedList<>();
  queue.add(0);
  int length = s.length();
  while (!queue.isEmpty()) {
    int index = queue.poll();
    //如果字符串到遍历完了，自己返回true
    if (index == length)
      return true;
    for (int i = index + 1; i <= length; i++) {
      if (setDict.contains(s.substring(index, i))) {
        queue.add(i);
      }
    }
  }
  return false;
}
```

这种也会出现重复计算的情况，所以这里我们也可以使用一个变量来记录下。

```
public boolean wordBreak(String s, List<String> wordDict) {
  //这里为了提高效率，把list转化为set，因为set的查找效率要比list高
  Set<String> setDict = new HashSet<>(wordDict);
  //记录当前层开始遍历字符串s的位置
  Queue<Integer> queue = new LinkedList<>();
  queue.add(0);
  int length = s.length();
  //记录访问过的位置，减少重复判断
  boolean[] visited = new boolean[length];
  while (!queue.isEmpty()) {
    int index = queue.poll();
    //如果字符串都遍历完了，直接返回true
    if (index == length)
      return true;
    //如果被访问过，则跳过
    if (visited[index])
      continue;
```

```
18        //标记为访问过
19        visited[index] = true;
20        for (int i = index + 1; i <= length; i++) {
21          if (setDict.contains(s.substring(index, i))) {
22            queue.add(i);
23          }
24        }
25      }
26      return false;
27    }
```

往期推荐

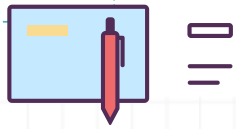
- 566，DFS解目标和问题
- 507，BFS和DFS解二叉树的层序遍历 II
- 470，DFS和BFS解合并二叉树
- 464. BFS和DFS解二叉树的所有路径

566，DFS解目标和问题

原创 博哥 数据结构和算法 6天前

We went through a lot, but we stayed together.

我们经历了各种风风雨雨，但依然团结一心。



问题描述

来源：LeetCode第494题

难度：中等

给你一个整数数组nums和一个整数target。

向数组中的每个整数前添加'+'或'-'，然后串联起所有整数，可以构造一个表达式：

- 例如，nums=[2,1]，可以在2之前添加'+'，在1之前添加'-'，然后串联起来得到表达式"+2-1"。

返回可以通过上述方法构造的、运算结果等于target的不同表达式的数目。

示例 1：

> **输入**：nums = [1,1,1,1,1], target = 3
> **输出**：5
> **解释**：一共有 5 种方法让最终目标和为 3 。
> -1 + 1 + 1 + 1 + 1 = 3
> +1 - 1 + 1 + 1 + 1 = 3
> +1 + 1 - 1 + 1 + 1 = 3
> +1 + 1 + 1 - 1 + 1 = 3
> +1 + 1 + 1 + 1 - 1 = 3

示例 2：

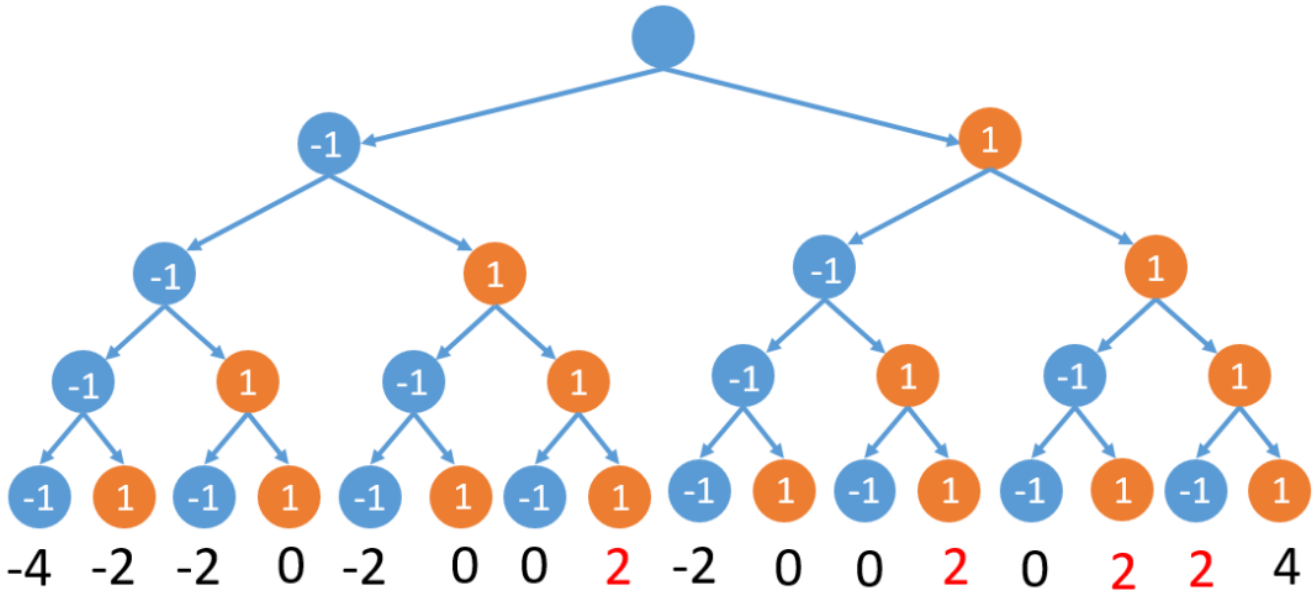
> **输入**：nums = [1], target = 1
> **输出**：1

提示：

- 1<=nums.length<=20
- 0<=nums[i]<=1000
- 0<=sum(nums[i])<=1000
- -1000<=target<=100

DFS解决

这题让在数组中每个元素前面添加'+'或'-'，组成算术表达式并且他的和等于target，问有多少种方式。每个元素只能有两种选择，要么添加'+'要么添加'-'，所以我们很容易想到的是二叉树。假如使用数组[1,1,1,1]中的所有元素通过'+'或'-'符号构成2，如下图所示



我们让每个节点的左子节点选择'-'，右子节点选择'+'，计算从根节点到叶子节点所有元素的和是否等于target，如果等于，说明找到了一个满足条件的表达式，只需要计算所有满足条件的个数即可，来看下代码

```
//不同表达式的数目
int count = 0;

public int findTargetSumWays(int[] nums, int target) {
    dfs(nums, target, 0, 0);
    return count;
}

//从根节点开始往下累加，到叶子节点的时候如果累加值等于target，
//说明找到了一条符合条件的表达式
private void dfs(int[] nums, int target, int sum, int index) {
    //判断从根节点到当前叶子节点这条路径是否走完了
    if (index == nums.length) {
        //如果当前累加值等于target，说明找到了一条符号条件的表达式
        if (target == sum)
            count++;
        return;
    }
    //左子树数负数，要减去
```

```
    dfs(nums, target, sum - nums[index], index + 1);
    //右子树是正数，要加上
    dfs(nums, target, sum + nums[index], index + 1);
  }
```

这题我们还可以修改一下，上面计算的时候是从根节点到叶子节点的累加，其实我们还可以从根节点到叶子节点往下减，根节点默认值是target，到叶子节点计算完的时候如果值为0，说明找到了一个满足条件的表达式，原理都差不多，来看下代码

```
//不同表达式的数目
int count = 0;

public int findTargetSumWays(int[] nums, int target) {
  dfs(nums, target, 0);
  return count;
}

//从更节点开始往下减
private void dfs(int[] nums, int target, int index) {
  //判断当前路径是否走完了
  if (index == nums.length) {
    //如果走完了，减到最后等于0，说明找到了一条符号条件的表达式
    if (target == 0)
      count++;
    return;
  }
  dfs(nums, target - nums[index], index + 1);
  dfs(nums, target + nums[index], index + 1);
}
```

或者还可以这样写，直接通过dfs返回

```
public int findTargetSumWays(int[] nums, int target) {
  return dfs(nums, target, 0);
}

private int dfs(int[] nums, int target, int index) {
  if (index == nums.length) {
    return target == 0 ? 1 : 0;
  }
  int res = 0;
  res += dfs(nums, target - nums[index], index + 1);
  res += dfs(nums, target + nums[index], index + 1);
  return res;
}
```

总结

这题还一种解决方式就是使用动态规划，解题思路和**组合总和 Ⅳ**以及**背包问题**很类似，太晚了这个先放到下次在讲。

这种类似的题估计大家很常见，就是连续的几个数通添加一些符号让他等于一个特定的值，但这题简化了，只能添加'+'或'-'，没有其他符号。

532，BFS解打开转盘锁

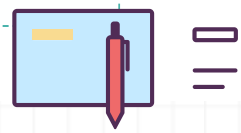
原创 博哥 数据结构和算法 3月30日

收录于话题
#算法图文分析

143个 >

You see, madness, as you know, is like gravity. All it takes is a little push!

疯狂就像地心引力，需要做的只是轻轻一推。



问题描述

你有一个带有四个圆形拨轮的转盘锁。每个拨轮都有10个数字：'0', '1', '2', '3', '4', '5', '6', '7', '8', '9' 。每个拨轮可以自由旋转：例如把 '9' 变为'0'，'0' 变为 '9' 。每次旋转都只能旋转一个拨轮的一位数字。

锁的初始数字为 '0000' ，一个代表四个拨轮的数字的字符串。

列表 deadends 包含了一组死亡数字，一旦拨轮的数字和列表里的任何一个元素相同，这个锁将会被永久锁定，无法再被旋转。

字符串target 代表可以解锁的数字，你需要给出最小的旋转次数，如果无论如何不能解锁，返回 -1。

示例 1:

> **输入**:
> deadends = ["0201","0101","0102","1212","2002"],
> target = "0202"
>
> **输出**: 6
> **解释**:
> 可能的移动序列为 "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202"。

> 注意 "0000" -> "0001" -> "0002" -> "0102" -> "0202" 这样的序列是不能解锁的，
> 因为当拨动到 "0102" 时这个锁就会被锁定。

示例 2:

> **输入**:
> deadends = ["8888"],
> target = "0009"
>
> **输出**: 1
> **解释**:
> 把最后一位反向旋转一次即可 "0000" -> "0009"。

示例 3:

> **输入**:
> deadends = ["8887","8889","8878","8898","8788","8988","7888","9888"],
> target = "8888"
>
> **输出**: -1
> **解释**:
> 无法旋转到目标数字且不被锁定。

示例 4:

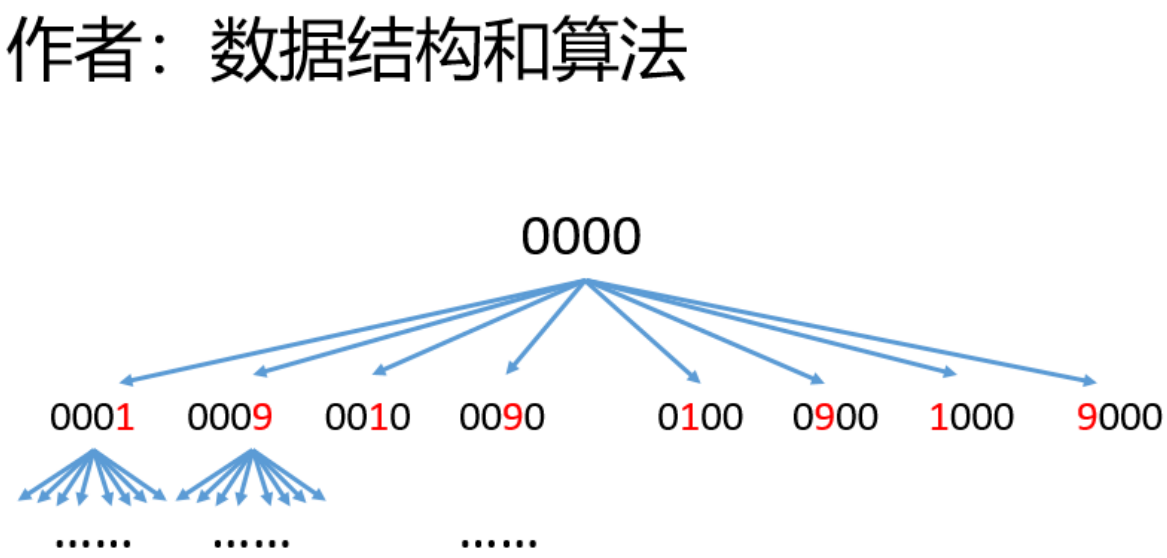
> **输入**:
> deadends = ["0000"],
> target = "8888"
>
> **输出**: -1

提示:

1. 死亡列表 deadends 的长度范围为 [1, 500]。
2. 目标数字 target 不会在 deadends 之中。
3. 每个 deadends 和 target 中的字符串的数字会在 10,000 个可能的情况 '0000' 到 '9999' 中产生。

BFS方式解决

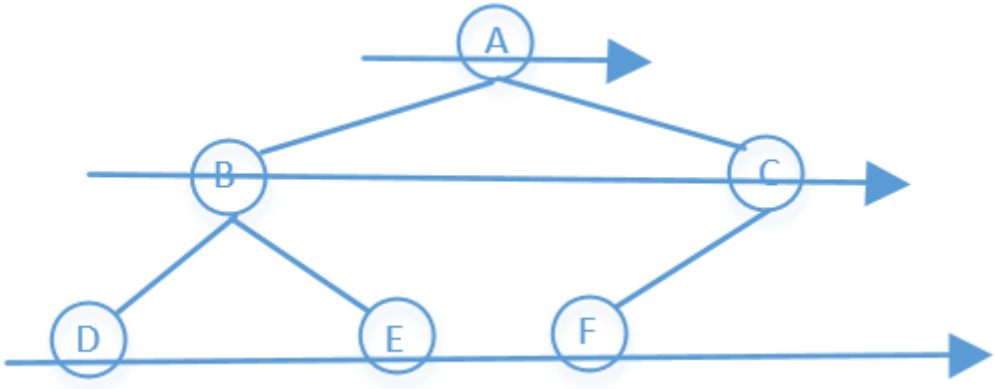
以字符串"0000"为起始点，把它的每一位都分别加1和减1，总共会有8个结果，如下图所示，细心的同学可能发现了，这不就是一棵8叉树吗，二叉树是有2个子节点，那么8叉树肯定就是8个子节点了。



- 这是一棵以"0000"为根节点的8叉树，我们一层一层的遍历他的每个节点，如果找到就返回他所在的层数即可，如果当前层遍历完了还没找到就遍历下一层，直到找到为止，如果都遍历完了还没找到就返回-1。所以我们很容易想到BFS
- 注意这棵树并不是无线的延伸下去的，因为树中所有的节点都不能重复，否则会出现死循环。比如"0000"的子节点包含"0001"，但"0001"的子节点不能再包含"0000"了。并且子节点中还不能包含死亡数字。

搞懂了上面的分析过程，代码就容易多了

之前讲过二叉树的BFS遍历，具体可以看下《373，数据结构-6,树》，他就是一层一层的往下遍历的，如下图所示



二叉树的BFS代码我们可以这样写

```
1   public void levelOrder(TreeNode tree) {
2     Queue<TreeNode> queue = new LinkedList<>();
3     queue.add(tree);
4     int level = 0;//统计有多少层
5     while (!queue.isEmpty()) {
```

```
      //每一层的节点数
      int size = queue.size();
      for (int i = 0; i < size; i++) {
        TreeNode node = queue.poll();
        //打印节点
        System.out.println(node.val);
        if (node.left != null)
          queue.add(node.left);
        if (node.right != null)
          queue.add(node.right);
      }
      level++;
      //打印第几层
      System.out.println(level);
    }
  }
```

二叉树的BFS打印我们搞懂了之后，那么不管是8叉树还是100叉树，打印其实都是一样的，我们来看下最终代码

```
public int openLock(String[] deadends, String target) {
  Set<String> set = new HashSet<>(Arrays.asList(deadends));
  //开始遍历的字符串是"0000"，相当于根节点
  String startStr = "0000";
  if (set.contains(startStr))
    return -1;
  //创建队列
  Queue<String> queue = new LinkedList<>();
  //记录访问过的节点
  Set<String> visited = new HashSet<>();
  queue.offer(startStr);
  visited.add(startStr);
  //树的层数
  int level = 0;
  while (!queue.isEmpty()) {
    //每层的子节点个数
    int size = queue.size();
    while (size-- > 0) {
      //每个节点的值
      String str = queue.poll();
      //对于每个节点中的4个数字分别进行加1和减1，相当于创建8个子节点，这八个子节点
      //可以类比二叉树的左右子节点
      for (int i = 0; i < 4; i++) {
        char ch = str.charAt(i);
        //strAdd表示加1的结果，strSub表示减1的结果
        String strAdd = str.substring(0, i) + (ch == '9' ? 0 : ch - '0' + 1) + str.substring(i + 1);
        String strSub = str.substring(0, i) + (ch == '0' ? 9 : ch - '0' - 1) + str.substring(i + 1);
        //如果找到直接返回
        if (str.equals(target))
          return level;
        //不能包含死亡数字也不能包含访问过的字符串
        if (!visited.contains(strAdd) && !set.contains(strAdd)) {
          queue.offer(strAdd);
          visited.add(strAdd);
        }
        if (!visited.contains(strSub) && !set.contains(strSub)) {
          queue.offer(strSub);
          visited.add(strSub);
        }
      }
    }
    //当前层访问完了，到下一层，层数要加1
    level++;
  }
  return -1;
}
```

总结

实际上并不是一棵树，但我们可以把它想象成为一棵树，就像图的BFS遍历一样，我们还需要使用一个变量来记录访问过的节点，如果被访问过之后，下次就不能再访问了。

往期推荐

- 531，BFS和动态规划解完全平方数
- 507，BFS和DFS解二叉树的层序遍历 II
- 473，BFS解单词接龙
- 464. BFS和DFS解二叉树的所有路径

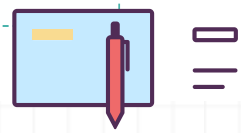
531，BFS和动态规划解完全平方数

原创 博哥 数据结构和算法 3月29日

收录于话题
#算法图文分析

143个 >

> Nothing in life is to be feared. It is only to be understood.
>
> 生活中没有可畏惧的，只要理解它就能战胜它。



问题描述

给定正整数n，找到若干个完全平方数（比如1，4，9，16,...）使得它们的和等于n。你需要让组成和的完全平方数的个数最少。

给你一个整数n，返回和为n的完全平方数的最少数量 。

完全平方数是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，1、4、9和16都是完全平方数，而3和11不是。

示例 1：

```
输入：n = 12
输出：3
解释：12 = 4 + 4 + 4
```

示例 2：

```
输入：n = 13
输出：2
解释：13 = 4 + 9
```

提示：

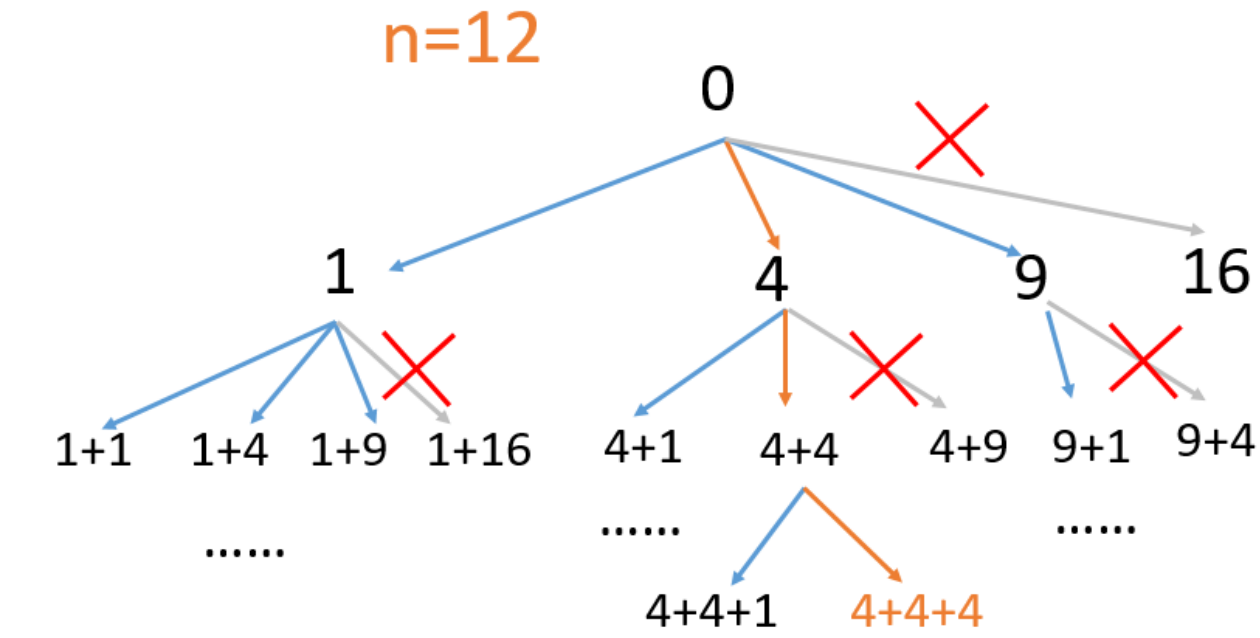
- 1 <= n <= 10^4

BFS解决

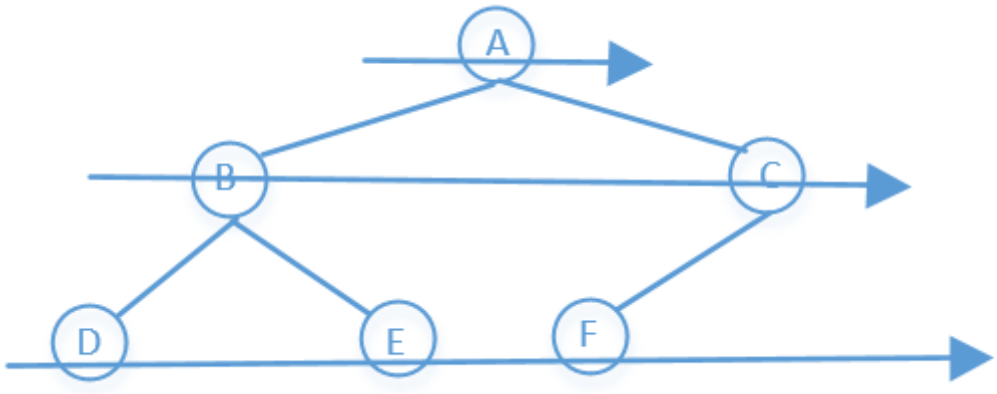
这题让求的是若干个平方数的和等于n，并且平方数的个数最少。首先我们可以把它想象成为一颗m叉树，树的每一个节点的值都是平方数的和，如下图所示。

每一个节点的值都是从根节点到当前节点的累加。而平方数的个数其实就是遍历到第几层的时候累加和等于target。我们只需要一层一层的遍历，也就是常说的BFS，当遇到累加的和等于target的时候直接返回当前的层数即可。

作者：数据结构和算法



二叉树的BFS遍历像下面这样



他的代码很简单

```java
public void levelOrder(TreeNode tree) {
  Queue<TreeNode> queue = new LinkedList<>();
  queue.add(tree);
  int level = 0;//统计有多少层
  while (!queue.isEmpty()) {
    //每一层的节点数
    int size = queue.size();
    for (int i = 0; i < size; i++) {
      TreeNode node = queue.poll();
      //打印节点
      System.out.println(node.val);
      if (node.left != null)
        queue.add(node.left);
      if (node.right != null)
        queue.add(node.right);
    }
    level++;
    //打印第几层
    System.out.println(level);
  }
}
```

我们只需要对他稍作修改就是今天这题的答案了，来看下最终代码

```java
public int numSquares(int n) {
  Queue<Integer> queue = new LinkedList<>();
  //记录访问过的节点值
  Set<Integer> visited = new HashSet<>();
  queue.offer(0);
  visited.add(0);
  //树的第几层
  int level = 0;
  while (!queue.isEmpty()) {
    //每一层的节点数量
    int size = queue.size();
    level++;
    //遍历当前层的所有节点
    for (int i = 0; i < size; i++) {
      //节点的值
      int digit = queue.poll();
      //访问当前节点的子节点，类比于二叉树的左右子节点
      for (int j = 1; j <= n; j++) {
        //子节点的值
        int nodeValue = digit + j * j;
        //nodeValue始终是完全平方数的和，当他等于n的
        //时候直接返回
        if (nodeValue == n)
          return level;
        //如果大于n，终止内层循环
        if (nodeValue > n)
          break;
        if (!visited.contains(nodeValue)) {
          queue.offer(nodeValue);
          visited.add(nodeValue);
        }
      }
    }
  }
  return level;
}
```

动态规划解决

这题除了使用BFS以外，还可以使用动态规划解决。

定义数组dp[]，其中dp[i]表示的是当n等于i的时候完全平方数的最少数量。比如dp[12]表示当n等于12的时候完全平方数的最少数量。这种解法比较类似于背包问题，具体可以看下《371，背包问题系列之-基础背包问题》。

比如当n等于60的时候，他的值是dp[60]，但是60还可以由11加上7的平方组成，我们还可以改为dp[11]+1，取最小的即可，即
dp[60]=min(dp[60],dp[11]+1,)

实际上60还可以由24加上6的平方组成……我们只需要找出所有的可能组合并记录最小的值即可。

所以递推公式我们很容易找出来

dp[i]=Math.min(dp[i],dp[i-j*j]+1);

那么初始条件是什么呢，我们默认任何正整数都是由1的平方组成，即dp[i]=i，也就是最大值，然后再通过递推公式找出最小值即可。

最后我们再来看下代码

```
public int numSquares(int n) {
    int[] dp = new int[n + 1];
    dp[0] = 0;
    for (int i = 1; i <= n; i++) {
        dp[i] = i;//最坏的情况都是由1的平方组成
        for (int j = 1; j * j <= i; j++) {
            //动态规划公式
            dp[i] = Math.min(dp[i], dp[i - j * j] + 1);
        }
    }
    return dp[n];
}
```

拉格朗日四平方和定理

拉格朗日四平方和定理说明任何一个数，都可以由小于等于4个的完全平方数相加得到。

当n=(8b+7)*4^n的时候，n是由4个完全平方数得到，否则n只能由1个，2个或者3个完全平方数得到。

由1个，2个和4个完全平方数得到的n我们很容易判断，所以剩下的就是由3个完全平方数得到的n。我们分为4步走的战略，

1. 先判断是否能由**1个平方数**组成
2. 在判断是否能由**4个平方数**组成
3. 接着判断是否能由**2个平方数**组成
4. 如果上面都不成立，只能由**3个平方数**组成了。

我们来看下代码

```
public int numSquares(int n) {
    //一，先判断由1个平方数组成的
    //如果n是平方数，直接返回1即可，表示n由
    //1个平方数组成
    if (is_square(n))
        return 1;
    //如果n是4的倍数，就除以4，因为4是2的平方，
    //如果n可以由m个完全平方数组成，那么4n也
    //可以由m个完全平方数组成
    while ((n & 3) == 0)
        n >>= 2;
    //二，在判断由4个平方数组成的
    //如果n是4的倍数，在上面代码的执行中就会一直除以4，
    //直到不是4的倍数为止，所以这里只需要判断n=(8b+7)
    //即可
    if ((n & 7) == 7)
        return 4;
    int sqrt_n = (int) (Math.sqrt(n));
    //三，接着判断由2个平方数组成的
    //下面判断是否能由2个平方数组成
    for (int i = 1; i <= sqrt_n; i++) {
        if (is_square(n - i * i)) {
            return 2;
        }
    }
    //四，剩下的只能由3个平方数组成了
    //如果上面都不成立，根据拉格朗日四平方和定理
    //只能由3个平方数组成了
    return 3;
}

//判断n是否是平方数
public boolean is_square(int n) {
    int sqrt_n = (int) (Math.sqrt(n));
    return sqrt_n * sqrt_n == n;
}
```


往期推荐

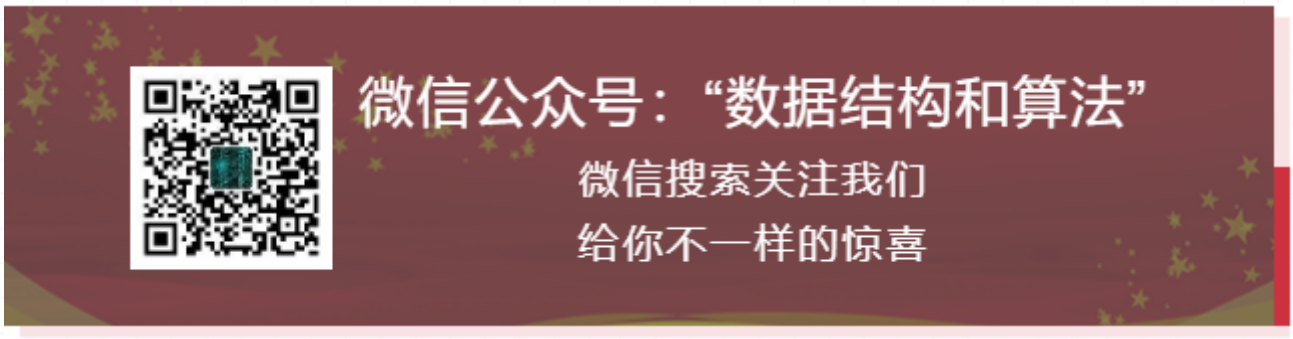
- 507，BFS和DFS解二叉树的层序遍历 II
- 470，DFS和BFS解合并二叉树
- 477，动态规划解按摩师的最长预约时间
- 465. 递归和动态规划解三角形最小路径和

507，BFS和DFS解二叉树的层序遍历 II

原创 山大王wld 数据结构和算法 1月13日

收录于话题
#算法图文分析

137个 >



Standing for right when it is unpopular is a true test of moral character.

站在不受欢迎但却正确的一边，才是真正的道德考验。

问题描述

给定一个二叉树，返回其节点值自底向上的层序遍历。（即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）

例如：
给定二叉树 [3,9,20,null,null,15,7],

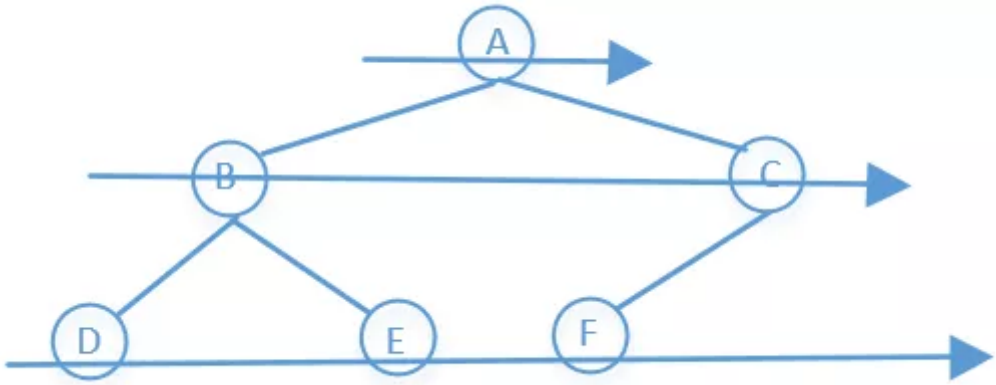
```
    3
   / \
  9  20
    /  \
   15   7
```

返回其自底向上的层序遍历为：

```
[
  [15,7],
  [9,20],
  [3]
]
```

BFS解决

这题类似于二叉树的BFS打印，就是一层一层的打印，一般情况下对于二叉树我们都是从上往下打印，但这题是从下往上打印。直接从下往上不太好操作，可以换种思路，因为题目要求的结果是从下往上就可以了，并没有要求打印的过程。



我们依然可以从上往下打印，只不过打印每一层的时候，结果都要插到列表的最前面，这样最终结果和从下往上打印的结果就完全一样了。

```java
public List<List<Integer>> levelOrderBottom(TreeNode root) {
  //边界条件判断
  if (root == null)
    return new ArrayList<>();
  //队列
  Queue<TreeNode> queue = new LinkedList<>();
  List<List<Integer>> res = new ArrayList<>();
  //根节点入队
  queue.add(root);
  //如果队列不为空就继续循环
  while (!queue.isEmpty()) {
    //BFS打印，levelCount表示的是每层的结点数
    int levelCount = queue.size();
    //subList存储的是每层的结点值
    List<Integer> subList = new ArrayList<>();
    for (int i = 0; i < levelCount; i++) {
      //出队
      TreeNode node = queue.poll();
      subList.add(node.val);
      //左右子节点如果不为空就加入到队列中
      if (node.left != null)
        queue.add(node.left);
      if (node.right != null)
        queue.add(node.right);
    }
    //把每层的结点值存储在res中，插入到最前面
    //（类似于从下往上打印，关键点在这）
    res.add(0, subList);
  }
```

```
30      return res;
31  }
```

DFS解决

在前面讲[373，数据结构-6,树]的时候提到过二叉树的BFS和DFS，其中DFS是一直往下走的，到叶子节点然后再返回。对于这道题我们从根节点往下走的时候，每一层都会有一个集合list，用来存放当前层的节点值，如果当前层的list没有创建，就先创建。原理也比较简单，来看下代码

```
1   public List<List<Integer>> levelOrderBottom(TreeNode root) {
2     List<List<Integer>> res = new ArrayList<>();
3     helper(res, root, 0);
4     return res;
5   }
6
7   public void helper(List<List<Integer>> list, TreeNode root, int level) {
8     //边界条件判断
9     if (root == null)
10      return;
11    //如果level等于list的长度，说明到下一层了，
12    //并且下一层的ArrayList还没有初始化，我们要
13    //先初始化一个ArrayList，然后放进去。
14    if (level == list.size()) {
15      list.add(0, new ArrayList<>());
16    }
17    //这里就相当于从后往前打印了
18    list.get(list.size() - level - 1).add(root.val);
19    //当前节点访问完之后，再使用递归的方式分别访问当前节点的左右子节点
20    helper(list, root.left, level + 1);
21    helper(list, root.right, level + 1);
22  }
```

总结

只要明白二叉树的BFS遍历，这题就很容易解决，虽然这题结果是从下往上，但我们只需要在每层节点值存储的时候修改一下位置即可。

往期推荐

- 470，DFS和BFS解合并二叉树
- 464，BFS和DFS解二叉树的所有路径
- 453，DFS和BFS解求根到叶子节点数字之和
- 417，BFS和DFS两种方式求岛屿的最大面积

473，BFS解单词接龙

原创 山大王wld 数据结构和算法 11月10日

收录于话题
#算法图文分析

95个 >



Dream is not about what you want, but what you do after knowing who you are.

理想不是想想而已，是看清自我后的不顾一切。

问题描述

给定两个单词（beginWord 和 endWord）和一个字典，找到从 beginWord 到 endWord的最短转换序列的长度。转换需遵循如下规则：

1. **每次转换只能改变一个字母。**
2. **转换过程中的中间单词必须是字典中的单词。**

说明:

- 如果不存在这样的转换序列，返回 0。
- 所有单词具有相同的长度。
- 所有单词只由小写字母组成。
- 字典中不存在重复的单词。
- 你可以假设 beginWord 和 endWord 是非空的，且二者不相同。

示例 1:

输入:
beginWord = "hit",
endWord = "cog",

wordList =
["hot","dot","dog","lot","log","cog"]

输出: 5

解释: 一个最短转换序列是
"hit" -> "hot" -> "dot" -> "dog" -> "cog"

返回它的长度 5。

示例 2:

输入:
beginWord = "hit"
endWord = "cog"

wordList =
["hot","dot","dog","lot","log"]

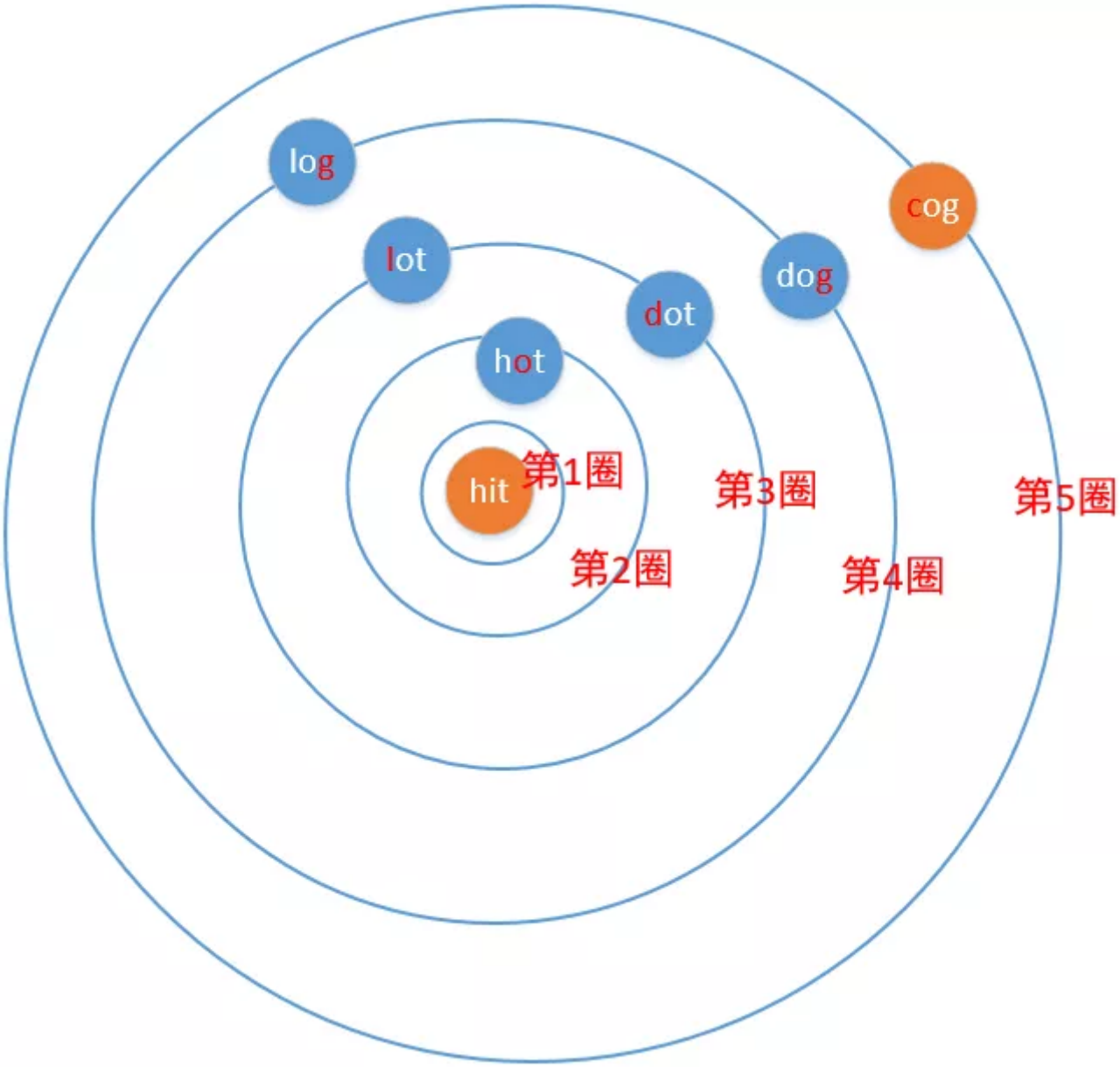
输出: 0

解释: endWord "cog" 不在字典中，所以无法进行转换。

1，一圈一圈往外扩散

这里以beginWord单词为中心点当做是第一圈，然后把字典中和beginWord只差一个字符的单词放到第二圈，然后再把和第二圈只差一个字符并且在字典中存在的单词放到第3圈……，一直这样放，放的时候要注意不能放之前放过的，并且在放的时候遇到endWord直接返回即可。

这里以示例1为例画个图来看下



代码如下，有详细的注释，应该不是很难理解

```
public int ladderLength(String beginWord, String endWord, List<String> wordList) {
  //把字典中的单词放入到set中，主要是为了方便查询
  Set<String> dictSet = new HashSet<>(wordList);
  //创建一个新的单词，记录单词是否被访问过，如果没被访问过就加入进来
  Set<String> visit = new HashSet<>();
  //BFS中常见的队列，我们可以把它想象成为一颗二叉树，记录每一层的节点。
  //或者把它想象成为一个图，记录挨着的节点，也就是每一圈的节点。这里我们把它想象成为一个图
  Queue<String> queue = new LinkedList<>();
  queue.add(beginWord);
  //这里的图是一圈一圈往外扩散的，这里的minlen可以看做扩散了多少圈，
  //也就是最短的转换序列长度
  int minlen = 1;
  while (!queue.isEmpty()) {
    //这里找出每个节点周围的节点数量，然后都遍历他们
    int levelCount = queue.size();
    for (int i = 0; i < levelCount; i++) {
      //出队
      String word = queue.poll();
      //这里遍历每一个节点的单词，然后修改其中一个字符，让他成为一个新的单词，
      //并查看这个新的单词在字典中是否存在，如果存在并且没有被访问过，就加入到队列中
      for (int j = 0; j < word.length(); j++) {
        char[] ch = word.toCharArray();
        for (char c = 'a'; c <= 'z'; c++) {
          if (c == word.charAt(j))
            continue;
```

```
            ch[j] = c;
            //修改其中的一个字符，然后重新构建一个新的单词
            String newWord = String.valueOf(ch);
            //查看字典中有没有这个单词，如果有并且没有被访问过，就加入到队列中
            //（Set的add方法表示把单词加入到队列中，如果set中没有这个单词
            //就会添加成功，返回true。如果有这个单词，就会添加失败，返回false）
            if (dictSet.contains(newWord) && visit.add(newWord)) {
              //如果新单词是endWord就返回,这里访问的是第minlen圈的节点，然后
              //新建的节点就是第minlen+1层
              if (newWord.equals(endWord))
                return minlen + 1;
              queue.add(newWord);
            }
          }
        }
      }
      //每往外扩一圈，长度就加1
      minlen++;
    }
    return 0;
  }
```

2，从两边往中间开始计算

上面是从start开始往外扩散，这里还可以一个从start，一个从end开始往中间走，哪个圈的元素少就先遍历哪个，这样可以减少循环的次数，如果能相遇就返回

```
  private int find(int minlen, Queue<String> startQueue, Queue<String> endQueue, Set<String> dictSe
    int startCount = startQueue.size();
    int endCount = endQueue.size();
    boolean start = startCount <= endCount;
    int count = start ? startCount : endCount;
    //哪个量少，遍历哪个
    for (int i = 0; i < count; i++) {
      //出队
      String word;
      if (start)
        word = startQueue.poll();
      else
        word = endQueue.poll();

      //这里遍历每一个节点的单词，然后修改其中一个字符，让他成为一个新的单词，
      //并查看这个新的单词在字典中是否存在，如果存在并且没有被访问过，就加入到队列中
      for (int j = 0; j < word.length(); j++) {
        char[] ch = word.toCharArray();
        for (char c = 'a'; c <= 'z'; c++) {
          if (c == word.charAt(j))
            continue;
          ch[j] = c;
          //修改其中的一个字符，然后重新构建一个新的单词
          String newWord = String.valueOf(ch);
          if (dictSet.contains(newWord)) {
            if (start) {//从前往后
              if (endQueue.contains(newWord))
                return minlen + 1;
              if (visit.add(newWord))
                startQueue.add(newWord);
            } else {//从后往前
              if (startQueue.contains(newWord))
                return minlen + 1;
              if (visit.add(newWord))
                endQueue.add(newWord);
            }
          }
        }
```

```
        }
    }
    //如果没有相遇就返回-1
    return -1;
}
```

总结

这道题上两种方式都能解决，第2种方式比第一种稍微要复杂一些，但运行效率要比第1种高。

往期推荐

- 470，DFS和BFS解合并二叉树
- 455，DFS和BFS解被围绕的区域
- 445，BFS和DFS两种方式解岛屿数量
- 422，剑指 Offer-使用DFS和BFS解机器人的运动范围

470，DFS和BFS解合并二叉树

原创 山大王wld 数据结构和算法 11月2日

收录于话题
#算法图文分析

95个 >



Sometimes the thing you're searching for your whole life, it's right there by your side all along.

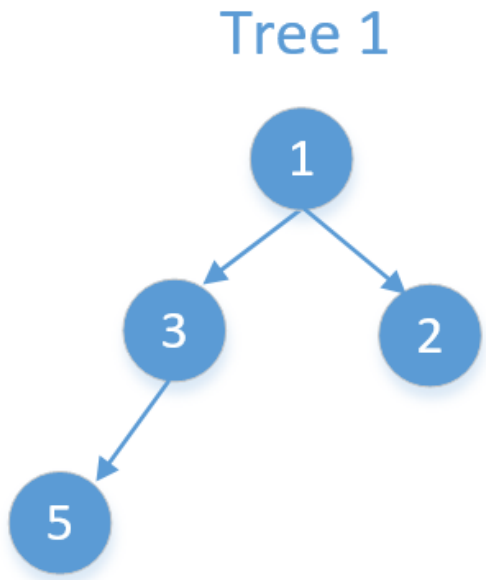
有时候你穷尽一生想要得到的，可能一直都在你身边。

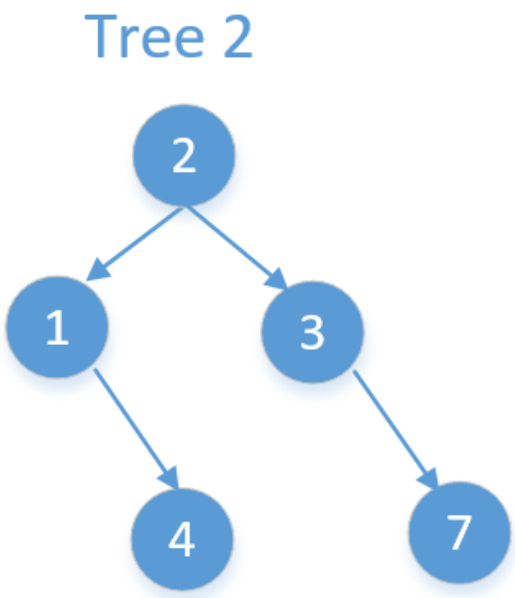
问题描述

给定两个二叉树，想象当你将它们中的一个覆盖到另一个上时，两个二叉树的一些节点便会重叠。

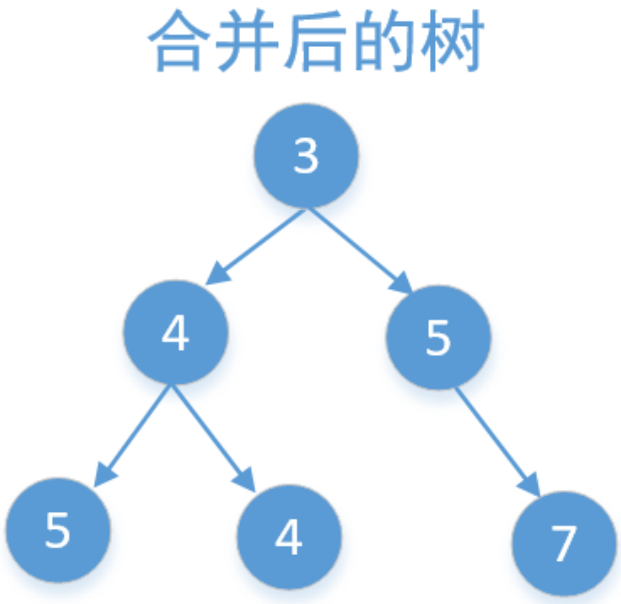
你需要将他们合并为一个新的二叉树。合并的规则是如果两个节点重叠，那么将他们的值相加作为节点合并后的新值，否则不为 NULL 的节点将直接作为新二叉树的节点。

示例 1：





合并后的树如下



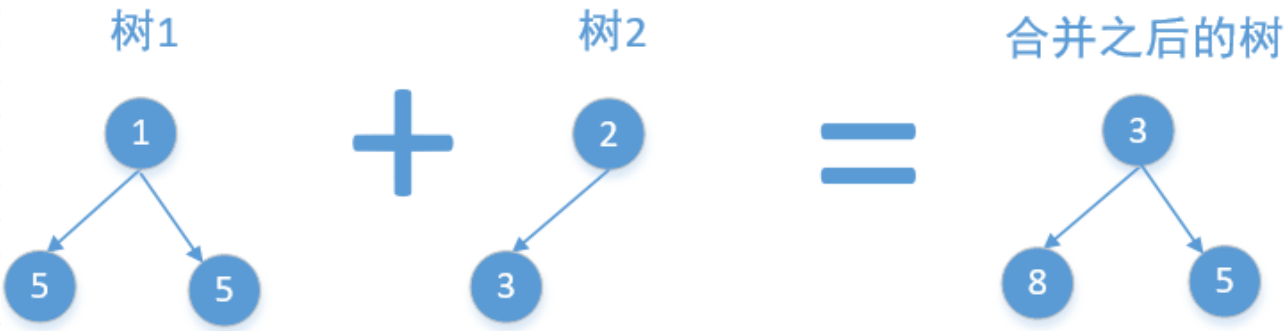
注意: 合并必须从两个树的根节点开始。

DFS解决

合并两棵二叉树，会有下面3种情况

- **树的两个节点都是空，那就不需要合并了**
- **树的两个节点一个为空一个不为空，那么合并的结果肯定是不为空的那个节点**
- **树的两个节点都不为空，那么合并的节点值就是这个两个节点的和**

来画个简图看一下

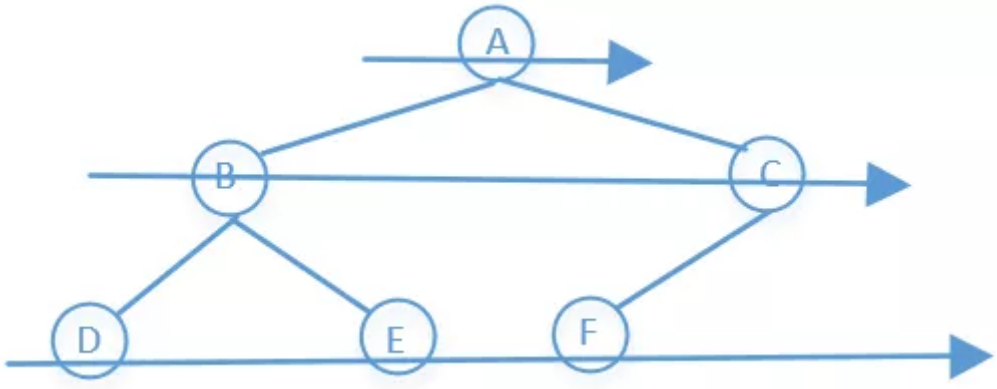


代码如下

```
public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {
    //如果两个节点都为空，直接返回空就行了
    if (t1 == null && t2 == null)
        return null;
    //如果t1节点为空，就返回t2节点
    if (t1 == null)
        return t2;
    //如果t2节点为空，就返回t1节点
    if (t2 == null)
        return t1;
    //走到这一步，说明两个节点都不为空，然后需要把这两个节点
    //合并成一个新的节点
    TreeNode newNode = new TreeNode(t1.val + t2.val);
    //当前节点t1和t2合并完之后，还要继续合并t1和t2的子节点
    newNode.left = mergeTrees(t1.left, t2.left);
    newNode.right = mergeTrees(t1.right, t2.right);
    return newNode;
}
```

BFS解决

除了DFS我们还可以使用BFS，就是一层一层的遍历，合并的原理和上面一样



这里是把第2棵树合并到第1棵树上，

- **如果树1的左子节点为空，直接把第2棵树的左子节点赋给第1棵树的左子节点即可**
- **如果树1的左子节点不为空，树2的左子节点为空，直接返回树1的左子节点即可**
- **如果树1的左子节点和树2的左子节点都不为空，直接相加。**

右子树和上面原理一样，来看下代码

```java
//把第2棵树合并到第1棵树上
public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {
    //如果t1节点为空，就返回t2节点
    if (t1 == null)
        return t2;
    //如果t2节点为空，就返回t1节点
    if (t2 == null)
        return t1;
    //队列中两棵树的节点同时存在，
    Queue<TreeNode> queue = new LinkedList<>();
    //把这两棵树的节点同时入队
    queue.add(t1);
    queue.add(t2);
    while (!queue.isEmpty()) {
        //两棵树的节点同时出队
        TreeNode node1 = queue.poll();
        TreeNode node2 = queue.poll();
        //把这两个节点的值相加，然后合并到第1棵树的节点上
        node1.val += node2.val;
        if (node1.left == null) {
            //如果node1左子节点为空，我们直接让node2的
            //左子结点成为node1的左子结点，
            node1.left = node2.left;
        } else {
            //执行到这一步，说明node1的左子节点不为空，
            //如果node2的左子节点为空就不需要合并了，
            //只有node2的左子节点不为空的时候才需要合并
            if (node2.left != null) {
                queue.add(node1.left);
                queue.add(node2.left);
            }
        }

        //原理同上，上面判断的是左子节点，这里判断的是右子节点
        if (node1.right == null) {
            node1.right = node2.right;
        } else {
            if (node2.right != null) {
                queue.add(node1.right);
                queue.add(node2.right);
            }
        }
    }
    //把第2棵树合并到第1棵树上，所以返回的是第1棵树
    return t1;
}
```

总结

合并两棵树，这里要分为3种情况来讨论，除了这3种情况，基本上也没有其他需要注意的地方。如果对二叉树的DFS和BFS比较熟悉的话，这题很容易写出来。

往期推荐

- 465. 递归和动态规划解三角形最小路径和
- 464. BFS和DFS解二叉树的所有路径

455，DFS和BFS解被围绕的区域

原创 山大王wld 数据结构和算法 9月24日

收录于话题
#算法图文分析

95个 >



Persist, and anything is within your reach.

坚持下来，做任何事情你都能够成功。

问题描述

给定一个二维的矩阵，包含 'X' 和 'O' （**字母 O**）。

找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充。

示例:

```
X X X X
X O O X
X X O X
X O X X
```

运行你的函数后，矩阵变为：

```
X X X X
X X X X
X X X X
```

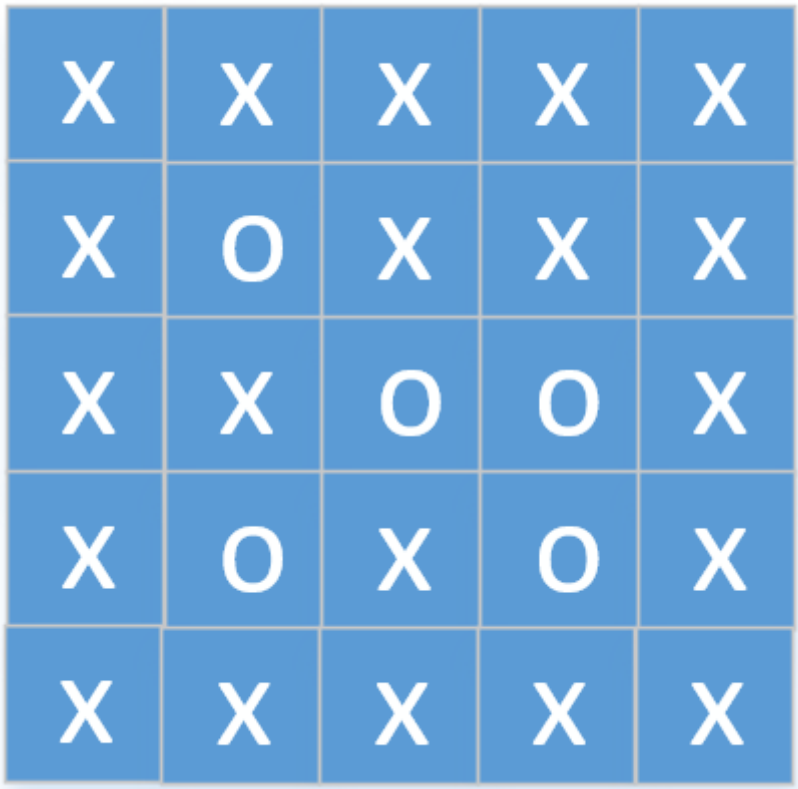
> X O X X

解释:

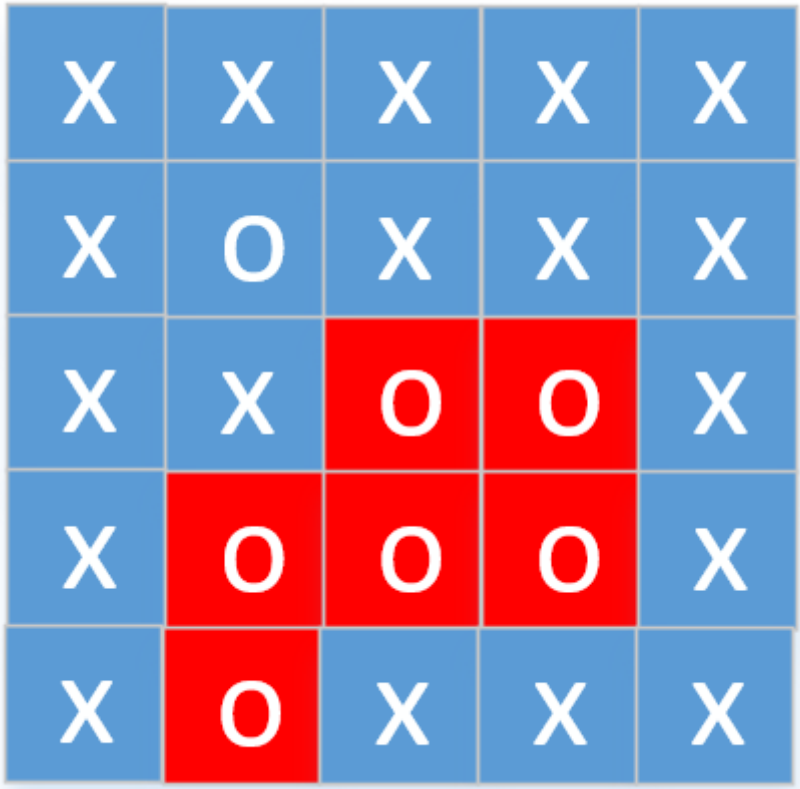
被围绕的区间不会存在于边界上，换句话说，任何边界上的 'O' 都不会被填充为 'X'。任何不在边界上，或不与边界上的 'O' 相连的 'O' 最终都会被填充为 'X'。如果两个元素在水平或垂直方向相邻，则称它们是“相连”的。

DFS解决

这题说的是如果被X围绕的区域有O，就用X把这个区域内的O给替换掉。那么我们怎么判断O是被X包围的呢，直接判断可能不太好操作，我们可以换种思路。如果矩阵的四周都是X，那么矩阵中只要有O，肯定是被X包围的，这个很好理解，就像下面这样



如果矩阵的四周只要有一个是O，那么和这个O挨着（挨着仅指上下左右，斜对角不算）的O都不可能被X包围，比如下面这样

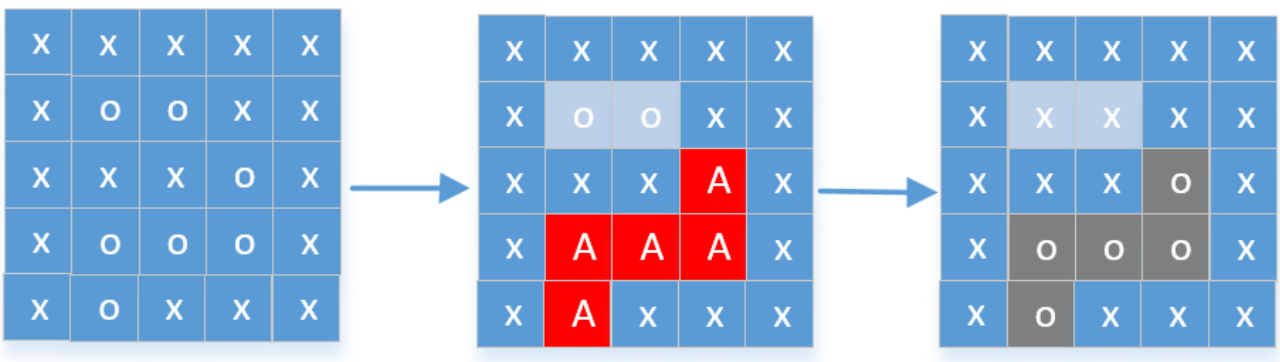


所以一种比较简单的判断方式就是查找这个矩阵的四周，查看有没有O，如果有O，说明他不能被X给包围，也就是不能被替换成X，我们先把他变为大写的A（其他值也可以，只要不是X和O就行），然后再遍历他的上下左右查看有没有O，如果有O，那么这个O也是不能被替换成X的，也要被标记为A……。

最后矩阵中最多会有3种状态，一种是X，一种是A，一种是O。

- 如果是X不用动，
- 如果是O，说明他是被X包围的，需要把它替换成X。
- 如果是A说明是不能被X包围的，还要把它还原为O。

我们随便举个找个数据画个图看一下



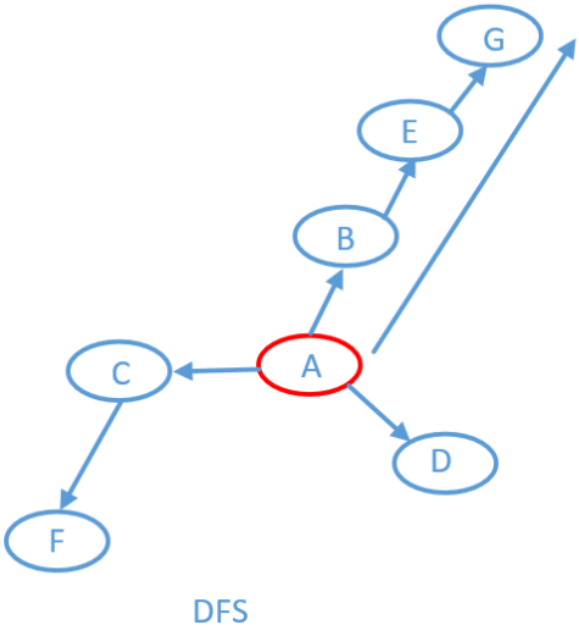
弄懂了上面的过程，代码就容易多了

```
public void solve(char[][] board) {
    //边界条件判断
```

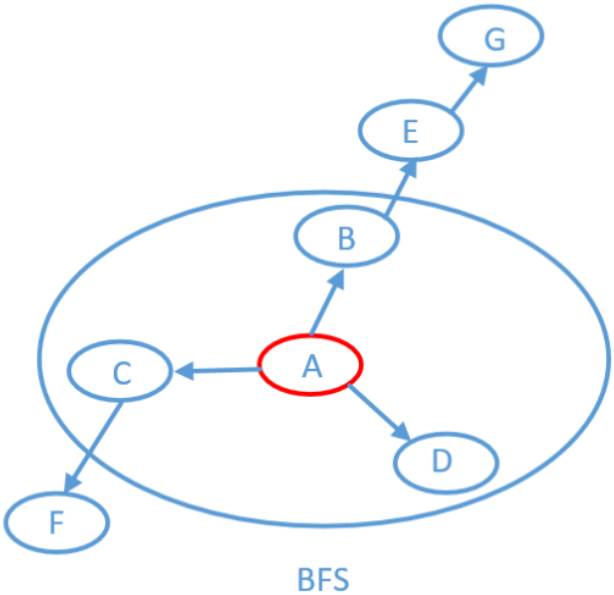
```
    if (board == null || board.length == 0)
      return;
    for (int i = 0; i < board.length; i++) {
      for (int j = 0; j < board[0].length; j++) {
        //从矩阵的四周开始判断，也就是矩阵的4条边上有O的地方开始遍历
        if (i == 0 || i == board.length - 1 || j == 0 || j == board[0].length - 1) {
          if (board[i][j] == 'O')
            dfs(i, j, board);
        }
      }
    }
    //重新复原
    for (int i = 0; i < board.length; i++) {
      for (int j = 0; j < board[0].length; j++) {
        //把矩阵中是'A'的还变为O，其他的都变成X
        if (board[i][j] == 'A')
          board[i][j] = 'O';
        else
          board[i][j] = 'X';
      }
    }
    return;
  }

  private void dfs(int i, int j, char[][] board) {
    //边界条件判断，首先不能跑到矩阵的外边
    if (i < 0 || i >= board.length || j < 0 || j >= board[0].length)
      return;
    //如果当前位置不是O，就不用再判断了
    if (board[i][j] != 'O')
      return;
    //如果当前位置是O，先把他变为'A'，然后往他的上下左右4个方向开始递归计算。
    board[i][j] = 'A';
    dfs(i - 1, j, board);//上
    dfs(i + 1, j, board);//下
    dfs(i, j - 1, board);//左
    dfs(i, j + 1, board);//右
  }
```

BFS解决

DFS是沿着一个方向一直走下去，BFS是先遍历四周的，然后再往外扩散，如下图所示



DFS



BFS

原理还是一样的，从矩阵的四周开始，找到一个O之后，把它变为A，然后把他的四周在遍历一遍，如果有O就加入到队列中，然后继续遍历队列中的元素……

```
public void solve(char[][] board) {
  //边界条件判断
  if (board == null || board.length == 0)
    return;
  int rows = board.length, columns = board[0].length;
  for (int i = 0; i < rows; i++)
    for (int j = 0; j < columns; j++) {
      //从矩阵的四周开始判断，也就是矩阵的4条边上有O的地方开始遍历
      if (i == 0 || i == rows - 1 || j == 0 || j == columns - 1) {
        if (board[i][j] == 'O')
          bfs(i, j, board);
      }
    }
  //重新复原
  for (int i = 0; i < board.length; i++) {
    for (int j = 0; j < board[0].length; j++) {
      //把矩阵中是'A'的还变为O，其他的都变成X
      if (board[i][j] == 'A')
        board[i][j] = 'O';
      else
        board[i][j] = 'X';
    }
  }
}

int[][] direction = {{-1, 0}, {1, 0}, {0, 1}, {0, -1}};

private void bfs(int i, int j, char[][] board) {
  Queue<Integer> queue = new LinkedList<>();
  //把当前位置变为A
  board[i][j] = 'A';
  //把当前的坐标加入到队列中
  queue.offer(i);
  queue.offer(j);
  while (!queue.isEmpty()) {
    //坐标出队
    int queueI = queue.poll();
    int queueJ = queue.poll();
    //沿着当前位置(queueI,queueJ)的上下左右四个方向查找
    for (int k = 0; k < 4; k++) {
      int x = direction[k][0] + queueI;
      int y = direction[k][1] + queueJ;
      //边界条件判断，首先不能跑到矩阵的外边
      if (x < 0 || x >= board.length || y < 0 || y >= board[0].length)
        continue;
      //如果当前位置不是O，就不用再判断了
      if (board[x][y] != 'O')
        continue;
      //否则就把他变为A
      board[x][y] = 'A';
      //然后再把这个位置的坐标存放到队列中
      queue.offer(x);
      queue.offer(y);
    }
  }
}
```

总结

要想找到被X包围的区域，最简单的一种方式就是从四周开始找，因为如果四周有O，那么他们肯定是不能被包围的，如果还有和这个O挨着的，也是不能被包围的，否则剩下

的如果有O，那么剩下的这些肯定是能被X包围的，理解这个思路很重要。

往期推荐

- 453，DFS和BFS解求根到叶子节点数字之和
- 450，什么叫回溯算法，一看就会，一写就废
- 445，BFS和DFS两种方式解岛屿数量
- 422，剑指 Offer-使用DFS和BFS解机器人的运动范围

453，DFS和BFS解求根到叶子节点数字之和

原创 山大王wld 数据结构和算法 9月21日

收录于话题
#算法图文分析

95个 >



If you wish to survive you need to cultivate a strong mental attitude.

如果你想活着，需要培养一颗坚强的心。

问题描述

给定一个二叉树，它的每个结点都存放一个 0-9 的数字，每条从根到叶子节点的路径都代表一个数字。

例如，从根到叶子节点路径 1->2->3 代表数字 123。
计算从根到叶子节点生成的所有数字之和。

说明: 叶子节点是指没有子节点的节点。

示例 1:

> **输入**: [1,2,3]
>
> 1
>
> / \

```
  2    3

输出: 25
解释:
从根到叶子节点路径 1->2 代表数字 12.
从根到叶子节点路径 1->3 代表数字 13.
因此，数字总和 = 12 + 13 = 25.
```

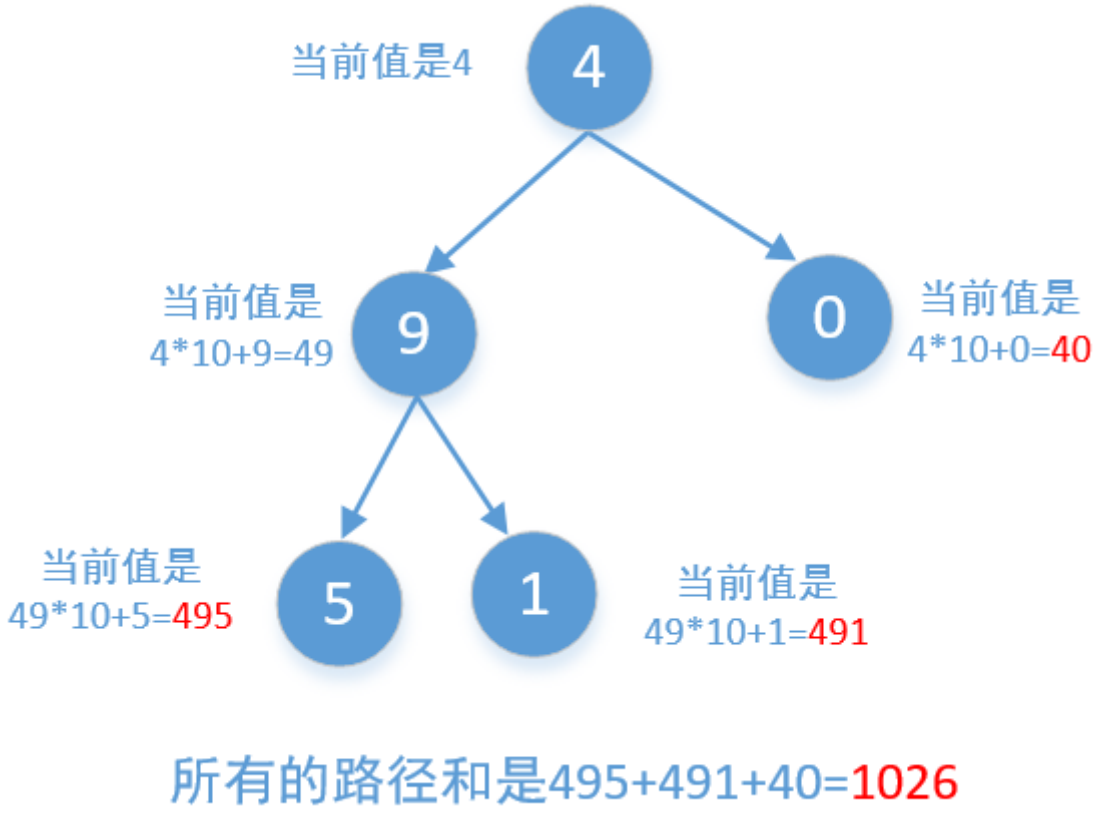
示例 2:

```
输入: [4,9,0,5,1]
    4
   / \
  9   0
 / \
5   1

输出: 1026
解释:
从根到叶子节点路径 4->9->5 代表数字 495.
从根到叶子节点路径 4->9->1 代表数字 491.
从根到叶子节点路径 4->0 代表数字 40.
因此，数字总和 = 495 + 491 + 40 = 1026.
```

DFS解决

这题说的是每条从根节点到叶子结点的路径都代表一个数字，然后再把这些数字加起来即可。遍历一棵树从根节点到叶子结点的所有路径，最容易想到的是DFS，所以这题使用DFS是最容易解决的。如果对二叉树的DFS不熟悉的话，可以看下373，数据结构-6，树

解决方式就是从根节点往下走的时候，那么当前节点的值就是父节点的值*10+当前节点的值。默认根节点的父节点的值是0，如果到达叶子结点，就用一个全局的变量把叶子结点的值加起来。这里就以示例2为例来画个图看一下



搞懂了上的分析过程，代码就容易多了

```java
public int sumNumbers(TreeNode root) {
    //如果根节点是空，直接返回0即可
    if (root == null)
        return 0;
    //两个栈，一个存储的是节点，一个存储的是节点对应的值
    Stack<TreeNode> nodeStack = new Stack<>();
    Stack<Integer> valueStack = new Stack<>();
    //全局的，统计所有路径的和
    int res = 0;
    nodeStack.add(root);
    valueStack.add(root.val);
    while (!nodeStack.isEmpty()) {
        //当前节点和当前节点的值同时出栈
        TreeNode node = nodeStack.pop();
        int value = valueStack.pop();
        if (node.left == null && node.right == null) {
            //如果当前节点是叶子结点，说明找到了一条路径，把这条
            //路径的值加入到全局变量res中
            res += value;
        } else {
            //如果不是叶子节点就执行下面的操作
            if (node.right != null) {
                //把子节点和子节点的值分别加入到栈中，这里子节点的值
                //就是父节点的值*10+当前节点的值
                nodeStack.push(node.right);
                valueStack.push(value * 10 + node.right.val);
            }
            if (node.left != null) {
                nodeStack.push(node.left);
                valueStack.push(value * 10 + node.left.val);
            }
        }
    }
    return res;
}
```

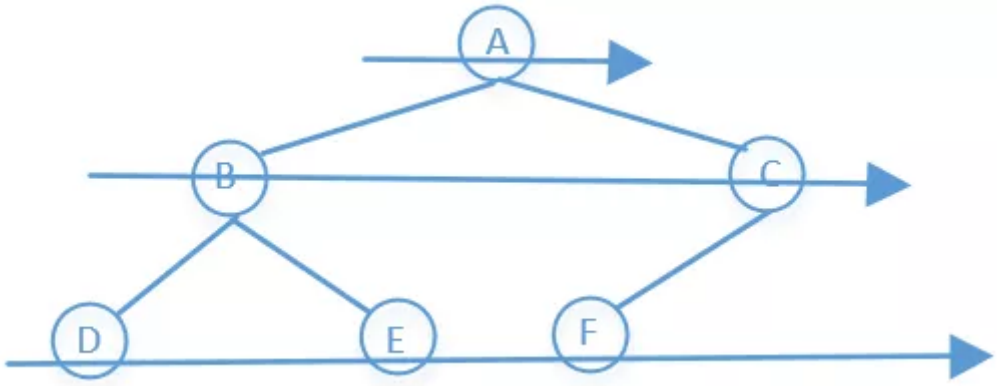
如果嫌上面代码多，也可以改为递归的方式

```java
public int sumNumbers(TreeNode root) {
```

```
    return dfs(root, 0);
}

private int dfs(TreeNode root, int sum) {
    //终止条件的判断
    if (root == null)
        return 0;
    //计算当前节点的值
    sum = sum * 10 + root.val;
    //如果当前节点是叶子节点，说明找到了一条完整路径，直接
    //返回这条路径的值即可
    if (root.left == null && root.right == null)
        return sum;
    //如果当前节点不是叶子结点，返回左右子节点的路径和
    return dfs(root.left, sum) + dfs(root.right, sum);
}
```

BFS解决

对于树的遍历，我们知道除了前序中序后续遍历以外，还有DFS和BFS，DFS上面已经讲过了，下面再来看一下BFS，他是一层一层的遍历的，就像下面这样，如果不太懂的也可以先看下[373，数据结构-6,树]



原理和上面DFS类似，每遍历一个结点，我们就要重新计算当前节点的值，那么当前节点的值就是父节点的值*10+当前节点的值。

```
public int sumNumbers(TreeNode root) {
    //边界条件的判断
    if (root == null)
        return 0;
    Queue<TreeNode> nodeQueue = new LinkedList<>();
    Queue<Integer> valueQueue = new LinkedList<>();
    int res = 0;
    nodeQueue.add(root);
    valueQueue.add(root.val);
    while (!nodeQueue.isEmpty()) {
        //节点和节点对应的值同时出队
        TreeNode node = nodeQueue.poll();
        int value = valueQueue.poll();
        if (node.left == null && node.right == null) {
            //如果当前节点是叶子结点，说明找到了一条路径，把这条
            //路径的值加入到全局变量res中
            res += value;
        } else {
            //如果不是叶子节点就执行下面的操作
            if (node.left != null) {
                //把子节点和子节点的值分别加入到队列中，这里子节点的值
                //就是父节点的值*10+当前节点的值
                nodeQueue.add(node.left);
                valueQueue.add(value * 10 + node.left.val);
```

```
            }
            if (node.right != null) {
                nodeQueue.add(node.right);
                valueQueue.add(value * 10 + node.right.val);
            }
        }
    }
    return res;
}
```

总结

这题从根节点到每个叶子结点都是一个完整的数字，我们要做的就是把这每个数字加起来。使用DFS应该是最容易理解的，其实每条路径也可以把它想象成为一个链表，每个链表代表一个数字，然后把所有的链表所代表的数字加起来就是这题要求的结果。

往期推荐

- 444，二叉树的序列化与反序列化
- 442，剑指 Offer-回溯算法解二叉树中和为某一值的路径
- 440，剑指 Offer-从上到下打印二叉树 II
- 434，剑指 Offer-二叉树的镜像

445，BFS和DFS两种方式解岛屿数量

原创 山大王wld 数据结构和算法 9月2日

收录于话题
#算法图文分析

95个 >



However dark and scary the world might be right now, there will be light.

无论世界现在有多黑暗，多可怕，终有一天会重现光明。

问题描述

给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

示例 1：

> **输入**:
> [
> ['1','1','1','1','0'],
> ['1','1','0','1','0'],
> ['1','1','0','0','0'],
> ['0','0','0','0','0']

]
输出: 1

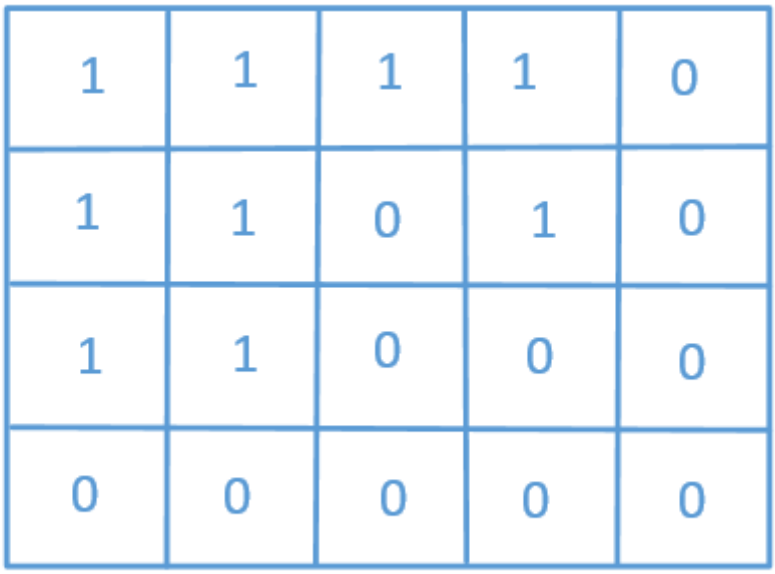
示例 2:

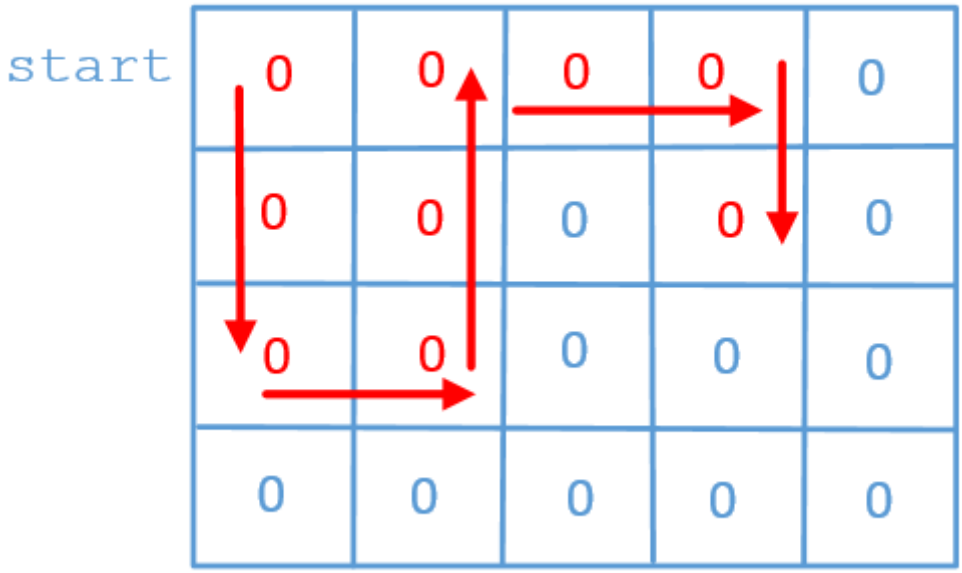
输入:
[
['1','1','0','0','0'],
['1','1','0','0','0'],
['0','0','1','0','0'],
['0','0','0','1','1']
]
输出: 3
解释: 每座岛屿只能由水平和/或竖直方向上相邻的陆地连接而成。

DFS解决

这题让求的是岛屿的面积，二维数组中值是1的都是岛屿，如果多个1是连着的，那么他们只能算一个岛屿。

最简单的一种方式就是遍历数组中的每一个值，如果是1就说明是岛屿，然后把它置为0或者其他的字符都可以，只要不是1就行，然后再遍历他的上下左右4个位置。如果是1，说明这两个岛屿是连着的，只能算是一个岛屿，我们还要把它置为0，然后再以它为中心遍历他的上下左右4个位置……。如果是0，就说明不是岛屿，就不在往他的上下左右4个位置遍历了。这里就以示例1为例来看一下





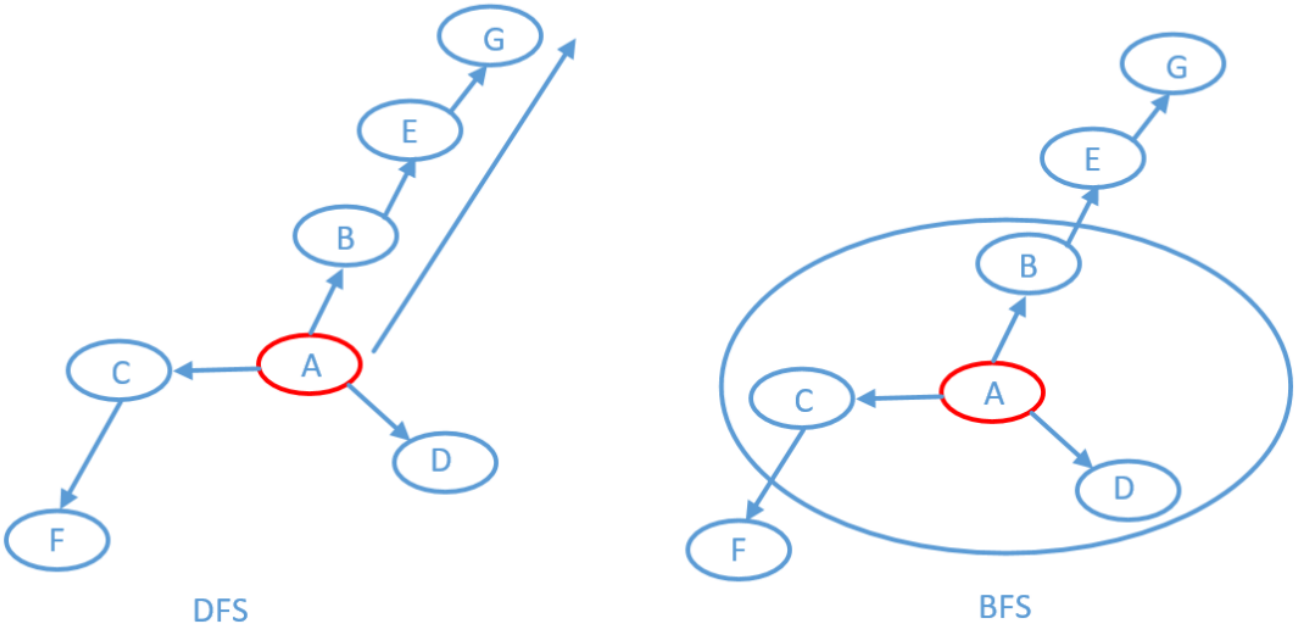
每个位置只要是1，先要把它置为0，然后沿着他的上下左右4个方向继续遍历，执行同样的操作，要注意边界条件的判断。代码比较简单，来看下

```
public int numIslands(char[][] grid) {
  //边界条件判断
  if (grid == null || grid.length == 0)
    return 0;
  //统计岛屿的个数
  int count = 0;
  //两个for循环遍历每一个格子
  for (int i = 0; i < grid.length; i++)
    for (int j = 0; j < grid[0].length; j++) {
      //只有当前格子是1才开始计算
      if (grid[i][j] == '1') {
        //如果当前格子是1，岛屿的数量加1
        count++;
        //然后通过dfs把当前格子的上下左右4
        //个位置为1的都要置为0，因为他们是连着
        //一起的算一个岛屿，
        dfs(grid, i, j);
      }
    }
  //最后返回岛屿的数量
  return count;
```

```
22    }
23
24    //这个方法会把当前格子以及他邻近的为1的格子都会置为1
25    public void dfs(char[][] grid, int i, int j) {
26      //边界条件判断，不能越界
27      if (i < 0 || i >= grid.length || j < 0 || j >= grid[0].length || grid[i][j] == '0')
28        return;
29      //把当前格子置为0，然后再从他的上下左右4个方向继续遍历
30      grid[i][j] = '0';
31      dfs(grid, i - 1, j);//上
32      dfs(grid, i + 1, j);//下
33      dfs(grid, i, j + 1);//左
34      dfs(grid, i, j - 1);//右
35    }
```

BFS解决

DFS就是沿着一条路径一直走下去，当遇到终止条件的时候才会返回，而BFS就是先把当前位置附近的访问一遍，就像下面这样先访问圈内的，然后再把圈放大继续访问，就像下面这样



这题使用BFS和DFS都能解决，如果遇到位置为1的格子，只要能把他们挨着的为1的全部置为0，然后挨着的挨着的为1的位置也置为0，然后……一直这样循环下去，看下代码

```
1     public int numIslands(char[][] grid) {
2       //边界条件判断
3       if (grid == null || grid.length == 0)
4         return 0;
5       //统计岛屿的个数
6       int count = 0;
7       //两个for循环遍历每一个格子
8       for (int i = 0; i < grid.length; i++)
9         for (int j = 0; j < grid[0].length; j++) {
10          //只有当前格子是1才开始计算
11          if (grid[i][j] == '1') {
12            //如果当前格子是1，岛屿的数量加1
13            count++;
14            //然后通过bfs把当前格子的上下左右4
15            //个位置为1的都要置为0，因为他们是连着
16            //一起的算一个岛屿，
```

```
            bfs(grid, i, j);
          }
        }
      return count;
    }

    private void bfs(char[][] grid, int x, int y) {
      //把当前格子先置为0
      grid[x][y] = '0';
      int n = grid.length;
      int m = grid[0].length;
      //使用队列，存储的是格子坐标转化的值
      Queue<Integer> queue = new LinkedList<>();
      //我们知道平面坐标是两位数字，但队列中存储的是一位数字，
      //所以这里是把两位数字转化为一位数字
      int code = x * m + y;
      //坐标转化的值存放到队列中
      queue.add(code);
      while (!queue.isEmpty()) {
        //出队
        code = queue.poll();
        //在反转成坐标值（i，j）
        int i = code / m;
        int j = code % m;
        if (i > 0 && grid[i - 1][j] == '1') {//上
          //如果上边格子为1，把它置为0，然后加入到队列中
          //下面同理
          grid[i - 1][j] = '0';
          queue.add((i - 1) * m + j);
        }
        if (i < n - 1 && grid[i + 1][j] == '1') {//下
          grid[i + 1][j] = '0';
          queue.add((i + 1) * m + j);
        }
        if (j > 0 && grid[i][j - 1] == '1') { //左
          grid[i][j - 1] = '0';
          queue.add(i * m + j - 1);
        }
        if (j < m - 1 && grid[i][j + 1] == '1') {//右
          grid[i][j + 1] = '0';
          queue.add(i * m + j + 1);
        }
      }
    }
```

总结

这题首先要搞懂岛屿是由什么组成的，如果都是1并且挨着的话那么他们只能算一个岛屿，所以当我们找到一个岛屿的时候，首先要把他变为0，然后再把它上下左右4个方向为1的也要变成0，因为他们挨着的算是一个岛屿，接着继续再把挨着的挨着的以同样的方式遍历……。

往期推荐

- 422，剑指 Offer-使用DFS和BFS解机器人的运动范围
- 417，BFS和DFS两种方式求岛屿的最大面积

422，剑指 Offer-使用DFS和BFS解机器人的运动范围

原创 山大王wld 数据结构和算法 8月5日

收录于话题
#剑指offer

27个 >



Knowing what you want is half the battle. Most people go through their whole lives not knowing what they want.

知道自己想要什么等于成功了一半，多数人一辈子浑浑噩噩也不知道自己想要什么。

问题描述

地上有一个m行n列的方格，从坐标 [0,0] 到坐标 [m-1,n-1] 。一个机器人从坐标 [0, 0] 的格子开始移动，它每次可以向左、右、上、下移动一格（不能移动到方格外），也不能进入行坐标和列坐标的数位之和大于k的格子。

例如，当k为18时，机器人能够进入方格 [35, 37] ，因为3+5+3+7=18。但它不能进入方格 [35, 38]，因为3+5+3+8=19。请问该机器人能够到达多少个格子？

示例 1：

> 输入：m = 2, n = 3, k = 1

> 输出：3

示例 2：

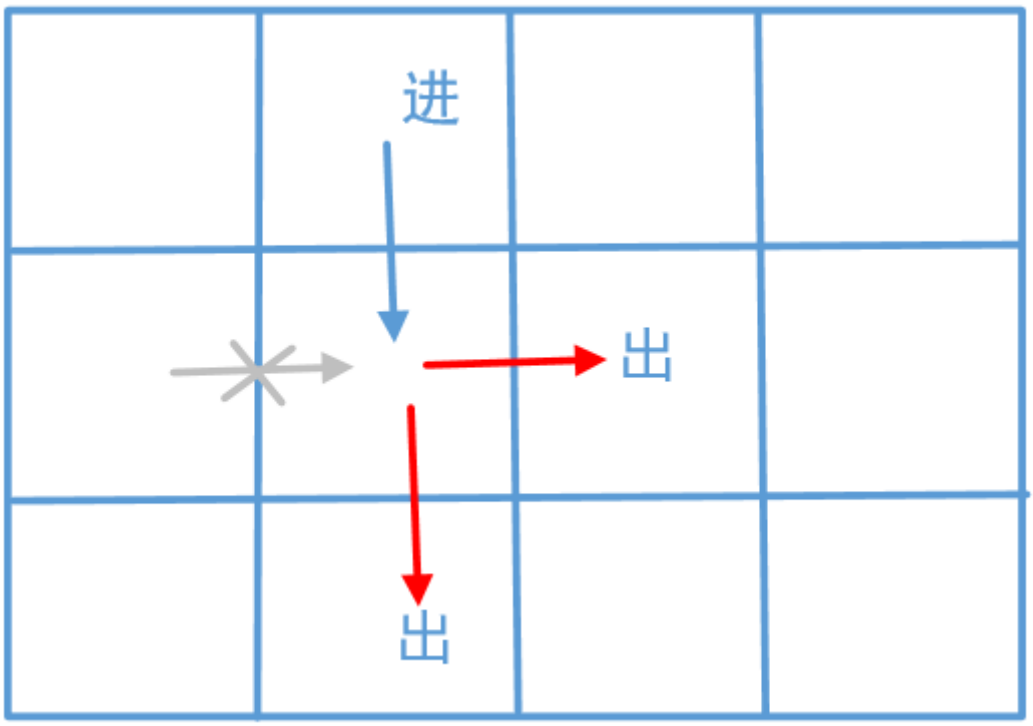
> 输入：m = 3，n = 1，k = 0
> 输出：1

提示：

- 1 <= n,m <= 100
- 0 <= k <= 20

DFS（深度优先搜索）

这道题说的是一个机器人从左上角开始，他可以沿着上下左右四个方向走，并且走到的每个格子坐标的数字和不大于k，问可以走多少个格子。我们先来画个图看一下



这里统计的是能走多少个格子，所以统计肯定是不能有重复的，题中说了，机器人是可以沿着上下左右四个方向走的。但你想一下，任何一个格子你从任何一个方向进来（比如从上面进来），那么他只能往其他3个方向走，因为如果在往回走就重复了。但实际上我们只要沿着两个方向走就可以了，一个是右边，一个是下边，也就是上面图中红色的箭头。我们来看下代码

```
1   public int movingCount(int m, int n, int k) {
2     //临时变量visited记录格子是否被访问过
3     boolean[][] visited = new boolean[m][n];
4     return dfs(0, 0, m, n, k, visited);
```

```
    }

    public int dfs(int i, int j, int m, int n, int k, boolean[][] visited) {
        //i >= m || j >= n是边界条件的判断，k < sum(i, j)判断当前格子坐标是否
        // 满足条件，visited[i][j]判断这个格子是否被访问过
        if (i >= m || j >= n || k < sum(i, j) || visited[i][j])
            return 0;
        //标注这个格子被访问过
        visited[i][j] = true;
        //沿着当前格子的右边和下边继续访问
        return 1 + dfs(i + 1, j, m, n, k, visited) + dfs(i, j + 1, m, n, k, visited);
    }

    //计算两个坐标数字的和
    private int sum(int i, int j) {
        int sum = 0;
        while (i != 0) {
            sum += i % 10;
            i /= 10;
        }
        while (j != 0) {
            sum += j % 10;
            j /= 10;
        }
        return sum;
    }
```

BFS（广度优先搜索）

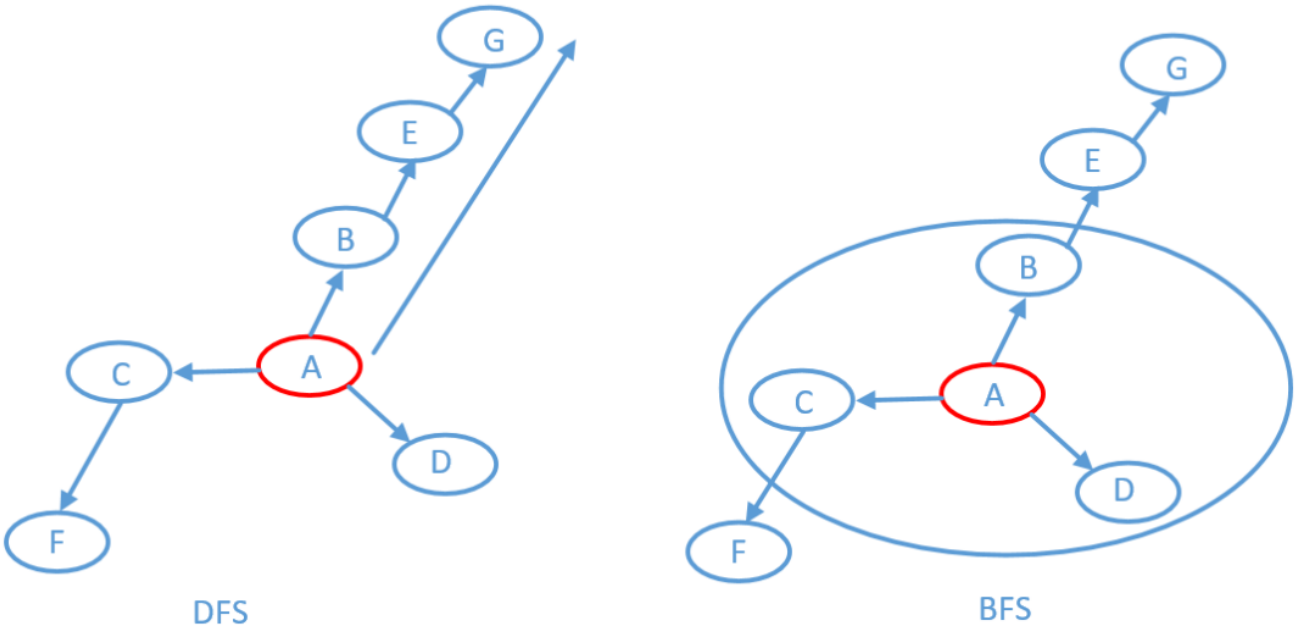
DFS是沿着一个方向一直往下走，有一种不撞南墙不回头的感觉，直到不满足条件才会回头。而BFS就显得有点博爱了，他不是一条道走下去，他会把离他最近的都访问一遍，访问完之后才开始访问第二近的……，一直这样下去，所以最好的一种数据结构就是使用队列，因为队列是先进先出，离他最近的访问完之后加入到队列中，最先入队的也是最先出队的，代码和上面有很多相似的地方，基本上没什么难度，来看下

```
public int movingCount(int m, int n, int k) {
    //临时变量visited记录格子是否被访问过
    boolean[][] visited = new boolean[m][n];
    int res = 0;
    //创建一个队列，保存的是访问到的格子坐标，是个二维数组
    Queue<int[]> queue = new LinkedList<>();
    //从左上角坐标[0,0]点开始访问，add方法表示把坐标
    // 点加入到队列的队尾
    queue.add(new int[]{0, 0});
    while (queue.size() > 0) {
        //这里的poll()函数表示的是移除队列头部元素，因为队列
        // 是先进先出，从尾部添加，从头部移除
        int[] x = queue.poll();
        int i = x[0], j = x[1];
        //i >= m || j >= n是边界条件的判断，k < sum(i, j)判断当前格子坐标是否
        // 满足条件，visited[i][j]判断这个格子是否被访问过
        if (i >= m || j >= n || k < sum(i, j) || visited[i][j])
            continue;
        //标注这个格子被访问过
        visited[i][j] = true;
        res++;
        //把当前格子右边格子的坐标加入到队列中
        queue.add(new int[]{i + 1, j});
        //把当前格子下边格子的坐标加入到队列中
        queue.add(new int[]{i, j + 1});
    }
    return res;
}
```

```
29
30  //计算两个坐标数字的和
31  private int sum(int i, int j) {
32    int sum = 0;
33    while (i != 0) {
34      sum += i % 10;
35      i /= 10;
36    }
37    while (j != 0) {
38      sum += j % 10;
39      j /= 10;
40    }
41    return sum;
42  }
```

总结

做这道题之前首先要明白DFS和BFS是什么意思，才能使用这两种方式。我们来画个图看一下



假如从A点开始访问，DFS就是沿着一条道走下去，然后再走其他的道……。BFS就是图中先访问圈内的部分，然后再把圈放大继续访问……。

往期推荐

- 420，剑指 Offer-回溯算法解矩阵中的路径
- 419，剑指 Offer-旋转数组的最小数字
- 417，BFS和DFS两种方式求岛屿的最大面积
- 413，动态规划求最长上升子序列

417，BFS和DFS两种方式求岛屿的最大面积

原创 山大王wld 数据结构和算法 7月31日

收录于话题
#算法图文分析

95个 >



You must practice being stupid, dumb, unthinking, empty.

你得学着痴一点，钝一些，少想一些，彻底放空自己。

问题描述

给定一个包含了一些0和1的非空二维数组grid 。

一个岛屿是由一些相邻的1(代表土地)构成的组合，这里的「相邻」要求两个1必须在水平或者竖直方向上相邻。你可以假设grid的四个边缘都被0（代表水）包围着。

找到给定的二维数组中最大的岛屿面积。(如果没有岛屿，则返回面积为0。)

示例 1:

```
[[0,0,1,0,0,0,0,1,0,0,0,0,0],
 [0,0,0,0,0,0,0,1,1,1,0,0,0],
 [0,1,1,0,1,0,0,0,0,0,0,0,0],
 [0,1,0,0,1,1,0,0,1,0,1,0,0],
 [0,1,0,0,1,1,0,0,1,1,1,0,0],
```

> [0,0,0,0,0,0,0,0,0,0,1,0,0],
> [0,0,0,0,0,0,0,1,1,1,0,0,0],
> [0,0,0,0,0,0,0,1,1,0,0,0,0]]

对于上面这个给定矩阵应返回 6。注意答案不应该是 11 ，因为岛屿只能包含水平或垂直的四个方向的 1 。

示例 2:

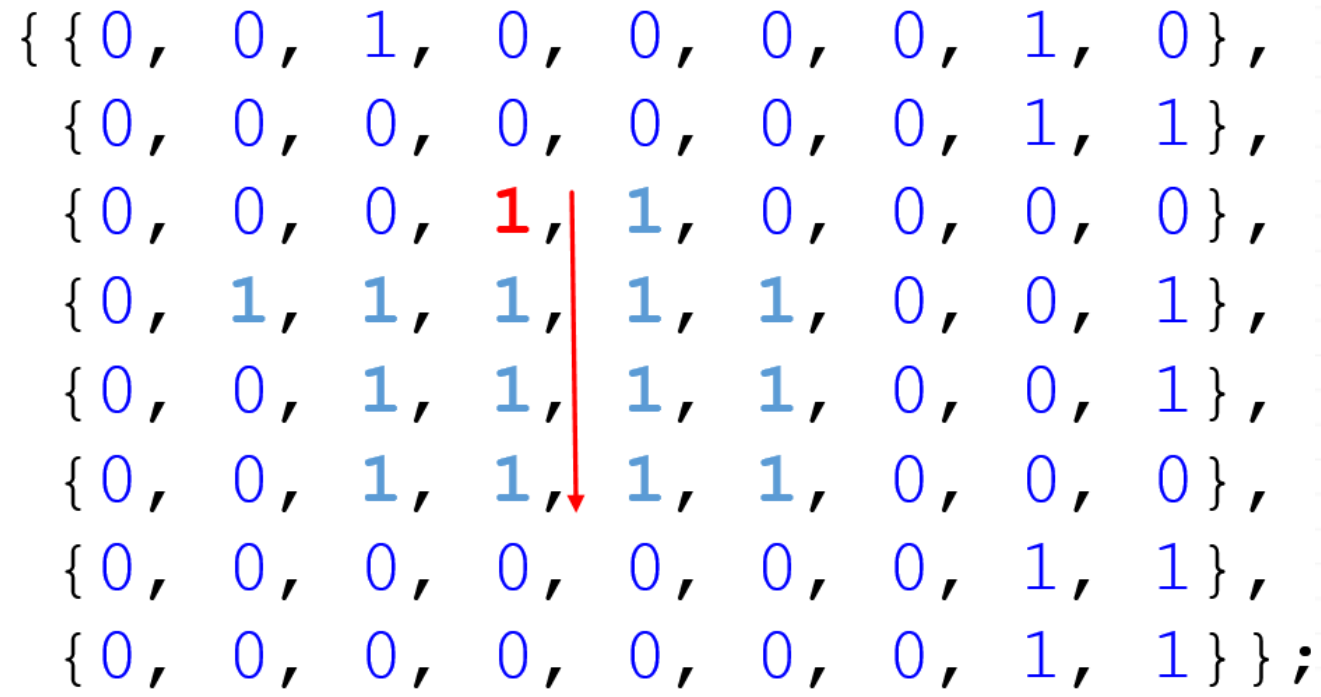
> [[0,0,0,0,0,0,0,0]]

对于上面这个给定的矩阵, 返回 0。

注意: 给定的矩阵grid 的长度和宽度都不超过 50。

DFS解决

这题无论使用DFS还是BFS都很好解决，DFS就是沿着一个方向一直走下去，直到不满足条件为止（要么走出grid的边缘，要么当前位置是0），就像下面这样，



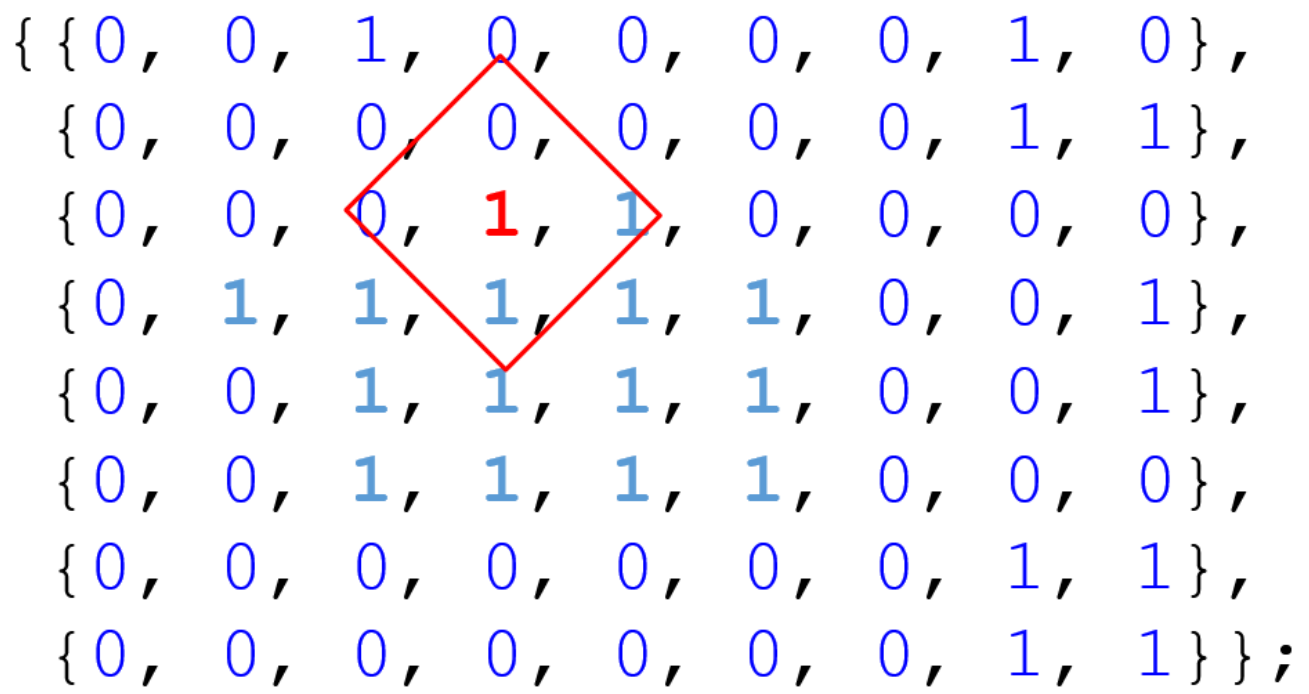
代码如下

```
1   public int maxAreaOfIsland(int[][] grid) {
2     int maxArea = 0;
3     for (int i = 0; i < grid.length; i++)
4       for (int j = 0; j < grid[0].length; j++)
5         if (grid[i][j] == 1) {//如果当前位置是1，开始计算
6           maxArea = Math.max(maxArea, dfs(grid, i, j));
7         }
8     return maxArea;
9   }
10
```

```java
public int dfs(int[][] grid, int i, int j) {
  //边界条件的判断
  if (i >= 0 && i < grid.length && j >= 0 && j < grid[0].length && grid[i][j] == 1) {
    //当前位置如果是1，为了防止重复计算就把他置为0，然后再从他的上下左右四个方向开始查找
    grid[i][j] = 0;
    return 1 + dfs(grid, i + 1, j) + dfs(grid, i - 1, j) + dfs(grid, i, j - 1) + dfs(grid, i, j + 1);
  }
  return 0;
}
```

BFS解决

BFS我们可以使用一个队列来实现，他的实现原理就是如果一个位置是1，我们就把他上下左右为1的点的坐标全部加入到队列中，然后改变当前位置的坐标为0，防止重复计算。加入队列之后再一个个出队，然后再以出队的那个点重复上面的操作……，直到队列为空为止。就像下面这样，假如遍历到红色的1，我们就把他上下左右为1的位置坐标全部加入到队列中。




```java
public int maxAreaOfIsland(int[][] grid) {
  int maxArea = 0;
  for (int i = 0; i < grid.length; i++)
    for (int j = 0; j < grid[0].length; j++)
      if (grid[i][j] == 1) {//如果当前位置是1，开始计算
        maxArea = Math.max(maxArea, bfs(grid, i, j));
      }
  return maxArea;
}

public int bfs(int[][] grid, int i, int j) {
  int m = grid.length, n = grid[0].length;
  if (grid[i][j] == 0)
    return 0;
  grid[i][j] = 0;
  //队列中存储的是个二维数组，这个二维数组就是格子的坐标
  Queue<int[]> queue = new LinkedList<>();
  //offer表示添加到队列的末尾
  queue.offer(new int[]{i, j});
```

```
    //分别表示右，左，下，上，四个方向
    int[][] dirs = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
    int res = 1;
    while (!queue.isEmpty()) {
      //poll表示从队列的头部移除一个元素
      int[] pos = queue.poll();
      //然后从pos坐标的4个方向再分别查找
      for (int[] dir : dirs) {
        int x = dir[0] + pos[0];
        int y = dir[1] + pos[1];
        //边界条件的判断
        if (x < 0 || x >= m || y < 0 || y >= n || grid[x][y] == 0) {
          continue;
        }
        grid[x][y] = 0;
        res++;
        queue.offer(new int[]{x, y});
      }
    }
    return res;
  }
```

总结

如果对图的遍历比较了解的话，这两种方式很容易想到，一个是沿着一个方向一直走下去，一个就像波浪一样，沿着一个点然后往四周一圈一圈的发散。

往期推荐

- 413，动态规划求最长上升子序列
- 405，换酒问题
- 398，双指针求无重复字符的最长子串
- 393，括号生成

597，双指针解验证回文字符串 II

原创 博哥 数据结构和算法 1周前

问题描述

来源：LeetCode第680题
难度：简单

给定一个非空字符串s，最多删除一个字符。判断是否能成为回文字符串。

示例 1:

> **输入**: s = "aba"
> **输出**: true

示例 2:

> **输入**: s = "abca"
> **输出**: true
> **解释**: 你可以删除c字符。

示例 3:

> **输入**: s = "abc"
> **输出**: false

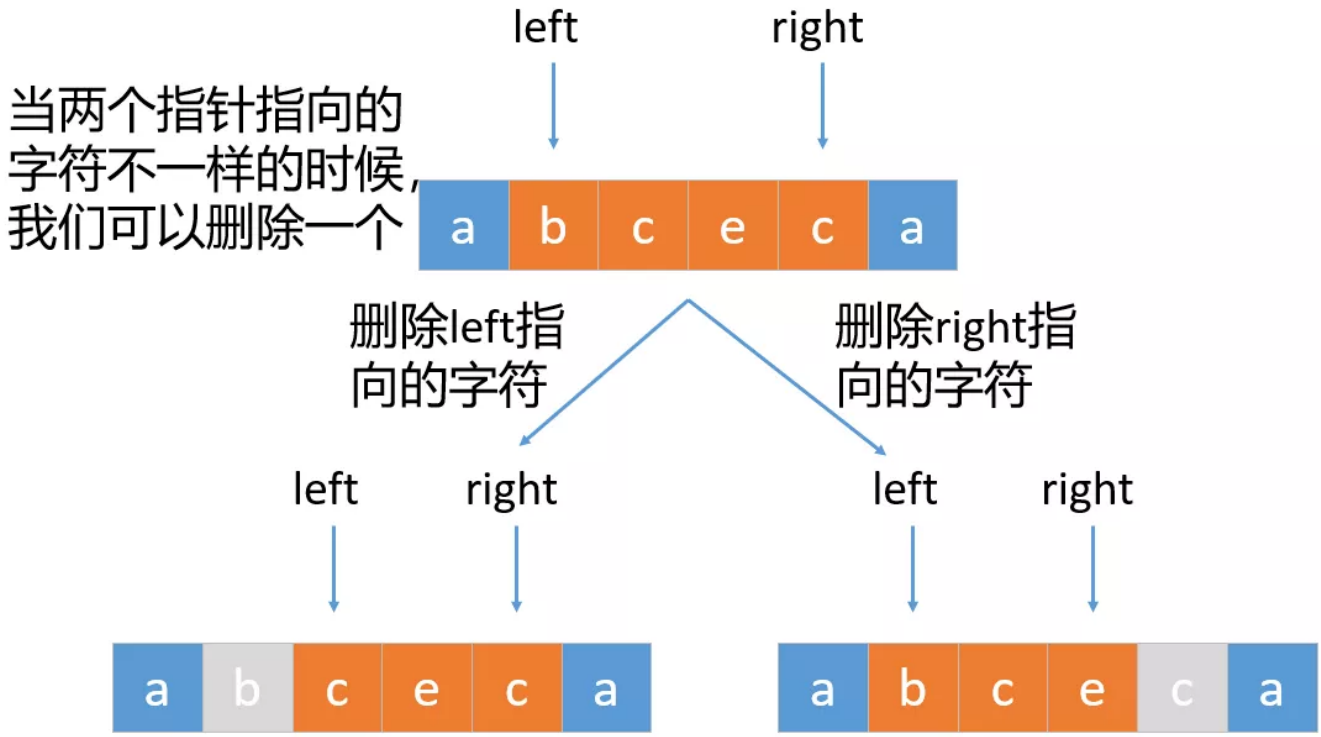
提示:

- 1 <= s.length <= 10^5
- s 由小写英文字母组成

双指针解决

如果只是验证是否是回文串，这个比较简单，之前也讲过497，双指针验证回文串。但这道题如果不是回文串，我们还可以删除一个字符，判断是否是回文的。

原理还和497题一样，使用两个指针left和right，从字符串的两边相向而行，如果两个指针指向的字符不相同，说明不能构成回文串，我们可以删除一个。可以删除left指向的字符也可以删除right指向的字符，如下图所示



原理比较简单，就不在过多介绍，我们直接看下代码。

```
public boolean validPalindrome(String s) {
  //左指针
  int left = 0;
  //右指针
  int right = s.length() - 1;
  while (left < right) {
    //如果两个指针指向的字符不一样，我们要删除一个，要么
    // 删除left指针指向的值，要么删除right指针指向的值
    if (s.charAt(left) != s.charAt(right)) {
      return isPalindromic(s, left + 1, right)
          || isPalindromic(s, left, right - 1);
    }
    left++;
    right--;
  }
  return true;
}

//判断子串[left,right]是否是回文串
private boolean isPalindromic(String s, int left, int right) {
  while (left < right) {
    if (s.charAt(left++) != s.charAt(right--)) {
      return false;
    }
  }
  return true;
}
```

时间复杂度：O（n），n是字符串的长度

空间复杂度：O（1），需要额外的常数大小的辅助空间。

549，滑动窗口解可获得的最大点数

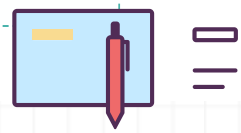
原创 博哥 数据结构和算法 5月9日

收录于话题
#算法图文分析

161个 >

Summer has filled her veins with light and her heart is washed with noon.

夏日使她血管里充满光，她温暖的心受午间洗沐。



问题描述

几张卡牌排成一行，每张卡牌都有一个对应的点数。点数由整数数组cardPoints给出。

每次行动，你可以从行的开头或者末尾拿一张卡牌，最终你必须正好拿k张卡牌。

你的点数就是你拿到手中的所有卡牌的点数之和。

给你一个整数数组cardPoints和整数k，请你返回可以获得的最大点数。

示例 1：

> **输入**：cardPoints = [1,2,3,4,5,6,1], k = 3
> **输出**：12
> **解释**：第一次行动，不管拿哪张牌，你的点数总是 1 。但是，先拿最右边的卡牌将会最大化你的可获得点数。最优策略是拿右边的三张牌，最终点数为 1 + 6 + 5 = 12 。

示例 2：

> **输入**：cardPoints = [2,2,2], k = 2
> **输出**：4
> **解释**：无论你拿起哪两张卡牌，可获得的点数总是 4 。

示例 3：

> **输入**：cardPoints = [9,7,7,9,7,7,9], k = 7
> **输出**：55
> **解释**：你必须拿起所有卡牌，可以获得的点数为所有卡牌的点数之和。

示例 4：

> **输入**：cardPoints = [1,1000,1], k = 1
> **输出**：1
> **解释**：你无法拿到中间那张卡牌，所以可以获得的最大点数为 1 。

示例 5：

> **输入**：cardPoints = [1,79,80,1,1,1,200,1], k = 3
> **输出**：202

提示：

- 1 <= cardPoints.length <= 10^5
- 1 <= cardPoints[i] <= 10^4
- 1 <= k <= cardPoints.length

滑动窗口解决

每次拿的时候只能从开头和末尾拿，而不能从中间拿。我们换种思路，如果把数组的首尾相连，串成一个环形，那么最终拿掉的k个元素肯定是连续的，问题就转化为求k个连续元素的最大和，所以我们很容易想到的就是**滑动窗口**。

但这个窗口有个限制条件，就是窗口内的元素至少包含原数组首尾元素中的一个。

我们就以示例一为例来看下视频。



最后再来看下代码

```java
public int maxScore(int[] cardPoints, int k) {
    int maxWindow = 0, length = cardPoints.length;
    //先统计前k个元素的和，也是窗口内元素的和
    for (int i = 0; i < k; i++)
        maxWindow += cardPoints[i];
    //然后窗口移动，更新当前窗口的值
    int curWindow = maxWindow;
    for (int i = length - 1; i >= length - k; i--) {
        //窗口移动的时候一个元素会出窗口，一个元素会进入窗口。
        //cardPoints[k - (length - i)]是移除窗口的元素
        curWindow -= cardPoints[k - (length - i)];
        //cardPoints[i]是进入窗口的元素
        curWindow += cardPoints[i];
        //记录窗口的最大值
        maxWindow = Math.max(maxWindow, curWindow);
    }
    return maxWindow;
}
```

总结

咋一看，无从下手，其实换个思路就很好解决了。

往期推荐

- 542，滑动窗口解最小覆盖子串
- 521，滑动窗口解最大连续1的个数 III
- 443，滑动窗口最大值
- 407，动态规划和滑动窗口解决最长重复子数组

542，滑动窗口解最小覆盖子串

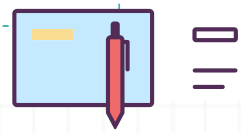
原创 博哥 数据结构和算法 今天

收录于话题
#算法图文分析

147个 >

Spring is when you feel like whistling even with a shoe full of slush.

所谓春天，就是即使鞋子灌满泥巴，仍然想吹起口哨。



问题描述

给你一个字符串s、一个字符串t。返回s中涵盖t所有字符的最小子串。如果s中不存在涵盖t所有字符的子串，则返回空字符串""。

注意：如果s中存在这样的子串，我们保证它是唯一的答案。

示例 1：

> **输入**：s = "ADOBECODEBANC", t = "ABC"
> **输出**："BANC"

示例 2：

> **输入**：s = "a", t = "a"
> **输出**："a"

提示：

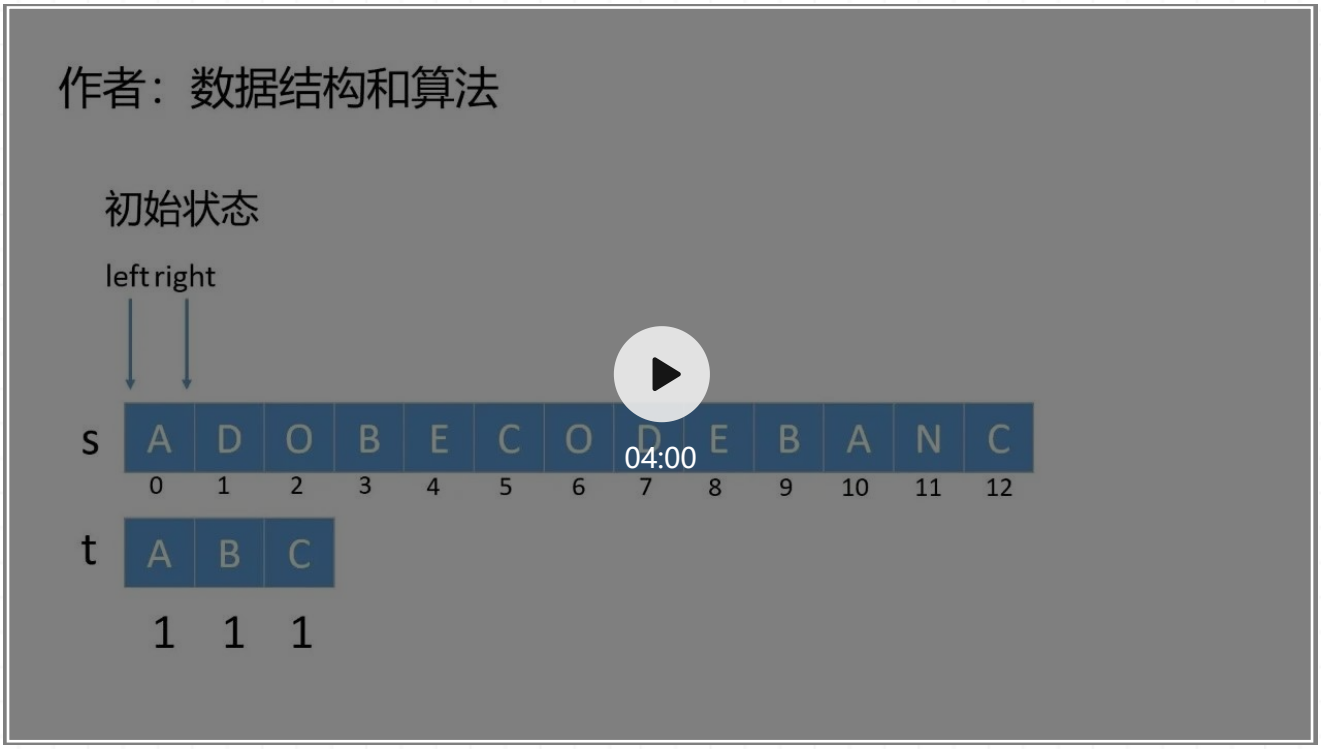
- 1 <= s.length, t.length <= 10^5
- s 和 t 由英文字母组成

滑动窗口解决

这题让求的是s中能覆盖t的最小子串，看到这题首先想到的就是滑动窗口。使用两个指针一个left，一个right，分别表示窗口的左边界和右边界。

- 当窗口内的所有字符不能覆盖t的时候，要扩大窗口，也就是right往右移。
- 当窗口内的所有字符可以覆盖t的时候，记录窗口的起始位置以及窗口的长度，然后缩小窗口（因为这里求的是能覆盖的最小子串），left往右移。如果缩小的窗口还能覆盖t，保存长度最小的窗口即可。
- 重复上面的操作，直到窗口的右边不能再移动为止。

这里我以示例1为例做个视频，来看一下具体操作过程，加深一下理解。（如果看不清，可以切换横向全屏观看）



原理搞懂了，代码就简单多了，但是这里有个关键点，就是怎么记录窗口内的元素，其实很简单，使用一个map就可以，来看下代码。

```
public String minWindow(String s, String t) {
  //把t中的字符全部放到map中
  Map<Character, Integer> map = new HashMap<>();
  for (char ch : t.toCharArray())
    map.put(ch, map.getOrDefault(ch, 0) + 1);

  int left = 0;//窗口的左边界
  int right = 0;//窗口的右边界

  //满足条件的窗口开始位置
  int strStart = 0;
  //满足条件的窗口的长度
  int windowLength = Integer.MAX_VALUE;

  while (right < s.length()) {
    //记录右指针扫描过的字符
    char rightChar = s.charAt(right);
    //如果右指针扫描的字符存在于map中，就减1
    if (map.containsKey(rightChar))
      map.put(rightChar, map.getOrDefault(rightChar, 0) - 1);
```

```
        //记录之后右指针要往右移
        right++;

        //检查窗口是否把t中字符全部覆盖了，如果覆盖了，要移动窗口的左边界
        //找到最小的能全部覆盖的窗口
        while (check(map)) {
          //如果现在窗口比之前保存的还要小，就更新窗口的长度
          //以及窗口的起始位置
          if (right - left < windowLength) {
            windowLength = right - left;
            strStart = left;
          }
          //移除窗口最左边的元素，也就是缩小窗口
          char leftChar = s.charAt(left);
          if (map.containsKey(leftChar))
            map.put(leftChar, map.getOrDefault(leftChar, 0) + 1);
          //左指针往右移
          left++;
        }
      }
      //如果找到合适的窗口就截取，否则就返回空
      if (windowLength != Integer.MAX_VALUE)
        return s.substring(strStart, strStart + windowLength);
      return "";
    }

    //检查窗口是否把字符串t中的所有字符都覆盖了，如果map中所有
    // value的值都不大于0，则表示全部覆盖
    private boolean check(Map<Character, Integer> map) {
      for (int value : map.values()) {
        //注意这里的value是可以为负数的，为负数的情况就是，相同的字符右
        // 指针扫描的要比t中的多，比如t是"ABC"，窗口中的字符是"ABBC"
        if (value > 0)
          return false;
      }
      return true;
    }
```

上面注释已经写的很详细了，基本上也都能看懂，实际上我们还可以把HashMap换成数组，原理其实都是一样的，来看下代码

```
1   public String minWindow(String s, String t) {
2     int[] map = new int[128];
3     //记录字符串t中每个字符的数量
4     for (char ch : t.toCharArray())
5       map[ch]++;
6     //字符串t的数量
7     int count = t.length();
8     int left = 0;//窗口的左边界
9     int right = 0;//窗口的右边界
10    //覆盖t的最小长度
11    int windowLength = Integer.MAX_VALUE;
12    //覆盖字符串t开始的位置
13    int strStart = 0;
14    while (right < s.length()) {
15      if (map[s.charAt(right++)]-- > 0)
16        count--;
17      //如果全部覆盖
18      while (count == 0) {
19        //如果有更小的窗口就记录更小的窗口
20        if (right - left < windowLength) {
21          windowLength = right - left;
22          strStart = left;
23        }
24        if (map[s.charAt(left++)]++ == 0)
25          count++;
26      }
27    }
28    //如果找到合适的窗口就截取，否则就返回空
```

```
29      if (windowLength != Integer.MAX_VALUE)
30          return s.substring(strStart, strStart + windowLength);
31      return "";
32  }
```

总结

滑动窗口类型的题也是最常见的，一般会有两个指针，分别指向窗口的左边界和右边界，如果窗口不满足条件我们就移动右边界来扩大窗口，如果满足条件我们可以移动左边界来缩小窗口，确定这个更小的窗口是否还满足条件……

往期推荐

- 521，滑动窗口解最大连续1的个数 III
- 443，滑动窗口最大值
- 407，动态规划和滑动窗口解决最长重复子数组
- 397，双指针求接雨水问题

539，双指针解删除有序数组中的重复项

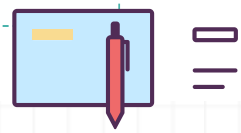
原创 博哥 数据结构和算法 今天

收录于话题
#算法图文分析

144个 >

Years may wrinkle the skin, but to give up enthusiasm wrinkles the soul.

岁月留痕，只及肌肤；激情不再，皱起心灵。



问题描述

给你一个有序数组nums ，请你原地删除重复出现的元素，使每个元素只出现一次 ，返回删除后数组的新长度。

不要使用额外的数组空间，你必须在原地修改输入数组并在使用O(1)额外空间的条件下完成。

示例 1：

> **输入**：nums = [1,1,2]
> **输出**：2, nums = [1,2]
> **解释**：函数应该返回新的长度 2 ，并且原数组 nums 的前两个元素被修改为 1，2 。不需要考虑数组中超出新长度后面的元素。

示例 2：

> **输入**：nums = [0,0,1,1,1,2,2,3,3,4]
> **输出**：5, nums = [0,1,2,3,4]
> **解释**：函数应该返回新的长度 5 ， 并且原数组 nums 的前五个元素被修改为 0，1，2，3，4 。不需要考虑数组中超出新长度后面的元素。

提示：

- 0 <= nums.length <= 3 * 10^4
- -10^4 <= nums[i] <= 10^4
- nums 已按升序排列

双指针解决

因为数组是**排序**的，只要是相同的肯定是挨着的，我们只需要遍历所有数组，然后前后两两比较，如果有相同的就把后面的给删除。

使用两个指针，**右指针始终往右移动**，

- **如果右指针指向的值等于左指针指向的值，左指针不动。**
- **如果右指针指向的值不等于左指针指向的值，那么左指针往右移一步，然后再把右指针指向的值赋给左指针。**

具体看下视频



来看下代码

```
1   //双指针解决
2   public int removeDuplicates(int[] A) {
3     //边界条件判断
4     if (A == null || A.length == 0)
5       return 0;
6     int left = 0;
7     for (int right = 1; right < A.length; right++)
8       //如果左指针和右指针指向的值一样，说明有重复的，
9       //这个时候，左指针不动，右指针继续往右移。如果他俩
10      //指向的值不一样就把右指针指向的值往前挪
11      if (A[left] != A[right])
12        A[++left] = A[right];
```

```
    return ++left;
  }
```

或者还可以换一种解法，其实原理都是一样的。

```
public int removeDuplicates(int[] A) {
  int count = 0;//重复的数字个数
  for (int right = 1; right < A.length; right++) {
    if (A[right] == A[right - 1]) {
      //如果有重复的，count要加1
      count++;
    } else {
      //如果没有重复，后面的就往前挪
      A[right - count] = A[right];
    }
  }
  //数组的长度减去重复的个数
  return A.length - count;
}
```

往期推荐

- 514，双指针解替换后的最长重复字符
- 497，双指针验证回文串
- 447，双指针解旋转链表
- 398，双指针求无重复字符的最长子串

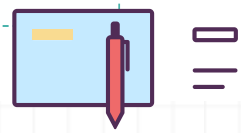
538，剑指 Offer-和为s的连续正数序列

原创 博哥 数据结构和算法 前天

收录于话题
#剑指offer

32个 >

> You can know everything in the world, but the only way you're findin' out that one is by givin' it a shot.
>
> 你可以了解世间万物，但追根溯源的唯一途径便是亲身尝试。



问题描述

输入一个正整数target，输出所有和为target的连续正整数序列（至少含有两个数）。

序列内的数字由小到大排列，不同序列按照首个数字从小到大排列。

示例 1：

> **输入：** target = 9
> **输出：** [[2,3,4],[4,5]]

示例 2：

> **输入：** target = 15
> **输出：** [[1,2,3,4,5],[4,5,6],[7,8]]

限制：

- 1 <= target <= 10^5

滑动窗口解决

滑动窗口，这里也叫双指针，因为题中要求的是正整数，连续的，并且至少含有两个数。所以我们使用两个指针，一个left指向1，一个right指向2，他们分别表示窗口的左边界和右边界。然后计算窗口内元素的和。

- **如果窗口内的值大于target，说明窗口大了，left往右移一步。**
- **如果窗口内的值小于target，说明窗口小了，right往右移一步。**
- **如果窗口内的值等于target，说明找到了一组满足条件的序列，把它加入到列表中**

我们以示例1为例来看下视频演示



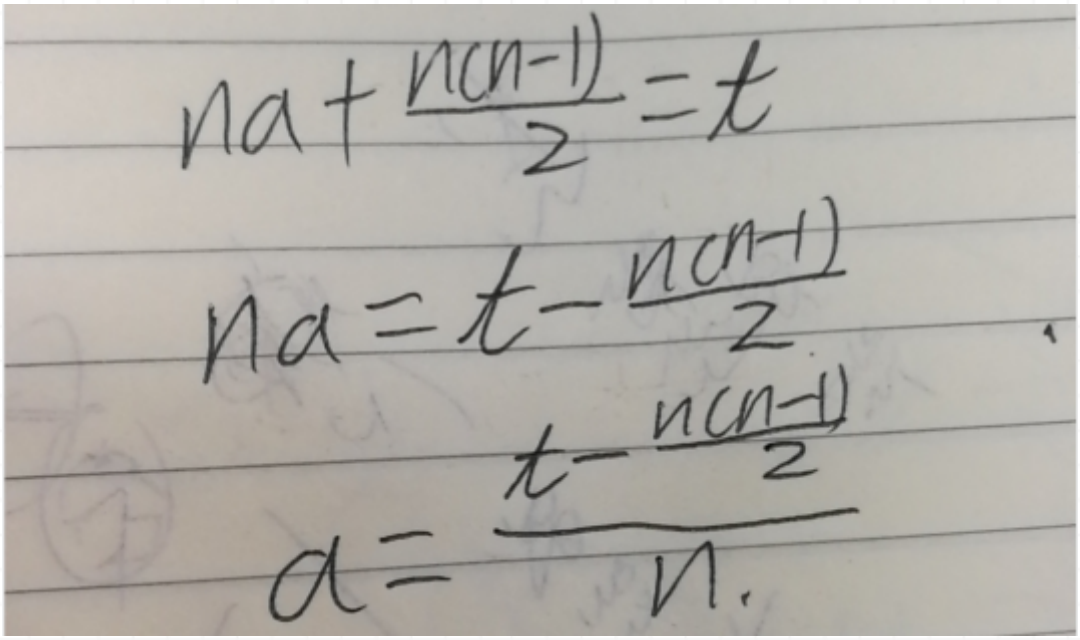
因为至少有两个数，所以窗口的左边界left <= target / 2，题中是把找到的序列添加到列表list中，最后在转化为二维数组，来看下代码

```
public int[][] findContinuousSequence(int target) {
    int left = 1; // 滑动窗口的左边界
    int right = 2; // 滑动窗口的右边界
    int sum = left + right; // 滑动窗口中数字的和
    List<int[]> res = new ArrayList<>();
    //窗口的左边是窗口内的最小数字，只能小于等于target / 2,
    //因为题中要求的是至少含有两个数
    while (left <= target / 2) {
        if (sum < target) {
            //如果窗口内的值比较小，右边界继续向右移动，
            //来扩大窗口
            sum += ++right;
        } else if (sum > target) {
            //如果窗口内的值比较大，左边边界往右移动，
            //缩小窗口
            sum -= left++;
        } else {
            //如果窗口内的值正好等于target，就把窗口内的值记录
            //下来，然后窗口的左边和右边同时往右移一步
            int[] arr = new int[right - left + 1];
            for (int k = left; k <= right; k++) {
                arr[k - left] = k;
            }
            res.add(arr);
```

```
            //左边和右边同时往右移一位
            sum -= left++;
            sum += ++right;
        }
    }
    //把结果转化为数组
    return res.toArray(new int[res.size()][]);
}
```

数学公式解决

我们假设有一组序列满足条件，其中序列的第一个数是a，他们分别是a，a+1，a+2，……，a+(n-1)，总共有n项，根据求和公式我们可以得出S=n*a+n*(n-1)/2，而S其实就是target，我们简写成t，来研究一下这个公式



要想求a，我们可以通过循环枚举n的值，只有a是正整数的时候才满足条件，那么这个循环什么时候终止呢，其实很简单，当分子t-n*(n-1)/2小于等于0的时候就可以终止了。

因为题中说了最少要有2个数，所以n从2开始，来看下代码

```
public int[][] findContinuousSequence(int target) {
    List<int[]> res = new ArrayList<>();
    int n = 2;
    //死循环
    while (true) {
        int total = target - n * (n - 1) / 2;
        //当分子小于等于0的时候，退出循环
        if (total <= 0)
            break;
        //如果首项是正整数，满足条件
        if (total % n == 0) {
            int[] arr = new int[n];
            //找出首项的值
            int startValue = total / n;
            for (int k = 0; k < n; k++) {
                arr[k] = startValue + k;
            }
            res.add(arr);
        }
        //继续找
        n++;
    }
```

```
    //反转，比如当target等于9的时候，结果是
    //[[4,5],[2,3,4]]，但题中要求的是不同
    // 序列按照首个数字从小到大排列，所以这里反转一下
    Collections.reverse(res);
    //把list转化为数组
    return res.toArray(new int[res.size()][]);
  }
```

数学的另一种解决方式

我们来思考这样一个问题

- **假如target是两个连续数字的和，那么这个序列的首项就是(target-1)/2。**
- **假如target是三个连续数字的和，那么这个序列的首项就是(target-1-2)/3。**
- **假如target是四个连续数字的和，那么这个序列的首项就是(target-1-2-3)/4。**
- **……**

证明也很好证，我们随便找一个，假如target是四个连续的序列和，那么这四个数字就是

a，a+1,a+2,a+3

也就是

4*a+1+2+3=target

所以他们的首项

a=(target-1-2-3)/4。

搞懂了上面的原理代码就简单了，我们来看下

```
public int[][] findContinuousSequence(int target) {
    List<int[]> res = new ArrayList<>();
    //因为至少是两个数，所以target先减1
    target--;
    for (int n = 2; target > 0; n++) {
      //找到了一组满足条件的序列
      if (target % n == 0) {
        int[] arr = new int[n];
        //找出首项的值
        int startValue = target / n;
        for (int k = 0; k < n; k++) {
          arr[k] = startValue + k;
        }
        res.add(arr);
      }
      target -= n;
    }
    Collections.reverse(res);
    //把list转化为数组
    return res.toArray(new int[res.size()][]);
  }
```

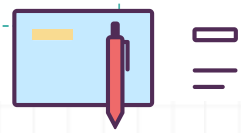
527，两个数组的交集 II

原创 博哥 数据结构和算法 今天

收录于话题
#算法图文分析

137个 >

> Doing things change things, no doing things, these things are exactly as they were.
>
> 行动才能改变，没有行动，一切也会原封不动。



问题描述

给定两个数组，编写一个函数来计算它们的**交集**。

示例 1：

> **输入**：nums1 = [1,2,2,1], nums2 = [2,2]
> **输出**：[2,2]

示例 2：

> **输入**：nums1 = [4,9,5], nums2 = [9,4,9,8,4]
> **输出**：[4,9]

说明：

- 输出结果中每个元素出现的次数，应与元素在两个数组中出现次数的最小值一致。
- 我们可以不考虑输出结果的顺序。

双指针解决

先对两个数组进行**排序**，然后使用**两个指针**，分别指向两个数组开始的位置。

- 如果两个指针指向的值**相同**，说明这个值是他们的交集，就把这个值加入到集合list中，然后两个指针在分别往后移一步。
- 如果两个指针指向的值**不同**，那么指向的值相对小的往后移一步，相对大的先不动，然后再比较

一直重复上面的操作，直到其中一个指针不能再移动为止，最后再把集合list转化为数组即可。来看下视频



在来看下代码

```java
public int[] intersect(int[] nums1, int[] nums2) {
  // 先对两个数组进行排序
  Arrays.sort(nums1);
  Arrays.sort(nums2);
  int i = 0;
  int j = 0;
  List<Integer> list = new ArrayList<>();
  while (i < nums1.length && j < nums2.length) {
    if (nums1[i] < nums2[j]) {
      // 如果i指向的值小于j指向的值，，说明i指向
      // 的值小了，i往后移一步
      i++;
    } else if (nums1[i] > nums2[j]) {
      // 如果i指向的值大于j指向的值，说明j指向的值
      // 小了，j往后移一步
      j++;
    } else {
      // 如果i和j指向的值相同，说明这两个值是重复的，
      // 把他加入到集合list中，然后i和j同时都往后移一步
      list.add(nums1[i]);
      i++;
      j++;
    }
  }
  //把list转化为数组
  int index = 0;
  int[] res = new int[list.size()];
  for (int k = 0; k < list.size(); k++) {
    res[index++] = list.get(k);
```

```
    }
    return res;
}
```

使用map解决

还可以使用map来解决，具体操作如下

- 遍历nums1中的所有元素，把它存放到map中，其中key就是nums1中的元素，value就是这个元素在数组nums1中出现的次数。
- 遍历nums2中的所有元素，查看map中是否包含nums2的元素，如果包含，就把当前值加入到集合list中，然后对应的value要减1。

最后再把集合list转化为数组即可，代码如下

```
public int[] intersect(int[] nums1, int[] nums2) {
    HashMap<Integer, Integer> map = new HashMap<>();
    ArrayList<Integer> list = new ArrayList<>();

    //先把数组nums1的所有元素都存放到map中，其中key是数组中
    //的元素，value是这个元素出现在数组中的次数
    for (int i = 0; i < nums1.length; i++) {
        map.put(nums1[i], map.getOrDefault(nums1[i], 0) + 1);
    }

    //然后再遍历nums2数组，查看map中是否包含nums2的元素，如果包含，
    //就把当前值加入到集合list中，然后再把对应的value值减1。
    for (int i = 0; i < nums2.length; i++) {
        if (map.getOrDefault(nums2[i], 0) > 0) {
            list.add(nums2[i]);
            map.put(nums2[i], map.get(nums2[i]) - 1);
        }
    }

    //把集合list转化为数组
    int[] res = new int[list.size()];
    for (int i = 0; i < list.size(); i++) {
        res[i] = list.get(i);
    }
    return res;
}
```


往期推荐

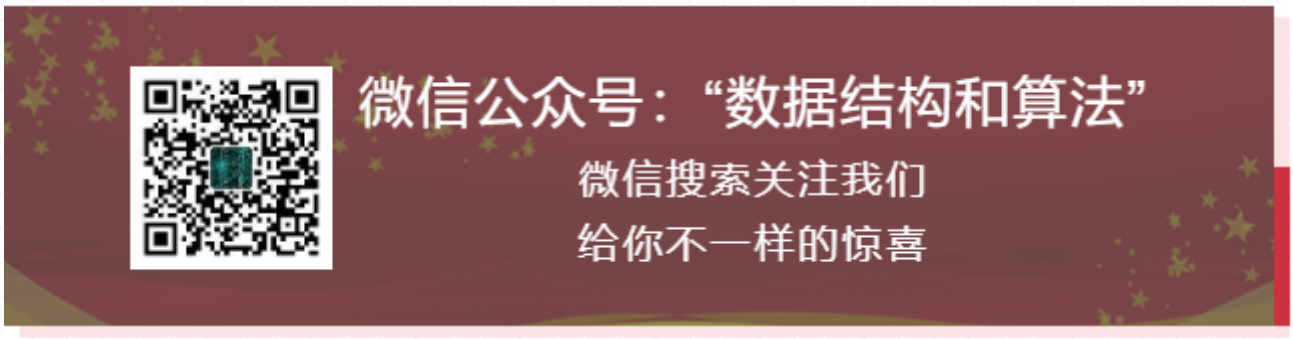
- 516，贪心算法解按要求补齐数组
- 509，数组中的第K个最大元素
- 504，旋转数组的3种解决方式
- 475，有效的山脉数组

514，双指针解替换后的最长重复字符

原创 山大王wld 数据结构和算法 2月4日

收录于话题
#算法图文分析

137个 >



In one second your whole life can change. It only takes a moment for everything to feel quite different.

生命真是瞬息万变，只要片刻，一切就截然不同了。

问题描述

给你一个仅由大写英文字母组成的字符串，你可以将任意位置上的字符替换成另外的字符，总共可最多替换k次。在执行上述操作后，找到包含重复字母的最长子串的长度。

注意：字符串长度 和 k 不会超过 10^4。

示例 1：

> **输入**：s = "ABAB", k = 2
> **输出**：4
> **解释**：用两个'A'替换为两个'B',反之亦然。

示例 2：

> **输入**：s = "AABABBA", k = 1
> **输出**：4
> **解释**：
> 将中间的一个'A'替换为'B',字符串变为 "AABBBBA"。
> 子串 "BBBB" 有最长重复字母，答案为 4。

双指针解决

这是一道典型的滑动窗口问题。在一个窗口内假如出现次数最多的那个字符出现的次数是a，窗口的长度是b，只要满足a+k>=b，我们就可以把窗口中的其他字符全部替换为出现次数最多的那个字符。
比如在窗口中有字符串"ABAABAB"，如果k大于等于3，我们就可以把字符串中的所有字符B替换为A。

相反如果a+k<b，我们是没法把窗口内的其他字符全部替换为出现次数最多的那个字符。比如字符串"ABAABBA"，如果k小于3，我们是不能把字符串中的所有字符B全部替换为A的。

搞懂了上面的分析过程，我们再来看一下这题的解决思路。

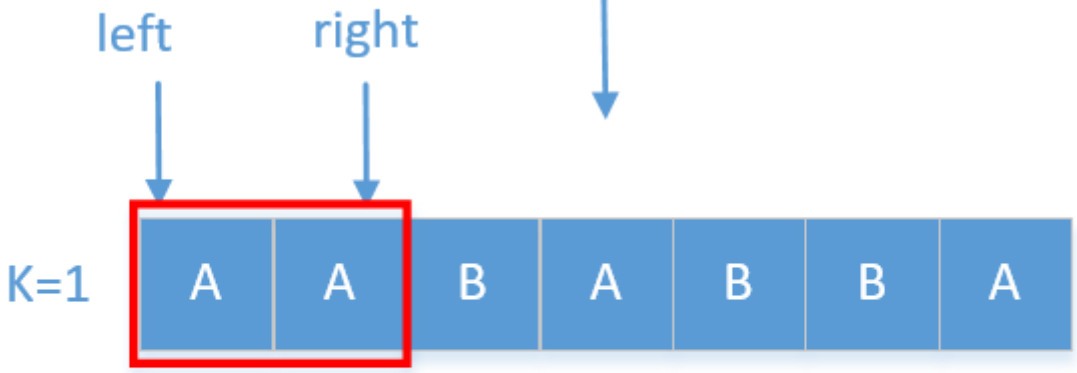
首先使用两个指针left和right，分别指向窗口的左边和右边。刚开始的时候left和right都指向第一个字符，也就是窗口的大小是1。

接着移动right，也就是扩大窗口，然后再判断窗口内相同字母最多的数量加上K是否小于窗口的大小。如果小于，说明窗口内其他的字母不能替换为最多的那个字母，我们要移动left，也就是缩小窗口的大小。如果大于，说明窗口内其他的字母是可以替换为最多的那个字母的，然后窗口左边界不变，移动右边界，扩大窗口，继续上面的循环……，直到右边界超出字符串为止。

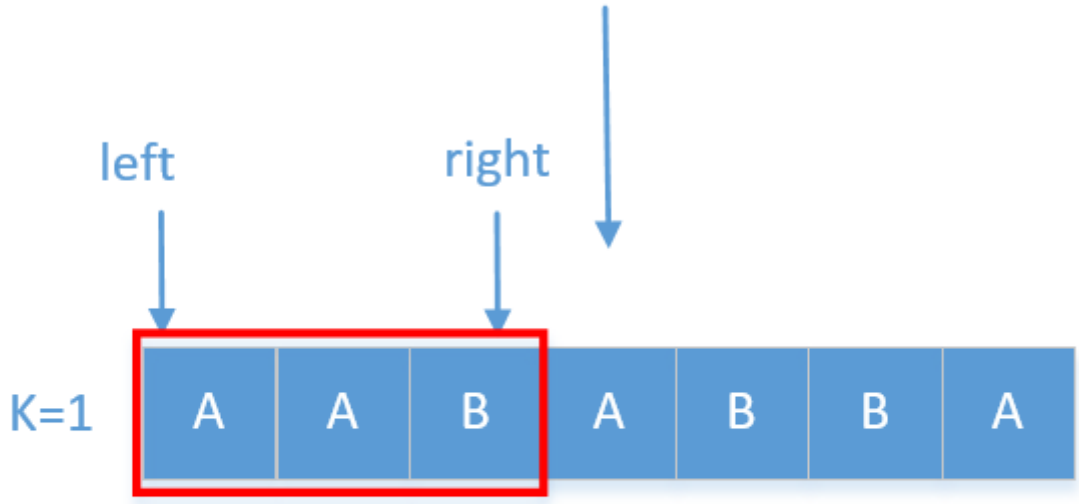
我们就以示例2为例画个图来看一下会更明白



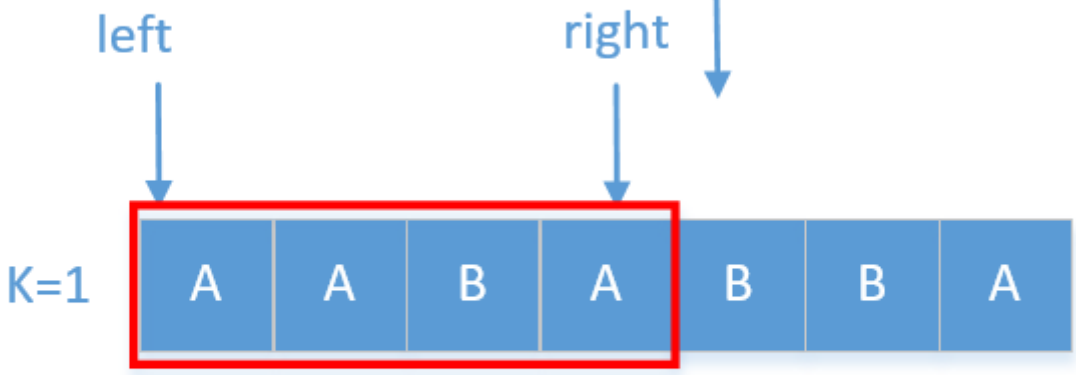
第1步窗口大小是1，最多的字母是A，个数是1，满足条件，左边不动，右边往右移，扩大窗口



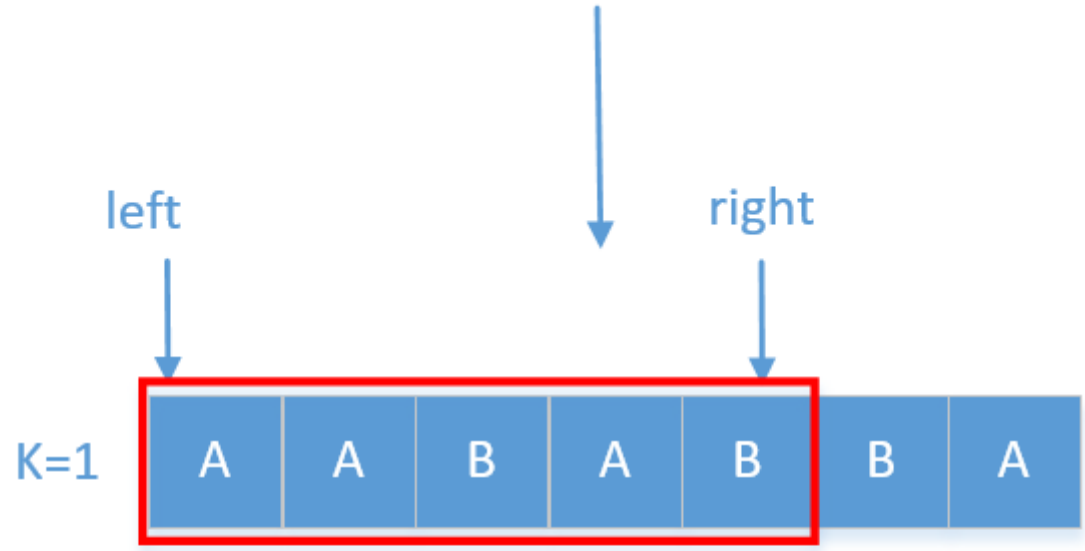
第2步窗口大小是2，最多的字母是A，个数是2，也满足条件，左边不动，右边往右移，扩大窗口



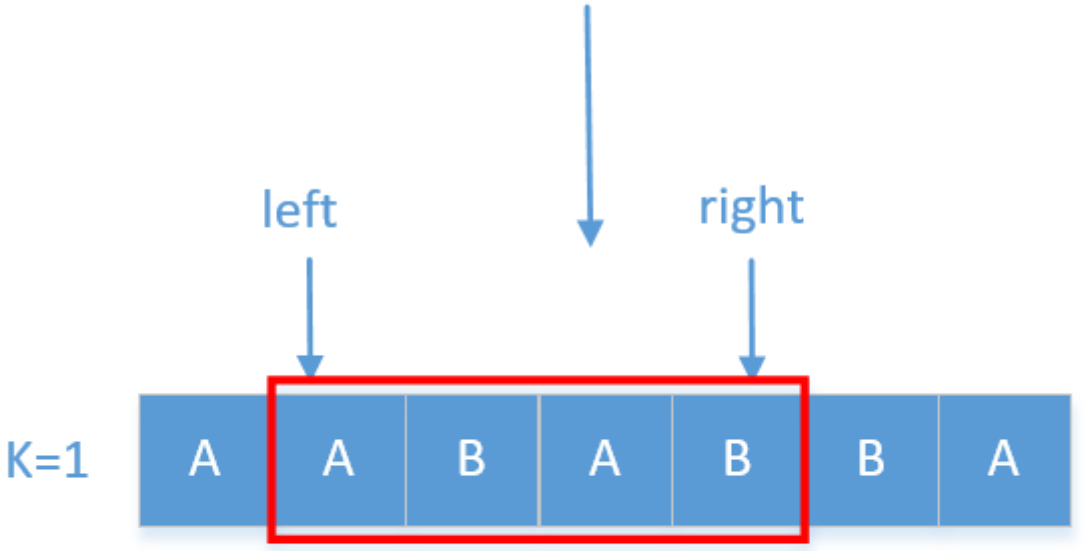
第3步窗口大小是3，最多的字母是A，个数是2，也满足条件，左边不动，右边往右移，扩大窗口



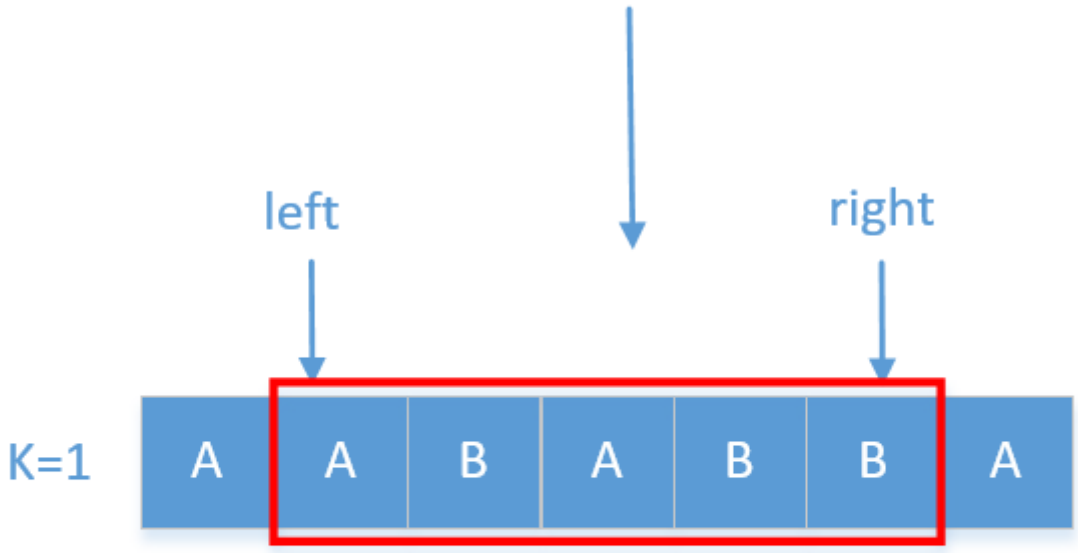
第4步窗口大小是4，最多的字母是A，个数是3，也满足条件，左边不动，右边往右移，扩大窗口



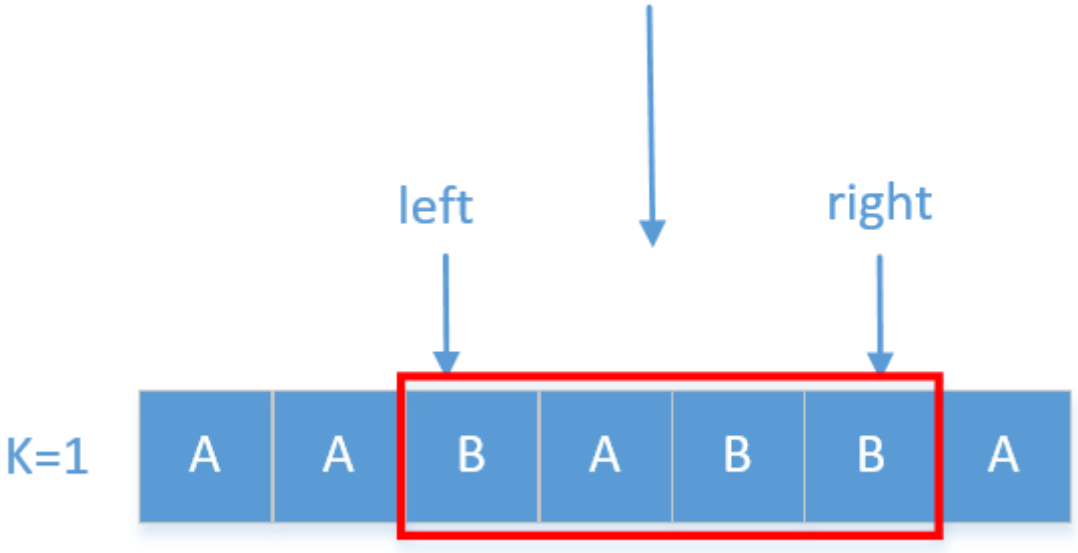
第5步窗口大小是5，最多的字母是A，个数是3，3+1<5，也就是窗口内的B不能全部替换成A，不满足条件，所以要缩小窗口的大小



第6步，虽然我们把窗口缩小了，但在这个窗口内实际上还是不满足的。我们不需要再继续缩小，因为这里right-left+1就是我们目前遇到的最长的可以替换的字符串，下一步继续扩大窗口



第7步窗口大小是5，最多的字母是B，个数是3，3+1<5，也就是窗口内的A不能全部替换成B，不满足条件，所以要缩小窗口的大小



……后面就不再画了

搞懂了上面的过程，我们再来看下代码

```
public int characterReplacement(String s, int k) {
    //字符串的长度
    int length = s.length();
    //用来存放对应字母的个数，比如字母A的个数是map[0],
    //字母B的个数是map[1]……
    int[] map = new int[26];
    int left = 0;//窗口左边的位置
    //窗口内曾经出现过相同字母最多的数量
    int maxSameCount = 0;
    int right = 0;//窗口右边的位置
    //满足条件的最大窗口，也就是可以替换的最长子串的长度
    int maxWindow = 0;
```

```
    //窗口的左边先不动，移动右边的位置
    for (; right < length; right++) {
      //统计窗口内曾经出现过相同字母最多的数量
      maxSameCount = Math.max(maxSameCount, ++map[s.charAt(right) - 'A']);
      //如果相同字母最多的数量加上k还小于窗口的大小，说明其他的字母不能全部替换为
      //最多的那个字母，我们要缩小窗口的大小，顺便减去窗口左边那个字母的数量，
      //因为他被移除窗口了，所以数量要减去
      if (k + maxSameCount < right - left + 1) {
        map[s.charAt(left) - 'A']--;
        left++;
      } else {//满足条件，要记录下最大的窗口，
        maxWindow = Math.max(maxWindow, right - left + 1);
      }
    }
    return maxWindow;
  }
```

总结

滑动窗口问题，如果窗口不满足的时候我们要缩小左边界，但没必要一直缩小到满足为止，这是因为最终求得的结果不会小于Math.min(k+maxSamCount,s.length);仔细想。

往期推荐

- 497，双指针验证回文串
- 490，动态规划和双指针解买卖股票的最佳时机
- 447，双指针解旋转链表
- 398，双指针求无重复字符的最长子串

497，双指针验证回文串

原创 山大王wld 数据结构和算法 1周前

收录于话题
#算法图文分析

111个 >



Books are the bees which carry the quickening pollen from one to another mind.

书籍是蜜蜂，将花粉从一个头脑传到另一个头脑。

问题描述

给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。

说明： 本题中，我们将空字符串定义为有效的回文串。

示例 1:

> **输入**: "A man, a plan, a canal: Panama"
> **输出**: true

示例 2:

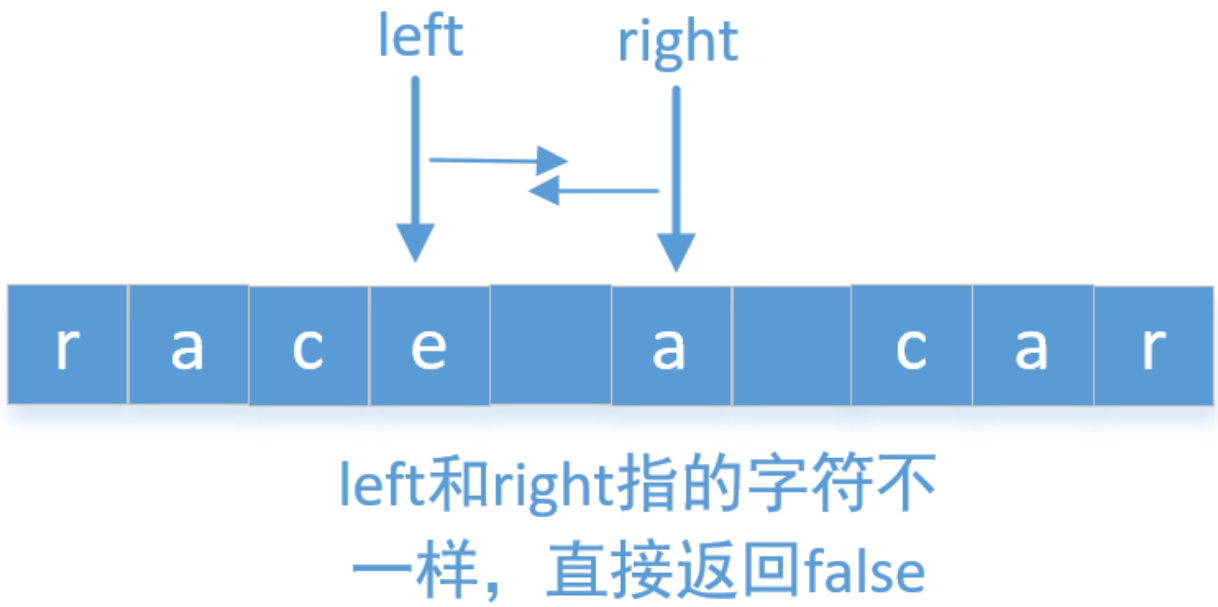
> **输入**: "race a car"
> **输出**: false

双指针解决

“回文串”是一个正读和反读都一样的字符串，也就是说他是左右两边对称的。验证一个字符串是否是回文串，最简单的一种方式就是使用两个指针，一个从前开始，一个从后开始，两个指针**同时**往中间走，如果他们指向的字符不一样，那么这个字符串肯定不是回文字符串，直接返回false即可，如果这两个指针相遇了，直接返回true。

但这题只需要判断字母和数字，因为字符串中可能含有其他字符，我们只需要跳过即可，画个图来看下





最后再来看下代码

```
public boolean isPalindrome(String s) {
  int left = 0, right = s.length() - 1;
  while (left < right) {
    //left是左指针，如果不是字母和数字要过滤掉
    while (left < right && !Character.isLetterOrDigit(s.charAt(left)))
      left++;
    //right也一样，如果不是字母和数字也要过滤掉
    while (left < right && !Character.isLetterOrDigit(s.charAt(right)))
      right--;
    //然后判断这两个字符是否相同，如果不相同直接返回false，这里是先把字符全部转化为小写
```

```
        if (Character.toLowerCase(s.charAt(left)) != Character.toLowerCase(s.charAt(right)))
            return false;
        //如果left和right指向的字符忽略大小写相等的话，这两个指针要分别往中间移一步
        left++;
        right--;
    }
    //如果都比较完了，说明是回文串，返回true
    return true;
}
```

我们还可以在比较之前字母全部转化为小写，这里改为for循环的方式，只不过是换汤不换药，原理还都是一样的，来看一下

```
public boolean isPalindrome(String s) {
    //先转为小写
    s = s.toLowerCase();
    for (int i = 0, j = s.length() - 1; i < j; i++, j--) {
        while (i < j && !Character.isLetterOrDigit(s.charAt(i)))
            i++;
        while (i < j && !Character.isLetterOrDigit(s.charAt(j)))
            j--;
        if (s.charAt(i) != s.charAt(j))
            return false;
    }
    return true;
}
```

递归方式解决

上面代码还可以写成递归的方式，无论怎么变，核心思路还是没变，可以参考一下，代码如下

```
public boolean isPalindrome(String s) {
    return isPalindromeHelper(s, 0, s.length() - 1);
}

public boolean isPalindromeHelper(String s, int left, int right) {
    if (left >= right)
        return true;
    while (left < right && !Character.isLetterOrDigit(s.charAt(left)))
        left++;
    while (left < right && !Character.isLetterOrDigit(s.charAt(right)))
        right--;
    return Character.toLowerCase(s.charAt(left)) == Character.toLowerCase(s.charAt(right))
        && isPalindromeHelper(s, ++left, --right);
}
```

总结

回文字符串的判断，和冒泡排序一样算是比较简单的一道算法题，基本上没什么难度。

往期推荐

- 493，动态规划解打家劫舍 III

466. 使用快慢指针把有序链表转换二叉搜索树

原创 山大王wld 数据结构和算法 10月21日

收录于话题
#算法图文分析

95个 >



Tomorrow is always fresh, with no mistakes in it.

明天始终崭新，无错可言。

问题描述

给定一个单链表，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点的左右两个子树的高度差的绝对值不超过 1。

示例:

```
给定的有序链表：[-10, -3, 0, 5, 9],

一个可能的答案是：[0, -3, 9, -10, null, 5],
它可以表示下面这个高度平衡二叉搜索树：

      0
     / \
   -3   9
```

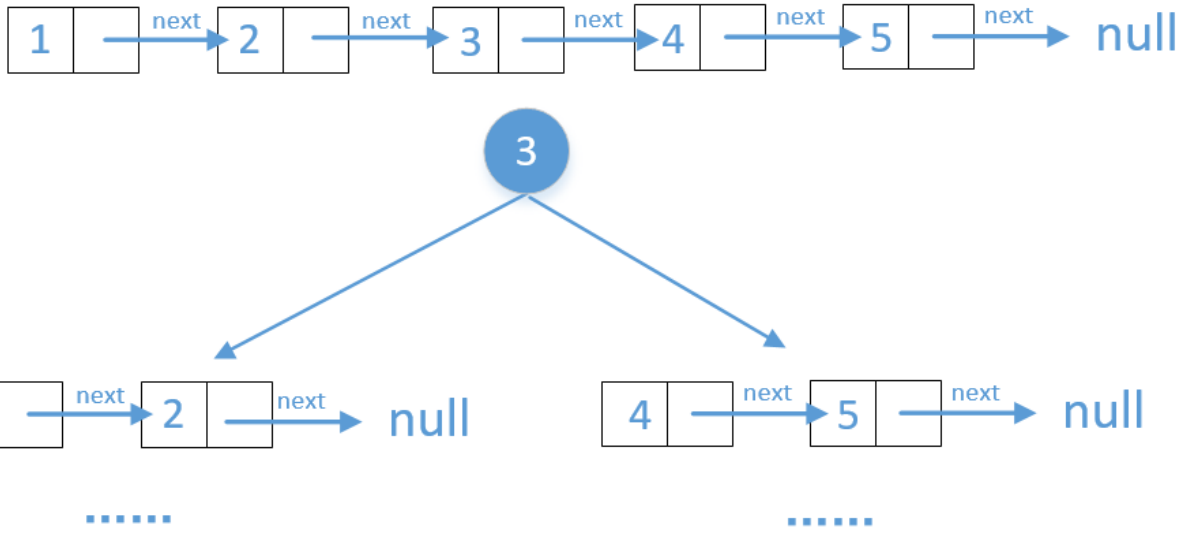
```
  /   /
-10  5
```

快慢指针解决

二叉搜索树的特点是当前节点大于左子树的所有节点，并且小于右子树的所有节点，并且每个节点都具有这个特性。

题中说了，是按照升序排列的单链表，我们只需要找到链表的中间节点，让他成为树的根节点，中间节点前面的就是根节点左子树的所有节点，中间节点后面的就是根节点右子树的所有节点，然后使用递归的方式再分别对左右子树进行相同的操作……

这里就以链表1→2→3→4→5为例来画个图看一下



我们看到上面链表的中间节点3就是二叉搜索树的根节点，然后再对左右子节点以同样的方式进行操作……，最后再来看下代码

```java
public TreeNode sortedListToBST(ListNode head) {
  //边界条件的判断
  if (head == null)
    return null;
  if (head.next == null)
    return new TreeNode(head.val);
  //这里通过快慢指针找到链表的中间结点slow，pre就是中间
  //结点slow的前一个结点
  ListNode slow = head, fast = head, pre = null;
  while (fast != null && fast.next != null) {
    pre = slow;
    slow = slow.next;
    fast = fast.next.next;
  }
  //链表断开为两部分，一部分是node的左子节点，一部分是node
  //的右子节点
  pre.next = null;
  //node就是当前节点
  TreeNode node = new TreeNode(slow.val);
  //从head节点到pre节点是node左子树的节点
  node.left = sortedListToBST(head);
  //从slow.next到链表的末尾是node的右子树的结点
```

```
    node.right = sortedListToBST(slow.next);
    return node;
  }
```

通过集合list解决

实际上还可以把链表中的值全都存储到集合list中，每次把list分为两部分，和上面原理一样

```
public TreeNode sortedListToBST(ListNode head) {
    List<Integer> list = new ArrayList<>();
    //把链表节点值全部提取到list中
    while (head != null) {
        list.add(head.val);
        head = head.next;
    }
    return sortedListToBSTHelper(list, 0, list.size() - 1);

}

TreeNode sortedListToBSTHelper(List<Integer> list, int left, int right) {
    if (left > right)
        return null;
    //把list中数据分为两部分
    int mid = left + (right - left) / 2;
    TreeNode root = new TreeNode(list.get(mid));
    root.left = sortedListToBSTHelper(list, left, mid - 1);
    root.right = sortedListToBSTHelper(list, mid + 1, right);
    return root;
}
```

总结

做这道题我们首先要明白什么是二叉搜索树，这题是让把升序的链表转化为二叉搜索树，如果是把升序的数组转化为二叉搜索树可能就更容易些了，但不管是什么数据结构，最终实现原理还是一样的。

往期推荐

- 462. 找出两个链表的第一个公共节点
- 461. 两两交换链表中的节点
- 457，二叉搜索树的最近公共祖先
- 453，DFS和BFS解求根到叶子节点数字之和

449，快慢指针解决环形链表

原创 山大王wld 数据结构和算法 9月10日

收录于话题
#算法图文分析

95个 >



Above all, don't lose hope.

总之，不要失去希望。

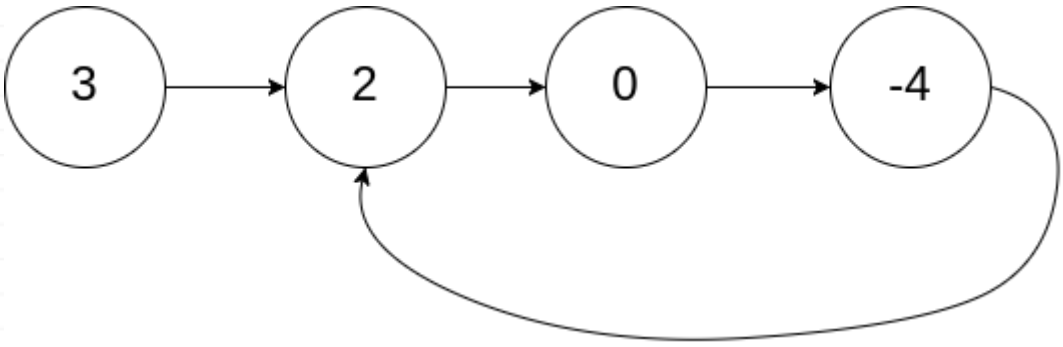
问题描述

给定一个链表，判断链表中是否有环。

为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 pos 是 -1，则在该链表中没有环。

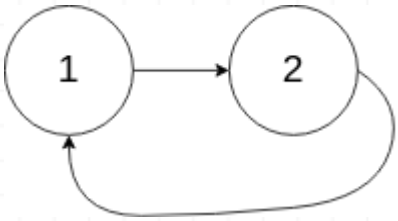
示例 1：

> **输入**：head = [3,2,0,-4], pos = 1
> **输出**：true
> **解释**：链表中有一个环，其尾部连接到第二个节点。



示例 2：

> **输入**：head = [1,2], pos = 0
> **输出**：true
> **解释**：链表中有一个环，其尾部连接到第一个节点。



示例 3：

> **输入**：head = [1], pos = -1
> **输出**：false
> **解释**：链表中没有环。



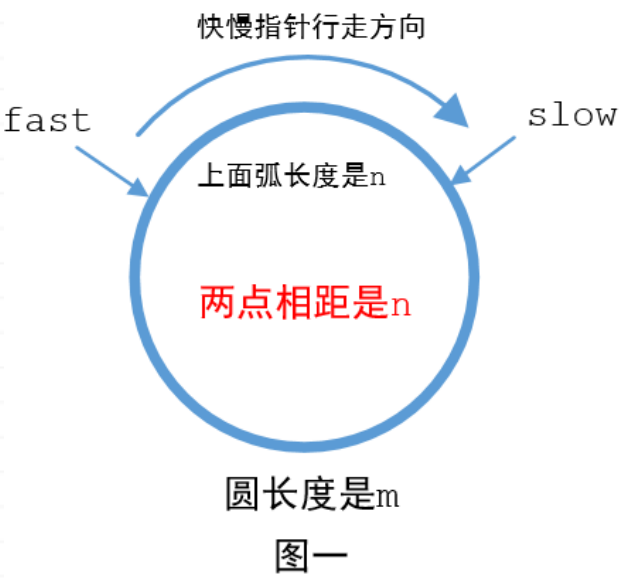
快慢指针解决

判断链表是否有环应该是老生常谈的一个话题了，最简单的一种方式就是快慢指针，慢指针针每次走一步，快指针每次走两步，如果相遇就说明有环，如果有一个为空说明没有环。代码比较简单

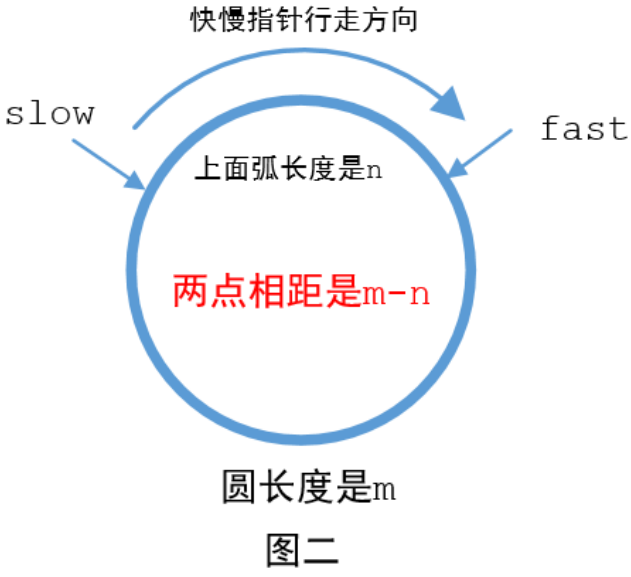
```
1   public boolean hasCycle(ListNode head) {
2     if (head == null)
3       return false;
4     //快慢两个指针
5     ListNode slow = head;
6     ListNode fast = head;
7     while (fast != null && fast.next != null) {
8       //慢指针每次走一步
9       slow = slow.next;
10      //快指针每次走两步
```

```
        fast = fast.next.next;
        //如果相遇，说明有环，直接返回true
        if (slow == fast)
            return true;
    }
    //否则就是没环
    return false;
}
```

到这里问题好像并没有结束，为什么快慢指针就一定能判断是否有环。我们可以这样来思考一下，假如有环，那么快慢指针最终都会走到环上，假如环的长度是m，快慢指针最近的间距是n，如下图中所示



图一



图二

快指针每次走两步，慢指针每次走一步，所以每走一次快慢指针的间距就要缩小一步，在图一中当走n次的时候就会相遇，在图二中当走m-n次的时候就会相遇。

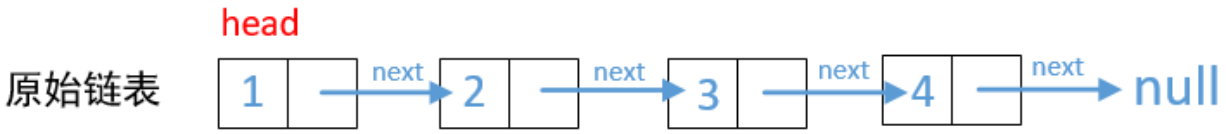
存放到集合中

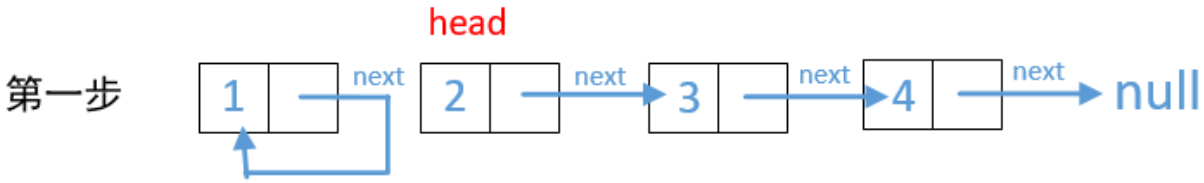
这题还可以把节点存放到集合set中，每次存放的时候判断当前节点是否存在，如果存在，说明有环，直接返回true，比较容易理解

```
public boolean hasCycle(ListNode head) {
    Set<ListNode> set = new HashSet<>();
    while (head != null) {
        //如果重复出现说明有环
        if (set.contains(head))
            return true;
        //否则就把当前节点加入到集合中
        set.add(head);
        head = head.next;
    }
    return false;
}
```

逐个删除

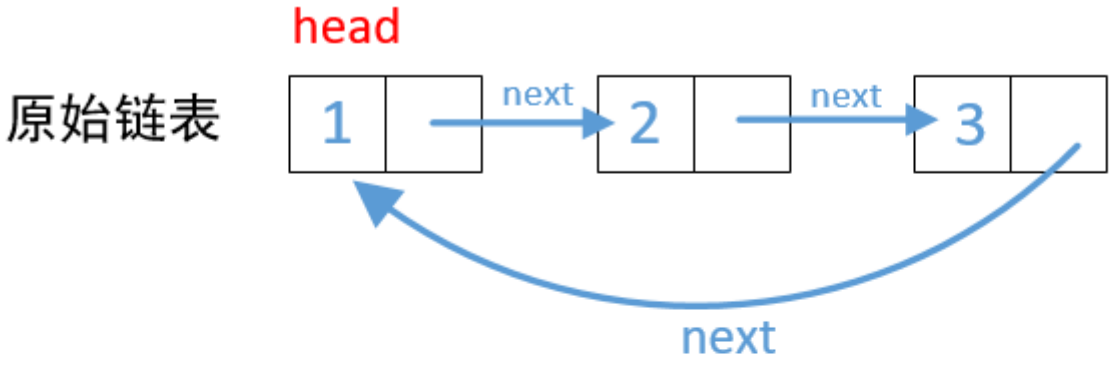
一个链表从头节点开始一个个删除，所谓删除就是让他的next指针指向他自己。如果没有环，从头结点一个个删除，最后肯定会删完，如下图所示

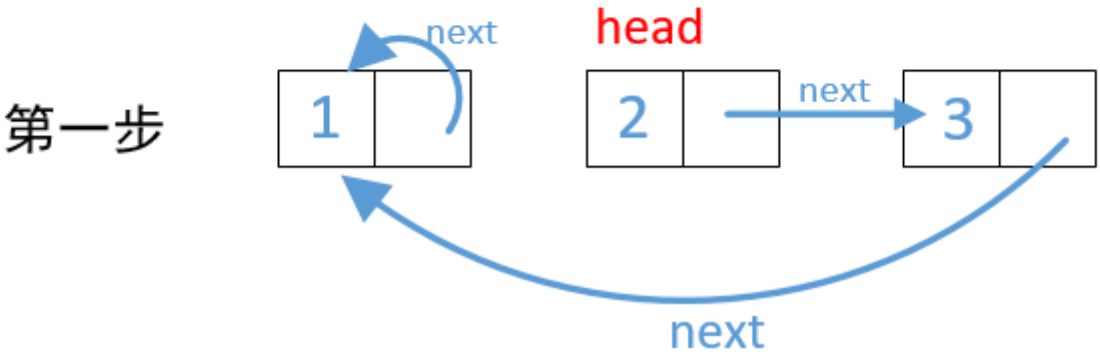


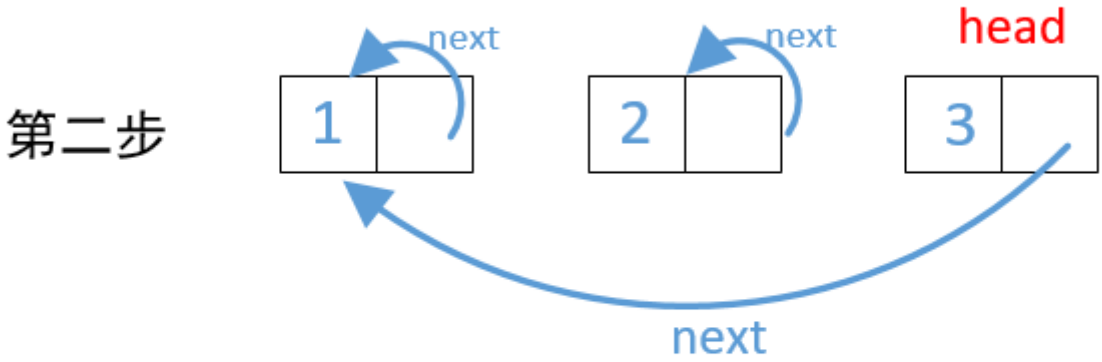


.....

如果是环形的，那么有两种情况，一种是o型的，一种是6型的。原理都是一样，我们就看一下o型的









如上图所示，如果删到最后，肯定会出现head=head.next;

```
public boolean hasCycle(ListNode head) {
  //如果head为空，或者他的next指向为空，直接返回false
  if (head == null || head.next == null)
    return false;
  //如果出现head.next = head表示有环
  if (head.next == head)
    return true;
  ListNode nextNode = head.next;
  //当前节点的next指向他自己，相当于把它删除了
  head.next = head;
  //然后递归，查看下一个节点
```

```
12      return hasCycle(nextNode);
13  }
```

总结

这题是很常见的一道题了。来思考这样一个问题，这里的快慢指针是快指针每次走2步，慢指针每次走1步。如果慢指针还是每次走1步，快指针每次走3步能不能判断。或者快指针每次走m步，慢指针每次都n步，并且m≠n，这种情况下能不能判断？

往期推荐

- 447，双指针解旋转链表
- 432，剑指 Offer-反转链表的3种方式
- 431，剑指 Offer-链表中倒数第k个节点
- 352，数据结构-2,链表

447，双指针解旋转链表

原创 山大王wld 数据结构和算法 9月7日

收录于话题
#算法图文分析

95个 >



I figure life's a gift, and I don't intend on wasting it.

我觉得人生就是份礼物，而我不愿白白浪费。

问题描述

给定一个链表，旋转链表，将链表每个节点向右移动 *k* 个位置，其中 *k* 是非负数。

示例 1:

> **输入**: 1->2->3->4->5->NULL, k = 2
> **输出**: 4->5->1->2->3->NULL
> **解释**:
> 向右旋转 1 步: 5->1->2->3->4->NULL
> 向右旋转 2 步: 4->5->1->2->3->NULL

示例 2:

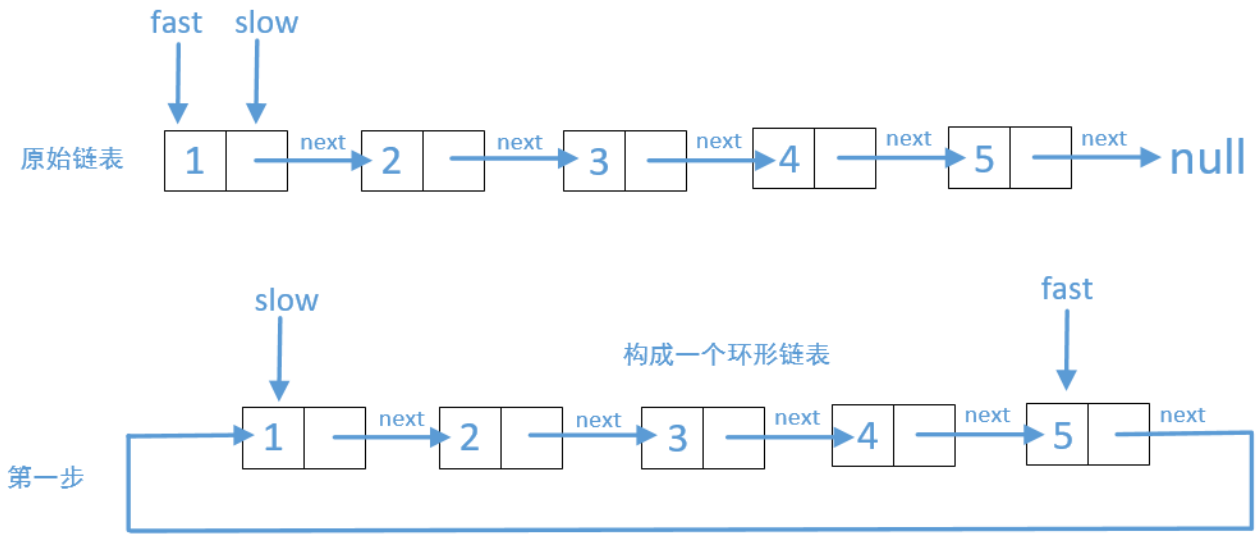
> **输入**: 0->1->2->NULL, k = 4
> **输出**: 2->0->1->NULL
> **解释**:
> 向右旋转 1 步: 2->0->1->NULL

向右旋转 2 步: 1->2->0->NULL
向右旋转 3 步: 0->1->2->NULL
向右旋转 4 步: 2->0->1->NULL

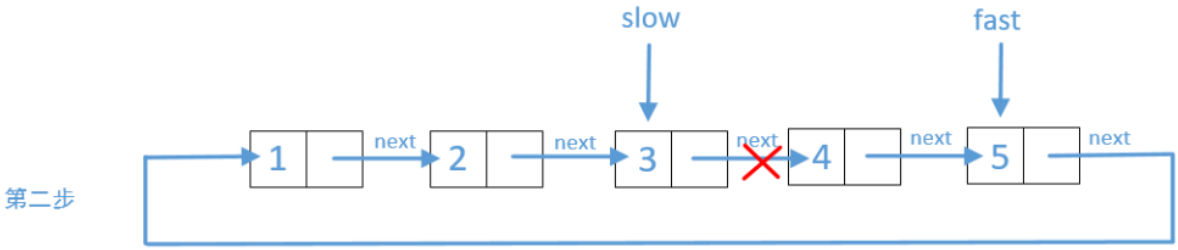
双指针解决

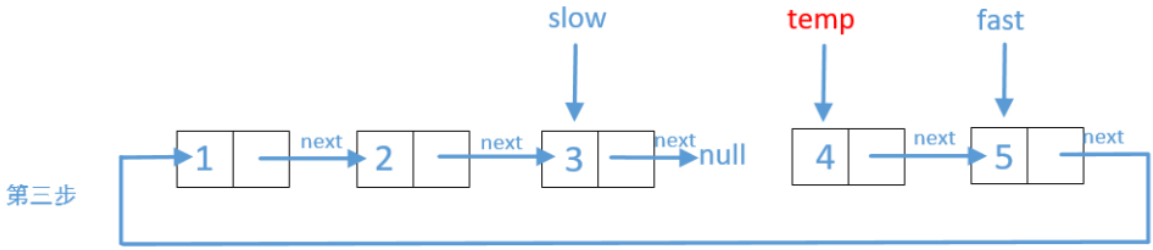
这题k是非负数，但k有可能比链表的长度还要大，所以先要计算链表的长度len，需要旋转的步数就是（k%len）。一种比较简单的方式就是先把链表连接成一个环，然后再把链表在某个合适的位置断开。

我们可以使用两个指针，一个快指针fast从头开始遍历直到走到链表的末尾，然后再把链表串成一个环形。还一个指针slow也是从头开始，走（len-k%len）步就是我们要返回的链表头，这里可能有点疑问，为什么不是走（k%len）步，这是因为我们需要把链表后面的（k%len）个移到前面，因为单向链表我们没法从后往前遍历，所以我们只能从前往后移动（len-k%len）步。但实际上操作的时候会少走一步，具体来举个例子看一下，这里就以示例1为例画个图来看一下



链表的长度是5，当k=2的时候，链表应该变为4→5→1→2→3，所以下面图中slow应该移动5-2=3步，也就是移动到节点值为4的位置，但因为链表变成了环形，我们还需要把结点4的前一个结点和节点4的连接断开，但我们不知道节点4的前一个结点是多少，如果再全部遍历一遍有点麻烦。所以这里slow移动的步数应该是5-2-1=2，也就是少移一步，指向下面的结点3，然后保存3的下一个结点4，也就是要返回的结点，接着把结点3和节点4断开，最后直接返回节点4为头的链表即可





断开链表，返回节点为4的头结点即可

原理比较简单，来直接看下代码

```java
public ListNode rotateRight(ListNode head, int k) {
    if (head == null)
        return head;
    ListNode fast = head, slow = head;
    //链表的长度
    int len = 1;
    //统计链表的长度，顺便找到链表的尾结点
    while (fast.next != null) {
        len++;
        fast = fast.next;
    }
    //首尾相连，先构成环
    fast.next = head;
    //慢指针移动的步数
    int step = len - k % len;
    //移动步数，这里大于1实际上是少移了一步
    while (step-- > 1) {
        slow = slow.next;
    }
    //temp就是需要返回的结点
    ListNode temp = slow.next;
    //因为链表是环形的，slow就相当于尾结点了，
    //直接让他的next等于空
    slow.next = null;
    return temp;
}
```

总结

这道题使用快慢指针解决，快指针主要是获取链表的尾结点然后把链表串起来，慢指针要找到需要把链表截断的位置。

往期推荐

- 432，剑指 Offer-反转链表的3种方式
- 431，剑指 Offer-链表中倒数第k个节点
- 429，剑指 Offer-删除链表的节点
- 352，数据结构-2,链表

398，双指针求无重复字符的最长子串

原创 山大王wld 数据结构和算法 7月9日

收录于话题
#算法图文分析

96个 >



You should never judge something you don't understand.

你不应该去评判你不了解的事物。

问题描述

给定一个字符串，请你找出其中不含有重复字符的最长子串的长度。

示例 1:

```
输入: "abcabcbb"
输出: 3
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。
```

示例 2:

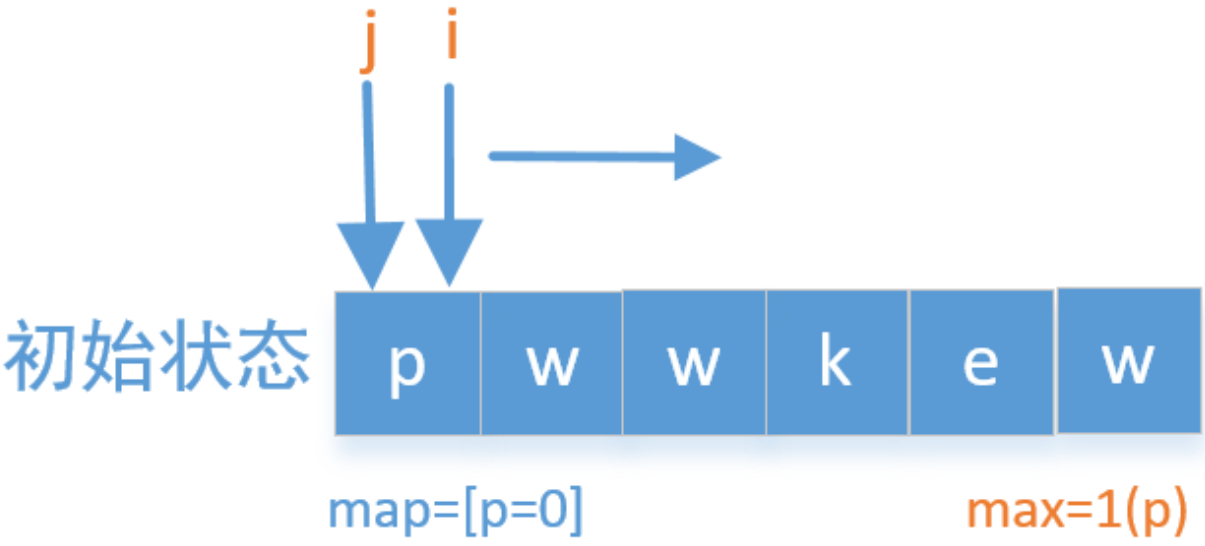
```
输入: "bbbbb"
输出: 1
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。
```

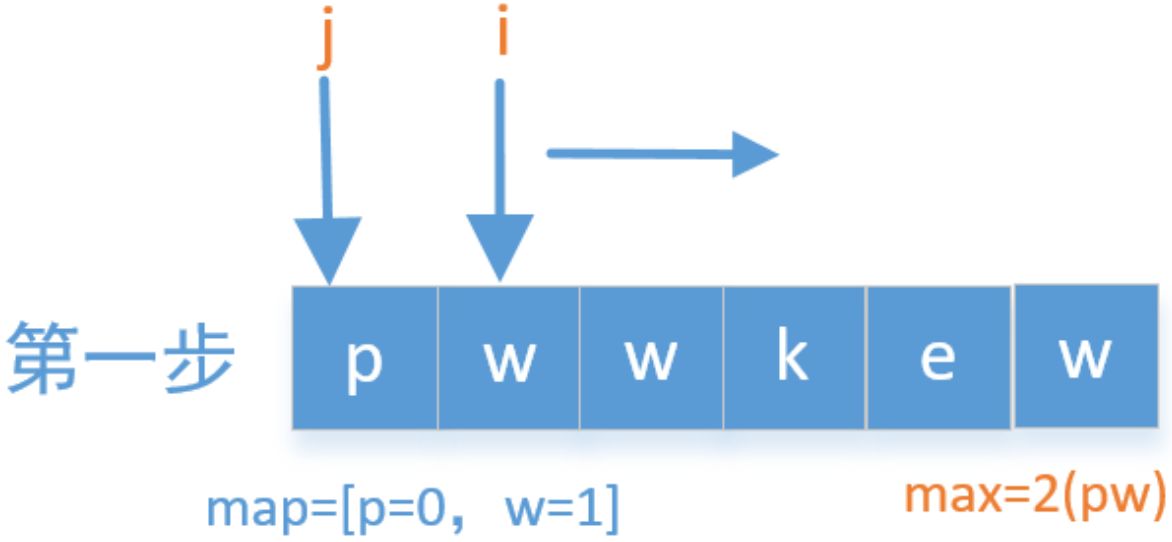
示例 3:

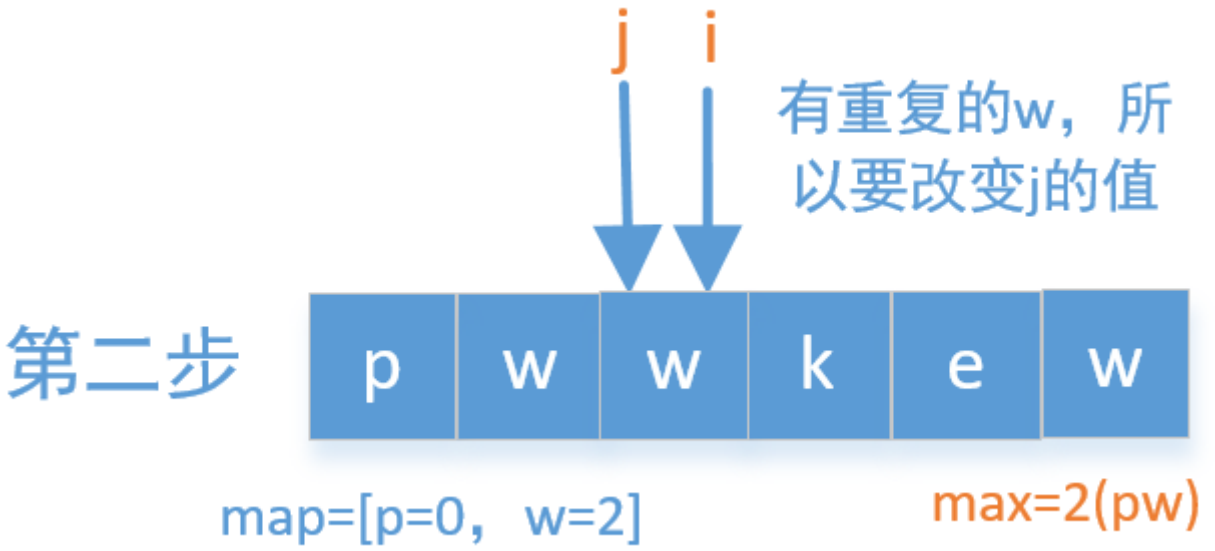
> 输入: "pwwkew"
> 输出: 3
> 解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
> 请注意，你的答案必须是子串的长度，"pwke" 是一个子序列不是子串。

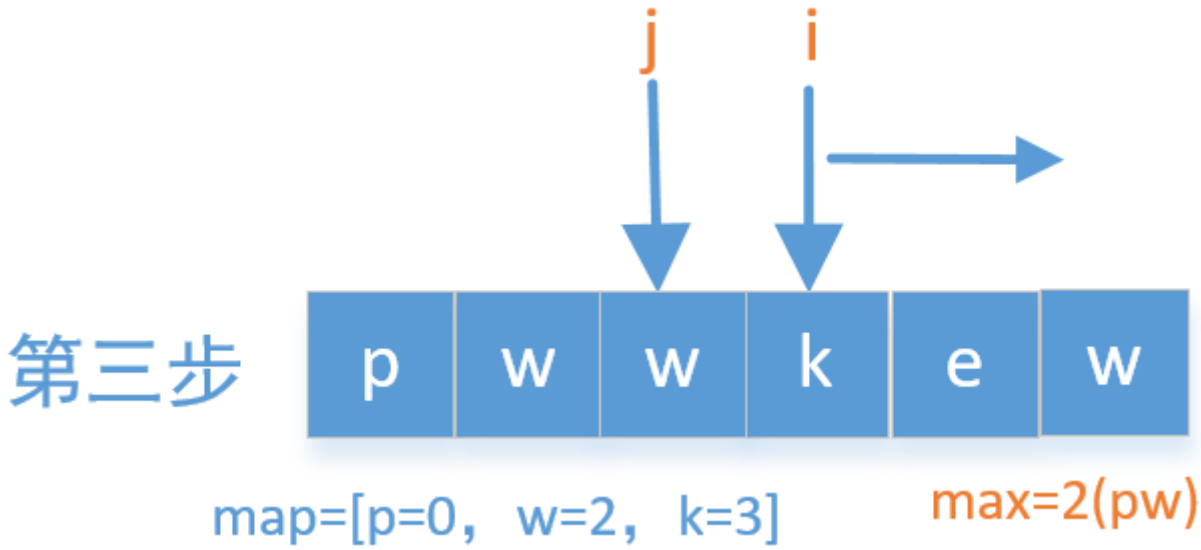
双指针求解

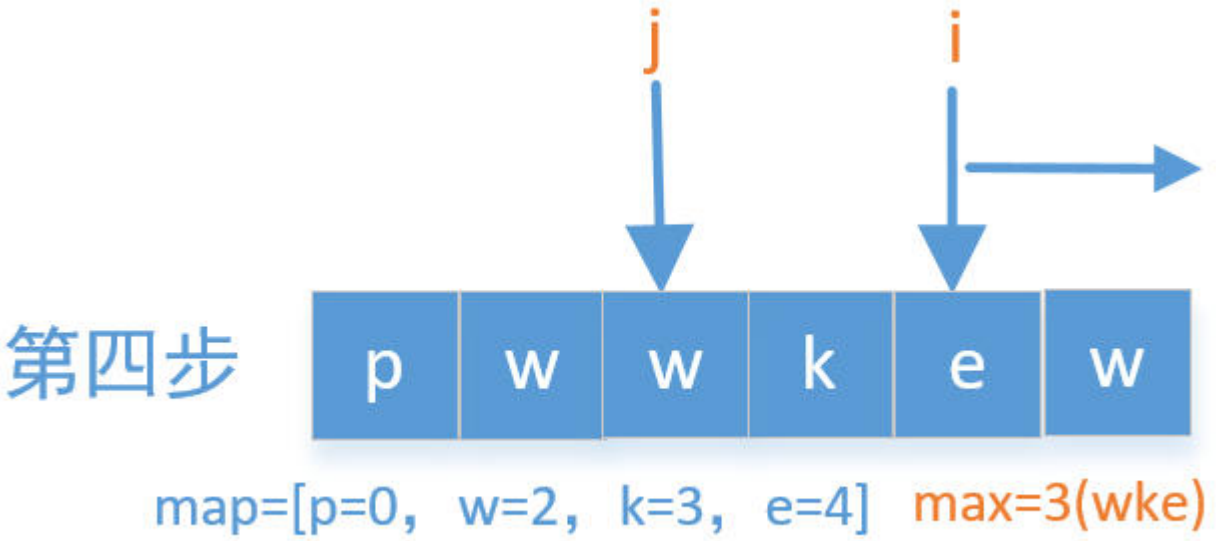
这题要求的是找出一个最长的字符串，并且这个字符串中没有重复的字符。最容易想到的就是双指针，最开始的时候两个指针i和j都指向第一个元素，然后i往后移，把扫描过的元素都放到map中，如果i扫描过的元素没有重复的，就顺便记录一下最大值max，如果i扫描过的元素有重复的，就改变j的位置，要保证j到i之间不能有重复的元素。我们就以pwwkew为例画个图看一下

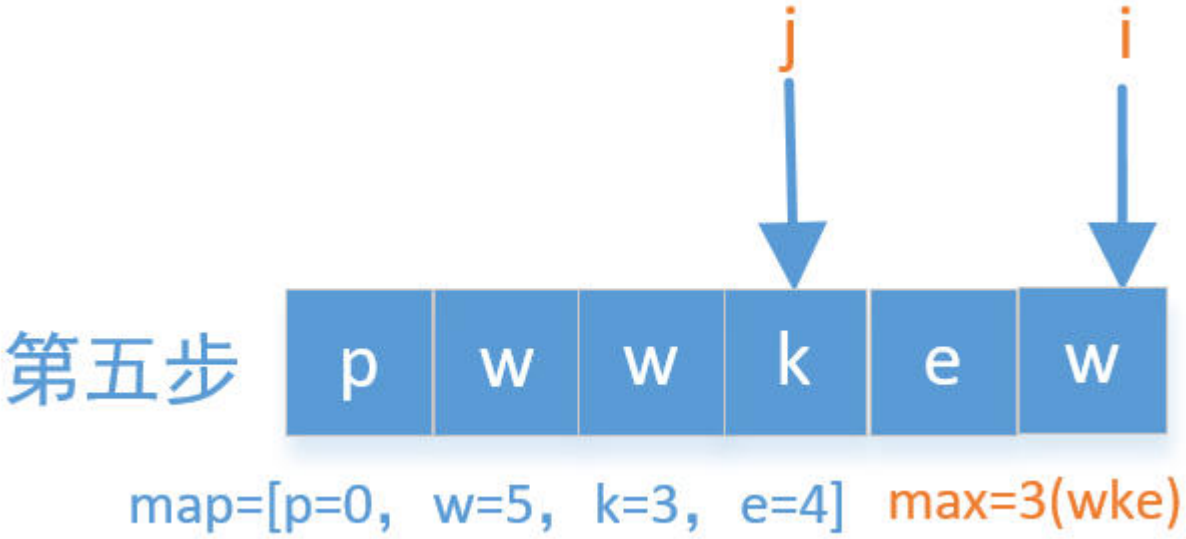





j
i
有重复的w，所以要改变j的值
第二步
p
w
w
k
e
w
map=[p=0，w=2]
max=2(pw)


j
i
第三步
p
w
w
k
e
w
map=[p=0，w=2，k=3]
max=2(pw)





我们使用一个map来存储扫描过的元素，其中i指针是一直往右移动的，如果i指向的元素在map中出现过，说明出现了重复的元素，要更新j的值。并且这个j的值只能增大不能减小，也就是说j只能往右移动，不能往左移动，所以下面代码中j取的是重复元素位置的下一个值和j这两个值的最大值，有点绕，我们直接看代码

```
if (map.containsKey(s.charAt(i))) {
    j = Math.max(j, map.get(s.charAt(i)) + 1);
}
```

再来看下完整代码

```
public int lengthOfLongestSubstring(String s) {
  if (s.length() == 0)
    return 0;
  HashMap<Character, Integer> map = new HashMap<>();
  int max = 0;
  for (int i = 0, j = 0; i < s.length(); ++i) {
    //如果有重复的，就修改j的值
    if (map.containsKey(s.charAt(i))) {
      j = Math.max(j, map.get(s.charAt(i)) + 1);
    }
```

```
            map.put(s.charAt(i), i);
            //记录查找的最大值
            max = Math.max(max, i - j + 1);
        }
        //返回最大值
        return max;
    }
```

使用队列求解

除了使用双指针以外，我们还可以使用队列来解决，这个原理也很简单。就是把元素不停的加入到队列中，如果有相同的元素，就把队首的元素移除，这样我们就可以保证队列中永远都没有重复的元素，每次计算的时候我们都要记录下最大长度，最后再返回即可。




```
public int lengthOfLongestSubstring(String s) {
    //用链表实现队列，队列是先进先出的
    Queue<Character> queue = new LinkedList<>();
    int max = 0;
    for (char c : s.toCharArray()) {
        while (queue.contains(c)) {
            //如果有重复的，队头出队，这里通过while循环，
            //如果还有重复的就继续出队，直到队列中没有
            // 重复的元素为止
            queue.poll();
        }
        //添加到队尾
        queue.add(c);
        //记录下最大长度
        max = Math.max(max, queue.size());
    }
    return max;
}
```

注意这里的while循环是不能改为if语句的，因为他移除的不一定都是重复的元素，他移除的是重复元素和他前面的元素。比如pweaw，当遇到第二个w的时候，因为出现了重复，所以我们要先把队首的元素p给移除，然后再把w给移除。

总结

这题没什么难度，使用双指针和队列都能很容易解决，代码量也不多，无论使用哪种方式，我们都要保证两个指针之间或者队列中不能出现重复的元素。

往期推荐

- 397，双指针求接雨水问题
- 396，双指针求盛最多水的容器
- 394，经典的八皇后问题和N皇后问题
- 373，数据结构-6,树

397，双指针求接雨水问题

原创 山大王wld 数据结构和算法 7月8日

收录于话题
#算法图文分析

96个 >



Work and acquire, and thou hast chained the wheel of chance.

边工作边探求，你便可拴住机会的车轮。

问题描述

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。



上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

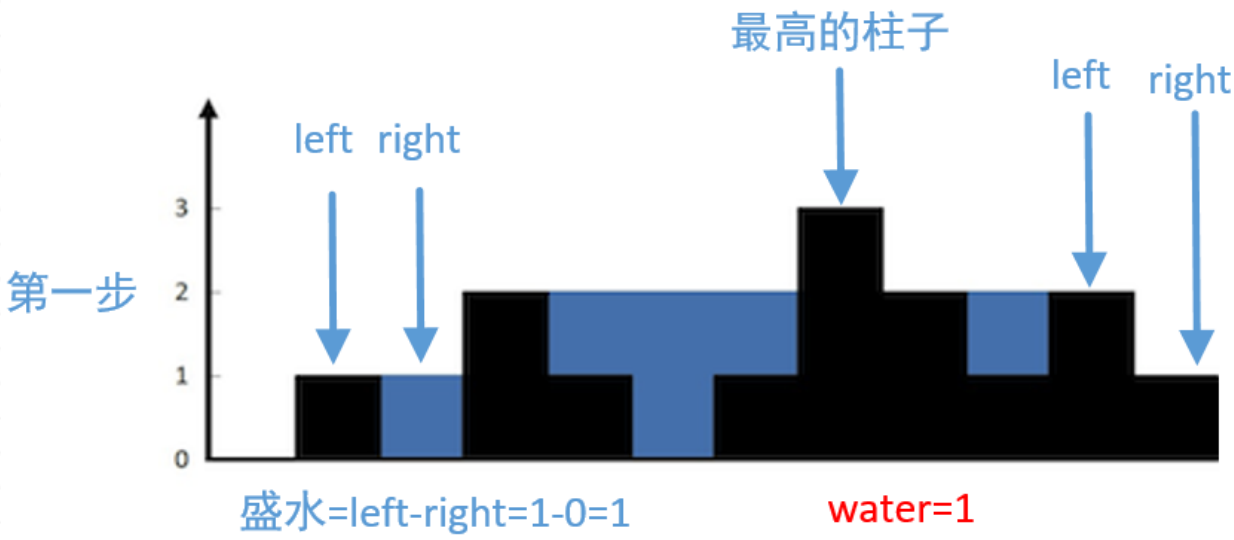
示例:

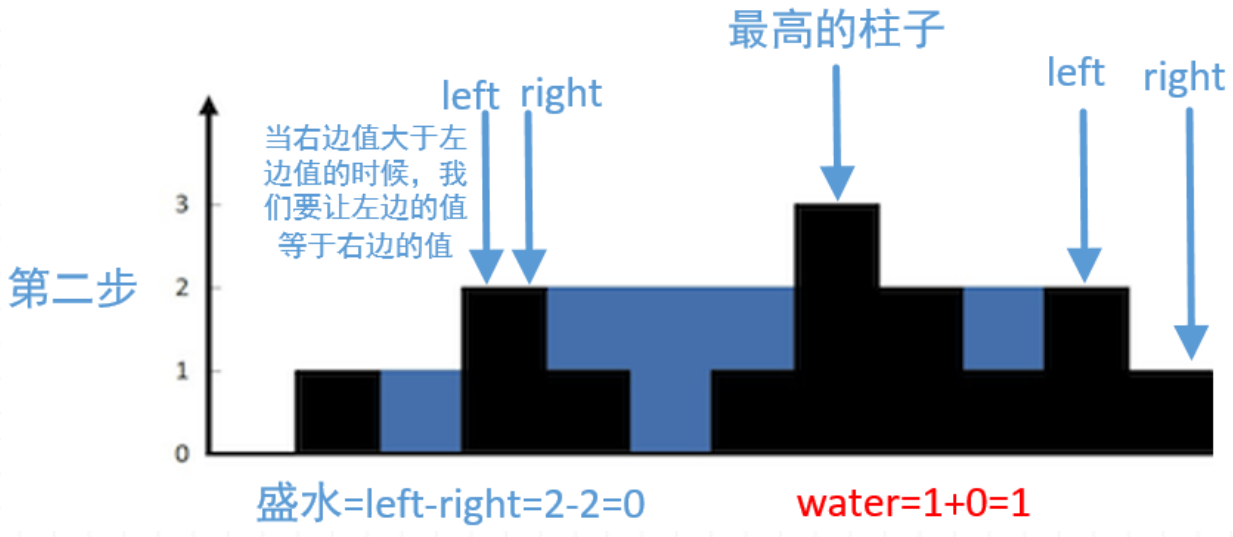
> 输入：[0,1,0,2,1,0,1,3,2,1,2,1]
> 输出：6

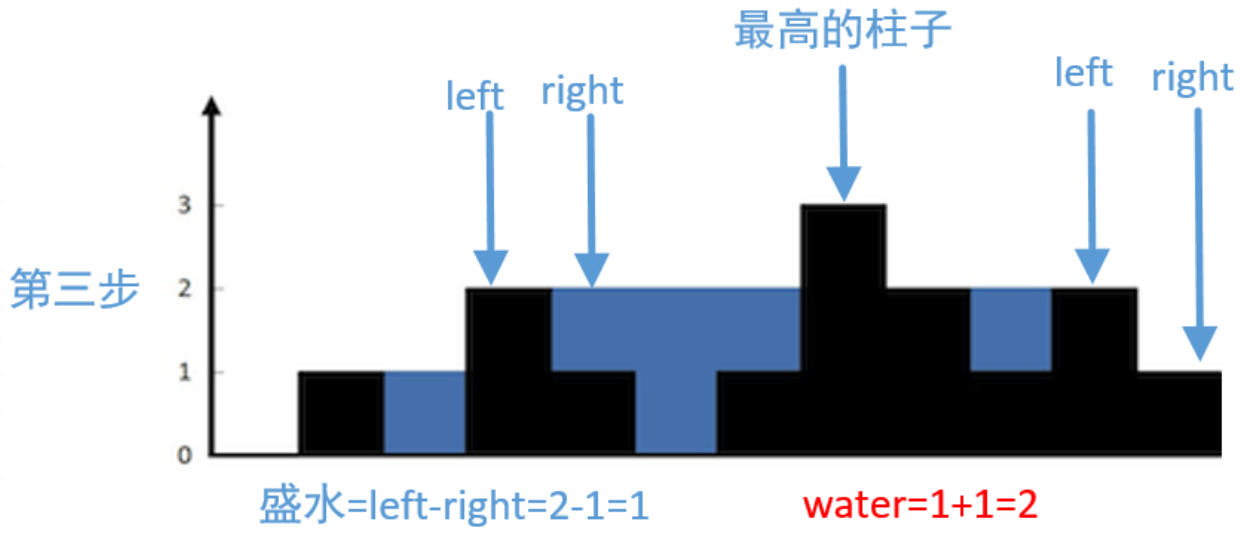
三指针求解

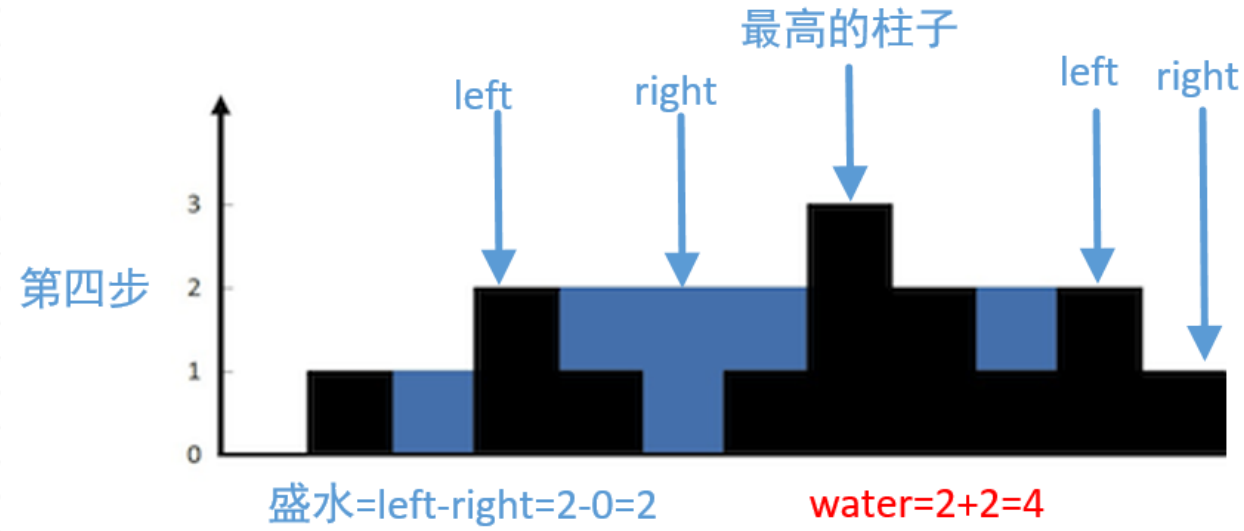
这题让求柱子中间能盛多少水，首先可以肯定两边的两个柱子是不能盛水的，只有两边之间的柱子有可能会盛水。最简单的一种方式就是使用3个指针，先找到最高的柱子，用一个指针top指向最高柱子，然后最高柱子左边用两个指针，一个left，一个right（这里的left和right指向柱子的高度）。

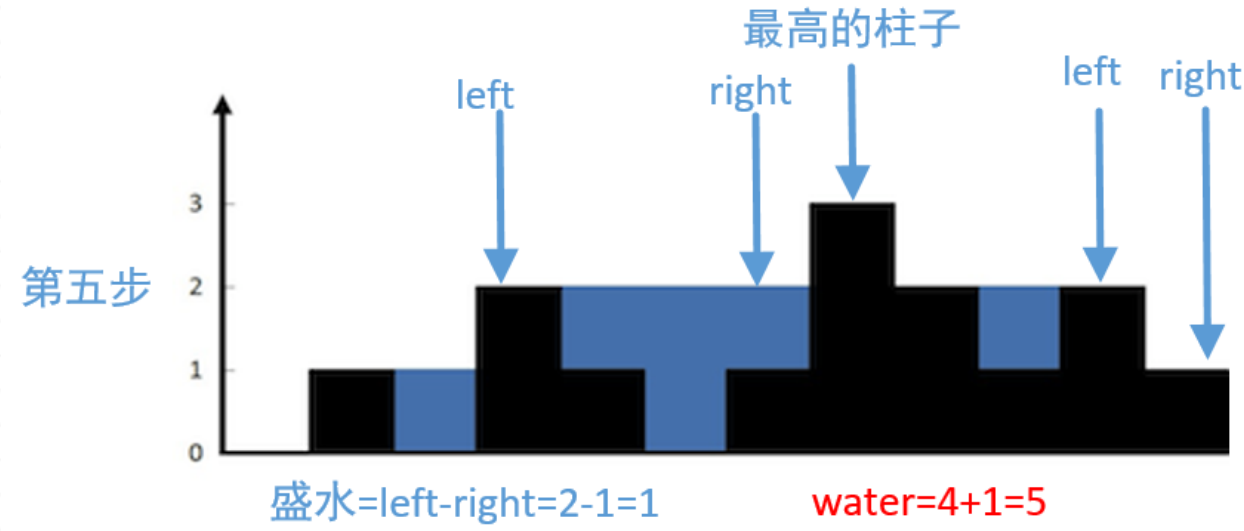
- 如果left大于right，那么肯定是能盛水的，因为left是小于等于最高柱子top的，并且right指向的柱子是在left和最高柱子top之间，根据木桶原理盛水量由最矮的柱子决定，所以盛水是left-right。
- 如果left不大于right，是不能盛水的，这时候我们要让left等于right。因为right是不能超过最高柱子的，我们增加left的高度，有利于后面计算的时候盛更多的水。











这里是计算最高柱子左边的盛水量，同理右边也一样，只不过右边计算的时候和左边有一点区别就是左边是left-right，右边是right-left

上面的代码如下

```
int left = height[0];//左边的柱子
int right = 0;//右边的柱子
```

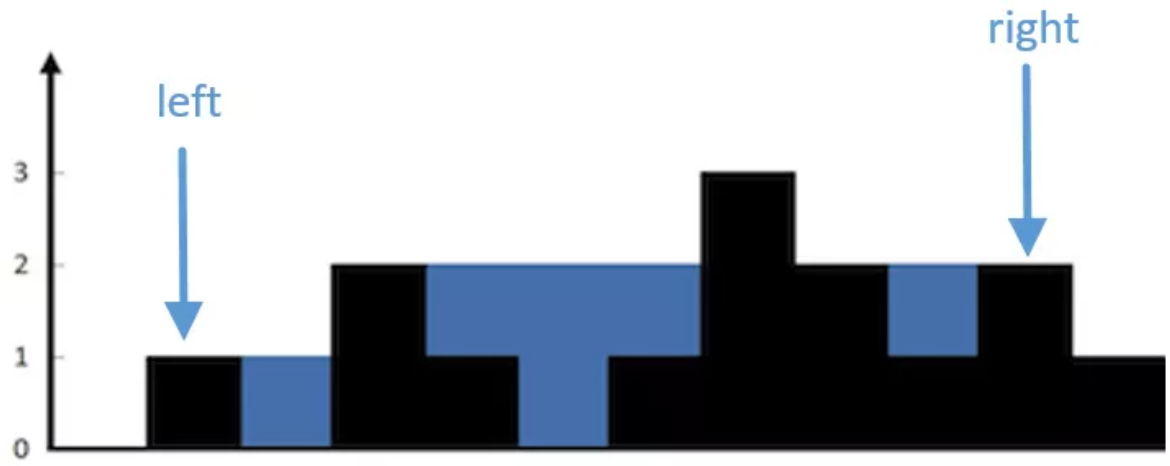
```
int water = 0;//盛水量
    for (int i = 1; i < 最高柱子的下标; i++) {
    right = height[i];
    //如果right大于left，我们要让更新left的值
    if (right > left) {
        left = right;
    } else {
        //否则我们计算盛水量
        water += left - right;
    }
}
```

这里我们只是计算了左边的盛水量，我们还需要计算右边的盛水量，完整代码如下

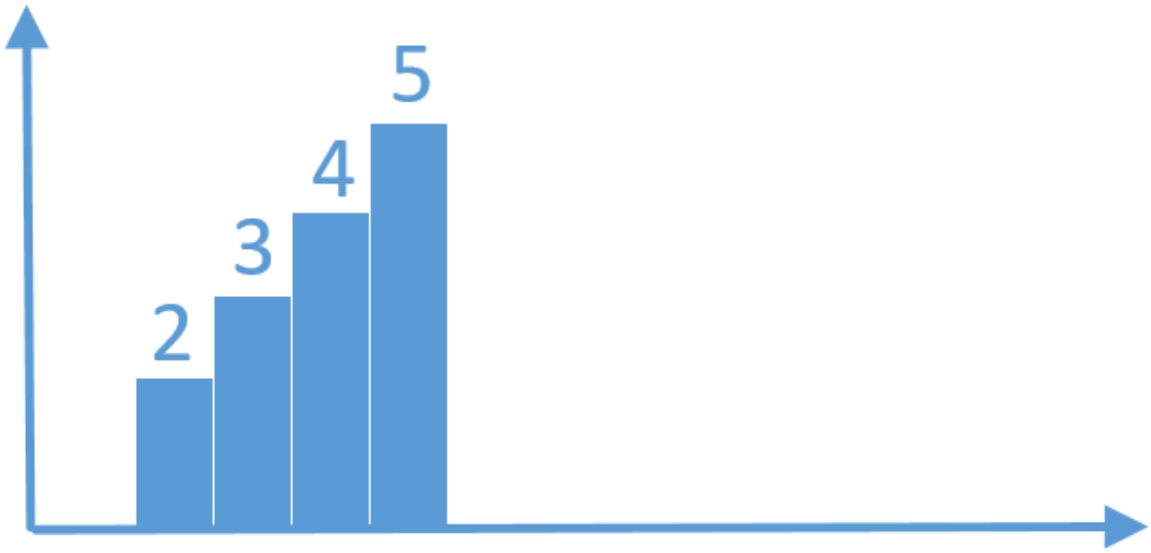
```
public int trap(int[] height) {
  if (height.length <= 2)
    return 0;
  //找到最高的柱子的下标
  int max = Integer.MIN_VALUE;
  int maxIndex = -1;
  for (int i = 0; i < height.length; i++) {
    if (height[i] > max) {
      max = height[i];
      maxIndex = i;
    }
  }

  //统计最高柱子左边能接的雨水数量
  int left = height[0];
  int right = 0;
  int water = 0;
  for (int i = 1; i < maxIndex; i++) {
    right = height[i];
    if (right > left) {
      left = right;
    } else {
      water += left - right;
    }
  }

  //统计最高柱子右边能接的雨水数量
  right = height[height.length - 1];
  for (int i = height.length - 2; i > maxIndex; i--) {
    left = height[i];
    if (height[i] > right) {
      right = left;
    } else {
      water += right - left;
    }
  }

  //返回盛水量
  return water;
}
```

双指针求解

这里我们还可以使用双指针，一个指向最左边，一个指向最右边，如下图所示。



这里要明白一点，最开始的时候如果左边柱子从左往右是递增的，那么这些柱子是不能盛水的，比如像下面这样



同理最开始的时候如果右边的柱子从右往左是递增的，也是不能盛水的。所以上面图中right指向的是右边第2根柱子。确定左右两边柱子的的代码如下

```
int left = 0, right = height.length - 1;
while (left < right && height[left] <= height[left + 1])
    left++;
while (left < right && height[right] <= height[right - 1])
    right--;
```

通过上面的计算，确定left和right的值之后，在left和right之间相当于构成了一个桶，桶的高度是最矮的那根柱子。然后我们从两边往中间逐个查找，如果查找的柱子高度小于桶的高度，那么盛水量就是桶的高度减去我们查找的柱子高度，如果查找的柱子大于桶的高度，我们要更新桶的高度。我们来看下最终代码

```
public int trap(int[] height) {
```

```
    if (height.length <= 2)
      return 0;
    int water = 0;
    int left = 0, right = height.length - 1;
    //最开始的时候确定left和right的边界，这里的left和right是
    //柱子的下标，不是柱子的高度
    while (left < right && height[left] <= height[left + 1])
      left++;
    while (left < right && height[right] <= height[right - 1])
      right--;

    while (left < right) {
      int leftValue = height[left];
      int rightValue = height[right];
      //在left和right两根柱子之间计算盛水量
      if (leftValue <= rightValue) {
        //如果左边柱子高度小于等于右边柱子的高度，根据木桶原理，
        // 桶的高度就是左边柱子的高度
        while (left < right && leftValue >= height[++left]) {
          water += leftValue - height[left];
        }
      } else {
        //如果左边柱子高度大于右边柱子的高度，根据木桶原理，
        // 桶的高度就是右边柱子的高度
        while (left < right && height[--right] <= rightValue) {
          water += rightValue - height[right];
        }
      }
    }
    return water;
  }
```

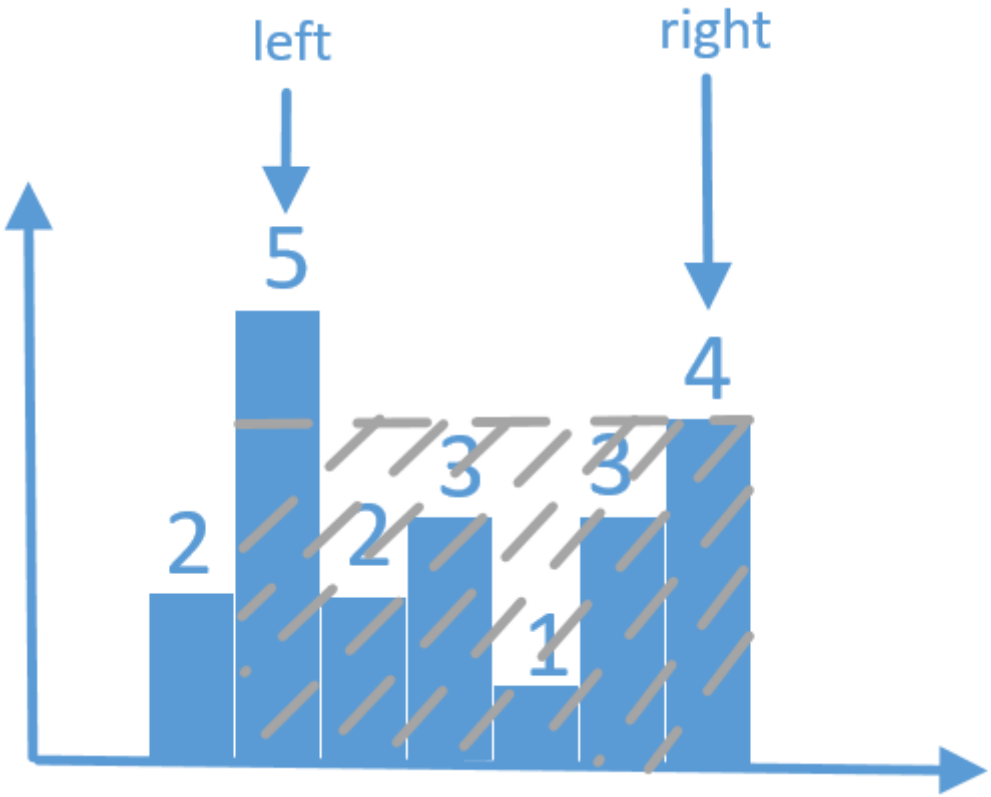
上面有3个while循环，看的有点眼花缭乱，实际上我们还可以把它合并为一个，代码如下

```
  public int trap(int[] height) {
    int left = 0;
    int right = height.length - 1;
    int water = 0;
    int leftmax = 0;
    int rightmax = 0;
    while (left < right) {
      //确定左边的最高柱子
      leftmax = Math.max(leftmax, height[left]);
      //确定左边的最高柱子
      rightmax = Math.max(rightmax, height[right]);
      //那么桶的高度就是leftmax和rightmax中最小的那个
      if (leftmax < rightmax) {
        //桶的高度是leftmax
        water += (leftmax - height[left++]);
      } else {
        //桶的高度是rightmax
        water += (rightmax - height[right--]);
      }
    }
    return water;
  }
```

双指针代码简化

实际上我们还可以再进一步简化，我们看下下面这个图。此时left和right围成的桶的高度是4，这个时候如果right往左移，那么移动之后这个值是小于4的，也就是小于桶的

高度，所以这个时候桶的高度是不变的。假如right往左移之后的值是大于4，比如5，那么桶的高度是要更新的。



我们只要确定桶的高度之后，那么盛水量就好求了。

```
1   public int trap(int[] height) {
2     int left = 0, right = height.length - 1, water = 0, bucketHeight = 0;
3     while (left < right) {
4       //取height[left]和height[right]的最小值
5       int minHeight = Math.min(height[left], height[right]);
6       //如果最小值minHeight大于桶的高度bucketHeight，要更新桶的高度到minHeight
7       bucketHeight = bucketHeight < minHeight ? minHeight : bucketHeight;
8       water += height[left] >= height[right] ? (bucketHeight - height[right--]) : (bucketHeight - he
9     }
10    return water;
11  }
```

总结

接雨水我们把它想象成两边的两根柱子围成一个桶，桶的高度就是最矮的那根柱子，只要确定了桶的高度，我们遍历中间柱子的时候就可以确定盛水量了。如果柱子的高度大于桶的高度，很明显是不能盛水的，只有柱子的高度小于桶的高度的时候才会盛水。这里有一点要注意的是当柱子的高度大于桶的高度的时候我们要更新桶的高度，当柱子的高度小于桶时候，桶的高度是不变的。这题使用双指针很巧妙的解决了上面的问题。

396，双指针求盛最多水的容器

原创 山大王wld 数据结构和算法 7月7日

收录于话题
#算法图文分析

96个 >



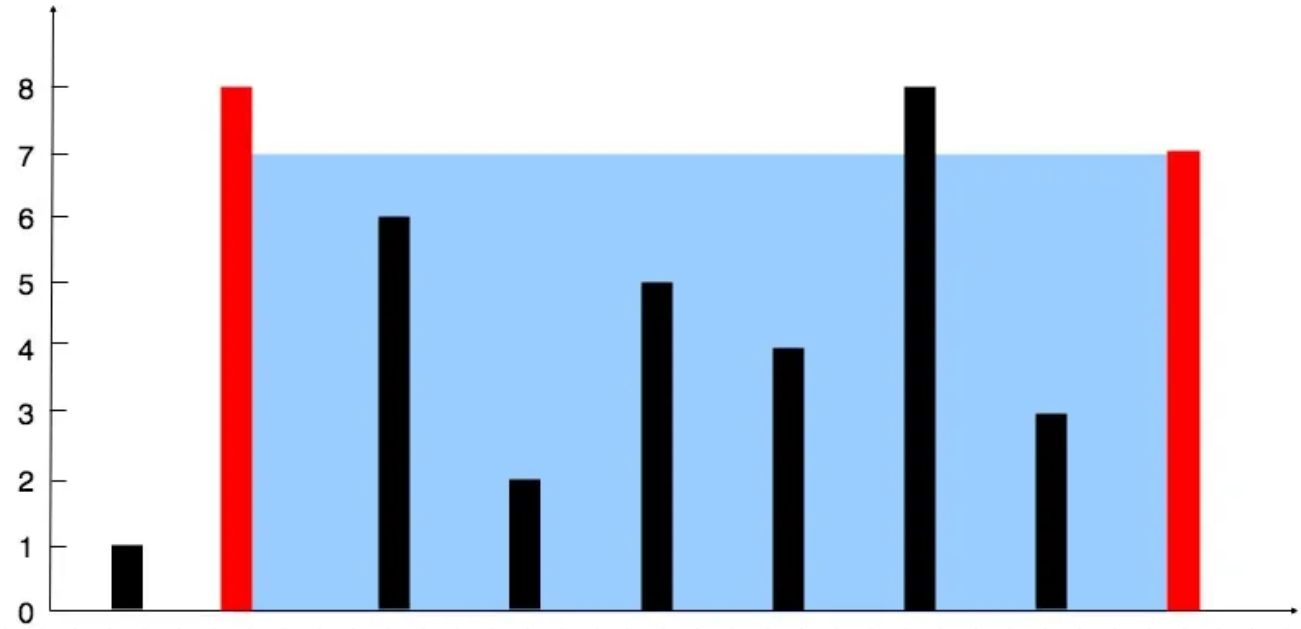
Whatever you do, you hold on to that foolishly hopeful smile.

不管你做什么，请留住你脸上那充满希望的傻笑。

问题描述

给你 n 个非负整数 a1，a2，...，an，每个数代表坐标中的一个点 (i, ai) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, ai) 和 (i, 0)。找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

说明： 你不能倾斜容器，且 n 的值至少为 2。



图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

示例：

> 输入：[1,8,6,2,5,4,8,3,7]
> 输出：49

暴力求解

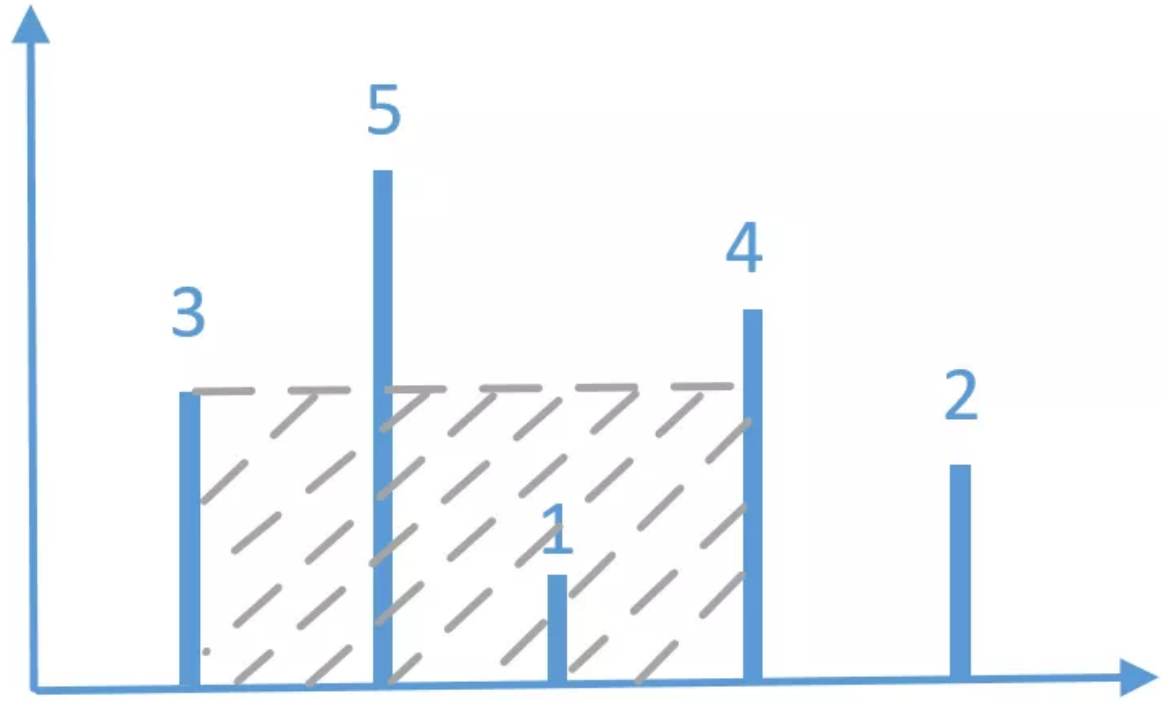
这种题最容易想到的是暴力求解，就是计算每两个柱子所围成的面积，把所有的都计算一遍，然后保留最大值即可。但暴力求解效率一般都不高，我们看看即可，代码如下

```
1   public int maxArea(int[] height) {
2     int maxarea = 0;
3     int area = 0;
4     int length = height.length;
5     for (int i = 0; i < length - 1; i++) {
6       for (int j = i + 1; j < length; j++) {
7         area = Math.min(height[i], height[j]) * (j - i);
8         maxarea = Math.max(maxarea, area);
9       }
10    }
11    return maxarea;
12  }
```

双指针求解

根据木桶原理，桶的容量是由最短的木板决定的，所以这里矩形的高度也是由最矮的柱子所决定的。我们可以使用两个指针，一个left指向左边的柱子，以他的高为矩形的高度，然后从最右边开始往左扫描，找到比left柱子高的为止（如果没找到，那么矩形的宽度就是0）。计算矩形面积之后，left再往右移一位，再以同样的方式继续查找……。

比如下面的图中计算以第1个柱子的高度为矩形的高度，因为高度一定，要想使矩形的面积最大，就只能是矩形的宽度最大，所以这里从数组的最后面开始找，找到一个比3大或者等于3的值即可，如果没找到那么宽度就是0。



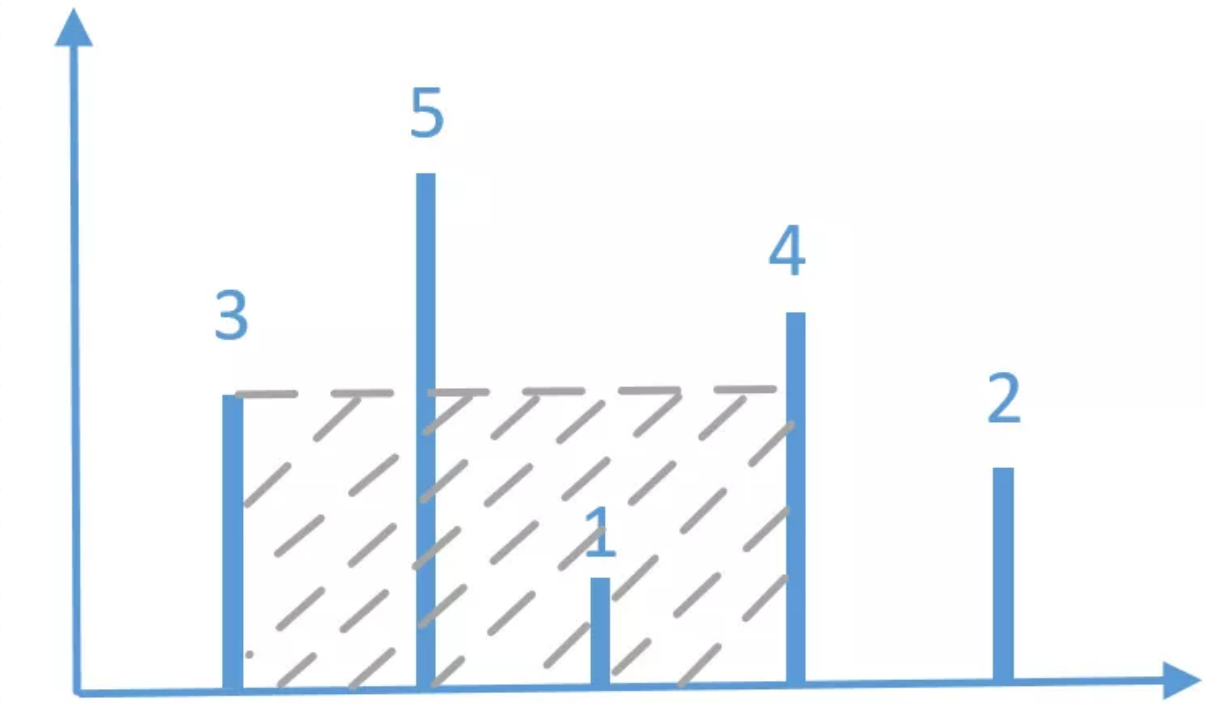
我们查找的时候为了防止遗漏，不光从前面开始找，而且还要从后面开始找，需要两遍查找，代码如下

```
public int maxArea(int[] height) {
  int maxarea = 0, left = 0, length = height.length;
  int area;
  int right;
  //从前面开始找
  while (left < length) {
    right = length - 1;
    while (right > left) {
      if (height[right] < height[left]) {
        right--;
      } else {
        break;
      }
    }
    //计算矩形的面积
    area = height[left] * (right - left);
    //保存计算过的最大的面积
    maxarea = Math.max(maxarea, area);
    left++;
  }
  //从后面开始找，和上面类似
  right = length - 1;
  while (right > 0) {
    left = 0;
    while (right > left) {
      if (height[right] > height[left]) {
        left++;
      } else {
        break;
      }
    }
    area = height[right] * (right - left);
```

```
33        maxarea = Math.max(maxarea, area);
34        right--;
35      }
36      return maxarea;
37    }
```

双指针优化

上面的代码我们两个方向都要查找，是不是感觉有点麻烦，我们再认真看下这个图



比如我们以3为矩形的高度，查找矩形宽度的时候从最右边开始往左找，找到比3大的为止，这里找到了4，那么柱子3到柱子4中间所围成的矩形高度就是柱子3的高度。如果我们从右边开始找的时候是小于3的，比如这里是2，那么我们这里是不是找到了以2为高度的矩形的最大面积。也就是相当于我们可以把从前往后和从后往前找合并为一个，所以这里代码就非常简洁了，我们来看下

```
1   public int maxArea(int[] height) {
2     int maxarea = 0, left = 0, right = height.length - 1;
3     int area = 0;
4     while (left < right) {
5       //计算面积，面积等于宽*高，宽就是left和right之间的距离，高就是
6       //left和right所对应的最低高度
7       area = Math.min(height[left], height[right]) * (right - left);
8       //保存计算过的最大的面积
9       maxarea = Math.max(maxarea, area);
10      //柱子矮的往中间靠
11      if (height[left] < height[right])
12        left++;
13      else
14        right--;
15    }
16    return maxarea;
17  }
```

这题基本上没什么难度，主要考察对双指针的使用。

往期推荐

- 395，动态规划解通配符匹配问题
- 394，经典的八皇后问题和N皇后问题
- 391，回溯算法求组合问题
- 387，二叉树中的最大路径和

591，二叉树的垂序遍历

原创 博哥 数据结构和算法 8月7日

问题描述

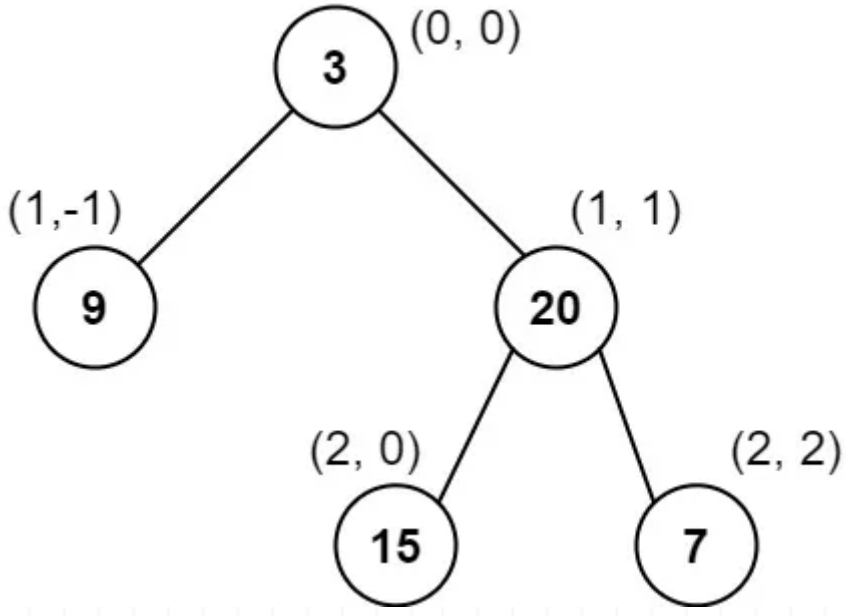
来源：LeetCode第987题
难度：困难

给你二叉树的根结点root，请你设计算法计算二叉树的垂序遍历序列。对位于(row,col)的每个结点而言，其左右子结点分别位于(row+1,col-1)和(row+1,col+1)。树的根结点位于(0,0)。

二叉树的垂序遍历从最左边的列开始直到最右边的列结束，按列索引每一列上的所有结点，形成一个按出现位置从上到下排序的有序列表。**如果同行同列上有多个结点，则按结点的值从小到大进行排序**。

返回二叉树的垂序遍历序列。

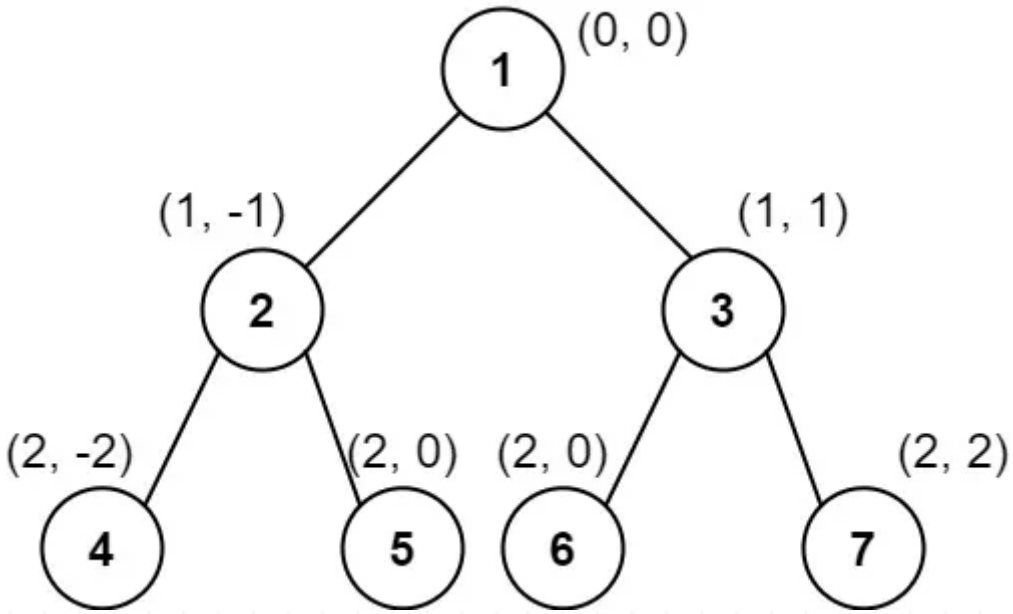
示例 1：



输入：root = [3,9,20,null,null,15,7]
输出：[[9],[3,15],[20],[7]]
解释：
列 -1 ：只有结点9在此列中。
列 0 ：只有结点3和15在此列中，按从上到下顺序。
列 1 ：只有结点20在此列中。

列 2 ：只有结点7在此列中。

示例 2：

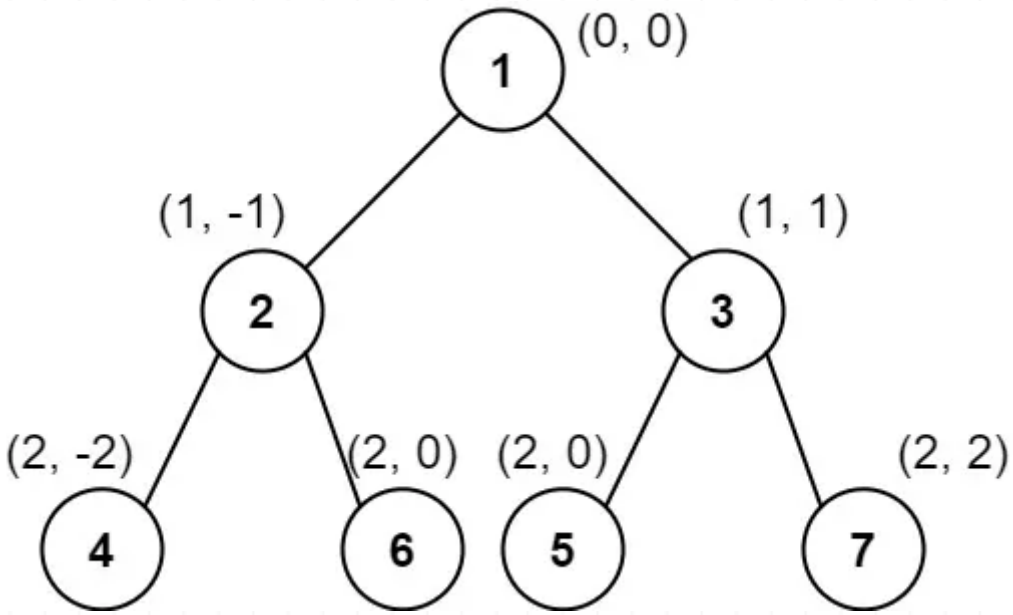


输入：root = [1,2,3,4,5,6,7]
输出：[[4],[2],[1,5,6],[3],[7]]
解释：
列 -2 ：只有结点4在此列中。
列 -1 ：只有结点2在此列中。
列 0 ：结点1、5和6都在此列中。

1在上面，所以它出现在前面。
5和6位置都是(2,0)，所以按值从小到大排序，5在6的前面。

列 1 ：只有结点3在此列中。
列 2 ：只有结点7在此列中。

示例 3：



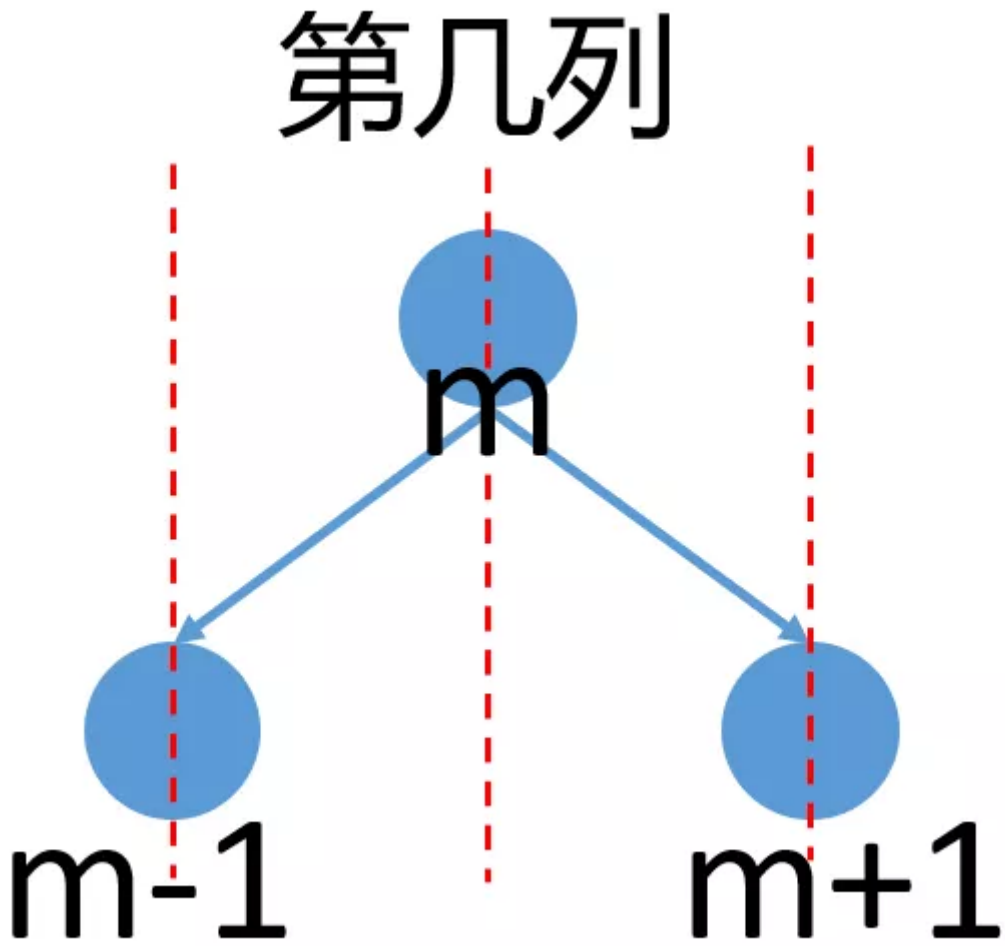
> **输入**：root = [1,2,3,4,6,5,7]
> **输出**：[[4],[2],[1,5,6],[3],[7]]
> **解释**：
> 这个示例实际上与示例2完全相同，只是结点5和6在树中的位置发生了交换。
> 因为5和6的位置仍然相同，所以答案保持不变，仍然按值从小到大排序。

提示：

- 树中结点数目总数在范围[1,1000]内
- 0 <= Node.val <= 1000

问题分析

这题虽然是hard，但其实没有什么难度，做这题我最先想到的就是BFS遍历，如果当前节点是第m列，那么他的左子节点就是第m-1列，右子节点就是m+1列



所以我们可以从上到下一层一层的遍历，使用一个map来存储，map的key存储的是第几列，value是那个列的集合。辛辛苦苦写完之后发现运行不通过（代码放在了下面，有兴趣的可以看下），这是因为题中有这样一句话，如果同行同列上有多个结点，则按结点的值从小到大进行排序。也就是说如果两个节点位置重合了还要按照大小进行排序。

所以这题我们只能按照常规方式来解决，就是先记录下每个节点的值和坐标，最后再把他们按照同一列的顺序放到集合中，关于二叉树的遍历有多种，我们之前也介绍过一些

373，数据结构-6,树

488，二叉树的Morris中序和前序遍历

来看下代码

```
public List<List<Integer>> verticalTraversal(TreeNode root) {
  //list集合中的元素是一个数组，每个数组的长度都是3，第1个值表示
  //节点的值，第2个和第3个值表示节点的横坐标和竖坐标
  List<int[]> mList = new ArrayList<>();
  //计算所有节点的值和坐标，根节点的坐标是（0，0）
  dfs(root, 0, 0, mList);
  //排序，排序的原则是先排左边一列，所以首先比较的是数组的第3个值（
  //纵坐标），然后每一列从上到下也就是数组的第2个值（横坐标），如果
  //前面两个值是一样的说明他们的坐标重合了，要按值从大到小排序
  Collections.sort(mList, (arr1, arr2) -> {
    //先按照纵坐标排序
    if (arr1[2] != arr2[2])
      return arr1[2] - arr2[2];
    if (arr1[1] != arr2[1])
      return arr1[1] - arr2[1];
    //如果坐标一样，再按照值排序
    return arr1[0] - arr2[0];
  });

  //把节点的值进行垂序分类
  List<List<Integer>> res = new ArrayList<>();
  res.add(new ArrayList<>());
  //因为上面排序了，所以这里首先遍历的就是最左边一列的值，
  //然后是第二列……
  for (int i = 0; i < mList.size(); i++) {
    //取出数组（包含当前节点的值和坐标）
    int[] arr = mList.get(i);
    //当前节点的值
    int value = arr[0];
    //如果当前节点和前一个节点不在同一列，说明到下一
    //列了，需要初始化下一列的集合
    if (i > 0 && arr[2] != mList.get(i - 1)[2])
      res.add(new ArrayList<>());
    //把当前节点的值添加到当前列中
    res.get(res.size() - 1).add(value);
  }
  return res;
}

private void dfs(TreeNode node, int i, int j, List<int[]> mList) {
  if (node == null)
    return;
  //把当前节点的值和坐标加入到集合中,当前节点的坐标是（i，j）
  mList.add(new int[]{node.val, i, j});
  //遍历左子节点
  dfs(node.left, i + 1, j - 1, mList);
  //遍历右子节点
  dfs(node.right, i + 1, j + 1, mList);
}
```

//错误的代码（同一行和同一列的两个值没有按照大小进行排序）

```
public List<List<Integer>> verticalTraversal(TreeNode root) {
  Map<Integer, ArrayList<Integer>> map = new HashMap<>();
  //存放节点的值
  Queue<TreeNode> nodeQueue = new LinkedList<>();
  //存放每个节点对应第几列
```

```
    Queue<Integer> lineQueue = new LinkedList<>();
    //根节点是第0列，他左子节点的列是负数，右子节点的列是正数
    nodeQueue.add(root);
    lineQueue.add(0);
    //记录最小的列，也就是最左边的列
    int mineLine = 0;
    while (!nodeQueue.isEmpty()) {
        //当前节点及对应的列出队
        TreeNode curNode = nodeQueue.poll();
        int line = lineQueue.poll();
        //记录最左边的列
        mineLine = Math.min(mineLine, line);
        //如果对应列的集合不存在就初始化一个
        if (!map.containsKey(line)) {
            map.put(line, new ArrayList<>());
        }
        //把当前节点对应的值加入到当前节点对应列的集合中
        map.get(line).add(curNode.val);
        //左子节点不为空，就把他以及他所在的列加入到队列中
        if (curNode.left != null) {
            nodeQueue.add(curNode.left);
            lineQueue.add(line - 1);
        }
        //右子节点同上
        if (curNode.right != null) {
            nodeQueue.add(curNode.right);
            lineQueue.add(line + 1);
        }
    }
    //把Map转化为List，map中key存储的是第几列，
    //value存储的是每一列的集合
    List<List<Integer>> res = new ArrayList<>();
    int end = mineLine + map.size();
    for (int i = mineLine; i < end; i++) {
        res.add(map.get(i));
    }
    return res;
}
```

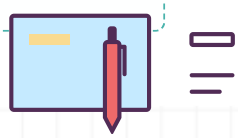
往期推荐

- 564，二叉树最大宽度
- 488，二叉树的Morris中序和前序遍历
- 464. BFS和DFS解二叉树的所有路径
- 456，解二叉树的右视图的两种方式

582，DFS解二叉树剪枝

原创 博哥 数据结构和算法 1周前

> The higher I got, the more amazed I was by the view.
>
> 我爬得越高，越为眼前的风景所惊叹。



问题描述

来源：LeetCode第814题
难度：中等

给定二叉树根结点root，此外树的每个结点的值要么是0，要么是1。返回移除了所有不包含1的子树的原二叉树。

（节点X的子树为X本身，以及所有X的后代。）

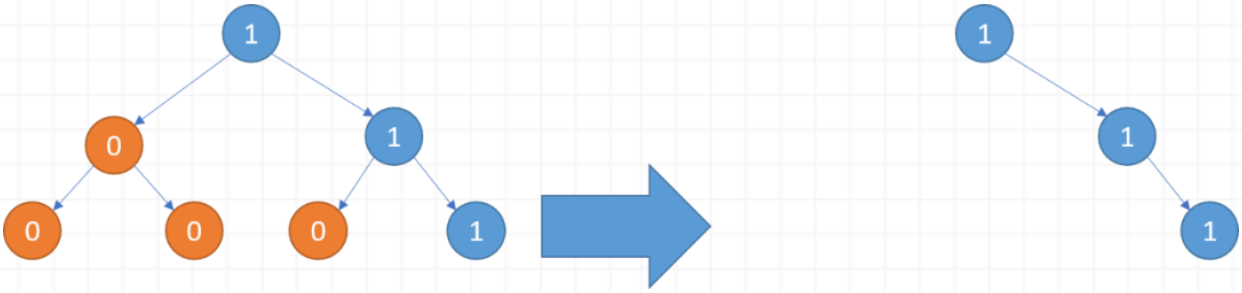
示例1：

> **输入**：[1,null,0,0,1]
> **输出**：[1,null,0,null,1]
>
> **解释**：
> 只有红色节点满足条件“所有不包含 1 的子树”。
> 右图为返回的答案。



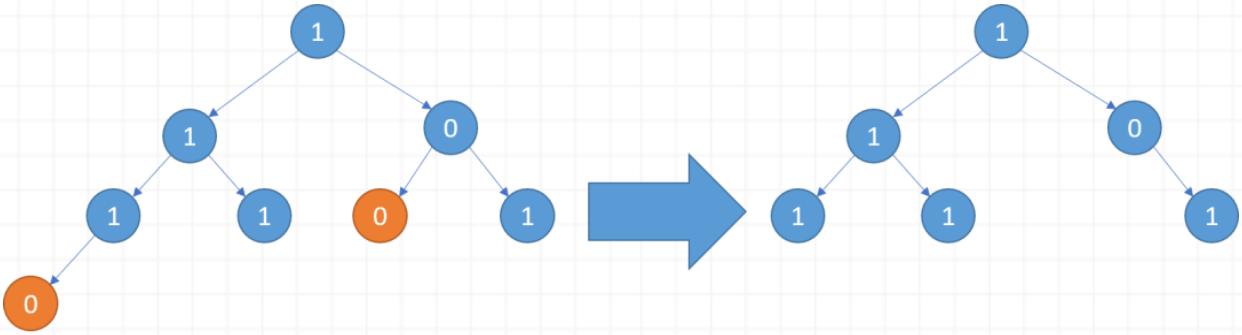
示例2：

> **输入**: [1,0,1,0,0,0,1]
> **输出**: [1,null,1,null,1]



示例3:

> **输入**: [1,1,0,1,1,0,1,0]
> **输出**: [1,1,0,1,1,null,1]



说明:

- 给定的二叉树最多有200个节点。
- 每个节点的值只会为0或1。

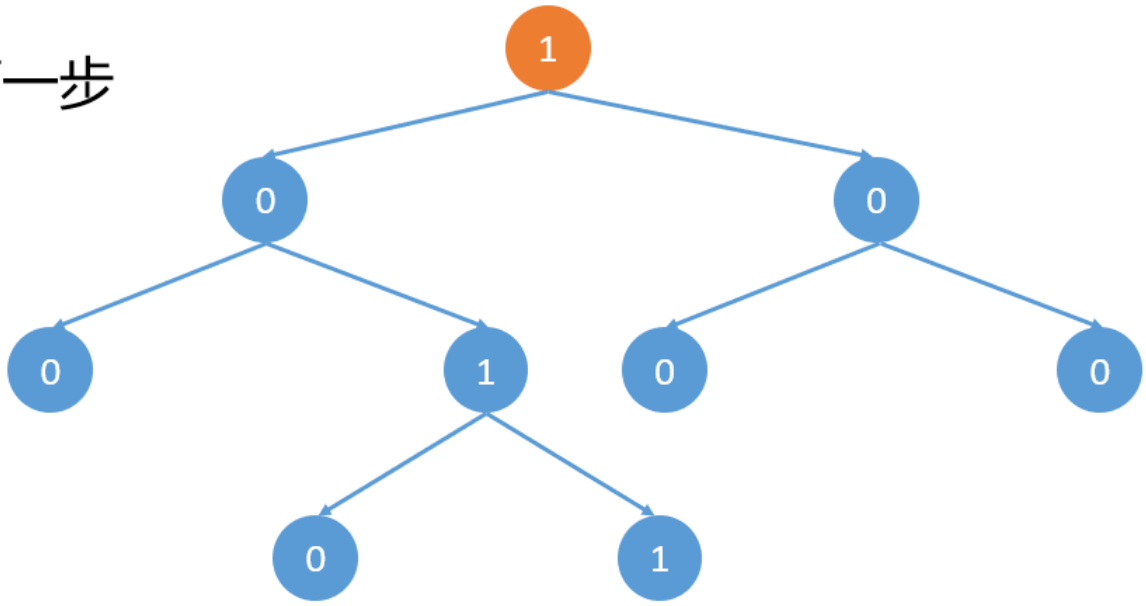
DFS解决

这题是说如果二叉树的任一节点的左子树都是0，就把他的左子树删除，同理如果右子树都是0，也把他的右子树给删除。

这题常规思路是从根节点开始统计他的左右子树节点是否都是0，如果任一子树的所有节点都是0，就把他给删除。否则不能删除，然后再统计不能删除节点的子节点……，那这样的话工作量就非常大，并且还会出现大量重复计算，如下图所示:

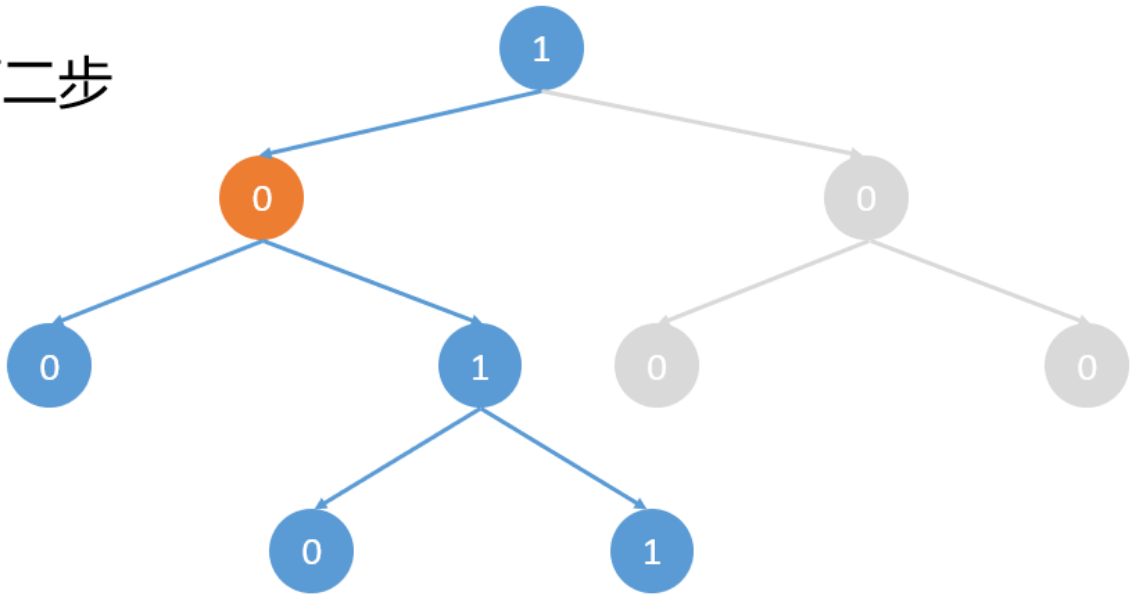
当前节点的左子树不全是0，所以不能删除，但他的右子树都是0，可以删除

第一步

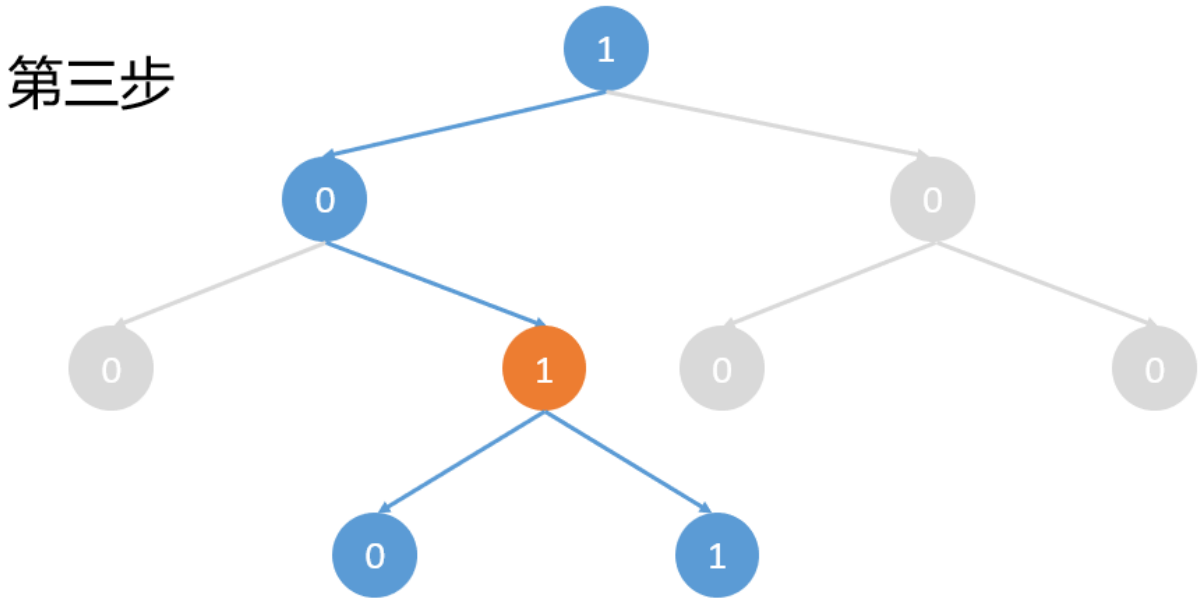


当前节点的左子树都是0，可以删除，但他的右子树不全是0，所以不可以删除

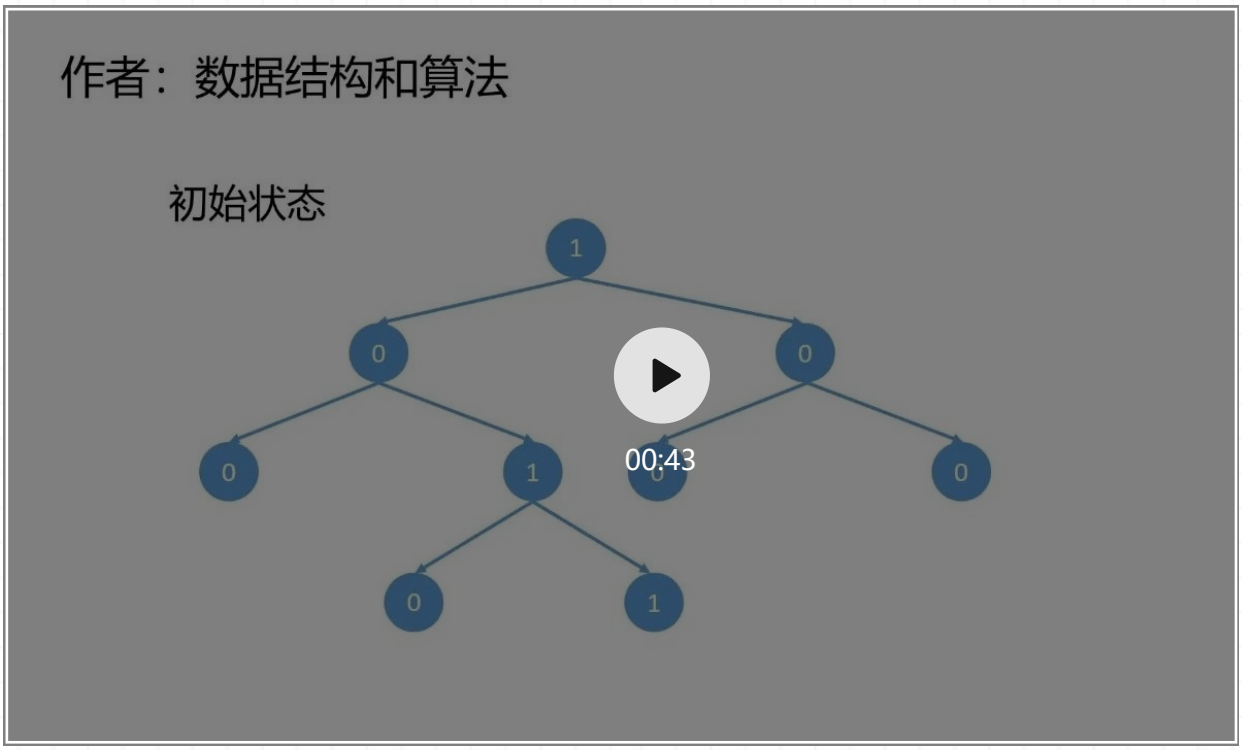
第二步



.....



上面方式明显是行不通的。我们可以这样来思考一下，从上到下删不行，那么从下到上呢，如果是叶子节点并且值为0，我们就把他给删除，否则不要删除，看下视频



看到这里大家可能就明白了，这就是二叉树的后续遍历，我们来看下代码

```
1   public TreeNode pruneTree(TreeNode root) {
2     if (root == null)
3       return null;
4     //这里要注意，当前节点可能不是叶子节点，但如果他的子节点
5     //都删除完了，它就变成了叶子节点。
6     //剪枝左子树
7     root.left = pruneTree(root.left);
8     //剪枝右子树
9     root.right = pruneTree(root.right);
10    //如果叶子节点的值是0，就把他给删除，返回一个空的节点
```

```
11      if (root.left == null && root.right == null && root.val == 0)
12        return null;
13      //否则不要删除，直接返回即可
14      return root;
15    }
```

时间复杂度：O(n)，n是二叉树中节点的个数。

空间复杂度：O(h)，h是树的高度，也是递归的深度。

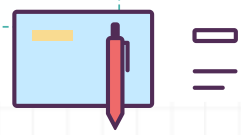
往期推荐

- 580，BFS和DFS解二叉树的堂兄弟节点
- 575，回溯算法和DFS解单词拆分 II
- 566，DFS解目标和问题
- 507，BFS和DFS解二叉树的层序遍历 II

564，二叉树最大宽度

原创 博哥 数据结构和算法 1周前

> It isn't the big pleasures that count the most; it's making a great deal out of the little ones.
>
> 最重要的不是有大快乐，而是能充分享受小快乐。



问题描述

来源：LeetCode第662题
难度：中等

给定一个二叉树，编写一个函数来获取这个树的最大宽度。树的宽度是所有层中的最大宽度。这个二叉树与满二叉树（full binary tree）结构相同，但一些节点为空。

每一层的宽度被定义为两个端点（该层最左和最右的非空节点，两端点间的null节点也计入长度）之间的长度。

示例 1：

```
输入：
           1
         /   \
        3     2
       / \     \
      5   3     9

输出：4
解释：最大值出现在树的第 3 层，宽度为 4 (5,3,null,9)。
```

示例 2：

```
输入：
          1
```

```
          /
         3
        / \
       5   3

输出：2
解释：最大值出现在树的第 3 层，宽度为 2 (5,3)。
```

示例 3:

```
输入：
          1
         / \
        3   2
       /
      5

输出：2
解释：最大值出现在树的第 2 层，宽度为 2 (3,2)。
```

示例 4:

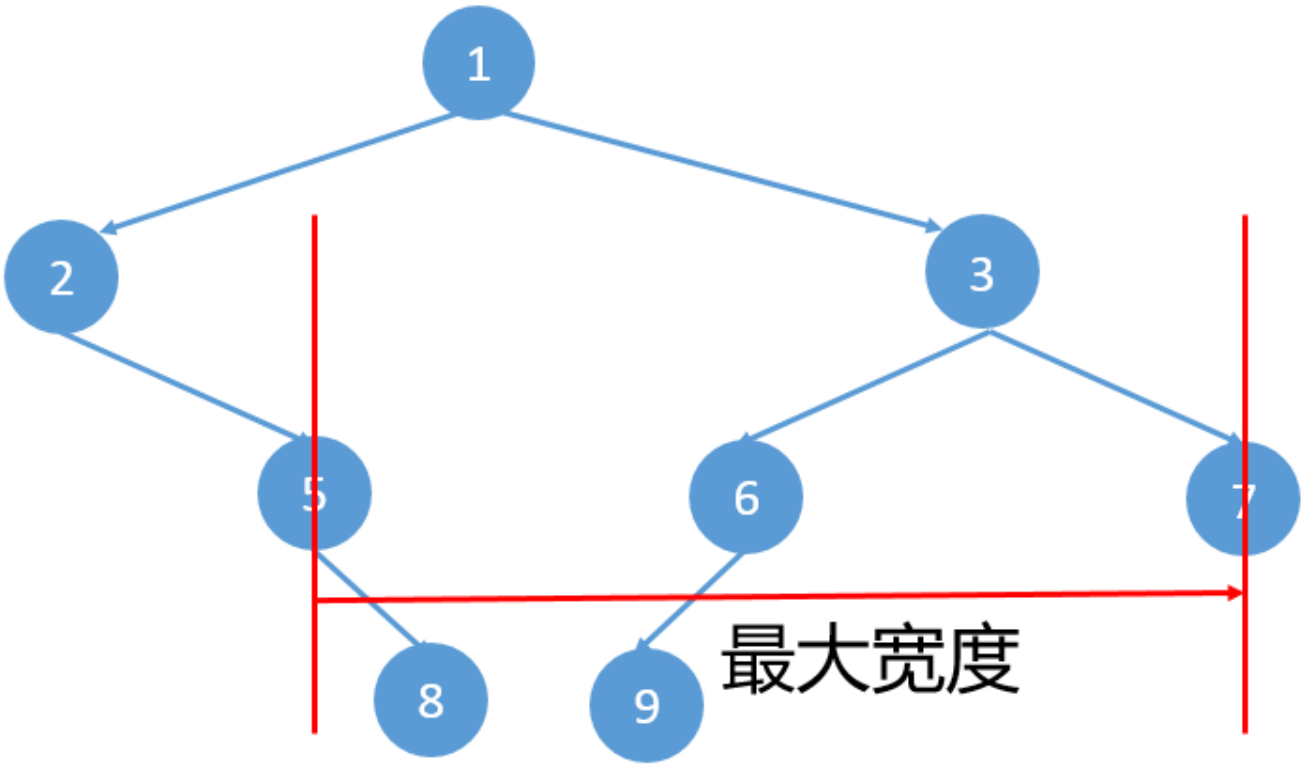
```
输入：
          1
         / \
        3   2
       /     \
      5       9
     /         \
    6           7
输出：8
解释：最大值出现在树的第 4 层，宽度为 8 (6,null,null,null,null,null,null,7)。
```

注意：答案在32位有符号整数的表示范围内。

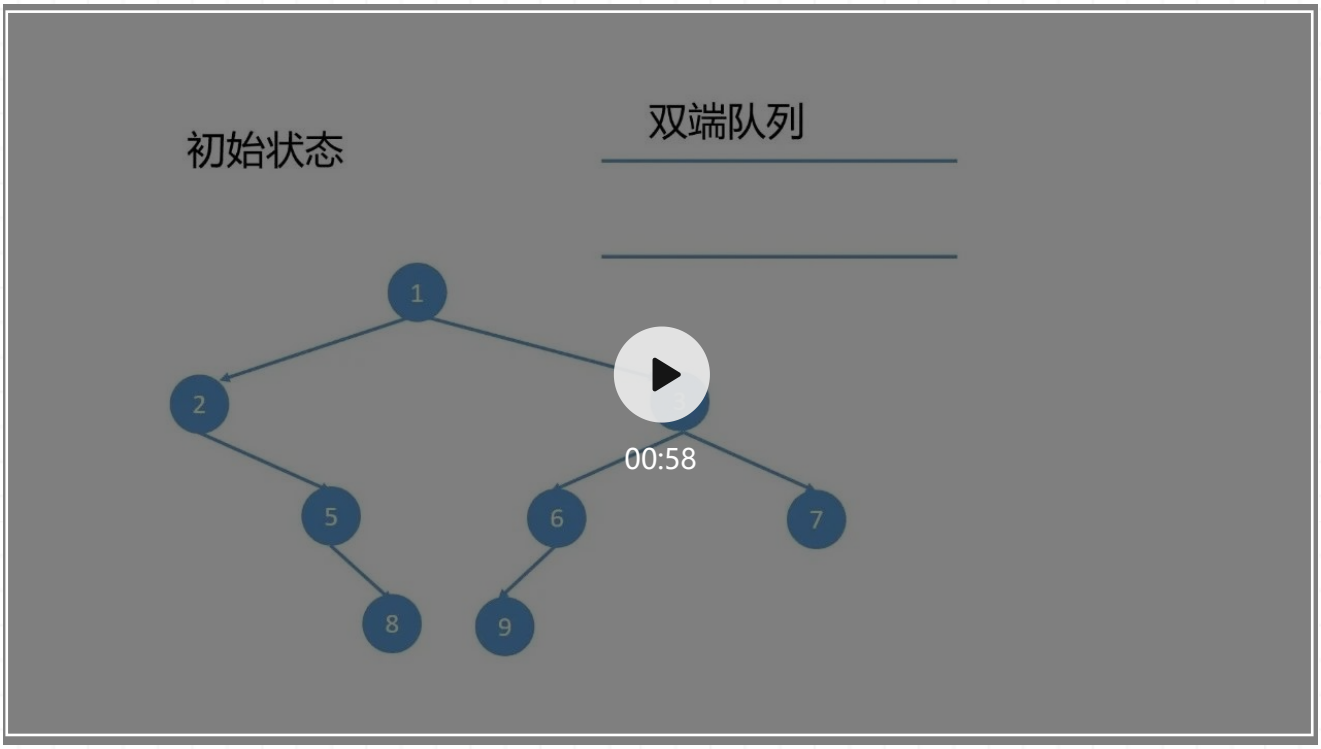
BFS解决

二叉树的最大宽度我们可以认为是从最左边到最右边的最大距离，假如是一棵满二叉树的话，每一层的最大距离就是最左边到最右边的节点数。因为二叉树不一定都是满二叉树，

有可能像下面这样。



每一行从最左边到最右边我们很容易想到的就是二叉树的BFS遍历，他就是一层一层遍历的，关于二叉树的BFS不明白的可以看下下面的视频。



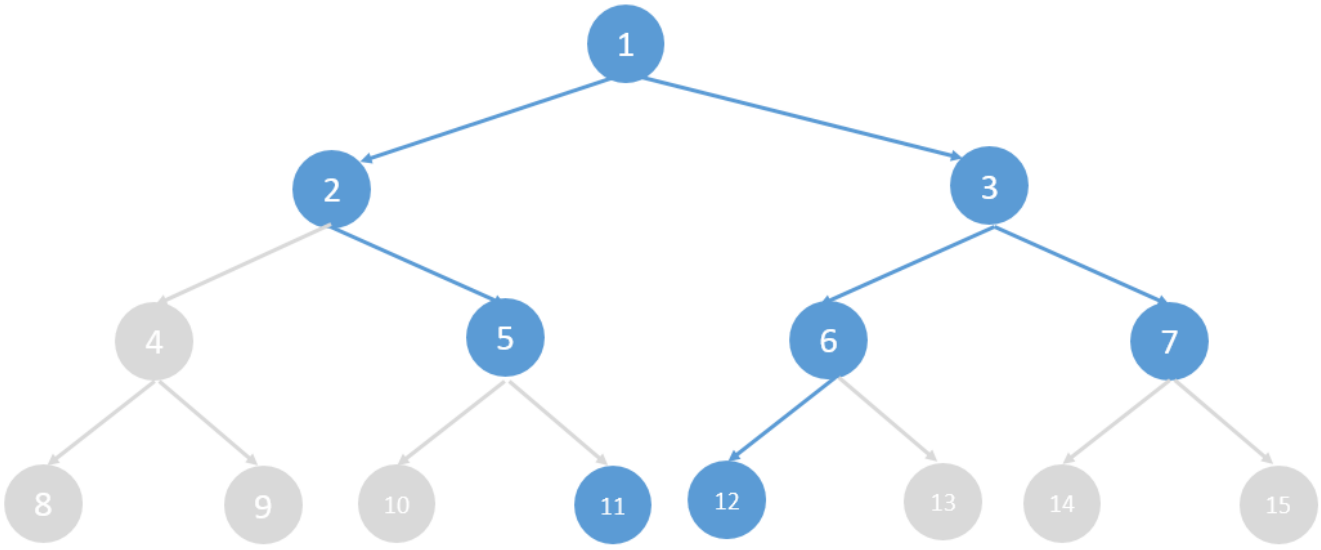
所以这题思路很容易想到，就是遍历每一层的时候计算这一层最左边节点到最右边节点的距离，大致代码如下

```
public int widthOfBinaryTree(TreeNode root) {
    if (root == null)
        return 0;
    //使用双端队列
    Deque<TreeNode> queue = new LinkedList<>();
    //把根节点加入到队列中
```

```
    queue.offer(root);
    //记录最大的宽度
    int maxWide = 0;
    while (!queue.isEmpty()) {
      //当前层节点的数量
      int levelCount = queue.size();
      // int gap = 当前层最左边到最右边的距离
      maxWide = Math.max(maxWide, gap);
      //遍历当前层的所有节点，把他们的子节点在加入到队列中
      for (int i = 0; i < levelCount; i++) {
        TreeNode node = queue.poll();
        //如果左子节点不为空，就把左子节点加入到队列中
        if (node.left != null) {
          queue.offer(node.left);
        }
        //如果右子节点不为空，就把右子节点加入到队列中
        if (node.right != null) {
          queue.offer(node.right);
        }
      }
    }
    return maxWide;
  }
```

他就是从上往下一层一层遍历的，遍历每一层的时候我们需要计算当前层的最大距离，保留最大的即可。这里关键是怎么计算当前层的最大距离。

我们可以这样来计算，把它想象成为一颗满二叉树，假如根节点是遍历的第1个节点，那么他的两个子节点分别是遍历的第2个和第3个节点。并且可以推算出如果一个节点是第n个遍历的，那么他的两个子节点分别是第n*2和n*2+1个遍历的，具体我们来画个图看一下



我们可以把这些值存到一个map中，也可以把它直接存到节点中，这里我们就把他存到节点中，在遍历每一层的时候用当前层最右边的值减去最左边的值+1就是当前层的最大距离，来看下最终代码。

```
  public int widthOfBinaryTree(TreeNode root) {
    if (root == null)
      return 0;
    //使用双端队列
    Deque<TreeNode> queue = new LinkedList<>();
    //把根节点加入到队列中
```

```
    queue.offer(root);
    //根节点的值我们把它修改为1
    root.val = 1;
    //记录最大的宽度
    int maxWide = 0;
    while (!queue.isEmpty()) {
      //当前层节点的数量
      int levelCount = queue.size();
      //当前层最左边节点的值
      int left = queue.peekFirst().val;
      //当前层最右边节点的值
      int right = queue.peekLast().val;
      //当前层的最大宽度就是right - left + 1，
      //这里计算之后要保留最大的
      maxWide = Math.max(maxWide, right - left + 1);
      //遍历当前层的所有节点，把他们的子节点在加入到队列中
      for (int i = 0; i < levelCount; i++) {
        TreeNode node = queue.poll();
        int position = node.val;
        //如果左子节点不为空，就把左子节点加入到队列中
        if (node.left != null) {
          node.left.val = position * 2;
          queue.offer(node.left);
        }
        //如果右子节点不为空，就把右子节点加入到队列中
        if (node.right != null) {
          node.right.val = position * 2 + 1;
          queue.offer(node.right);
        }
      }
    }
    return maxWide;
  }
```

这里因为左边节点先入队，所以peekFirst()返回的就是当前层最左边的节点，右边节点是最后入队的，所以peekLast()返回的是当前层最右边的节点。

DFS解决

BFS是一层一层的打印，DFS是沿着一个分支一直走下去，当到达叶子节点的时候在往回走。所以这题解题思路也很简单，我们使用两个变量left和right

- left变量记录每一层最左边的节点。
- right变量记录每一层遍历过的最右边节点。

然后再每一层计算最右边到最左边的距离，也就是当前层的最大宽度，来看下代码

```
  //记录最大的宽度
  int maxWide = 0;

  public int widthOfBinaryTree(TreeNode root) {
    dfs(root, 0, 1, new ArrayList<>(), new ArrayList<>());
    return maxWide;
  }

  /**
   * @param root
   * @param level    遍历到第几层
   * @param position 每个节点在满二叉树中的位置
   * @param left     只记录最左边的节点，每层只记录一个
   * @param right    只记录遍历过的最右边节点，每层只记录一个（这里是遍历过的，如果当前层有更右边的节点，
   *                 会把当前层的替换）
   */
```

```java
public void dfs(TreeNode root, int level, int position, List<Integer> left, List<Integer> right) {
    if (root == null)
        return;
    //首次到当前层，要把当前值分别加入到left和right中
    if (left.size() == level) {
        left.add(position);
        right.add(position);
    } else {//如果当前层已经遍历过，会替换掉
        right.set(level, position);
    }
    //递归遍历下一层的左右子节点
    dfs(root.left, level + 1, position << 1, left, right);
    dfs(root.right, level + 1, position * 2 + 1, left, right);
    //计算当前层的最大间距，保留最大值
    maxWide = Math.max(maxWide, right.get(level) - left.get(level) + 1);
}
```

往期推荐

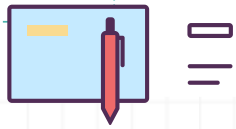
- 507，BFS和DFS解二叉树的层序遍历 II
- 488，二叉树的Morris中序和前序遍历
- 464. BFS和DFS解二叉树的所有路径
- 456，解二叉树的右视图的两种方式

563，N叉树的最大深度

原创 博哥 数据结构和算法 1周前

We cherish each moment of our lives.

珍惜生命中的每一刻。



问题描述

来源：LeetCode第559题

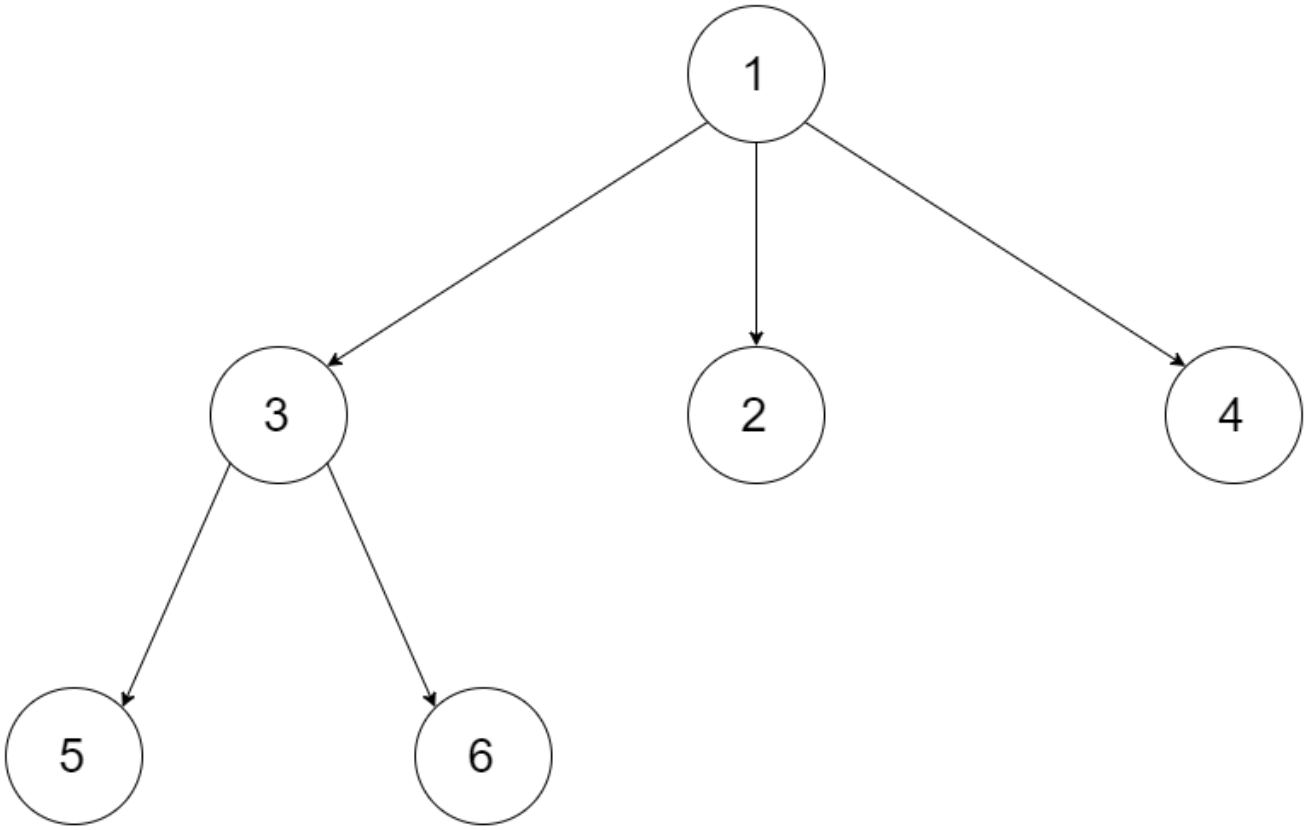
难度：简单

给定一个N叉树，找到其最大深度。

最大深度是指从根节点到最远叶子节点的最长路径上的节点总数。

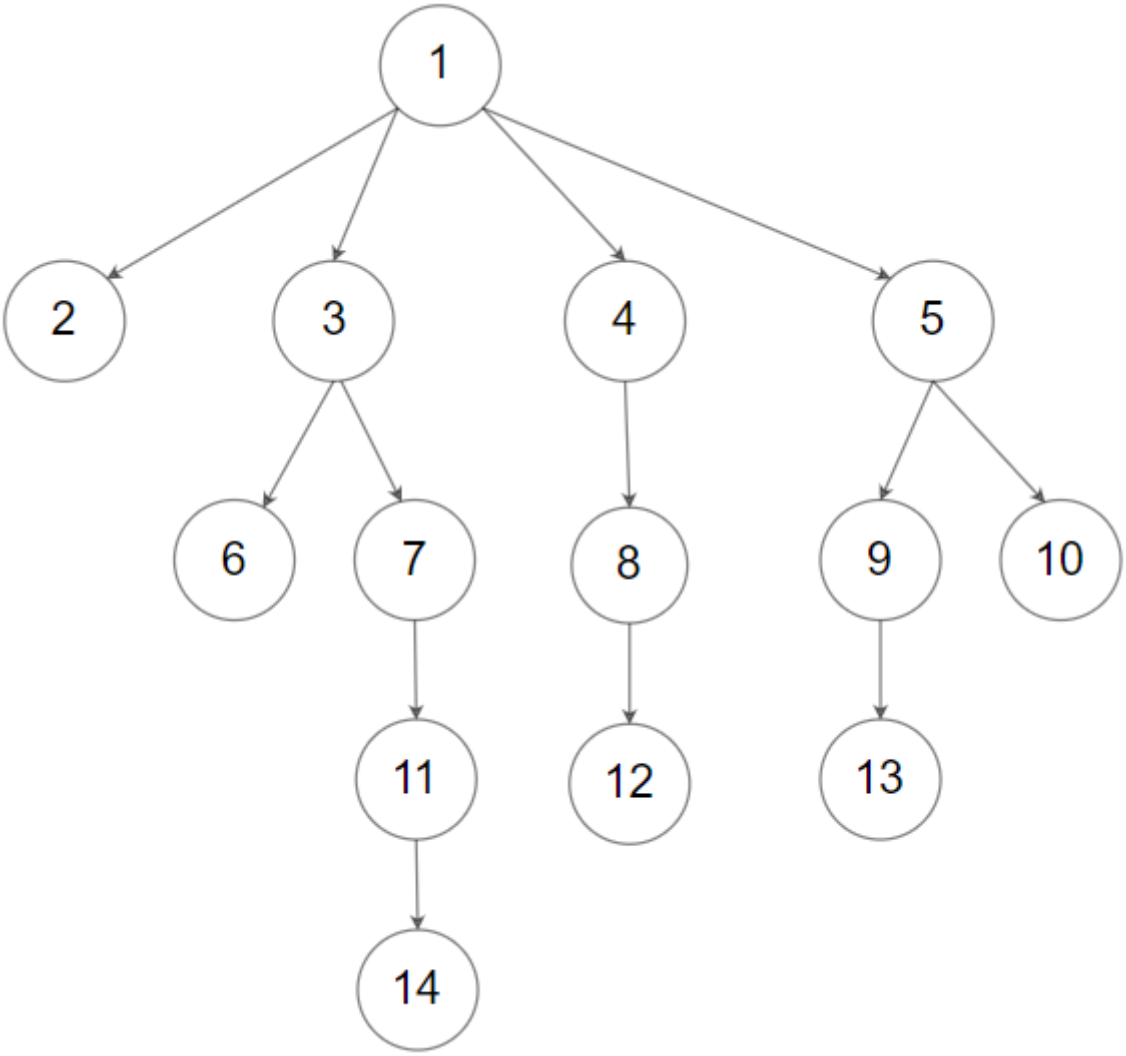
N 叉树输入按层序遍历序列化表示，每组子节点由空值分隔（请参见示例）。

示例 1：



> **输入**：root = [1,null,3,2,4,null,5,6]
> **输出**：3

示例 2：



> **输入**：root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]
> **输出**：5

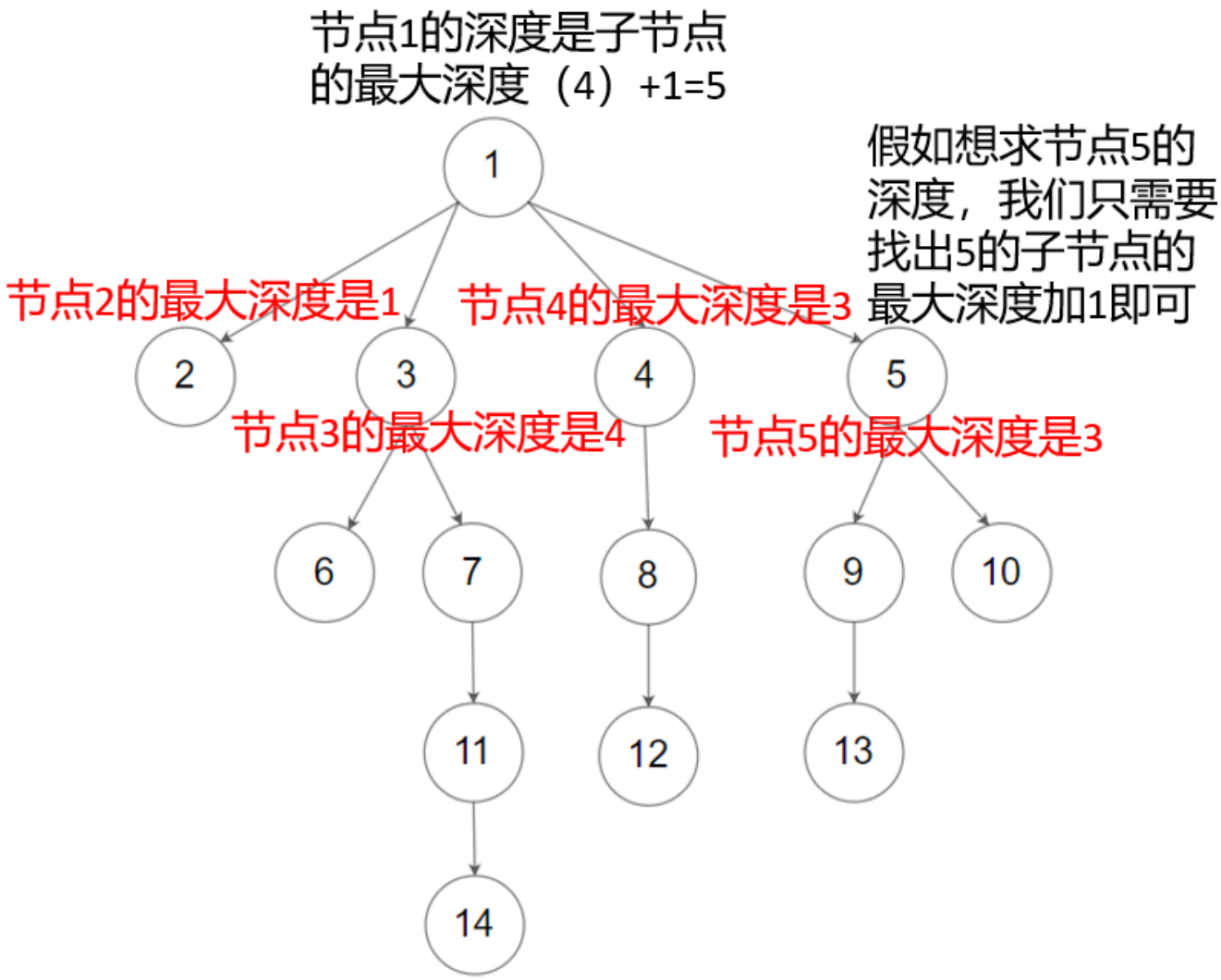
提示：

- 树的深度不会超过1000 。
- 树的节点数目位于[0，10^4] 之间。

DFS解决

之前讲过367，二叉树的最大深度，第367题是二叉树，而这题是N叉树。二叉树的每个节点最多有两个子节点，N叉树的每个节点最多有N个子节点，其实解法都差不多。

计算每一个节点的最大深度，我们需要找出他所有子节点的最大深度，然后再加上1，画个图来看一下

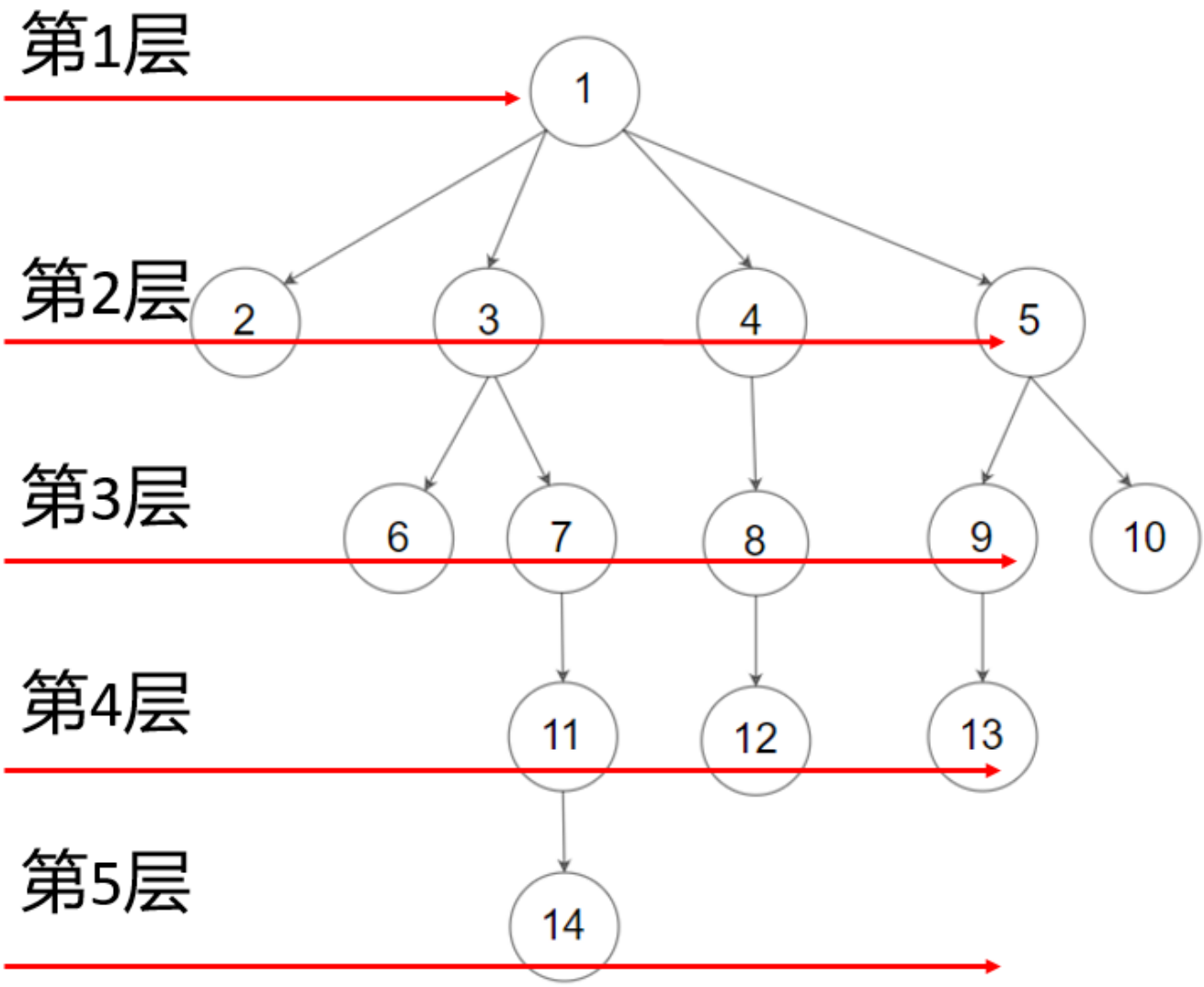


来看下代码

```
1   public int maxDepth(Node root) {
2     if (root == null)
3       return 0;
4     //当前节点子节点的个数
5     int size = root.children.size();
6     int max = 0;
7     //递归计算所有子节点的深度，保留最大值
8     for (int i = 0; i < size; i++) {
9       max = Math.max(max, maxDepth(root.children.get(i)));
10    }
11    //当前树的最大深度就是子节点的最大深度加上1
12    return max + 1;
13  }
```

BFS解决

BFS是一层一层的遍历，当我们把这棵N叉树遍历完的时候，只需要记录遍历了多少层，总共遍历的层数就是这题的答案，如下图所示



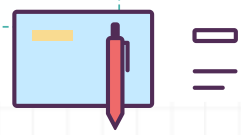
来看下代码

```
1   public int maxDepth(Node root) {
2     if (root == null)
3       return 0;
4     Queue<Node> queue = new LinkedList<>();
5     queue.offer(root);
6     int depth = 0;
7     while (!queue.isEmpty()) {
8       //到下一层了，深度要加1
9       depth++;
10      //levelCount是当前层的节点数，
11      int levelCount = queue.size();
12      for (int i = 0; i < levelCount; i++) {
13        //当前层的每一个节点都要出队，然后再
14        //把他们的子节点加入到队列中
15        Node current = queue.poll();
16        for (Node child : current.children)
17          queue.offer(child);
18      }
19    }
20    return depth;
21  }
```

561，二叉搜索树中第K小的元素

博哥 数据结构和算法 5月30日

> Instead of holding on to those who have already left, cherish those who stayed behind.
>
> 与其执著于谁当初离你而去，不如感谢谁最后留了下来。

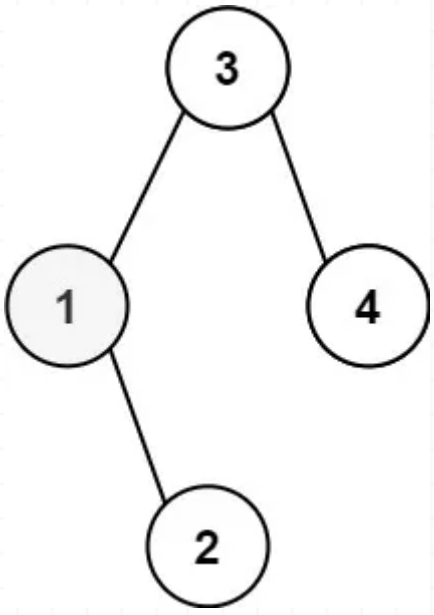


问题描述

来源：LeetCode第230题
难度：中等

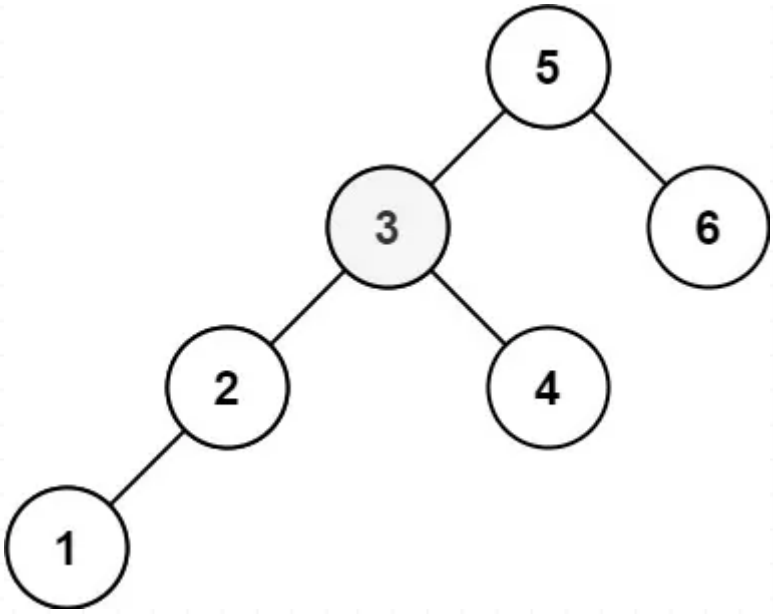
给定一个二叉搜索树的根节点root，和一个整数k，请你设计一个算法查找其中第k个最小元素（从1开始计数）。

示例 1：



> **输入**：root = [3,1,4,null,2], k = 1
> **输出**：1

示例 2：



> **输入**：root = [5,3,6,2,4,null,null,1], k = 3
> **输出**：3

提示：

- 树中的节点数为n。
- 1<=k<=n<=10^4
- 0<=Node.val<=10^4

中序遍历解决

这题说的是取二叉搜索树的第k小的元素，做这题之前首先要明白什么是二叉搜索树。wiki百科上对二叉搜索树是这样定义的：

二叉查找树（英语：Binary Search Tree），也称为二叉查找树、有序二叉树（ordered binary tree）或排序二叉树（sorted binary tree），是指一棵空树或者具有下列性质的二叉树：

- **若任意节点的左子树不空，则左子树上所有节点的值均小于它的根节点的值；**
- **若任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值；**
- **任意节点的左、右子树也分别为二叉查找树；**

二叉搜索树有一个非常重要的特性就是：二叉搜索树的中序遍历结果一定是有序的。有了这个特性这题就简单多了，我们只需要按照中序遍历的顺序，取他的第k个元素即可。

二叉树的中序遍历之前讲过，有递归的，非递归的，还有Morris，这里就不在重复介绍，具体可以看下

373，数据结构-6,树

488，二叉树的Morris中序和前序遍历

我们随便挑一个，比如二叉树中序遍历的递归写法如下

```
public void inOrderTraversal(TreeNode node) {
  if (node == null)
    return;
  inOrderTraversal(node.left);
  System.out.println(node.val);
  inOrderTraversal(node.right);
}
```

我们来对他进行修改一下，就是这题的答案了

```
int target = -1;
int count;

public int kthSmallest(TreeNode root, int k) {
  count = k;
  inOrderTraversal(root);
  return target;
}

public void inOrderTraversal(TreeNode node) {
  if (node == null)
    return;
  //访问左子节点
  inOrderTraversal(node.left);
  //访问当前节点，如果访问到第k个就把target赋值
  if (--count == 0) {
    target = node.val;
    return;
  }
  //访问右子节点
  inOrderTraversal(node.right);
}
```

统计节点个数

因为是中序遍历，先统计左子节点的个数，

- 如果左子节点的个数大于等于k，说明要找的元素就在左子节点中
- 否则如果左子节点的个数加上当前节点个数（1）等于k，说明当前节点就是要找的元素，直接返回即可。
- 否则我们要找的元素在右子节点中，直接到右子节点中查找。

```
public int kthSmallest(TreeNode root, int k) {
  //先统计左子节点的个数
  int leftCount = countNodes(root.left);
  if (leftCount >= k) {
    //如果左子节点的个数大于等于k，说明我们要找的元素就在左子节点中，
    //直接在左子节点中查找即可
    return kthSmallest(root.left, k);
  } else if (leftCount + 1 == k) {
    //如果左子节点的个数加当前节点（1）正好等于k，说明根节点
    //就是要找到元素
    return root.val;
  } else {
    //否则要找的元素在右子节点中，到右子节点中查找
    return kthSmallest(root.right, k - 1 - leftCount);
  }
}
```

```
17
18  //统计节点个数
19  public int countNodes(TreeNode n) {
20    if (n == null)
21      return 0;
22    return 1 + countNodes(n.left) + countNodes(n.right);
23  }
```

往期推荐

- 544，剑指 Offer-平衡二叉树
- 507，BFS和DFS解二叉树的层序遍历 II
- 488，二叉树的Morris中序和前序遍历
- 483，完全二叉树的节点个数

547，叶子相似的树

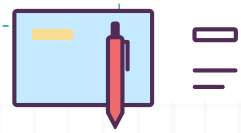
原创 博哥 数据结构和算法 5月5日

收录于话题
#算法图文分析

161个 >

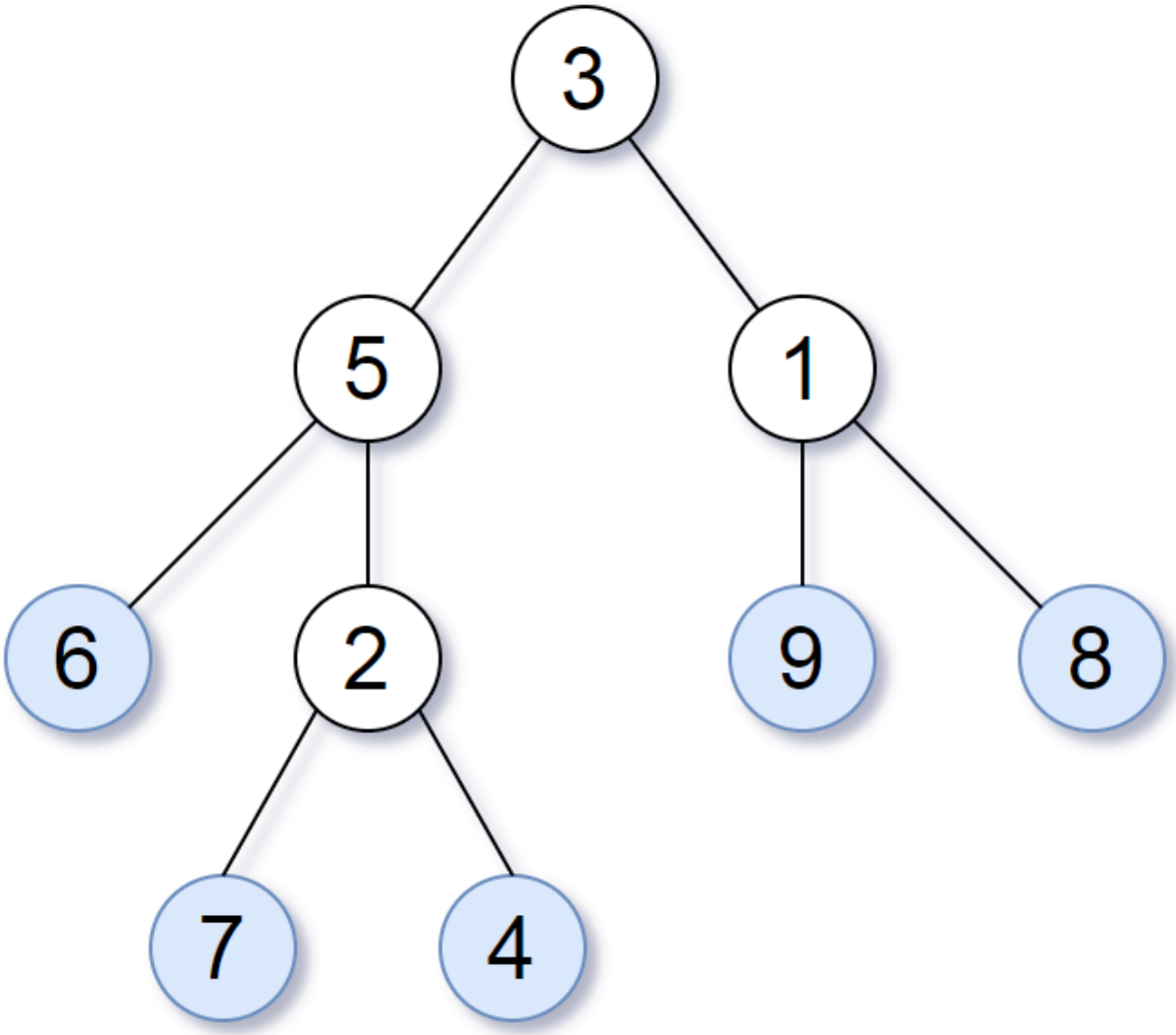
Gentle attitude for a heart to be contempt indeed is a great comfort.

柔和的态度对于一颗被轻蔑的心是很大的安慰。



问题描述

请考虑一棵二叉树上所有的叶子，这些**叶子的值**按从左到右的顺序排列形成一个叶值序列 。

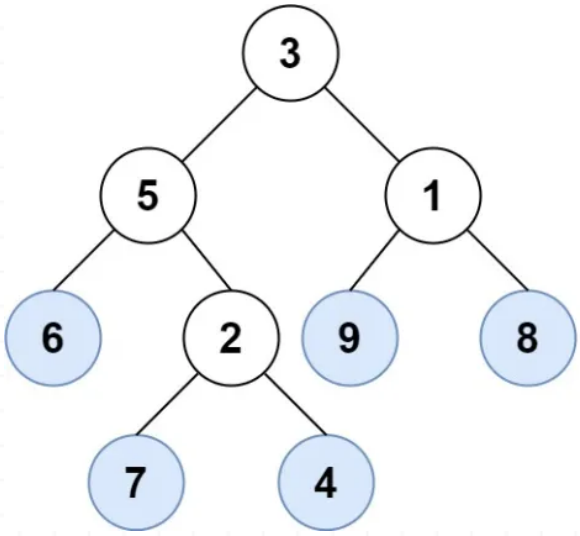


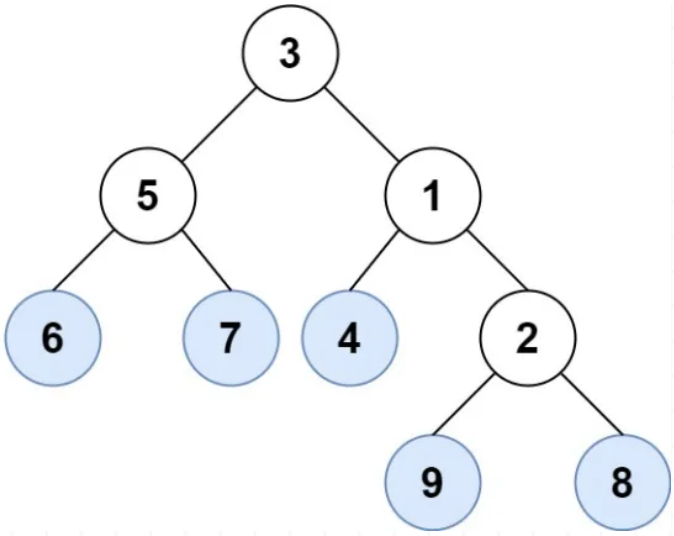
举个例子，如上图所示，给定一棵叶值序列为（6，7，4，9，8）的树。

如果有两棵二叉树的叶值序列是相同，那么我们就认为它们是叶相似的。

如果给定的两个头结点分别为root1和root2的树是叶相似的，则返回 true；否则返回 false 。

示例 1：





> **输入**：
> root1 = [3,5,1,6,2,9,8,null,null,7,4],
> root2 = [3,5,1,6,7,4,2,null,null,null,null,null,null,9,8]
> **输出**：true

示例 2：

> **输入**：root1 = [1], root2 = [1]
> **输出**：true

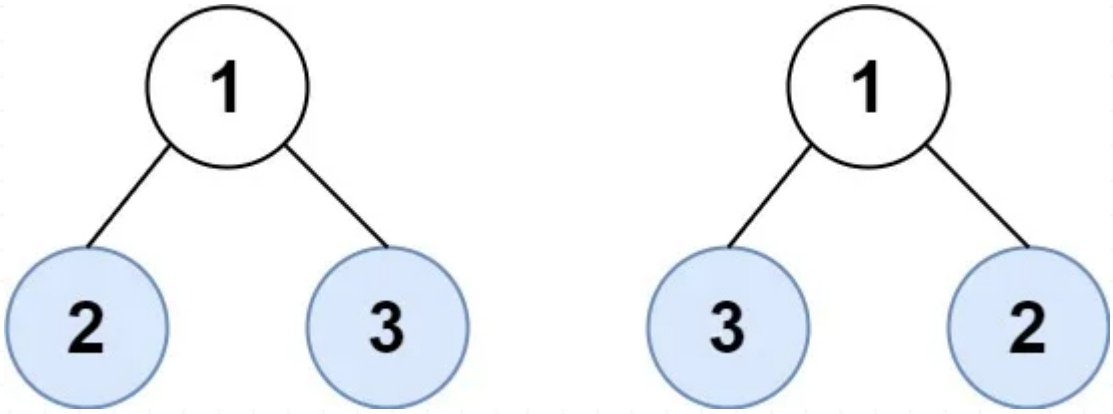
示例 3：

> **输入**：root1 = [1], root2 = [2]
> **输出**：false

示例 4：

> **输入**：root1 = [1,2], root2 = [2,2]
> **输出**：true

示例5：



> **输入**：root1 = [1,2,3], root2 = [1,3,2]
> **输出**：false

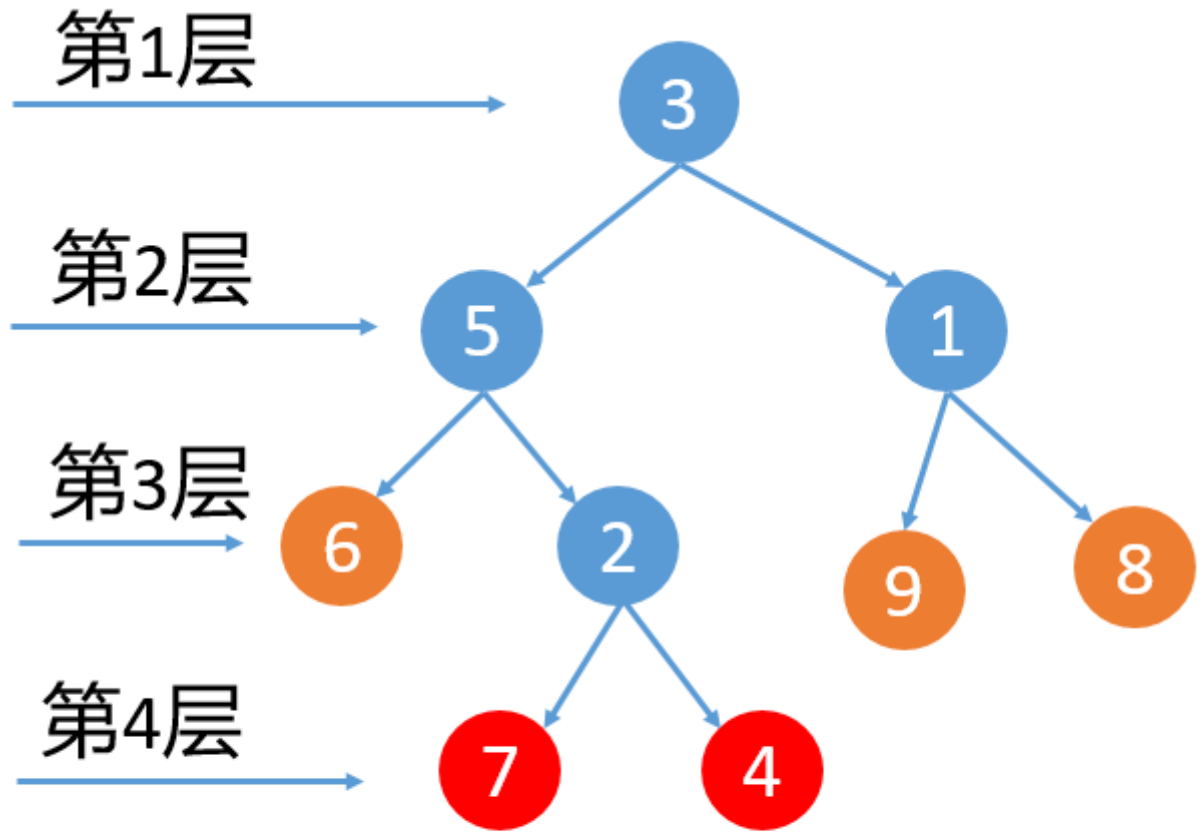
提示：

- 给定的两棵树可能会有1到200个结点。
- 给定的两棵树上的值介于0到200之间。

dfs解决

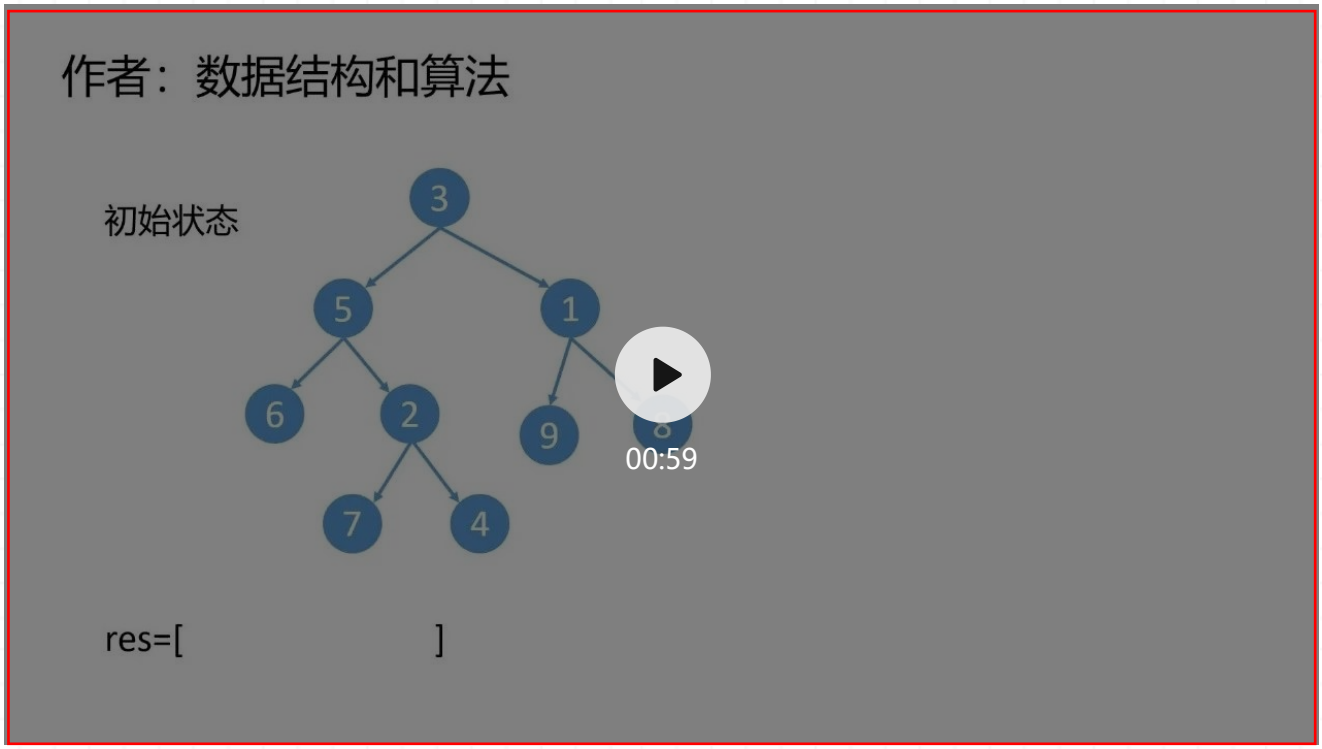
这题是让判断两棵树的叶子节点从左往右的排列顺序是否一样，一种常见的方式就是先统计每棵树叶子节点的值。统计树的叶子节点比较简单，只需要遍历每一个节点，判断是否是叶子节点，如果是叶子节点就把它的值加入到list集合中。最后在判断这两棵树的叶子节点集合是否完全相同即可。

遍历树的每一个节点会有多种方式，我们首先来看一下BFS是否可以，BFS是一层层的遍历的，如下图所示，遍历的结果是[6，9，8，7，4]，很明显是错误的，因为顺序弄乱了。



res=[6, 9, 8, 7, 4]

如果想要保证顺序，我们可以使用DFS，具体统计可以看下视频



叶子节点统计出来了，我们只需要判断统计结果的是否完全一致即可，来看下代码。

```
1   public boolean leafSimilar(TreeNode root1, TreeNode root2) {
2     //记录root1的的叶子节点
```

```
    List<Integer> mListLeaf1 = new ArrayList<>();
    //记录root2的的叶子节点
    List<Integer> mListLeaf2 = new ArrayList<>();
    dfs(root1, mListLeaf1);
    dfs(root2, mListLeaf2);
    //下面是判断统计两棵树的叶子节点值是否一样
    if (mListLeaf1.size() != mListLeaf2.size())
      return false;
    for (int i = 0; i < mListLeaf1.size(); i++) {
      if (mListLeaf1.get(i) != mListLeaf2.get(i))
        return false;
    }
    return true;
  }

  //统计树的叶子节点
  private void dfs(TreeNode root, List<Integer> mList) {
    //边界条件判断，如果是空，直接返回
    if (root == null)
      return;
    //如果是叶子节点，就把叶子节点的值加入到集合mList中
    if (root.left == null && root.right == null) {
      mList.add(root.val);
      //叶子节点是没有子节点的，不需要再往下找了
      return;
    }
    //如果不是叶子节点，分别统计当前节点左右子树的叶子节点
    dfs(root.left, mList);
    dfs(root.right, mList);
  }
```

StringBuilder解决

除了上面使用集合List，我们还可以使用StringBuilder，原理都是一样的，换汤不换药，来看下代码。

```
  public boolean leafSimilar(TreeNode root1, TreeNode root2) {
    //sb1和sb2分别记录root1和root2的叶子节点的值
    StringBuilder sb1 = new StringBuilder();
    StringBuilder sb2 = new StringBuilder();
    dfs(root1, sb1);
    dfs(root2, sb2);
    return sb1.toString().equals(sb2.toString());
  }

  private void dfs(TreeNode root, StringBuilder sb) {
    //边界条件判断，如果是空，直接返回
    if (root == null)
      return;
    //如果是叶子节点，就把叶子节点的值加入到StringBuilder中
    if (root.left == null && root.right == null) {
      sb.append(root.val + "#");
      return;
    }
    //如果不是叶子节点，分别统计当前节点左右子树的叶子节点
    dfs(root.left, sb);
    dfs(root.right, sb);
  }
```

545，二叉搜索树的范围和

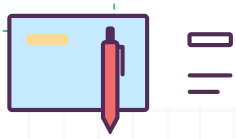
原创 博哥 数据结构和算法 4天前

收录于话题
#算法图文分析

148个 >

Nothing is impossible.

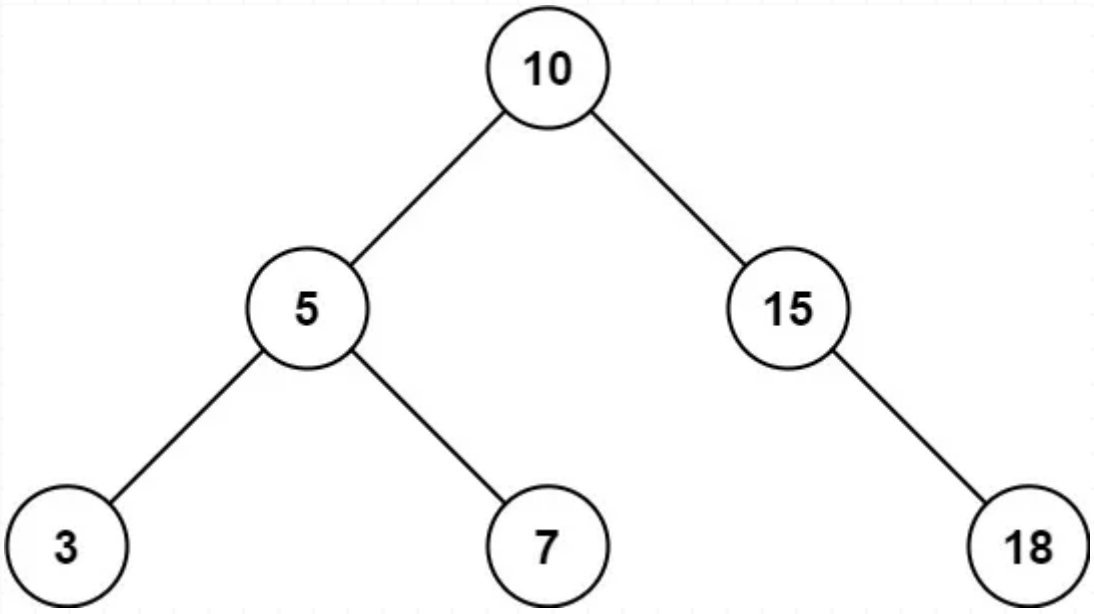
没有什么是不可能的。



问题描述

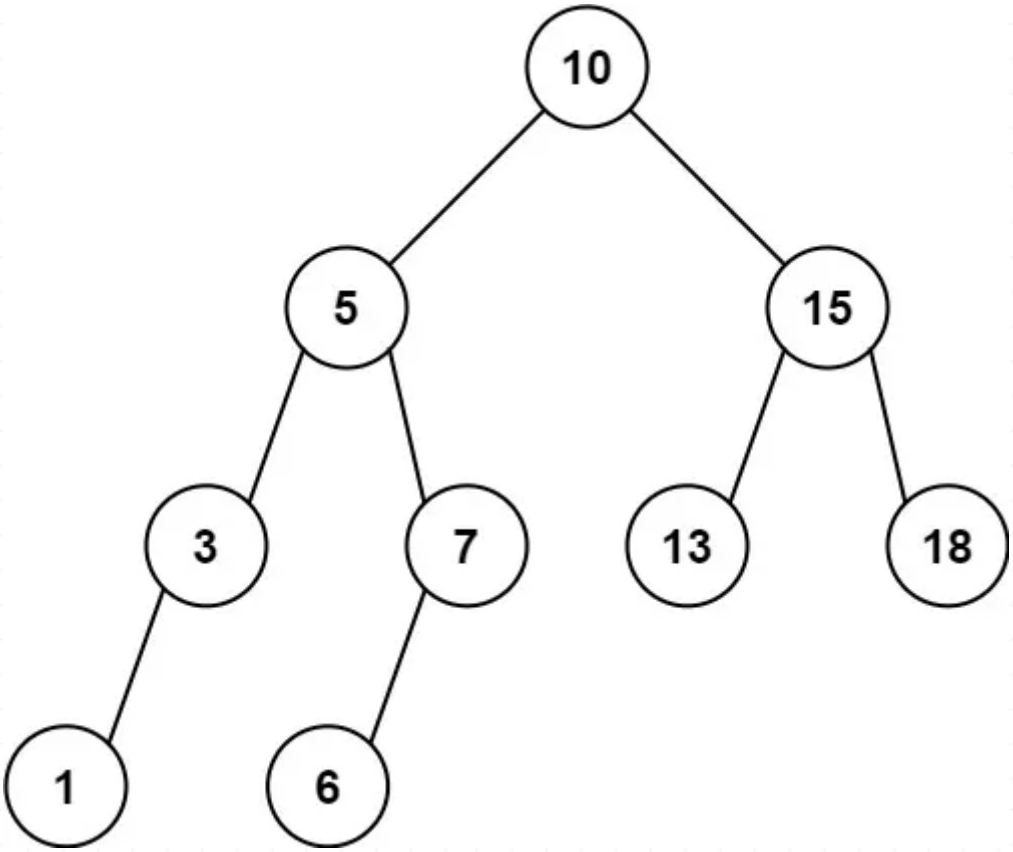
给定**二叉搜索树**的根结点root，返回值位于范围[low, high]之间的所有结点的值的和。

示例 1：



> **输入**：root = [10,5,15,3,7,null,18],
> low = 7, high = 15
>
> **输出**：32

示例 2：



> **输入**：root = [10,5,15,3,7,13,18,1,null,6],
> low = 6, high = 10
>
> **输出**：23

提示：

- 树中节点数目在范围 [1, 2 * 10^4] 内
- 1 <= Node.val <= 10^5
- 1 <= low <= high <= 10^5
- 所有Node.val互不相同

逐个遍历

我们遍历二叉树的每一个节点，判断他的值是否在[low, high]之间，如果在这个之间，我们就统计他们的和。

而遍历二叉树的方式有很多，前序，中序，后续，BFS，并且每种方式都有递归和非递归等多种实现方式，如果还嫌不够，还有Morris的前序，中序，后续。那这样写下来答案就比较多了。

关于二叉树的前中后，以及BFS遍历可以看下《373，数据结构-6,树》

关于二叉树的Morris遍历方式可以看下《488，二叉树的Morris中序和前序遍历》

解法比较多，这里就随便挑一个来写，比如二叉树的中序遍历递归写法如下

```
public void inOrderTraversal(TreeNode node) {
  if (node == null)
    return;
  inOrderTraversal(node.left);
  System.out.println(node.val);
  inOrderTraversal(node.right);
}
```

我们来对他改造一下，遍历每个节点的时候判断他的值

```
int res = 0;

public int rangeSumBST(TreeNode root, int low, int high) {
  inOrderTraversal(root, low, high);
  return res;
}

public void inOrderTraversal(TreeNode node, int low, int high) {
  if (node == null)
    return;
  inOrderTraversal(node.left, low, high);
  //如果当前节点的值在[low, high]之间，就累加
  if (node.val >= low && node.val <= high)
    res += node.val;
  inOrderTraversal(node.right, low, high);
}
```

代码优化

这题有个条件就是**二搜索叉树**，我们知道**二叉搜索树的特点就是左子树的所有节点值都比当前节点值小，右子树的所有节点值都比当前节点值大**，如果我们按照**中序遍历**的方式打印的话，就会发现打印的结果是**升序**排列的。

上面那种解法遍历每一个节点，虽然也能解决问题，但很效率不是最好的，对于二叉搜索树

- 如果当前节点**小于low**，那么他它的**左子树**的所有节点肯定也都小于low，我们没必要在遍历了，直接放弃
- 如果当前节点**大于high**，那么他的**右子树**的所有节点肯定也都大于high，也可以放弃

来看下代码

```
public int rangeSumBST(TreeNode root, int low, int high) {
  //递归边界条件判断
  if (root == null)
    return 0;
  //当前节点以及他的右子树的值都太大了，不要了
  if (root.val > high) {
    return rangeSumBST(root.left, low, high);
  }
  //当前节点以及他的左子树的值都太小了，也不要了
```

```
10      if (root.val < low) {
11          return rangeSumBST(root.right, low, high);
12      }
13      //如果当前节点值在[low, high]之间，就留下
14      return root.val + rangeSumBST(root.left, low, high) + rangeSumBST(root.right, low, high);
15  }
```

往期推荐

- 510，将有序数组转换为二叉搜索树
- 503，二叉搜索树中的众数
- 488，二叉树的Morris中序和前序遍历
- 466. 使用快慢指针把有序链表转换二叉搜索树

544，剑指 Offer-平衡二叉树

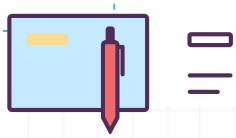
原创 博哥 数据结构和算法 6天前

收录于话题
#剑指offer

34个 >

Time goes on and on, never to an end but crossings.

时间一直走，没有尽头，只有路口。



问题描述

输入一棵二叉树的根节点，判断该树是不是平衡二叉树。如果某二叉树中任意节点的左右子树的高度相差不超过1，那么它就是一棵平衡二叉树。

示例 1:

给定二叉树 [3,9,20,null,null,15,7]

```
    3
   / \
  9  20
    /  \
   15   7
```

返回 true 。

示例 2:

给定二叉树 [1,2,2,3,3,null,null,4,4]

```
      1
     / \
    2   2
   / \
  3   3
 / \
```

```
7   4   4
```

返回 false 。

限制：

- 0 <= 树的结点个数 <= 10000

从上到下

平衡二叉树要求的是左右子节点的高度不能超过1，所以我们可以判断树的左右两个子节点的高度只要不超过1就行，而树的高度怎么计算呢，其实很简单，代码如下

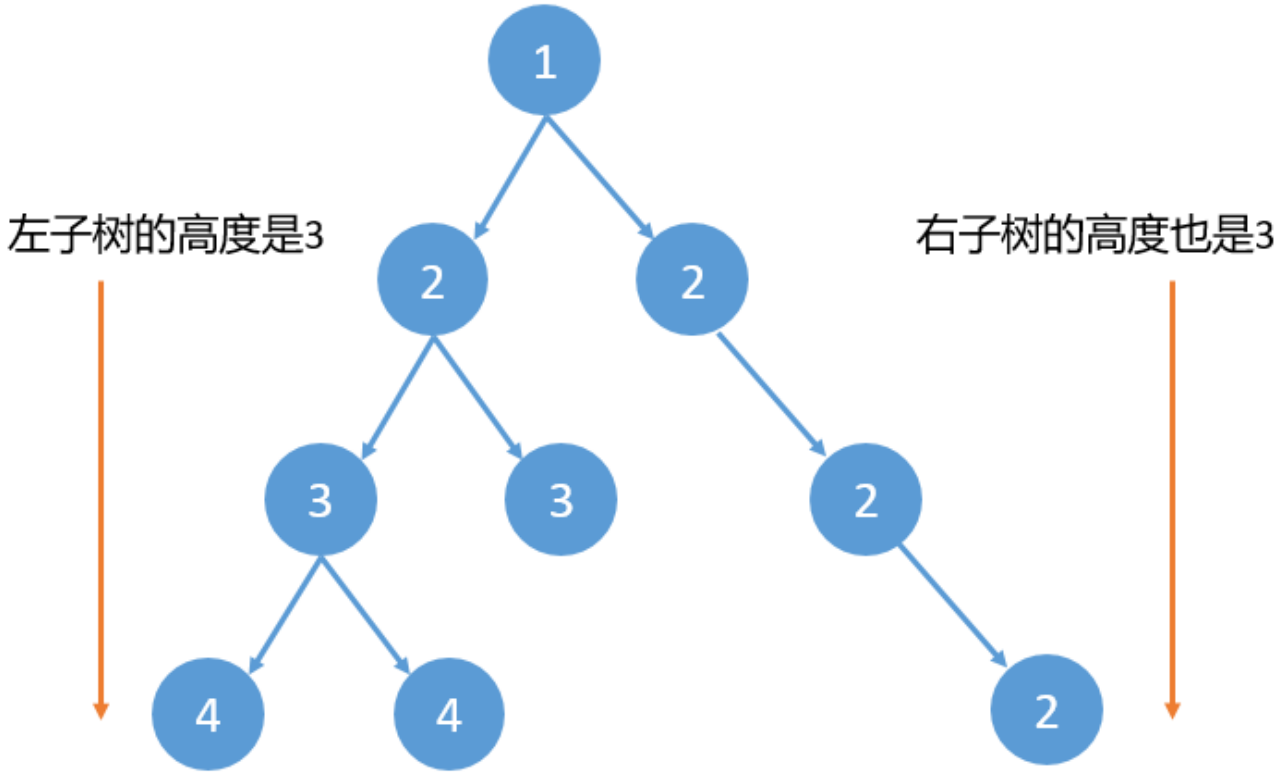
```
//计算树中节点的高度
public int depth(TreeNode root) {
  if (root == null)
    return 0;
  return Math.max(depth(root.left), depth(root.right)) + 1;
}
```

所以这题的代码我们也很容易写出来

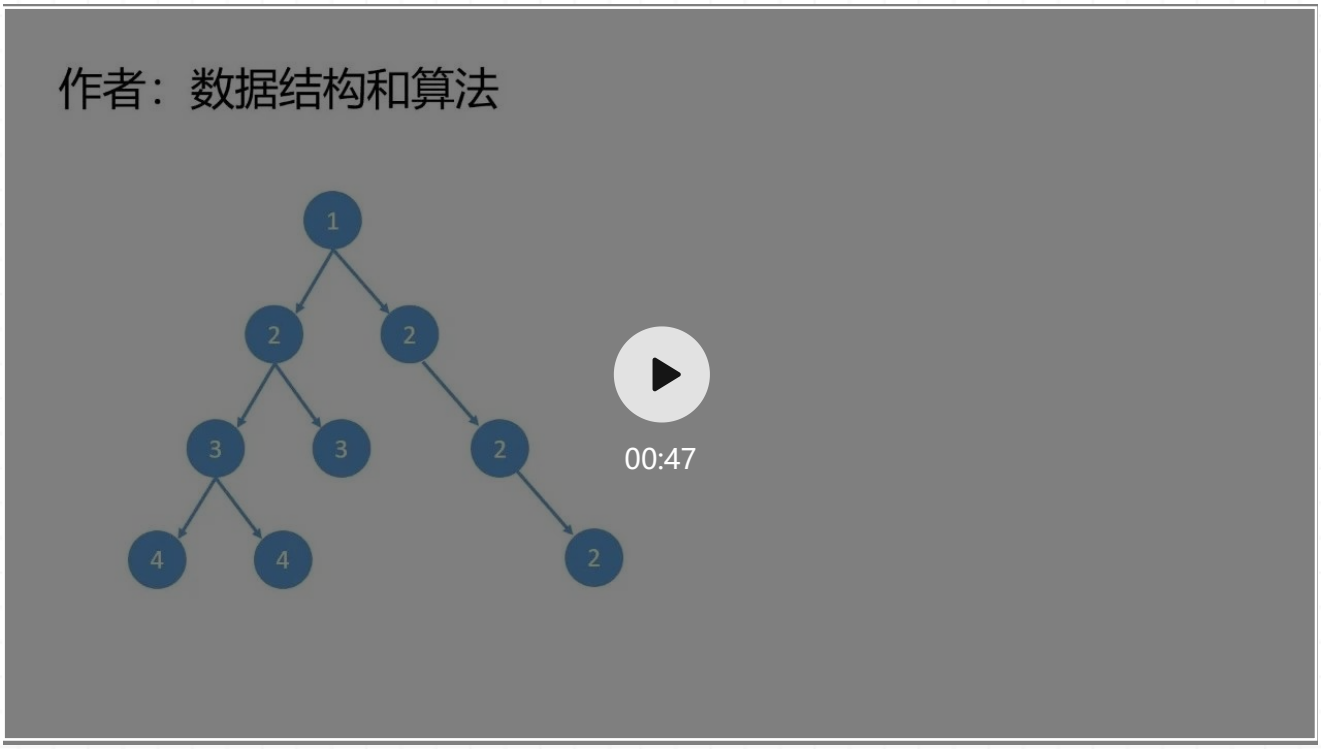
```
public boolean isBalanced(TreeNode root) {
  if (root == null)
    return true;
  //分别计算左子树和右子树的高度
  int left = depth(root.left);
  int right = depth(root.right);
  //这两个子树的高度不能超过1
  return Math.abs(left - right) <= 1;
}

//计算树中节点的高度
public int depth(TreeNode root) {
  if (root == null)
    return 0;
  return Math.max(depth(root.left), depth(root.right)) + 1;
}
```

但这里会有个问题，因为二叉平衡树的任何一棵子树也都必须是平衡的，上面我们只判断了根节点的两个子节点的高度是否小于等于1，没有判断子树是否是平衡的。

如下图所示，虽然根节点的两个子节点的高度是一样的，但很明显根节点的右子树不是平衡的，也就是说这棵树不是平衡二叉树。



所以除了判断根节点以外，还需要判断所有的子节点，具体看下视频



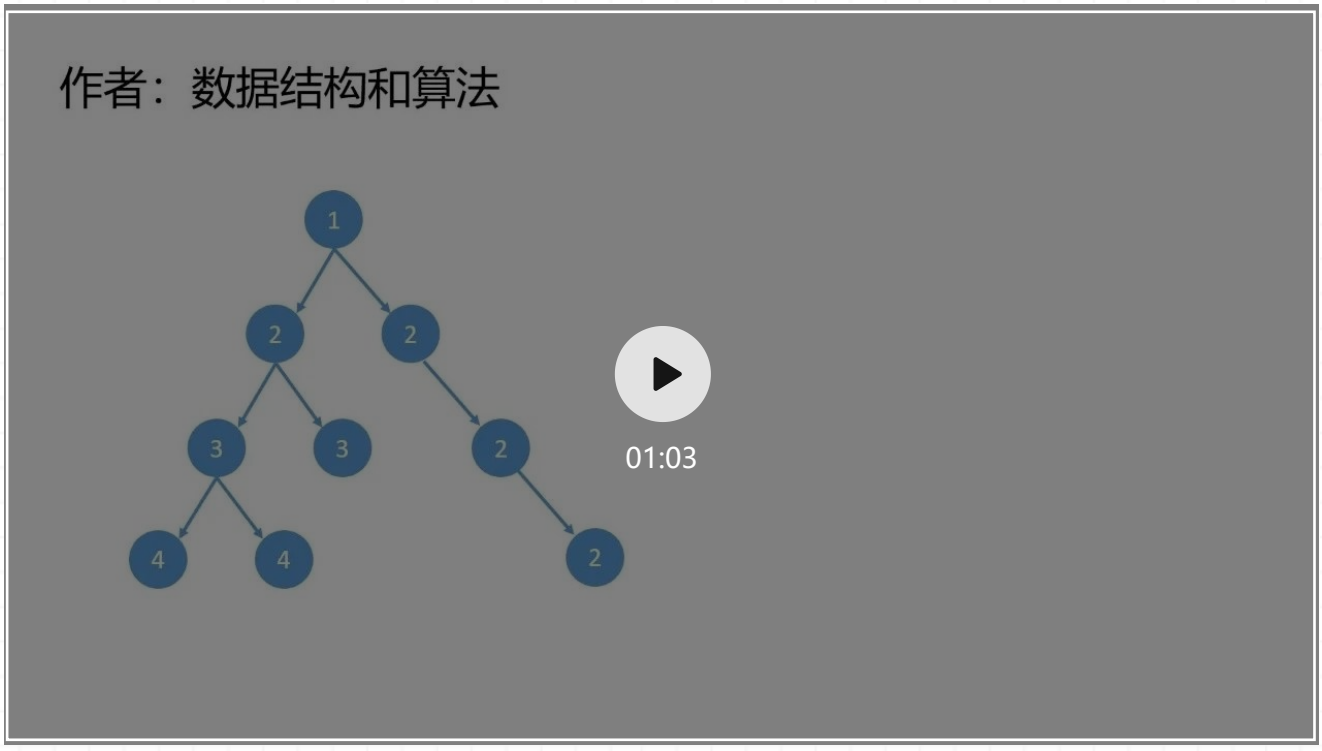
再来看下代码

```
1   public boolean isBalanced(TreeNode root) {
2     if (root == null)
3       return true;
4     //分别计算左子树和右子树的高度
5     int left = depth(root.left);
6     int right = depth(root.right);
7     //这两个子树的高度不能超过1，并且他的两个子树也必须是平衡二叉树
8     return Math.abs(left - right) <= 1 && isBalanced(root.left) && isBalanced(root.right);
9   }
10
11  //计算树中节点的高度
12  public int depth(TreeNode root) {
```

```
    if (root == null)
      return 0;
    return Math.max(depth(root.left), depth(root.right)) + 1;
  }
```

从下到上

上面的计算过程是从上往下判断的，其实我们还可以从下往上判断，就是从叶子节点开始往上，如果某一个子树不是平衡的就返回false，具体看视频



先判断两个子树是否是平衡的，然后再判断以当前节点为根节点的子树是否是平衡的……。来看下代码

```
//如果等于-1就表示不是平衡的
private static final int UNBALANCED = -1;

public boolean isBalanced(TreeNode root) {
  return helper(root) != UNBALANCED;
}

public int helper(TreeNode root) {
  if (root == null)
    return 0;

  //如果左子节点不是平衡二叉树，直接返回UNBALANCED
  int left = helper(root.left);
  if (left == UNBALANCED)
    return UNBALANCED;

  //如果右子节点不是平衡二叉树，直接返回UNBALANCED
  int right = helper(root.right);
  if (right == UNBALANCED)
    return UNBALANCED;

  //如果左右子节点都是平衡二叉树，但他们的高度相差大于1，
  //直接返回UNBALANCED
  if (Math.abs(left - right) > 1)
    return UNBALANCED;

  //否则就返回二叉树中节点的最大高度
```

```
28      return Math.max(left, right) + 1;
29  }
```

总结

关于二叉树的一些常用术语我们来总结一下：

节点的度：一个节点含有的子树的个数称为该节点的度；
树的度：一棵树中，最大的节点度称为树的度；
叶节点或终端节点：度为零的节点；
非终端节点或分支节点：度不为零的节点；
父亲节点或父节点：若一个节点含有子节点，则这个节点称为其子节点的父节点；
孩子节点或子节点：一个节点含有的子树的根节点称为该节点的子节点；
兄弟节点：具有相同父节点的节点互称为兄弟节点；
节点的层次：从根开始定义起，根为第1层，根的子节点为第2层，以此类推；
深度：对于任意节点n,n的深度为从根到n的唯一路径长，根的深度为0；
高度：对于任意节点n,n的高度为从n到一片树叶的最长路径长，所有树叶的高度为0；
堂兄弟节点：父节点在同一层的节点互为堂兄弟；
节点的祖先：从根到该节点所经分支上的所有节点；
子孙：以某节点为根的子树中任一节点都称为该节点的子孙。
森林：由m（m>=0）棵互不相交的树的集合称为森林；

往期推荐

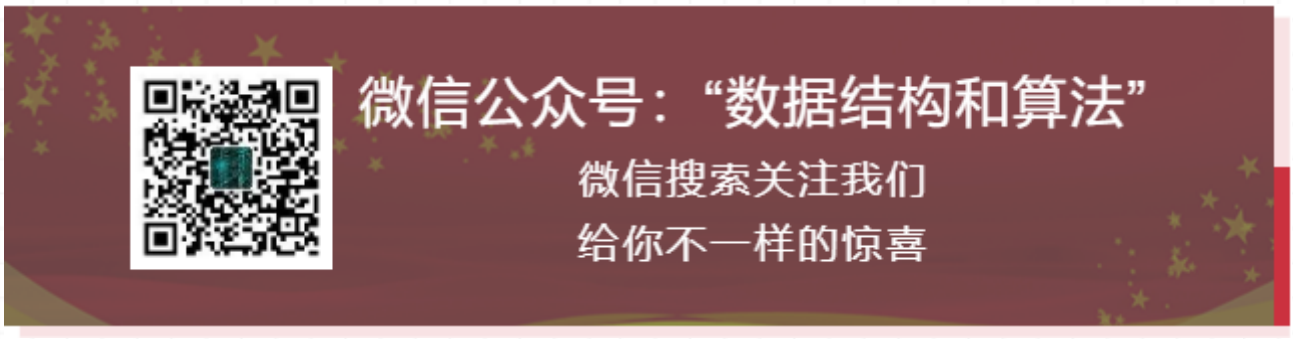
- 507，BFS和DFS解二叉树的层序遍历 II
- 503，二叉搜索树中的众数
- 485，递归和非递归两种方式解相同的树
- 470，DFS和BFS解合并二叉树

510，将有序数组转换为二叉搜索树

原创 山大王wld 数据结构和算法 1月21日

收录于话题
#算法图文分析

137个 >



Opportunity does not knock, it presents itself when you beat down the door.

机遇不会自己找上门来，它只会在你开门时出现。

问题描述

将一个按照升序排列的有序数组，转换为一棵高度平衡二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点的左右两个子树的高度差的绝对值不超过 1。

示例：

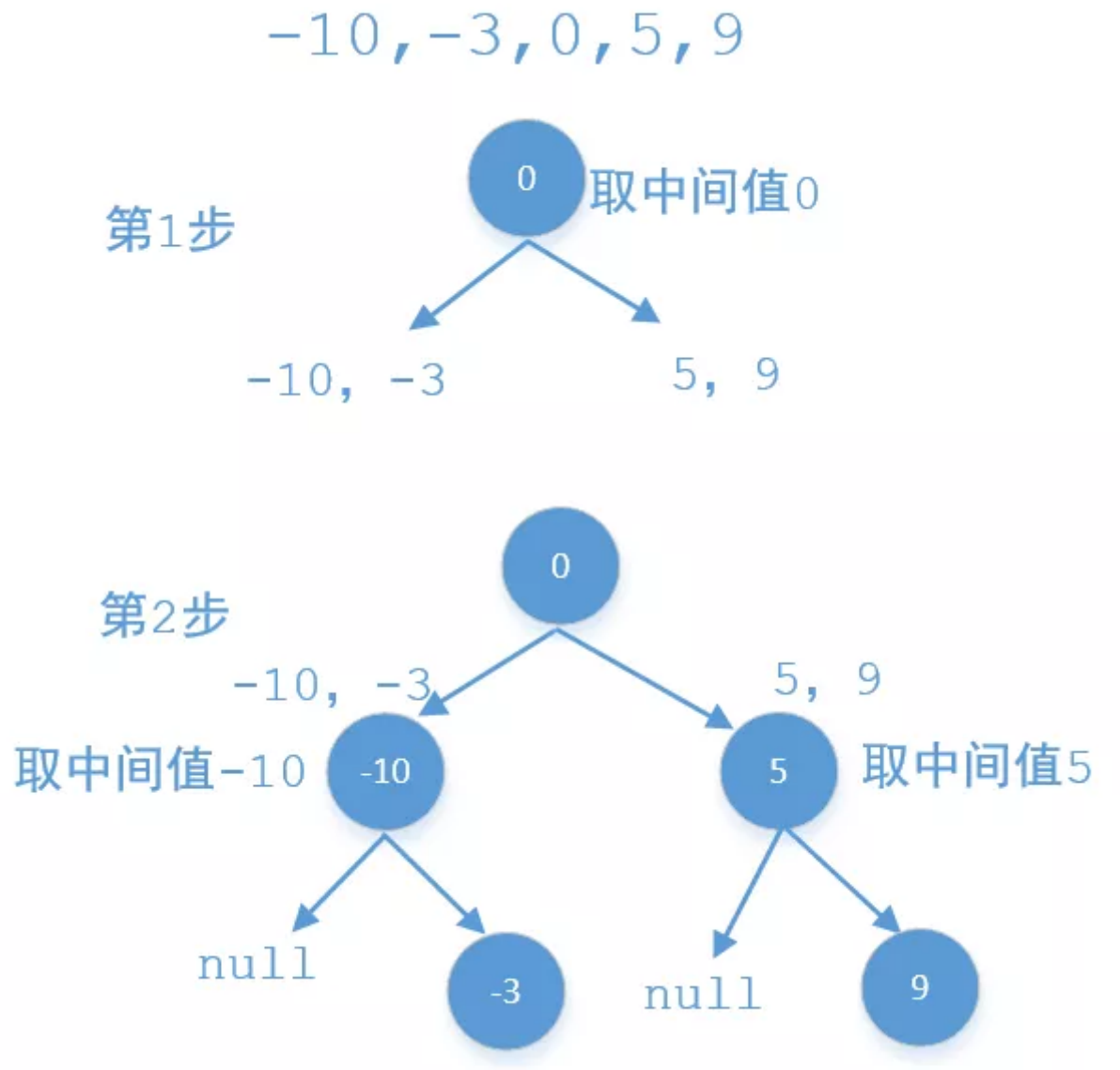
```
给定有序数组：[-10,-3,0,5,9],

一个可能的答案是：[0,-3,9,-10,null,5],
它可以表示下面这个高度平衡二叉搜索树：

      0
```

```
      / \
    -3   9
    /   /
  -10  5
```

递归方式解决

题中说了要转换为一棵高度平衡的二叉搜索树，并且数组又是排过序的，这就好办了。

我们可以使用递归的方式，每次取数组中间的值比如m作为当前节点，m前面的值都是比他小的，作为他左子树的结点值。m后面的值都是比他大的，作为他右子树的节点值，示例中一个可能的结果是。



代码如下

```
public TreeNode sortedArrayToBST(int[] num) {
  //边界条件判断
  if (num.length == 0)
    return null;
  return sortedArrayToBST(num, 0, num.length - 1);
```

```
}

//start表示数组开始的位置，end表示的结束的位置
public TreeNode sortedArrayToBST(int[] num, int start, int end) {
    if (start > end)
        return null;
    int mid = (start + end) >> 1;
    //取中间值作为当前节点
    TreeNode root = new TreeNode(num[mid]);
    //然后递归的方式，中间值之前的是当前节点左子树的所有节点
    root.left = sortedArrayToBST(num, start, mid - 1);
    //中间值之后的是当前节点的右子树的所有节点
    root.right = sortedArrayToBST(num, mid + 1, end);
    return root;
}
```

总结

之前讲过466. 使用快慢指针把有序链表转换二叉搜索树，和这题非常类似，只不过第466题是链表，这里改为了数组，更简单了。

往期推荐

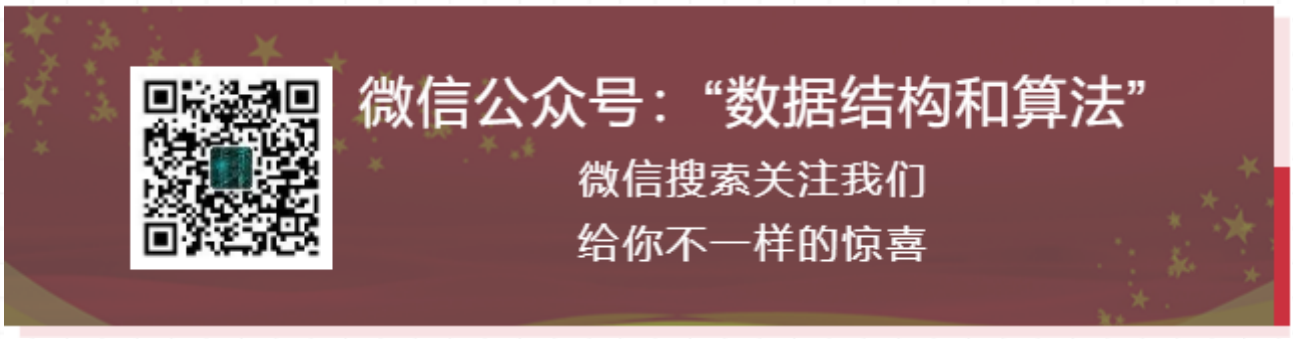
- 488，二叉树的Morris中序和前序遍历
- 483，完全二叉树的节点个数
- 464. BFS和DFS解二叉树的所有路径
- 399，从前序与中序遍历序列构造二叉树

503，二叉搜索树中的众数

原创 山大王wld 数据结构和算法 1月7日

收录于话题
#算法图文分析

137个 >



What drains your spirit drains your body. What fuels your spirit fuels your body.

消耗你心灵的事物也会消耗你的健康，滋补你心灵的事物也会滋补你的身体。

问题描述

给定一个有相同值的二叉搜索树（BST），找出 BST 中的所有众数（出现频率最高的元素）。

假定 BST 有如下定义：

- 结点左子树中所含结点的值小于等于当前结点的值
- 结点右子树中所含结点的值大于等于当前结点的值
- 左子树和右子树都是二叉搜索树

例如：
给定 BST [1,null,2,2],

```
1    1
```

```
    \
     2
    /
   2
```

返回[2].

提示：如果众数超过1个，不需考虑输出顺序

二叉树的中序遍历解决

这题是让求二叉搜索树中的众数，也就是出现次数最多的那个数。如果是在一个排序的数组中找众数，这个可能就比较简单，但这题是让在二叉搜索树中查找。

我们都知道**二叉搜索树的中序遍历是有序的**，有一种方式就是先把二叉搜索树中序遍历的结果存放到一个数组中，因为这个数组是升序的（虽然有重复的），我们可以遍历这个数组，这里使用几个变量。

使用变量curent表示当前的值，count表示当前值的数量，maxCount表示重复数字最大的数量。list集合存放结果。

1，如果当前节点的值等于curent，count就加1，

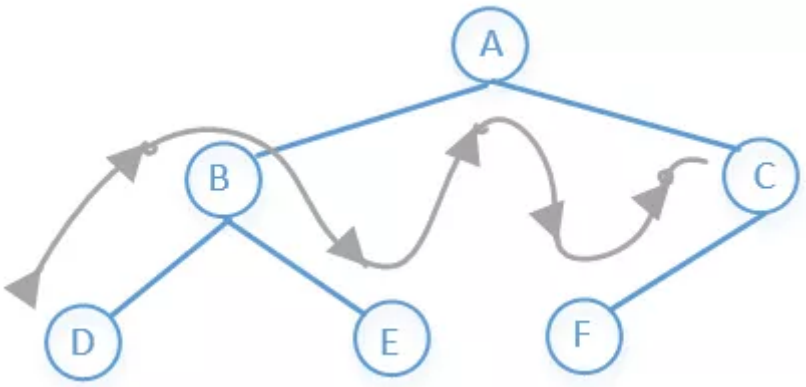
2，如果当前节点的值不等于current，说明遇到了下一个新的值，更新current为新的值，然后让count=1；

接着比较count和maxCount的大小，

- 如果count==maxCount，就把当前节点的值加入到集合list中。
- 如果count>maxCount，说明遇到重复次数最高的数了，这里先把list集合清空，然后再把当前节点的值加入到集合list中，最后在更新maxCount的值。

如果先把树的节点找出来在计算有点麻烦，我们可以在遍历树的节点的时候就判断，这样就会好一些，下面使用两种方式，一种是递归的，一种是非递归的。树的中序遍历顺序如下

左子节点→当前节点→右子节点



二叉树的中序遍历代码

```java
public void inOrderTraversal(TreeNode node) {
  if (node == null)
    return;
  inOrderTraversal(node.left);
  System.out.println(node.val);
  inOrderTraversal(node.right);
}
```

我们来对他进行改造一下

```java
List<Integer> mList = new ArrayList<>();
int curent = 0;//表示当前节点的值
int count = 0;//和当前节点值相同的节点数量
int maxCount = 0;//最大的重复数量

public int[] findMode(TreeNode root) {
  inOrderTraversal(root);
  int[] res = new int[mList.size()];
  //把集合list转化为数组
  for (int i = 0; i < mList.size(); i++) {
    res[i] = mList.get(i);
  }
  return res;
}

//递归方式
public void inOrderTraversal(TreeNode node) {
  //终止条件判断
  if (node == null)
    return;
  //遍历左子树
  inOrderTraversal(node.left);

  //下面是对当前节点的一些逻辑操作
  int nodeValue = node.val;
  if (nodeValue == curent) {
    //如果节点值等于curent，count就加1
    count++;
  } else {
    //否则，就表示遇到了一个新的值，curent和count都要
    //重新赋值
    curent = nodeValue;
    count = 1;
  }
  if (count == maxCount) {
    //如果count == maxCount，就把当前节点加入到集合中
    mList.add(nodeValue);
  } else if (count > maxCount) {
    //否则，当前节点的值重复量是最多的，直接把list清空，然后
    //把当前节点的值加入到集合中
    mList.clear();
    mList.add(nodeValue);
    maxCount = count;
  }
```

```
45
46      //遍历右子树
47      inOrderTraversal(node.right);
48    }
```

之前在讲到[373，数据结构-6,树]的时候，提到二叉树中序遍历的非递归写法

```
1   public void inOrderTraversal(TreeNode tree) {
2     Stack<TreeNode> stack = new Stack<>();
3     while (tree != null || !stack.isEmpty()) {
4       while (tree != null) {
5         stack.push(tree);
6         tree = tree.left;
7       }
8       if (!stack.isEmpty()) {
9         tree = stack.pop();
10        System.out.println(tree.val);
11        tree = tree.right;
12      }
13    }
14  }
```

再来改造一下

```
1   List<Integer> mList = new ArrayList<>();
2   int curent = 0;
3   int count = 0;
4   int maxCount = 0;
5
6   public int[] findMode(TreeNode root) {
7     inOrderTraversal(root);
8     int[] res = new int[mList.size()];
9     //把集合list转化为数组
10    for (int i = 0; i < mList.size(); i++) {
11      res[i] = mList.get(i);
12    }
13    return res;
14  }
15
16  //非递归方式
17  public void inOrderTraversal(TreeNode tree) {
18    Stack<TreeNode> stack = new Stack<>();
19    while (tree != null || !stack.isEmpty()) {
20      while (tree != null) {
21        stack.push(tree);
22        tree = tree.left;
23      }
24      if (!stack.isEmpty()) {
25        tree = stack.pop();
26        int nodeValue = tree.val;
27        if (nodeValue == curent) {
28          //如果节点值等于curent，count就加1
29          count++;
30        } else {
31          //否则，就表示遇到了一个新的值，curent和count都要
32          //重新赋值
33          curent = nodeValue;
34          count = 1;
35        }
36        if (count == maxCount) {
37          //如果count == maxCount，就把当前节点加入到集合中
38          mList.add(nodeValue);
39        } else if (count > maxCount) {
40          //否则，当前节点的值重复量是最多的，直接把list清空，然后
41          //把当前节点的值加入到集合中
42          mList.clear();
43          mList.add(nodeValue);
44          maxCount = count;
```

```
            }
            tree = tree.right;
        }
    }
}
```

总结

做这题首先要搞懂二叉搜索树的中序遍历结果是排序的，这题就容易解了。除了常规的二叉树的中序遍历，我们还可以使用二叉树的Morris中序遍历，具体可以看下488，二叉树的Morris中序和前序遍历。

往期推荐

- 488，二叉树的Morris中序和前序遍历
- 483，完全二叉树的节点个数
- 464. BFS和DFS解二叉树的所有路径
- 456，解二叉树的右视图的两种方式

488，二叉树的Morris中序和前序遍历

原创 山大王wld 数据结构和算法 2020-12-09

收录于话题
#算法图文分析
111个 >



Never wasting an hour, never letting one moment go cold.

不要虚度光阴，即使是一瞬，也要用力把握。

问题描述

之前介绍过二叉树的前中后，以及DFS和BFS等遍历方式，并且每种都写了递归和非递归的解法。有的说二叉树的前中后遍历方式都属于DFS的一种，其实也可以这样理解。关于这几种遍历方式具体可以看下373，数据结构-6,树。

对于二叉树的遍历除了前面介绍的几种常见的以外，还有Morris的前中后3种遍历方式。前面讲的二叉树的几种遍历方式，如果使用的是非递归方式，我们要么需要一个栈，要么需要一个队列来维护节点之间的关系。如果使用Morris遍历就不需要了，他的实现过程其实就是把叶子节点的指针给利用起来，因为一般情况下，二叉树的叶子节点是没有子节点的，也就是说他们是指向空，这样总感觉有点浪费，Morris的实现原理其实就是把叶子节点的指针给利用了起来。Morris的后续遍历方式稍微有点复杂，这个以后再讲，这里主要看一下Morris的前序和中序遍历方式。

Morris的中序遍历

对于Morris的中序遍历可以把它看做是把二叉树拉直变成了链表，我们先来看一下他的实现步骤：
记当前节点为cur，从根节点开始遍历。

1，判断cur是否为空，如果为空就停止遍历。

2，如果cur不为空

1）如果cur没有左子节点，打印cur的值，然后让cur指向他的右子节点，即cur=cur.right

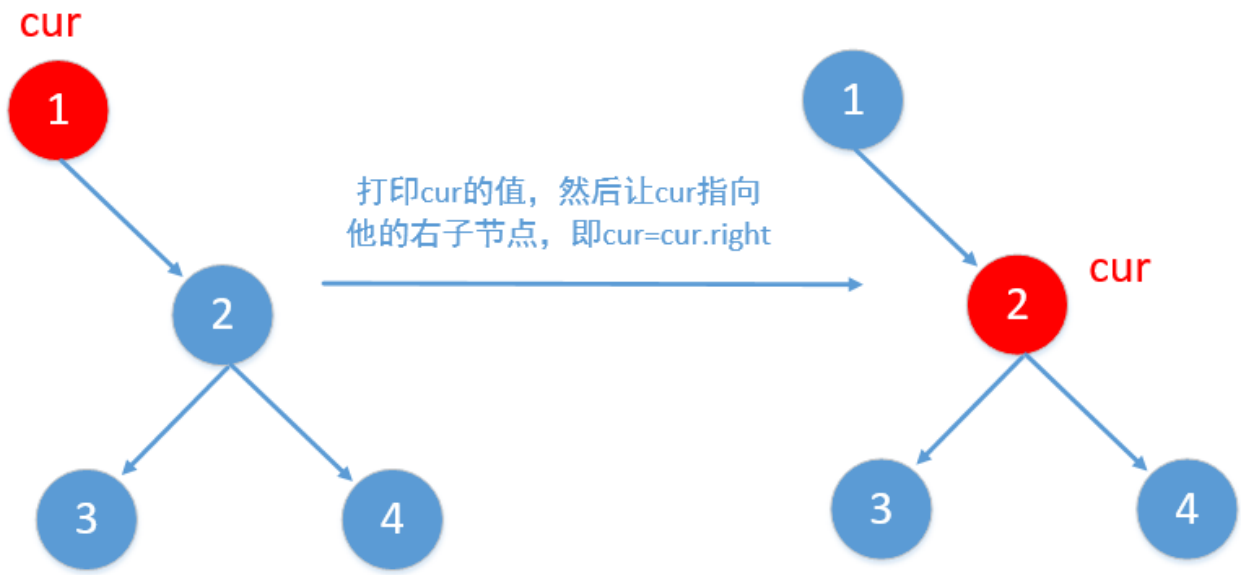
2）如果cur有左子节点，则从左子节点中找到最右边的结点pre。

1))如果pre的右子节点为空，就让pre的右子节点指向cur，即pre.right=cur，然后cur指向他的左子节点，即cur=cur.left。

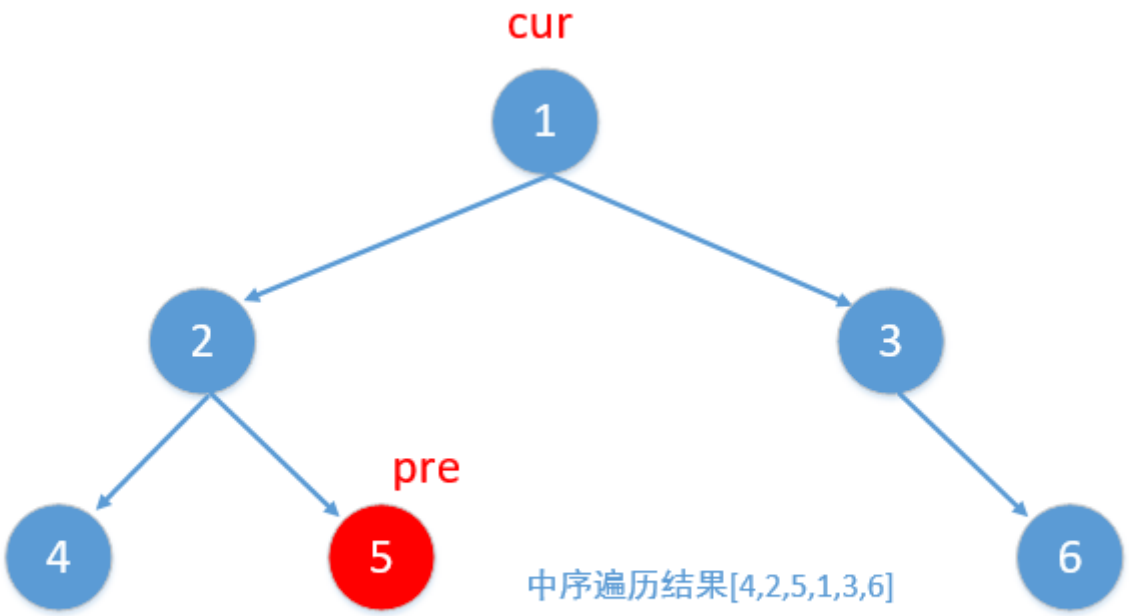
2))如果pre的右子节点不为空，就让他指向空，即pre.right=null，然后输出cur节点的值，并将节点cur指向其右子节点，即cur=cur.right。

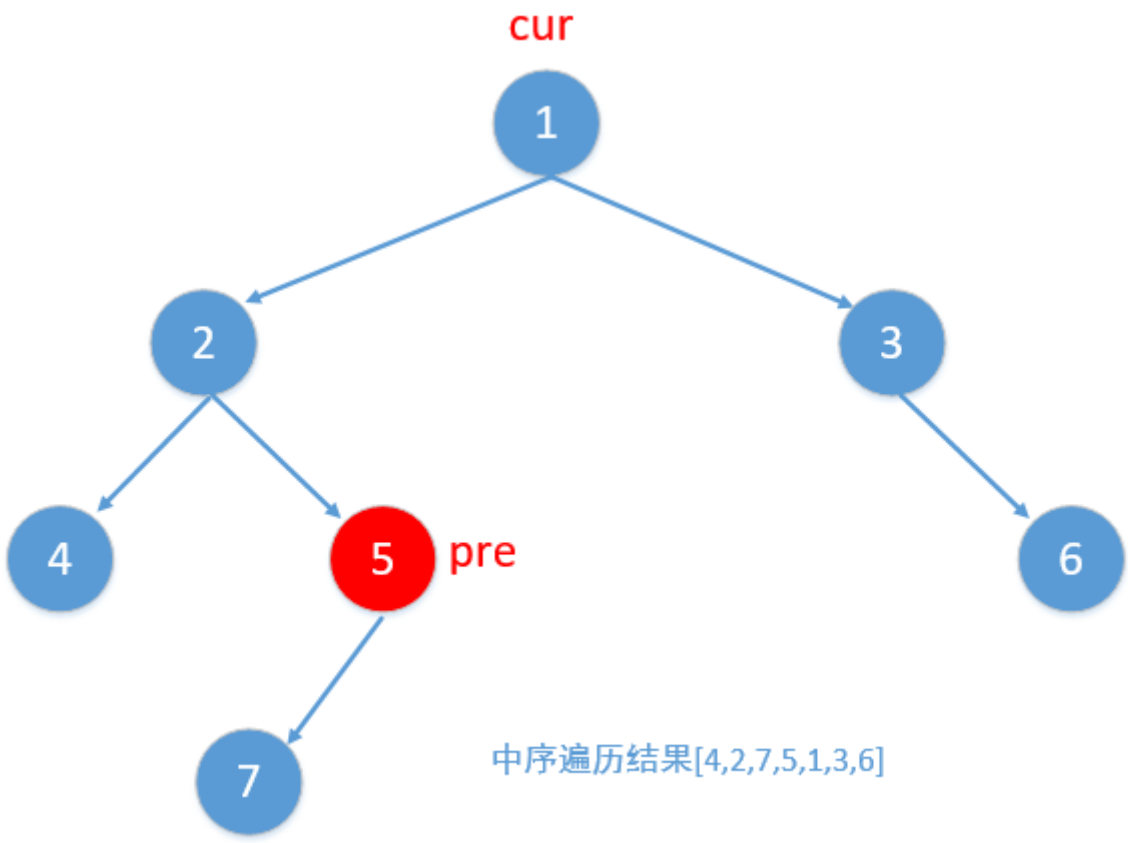
3，重复步骤2，直到节点cur为空为止。

上面叙述了一大堆，懂的一眼就能看明白，不懂的肯定会绕晕。我来一条条解释一下。
1，首先第一条，cur为空就停止遍历，这没什么好说的，为空了当然就没法遍历了。
2，cur不为空，cur的左子节点为空，打印当前节点cur的值，然后让cur=cur.right，我们来画个图看一下

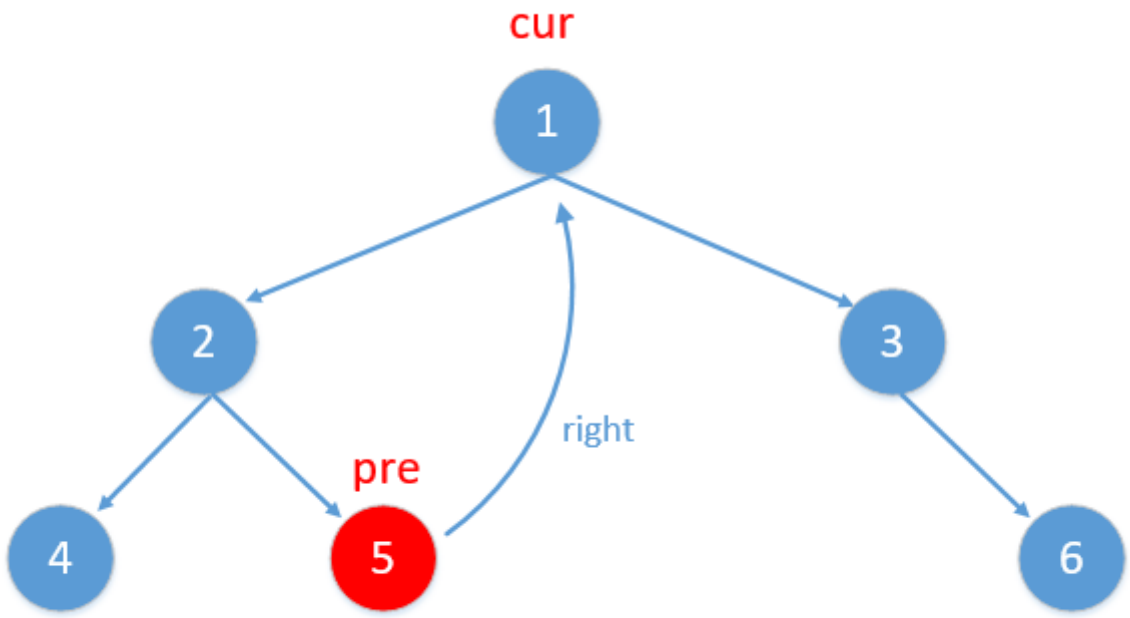


2，cur不为空，他的左子节点也不为空，我们要找到他左子节点中的最右子节点pre，其实也就是中序遍历中cur的前一个节点，我们随便画两棵树来看一下，很明显就是要找到当前节点cur在中序遍历的前一个节点。

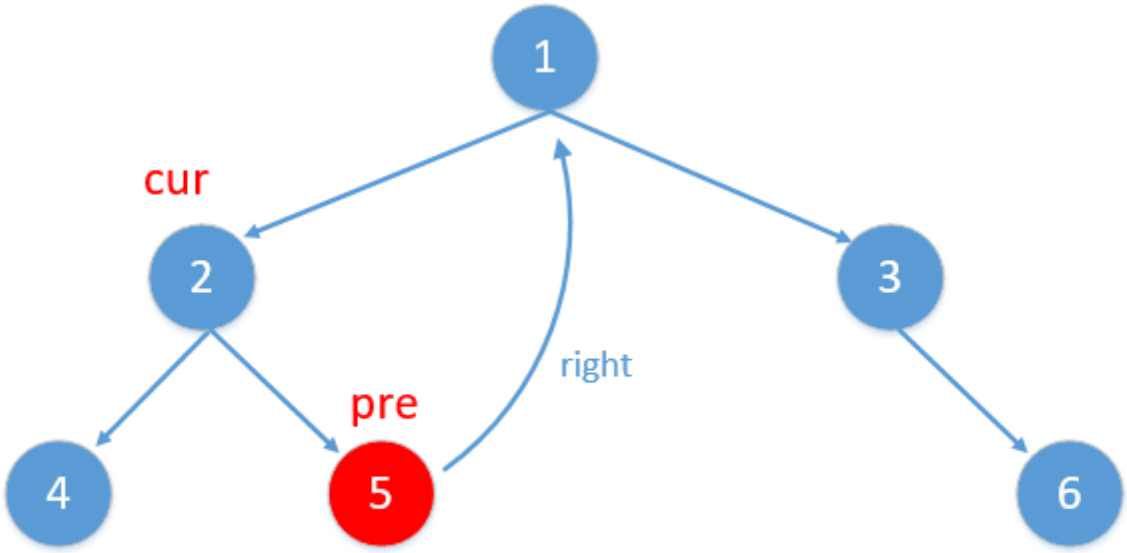




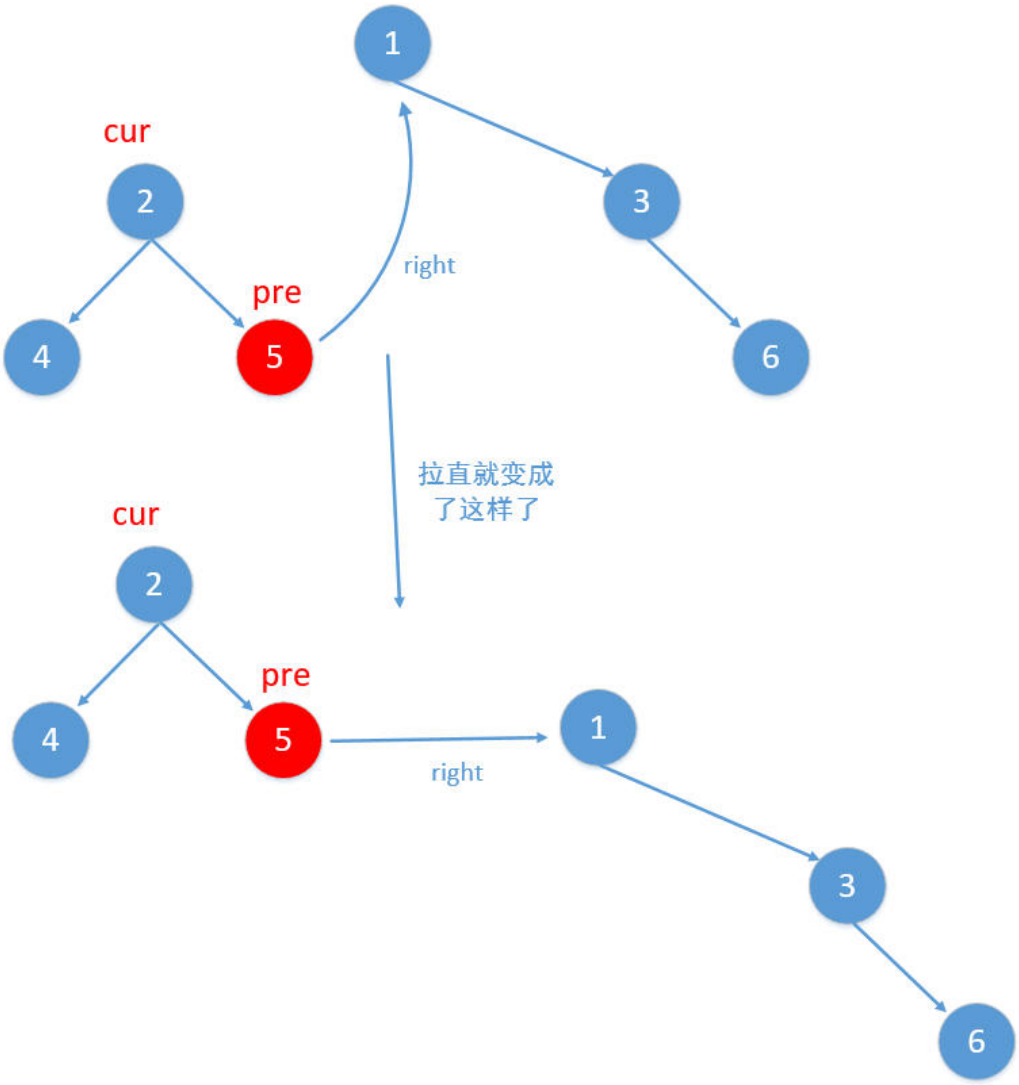
然后判断pre的右子节点是否为空，这一步可能是大家最疑惑的地方，这还需要判断吗，其实是需要的，第一次查找的时候，pre的右子节点肯定是为空的，我们让他指向cur即可，也就是pre.right=cur。如下图所示



然后再让cur指向他的左子节点，如下图所示



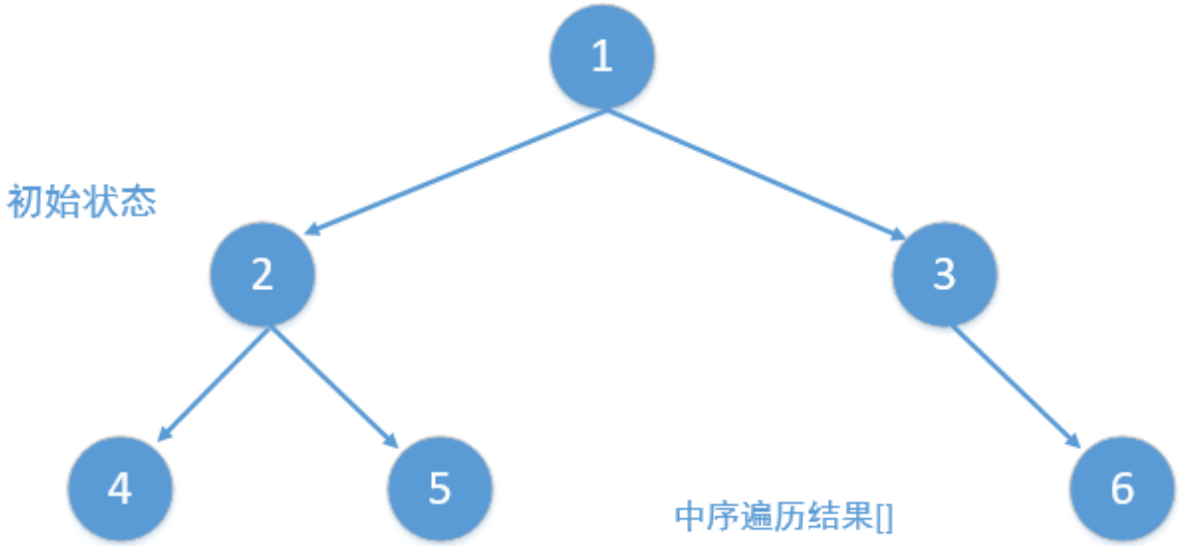
虽然节点1和他的左子节点没有断开，但我们可以认为他是断开了（实际上没有断开），我们可以把它想象成这样

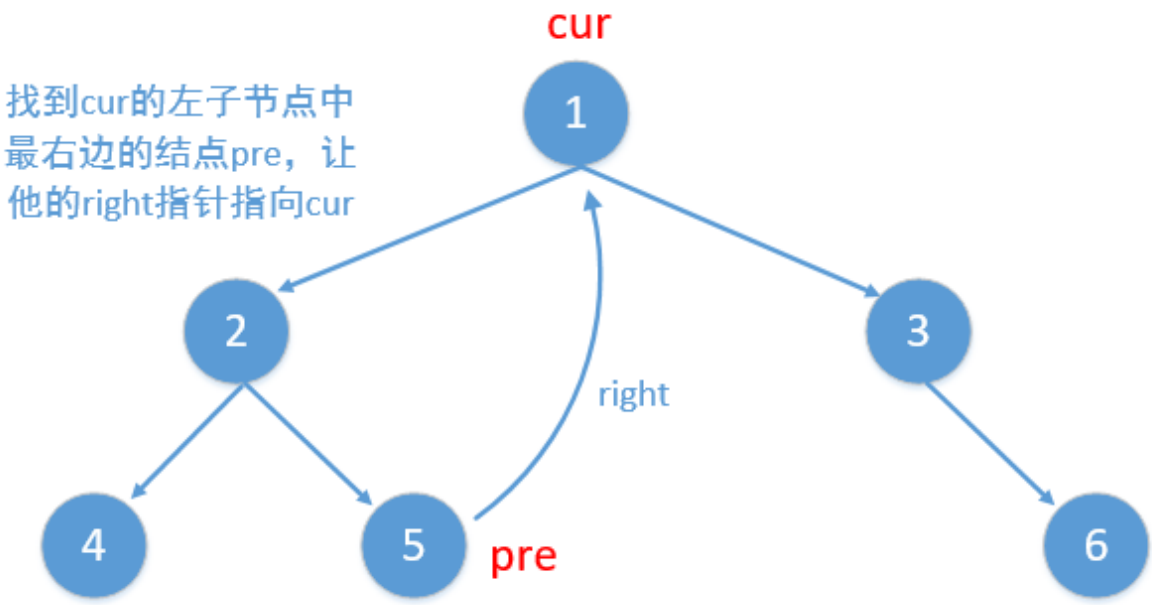


最终我们可以把它想象成转化为一个链表（实际上并没有）。

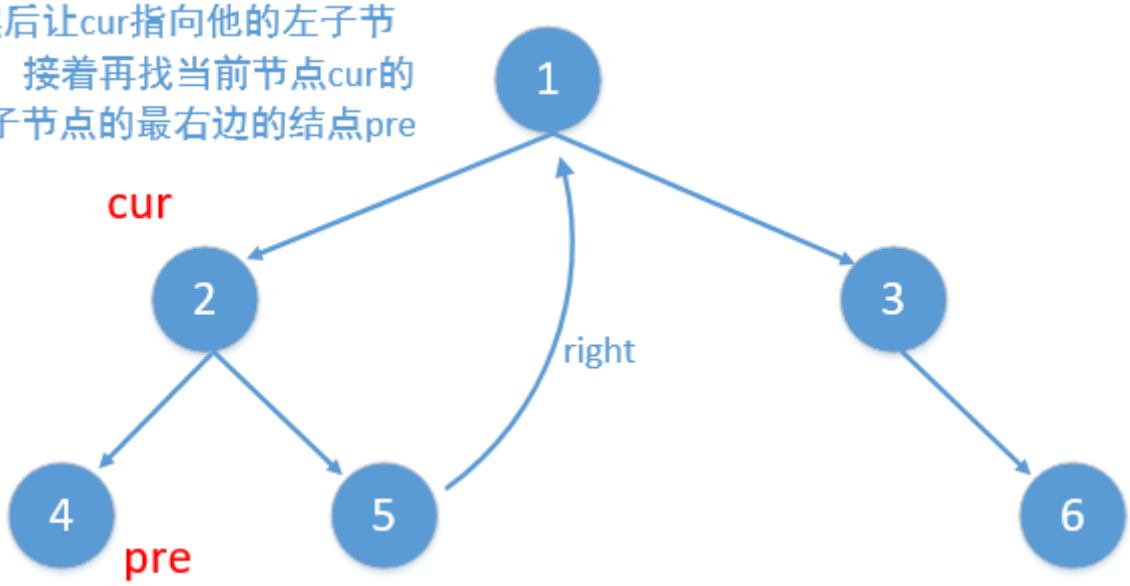
如果pre的右子节点不为空，那么他肯定是指向cur节点的，也就表示cur节点的左子节点都已经遍历完了，只需要打印当前节点cur的值，然后让cur指向右子节点，以同样的操作开始遍历右子节点。

文字描述比较绕，我们就随便举个例子画个图来看下

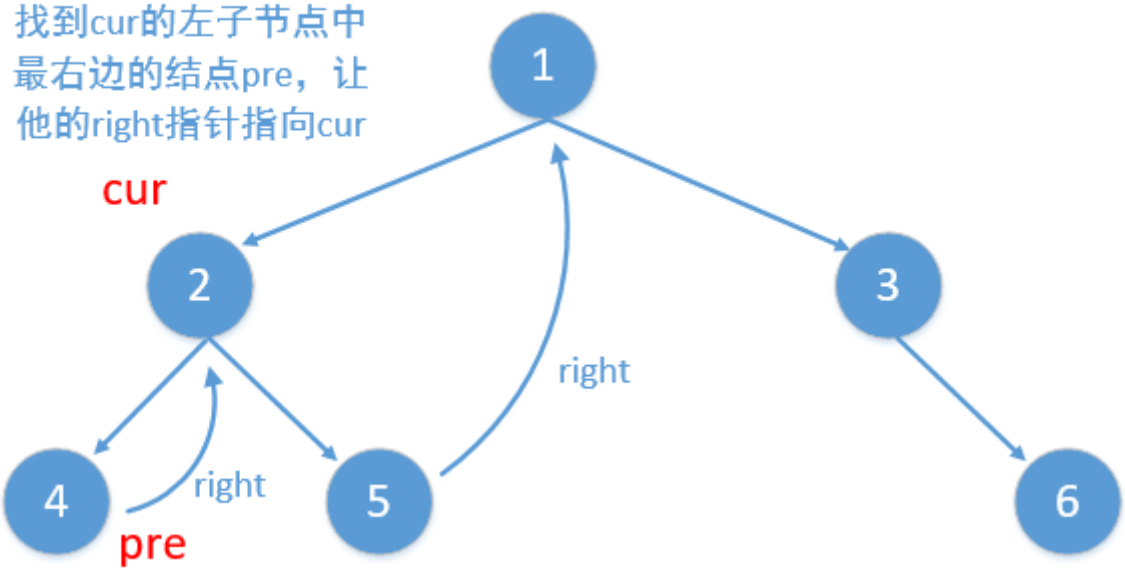

初始状态
1
2
3
4
5
6
中序遍历结果[]

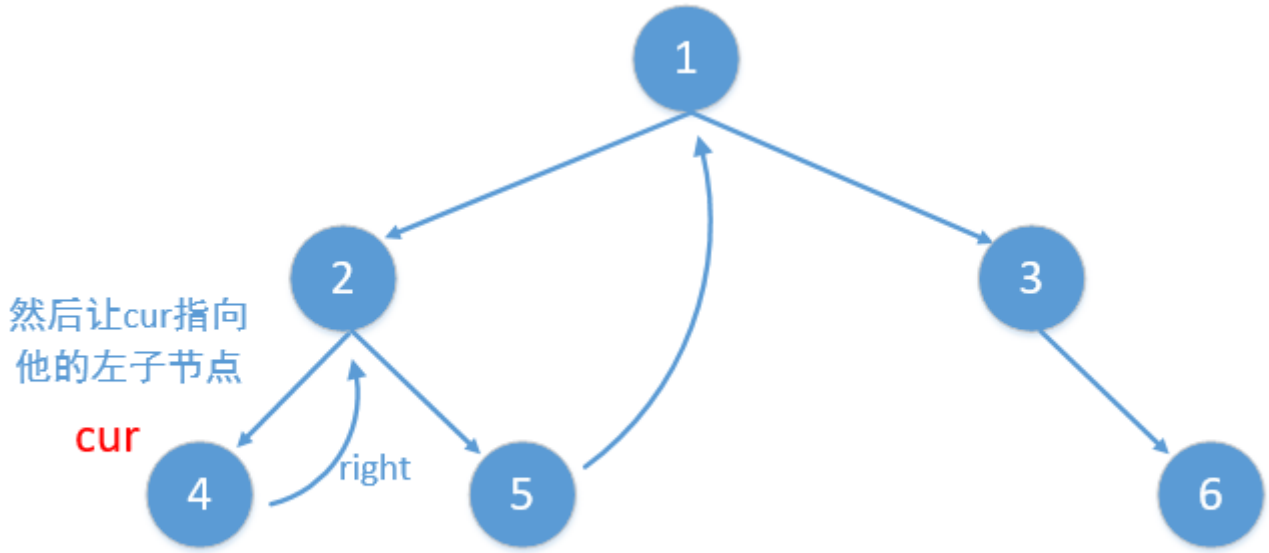

找到cur的左子节点中
最右边的结点pre，让
他的right指针指向cur
cur
1
2
3
right
4
5
pre
6

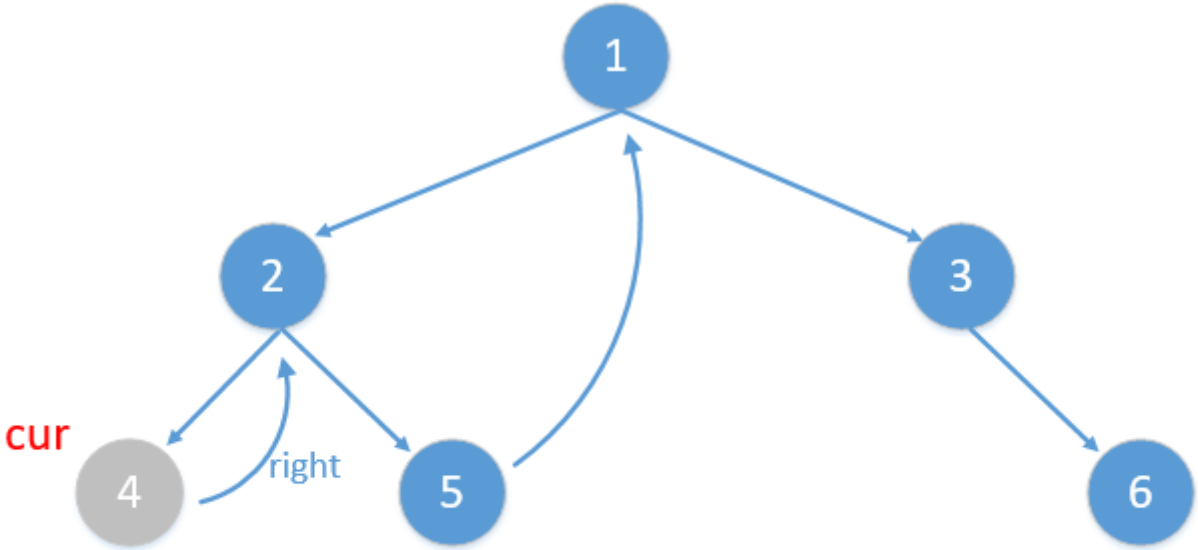
然后让cur指向他的左子节点，接着再找当前节点cur的左子节点的最右边的结点pre



找到cur的左子节点中最右边的结点pre，让他的right指针指向cur

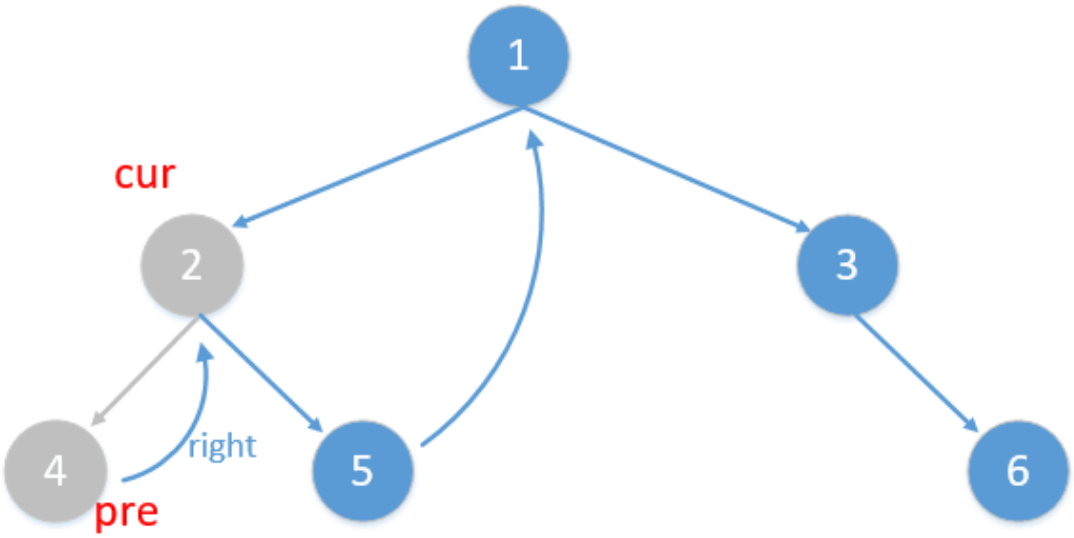






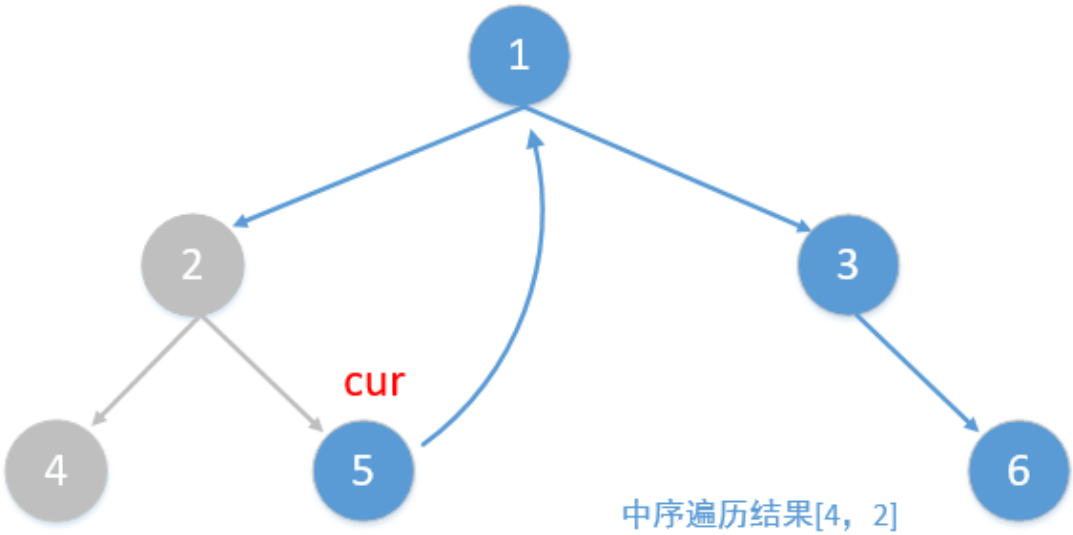
由于cur的左子节点为空，不能
再找了，直接输出cur的值4，
然后让他指向他的右子节点2，

中序遍历结果[4]

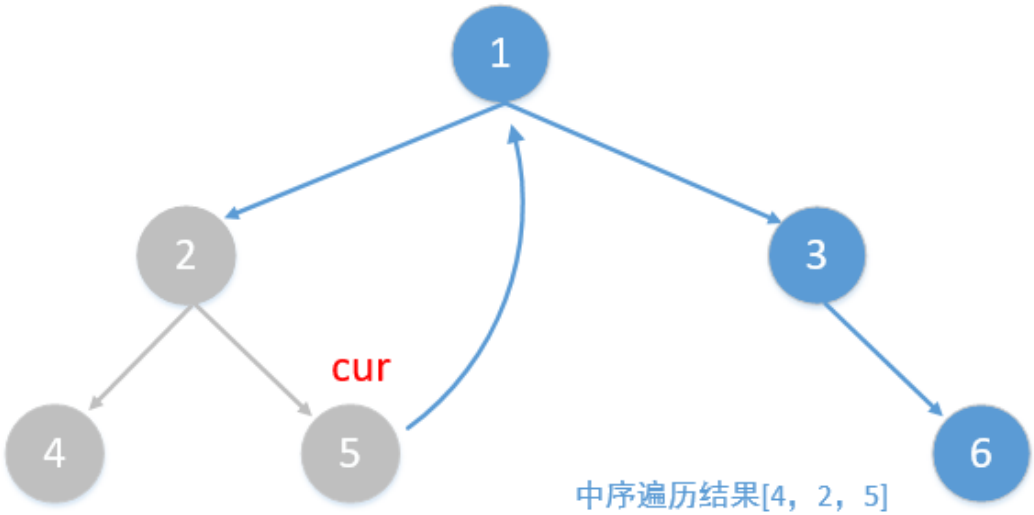


接着我们再找cur左子节点的最右节点pre，结果发现这个pre的右指针指向cur，说明cur的左子节点都已经遍历完了，我们还需要把这棵树给还原，让pre.right=null，然后打印cur的值2，接着让cur指向他的右子节点。如下图所示

中序遍历结果[4，2]

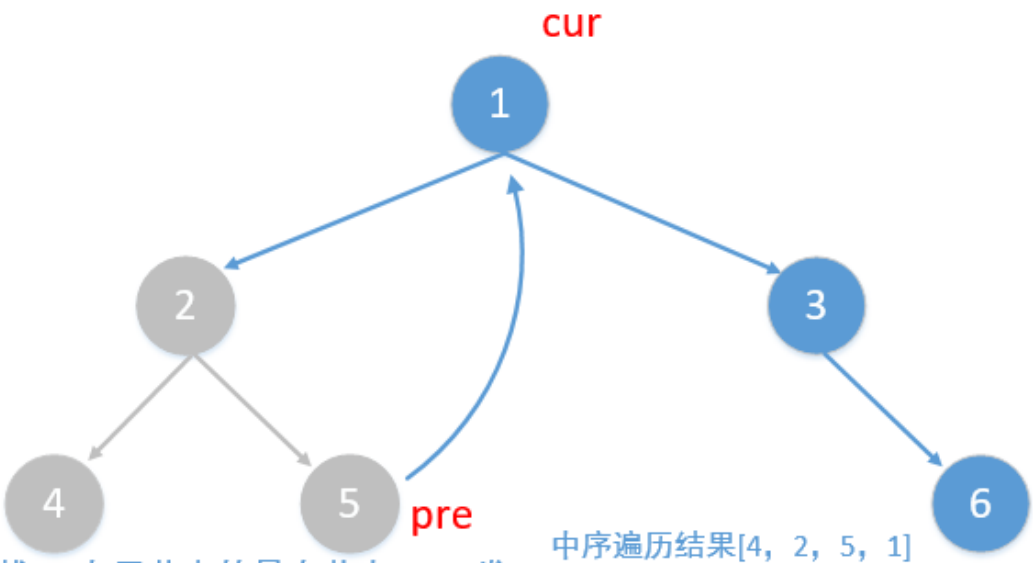


中序遍历结果[4，2]



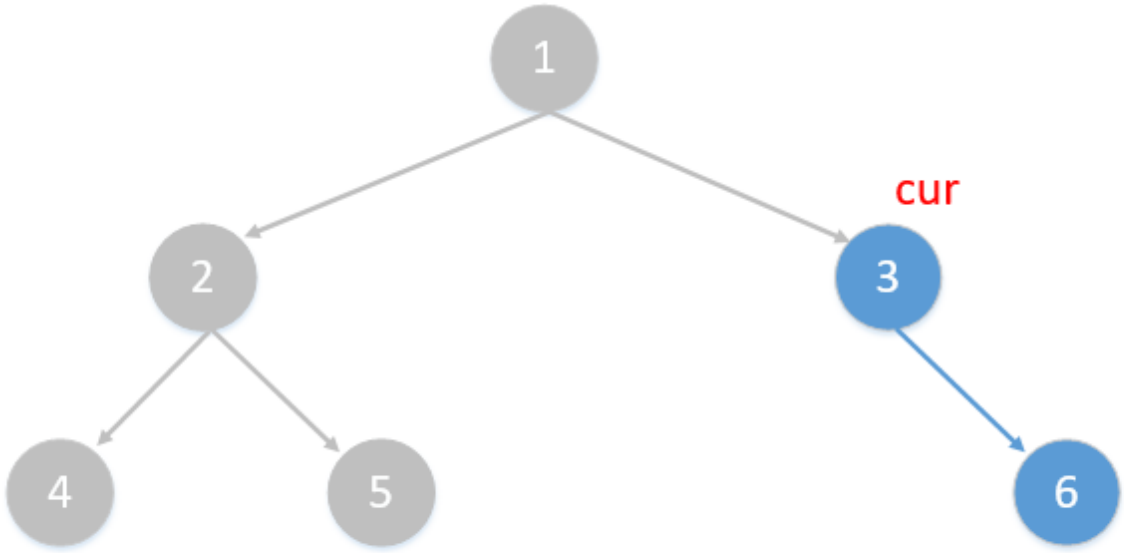
中序遍历结果[4，2，5]

同样，cur没有左子节点，然后打印cur的值，接着再让他指向他的右子节点



中序遍历结果[4，2，5，1]

然后再找cur左子节点的最右节点pre，发现他的right指针指向cur，说明节点cur的左子节点都遍历完了，要让pre的右指针指向空，是为了把树还原，然后打印cur的值，接着让cur指向他的右子节点。



然后右子节点遍历的方式也是同样的原理，这里图就不在往下画了

.....

搞懂了上面的原理，代码就简单多了

```java
public List<Integer> inorderTraversal(TreeNode root) {
  List<Integer> res = new ArrayList<>();
  //首先把根节点赋值给cur
  TreeNode cur = root;
  //如果cur不为空就继续遍历
  while (cur != null) {
    if (cur.left == null) {
      //如果当前节点cur的左子节点为空，就访问当前节点cur，
      //接着让当前节点cur指向他的右子节点
      res.add(cur.val);
      cur = cur.right;
    } else {
      TreeNode pre = cur.left;
      //查找pre节点，注意这里有个判断就是pre的右子节点不能等于cur
      while (pre.right != null && pre.right != cur)
        pre = pre.right;
      //如果pre节点的右指针指向空，我们就让他指向当前节点cur，
      //然后当前节点cur指向他的左子节点
      if (pre.right == null) {
        pre.right = cur;
        cur = cur.left;
      } else {
        //如果pre节点的右指针不为空，那么他肯定是指向cur的，
        //表示cur的左子节点都遍历完了，我们需要让pre的右
        //指针指向null，目的是把树给还原，然后再访问当前节点
        //cur，最后再让当前节点cur指向他的右子节点。
        pre.right = null;
        res.add(cur.val);
        cur = cur.right;
      }
    }
  }
  return res;
}
```

Morris的前序遍历

前序遍历和中序遍历的方式是一样的，只不过访问节点的时机不一样。图就不在画了，代码中有注释，可以看下

```java
public List<Integer> preorderTraversal(TreeNode root) {
    List<Integer> res = new ArrayList<>();
    //首先把根节点赋值给cur
    TreeNode cur = root;
    //如果cur不为空就继续遍历
    while (cur != null) {
        if (cur.left == null) {
            //如果当前节点cur的左子节点为空，就访问当前节点cur，
            //接着让当前节点cur指向他的右子节点
            res.add(cur.val);
            cur = cur.right;
        } else {
            TreeNode pre = cur.left;
            //查找pre节点，注意这里有个判断就是pre的右子节点不能等于cur
            while (pre.right != null && pre.right != cur)
                pre = pre.right;
            //如果pre节点的右指针指向空，我们就让他指向当前节点cur，
            //然后打印当前节点cur的值，最后再让当前节点cur指向他的左子节点
            if (pre.right == null) {
                pre.right = cur;
                res.add(cur.val);
                cur = cur.left;
            } else {
                //如果pre节点的右指针不为空，那么他肯定是指向cur的，
                //表示当前节点cur和他的的左子节点都遍历完了，直接
                //让他指向他的右子节点即可。
                pre.right = null;
                cur = cur.right;
            }
        }
    }
    return res;
}
```

总结

上面就是Morris的中序和前序遍历，关于后续遍历有一点复杂，后面单独有一篇文章来讲。

485，递归和非递归两种方式解相同的树

原创 山大王wld 数据结构和算法 今天

收录于话题
#算法图文分析

95个 >



It takes more than intelligence to act intelligently.

头脑聪明不代表就能明智地行事。

问题描述

给定两个二叉树，编写一个函数来检验它们是否相同。

如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

示例 1:

```
输入:       1          1
           / \        / \
          2   3      2   3

         [1,2,3],   [1,2,3]

输出: true
```

示例 2:

```
输入:       1          1
           /            \
          2              2

         [1,2],     [1,null,2]

输出: false
```

示例 3:

```
输入:        1          1
            / \        / \
           2   1      1   2

         [1,2,1],   [1,1,2]

输出: false
```

递归解法

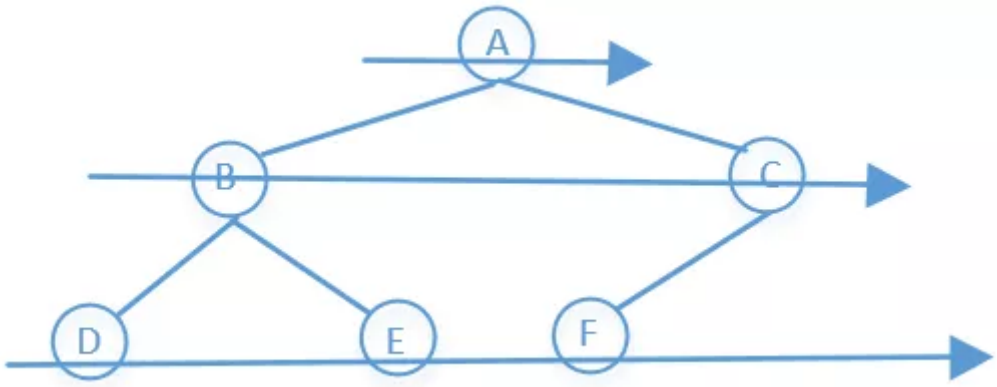
这题是让判断两棵树是否相同，最简单的方式就是使用递归。先判断根节点是否相同，如果相同再分别判断左右子节点是否相同，判断的过程中只要有一个不相同就返回false，如果全部相同才会返回true。

来看下代码

```
public boolean isSameTree(TreeNode p, TreeNode q) {
  //如果都为空我们就认为他是相同的
  if (p == null && q == null)
    return true;
  //如果一个为空，一个不为空，很明显不可能是相同的树，直接返回false即可
  if (p == null || q == null)
    return false;
  //如果这两个节点都不为空并且又不相等，所以他也不可能是相同的树，直接返回false
  if (p.val != q.val)
    return false;
  //走到这一步说明节点p和q是完全相同的，我们只需要在比较他们的左右子节点即可
  return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
}
```

非递归解决

非递归解决可以参照树的BFS遍历，也就是宽度优先搜索，也称广度优先搜索。关于树的BFS遍历可以参照373，数据结构-6,树，他是一层一层遍历的，如下图所示



二叉树的BFS代码如下

```
public void bfsPrint(TreeNode tree) {
  if (tree == null)
    return;
  Queue<TreeNode> queue = new LinkedList<>();
  queue.add(tree);//相当于把数据加入到队列尾部
  while (!queue.isEmpty()) {
    //poll方法相当于移除队列头部的元素
    TreeNode node = queue.poll();
    System.out.println(node.val);
    if (node.left != null)
      queue.add(node.left);
    if (node.right != null)
      queue.add(node.right);
  }
}
```

他使用的是一个队列，把每层的结点不停的放入到队列中，然后再出队，再把子节点放入到队列中……

对于这道题我们可以使用两个队列，一个是存放第一棵树节点的队列，一个是存放第二棵树节点的队列。

1，每次这两个队列的元素同时出队，出队之后都要判断出队的这两个节点值是否相同，如果不相同直接返回false，如果相同再往下走。
2，然后再判断他们的左子节点是否都存在，如果一个存在一个不存在就直接返回false，如果都存在就加入到队列中
3，右子节点同左子节点
4，最后再判断这两个队列是否都为空

原理比较简单，看下代码

```
public boolean isSameTree(TreeNode p, TreeNode q) {
  //如果都为空我们就认为他们是相同的
  if (p == null && q == null)
    return true;
  //如果一个为空，一个不为空，很明显不可能是相同的树，直接返回false即可
  if (p == null || q == null)
    return false;
  Queue<TreeNode> queueP = new LinkedList<>();
  Queue<TreeNode> queueQ = new LinkedList<>();
  //如果p和q两个节点都不为空，就把他们加入到队列中
  queueP.add(p);
  queueQ.add(q);
```

```
    while (!queueP.isEmpty() && !queueQ.isEmpty()) {
      //分别出队
      TreeNode tempP = queueP.poll();
      TreeNode tempQ = queueQ.poll();
      //如果这两个节点的值不相同，直接返回false
      if (tempP.val != tempQ.val)
        return false;

      //如果对应的左子节点不为空就加入到队列中
      if (tempP.left != null)
        queueP.add(tempP.left);
      if (tempQ.left != null)
        queueQ.add(tempQ.left);
      //注意这里没有直接判断两个左子节点是否一个为空一个
      //不为空，而是通过队列的长度来判断的，只有两个左子节点
      //都为空或者都不为空的时候，队列长度才会一样
      if (queueP.size() != queueQ.size())
        return false;

      //右子节点同上
      if (tempP.right != null)
        queueP.add(tempP.right);
      if (tempQ.right != null)
        queueQ.add(tempQ.right);
      if (queueP.size() != queueQ.size())
        return false;
    }
    //最后再判断这两个队列是否都为空
    return queueP.isEmpty() && queueQ.isEmpty();
  }
```

总结

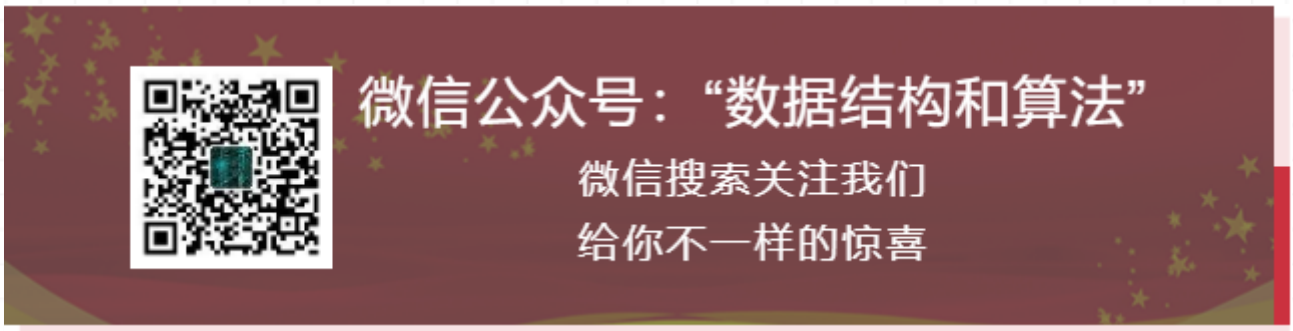
我们从代码层面可以看到第一种方式要简单的多，当然第二种解法也提供了一种思路。

往期推荐

- 474，翻转二叉树的多种解决方式
- 470，DFS和BFS解合并二叉树
- 456，解二叉树的右视图的两种方式
- 434，剑指 Offer-二叉树的镜像

483，完全二叉树的节点个数

原创 山大王wld 数据结构和算法 前天



You got to put the past behind before you can move on.

只有忘记过去，才能继续前行。

问题描述

给出一个**完全二叉树**，求出该树的节点个数。

说明：

完全二叉树的定义如下：在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该层**最左边**的若干位置。若最底层为第 h 层，则该层包含 1~ 2^h 个节点。

示例：

```
输入：
      1
    /  \
   2    3
  / \   /
 4  5 6
```

```
输出：6
```

DFS解决

这题是让求完全二叉树的节点个数，最简单的一种方式就是使用DFS，也就是递归解决。如果当前节点为空，直接返回0即可，否则就返回左子节点的个数+右子节点的个数+1。原理比较简单，直接一行代码搞定

```
public int countNodes(TreeNode root) {
  return root == null ? 0 : countNodes(root.left) + countNodes(root.right) + 1;
}
```

BFS解决

之前讲过二叉树的几种遍历方式373，数据结构-6,树，有前序遍历，中序遍历，后序遍历，BFS，DFS，每种写法都包含递归和非递归，我们只需要把所有的节点都遍历一遍就可以统计出来了，如果每个都写一遍，有点多了，这里就使用BFS来写一个，二叉树的BFS代码如下。

```
public void levelOrder(TreeNode tree) {
  if (tree == null)
    return;
  Queue<TreeNode> queue = new LinkedList<>();
  queue.add(tree);
  while (!queue.isEmpty()) {
    //poll方法相当于移除队列头部的元素
    TreeNode node = queue.poll();
    System.out.println(node.val);
    //如果左子节点不为空就把他加入到队列中
    if (node.left != null)
      queue.add(node.left);
    //如果右子节点不为空也把他加入到队列中
    if (node.right != null)
      queue.add(node.right);
  }
}
```

我们来对它进行改造一下，统计节点的个数

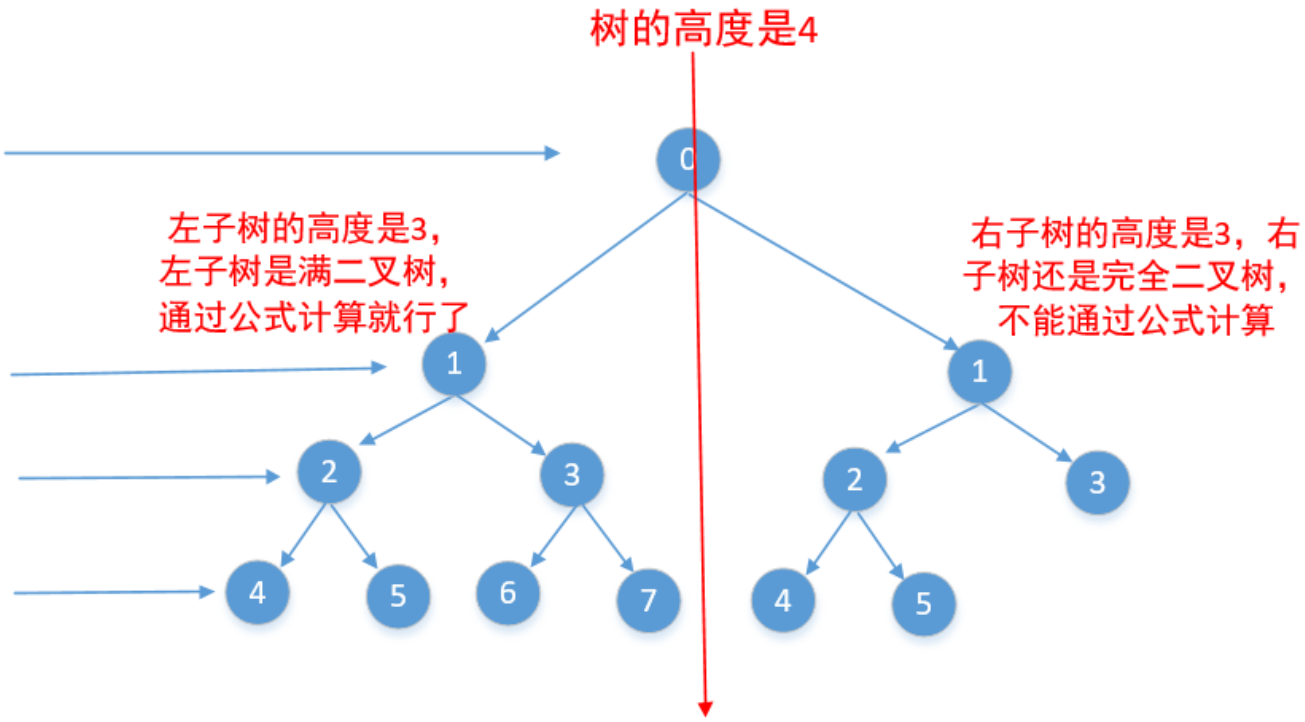
```
public int countNodes(TreeNode root) {
  if (root == null)
    return 0;
  int count = 0;
  Queue<TreeNode> queue = new LinkedList<>();
  queue.add(root);
  while (!queue.isEmpty()) {
    //poll方法相当于移除队列头部的元素
    TreeNode node = queue.poll();
    count++;//统计节点的个数
    if (node.left != null)
      queue.add(node.left);
    if (node.right != null)
      queue.add(node.right);
  }
  return count;
}
```

从左子树找树的高度

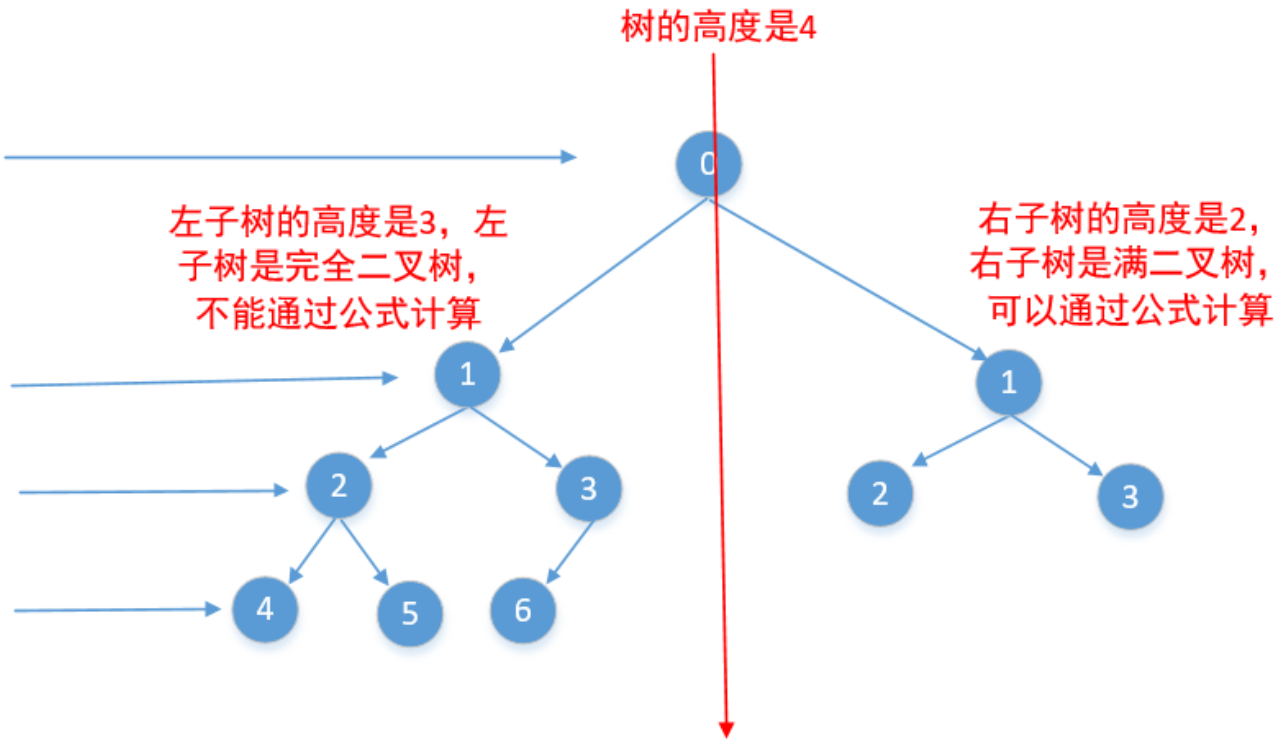
题中对完全二叉树的描述已经很清晰了，如果我们还是用上面的两种方式一个个遍历的话，效果明显不是很好，可以考虑下面这种方式

先计算树的高度height，然后计算右子树的高度

1，如果右子树的高度等于height-1，说明**左子树是满二叉树**（如下图所示），可以通过公式(2^(height-1))-1计算即可，不需要全部遍历，然后再通过递归的方式计算右子树……，



2，如果右子树的高度不等于height-1，说明**右子树是满二叉树**（如下图所示），只不过比上面那种少了一层，也就是height-2，也可以通过公式(2^(height-2))-1计算，然后再通过递归的方式计算左子树……，



搞懂了上面的原理，代码就简单多了

```java
public int countNodes(TreeNode root) {
    //计算树的高度，
    int height = treeHeight(root);
    //如果树是空的，或者高度是1，直接返回
    if (height == 0 || height == 1)
        return height;
    //如果右子树的高度是树的高度减1，说明左子树是满二叉树，
    //左子树可以通过公式计算，只需要递归右子树就行了
    if (treeHeight(root.right) == height - 1) {
        //注意这里的计算，左子树的数量是实际上是(1 << (height - 1))-1，
        //不要把根节点给忘了，在加上1就是(1 << (height - 1))
        return (1 << (height - 1)) + countNodes(root.right);
    } else {
        //如果右子树的高度不是树的高度减1，说明右子树是满二叉树，可以通过
        //公式计算右子树，只需要递归左子树就行了
        return (1 << (height - 2)) + countNodes(root.left);
    }
}

//计算树的高度
private int treeHeight(TreeNode root) {
    return root == null ? 0 : 1 + treeHeight(root.left);
}
```

或者我们还可以把它改为非递归的

```java
public int countNodes(TreeNode root) {
    int count = 0, height = treeHeight(root);
    while (root != null) {
        //如果右子树的高度是树的高度减1，那么左子树就是满二叉树
        if (treeHeight(root.right) == height - 1) {//左子树是满二叉树
            count += 1 << height - 1;
            root = root.right;
        } else {//右子树是满二叉树
            count += 1 << height - 2;
            root = root.left;
        }
        height--;
    }
    return count;
}

```

```
  //计算树的高度
  private int treeHeight(TreeNode root) {
    return root == null ? 0 : 1 + treeHeight(root.left);
  }
```

从右子树找树的高度

上面是先计算二叉树的高度，它是从左子节点一直往下走，找到树的高度。还可以换种思路从树的右子节点往下走，找到树的高度，原理都差不多，代码中有详细注释，就不在过多介绍

```
public int countNodes(TreeNode root) {
  if (root == null)
    return 0;
  //计算高度，注意这里不是树的实际高度
  int height = treeHeight(root);
  if (treeHeight(root.left) == height) {//左子树是满二叉树，通过公式计算
    return (1 << height) + countNodes(root.right);
  } else {//右子树是满二叉树，通过公式计算
    return (1 << height - 1) + countNodes(root.left);
  }
}

//计算树的高度，注意这个结果不是树的实际高度，如果树是满二叉树，他就是树的
//高度，如果不是满二叉树，他就是树的高度减1
private int treeHeight(TreeNode root) {
  return root == null ? 0 : 1 + treeHeight(root.right);//注意这里遍历的是树的右结点
}
```

总结

二叉树的遍历方式除了之前讲373，数据结构-6,树中的前序，中序，后续，以及BFS以外，还有莫里斯的前中后3种遍历方式。这些遍历方式有的还包含递归以及非递归等多种写法，如果都写一遍的话答案就非常多了。但这里说的是完全二叉树，我们可以根据完全二叉树的特性来计算，没必要把所有的节点都要遍历一遍。

二叉树常见的几种遍历方式（包括前序，中序，后序，DFS，BFS）在第373题都有过介绍，并且都有递归和非递归等多种实现方式。关于二叉树的莫里斯（Morris）的3种遍历方式后面有时间也会做介绍，期待大家一块学习。

往期推荐

- 464. BFS和DFS解二叉树的所有路径
- 456，解二叉树的右视图的两种方式
- 387，二叉树中的最大路径和
- 374，二叉树的最小深度

474，翻转二叉树的多种解决方式

原创 山大王wld 数据结构和算法 11月10日

收录于话题
#算法图文分析

95个 >



Dream is not about what you want, but what you do after knowing who you are.

理想不是想想而已，是看清自我后的不顾一切。

问题描述

翻转一棵二叉树。

示例：

输入：

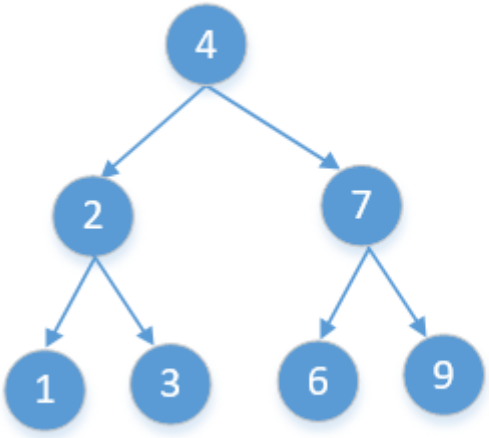
```
     4
   /   \
  2     7
 / \   / \
1   3 6   9
```

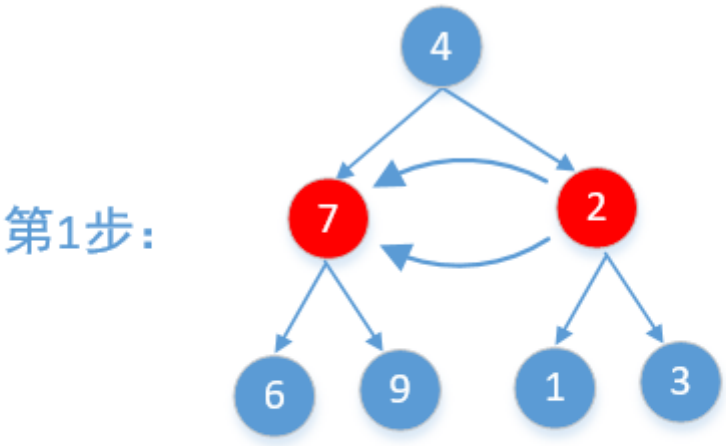
输出：

```
     4
   /   \
  7     2
 / \   / \
9   6 3   1
```

递归方式解决

翻转二叉树，可以先交换根节点的两个子节点，然后通过同样的方式在交换根节点的子节点的两个子节点……一直这样交换下去，画个图看一下





……

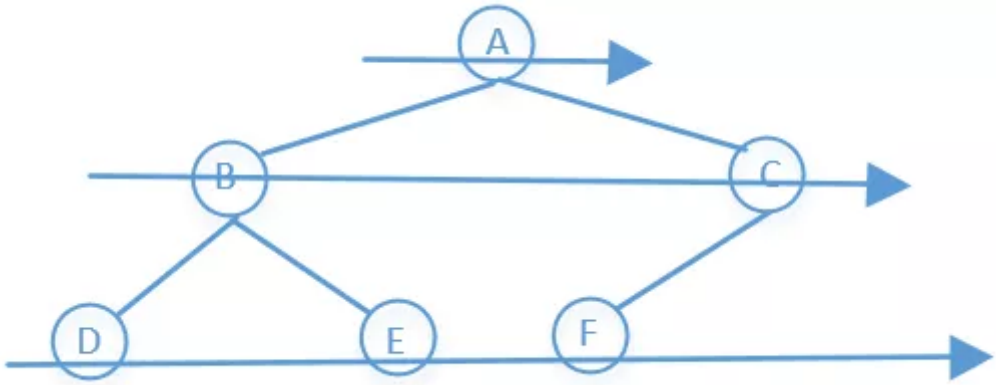
代码比较简单

```java
public TreeNode invertTree(TreeNode root) {
    //递归的边界条件判断
    if (root == null)
        return null;
    //先交换子节点
    TreeNode left = root.left;
    root.left = root.right;
    root.right = left;
```

```
    //递归调用
    invertTree(root.left);
    invertTree(root.right);
    return root;
  }
```

BFS解决

如果对树的遍历比较熟悉的话，我们只要遍历树的所有节点，然后把他们的左右子节点相互交换即可，如果这样写，那么答案就比较多了。这里来看下二叉树的BFS解法，二叉树的BFS就是一层一层的遍历，像下面这样



二叉树的BFS代码如下

```
public void levelOrder(TreeNode tree) {
  if (tree == null)
    return;
  Queue<TreeNode> queue = new LinkedList<>();
  queue.add(tree);//相当于把数据加入到队列尾部
  while (!queue.isEmpty()) {
    //poll方法相当于移除队列头部的元素
    TreeNode node = queue.poll();
    System.out.println(node.val);
    if (node.left != null)
      queue.add(node.left);
    if (node.right != null)
      queue.add(node.right);
  }
}
```

我们就参照这种方式来写下，每次遍历节点的时候都要交换子节点，所以代码很容易写

```
public TreeNode invertTree(TreeNode root) {
  if (root == null)
    return root;
  Queue<TreeNode> queue = new LinkedList<>();
  queue.add(root);//相当于把数据加入到队列尾部
  while (!queue.isEmpty()) {
    //poll方法相当于移除队列头部的元素
    TreeNode node = queue.poll();
    //先交换子节点
    TreeNode left = node.left;
    node.left = node.right;
    node.right = left;

    if (node.left != null)
      queue.add(node.left);
    if (node.right != null)
      queue.add(node.right);
  }
```

```
    return root;
}
```

DFS解决

上面说了只要能遍历二叉树的所有节点，然后交换子节点，就能完成这道题，二叉树还有一种方式是DFS遍历，他的代码如下

```
public static void treeDFS(TreeNode root) {
    Stack<TreeNode> stack = new Stack<>();
    stack.add(root);
    while (!stack.empty()) {
        TreeNode node = stack.pop();
        System.out.println(node.val);
        if (node.right != null) {
            stack.push(node.right);
        }
        if (node.left != null) {
            stack.push(node.left);
        }
    }
}
```

我们来参照这种方式写下

```
public TreeNode invertTree(TreeNode root) {
    if (root == null)
        return root;
    Stack<TreeNode> stack = new Stack<>();
    stack.add(root);
    while (!stack.empty()) {
        TreeNode node = stack.pop();
        //先交换子节点
        TreeNode left = node.left;
        node.left = node.right;
        node.right = left;
        if (node.right != null) {
            stack.push(node.right);
        }
        if (node.left != null) {
            stack.push(node.left);
        }
    }
    return root;
}
```

总结

这道题其实很简单，基本上没什么难度，只要能遍历二叉树的所有节点都可以轻松完成。之前在[373，数据结构-6,树]中讲过二叉树的几种遍历方式，包括递归和非递归的。我们还可以使用二叉树的中序遍历来解决，二叉树的中序遍历代码如下

```
public static void inOrderTraversal(TreeNode tree) {
    Stack<TreeNode> stack = new Stack<>();
    while (tree != null || !stack.isEmpty()) {
        while (tree != null) {
            stack.push(tree);
            tree = tree.left;
        }
        if (!stack.isEmpty()) {
            tree = stack.pop();
            System.out.println(tree.val);
```

```
            tree = tree.right;
        }
    }
}
```

修改一下，就变成这题的答案了。

```
public TreeNode invertTree(TreeNode root) {
    if (root == null)
        return root;
    Stack<TreeNode> stack = new Stack<>();
    TreeNode node = root;
    while (node != null || !stack.isEmpty()) {
        while (node != null) {
            stack.push(node);
            node = node.left;
        }
        if (!stack.isEmpty()) {
            node = stack.pop();
            //先交换子节点
            TreeNode left = node.left;
            node.left = node.right;
            node.right = left;
            //注意，这里是交换之后的，所以要修改
            node = node.left;
        }
    }
    return root;
}
```

往期推荐

- 470，DFS和BFS解合并二叉树
- 464. BFS和DFS解二叉树的所有路径
- 456，解二叉树的右视图的两种方式
- 442，剑指 Offer-回溯算法解二叉树中和为某一值的路径

471，二叉搜索树中的插入操作

原创 山大王wld 数据结构和算法 11月3日

收录于话题
#算法图文分析

95个 >



True courage is about knowing not when to take a life, but when to spare one.

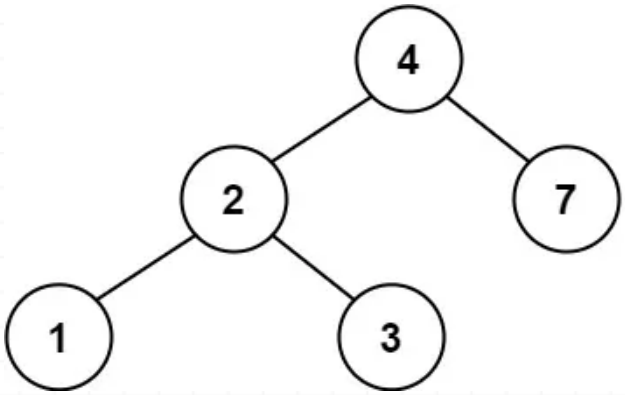
真正的勇气不在于取人性命，而在于宽恕。

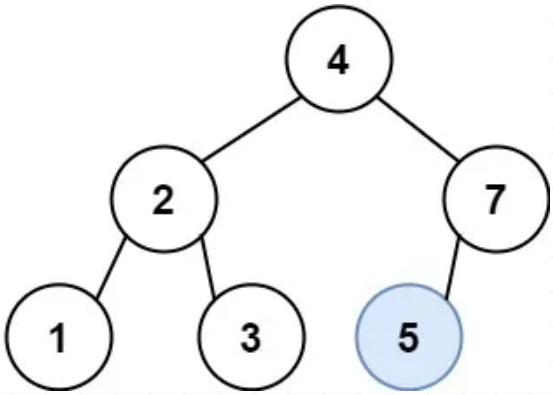
问题描述

给定二叉搜索树（BST）的根节点和要插入树中的值，将值插入二叉搜索树。返回插入后二叉搜索树的根节点。输入数据保证新值和原始二叉搜索树中的任意节点值都不同。

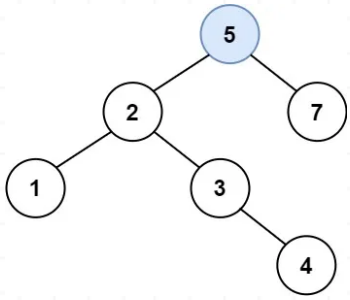
注意，可能存在多种有效的插入方式，只要树在插入后仍保持为二叉搜索树即可。你可以返回任意有效的结果 。

示例 1：





> **输入**：root = [4,2,7,1,3], val = 5
> **输出**：[4,2,7,1,3,5]
> **解释**：另一个满足题目要求可以通过的树是：



示例 2：

> **输入**：
> root = [40,20,60,10,30,50,70],
> val = 25
>
> **输出**：
> [40,20,60,10,30,50,70,null,null,25]

示例 3：

> **输入**：
> root =
> [4,2,7,1,3,null,null,null,null,null,null]
> val = 5
>
> **输出**：[4,2,7,1,3,5]

提示：

- 给定的树上的节点数介于 0 和 10^4 之间
- 每个节点都有一个唯一整数值，取值范围从 0 到 10^8

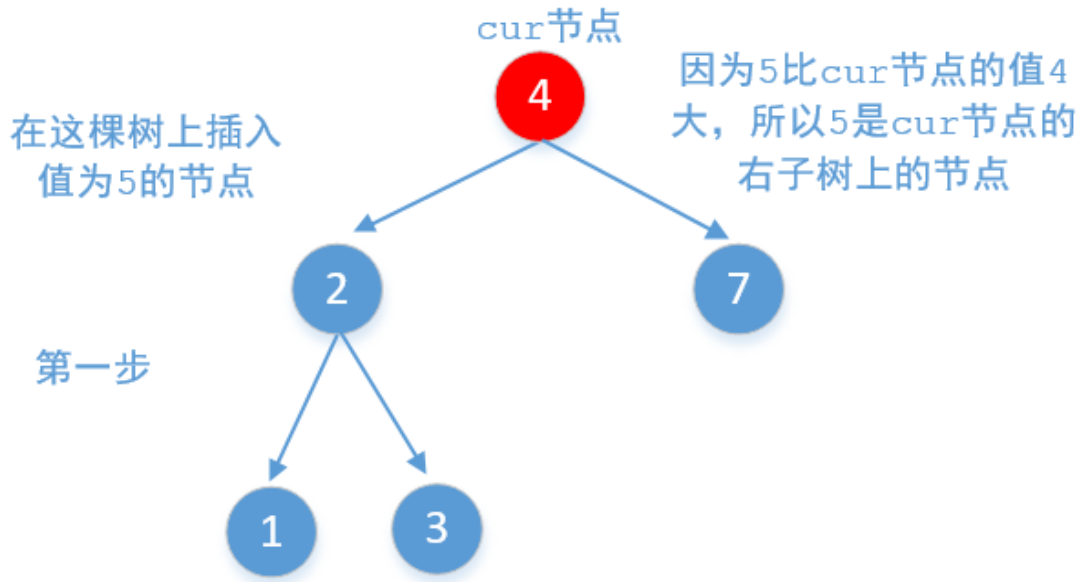
- -10^8 <= val <= 10^8
- 新值和原始二叉搜索树中的任意节点值都不同

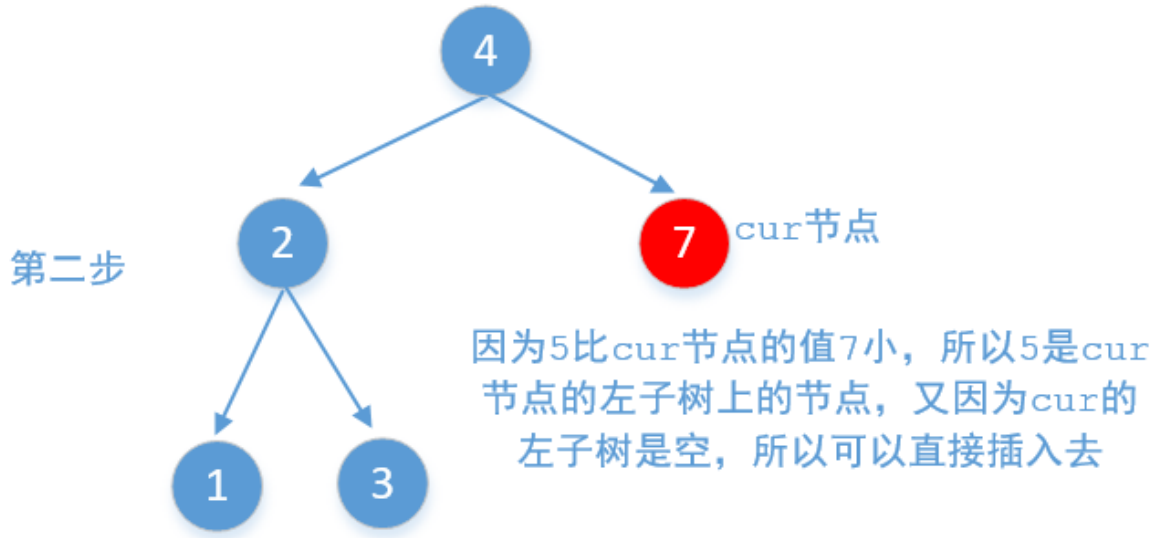
非递归方式解决

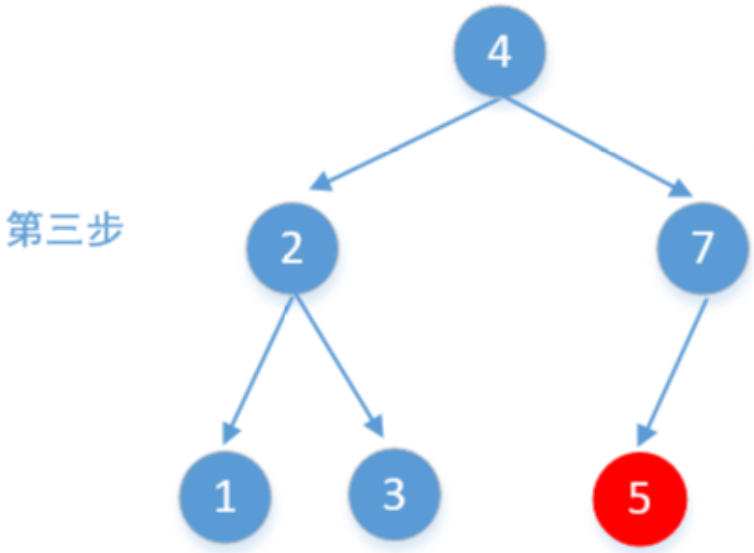
这题说的是让在二叉搜索树中插入一个节点，最简单的一种方式就是插入到叶子节点。二叉搜索树的特点是左子树的值都小于当前节点，右子树的值都大于当前节点，并且左右子树都具有这个特性。所以我们需要用插入的值val和根节点比较，

- **如果val大于根节点，说明值为val的节点应该插入到root节点的右子树上**
- **如果val小于根节点，说明值为val的节点应该插入到root节点的左子树上**

然后再继续执行上面的操作，直到找到叶子节点为止，然后再把它插进去，就以题中示例为例画个图来看一下







代码如下

```
public TreeNode insertIntoBST(TreeNode root, int val) {
  //边界条件判断
  if (root == null)
    return new TreeNode(val);
  TreeNode cur = root;
  while (true) {
    //如果当前节点cur的值大于val，说明val值应该插入到
    //当前节点cur的左子树，否则就插入到当前节点cur的右子树
    if (cur.val > val) {
      //如果左子节点不为空，就继续往下找
      if (cur.left != null) {
        cur = cur.left;
      } else {//如果左子节点为空，就直接插入去，然后再返回root节点
        cur.left = new TreeNode(val);
        return root;
      }
    } else {//同上
      if (cur.right != null) {
        cur = cur.right;
      } else {
        cur.right = new TreeNode(val);
        return root;
      }
    }
  }
}
```

递归方式解决

递归的方式原理还和上面一样，一直往下找，直到找到叶子节点为止，代码如下

```
public TreeNode insertIntoBST(TreeNode root, int val) {
  //如果root为空，就直接创建一个新的节点
  if (root == null)
    return new TreeNode(val);
  //如果root节点的值大于val，说明值为val的节点应该在root
  //节点的左子树上
  if (root.val > val)
    root.left = insertIntoBST(root.left, val);
  else
    root.right = insertIntoBST(root.right, val);
  return root;
}
```

问题分析

做这题之前首先要弄懂什么是二叉搜索树，然后才能进行后面的操作，最简单的方式就是把值插入到二叉搜索树的叶子节点。

往期推荐

- 466. 使用快慢指针把有序链表转换二叉搜索树
- 464. BFS和DFS解二叉树的所有路径
- 457，二叉搜索树的最近公共祖先
- 442，剑指 Offer-回溯算法解二叉树中和为某一值的路径

464. BFS和DFS解二叉树的所有路径

原创 山大王wld 数据结构和算法 10月19日

收录于话题
#算法图文分析

95个 >



You've gotta let go that stuff in the past, cause it just doesn't matter.

过去的事就让它过去，因为那些都无关紧要。

问题描述

给定一个二叉树，返回所有从根节点到叶子节点的路径。

说明: 叶子节点是指没有子节点的节点。

示例:

输入:

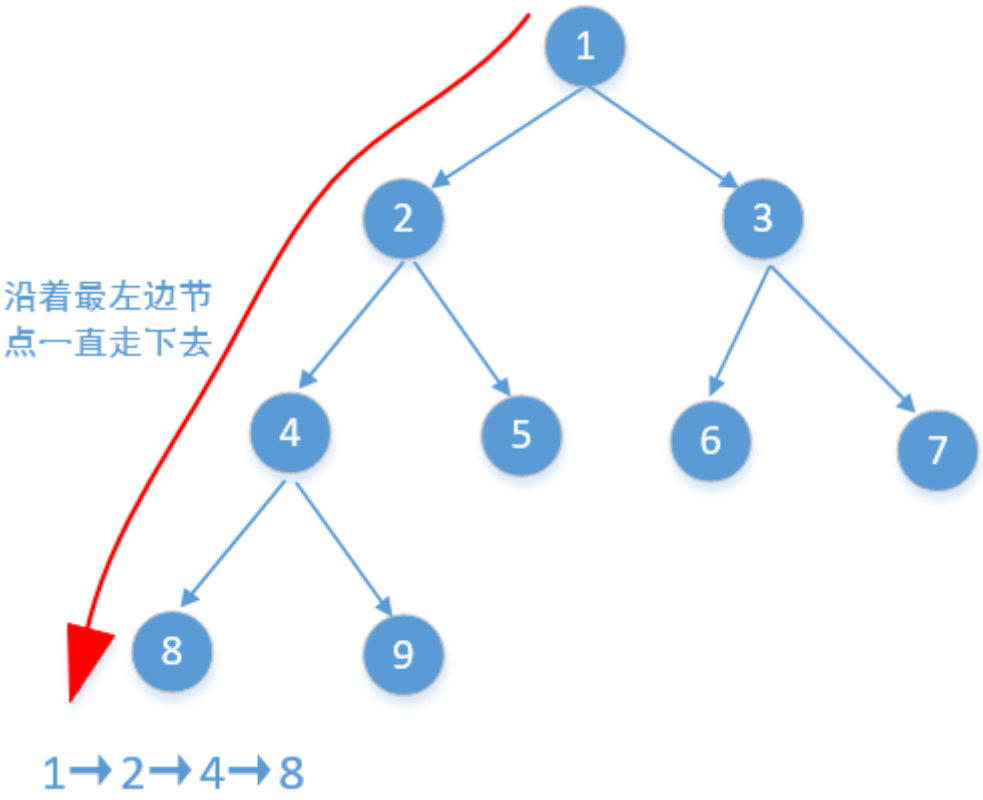
```
   1
 /   \
2     3
 \
  5
```

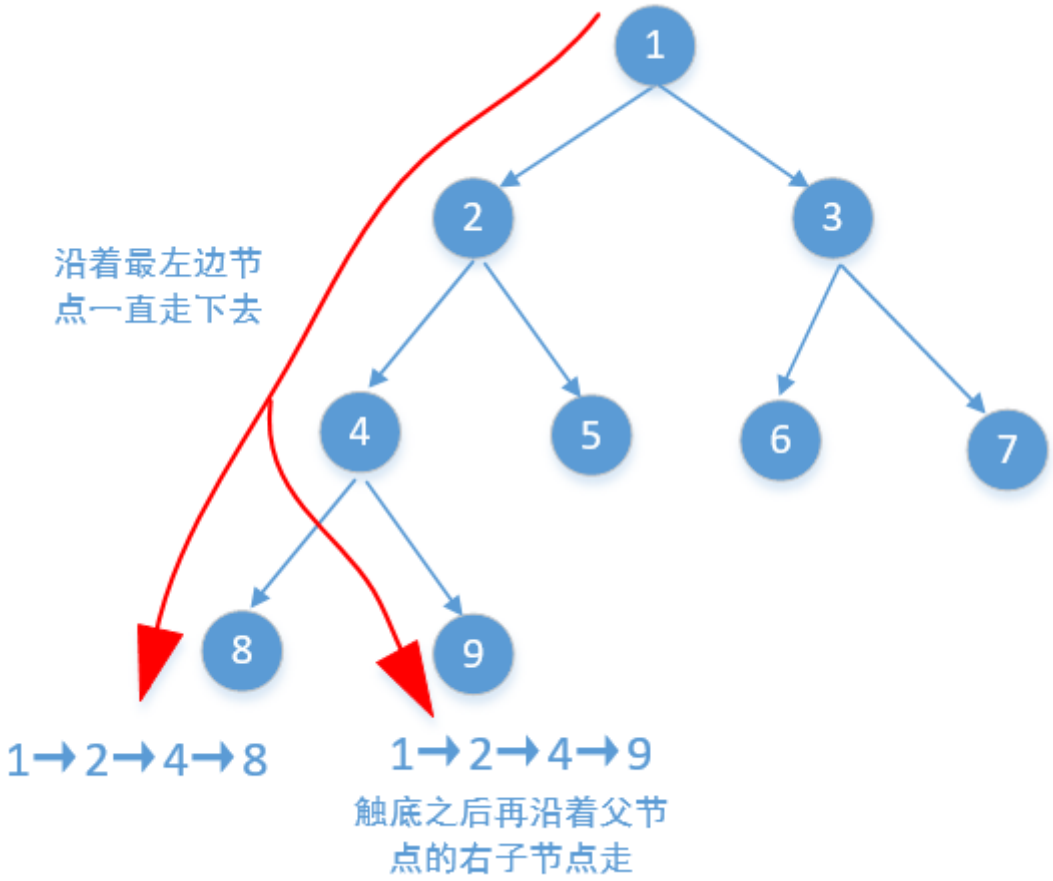
输出: ["1->2->5", "1->3"]

> **解释：** 所有根节点到叶子节点的路径为：
> 1->2->5，1->3

DFS解决

这题让求的是从根节点到叶子节点的所有路径，最常见的一种方式就是DFS（深度优先搜索），也就是从根节点沿着最左边节点一直走下去（如果没有左子节点，有右子节点，会沿着右子节点走下去），当到达叶子节点的时候在返回到父节点，然后沿着父节点的右子节点开始走下去，如下图所示





在前面讲过二叉树的dfs，[373，数据结构-6,树](#)，他的代码如下

```
public static void treeDFS(TreeNode root) {
  if (root == null)
    return;
  System.out.println(root.val);
  treeDFS(root.left);
  treeDFS(root.right);
}
```

他就是从根节点到叶子节点的所有路径都会访问一遍，我们只需要把这个路径串联起来即可，这里来仿照上面的代码改造一下

```
public List<String> binaryTreePaths(TreeNode root) {
  List<String> res = new ArrayList<>();
  if (root == null)
    return res;
  dfs(root, "", res);
  return res;
}

private void dfs(TreeNode root, String path, List<String> res) {
  //如果到达叶子节点，就把结果存放到集合res中
  if (root.left == null && root.right == null)
    res.add(path + root.val);
  //如果左子节点不为空，就沿着左子节点走下去
  if (root.left != null)
    dfs(root.left, path + root.val + "->", res);
  //如果右子节点不为空，就沿着右子节点走下去
  if (root.right != null)
    dfs(root.right, path + root.val + "->", res);
}
```

在第373题讲到二叉树DFS遍历的时候，还有一种非递归的写法，代码如下

```
public static void treeDFS(TreeNode root) {
  Stack<TreeNode> stack = new Stack<>();
```

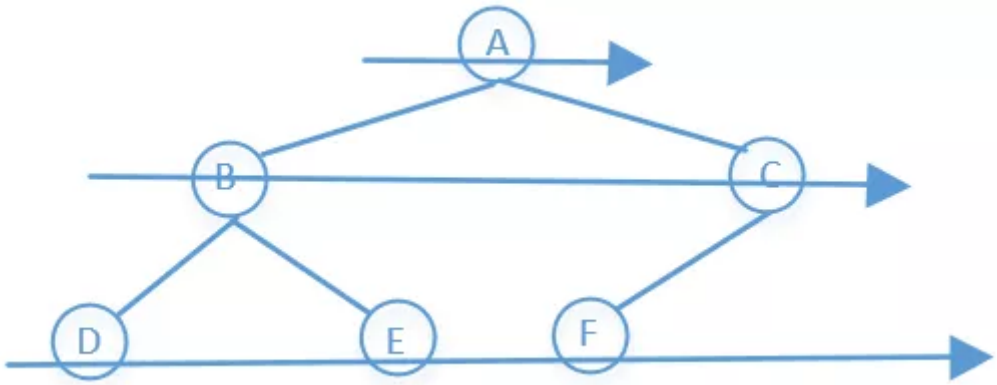
```
    stack.add(root);
    while (!stack.empty()) {
      TreeNode node = stack.pop();
      System.out.println(node.val);
      if (node.right != null) {
        stack.push(node.right);
      }
      if (node.left != null) {
        stack.push(node.left);
      }
    }
  }
```

因为他的遍历过程没变，只不过写法改变了，我们也可以来对他进行改造，看下最终代码

```
public List<String> binaryTreePaths(TreeNode root) {
    List<String> res = new ArrayList<>();
    if (root == null)
      return res;
    //存储节点的栈
    Stack<TreeNode> stackNode = new Stack<>();
    //存储路径的栈，和上面的栈是同步进行的，这里路径指的是
    //从根节点到当前节点的路径
    Stack<String> stackPath = new Stack<>();
    //根节点和根节点的路径同时入栈
    stackNode.push(root);
    stackPath.push(root.val + "");
    while (!stackNode.empty()) {
      //当前节点和对应的路径同时出栈
      TreeNode node = stackNode.pop();
      String path = stackPath.pop();
      //如果到达叶子节点，就把路径加入到集合res中
      if (node.left == null && node.right == null) {
        res.add(path);
      }
      //如果右子节不为空，就把右子节点和对应的路径分别加入到栈中
      if (node.right != null) {
        stackPath.push(path + "->" + node.right.val);
        stackNode.push(node.right);
      }
      //同上
      if (node.left != null) {
        stackPath.push(path + "->" + node.left.val);
        stackNode.push(node.left);
      }
    }
    return res;
  }
```

BFS解决

BFS就是一层一层的打印，如下图所示



只需要使用一个队列，把每层的节点都存放到队列中，然后再一个个出队，顺便把子节点在一个个存放到队列中……一直这样循环下去，直到队列为空为止，在373，数据结构-6,树的时候也提到过二叉树的BFS遍历，他的代码如下

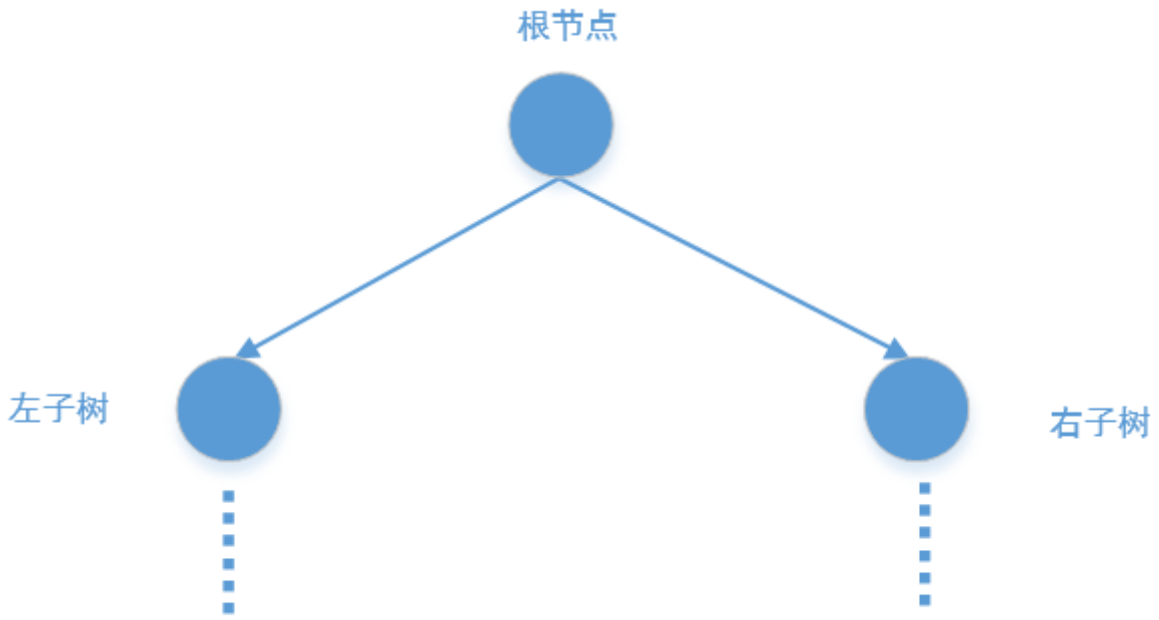
```java
public void levelOrder(TreeNode tree) {
  if (tree == null)
    return;
  Queue<TreeNode> queue = new LinkedList<>();
  queue.add(tree);//相当于把数据加入到队列尾部
  while (!queue.isEmpty()) {
    //poll方法相当于移除队列头部的元素
    TreeNode node = queue.poll();
    System.out.println(node.val);
    if (node.left != null)
      queue.add(node.left);
    if (node.right != null)
      queue.add(node.right);
  }
}
```

也可以对上面的代码进行改造，改造的原理和DFS的非递归解法一样，就是使用一个变量存放从根节点到当前节点的路径，代码如下

```java
public List<String> binaryTreePaths(TreeNode root) {
  List<String> res = new ArrayList<>();
  if (root == null)
    return res;
  //队列，节点和路径成对出现，路径就是从根节点到当前节点的路径
  Queue<Object> queue = new LinkedList<>();
  queue.add(root);
  queue.add(root.val + "");
  while (!queue.isEmpty()) {
    TreeNode node = (TreeNode) queue.poll();
    String path = (String) queue.poll();
    //如果到叶子节点，说明找到了一条完整路径
    if (node.left == null && node.right == null) {
      res.add(path);
    }

    //右子节点不为空就把右子节点和路径存放到队列中
    if (node.right != null) {
      queue.add(node.right);
      queue.add(path + "->" + node.right.val);
    }

    //左子节点不为空就把左子节点和路径存放到队列中
    if (node.left != null) {
      queue.add(node.left);
      queue.add(path + "->" + node.left.val);
    }
  }
  return res;
}
```

递归解法

我们来思考这样一个问题，如果知道了左子树和右子树的所有路径，我们在用根节点和他们连在一起，是不是就是从根节点到所有叶子节点的所有路径，所以这里最容易想到的就是递归



最后再来看下代码

```
1   public List<String> binaryTreePaths(TreeNode root) {
2     List<String> res = new ArrayList<>();
3     if (root == null)
4       return res;
5     //到达叶子节点，把路径加入到集合中
6     if (root.left == null && root.right == null) {
7       res.add(root.val + "");
8       return res;
9     }
10    //遍历左子节点的所有路径
11    for (String path : binaryTreePaths(root.left)) {
12      res.add(root.val + "->" + path);
13    }
14    //遍历右子节点的所有路径
15    for (String path : binaryTreePaths(root.right)) {
16      res.add(root.val + "->" + path);
17    }
18    return res;
19  }
```

总结

二叉树的遍历常见的也就是前序遍历，中序遍历，后续遍历，BFS，DFS，以及莫里斯的前序，中序和后续这几种，有的还说前序遍历就是DFS，其实可以这么说。但DFS却不是前序遍历，因为DFS不光可以先从左子树开始遍历还可以先从右子树开始遍历。前

面五种之前在373，数据结构-6,树已经讲过，后面的3种后续有时间会再做介绍。只要掌握二叉树的这几种遍历方式，对于大部分关于二叉树的问题都可以参照这几种方式进行修改来解决。

往期推荐

- 456，解二叉树的右视图的两种方式
- 444，二叉树的序列化与反序列化
- 442，剑指 Offer-回溯算法解二叉树中和为某一值的路径
- 372，二叉树的最近公共祖先

458，填充每个节点的下一个右侧节点指针 II

原创 山大王wld 数据结构和算法 9月29日

收录于话题
#算法图文分析

95个 >



What I see here is nothing but a shell. What is most important is what is invisible.

我看到的都是表象，最重要的东西肉眼是看不见的。

问题描述

给定一个二叉树

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 NULL。

初始状态下，所有 next 指针都被设置为 NULL。

进阶：

- 你只能使用常量级额外空间。
- 使用递归解题也符合要求，本题中递归程序占用的栈空间不算做额外的空间复杂度。

示例：

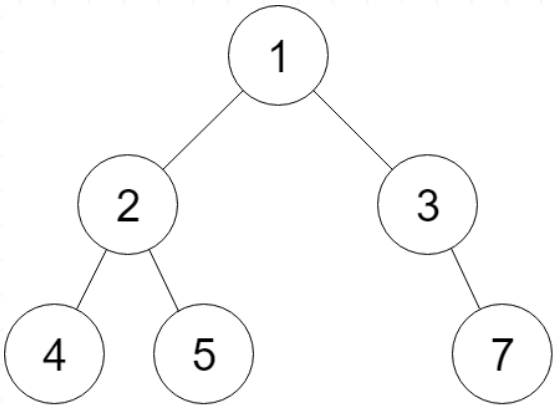


Figure A

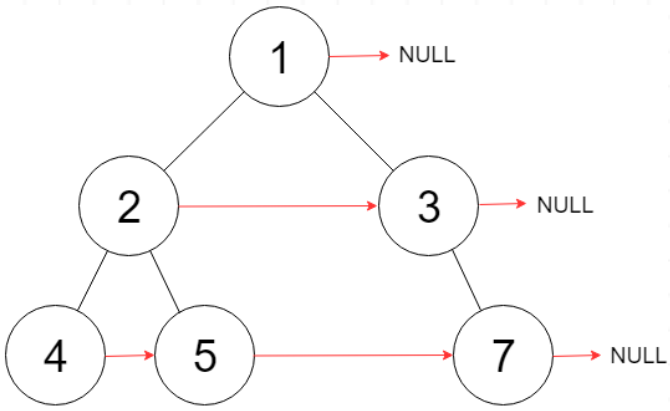


Figure B

> **输入**：root = [1,2,3,4,5,null,7]
> **输出**：[1,#,2,3,#,4,5,7,#]
> **解释**：给定二叉树如图 A 所示，你的函数应该填充它的每个 next 指针，以指向其下一个右侧节点，如图 B 所示。

提示：

- 树中的节点数小于 `6000`
- `-100 <= node.val <= 100`

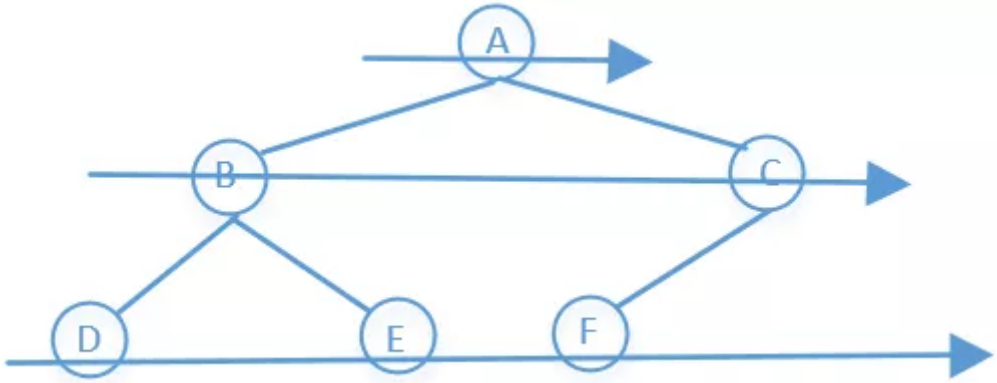
BFS解决

看到关于二叉树的问题，首先要想到关于二叉树的一些常见遍历方式，
对于二叉树的遍历有

1. **前序遍历**
2. **中序遍历**
3. **后续遍历**
4. **深度优先搜索（DFS）**
5. **宽度优先搜索（BFS）**

除了上面介绍的5种以外，还有**Morris（莫里斯）的前中后3种遍历方式**，总共也就这8种。所以只要遇到二叉树相关的算法题，首先想到的就是上面的几种遍历方式，然后再稍加修改，基本上也就这个套路。

这题让求的就是让把二叉树中每行都串联起来，对于这道题来说最适合的就是BFS。也就是一行一行的遍历，如下图所示



他的代码如下

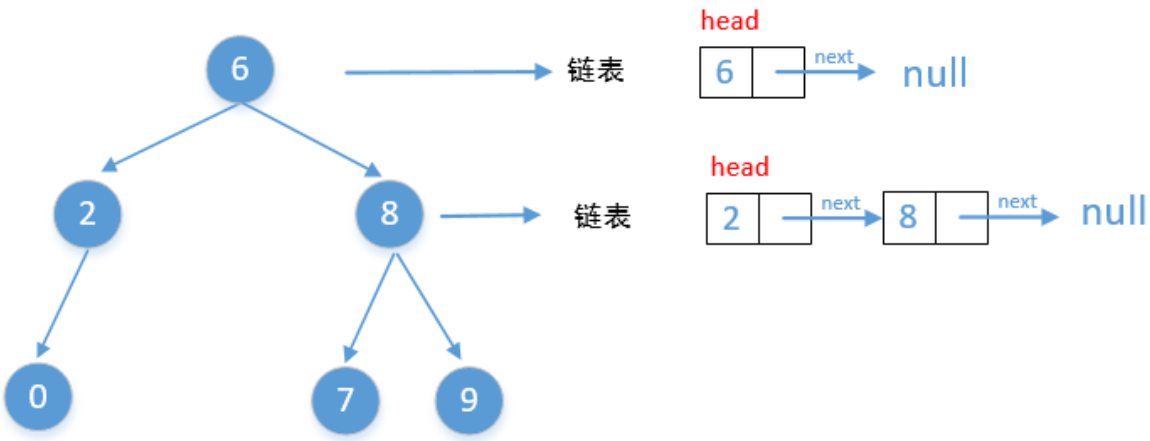
```
1   public void levelOrder(TreeNode tree) {
2     if (tree == null)
3       return;
4     Queue<TreeNode> queue = new LinkedList<>();
5     queue.add(tree);//相当于把数据加入到队列尾部
6     while (!queue.isEmpty()) {
7       //poll方法相当于移除队列头部的元素
8       TreeNode node = queue.poll();
9       System.out.println(node.val);
10      if (node.left != null)
11        queue.add(node.left);
12      if (node.right != null)
13        queue.add(node.right);
14    }
15  }
```

在遍历每一行的时候，只要把他们串联起来就OK，下面就来把上面的代码改造一下

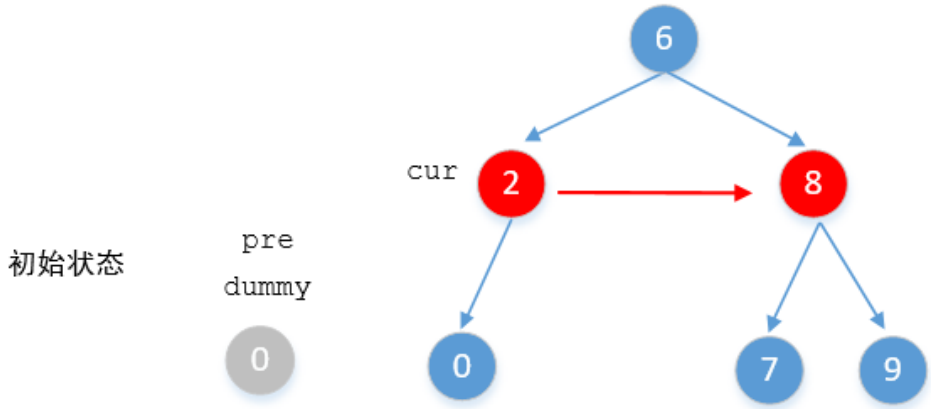
```
1   public Node connect(Node root) {
2     if (root == null)
3       return root;
4     Queue<Node> queue = new LinkedList<>();
5     queue.add(root);
6     while (!queue.isEmpty()) {
7       //每一层的数量
8       int levelCount = queue.size();
9       //前一个节点
10      Node pre = null;
11      for (int i = 0; i < levelCount; i++) {
12        //出队
13        Node node = queue.poll();
14        //如果pre为空就表示node节点是这一行的第一个，
15        //没有前一个节点指向他，否则就让前一个节点指向他
16        if (pre != null) {
17          pre.next = node;
18        }
19        //然后再让当前节点成为前一个节点
20        pre = node;
21        //左右子节点如果不为空就入队
22        if (node.left != null)
23          queue.add(node.left);
24        if (node.right != null)
25          queue.add(node.right);
26      }
27    }
28    return root;
29  }
```

每一层看做一个链表

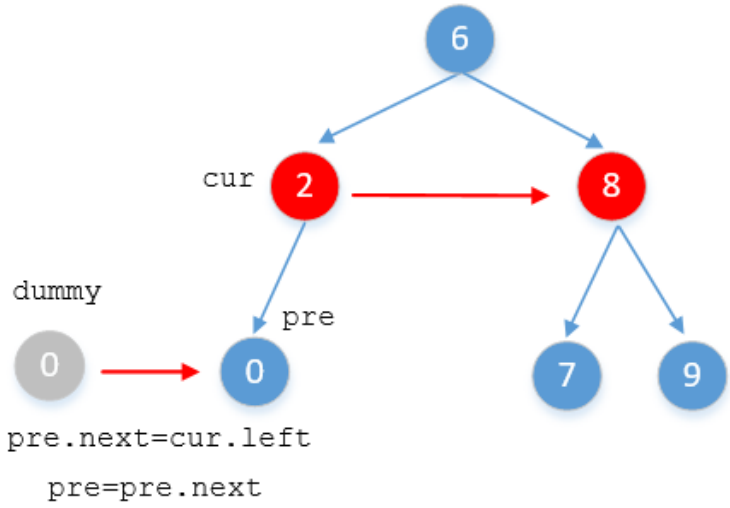
上面计算的时候把节点不停的入队然后再不停的出队，其实可以不需要队列，**每一行都可以看成一个链表**，比如第一行就是只有一个节点的链表，第二行是只有两个节点的链表（假如根节点的左右两个子节点都不为空）……，每次只遍历一层节点，然后顺便把子节点串成一个链表，接着遍历下一层节点的时候再把下下一层的结点串成一个链表……。画个图来看一下



假如我们要访问第2行，也就是把第3行的节点给串起来，过程如下



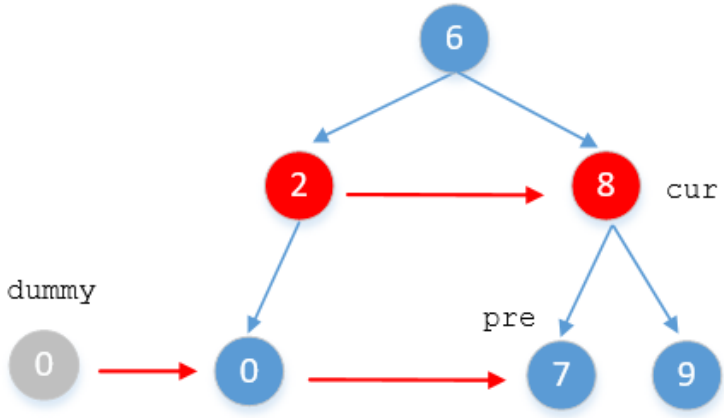
第1步



pre.next=cur.left

pre=pre.next

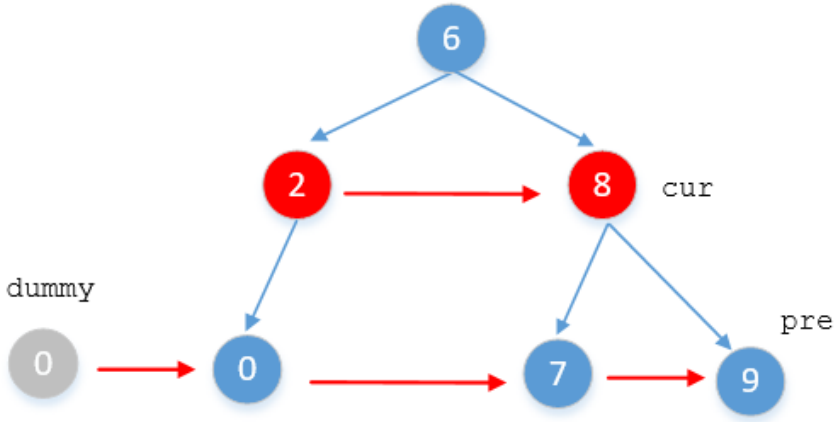
第2步



pre.next=cur.left

pre=pre.next

第3步



pre.next=cur.right

pre=pre.next

代码如下

```
public Node connect(Node root) {
  if (root == null)
    return root;
  //cur我们可以把它看做是每一层的链表
  Node cur = root;
  while (cur != null) {
    //遍历当前层的时候，为了方便操作在下一
```

```
        //层前面添加一个哑结点（注意这里是访问
        //当前层的节点，然后把下一层的节点串起来）
        Node dummy = new Node(0);
        //pre表示访下一层节点的前一个节点
        Node pre = dummy;
        //然后开始遍历当前层的链表
        while (cur != null) {
          if (cur.left != null) {
            //如果当前节点的左子节点不为空，就让pre节点
            //的next指向他，也就是把它串起来
            pre.next = cur.left;
            //然后再更新pre
            pre = pre.next;
          }
          //同理参照左子树
          if (cur.right != null) {
            pre.next = cur.right;
            pre = pre.next;
          }
          //继续访问这一行的下一个节点
          cur = cur.next;
        }
        //把下一层串联成一个链表之后，让他赋值给cur，
        //后续继续循环，直到cur为空为止
        cur = dummy.next;
      }
      return root;
    }
```

总结

看到二叉树首先要想到那8种遍历方式，然后根据题的要求再稍加修改，基本上可以完成大部分关于二叉树的算法题，所以二叉树的那几种遍历方式非常重要，熟练掌握之后对于二叉树的解题会有很大帮助，前面讲过二叉树的5种遍历方式[373，数据结构-6,树]，其中包括递归和非递归的。还有二叉树莫里斯的3种遍历方式，后续有时间会再做介绍。

往期推荐

- 440，剑指 Offer-从上到下打印二叉树 II
- 435，剑指 Offer-对称的二叉树
- 434，剑指 Offer-二叉树的镜像
- 400，二叉树的锯齿形层次遍历

457，二叉搜索树的最近公共祖先

原创 山大王wld 数据结构和算法 9月29日

收录于话题
#算法图文分析

95个 >



A mind troubled by doubt cannot focus on the course to victory.

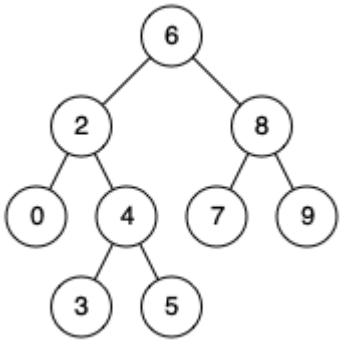
被不确定所困扰的人，是无法专注在成功的道路上的。

问题描述

给定一个**二叉搜索树**，找到该树中两个指定节点的**最近公共祖先**。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。”

例如，给定如下二叉搜索树:
root = [6,2,8,0,4,7,9,null,null,3,5]



示例 1:

> **输入**:
> root = [6,2,8,0,4,7,9,null,null,3,5],
> p = 2, q = 8
>
> **输输出**: 6
> **解释**: 节点2和节点8的最近公共祖先是6。

示例 2:

> **输入**:
> root = [6,2,8,0,4,7,9,null,null,3,5],
> p = 2, q = 4
>
> **输出**: 2
> **解释**: 节点 2 和节点 4 的最近公共祖先是 2，因为根据定义最近公共祖先节点可以为节点本身。

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉搜索树中。

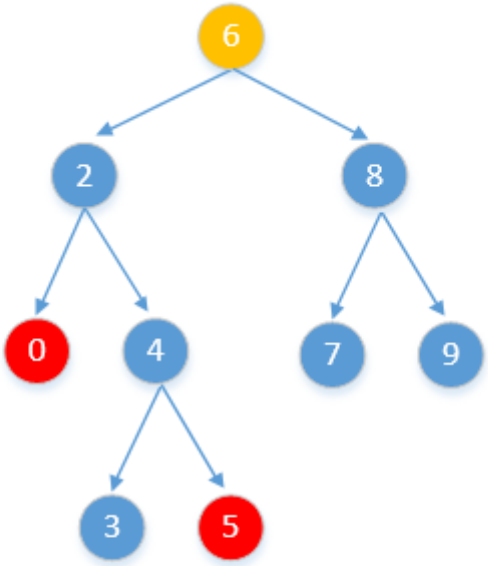
非递归解决

这题让求二叉搜索树的最近公共祖先，而二叉搜索树的特点就是**左子树的所有节点都小于当前节点，右子树的所有节点都大于当前节点，并且每棵子树都具有上述特点**，所以这题就好办了，从更节点开始遍历

- **如果两个节点值都小于根节点，说明他们都在根节点的左子树上，我们往左子树上找**
- **如果两个节点值都大于根节点，说明他们都在根节点的右子树上，我们往右子树上找**
- **如果一个节点值大于根节点，一个节点值小于根节点，说明他们他们一个在根节点的左子树上一个在根节点的右子树上，那么根节点就是他们的最近公共祖先节点。**

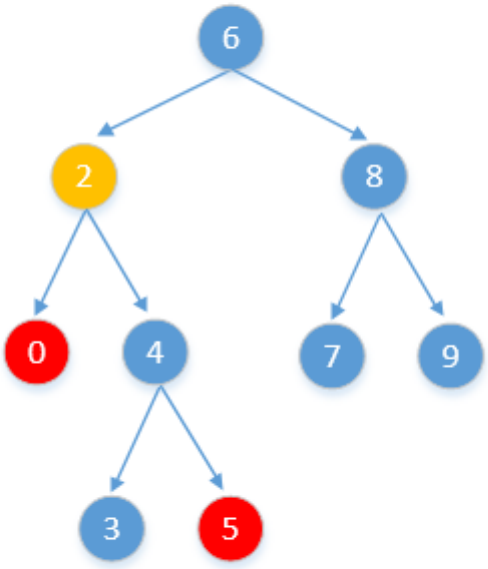
画个图看一下，比如要找0和5的最近公共祖先节点，如下图所示

第一步



因为6比0和5都大，所以0和5是在节点6的左子树上

第二步



节点6的左子树的节点值是2，2比0大，并且小于5，所以0和5一个是在节点2的左子树，一个在节点2的右子树，那么节点2就是0和5的最近祖先节点

搞懂了上面的过程，代码就容易写了，我们来看下

```
1   public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
2     //如果根节点和p,q的差相乘是正数，说明这两个差值要么都是正数要么都是负数，也就是说
3     //他们肯定都位于根节点的同一侧，就继续往下找
```

```
    while ((root.val - p.val) * (root.val - q.val) > 0)
      root = p.val < root.val ? root.left : root.right;
    //如果相乘的结果是负数，说明p和q位于根节点的两侧，如果等于0，说明至少有一个就是根节点
    return root;
  }
```

递归解决

也可把它改为递归的方式

```
  public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    //如果小于等于0，说明p和q位于root的两侧，直接返回即可
    if ((root.val - p.val) * (root.val - q.val) <= 0)
      return root;
    //否则，p和q位于root的同一侧，就继续往下找
    return lowestCommonAncestor(p.val < root.val ? root.left : root.right, p, q);
  }
```

如果嫌代码行数太多，那就一行解决

```
  public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    return (root.val - p.val) * (root.val - q.val) <= 0 ? root : lowestCommonAncestor(p.val < root.val
  }
```

之前讲过372，二叉树的最近公共祖先，也可以参照这道题看一下，第372题的树不是二叉搜索树，而是一般普通的树，所以第372题的解都可以拿到这题来用，这里代码就不在写了，有兴趣的可以看下

总结

这题相对于372还是比较简单的，可以使用二叉搜索树的规律，就是左子树的所有节点都小于当前节点，右子树的所有节点都大于当前节点。

往期推荐

- 399，从前序与中序遍历序列构造二叉树
- 387，二叉树中的最大路径和
- 374，二叉树的最小深度
- 372，二叉树的最近公共祖先

456，解二叉树的右视图的两种方式

原创 山大王wld 数据结构和算法 9月27日

收录于话题
#算法图文分析

95个 >



Even though it may seem silly or wrong, you must try.

即使那看来似乎愚笨或错误，你们都必须试试。

问题描述

给定一棵二叉树，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。

示例:

> **输入**: [1,2,3,null,5,null,4]
> **输出**: [1, 3, 4]
> **解释**:
>
>
>   1   <---
> /   \
> 2   3   <---
> \   \
>   5   4   <---

问题分析

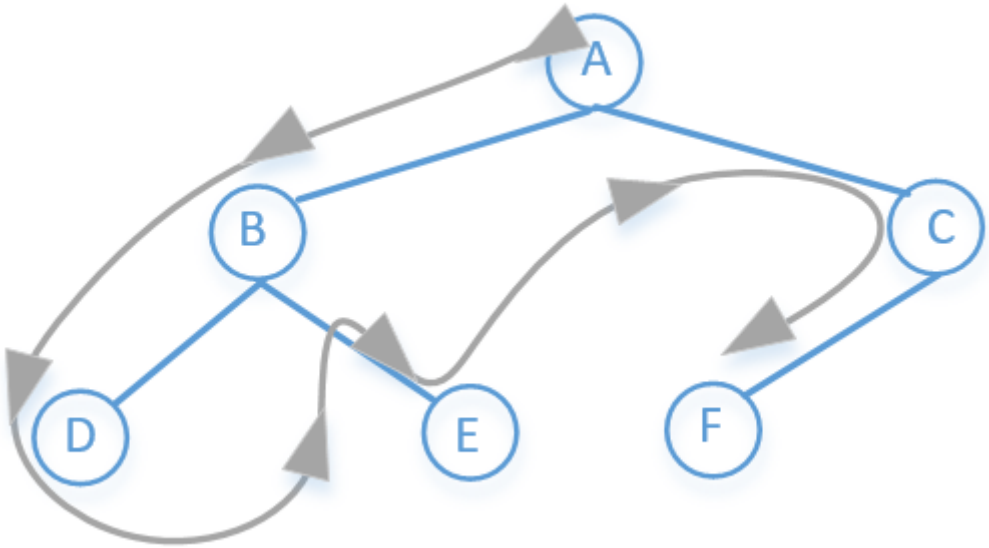
这题说的很明白，就是站在一棵二叉树的右边，你所能看到的结点值。对于二叉树的遍历，前面有简单的介绍过5种遍历方式（有兴趣的可以看下373，数据结构-6,树），分别是：

1. 前序遍历
2. 中序遍历
3. 后续遍历
4. 深度优先搜索（DFS）
5. 宽度优先搜索（BFS）

除了上面介绍的5种以外，还有Morris（莫里斯）的前中后3种遍历方式，后面的3种后续会介绍。所以只要遇到二叉树相关的算法题，首先想到的就是上面的几种遍历方式，基本上也就这个套路，没有别的可选择。对于这道题来说最适合的两种遍历方式就是DFS和BFS。

DFS解决

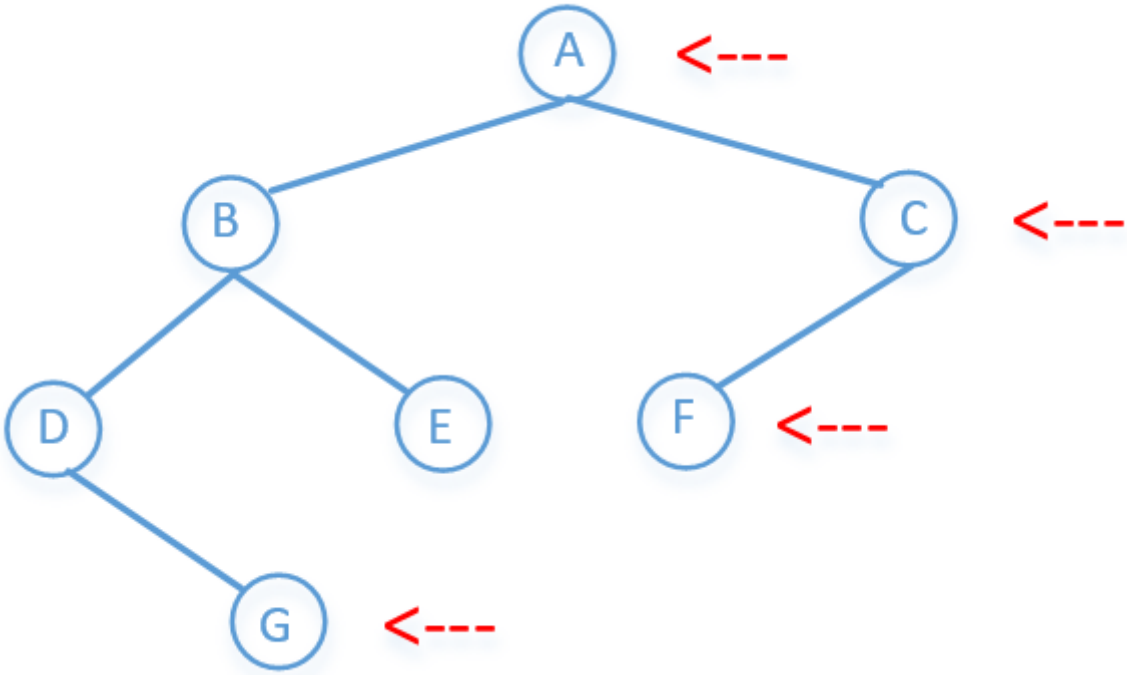
DFS的遍历顺序是从根节点开始一直往左子节点走下去，当走到叶子节点的时候会回到上一个结点，然后从上一个结点的右子节点继续同样的操作……，如下图所示



二叉树的DFS代码如下

```
1   public void treeDFS(TreeNode root) {
2     if (root == null)
3       return;
4     System.out.println(root.val);
5     treeDFS(root.left);
6     treeDFS(root.right);
7   }
```

为了做这道题我们来对上面代码进行改造，上面代码先遍历的是左子树，而这题求的是二叉树的右视图，我们应该先遍历右子树才对。这里随便举个例子来画个图看一下，如下图所示



从根节点开始往右子节点开始遍历，这么我们可以发现一个规律就是每一层最先遍历的结点就是从右边最先看到的结点。如上图所示，我们可以看到，第一层最先遍历的结点是A，第二层最先遍历的结点是C，第三层最先遍历的结点是F，第四层最先看到的是G，而这4个节点值[A，C，F，G]就是我们要求的结果。搞懂了上面的分析过程，代码就so easy了。

```java
public List<Integer> rightSideView(TreeNode root) {
    List<Integer> res = new ArrayList<>();
    dfs(root, res, 0);
    return res;
}

public void dfs(TreeNode curr, List<Integer> res, int level) {
    //递归的终止条件判断
    if (curr == null) {
        return;
    }
    //level表示的是第几层，因为是先遍历右子树，所以每一层最先遍历
    //的结点值就是我们所需要的，当下面语句成立的时候，就表示当前节
    //点值所在的那一行是最先遍历的，所以要把他加入到集合res中
    if (level == res.size()) {
        res.add(curr.val);
    }
    //先遍历右子树，在遍历左子树
    dfs(curr.right, res, level + 1);
    dfs(curr.left, res, level + 1);
}
```

上是递归的方式解决，在373，数据结构-6,树提到过二叉树DFS非递归的代码

```java
public void treeDFS(TreeNode root) {
    Stack<TreeNode> stack = new Stack<>();
    stack.add(root);
    while (!stack.empty()) {
        TreeNode node = stack.pop();
        System.out.println(node.val);
        if (node.right != null) {
            stack.push(node.right);
        }
```

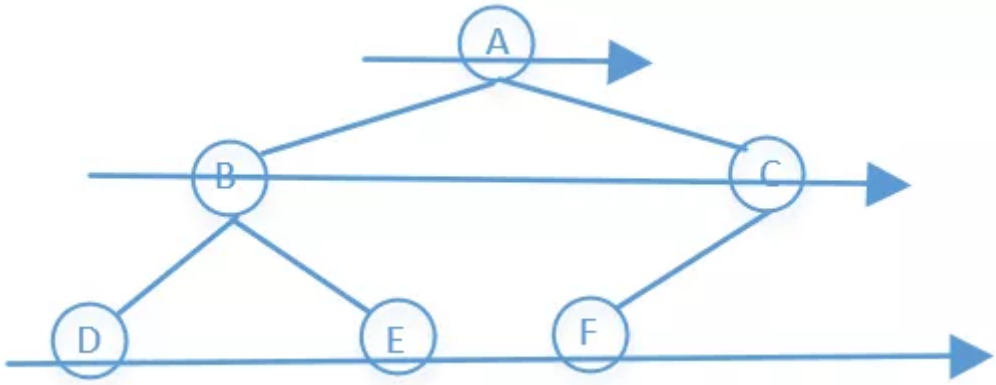
```
        if (node.left != null) {
          stack.push(node.left);
        }
      }
    }
```

我们也可以仿照上面代码来写下，这里使用两个栈，一个是存储当前节点的，一个是存储当前节点所对应的层数，代码如下

```
public List<Integer> rightSideView(TreeNode root) {
    List<Integer> res = new ArrayList<>();
    //终止条件判断
    if (root == null)
        return res;
    //两个栈，一个存储当前节点，一个存储当期节点在第几层
    Stack<TreeNode> stackNode = new Stack<>();
    Stack<Integer> stackLevel = new Stack<>();
    //当前节点和当前节点的层数同时入栈
    stackNode.add(root);
    stackLevel.add(0);
    while (!stackNode.empty()) {
        //当前节点和当前节点的层数同时出栈
        TreeNode node = stackNode.pop();
        int level = stackLevel.pop();
        //下一层最先访问的结点就是我们需要的值
        if (res.size() == level)
            res.add(node.val);
        //先访问左子节点，在访问右子节点
        if (node.left != null) {
            stackNode.push(node.left);
            stackLevel.push(level + 1);
        }
        if (node.right != null) {
            stackNode.push(node.right);
            stackLevel.push(level + 1);
        }
    }
    return res;
}
```

BFS解决

这里只是换了个写法，其实整体思路还是没变，二叉树的BFS是一层一层的往下访问，就像下面图中这样



二叉树的BFS代码如下

```
public void levelOrder(TreeNode tree) {
    if (tree == null)
        return;
    //创建队列
```

```
    Queue<TreeNode> queue = new LinkedList<>();
    //入队
    queue.add(tree);
    while (!queue.isEmpty()) {
      //出队
      TreeNode node = queue.poll();
      System.out.println(node.val);
      if (node.left != null)
        queue.add(node.left);
      if (node.right != null)
        queue.add(node.right);
    }
  }
```

我们来对他进行改造一下

```
public List<Integer> rightSideView(TreeNode root) {
  List<Integer> res = new ArrayList<>();
  //终止条件判断
  if (root == null)
    return res;
  //创建队列
  Queue<TreeNode> queue = new LinkedList();
  queue.offer(root);
  while (!queue.isEmpty()) {
    //每层的数量
    int count = queue.size();
    while (count-- > 0) {
      //当前节点出队
      TreeNode cur = queue.poll();
      //因为每层是从左往右依次入队的，所以每层的
      //最后一个就是我们所需要的
      if (count == 0)
        res.add(cur.val);
      //左子树如果不为空，左子树入队，右子树如果不为空
      //右子树入队
      if (cur.left != null)
        queue.offer(cur.left);
      if (cur.right != null)
        queue.offer(cur.right);
    }
  }
  return res;
}
```

总结

对于二叉树的一些常见遍历一定要熟练掌握，总共加起来也就那8种，不是很多，如果掌握了那些遍历方式，对于二叉树的一些算法题只要是稍加修改基本上都能做的出来。

往期推荐

- 444，二叉树的序列化与反序列化
- 442，剑指 Offer-回溯算法解二叉树中和为某一值的路径
- 440，剑指 Offer-从上到下打印二叉树 II
- 372，二叉树的最近公共祖先

444，二叉树的序列化与反序列化

原创 山大王wld 数据结构和算法 8月27日

收录于话题
#算法图文分析

95个 >



It's easy to find if you know what you're looking for.

如果你知道自己想追求什么，就很容易成功。

问题描述

序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。

请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

示例：

> 你可以将以下二叉树：
>
> 1
> / \
> 2 3

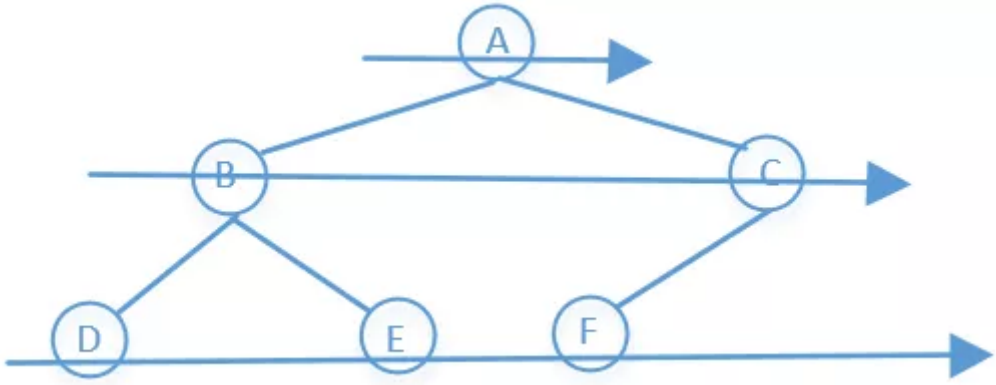
```
  / \
 4   5
```

序列化为 "[1,2,3,null,null,4,5]"

BFS解决

这题上面说了一大堆，其实就是把二叉树转化为一个字符串，并且还能把这个字符串还原成原来的二叉树就可以了。

把二叉树转化为字符串可以有很多种方式，比如前序遍历，中序遍历，后续遍历，BFS，DFS都是可以的，对于树的各种遍历具体可以看下373，数据结构-6,树。但这题还要求把字符串再还原成原来的二叉树。最容易想到的就是BFS，就是一层一层从往下遍历



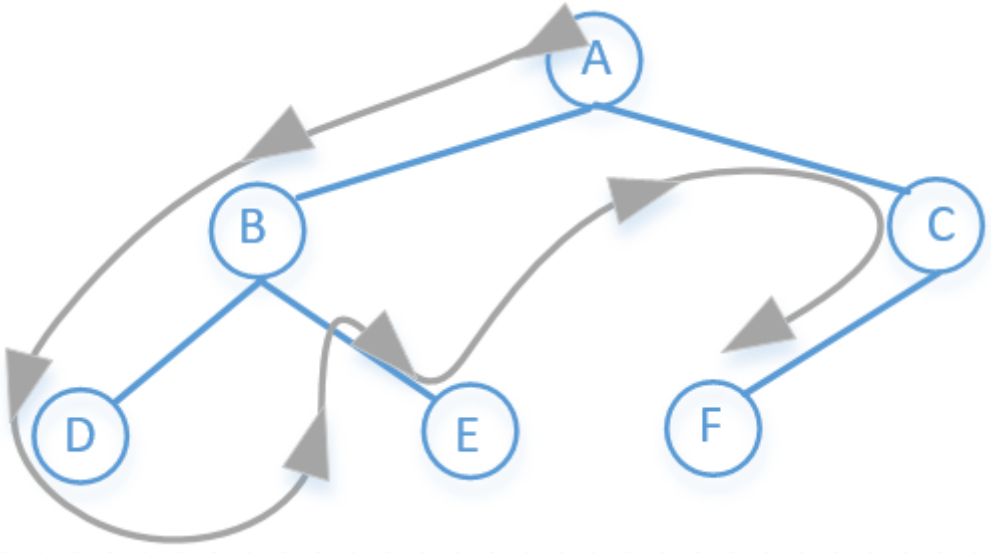
来看下代码

```java
public class Codec {

  //把树转化为字符串（使用BFS遍历）
  public String serialize(TreeNode root) {
    //边界判断，如果为空就返回一个字符串"#"
    if (root == null)
      return "#";
    //创建一个队列
    Queue<TreeNode> queue = new LinkedList<>();
    StringBuilder res = new StringBuilder();
    //把根节点加入到队列中
    queue.add(root);
    while (!queue.isEmpty()) {
      //节点出队
      TreeNode node = queue.poll();
      //如果节点为空，添加一个字符"#"作为空的节点
      if (node == null) {
        res.append("#,");
        continue;
      }
      //如果节点不为空，把当前节点的值加入到字符串中，
      //注意节点之间都是以逗号","分隔的，在下面把字符
      //串还原二叉树的时候也是以逗号","把字符串进行拆分
      res.append(node.val + ",");
      //左子节点加入到队列中（左子节点有可能为空）
      queue.add(node.left);
      //右子节点加入到队列中（右子节点有可能为空）
```

```
            queue.add(node.right);
        }
        return res.toString();
    }

    //把字符串还原为二叉树
    public TreeNode deserialize(String data) {
        //如果是"#"，就表示一个空的节点
        if (data == "#")
            return null;
        Queue<TreeNode> queue = new LinkedList<>();
        //因为上面每个节点之间是以逗号","分隔的，所以这里
        //也要以逗号","来进行拆分
        String[] values = data.split(",");
        //上面使用的是BFS，所以第一个值就是根节点的值，这里创建根节点
        TreeNode root = new TreeNode(Integer.parseInt(values[0]));
        queue.add(root);
        for (int i = 1; i < values.length; i++) {
            //队列中节点出栈
            TreeNode parent = queue.poll();
            //因为在BFS中左右子节点是成对出现的，所以这里挨着的两个值一个是
            //左子节点的值一个是右子节点的值，当前值如果是"#"就表示这个子节点
            //是空的，如果不是"#"就表示不是空的
            if (!"#".equals(values[i])) {
                TreeNode left = new TreeNode(Integer.parseInt(values[i]));
                parent.left = left;
                queue.add(left);
            }
            //上面如果不为空就是左子节点的值，这里是右子节点的值，注意这里有个i++，
            if (!"#".equals(values[++i])) {
                TreeNode right = new TreeNode(Integer.parseInt(values[i]));
                parent.right = right;
                queue.add(right);
            }
        }
        return root;
    }
}
```

DFS解决

DFS遍历是从根节点开始，一直往左子节点走，当到达叶子节点的时候会返回到父节点，然后从从父节点的右子节点继续遍历……




```
class Codec {

    //把树转化为字符串（使用DFS遍历，也是前序遍历，顺序是：根节点→左子树→右子树）
```

```
    public String serialize(TreeNode root) {
        //边界判断，如果为空就返回一个字符串"#"
        if (root == null)
            return "#";
        return root.val + "," + serialize(root.left) + "," + serialize(root.right);
    }

    //把字符串还原为二叉树
    public TreeNode deserialize(String data) {
        //把字符串data以逗号","拆分，拆分之后存储到队列中
        Queue<String> queue = new LinkedList<>(Arrays.asList(data.split(",")));
        return helper(queue);
    }

    private TreeNode helper(Queue<String> queue) {
        //出队
        String sVal = queue.poll();
        //如果是"#"表示空节点
        if ("#".equals(sVal))
            return null;
        //否则创建当前节点
        TreeNode root = new TreeNode(Integer.valueOf(sVal));
        //分别创建左子树和右子树
        root.left = helper(queue);
        root.right = helper(queue);
        return root;
    }
}
```

总结

把二叉树转化为字符串很简单，关键是怎么把转化的字符串再还原成原来的二叉树，这里使用BFS和DFS都很容易实现。

往期推荐

- 442，剑指 Offer-回溯算法解二叉树中和为某一值的路径
- 440，剑指 Offer-从上到下打印二叉树 II
- 434，剑指 Offer-二叉树的镜像
- 387，二叉树中的最大路径和

441，剑指 Offer-二叉搜索树的后序遍历序列

原创 山大王wld 数据结构和算法 8月21日

收录于话题
#剑指offer

27个 >



Maybe you don't have to do this all by yourself, mate.

也许你没必要一个人扛，哥们儿。

问题描述

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历结果。如果是则返回 true，否则返回 false。假设输入的数组的任意两个数字都互不相同。

参考以下这颗二叉搜索树：

```
    5
   / \
  2   6
 / \
1   3
```

示例 1：

```
输入: [1,6,3,2,5]
输出: false
```

示例 2：

```
输入: [1,3,2,6,5]
输出: true
```

提示：

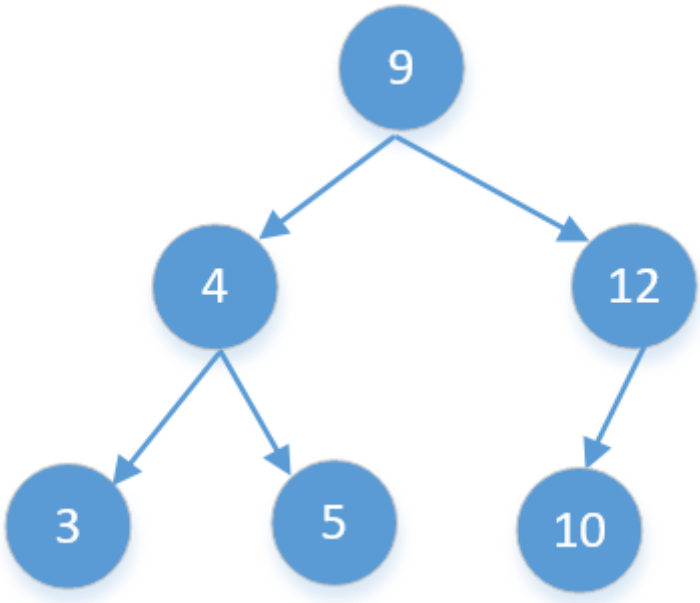
1. 数组长度 <= 1000

递归方式解决

如果这题说的是判断该数组是不是某二叉搜索树的中序遍历结果，那么这道题就非常简单了，因为二叉搜索树的中序遍历结果一定是有序的，我们只需要判断数组是否有序就行了。但这道题要判断的是不是某二叉搜索树的后序遍历结果，这样就有点难办了。

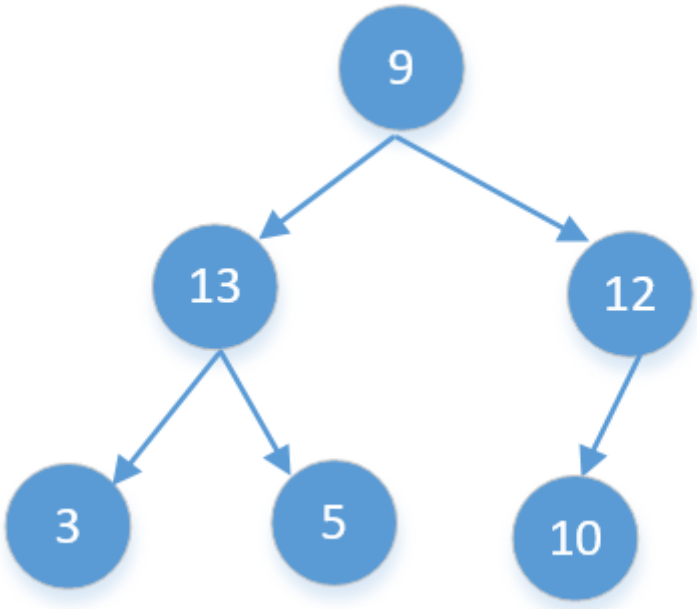
二叉搜索树的特点是左子树的值<根节点<右子树的值。而后续遍历的顺序是：
左子节点→右子节点→根节点；

比如下面这棵二叉树，他的后续遍历是
[3，5，4，10，12，9]



我们知道后续遍历的最后一个数字一定是根节点，所以数组中最后一个数字9就是根节点，我们从前往后找到第一个比9大的数字10，那么10后面的[10，12]（除了9）都是9的右子节点，10前面的[3，5，4]都是9的左子节点，后面的需要判断一下，如果有小于9的，说明不是二叉搜索树，直接返回false。然后再以递归的方式判断左右子树。

再来看一个，他的后续遍历是[3，5，13，10，12，9]

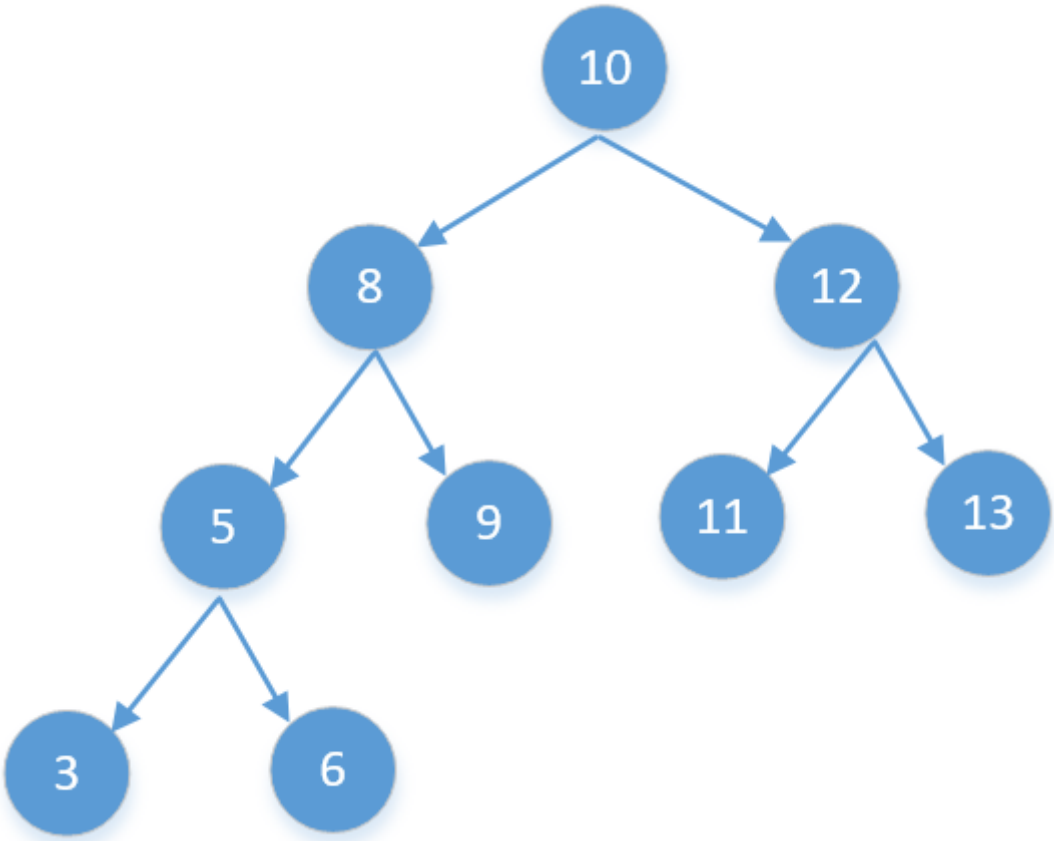


我们来根据数组拆分，第一个比9大的后面都是9的右子节点[13，10，12]。然后再拆分这个数组，12是根节点，第一个比12大的后面都是12的右子节点[13，10]，但我们看到10是比12小的，他不可能是12的右子节点，所以我们能确定这棵树不是二叉搜索树。搞懂了上面的原理我们再来看下代码。

```
public boolean verifyPostorder(int[] postorder) {
  return helper(postorder, 0, postorder.length - 1);
}

boolean helper(int[] postorder, int left, int right) {
  //如果left==right，就一个节点不需要判断了，如果left>right说明没有节点，
  //也不用再看了,否则就要继续往下判断
  if (left >= right)
    return true;
  //因为数组中最后一个值postorder[right]是根节点，这里从左往右找出第一个比
  //根节点大的值，他后面的都是根节点的右子节点（包含当前值，不包含最后一个值，
  //因为最后一个是根节点），他前面的都是根节点的左子节点
  int mid = left;
  int root = postorder[right];
  while (postorder[mid] < root)
    mid++;
  int temp = mid;
  //因为postorder[mid]前面的值都是比根节点root小的，
  //我们还需要确定postorder[mid]后面的值都要比根节点root大，
  //如果后面有比根节点小的直接返回false
  while (temp < right) {
    if (postorder[temp++] < root)
      return false;
  }
  //然后对左右子节点进行递归调用
  return helper(postorder, left, mid - 1) && helper(postorder, mid, right - 1);
}
```

使用栈解决

我们先来画一个节点多一些的二叉搜索树，然后观察一下他的规律



他的后续遍历结果是
[3,6,5,9,8,11,13,12,10]

从前往后不好看，我们来从后往前看
[10,12,13,11,8,9,5,6,3]
如果你仔细观察会发现一个规律，就是挨着的两个数如果arr[i]<arr[i+1]，那么arr[i+1]一定是arr[i]的右子节点，这一点是毋庸置疑的，我们可以看下上面的10和12是挨着的并且10<12，所以12是10的右子节点。同理12和13，8和9，5和6，他们都是挨着的，并且前面的都是小于后面的，所以后面的都是前面的右子节点。如果想证明也很简单，因为比arr[i]大的肯定都是他的右子节点，如果还是挨着他的，肯定是在后续遍历中所有的右子节点最后一个遍历的，所以他一定是arr[i]的右子节点。

我们刚才看的是升序的，再来看一下降序的（这里的升序和降序都是基于后续遍历从后往前看的，也就是上面蓝色数组）。如果arr[i]>arr[i+1]，那么arr[i+1]一定是arr[0]……arr[i]中某个节点的左子节点，并且这个值是大于arr[i+1]中最小的。我们来看一下上面的数组，比如13，11是降序的，那么11肯定是他前面某一个节点的左子节点，并且这个值是大于11中最小的，我们看到12和13都是大于11的，但12最小，所以11就是12的左子节点。同理我们可以观察到11和8是降序，8前面大于8中最小的是10，所以8就是10的左子节点。9和5是降序，6和3是降序，都遵守这个规律。

根据上面分析的过程，很容易想到使用栈来解决。遍历数组的所有元素，如果栈为空，就把当前元素压栈。如果栈不为空，并且当前元素大于栈顶元素，说明是升序的，那么

就说明当前元素是栈顶元素的右子节点，就把当前元素压栈，如果一直升序，就一直压栈。当前元素小于栈顶元素，说明是倒序的，说明当前元素是某个节点的左子节点，我们目的是要找到这个左子节点的父节点，就让栈顶元素出栈，直到栈为空或者栈顶元素小于当前值为止，其中最后一个出栈的就是当前元素的父节点。我们来看下代码

```
public boolean verifyPostorder(int[] postorder) {
  Stack<Integer> stack = new Stack<>();
  int parent = Integer.MAX_VALUE;
  //注意for循环是倒叙遍历的
  for (int i = postorder.length - 1; i >= 0; i--) {
    int cur = postorder[i];
    //当如果前节点小于栈顶元素，说明栈顶元素和当前值构成了倒叙，
    //说明当前节点是前面某个节点的左子节点，我们要找到他的父节点
    while (!stack.isEmpty() && stack.peek() > cur)
      parent = stack.pop();
    //只要遇到了某一个左子节点，才会执行上面的代码，才会更
    //新parent的值，否则parent就是一个非常大的值，也就
    //是说如果一直没有遇到左子节点，那么右子节点可以非常大
    if (cur > parent)
      return false;
    //入栈
    stack.add(cur);
  }
  return true;
}
```

上面代码可能大家有点蒙的是if(cur>parent)这一行的判断。二叉搜索树应该是左子节点小于根节点，右子节点大于根节点，但上面为什么大于父节点的时候要返回false，注意这里的parent是在什么情况下赋的值，parent并不一定都是父节点的值，相对于遇到了左子节点的时候他是左子节点的父节点。如果是右子节点，parent就是他的某一个祖先节点，并且这个右子节点是这个祖先节点的一个左子树的一部分，所以不能超过他，有点绕，慢慢体会。

总结

这题第一种方式是最容易想到的，每次把数组劈两半，因为通过第一个while循环，左边的都是小于根节点的，然后再判断右边的是不是都大于根节点，然后左右两边再以同样的方式计算……。第二种方式也能解决，但比较绕，相对来说不太好容易理解，但如果真的搞懂了，会豁然开朗，也会有很大的收获。

往期推荐

- 400，二叉树的锯齿形层次遍历
- 399，从前序与中序遍历序列构造二叉树
- 387，二叉树中的最大路径和
- 374，二叉树的最小深度

440，剑指 Offer-从上到下打印二叉树 II

原创 山大王wld 数据结构和算法 8月21日

收录于话题
#剑指offer

27个 >



You should never judge something you don't understand.

你不应该去评判你不了解的事物。

问题描述

从上到下按层打印二叉树，同一层的节点按从左到右的顺序打印，每一层打印到一行。

例如:
给定二叉树: [3,9,20,null,null,15,7],

```
  3
 /\
9  20
  / \
 15  7
```

返回其层次遍历结果:

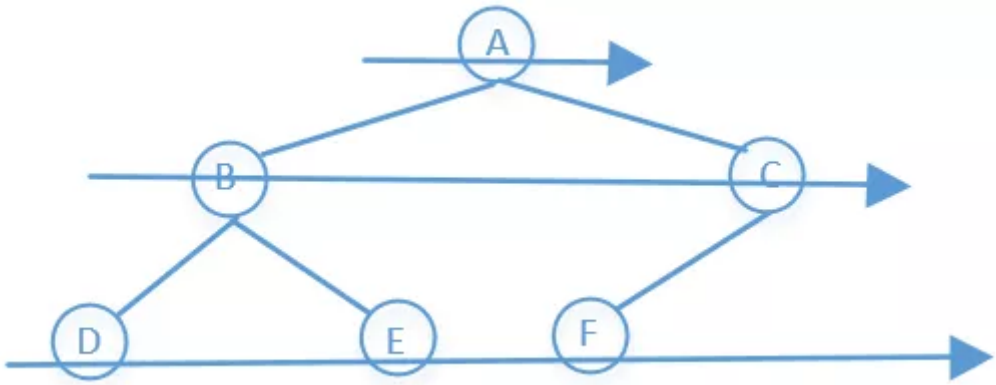
```
[
  [3],
  [9,20],
  [15,7]
]
```

提示：

1. 节点总数 <= 1000

BFS解决

这题和上一题439，剑指 Offer-从上到下打印二叉树其实是一样的，只不过上一题返回的是数组，这一题返回的是list。返回数组，我们还要初始化数组，但不知道数组的大小，所以一般是先储存在list中再转化为数组，返回list就比较简单了。




```java
public List<List<Integer>> levelOrder(TreeNode root) {
  //边界条件判断
  if (root == null)
    return new ArrayList<>();
  //队列
  Queue<TreeNode> queue = new LinkedList<>();
  List<List<Integer>> res = new ArrayList<>();
  //根节点入队
  queue.add(root);
  //如果队列不为空就继续循环
  while (!queue.isEmpty()) {
    //BFS打印，levelNum表示的是每层的结点数
    int levelNum = queue.size();
    //subList存储的是每层的结点值
    List<Integer> subList = new ArrayList<>();
    for (int i = 0; i < levelNum; i++) {
      //出队
      TreeNode node = queue.poll();
      subList.add(node.val);
      //左右子节点如果不为空就加入到队列中
      if (node.left != null)
        queue.add(node.left);
      if (node.right != null)
        queue.add(node.right);
    }
    //把每层的结点值存储在res中，
    res.add(subList);
```

```
    }
    return res;
}
```

DFS解决

这题让一层一层的打印，其实就是BFS，但使用DFS也是可以解决的，看一下

```
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> res = new ArrayList<>();
    levelHelper(res, root, 0);
    return res;
}

public void levelHelper(List<List<Integer>> list, TreeNode root, int level) {
    //边界条件判断
    if (root == null)
        return;
    //level表示的是层数，如果level >= list.size()，说明到下一层了，所以
    //要先把下一层的list初始化，防止下面add的时候出现空指针异常
    if (level >= list.size()) {
        list.add(new ArrayList<>());
    }
    //level表示的是第几层，这里访问到第几层，我们就把数据加入到第几层
    list.get(level).add(root.val);
    //当前节点访问完之后，再使用递归的方式分别访问当前节点的左右子节点
    levelHelper(list, root.left, level + 1);
    levelHelper(list, root.right, level + 1);
}
```

总结

这题其实就是二叉树的宽度优先搜索，前面讲373，数据结构-6,树的时候也提到过树的各种遍历方式。

往期推荐

- 439，剑指 Offer-从上到下打印二叉树
- 435，剑指 Offer-对称的二叉树
- 434，剑指 Offer-二叉树的镜像
- 414，剑指 Offer-重建二叉树

439，剑指 Offer-从上到下打印二叉树

原创 山大王wld 数据结构和算法 8月20日

收录于话题
#剑指offer

27个 >



Happiness can be found, even in the darkest of times.

可我们总能找到快乐，哪怕处在最黑暗的时期。

问题描述

从上到下打印出二叉树的每个节点，同一层的节点按照从左到右的顺序打印。

例如:
给定二叉树: `[3,9,20,null,null,15,7]`,

```
  3
 /\
9  20
  / \
 15  7
```

返回:

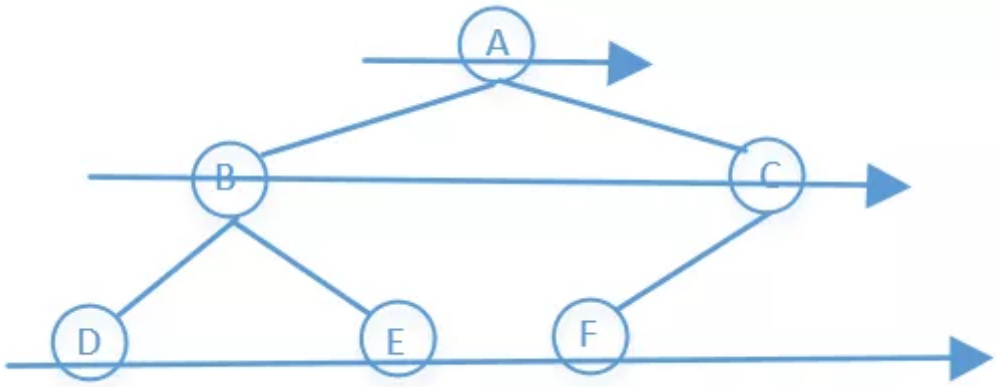
```
[3,9,20,15,7]
```

提示：

1. 节点总数 <= 1000

BFS解决

其实这就是二叉树的BFS，也可以看下之前讲的373，数据结构-6,树，



就是这样，一层一层打印，使用队列解决

```
public int[] levelOrder(TreeNode root) {
  if (root == null)
    return new int[0];
  //队列
  Queue<TreeNode> queue = new LinkedList<>();
  List<Integer> list = new ArrayList<>();
  //根节点入队
  queue.add(root);
  while (!queue.isEmpty()) {
    //出队
    TreeNode node = queue.poll();
    //把结点值存放到list中
    list.add(node.val);
    //左右子节点如果不为空就加入到队列中
    if (node.left != null)
      queue.add(node.left);
    if (node.right != null)
      queue.add(node.right);
  }
  //把list转化为数组
  int[] res = new int[list.size()];
  for (int i = 0; i < list.size(); i++) {
    res[i] = list.get(i);
  }
  return res;
}
```

递归方式解决

其实这题很明显就是二叉树的宽度优先搜索，使用上面代码就对了。实际上我们还可以改一下，改成DFS并且还是递归的，我想除了我应该没人会这么无聊吧，有兴趣的也可以看下

```
public int[] levelOrder(TreeNode root) {
  List<List<Integer>> list = new ArrayList<>();
```

```
    levelHelper(list, root, 0);
    List<Integer> tempList = new ArrayList<>();
    for (int i = 0; i < list.size(); i++) {
      tempList.addAll(list.get(i));
    }

    //把list转化为数组
    int[] res = new int[tempList.size()];
    for (int i = 0; i < tempList.size(); i++) {
      res[i] = tempList.get(i);
    }
    return res;
  }

  public void levelHelper(List<List<Integer>> list, TreeNode root, int height) {
    if (root == null)
      return;
    if (height >= list.size()) {
      list.add(new ArrayList<>());
    }
    list.get(height).add(root.val);
    levelHelper(list, root.left, height + 1);
    levelHelper(list, root.right, height + 1);
  }
```

总结

这题实际上就是二叉树的宽度优先遍历，一层一层打印。

往期推荐

- 414，剑指 Offer-重建二叉树
- 400，二叉树的锯齿形层次遍历
- 373，数据结构-6,树
- 372，二叉树的最近公共祖先

435，剑指 Offer-对称的二叉树

原创 山大王wld 数据结构和算法 8月18日

收录于话题
#剑指offer

27个 >



It takes a great deal of bravery to stand up to your enemies, but a great deal more to stand up to your friends.

挺身而出对抗敌人需要勇气，但在朋友面前坚定立场，更需要勇气。

问题描述

请实现一个函数，用来判断一棵二叉树是不是对称的。如果一棵二叉树和它的镜像一样，那么它是对称的。

例如，二叉树 [1,2,2,3,4,4,3] 是对称的。

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

但是下面这个 [1,2,2,null,3,null,3] 则不是镜像对称的:

```
    1
   / \
  2   2
   \   \
   3    3
```

示例 1：

```
输入：root = [1,2,2,3,4,4,3]
输出：true
```

示例 2：

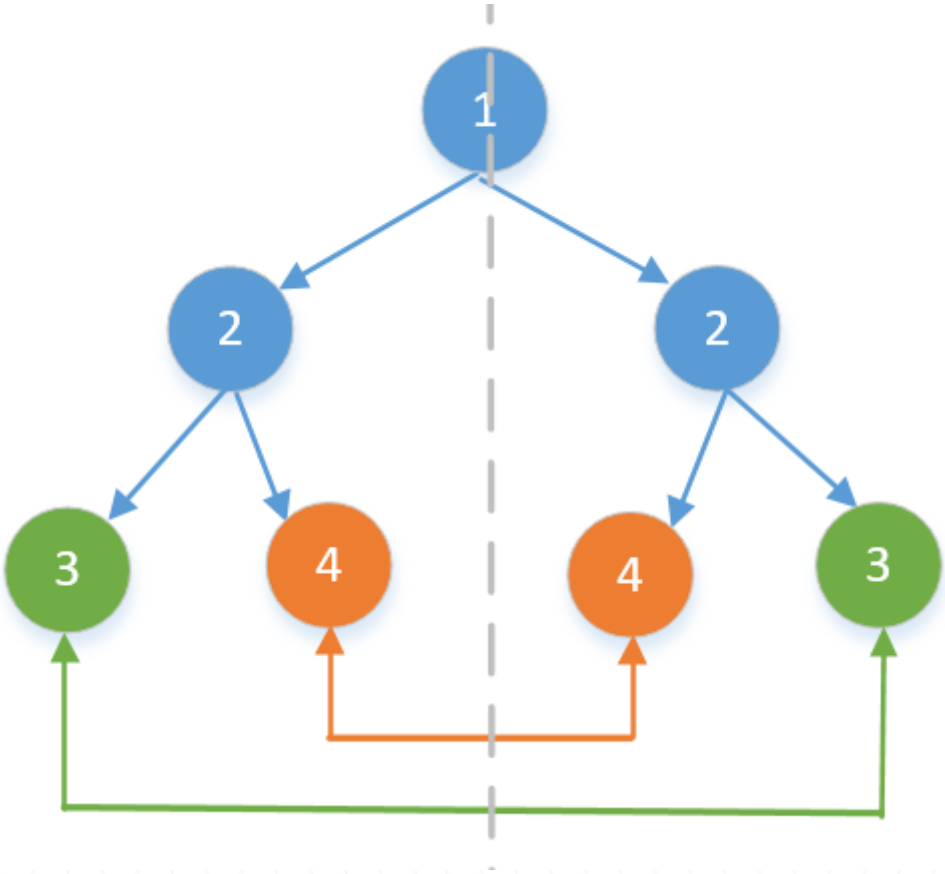
```
输入：root = [1,2,2,null,3,null,3]
输出：false
```

限制：

0 <= 节点个数 <= 1000

递归解决

判断二叉树是否是对称，需要从子节点开始比较，两个子节点的值必须相同，并且左子节点的右子节点（如果有）必须等于右子节点的左子节点，左子节点的左子节点必须等于右子节点的右子节点。就像下面图中那样




```java
public boolean isSymmetric(TreeNode root) {
    if (root == null)
        return true;
    //从两个子节点开始判断
    return isSymmetricHelper(root.left, root.right);
}

public boolean isSymmetricHelper(TreeNode left, TreeNode right) {
    //如果左右子节点都为空，说明当前节点是叶子节点，返回true
    if (left == null && right == null)
        return true;
    //如果当前节点只有一个子节点或者有两个子节点，但两个子节点的值不相同，直接返回false
    if (left == null || right == null || left.val != right.val)
        return false;
    //然后左子节点的左子节点和右子节点的右子节点比较，左子节点的右子节点和右子节点的左子节点比较
    return isSymmetricHelper(left.left, right.right) && isSymmetricHelper(left.right, right.left);
}
```

非递归解决

非递归解决和上面原理一样，直接看下代码

```java
public boolean isSymmetric(TreeNode root) {
    //队列
    Queue<TreeNode> queue = new LinkedList<>();
    if (root == null)
        return true;
    //左子节点和右子节点同时入队
    queue.add(root.left);
    queue.add(root.right);
    //如果队列不为空就继续循环
    while (!queue.isEmpty()) {
        //每两个出队
        TreeNode left = queue.poll(), right = queue.poll();
        //如果都为空继续循环
        if (left == null && right == null)
```

```
            continue;
        //如果一个为空一个不为空，说明不是对称的，直接返回false
        if (left == null ^ right == null)
            return false;
        //如果这两个值不相同，也不是对称的，直接返回false
        if (left.val != right.val)
            return false;
        //这里要记住入队的顺序，他会每两个两个的出队。
        //左子节点的左子节点和右子节点的右子节点同时
        //入队，因为他俩会同时比较。
        //左子节点的右子节点和右子节点的左子节点同时入队，
        //因为他俩会同时比较
        queue.add(left.left);
        queue.add(right.right);
        queue.add(left.right);
        queue.add(right.left);
    }
    return true;
}
```

总结

树的镜像判断，首先需要要找准判断的两个节点，然后再比较，比较完之后如果值相同，还有继续比较两个子节点。

往期推荐

- 400，二叉树的锯齿形层次遍历
- 387，二叉树中的最大路径和
- 374，二叉树的最小深度
- 373，数据结构-6,树

434，剑指 Offer-二叉树的镜像

原创 山大王wld 数据结构和算法 8月18日

收录于话题
#剑指offer

27个 >



The more you know who you are and what you want, the less you let things upset you.

你越了解自己以及自己想要的东西，你就越不会被外界困扰。

问题描述

请完成一个函数，输入一个二叉树，该函数输出它的镜像。

例如输入：

```
    4
   / \
  2   7
 /\   /\
1  3 6  9
```

镜像输出：

```
    4
   / \
  7   2
```

```
 /\  /\
9  6 3  1
```

示例 1：

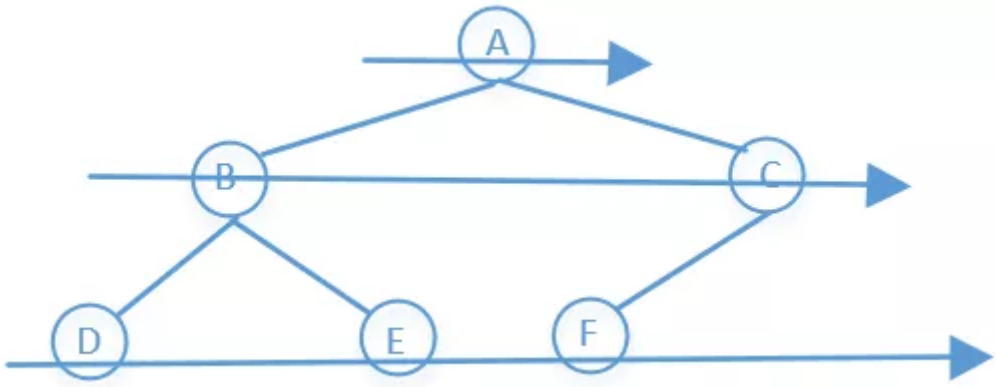
```
输入：root = [4,2,7,1,3,6,9]
输出：[4,7,2,9,6,3,1]
```

限制：

```
0 <= 节点个数 <= 1000
```

BFS解决

之前讲[373，数据结构-6,树](#)的时候，提到过二叉树的广度优先搜索，就是一层一层的访问，像下面这样

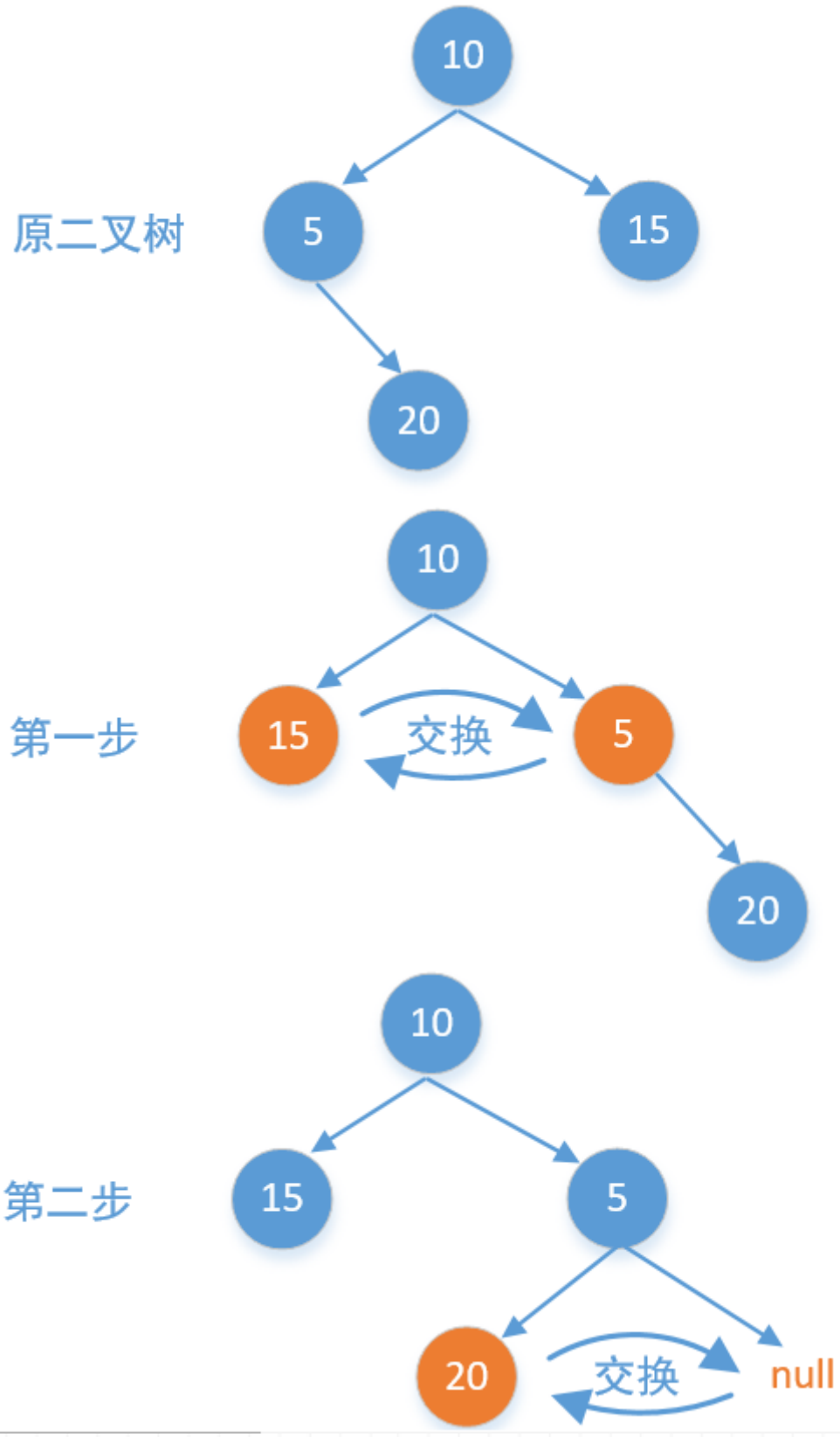


二叉树的BFS代码如下

```java
public static void treeBFS(TreeNode root) {
  //如果为空直接返回
  if (root == null)
    return;
  //队列
  Queue<TreeNode> queue = new LinkedList<>();
  //首先把根节点加入到队列中
  queue.add(root);
  //如果队列不为空就继续循环
  while (!queue.isEmpty()) {
    //poll方法相当于移除队列头部的元素
    TreeNode node = queue.poll();
    //打印当前节点
    System.out.println(node.val);
    //如果当前节点的左子树不为空，就把左子树
    //节点加入到队列中
    if (node.left != null)
      queue.add(node.left);
    //如果当前节点的右子树不为空，就把右子树
    //节点加入到队列中
    if (node.right != null)
      queue.add(node.right);
```

```
23      }
24  }
```

这题要求的是输出二叉树的镜像，就是每一个节点的左右子节点进行交换，随便画个二叉树看一下



我们需要遍历每一个节点，然后交换他的两个子节点，一直循环下去，直到所有的节点都遍历完为止，代码如下

```
1   public TreeNode mirrorTree(TreeNode root) {
```

```
    //如果为空直接返回
    if (root == null)
      return null;
    //队列
    final Queue<TreeNode> queue = new LinkedList<>();
    //首先把根节点加入到队列中
    queue.add(root);
    while (!queue.isEmpty()) {
      //poll方法相当于移除队列头部的元素
      TreeNode node = queue.poll();
      //交换node节点的两个子节点
      TreeNode left = node.left;
      node.left = node.right;
      node.right = left;
      //如果当前节点的左子树不为空，就把左子树
      //节点加入到队列中
      if (node.left != null) {
        queue.add(node.left);
      }
      //如果当前节点的右子树不为空，就把右子树
      //节点加入到队列中
      if (node.right != null) {
        queue.add(node.right);
      }
    }
    return root;
  }
```

DFS解决

无论是BFS还是DFS都会访问到每一个节点，访问每个节点的时候交换他的左右子节点，直到所有的节点都访问完为止，代码如下

```
  public TreeNode mirrorTree(TreeNode root) {//DFS
    //如果为空直接返回
    if (root == null)
      return null;
    //栈
    Stack<TreeNode> stack = new Stack<>();
    //根节点压栈
    stack.push(root);
    //如果栈不为空就继续循环
    while (!stack.empty()) {
      //出栈
      TreeNode node = stack.pop();
      //子节点交换
      TreeNode temp = node.left;
      node.left = node.right;
      node.right = temp;
      //左子节点不为空入栈
      if (node.left != null)
        stack.push(node.left);
      //右子节点不为空入栈
      if (node.right != null)
        stack.push(node.right);
    }
    return root;
  }
```

中序遍历解决

这题其实解法比较多，只要访问他的每一个节点，然后交换子节点即可，我们知道二叉树不光有BFS和DFS访问顺序，而且还有前序遍历，中序遍历和后续遍历等，不管哪种访问方式，只要能把所有节点都能访问一遍然后交换子节点就能解决，我们这里就以中序遍历来看下，前序和后序就不在看了。在373，数据结构-6,树中，提到二叉树中序遍历的非递归写法如下

```
public static void inOrderTraversal(TreeNode tree) {
  Stack<TreeNode> stack = new Stack<>();
  while (tree != null || !stack.isEmpty()) {
    while (tree != null) {
      stack.push(tree);
      tree = tree.left;
    }
    if (!stack.isEmpty()) {
      tree = stack.pop();
      System.out.println(tree.val);
      tree = tree.right;
    }
  }
}
```

我们来对他改造一下，就是在访问每个节点的时候交换，代码如下

```
public static TreeNode mirrorTree(TreeNode root) {
  //如果为空直接返回
  if (root == null)
    return null;
  Stack<TreeNode> stack = new Stack<>();
  TreeNode node = root;
  while (node != null || !stack.isEmpty()) {
    while (node != null) {
      stack.push(node);
      node = node.left;
    }
    if (!stack.isEmpty()) {
      node = stack.pop();
      //子节点交换
      TreeNode temp = node.left;
      node.left = node.right;
      node.right = temp;
      //注意这里以前是node.right，因为上面已经交换了
      //，所以这里要改为node.left
      node = node.left;
    }
  }
  return root;
}
```

递归方式解决

二叉树中序遍历的递归代码如下

```
public void inOrderTraversal(TreeNode node) {
  if (node == null)
    return;
  inOrderTraversal(node.left);
  System.out.println(node.val);
  inOrderTraversal(node.right);
}
```

上面说了，只要能访问二叉树的每一个节点，然后交换左右子节点就行了，这里就以二叉树中序遍历递归的方式来看下

```
public TreeNode mirrorTree(TreeNode root) {
```

```
    if (root == null)
      return null;
    mirrorTree(root.left);
    //子节点交换
    TreeNode temp = root.left;
    root.left = root.right;
    root.right = temp;
    //上面交换过了，这里root.right要变成root.left
    mirrorTree(root.left);
    return root;
  }
```

再来看一个后续遍历的

```
public TreeNode mirrorTree(TreeNode root) {
  if (root == null)
    return null;
  TreeNode left = mirrorTree(root.left);
  TreeNode right = mirrorTree(root.right);
  root.left = right;
  root.right = left;
  return root;
}
```

总结

这题没什么难度，但解法比较多，主要是因为二叉树的遍历方式比较多，如果每一种方式递归和非递归都写的话就更多了。

往期推荐

- 433，剑指 Offer-树的子结构
- 414，剑指 Offer-重建二叉树
- 403，验证二叉搜索树
- 401，删除二叉搜索树中的节点

433，剑指 Offer-树的子结构

原创 山大王wld 数据结构和算法 8月17日

收录于话题
#剑指offer

27个 >



No amount of money ever bought a second of time.

再多的钱也无法买回逝去的光阴。

问题描述

输入两棵二叉树A和B，判断B是不是A的子结构。(约定空树不是任意一个树的子结构)

B是A的子结构， 即A中有出现和B相同的结构和节点值。

例如:
给定的树 A:

```
     3
    / \
   4   5
  / \
 1   2
```

给定的树 B:

```
  4
 /
1
```

返回 true，因为 B 与 A 的一个子树拥有相同的结构和节点值。

示例 1：

```
输入：A = [1,2,3], B = [3,1]
输出：false
```

示例 2：

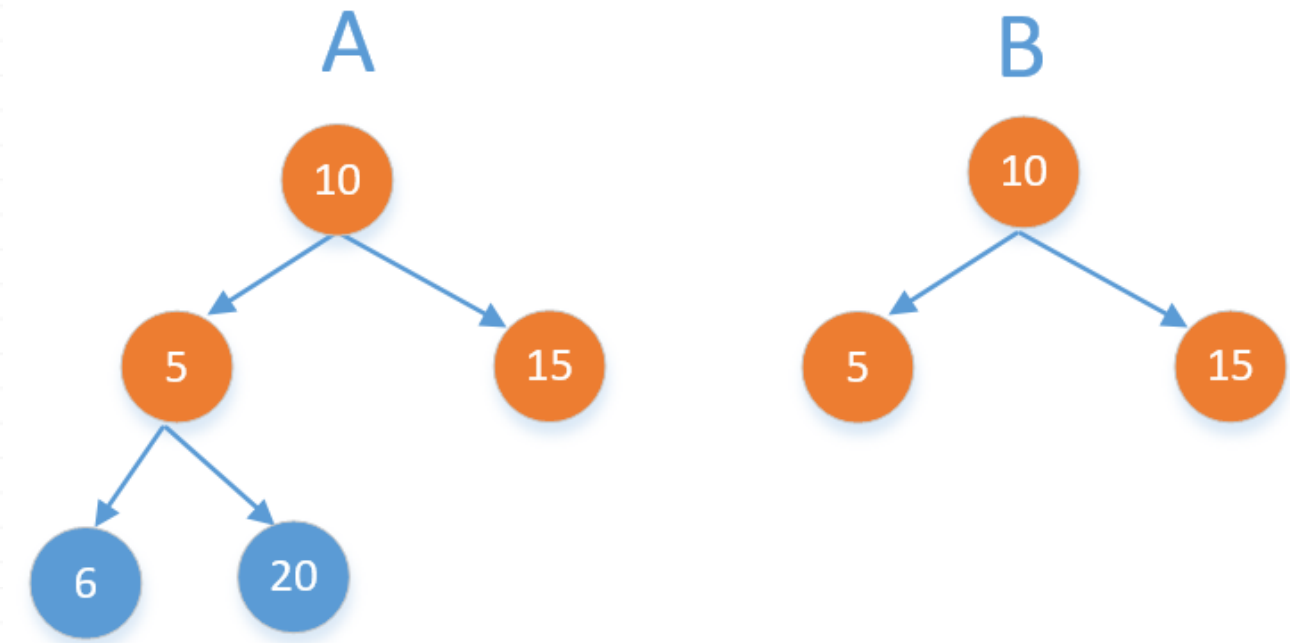
```
输入：A = [3,4,5,1,2], B = [4,1]
输出：true
```

限制：

0 <= 节点个数 <= 10000

问题分析

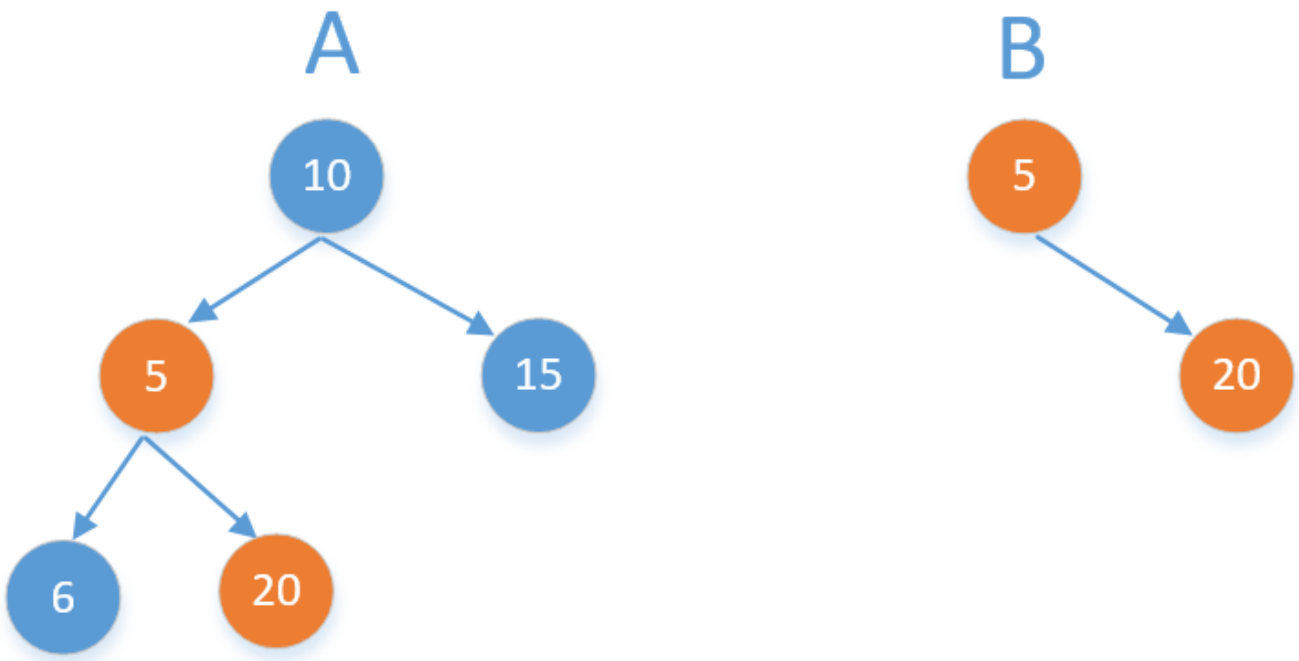
要判断B是否是A的子结构，像下面这样，我们只需要从根节点开始判断，通过递归的方式比较他的每一个子节点即可，所以代码也很容易写




```
1   public boolean isSubStructure(TreeNode A, TreeNode B) {
```

```
    //边界条件判断，如果A和B有一个为空，返回false
    if (A == null || B == null)
      return false;
    return isSub(A, B);
  }

  boolean isSub(TreeNode A, TreeNode B) {
    //这里如果B为空，说明B已经访问完了，确定是A的子结构
    if (B == null)
      return true;
    //如果B不为空A为空，或者这两个节点值不同，说明B树不是
    //A的子结构，直接返回false
    if (A == null || A.val != B.val)
      return false;
    //当前节点比较完之后还要继续判断左右子节点
    return isSub(A.left, B.left) && isSub(A.right, B.right);
  }
```

但实际上B如果是A的子结构的话，不一定是从根节点开始的，也可能是下面这样



也就是说B不光有可能是A的子结构，也有可能是A左子树的子结构或者右子树的子结构，所以如果从根节点判断B不是A的子结构，还要继续判断B是不是A左子树的子结构和右子树的子结构，代码如下

```
  public boolean isSubStructure(TreeNode A, TreeNode B) {
    if (A == null || B == null)
      return false;
    //先从根节点判断B是不是A的子结构，如果不是在分别从左右两个子树判断，
    //只要有一个为true，就说明B是A的子结构
    return isSub(A, B) || isSubStructure(A.left, B) || isSubStructure(A.right, B);
  }

  boolean isSub(TreeNode A, TreeNode B) {
    //这里如果B为空，说明B已经访问完了，确定是A的子结构
    if (B == null)
      return true;
    //如果B不为空A为空，或者这两个节点值不同，说明B树不是
    //A的子结构，直接返回false
    if (A == null || A.val != B.val)
      return false;
    //当前节点比较完之后还要继续判断左右子节点
    return isSub(A.left, B.left) && isSub(A.right, B.right);
  }
```

总结

B是A的子结构不一定是从根节点开始判断B是否是A的子结构，也有可能B是A左子树或右子树的子结构，所以如果从根节点判断B不是A的子结构的时候还要分别判断A的子树中是否包含B。

往期推荐

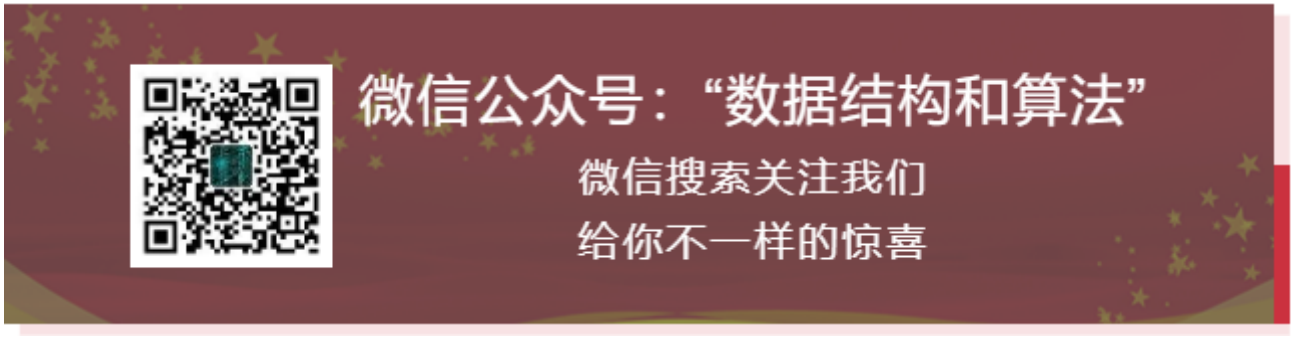
- 374，二叉树的最小深度
- 373，数据结构-6,树
- 372，二叉树的最近公共祖先
- 367，二叉树的最大深度

414，剑指 Offer-重建二叉树

原创 山大王wld 数据结构和算法 7月29日

收录于话题
#剑指offer

27个 >



Tough time don't last, tough people do.

没有过不去的坎，只有打不倒的人。

问题描述

输入某二叉树的前序遍历和中序遍历的结果，请重建该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。

例如，给出

```
前序遍历 preorder = [3,9,20,15,7]
中序遍历 inorder = [9,3,15,20,7]
```

返回如下的二叉树：

```
  3
 / \
9  20
  /  \
```

```
15   7
```

限制：

- 0 <= 节点个数 <= 5000

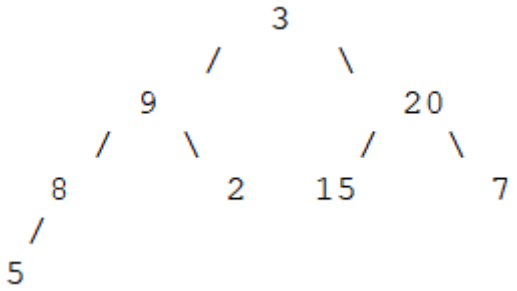
问题分析

这题和之前讲过的一道题重复了，399，从前序与中序遍历序列构造二叉树，这两道题其实是完全一样的，除了之前讲过的3种方法以外，我们今天再来讲一种解法，这种思想来源于403，验证二叉搜索树的第一种解法。前序遍历的第一个元素肯定是根节点，那么前序遍历的第一个节点在中序位置之前的都是根节点的左子节点，之后的都是根节点的右子节点，我们来简单画个图看一下





这里是随便举个例子，我们看到前序遍历的3肯定是根节点，那么在中序遍历中，3前面的都是3左子节点的值，3后面的都是3右子节点的值，
他真正的结构是这样的



我们来看下代码

```
1   private int in = 0;
2   private int pre = 0;
3
4   public TreeNode buildTree(int[] preorder, int[] inorder) {
5     return build(preorder, inorder, Integer.MIN_VALUE);
6   }
7
8   private TreeNode build(int[] preorder, int[] inorder, int stop) {
9     if (pre >= preorder.length)
10      return null;
11    if (inorder[in] == stop) {
12      in++;
13      return null;
14    }
15
16    TreeNode node = new TreeNode(preorder[pre++]);
```

```
17      node.left = build(preorder, inorder, node.val);
18      node.right = build(preorder, inorder, stop);
19      return node;
20  }
```

总结

关于二叉树的算法题其实有很多，这里讲的也只是冰山一角，搞懂了上面和前面的几个关于二叉树的题，对二叉树算法相关题的理解也会进一步加深

往期推荐

- 410，剑指 Offer-从尾到头打印链表
- 408，剑指 Offer-替换空格
- 406，剑指 Offer-二维数组中的查找
- 404，剑指 Offer-数组中重复的数字

403，验证二叉搜索树

原创 山大王wld 数据结构和算法 7月18日

收录于话题
#算法图文分析

95个 >



Sometimes I feel I'm fighting for a life I just ain't got the time to live. I want it all to mean something.

我常常觉得我在为一个没时间享受的人生奋斗，我希望它能有价值。

问题描述

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

- 节点的左子树只包含小于当前节点的数。
- 节点的右子树只包含大于当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例 1:

```
输入:
    2
   / \
  1   3
```

> 输出: true

示例 2:

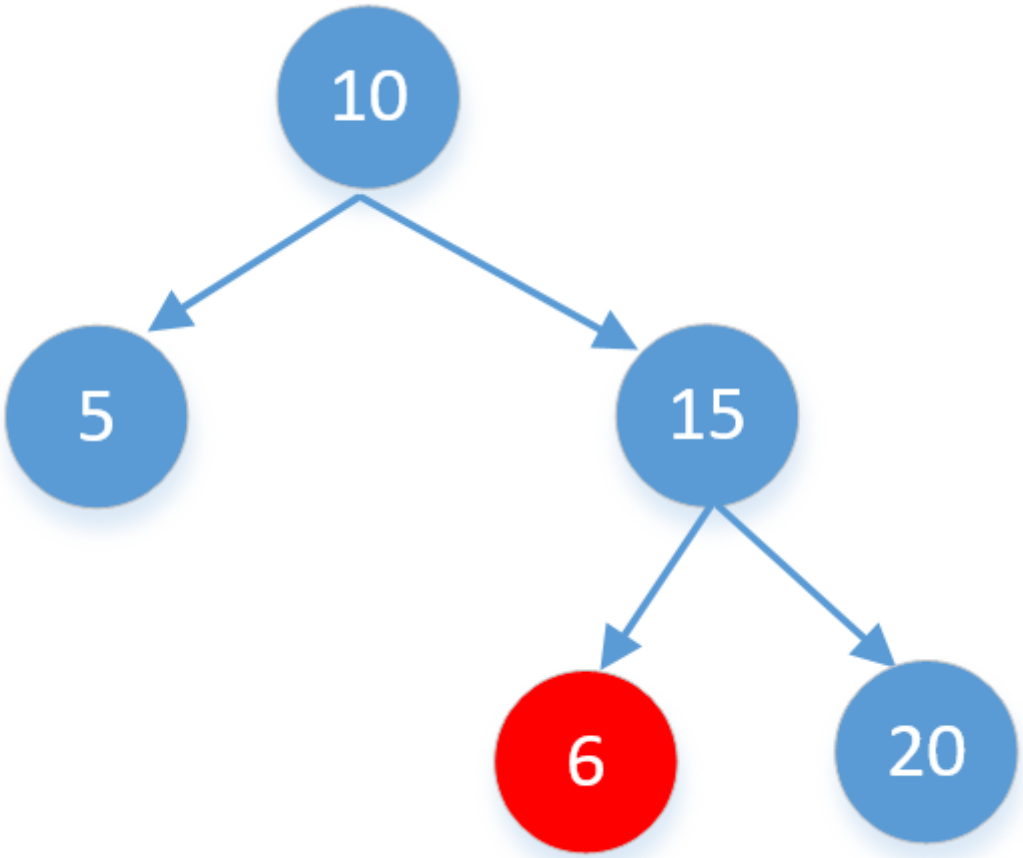
> 输入:
>
> 5
>
> / \
>
> 1 4
>
> / \
>
> 3 6
>
> 输出: false
>
> 解释: 输入为: [5,1,4,null,null,3,6]。
>
> 根节点的值为 5 ，但是其右子节点值为 4 。

递归写法

做这题之前我们首先要明白什么是二叉搜索树，就是每个节点左子树的值都比当前节点小，右子树的值都比当前节点大。所以看到这里我们最先想到的就是递归，我最先想到的是下面这种写法（注意是错误的）

```
1  public boolean isValidBST(TreeNode root) {
2    if (root == null)
3      return true;
4    if (root.left != null && root.val <= root.left.val || root.right != null && root.val >= root.right.
5      return false;
6    return isValidBST(root.left) && isValidBST(root.right);
7  }
```

如果一个结点是空的，我们默认他是有效的二叉搜索树，否则如果左节点不为空，我们要判断是否大于左节点的值，如果右节点不为空，我们还要判断小于右节点的值，然后我们再以左右两个子节点用相同的方式判断。看起来好像没什么问题，但我们好像忽略了一个每个节点的上限和下限，比如下面这棵树



注意6这个节点不光要小于15而且还要大于10，所以这里的每一个节点都是有一个范围的，上面的代码我只判断了6比15小，但没有和10进行比较，所以代码是错误的。这里我们来给每个节点添加一个范围，如果不在这个范围之内直接返回false，比如6的范围是(10,15)，很明显他不在这个范围内，所以他不是二叉搜索树。根节点的范围我们从Long.MIN_VALUE到Long.MAX_VALUE，来看下代码

```
1   public boolean isValidBST(TreeNode root) {
2     return isValidBST(root, Long.MIN_VALUE, Long.MAX_VALUE);
3   }
4
5   public boolean isValidBST(TreeNode root, long minVal, long maxVal) {
6     if (root == null)
7       return true;
8     //每个节点如果超过这个范围，直接返回false
9     if (root.val >= maxVal || root.val <= minVal)
10      return false;
11    //这里再分别以左右两个子节点分别判断，
12    //左子树范围的最小值是minVal，最大值是当前节点的值，也就是root的值，因为左子树的值要比当前节点小
13    //右子数范围的最大值是maxVal，最小值是当前节点的值，也就是root的值，因为右子树的值要比当前节点大
14    return isValidBST(root.left, minVal, root.val) && isValidBST(root.right, root.val, maxVal);
15  }
```

中序遍历递归

根据二叉搜索树的性质我们知道，中序遍历二叉搜索树，遍历的结果一定是有序的，如果不明白中序遍历的可以看下前面的373，数据结构-6,树。中序遍历时，判断当前节点是否大于中序遍历的前一个节点，也就是判断是否有序，如果不大于直接返回 false。

```
1   //前一个结点，全局的
```

```
    TreeNode prev;

    public boolean isValidBST(TreeNode root) {
        if (root == null)
            return true;
        //访问左子树
        if (!isValidBST(root.left))
            return false;
        //访问当前节点：如果当前节点小于等于中序遍历的前一个节点直接返回false。
        if (prev != null && prev.val >= root.val)
            return false;
        prev = root;
        //访问右子树
        if (!isValidBST(root.right))
            return false;
        return true;
    }
```

中序遍历非递归

如果对树的中序遍历比较熟悉的话，或者看过之前写的《373，数据结构-6,树》，这里面也有树的中序遍历的递归和非递归两种写法。我们完全可以把上面中序遍历的递归改为非递归。

```
public boolean isValidBST(TreeNode root) {
    if (root == null)
        return true;
    Stack<TreeNode> stack = new Stack<>();
    TreeNode pre = null;
    while (root != null || !stack.isEmpty()) {
        while (root != null) {
            stack.push(root);
            root = root.left;
        }
        root = stack.pop();
        if (pre != null && root.val <= pre.val)
            return false;
        //保存前一个访问的结点
        pre = root;
        root = root.right;
    }
    return true;
}
```

总结

这题可能最容易理解的是第一种解法，我们只需要给每个节点添加一个范围，然后再分别遍历每个节点，查看是否都在指定的范围内，只要有一个不在范围内，说明不是二叉搜索树，直接返回false。后面两种写法是根据二叉搜索树中序遍历的特点来判断，因为二叉搜索树中序遍历的结果是升序的，我们就按照二叉树中序遍历的方式来遍历这棵二叉树，然后在遍历的时候顺便保存一下前一个访问的结点，判断当前节点是否大于前一个结点的值，如果不大于直接返回false。

401，删除二叉搜索树中的节点

原创 山大王wld 数据结构和算法 7月15日

收录于话题
#算法图文分析

95个 >



I wanted to live deep and suck out all the marrow of life.

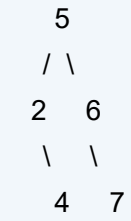
我希望活得深刻，吸取生命中所有精华。

问题描述

给定一个二叉搜索树的根节点 root 和一个值 key，删除二叉搜索树中的 key 对应的节点，并保证二叉搜索树的性质不变。返回二叉搜索树（有可能被更新）的根节点的引用。

示例:

```
root = [5,3,6,2,4,null,7]
key = 3

    5
   / \
  3   6
 / \   \
2   4   7
```

> 给定需要删除的节点值是 3，所以我们首先找到 3 这个节点，然后删除它。
>
> 一个正确的答案是 [5,4,6,2,null,null,7]，如下图所示。
>
> 
>
> 另一个正确答案是 [5,2,6,null,4,null,7]。
>
> 

问题分析

二叉搜索树的特点是左子树的值都比他小，右子树的值都比他大，删除一个节点之后我们还要保证二叉搜索树的这个特点不变。如果要删除一个结点，我们先要找到这个节点，然后才能删除，但这里要分几种情况。

- 如果要删除的节点是叶子节点，我们直接删除即可。
- 如果删除的结点不是叶子节点，并且有一个子节点为空，我们直接返回另一个不为空的子节点即可。
- 如果删除的结点不是叶子节点，并且左右子树都不为空，我们可以用左子树的最大值替换掉要删除的节点或者用右子树的最小值替换掉要删除的节点都是可以的。

这里使用递归的方式是最容易理解的，我们来看下代码

```
public TreeNode deleteNode(TreeNode root, int key) {
  if (root == null)
    return null;
  //通过递归的方式要先找到要删除的结点
  if (key < root.val) {
    //要删除的节点在左子树上
    root.left = deleteNode(root.left, key);
  } else if (key > root.val) {
    //要删除的节点在右子树上
    root.right = deleteNode(root.right, key);
  } else {
    //找到了要删除的节点。
    //如果左子树为空，我们只需要返回右子树即可
    if (root.left == null)
      return root.right;
    //如果右子树为空，我们只需要返回左子树即可
    if (root.right == null)
```

```
        return root.left;
      //说明两个子节点都不为空，我们可以找左子树的最大值，
      //也可以找右子树的最小值替换

      //这里是用右子树的最小值替换
      //TreeNode minNode = findMin(root.right);
      //root.val = minNode.val;
      //root.right = deleteNode(root.right, root.val);

      //这里是用左子树的最大值替换
      TreeNode maxNode = findMax(root.left);
      root.val = maxNode.val;
      root.left = deleteNode(root.left, root.val);
    }
    return root;
  }

  //  找右子树的最小值
  //  private TreeNode findMin(TreeNode node) {
  //    while (node.left != null)
  //      node = node.left;
  //    return node;
  //  }

  //找左子树的最大值
  private TreeNode findMax(TreeNode node) {
    while (node.right != null)
      node = node.right;
    return node;
  }
```

上面节点删除的时候我们使用左子树的最大值或者右子树的最小值替换都是可以的。其实我们还可以改一下，如果要删除结点左右子树只要有一个为空，我们就返回另一棵子树，如果都不为空，我们可以让左子树成为右子树最小结点的左子树或者让右子树成为左子树最大结点的右子树，我们来看下代码。

```
public TreeNode deleteNode(TreeNode root, int key) {
  if (root == null)
    return null;
  if (root.val > key) {
    //要删除的节点在左子树上
    root.left = deleteNode(root.left, key);
  } else if (root.val < key) {
    //要删除的节点在右子树上
    root.right = deleteNode(root.right, key);
  } else {
    //找到要删除的结点之后
    if (root.left == null)
      return root.right;
    if (root.right == null)
      return root.left;

    /*
    //左右子树都不为空，找到要删除结点右子树的最小值
    TreeNode rightSmallest = root.right;
    while (rightSmallest.left != null)
      rightSmallest = rightSmallest.left;
    //这个最小值对应的节点一定是没有左子树的，
    // 如果有他肯定不是最小的，然后让删除结点的
    //左子树成为这个最小值的左子树
    rightSmallest.left = root.left;
    //直接返回要删除结点的右子树
    return root.right;
     */

    //左右子树都不为空，找到要删除结点左子树的最大值
    TreeNode leftBig = root.left;
```

```
32          while (leftBig.right != null)
33              leftBig = leftBig.right;
34          //这个最大值对应的节点一定是没有右子树的，
35          // 如果有他肯定不是最大的，然后让删除结点的
36          //右子树成为这个最大值的右子树
37          leftBig.right = root.right;
38          //直接返回要删除结点的左子树
39          return root.left;
40      }
41      return root;
42  }
```

总结

删除结点的时候并不一定要直接删除，在之前讲[378，数据结构-7,堆]的时候删除结点直接使用其他节点来替换掉要删除的结点即可。这里也是使用同样的方式，对于二叉搜索树节点的删除，我们可以用它左子树的最大值或者右子树的最小值来替换，如果没有左子树或者右子树那就更方便了。

往期推荐

- 399，从前序与中序遍历序列构造二叉树
- 388，先序遍历构造二叉树
- 372，二叉树的最近公共祖先
- 367，二叉树的最大深度

400，二叉树的锯齿形层次遍历

原创 山大王wld 数据结构和算法 7月14日

收录于话题
#算法图文分析

95个 >



I may not be able to change the past, but I can learn from it.

我也许不能改变过去发生的事情，但能向过去学习。

问题描述

今天来看一道比较简单的题，给定一个二叉树，返回其节点值的锯齿形层次遍历。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

例如：
给定二叉树 [3,9,20,null,null,15,7]

```
  3
 / \
9  20
  /  \
 15   7
```

返回锯齿形层次遍历如下：

```
[
  [3],
  [20,9],
  [15,7]
]
```

BFS打印

二叉树的的层次遍历就是一层一层的遍历，也就是我们俗称的BFS（宽度优先搜索算法（又称广度优先搜索）），之前在373，数据结构-6,树中讲过树的宽度优先搜索，最简单的方式就是使用队列。但这题打印的时候多了一个条件，就是不能一直从一个方向打印，要先从左边打印然后再从右边打印……，就这样交替进行，所以这里要有个变量来判断是从左往右还是从右往左打印，代码比较简单，我们来看下。

```
1  public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
2    List<List<Integer>> res = new ArrayList<>();
3    if (root == null)
4      return res;
5    Queue<TreeNode> queue = new LinkedList<>();
6    queue.add(root);
7    boolean leftToRight = true;
8    while (!queue.isEmpty()) {
9      List<Integer> level = new ArrayList<>();
10     //统计这一行有多少个节点
11     int count = queue.size();
12     //遍历这一行的所有节点
13     for (int i = 0; i < count; i++) {
14       //poll移除队列头部元素（队列在头部移除，尾部添加）
15       TreeNode node = queue.poll();
16       //判断是从左往右打印还是从右往左打印。
17       if (leftToRight) {
18         level.add(node.val);
19       } else {
20         level.add(0, node.val);
21       }
22       //左右子节点如果不为空会被加入到队列中
23       if (node.left != null)
24         queue.add(node.left);
25       if (node.right != null)
26         queue.add(node.right);
27     }
28     res.add(level);
29     leftToRight = !leftToRight;
30   }
31   return res;
32 }
```

上面代码中如果把第17-21行的代码直接换成第18行的代码就是我们之前讲过的BFS，就是一层一层往下打印。只不过这里多了一个条件的判断。

当然我们还可以根据每一层是第几层来判断，如果根节点是第1层的话，那么我们在层数是奇数的时候从左往右打印，如果层数是偶数的时候从右往左打印。在前面我们讲队列的时候359，数据结构-3,队列我们讲到了双端队列，就是一个可以在两边同时添加和删除的队列。这里我们使用上面两种方式的结合，来看下代码。

```java
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    List<List<Integer>> res = new ArrayList<>();
    if (root == null)
        return res;
    //双端队列，两边都可以操作
    Deque<TreeNode> deque = new LinkedList<>();
    //添加到队列的头
    deque.addFirst(root);
    while (!deque.isEmpty()) {
        List<Integer> level = new ArrayList<>();
        //统计这一行有多少个节点
        int count = deque.size();
        //遍历这一行的所有节点
        TreeNode cur;
        for (int i = 0; i < count; i++) {
            if (res.size() % 2 == 1) {
                //从左边往右边打印
                //移除队列头部的元素，如果子节点不为空加入到队列的尾部
                cur = deque.removeFirst();
                if (cur.right != null)
                    deque.addLast(cur.right);
                if (cur.left != null)
                    deque.addLast(cur.left);
            } else {
                //从右边往左边打印
                //移除队列尾部的元素，如果子节点不为空加入到队列的头部
                cur = deque.removeLast();
                if (cur.left != null)
                    deque.addFirst(cur.left);
                if (cur.right != null)
                    deque.addFirst(cur.right);
            }
            level.add(cur.val);
        }
        res.add(level);
    }
    return res;
}
```

DFS打印

这题除了使用BFS以外，我们还可以使用DFS。但这里我们要有个判断，如果走到下一层的时候集合没有创建，我们要先创建下一层的集合，代码也很简单，我们来看下。

```java
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    List<List<Integer>> res = new ArrayList<>();
    travel(root, res, 0);
    return res;
}

private void travel(TreeNode cur, List<List<Integer>> res, int level) {
    if (cur == null)
        return;
    //如果res.size() <= level说明下一层的集合还没创建，所以要先创建下一层的集合
    if (res.size() <= level) {
        List<Integer> newLevel = new LinkedList<>();
        res.add(newLevel);
    }
    //遍历到第几层我们就操作第几层的数据
    List<Integer> list = res.get(level);
    //这里默认根节点是第0层，偶数层相当于从左往右遍历，
    // 所以要添加到集合的末尾，如果是奇数层相当于从右往左遍历，
    // 要把数据添加到集合的开头
    if (level % 2 == 0)
        list.add(cur.val);
    else
```

```
23          list.add(0, cur.val);
24      //分别遍历左右两个子节点，到下一层了，所以层数要加1
25      travel(cur.left, res, level + 1);
26      travel(cur.right, res, level + 1);
27  }
```

总结

这题最简单的一种方式就是参照二叉树的BFS打印，然后稍作修改，如果当前行是从左往右打印，那么下一行就从右往左打印。如果当前行是从右往左打印，那么下一行就从左往右打印，代码基本上没什么难度。

往期推荐

- 399，从前序与中序遍历序列构造二叉树
- 388，先序遍历构造二叉树
- 373，数据结构-6,树
- 372，二叉树的最近公共祖先

399，从前序与中序遍历序列构造二叉树

原创 山大王wld 数据结构和算法 7月13日

收录于话题
#算法图文分析

96个 >



Work and acquire, and thou hast chained the wheel of chance.

边工作边探求，你便可拴住机会的车轮。

问题描述

今天我们就不做关于双指针的了，我们爬到树上玩会儿，做一道关于二叉树的题。今天的题就一句话，根据一棵树的前序遍历与中序遍历构造二叉树。

注意:
你可以假设树中没有重复的元素。

例如，给出

```
前序遍历 preorder = [3,9,20,15,7]
中序遍历 inorder = [9,3,15,20,7]
```

返回如下的二叉树：

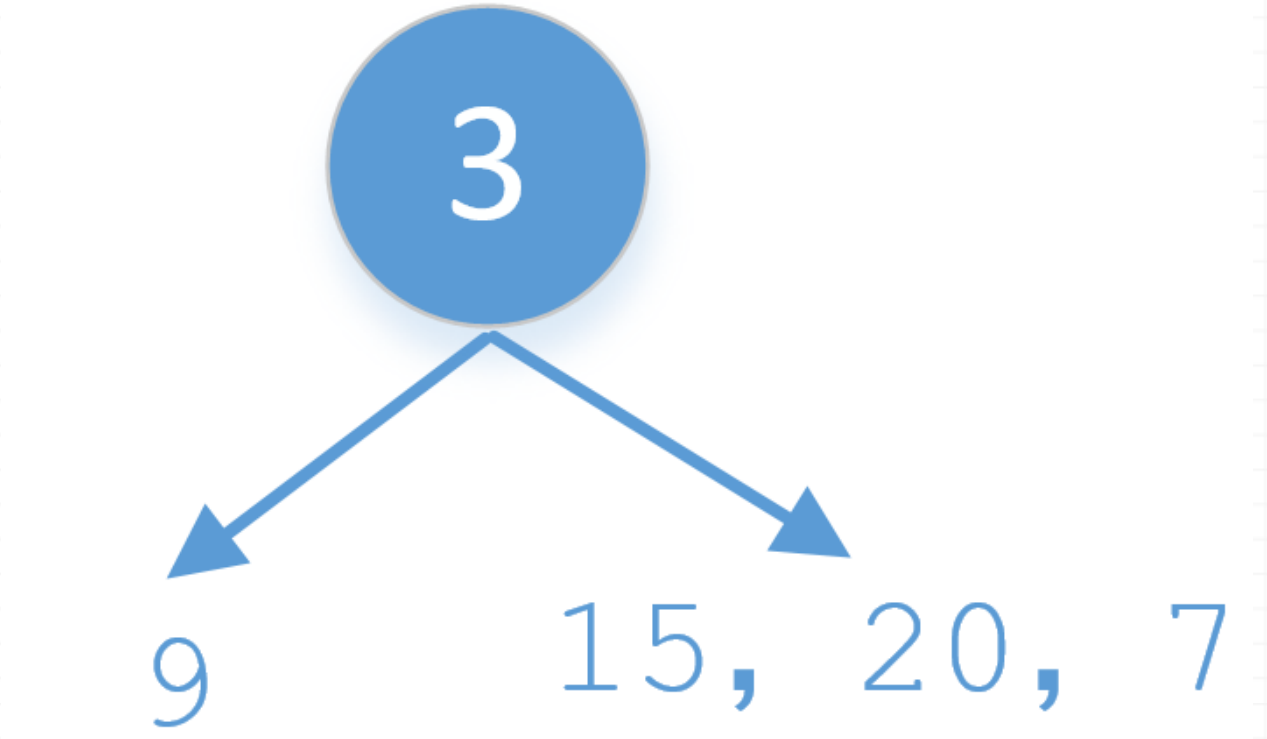
```
  3
 / \
9  20
  /  \
 15   7
```

问题分析

做这题之前我们先来看一下树的几种遍历顺序。
先序遍历：根节点→左子树→右子树。
中序遍历：左子树→根节点→右子树。
后续遍历：左子树→右子树→根节点。

其实也很好记，他是根据根节点遍历的顺序来定义的，比如先遍历根节点就是先序遍历，中间遍历根节点就是中序遍历，最后遍历根节点就是后续遍历，至于左子树和右子树哪个先遍历，记住一点，这3种遍历顺序右节点永远都不可能比左节点先遍历。如果还不懂的可以看下之前写的373，数据结构-6,树。

我们就以上面的示例数据来看下，前序遍历是[3,9,20,15,7]，前序遍历先访问的是根节点，所以3就是根节点。中序遍历是[9,3,15,20,7]，由于中序遍历是在左子树都遍历完的时候才遍历根节点，所有在中序遍历中3前面的都是3的左子树节点，3后面的都是3的右子树节点。也就是下面这样



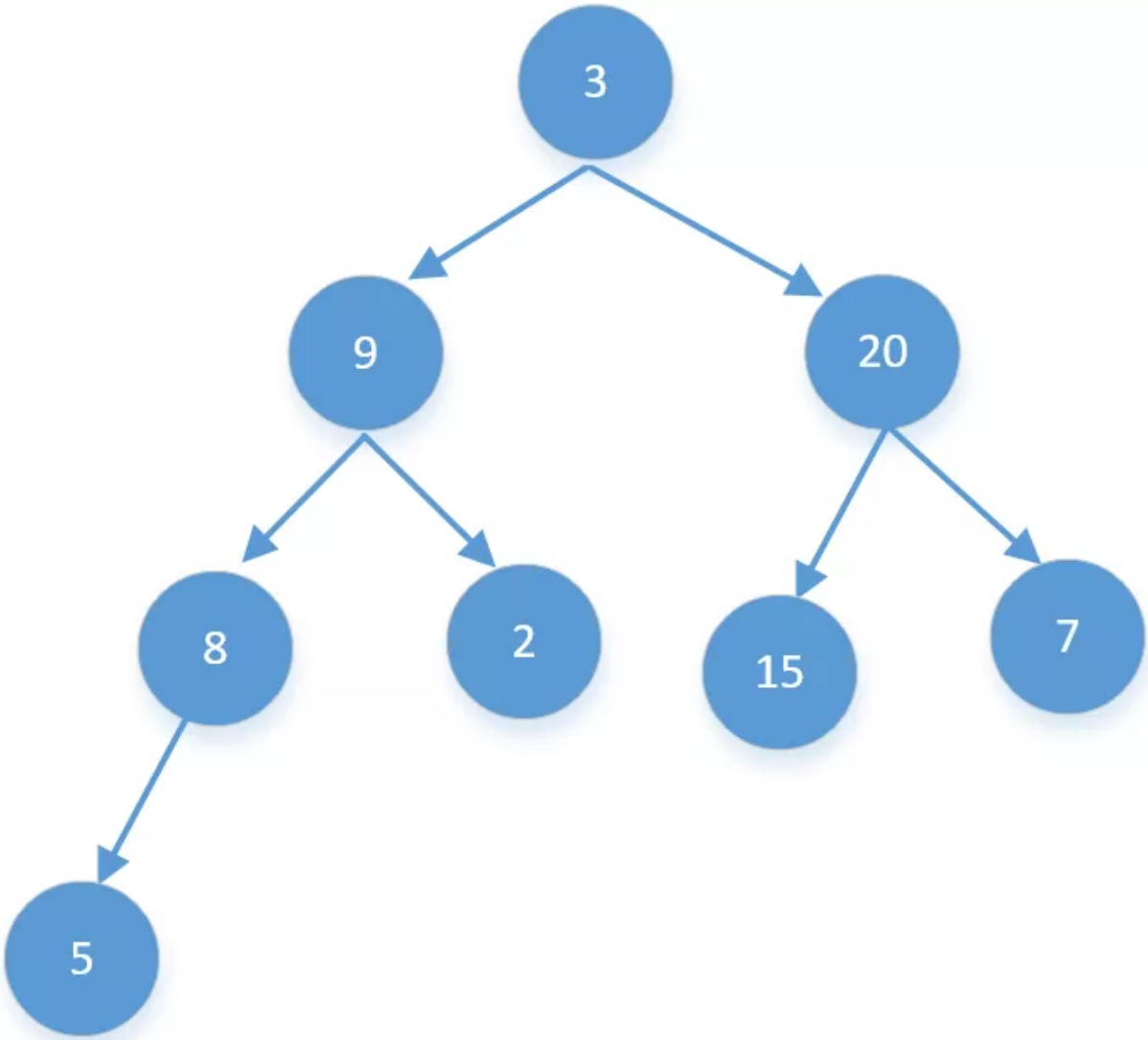
然后我们再使用同样的方式对左右子树继续划分，一直这样下去，直到不能再分为止，我们来看下代码

```java
public TreeNode buildTree(int[] preorder, int[] inorder) {
    //把前序遍历的值和中序遍历的值放到list中
    List<Integer> preorderList = new ArrayList<>();
    List<Integer> inorderList = new ArrayList<>();
    for (int i = 0; i < preorder.length; i++) {
        preorderList.add(preorder[i]);
        inorderList.add(inorder[i]);
    }
    return helper(preorderList, inorderList);
}

private TreeNode helper(List<Integer> preorderList, List<Integer> inorderList) {
    if (inorderList.size() == 0)
        return null;
    //前序遍历的第一个值就是根节点
    int rootVal = preorderList.remove(0);
    //创建跟结点
    TreeNode root = new TreeNode(rootVal);
    //查看根节点在中序遍历中的位置，然后再把中序遍历的数组劈两半，前面部分是
    //根节点左子树的所有值，后面部分是根节点右子树的所有值
    int mid = inorderList.indexOf(rootVal);
    //[0，mid)是左子树的所有值，inorderList.subList(0，mid)表示截取inorderList
    //的值，截取的范围是[0，mid)，包含0不包含mid。
    root.left = helper(preorderList, inorderList.subList(0, mid));
    //[mid+1，inorderList.size())是右子树的所有值，
    // inorderList.subList(mid + 1, inorderList.size())表示截取inorderList
    //的值，截取的范围是[mid+1，inorderList.size())，包含mid+1不包含inorderList.size()。
    root.right = helper(preorderList, inorderList.subList(mid + 1, inorderList.size()));
    return root;
}
```

上面代码中是先把数组转化为list集合，然后在list集合中进行截取，这样效率明显不是很高，实际上我们还可以不使用list，不对数组进行截取。

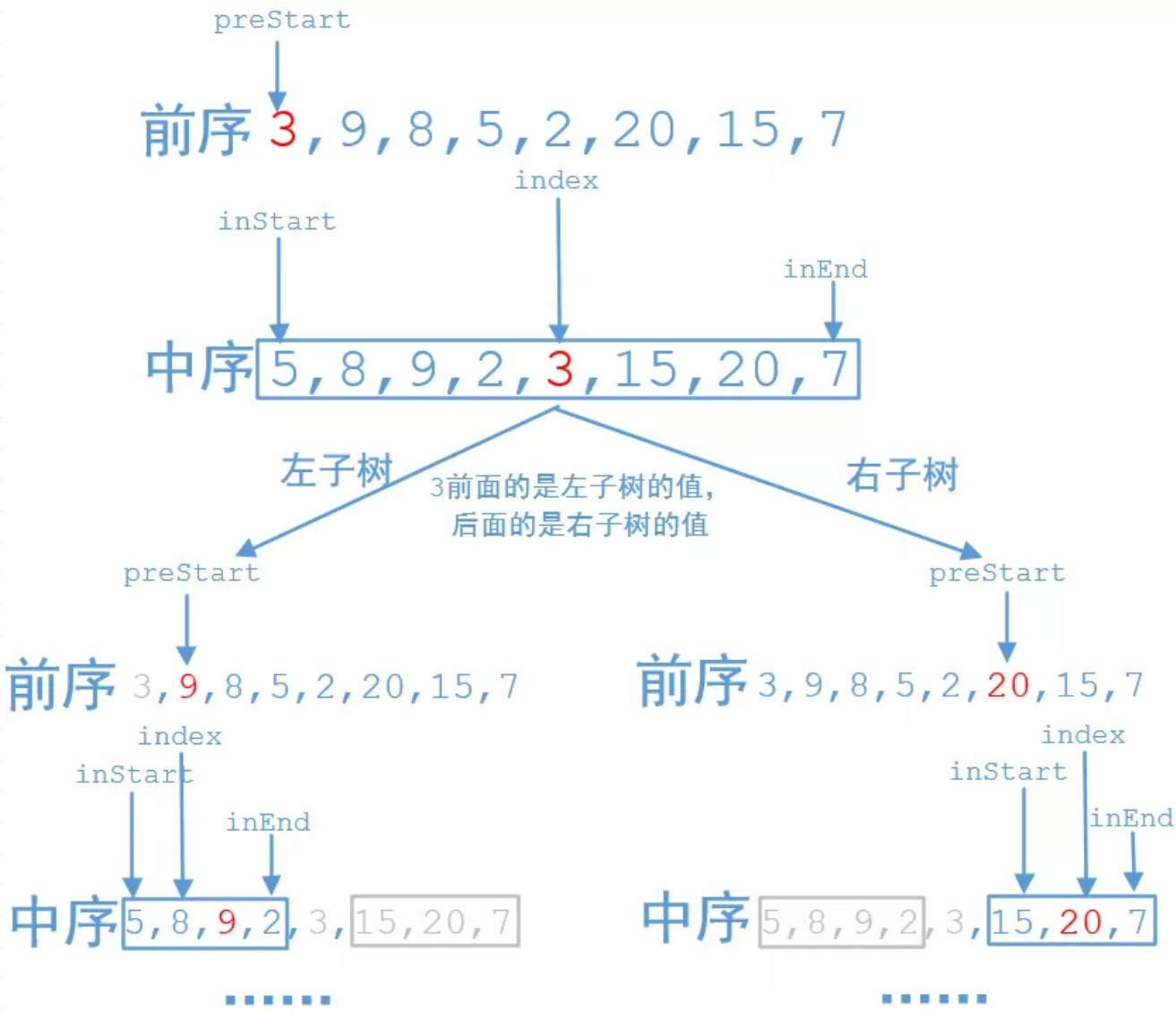
使用指针解决

我们只需要使用3个指针即可。一个是preStart，他表示的是前序遍历开始的位置，一个是inStart，他表示的是中序遍历开始的位置。一个是inEnd，他表示的是中序遍历结束的位置，我们主要是对中序遍历的数组进行拆解，下面就以下面的这棵树来画个图分析下



他的前序遍历是：[3,9,8,5,2,20,15,7]

他的中序遍历是：[5,8,9,2,3,15,20,7]



这里只要找到了前序遍历的结点在中序遍历的位置，我们就可以把中序遍历数组分解为两部分了。如果index是前序遍历的某个值在中序遍历数组中的索引，以index为根节点划分的话，那么中序遍历中

[0，index-1]就是根节点左子树的所有节点，

[index+1，inorder.length-1]就是根节点右子树的所有节点。

中序遍历好划分，那么前序遍历呢，如果是左子树：

preStart=index+1;

如果是右子树就稍微麻烦点，

preStart=preStart+(index-instart+1);

preStart是当前节点比如m先序遍历开始的位置，index-instart+1就是当前节点m左子树的数量加上当前节点的数量，所以preStart+(index-instart+1)就是当前节点m右子树前序遍历开始的位置，我们来看下完整代码

```
1   public TreeNode buildTree(int[] preorder, int[] inorder) {
2     return helper(0, 0, inorder.length - 1, preorder, inorder);
3   }
4
5   public TreeNode helper(int preStart, int inStart, int inEnd, int[] preorder, int[] inorder) {
6     if (preStart > preorder.length - 1 || inStart > inEnd) {
```

```
        return null;
    }
    //创建结点
    TreeNode root = new TreeNode(preorder[preStart]);
    int index = 0;
    //找到当前节点root在中序遍历中的位置，然后再把数组分两半
    for (int i = inStart; i <= inEnd; i++) {
        if (inorder[i] == root.val) {
            index = i;
            break;
        }
    }
    root.left = helper(preStart + 1, inStart, index - 1, preorder, inorder);
    root.right = helper(preStart + index - inStart + 1, index + 1, inEnd, preorder, inorder);
    return root;
}
```

使用栈解决

如果使用栈来解决首先要搞懂一个知识点，就是前序遍历挨着的两个值比如m和n，他们会有下面两种情况之一的关系。

1，n是m左子树节点的值。
2，n是m右子树节点的值或者是m某个祖先节点的右节点的值。

- 对于第一个知识点我们很容易理解，如果m的左子树不为空，那么n就是m左子树节点的值。

- 对于第二个问题，如果一个结点没有左子树只有右子树，那么n就是m右子树节点的值，如果一个结点既没有左子树也没有右子树，那么n就是m某个祖先节点的右节点，我们只要找到这个祖先节点就好办了。

搞懂了这点，代码就很容易写了，下面看下完整代码

```
public TreeNode buildTree(int[] preorder, int[] inorder) {
    if (preorder.length == 0)
        return null;
    Stack<TreeNode> s = new Stack<>();
    //前序的第一个其实就是根节点
    TreeNode root = new TreeNode(preorder[0]);
    TreeNode cur = root;
    for (int i = 1, j = 0; i < preorder.length; i++) {
        //第一种情况
        if (cur.val != inorder[j]) {
            cur.left = new TreeNode(preorder[i]);
            s.push(cur);
            cur = cur.left;
        } else {
            //第二种情况
            j++;
            //找到合适的cur，然后确定他的右节点
            while (!s.empty() && s.peek().val == inorder[j]) {
                cur = s.pop();
                j++;
            }
            //给cur添加右节点
            cur = cur.right = new TreeNode(preorder[i]);
        }
```

```
    }
    return root;
}
```

总结

这题如果直接在纸上推算出来还是很简单的，如果写成代码就稍微有一点难度。当然第一种写法还是非常简单，他是每次遍历都会把数组截取，但截取效率不高，所以第二种方式就使用指针的方式，每次遍历的时候通过指针来固定左子树和右子树在数组中的范围。第3种方式是巧妙的运用了前序遍历的特点，然后使用栈的方式解决，这种方式也是非常经典的，一般不太容易想到。

往期推荐

- 397，双指针求接雨水问题
- 396，双指针求盛最多水的容器
- 388，先序遍历构造二叉树
- 373，数据结构-6,树

388，先序遍历构造二叉树

原创 山大王wld 数据结构和算法 6月23日

收录于话题
#算法图文分析

96个 >



Love is the one thing that transcends time and space.

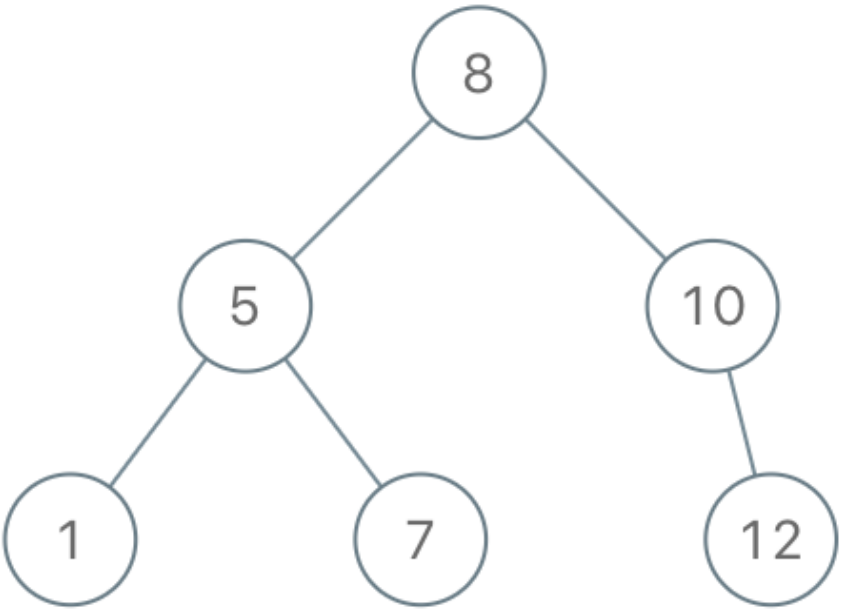
只有爱可以穿越时空。

问题描述

> 返回与给定先序遍历相匹配的二叉搜索树的根结点。

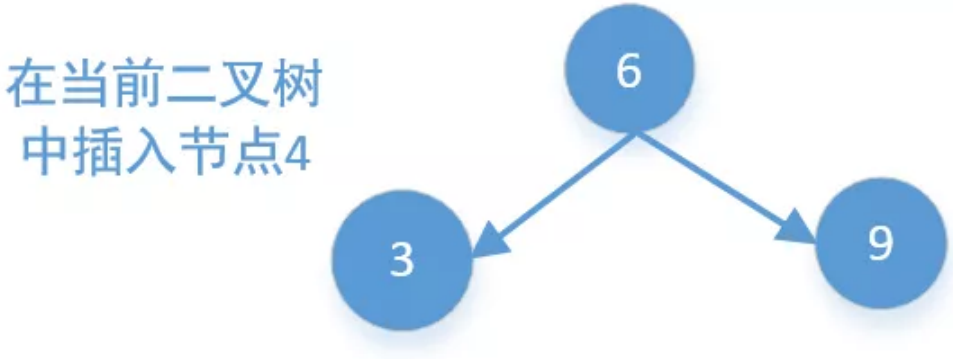
示例：

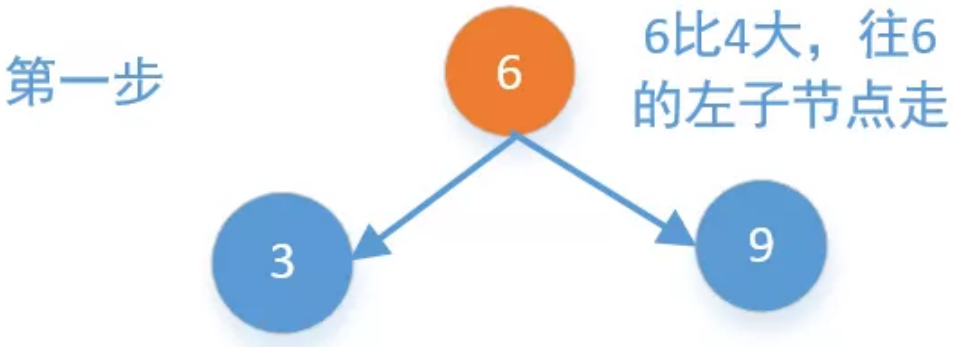
输入：[8，5，1，7，10，12]
输出：[8，5，10，1，7，null，12]

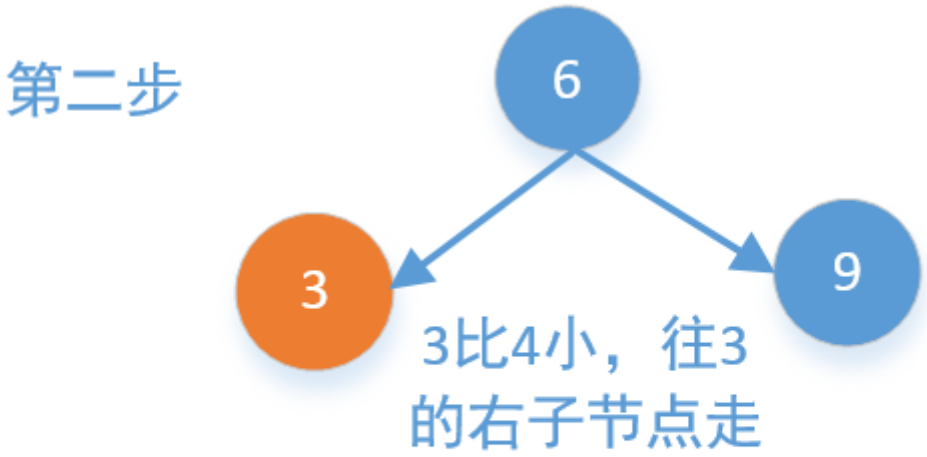


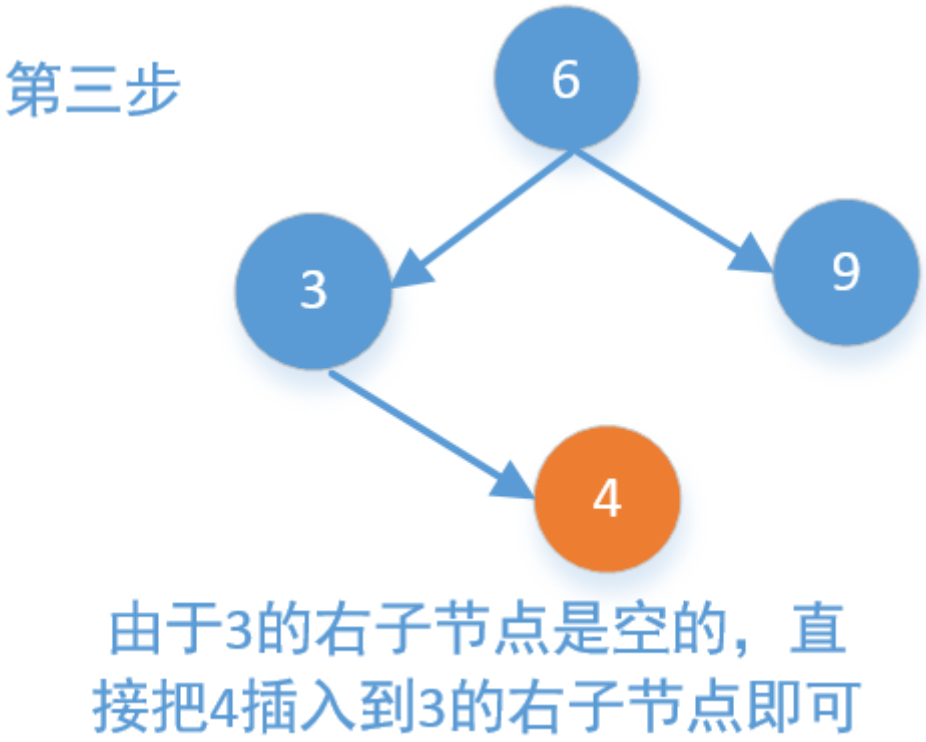
问题分析

我们知道先序遍历的顺序是：根节点→左子树→右子树。二叉搜索树的特点是当前节点左子树的值都小于当前节点，当前节点右子树的值都大于当前节点。比如我们在下面的搜索二叉树中插入节点4









原理很简单，我们来看下如果插入一个结点的时候代码该怎么写

```
//data是插入的结点
private static void addTreeNode(TreeNode root, int data) {
  TreeNode node = new TreeNode(data);
  TreeNode p = root;
  while (true) {
    //如果要插入的结点data比结点p的值小，就往p结点的左
    //子节点找，否则往p的右子节点找
    if (p.val > data) {
      //如果p的左子节点等于空，直接放进去
      if (p.left == null) {
        p.left = node;
        break;
      } else {
        p = p.left;
      }
    } else {
      //如果p的右子节点等于空，直接放进去
```

```
            if (p.right == null) {
                p.right = node;
                break;
            } else {
                p = p.right;
            }
        }
    }
}
```

上面代码很简单，插入一个结点的代码写出来了，我们只需要把数组中的元素全部遍历一遍然后再一个个插入即可，代码如下

```
public TreeNode bstFromPreorder(int[] preorder) {
    TreeNode root = new TreeNode();
    root.val = preorder[0];
    for (int i = 1; i < preorder.length; i++)
        addTreeNode(root, preorder[i]);
    return root;
}
```

递归方式

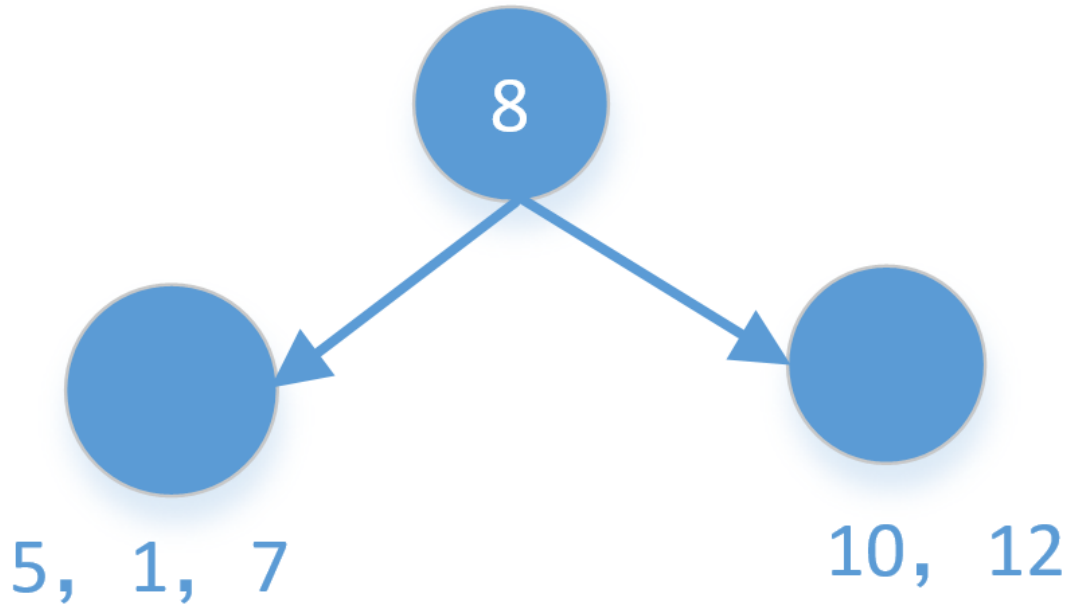
上面节点插入的时候我们使用的是while循环的方式，这种比较容易理解，但代码量相对比较多，我们还可以改为递归的方式

```
private TreeNode addTreeNode(TreeNode root, int val) {
    if (root == null)
        return new TreeNode(val);
    else if (root.val > val)
        root.left = addTreeNode(root.left, val);
    else
        root.right = addTreeNode(root.right, val);
    return root;
}
```

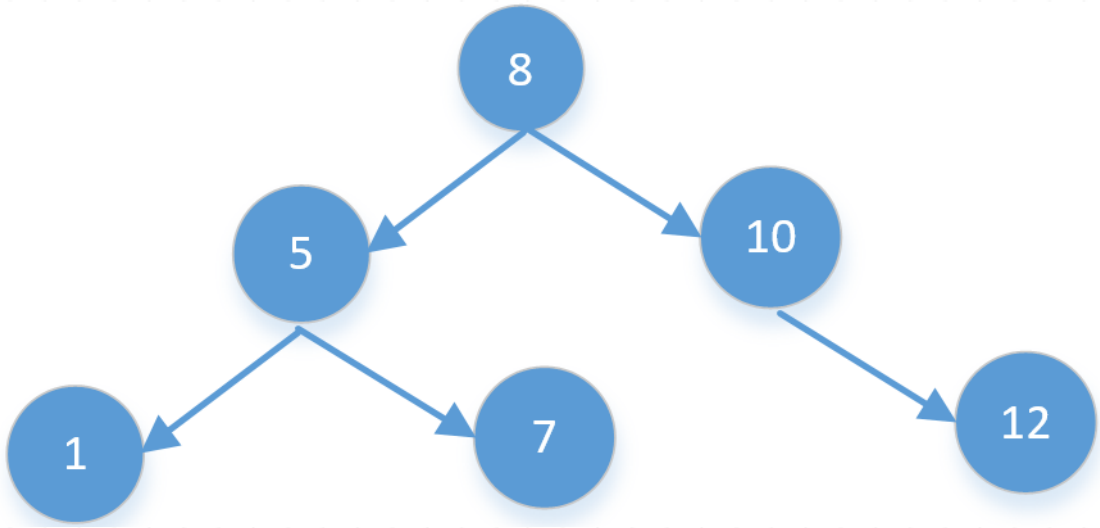
这种递归的方式代码会更简洁一些。如果root为空的话会新建一个节点。否则会一直走下去，他会根据root节点的大小判断往左走还是往右走，注意这里的root节点不一定是根节点，在递归的时候他是一直变的。

二分法构造

我们知道输入的数据是二叉树的先序遍历，那么第一个节点肯定是头结点，比他小的是他左子树的节点值，比他大的是他右子树的节点值，我们就拿上面的[8,5,1,7,10,12]来说，8是根节点，比8小的[5，1，7]是他左子树上的值，比他大的[10，12]是他右子树上的值。所以可以参照二分法查找的方式，把数组分为两部分，他是这样的



然后左边的[5，1，7]我们再按照上面的方式拆分，5是根节点，比5小的1是左子节点，比5大的7是右子节点。同理右边的[10，12]中10是根节点，比10大的12是右子节点，这样我们一直拆分下去，直到不能拆分为止，所以结果是下面这样



我们再来看下代码

```
public TreeNode bstFromPreorder(int[] preorder) {
    return buildBST(preorder, 0, preorder.length - 1);
}

//数组的范围从left到right
private TreeNode buildBST(int[] preorder, int left, int right) {
    if (left > right)
        return null;
    TreeNode root = new TreeNode(preorder[left]);
    //如果left==right说明只有一个元素，没法再拆分了
    if (left == right)
        return root;
    int i = left;
    //拆分为两部分，一部分是比preorder[left]大的，一部分是比preorder[left]小的
    while (i + 1 <= right && preorder[i + 1] < preorder[left])
        i++;
    //区间[left + 1，i]所有元素都在root节点的左子树
    //区间[i + 1，right]所有元素都在root节点的右子树
    root.left = buildBST(preorder, left + 1, i);
    root.right = buildBST(preorder, i + 1, right);
    return root;
}
```

先序遍历

我们还可以参照先序遍历的方式把数组元素一个个取出来，也很好理解，直接上代码。

```
int index = 0;

public TreeNode bstFromPreorder(int[] preorder) {
  return bstFromPreorder(preorder, Integer.MAX_VALUE);
}

public TreeNode bstFromPreorder(int[] preorder, int max) {
  if (index == preorder.length || preorder[index] > max)
    return null;
  //把数组中的元素一个个取出来创建节点
  TreeNode root = new TreeNode(preorder[index++]);
  //左子树的最大值不能超过root.val
  root.left = bstFromPreorder(preorder, root.val);
  //右子树的最大值不能超过max
  root.right = bstFromPreorder(preorder, max);
  return root;
}
```

使用栈来解决

这题解法比较多，再来看最后一种解题思路。我们还可以使用一个栈来维护二叉搜索树中的节点，栈中存放的是已经构建好的二叉搜索树的结点（但不是全部，有些可能已经出栈了），其中栈中元素从栈底到栈顶是递减的，我们遍历数组的时候如果当前值小于栈顶元素的值，我们直接让当前值成为栈顶元素节点的左子节点，然后压栈。

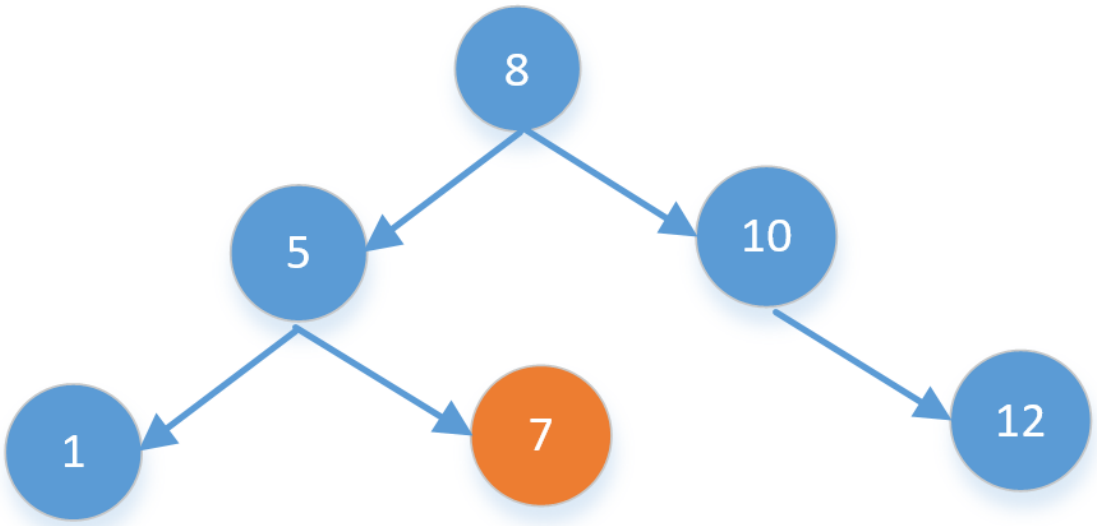
```
if (preorder[i] < stack.peek().val) {
    stack.peek().left = node;
}
```

如果当前元素的值大于栈顶元素的值，我们就让栈顶元素出栈，直到当前元素的值小于栈顶元素的值为止（或者栈为空为止）。而前一个比当前元素值小的节点就是当前元素的父节点。而当前元素是他父节点的右子节点。

```
TreeNode parent = stack.peek();
//栈从栈底到栈顶是递减的
while (!stack.isEmpty() && preorder[i] > stack.peek().val) {
    parent = stack.pop();
}
parent.right = node;
```

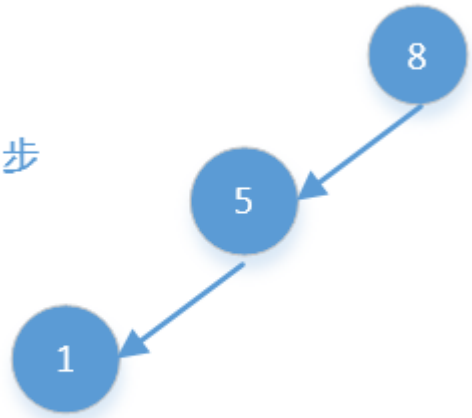
解惑：

这里如果思路不是很清晰的可能会有点疑问，出栈的时候把小于当前元素的值出栈了，如果再遇到比出栈的元素还要小的值那不是完蛋了，因为那个值已经出栈了，找不到了。其实有这个想法是正确的，但这种想法有点多余了，我们就拿下面的图来说吧[8,5,1,7,10,12]



比如当我们插入节点7的时候，节点1，5都已经全部出栈，但7后面无论如何都不会再出现比1或者5还小的值了，因为他是二叉搜索树，5的右节点的所有值都是比5大的。我们来画个简单的图看下

第一步

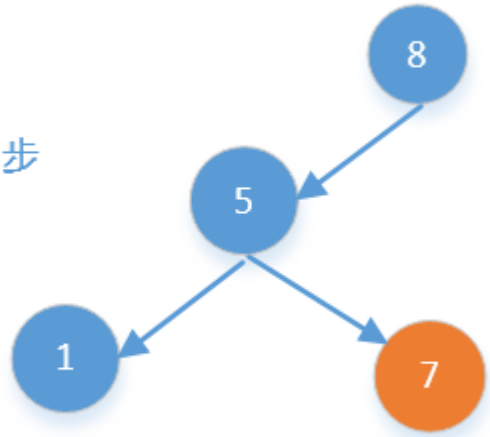


8，5，1分别入栈



因为8，5，1是递减的，所以5是8
的左子节点，1是5的左子节点

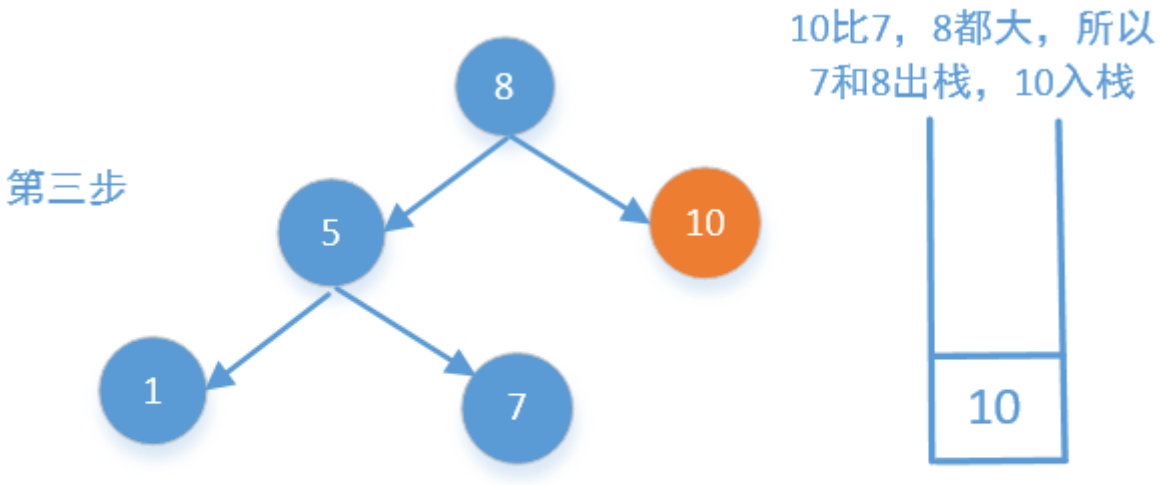
第二步



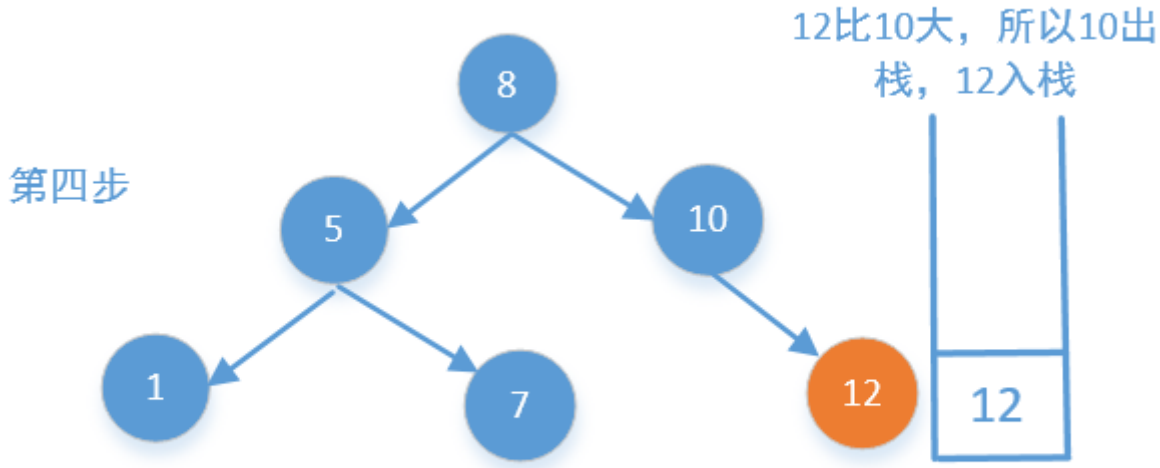
7比1，5都大，所以
1和5出栈，7入栈



7比栈中元素1，5都大，所以7是
最后一个出栈5的右子节点



10比栈中元素7，8都大，所以10
是最后一个出栈8的右子节点



12比栈中元素10大，所以12是最
后一个出栈10的右子节点

所以我们看到后面无论走到哪一步都不可能在遇到比出栈元素更小的值了，最后我们再来看下完整代码

```
public TreeNode bstFromPreorder(int[] preorder) {
   Stack<TreeNode> stack = new Stack<>();
   TreeNode root = new TreeNode(preorder[0]);
   stack.push(root);
   for (int i = 1; i < preorder.length; i++) {
      TreeNode node = new TreeNode(preorder[i]);
      //小于栈顶元素的值，说明应该在栈顶元素的左子树
      if (preorder[i] < stack.peek().val) {
         stack.peek().left = node;
      } else {//大于栈顶元素的值，我们要找到当前元素的父节点
         TreeNode parent = stack.peek();
         //栈从栈底到栈顶是递减的
         while (!stack.isEmpty() && preorder[i] > stack.peek().val) {
            parent = stack.pop();
         }
         parent.right = node;
      }
```

```
18          //节点压栈
19          stack.push(node);
20      }
21      return root;
22  }
```

往期推荐

- 374，二叉树的最小深度
- 373，数据结构-6,树
- 372，二叉树的最近公共祖先
- 367，二叉树的最大深度

387，二叉树中的最大路径和

原创 山大王wld 数据结构和算法 6月21日

收录于话题
#算法图文分析

96个 >



You never want your kids to see you scared. You wanna be that rock that they can grab ahold of in a stormy sea.

当父亲的永远都不想在孩子面前露怯，你想在暴风来临时让他们依靠，成为他们心中的坚石。

问题描述

给定一个非空二叉树，返回其最大路径和。

路径被定义为一条从树中任意节点出发，达到任意节点的序列。该路径至少包含一个节点，且不一定经过根节点。

示例 1:

```
输入: [1, 2, 3]

      1
     / \
    2   3

输出: 6
```

示例 2:

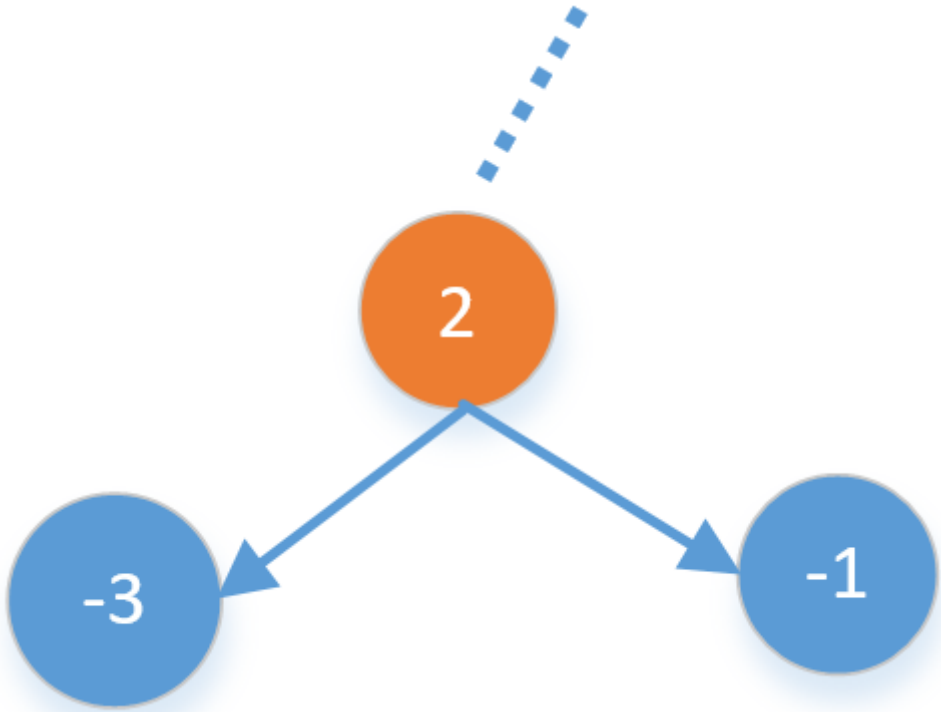
```
输入: [-10, 9, 20, null, null, 15, 7]

  -10
  / \
 9  20
    /  \
   15   7

输出: 42
```

问题分析

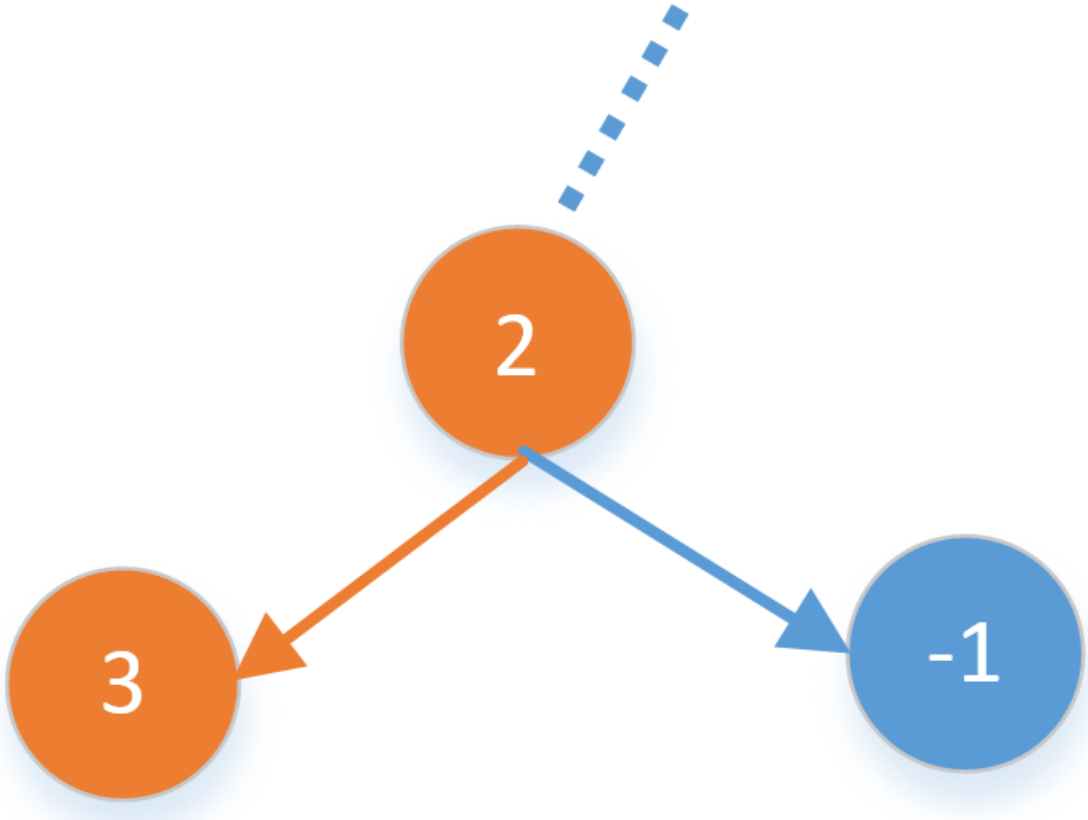
这道题要求的最大路径和如果是从根节点开始到叶子节点就好办了，我们可以通过递归的方式，从下往上，舍去比较小的路径节点，保留比较大的节点。

但这道题要求的最大路径和并不一定经过根节点，如果再使用上面的方式就行不通了，对于这道题我们可以分为4种情况来讨论

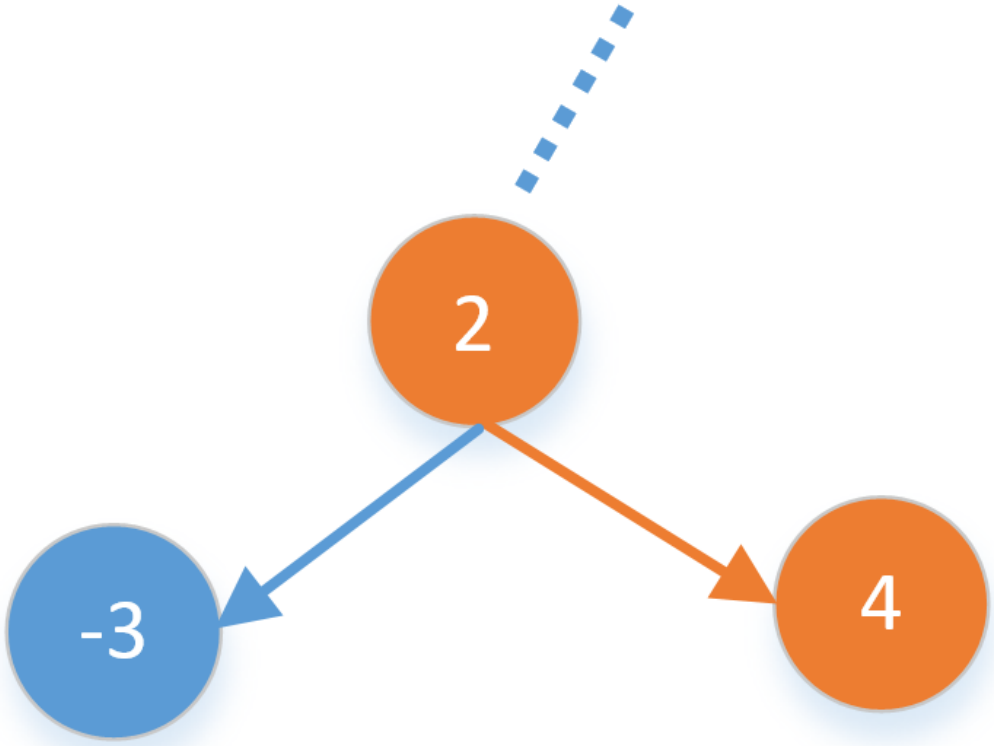
1，只要当前节点，舍弃子节点。比如下面结点2的左右子节点都是负数，如果是负数我们还不如不要，所以直接舍弃子节点。



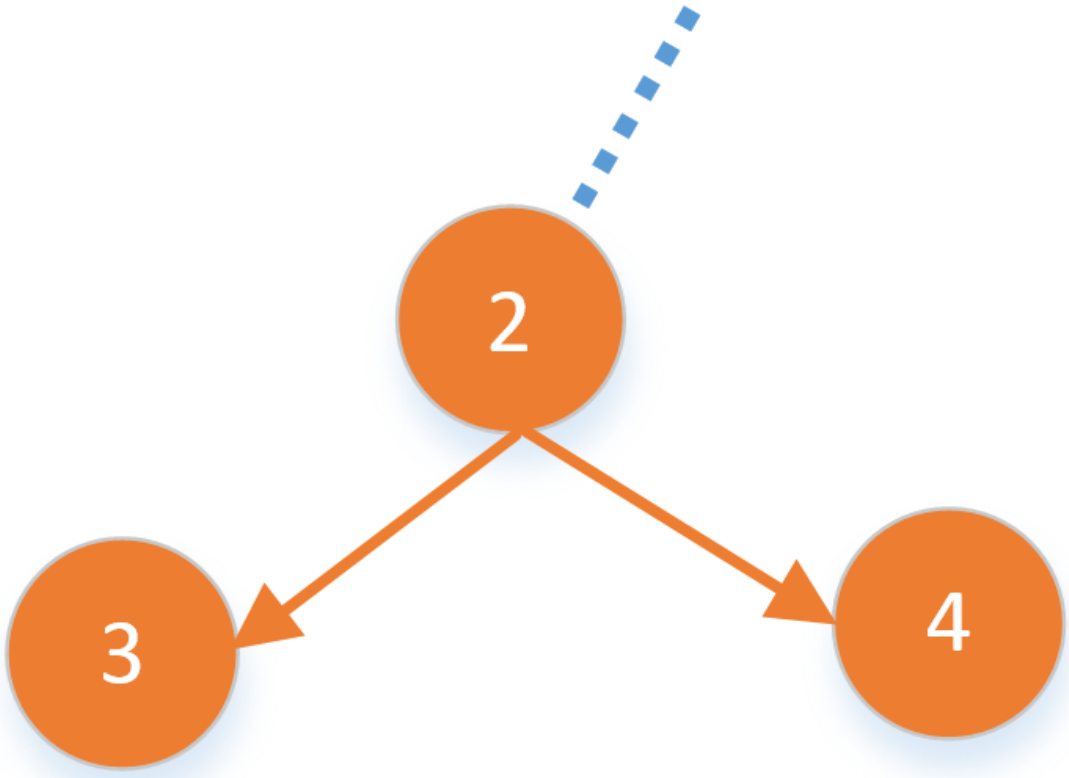
2，保留当前节点和左子节点。比如下面结点2的右子节点是负数，我们直接舍弃右子节点，但左子节点不是负数，我们可以保留左子节点。



3，保留当前节点和右子节点。比如下面结点2的左子节点是负数，我们直接舍弃左子节点，但右子节点不是负数，我们可以保留右子节点。



4，保留当前节点和两个子节点。比如下面结点2的左右子节点都不是负数，我们都可以留下。

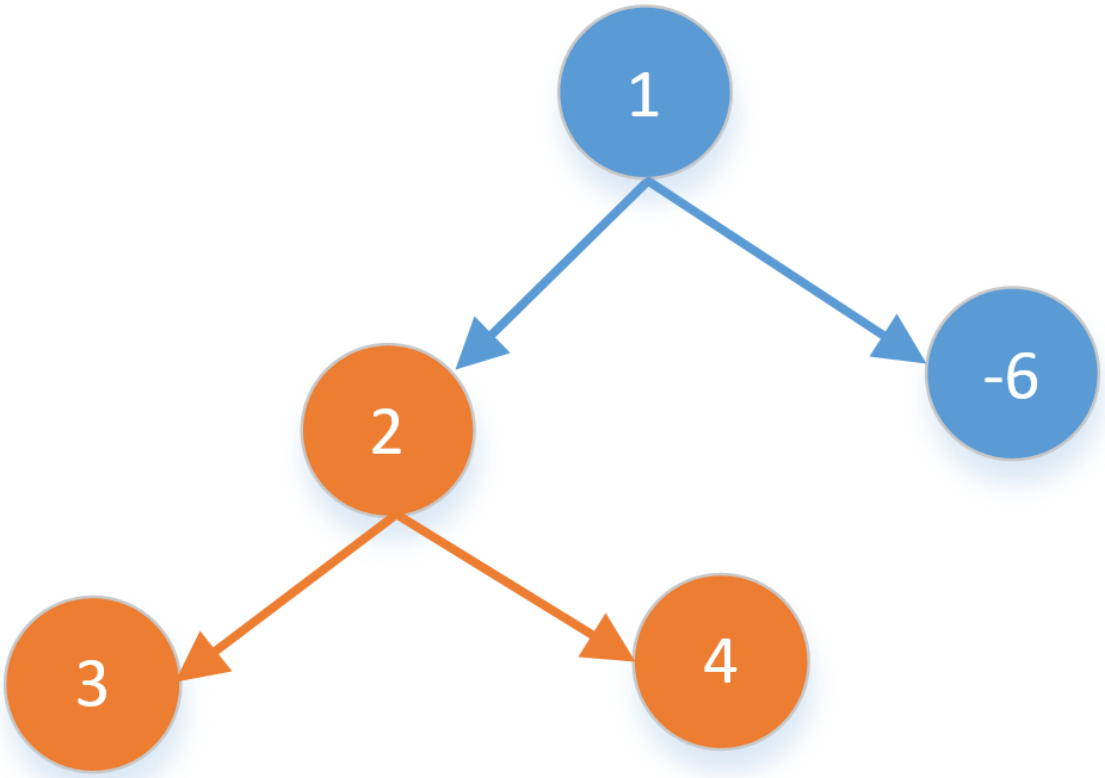


上面的1，2，3都可以作为子树的一部分再继续计算，我们可以使用同一个公式，取左右子节点最大的那个即可，如果都小于0我们不要了，下面公式中left是左子树的值，right是右子树的值

```
int res = root.val + Math.max(0, Math.max(left, right));
```

而4是不能在作为子树的一部分参与计算的，因为已经分叉了，比如下面的3→2→4是不能再和结点1进行组合的。第4种情况如果左右子树有一个是小于0的我们还不如不选，如果都大于0我们都要选的。

```
int cur = root.val + Math.max(0, left) + Math.max(0, right);
```



搞懂了上面的分析过程，代码就很容易写出来了，我们最后来看下代码

代码部分

```
1   private int maxValue = Integer.MIN_VALUE;
2
3   public int maxPathSum(TreeNode root) {
4     maxPathSumHelper(root);
5     return maxValue;
6   }
7
8   public int maxPathSumHelper(TreeNode root) {
9     if (root == null)
10      return 0;
11    //左子节点的值
12    int left = maxPathSumHelper(root.left);
13    //右子节点的值
14    int right = maxPathSumHelper(root.right);
15    //第4种情况
16    int cur = root.val + Math.max(0, left) + Math.max(0, right);
17    //第1,2,3三种情况,返回当前值加上左右子节点的最大值即可，如果左右子节点都
18    //小于0，还不如不选
19    int res = root.val + Math.max(0, Math.max(left, right));
20    //记录最大value值
21    maxValue = Math.max(maxValue, Math.max(cur, res));
22    //第1,2,3种情况还可以再计算，所以返回的是res
```

```
23      return res;
24  }
```

往期推荐

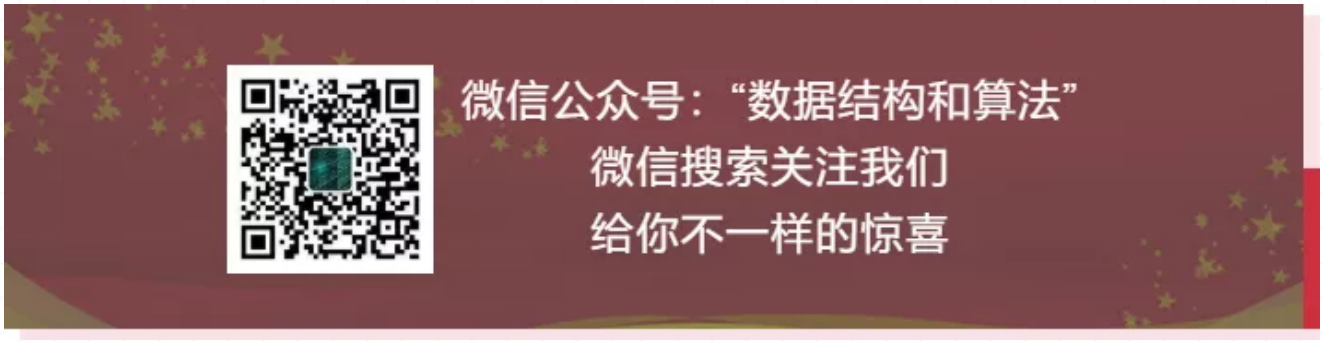
- 383，不使用“+”,“-”,“×”,“÷”实现四则运算
- 375，在每个树行中找最大值
- 373，数据结构-6,树
- 372，二叉树的最近公共祖先

375，在每个树行中找最大值

原创 山大王wld 数据结构和算法 6月3日

收录于话题
#算法图文分析

96个 >



Don't spend another minute being angry about yesterday.

不要再浪费时间为昨天而懊恼。

问题描述

在二叉树的每一行中找到最大的值。
比如
输入:

```
      1
     / \
    3   2
   / \   \
  5   3   9
```

输出: [1, 3, 9]

问题分析：

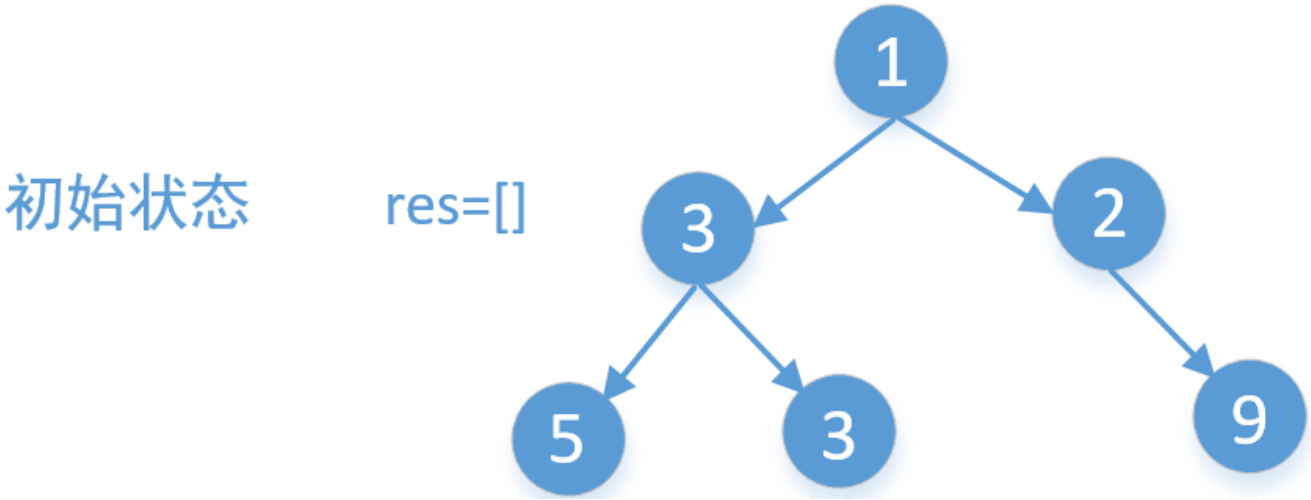
01 BFS求解

关于这道题我们最容易想到的也就是BFS，一层一层遍历，然后在每一层中再找出最大值。前面已经讲过很多BFS的题，这题不是很难。我们来直接看下代码。

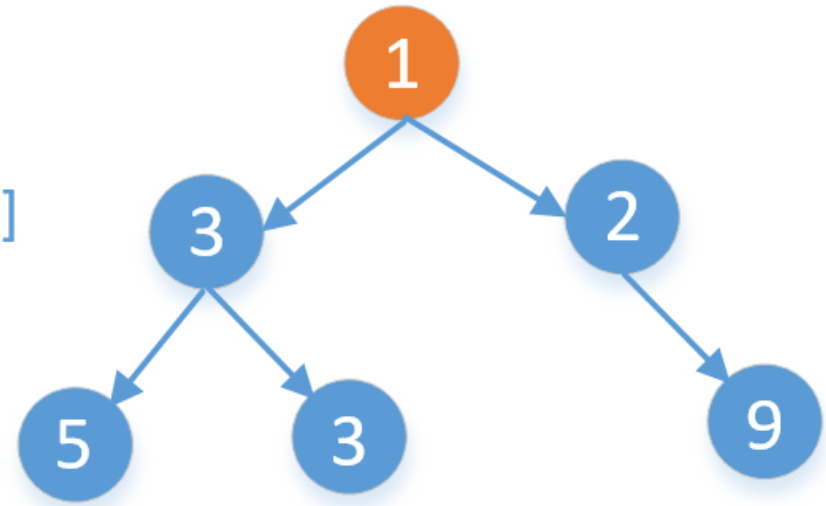
```java
public List<Integer> largestValues(TreeNode root) {
    //LinkedList实现队列
    Queue<TreeNode> queue = new LinkedList<>();
    List<Integer> values = new ArrayList<>();
    if (root != null)
        queue.add(root);//入队
    while (!queue.isEmpty()) {
        int max = Integer.MIN_VALUE;
        int levelSize = queue.size();//每一层的数量
        for (int i = 0; i < levelSize; i++) {
            TreeNode node = queue.poll();//出队
            max = Math.max(max, node.val);//记录每层的最大值
            if (node.left != null)
                queue.add(node.left);
            if (node.right != null)
                queue.add(node.right);
        }
        values.add(max);
    }
    return values;
}
```

02 DFS求解

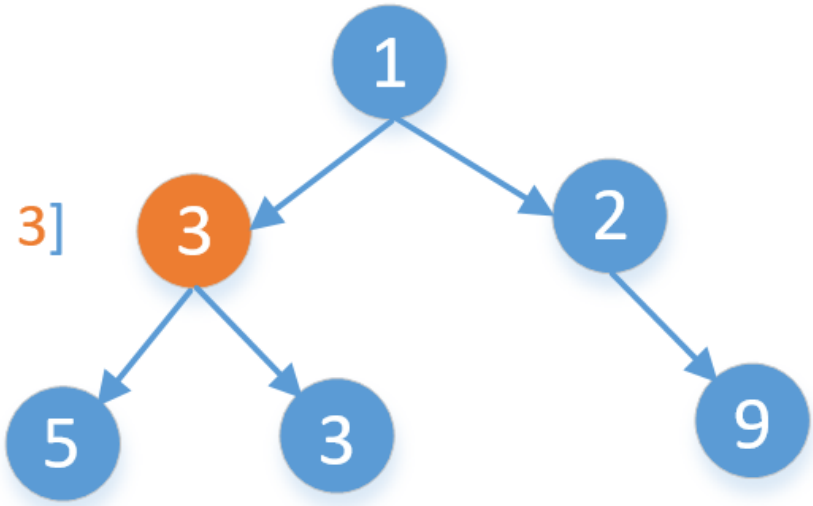
除了一层一层遍历以外，我们还可以使用DFS（深度优先搜索算法）来求解。我们就以上面的举例来画个图分析一下



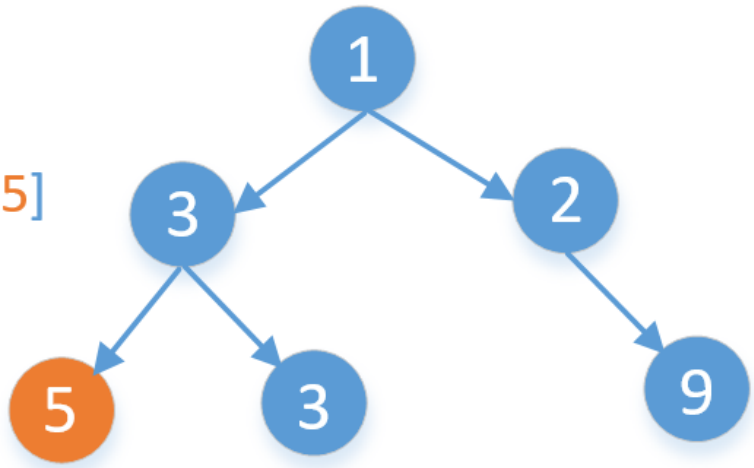
第一步　　res=[1]



第二步　　res=[1，3]



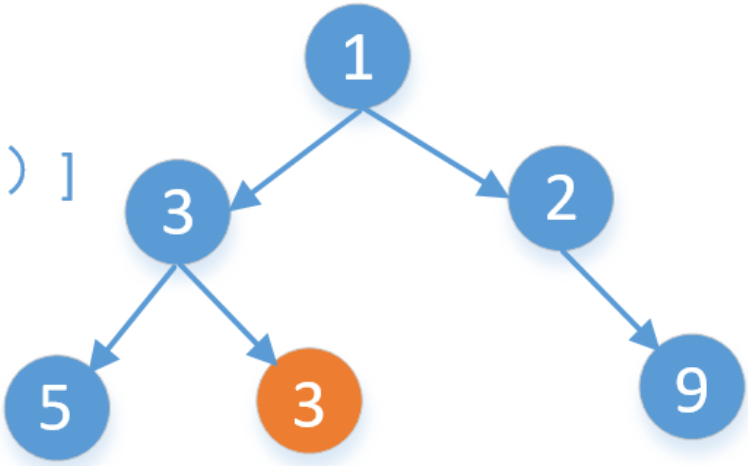
第三步　res=[1，3，5]



第四步

res=[1，3，max（5,3）]

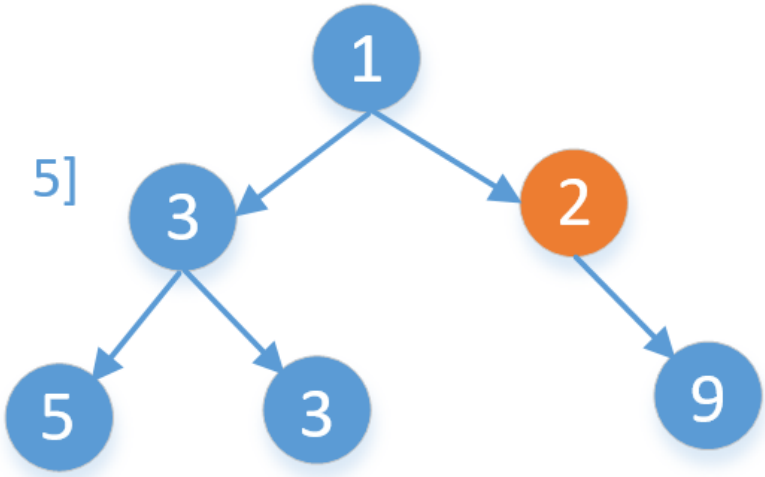
res=[1，3，5]



第五步

res=[1，max（3,2），5]

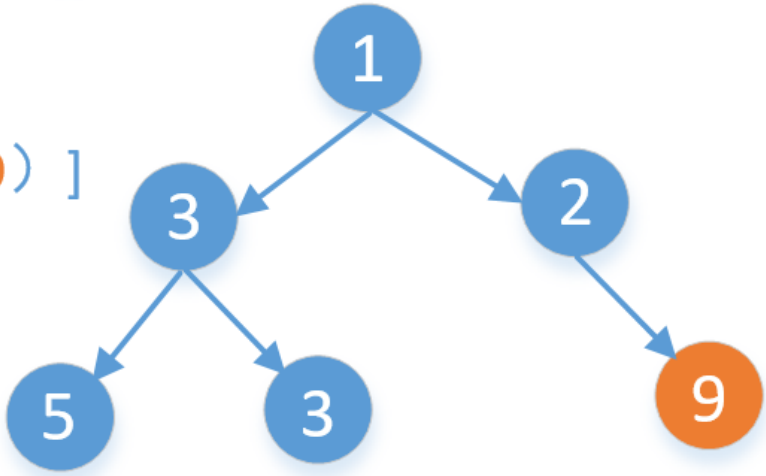
res=[1，3，5]



第六步

res=[1，3，max（5,9）]

res=[1，3，9]



上面的橙色结点就是遍历的顺序，看明白了上面的图，代码就很容易写出来了，我们再来看下代码

```java
public List<Integer> largestValues(TreeNode root) {
    List<Integer> res = new ArrayList<>();
    helper(root, res, 1);
    return res;
}

//level表示的是第几层，集合res中的第一个数据表示的是
// 第一层的最大值，第二个数据表示的是第二层的最大值……
private void helper(TreeNode root, List<Integer> res, int level) {
    if (root == null)
        return;
    //如果走到下一层了直接加入到集合中
    if (level == res.size() + 1) {
        res.add(root.val);
    } else {
        //注意：我们的level是从1开始的，也就是说root
        // 是第一层，而集合list的下标是从0开始的，
        // 所以这里level要减1。
        // Math.max(res.get(level - 1), root.val)表示的
        // 是遍历到的第level层的root.val值和集合中的第level
        // 个元素的值哪个大，就要哪个。
        res.set(level - 1, Math.max(res.get(level - 1), root.val));
    }
    //下面两行是DFS的核心代码
    helper(root.left, res, level + 1);
    helper(root.right, res, level + 1);
}
```

往期推荐

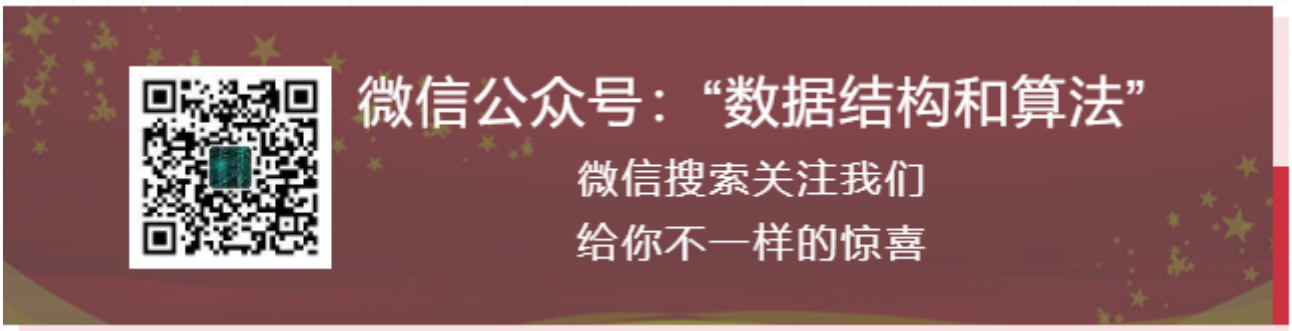
- 374，二叉树的最小深度
- 373，数据结构-6,树
- 372，二叉树的最近公共祖先
- 367，二叉树的最大深度

374，二叉树的最小深度

原创 山大王wld 数据结构和算法 6月2日

收录于话题
#算法图文分析

96个 >



What they don’t get is that when someone’s struggling, it means he’s strong.

他们不懂，当一个人在挣扎，那意味着他很强大。

题目描述

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

说明: 叶子节点是指没有子节点的节点。

示例:

给定二叉树 [3,9,20,null,null,15,7],

```
    3
   / \
```

```
  9  20
    /  \
   15   7
```

返回它的最小深度 2。

问题分析：

这题其实不难，看到这道题我们首先想到的是BFS，就是一层一层的遍历，如果某一层的某个节点没有子节点了，我们就返回这个节点的层数即可。

比如上面的9在第二层，他没有子节点了，我们直接返回他所在的层数2即可，没必要在遍历第3层了。代码很简单，我们来看下。

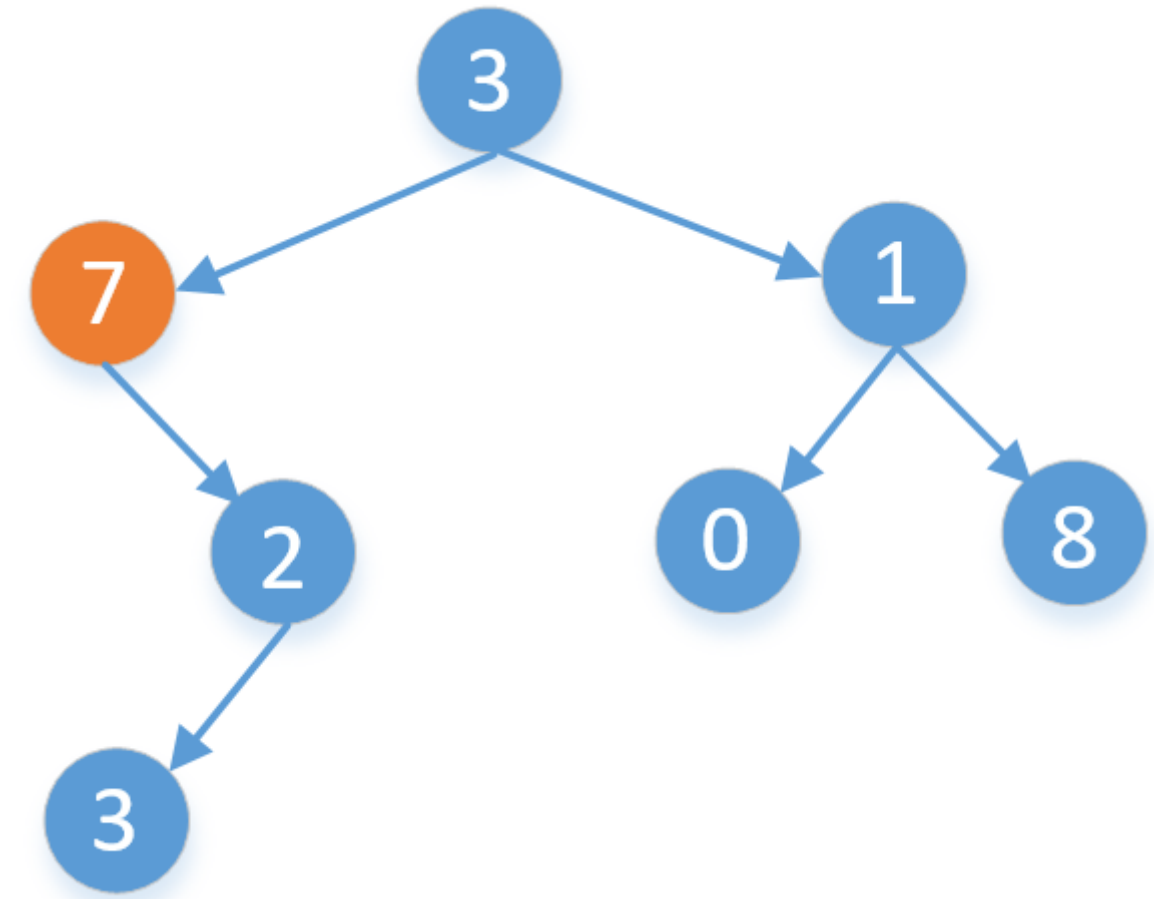
01 非递归写法

```
1   public int minDepth(TreeNode root) {
2     if (root == null)
3       return 0;
4     Queue<TreeNode> queue = new LinkedList<>();
5     queue.add(root);//入队
6     int level = 0;
7     while (!queue.isEmpty()) {//队列不为空就继续循环
8       level++;
9       int levelCount = queue.size();
10      for (int j = 0; j < levelCount; j++) {
11        TreeNode node = queue.poll();//出队
12        //如果当前node节点的左右子树都为空，直接返回level即可
13        if (node.left == null && node.right == null)
14          return level;
15        if (node.left != null)
16          queue.add(node.left);
17        if (node.right != null)
18          queue.add(node.right);
19      }
20    }
21    return -1;
22  }
```

02 递归写法

我们还可以使用递归的方式，返回Math.min(左子树的深度，右子树的深度)+1，看起来很有道理，但有一个问题，如果左右子树都不为空或者都为空是没问题的。但如果左右子树一个为空一个不为空，就会有问题了，因为为空的那个子节点的深度是0，我们不能用它，所以这里要有个判断。

> 比如下面7的左子树的深度是0，但他还有右子树，所以我们不能选择深度最小的（因为这时7的左子树的深度是0）。




```java
public int minDepth(TreeNode root) {
  if (root == null)
    return 0;
  //左子树的最小深度
  int left = minDepth(root.left);
  //右子树的最小深度
  int right = minDepth(root.right);
  //如果left和right都为0，我们返回1即可，
  //如果left和right只有一个为0，说明他只有一个子结点，我们只需要返回它另一个子节点的最小深度+1即可。
  //如果left和right都不为0，说明他有两个子节点，我们只需要返回最小深度的+1即可。
  return (left == 0 || right == 0) ? left + right + 1 : Math.min(left, right) + 1;
}
```

或者我们还可以换种方式

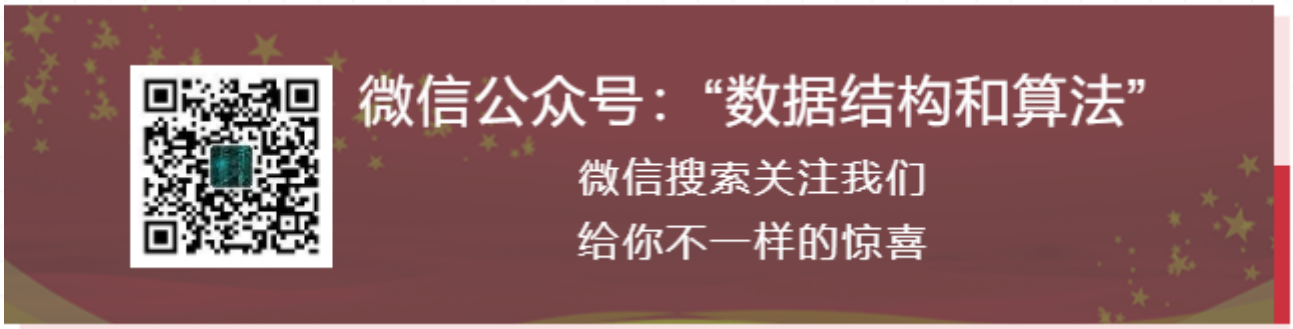
```java
public static int minDepth(TreeNode root) {
  if (root == null)
    return 0;
  //如果左子树等于空，我们返回右子树的最小高度+1
  if (root.left == null)
    return minDepth(root.right) + 1;
  //如果右子树等于空，我们返回左子树的最小高度+1
  if (root.right == null)
    return minDepth(root.left) + 1;
  //如果左右子树都不为空，我们返回左右子树深度最小的那个+1
  return Math.min(minDepth(root.left), minDepth(root.right)) + 1;
}
```

372，二叉树的最近公共祖先

原创 山大王wld 数据结构和算法 5月30日

收录于话题
#算法图文分析

96个 >

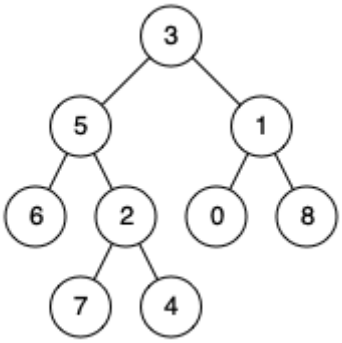


Believe you can and you're halfway there.

相信你自己能做到，你就已经成功一半了。

问题描述：

给定一个二叉树，找到该树中两个指定节点的最近公共祖先。
例如，给定如下二叉树:
root = [3,5,1,6,2,0,8,null,null,7,4]



示例 1:

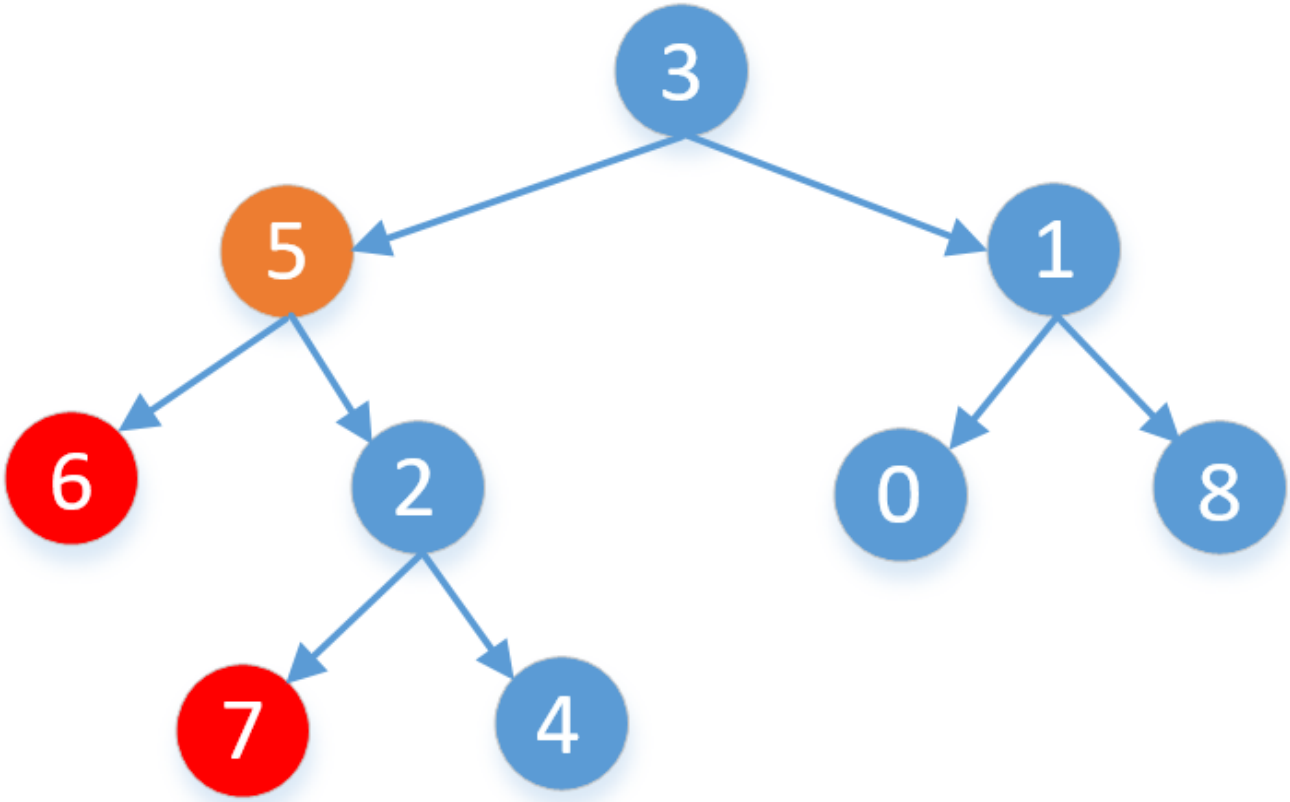
> 输入:
> root = [3,5,1,6,2,0,8,null,null,7,4]
> p = 5, q = 1
>
> 输出: 3
> 解释: 节点 5 和节点 1 的最近公共祖先是节点 3。

示例 2:

> 输入:
> root = [3,5,1,6,2,0,8,null,null,7,4]
> p = 5, q = 4
>
> 输出: 5
> 解释: 节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。

问题分析:

> 要想找到两个节点的最近公共祖先节点，我们可以从两个节点往上找，每个节点都往上走，一直走到根节点，那么根节点到这两个节点的连线肯定有相交的地方，如果是从上往下走，那么最后一次相交的节点就是他们的最近公共祖先节点。我们就以找6和7的最近公共节点来画个图看一下



我们看到6和7公共祖先有5和3，但最近的是5。我们只要往上找，找到他们第一个相同的公共祖先节点即可，但怎么找到每个节点的父节点呢，我们只需要把每个节点都遍历一遍，然后顺便记录他们的父节点存储在Map中。我们先找到其中的一条路径，比如6→5→3，然后在另一个节点往上找，由于7不在那条路径上，我们找7的父节点是2，2也不在那条路径上，我们接着往上找，2的父节点是5，5在那条路径上，所以5就是他们的最近公共子节点。

其实这里我们可以优化一下，我们没必要遍历所有的结点，我们一层一层的遍历（也就是BFS），只需要这两个节点都遍历到就可以了，比如上面2和8的公共结点，我们只需要遍历到第3层，把2和8都遍历到就行了，没必要再遍历第4层了。

我们来看下代码

01 非递归写法

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    //记录遍历到的每个节点的父节点。
    Map<TreeNode, TreeNode> parent = new HashMap<>();
    Queue<TreeNode> queue = new LinkedList<>();
    parent.put(root, null);//根节点没有父节点，所以为空
    queue.add(root);
    //直到两个节点都找到为止。
    while (!parent.containsKey(p) || !parent.containsKey(q)) {
        //队列是一边进一边出，这里poll方法是出队，
        TreeNode node = queue.poll();
        if (node.left != null) {
            //左子节点不为空，记录下他的父节点
            parent.put(node.left, node);
```

```
            //左子节点不为空，把它加入到队列中
            queue.add(node.left);
        }
        //右节点同上
        if (node.right != null) {
            parent.put(node.right, node);
            queue.add(node.right);
        }
    }
    Set<TreeNode> ancestors = new HashSet<>();
    //记录下p和他的祖先节点，从p节点开始一直到根节点。
    while (p != null) {
        ancestors.add(p);
        p = parent.get(p);
    }
    //查看p和他的祖先节点是否包含q节点，如果不包含再看是否包含q的父节点……
    while (!ancestors.contains(q))
        q = parent.get(q);
    return q;
}
```

02 递归写法

这题我们还可以改一下，使用递归的写法，代码中有注释，就不在详细介绍。

```
public TreeNode lowestCommonAncestor(TreeNode cur, TreeNode p, TreeNode q) {
    if (cur == null || cur == p || cur == q)
        return cur;
    TreeNode left = lowestCommonAncestor(cur.left, p, q);
    TreeNode right = lowestCommonAncestor(cur.right, p, q);
    //如果left为空，说明这两个节点在cur结点的右子树上，我们只需要返回右子树查找的结果即可
    if (left == null)
        return right;
    //同上
    if (right == null)
        return left;
    //如果left和right都不为空，说明这两个节点一个在cur的左子树上一个在cur的右子树上，
    //我们只需要返回cur结点即可。
    return cur;
}
```

03 总结

这道题如果一开始就知道每个节点的父节点就更简单了，从每个节点到根节点我们都可以把它看成是一个链表，如果求两个节点的最近公共祖先节点，我们只需要找到这两个链表第一次的交点即可，所以这个时候又是另外一道算法题了。

367，二叉树的最大深度

原创 山大王wld 数据结构和算法 5月22日

收录于话题
#算法图文分析

96个 >

问题

> 给定一个二叉树，找出其最大深度。
> 二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。
> 说明: 叶子节点是指没有子节点的节点。
>
> 示例：
> 给定二叉树 [3,9,20,null,null,15,7]

```
    3
   / \
  9  20
    /  \
   15   7
```

返回它的最大深度 3 。

数据结构：

java：树节点的数据结构

```java
public class TreeNode {
  int val;
  TreeNode left;
  TreeNode right;

  TreeNode(int x) {
    val = x;
  }
}
```

C语言：树节点的数据结构

```c
struct TreeNode {
  int val;
  struct TreeNode *left;
```

```
    struct TreeNode *right;
};
```

C++：树节点的数据结构

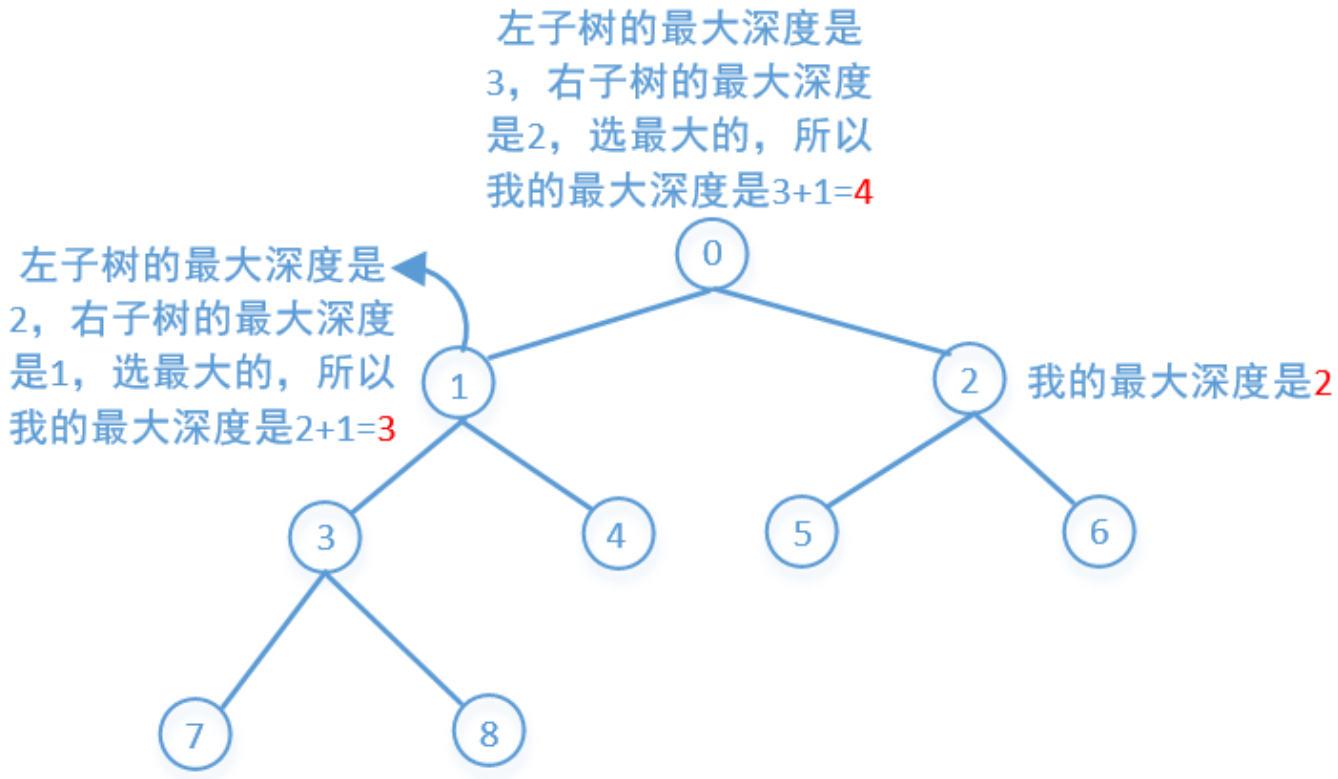
```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

递归写法

> 我们能想到的最简单的方式估计就是递归了，也就是下面这个图



> 如果对递归不熟悉的话可以看下我前面讲的关于复仇一个故事362，汉诺塔。下面我们来画个图来分析下



> 看明白了上面的过程，代码就容易多了，我们看下

java

```java
public int maxDepth(TreeNode root) {
  if (root == null)
    return 0;
  return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
}
```

C语言

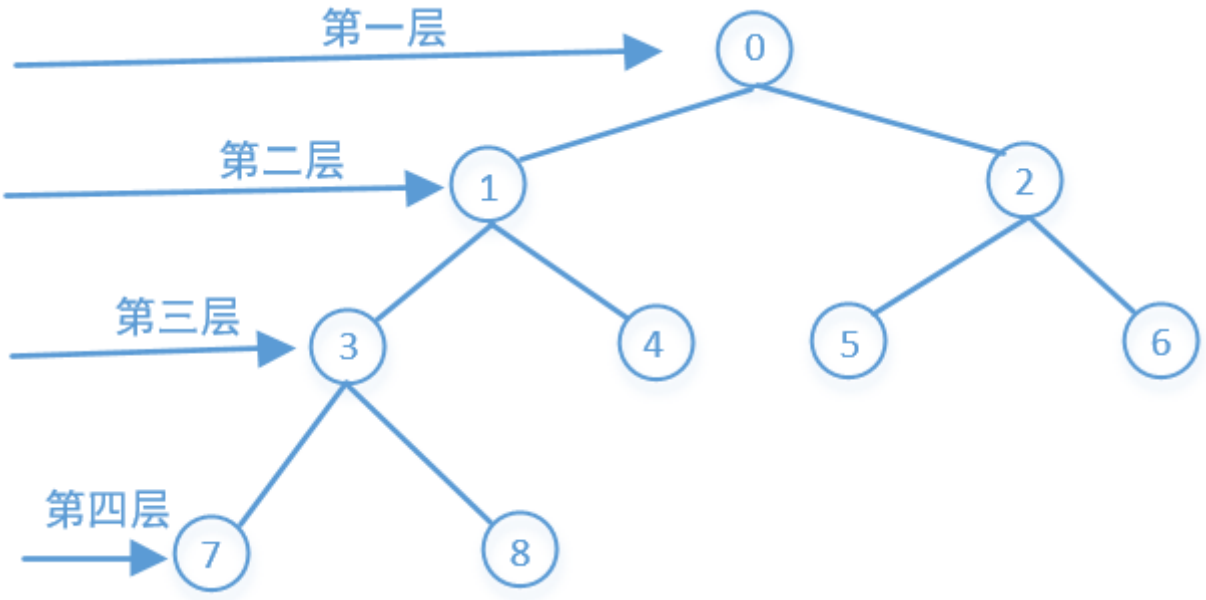
```c
int maxDepth(struct TreeNode* root) {
  if (root == NULL)
    return 0;
  return max(maxDepth(root -> left), maxDepth(root -> right)) + 1;
}

int max(int left, int right) {
  return left > right ? left : right;
}
```

C++

```cpp
public:
int maxDepth(TreeNode* root) {
  if (root == NULL)
    return 0;
  return max(maxDepth(root -> left), maxDepth(root -> right)) + 1;
}
```

BFS:

除了递归，我们还可能想到的就是BFS（宽度优先搜索算法（又称广度优先搜索）），他的实现原理就是一层层遍历，统计一下总共有多少层，我们来画个图分析一下。



一层一层往下走，统计总共有多少层，我们来看下代码

java

```java
public int maxDepth(TreeNode root) {
  if (root == null)
    return 0;
  Deque<TreeNode> stack = new LinkedList<>();
  stack.push(root);
  int count = 0;
  while (!stack.isEmpty()) {
    int size = stack.size();
    while (size-- > 0) {
      TreeNode cur = stack.pop();
      if (cur.left != null)
        stack.addLast(cur.left);
      if (cur.right != null)
        stack.addLast(cur.right);
    }
    count++;
  }
  return count;
}
```

C++

```cpp
public:
int maxDepth(TreeNode* root) {
  if (root == NULL)
    return 0;
  int res = 0;
  queue<TreeNode *>q;
  q.push(root);
  while (!q.empty()) {
    ++res;
    for (int i = 0, n = q.size(); i < n; ++i) {
      TreeNode * p = q.front();
      q.pop();
      if (p -> left != NULL)
```

```
            q.push(p -> left);
          if (p -> right != NULL)
            q.push(p -> right);
        }
      }
      return res;
    }
```

DFS:

> 想到BFS我们一般会和DFS联想到一起，DFS是深度优先搜索算法，我们先来看下代码

java

```java
public int maxDepth(TreeNode root) {
  if (root == null)
    return 0;
  Stack<TreeNode> stack = new Stack<>();
  Stack<Integer> value = new Stack<>();
  stack.push(root);
  value.push(1);
  int max = 0;
  while (!stack.isEmpty()) {
    TreeNode node = stack.pop();
    int temp = value.pop();
    max = Math.max(temp, max);
    if (node.left != null) {
      stack.push(node.left);
      value.push(temp + 1);
    }
    if (node.right != null) {
      stack.push(node.right);
      value.push(temp + 1);
    }
  }
  return max;
}
```

C++

```cpp
public:
int maxDepth(TreeNode*root) {
  if (root == NULL)
    return 0;
  stack<TreeNode *>nodeStack;
  stack<int> value;
  nodeStack.push(root);
  value.push(1);
  int max = 0;
  while (!nodeStack.empty()) {
    TreeNode * node = nodeStack.top();
    nodeStack.pop();
    int temp = value.top();
    value.pop();
    max = temp > max ? temp : max;
    if (node -> left != NULL) {
      nodeStack.push(node -> left);
      value.push(temp + 1);
    }
    if (node -> right != NULL) {
      nodeStack.push(node -> right);
```

```
22            value.push(temp + 1);
23          }
24        }
25        return max;
26    }
```

这里使用了两个栈，一个是存储节点的，一个是存储每个节点到根节点总共经过多少个节点（包含根节点和当前节点）。

596，删除排序链表中的重复元素 II

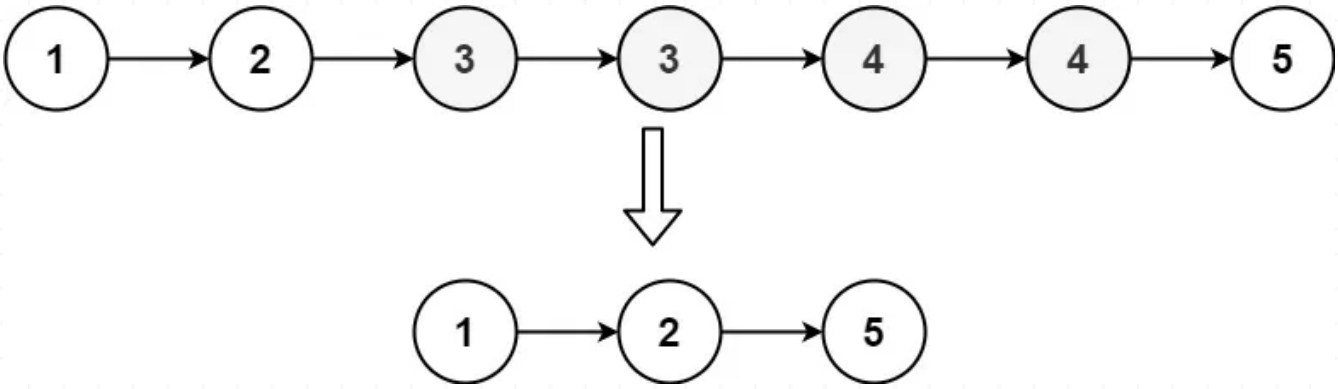
原创 彭哥 数据结构和算法 1周前

问题描述

来源：LeetCode第82题
难度：中等

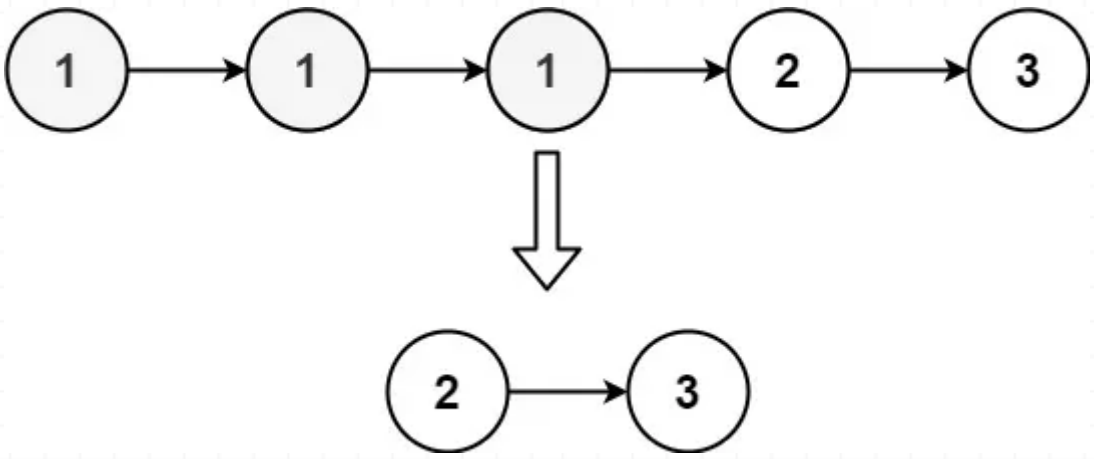
存在一个按升序排列的链表，给你这个链表的头节点head，请你删除链表中所有存在数字重复情况的节点，只保留原始链表中没有重复出现的数字。返回同样按升序排列的结果链表。

示例 1：



> **输入**：head = [1,2,3,3,4,4,5]
> **输出**：[1,2,5]

示例 2：



> **输入**：head = [1,1,1,2,3]

> **输出**：[2,3]

提示：

- 链表中节点数目在范围 [0, 300] 内
- -100 <= Node.val <= 100
- 题目数据保证链表已经按升序排列

双指针解决

前面我们刚讲过595，删除排序链表中的重复元素，这题和595题不同的是，如果有数字相同的节点，那么这些数字相同的节点要全部删除。

这题的解决思路就是使用两个指针，一个指针cur指向当前节点，一个指针pre指向当前节点cur的前一个节点。cur始终和他的下一个节点比较，如果相同就往后移，如果不相同我们就需要判断pre的下一个节点是否是cur，如果是cur说明没有相同的节点，如果不是cur说明有相同的节点，我们就要删除，叙述不太好理解，我做个视频来看一下



最后在来看下代码

```
public ListNode deleteDuplicates(ListNode head) {
   if (head == null || head.next == null)
      return head;
   //添加一个dummy节点
   ListNode dummy = new ListNode(0);
   //让dummy节点的next指针指向head。
   dummy.next = head;
   //指向当前遍历的节点
   ListNode cur = head;
   //指向当前节点pre的前一个节点
   ListNode pre = dummy;
   while (cur != null) {
```

```
    while (cur.next != null && cur.val == cur.next.val) {
      //如果有重复的，cur就一直往下走
      cur = cur.next;
    }
    //判断上面有没有重复的节点，如果pre.next == cur，说明没有
    //重复的节点。否则说明有重复的节点，然后还要把重复的节点给删除
    if (pre.next == cur) {
      pre = pre.next;
    } else {
      //有重复的就删除
      pre.next = cur.next;
    }
    cur = cur.next;
  }
  return dummy.next;
}
```

递归方式解决

除了上面的方式以外，我们还可以使用递归的方式来解决，我们先定义一个函数deleteDuplicates(ListNode head)表示删除重复的节点。

如果head.val != head.next.val，也就是说当前节点和他的下一个节点值不一样，我们不做任何的删除，直接递归head节点的下一个节点，也就是

```
head.next = deleteDuplicates(head.next);
```

如果head.val == head.next.val，说明有重复的节点，这里到底是重复一个还是重复多个，我们不知道，需要通过一个循环来确定。然后把重复的全部删除，也就是

```
while (head.next != null && head.val == head.next.val)
    head = head.next;
return deleteDuplicates(head.next);
```

那递归的终止条件是什么呢，就是节点为空，或者只有一个节点，这种情况下是不可能有重复的，直接返回即可。我们来看下完整代码

```
public ListNode deleteDuplicates(ListNode head) {
  if (head == null || head.next == null)
    return head;
  if (head.val != head.next.val) {
    //如果当前节点和下一个节点的值不相同
    head.next = deleteDuplicates(head.next);
    return head;
  } else {
    //如果当前节点和下一个节点的值相同，说明出现了重复的，
    //把重复的全部给删除
    while (head.next != null && head.val == head.next.val)
      head = head.next;
    return deleteDuplicates(head.next);
  }
}
```

595，删除排序链表中的重复元素

原创 博哥 数据结构和算法 1周前

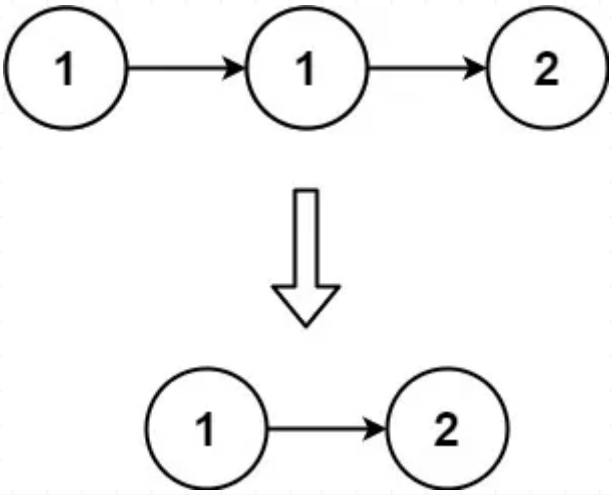
问题描述

来源：LeetCode第83题
难度：简单

存在一个按升序排列的链表，给你这个链表的头节点head，请你删除所有重复的元素，使每个元素只出现一次。

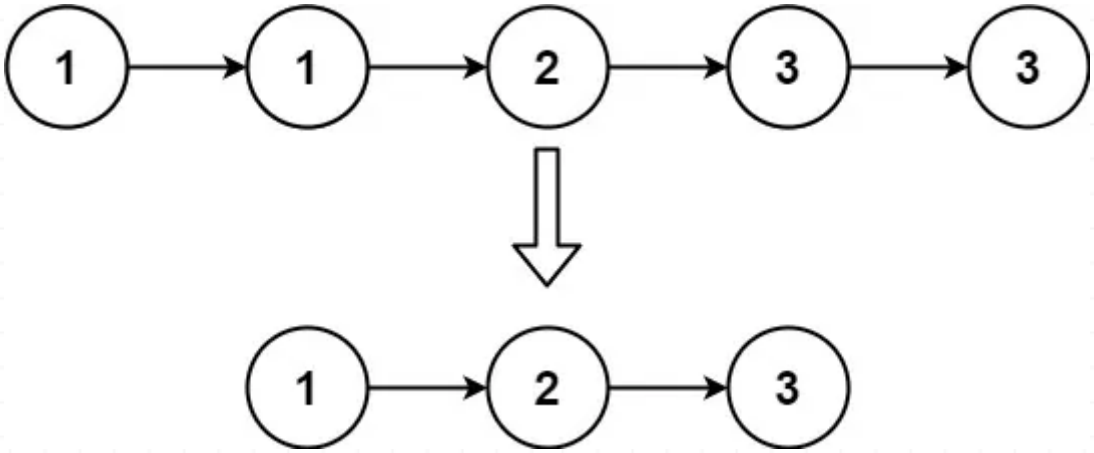
返回同样按升序排列的结果链表。

示例 1：



> **输入**：head = [1,1,2]
> **输出**：[1,2]

示例 2：



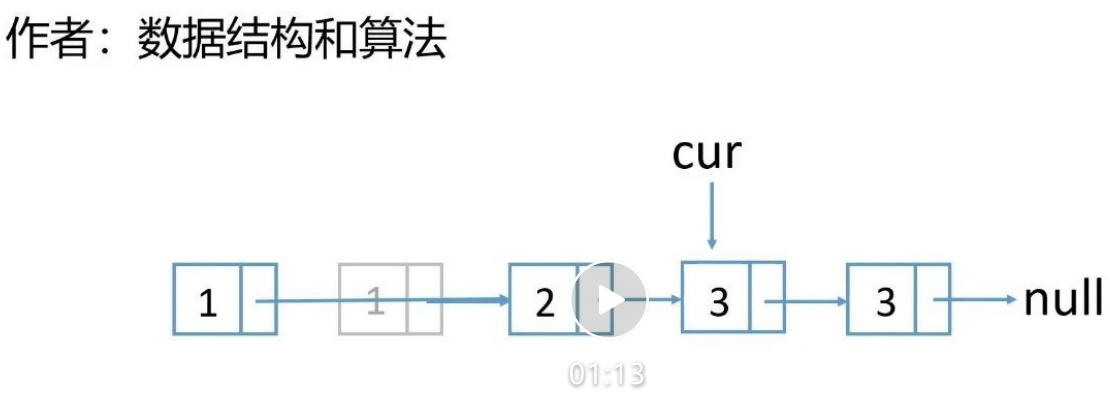
> **输入**：head = [1,1,2,3,3]
> **输出**：[1,2,3]

提示：

- 链表中节点数目在范围 [0, 300] 内
- -100 <= Node.val <= 100
- 题目数据保证链表已经按升序排列

使用一个指针解决

这题说了链表中的值是按照升序排列的，既然是排过序的，那么相同的节点肯定是挨着的。我们可以使用一个指针cur，每次都要判断是否和他后面的节点值相同，如果相同就把后面的那个节点给删除，这里就以示例2为例来看个视频



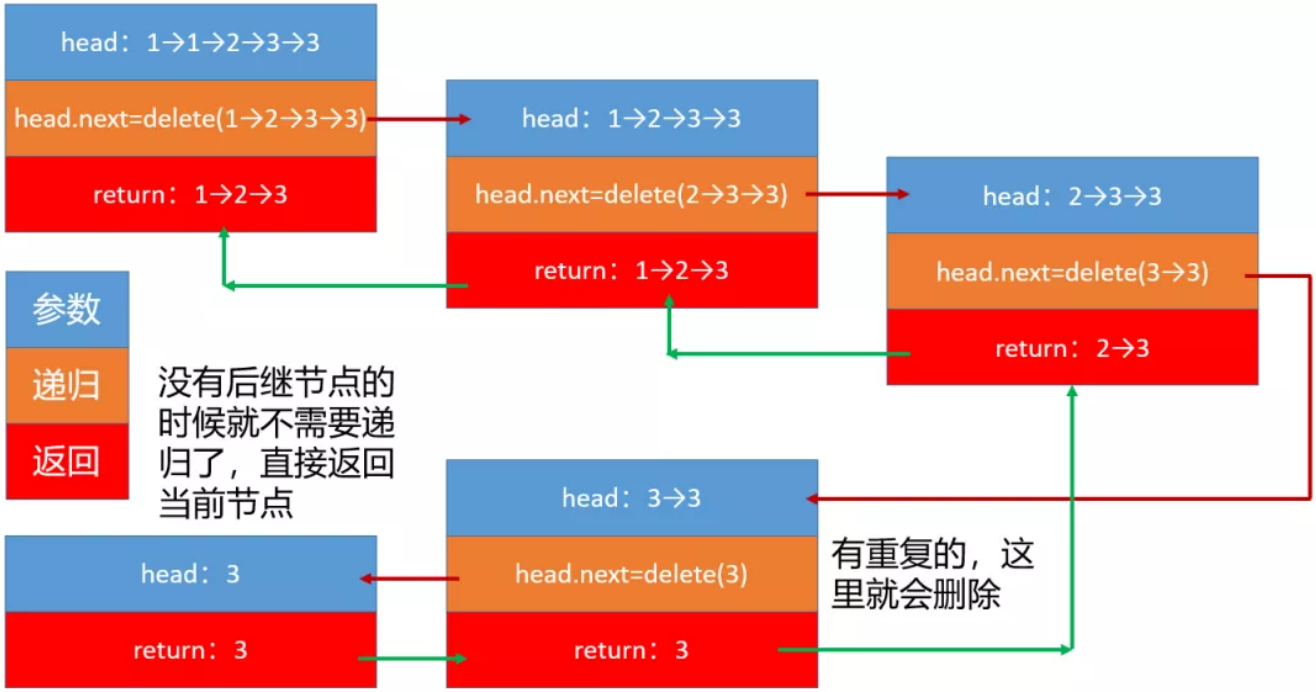
最后再来看下代码

```
public ListNode deleteDuplicates(ListNode head) {
```

```
    //如果但前节点是空，或者是单个节点，直接返回
    if (head == null || head.next == null)
        return head;
    //只用一个指针cur指向当前节点
    ListNode cur = head;
    while (cur.next != null) {
        //如果当前节点的值和下一个节点的值相同，
        //就把下一个节点值给删除
        if (cur.val == cur.next.val) {
            cur.next = cur.next.next;
        } else {
            //否则cur就往后移一步
            cur = cur.next;
        }
    }
    return head;
}
```

递归方式解决

除了上面使用一个指针以外，我们还可以使用递归的方式来解决。这个需要对链表的逆序访问比较熟悉，关于链表的逆序访问也可以看下1，倒叙打印链表。我们还以示例1为例来画个图看一下（如果看不清，图片可点击放大）



最后再来看下代码

```
public ListNode deleteDuplicates(ListNode head) {
    //递归的边界条件判断
    if (head == null || head.next == null)
        return head;
    //递归，相当于从后往前遍历
    head.next = deleteDuplicates(head.next);
    //如果当前节点和下一个一样，直接返回下一个节点，否则
    //返回当前节点
    return head.val == head.next.val ? head.next : head;
}
```

554，反转链表 II

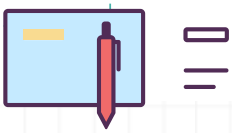
原创 博哥 数据结构和算法 5月18日

收录于话题
#算法图文分析

161个 >

History is apt to judge harshly those who sacrifice tomorrow for today.

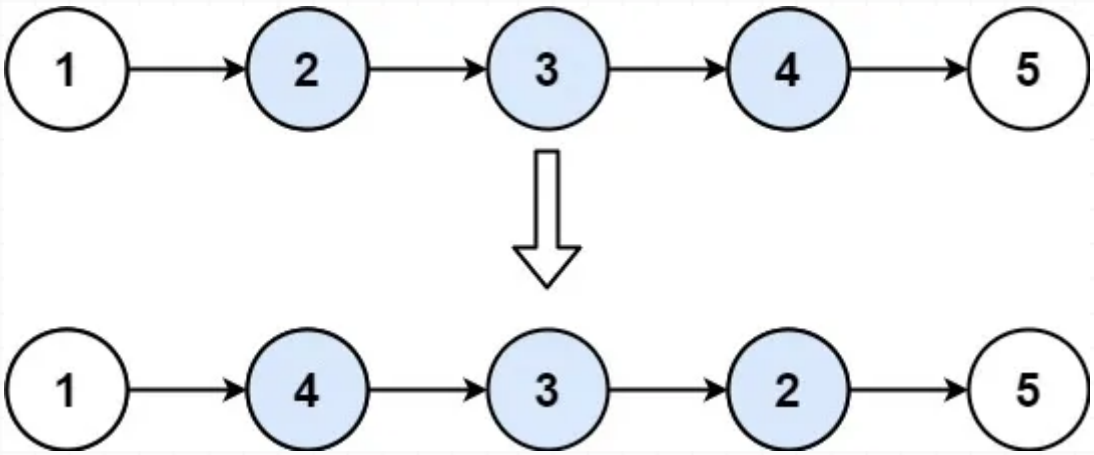
历史往往对那些为了今天而牺牲明天的人作出严厉的判决。



问题描述

来源：LeetCode第92题
难度：中等

给你单链表的头指针head和两个整数left和right，其中left<=right。请你反转从位置left到位置right的链表节点，返回反转后的链表 。



示例 1：

```
输入：head = [1,2,3,4,5], left = 2, right = 4
输出：[1,4,3,2,5]
```

示例 2：

> 输入：head = [5], left = 1, right = 1
> 输出：[5]

提示：

- 链表中节点数目为 n
- 1 <= n <= 500
- -500 <= Node.val <= 500
- 1 <= left <= right <= n

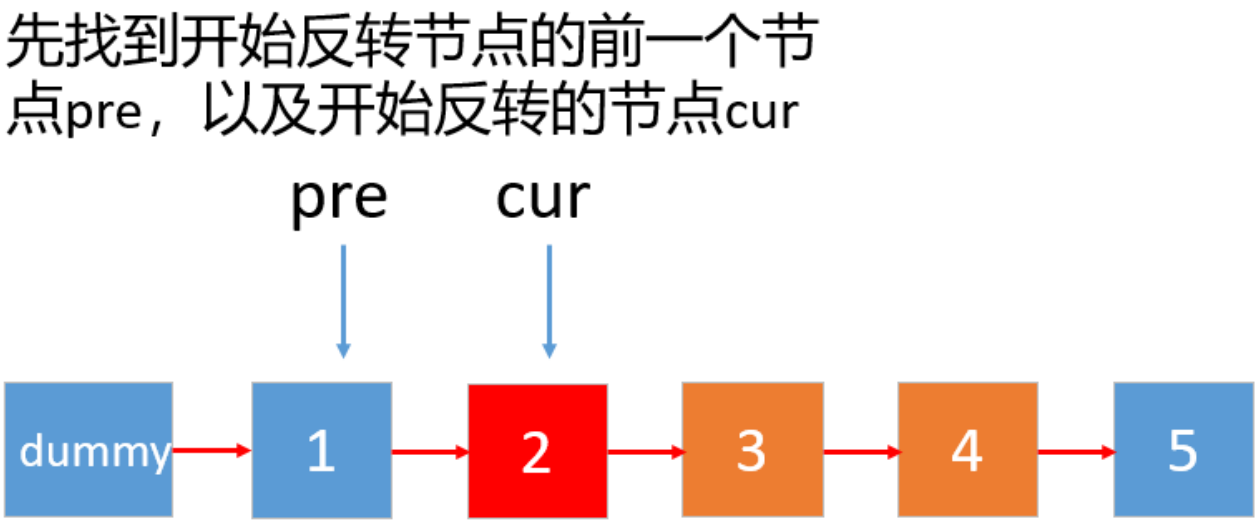
头插法解决

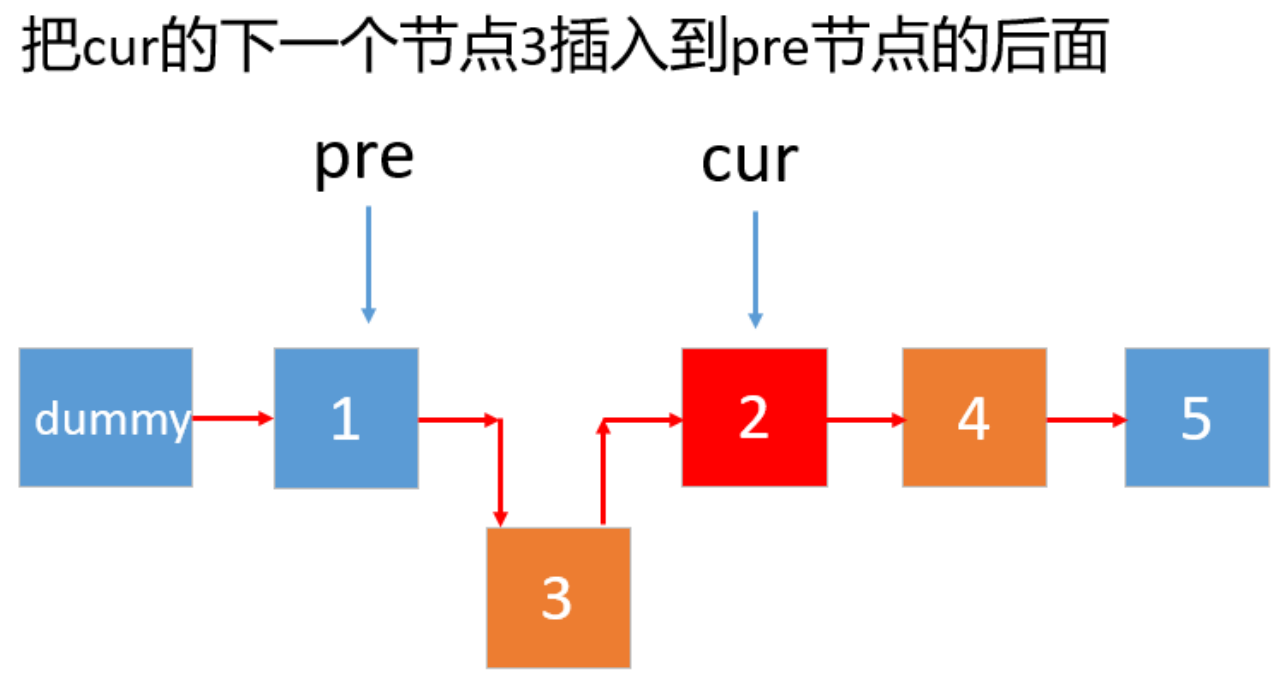
之前讲过链表的全部反转《432，剑指 Offer-反转链表的3种方式》，而这题只要求反转链表的部分节点，如果直接使用多个指针对需要反转的节点前后两两交换，也是可以解决的。

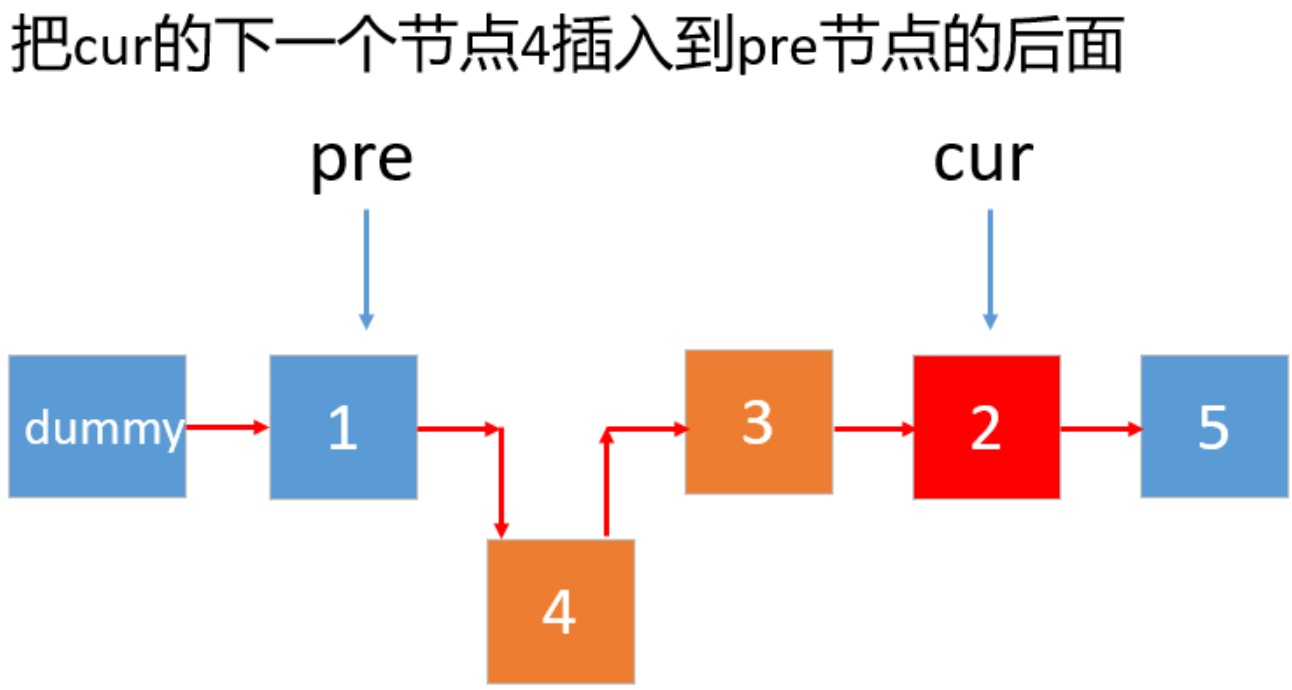
但这里我们使用一种更加容易理解的方式来解决，就是使用头插法，举个例子，比如我们要反转[1,2,3,4]

- 第一步2插入到前面[2,1,3,4]
- 第二步3插入到前面[3,2,1,4]
- 第三步4插入到前面[4,3,2,1]

只需要把后面不停的往前面插入即可完成反转，这里以示例一为例画个图来看下







再来看下代码

```
public ListNode reverseBetween(ListNode head, int m, int n) {
  //为了方便处理，先创建一个哑节点，让他指向head
  ListNode dummy = new ListNode(0);
  dummy.next = head;

  //记录开始反转节点的前一个节点
  ListNode pre = dummy;
  for (int i = 0; i < m - 1; i++) {
    pre = pre.next;
  }
  //记录开始反转的节点，我们把它后面需要反转的的节点
  //都移动到前面
  ListNode cur = pre.next;

  //采用头插法，把后面的节点都插入到前面
  for (int i = 0; i < n - m; i++) {
    ListNode next = cur.next;
    cur.next = next.next;
    next.next = pre.next;
    pre.next = next;
```

```
        }
        return dummy.next;
    }
```

总结

链表的节点交换一般没什么难度，但如果不仔细很容易出错，对于链表的交换最好在纸上一步步把它画出来，这样才更容易理解。

往期推荐

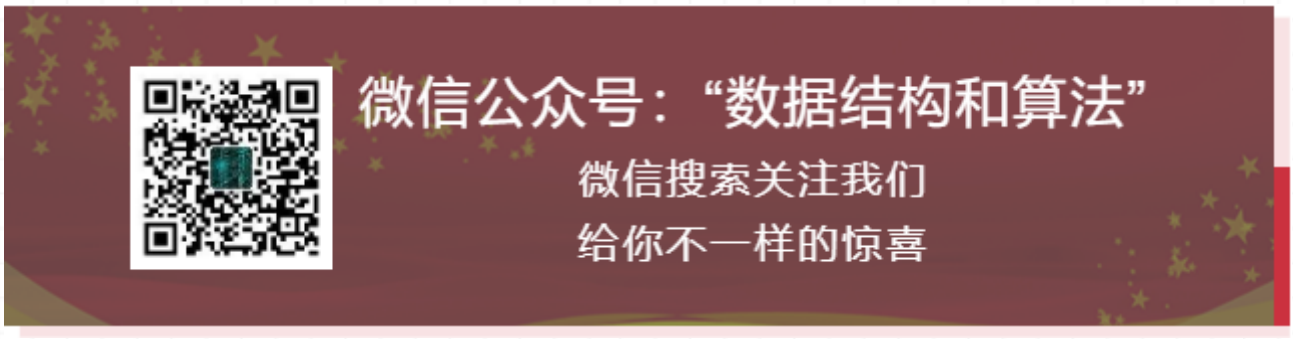
- 502，分隔链表的解决方式
- 466. 使用快慢指针把有序链表转换二叉搜索树
- 463. 判断回文链表的3种方式
- 432，剑指 Offer-反转链表的3种方式

502，分隔链表的解决方式

原创 山大王wld 数据结构和算法 1月5日

收录于话题
#算法图文分析

137个 >



It's when you start to become really afraid of death that you learn to appreciate life.

只有当你真正感受到对死亡的恐惧，你才会学到要珍惜生命。

问题描述

给你一个链表和一个特定值x，请你对链表进行分隔，使得所有小于x的节点都出现在大于或等于x的节点之前。

你应当保留两个分区中每个节点的初始相对位置。

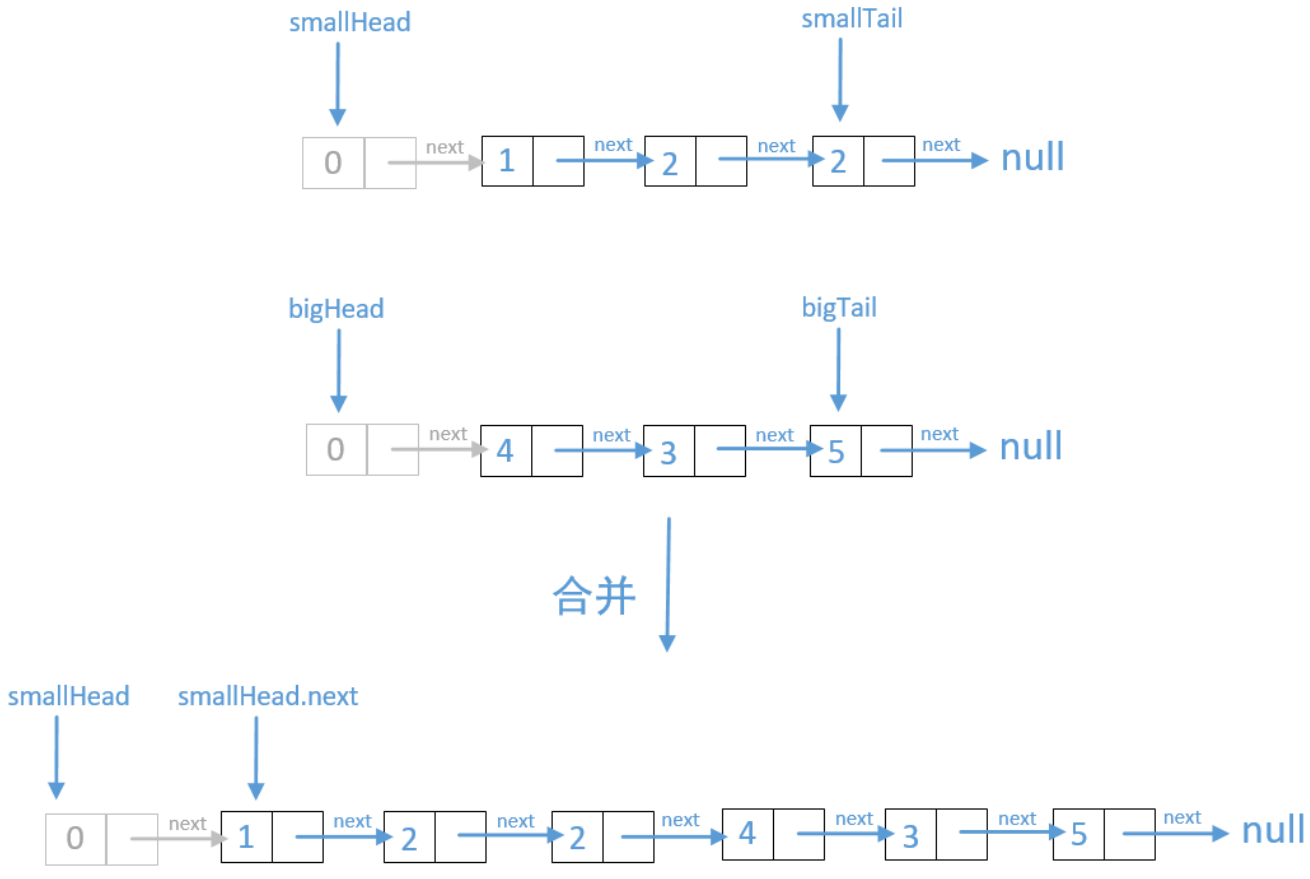
示例：

> **输入**：head = 1->4->3->2->5->2, x = 3
> **输出**：1->2->2->4->3->5

四指针解决

在算法中双指针我们经常听过，但四指针还是比较少的，四指针顾名思义就是使用4个指针来解决。

这题是让把节点值小于x的节点都放到前面，最简单的一种解决方式就是把原链表的节点分隔为两个链表，其中一个链表节点的值都是小于x的，另一个链表节点的值都是大于或等于x的，最后再把这两个链表合并即可。这里要使用四个指针，其中两个指向小的链表，两个指向大的链表，原理如下图所示



最后我们再来看下代码

```
1   public ListNode partition(ListNode head, int x) {
2     //小链表的头
3     ListNode smallHead = new ListNode(0);
4     //大链表的头
5     ListNode bigHead = new ListNode(0);
6     //小链表的尾
7     ListNode smallTail = smallHead;
8     //大链表的尾
9     ListNode bigTail = bigHead;
10    //遍历head链表
11    while (head != null) {
12      if (head.val < x) {
13        //如果当前节点的值小于x，则把当前节点挂到小链表的后面
14        smallTail = smallTail.next = head;
15      } else {//否则挂到大链表的后面
16        bigTail = bigTail.next = head;
17      }
18
19      //继续循环下一个结点
20      head = head.next;
21    }
22    //最后再把大小链表拼接在一块即可。
23    smallTail.next = bigHead.next;
24    bigTail.next = null;
```

```
25      return smallHead.next;
26  }
```

总结

这题不算难，注意最后把两个链表串起来的时候，大的链表是在后面，最后一定要让他的尾指针指向空，否则有可能会构成环。

往期推荐

- 466. 使用快慢指针把有序链表转换二叉搜索树
- 463. 判断回文链表的3种方式
- 462. 找出两个链表的第一个公共节点
- 460. 快慢指针解环形链表 II

463. 判断回文链表的3种方式

原创 山大王wld 数据结构和算法 10月17日

收录于话题
#算法图文分析

95个 >



Grow your way forward, through the triumphs and the setbacks.

在胜利与挫折的交错中不断成长。

问题描述

请判断一个链表是否为回文链表。链表为单向无环链表

示例 1:

> **输入**: 1->2
> **输出**: false

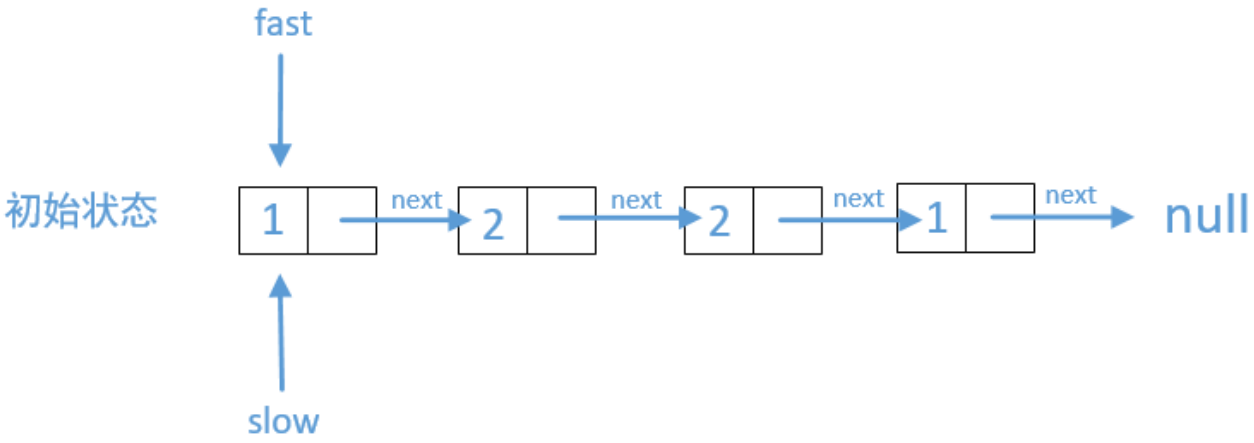
示例 2:

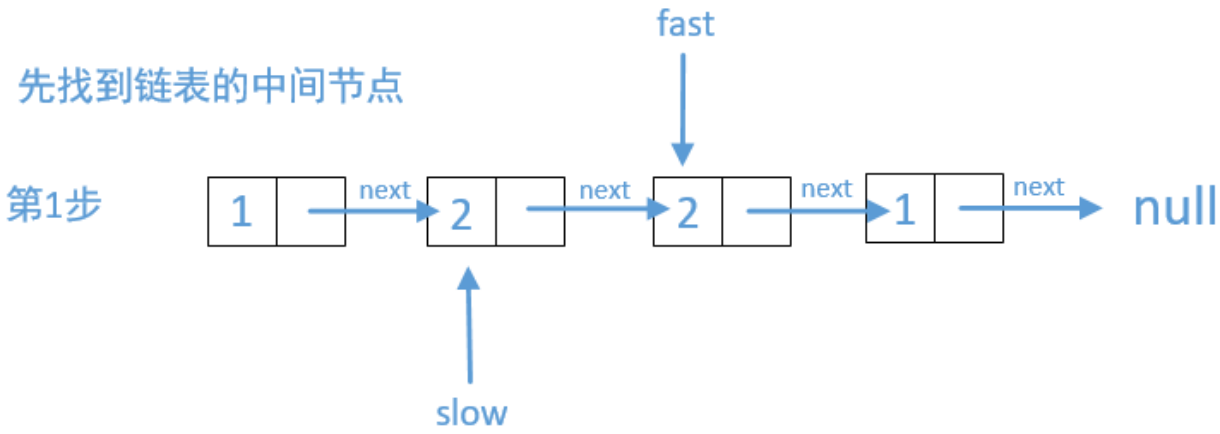
> **输入**: 1->2->2->1
> **输出**: true

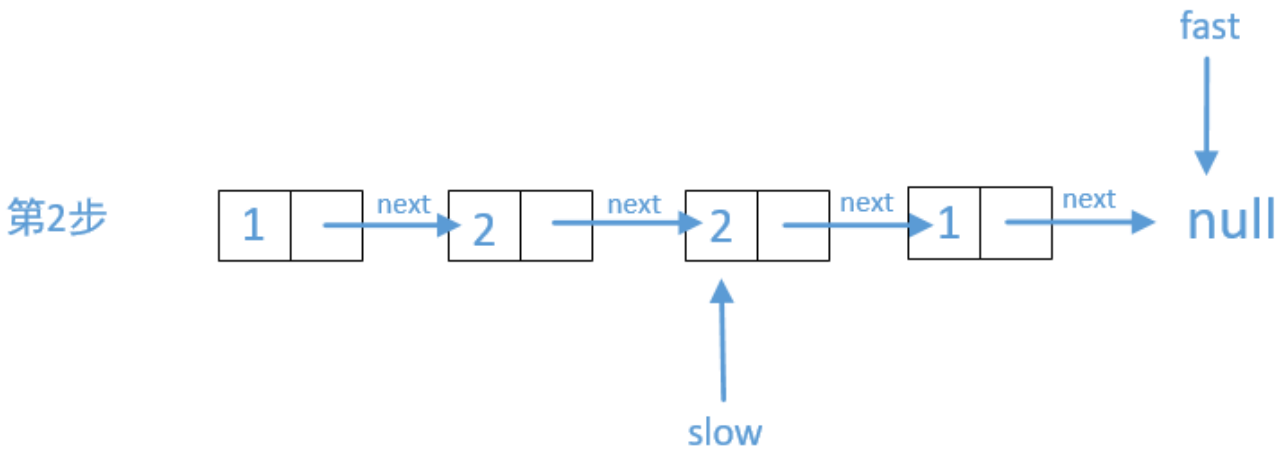
反转后半部分链表

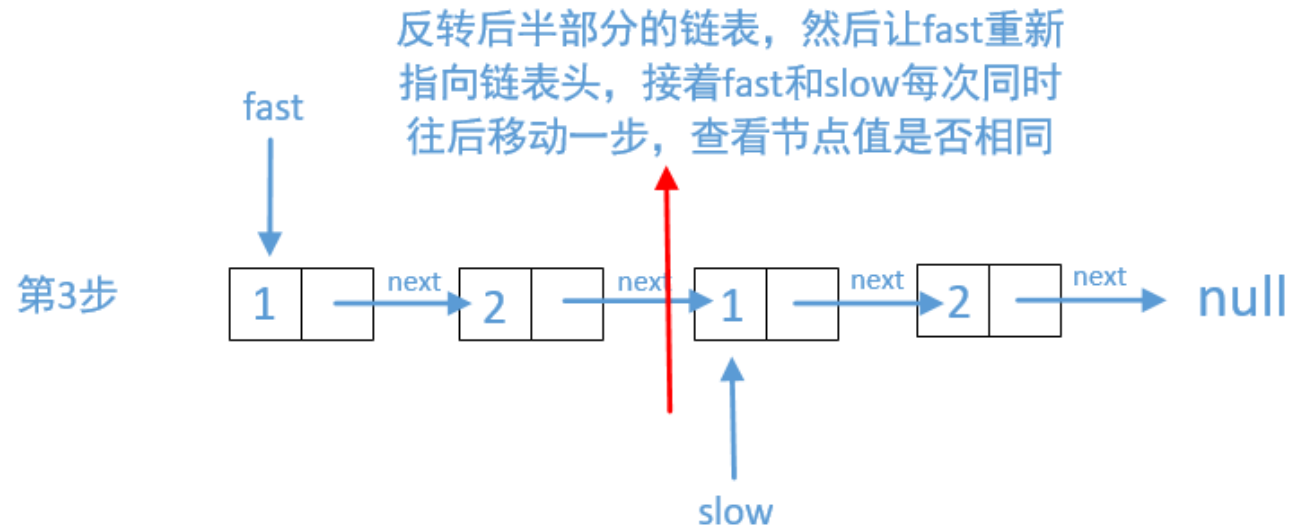
这题是让判断链表是否是回文链表，所谓的回文链表就是以链表中间为中心点两边对称。我们常见的有判断一个字符串是否是回文字符串，这个比较简单，可以使用两个指针，一个最左边一个最右边，两个指针同时往中间靠，判断所指的字符是否相等。

但这题判断的是链表，因为这里是单向链表，只能从前往后访问，不能从后往前访问，所以使用判断字符串的那种方式是行不通的。但我们可以通过找到链表的中间节点然后把链表后半部分反转（关于链表的反转可以看下432，剑指 Offer-反转链表的3种方式），最后再用后半部分反转的链表和前半部分一个个比较即可。这里以示例2为例画个图看一下。









最后再来看下代码

```
1   public boolean isPalindrome(ListNode head) {
2     ListNode fast = head, slow = head;
3     //通过快慢指针找到中点
4     while (fast != null && fast.next != null) {
5       fast = fast.next.next;
6       slow = slow.next;
7     }
8     //如果fast不为空，说明链表的长度是奇数个
9     if (fast != null) {
10      slow = slow.next;
11    }
12    //反转后半部分链表
13    slow = reverse(slow);
14
15    fast = head;
16    while (slow != null) {
17      //然后比较，判断节点值是否相等
18      if (fast.val != slow.val)
19        return false;
20      fast = fast.next;
21      slow = slow.next;
22    }
23    return true;
24  }
25
26  //反转链表
27  public ListNode reverse(ListNode head) {
28    ListNode prev = null;
29    while (head != null) {
30      ListNode next = head.next;
31      head.next = prev;
```

```
        prev = head;
        head = next;
    }
    return prev;
}
```

使用栈解决

我们知道栈是先进后出的一种数据结构，这里还可以使用栈先把链表的节点全部存放到栈中，然后再一个个出栈，这样就相当于链表从后往前访问了，通过这种方式也能解决，看下代码

```
public boolean isPalindrome(ListNode head) {
    ListNode temp = head;
    Stack<Integer> stack = new Stack();
    //把链表节点的值存放到栈中
    while (temp != null) {
        stack.push(temp.val);
        temp = temp.next;
    }

    //然后再出栈
    while (head != null) {
        if (head.val != stack.pop()) {
            return false;
        }
        head = head.next;
    }
    return true;
}
```

这里相当于链表从前往后全部都比较了一遍，其实我们只需要拿链表的后半部分和前半部分比较即可，没必要全部比较，所以这里可以优化一下

```
public boolean isPalindrome(ListNode head) {
    if (head == null)
        return true;
    ListNode temp = head;
    Stack<Integer> stack = new Stack();
    //链表的长度
    int len = 0;
    //把链表节点的值存放到栈中
    while (temp != null) {
        stack.push(temp.val);
        temp = temp.next;
        len++;
    }
    //len长度除以2
    len >>= 1;
    //然后再出栈
    while (len-- >= 0) {
        if (head.val != stack.pop())
            return false;
        head = head.next;
    }
    return true;
}
```

递归方式解决

我们知道，如果对链表逆序打印可以这样写

```java
private void printListNode(ListNode head) {
  if (head == null)
    return;
  printListNode(head.next);
  System.out.println(head.val);
}
```

也就是说最先打印的是链表的尾结点，他是从后往前打印的，看到这里是不是有灵感了，我们来对上面的对面进行改造一下

```java
ListNode temp;

public boolean isPalindrome(ListNode head) {
  temp = head;
  return check(head);
}

private boolean check(ListNode head) {
  if (head == null)
    return true;
  boolean res = check(head.next) && (temp.val == head.val);
  temp = temp.next;
  return res;
}
```

问题分析

回文链表的判断，相比回文字符串的判断稍微要麻烦一点，但难度也不是很大，如果对链表比较熟悉的话，这3种解决方式都很容易想到，如果不熟悉的话，可能最容易想到的就是第2种了，也就是栈和链表的结合。

如果对栈和链表不熟悉的话，可以看下352，数据结构-2,链表，这里详细介绍了单向链表，双向链表，以及环形链表的断开和连接。也可以看下363，数据结构-4,栈，这里有对栈的一些简单介绍和实例讲解。

往期推荐

- 460. 快慢指针解环形链表 II
- 455，DFS和BFS解被围绕的区域
- 450，什么叫回溯算法，一看就会，一写就废
- 446，回溯算法解黄金矿工问题

462. 找出两个链表的第一个公共节点

原创 山大王wld 数据结构和算法 10月16日

收录于话题
#算法图文分析

95个 >



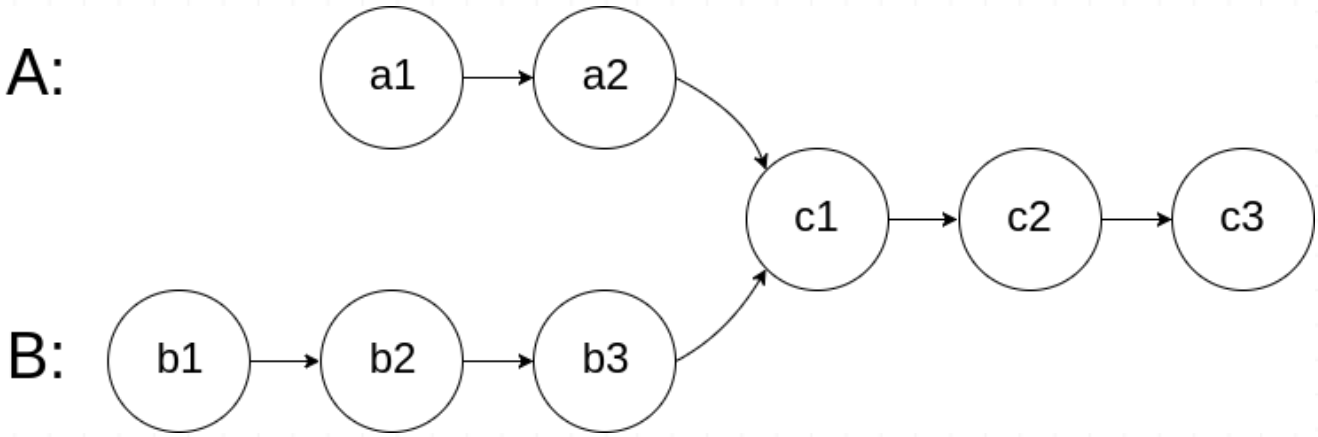
I am a slow walker, but I never walk backwards.

我走得很慢，但是我从来不会后退。

问题描述

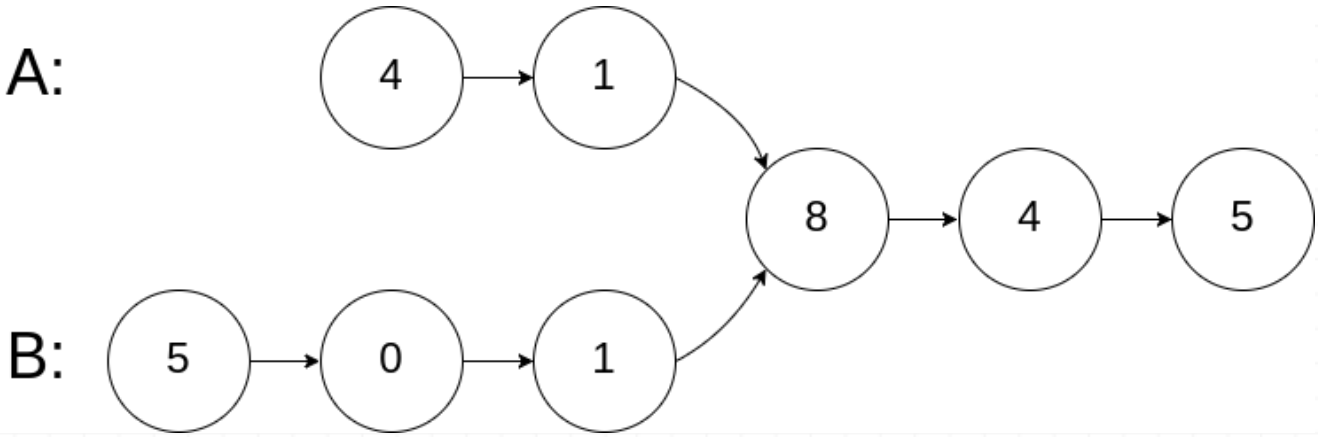
输入两个链表，找出它们的第一个公共节点。

如下面的两个链表：



在节点 c1 开始相交。

示例 1：

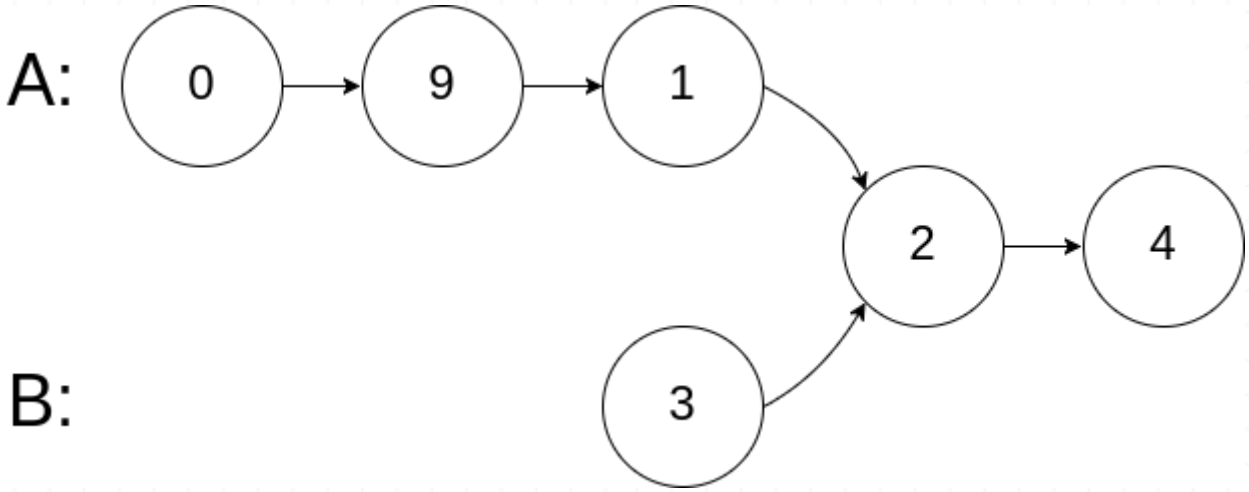


输入： intersectVal = 8,
listA = [4,1,8,4,5],
listB = [5,0,1,8,4,5],
skipA = 2, skipB = 3

输出： Reference of the node with value = 8

输入解释： 相交节点的值为 8 （注意，如果两个列表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

示例 2：

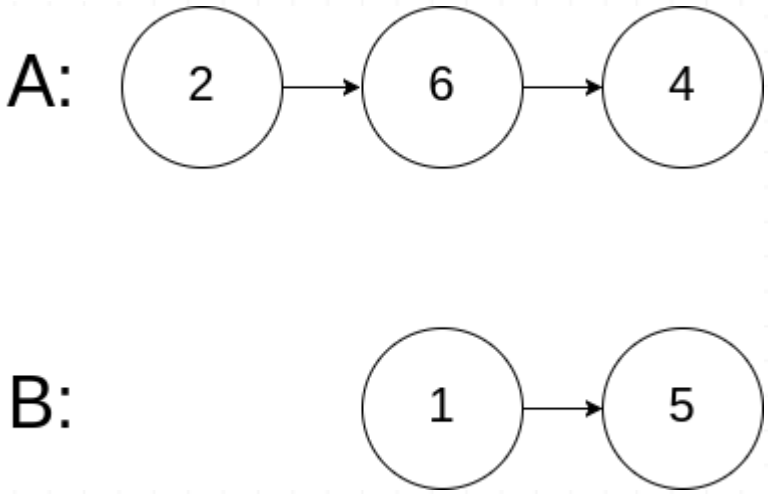


输入： intersectVal = 2,
listA = [0,9,1,2,4],
listB = [3,2,4],
skipA = 3, skipB = 1

输出： Reference of the node with value = 2

输入解释：相交节点的值为 2 （注意，如果两个列表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [0,9,1,2,4]，链表 B 为 [3,2,4]。在 A 中，相交节点前有 3 个节点；在 B 中，相交节点前有 1 个节点。

示例 3：



输入： intersectVal = 0,
listA = [2,6,4],
listB = [1,5],
skipA = 3, skipB = 2

输出： null

输入解释：从各自的表头开始算起，链表 A 为 [2,6,4]，链表 B 为 [1,5]。由于这两个链表不相交，所以 intersectVal 必须为 0，而 skipA 和 skipB 可以是任意值。
解释：这两个链表不相交，因此返回 null。

注意：

1. 如果两个链表没有交点，返回 null.
2. 在返回结果后，两个链表仍须保持原有的结构。
3. 可假定整个链表结构中没有循环。
4. 程序尽量满足 O(n) 时间复杂度，且仅用 O(1) 内存。

通过集合set解决

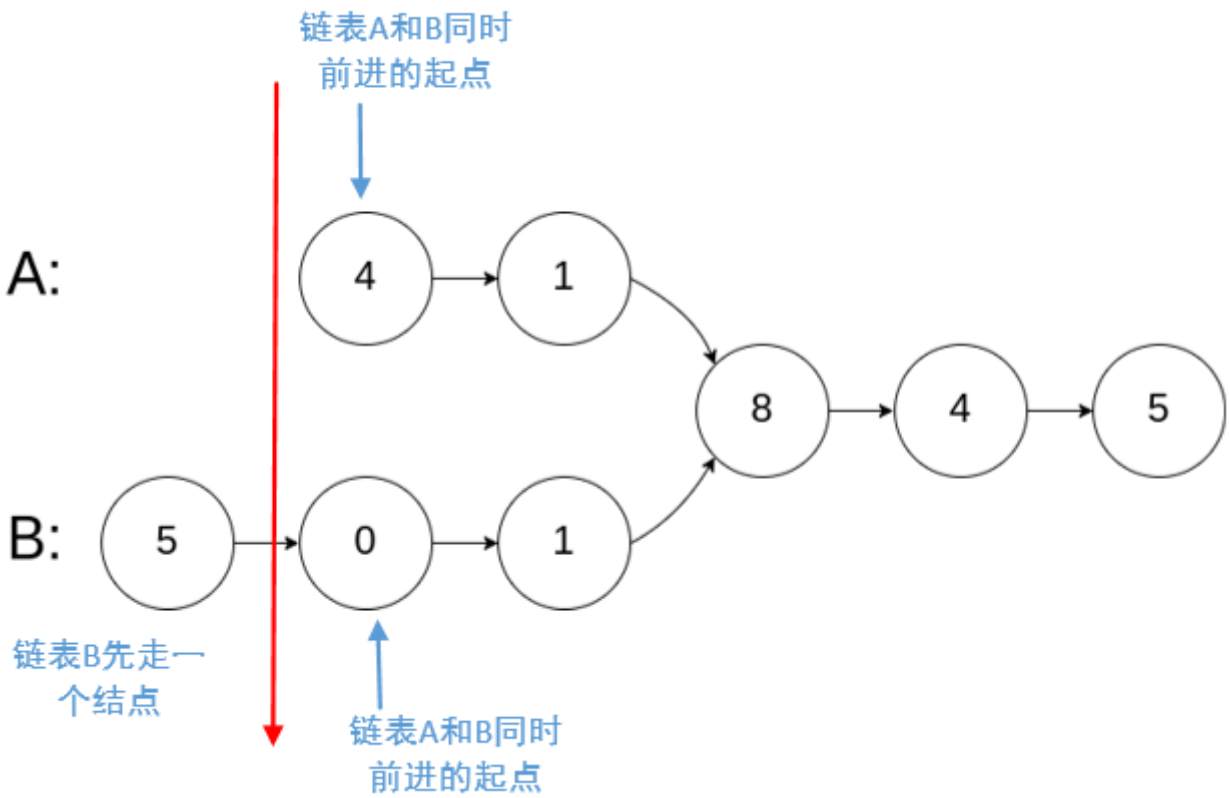
上面说了一大堆，其实就是判断两个链表是否相交，如果相交就返回他们的相交的交点，如果不相交就返回null。

做这题最容易想到的一种解决方式就是先把第一个链表的节点全部存放到集合set中，然后遍历第二个链表的每一个节点，判断在集合set中是否存在，如果存在就直接返回这个存在的结点。如果遍历完了，在集合set中还没找到，说明他们没有相交，直接返回null即可，原理比较简单，直接看下代码

```
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
  //创建集合set
  Set<ListNode> set = new HashSet<>();
  //先把链表A的结点全部存放到集合set中
  while (headA != null) {
    set.add(headA);
    headA = headA.next;
  }

  //然后访问链表B的结点，判断集合中是否包含链表B的结点，如果包含就直接返回
  while (headB != null) {
    if (set.contains(headB))
      return headB;
    headB = headB.next;
  }
  //如果集合set不包含链表B的任何一个结点，说明他们没有交点，直接返回null
  return null;
}
```

先统计两个链表的长度

还可以先统计两个链表的长度，如果两个链表的长度不一样，就让链表长的先走，直到两个链表长度一样，这个时候两个链表再同时每次往后移一步，看节点是否一样，如果有相等的，说明这个相等的节点就是两链表的交点，否则如果走完了还没有找到相等的节点，说明他们没有交点，直接返回null即可，来画个图看一下。



最后再来看下代码

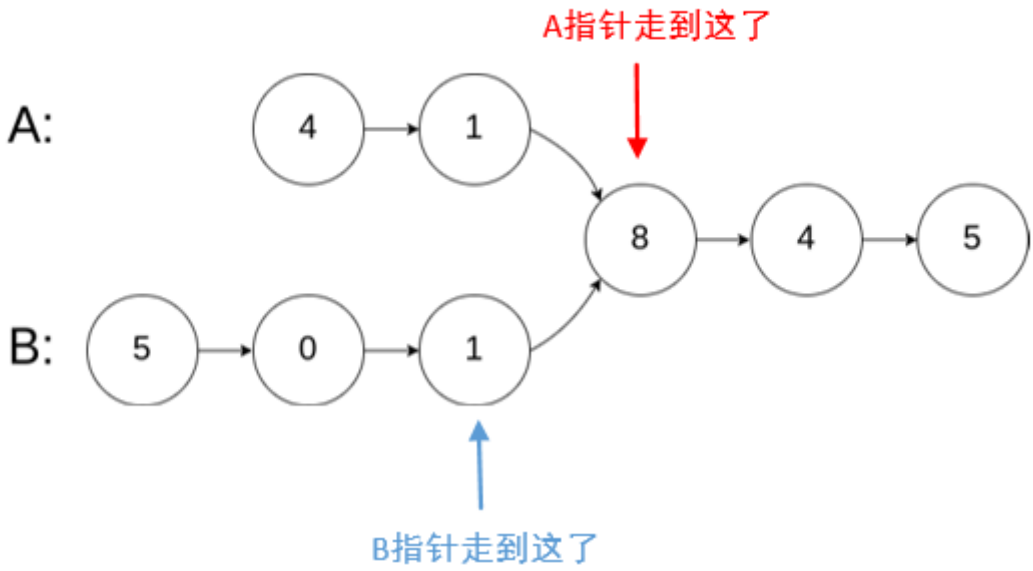
```java
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    //统计链表A和链表B的长度
    int lenA = length(headA), lenB = length(headB);

    //如果节点长度不一样，节点多的先走，直到他们的长度一样为止
    while (lenA != lenB) {
        if (lenA > lenB) {
            //如果链表A长，那么链表A先走
            headA = headA.next;
            lenA--;
        } else {
            //如果链表B长，那么链表B先走
            headB = headB.next;
            lenB--;
        }
    }

    //然后开始比较，如果他俩不相等就一直往下走
    while (headA != headB) {
        headA = headA.next;
        headB = headB.next;
    }
    //走到最后，最终会有两种可能，一种是headA为空，
    //也就是说他们俩不相交。还有一种可能就是headA
    //不为空，也就是说headA就是他们的交点
    return headA;
}

//统计链表的长度
private int length(ListNode node) {
    int length = 0;
    while (node != null) {
        node = node.next;
        length++;
    }
    return length;
}
```

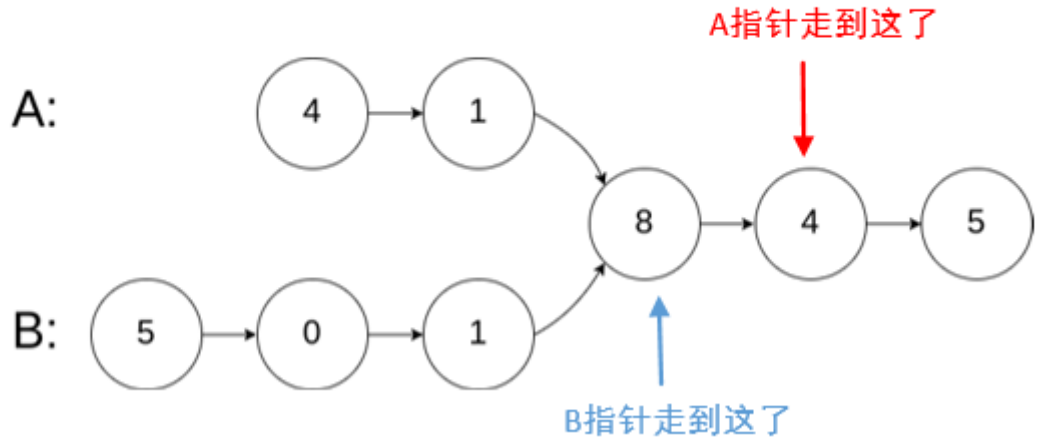
双指针解决

我们还可以使用两个指针，最开始的时候一个指向链表A，一个指向链表B，然后他们每次都要往后移动一位，顺便查看节点是否相等。如果链表A和链表B不相交，基本上没啥可说的，我们这里假设链表A和链表B相交。那么就会有两种情况，

一种是链表A的长度和链表B的长度相等，他们每次都走一步，最终在相交点肯定会相遇。

一种是链表A的长度和链表B的长度不相等，如下图所示

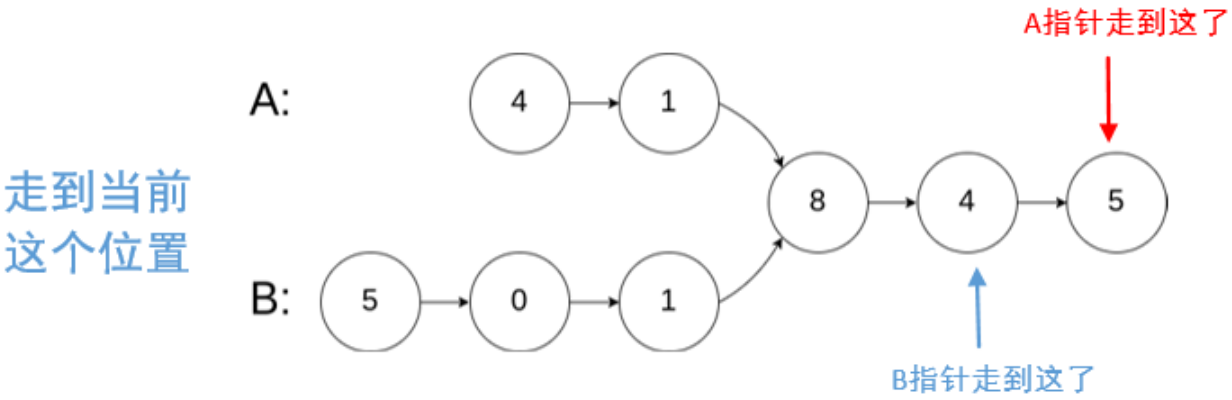


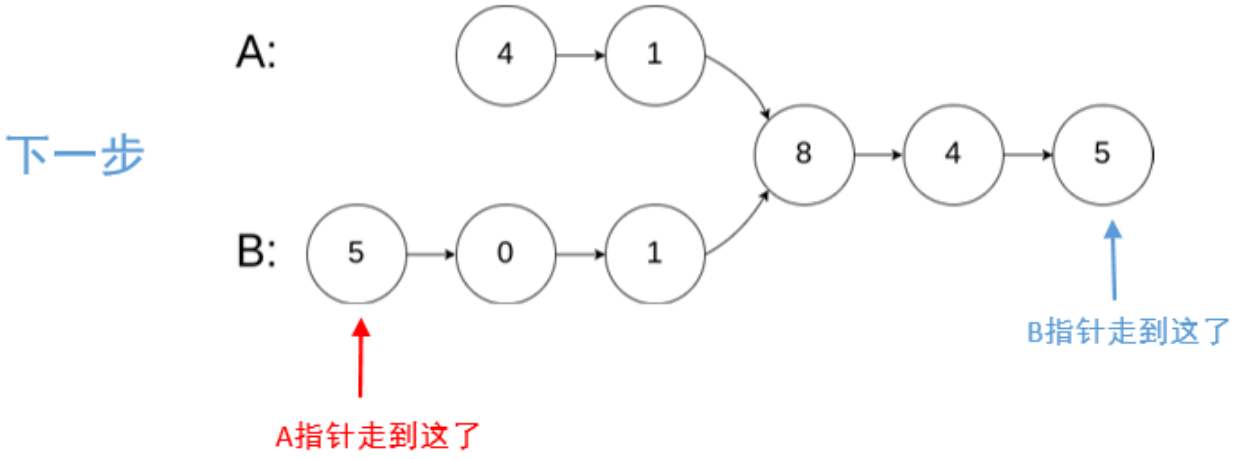


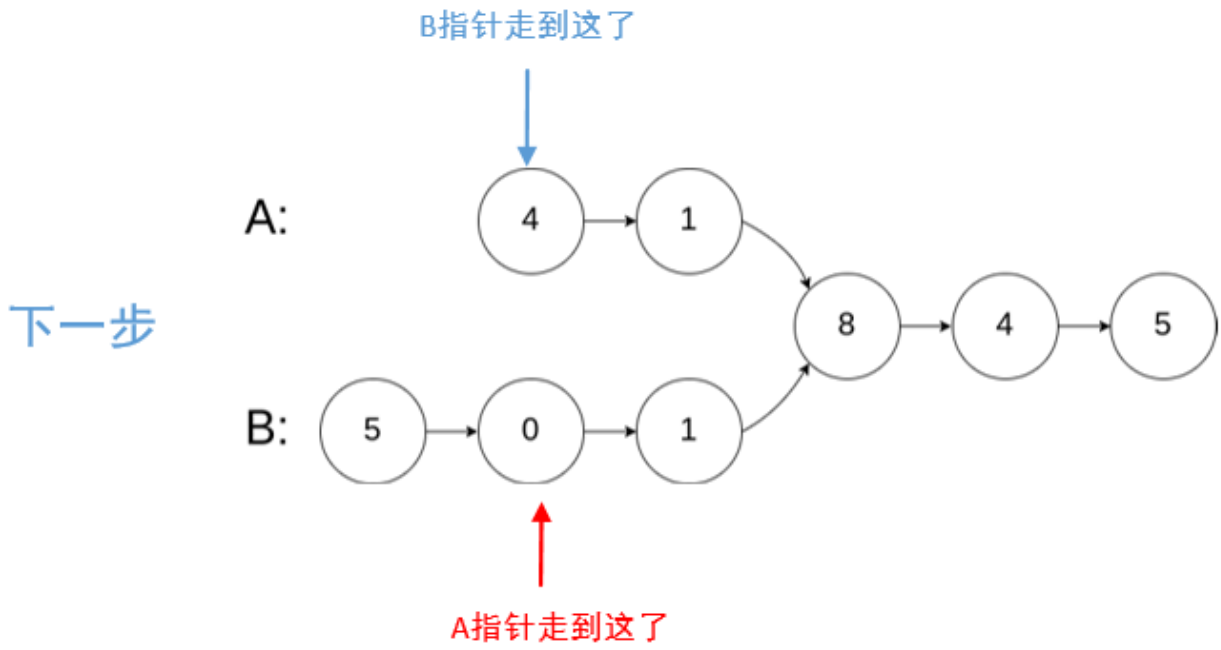
虽然他们有交点，但他们的长度不一样，所以他们完美的错开了，即使把链表都走完了也找不到相交点。

我们仔细看下上面的图，如果A指针把链表A走完了，然后再从链表B开始走到相遇点就相当于把这两个链表的所有节点都走了一遍，同理如果B指针把链表B走完了，然后再从链表A开始一直走到相遇点也相当于把这两个链表的所有节点都走完了

所以如果A指针走到链表末尾，下一步就让他从链表B开始。同理如果B指针走到链表末尾，下一步就让他从链表A开始。只要这两个链表相交最终肯定会在相交点相遇，如果不相交，最终他们都会同时走到两个链表的末尾，我们来画个图看一下







如上图所示，A指针和B指针如果一直走下去，那么他们最终会在相交点相遇，最后再来看下代码

```
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    //tempA和tempB我们可以认为是A,B两个指针
    ListNode tempA = headA;
    ListNode tempB = headB;
    while (tempA != tempB) {
        //如果指针tempA不为空，tempA就往后移一步。
        //如果指针tempA为空，就让指针tempA指向headB（注意这里是headB不是tempB）
        tempA = tempA == null ? headB : tempA.next;
        //指针tempB同上
        tempB = tempB == null ? headA : tempB.next;
    }
    //tempA要么是空，要么是两链表的交点
    return tempA;
}
```

注意：这里所说的指针并不是C语言中的指针，在这里其实他就是一个变量，千万不要搞混了。

问题分析

第一种解法应该是都容易想到的，但效率不高，一个个存储，然后再拿另一个链表的节点一个个判断。最后一种解法没有统计链表的长度，而是当一个链表访问完如果没有找到相交点，就从下一个链表继续访问，一般不太容易想到，也算是比较经典的。

往期推荐

- 461. 两两交换链表中的节点
- 459. 删除链表的倒数第N个节点的3种方式

461. 两两交换链表中的节点

原创 山大王wld 数据结构和算法 10月13日

收录于话题
#算法图文分析

95个 >



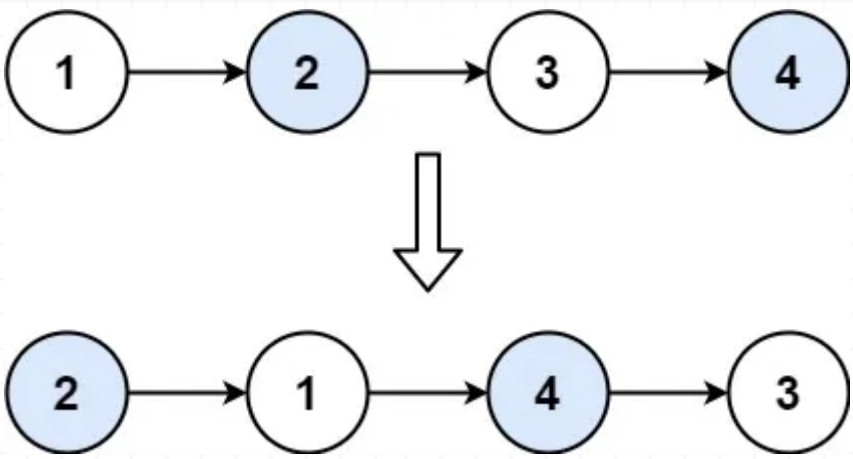
My friends are the most important thing in my life.

朋友是我生命中最重要的部分。

问题描述

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

示例 1：



> **输入：** head = [1,2,3,4]
> **输出：** [2,1,4,3]

示例 2：

> **输入**：head = []
> **输出**：[]

示例 3：

> **输入**：head = [1]
> **输出**：[1]

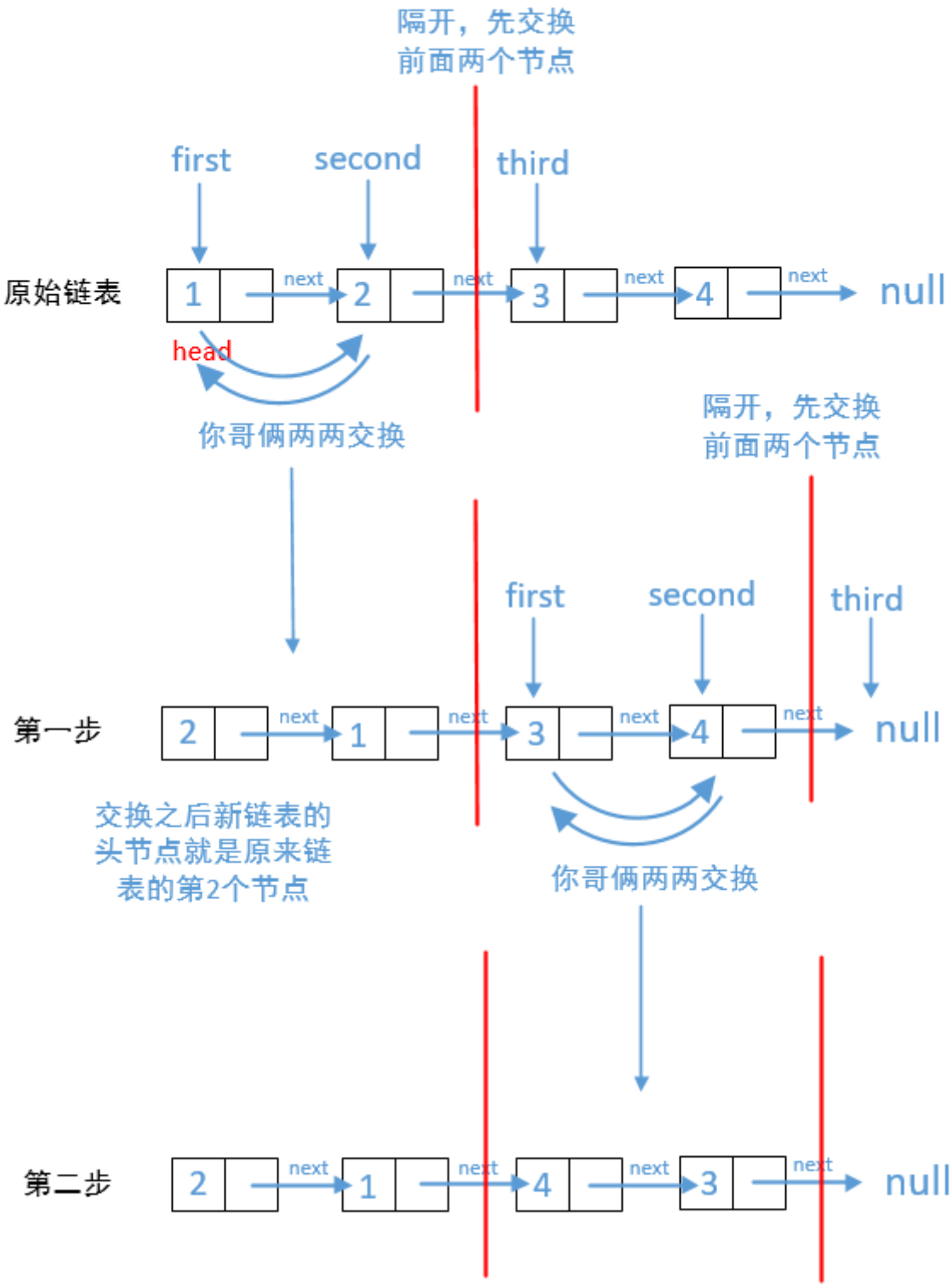
提示：

- 链表中节点的数目在范围 [0, 100] 内
- 0 <= Node.val <= 100

非递归解决

这题主要考察的是链表节点的交换，要交换链表的节点首先要搞懂链表节点的断开和连接，之前写过一篇文章，《352，数据结构-2,链表》，图文详细介绍了单向链表，双向链表，还有环形链表的断开和链接。

这道题我们可以人为的把链表分为两组，前面两个节点为一组，后面的为一组，我们首先交换第一组的两个节点，交换完之后把后面的节点再分为两组，继续这样交换下去……，直到不能交换为止。注意，如果能交换，那么原来链表的第二个节点就是我们要返回新链表的头结点。
这里就以示例为例画个图来看下



再来看下代码

```
public ListNode swapPairs(ListNode head) {
  //链表至少有2个节点才能交换，否则就不要交换
  if (head == null || head.next == null)
    return head;
  //这里的first，second，third可以参照图中的标注
  ListNode first = head;
  ListNode second;
  ListNode third;
  //这个是交换链表之后的尾结点，他的next要指向新交换的链表
  ListNode preLast = null;
  //这个只赋值一次，它是要返回的新链表的头结点
  ListNode newHead = head.next;
  //如果能交换就继续操作
  while (first != null && first.next != null) {
    //给second，third赋值
```

```
        second = first.next;
        third = second.next;
        //first和second这两个节点交换
        first.next = third;
        second.next = first;
        //这个时候second就是交换之后新链表的头结点，
        //如果preLast不为空，说明前面还有交换完成的链表
        //，要让preLast的next指向新链表的头结点
        if (preLast != null) {
          preLast.next = second;
        }
        //因为first和second交换之后，first就变成新链表
        //的尾结点了，把它保存在preLast中
        preLast = first;
        //前两个交换了，然后从第3个在继续操作
        first = third;
      }
      //返回新链表
      return newHead;
    }
```

递归解决

之前讲过《426，什么是递归，通过这篇文章，让你彻底搞懂递归》，递归有2个条件，一个是终止条件，一个是调用自己，并且要同时具备，所以这题的递归模板很容易写出来。

```
public ListNode swapPairs(ListNode head) {
    //终止条件，如果节点为空或者没有后继节点，就不能交换
    if (head == null || head.next == null)
        return head;

    //一些逻辑运算，可有可无，视情况而定

    //前面两个节点我们保留不要动，从第3个节点往后开始
    //两两交换
    ListNode newListNode = swapPairs(head.next.next);

    //一些逻辑运算，可有可无，视情况而定

    return 交换之后的链表;
}
```

假如使用递归从第3个节点往后的节点全部两两交换了，这个时候我们可以把链表分为3部分，**第一个节点**，**第二个节点**和**后面交换完成的链表**，就是1→2→3，这种形式，我们只要再把1和2位置交换了，那么这道题就完成了。来看下代码

```
public ListNode swapPairs(ListNode head) {
    //边界条件判断
    if (head == null || head.next == null)
        return head;
    //从第3个链表往后进行交换
    ListNode third = swapPairs(head.next.next);
    //从第3个链表往后都交换完了，我们只需要交换前两个链表即可，
    //这里我们把链表分为3组，分别是第1个节点，第2个节点，后面
    //的所有节点，也就是1→2→3，我们要把它变为2→1→3
    ListNode second = head.next;
    head.next = third;
    second.next = head;

    return second;
}
```

交换节点的值

还可以换种思路，因为交换链表的结点需要不停的断开和连接，比较麻烦。所以我们还可以不交换链表的节点，直接交换链表节点的值即可，这样就非常简单了

```
public ListNode swapPairs(ListNode head) {
  int first;
  ListNode temp = head;
  while (temp != null && temp.next != null) {
    //直接交换链表节点的值
    first = temp.val;
    temp.val = temp.next.val;
    temp.next.val = first;

    temp = temp.next.next;
  }
  return head;
}
```

总结

链表节点的交换要注意，在交换前两个节点之前，要把第3个节点给先保存起来，否则交换完了就找不到第3个节点了。

往期推荐

- 460. 快慢指针解环形链表 II
- 449，快慢指针解决环形链表
- 447，双指针解旋转链表
- 352，数据结构-2,链表

460. 快慢指针解环形链表 II

原创 山大王wld 数据结构和算法 10月12日

收录于话题
#算法图文分析

95个 >



Questions can't change the truth. But they give it motion.

问题不能改变事实，但是能够推动改变。

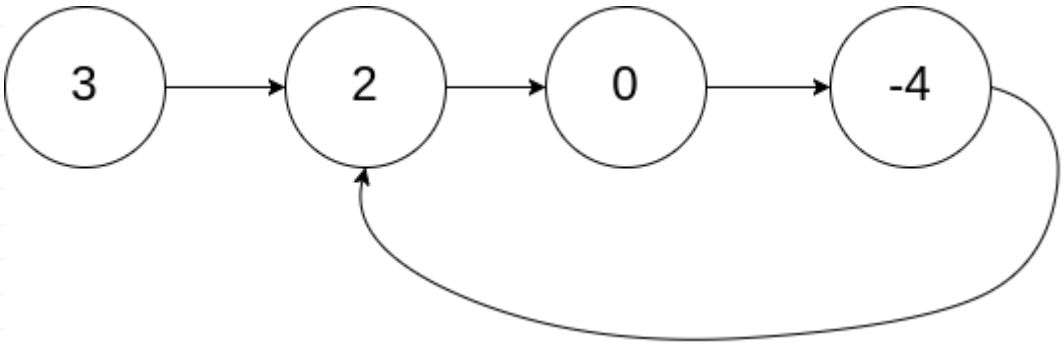
问题描述

给定一个链表，返回链表开始入环的第一个节点。如果链表无环，则返回 null。

为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 pos 是 -1，则在该链表中没有环。**注意，pos 仅仅是用于标识环的情况，并不会作为参数传递到函数中。**

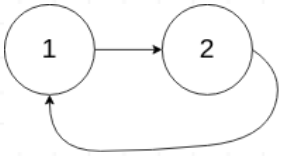
说明：不允许修改给定的链表。

示例 1：



> **输入**：head = [3,2,0,-4], pos = 1
> **输出**：返回索引为 1 的链表节点
> **解释**：链表中有一个环，其尾部连接到第二个节点。

示例 2：



> **输入**：head = [1,2], pos = 0
> **输出**：返回索引为 0 的链表节点
> **解释**：链表中有一个环，其尾部连接到第一个节点。

示例 3：



> **输入**：head = [1], pos = -1
> **输出**：返回 null
> **解释**：链表中没有环。

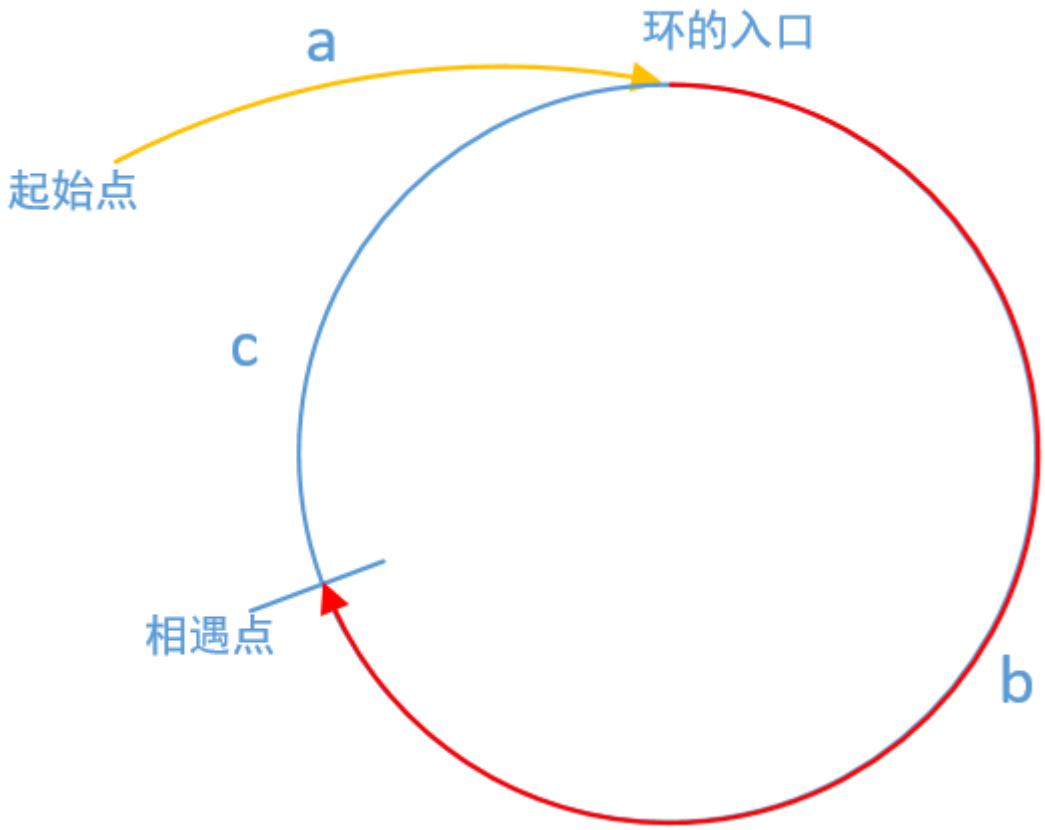
提示：

- 链表中节点的数目范围在范围 [0, 10^4] 内
- -10^5 <= Node.val <= 10^5
- pos 的值为 -1 或者链表中的一个有效索引

双指针解决

在前面讲过449，快慢指针解决环形链表，判断一个链表是否是环形链表，而这题比第449题稍微麻烦一点，就是如果确定链表有环，还要找出环的入口。如果一个链表没有环，就不存在环的入口。这里主要来看一下，如果一个链表有环怎么找到他的入口。

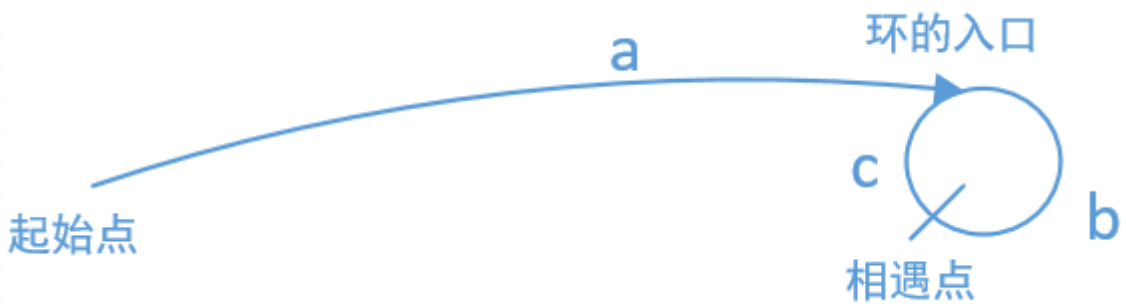
对于链表组成的环一般有两种，一种是O型，一种是6型，其实原理都一样，这里主要看一下6字型的链表，他会有两种情况，一种是环很大，如下图所示。



根据第449题的分析，通过快慢指针可以判断一个链表是否有环。如果有环，那么**快指针走过的路径就是图中a+b+c+b，慢指针走过的路径就是图中a+b**，因为在相同的时间内，快指针走过的路径是慢指针的2倍，所以这里有a+b+c+b=2*(a+b)，整理得到a=c，也就是说图中a的路径长度和c的路径长度是一样的。

在相遇的时候再使用两个指针，一个从链表起始点开始，一个从相遇点开始，每次他们都走一步，直到再次相遇，那么这个相遇点就是环的入口。

还有一种情况就是环很小，如下图所示



这种情况下当快慢指针在环上相遇的时候，快指针有可能在环上转了好几圈，我们假设相遇的时候快指针已经在环上转了n圈。

那么相遇的时候**快指针走过的路径就是a+b+(b+c)*n**，（b+c其实就是环的长度），**慢指针走过的路径是a+b**。由于相同时间内快指针走过的路径是慢指针的2倍。

所以有a+b+(b+c)*n=2*(a+b)，整理得到a+b=n*(b+c)，也就是说a+b等于n个环的长度。

我们还可以使用两个指针，一个从链表的起点开始，一个从相遇点开始，那么就会出现一种情况就是，一个指针在路径a上走，一个指针一直在环上转圈，最终会走到第一种情况。

所以无论哪种情况我们都可以使用第一种方式解决，代码如下

```
public ListNode detectCycle(ListNode head) {
  ListNode slow = head, fast = head;
  while (fast != null && fast.next != null) {
    //快慢指针，快指针每次走两步，慢指针每次走一步
    fast = fast.next.next;
    slow = slow.next;
    //先判断是否有环，
    if (slow == fast) {
      //确定有环之后才能找环的入口
      while (head != slow) {
        //两指针，一个从头结点开始，
        //一个从相遇点开始每次走一步，直到
        //再次相遇为止
        head = head.next;
        slow = slow.next;
      }
      return slow;
    }
  }
  return null;
}
```

通过集合set解决

上面解法可能不是太好理解，这里再来看一种比较好理解的，就把结点一个个全部存放到集合set中，在存放的时候如果出现了重复的结点，说明这个链表是有环的，并且首次重复的这个节点就是环的入口

```
public ListNode detectCycle(ListNode head) {
  Set<ListNode> set = new HashSet<>();
  while (head != null) {
    //如果重复出现说明有环
    if (!set.add(head))
      return head;
    //否则就把当前节点加入到集合中
    head = head.next;
  }
  return null;
}
```

set的add方法表示往集合中添加元素，添加的时候如果没有重复的就会返回true，如果有重复的就会返回false。

总结

这道题最容易想到的应该是最后一种解法，比较简单，也很容易理解。但这道题有个要求，就是不能修改链表的结构，如果没这要求的话，还可以参照449，快慢指针解决环形链表的最后一种解法，就是把链表节点一个个删除。

往期推荐

- 449，快慢指针解决环形链表
- 447，双指针解旋转链表
- 432，剑指 Offer-反转链表的3种方式
- 429，剑指 Offer-删除链表的节点

459. 删除链表的倒数第N个节点的3种方式

原创 山大王wld 数据结构和算法 10月10日

收录于话题
#算法图文分析

95个 >



Why can't we savor the precious moments?

我们为什么不慢慢享受美好时光?

问题描述

给定一个链表，删除链表的倒数第 *n* 个节点，并且返回链表的头结点。

示例：

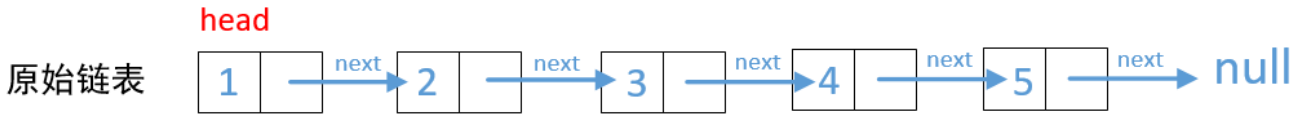
> 给定一个链表: 1->2->3->4->5, 和 n = 2.
>
> 当删除了倒数第二个节点后，链表变为
> 1->2->3->5.

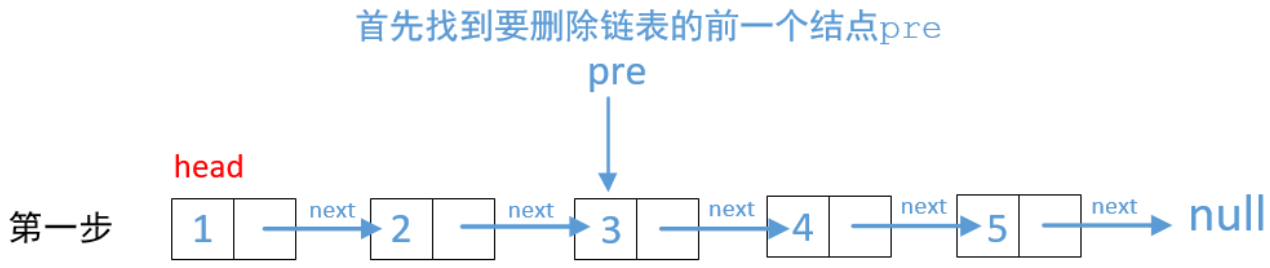
说明：

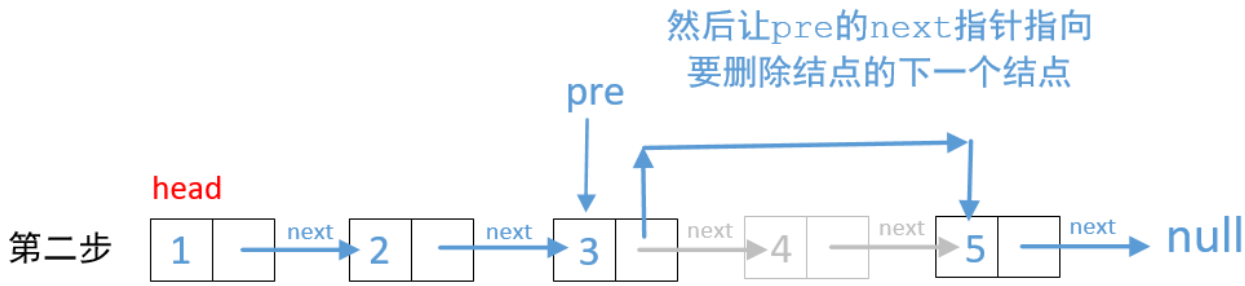
给定的 *n* 保证是有效的。

非递归解决

这题让删除链表的倒数第n个节点，首先最容易想到的就是先求出链表的长度length，然后就可以找到要删除链表的前一个结点，让他的前一个结点指向要删除结点的下一个结点即可，这里就以示例为例画个图看一下







再来看下代码

```java
public ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode pre = head;
    int last = length(head) - n;
    //如果last等于0表示删除的是头结点
    if (last == 0)
        return head.next;
    //这里首先要找到要删除链表的前一个结点
    for (int i = 0; i < last - 1; i++) {
        pre = pre.next;
    }
    //然后让前一个结点的next指向要删除节点的next
    pre.next = pre.next.next;
    return head;
}

//求链表的长度
private int length(ListNode head) {
    int len = 0;
    while (head != null) {
        len++;
        head = head.next;
    }
    return len;
}
```

双指针求解

上面是先计算链表的长度，其实不计算链表的长度也是可以，我们可以使用两个指针，一个指针fast先走n步，然后另一个指针slow从头结点开始，找到要删除结点的前一个结点，这样就可以完成结点的删除了。

```
public ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode fast = head;
    ListNode slow = head;
    //fast移n步，
    for (int i = 0; i < n; i++) {
        fast = fast.next;
    }
    //如果fast为空，表示删除的是头结点
    if (fast == null)
        return head.next;

    while (fast.next != null) {
        fast = fast.next;
        slow = slow.next;
    }
    //这里最终slow不是倒数第n个节点，他是倒数第n+1个节点，
    //他的下一个结点是倒数第n个节点,所以删除的是他的下一个结点
    slow.next = slow.next.next;
    return head;
}
```

递归解决

我们知道获取链表的长度除了上面介绍的一种方式以外，还可以写成递归的方式，比如

```
//求链表的长度
private int length(ListNode head) {
    if (head == null)
        return 0;
    return length(head.next) + 1;
}
```

上面计算链表长度的递归其实可以把它看做是从后往前计算，当计算的长度是n的时候就表示遍历到了倒数第n个节点了，这里只要求出倒数第n+1个节点，问题就迎刃而解了，来看下代码

```
public ListNode removeNthFromEnd(ListNode head, int n) {
    int pos = length(head, n);
    // 说明删除的是头节点
    if (pos == n)
        return head.next;
    return head;
}

// 获取节点所在位置，逆序
public int length(ListNode node, int n) {
    if (node == null)
        return 0;
    int pos = length(node.next, n) + 1;
    //获取要删除链表的前一个结点，就可以完成链表的删除
    if (pos == n + 1)
        node.next = node.next.next;
    return pos;
}
```

总结

要删除链表的倒数第n个节点，首先要找到链表的倒数第n+1个节点，这样就可以完成链表的删除了，关于怎么找，方式有多种。但要注意一点如果删除的是头结点的话，那么就不存在倒数第n+1个节点，所以这个要注意一下。

往期推荐

- 449，快慢指针解决环形链表
- 447，双指针解旋转链表
- 431，剑指 Offer-链表中倒数第k个节点
- 429，剑指 Offer-删除链表的节点

432，剑指 Offer-反转链表的3种方式

原创 山大王wld 数据结构和算法 8月17日

收录于话题
#剑指offer

27个 >



Success is stumbling from failure to failure with no loss of enthusiasm.

成功是在失败中摸索，同时不失去热情。

问题描述

定义一个函数，输入一个链表的头节点，反转该链表并输出反转后链表的头节点。

示例：

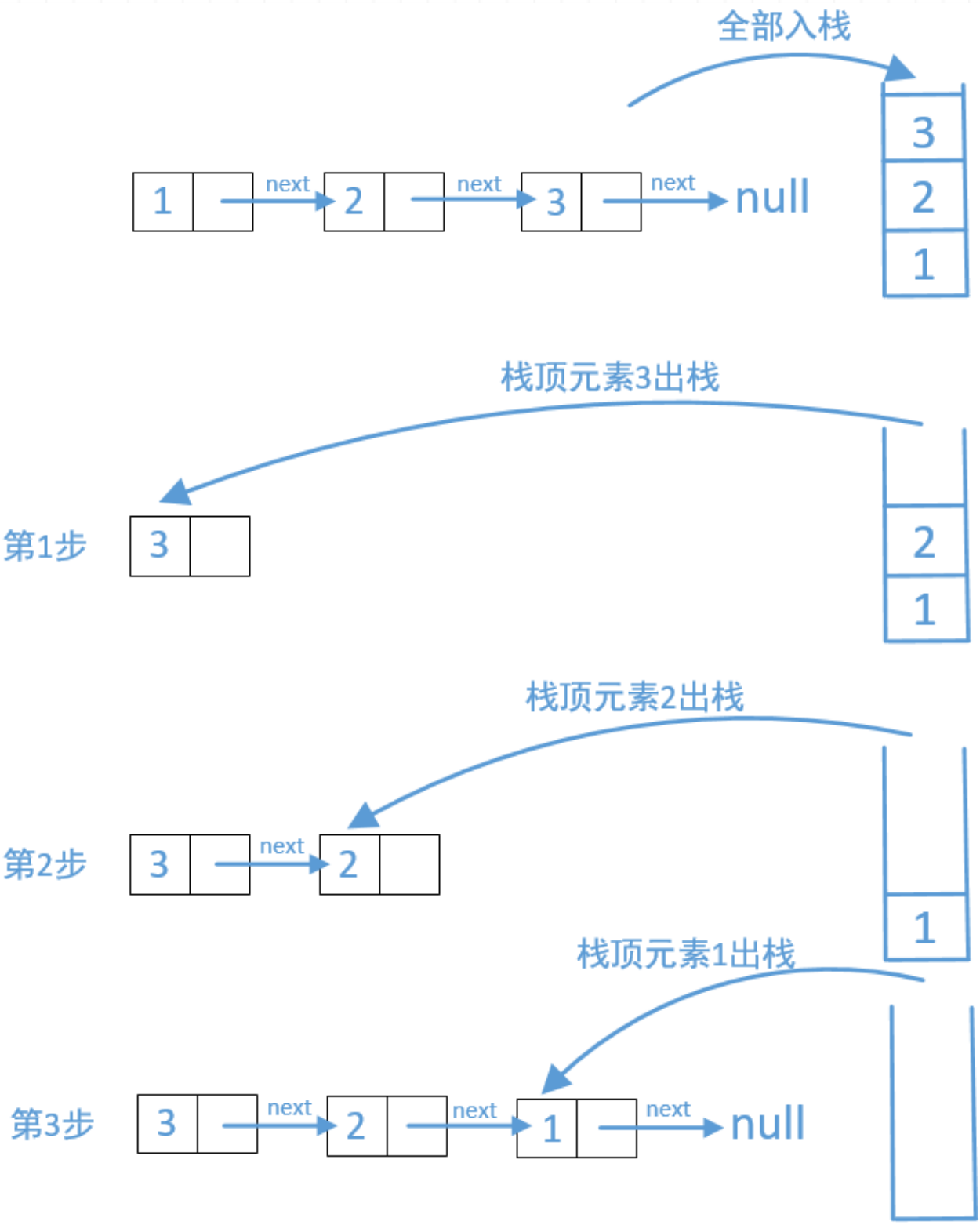
```
输入：1->2->3->4->5->NULL
输出：5->4->3->2->1->NULL
```

限制：

`0 <= 节点个数 <= 5000`

使用栈解决

链表的反转是老生常谈的一个问题了，同时也是面试中常考的一道题。最简单的一种方式就是使用**栈**，因为**栈是先进后出**的。实现原理就是把链表节点一个个入栈，当全部入栈完之后再一个个出栈，出栈的时候在把出栈的结点串成一个新的链表。原理如下



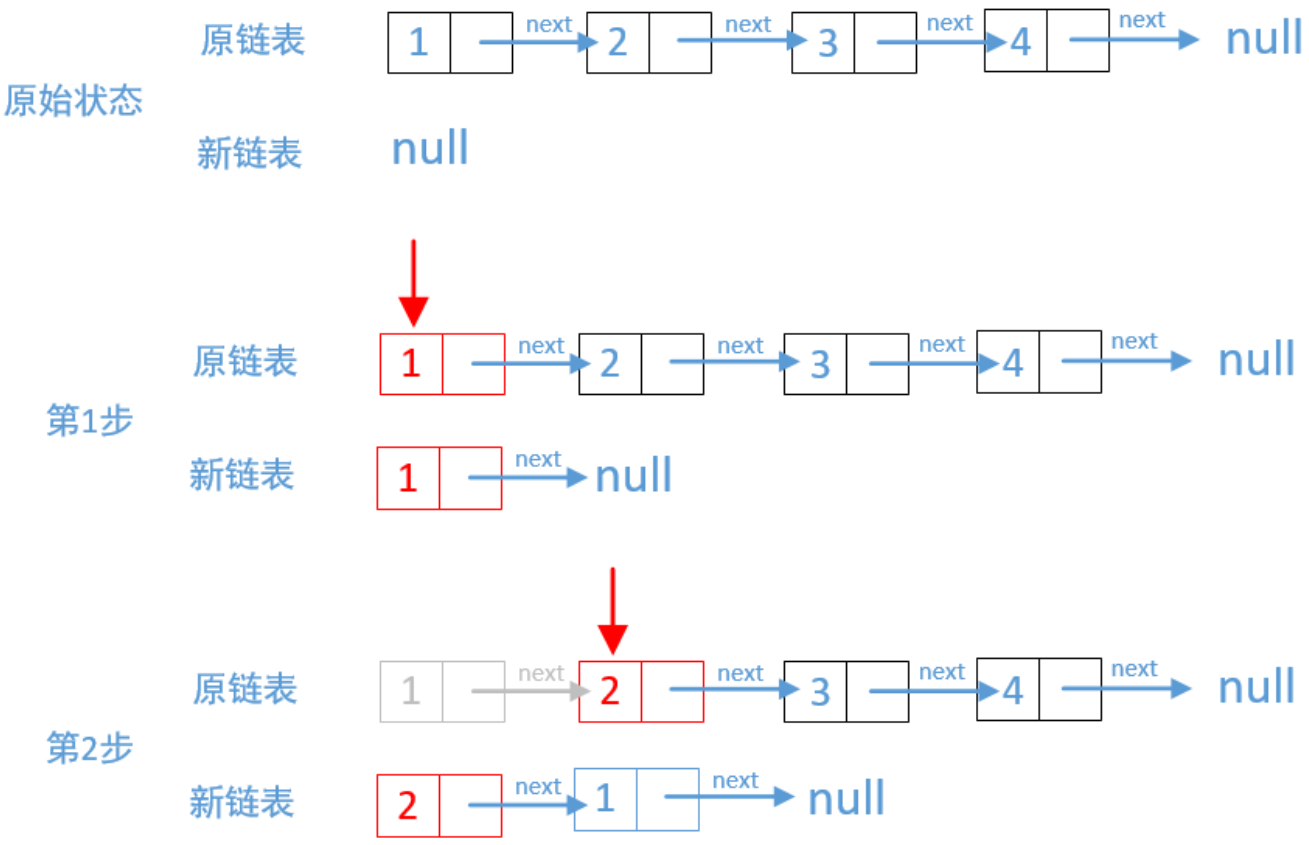
代码比较简单，来看下

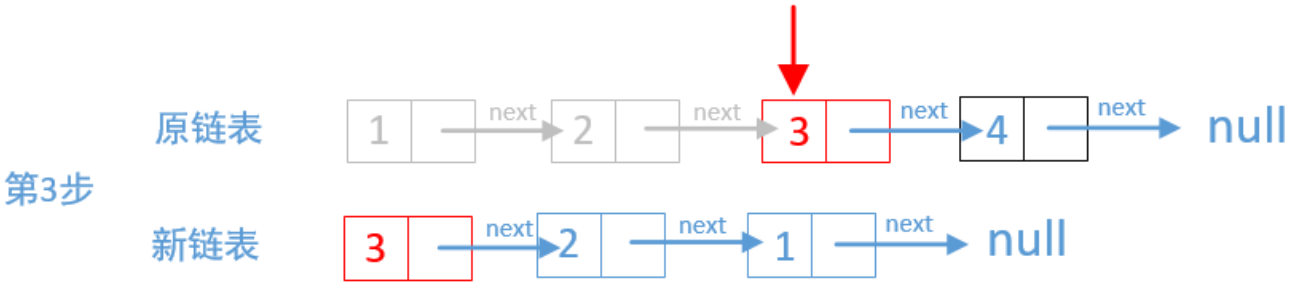
```
public ListNode reverseList(ListNode head) {
  Stack<ListNode> stack = new Stack<>();
  //把链表节点全部摘掉放到栈中
  while (head != null) {
    stack.push(head);
    head = head.next;
  }
```

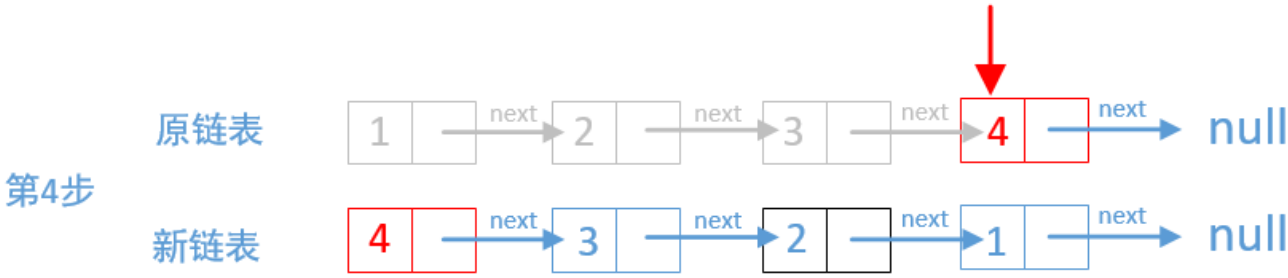
```
8       if (stack.isEmpty())
9           return null;
10      ListNode node = stack.pop();
11      ListNode dummy = node;
12      //栈中的结点全部出栈，然后重新连成一个新的链表
13      while (!stack.isEmpty()) {
14          ListNode tempNode = stack.pop();
15          node.next = tempNode;
16          node = node.next;
17      }
18      //最后一个结点就是反转前的头结点，一定要让他的next
19      //等于空，否则会构成环
20      node.next = null;
21      return dummy;
22  }
```

双链表求解

双链表求解是把原链表的结点一个个摘掉，每次摘掉的链表都让他成为新的链表的头结点，然后更新新链表。下面以链表1→2→3→4为例画个图来看下。







他每次访问的原链表节点都会成为新链表的头结点，最后再来看下代码

```java
public ListNode reverseList(ListNode head) {
  //新链表
  ListNode newHead = null;
  while (head != null) {
    //先保存访问的节点的下一个节点，保存起来
    //留着下一步访问的
    ListNode temp = head.next;
    //每次访问的原链表节点都会成为新链表的头结点，
    //其实就是把新链表挂到访问的原链表节点的
    //后面就行了
    head.next = newHead;
    //更新新链表
    newHead = head;
    //重新赋值，继续访问
    head = temp;
  }
  //返回新链表
  return newHead;
}
```

递归解决

我们再来回顾一下递归的模板，终止条件，递归调用，逻辑处理。

```java
public ListNode reverseList(参数0) {
  if (终止条件)
    return;

  逻辑处理（可能有，也可能没有，具体问题具体分析）

  //递归调用
  ListNode reverse = reverseList(参数1);

  逻辑处理（可能有，也可能没有，具体问题具体分析）
}
```

终止条件就是链表为空，或者是链表没有尾结点的时候，直接返回

```java
if (head == null || head.next == null)
```

```
    return head;
```

递归调用是要从当前节点的下一个结点开始递归。逻辑处理这块是要把当前节点挂到递归之后的链表的末尾，看下代码

```
public ListNode reverseList(ListNode head) {
  //终止条件
  if (head == null || head.next == null)
    return head;
  //保存当前节点的下一个结点
  ListNode next = head.next;
  //从当前节点的下一个结点开始递归调用
  ListNode reverse = reverseList(next);
  //reverse是反转之后的链表，因为函数reverseList
  // 表示的是对链表的反转，所以反转完之后next肯定
  // 是链表reverse的尾结点，然后我们再把当前节点
  //head挂到next节点的后面就完成了链表的反转。
  next.next = head;
  //这里head相当于变成了尾结点，尾结点都是为空的，
  //否则会构成环
  head.next = null;
  return reverse;
}
```

因为递归调用之后head.next节点就会成为reverse节点的尾结点，我们可以直接让head.next.next = head;，这样代码会更简洁一些，看下代码

```
public ListNode reverseList(ListNode head) {
  if (head == null || head.next == null)
    return head;
  ListNode reverse = reverseList(head.next);
  head.next.next = head;
  head.next = null;
  return reverse;
}
```

这种递归往下传递的时候基本上没有逻辑处理，当往回反弹的时候才开始处理，也就是从链表的尾端往前开始处理的。我们还可以再来改一下，在链表递归的时候从前往后处理，处理完之后直接返回递归的结果，这就是所谓的**尾递归**，这种运行效率要比上一种好很多

```
public ListNode reverseList(ListNode head) {
  return reverseListInt(head, null);
}

private ListNode reverseListInt(ListNode head, ListNode newHead) {
  if (head == null)
    return newHead;
  ListNode next = head.next;
  head.next = newHead;
  return reverseListInt(next, head);
}
```

尾递归虽然也会不停的压栈，但由于最后返回的是递归函数的值，所以在返回的时候都会一次性出栈，不会一个个出栈这么慢。但如果我们再来改一下，像下面代码这样又会一个个出栈了

```
public ListNode reverseList(ListNode head) {
  return reverseListInt(head, null);
}

private ListNode reverseListInt(ListNode head, ListNode newHead) {
  if (head == null)
    return newHead;
```

```
8       ListNode next = head.next;
9       head.next = newHead;
10      ListNode node = reverseListInt(next, head);
11      return node;
12  }
```

总结

链表反转使用栈虽然也能实现，但一般不是很推荐，下面两种实现方式会好一些。使用栈能实现链表的反转，那么使用队列呢，如果使用双端队列也是可以的，从一端全部入队，然后再从这一端全部出队，说了半天这不还是和栈一样吗……

往期推荐

- 431，剑指 Offer-链表中倒数第k个节点
- 429，剑指 Offer-删除链表的节点
- 410，剑指 Offer-从尾到头打印链表
- 352，数据结构-2,链表

431，剑指 Offer-链表中倒数第k个节点

原创 山大王wld 数据结构和算法 8月14日

收录于话题
#剑指offer

27个 >



Every passing minute is another chance to turn it all around.

过去的每一分钟都是改变现状的一次机会。

问题描述

输入一个链表，输出该链表中倒数第k个节点。为了符合大多数人的习惯，本题从1开始计数，即链表的尾节点是倒数第1个节点。

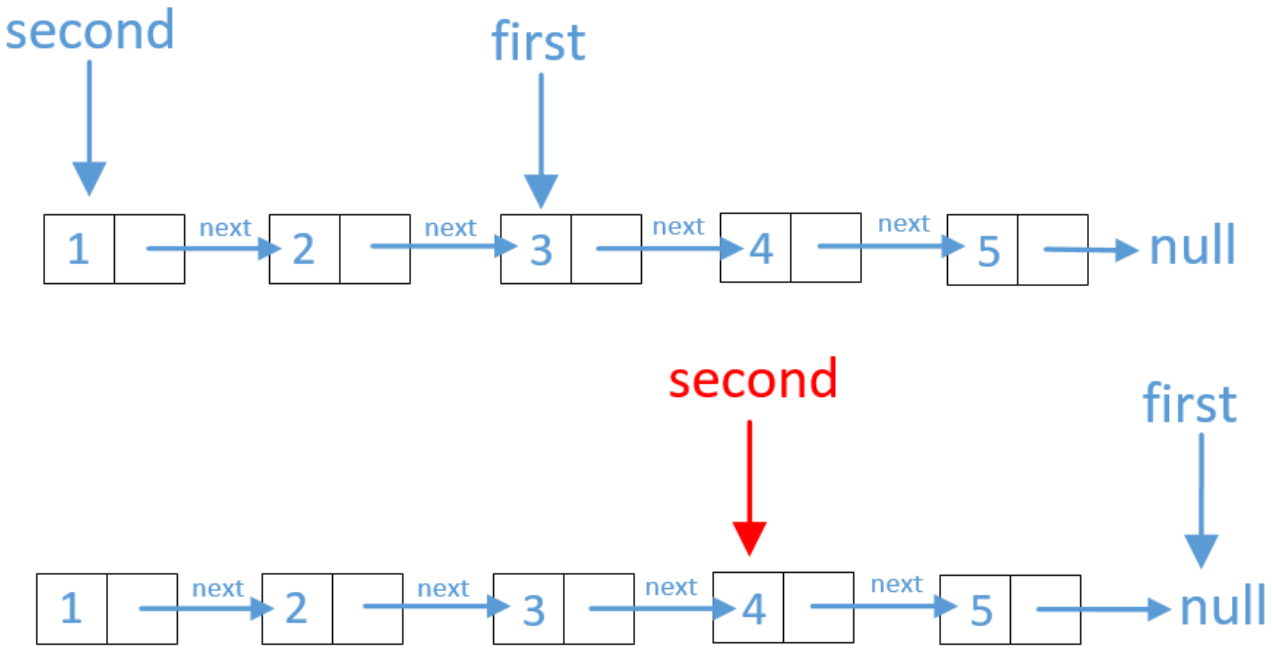
例如，一个链表有6个节点，从头节点开始，它们的值依次是1、2、3、4、5、6。这个链表的倒数第3个节点是值为4的节点。

示例：

> 给定一个链表: 1->2->3->4->5, 和 k = 2.
>
> 返回链表 4->5.

双指针求解

这题要求链表的倒数第k个节点，最简单的方式就是使用两个指针，第一个指针先移动k步，然后第二个指针再从头开始，这个时候这两个指针同时移动，当第一个指针到链表的末尾的时候，返回第二个指针即可。




```java
public ListNode getKthFromEnd(ListNode head, int k) {
    ListNode first = head;
    ListNode second = head;
    //第一个指针先走k步
    while (k-- > 0) {
        first = first.next;
    }
    //然后两个指针在同时前进
    while (first != null) {
        first = first.next;
        second = second.next;
    }
    return second;
}
```

使用栈解决

这题要求的是返回后面的k个节点，我们只要把原链表的结点全部压栈，然后再把栈中最上面的k个节点出栈，出栈的结点重新串成一个新的链表即可，原理也比较简单，直接看下代码。

```java
public ListNode getKthFromEnd(ListNode head, int k) {
    Stack<ListNode> stack = new Stack<>();
    //链表节点压栈
    while (head != null) {
        stack.push(head);
        head = head.next;
    }
    //在出栈串成新的链表
    ListNode firstNode = stack.pop();
    while (--k > 0) {
        ListNode temp = stack.pop();
```

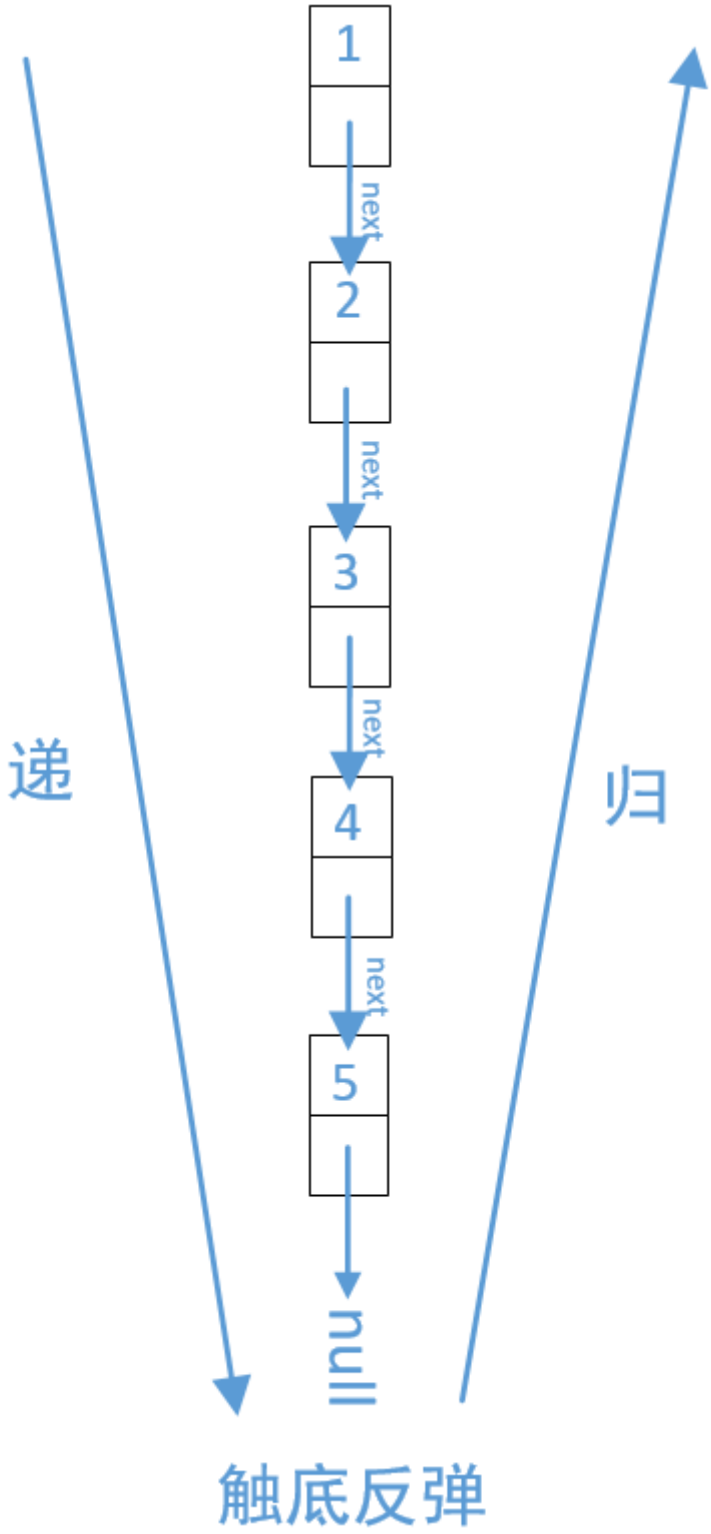
```
        temp.next = firstNode;
        firstNode = temp;
      }
      return firstNode;
    }
```

递归求解

我们之前讲过链表的逆序打印410，剑指 Offer-从尾到头打印链表，其中有这样一段代码

```
1   public void reversePrint(ListNode head) {
2     if (head == null)
3       return;
4     reversePrint(head.next);
5     System.out.println(head.val);
6   }
```

这段代码其实很简单，我们要理解他就要弄懂递归的原理，递归分为两个过程，**递**和**归**，看一下下面的图就知道了，先往下传递，当遇到终止条件的时候开始往回走。



前面也刚讲过递归的原理[426，什么是递归，通过这篇文章，让你彻底搞懂递归]，这题如果使用递归的话，我们先来看一下递归的模板

```
public ListNode getKthFromEnd(ListNode head, int k) {
  //终止条件
  if (head == null)
    return head;
  //递归调用
  ListNode node = getKthFromEnd(head.next, k);
  //逻辑处理
  ……
}
```

终止条件很明显就是当节点head为空的时候，就没法递归了，这里主要看的是逻辑处理部分，当递归往下传递到最底端的时候，就会触底反弹往回走，在往回走的过程中记录

下走过的节点，当达到k的时候，说明到达的那个节点就是倒数第k个节点，直接返回即可，如果没有达到k，就返回空，搞懂了上面的过程，代码就很容易写了

```
//全局变量，记录递归往回走的时候访问的结点数量
int size;

public ListNode getKthFromEnd(ListNode head, int k) {
  //边界条件判断
  if (head == null)
    return head;
  ListNode node = getKthFromEnd(head.next, k);
  ++size;
  //从后面数结点数小于k，返回空
  if (size < k) {
    return null;
  } else if (size == k) {
    //从后面数访问结点等于k，直接返回传递的结点k即可
    return head;
  } else {
    //从后面数访问的结点大于k，说明我们已经找到了，
    //直接返回node即可
    return node;
  }
}
```

上面代码在仔细一看，当size小于k的时候node节点就是空，所以我们可以把size大于k和小于k合并为一个，这样代码会更简洁一些

```
int size;

public ListNode getKthFromEnd(ListNode head, int k) {
  if (head == null)
    return head;
  ListNode node = getKthFromEnd(head.next, k);
  if (++size == k)
    return head;
  return node;
}
```

总结

这题最简单的估计就是第一种使用双指针的方式了，关于链表的知识也可以看下前面讲的352，数据结构-2,链表

往期推荐

- 429，剑指 Offer-删除链表的节点
- 410，剑指 Offer-从尾到头打印链表
- 386，链表中的下一个更大节点
- 352，数据结构-2,链表

429，剑指 Offer-删除链表的节点

原创 山大王wld 数据结构和算法 8月11日

收录于话题
#剑指offer

27个 >



The hardships that I encountered in the past will help me succeed in the future.

我过去遇到的困难会在未来帮助我成功。

问题描述

给定单向链表的头指针和一个要删除的节点的值，定义一个函数删除该节点。

返回删除后的链表的头节点。

注意： 此题对比原题有改动

示例 1：

```
输入: head = [4,5,1,9], val = 5
输出: [4,1,9]

解释: 给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.
```

示例 2：

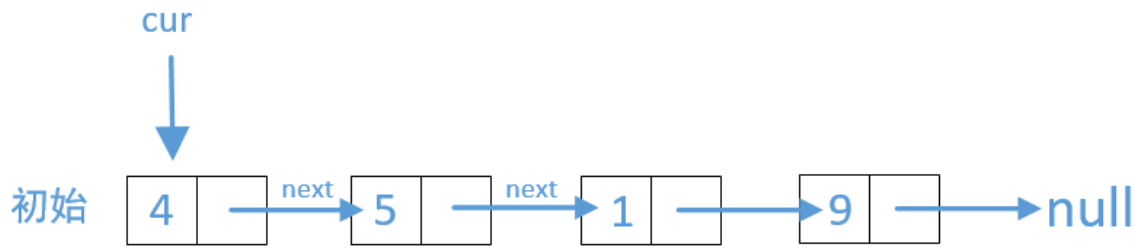
> 输入: head = [4,5,1,9], val = 1
> 输出: [4,5,9]
>
> 解释: 给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9.

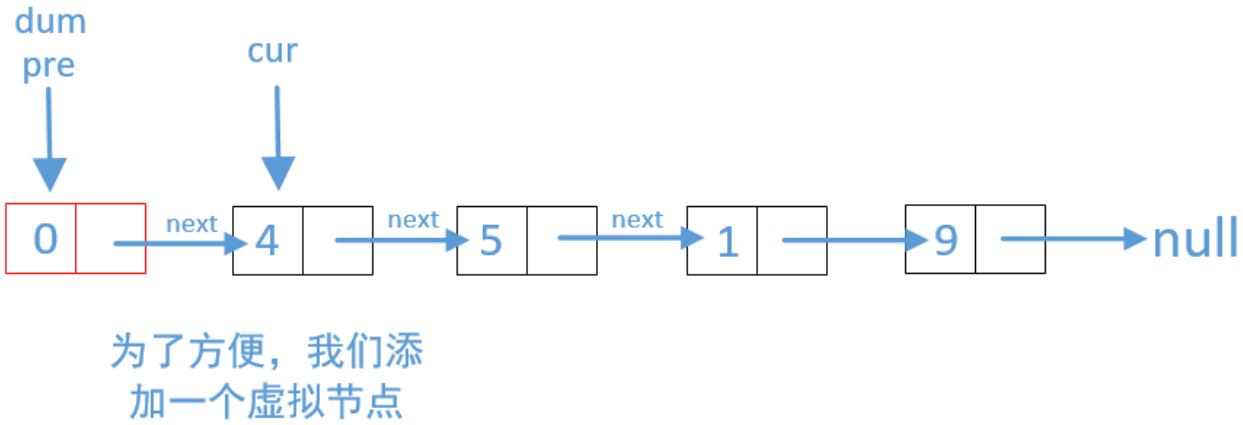
说明：

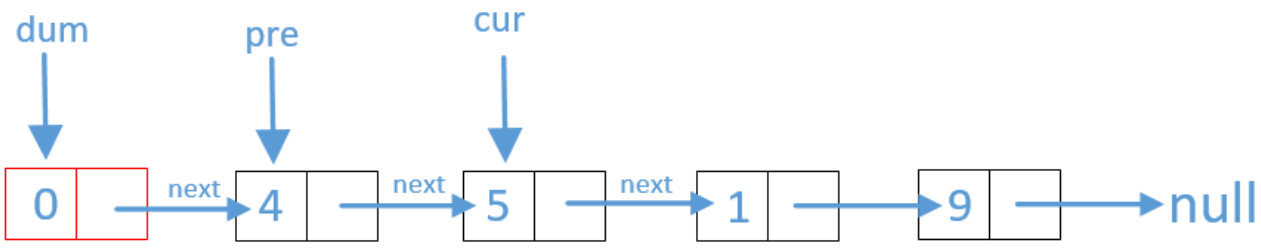
- 题目保证链表中节点的值互不相同
- 若使用 C 或 C++ 语言，你不需要 `free` 或 `delete` 被删除的节点

双指针求解

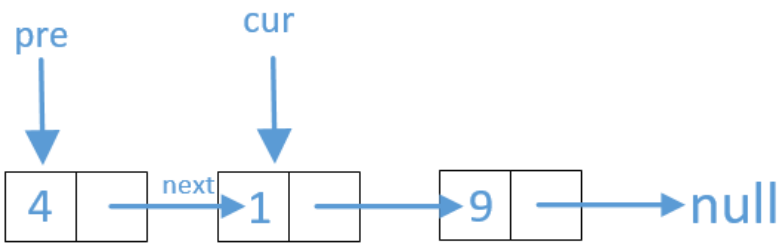
题中说了链表中的节点值互不相同，也就是说最多只能删除一个。最简单的一种方式就是双指针求解，我们使用两个指针一个指向当前的节点，一个指向当前的上一个节点，以示例1为例画个图看下







当cur等于5的时候，说明我们找到了，
然后让cur的上一个结点指向cur的下一
个结点，也就是pre.next=cur.next



返回dum的下一个结点

```java
public ListNode deleteNode(ListNode head, int val) {
    //初始化一个虚拟节点
    ListNode dummy = new ListNode(0);
    //让虚拟节点指向头结点
    dummy.next = head;
    ListNode cur = head;
    ListNode pre = dummy;
    while (cur != null) {
        if (cur.val == val) {
            //如果找到要删除的结点，直接把他删除
            pre.next = cur.next;
            break;
        }
        //如果没找到，pre指针和cur指针都同时往后移一步
        pre = cur;
        cur = cur.next;
    }
    //最后返回虚拟节点的下一个结点即可
    return dummy.next;
}
```

上面代码添加虚拟节点是为了方便操作，当然不添加虚拟节点也是可以的，像下面这样

```java
public ListNode deleteNode(ListNode head, int val) {
    //边界条件判断
    if (head == null)
        return head;
    //如果要删除的是头结点，直接返回头结点的下一个结点即可
    if (head.val == val)
        return head.next;
    ListNode cur = head;
    //找到要删除结点的上一个结点
    while (cur.next != null && cur.next.val != val) {
        cur = cur.next;
    }
    //删除结点
    cur.next = cur.next.next;
    return head;
}
```

递归解决

前面刚讲过递归，426，什么是递归，通过这篇文章，让你彻底搞懂递归，提到递归的模板，我们看下

```
public void recursion(参数0) {
    if (终止条件) {
        return;
    }

    可能有一些逻辑运算
    recursion(参数1)
    可能有一些逻辑运算
    recursion(参数2)
            ……
    recursion(参数n)
    可能有一些逻辑运算
}
```

我们来定义一个递归的函数
public ListNode deleteNode(ListNode head, int val)，
他表示的是删除链表中值等于val的结点，那么递归的终止条件就是当head等于空的时候，我们直接返回head，因为一个空的链表我们是没法删除的，也就是下面这样

```
if (head == null)
    return head;
```

如果head结点不等于空，并且head结点的值等于val，我们直接返回head结点的下一个结点

```
if (head.val == val)
    return head.next;
```

否则也就是说头结点是删不掉的，我们就递归调用，从头结点的下一个开始继续上面的操作，直到删除为止。

```
head.next = deleteNode(head.next, val);
return head;
```

完整代码如下

```
1   public ListNode deleteNode(ListNode head, int val) {
2     if (head == null)
3       return head;
4     if (head.val == val)
5       return head.next;
6     head.next = deleteNode(head.next, val);
7     return head;
8   }
```

总结

这题递归和非递归两种方式都很容易解决，对应链表的操作也可以看下前面的352，数据结构-2,链表

往期推荐

- 426，什么是递归，通过这篇文章，让你彻底搞懂递归
- 422，剑指 Offer-使用DFS和BFS解机器人的运动范围
- 420，剑指 Offer-回溯算法解矩阵中的路径
- 416，剑指 Offer-用两个栈实现队列

410，剑指 Offer-从尾到头打印链表

原创 山大王wld 数据结构和算法 7月22日

收录于话题
#剑指offer

27个 >



Every strike brings me closer to the next home run.

每一次挥棒落空都让我更接近下一个全垒打。

问题描述

输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）。

示例 1：

```
输入：head = [1,3,2]
输出：[2,3,1]
```

限制：

- 0 <= 链表长度 <= 10000

链表节点类

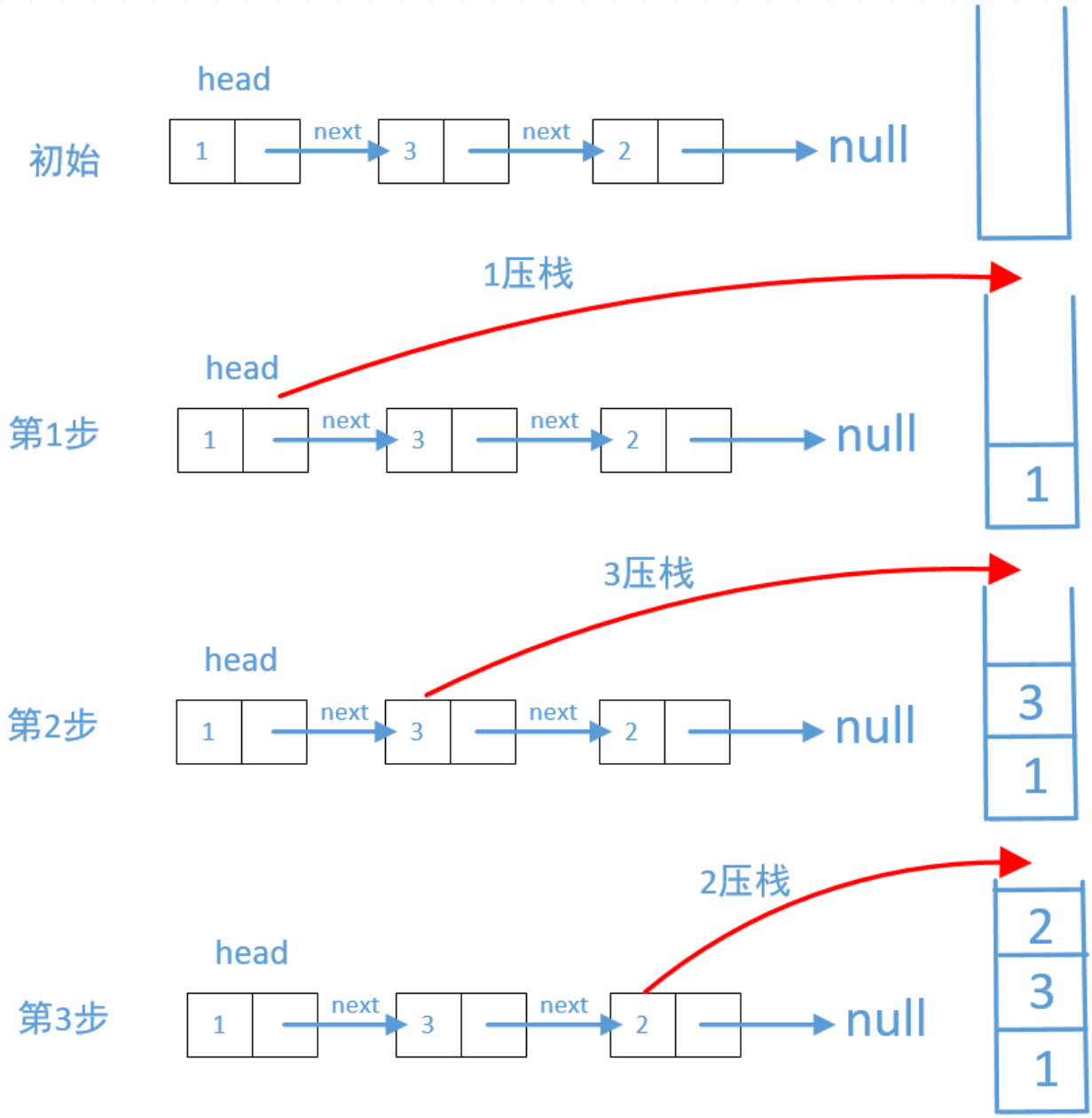
```
public class ListNode {
    int val;
    ListNode next;
```

```
4
5       ListNode(int x) {
6           val = x;
7       }
8   }
```

使用栈来解决

从尾到头打印链表，首先这个链表是单向的，如果是双向的，直接从后往前打印就行了，这里链表不是单向的。

这里最容易想到的一种方式就是把链表的节点全部压栈，因为栈是先进后出的一种数据结构，全部压栈之后再一个个出栈即可，



压栈完之后再一个个出栈

```
1   public int[] reversePrint(ListNode head) {
2       Stack<ListNode> stack = new Stack<>();
3       ListNode temp = head;
4       while (temp != null) {
5           stack.push(temp);
```

```
      temp = temp.next;
    }
    int size = stack.size();
    int[] res = new int[size];
    for (int i = 0; i < size; i++) {
      res[i] = stack.pop().val;
    }
    return res;
  }
```

递归方式解决

我们知道如果逆序打印一个链表使用递归的方式还是很简单的，像下面这样

```
public void reversePrint(ListNode head) {
  if (head == null)
    return;
  reversePrint(head.next);
  System.out.println(head.val);
}
```

但这里实际上是要我们返回逆序打印的结果，也就是返回一个数组，所以我们可以改一下

```
public int[] reversePrint(ListNode head) {
  int cout = length(head);
  int[] res = new int[cout];
  reversePrintHelper(head, res, cout - 1);
  return res;
}

//计算链表的长度
public int length(ListNode head) {
  int cout = 0;
  ListNode dummy = head;
  while (dummy != null) {
    cout++;
    dummy = dummy.next;
  }
  return cout;
}

public void reversePrintHelper(ListNode head, int[] res, int index) {
  if (head == null)
    return;
  reversePrintHelper(head.next, res, index - 1);
  res[index] = head.val;
}
```

先反转再打印

关于链表的反转其实解法也比较多，这里先列出简单的两种，一个是递归的，一个是非递归的。

递归的方式

```
public ListNode reverseList(ListNode head) {
  if (head == null || head.next == null)
    return head;
  ListNode tempList = reverseList(head.next);
  head.next.next = head;
  head.next = null;
```

```
    return tempList;
  }
```

非递归的方式

```
public ListNode reverseList(ListNode head) {
    ListNode pre = null;
    while (head != null) {
        ListNode next = head.next;
        head.next = pre;
        pre = head;
        head = next;
    }
    return pre;
}
```

链表反转之后在打印就方便多了，这里就不在写了。

总结

关于链表的逆序打印应该算是一道比较简单的题了，使用栈，递归，反转都能轻松实现。

往期推荐

- 408，剑指 Offer-替换空格
- 406，剑指 Offer-二维数组中的查找
- 404，剑指 Offer-数组中重复的数字

386，链表中的下一个更大节点

原创 山大王wld 数据结构和算法 6月17日

收录于话题
#算法图文分析

96个 >



Zeal without knowledge is fire without light.

没有知识的热忱犹如火之无光。

问题描述

给一个链表，返回每个节点下一个比他大的值，如果不存在就返回0 。

示例 1：

输入：[2,1,5]
输出：[5,5,0]
解释：2的下一个比他大的是5，1的下一个比他大的也是5，5后面没有比他大的值，所以是0。

示例 2：

> 输入：[2,7,4,3,5]
> 输出：[7,0,5,5,0]

示例 3：

> 输入：[1,7,5,1,9,2,5,1]
> 输出：[7,9,9,9,0,5,0,0]

问题分析

这题和382，每日温度的5种解题思路很像，不过有点不同的是第382题求的是下一个比他大的数的位置和当前数位置的差值，而这里求的是下一个比他大的数的值。还有一点是382题使用的是数组，我们这道题使用的是链表。

所以我们完全可以参照382题的做法，由于链表访问比较麻烦，需要从前往后一个个遍历，我们这里直接把链表转化为list集合的方式来求解。我们不把链表转为数组，因为数组需要先声明大小，而链表的大小我们是不知道的（如果先计算链表的长度再转化为数组感觉有点多余），我们直接把链表转化为list集合，这样会更方便些。解题思路完全可以参照第382题，我们来看下代码

暴力求解

```
1   public int[] nextLargerNodes(ListNode head) {
2     List<Integer> list = new ArrayList<>();
3     //链表元素存储到集合中
4     while (head != null) {
5       list.add(head.val);
6       head = head.next;
7     }
8     int[] res = new int[list.size()];
9     for (int i = 0; i < list.size() - 1; i++) {
10      for (int j = i + 1; j < list.size(); j++) {
11        if(list.get(j)>list.get(i)){
12          res[i]=list.get(j);
13          break;
14        }
15      }
16    }
17    return res;
18  }
```

使用栈求解

```
1   public int[] nextLargerNodes(ListNode head) {
2     List<Integer> list = new ArrayList<>();
3     //链表元素存储到集合中
4     while (head != null) {
5       list.add(head.val);
6       head = head.next;
7     }
```

```
    //栈中存储的是元素的下标，并且从栈底到栈顶元素在集合中对应的
    //值是从大到小的
    Stack<Integer> stack = new Stack<>();
    int[] res = new int[list.size()];
    for (int i = 0; i < list.size(); i++) {
      while (!stack.empty() && list.get(stack.peek()) < list.get(i)) {
        //如果栈顶元素对应的值小于当前值，说明栈顶元素遇到了比他小的
        int index = stack.pop();
        res[index] = list.get(i);
      }
      stack.push(i);
    }
    return res;
  }
```

剪枝计算

> 这种是从后往前计算，通过剪枝，减少运算

```
  public int[] nextLargerNodes(ListNode head) {
    List<Integer> list = new ArrayList<>();
    //链表元素存储到集合中
    while (head != null) {
      list.add(head.val);
      head = head.next;
    }
    int[] res = new int[list.size()];
    for (int i = res.length - 1; i >= 0; i--) {
      int j = i + 1;
      int num = 0;
      if (j < res.length)
        num = list.get(j);
      while (j < res.length) {
        if (num > list.get(i)) {
          //如果找到就停止while循环
          res[i] = num;
          break;
        } else if (num == 0) {
          break;
        } else {
          num = res[j++];
        }
      }
    }
    return res;
  }
```

不转化为list集合求解

上面3种解法我们完全是参照第382题的答案写的，并且在计算之前都首先把链表转化为list集合，我们能不能直接使用链表，不把他转化为list集合呢，其实也是可以的，我们只需要使用一个栈来存储链表中每个节点在list中的位置即可，并且栈中元素在集合list中对应的值从栈底到栈顶是递减的，原理还是和上面类似，我们就以上面例二的数据[2,7,4,3,5]来画个图看一下是怎么样的一个实现过程

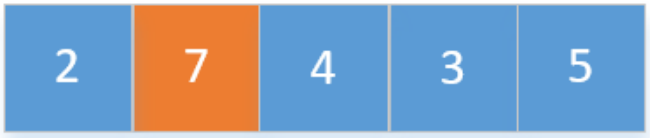

原始状态
list={}
2
7
4
3
5
栈为空，2的下标0入栈
第1步
list={2}
2加入到集合list中
2
7
4
3
5
0

栈不为空，由于当前值7比栈顶元素0对应的值2大，所以栈顶元素0出栈，栈顶元素0对应的值2变成当前值7

0出栈

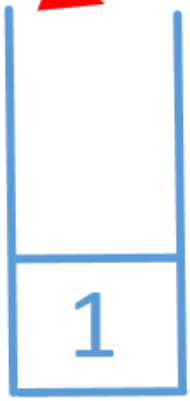


第2步

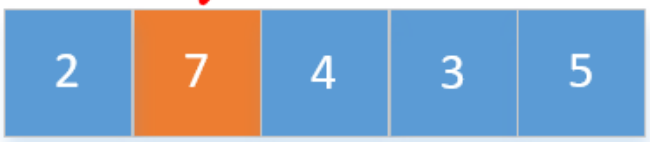


list={7}　这里2变成7

栈为空，7的下标1入栈

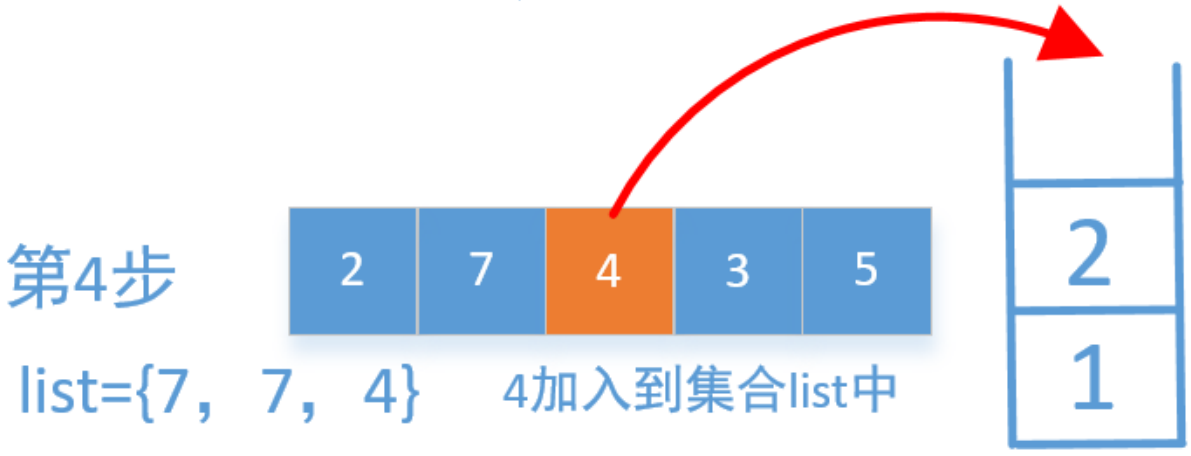


第3步



list={7，7}　7加入到集合list中

当前值4比栈顶元素1对应的
值7小，所以4的下标2入栈



list={7，7，4}　4加入到集合list中

当前值3比栈顶元素2对应的
值4小，所以3的下标3压栈



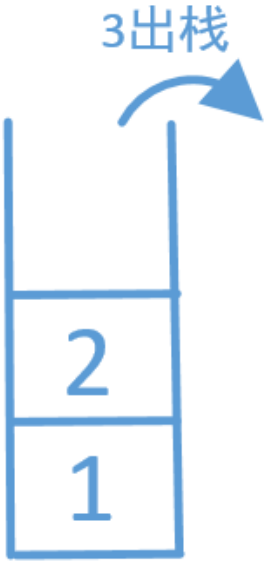
list={7，7，4，3}　3加入到集合list中

栈不为空，由于当前值5比栈顶元素3对应的值3大，所以栈顶元素3出栈，栈顶元素3对应的值3变成当前值5

第6步



list={7，7，4，5}　这里3变成5



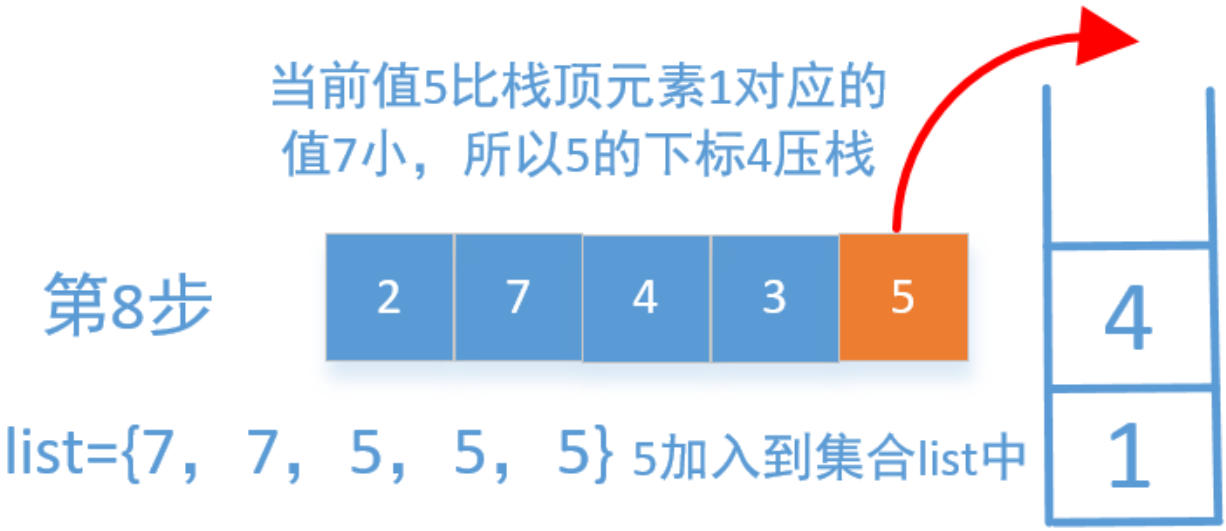
栈不为空，由于当前值5比栈顶元素2对应的值4大，所以栈顶元素2出栈，栈顶元素2对应的值4变成当前值5

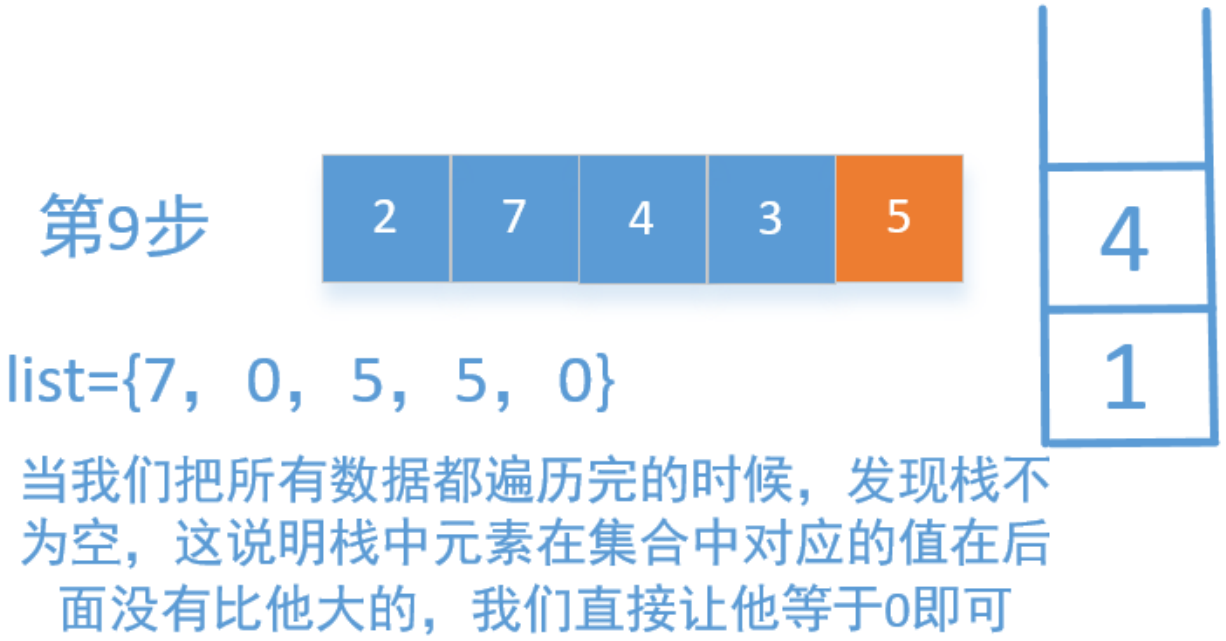
第7步



list={7，7，5，5}　这里4变成5







我们来看下代码

```java
public int[] nextLargerNodes(ListNode head) {
  List<Integer> list = new ArrayList<>();
  Stack<Integer> stack = new Stack();
  for (ListNode node = head; node != null; node = node.next) {
    while (!stack.isEmpty() && node.val > list.get(stack.peek())) {
      list.set(stack.pop(), node.val);
    }
    stack.push(list.size());
    list.add(node.val);
  }
  for (int i : stack) {
    list.set(i, 0);
  }
  int[] res = new int[list.size()];
  for (int i = 0; i < res.length; i++) {
    res[i] = list.get(i);
  }
  return res;
}
```

上面代码核心部分在4-10行，原理和前几种写法类似，14-17行是把list转化为数组，第11-13行遍历这个栈，这个时候栈中的每个元素在list中对应的值在后面没有比他更大的，我们直接让他等于0即可。

递归方式

当然我们还可以改为递归的方式，有兴趣的可以自己看下

```
1   int[] res;
2
3   public int[] nextLargerNodes(ListNode head) {
4     calNode(head, 0, new Stack<>());
5     return res;
6   }
7
8   public void calNode(ListNode node, int index, Stack<Integer> stack) {
9     if (node == null) {
10      res = new int[index];//初始化数组的大小
11      return;
12    }
13    calNode(node.next, index + 1, stack);
14    while (!stack.empty() && stack.peek() <= node.val)
15      stack.pop();
16    res[index] = stack.empty() ? 0 : stack.peek();
17    stack.push(node.val);
18  }
```

注意这里的第14行如果栈顶元素不小于当前值就要出栈，一直到栈顶元素大于当前值为止，为什么要这样写，因为这里的递归对链表的访问实际上相当于从链表的尾到头开始的，也就是逆序。这个可以参照前面的剪枝计算，因为他也在集合中从后往前开始计算的。

往期推荐

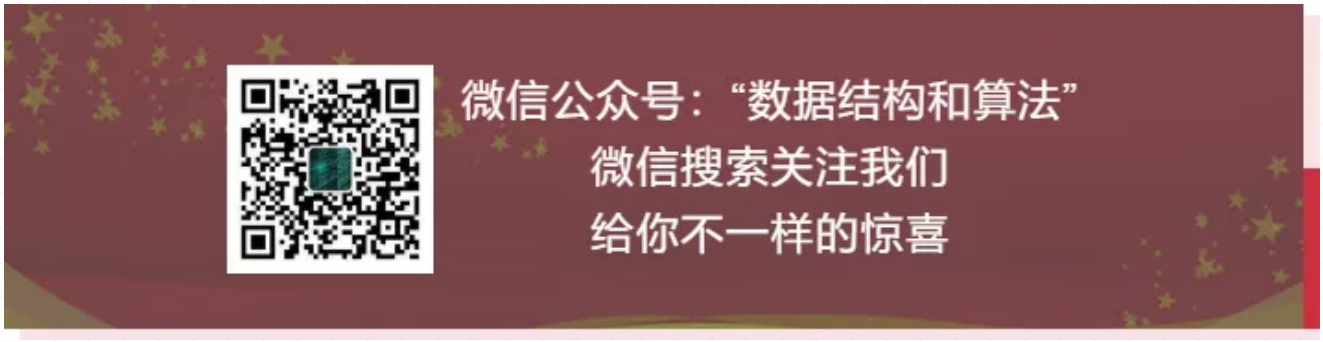
- 382，每日温度的5种解题思路
- 379，柱状图中最大的矩形（难）

381，合并两个有序链表（易）

原创 山大王wld 数据结构和算法 6月9日

收录于话题
#算法图文分析

96个 >



Correction does much, but encouragement does more.

纠正很有用，但鼓励的作用更大。

问题描述

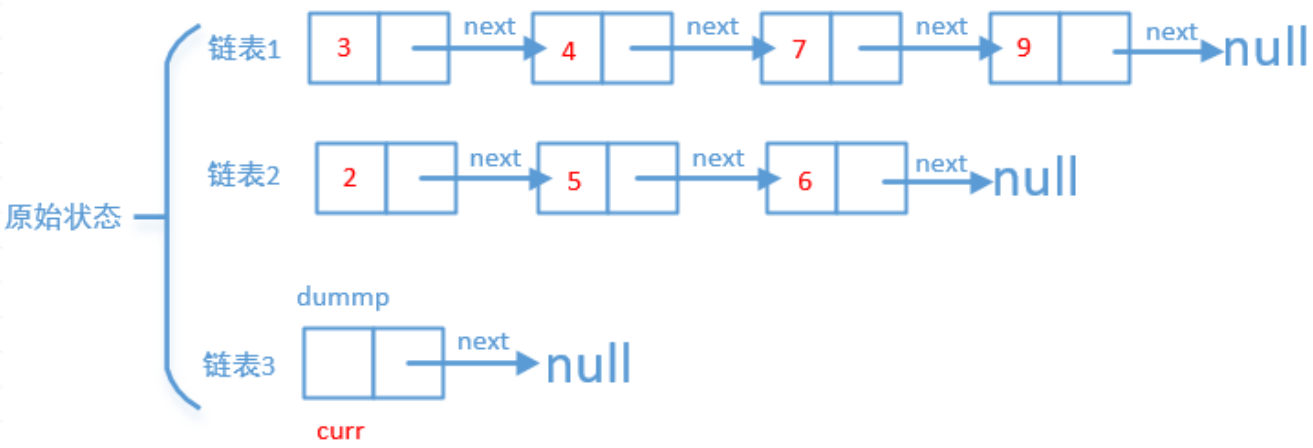
将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

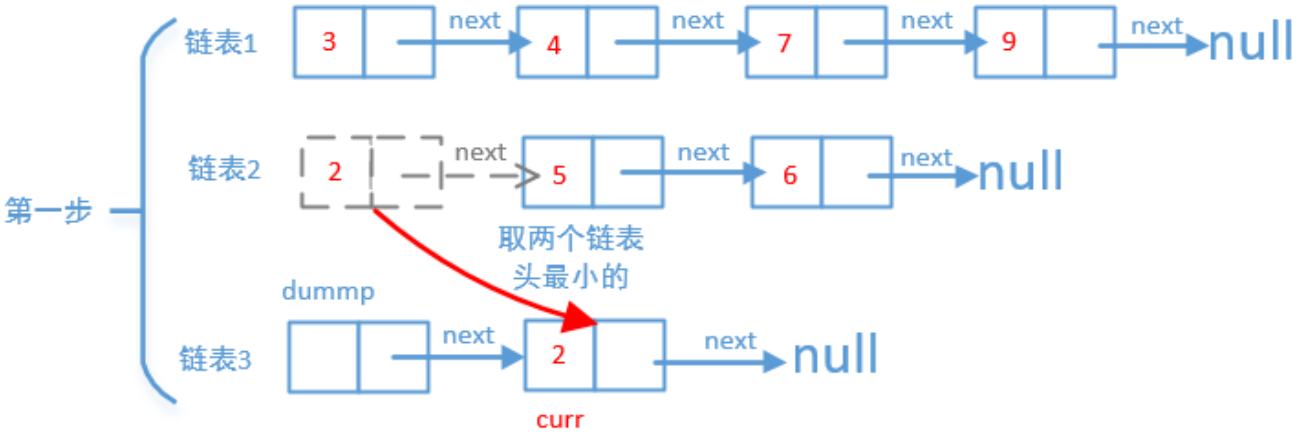
示例：

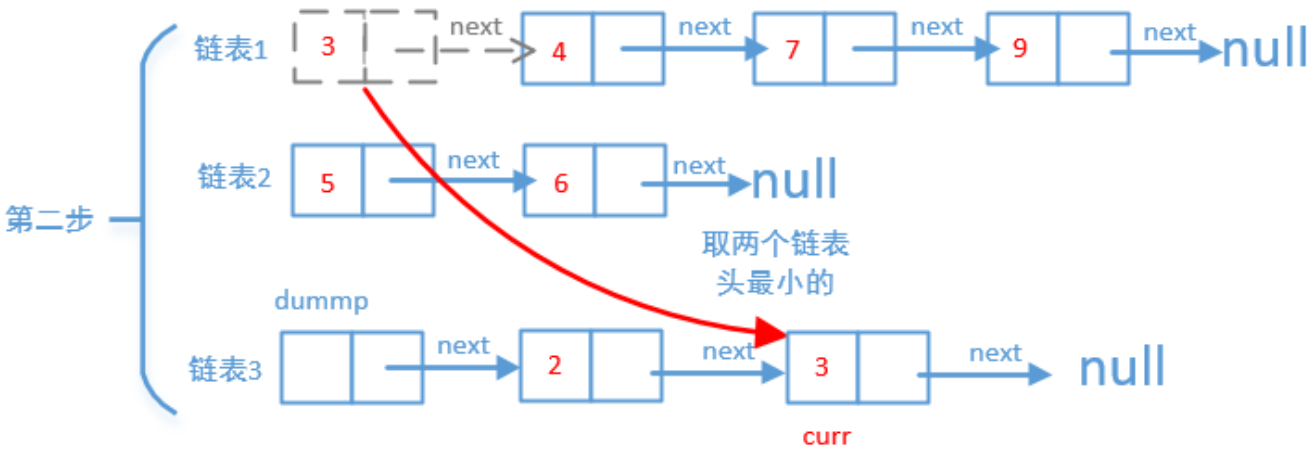
输入：1->2->4，1->3->4
输出：1->1->2->3->4->4

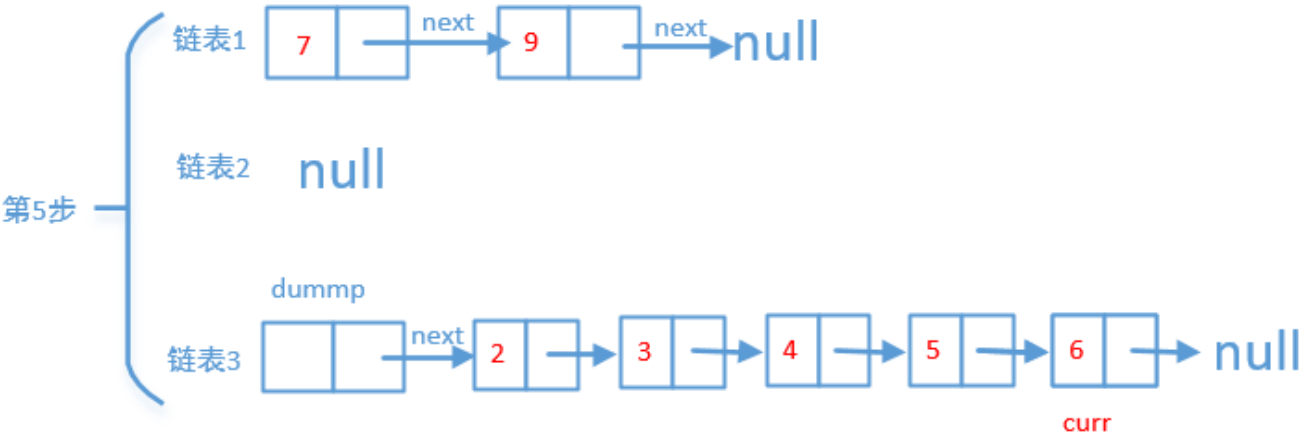
问题分析

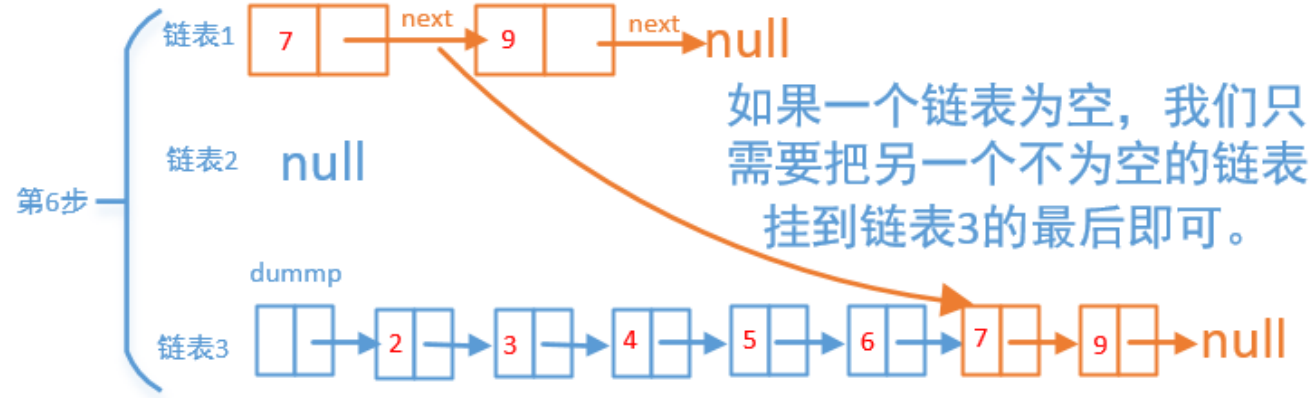
这题比较简单，因为链表是升序的，我们只需要遍历每个链表的头，比较一下哪个小就把哪个链表的头拿出来放到新的链表中，一直这样循环，直到有一个链表为空，然后我们再把另一个不为空的链表挂到新的链表中。我们就以3→4→7→9和2→5→6两个链表来画个图看一下是怎么合并的。











最后返回dummp.next即可

01 链表合并代码

```
1   public ListNode mergeTwoLists(ListNode linked1, ListNode linked2) {
2     //下面4行是空判断
3     if (linked1 == null)
4       return linked2;
5     if (linked2 == null)
6       return linked1;
7     ListNode dummy = new ListNode(0);
8     ListNode curr = dummy;
```

```
    while (linked1 != null && linked2 != null) {
      //比较一下，哪个小就把哪个放到新的链表中
      if (linked1.val <= linked2.val) {
        curr.next = linked1;
        linked1 = linked1.next;
      } else {
        curr.next = linked2;
        linked2 = linked2.next;
      }
      curr = curr.next;
    }
    //然后把那个不为空的链表挂到新的链表中
    curr.next = linked1 == null ? linked2 : linked1;
    return dummy.next;
  }
```

02 链表合并递归写法

```
  public ListNode mergeTwoLists(ListNode linked1, ListNode linked2) {
    if (linked1 == null)
      return linked2;
    if (linked2 == null)
      return linked1;
    if (linked1.val < linked2.val) {
      linked1.next = mergeTwoLists(linked1.next, linked2);
      return linked1;
    } else {
      linked2.next = mergeTwoLists(linked1, linked2.next);
      return linked2;
    }
  }
```

> 递归写法其实我们还可以写的更简洁一些

```
  public ListNode mergeTwoLists(ListNode linked1, ListNode linked2) {
    //只要有一个为空，就返回另一个
    if (linked1 == null || linked2 == null)
      return linked2 == null ? linked1 : linked2;
    //把小的赋值给first
    ListNode first = (linked2.val < linked1.val) ? linked2 : linked1;
    first.next = mergeTwoLists(first.next, first == linked1 ? linked2 : linked1);
    return first;
  }
```

往期推荐

- 352，数据结构-2,链表
- 333，奇偶链表

528，使用栈解基本计算器 II

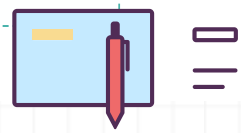
原创 博哥 数据结构和算法 1周前

收录于话题
#算法图文分析

140个 >

If no one else guards the world, then I will come forward.

如果没有别人保卫这个世界，那么我将挺身而出。



问题描述

给你一个字符串表达式 s ，请你实现一个基本计算器来计算并返回它的值。

整数除法仅保留整数部分。

示例 1：

> **输入**：s = "3+2*2"
> **输出**：7

示例 2：

> **输入**：s = " 3/2 "
> **输出**：1

示例 3：

> **输入**：s = " 3+5 / 2 "
> **输出**：5

提示：

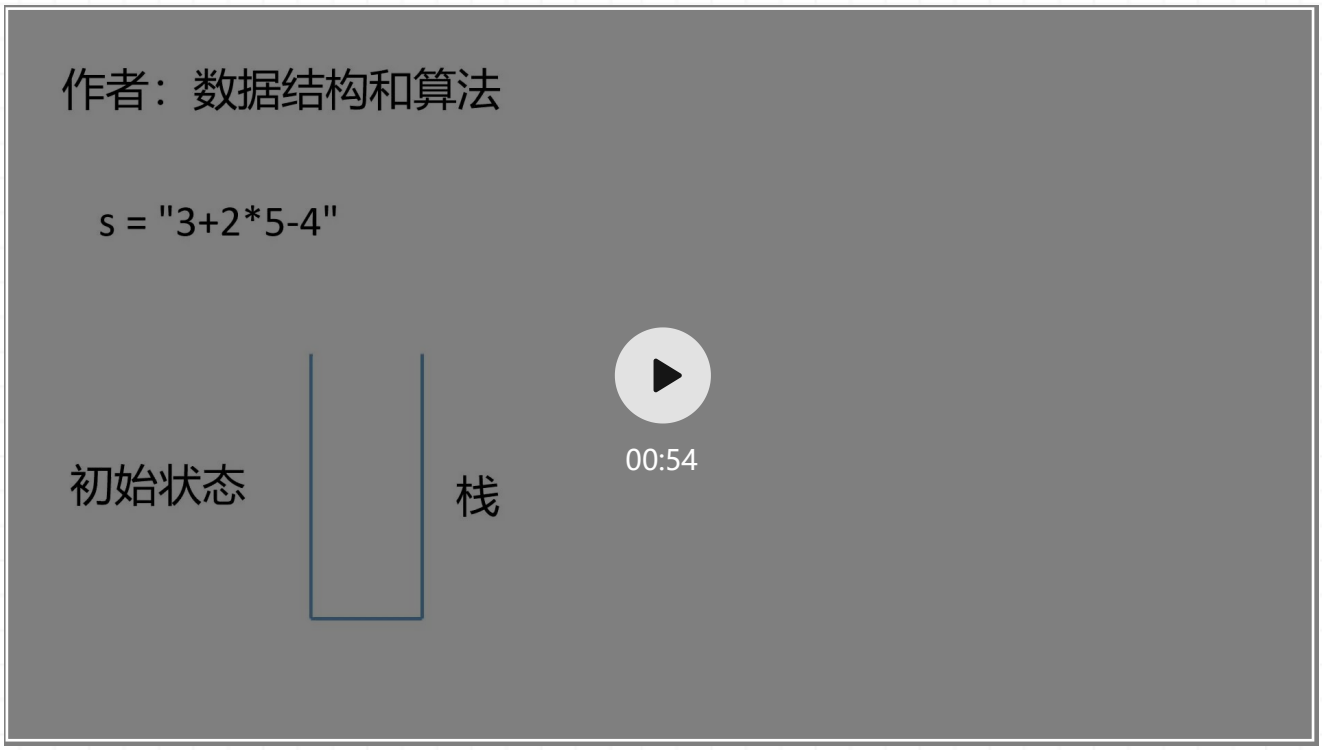
- 1<=s.length<=3*10^5
- s由整数和算符('+', '-', '*', '/') 组成，中间由一些空格隔开
- s表示一个**有效表达式**
- 表达式中的所有整数都是非负整数，且在范围[0, 2^31 - 1]内
- 题目数据保证答案是一个32-bit整数

使用栈解决

提示中说了，是一个有效的表达式，并且只有数字，空格和'+', '-', '*', '/'组成。我们知道算术运算式先算乘除后算加减。

- 当遇到加法或减法的时候我们没法确定要不要计算，比如2+3*4。
- 当遇到乘法或者除法的时候我们是可以直接计算的，因为他们的优先级比较高，比如4*2-3我们可以直接计算4*2的值。或者3-4*2也是可以的。

所以我们可以把不能确定是否要计算的先加入到栈中，可以直接计算的先计算，然后再加入到栈中，我们随便举个例子画个图来看下



代码比较简单，我们来看下

```
1   public int calculate(String s) {
2     //记录每个数字前面的符号，如果是乘法和除法就直接和前面的数字运算，
3     //然后在存放到栈中，如果是加法和减法直接存放到栈中
4     int preSign = '+';
5     Stack<Integer> stack = new Stack<>();
6     int length = s.length();
7     for (int i = 0; i < length; i++) {
8       int ch = s.charAt(i);
9       if (ch == ' ')//过滤掉空格
10        continue;
11      //如果是数字
12      if (ch >= '0' && ch <= '9') {
```

```
            //找到连续的数字字符串，把它转化为整数
            int num = 0;
            while (i < length && (ch = s.charAt(i)) >= '0' && ch <= '9') {
                num = num * 10 + ch - '0';
                i++;
            }
            //这个是为了抵消上面for循环中的i++
            i--;
            //乘法和除法，运算之后在存放到栈中。加法和减法直接存放到栈中
            if (preSign == '*') {
                stack.push(num * stack.pop());
            } else if (preSign == '/') {
                stack.push(stack.pop() / num);
            } else if (preSign == '+') {
                stack.push(num);
            } else if (preSign == '-') {
                stack.push(-num);
            }
        } else {//记录前一个的符号
            preSign = ch;
        }
    }
    //把栈中的所有元素都取出来，计算他们的和
    int res = 0;
    while (!stack.empty()) {
        res += stack.pop();
    }
    return res;
}
```

总结

算术计算，优先级高的先计算，计算的结果入栈，优先级低的直接入栈，最后再统计栈中所有元素的和即可。

往期推荐

- 519，单调栈解下一个更大元素 I
- 508，使用栈来判断有效的括号
- 500，验证栈序列
- 416，剑指 Offer-用两个栈实现队列

526，删除字符串中的所有相邻重复项

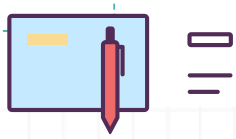
原创 博哥 数据结构和算法 3天前

收录于话题
#算法图文分析

137个 >

It may not be pretty, but we headed to the city.

其貌虽不扬，扬帆亦远航。



问题描述

给出由小写字母组成的字符串S，**重复项删除操作**会选择两个相邻且相同的字母，并删除它们。

在S上反复执行重复项删除操作，直到无法继续删除。

在完成所有重复项删除操作后返回最终的字符串。答案保证唯一。

示例：

> **输入**：`"abbaca"`
> **输出**：`"ca"`
> **解释**：
> 例如，在 `"abbaca"` 中，我们可以删除 `"bb"` 由于两字母相邻且相同，这是此时唯一可以执行删除操作的重复项。之后我们得到字符串 `"aaca"`，其中又只有 `"aa"` 可以执行重复项删除操作，所以最后的字符串为 `"ca"`。

提示：

- `1 <= S.length <= 20000`
- `S` 仅由小写英文字母组成。

使用栈解决

这题让把挨着的并且相同的字符同时给删除，最简单的实现方式就是使用栈来解决。

我们遍历字符串中的所有字母：

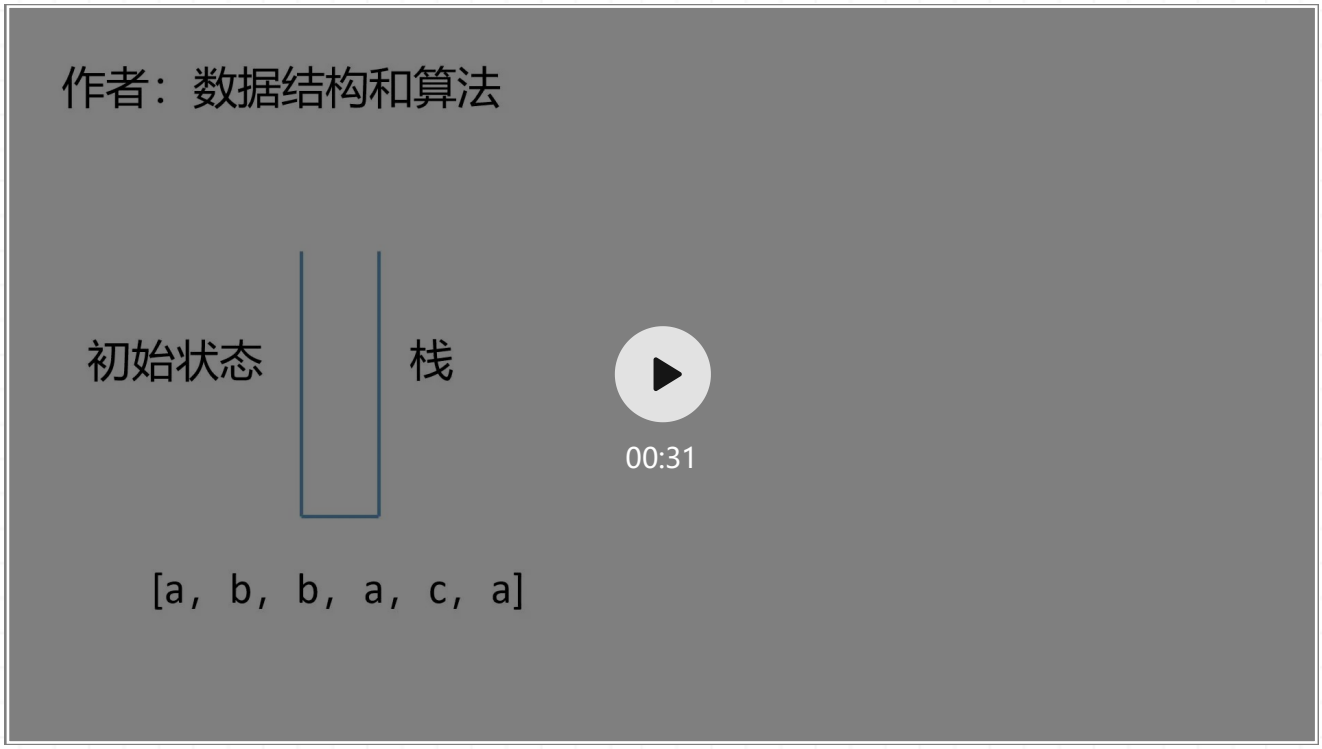
1，如果栈为空

- 就把当前字母加入到栈中。

2，如果栈不为空

- 如果当前字母等于栈顶元素，说明他俩是相邻且相同的，让他俩同时消失。
- 如果当前字母不等于栈顶元素，说明他俩是相邻但不相同，直接让当前字母入栈。

具体以示例为例来看下视频



上面视频中是先入栈之后，如果两个有相同的在同时消失，其实我们不需要先入栈，而是先比较，如果相同让栈顶元素出栈即可，来看下代码

```
public String removeDuplicates(String S) {
    char[] chars = S.toCharArray();
    Stack<Character> stack = new Stack<>();
    int index = 0;
    int length = S.length();
    while (index < length) {
        char current = chars[index++];
        if (!stack.empty() && stack.peek() == current) {
            //如果栈顶的值和当前遍历的值相同，他两直接消失
            stack.pop();
        } else {
            //如果栈为空，或者栈顶元素和当前值不相同，就把当前值压入栈
            stack.push(current);
        }
    }
    //下面是把栈中的元素转化为字符串
    StringBuilder stringBuilder = new StringBuilder(stack.size());
    while (!stack.empty())
        stringBuilder.append(stack.pop());
```

```
    return stringBuilder.reverse().toString();
}
```

使用双指针解决

大家应该都玩过连连看吧，就是挨着的多个相同的会同时消失。我们这里类似于连连看，挨着两个相同的同时消失，可以使用两个指针。

- 一个right一直往右移动，然后把指向的值递给left指向的值即可。
- 一个left每次都会比较挨着的两个是否相同，如果相同，他两同时消失。

具体看下视频



代码如下

```
public String removeDuplicates(String S) {
    int left = 0;
    int right = 0;
    int length = S.length();
    char[] chars = S.toCharArray();
    while (right < length) {
        //先把右边的字符赋值给左边
        chars[left] = chars[right];
        //然后判断左边挨着的两个字符是否相同，如果相同，
        //他两同时消失，也就是left往回退两步
        if (left > 0 && chars[left - 1] == chars[left])
            left -= 2;
        ++right;
        ++left;
    }
    return new String(chars, 0, left);
}
```

还可以使用StringBuilder，原理都是一样的，来看下代码

```
public String removeDuplicates(String S) {
    StringBuilder stringBuilder = new StringBuilder();
```

```
    char[] chars = S.toCharArray();
    for (char ch : chars) {
      int size = stringBuilder.length();
      if (size > 0 && stringBuilder.charAt(size - 1) == ch) {
        stringBuilder.deleteCharAt(size - 1);
      } else {
        stringBuilder.append(ch);
      }
    }
    return stringBuilder.toString();
  }
```

总结

这题使用栈是最容易想到的，每次只需要比较要入栈的值和栈顶元素即可，如果一样，就同时消失。当然还可以使用双端队列，他是两头都可以添加和删除的一种数据结构，具体可以看下[359，数据结构-3,队列]

往期推荐

- 508，使用栈来判断有效的括号
- 500，验证栈序列
- 497，双指针验证回文串
- 398，双指针求无重复字符的最长子串

523，单调栈解下一个更大元素 II

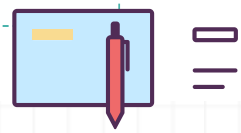
原创 博哥 数据结构和算法 6天前

收录于话题
#算法图文分析

137个 >

Women's value has been under-recognized for far too long.

女性的价值被低估太久了。



问题描述

给定一个**循环数组**（最后一个元素的下一个元素是数组的第一个元素），输出每个元素的下一个更大元素。数字x的下一个更大的元素是按数组遍历顺序，这个数字之后的第一个比它更大的数，这意味着你应该循环地搜索它的下一个更大的数。如果不存在，则输出-1。

示例 1:

> **输入**: [1,2,1]
> **输出**: [2,-1,2]
> **解释**: 第一个 1 的下一个更大的数是 2;
> 数字 2 找不到下一个更大的数;
> 第二个 1 的下一个最大的数需要循环搜索，结果也是 2。

注意: 输入数组的长度不会超过10000。

单调栈解决

这题可以参照前面讲的

379，柱状图中最大的矩形（难）

382，每日温度的5种解题思路

386，链表中的下一个更大节点

[519，单调栈解下一个更大元素 I](#)

他们都使用了单调栈的特性。这道题相对于上面几道题还是比较简单的，主要考察对栈的使用。所谓单调栈，也就是说栈中的元素（这个元素不一定是具体的值，也有可能是数组的下标）要么是递增的要么是递减的。

具体操作是，首先遍历数组中的所有元素，遍历的时候判断栈是否为空：

一，如果栈为空，就把遍历的元素下标加入到栈中。

二，如果栈不为空，查看栈顶元素在数组中对应的值是否小于这个遍历的元素：

1，如果小于，说明栈顶元素遇到右边第一个比他大的值，然后栈顶元素出栈，记录下这个值。如果栈还不为空，继续判断……

2，如果不小于，把当前遍历的元素下标加入到栈中。

这里栈中存储的不是数组中元素具体的值，而是数组中元素对应的下标，模板如下所示。

```
int res[] = new int[length];
Stack<Integer> stack = new Stack<>();
for (int i = 0; i < length; i++) {
  while (!stack.isEmpty() && nums[stack.peek()] < nums[i])
    res[stack.pop()] = nums[i];
  //当前元素的下标入栈
  stack.push(i);
}
```

对于这道题完全可以套用上面的模板，因为题中说了是循环数组，我们只需要把数组遍历两遍即可，代码如下

```
public int[] nextGreaterElements(int[] nums) {
  int length = nums.length;
  int res[] = new int[length];
  Arrays.fill(res, -1);//默认都为-1
  Stack<Integer> stack = new Stack<>();
  //相当于把数组循环两遍
  for (int i = 0; i < length * 2; i++) {
    //遍历数组的第index（index从0开始）个元素，因为数组会遍历
    //两遍，会导致数组越界异常，所以这里要对数组长度进行求余
    int index = i % length;
    //单调栈，他存储的是元素的下标，不是元素具体值，从栈顶
    //到栈底所对应的值是递增的（栈顶元素在数组中对应的值最小，
    //栈底元素对应的值最大），如果栈顶元素对应的值比nums[index]小，
    //说明栈顶元素对应的值遇到了右边第一个比他大的值，然后栈顶元素出栈，
    //让他对应的位置变为nums[index]，也就是他右边第一个比他大的值，
    //然后继续判断……
    while (!stack.isEmpty() && nums[stack.peek()] < nums[index])
      res[stack.pop()] = nums[index];
    //当前元素的下标入栈
    stack.push(index);
  }
  return res;
}
```

519，单调栈解下一个更大元素 I

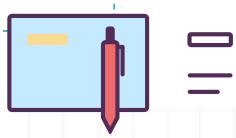
原创 博哥 数据结构和算法 2月24日

收录于话题
#算法图文分析

137个 >

There is no birth of consciousness without pain.

没有伤痛，就不会有醒觉。



问题描述

给你两个**没有重复元素**的数组nums1和nums2，其中nums1是nums2 的子集。

请你找出nums1中每个元素在nums2中的下一个比其大的值。

nums1中数字x的下一个更大元素是指x在nums2中对应位置的右边的第一个比x大的元素。如果不存在，对应位置输出-1。

示例 1:

```
输入:nums1=[4,1,2],nums2=[1,3,4,2].
输出:[-1,3,-1]
解释:
    对于num1中的数字4 ，你无法在第二个数组中找到下一个更大的数字，因此输出-1。
    对于num1中的数字1，第二个数组中数字1右边的下一个较大数字是 3 。
    对于num1中的数字2，第二个数组中没有下一个更大的数字，因此输出-1 。
```

示例 2:

```
输入:nums1=[2,4],nums2=[1,2,3,4].
输出:[3,-1]
解释:
    对于num1中的数字2，第二个数组中的下一个较大数字是3 。
    对于num1中的数字4，第二个数组中没有下一个更大的数字，因此输出-1 。
```

提示：

- 1<=nums1.length<=nums2.length<=1000
- 0<=nums1[i],nums2[i]<=10^4
- nums1和nums2中所有整数**互不相同**
- nums1中的所有整数同样出现在nums2中

单调栈解决

这题说的很明白，nums1是nums2的子集，并且nums1和nums2中都没有重复的元素。最简单的一种解决方式就是暴力求解，使用两个for循环，先找到nums1中每个元素在nums2中的位置，然后再从那个位置之后查找第一个比他大的，这个比较简单，就不在写，我们看一下使用单调栈该怎么解决。

首先遍历nums2的所有元素，遍历的时候判断栈是否为空：
一，如果栈为空，就把遍历的元素加入到栈中。
二，如果栈不为空，查看栈顶元素是否小于这个遍历的元素：
1，如果小于，说明栈顶元素遇到右边第一个比他大的值，然后栈顶元素出栈，记录下这个值。如果栈还不为空，继续判断……
2，如果不小于，把当前遍历的元素加入到栈中。

上面关键一点是怎么记录右边第一个比他大的，我们可以使用Map，来看下视频



最后再来看下代码

```
1   public int[] nextGreaterElement(int[] nums1, int[] nums2) {
```

```
    //map中的key是数组中元素的值，value是这个值遇到的
    //右边第一个比他大的值
    Map<Integer, Integer> map = new HashMap<>();
    //单调栈，从栈顶到栈底是递增的
    Stack<Integer> stack = new Stack<>();
    //遍历nums2的所有元素
    for (int num : nums2) {
      //如果栈顶元素小于num，说明栈顶元素遇到了右边
      //第一个比他大的值，然后栈顶元素出栈，记录下
      //这个值。
      while (!stack.empty() && stack.peek() < num)
        map.put(stack.pop(), num);
      //当前元素入栈
      stack.push(num);
    }
    //遍历nums1的所有元素，然后在map中查找，如果没有查找到，
    //说明没有遇到右边比他大的值，默认给-1。
    int[] res = new int[nums1.length];
    for (int i = 0; i < nums1.length; i++) {
      res[i] = map.getOrDefault(nums1[i], -1);
    }
    return res;
  }
```

总结

栈是一种**先进后出**的数据结构，使用单调栈先找出nums2中每个元素下一个比他大的值，存储在map中，然后在遍历nums1查找。

往期推荐

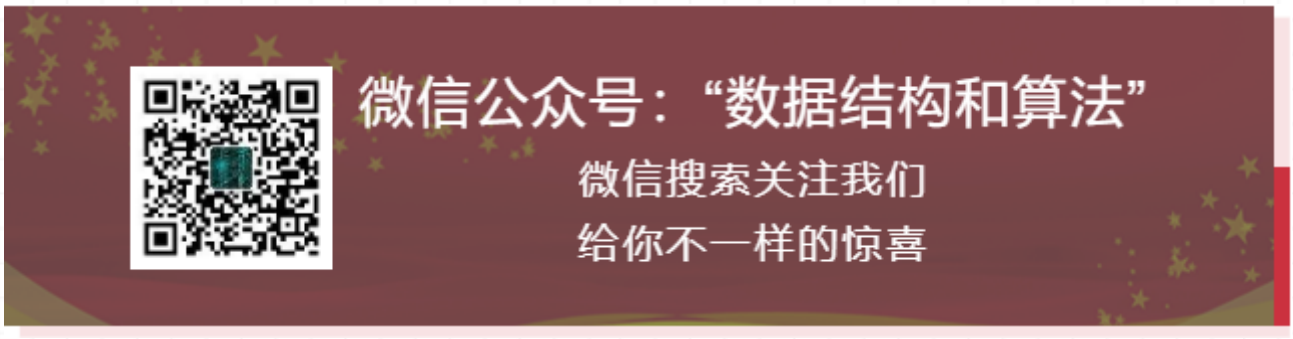
- 508，使用栈来判断有效的括号
- 500，验证栈序列
- 438，剑指 Offer-栈的压入、弹出序列
- 416，剑指 Offer-用两个栈实现队列

508，使用栈来判断有效的括号

原创 山大王wld 数据结构和算法 1月17日

收录于话题
#算法图文分析

137个 >



Life is 10% what happens to you and 90% how you react to it.

生活的10%是发生在你身上的事，剩下的90%是你如何应对。

问题描述

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

示例 1:

> **输入**: "()"
> **输出**: true

示例 2:

> **输入**: "()[]{}"
> **输出**: true

示例 3:

> **输入**: "(]"
> **输出**: false

示例 4:

> **输入**: "([)]"
> **输出**: false

示例 5:

> **输入**: "{[]}"
> **输出**: true

使用栈来解决

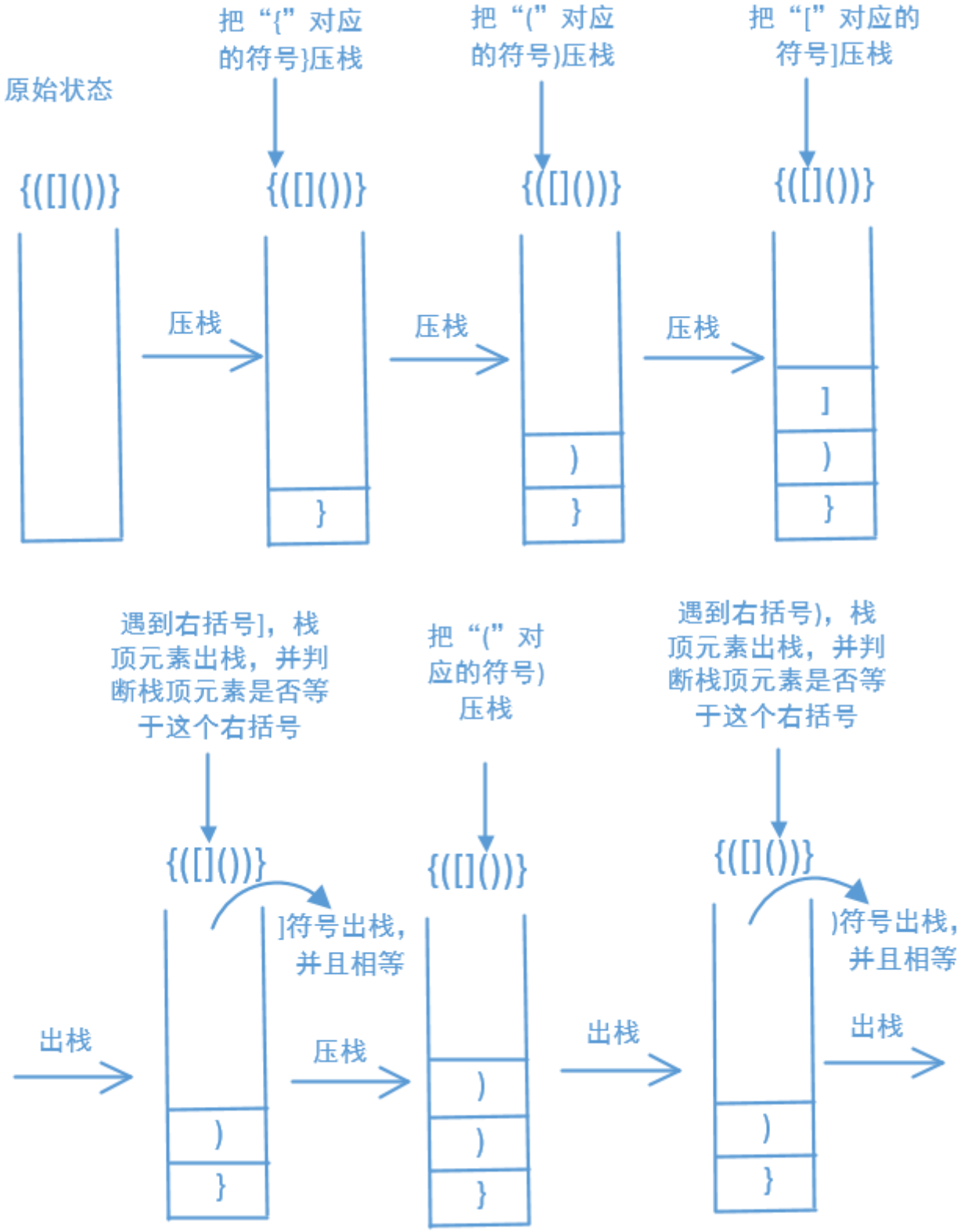
要判断括号的有效性，左括号必须和右括号相对应。如果是有效括号，并且他们中间还有括号，那么他们必须也是有效的，所以最简单的一种方式就是使用栈来解决。

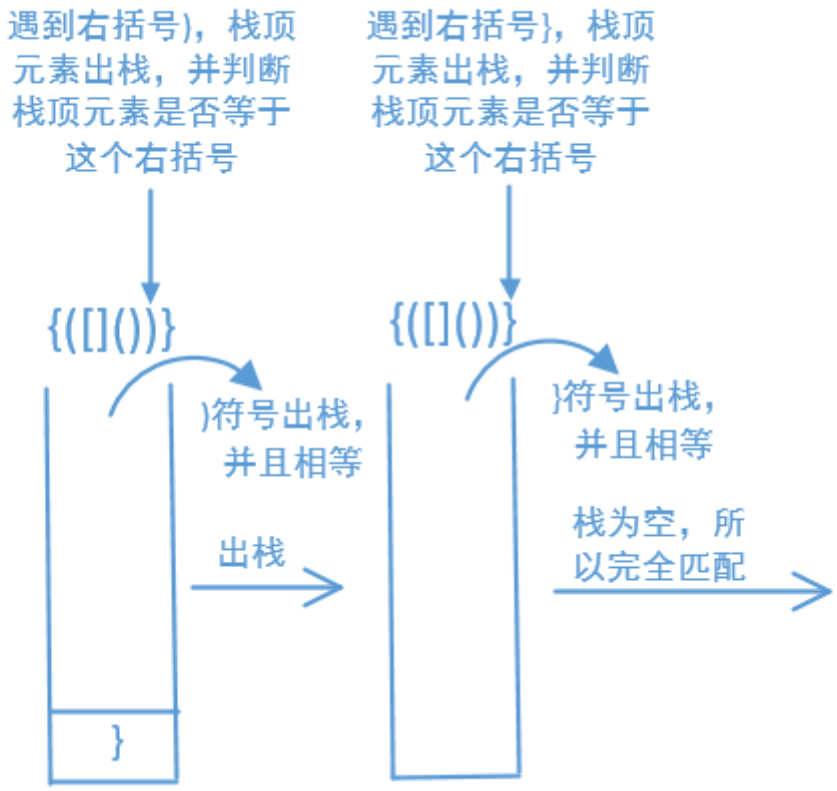
我们遍历字符串中的所有字符
1，如果遇到了左括号，就把对应的右括号压栈（比如遇到了字符'('，就把字符')'压栈）。

2，如果遇到了右括号
　　1）查看栈是否为空，如果为空，说明不能构成有效的括号，直接返回false。
　　2）如果栈不为空，栈顶元素出栈，然后判断出栈的这个元素是否等于这个右括号，如果不等于，说明不匹配，直接返回false。如果匹配，就继续判断字符串的下一个字符。

3，最后如果栈为空，说明是完全匹配，是有效的括号，否则如果栈不为空，说明不完全匹配，不是有效的括号。

这里随便举个例子比如字符串是“{([]())}”，画个图来看一下。





代码如下

```java
public boolean isValid(String s) {
    Stack<Character> stack = new Stack<>();
    char[] chars = s.toCharArray();
    //遍历所有的元素
    for (char c : chars) {
        //如果是左括号，就把他们对应的右括号压栈
        if (c == '(') {
            stack.push(')');
        } else if (c == '{') {
            stack.push('}');
        } else if (c == '[') {
            stack.push(']');
        } else if (stack.isEmpty() || stack.pop() != c) {
            //否则就只能是右括号。
            //1，如果栈为空，说明括号无法匹配。
            //2，如果栈不为空，栈顶元素就要出栈，和这个右括号比较。
            //如果栈顶元素不等于这个右括号，说明无法匹配，
            //直接返回false。
            return false;
        }
    }
    //最后如果栈为空，说明完全匹配，是有效的括号。
    //否则不完全匹配，就不是有效的括号
    return stack.isEmpty();
}
```

总结

栈是一种先进后出的数据结构，使用栈来判断括号的有效性是最简单的一种解决方式。

500，验证栈序列

原创 山大王wld 数据结构和算法 3天前

收录于话题
#算法图文分析

111个 >



Travel can be one of the most rewarding forms of introspection.

旅行可能是最有益的自省方式之一。

问题描述

给定pushed和popped两个序列，每个序列中的值都不重复，只有当它们可能是在最初空栈上进行的推入push和弹出pop操作序列的结果时，返回true；否则，返回false。

示例 1：

> **输入**：pushed = [1,2,3,4,5],
> popped = [4,5,3,2,1]
>
> **输出**：true
> **解释**：我们可以按以下顺序执行：
> push(1), push(2), push(3), push(4), pop() -> 4,
> push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1

示例 2：

> **输入**：pushed = [1,2,3,4,5],
> popped = [4,3,5,1,2]
>
> **输出**：false
> **解释**：1 不能在 2 之前弹出。

提示：

- 0<=pushed.length==popped.length<=1000
- 0<=pushed[i],popped[i]<1000
- pushed是popped的排列。

使用栈来解决

这题说的pushed数组是入栈的顺序，popped数组是否是其中的一个出栈的顺序。注意这里的入栈和出栈并不是一直入栈，也不是一直出栈，有可能在入栈某个值的时候会出栈，也有可能在出栈某个值的时候在继续入栈。

我们首先要明白栈是一种先进后出的数据结构，在前面也有过介绍363，数据结构-4，栈，栈就好比把书一本本的摞起来，放的时候只能从上面放，拿的时候也只能从上面拿。

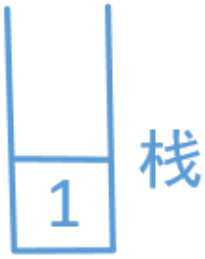
对于这道题，我们可以把数组pushed中的元素一个个入栈，每入栈一个元素就要拿栈顶元素和popped中的第一个元素比较，看是否一样，如果一样，就出栈，然后再拿新的栈顶元素和popped中第2个元素比较……。如果栈顶元素不等于popped中的第一个元素，那么数组pushed中的元素就继续入栈，入栈之后再和popped中的第一个元素比较……。

最后再判断栈是否为空，如果为空，说明pushed数组通过一系列的出栈和入栈能得到popped数组，直接返回true。否则就表示没法得到popped数组，直接返回false。

描述有点绕，我们以示例1为例画个图来看一下

pushed=[1,2,3,4,5] popped=[4,5,3,2,1]

第1步，1入栈



因为栈顶元素不等于popped的第一个元素，所以栈顶元素不用出栈

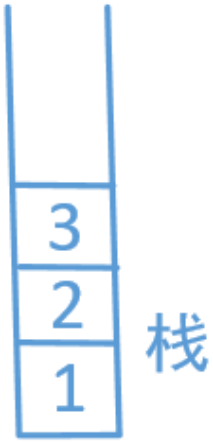
.....

同理，第2，3，4步，数组pushed中的值2，3，4分别入栈



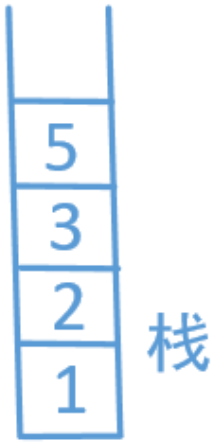
这个时候栈顶元素4等于数组popped中的第一个元素，所以栈顶元素4出栈

第5步



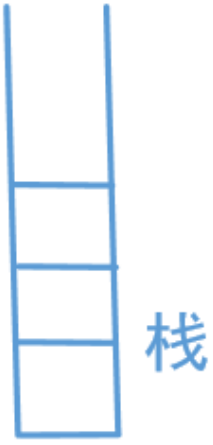
这个时候新的栈顶元素要和popped的第二个元素比较，因为他俩不相同，所以栈顶元素就不要出栈

第6步，pushed数组中的最后一个元素5入栈



这个时候栈顶元素5等于数组popped中的第二个元素，所以栈顶元素5出栈

第7，8，9步



5出栈之后新的栈顶元素是3，等于popped数组中的第三个元素，所以栈顶元素要出栈，同理，2，1也要出栈，这个时候栈就为空了。

原理比较简单，我们来看下代码

```java
public boolean validateStackSequences(int[] pushed, int[] popped) {
    //创建一个栈
    Stack<Integer> stack = new Stack<>();
    //记录popped数组访问到哪一个元素了
    int index = 0;
    //遍历pushed数组中的所有元素
    for (int num : pushed) {
        stack.push(num);//把当前元素push到栈中
        while (!stack.empty() && stack.peek() == popped[index]) {
            //如果栈顶元素等于popped[index]，就让栈顶元素出栈
            stack.pop();
            index++;
        }
```

```
    }
    return stack.empty();
}
```

使用单个指针

我们还可以只用单个变量i来结合pushed数组来模拟栈的存储，原理和上面类似，来看下代码

```
public boolean validateStackSequences(int[] pushed, int[] popped) {
    int i = 0;//相当于栈中元素的个数
    int j = 0;//记录popped数组访问到哪个元素了
    //遍历pushed数组中的所有元素
    for (int num : pushed) {
        pushed[i++] = num;//把当前元素在从新放到pushed数组中
        while (i > 0 && pushed[i - 1] == popped[j]) {
            // pushed[i - 1]类似于栈顶的元素
            --i;
            ++j;
        }
    }
    return i == 0;
}
```

问题分析

这道题相对来说还是比较简单的，首先明白栈是先进后出的一种数据结构，理解这点，这题很容易就能写出来。

往期推荐

- 438，剑指 Offer-栈的压入、弹出序列
- 437，剑指 Offer-包含min函数的栈
- 416，剑指 Offer-用两个栈实现队列
- 363，数据结构-4,栈

438，剑指 Offer-栈的压入、弹出序列

原创 山大王wld 数据结构和算法 8月20日

收录于话题
#剑指offer

27个 >



Nothing in the world can take the place of persistence.

世界上没有什么可以取代坚持。

问题描述

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。

例如，序列 {1,2,3,4,5} 是某栈的压栈序列，序列 {4,5,3,2,1} 是该压栈序列对应的一个弹出序列，但 {4,3,5,1,2} 就不可能是该压栈序列的弹出序列。

示例 1：

> **输入**：pushed = [1,2,3,4,5],
> popped = [4,5,3,2,1]
>
> **输出**：true
> **解释**：我们可以按以下顺序执行：
> push(1), push(2), push(3), push(4), pop() -> 4,

> push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1

示例 2：

> **输入**：pushed = [1,2,3,4,5],
> popped = [4,3,5,1,2]
>
> **输出**：false
> **解释**：1 不能在 2 之前弹出。

提示：

1. 0 <= pushed.length == popped.length <= 1000
2. 0 <= pushed[i], popped[i] < 1000
3. pushed 是 popped 的排列。

问题分析

这题主要考察对栈的理解，栈是一种先进后出的数据结构，如果栈顶元素不出栈，那么栈顶元素下面的元素都是不能出栈的。

一种解决方式就是把pushed数组的元素逐个压栈，当栈顶元素等于popped数组中第一个元素的时候，就让栈顶元素出栈，这个时候再用popped数组的第2个元素和栈顶元素比较，如果相同继续出栈……，最后判断栈是否为空即可，来看下代码

```java
public boolean validateStackSequences(int[] pushed, int[] popped) {
    Stack<Integer> stack = new Stack<>();
    int index = 0;
    for (int val : pushed) {
        //pushed数组中的元素逐个压栈
        stack.push(val);
        while (!stack.empty() && stack.peek() == popped[index]) {
            stack.pop();
            index++;
        }
    }
    return stack.empty();
}
```

总结

这题主要还是考察对栈的熟练使用。

437，剑指 Offer-包含min函数的栈

原创 山大王wld 数据结构和算法 8月19日

收录于话题
#剑指offer

27个 >



The best way out is always through.

最好的出路永远都是勇往直前。

问题描述

定义栈的数据结构，请在该类型中实现一个能够得到栈的最小元素的 min 函数在该栈中，调用 min、push 及 pop 的时间复杂度都是 O(1)。

示例:

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.min();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.min();   --> 返回 -2.
```

提示：

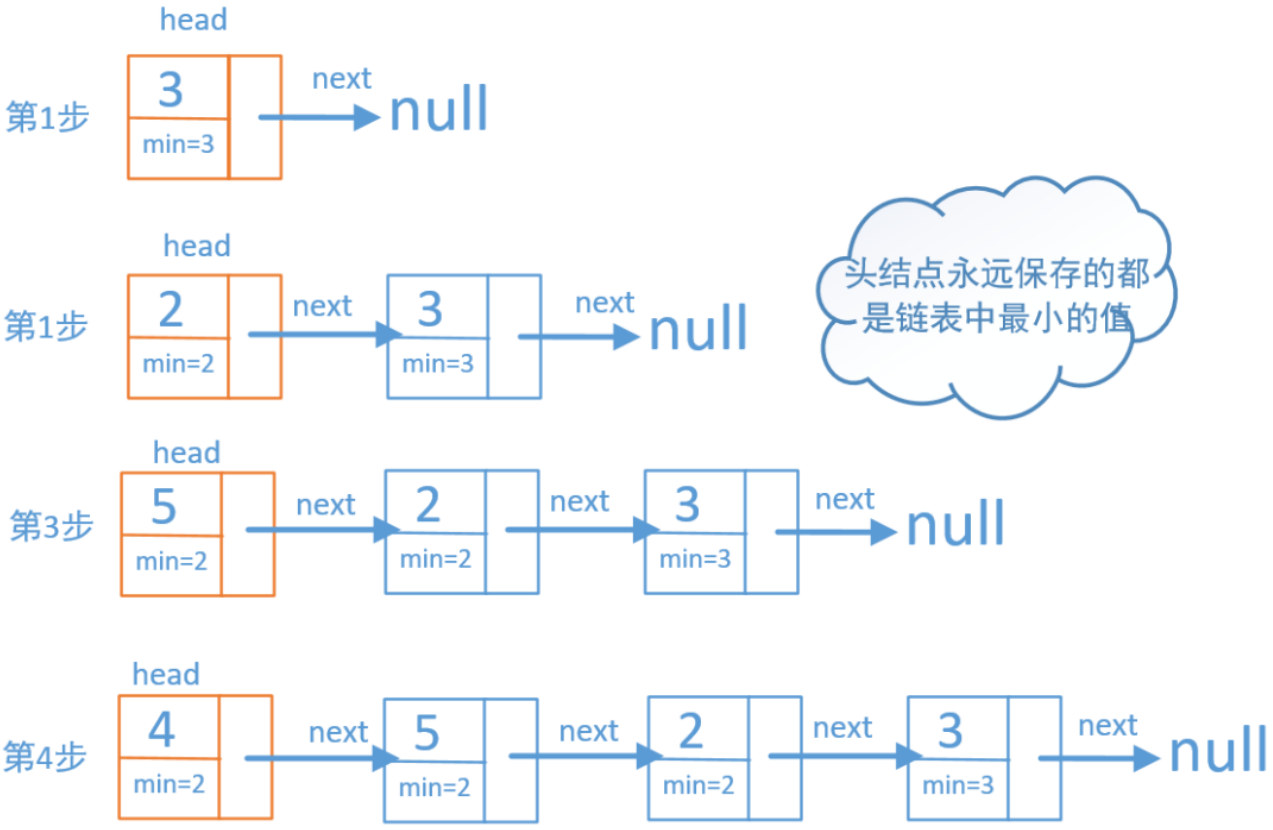
1. 各函数的调用总次数不超过 20000 次

使用辅助类解决

这道题让我们自定义一个栈，有push，pop，top，min四个函数。这题和官方的Stack相比就多了一个min函数。栈的实现我们可以使用链表，先来定义一个链表类

```
class ListNode {
  public int val;
  public int min;//最小值
  public ListNode next;

  public ListNode(int val, int min, ListNode next) {
    this.val = val;
    this.min = min;
    this.next = next;
  }
}
```

这里对链表的操作永远都是链表的头，假如往栈中加入3→2→5→4，画个图来看一下使用链表怎么操作的



代码比较简单，来看下

```
class MinStack {
  //链表头，相当于栈顶
  private ListNode head;

  //压栈，需要判断栈是否为空
  public void push(int x) {
    if (empty())
      head = new ListNode(x, x, null);
    else
      head = new ListNode(x, Math.min(x, head.min), head);
```

```
    }

    //出栈，相当于把链表头删除
    public void pop() {
      if (empty())
        throw new IllegalStateException("栈为空……");
      head = head.next;
    }

    //栈顶的值也就是链表头的值
    public int top() {
      if (empty())
        throw new IllegalStateException("栈为空……");
      return head.val;
    }

    //链表中头结点保存的是整个链表最小的值，所以返回head.min也就是
    //相当于返回栈中最小的值
    public int min() {
      if (empty())
        throw new IllegalStateException("栈为空……");
      return head.min;
    }

    //判断栈是否为空
    private boolean empty() {
      return head == null;
    }
}
```

上面解决方式是使用一个辅助的类，实际上如果使用辅助类，我们也可以使用官方提供的栈，像下面这样。

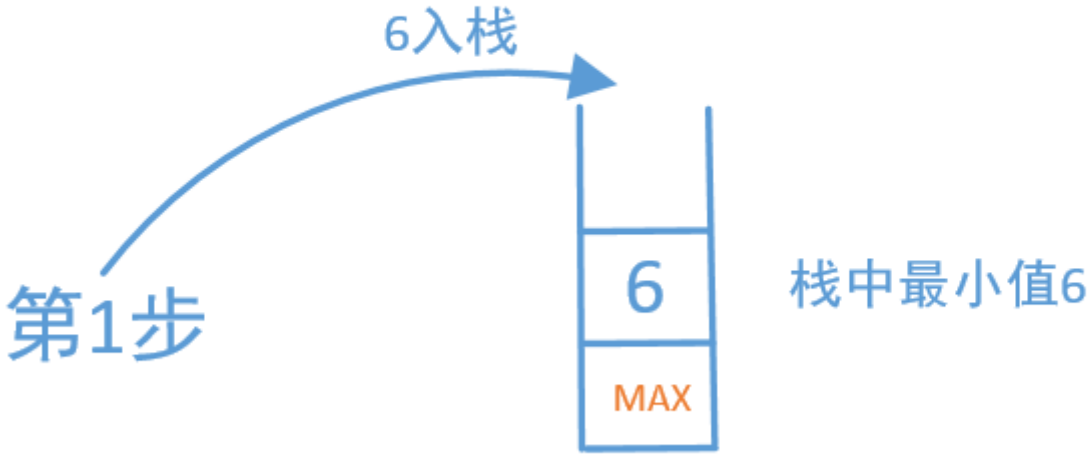
```
class MinStack {
    private Stack<StackNode> stack = new Stack<>();

    //压栈
    public void push(int x) {
      if (empty()) {
        stack.push(new StackNode(x, x));
      } else {
        stack.push(new StackNode(x, Math.min(x, min())));
      }
    }

    //出栈
    public void pop() {
      if (empty())
        throw new IllegalStateException("栈为空……");
      stack.pop();
    }

    public int top() {
      if (empty())
        throw new IllegalStateException("栈为空……");
      return stack.peek().val;
    }

    public int min() {
      if (empty())
        throw new IllegalStateException("栈为空……");
      return stack.peek().min;
    }

    //判断栈是否为空
    private boolean empty() {
      return stack.isEmpty();
```

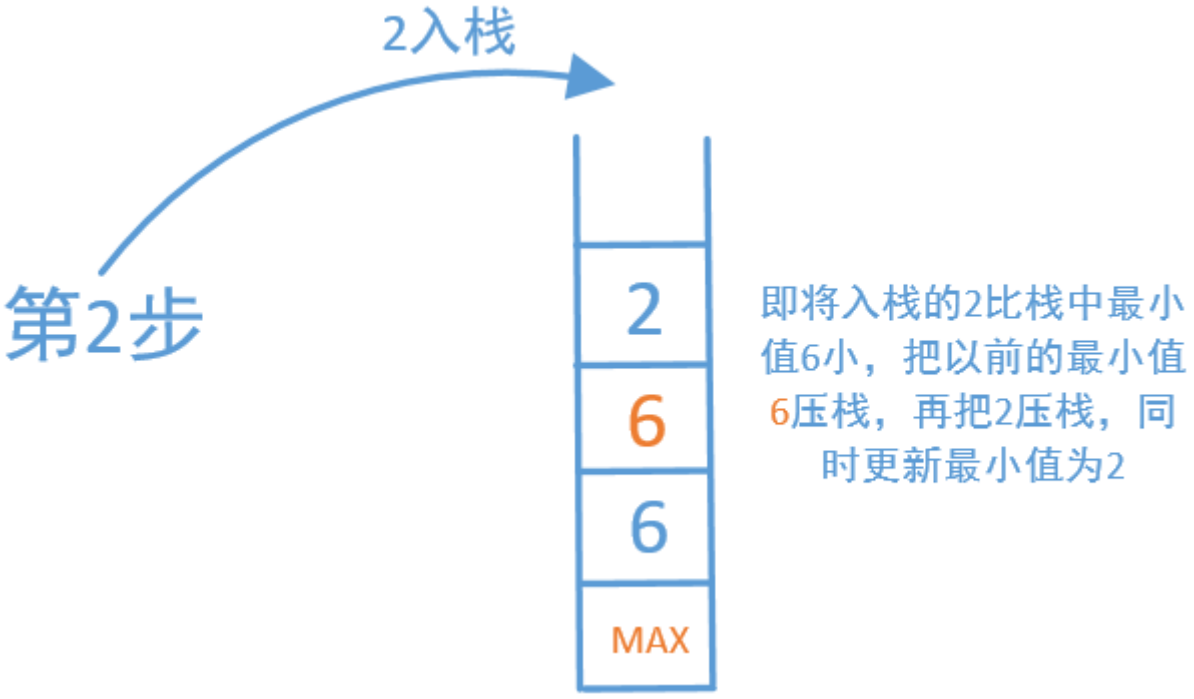
```
    }
  }

  class StackNode {
    public int val;
    public int min;

    public StackNode(int val, int min) {
      this.val = val;
      this.min = min;
    }
  }
```

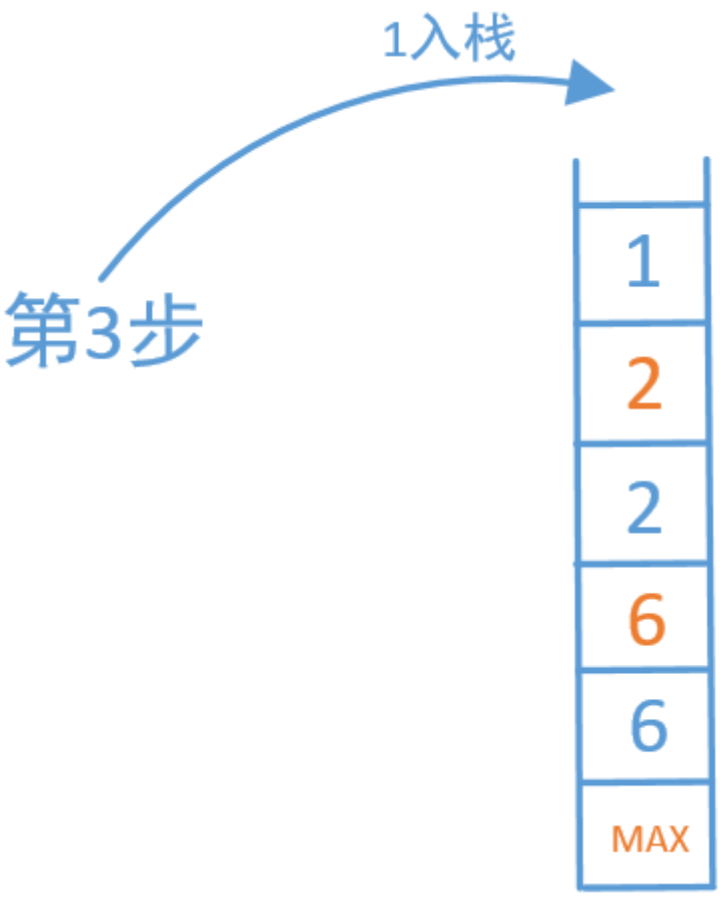
使用单个栈解决

也可以使用官方提供的栈，当压栈的值小于栈中最小值时，先把最小值入栈，然后再把需要压栈的值入栈，最后再更新栈中最小值。如果压栈的值大于栈中最小值的时候，直接压栈，这里就以[6，2，1，4]分别入栈来看一下

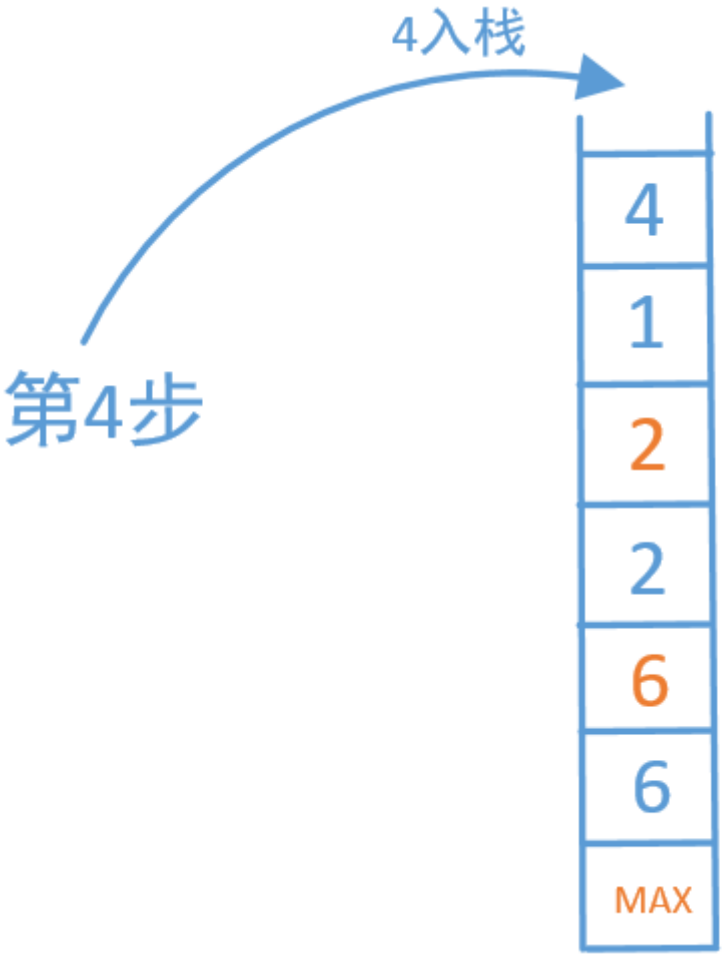
6,2,1,4


6入栈
第1步
6
MAX
栈中最小值6


2入栈
第2步
2
6
6
MAX
即将入栈的2比栈中最小值6小，把以前的最小值6压栈，再把2压栈，同时更新最小值为2



即将入栈的1比栈中最小值2小，把以前的最小值2压栈，再把1压栈，同时更新最小值为1



即将入栈的4比栈中最小值1大，直接压栈

这是压栈的过程，出栈的时候如果出栈的值等于最小值，说明最小值已经出栈了，要更新最小值，估计直接看代码会更明白一些

```java
class MinStack {//push方法可能会加入很多min
  int min = Integer.MAX_VALUE;
  Stack<Integer> stack = new Stack<>();

  public void push(int x) {
    //如果加入的值小于最小值，要更新最小值
    if (x <= min) {
      stack.push(min);
      min = x;
    }
    stack.push(x);
  }

  public void pop() {
    //如果把最小值出栈了，就更新最小值
    if (stack.pop() == min)
      min = stack.pop();
  }

  public int top() {
    return stack.peek();
  }

  public int min() {
    return min;
  }
}
```

这种方式虽然也能解决，但如果压栈的值一直递减的话，栈中会压入很多的min，实际上我们还可以在改一下，栈中压入的是需要压栈的值和最小值的差值，这样就不会压入min了，看下代码

```java
public class MinStack {
  long min;
  Stack<Long> stack = new Stack<>();

  public void push(int x) {
    if (stack.isEmpty()) {
      stack.push(0L);
      min = x;
    } else {
      //这里入栈的是入栈的值和最小值的差值，有可能为负，也有可能为正。
      stack.push(x - min);
      if (x < min)
        min = x;
    }
  }

  public void pop() {
    if (stack.isEmpty())
      return;
    long pop = stack.pop();
    //因为入栈的是差值，当出栈的为负数的时候，说明栈中最小值已经出栈了，
    //这里要重新更新最小值
    if (pop < 0)
      min -= pop;
  }

  public int top() {
    long top = stack.peek();
    if (top > 0) {
      //栈顶元素如果是正的，说明栈顶元素压栈的时候是比栈中最小值大的，根据
      //top=x - min，可以计算x=top+min
      return (int) (top + min);
    } else {
      //当栈顶元素是负数的时候，说明栈顶元素压栈的时候是比栈中最小值小的，
```

```
        //而压栈完之后他会更新最小值min，所以如果在使用上面公式肯定是不行
        //的。如果栈顶元素压栈的时候比最小值小，他会更新最小值，这个最小值
        //就是我们要压栈的值，所以这里直接返回min就行了。
        return (int) (min);
      }
    }

    public int min() {
      return (int) min;
    }
  }
```

使用双栈解决

这个代码比较简洁，就不在说了，直接看下代码

```
class MinStack {
  //栈1存放的是需要压栈的值
  Stack<Integer> stack1 = new Stack<>();
  //栈2存放的是最小值
  Stack<Integer> stack2 = new Stack<>();

  public void push(int x) {
    stack1.push(x);
    if (stack2.empty() || x <= min())
      stack2.push(x);
  }

  public void pop() {
    //如果出栈的值等于最小值，说明栈中的最小值
    //已经出栈了，因为stack2中的栈顶元素存放的
    //就是最小值，所以stack2栈顶元素也要出栈
    if (stack1.pop() == min())
      stack2.pop();
  }

  public int top() {
    return stack1.peek();
  }

  public int min() {
    return stack2.peek();
  }
}
```

问题分析

这道题解法比较多，不算太难，栈的实现也可以有多种方式。

往期推荐

- 416，剑指 Offer-用两个栈实现队列
- 399，从前序与中序遍历序列构造二叉树
- 397，双指针求接雨水问题

416，剑指 Offer-用两个栈实现队列

原创 山大王wld 数据结构和算法 7月30日

收录于话题
#剑指offer

27个 >



Everyone's the hero in their own story.

每个人都是自己故事里的英雄。

问题描述

用两个栈实现一个队列。队列的声明如下，请实现它的两个函数 appendTail 和 deleteHead ，分别完成在队列尾部插入整数和在队列头部删除整数的功能。(若队列中没有元素，deleteHead 操作返回 -1)

示例 1：

> **输入：**
> ["CQueue","appendTail","deleteHead","deleteHead"]
> [[],[3],[],[]]
> **输出：** [null,null,3,-1]

示例 2：

> **输入：**

> ["CQueue","deleteHead","appendTail","appendTail","deleteHead","deleteHead"]
> [[],[],[5],[2],[],[]]
> **输出：** [null,-1,null,null,5,2]

问题分析

做这题之前我们首先要明白一点就是，栈是先进后出的，队列是先进先出的。我们可以使用两个栈stackPop和stackPush，往队列中添加元素的时候直接把要添加的值压入到stackPush栈中。往队列中删除元素的时候如果stackPop中有元素我们就直接删除，如果没有元素，我们需要把stackPush中的元素全部出栈放到stackPop中，然后再删除stackPop中的元素。这样做的目的我们就可以保证stackPop中的元素永远都是比stackPush中的元素更老。

```
class CQueue {
    public Stack<Integer> stackPush;
    public Stack<Integer> stackPop;

    public CQueue() {
        stackPush = new Stack<>();
        stackPop = new Stack<>();
    }

    public void appendTail(int value) {
        stackPush.push(value);
    }

    public int deleteHead() {
        if (stackPop.isEmpty())
            while (!stackPush.isEmpty())
                stackPop.push(stackPush.pop());
        return stackPop.isEmpty() ? -1 : stackPop.pop();
    }
}
```

总结

用栈实现队列也是比较常见的一道题，类似的还有用队列实现栈。

往期推荐

- 414，剑指 Offer-重建二叉树
- 410，剑指 Offer-从尾到头打印链表
- 408，剑指 Offer-替换空格
- 404，剑指 Offer-数组中重复的数字

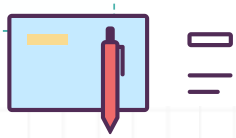
Manacher(马拉车)算法

原创 博哥 数据结构和算法 5月20日

收录于话题
#算法图文分析

161个 >

> It's great to be great, but it's greater to be human.
>
> 成为一个伟人很伟大，但是成为一个充满人性的人更伟大。



Manacher算法

Manacher于1975年发现了一种线性时间算法，可以在列出给定字符串中从任意位置开始的所有回文子串。同样的算法也可以在任意位置查找全部极大回文子串，并且时间复杂度是线性的。

之前在讲《517，最长回文子串的3种解决方式》的时候，在最后提到过Manacher算法，但是没有写，这里单独拿出来写。

我们先看一下回文串，回文串有两种形式，一种是奇数的比如"aba"，一种是偶数的比如"abba"。这里使用Manacher算法的时候，会在每个字符之间都会插入一个特殊字符，并且两边也会插入，这个特殊字符要保证不能是原字符串中的字符，这样无论原来字符串长度是奇数还是偶数，添加之后长度都会变成奇数。例如

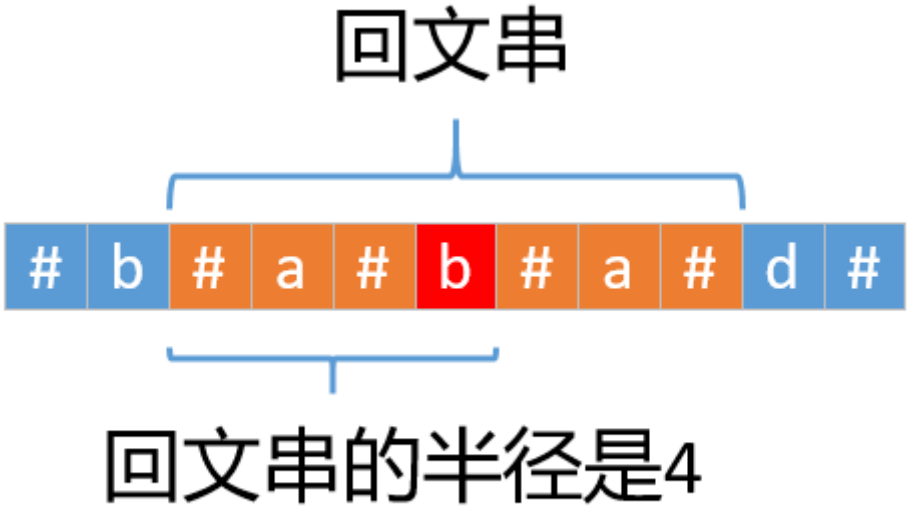
- "aba"-->"#a#b#a#"（长度是7）
- "abba"-->"#a#b#b#a#"（长度是9）

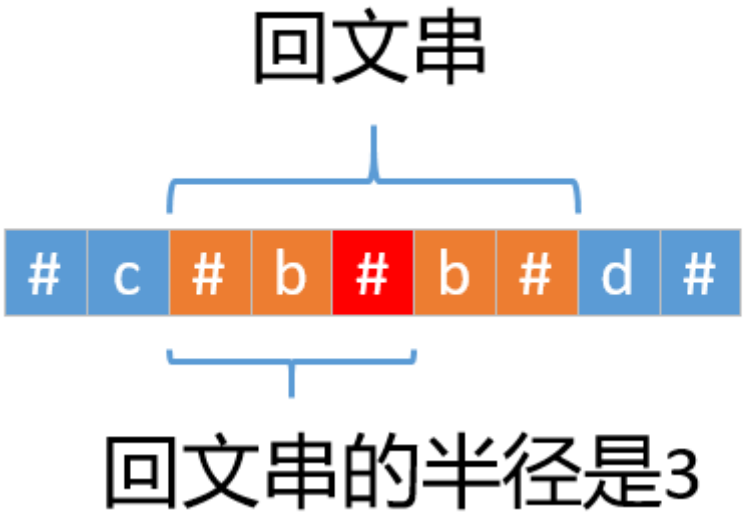
这里再来引用一个变量叫回文半径，通过添加特殊字符，原来字符串长度无论是奇数还是偶数最终都会变为奇数，因为特殊字符的引用，改变之后的字符串的所有回文子串长度一定都是奇数。并且回文子串的第一个和最后一个字符一定是你添加的那个特殊字符。其实很好证明

- 如果原来回文子串的长度是奇数，通过中间插入特殊字符，特殊字符的个数必定是偶数，在加上两边的特殊字符，长度必然是奇数

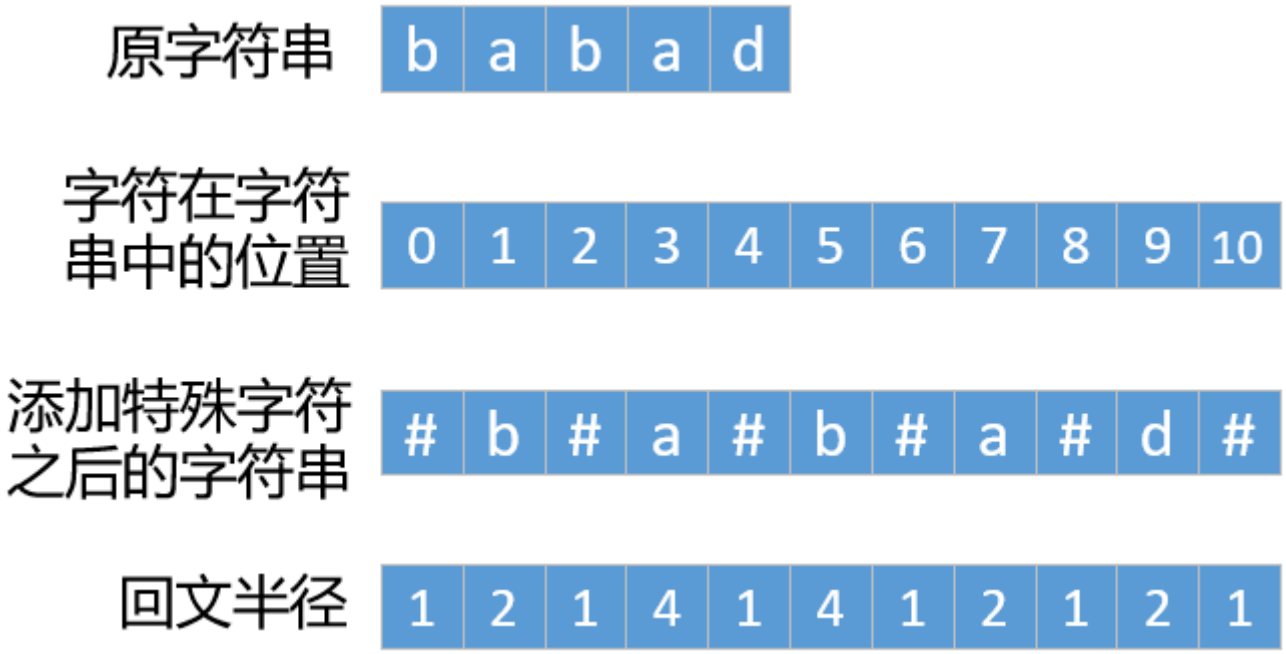
- 如果原来回文子串的长度是偶数，通过中间插入特殊字符，特殊字符的个数必定是奇数，在加上两边的特殊字符，长度必然是奇数

因为添加特殊字符之后所有回文子串的长度都是奇数，我们定义回文子串最中间的那个字符到回文子串最左边的长度叫回文半径，如下图所示。





我们来看个例子，比如字符串"babad"在添加特殊字符之后每个字符的回文半径

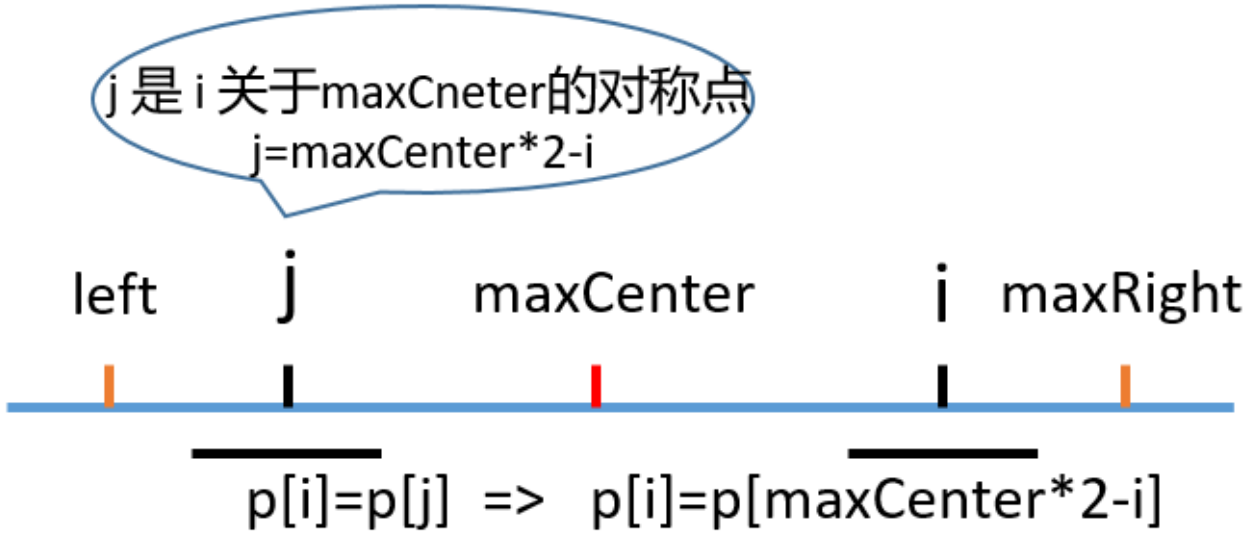


搞懂了这个我们再来看一下最长回文子串该怎么求。在第517题的时候我们讲过中心扩散法，我们会以每一个字符（中间会过滤掉重复的）为中心往两边扩散，如果以当前字符为中心往两边扩散计算完的时候，到下一个字符在往两边扩散的时候还要重新计算，那么有没有一种方法不用重新计算，而利用之前计算的结果呢，答案是肯定的。

假如以当前字符s[maxCenter]为回文中心的最大回文长度是从left到maxRight，如下图所示

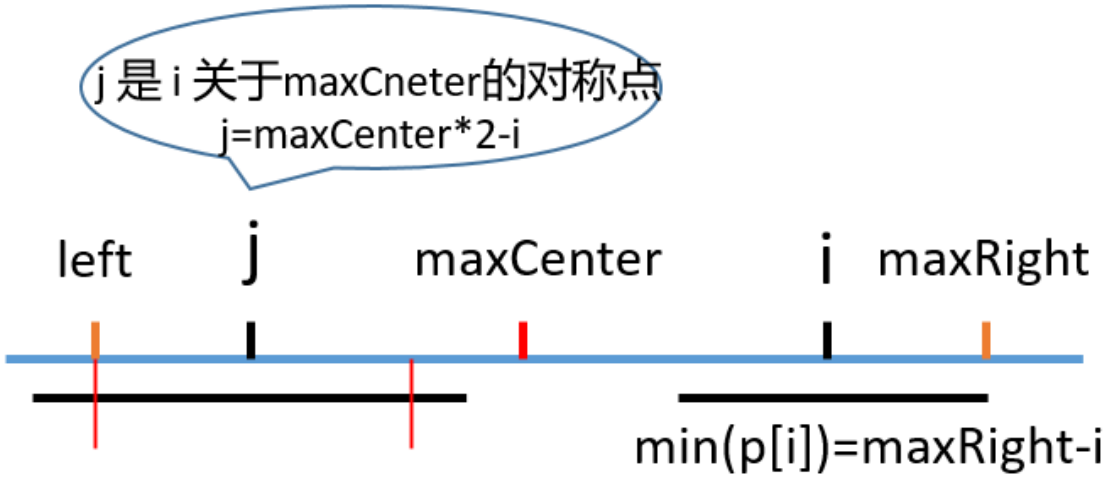
如果我们想求以字符s[i]为回文中心的最大回文长度，我们只需要找到i关于maxCenter的对称点j，看下j的回文长度，因为j已经计算过了。

1，如果i在maxRight的左边，并且j的最大回文长度左边没有到达left，根据对称性，i的最大回文长度就等于j的最大回文长度，如下图所示



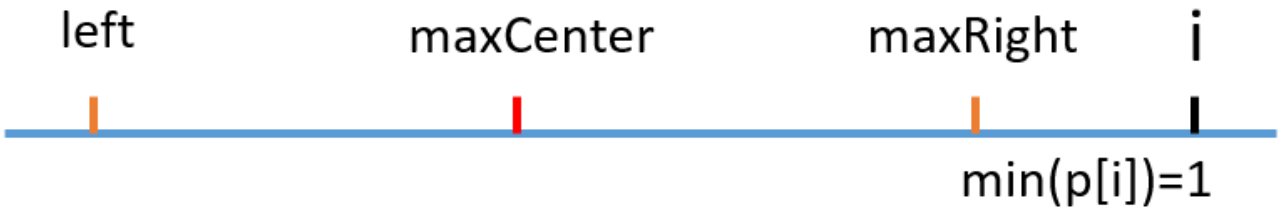
i 的对称点 j 的回文半径没有超出范围[left,maxRight]，因为 i 和 j 是对称的，所以 i 的回文半径等于 j 的回文半径，即p[i]=p[j]

2，如果i在maxRight的左边，并且j的最大回文长度左边到达或者超过left，根据对称性，i的最小回文长度等于j-left也等于maxRight-i，至于最大能有多大，还需要在继续判断，如下图所示



i 的对称点 j 的回文半径超出了left的左边界，超出的部分就没法在确定了，因为 i 和 j 是关于maxCenter对称的，所以我们可以确定 i 的回文半径最小值是maxRight-i，也是j-left，他俩的值是一样的，至于 i 的回文半径到底有多大还需要在继续判断

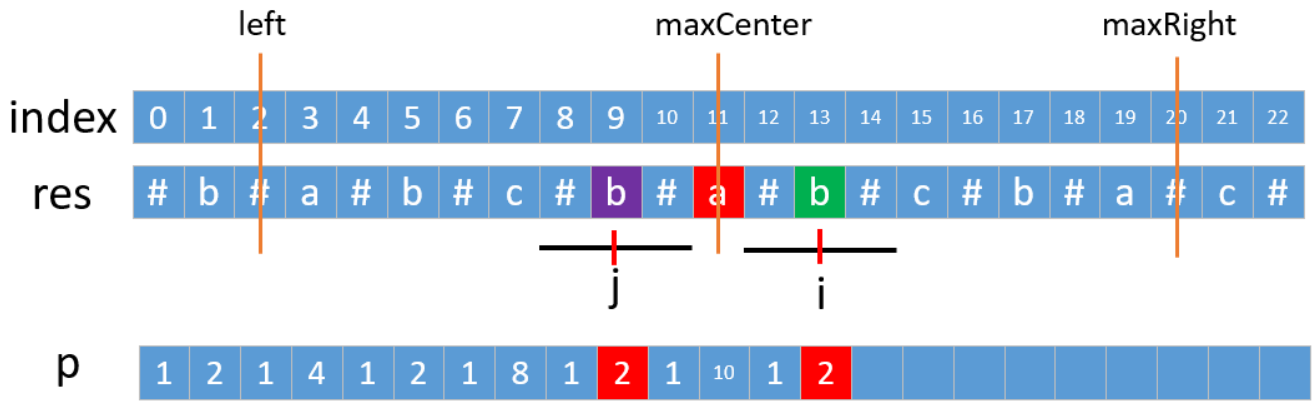
3，如果i在maxRight的右边，我们就没法利用之前计算的结果了，这个时候就需要一个个判断了，如下图所示



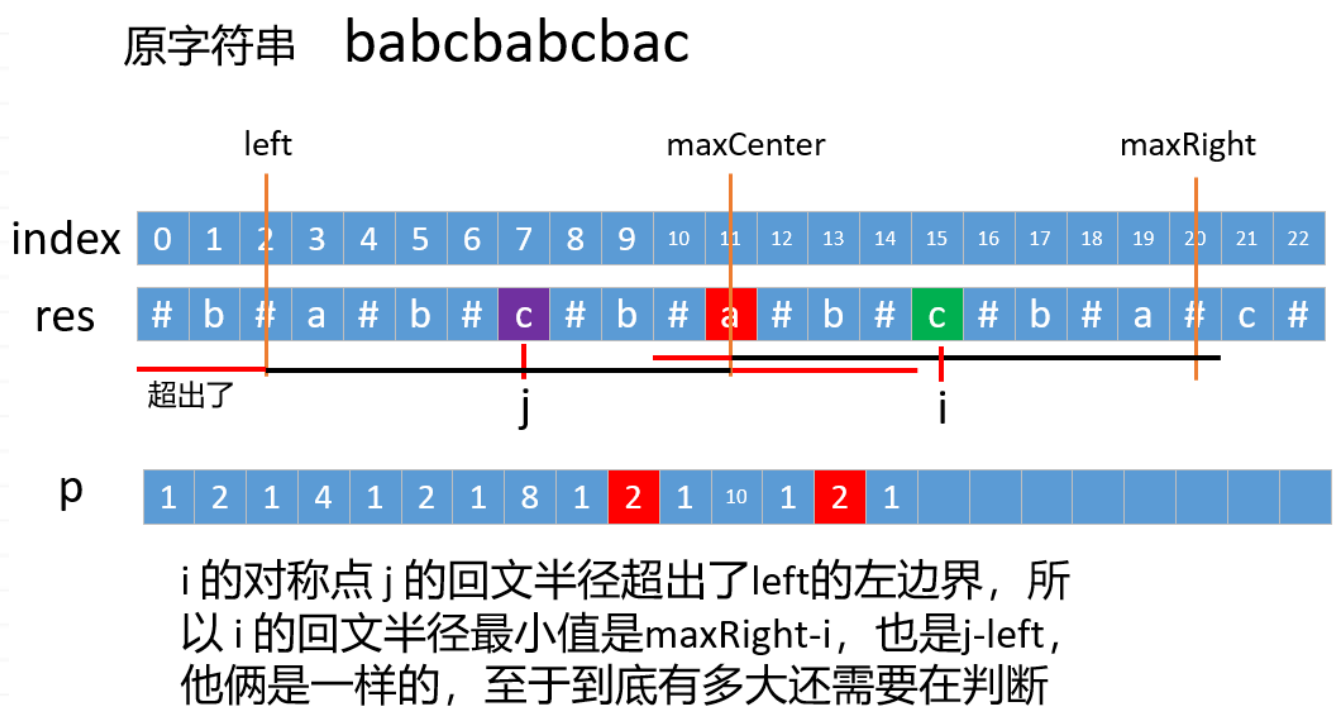
i 超出了范围[left,maxRight]，没法利用之前已知的数据了，这个时候就要一个个判断了。

如果还看不明白，我们来随便找个字符串"babcbabcbac"画个图来看下

原字符串 babcbabcbac



i 的对称点 j 的回文半径没有超出[left，maxRight]，所以p[i]=p[j]



i 的对称点 j 的回文半径超出了left的左边界，所以 i 的回文半径最小值是maxRight-i，也是j-left，他俩是一样的，至于到底有多大还需要在判断

代码如下，分三种情况判断

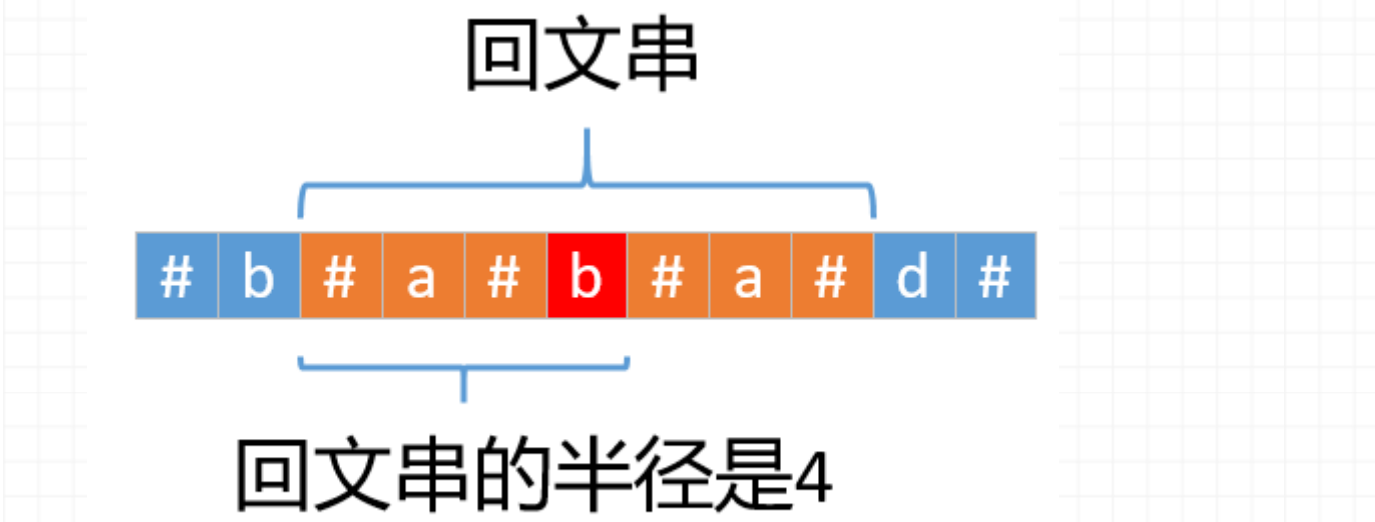
```
for (int i = 0; i < length; i++) {
  if (i < maxRight) {
    //情况一，i没有超出范围[left,maxRight]
    //2 * maxCenter - i其实就是j的位置，实际上是判断p[j]<maxRight - i
    if (p[2 * maxCenter - i] < maxRight - i) {
      //j的回文半径没有超出范围[left,maxRight]，直接让p[i]=p[j]即可
      p[i] = p[2 * maxCenter - i];
    } else {
      //情况二，j的回文半径已经超出了范围[left,maxRight]，我们可以确定p[i]的最小值
      //是maxRight - i，至于到底有多大，后面还需要在计算
      p[i] = maxRight - i;
      //继续计算
      while (i - p[i] >= 0 && i + p[i] < length && res[i - p[i]] == res[i + p[i]])
        p[i]++;
    }
  } else {
    //情况三，i超出了范围[left,maxRight]，就没法利用之前的已知数据，而是要一个个判断了
    p[i] = 1;
    //继续计算
    while (i - p[i] >= 0 && i + p[i] < length && res[i - p[i]] == res[i + p[i]])
      p[i]++;
  }
}
```

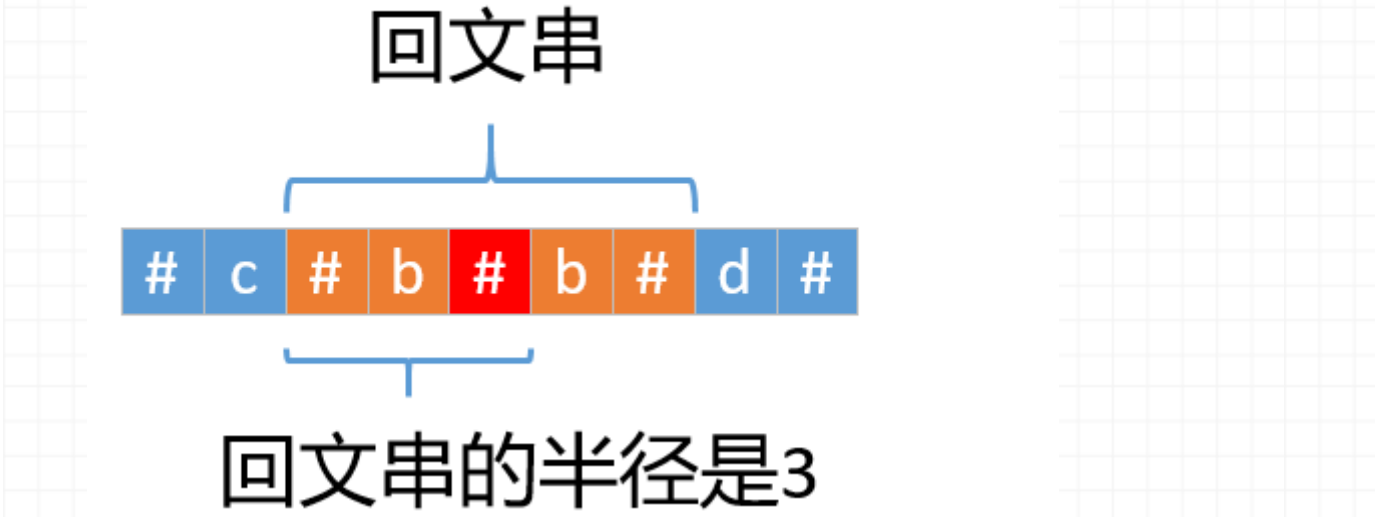
在来看下最终代码

```
public String longestPalindrome(String s) {
  int charLen = s.length();//源字符串的长度
  int length = charLen * 2 + 1;//添加特殊字符之后的长度
  char[] chars = s.toCharArray();//源字符串的字符数组
  char[] res = new char[length];//添加特殊字符的字符数组
  int index = 0;
  //添加特殊字符
  for (int i = 0; i < res.length; i++) {
    res[i] = (i % 2) == 0 ? '#' : chars[index++];
  }

  //新建p数组，p[i]表示以res[i]为中心的回文串半径
  int[] p = new int[length];
  //maxRight(某个回文串延伸到的最右边下标)
  //maxCenter(maxRight所属回文串中心下标),
  //resCenter（记录遍历过的最大回文串中心下标）
```

```
    //resLen（记录遍历过的最大回文半径）
    int maxRight = 0, maxCenter = 0, resCenter = 0, resLen = 0;
    //遍历字符数组res
    for (int i = 0; i < length; i++) {
      if (i < maxRight) {
        //情况一，i没有超出范围[left,maxRight]
        //2 * maxCenter - i其实就是j的位置，实际上是判断p[j]<maxRight - i
        if (p[2 * maxCenter - i] < maxRight - i) {
          //j的回文半径没有超出范围[left,maxRight]，直接让p[i]=p[j]即可
          p[i] = p[2 * maxCenter - i];
        } else {
          //情况二，j的回文半径已经超出了范围[left,maxRight]，我们可以确定p[i]的最小值
          //是maxRight - i，至于到底有多大，后面还需要在计算
          p[i] = maxRight - i;
          while (i - p[i] >= 0 && i + p[i] < length && res[i - p[i]] == res[i + p[i]])
            p[i]++;
        }
      } else {
        //情况三，i超出了范围[left,maxRight]，就没法利用之前的已知数据，而是要一个个判断了
        p[i] = 1;
        while (i - p[i] >= 0 && i + p[i] < length && res[i - p[i]] == res[i + p[i]])
          p[i]++;
      }
      //匹配完之后，如果右边界i + p[i]超过maxRight，那么就更新maxRight和maxCenter
      if (i + p[i] > maxRight) {
        maxRight = i + p[i];
        maxCenter = i;
      }
      //记录最长回文串的半径和中心位置
      if (p[i] > resLen) {
        resLen = p[i];
        resCenter = i;
      }
    }
    //计算最长回文串的长度和开始的位置
    resLen = resLen - 1;
    int start = (resCenter - resLen) >> 1;
    //截取最长回文子串
    return s.substring(start, start + resLen);
  }
```

上面都通过画图分析很好理解，可能稍微有点不好理解的是后面3行代码，resLen就是最大回文半径，resCenter就是最大回文子串（添加特殊字符之后的）中间的那个字符。我们可以根据下面这个图可以看到，原字符串中回文串的长度就是添加特殊字符之后的**回文半径-1**。





上面是分为3种情况来判断的，实际上我们还可以把上面3种情况合并

```
      //合并后的代码
      p[i] = maxRight > i ? Math.min(maxRight - i, p[2 * maxCenter - i]) : 1;
      //上面的语句只能确定i~maxRight的回文情况，至于maxRight之后的部分是否对称，
      //就只能一个个去匹配了，匹配的时候首先数组不能越界
      while (i - p[i] >= 0 && i + p[i] < length && res[i - p[i]] == res[i + p[i]])
        p[i]++;
```

我们来看下合并后的最终代码

```
// 返回最长回文串长度
public String longestPalindrome(String s) {
  int charLen = s.length();//源字符串的长度
  int length = charLen * 2 + 1;//添加特殊字符之后的长度
  char[] chars = s.toCharArray();//源字符串的字符数组
  char[] res = new char[length];//添加特殊字符的字符数组
  int index = 0;
  //添加特殊字符
  for (int i = 0; i < res.length; i++) {
    res[i] = (i % 2) == 0 ? '#' : chars[index++];
  }

  //新建p数组，p[i]表示以res[i]为中心的回文串半径
  int[] p = new int[length];
  //maxRight(某个回文串延伸到的最右边下标)
  //maxCenter(maxRight所属回文串中心下标),
  //resCenter（记录遍历过的最大回文串中心下标）
```

```
    //resLen（记录遍历过的最大回文半径）
    int maxRight = 0, maxCenter = 0, resCenter = 0, resLen = 0;
    //遍历字符数组res
    for (int i = 0; i < length; i++) {
      //合并后的代码
      p[i] = maxRight > i ? Math.min(maxRight - i, p[2 * maxCenter - i]) : 1;
      //上面的语句只能确定i~maxRight的回文情况，至于maxRight之后的部分是否对称，
      //就只能一个个去匹配了，匹配的时候首先数组不能越界
      while (i - p[i] >= 0 && i + p[i] < length && res[i - p[i]] == res[i + p[i]])
        p[i]++;
      //匹配完之后，如果右边界i + p[i]超过maxRight，那么就更新maxRight和maxCenter
      if (i + p[i] > maxRight) {
        maxRight = i + p[i];
        maxCenter = i;
      }
      //记录最长回文串的半径和中心位置
      if (p[i] > resLen) {
        resLen = p[i];
        resCenter = i;
      }
    }
    //计算最长回文串的长度和开始的位置
    resLen = resLen - 1;
    int start = (resCenter - resLen) >> 1;
    //截取最长回文子串
    return s.substring(start, start + resLen);
  }
```

总结

Manacher算法，很多人习惯称它为马拉车算法，是一道非常经典的算法，搞懂他的原理，其实他的解题思路并不难。

往期推荐

- 540，动态规划和中心扩散法解回文子串
- 529，动态规划解最长回文子序列
- 517，最长回文子串的3种解决方式
- 497，双指针验证回文串

426，什么是递归，通过这篇文章，让你彻底搞懂递归

原创 山大王wld 数据结构和算法 8月10日

收录于话题
#算法图文分析

95个 >



Beauty begins the moment you decide to be yourself.

美丽开始于你决定做自己的那一刻。

啥叫递归

tips：文章有点长，可以慢慢看，如果来不及看，也可以先收藏以后有时间在看。

聊递归之前先看一下什么叫递归。

递归，就是在运行的过程中调用自己。

构成递归需具备的条件：

1. 子问题须与原始问题为同样的事，且更为简单；
2. 不能无限制地调用本身，须有个出口，化简为非递归状况处理。

递归语言例子

我们用2个故事来阐述一下什么叫递归。

1，从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？“从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？‘从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？……’”

2，大雄在房里，用时光电视看着从前的情况。电视画面中的那个时候，他正在房里，用时光电视，看着从前的情况。电视画面中的电视画面的那个时候，他正在房里，用时光电视，看着从前的情况……

递归模板

我们知道递归必须具备两个条件，一个是调用自己，一个是有终止条件。这两个条件必须同时具备，且一个都不能少。并且终止条件必须是在递归最开始的地方，也就是下面这样

```
public void recursion(参数0) {
    if (终止条件) {
        return;
    }
    recursion(参数1);
}
```

不能把终止条件写在递归结束的位置，下面这种写法是错误的

```
public void recursion(参数0) {
    recursion(参数1);
    if (终止条件) {
        return;
    }
}
```

如果这样的话，递归永远退不出来了，就会出现堆栈溢出异常(StackOverflowError)。

但实际上递归可能调用自己不止一次，并且很多递归在调用之前或调用之后都会有一些逻辑上的处理，比如下面这样。

```
public void recursion(参数0) {
    if (终止条件) {
```

```
            return;
        }

        可能有一些逻辑运算
        recursion(参数1)
        可能有一些逻辑运算
        recursion(参数2)
                ……
        recursion(参数n)
        可能有一些逻辑运算
    }
```

实例分析

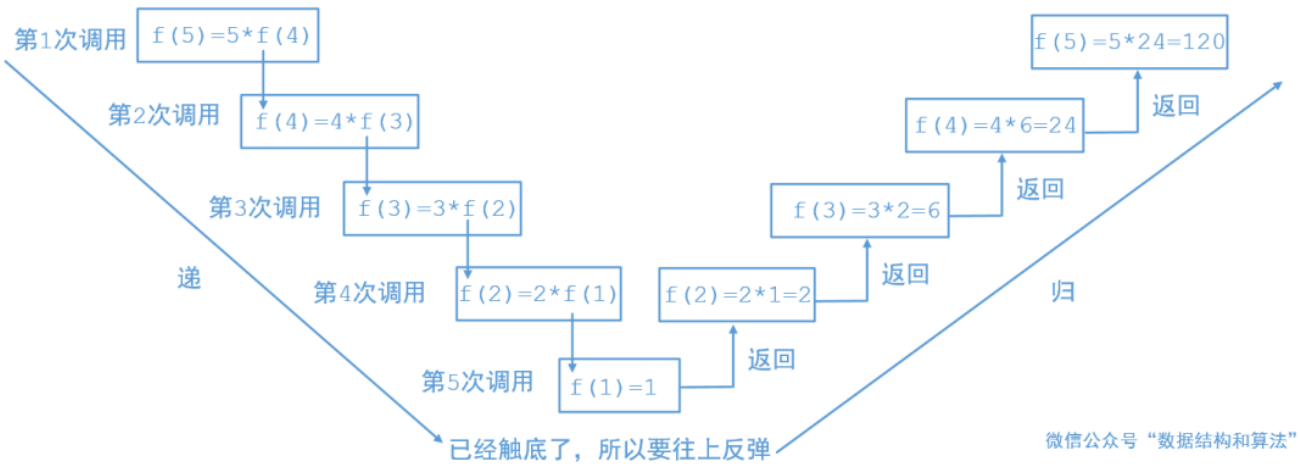
我对递归的理解是先往下一层层传递，当碰到终止条件的时候会反弹，最终会反弹到调用处。下面我们就以5个最常见的示例来分析下

1，阶乘

我们先来看一个最简单的递归调用-阶乘，代码如下

```java
public int recursion(int n) {
  if (n == 1)
    return 1;
  return n * recursion(n - 1);
}
```

这个递归在熟悉不过了，第2-3行是终止条件，第4行是调用自己。我们就用n等于5的时候来画个图看一下递归究竟是怎么调用的



如果看不清，图片可点击放大。

这种递归还是很简单的，我们求f(5)的时候，只需要求出f(4)即可，如果求f(4)我们要求出f(3)……，一层一层的调用，当n=1的时候，我们直接返回1，然后再一层一层的返回，直到返回f(5)为止。

递归的目的是把一个大的问题细分为更小的子问题，我们只需要知道递归函数的功能即可，不要把递归一层一层的拆开来想，如果同时调用多次的话这样你很可能会陷入循环而出不来。比如上面的题中要求f(5)，我们只需要计算f(4)即可，即f(5)=5*f(4)；至于f(4)是怎么计算的，我们就不要管了。因为我们知道f(n)中的n可以代表任何正整数，我们只需要传入4就可以计算f(4)。

2，斐波那契数列

我们再来看另一道经典的递归题，就是斐波那契数列，数列的前几项如下所示

[1，1，2，3，5，8，13……]

我们参照递归的模板来写下，首先终止条件是当n等于1或者2的时候返回1，也就是数列的前两个值是1，代码如下

```
1   public int fibonacci(int n) {
2     if (n == 1 || n == 2)
3       return 1;
4     这里是递归调用;
5   }
```

递归的两个条件，一个是终止条件，我们找到了。还一个是调用自己，我们知道斐波那契数列当前的值是前两个值的和，也就是

fibonacci(n) =fibonacci(n - 1) + fibonacci(n - 2)

所以代码很容易就写出来了

```
1   //1,1,2,3,5,8,13……
2   public int fibonacci(int n) {
3     if (n == 1 || n == 2)
4       return 1;
5     return fibonacci(n - 1) + fibonacci(n - 2);
6   }
```

3，汉诺塔

通过前面两个示例的分析，我们对递归有一个大概的了解，下面我们再来看另一个示例-汉诺塔，这个其实前面讲过，有兴趣的可以看下362，汉诺塔



汉诺塔的原理这里再简单提一下，就是有3根柱子A，B，C。A柱子上由上至下依次由小至大排列的圆盘。把A柱子上的圆盘借B柱子全部移动到C柱子上，并且移动的过程始终是小的圆盘在上，大的在下。我们还是用递归的方式来解这道题，先来定义一个函数

public void hanoi(int n, char A, char B, char C)

他表示的是把n个圆盘从A借助B成功的移动到C。

我们先来回顾一下递归的条件，一个是终止条件，一个是调用自己。我们先来看下递归的终止条件就是当n等于1的时候，也就是A柱子上只有一个圆盘的时候，我们直接把A柱子上的圆盘移动到C柱子上即可。

```
//表示的是把n个圆盘借助柱子B成功的从A移动到C
public static void hanoi(int n, char A, char B, char C) {
  if (n == 1) {
    //如果只有一个，直接从A移动到C即可
    System.out.println("从" + A + "移动到" + C);
    return;
  }
  这里是递归调用
}
```

再来看一下递归调用，如果n不等于1，我们要分3步，

1，先把n-1个圆盘从A借助C成功的移动到B

2，然后再把第n个圆盘从A移动到C

3，最后再把n-1个圆盘从B借助A成功的移动到C。

那代码该怎么写呢，我们知道函数

hanoi(n, 'A', 'B', 'C')表示的是把n个圆盘从A借助B成功的移动到C

所以hanoi(n-1, 'A', 'C', 'B')就表示的是把n-1个圆盘从A借助C成功的移动到B

hanoi(n-1, 'B', 'A', 'C')就表示的是把n-1个圆盘从B借助A成功的移动到C

所以上面3步如果用代码就可以这样来表示

1，hanoi(n-1, 'A', 'C', 'B')

2，System.out.println("从" + A + "移动到" + C);

3，hanoi(n-1, 'B', 'A', 'C')

所以最终完整代码如下

```
//表示的是把n个圆盘借助柱子B成功的从A移动到C
public static void hanoi(int n, char A, char B, char C) {
  if (n == 1) {
    //如果只有一个，直接从A移动到C即可
    System.out.println("从" + A + "移动到" + C);
    return;
  }
  //表示先把n-1个圆盘成功从A移动到B
  hanoi(n - 1, A, C, B);
  //把第n个圆盘从A移动到C
  System.out.println("从" + A + "移动到" + C);
  //表示把n-1个圆盘再成功从B移动到C
  hanoi(n - 1, B, A, C);
}
```

通过上面的分析，是不是感觉递归很简单。所以我们写递归的时候完全可以套用上面的模板，先写出终止条件，然后在写递归的逻辑调用。还有一点非常重要，就是一定要明白递归函数中每个参数的含义，这样在逻辑处理和函数调用的时候才能得心应手，函数的调用我们一定不要去一步步拆开去想，这样很有可能你会奔溃的。

4，二叉树的遍历

再来看最后一个常见的示例就是二叉树的遍历，在前面也讲过，如果有兴趣的话可以看下373，数据结构-6,树，我们主要来看一下二叉树的前中后3种遍历方式，

1，先看一下前序遍历（根节点最开始），他的顺序是

根节点→左子树→右子树

我们来套用模板看一下

```
public void preOrder(TreeNode node) {
  if (终止条件)//(必须要有)
    return;
  逻辑处理//（不是必须的）
  递归调用//(必须要有)
}
```

终止条件是node等于空，**逻辑处理**这块直接打印当前节点的值即可，**递归调用**是先打印左子树在打印右子树，我们来看下

```
public static void preOrder(TreeNode node) {
  if (node == null)
    return;
  System.out.printf(node.val + "");
  preOrder(node.left);
  preOrder(node.right);
}
```

中序遍历和后续遍历直接看下

2，中序遍历（根节点在中间）

左子树→**根节点**→右子树

```
public static void inOrder(TreeNode node) {
  if (node == null)
    return;
  inOrder(node.left);
  System.out.println(node.val);
  inOrder(node.right);
}
```

3，后序遍历（根节点在最后）

左子树→右子树→**根节点**

```
public static void postOrder(TreeNode tree) {
  if (tree == null)
    return;
  postOrder(tree.left);
  postOrder(tree.right);
  System.out.println(tree.val);
}
```

5，链表的逆序打印

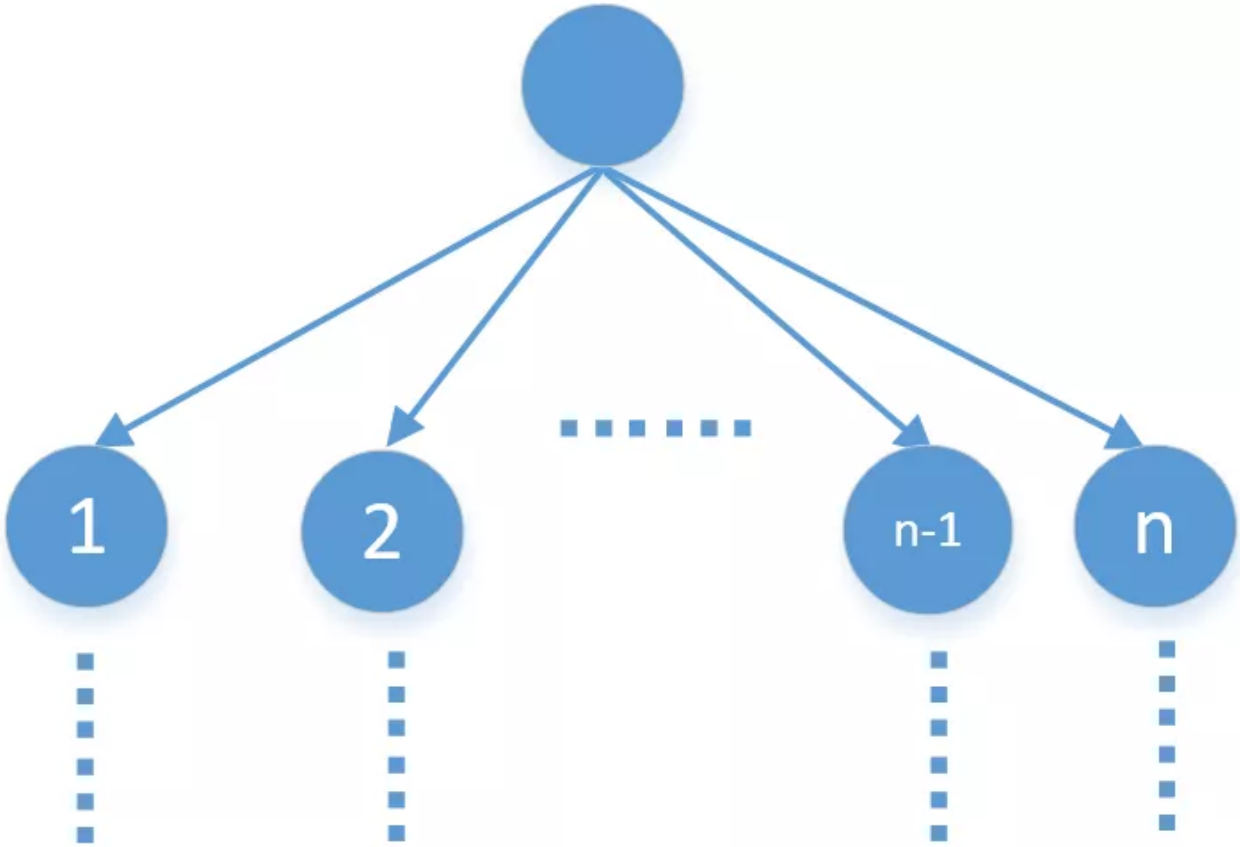
这个就不在说了，直接看下

```
public void printRevers(ListNode root) {
  //（终止条件）
  if (root == null)
    return;
  //（递归调用）先打印下一个
  printRevers(root.next);
  //（逻辑处理）把后面的都打印完了在打印当前节点
```

```
8       System.out.println(root.val);
9   }
```

分支污染问题

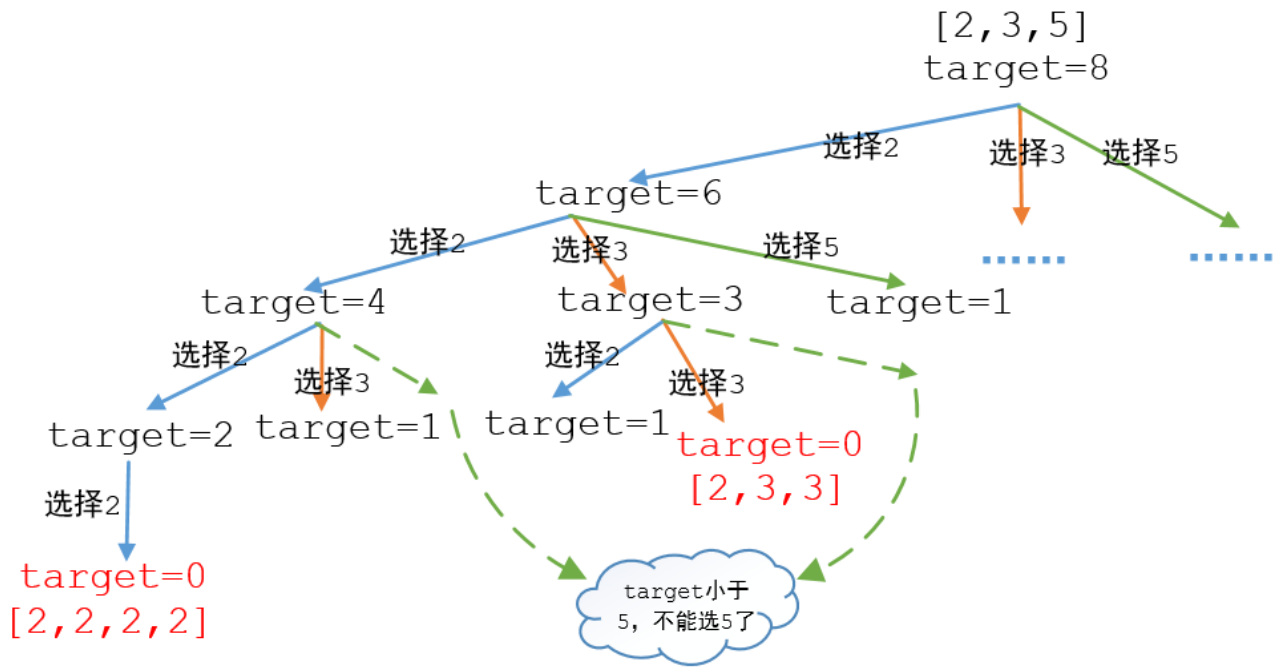
通过上面的分析，我们对递归有了更深一层的认识。但总觉得还少了点什么，其实递归我们还可以通过另一种方式来认识他，就是n叉树。在递归中如果只调用自己一次，我们可以把它想象为是一棵一叉树（这是我自己想的，我们可以认为只有一个子节点的树），如果调用自己2次，我们可以把它想象为一棵二叉树，如果调用自己n次，我们可以把它想象为一棵n叉树……。就像下面这样，当到达叶子节点的时候开始往回反弹。



递归的时候如果处理不当可能会出现分支污染导致结果错误。为什么会出现这种情况，我先来解释一下，因为除了基本类型是值传递以外，其他类型基本上很多都是引用传递。看一下上面的图，比如我开始调用的时候传入一个list对象，在调用第一个分支之后list中的数据修改了，那么后面的所有分支都能感知到，实际上也就是对后面的分支造成了污染。

我们先来看一个例子吧

给定一个数组nums=[2，3，5]和一个固定的值target=8。找出数组sums中所有可以使数字和为target的组合。先来画个图看一下



图中红色的表示的是选择成功的组合，这里只画了选择2的分支，由于图太大，所以选择3和选择5的分支没画。在仔细一看这不就是一棵3叉树吗，OK，我们来使用递归的方式，先来看一下函数的定义

```
private void combinationSum(List<Integer> cur, int sums[], int target) {

}
```

在把递归的模板拿出来

```
private void combinationSum(List<Integer> cur, int sums[], int target) {
  if (终止条件) {
    return;
  }
  //逻辑处理

  //因为是3叉树，所以这里要调用3次
  //递归调用
  //递归调用
  //递归调用

  //逻辑处理
}
```

这种解法灵活性不是很高，如果nums的长度是3，我们3次递归调用，如果nums的长度是n，那么我们就要n次调用……。所以我们可以直接写成for循环的形式，也就是下面这样

```
private void combinationSum(List<Integer> cur, int sums[], int target) {
  //终止条件必须要有
  if (终止条件) {
    return;
  }
  //逻辑处理(可有可无，是情况而定)
  for (int i = 0; i < sums.length; i++) {
    //逻辑处理(可有可无，是情况而定)
    //递归调用(递归调用必须要有)
    //逻辑处理(可有可无，是情况而定)
  }
  //逻辑处理(可有可无，是情况而定)
}
```

下面我们再来一步一步看

1，终止条件是什么？

当target等于0的时候，说明我们找到了一组组合，我们就把他打印出来，所以终止条件很容易写，代码如下

```
    if (target == 0) {
      System.out.println(Arrays.toString(cur.toArray()));
      return;
    }
```

2，逻辑处理和递归调用

我们一个个往下选的时候如果要选的值比target大，我们就不要选了，如果不比target大，就把他加入到list中，表示我们选了他，如果选了他之后在递归调用的时候target值就要减去选择的值，代码如下

```
      //逻辑处理
      //如果当前值大于target我们就不要选了
      if (target < sums[i])
        continue;
      //否则我们就把他加入到集合中
      cur.add(sums[i]);
      //递归调用
      combinationSum(cur, sums, target - sums[i]);
```

终止条件和递归调用都已经写出来了，感觉代码是不是很简单，我们再来把它组合起来看下完整代码

```
  private void combinationSum(List<Integer> cur, int sums[], int target) {
    //终止条件必须要有
    if (target == 0) {
      System.out.println(Arrays.toString(cur.toArray()));
      return;
    }
    for (int i = 0; i < sums.length; i++) {
      //逻辑处理
      //如果当前值大于target我们就不要选了
      if (target < sums[i])
        continue;
      //否则我们就把他加入到集合中
      cur.add(sums[i]);
      //递归调用
      combinationSum(cur, sums, target - sums[i]);
    }
```

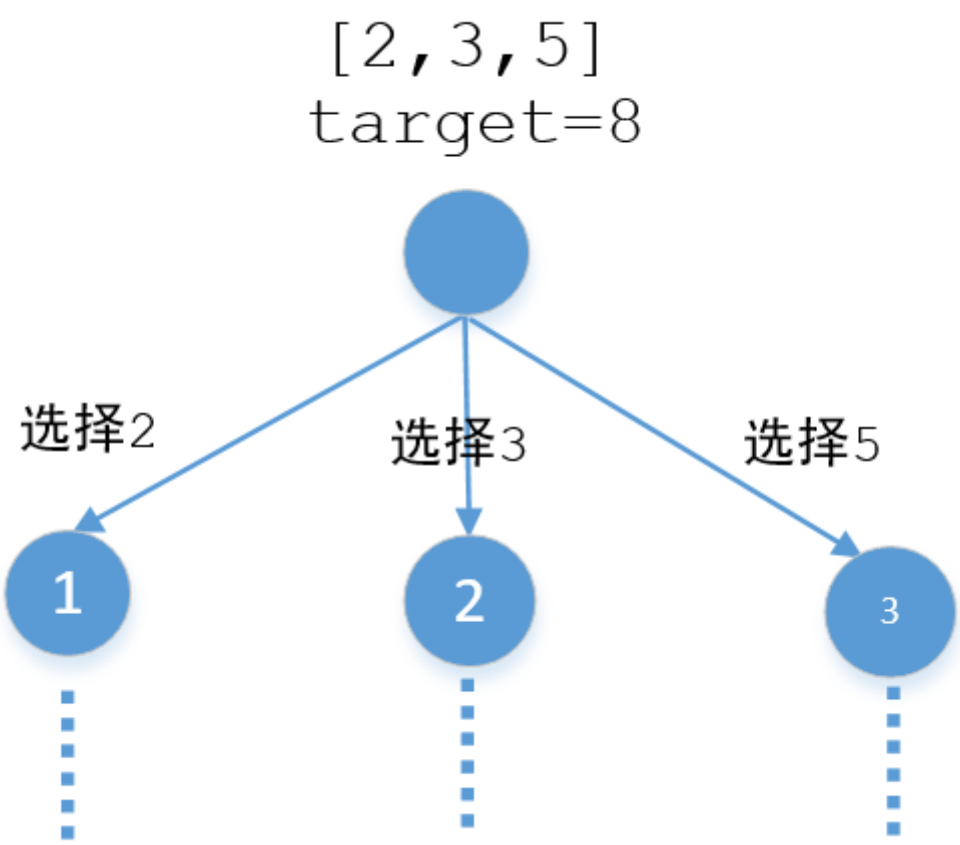
我们还用上面的数据打印测试一下

```
  public static void main(String[] args) {
    new Recursion().combinationSum(new ArrayList<>(), new int[]{2, 3, 5}, 8);
  }
```

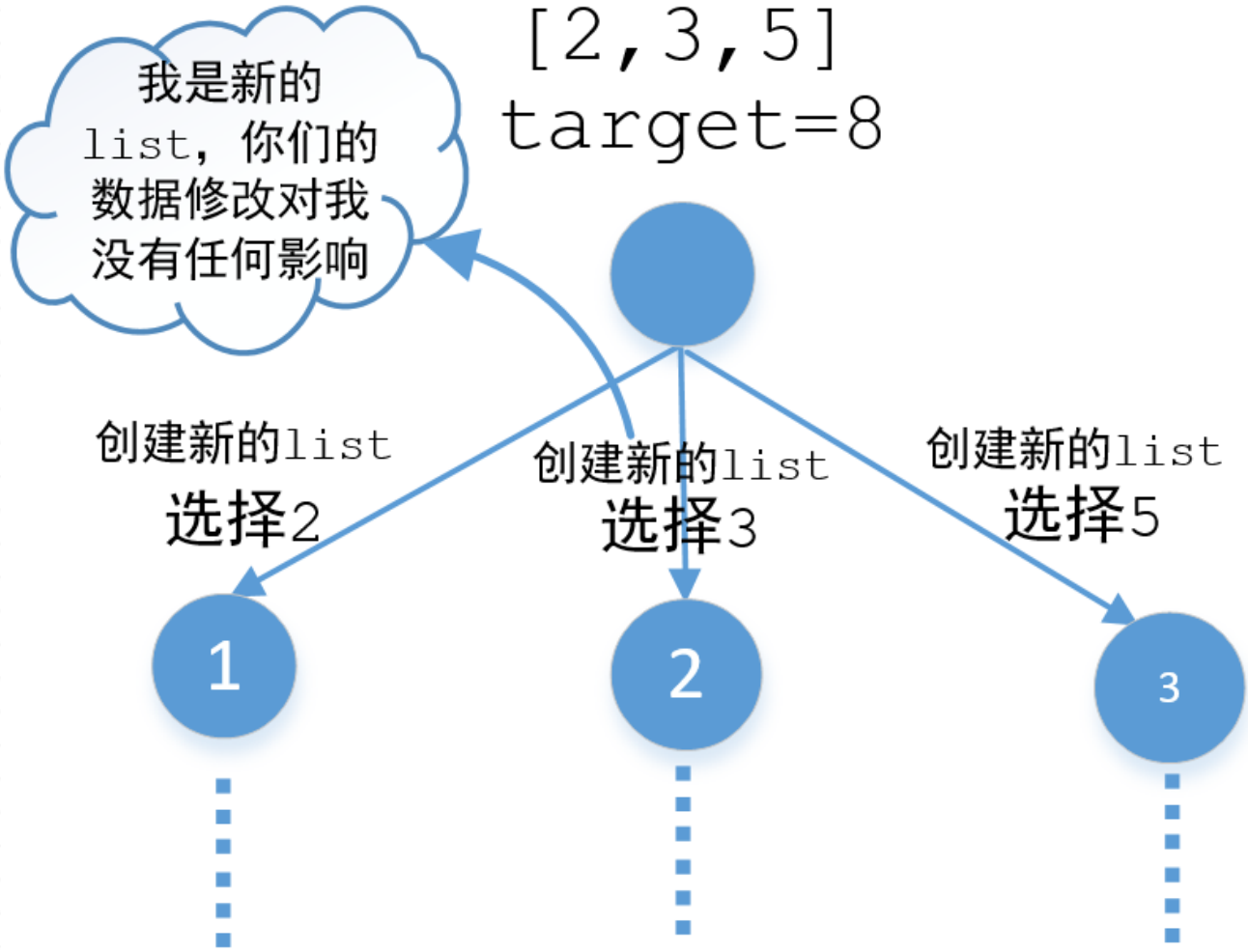
运行结果如下

```
[2, 2, 2, 2]
[2, 2, 2, 3, 3, 2, 3]
[2, 2, 2, 3, 3, 2, 3, 5, 3, 2, 2, 3]
[2, 2, 2, 3, 3, 2, 3, 5, 3, 2, 2, 3, 3, 2]
[2, 2, 2, 3, 3, 2, 3, 5, 3, 2, 2, 3, 3, 2, 5]
[2, 2, 2, 3, 3, 2, 3, 5, 3, 2, 2, 3, 3, 2, 5, 5, 2, 3]
```

是不是很意外，我们思路并没有出错，结果为什么不对呢，其实这就是典型的分支污染，我们再来看一下图



当我们选择2的时候是一个分支，当我们选择3的时候又是另外一个分支，这两个分支的数据应该是互不干涉的，但实际上当我们沿着选择2的分支走下去的时候list中会携带选择2的那个分支的数据，当我们再选择3的那个分支的时候这些数据还依然存在list中，所以对选择3的那个分支造成了污染。有一种解决方式就是每个分支都创建一个新的list，也就是下面这样，这样任何一个分支的修改都不会影响到其他分支。



再来看下代码

```
1   private void combinationSum(List<Integer> cur, int sums[], int target) {
2     //终止条件必须要有
3     if (target == 0) {
4       System.out.println(Arrays.toString(cur.toArray()));
5       return;
6     }
7     for (int i = 0; i < sums.length; i++) {
8       //逻辑处理
9       //如果当前值大于target我们就不要选了
10      if (target < sums[i])
11        continue;
12      //由于List是引用传递，所以这里要重新创建一个
13      List<Integer> list = new ArrayList<>(cur);
14      //把数据加入到集合中
15      list.add(sums[i]);
16      //递归调用
17      combinationSum(list, sums, target - sums[i]);
18    }
19  }
```

我们看到第13行是重新创建了一个list。再来打印一下看下结果，结果完全正确，每一组数据的和都是8

```
[2, 2, 2, 2]
[2, 3, 3]
[3, 2, 3]
[3, 3, 2]
[3, 5]
[5, 3]
```

上面我们每一个分支都创建了一个新的list，所以任何分支修改都只会对当前分支有影响，不会影响到其他分支，也算是一种解决方式。但每次都重新创建数据，运行效率很差。我们知道当执行完分支1的时候，list中会携带分支1的数据，当执行分支2的时候，实际上我们是不需要分支1的数据的，所以有一种方式就是从分支1执行到分支2的时候要把分支1的数据给删除，这就是大家经常提到的**回溯算法**，我们来看下

```java
private void combinationSum(List<Integer> cur, int sums[], int target) {
  //终止条件必须要有
  if (target == 0) {
    System.out.println(Arrays.toString(cur.toArray()));
    return;
  }
  for (int i = 0; i < sums.length; i++) {
    //逻辑处理
    //如果当前值大于target我们就不要选了
    if (target < sums[i])
      continue;
    //把数据sums[i]加入到集合中，然后参与下一轮的递归
    cur.add(sums[i]);
    //递归调用
    combinationSum(cur, sums, target - sums[i]);
    //sums[i]这个数据你用完了吧，我要把它删了
    cur.remove(cur.size() - 1);
  }
}
```

我们再来看一下打印结果，完全正确

```
[2, 2, 2, 2]
[2, 3, 3]
[3, 2, 3]
[3, 3, 2]
[3, 5]
[5, 3]
```

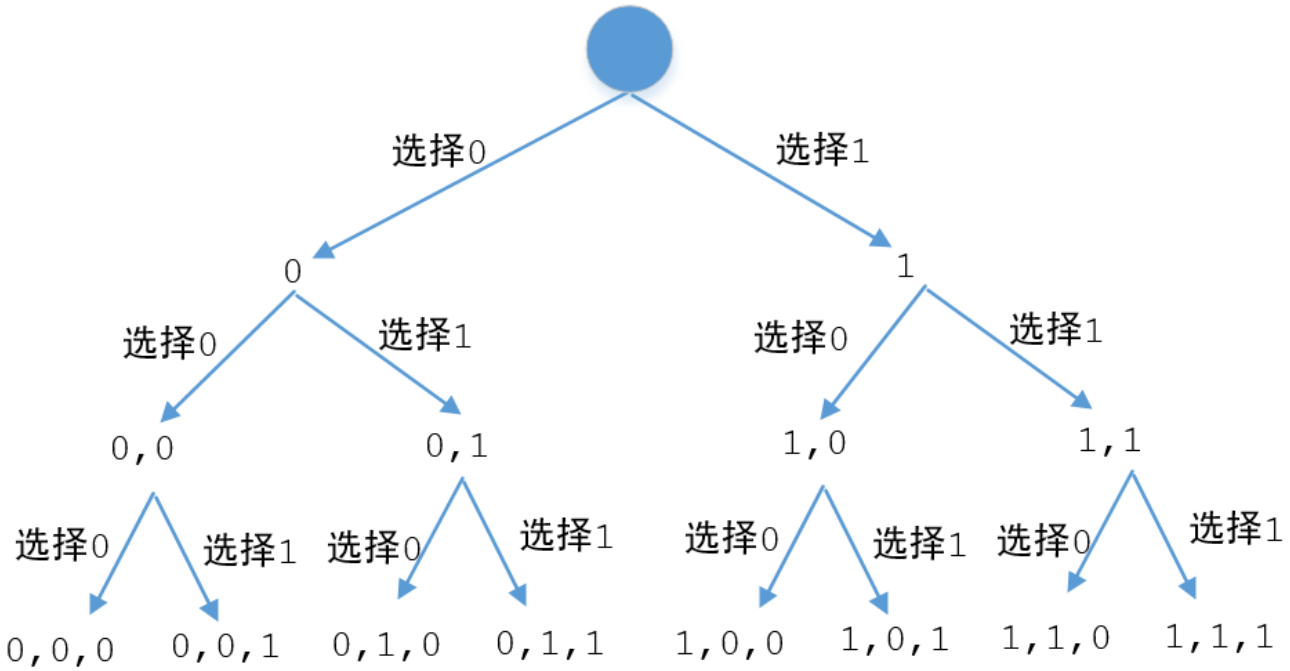
递归分支污染对结果的影响

分支污染一般会对结果造成致命错误，但也不是绝对的，我们再来看个例子。生成一个2^n长的数组，数组的值从0到(2^n)-1，比如n是3，那么要生成

```
[0, 0, 0]
[0, 0, 1]
[0, 1, 0]
[0, 1, 1]
[1, 0, 0]
[1, 0, 1]
```

```
[1, 1, 0]
[1, 1, 1]
```

我们先来画个图看一下



这不就是个二叉树吗，对于递归前面已经讲的很多了，我们来直接看代码

```
private void binary(int[] array, int index) {
  if (index == array.length) {
    System.out.println(Arrays.toString(array));
  } else {
    int temp = array[index];
    array[index] = 0;
    binary(array, index + 1);
    array[index] = 1;
    binary(array, index + 1);
    array[index] = temp;
  }
}
```

上面代码很好理解，首先是终止条件，然后是递归调用，在调用之前会把array[index]的值保存下来，最后再还原。我们来测试一下

```
new Recursion().binary(new int[]{0, 0, 0}, 0);
```

看下打印结果

```
[0, 0, 0]
[0, 0, 1]
[0, 1, 0]
[0, 1, 1]
[1, 0, 0]
[1, 0, 1]
[1, 1, 0]
[1, 1, 1]
```

结果完全正确，我们再来改一下代码

```
1   private void binary(int[] array, int index) {
2     if (index == array.length) {
3       System.out.println(Arrays.toString(array));
4     } else {
5       array[index] = 0;
6       binary(array, index + 1);
7       array[index] = 1;
8       binary(array, index + 1);
9     }
10  }
```

再来看一下打印结果

```
[0, 0, 0]
[0, 0, 1]
[0, 1, 0]
[0, 1, 1]
[1, 0, 0]
[1, 0, 1]
[1, 1, 0]
[1, 1, 1]
```

和上面结果一模一样，开始的时候我们没有把array[index]的值保存下来，最后也没有对他进行复原，但结果丝毫不差。原因就在上面代码第5行array[index]=0，这是因为，上一分支执行的时候即使对array[index]造成了污染，在下一分支又会对他进行重新修改。即使你把它改为任何数字也都不会影响到最终结果，比如我们在上一分支执行完了时候我们把它改为100，你在试试

```
1   private void binary(int[] array, int index) {
2     if (index == array.length) {
3       System.out.println(Arrays.toString(array));
4     } else {
5       array[index] = 0;
6       binary(array, index + 1);
7       array[index] = 1;
8       binary(array, index + 1);
9       //注意，这里改成100了
10      array[index] = 100;
11    }
12  }
```

我们看到第10行，把array[index]改为100了，最终打印结果也是不会变的，所以这种分支污染并不会造成最终的结果错误。

总结

对递归的理解，看完这篇文章应该没有什么疑问了，记住上面模板，其实代码很好写的，后面也会再写一些关于递归的算法题的，让你彻底搞懂递归。

往期推荐

394，经典的八皇后问题和N皇后问题

原创 山大王wld 数据结构和算法 7月4日

收录于话题
#算法图文分析

96个 >



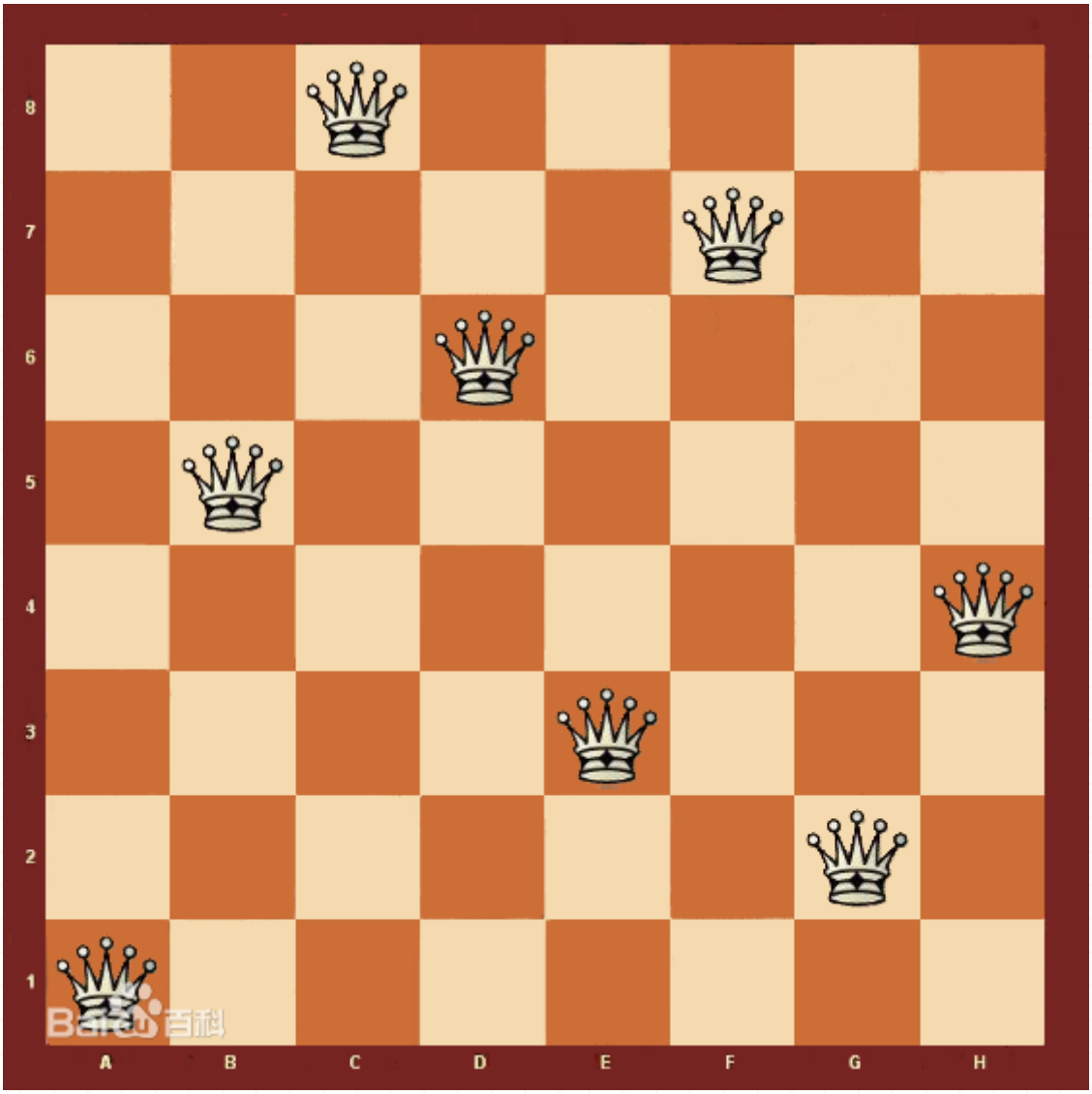
Accept the things you cannot change. Have the courage to change the things you can.

接受那些你也无法改变的事，而能够改变的事则要勇于行动。

八皇后的来源

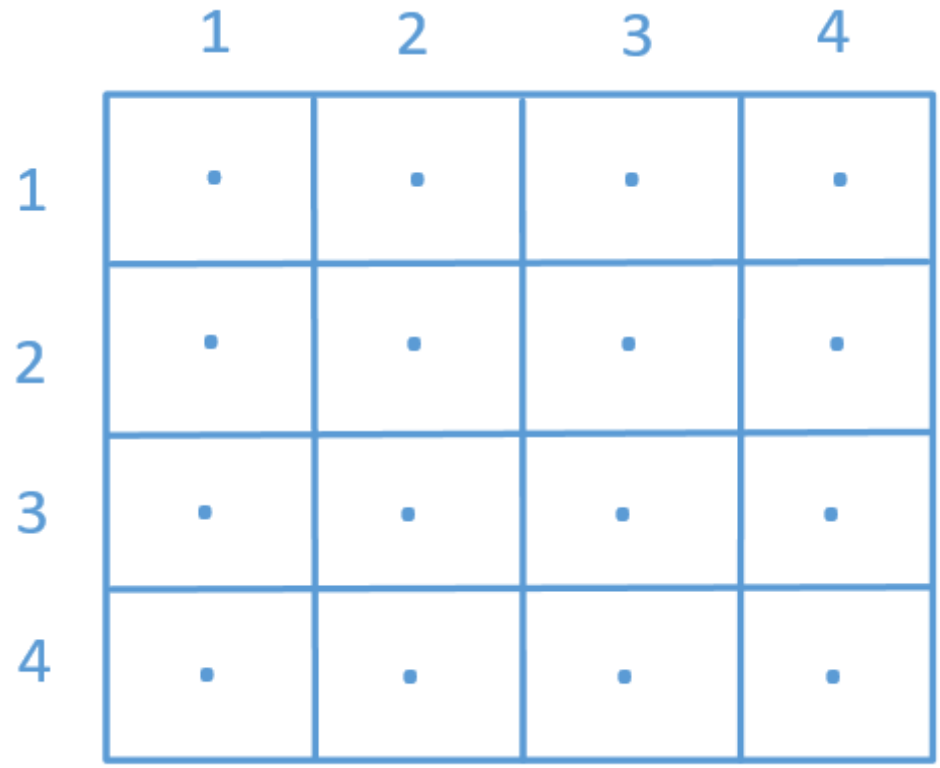
八皇后问题是一个以国际象棋为背景的问题：如何能够在8×8的国际象棋棋盘上放置八个皇后，使得任何一个皇后都无法直接吃掉其他的皇后？为了达到此目的，任两个皇后都不能处于同一条横行、纵行或斜线上。八皇后问题可以推广为更一般的n皇后摆放问题：这时棋盘的大小变为n×n，而皇后个数也变成n。当且仅当n = 1或n ≥ 4时问题有解。

八皇后问题最早是由国际象棋棋手马克斯·贝瑟尔（Max Bezzel）于1848年提出。第一个解在1850年由弗朗兹·诺克（Franz Nauck）给出。并且将其推广为更一般的n皇后摆放问题。诺克也是首先将问题推广到更一般的n皇后摆放问题的人之一。



问题分析

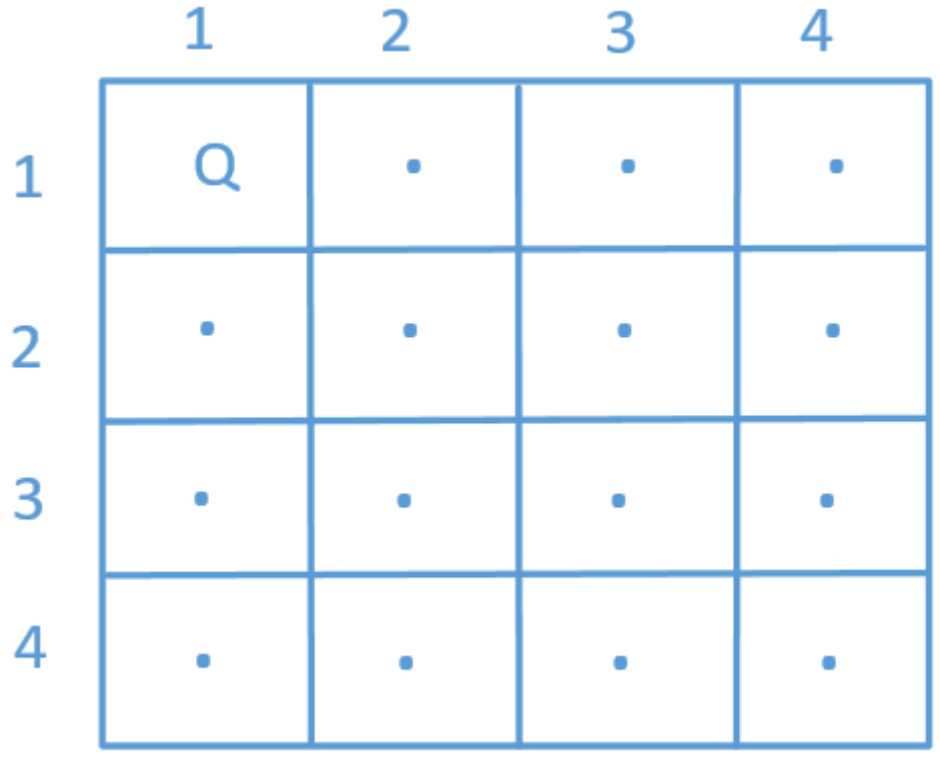
我们先不看8皇后，我们先来看一下4皇后，其实原理都是一样的，比如在下面的4*4的格子里，如果我们在其中一个格子里输入了皇后，那么在这一行这一列和这左右两边的对角线上都不能有皇后。



所以有一种方式就是我们一个个去试

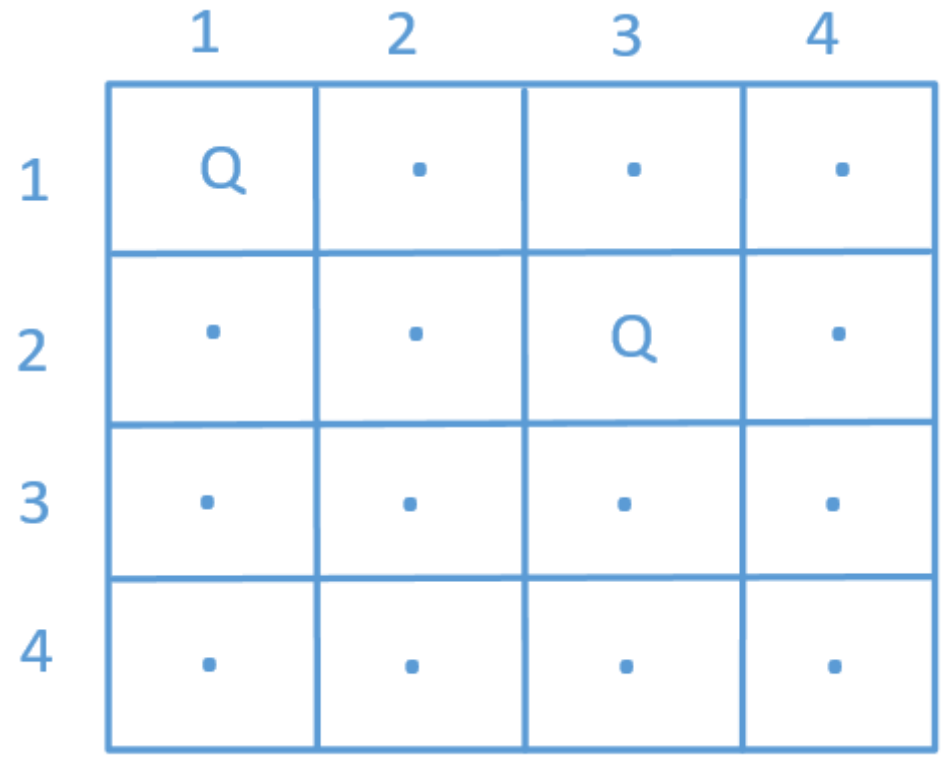
第一行

比如我们在第一行第一列输入了一个皇后



第二行

第二行我们就不能在第一列和第二列输入皇后了，因为有冲突了。但我们可以在第3列输入皇后

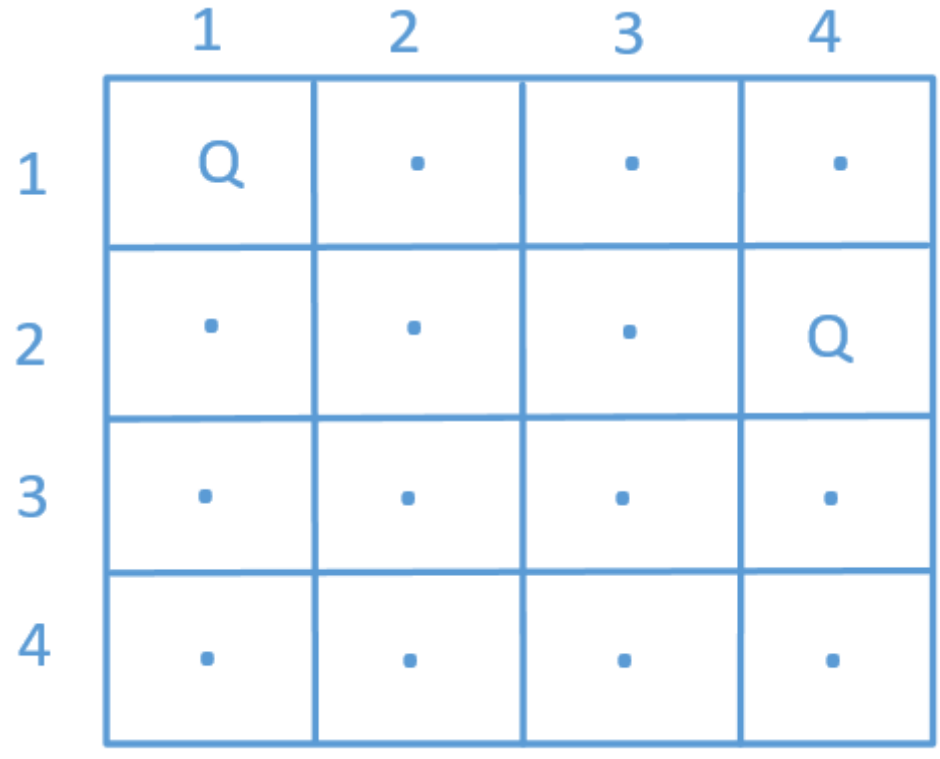


第三行

第三行我们发现在任何位置输入都会有冲突。这说明我们之前选择的是错误的，再回到上一步，我们发现第二步不光能选择第3列，而且还能选择第4列，既然选择第3列不合适，那我们就选择第4列吧

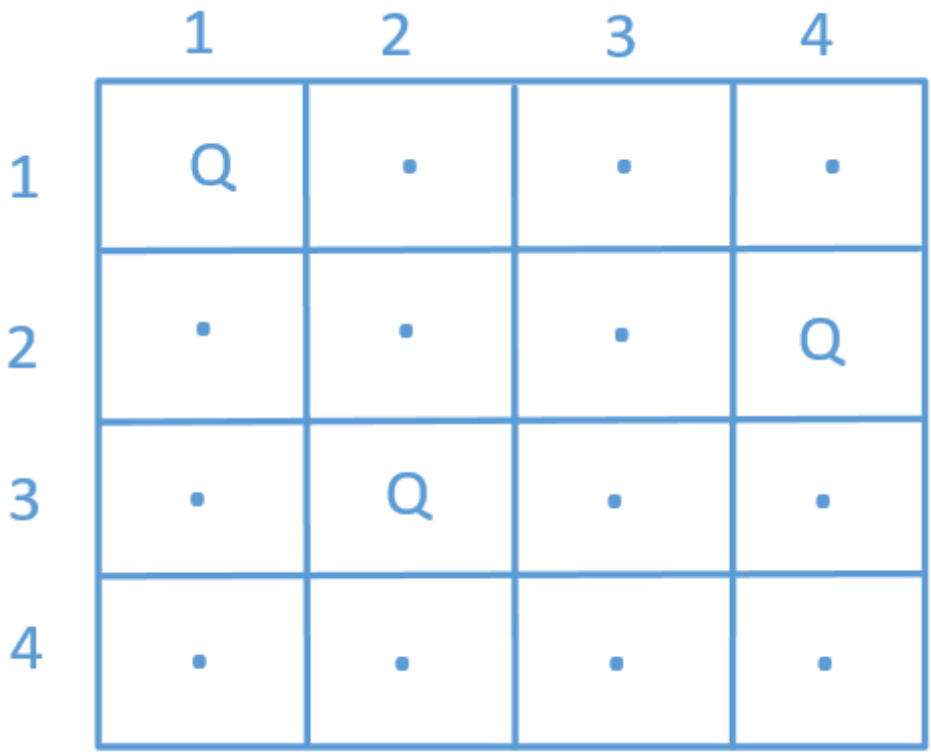
第二行（重新选择）

第二行我们选择第4列



第三行（重新选择）

第3行我们只有选择第2列不会有冲突



第四行

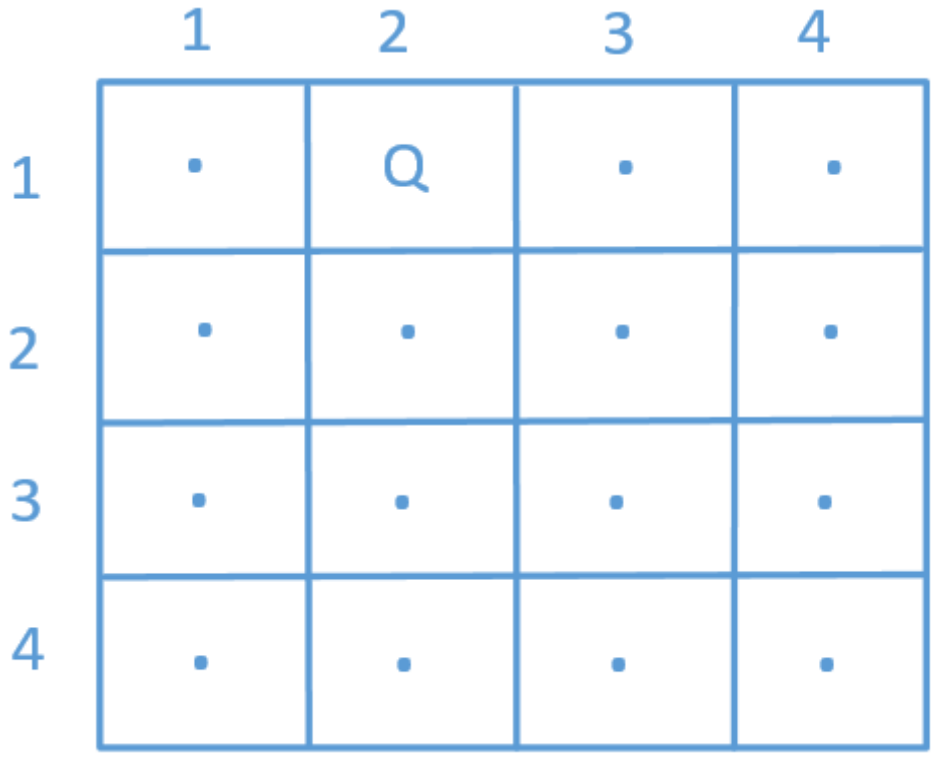
我们发现第4行又没有可选择的了。第一次重试失败

第二次重试

到这里我们只有重新回到第一步了，这说明我们之前第一行选择第一列是无解的，所以我们第一行不应该选择第一列，我们再来选择第二列来试试

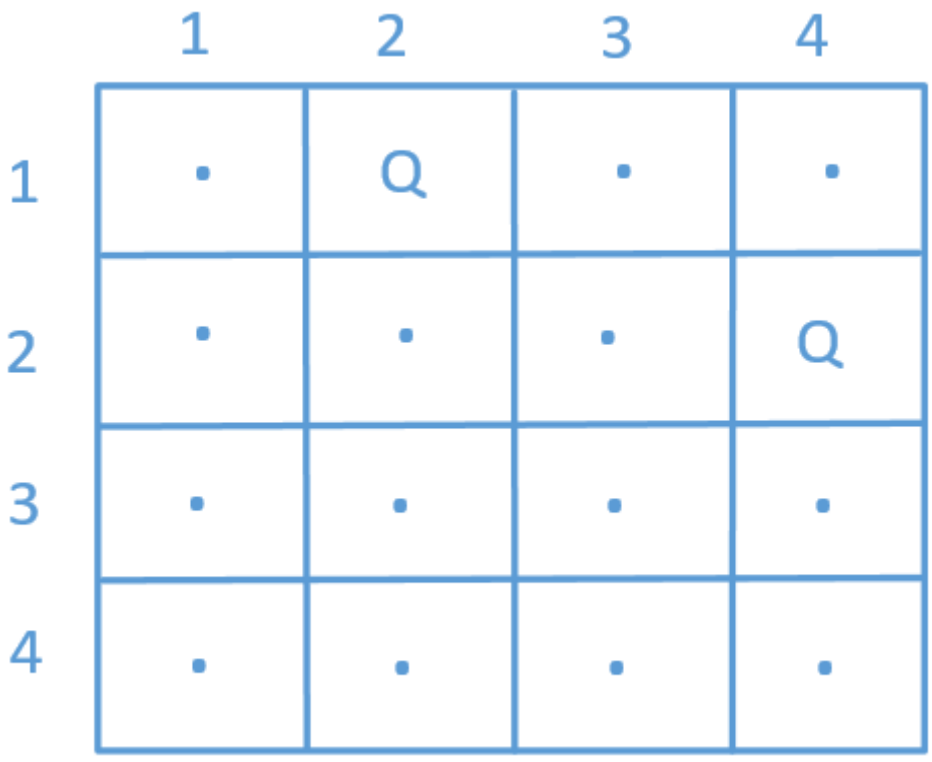
第一行

这一行我们选择第2列



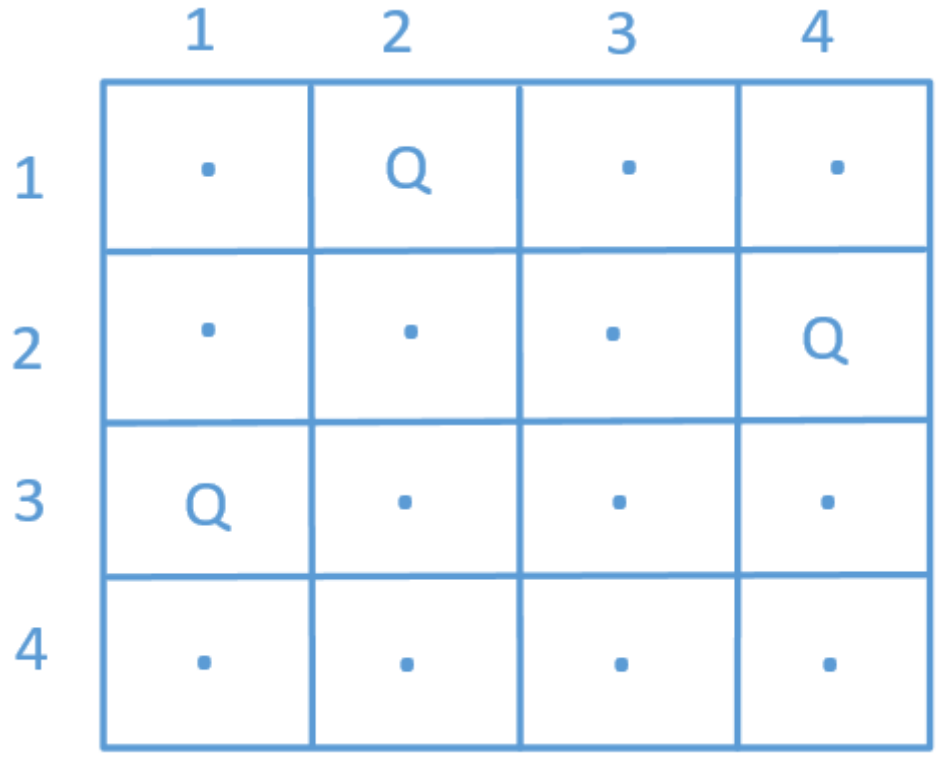
第二行

第二行我们前3个都不能选，只能选第4列



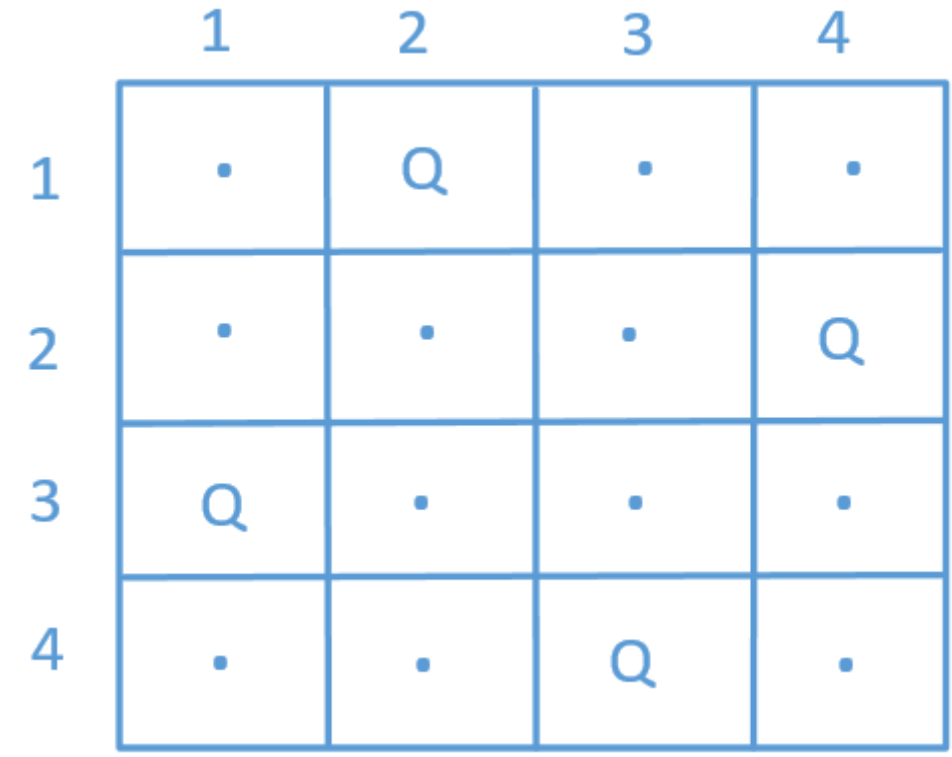
第三行

第三行我们只能选第1列



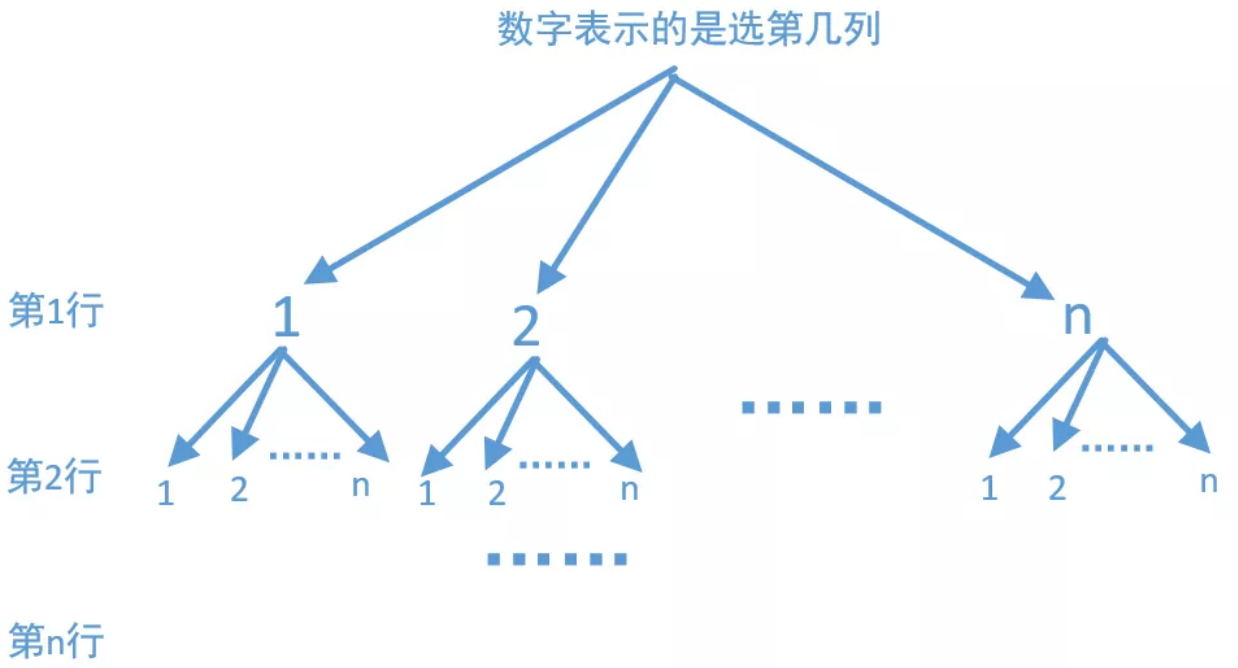
第四行

第四行我们只能选第3列



最后我们终于找到了一组解。除了这组解还有没有其他解呢，肯定还是有的，因为4皇后是有两组解的，这里我们就不在一个个试了。

我们来看一下他查找的过程，就是先从第1行的第1列开始往下找，然后再从第1行的第2列……，一直到第1行的第n列。代码该怎么写呢，看到这里估计大家都已经想到了，这不就是一棵N叉树的前序遍历吗，我们来看下，是不是下面这样的。



我们先来看一下二叉树的前序遍历怎么写，不明白的可以参考下373，数据结构-6,树

```
public static void preOrder(TreeNode tree) {
    if (tree == null)
        return;
    System.out.printf(tree.val + "");
    preOrder(tree.left);
    preOrder(tree.right);
}
```

二叉树是有两个子节点，那么N叉树当然是有N个子节点了，所以N叉树的前序遍历是这样的

```
public static void preOrder(TreeNode tree) {
    if (tree == null)
        return;
    System.out.printf(tree.val + "");
    preOrder("第1个子节点");
    preOrder("第2个子节点");
    ……
    preOrder("第n个子节点");
}
```

如果N是一个很大的值，这样写要写死人了，我们一般使用循环的方式

```
public static void preOrder(TreeNode tree) {
```

```
    if (tree == null)
        return;
    System.out.printf(tree.val + "");
    for (int i = 0; i <n ; i++) {
        preOrder("第i个子节点");
    }
}
```

搞懂了上面的分析过程，那么这题的代码轮廓就呼之欲出了

```
private void solve(char[][] chess, int row) {
  "终止条件"
  return;

  for (int col = 0; col < chess.length; col++) {
    //判断这个位置是否可以放皇后
    if (valid(chess, row, col)) {
      //如果可以放，我们就把原来的数组chess复制一份，
      char[][] temp = copy(chess);
      //然后在这个位置放上皇后
      temp[row][col] = 'Q';
      //下一行
      solve(temp, row + 1);
    }
  }
}
```

我们来分析下上面的代码，因为是递归所以必须要有终止条件，那么这题的终止条件就是我们最后一行已经走完了，也就是

```
if (row == chess.length) {
        return;
}
```

第7行就是判断在这个位置我们能不能放皇后，如果不能放我们就判断这一行的下一列能不能放……，如果能放我们就把原来数组chess复制一份，然后把皇后放到这个位置，然后再判断下一行，这和我们上面画图的过程非常类似。注意这里的第9行为什么要复制一份，因为数组是引用传递，这涉及到递归的时候分支污染问题（后面有时间我会专门写一篇关于递归的时候分支污染问题）。当然不复制一份也是可以的，我们下面再讲。当我们把上面的问题都搞懂的时候，代码也就很容易写出来了，我们来看下N皇后的最终代码

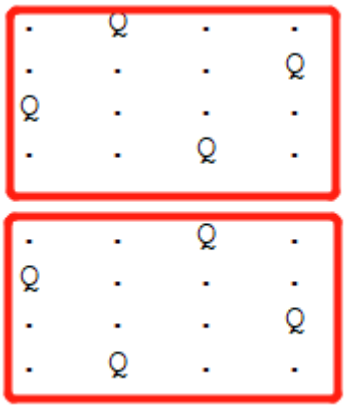
```
public void solveNQueens(int n) {
  char[][] chess = new char[n][n];
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      chess[i][j] = '.';
  solve(chess, 0);
}

private void solve(char[][] chess, int row) {
  if (row == chess.length) {
    //自己写的一个公共方法，打印二维数组的，
    // 主要用于测试数据用的
```

```
13        Util.printTwoCharArrays(chess);
14        System.out.println();
15        return;
16      }
17      for (int col = 0; col < chess.length; col++) {
18        if (valid(chess, row, col)) {
19          char[][] temp = copy(chess);
20          temp[row][col] = 'Q';
21          solve(temp, row + 1);
22        }
23      }
24    }
25
26    //把二维数组chess中的数据测下copy一份
27    private char[][] copy(char[][] chess) {
28      char[][] temp = new char[chess.length][chess[0].length];
29      for (int i = 0; i < chess.length; i++) {
30        for (int j = 0; j < chess[0].length; j++) {
31          temp[i][j] = chess[i][j];
32        }
33      }
34      return temp;
35    }
36
37    //row表示第几行，col表示第几列
38    private boolean valid(char[][] chess, int row, int col) {
39      //判断当前列有没有皇后,因为他是一行一行往下走的，
40      //我们只需要检查走过的行数即可，通俗一点就是判断当前
41      //坐标位置的上面有没有皇后
42      for (int i = 0; i < row; i++) {
43        if (chess[i][col] == 'Q') {
44          return false;
45        }
46      }
47      //判断当前坐标的右上角有没有皇后
48      for (int i = row - 1, j = col + 1; i >= 0 && j < chess.length; i--, j++) {
49        if (chess[i][j] == 'Q') {
50          return false;
51        }
52      }
53      //判断当前坐标的左上角有没有皇后
54      for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
55        if (chess[i][j] == 'Q') {
56          return false;
57        }
58      }
59      return true;
60    }
```

代码看起来比较多，我们主要看下solve方法即可，其他的方法不看也可以，知道有这个功能就行。solve代码中其核心代码是在17-23行，上面是终止条件的判断，我们就用4皇后来测试一下

```
1  solveNQueens(4);
```

看一下打印的结果



我们看到4皇后的时候有两组解，其中第一组和我们上面图中分析的完全一样。

4皇后解决了，那么8皇后也一样，我们只要在函数调用的时候传入8就可以了。同理，要想计算10皇后，20皇后，100皇后……也都是可以的。

回溯解决

上面代码中每次遇到能放皇后的时候，我们都会把原数组复制一份，这样对新数据的修改就不会影响到原来的，也就是不会造成分支污染。但这样每次尝试的时候都都把原数组复制一份，影响效率，有没有其他的方法不复制呢，是有的。就是每次我们选择把这个位置放置皇后的时候，如果最终不能成功，那么返回的时候我们就还要把这个位置还原。这就是回溯算法，也是试探算法。我们来看下代码

```
1   private void solve(char[][] chess, int row) {
2     if (row == chess.length) {
3       //自己写的一个公共方法，打印二维数组的，
4       // 主要用于测试数据用的
5       Util.printTwoCharArrays(chess);
6       System.out.println();
7       return;
8     }
9     for (int col = 0; col < chess.length; col++) {
10      if (valid(chess, row, col)) {
11        chess[row][col] = 'Q';
12        solve(chess, row + 1);
13        chess[row][col] = '.';
14      }
15    }
16  }
```

主要来看下11-13行，其他的都没变，还和上面的一样。这和我们之前讲的391，回溯算法求组合问题很类似。他是先假设[row][col]这个位置可以放皇后，然后往下找，无论找到找不到最后都会回到这个地方，因为这里是递归调用，回到这个地方的时候我们再把它复原，然后走下一个分支。最后我们再来看下使用回溯算法解N皇后的完整代码

```
1   public void solveNQueens(int n) {
2     char[][] chess = new char[n][n];
3     for (int i = 0; i < n; i++)
4       for (int j = 0; j < n; j++)
5         chess[i][j] = '.';
6     solve(chess, 0);
7   }
8
9   private void solve(char[][] chess, int row) {
```

```
10      if (row == chess.length) {
11          //自己写的一个公共方法，打印二维数组的，
12          // 主要用于测试数据用的
13          Util.printTwoCharArrays(chess);
14          System.out.println();
15          return;
16      }
17      for (int col = 0; col < chess.length; col++) {
18          if (valid(chess, row, col)) {
19              chess[row][col] = 'Q';
20              solve(chess, row + 1);
21              chess[row][col] = '.';
22          }
23      }
24  }
25
26  //row表示第几行，col表示第几列
27  private boolean valid(char[][] chess, int row, int col) {
28      //判断当前列有没有皇后,因为他是一行一行往下走的，
29      //我们只需要检查走过的行数即可，通俗一点就是判断当前
30      //坐标位置的上面有没有皇后
31      for (int i = 0; i < row; i++) {
32          if (chess[i][col] == 'Q') {
33              return false;
34          }
35      }
36      //判断当前坐标的右上角有没有皇后
37      for (int i = row - 1, j = col + 1; i >= 0 && j < chess.length; i--, j++) {
38          if (chess[i][j] == 'Q') {
39              return false;
40          }
41      }
42      //判断当前坐标的左上角有没有皇后
43      for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
44          if (chess[i][j] == 'Q') {
45              return false;
46          }
47      }
48      return true;
49  }
```

总结

8皇后问题其实是一道很经典的回溯算法题，我们这里并没有专门针对8皇后来讲，我们这里讲的是N皇后，如果这道题搞懂了，那么8皇后自然也就懂了，因为这里的N可以是任何正整数。

递归一般可能会有多个分支，我们要保证每个分支的修改不会污染其他分支，也就是不要对其他分支造成影响，这一点很重要。由于一些语言中除了基本类型以外，其他的大部分都是引用传递，所以我们在一个分支修改之后可能就会对其他分支产生一些我们不想要的垃圾数据，这个时候我们就有两中解决方式，一种就是我们上面讲到的第一种，复制一份新的数据，这样每个分支都会产生一些新的数据，所以即使修改了也不会对其他分支有影响。还一种方式就是我们每次使用完之后再把它复原，一般的情况下我们都会选择第二种，因为这种代码更简洁一些，也不会重复的复制数据，造成大量的垃圾数据。

389，两个超级大数相加

原创 山大王wld 数据结构和算法 6月24日

收录于话题
#算法图文分析

96个 >



If you want the rainbow, you have to deal with the rain.

你若想要彩虹，必须经历风雨。

问题描述

给定两个字符串形式的非负整数 num1 和num2 ，计算它们的和。

注意：

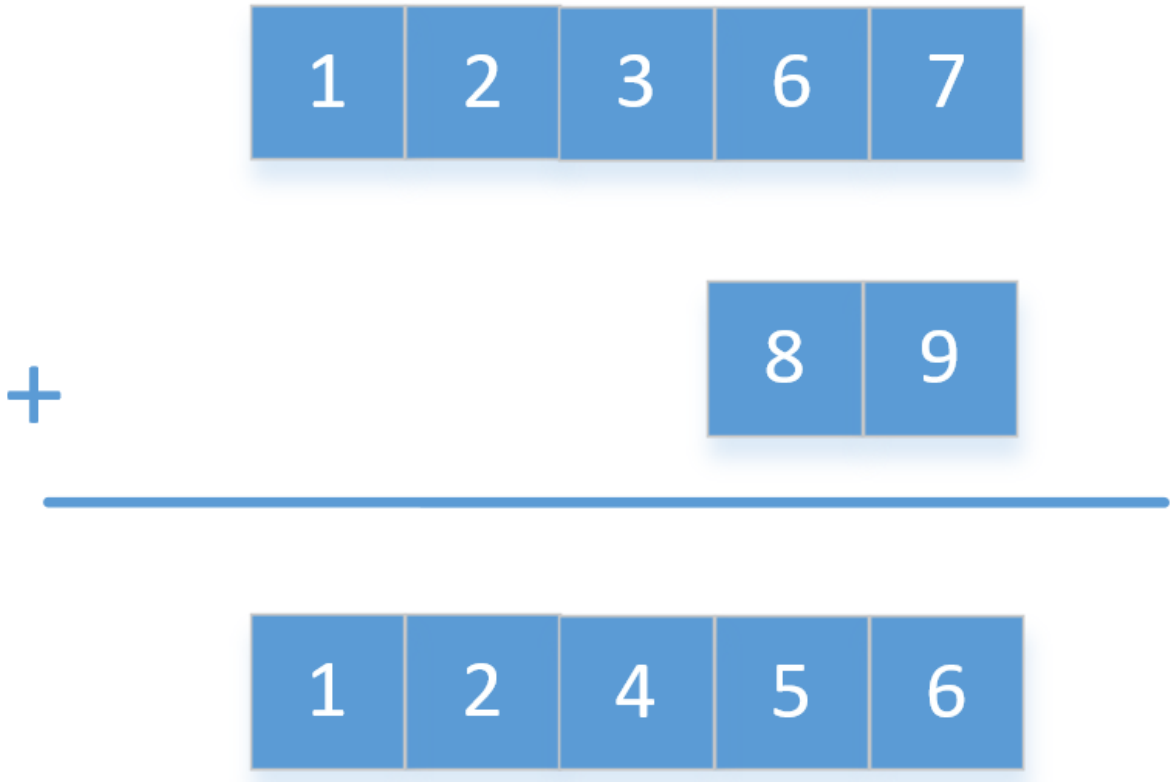
num1 和num2 的长度都小于 5100.
num1 和num2 都只包含数字 0-9.
num1 和num2 都不包含任何前导零。
你不能使用任何内建 BigInteger 库， 也不能直接将输入的字符串转换为整数形式。

示例 1：

```
"12345678901278"+"234"="12345678901512"
```

问题分析

实际上这道题求的是两个字符串相加，我们就用两个很短的字符串"12367"+"89"为例画个图来看下是怎么计算的



它相当于两个字符串从最右边开始相加，比如我们要计算num1字符串的最右边的那个数字和num2字符串最右边的那个字符相加

```
int i = num1.length() - 1, j = num2.length() - 1;
int x = num1.charAt(i) - '0';
int y = num2.charAt(j) - '0';
int sum = x + y;
```

把计算的结果放到一个新的字符串后面，但字符串每一位只能保存一位数字，而我们相加的结果sum可能是个两位数，所以这里我们只取他的个位数，十位数要往前进一位。比如我们要计算num1和num2的倒数第二位

```
int x = num1.charAt(i - 1) - '0';
int y = num2.charAt(j - 1) - '0';
int sum = x + y + carry;
```

carry就是上一步相加结果的进位，上一步如果进位了就是1，如果没进位就是0。搞懂了上面的相加过程，剩下的就是一些边界条件的判断。最后不要忘了对字符串进行反转，因为我们相加的时候是从num1和num2的尾部开始加的，下面我们来看下完整代码

```java
public String addStrings(String num1, String num2) {
    StringBuilder s = new StringBuilder();
    int i = num1.length() - 1, j = num2.length() - 1, carry = 0;
    while (i >= 0 || j >= 0 || carry != 0) {
        int x = i < 0 ? 0 : num1.charAt(i--) - '0';
        int y = j < 0 ? 0 : num2.charAt(j--) - '0';
        int sum = x + y + carry;
        s.append(sum % 10);//添加到字符串尾部
        carry = sum / 10;
    }
    return s.reverse().toString();//对字符串反转
}
```

从头部插入

上面是插入到字符串的尾部，最后再反转。实际上我们在计算的时候还可以先插入到字符串的头部，最后直接返回即可，不需要再反转了，代码和上面差不多，我们来看下

```java
public String addStrings(String num1, String num2) {
    StringBuilder s = new StringBuilder();
    int carry = 0, i = num1.length() - 1, j = num2.length() - 1;
    while (i >= 0 || j >= 0 || carry != 0) {
        int x = i < 0 ? 0 : num1.charAt(i--) - '0';
        int y = j < 0 ? 0 : num2.charAt(j--) - '0';
        int sum = x + y + carry;
        s.insert(0, sum % 10);//插入到s字符串的第一个位置
        carry = sum / 10;
    }
    return s.toString();
}
```

使用栈来解决

我们还可以先把相加的结果放到一个栈中，最后再一个个出栈。其实也是换汤不换药，代码都差不多，我们来看下

```java
public String addStrings(String num1, String num2) {
    Stack<Integer> stack = new Stack<>();
```

```
    StringBuilder sb = new StringBuilder();
    int i = num1.length() - 1, j = num2.length() - 1, carry = 0;
    while (i >= 0 || j >= 0 || carry != 0) {
        carry += i >= 0 ? num1.charAt(i--) - '0' : 0;
        carry += j >= 0 ? num2.charAt(j--) - '0' : 0;
        stack.push(carry % 10);
        carry = carry / 10;
    }
    while (!stack.isEmpty())
        sb.append(stack.pop());
    return sb.toString();
}
```

递归的方式解决

除了上面介绍的几种以外，我们还可以把它改为递归的方式

```
public String addStrings(String num1, String num2) {
    return addBinaryHelper(num1, num1.length() - 1, num2, num2.length() -
}

public String addBinaryHelper(String num1, int indexA, String num2, int in
    if (indexA < 0 && indexB < 0 && carry == 0)
        return "";
    carry += indexA < 0 ? 0 : num1.charAt(indexA--) - '0';
    carry += indexB < 0 ? 0 : num2.charAt(indexB--) - '0';
    int digit = carry % 10;
    carry = carry / 10;
    String res = addBinaryHelper(num1, indexA, num2, indexB, carry);
    return res + digit;
}
```

总结

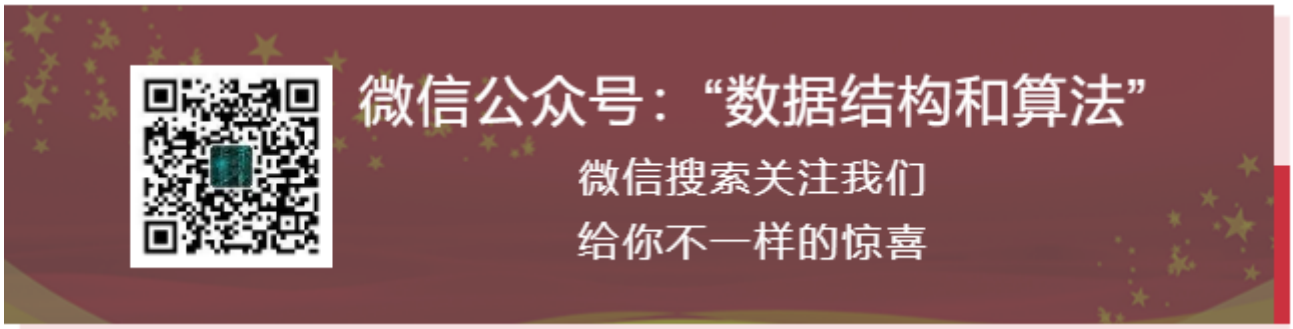
这题非常简单，解题思路大同小异，都是先从两个字符串的最后一位开始相加，只不过写的时候会有多种方式。

371，背包问题系列之-基础背包问题

原创 山大王wld 数据结构和算法 5月28日

收录于话题
#算法图文分析

96个 >



描述

背包问题是动态规划中最经典的一道算法题。背包问题的种类比较多，我们先来看一个最简单的背包问题-基础背包。他是这样描述的。
有N件物品和一个容量为V的包，第i件物品的重量是w[i]，价值是v[i]，求将哪些物品装入背包可使这些物品的重量总和不能超过背包容量，且价值总和最大。我们先来举个例子分析一下

举例分析

假设我们背包可容纳的重量是4kg，有3样东西可供我们选择，一个是高压锅有4kg，价值300元，一个是风扇有3kg，价值200元，最后一个是一双运动鞋有1kg，价值150元。问要装哪些东西在重量不能超过背包容量的情况下价值最大。如果只装高压锅价值才300元，如果装风扇和运动鞋价值将达到350元，所以装风扇和运动鞋才是最优解，我们来画个图分析一下

01 结合图形分析



高压锅
重量：4kg
价值：300元

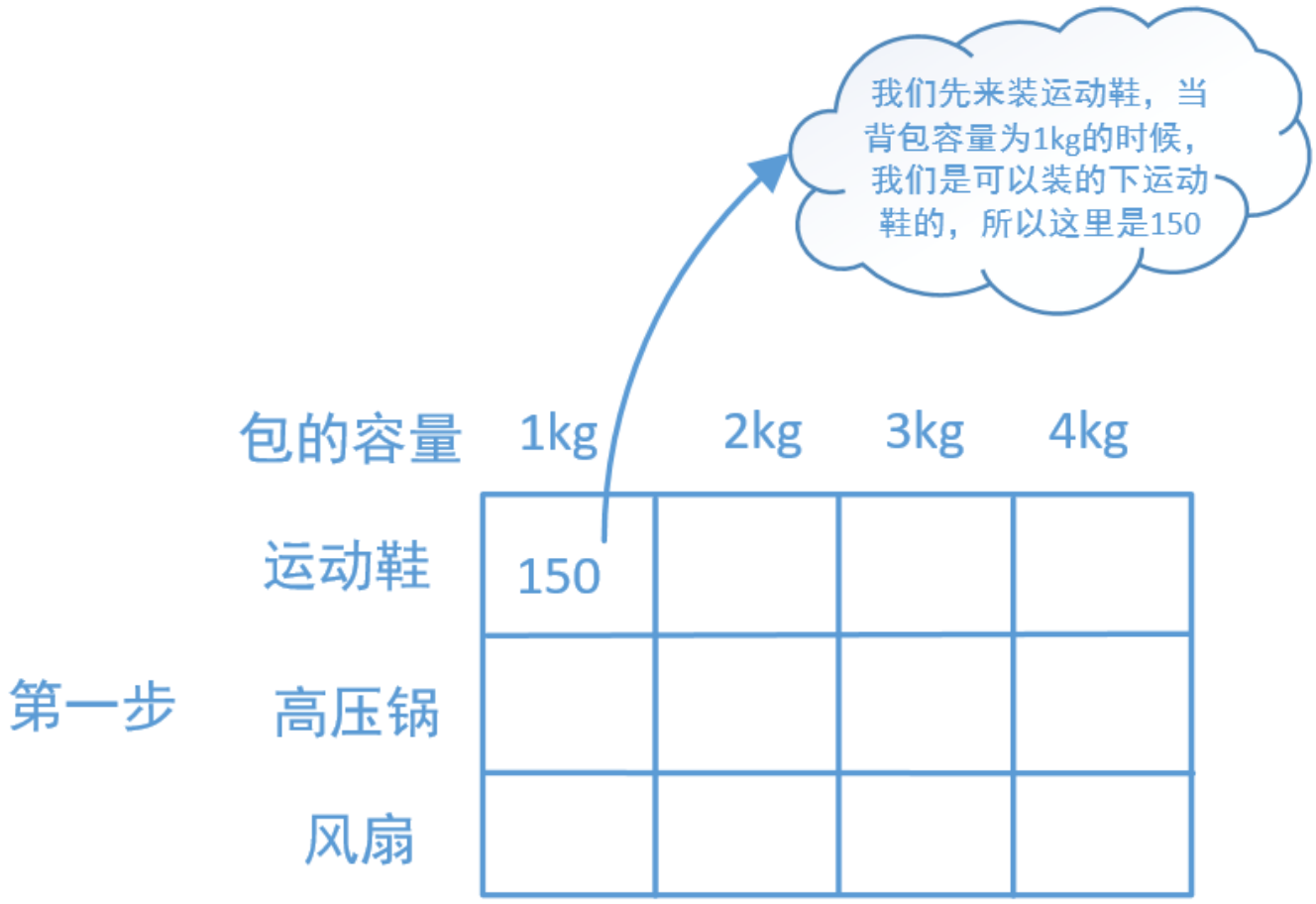


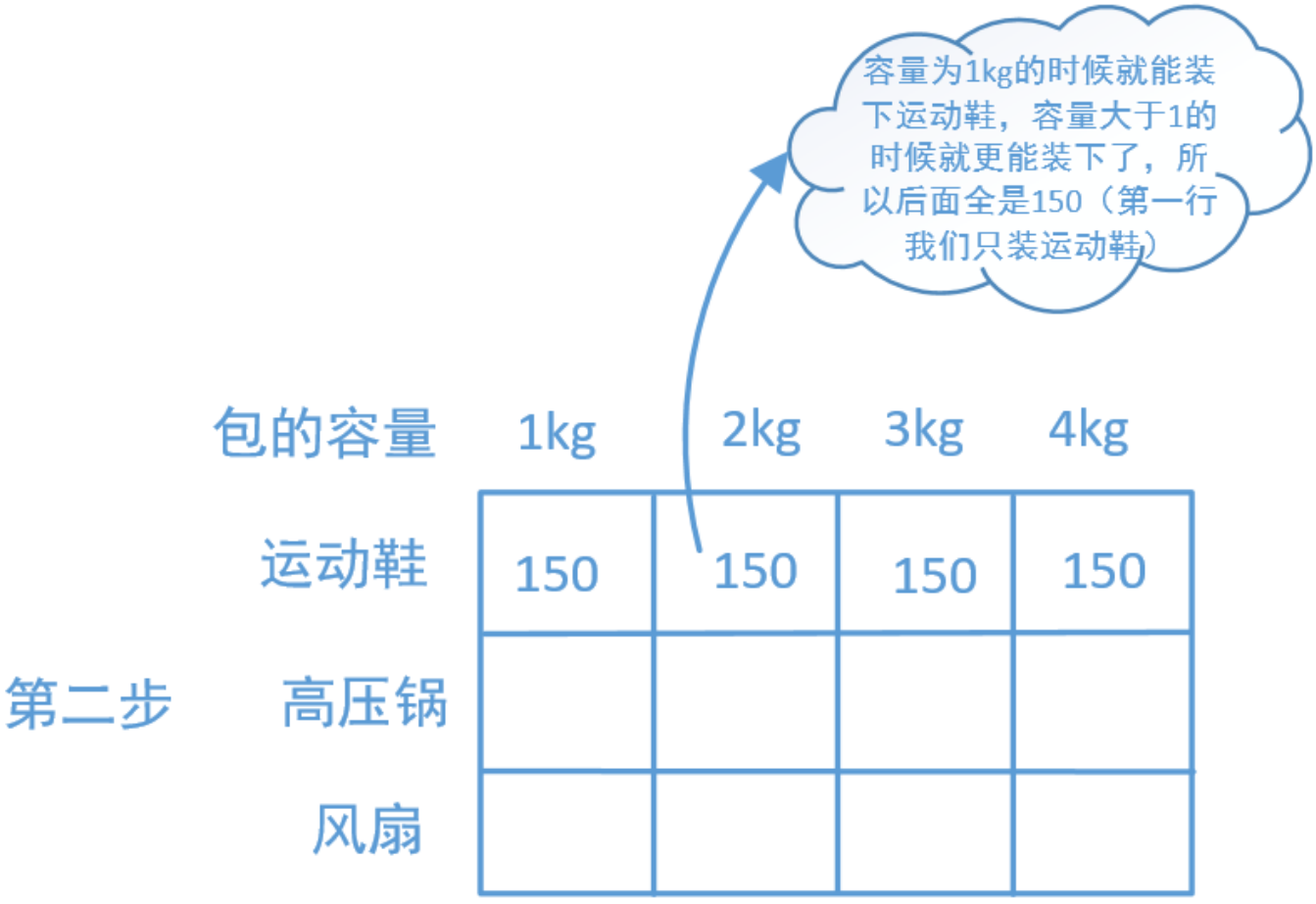
风扇
重量：3kg
价值：200元



运动鞋
重量：1kg
价值：150元

原始状态	包的容量	1kg	2kg	3kg	4kg
	运动鞋				
	高压锅				
	风扇				


我们先来装运动鞋，当背包容量为1kg的时候，我们是可以装的下运动鞋的，所以这里是150
包的容量
1kg
2kg
3kg
4kg
运动鞋
150
高压锅
风扇
第一步


容量为1kg的时候就能装下运动鞋，容量大于1的时候就更能装下了，所以后面全是150（第一行我们只装运动鞋）
包的容量
1kg
2kg
3kg
4kg
运动鞋
150
150
150
150
高压锅
风扇
第二步

第二行我们可以选择高压锅了，因为1kg的容量我们装不下高压锅，只能装运动鞋，所以最大值还是150

第三步

包的容量	1kg	2kg	3kg	4kg
运动鞋	150	150	150	150
高压锅	150			
风扇				

同理当包的容量是2，3的时候还是装不下高压锅的（在这一行我们还没到风扇这一步，所以还不能选择风扇，只能选择运动鞋和高压锅），只能装运动鞋，所以最大值还是150

第四步

包的容量	1kg	2kg	3kg	4kg
运动鞋	150	150	150	150
高压锅	150	150	150	
风扇				

包的容量为4的时候我们可以装下高压锅了，如果装高压锅的话，最大价值是300元，比运动鞋的价值大，所以我们选择装高压锅

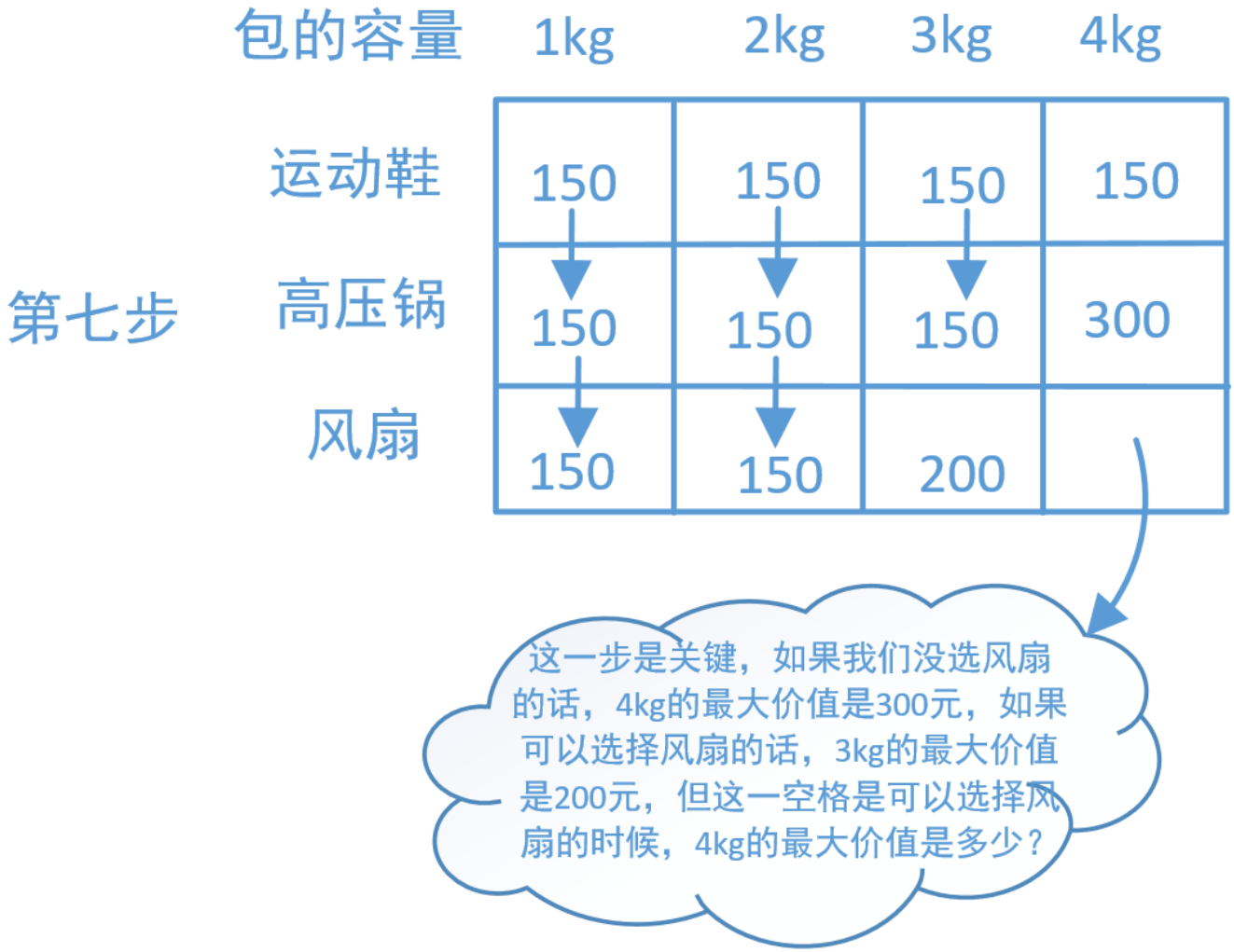
第五步

包的容量	1kg	2kg	3kg	4kg
运动鞋	150	150	150	150
高压锅	150	150	150	300
风扇				

第三行我们可以3个都能选择了，当包容量为1，2的时候，我们只能选择运动鞋。但当包容量为3的时候我们可以选择风扇了，风扇的价值大于150，所以当包容量为3kg的时候我们选择风扇。

第六步

包的容量	1kg	2kg	3kg	4kg
运动鞋	150	150	150	150
高压锅	150	150	150	300
风扇	150	150	200	


包的容量
1kg
2kg
3kg
4kg
第七步
运动鞋
高压锅
风扇
150
150
150
150
150
150
150
300
150
150
200
这一步是关键，如果我们没选风扇的话，4kg的最大价值是300元，如果可以选择风扇的话，3kg的最大价值是200元，但这一空格是可以选择风扇的时候，4kg的最大价值是多少？

不选风扇
4kg，300元

?

选风扇
3kg，200元

+

1kg可装
的价值

这1kg可装运动鞋，价值150元



不选风扇
4kg，300元

?

选风扇
3kg，200元

+

运动鞋
1kg，150元

300元 ? 200+150=350元 （很明显我们选风扇和运动鞋的价值是最大的）

所以递推公式我们很容易就能得出

dp[i][j] = max
- dp[i-1][j]（不选当前商品）
- 当前商品的价值+剩余空间可容纳的价值(选当前商品)

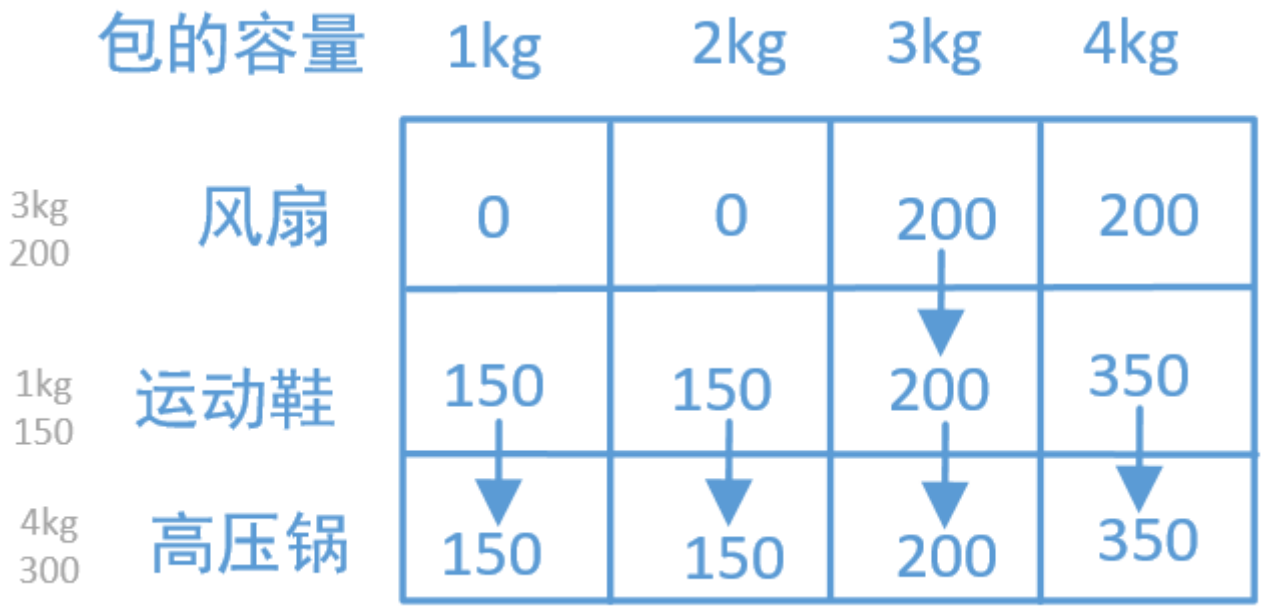
转换成公式



dp[i][j] = max
- dp[i-1][j]（不选当前商品）
- dp[i-1][j-weight(i)]+value[i]（选当前商品）

02 改变选择的顺序

我们上面选择的顺序是：运动鞋→高压锅→风扇，如果我们改变选择的顺序，结果会不会改变，比如我们选择的顺序是：风扇→运动鞋→高压锅，我们还是来画个图看一下



我们发现无论选择顺序怎么改变都不会改变最终的结果。

数据测试：

我们就用上面的图形分析的数据来测试一下，看一下最终结果

```
public static void main(String[] args) {
  System.out.println("最终结果是：" + packageProblem1());
}

public static int packageProblem1() {
  int packageContainWeight = 4;//包最大可装重量
  int[] weight = {1, 4, 3};//3个物品的重量
  int[] value = {150, 300, 200};//3个物品的价值
  int[][] dp = new int[weight.length + 1][packageContainWeight + 1];
  for (int i = 1; i <= value.length; i++) {
    for (int j = 1; j <= packageContainWeight; j++) {
      if (j >= weight[i - 1]) {
        dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - weight[i - 1]] + value[i - 1]);
      } else {
        dp[i][j] = dp[i - 1][j];
      }
    }
  }
  Util.printTwoIntArrays(dp);//这一行仅做打印观测数据，也可以去掉
```

```
20      return dp[weight.length][packageContainWeight];
21  }
```

03 运行结果

```
0    0    0    0    0
0    150  150  150  150
0    150  150  150  300
0    150  150  200  350
最终结果是：350
```

和我们上面分析的完全一致。（为了测试方便，这里的所有数据我都是写死的，我们也可以把这些数据提取出来，作为函数参数传进来。）

空间优化：

其实这题还可以优化一下，这里的二维数组我们每次计算的时候都是只需要上一行的数字，其他的我们都用不到，所以我们可以用一维空间的数组来记录上一行的值即可，但要记住一维的时候一定要逆序，否则会导致重复计算。我们来看下代码

```
1   public static int packageProblem2() {
2     int packageContainWeight = 4;
3     int[] weight = {1, 4, 3};
4     int[] value = {150, 300, 200};
5     int[] dp = new int[packageContainWeight + 1];
6     for (int i = 1; i <= value.length; i++) {
7       for (int j = packageContainWeight; j >= 1; j--) {
8         if (j - weight[i - 1] >= 0)
9           dp[j] = Math.max(dp[j], dp[j - weight[i - 1]] + value[i - 1]);
10      }
11      Util.printIntArrays(dp);
12      System.out.println();
13    }
14    return dp[packageContainWeight];
15  }
```

注意：

我们看到第7行在遍历重量的时候采用的是逆序的方式，因为第9行在计算dp[j]的值的时候，数组后面的值会依赖前面的值，而前面的值不会依赖后面的值，如果不采用逆序的方式，数组前面的值更新了会对后面产生影响。

04 运行结果

```
0    150 150 150 150
0    150 150 150 300
0    150 150 200 350
最终结果是：350
```

C++：

```cpp
#include<iostream>
#include <algorithm>

using namespace std;

int main()
{
  int weight[] = { 1,4,3 };
  int value[] = {150, 300, 200 };
  int packageContainWeight = 4;
  int dp[4][5]= { { 0 } };
  for (int i = 1; i <4 ; i++)
  {
    for (int j = 1; j < 5; j++)
    {
      if (j >= weight[i - 1])
        dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i - 1]] + value[i - 1]);
      else
        dp[i][j] = dp[i - 1][j];
    }
  }

  for (int i = 0; i < 4; i++)
  {
    for (int j = 0; j < 5; j++)
    {
      cout << dp[i][j] << ' ';
    }
    cout << endl;
  }

  return 0;
}

```

05 运行结果

```
0 0 0 0 0
0 150 150 150 150
0 150 150 150 300
0 150 150 200 350

Process returned 0 (0x0)   execution time : 0.058 s
Press any key to continue.
```

递归写法：

除了上面的两种写法以外，我们还可以使用递归的方式，代码中有注释，有兴趣的可以自己看，就不在详细介绍。

```
int[] weight = {1, 4, 3};//3个物品的重量
int[] value = {150, 300, 200};//3个物品的价值

// i：处理到第i件物品，j可容纳的重量
public int packageProblem3(int i, int j) {
  if (i == -1)
    return 0;
  int v1 = 0;
  if (j >= weight[i]) {//如果剩余空间大于所放的物品
    v1 = packageProblem3(i - 1, j - weight[i]) + value[i]; //选第i件
  }
  int v2 = packageProblem3(i - 1, j);//不选第i件
  return Math.max(v1, v2);
}
```

366，约瑟夫环

原创 山大王wld 数据结构和算法 5月21日

收录于话题
#算法图文分析
96个 >

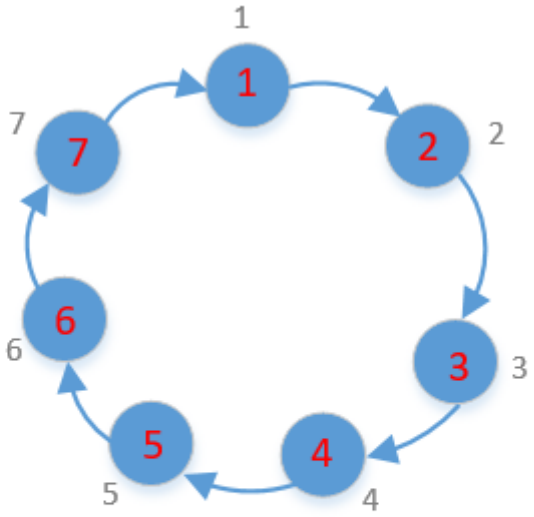
问题来源：

据说著名犹太历史学家 Josephus有过以下的故事：在罗马人占领乔塔帕特后，39 个犹太人与Josephus及他的朋友躲到一个洞中，39个犹太人决定宁愿死也不要被敌人抓到，于是决定了一个自杀方式，41个人排成一个圆圈，由第1个人开始报数，每报数到第3人该人就必须自杀，然后再由下一个重新报数，直到所有人都自杀身亡为止。然而Josephus 和他的朋友并不想遵从。首先从一个人开始，越过k-2个人（因为第一个人已经被越过），并杀掉第k个人。接着，再越过k-1个人，并杀掉第k个人。这个过程沿着圆圈一直进行，直到最终只剩下一个人留下，这个人就可以继续活着。问题是，给定了和，一开始要站在什么地方才能避免被处决？Josephus要他的朋友先假装遵从，他将朋友与自己安排在第16个与第31个位置，于是逃过了这场死亡游戏。

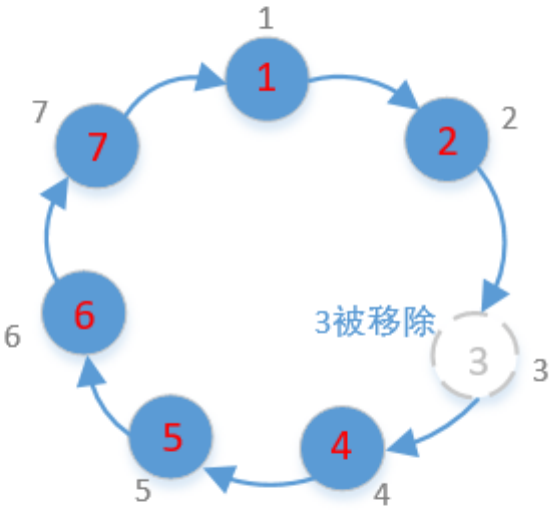
一，留下K-1个

41数字太大了，我们就以7为例，来画个图看一下

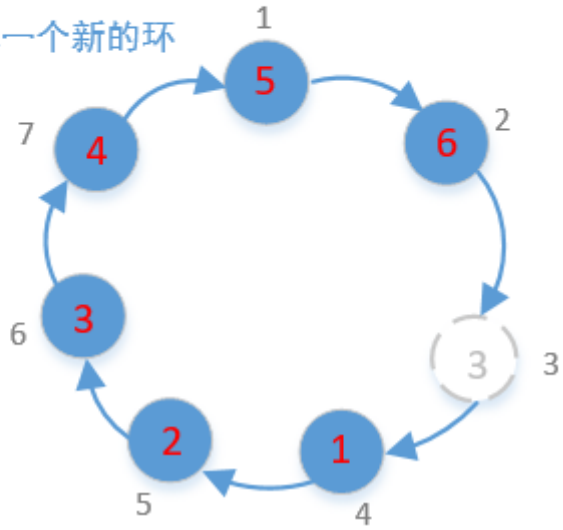
原始状态



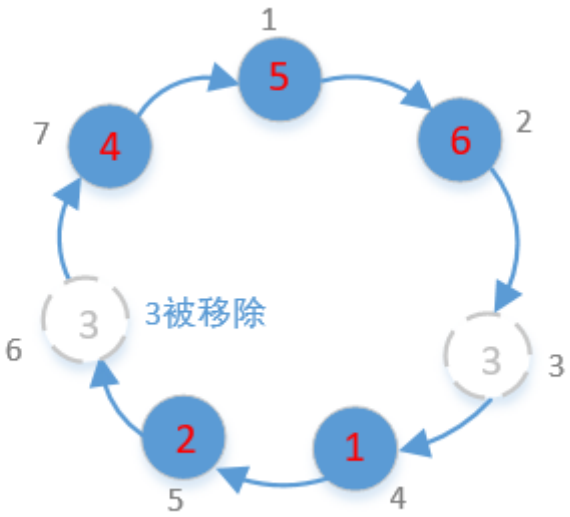
第一步



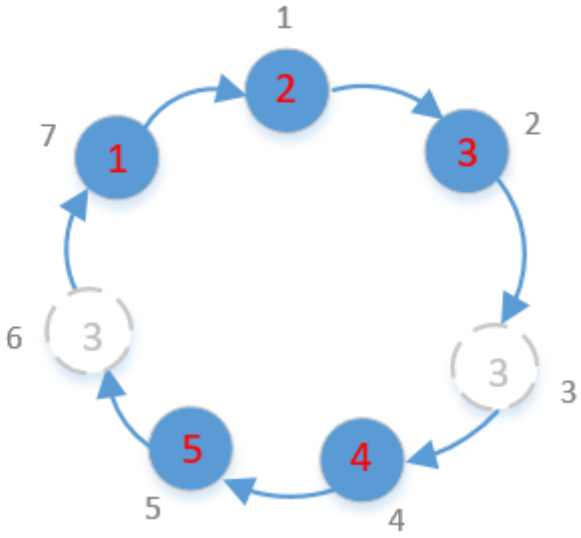
重新构成一个新的环



第二步



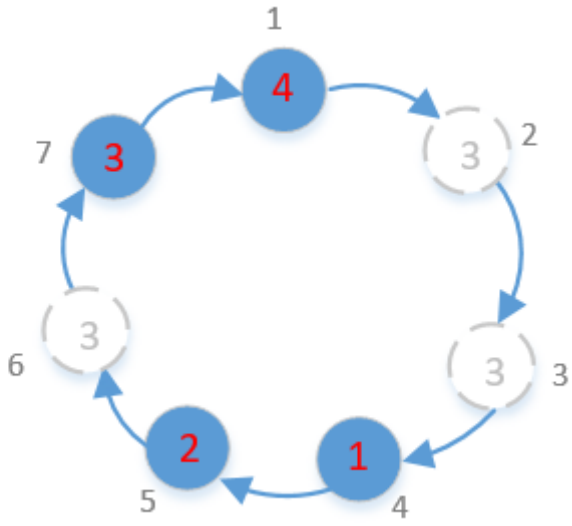
重新构成一个新的环

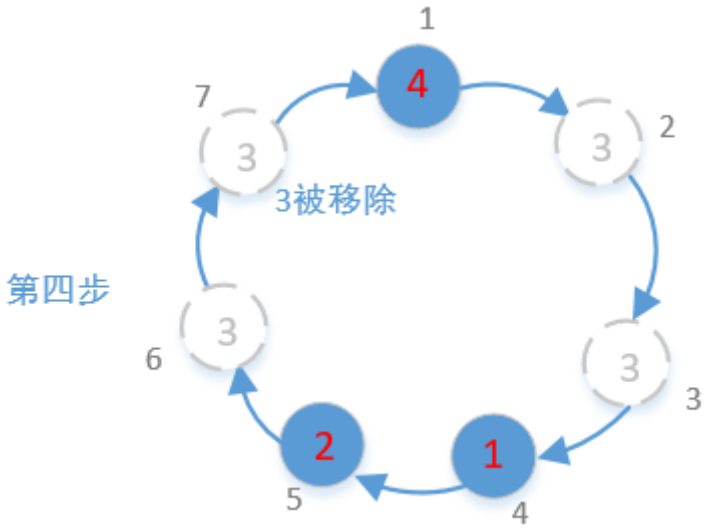


第三步

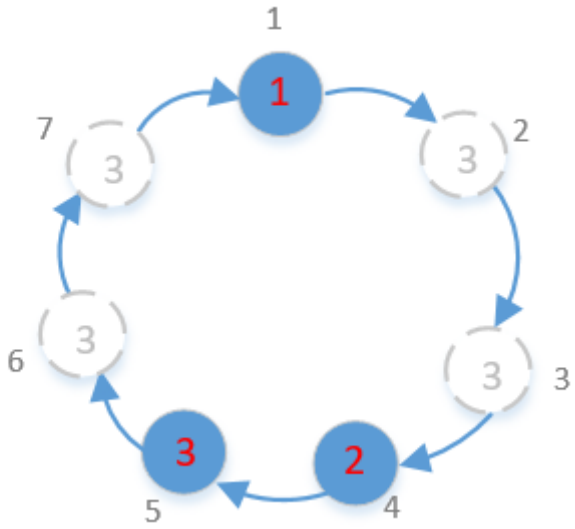


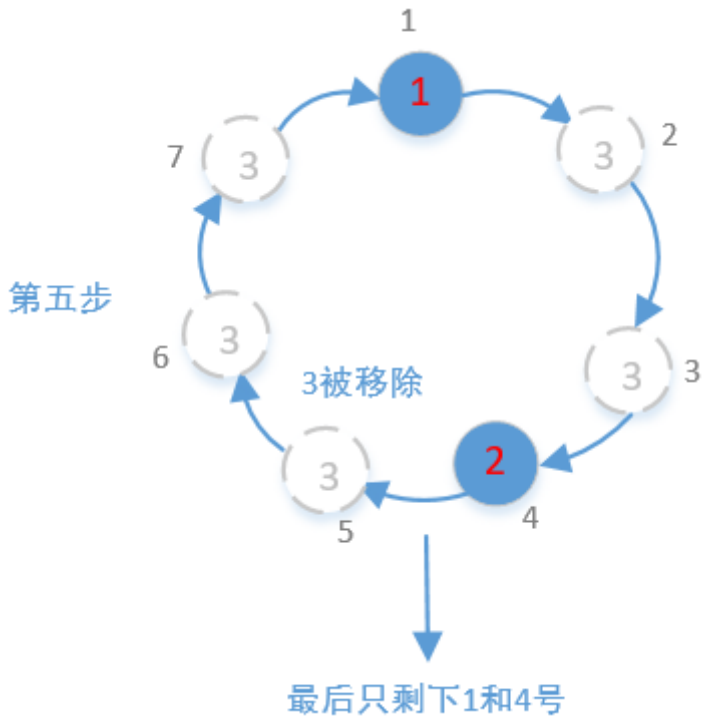
重新构成一个新的环





重新构成一个新的环





我们再来看下代码

1，数组的实现

```
1   public static Integer[] solution(int count, int k) {
2     Integer live[] = new Integer[Math.min(count, k - 1)];
3     if (count < k) {
4       int index = 0;
5       while (index < count) {
6         live[index++] = index;
7       }
8       return live;
9     }
10    List<Integer> mList = new ArrayList<>(count);
11    for (int i = 0; i < count; i++) {
12      mList.add(i + 1);
13    }
14
15    int point = 0;
16    int number = 0;
17    while (mList.size() >= k) {
18      number++;
19      if (point >= mList.size()) {
20        point = 0;
21      }
22      if (number % k == 0) {
23        mList.remove(point);
24        continue;
25      }
26      point++;
27    }
28    return mList.toArray(live);
29  }
```

1，3-9行表示如果k大于count就直接把所有人的编号都返回即可，不再删除了。

2，11-13行生成从1到count的所有值（包含1和count）

3，16行number统计数量，在第22-25行如果统计的数量是k的倍数就把他移除。其实我们也可以在22行成立的时候让number重新归0。这里使用的是对k求余也是可以的。

4，我们就用上面已知的两组数据测试一下，当count等于41的时候结果是16和31，当count等于7的时候结果是1和4

```
1     Util.printObjectArrays(solution(41, 3));
2     System.out.println("----------------------");
3     Util.printObjectArrays(solution(7, 3));
```

运行结果是

```
1       16
2       31
3       ----------------------
4       1
5       4
```

结果完全正确。

数组的删除会导致后面的元素都会往前移，频繁的删除效率肯定不是很高，其实我们还可以使用链表。因为链表的删除不需要移动后面的元素，效率还是比较高的。如果不使用链表，我们还可以把数组中删除的元素用一个负数来填充，这样也是可以的。我们来看下

2，数组实现的另一种方式

```
1   /**
2    * @param count 总人数
3    * @param k    每隔几个人杀掉
4    * @return
5    */
```

```
6   public static Integer[] solution(int count, int k) {
7     Integer live[] = new Integer[Math.min(count, k - 1)];
8     if (count < k) {
9       int index = 0;
10      while (index < count) {
11        live[index++] = index;
12      }
13      return live;
14    }
15    List<Integer> mList = new ArrayList<>(count);
16    for (int i = 0; i < count; i++) {
17      mList.add(i + 1);
18    }
19
20    int point = 0;
21    int number = 0;
22    int total = count - k + 1;//记录总共删除的个数
23    while (true) {
24      if (total <= 0)
25        break;
26      if (point >= mList.size()) {
27        point = 0;
28      }
29      if (mList.get(point) < 0) {
30        point++;
31        continue;
32      }
33      number++;
34      if (number % k == 0) {
35        mList.set(point, -1);//如果是第k个，就把他变为负数
36        total--;
37        continue;
38      }
39      point++;
40    }
41    int index = 0;
42    for (int i = 0; i < mList.size(); i++) {
43      if (mList.get(i) > 0)
44        live[index++] = mList.get(i);
45    }
46    return live;
47  }
```

第35行该删除的我们没有删除，直接把他变为-1。在29行统计的时候如果为负数表示已经被删除了，就直接跳过，执行下一轮循环。第42-45行把最后没有被删除的放到数组live中。

3，链表实现

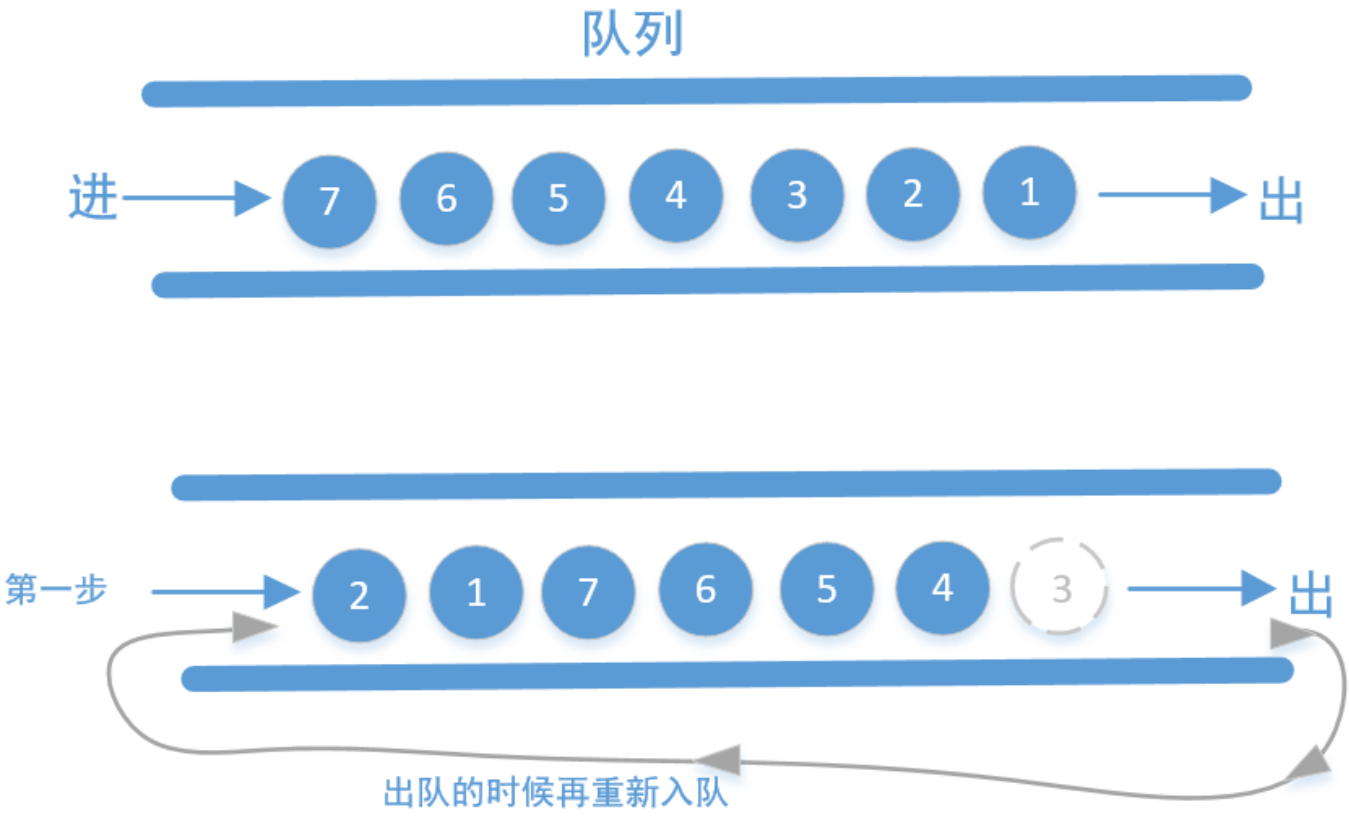
一般来说链表的断开要比数组的删除效率要高一些，因为数组删除某个元素之后，它后面的元素都还要往前移。使用链表会更简单一些，我们可以把它想象为一圈人大家都手牵着手，然后再一个个报数，当报到k的时候就自动退出，退出的时候左右两边人的手要牵到一块重新构成一个新的环，代码很简单，我们看下

```
1   public static Integer[] solution(int count, int k) {
2     Integer live[] = new Integer[Math.min(count, k - 1)];
3     if (count < k) {
4       int index = 0;
5       while (index < count) {
6         live[index++] = index;
7       }
8       return live;
9     }
10    LinkedList<Integer> mList = new LinkedList<>();
11    for (int i = 0; i < count; i++) {
12      mList.addLast(i + 1);
```

```
    }

    int point = 0;
    int number = 0;
    while (mList.size() >= k) {
      number++;
      if (point >= mList.size()) {
        point = 0;
      }
      if (number % k == 0) {
        mList.remove(point);
        continue;
      }
      point++;
    }
    return mList.toArray(live);
  }
```

队列实现：

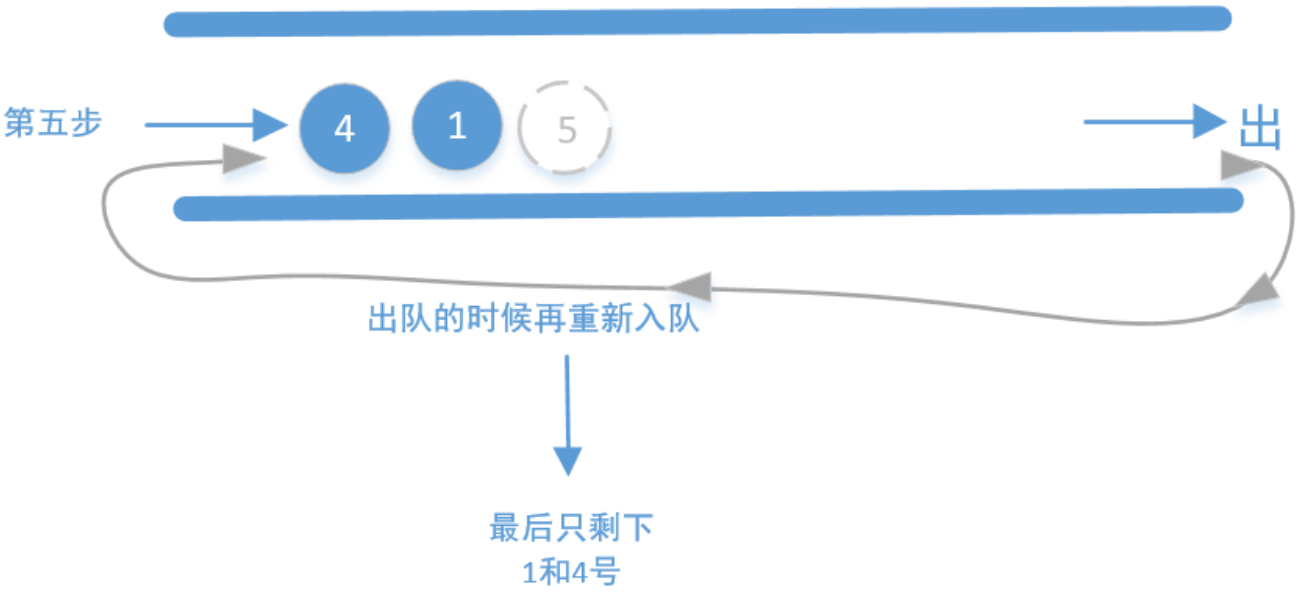
除了上面说的数组和链表以外，我们还可以使用队列。这个实现起来也非常简单。我们把所有的元素全部入队，然后再一个个出队，出队的时候记录出队的个数，如果不是第k个就让他重新入队，如果是第k个就不用了入队了，然后下一个出队的再重新从1开始计算。我们还是以7来画个图看一下。











我们先来看一下代码，队列就是用之前写的359，数据结构-3,队列中的双端队列。

```
1   /**
```

```
 * @param count 总人数
 * @param k    每隔几个人杀掉
 * @return
 */
public static Integer[] solution(int count, int k) {
  Integer live[] = new Integer[Math.min(count, k - 1)];
  if (count < k) {
    int index = 0;
    while (index < count) {
      live[index++] = index;
    }
    return live;
  }
  MyQueue<Integer> queue = new MyQueue<>(count + 1);
  for (int i = 0; i < count; i++) {
    queue.addLast(i + 1);
  }

  int number = 1;
  while (queue.size() >= k) {
    Integer item = queue.removeFirst();
    if (number % k == 0) {
      number = 1;
      continue;
    }
    queue.addLast(item);
    number++;
  }
  int index = 0;
  while (!queue.isEmpty()) {
    live[index++] = queue.removeFirst();
  }
  return live;
}
```

二，只留下一个

上面我们讲的是每到第K个删除，如果count大于等于k的话，最终会留下k-1个。但对这题还有另一个版本，就是无论多少个，最后只留下1个，就是说如果数量小于k个的时候我们继续循环删除，直到留下最后一个的为止。原理和上面非常类似，只不过当删除到最后小于K个的时候我们还要继续循环即可。图就不再画了，我们就用最后双端队列这种实现来改一下。

1，双端队列解决

```
public static Integer solution(int count, int k) {
  MyQueue<Integer> queue = new MyQueue<>(count + 1);
  for (int i = 0; i < count; i++) {
    queue.addLast(i + 1);
  }
  int number = 1;
  while (queue.size() > 1) {
    Integer item = queue.removeFirst();
    if (number % k == 0) {
      number = 1;
      continue;
    }
    queue.addLast(item);
    number++;
  }
  return queue.removeFirst();
}
```

2，递归解决

我们用f(n,k)表示有n个人，第k个出列，最后列出的人的编号。

那么f(n-1,k)就表示有n-1个人，第k个出列，最后列出的人的编号。

所以我们可以找到递归的公式f(n,k)=f(n-1,k)+k；也就是说n-1个人组成的环相对于n个人组成的环相当于顺时针旋转了k个单位。因为是环形的，当超出环的大小的时候我们要对它求余，所以为了防止越界问题我们要这样写f(n,k)=(f(n-1,k)+k)%n。

当f(1,k)的时候就表示剩下最后一个元素了，我们直接返回即可。

我们来看下代码

```
public static int solution(int n, int k) {
  return helper(n, k) + 1;
}

public static int helper(int n, int m) {
  if (n == 1)
    return 0;
  return (helper(n - 1, m) + m) % n;
}
```

因为人的编号是从1开始的，所以这里要加1。当然我们还可以再来改一下，这样就不用在加1，就可以直接返回了。

```
public static int solution(int n, int k) {
  if (n == 1)
    return n;
  return (solution(n - 1, k) + k - 1) % n + 1;
}
```

3，非递归写法

看明白了上面的递归的思路，我们还可以把它改为非递归的写法

```
public static int solution(int n, int k) {
  int m = 0;
  for (int i = 2; i <= n; i++) {
    m = (m + k) % i;
  }
  return m + 1;
}
```

他的原理是这样的，从前往后推，如果当n=i的时候，最终留下的是m，那么当n=i+1的时候，最终留下的就是m+k，考虑到m+k可能大于环的长度，所以要对m+k进行求余，结果就是（m+k）%i，一直循环到i等于n就是最终结果。

总结:

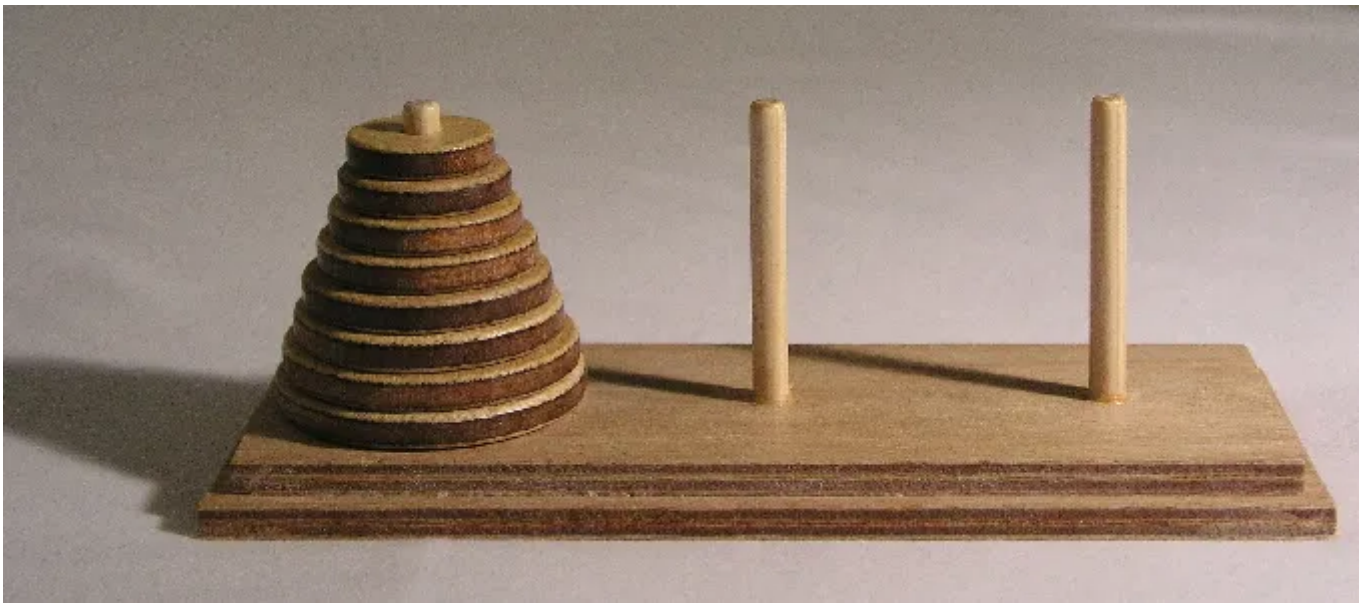
这题只要是学过编程的大多数应该都听过，无论是使用数组，链表，还是队列都很容易解决，具有一定的代表性，希望大家能够熟练掌握。

362，汉诺塔

原创 山大王wld 数据结构和算法 5月14日

收录于话题
#算法图文分析
96个 >

算法题尽量做到难易结合，在我们看一些难题的时候偶尔也可以看一些非常简单又非常容易看懂的题，要不然算法题一直看不懂，容易打击大家的兴趣，今天来看一道非常简单又非常常见的算法题，就是汉诺塔问题，这道题我想只要是学过编程的同学都应该知道的。



关于汉诺塔的传说

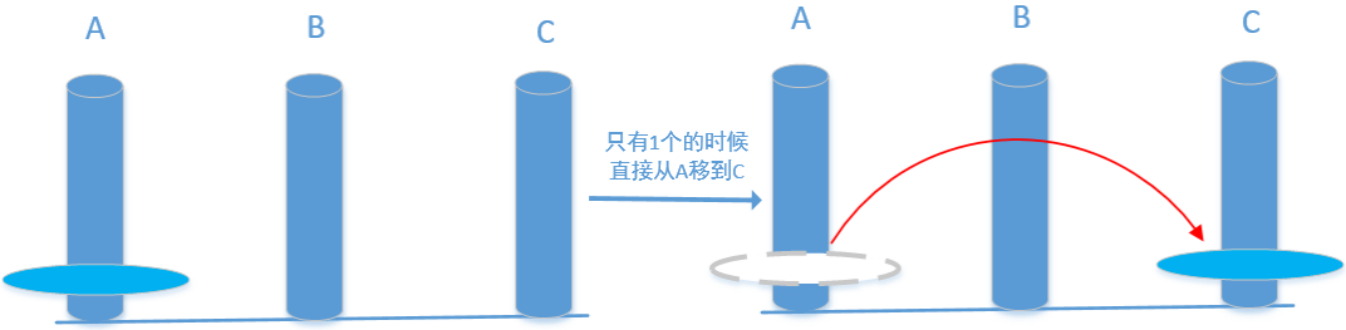
汉诺塔：汉诺塔（又称河内塔）问题是源于印度一个古老传说的益智玩具。大梵天创造世界的时候做了三根金刚石柱子，在一根柱子上从下往上按照大小顺序摞着64片黄金圆盘。大梵天命令婆罗门把圆盘从下面开始按大小顺序重新摆放在另一根柱子上。并且规定，在小圆盘上不能放大圆盘，在三根柱子之间一次只能移动一个圆盘。

汉诺塔还有另外一个版本

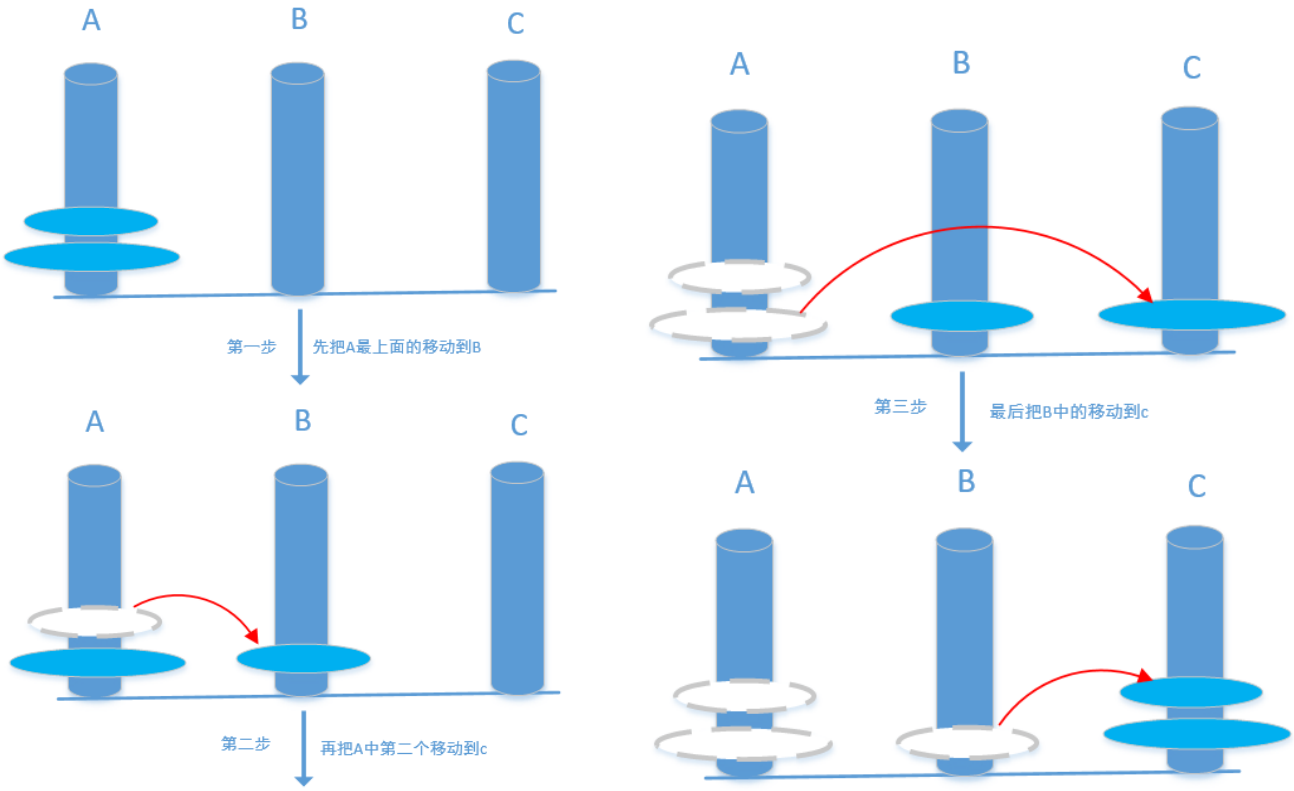
汉诺塔(Towers of Hanoi)是法国人M.Claus(Lucas)于1883年从泰国带至法国的，河内为越战时北越的首都，即现在的胡志明市；1883年法国数学家 Edouard Lucas曾提及这个故事，据说创世纪时Benares有一座波罗教塔，是由三支钻石棒（ Pag）所支撑，开始时神在第一根棒上放置64个由上至下依由小至大排列的金盘（Disc） ，并命令僧侣将所有的金盘从第一根石棒移至第三根石棒，且搬运过程中遵守大盘子在小盘子之下的原则，若每日仅搬一个盘子，则当盘子全数搬运完毕之时，此塔将毁损， 而也就是世界末日来临之时

我们这里不去追本溯源，追究他哪个版本是正确的，我们知道有这样一道题就行，直接文字叙述可能不太直观，我们画个图来分析一下

1，当只有一个的时候



2，当只有2个的时候



3，当只有3个的时候

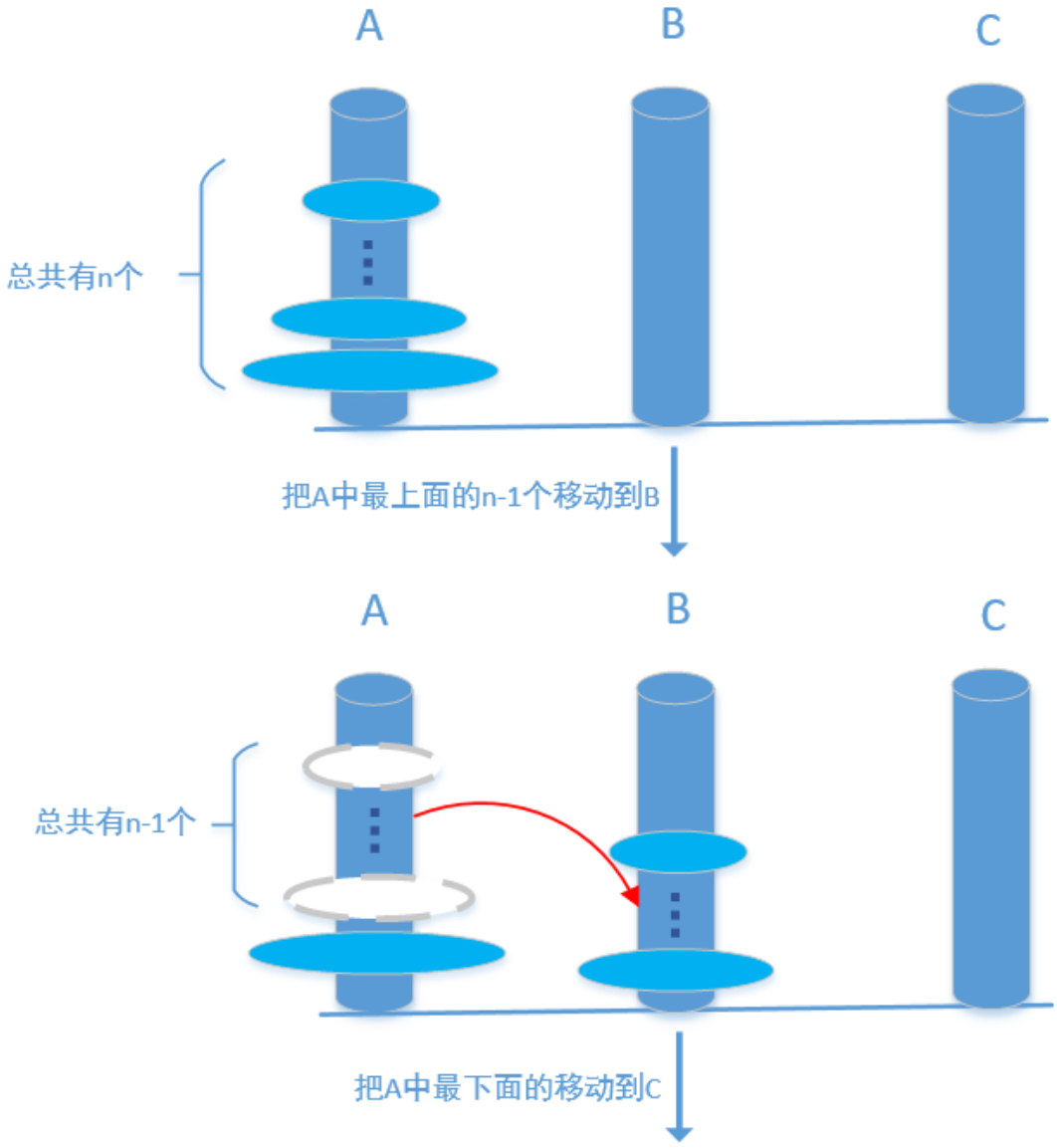
当n等于3的时候移动的步数就比较多了，我们就不再画图了，我们来看一个动画

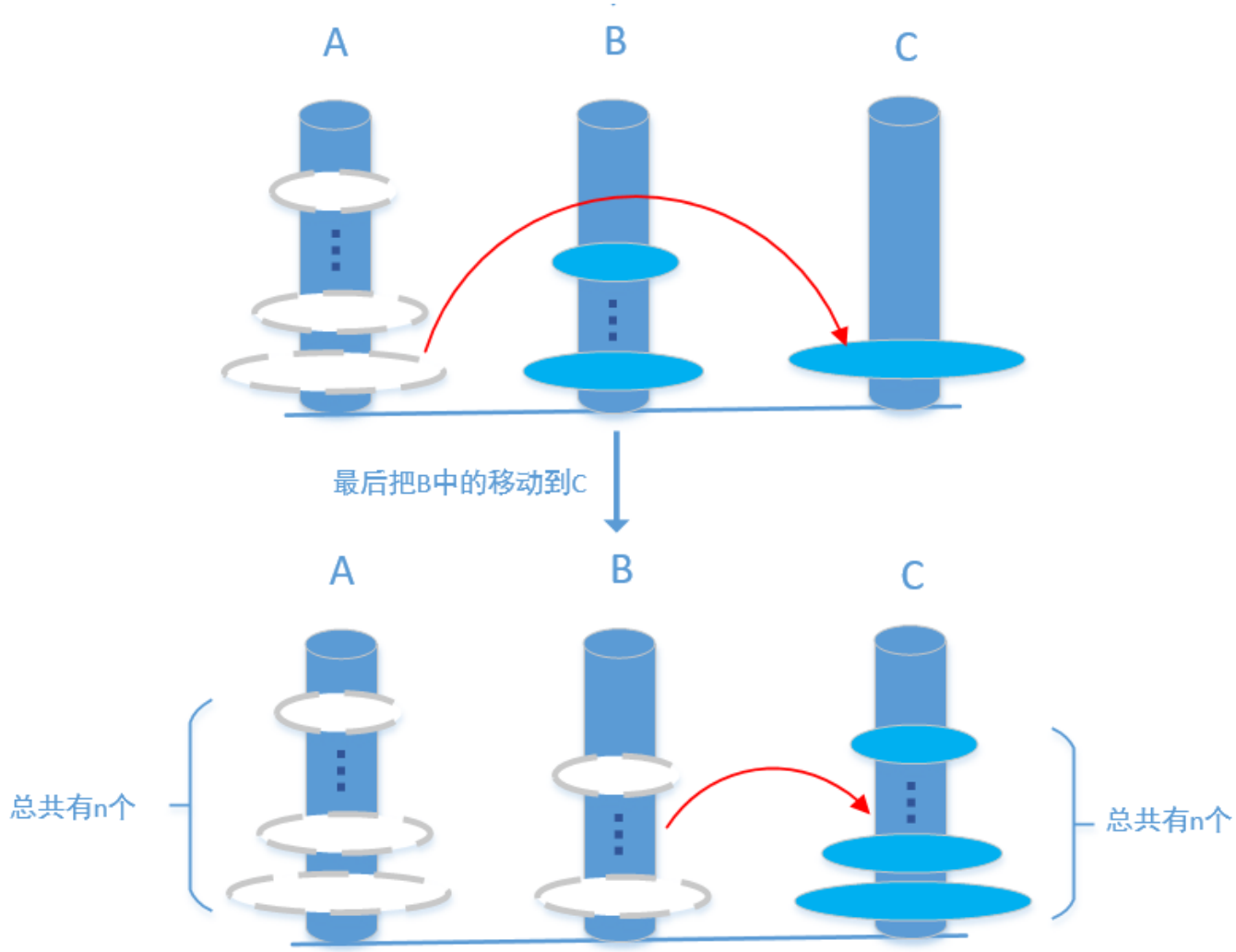


4，当只有4个的时候



5，当有n个的时候





这里主要还是考察对递归的理解，其实递归我们没必要把每一步都推出来，我们只需要找到他的规律和边界条件就行了，就像前面我们讲356，青蛙跳台阶相关问题的时候，我们提到了斐波那契数列，我们只需要找到斐波那契的规律f(n)=f(n-1)+f(n-2)，和他的边界条件f(0)=f(1)=1我们就可以写出代码了，其他的我们就不用管了。

故事篇

这里我来给大家讲个故事吧，很久很久以前有一个人，名叫“孤独求败”，他和家人住在山下过着与世无争的生活，就这样每天日出而作日落而息。有一天他从山上砍柴回来，突然看到家人遇害，他悲痛欲绝，于是决定给家人报仇，但他武艺不行，而仇人的功夫很高，如果硬拼不但报不了仇，而且还会白白丢了性命，所以他决定先去习武。

那天他来到一座山上，看到一个老和尚，向他说明了缘由，老和尚听闻之后怒火中烧，一脚把门前的一个千斤石狮踢开，想帮孤独求败报仇，但又怕得罪他的仇人导致引火烧身，所以他就对孤独求败说：“此仇只有你自己能报”。

孤独求败：“可我功夫不行，根本报不了仇”。

老和尚；“别急，请跟我来”。

于是老和尚带他来到了一间很破旧的房子里，指着里面的一个满是灰尘的破柜子说；“武功秘籍就锁在这里面，你拿到钥匙打开它，把它拿回去自己修炼，等你功夫达到十级之后就可以找你的仇人报仇了”。

然后找来了他的大徒弟对孤独求败说，这是你的大师兄，钥匙你找他要，我要回去睡觉了。然后大师兄说钥匙锁在我自己的小盒子里了，我小盒子的钥匙在二师兄拿着，你先去找二师兄把我这小盒子的钥匙拿到，我把这个小盒子打开，就可以把锁柜子的钥匙给你了。

于是孤独求败又去找到了二师兄，然后二师兄说，大师兄的钥匙被我锁在了我自己的小盒子里了，我自己的小盒子的钥匙在三师兄拿着……

就这样，孤独求败一直找下去，直到找到最小的100师兄拿到钥匙，然后交给第99师兄，打开第99师兄的小盒子，拿到第98师兄的钥匙，又交给第98师兄……一直到大师兄，然后大师兄拿到钥匙打开自己的小盒子，把藏有武功秘籍的柜子钥匙交给孤独求败，这样孤独求败就拿到了武功秘籍，回家修炼去了……

孤独求败从大师兄开始到拿到武功秘籍的过程其实就是**递归**。

读懂了上面的故事，再来看汉诺塔问题就容易多了，我们来看下代码。

```
//表示的是把n个圆盘成功的从A移动到C
public static void hanoi(int n, char A, char B, char C) {
  if (n == 1) {
    //如果只有一个，直接从A移动到C即可
    System.out.println("从" + A + "移动到" + C);
  } else {
    //表示先把n-1个圆盘成功从A移动到B
    hanoi(n - 1, A, C, B);
    //把第n个圆盘从A移动到C
    System.out.println("从" + A + "移动到" + C);
    //表示把n-1个圆盘再成功从B移动到C
    hanoi(n - 1, B, A, C);
  }
}
```

我们还可以把每个柱子看作是一个栈，大的在下面，小的在上面，所以我们也可以使用栈来实现

```
//stackA 源柱  stackB 借助柱  stackC目的柱
public static void move(Stack stackA, Stack stackB, Stack stackC, int n) {
  if (n == 1) {
    stackC.push(stackA.pop());
  } else {
    move(stackA, stackC, stackB, n - 1);
    stackC.push(stackA.pop());
    move(stackB, stackA, stackC, n - 1);
  }
}
```

如果想要统计总共移了多少次，可以使用公式(2^n)-1，代码如下

```
public static long hanoiCount(int n) {
  return (1L << n) - 1;
}
```

当n=63的时候，我们得到9223372036854775807，当n=64的时候就已经出现了数字溢出。

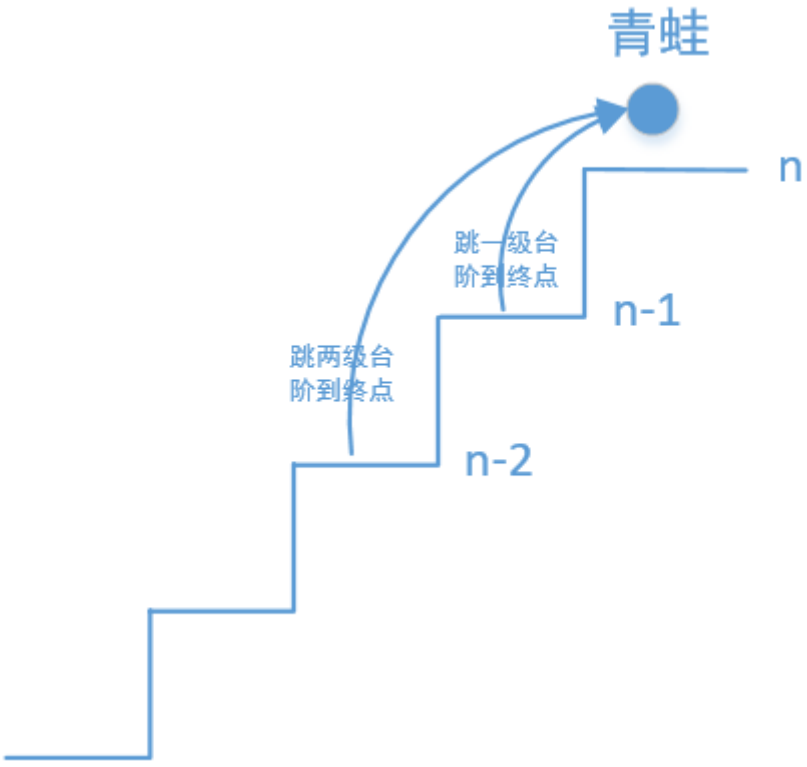
356，青蛙跳台阶相关问题

原创 山大王wld 数据结构和算法 5月6日

收录于话题
#算法图文分析
96个 >

问题一：
一只青蛙一次可以跳上一级台阶，也可以跳上二级台阶，求该青蛙跳上一个n级的台阶总共需要多少种跳法。

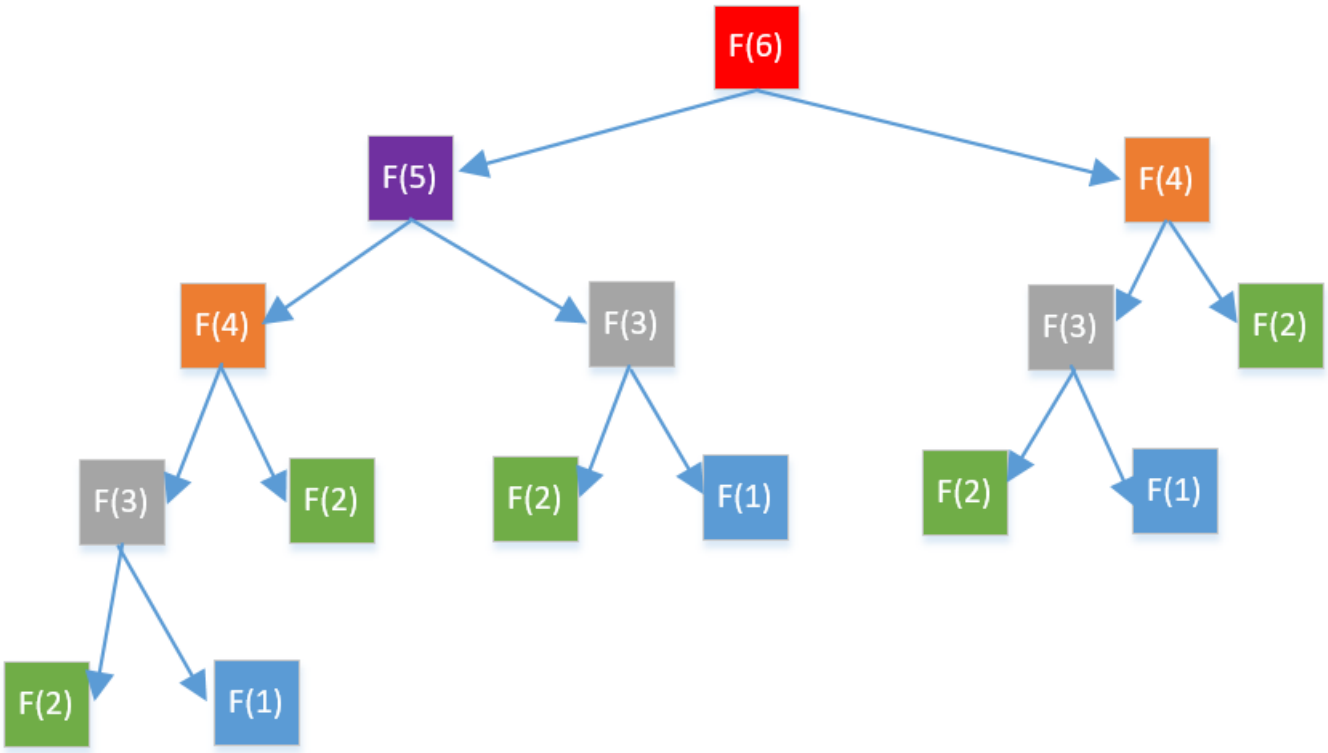
我们来分析一下：
当n等于1的时候，只需要跳一次即可，只有一种跳法，记f(1)=1
当n等于2的时候，可以先跳一级再跳一级，或者直接跳二级，共有2种跳法，记f(2)=2
当n等于3的时候，他可以从一级台阶上跳两步上来，也可以从二级台阶上跳一步上来，所以总共有f(3)=f(2)+f(1)；
同理当等于n的时候，总共有f(n)=f(n-1)+f(n-2)(这里n>2)种跳法。



所以大家一看就知道这就是个斐波那契数列，只不过有一点不同的是斐波那契数列一般是以1,1,2,3,5,8,13……开始的，而我们这是以1,2,3,5,8,13……开始的，少了最前面的一个1。最代码很简单

```
1   public static int f(int n) {
2     if (n < 3)
3       return n;
4     return f(n - 1) + f(n - 2);
5   }
```

我们以计算f(6)为例画个图看一下计算的过程



我们看到递归会重复计算已经计算过的值，效率明显不是很高，我们还可以把计算过的值储存起来，防止重复计算，我们来看下代码

```
1   private static int f2(int n, HashMap<Integer, Integer> map) {
2     if (n < 3) return n;
3     if (map.containsKey(n))
4       return map.get(n);
5     int first = f2(n - 1, map);
6     int second = f2(n - 2, map);
7     int sum = first + second;
8     map.put(n, sum);
9     return sum;
10  }
```

我们还可以把递归改为非递归的形式，看下代码

```
1   private static int f3(int n) {
2     if (n < 3)
3       return n;
4     int first = 1, second = 2, sum = 0;
5     while (n-- > 2) {
6       sum = first + second;
7       first = second;
8       second = sum;
9     }
10    return sum;
11  }
```

上面3种方式都可以实现青蛙跳台阶问题，那么哪种效率更高呢，我们来找个比较大的数据测试一下

```
1   public static void main(String[] args) {
2     int step = 45;
3     long time = System.nanoTime();
4     System.out.println(f(step));
5     System.out.println("代码优化前时间：" + (System.nanoTime() - time));
6     time = System.nanoTime();
7     System.out.println(f2(step, new HashMap<Integer, Integer>()));
8     System.out.println("代码优化后时间：" + (System.nanoTime() - time));
9     time = System.nanoTime();
```

```
10      System.out.println(f3(step));
11      System.out.println("代码非递归时间：" + (System.nanoTime() - time));
12    }
```

来看一下运行的时间

```
1   1836311903
2   代码优化前时间：2221741900
3   1836311903
4   代码优化后时间：108000
5   1836311903
6   代码非递归时间：17600
```

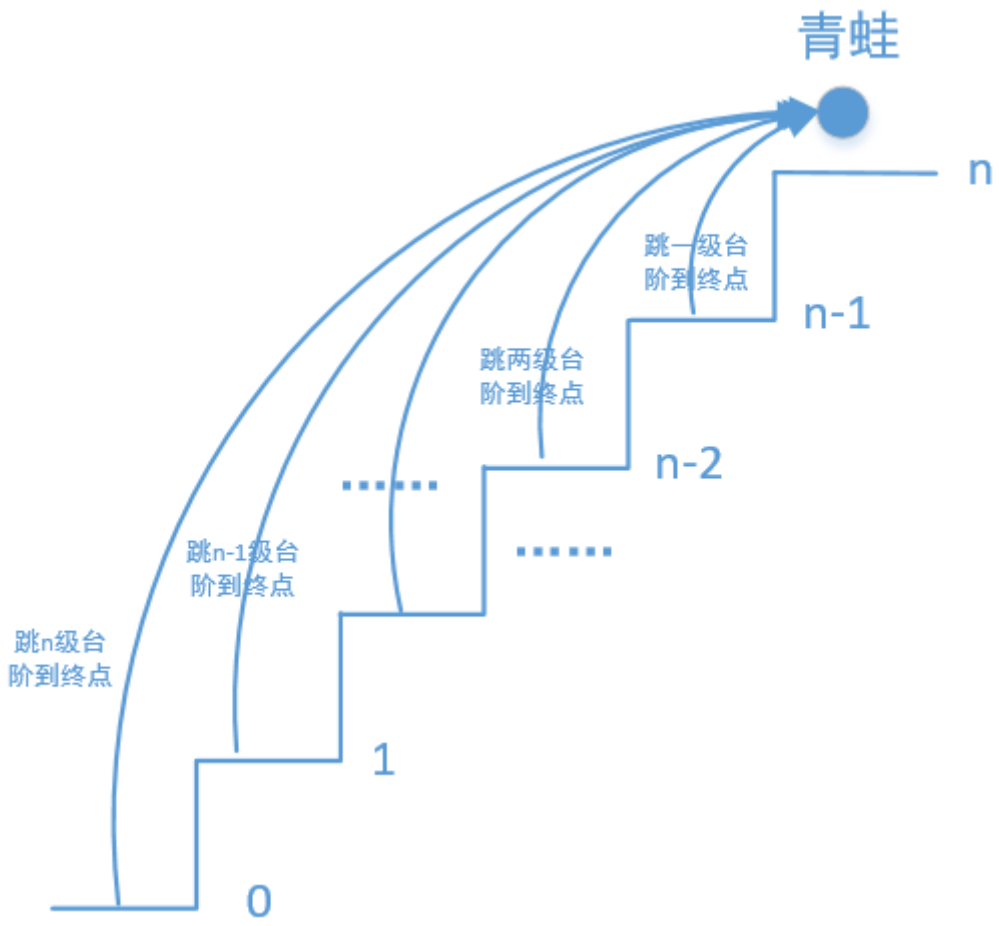
我们看到递归优化之前运行时间是非常长的，优化之后时间大幅下降，但对于非递归来说又稍逊色了一些。

问题二：

一只青蛙一次可以跳上一级台阶，也可以跳上二级台阶……，也可以跳n级，求该青蛙跳上一个n级的台阶总共需要多少种跳法。

我们来分析一下

一只青蛙要想跳到n级台阶，可以从一级，二级……，也就是说可以从任何一级跳到n级，



所以递推公式我们很容易就能想到

f(n)=f(n-1)+f(n-2)+……+f(2)+f(1)+f(0)；最后这个f(0)是可以去掉的，因为0级就相当于没跳，所以f(0)=0;

然后我们把f(0)去掉在转换一下：

f(n-1)=f(n-2)+f(n-3)+……+f(2)+f(1);

所以f(n)=f(n-1)+f(n-1)=2*f(n-1)；他是一个等比数列，且f(1)=1;

我们我们可以得出f(n)=2^(n-1)；代码如下

```
private static int f4(int n) {
  if (n == 1)
    return 1;
  return f4(n - 1) * 2;
}
```

或者还可以改为非递归的

```
private static int f5(int n) {
  if (n == 1)
    return 1;
  return 1 << (n - 1);
}
```

问题三：

一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上m级。求该青蛙跳上一个n级的台阶总共有多少种跳法

这道题我们要分开讨论：

1，如果n<=m；因为只能往上跳不能往下跳，所以大于n的都不可以跳，如果跳了就直接超过了，只能跳小于等于n的数字，那么这个问题就直接退到问题2了。

2，如果n>m；我们要想跳到n级台阶，我们可以从n-1级跳一步上来，或者从n-2级跳两步上来……，或者从n-m级跳m步上来，所以我们可以找出递归公式

f(n) = f(n-1) + f(n-2) + f(n-3) + ... + f(n-m);

进一步可以推出：

f(n-1) = f(n-2) + f(n-3) + ... + f(n-m) + f(n-m-1);

化简结果为：

f(n) = 2f(n-1) - f(n-m-1); **(n>m)**

所以代码我们要分为两部分，一部分是n>m，另一部分是n<=m，我们来看下代码

```
public static int f6(int n, int m) {
  if (n <= 1)
    return 1;
  //总台阶大于跳的最高级台阶
  if (n > m)
    return 2 * f6(n - 1, m) - f6(n - 1 - m, m);
  //回退到上面的问题二了
  return 2 * f6(n - 1, n);
}
```

斐波那契数列又称黄金分割数列，他有很多的特性，比如兔子的繁殖，他的通项公式如下

$$a_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

592，位运算解颠倒二进制位

原创 博哥 数据结构和算法 8月9日

问题描述

来源：LeetCode第190题
难度：简单

颠倒给定的32位无符号整数的二进制位。

提示：

- 请注意，在某些语言（如 Java）中，没有无符号整数类型。在这种情况下，输入和输出都将被指定为有符号整数类型，并且不应影响您的实现，因为无论整数是有符号的还是无符号的，其内部的二进制表示形式都是相同的。
- 在 Java 中，编译器使用二进制补码记法来表示有符号整数。因此，在上面的 示例 2 中，输入表示有符号整数 -3，输出表示有符号整数 -1073741825。

示例 1：

> **输入**:
> 00000010100101000001111010011100
> **输出**:
> 00111001011110000010100101000000
>
> **解释**: 输入的二进制串 00000010100101000001111010011100 表示无符号整数 43261596，因此返回 964176192，其二进制表示形式为 00111001011110000010100101000000。

示例 2：

> **输入**:
> 11111111111111111111111111111101
> **输出**:
> 10111111111111111111111111111111
>
> **解释**：输入的二进制串 11111111111111111111111111111101 表示无符号整数 4294967293，因此返回 3221225471 其二进制表示形式为 10111111111111111111111111111111 。

提示：

- 输入是一个长度为32的二进制字符串

问题分析

这题是让把一个int类型的二进制位反过来，比如abcd，要把他变成dcba。最简单的一种方式就是通过一个循环，每次循环的时候把n的最后一位数字（二进制的）截取掉，放到一个新的数字中的末尾，可以看下视频



原理比较简答，我们来看下代码

```
public int reverseBits(int n) {
  int res = 0;
  for (int i = 0; i < 32; i++) {
    //res先往左移一位，把最后一个位置空出来，
    //用来存放n的最后一位数字
    res <<= 1;
    //res加上n的最后一位数字
    res |= n & 1;
    //n往右移一位，把最后一位数字去掉
    n >>= 1;
  }
  return res;
}
```

我们知道在java中int类型是32位的，我们还可以使用一个临时变量res，把

- n的二进制位中右边第1位放到res二进制位右边第32位，
- n的二进制位中右边第2位放到res二进制位右边第31位，
- n的二进制位中右边第3位放到res二进制位右边第30位，
- ……

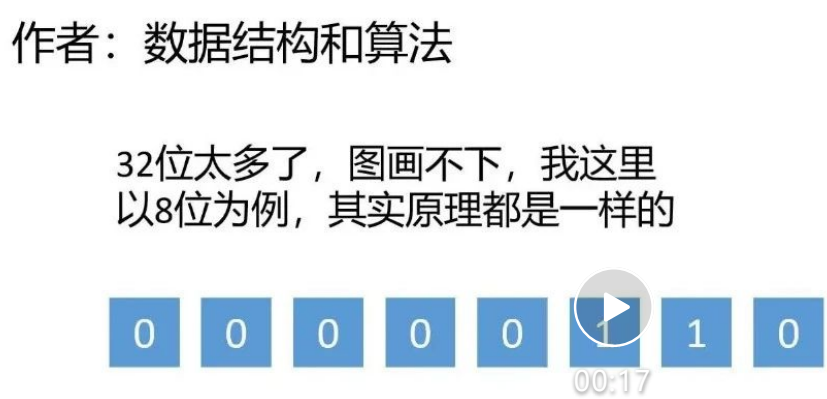
这样最终也可以实现，原理比较简单，来看下代码

```
public int reverseBits(int n) {
  int res = 0;
  //把低16位移到高16上
  for (int i = 0; i < 16; i++) {
    res |= (n & (1 << i)) << (31 - i * 2);
```

```
    }
    //把高16位移到低16位上
    for (int i = 16; i < 32; i++) {
        res |= (n & (1 << i)) >>> (i * 2 - 31);
    }
    return res;
}
```

我们还可以不使用循环，就是前面16个和后面16个交换，然后前面16个和后面16个中的8个8个在交换……直到两两交换为止。

看下视频



代码如下

```
public int reverseBits(int n) {
    n = (n >>> 16) | (n << 16);
    n = ((n & 0xff00ff00) >>> 8) | ((n & 0x00ff00ff) << 8);
    n = ((n & 0xf0f0f0f0) >>> 4) | ((n & 0x0f0f0f0f) << 4);
    n = ((n & 0xcccccccc) >>> 2) | ((n & 0x33333333) << 2);
    n = ((n & 0xaaaaaaaa) >>> 1) | ((n & 0x55555555) << 1);
    return n;
}
```

除了上面介绍的3种解法以外，还有一种解决方式就是每2位之间相互交换，然后是每4位之间相互交换，接着是每8位之间相互交换，但最后不是每16位之间交换，而是把32位分为4个8位，这4个8位之间相互交换，具体也可以看下

364，位1的个数系列（一）

385，位1的个数系列（二）

402，位1的个数系列（三）

来看下代码（经常看源码的同学会发现，这个就是源码Integer中的代码。）

```
public static int reverseBits(int var0) {
    var0 = (var0 & 1431655765) << 1 | var0 >>> 1 & 1431655765;
    var0 = (var0 & 858993459) << 2 | var0 >>> 2 & 858993459;
    var0 = (var0 & 252645135) << 4 | var0 >>> 4 & 252645135;
```

```
  var0 = var0 << 24 | (var0 & '\uff00') << 8 | var0 >>> 8 & '\uff00' | var0 >>> 24;
  return var0;
}
```

上面数字的二进制位如下：

```
1431655765的二进制是：01010101 01010101 01010101 01010101
858993459 的二进制是：00110011 00110011 00110011 00110011
252645135 的二进制是：00001111 00001111 00001111 00001111
'\uff00'  的二进制是：00000000 00000000 11111111 00000000
```

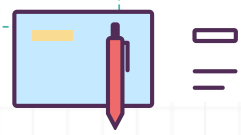
往期推荐

- 560，位运算解只出现一次的数字 II
- 499，位运算解只出现一次的数字 III
- 495，位运算等多种方式解找不同
- 494，位运算解只出现一次的数字

565，多种方式解2的幂

原创 博哥 数据结构和算法 1周前

> Yesterday is history, tomorrow is a mystery. But today is a gift.
>
> 昨日已成往事，未来还未可知。但是今天是上天的馈赠。



问题描述

来源：LeetCode第231题
难度：简单

给你一个**整数n**，请你判断该整数是否是2的幂次方。如果是，返回true；否则，返回false。

如果存在一个**整数x**使得n==2^x，则认为n是2的幂次方。

示例 1：

> **输入**：n = 1
> **输出**：true
> **解释**：2^0 = 1

示例 2：

> **输入**：n = 16
> **输出**：true
> **解释**：2^4 = 16

示例 3：

> **输入**：n = 3
> **输出**：false

示例 4：

> **输入**：n = 4
> **输出**：true

示例 5：

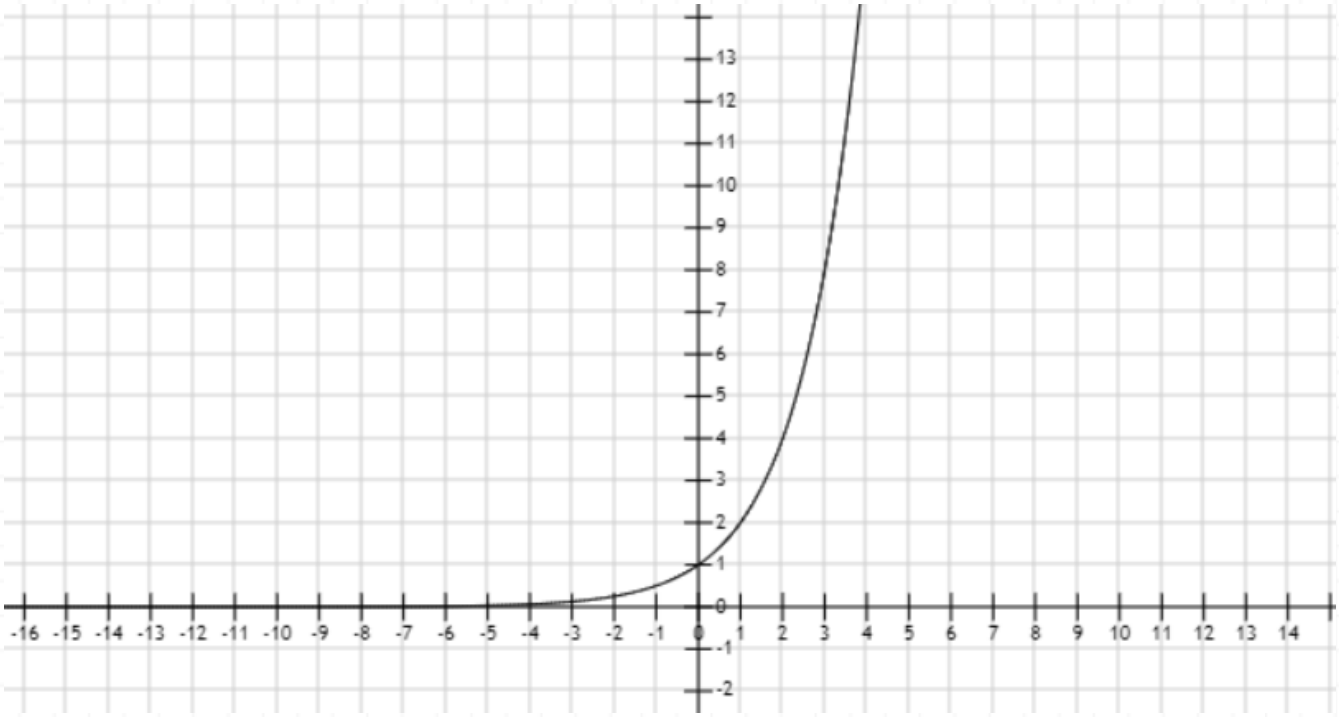
> **输入**：n = 5
> **输出**：false

提示：

- -2^31 <= n <= 2^31 - 1

递归解决

这题让判断一个数是否是2的幂次方，来看一下2的幂次方函数图像，可以看到2的幂次方一定是大于0的。



因为题中说了n和x都是**整数**，那么n如果是2的幂次方，他只能是2^0，2^1，2^2,……2^31。所以很容易想到的就是判断n是否是偶数，如果是偶数就一直除以2，直到是奇数为止，最后在判断这个奇数是否等于1，如果等于1返回true，否则返回false，比如8→4→2→1。代码如下

```
1   public boolean isPowerOfTwo(int n) {
2     //非正数不可能是2的幂，直接返回false
3     if (n <= 0)
4       return false;
5     //如果是偶数就一直除以2，直到不是偶数为止
6     while (n % 2 == 0)
```

```
        n /= 2;
    //判断是否等于1
    return n == 1;
  }
```

位运算解决

通过上面分析我们知道，因为n和x都是整数，如果n是2的幂次方，那么n就只能是1，2，4，8，16……这样的数字。其实这些数字都有一个特点。

- 1的二进制位中只有一个1
- 2是1往左移一位，所以他也只有一个1
- 4是2往左移一位，所以他也只有一个1
- 8是4往左移一位，所以他也只有一个1
- ……

所以一个数的二进制位中如果只有一个1（符号位不算），那么这个数肯定是2的幂次方，前面我们讲过《425，剑指 Offer-二进制中1的个数》，列出了18种解法，我们可以随便找一种修改一下就是今天这题的答案，比如我们就拿第一种来修改一下，代码如下

```
public boolean isPowerOfTwo(int n) {
    //首先要保证n是大于0的，然后再判断n的
    //二进制位中1的个数是否等于1
    return n > 0 && hammingWeight(n) == 1;
}

//二进制中1的个数
public int hammingWeight(int n) {
    int count = 0;
    for (int i = 0; i < 32; i++) {
        if (((n >>> i) & 1) == 1) {
            count++;
        }
    }
    return count;
}
```

在425题中讲到第5种解法（具体可以看下《364，位1的个数系列（一）》）的时候，我们知道n&(n-1)实际上就是消去n的二进制位中最右边的1，如果n的二进制位中只有一个1，那么n&(n-1)的结果肯定是0，所以我们只需要判断n大于0的时候，n&(n-1)是否等于0即可，一行代码搞定。

```
public boolean isPowerOfTwo(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}
```

如果对位运算比较熟悉的同学应该能明白n和-n在二进制位中的区别，因为-n是n每一个都取反然后再加上1的结果，所以n和-n的区别就是n原来右边第一个1以及他右边的都不变，其他各位都是取反，具体我们来看下

```
public static void main(String args[]) {
    System.out.println("8的二进制：" + Util.bitInt32(8));
    System.out.println("-8的二进制：" + Util.bitInt32(-8));
```

```
        System.out.println();
        System.out.println("50的二进制：" + Util.bitInt32(50));
        System.out.println("-50的二进制：" + Util.bitInt32(-50));
        System.out.println();
        System.out.println("24的二进制：" + Util.bitInt32(24));
        System.out.println("-24的二进制：" + Util.bitInt32(-24));
    }
```

我们来看一下打印结果

```
8的二进制：   00000000 00000000 00000000 00001000
-8的二进制：  11111111 11111111 11111111 11111000

50的二进制：  00000000 00000000 00000000 00110010
-50的二进制： 11111111 11111111 11111111 11001110

24的二进制：  00000000 00000000 00000000 00011000
-24的二进制： 11111111 11111111 11111111 11101000
```

所以对于这道题来说，如果n是2的幂次方，在确定n大于0的情况下，只需要判断(n&-n)==n即可，也是一行代码搞定

```
public boolean isPowerOfTwo(int n) {
  return n > 0 && (n & -n) == n;
}
```

其实还有一种数学的方式，题中给出的条件是-2^31 <= n <= 2^31 - 1，所以我们可以找出在int范围类最大的2的幂次方，然后再判断这个数是否能被n整除

```
public boolean isPowerOfTwo(int n) {
  return n > 0 && Math.pow(2, 31) % n == 0;
}
```

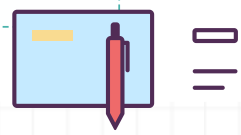
往期推荐

- 556，位运算解形成两个异或相等数组的三元组数目
- 512，反转二进制位
- 499，位运算解只出现一次的数字 III
- 494，位运算解只出现一次的数字

560，位运算解只出现一次的数字 II

原创 博哥 数据结构和算法 5月28日

> A lie can travel halfway around the world while the truth is still putting on its shoes.
>
> 当真理还正在穿鞋的时候，谎言就能走遍半个世界。



问题描述

来源：剑指 Offer 56 - II
难度：中等

在一个数组nums中除一个数字只出现一次之外，其他数字都出现了三次。请找出那个只出现一次的数字。

示例 1：

> **输入**：nums = [3,4,3,3]
> **输出**：4

示例 2：

> **输入**：nums = [9,1,7,9,7,9,7]
> **输出**：1

限制：

- 1 <= nums.length <= 10000
- 1 <= nums[i] < 2^31

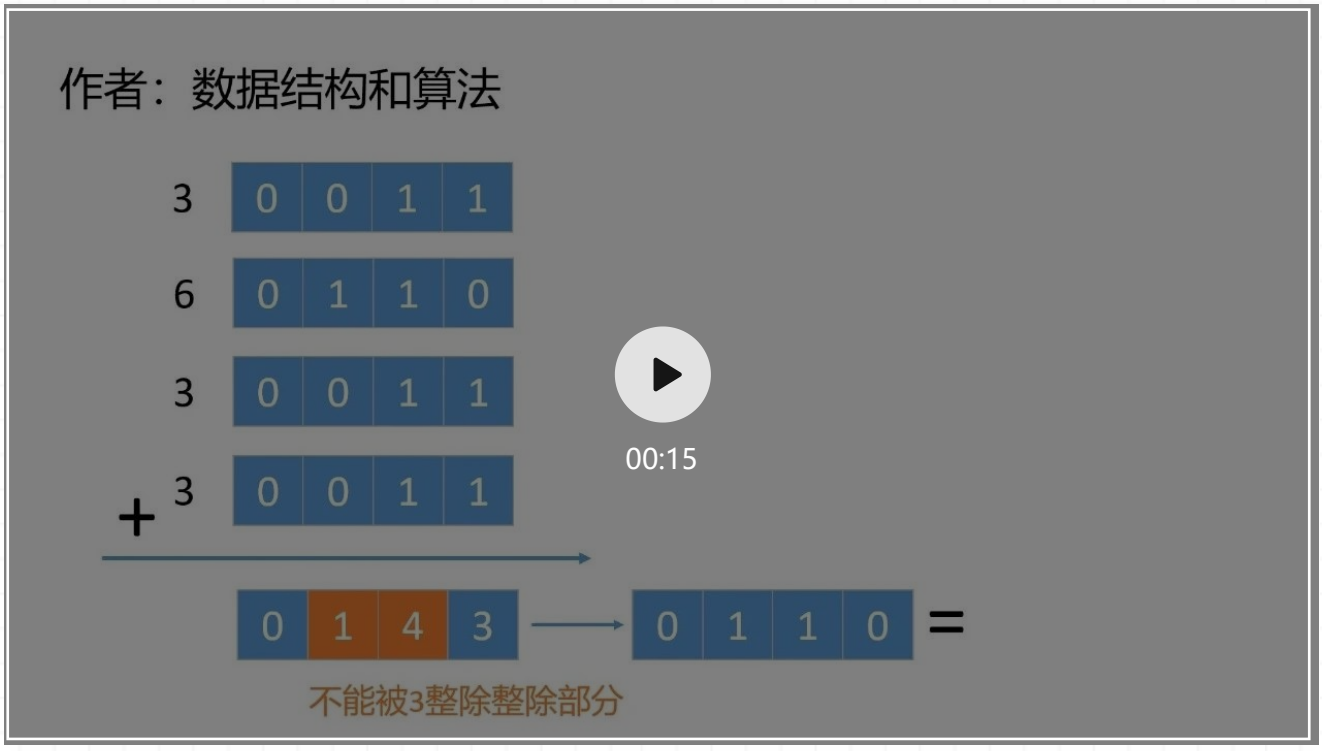
使用HashMap

这题说的很明白，只有一个数字出现了一次，其他的数字都出现了3次，找出那个出现一次的数字。最简单的一种方式就是使用HashMap统计每个数字出现的次数，因为只有一个数字出现一次，其他都出现3次，我们只需要返回那个出现一次的即可，原理比较简单，来看下代码

```
public int singleNumber(int[] nums) {
    Map<Integer, Integer> map = new HashMap<>();
    //先把数字存储到map中，其中key存储的是当前数字，value是
    //数字的出现的次数
    for (int num : nums) {
        map.put(num, map.getOrDefault(num, 0) + 1);
    }
    //最后在遍历map中的所有元素，返回value值等于1的
    for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
        if (entry.getValue() == 1)
            return entry.getKey();
    }
    return -1;
}
```

位运算解决

在java中int类型是32位，我们需要统计所有数字在某一位置的和能不能被3整除，如果不能被3整除，说明那个只出现一次的数字的二进制在那个位置是1……把32位全部统计完为止，来看个视频



再来看下代码

```
public int singleNumber(int[] nums) {
    //最终的结果值
    int res = 0;
    //int类型有32位，统计每一位1的个数
    for (int i = 0; i < 32; i++) {
        //统计第i位中1的个数
        int oneCount = 0;
        for (int j = 0; j < nums.length; j++) {
            oneCount += (nums[j] >>> i) & 1;
        }
```

```
        //如果1的个数不是3的倍数，说明那个只出现一次的数字
        //的二进制位中在这一位是1
        if (oneCount % 3 == 1)
            res |= 1 << i;
    }
    return res;
}
```

这题我们还可以扩展一下

一，如果只有一个数字出现一次，其他数字都出现偶数次，我们只需要把所有数字异或一遍即可。

因为异或有下面几条性质

- **a^a=0 任何数字和自己异或结果是0**
- **a^0=a 任何数字和0异或还是他自己**
- **a^b^c=a^c^b 异或运算具有交换律**

二，如果只有一个数字出现一次，其他数字都出现奇数次，我们可以用下面代码来解决。

```
// n是出现的次数
public int findOnce(int[] nums, int n) {
    int bitLength = 32;
    int res = 0;
    for (int i = 0; i < bitLength; i++) {
        int oneCount = 0;
        for (int j = 0; j < nums.length; j++) {
            oneCount += (nums[j] >>> i) & 1;
        }
        if (oneCount % n != 0)
            res |= (1 << i);
    }
    return res;
}
```

状态机1

按照题意的要求，我们定义一种运算如果某个数出现3次，通过这种运算就让他的结果变成0，也就是说周期是3。每个数都会有下面几种状态

- **出现0次**
- **出现1次**
- **出现2次**
- **出现3次**

因为周期是3，当出现3次的时候可以认为出现了0次，也就是下面几种状态

- **出现0次**
- **出现1次**
- **出现2次**

看到这里其实大家已经想到了，这不就是传说中的**3进制**吗。

在二进制中一个位置要么是1要么是0，只能表示一种状态，如果要表示3种状态我们可以使用两位数字来表示

我们选择

- **00表示出现0次**
- **01表示出现1次**
- **10表示出现2次**

但这里好像没有出现3次的，其实上面已经说了，出现3次的可以认为是出现0次。对于每一个数字，如果是0我们就不用了管他，只有是1的时候状态才会改变（这里数字展示会出现错乱，下面全是截图，如果看不清楚，可以点击放大）

3 种状态

存储	输入	结果	
ab	c	ab	（输入是1的时候，输出会变为下一个状态）
00	1	01	（这里是关键，如果只出现一次，结果会保存在b中，所以最后只需要返回b的值即可）
01	1	10	
10	1	00	
00	0	00	（输入是0的时候，输出不变）
01	0	01	
10	0	10	

所以可以列出下面公式

a= ~a&b&c|a&~b&~c

b= ~a&~b&c|~a&b&~c

来看下代码

```java
public int singleNumber(int[] nums) {
    int a = 0, b = 0;
    for (int c : nums) {
        //防止a的值被修改，在计算b的时候有影响，
        //这里在b计算完之后再对a赋值
        int tempa = ~a & b & c | a & ~b & ~c;
        b = ~a & ~b & c | ~a & b & ~c;
        a = tempa;
    }
    return b;
}
```

状态机2

上面我们选择的是 00，01，01 三种状态。那么能不能选择其他状态能，当然是可以的，比如我们选择 00，01，11 三种状态

```
3 种状态

存储    输入    结果

 ab      c      ab    （输入是1的时候，输出会变为下一个状态）

 00      1      01    （这里是关键，如果只出现一次，结果会保存在b中，所以最后只需要返回b的值即可）

 01      1      11

 11      1      00


 00      0      00    （输入是0的时候，输出不变）

 01      0      01

 11      0      11


所以可以列出下面公式

a= ~a&b&c|a&b&~c

b= ~a&~b&c|~a&b&c|~a&b&~c|a&b&~c
```

来看下代码

```java
public int singleNumber(int[] nums) {
    int a = 0, b = 0;
    for (int c : nums) {
        //防止a的值被修改，在计算b的时候有影响，
        //这里在b计算完之后再对a赋值
        int tempa = ~a & b & c | a & b & ~c;
        b = ~a & ~b & c | ~a & b & c | ~a & b & ~c | a & b & ~c;
        a = tempa;
    }
    return b;
}
```

状态机3

除了上面提到的使用两位数字，难道就不能使用三位数字吗，当然也是可以的，比如我们使用3个数字 001，010，100 来表示，我们来看一下

3 种状态

存储	输入	结果	
abc	d	abc	（输入是1的时候，输出会变为下一个状态）
001	1	010	（这里是关键，如果只出现一次，结果会保存在b中，所以最后只需要返回b的值即可）
010	1	100	
100	1	001	
001	0	001	（输入是0的时候，输出不变）
010	0	010	
100	0	100	

所以可以列出下面公式

```
a = ~a & b & ~c & d | a & ~b & ~c & ~d
b = ~a & ~b & c & d | ~a & b & ~c & ~d
c = a & ~b & ~c & d | ~a & ~b & c & ~d
```

来看下代码

```java
public int singleNumber(int[] nums) {
  //因为默认是001，所以c的位置我们让他全部变为1
  int a = 0, b = 0, c = - 1;
  for (int d : nums) {
    int tempa = ~a & b & ~c & d | a & ~b & ~c & ~d;
    int tempb = ~a & ~b & c & d | ~a & b & ~c & ~d;
    c = a & ~b & ~c & d | ~a & ~b & c & ~d;
    a = tempa;
    b = tempb;
  }
  return b;
}
```

看到这里大家是不是有想法了，上面选择两位，三位都可以计算，那么四位能不能计算呢，其实也是可以的。在java中int是32位，只要不是选择1位，无论你选择2位还是28位还是32位其实都是可以的，只要满足让他出现3次的时候回到初始状态即可。那这样写下去答案就比较多了，这里就不在一直往下写了,如果感兴趣的大家可以试着写下。

总结

之前我在LeetCode上写这题解的时候，很多同学评论不知道公式怎么推导的，这里再来补充一下

其实公式的推理很简单，就拿我上面写的状态机 2 来说

```
3 种状态

存储     输入     结果

 ab       c       ab

 00       1       01

 01       1       11 (注意这个地方a是1)，所以a= ~a&b&c

 11       1       00


 00       0       00

 01       0       01

 11       0       11 (注意这个地方a是1)，所以a=a&b&~c
```

我们看到有两个地方a是1，所以 a= ~a&b&c|a&b&~c，如果abc那个是1我们就用原来的字符表示，如果是0就取反，多个是1的地方用运算符|表示。

再比如有4个地方b是1，他们分别是

```
00      1      01        ~a & ~b & c

01      1      11        ~a & b & c

01      0      01        ~a & b & ~c

11      0      11         a & b & ~c
```

所以b=~a&~b&c|~a&b&c|~a&b&~c|a&b& ~c。

往期推荐

- 556，位运算解形成两个异或相等数组的三元组数目
- 499，位运算解只出现一次的数字 III
- 495，位运算等多种方式解找不同
- 494，位运算解只出现一次的数字

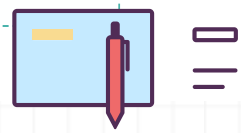
556，位运算解形成两个异或相等数组的三元组数目

原创 博哥 数据结构和算法 5月21日

收录于话题
#算法图文分析

161个 >

> Art is the stored honey of the human soul, gathered on wings of misery and travail.
>
> 艺术乃贮存人类灵魂的蜂蜜，由痛苦和辛劳的翅膀采集。



问题描述

给你一个整数数组arr。现需要从数组中取三个下标i、j和k，其中(0<=i<j<=k<arr.length)。

a和b定义如下：

- **a=arr[i]^arr[i+1]^...^arr[j-1]**
- **b=arr[j]^arr[j+1]^...^arr[k]**

注意：^表示按位异或操作。
请返回能够令a==b成立的三元组(i,j,k)的数目。

示例 1：

> **输入**：arr = [2,3,1,6,7]
> **输出**：4
> **解释**：满足题意的三元组分别是 (0,1,2), (0,2,2), (2,3,4) 以及 (2,4,4)

示例 2：

> **输入**：arr = [1,1,1,1,1]
> **输出**：10

示例 3：

> **输入**：arr = [2,3]
> **输出**：0

示例 4：

> **输入**：arr = [1,3,5,7,9]
> **输出**：3

示例 5：

> **输入**：arr = [7,11,12,9,5,2,7,17,22]
> **输出**：8

提示：

- 1 <= arr.length <= 300
- 1 <= arr[i] <= 10^8

位运算解决

做这道题之前我们来看一下异或的几个特性

- **a^0=a；任何数字和0异或还是他自己**
- **a^a=0；任何数字和自己异或都是0**
- **a^b^c=a^c^b；异或运算具有交换律**

我们看一下这题a的值是数组[i……j-1]中所有元素的异或结果，b的值是数组[j……k]中所有元素的异或结果，并且a中异或的元素和b中异或的元素是连续的并且没有重叠。如果要让a==b，那么a^b=0，也就是
arr[i]^arr[i+1]^……^arr[j]^……^arr[k]=0;

那这个问题就好办了，我们只需要从数组arr中找到一些**连续**的元素，他们的异或结果等于0即可。

那么一些连续的元素至少需要多少个呢，因为题中的条件是i<j，并且j可以等于k，这个k我们不需要管，所以至少需要2个元素。也就是说从数组arr中找到至少2个以上的连续的元素他们的异或结果是0即可成立三元组 (i, j , k)。

这里还要再来看一个问题，假如数组[1,2,5,6]的异或结果是0，那么可能的组合有哪些

- a=1，b=2^3^4--->[i,j,k]的值是[0,1,3]
- a=1^2，b=^3^4->[i,j,k]的值是[0,2,3]
- a=1^2^3，b=4--->[i,j,k]的值是[0,3,3]

也就是说如果数组中连续n个元素的异或结果是0，那么可能的组合就有n-1种。搞懂了上面的分析过程，代码就简单多了。

来看下代码

```
public int countTriplets(int[] arr) {
  //所有可能的组合
  int total = 0;
  int length = arr.length;
  //判断数组从i到j的元素异或结果是否是0
  for (int i = 0; i < length - 1; i++) {
    int xor = arr[i];
    for (int j = i + 1; j < length; j++) {
      xor ^= arr[j];
      //如果数组从i到j的异或结果是0，那么他们
      //可能的组合就是j-i
      if (xor == 0) {
        total += (j - i);
      }
    }
  }
  return total;
}
```


往期推荐

- 499，位运算解只出现一次的数字 III
- 495，位运算等多种方式解找不同
- 494，位运算解只出现一次的数字
- 451，回溯和位运算解子集

534，剑指 Offer-0～n-1中缺失的数字

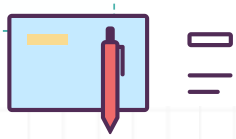
原创 博哥 数据结构和算法 1周前

收录于话题
#剑指offer

32个 >

Be yourself; everyone else is already taken.

做你自己，因为别人都有人做了。



问题描述

一个长度为n-1的递增排序数组中的所有数字都是唯一的，并且每个数字都在范围0～n-1之内。在范围0～n-1内的n个数字中有且只有一个数字不在该数组中，请找出这个数字。

示例 1:

> **输入**: [0,1,3]
> **输出**: 2

示例 2:

> **输入**: [0,1,2,3,4,5,6,7,9]
> **输出**: 8

限制:

1 <= 数组长度 <= 10000

位运算求解

题中的意思是数字[0,n]之间的n+1个数字少了一个，而其他的数字都存在。

如果我们把这个数组添加从0~n的n+1个元素，就变成了数组中只有一个数出现了一次，其他数字都出现了2次，让我们求这个只出现一次的数字。这题使用位运算是最容易解决的，关于位运算有下面几个规律

1^1=0;
1^0=1;
0^1=1;
0^0=0;

也就说0和1异或的时候相同的异或结果为0，不同的异或结果为1，根据上面的规律我们得到

a^a=0；自己和自己异或等于0
a^0=a；任何数字和0异或还等于他自己
a^b^c=a^c^b；异或运算具有交换律

有了这3个规律，这题就很容易解了，我们只需要把所有的数字都异或一遍，最终的结果就是我们要求的那个数字。来看下代码

```
1   public int missingNumber(int[] nums) {
2     int xor = 0;
3     for (int i = 0; i < nums.length; i++)
4       xor ^= nums[i] ^ (i + 1);
5     return xor;
6   }
```

或者还可以这样写，原理都是一样的

```
1   public int missingNumber(int[] nums) {
2     int xor = 0;
3     for (int i = 0; i < nums.length; i++)
4       xor ^= nums[i] ^ i;
5     return xor ^ nums.length;
6   }
```

求和

如果不缺那个数字的话，这个数组的所有数字可以组成一个等差数列，我们只需要根据公式求和，然后再减去数组中所有的数字即可，代码如下

```
1   public int missingNumber(int[] nums) {
2     int length = nums.length;
3     int sum = (0 + length) * (length + 1) / 2;
4     for (int i = 0; i < length; i++)
5       sum -= nums[i];
6     return sum;
7   }
```

暴力求解

题中说了是**递增排序数组**，所以我们只需要从前往后逐个遍历，少了哪个就返回哪个

```
public int missingNumber(int[] nums) {
  int length = nums.length;
  for (int i = 0; i < length; i++) {
    if (nums[i] != i)
      return i;
  }
  return length;
}
```

二分法查找

一般的二分法查找，找到之后会直接返回，这里使用二分法主要是在不断的缩小区间，直到找到为止。

```
public int missingNumber(int[] nums) {
  int start = 0;
  int end = nums.length - 1;
  while (start < end) {
    int mid = start + (end - start) / 2;
    if (nums[mid] == mid) {
      //如果nums[mid] == mid也就是说当前元素的
      //下标等于他自己，比如数组[0,1,2,3,4,5]每
      //个元素的下标都等于他自己，说明[start,mid]
      //没有缺少任何数字，那么缺少的肯定是在[mid+1,end]
      start = mid + 1;
    } else {
      //如果当前元素的下标不等于他自己，比如[0,1,2,4]中
      //nums[3]==4，说明说明缺少的数字就在这个区间内
      end = mid;
    }
  }
  //如果类似于[0,1,2,3]这种start指向了数组的最后一个，我们让他加1
  return start == nums[start] ? start + 1 : start;
}
```

当然还可以换种写法

```
public int missingNumber(int[] nums) {
  int start = 0;
  int end = nums.length - 1;
  while (start <= end) {
    int mid = start + (end - start) / 2;
    if (nums[mid] == mid) {
      //如果nums[mid] == mid也就是说当前元素的
      //下标等于他自己，比如数组[0,1,2,3,4,5]每
      //个元素的下标都等于他自己，说明[start,mid]
      //没有缺少任何数字，那么缺少的肯定是在[mid+1,end]
      start = mid + 1;
    } else {
      //注意这里写法和上面代码不一样
      end = mid - 1;
    }
  }
  return start;
}
```

总结

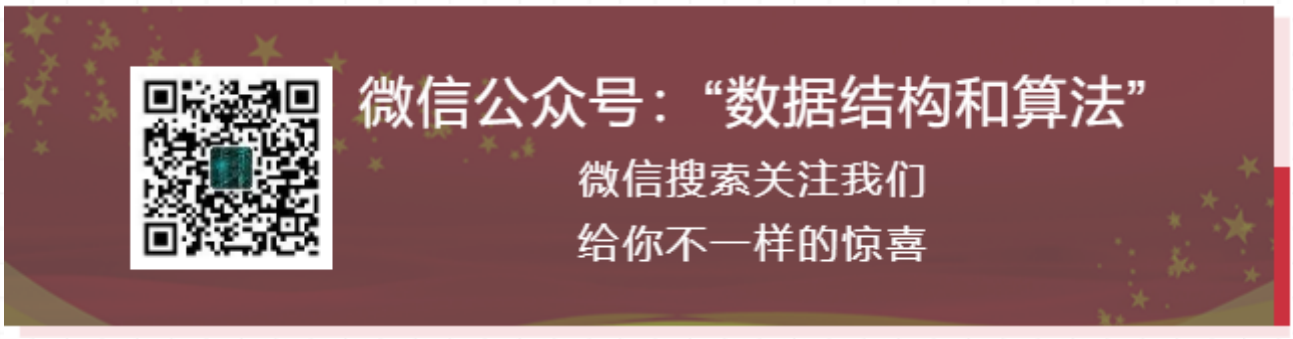
这题算是比较简单的一道题，基本上没什么难度。

513，汉明距离

原创 山大王wld 数据结构和算法 2月1日

收录于话题
#算法图文分析

137个 >



Just don't be disappointed if there's not a pot of gold at the end of the rainbow.

千万别失望，就算彩虹尽头没有你期待的美好也没关系。

问题描述

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。
给出两个整数 x 和 y，计算它们之间的汉明距离。

注意：
0 ≤ x, y < 2^31.

示例：

> 输入: x = 1, y = 4
>
> 输出: 2
>
> 解释:
> 1 (0 0 0 1)

4 (0 1 0 0)
↑ ↑

上面的箭头指出了对应二进制位不同的位置。

问题分析

x和y都转化为二进制的时候，在相同的位置上如果值都一样，他们的汉明距离就是0。如果在相同的位置上值不一样，有多少个不一样的位置，那么汉明距离就是多少。所以看到这道题，我们应该最容易想到的就是先异或运算，然后再计算这个异或运算的结果在二进制表示中1的个数。代码如下

```
public int hammingDistance(int x, int y) {
  return Integer.bitCount(x ^ y);
}
```

一行代码搞定，这题实际上没什么难度，我们只需要计算x和y的异或结果，然后再计算这个结果的二进制中1的个数即可。在之前我们分3个系列分别讲到了二进制中1的个数

364，位1的个数系列（一）

385，位1的个数系列（二）

402，位1的个数系列（三）

当然这题答案非常多，下面我们再来看两种写法

```
public int hammingDistance(int x, int y) {
  int xor = x ^ y;
  int res = 0;
  while (xor != 0) {
    res += xor & 1;
    xor = xor >>> 1;
  }
  return res;
}
```

或者

```
public int hammingDistance(int x, int y) {
  int xor = x ^ y;
  int res = 0;
  while (xor != 0) {
    res += 1;
    xor &= xor - 1;
  }
  return res;
}
```


往期推荐

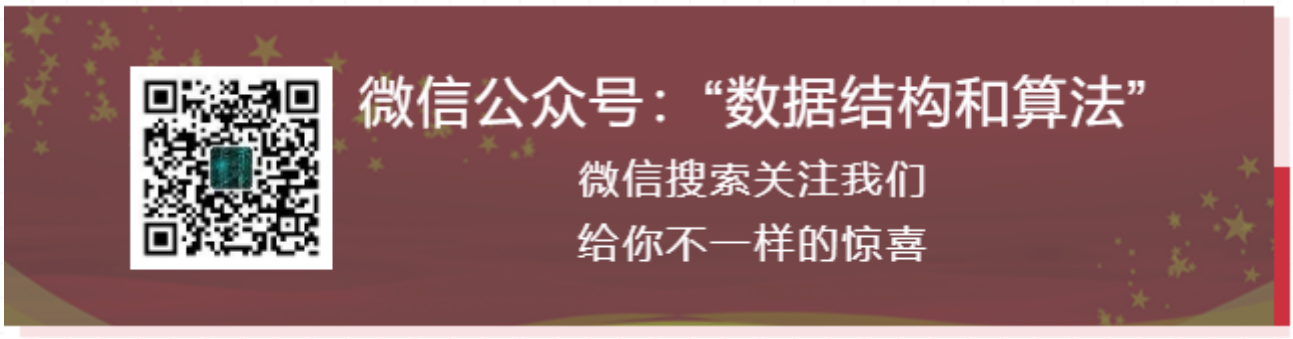
- 499，位运算解只出现一次的数字 III

512，反转二进制位

原创 山大王wld 数据结构和算法 1月25日

收录于话题
#算法图文分析

137个 >



Life is just like that sometimes, we're hoping for a unicorn and we get a goat.

有时候人生就是如此，我们期待的是独角兽，得到的却是山羊。

问题描述

在Java语言中有一个类叫Integer，他是int类型的包装类。这个类里面有很多关于二进制的操作，之前在讲

364，位1的个数系列（一）

385，位1的个数系列（二）

402，位1的个数系列（三）

提到过二进制中1的个数的计算方式

```
/**
 * Returns the number of one-bits in the two's complement binary
 * representation of the specified {@code int} value.  This function is
 * sometimes referred to as the <i>population count</i>.
 *
 * @param i the value whose bits are to be counted
```

```
     * @return the number of one-bits in the two's complement binary
     *     representation of the specified {@code int} value.
     * @since 1.5
     */
    public static int bitCount(int i) {
        // HD, Figure 5-2
        i = i - ((i >>> 1) & 0x55555555);
        i = (i & 0x33333333) + ((i >>> 2) & 0x33333333);
        i = (i + (i >>> 4)) & 0x0f0f0f0f;
        i = i + (i >>> 8);
        i = i + (i >>> 16);
        return i & 0x3f;
    }
```

今天讲的不是二进制中1的个数，而是对二进制进行反转。比如1000的二进制是

```
00000000 00000000 00000011 11101000
```

反转的结果就是

```
00010111 11000000 00000000 00000000
```

我们先来看一下代码，这是Integer类中的代码

```
/**
 * Returns the value obtained by reversing the order of the bits in the
 * two's complement binary representation of the specified {@code int}
 * value.
 *
 * @param i the value to be reversed
 * @return the value obtained by reversing order of the bits in the
 *     specified {@code int} value.
 * @since 1.5
 */
public static int reverse(int i) {
    // HD, Figure 7-1
    i = (i & 0x55555555) << 1 | (i >>> 1) & 0x55555555;
    i = (i & 0x33333333) << 2 | (i >>> 2) & 0x33333333;
```

```
    i = (i & 0x0f0f0f0f) << 4 | (i >>> 4) & 0x0f0f0f0f;
    i = (i << 24) | ((i & 0xff00) << 8) |
      ((i >>> 8) & 0xff00) | (i >>> 24);
    return i;
}
```

代码解析

上面代码咋一看有点懵，因为他们要么是十进制要么是16进制，16进制看的不是很明白，如果我们把它转化为二进制就比较容易理解了。

```
0x55555555的二进制是：
01010101 01010101 01010101 01010101

0x33333333的二进制是：
00110011 00110011 00110011 00110011

0x0f0f0f0f的二进制是：
00001111 00001111 00001111 00001111
```

我们看到在二进制中
0x55555555是每1对0和1交替出现
0x33333333是每2对0和1交替出现
0x0f0f0f0f 是每4对0和1交替出现。

1，先来看第一行代码
`i = (i & 0x55555555) << 1 | (i >>> 1) & 0x55555555;`
因为0x55555555在二进制中是0和1交替出现的，(i & 0x55555555)相当于把i的二进制中奇数位（从右边数）不变，偶数位全部变为0，然后再左移一位，相当于把奇数位的值全部变成了偶数位。
(i >>> 1) & 0x55555555是先往右无符号右移一位，然后在和0x55555555进行与运算，相当于把原来偶数位上的值移到了奇数位上。
最后再执行或（|）运算，完美的实现了奇偶位上数值的互换。

我们就用0x55555555来测试一下

```
int num = 0x55555555;
System.out.println("num的二进制表示：");
```

```
System.out.println(Util.bitInt32(num));
System.out.println("第一步计算之后num的二进制表示：");
num = (num & 0x55555555) << 1 | (num >>> 1) & 0x55555555;
System.out.println(Util.bitInt32(num));
```

来看一下打印结果

```
num的二进制表示：
01010101 01010101 01010101 01010101
第一步计算之后num的二进制表示：
10101010 10101010 10101010 10101010
```

同理，上面的0x33333333和0x0f0f0f0f分别完成了每4位和每8位之间的前半部分和后半部分的交换。后面的就很好理解了，因为在java语言中int是32位的，上面已经完成了8位之间的交换，后面我们只需要把int类型的二进制分为4份，然后再交换。

总结

上面的代码实现过程不是很难，但是对于初学者估计还是有一定的难度的。我们再来看下下面一段代码

```
public static int reverse(int var0) {
  var0 = (var0 & 1431655765) << 1 | var0 >>> 1 & 1431655765;
  var0 = (var0 & 858993459) << 2 | var0 >>> 2 & 858993459;
  var0 = (var0 & 252645135) << 4 | var0 >>> 4 & 252645135;
  var0 = var0 << 24 | (var0 & '\uff00') << 8 | var0 >>> 8 & '\uff00' | var0
  return var0;
}
```

其实上面代码是一样的，只不过一个都是十进制，一个是十进制和16进制的混合。

往期推荐

- 499，位运算解只出现一次的数字 III
- 495，位运算等多种方式解找不同
- 494，位运算解只出现一次的数字
- 469，位运算求最小的2的n次方

499，位运算解只出现一次的数字 III

原创 山大王wld 数据结构和算法 4天前

收录于话题
#算法图文分析

111个 >



It’s not the load that breaks you down, it’s the way you carry it.

压垮你的不是那些重担，而是你背负的方式。

问题描述

给定一个整数数组 `nums` ，其中恰好有两个元素只出现一次，其余所有元素均出现两次。找出只出现一次的那两个元素。

示例 ：

> **输入**: [1,2,1,3,2,5]
> **输出**: [3,5]

位运算解决

前面刚讲过一个和这题类似的题494，位运算解只出现一次的数字，只不过第494题只有一个数字出现一次，但这题是有两个数字只出现一次。我们知道在位运算中**异或运算**具有交换律，也就是

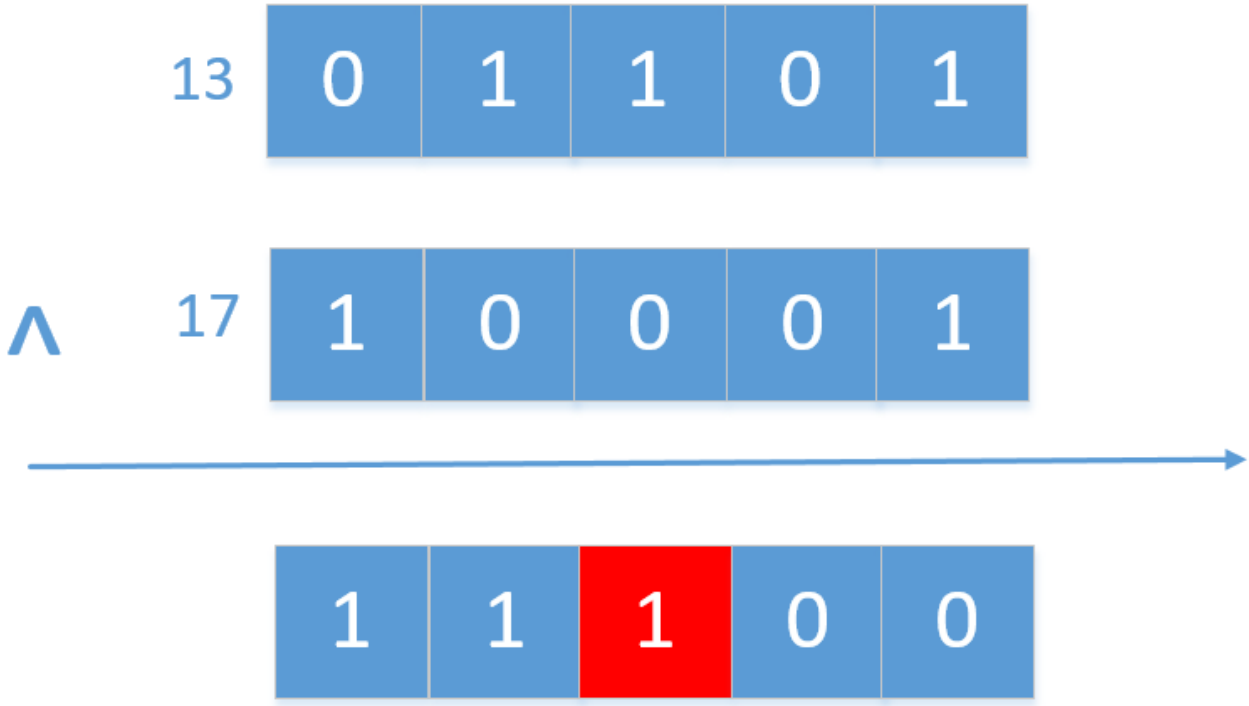
A^B^C=A^C^B

我们还知道一个数字和自己异或，结果是0，也就是
A^A=0；
任何数字和0异或结果还是他自己
A^0=A；

有了上面的3个公式，这题就很容易解了，假如数组的元素是
[a，e，f，h，b，f，h，e]
我们看到这个数组中只有a和b出现了一次，其他的元素都出现了2次。如果我们把数组中的所有元素全部都异或一遍,也就是下面这样
a^e^f^h^b^f^h^e
因为异或具有交换律，我们可以把它整理成
a^b^(f^f)^(h^h)^(e^e)
结果就是a^b^0^0^0=a^b

因为a和b是不相等的，所以他俩的异或结果不可能是0，只要不为0，那么这个结果转化为二进制的时候肯定有1。关于异或运算有下面几个规律
1^1=0;
1^0=1;
0^1=1;
0^0=0;
我们看到结果为1的要么是0和1异或，要么是1和0异或。也就是说我们可以根据a和b某一位是否是0和1来把数组分为两组，并且a和b都不在同一组

举个例子，比如数组
[12，13，14，17，14，12]
那么他们异或的结果就是13^17



我们看到异或结果最右边的1，也就是红色部分，根据这个位置是0还是1把原数组分为两组，那么13和17肯定不在同一组。那么每组就变成了只有一个数字出现一次，其他数字都出现两次。然后我们就可以使用494，位运算解只出现一次的数字的方式来解了。代码如下

```
public int[] singleNumber(int[] nums) {
    int bitmask = 0;
    //把数组中的所有元素全部都异或一遍
    for (int num : nums) {
        bitmask ^= num;
    }
    //因为异或运算的结果不一定都是2的n次幂，
    //在二进制中可能会有多个1，为了方便计算
    //我们只需保留其中的任何一个1，其他的都
    //让他变为0，这里保留的是最右边的1
    bitmask &= -bitmask;
    int[] rets = {0, 0};
    for (int num : nums) {
        //然后再把数组分为两部分，每部分在
        //分别异或
        if ((num & bitmask) == 0) {
            rets[0] ^= num;
        } else {
            rets[1] ^= num;
        }
    }
    return rets;
}
```

上面的位运算bitmask &= -bitmask;表示的是把bitmask二进制中最右边的1保留，其他位置全部变为0，随便找个数据打印一下

```
public static void main(String args[]) {
    int a = 76;
    System.out.println(a + "的二进制是:              " + Util.bitInt32(a));
    System.out.println(-a + "的二进制是:             " + Util.bitInt32(-a));
    System.out.println(a + " & " + -a + "运算结果的二进制是:" + Util.bitInt32( num: a & -a));
}
```

再来看一下运算结果

```
76的二进制是:                  00000000 00000000 00000000 01001100
-76的二进制是:                 11111111 11111111 11111111 10110100
76 & -76运算结果的二进制是:00000000 00000000 00000000 00000100
```

总结

这题不是很难，但做这题需要对异或运算熟练掌握才行。

往期推荐

- 469，位运算求最小的2的n次方
- 451，回溯和位运算解子集
- 425，剑指 Offer-二进制中1的个数
- 417，BFS和DFS两种方式求岛屿的最大面积

495，位运算等多种方式解找不同

原创 山大王wld 数据结构和算法 1周前

收录于话题
#算法图文分析

111个 >



If you cannot be a poet, be the poem.

如果你不能成为一个诗人，就活成一首诗。

问题描述

给定两个字符串s和t，它们只包含小写字母。

字符串t由字符串s随机重排，然后在随机位置添加一个字母。

请找出在t中被添加的字母。

示例 1：

> **输入**：s = "abcd", t = "abcde"
> **输出**："e"
> **解释**：'e' 是那个被添加的字母。

示例 2：

> **输入**：s = "", t = "y"
> **输出**："y"

示例 3：

> **输入**：s = "a", t = "aa"
> **输出**："a"

示例 4：

> **输入**：s = "ae", t = "aea"
> **输出**："a"

提示：

- 0<=s.length<=1000
- t.length==s.length+1
- s和t只包含小写字母

位运算解决

这题说的是字符串t只比s多了一个字符，其他字符他们的数量都是一样的。如果我们把字符串s和t合并就会发现，除了那个多出的字符出现奇数次，其他的所有字符都是出现偶数次，看到这里我们很容易想到494，位运算解只出现一次的数字。所以只要是第494题的解我们都可以拿来用。

所以这题最简单的一种解决方式就是使用异或运算，关于异或运算有下面几个规律

- a^a=0；任何数字和自己**异或**都是0
- a^0=a；任何数字和0**异或**还是他自己
- a^b^c=a^c^b **异或**运算具有交换律

因为s和t合并之后，偶数个的字符通过异或都会变为0，奇数个的字符异或之后还是他自己，我们只需要把合并的字符全部异或一遍即可，代码如下

```
1  public char findTheDifference(String s, String t) {
2    char[] charArr = s.concat(t).toCharArray();
3    char res = 0;
4    for (char c : charArr) {
5      res ^= c;
6    }
7    return res;
8  }
```

纯数学的方式解决

既然字符串s比t少一个字符，我们先统计字符串s中每个字符的数量，然后减去字符串t中的每个字符，如果小于0，说明字符串s比t少的就是这个字符，直接返回即可，代码如下

```
public char findTheDifference(String s, String t) {
  int count[] = new int[26];
  for (int i = 0; i < s.length(); i++) {
    count[s.charAt(i) - 'a']++;
  }
  for (int i = 0; i < t.length(); i++) {
    if (--count[t.charAt(i) - 'a'] < 0)
      return t.charAt(i);
  }
  return 'a';
}
```

使用结合Set解决

把字符串s和t合并，然后遍历合并的每个字符，判断集合set中是否有这个字符，如果有就移除，否则就加入到集合set中。最后集合set中只有一个字符，这个字符就是我们所求的。

```
public char findTheDifference(String s, String t) {
  Set<Character> set = new HashSet<>();
  char[] charArr = s.concat(t).toCharArray();
  for (int i = 0; i < charArr.length; i++) {
    if (set.contains(charArr[i]))
      set.remove(charArr[i]);
    else
      set.add(charArr[i]);
  }
  return (char) set.toArray()[0];
}
```

其实还可以把contains和add方法合并，如果add失败，说明集合Set中有这个数字，然后再把它给移除即可。

```
public char findTheDifference(String s, String t) {
  Set<Character> set = new HashSet<>();
  char[] charArr = s.concat(t).toCharArray();
  for (int i = 0; i < charArr.length; i++) {
    if (!set.add(charArr[i]))
      set.remove(charArr[i]);
  }
  return (char) set.toArray()[0];
}
```

计算两个字符串的差值

还可以用t中所有字符的和减去s中所有字符的和，最后结果就是要求的那个字符

```
public char findTheDifference(String s, String t) {
  int distance = 0;
  for (int i = 0; i < s.length(); ++i) {
    distance -= s.charAt(i);
    distance += t.charAt(i);
  }
  distance += t.charAt(t.length() - 1);
  return (char) distance;
}
```

总结

这题不算难，但解法比较多，位运算应该是最简单的解决方式。

往期推荐

- 493，动态规划解打家劫舍 III
- 492，动态规划和贪心算法解买卖股票的最佳时机 II
- 470，DFS和BFS解合并二叉树
- 464. BFS和DFS解二叉树的所有路径

494，位运算解只出现一次的数字

原创 山大王wld 数据结构和算法 1周前

收录于话题
#算法图文分析

111个 >



By letting go of something, your arms are free to grab hold of something else.

学会放手，才能获得更多。

问题描述

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：
你的算法应该具有线性时间复杂度。你可以不使用额外空间来实现吗？

示例 1：

> **输入**: [2,2,1]
> **输出**: 1

示例 2：

> **输入**：[4,1,2,1,2]
> **输出**：4

位运算解决

这题说的是只有一个数出现了一次，其他数字都出现了2次，让我们求这个只出现一次的数字。这题使用位运算是最容易解决的，关于位运算有下面几个规律

1^1=0;

1^0=1;

0^1=1;

0^0=0;

也就说0和1异或的时候相同的**异或结果**为0，不同的**异或结果**为1，根据上面的规律我们得到

a^a=0；自己和自己**异或**等于0

a^0=a；任何数字和0**异或**还等于他自己

a^b^c=a^c^b；**异或**运算具有交换律

有了这3个规律，这题就很容易解了，我们只需要把所有的数字都**异或**一遍，最终的结果就是我们要求的那个数字。来看下代码

```
1   public int singleNumber(int nums[]) {
2     int result = 0;
3     for (int i = 0; i < nums.length; i++)
4       result ^= nums[i];
5     return result;
6   }
```

使用集合Set解决

这个应该是最容易想到的，我们遍历数组中的元素，然后在一个个添加到集合Set中，如果添加失败，说明以前添加过，就把他给移除掉。当我们把数组中的所有元素都遍历完的时候，集合Set中只会有一个元素，这个就是我们要求的值。

```
1   public int singleNumber(int[] nums) {
2     Set<Integer> set = new HashSet<>();
3     for (int num : nums) {
4       if (!set.add(num)) {
5         //如果添加失败，说明这个值
6         //在集合Set中存在，我们要
7         //把他给移除掉
8         set.remove(num);
9       }
10    }
11    //最终集合Set中只有一个元素，我们直接返回
12    return (int) set.toArray()[0];
13  }
```

总结

还有一种解题思路就是使用HashMap来统计，但无论哪种方式都没有位运算来的快。

往期推荐

- 469，位运算求最小的2的n次方
- 451，回溯和位运算解子集
- 425，剑指 Offer-二进制中1的个数
- 383，不使用“+”,“-”,“×”,“÷”实现四则运算

476，根据数字二进制下1的数目排序

原创 山大王wld 数据结构和算法 11月13日

收录于话题
#算法图文分析

95个 >



Don't just follow the path. Make your own trail.

不要随波逐流，要找到自己的路。

问题描述

给你一个整数数组arr。请你将数组中的元素按照其二进制表示中数字1的数目升序排序。

如果存在多个数字二进制中1的数目相同，则必须将它们按照数值大小升序排列。

请你返回排序后的数组。

示例 1：

> **输入**：arr = [0,1,2,3,4,5,6,7,8]
> **输出**：[0,1,2,4,8,3,5,6,7]
> **解释**：[0] 是唯一一个有 0 个 1 的数。
> [1,2,4,8] 都有 1 个 1 。
> [3,5,6] 有 2 个 1 。
> [7] 有 3 个 1 。

> 按照 1 的个数排序得到的结果数组为 [0,1,2,4,8,3,5,6,7]

示例 2：

> **输入**：arr = [1024,512,256,128,64,32,16,8,4,2,1]
> **输出**：[1,2,4,8,16,32,64,128,256,512,1024]
>
> **解释**：数组中所有整数二进制下都只有 1 个 1 ，所以你需要按照数值大小将它们排序。

示例 3：

> **输入**：arr = [10000,10000]
> **输出**：[10000,10000]

示例 4：

> **输入**：arr = [2,3,5,7,11,13,17,19]
> **输出**：[2,3,5,17,7,11,13,19]

示例 5：

> **输入**：arr = [10,100,1000,10000]
> **输出**：[10,100,10000,1000]

提示：

- 1 <= arr.length <= 500
- 0 <= arr[i] <= 10^4

位运算和数字合并解决

这道题是让求数组中每个数字二进制中1的个数，然后以1的个数从大到小排序，如果1的个数相同，就按照原来数字的大小进行排序。

我们仔细看这题有个限制条件就是0<=arr[i]<=10^4，也就是说数组元素都是非负数，并且都是小于等于10000的。不知道大家有没有学Android的，看过Android源

码，你会看到里面经常会用一个数字表示多种状态，有的是用二进制中某一位来表示，有的是用二进制中的多位来表示，看到这里是不是也有了灵感。

所以一种最简单的方式，先计算出数组中每个元素二进制中1的个数乘以100000+原来的数，让他成为一个新的数，接着再对他们进行排序，最后再还原，看下代码

```
public int[] sortByBits(int[] arr) {
    int length = arr.length;
    //数组中每个数字的二进制位乘以100000再加上原来的数值，
    //成为一个新的数
    for (int i = 0; i < length; i++) {
        arr[i] = Integer.bitCount(arr[i]) * 100000 + arr[i];
    }
    //对这个新的数进行排序
    Arrays.sort(arr);
    //然后再把新的数字还原成原来的数字
    for (int i = 0; i < length; i++) {
        arr[i] %= 100000;
    }
    return arr;
}
```

先计算再排序

还有一种方式就是先计算数组中每个数字二进制中1的个数，然后再排序，这种就比较简单了。关于一个数字二进制中1的个数可以看下425，剑指 Offer-二进制中1的个数，这里列出了18种方式，具体细节可以看下

364，位1的个数系列（一）

385，位1的个数系列（二）

402，位1的个数系列（三）

关于排序，在两年前写过十几种排序算法，也可以看下

101，排序-冒泡排序

102，排序-选择排序

103，排序-插入排序

104，排序-快速排序

105，排序-归并排序

106，排序-堆排序

107，排序-桶排序

108，排序-基数排序

109，排序-希尔排序

110，排序-计数排序

111，排序-位图排序

112，排序-其他排序

如果这样两者结合的话，那么答案就比较多了，我们来随便挑一组组合。排序就使用插入排序，计算二进制中1的个数我们随便挑一个，答案如下

```
public int[] sortByBits(int[] arr) {
    //二维数组，temp[i][0]存储的是当前的值
    //temp[i][1]存储的是当前值的二进制中1的个数
    int[][] temp = new int[arr.length][2];
    for (int i = 0; i < arr.length; i++) {
        temp[i][0] = arr[i];
        temp[i][1] = hammingWeight(arr[i]);
    }
    //使用插入排序
    insertSort(temp);
    for (int i = 0; i < arr.length; i++) {
        arr[i] = temp[i][0];
    }
    return arr;
}

//插入排序
private void insertSort(int[][] array) {
    for (int i = 1; i < array.length; i++) {
        int j;
        int[] temp = array[i];
        for (j = i - 1; j >= 0; j--) {
            //先比较位1的大小，如果相同再比较数字的大小
            if (array[j][1] > temp[1] || (array[j][1] == temp[1] && array[j][0] > temp[0])) {
                array[j + 1] = array[j];//往后挪
            } else {
                break;//没有交换就break
            }
        }
        array[j + 1] = temp;
    }
}

//计算二进制中1的个数
public int hammingWeight(int n) {
    int count = 0;
    while (n != 0) {
        n &= n - 1;
        count++;
    }
    return count;
}
```

问题分析

这题结合了计算二进制中1的个数和排序两种方式，这两种方式之前都有大量的介绍，如果结合写的话，答案还是比较多的。

往期推荐

- 425，剑指 Offer-二进制中1的个数
- 469，位运算求最小的2的n次方
- 451，回溯和位运算解子集

469，位运算求最小的2的n次方

原创 山大王wld 数据结构和算法 10月29日

收录于话题
#算法图文分析

95个 >



Promise yourself to be so strong that nothing can disturb your peace of mind.

答应自己要坚强，不让任何事物烦扰内心的平静。

问题描述

给一个函数f(x)，返回一个不小于x的最小的2的n次方。描述的比较绕口，举个例子。

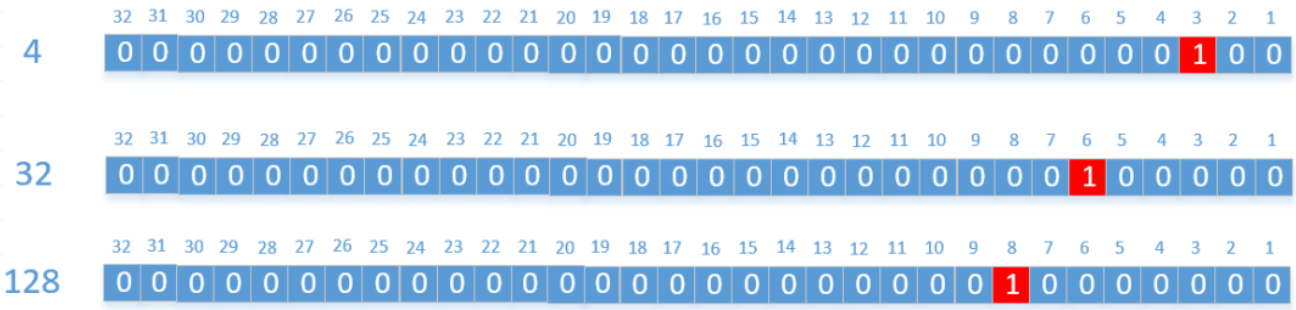
```
f(7)=8
f(9)=16
f(30)=32
f(64)=64
```

注意：

- x是正整数
- 0<x<Integer.MAX_VALUE

循环解决

我们看到返回的结果都是2的n次方，并且是不小于x的最小的2的n次方。如果把2的n次方转换成二进制我们就会发现，只有一个位上是1，其他位上全部是0，随便举几个数字看一下



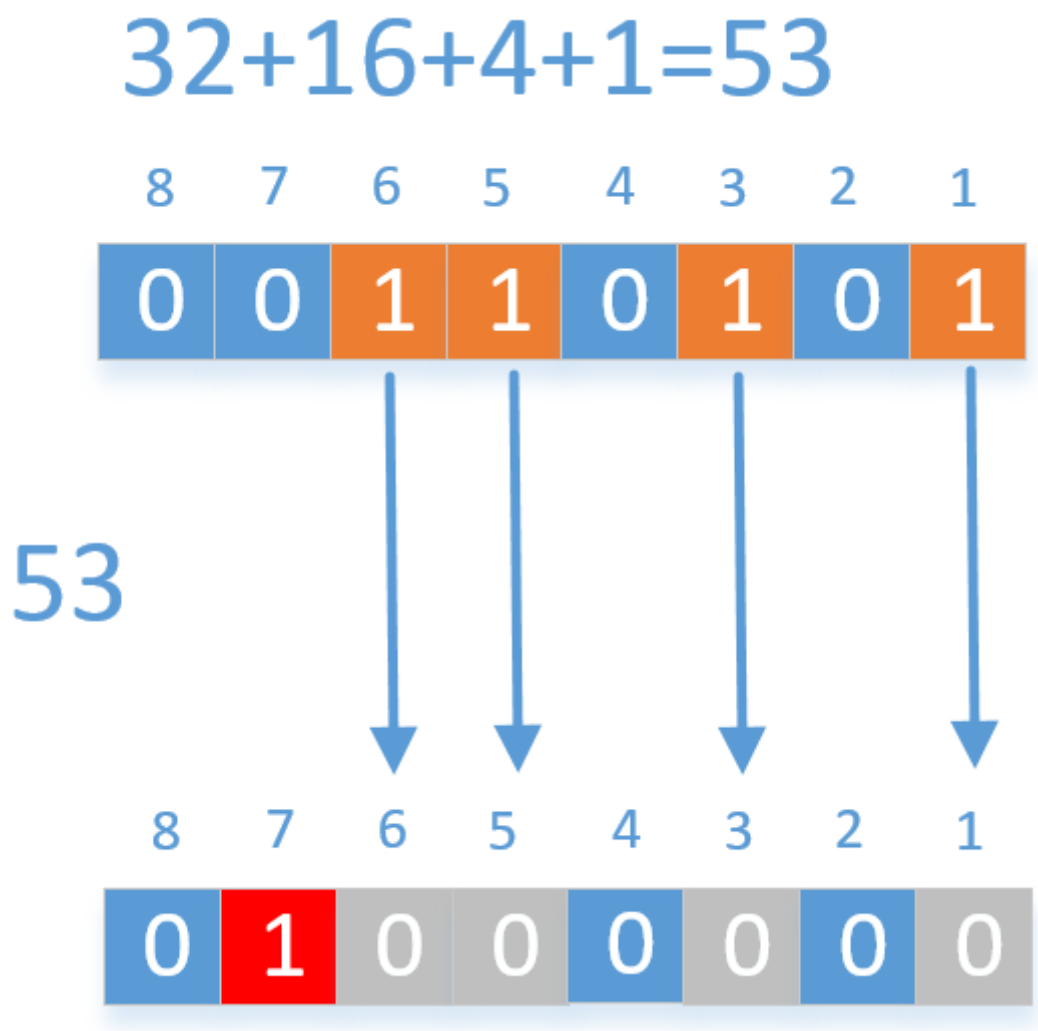
所以最简单的一种方式就是通过循环来计算，先用x和1比较，如果大于1，那么1就往左移一位，继续比较……，一直这样下去，直到不小于为止，原理比较简单，来看下代码

```
1   public int lowPower(int x) {
2     int n = 1;
3     while (x > n)
4       n <<= 1;
5     return n;
6   }
```

代码非常简洁，基本没什么难度

位运算解决

再来看下位运算，我们随便举个例子，比如53，他的二进制位如下，如果要找到不小于53的最小的2的n次方，我们只需要把53的二进制位中最左边的1往左移一位，其他的全部变为0即可。



所以一种最简单的方式就是通过移位运算，把53最左边的1全部往右边铺开，就变成了00111111，然后再加1就变成了了01000000。最后来看下代码

```
public int lowPower(int x) {
  //这里把最左边的1全部往右边铺开
  x |= x >>> 1;
  x |= x >>> 2;
  x |= x >>> 4;
  x |= x >>> 8;
  x |= x >>> 16;
  //最后再加1返回
  return x + 1;
}
```

但是这里有个小问题就是，如果x本来就是2的n次方，比如x是16，运算的结果就会变成32，与我们实际要求不符。所以这里我们可以先让x-1，然后再进行运算，所以正确答案应该是下面这样

```
public int lowPower(int x) {
  //这里把最左边的1全部往右边铺开
  //x--;
  x -= 1;
  x |= x >>> 1;
  x |= x >>> 2;
  x |= x >>> 4;
  x |= x >>> 8;
  x |= x >>> 16;
  //最后再加1返回
  return x + 1;
}
```

或者也可以改成这样，当然没有上面的代码简洁

```java
public int lowPower(int x) {
  if (x == 1)
     return 1;
  //这里把最左边的1全部往右边铺开
  x -= 1;
  x |= x >>> 1;
  x |= x >>> 2;
  x |= x >>> 4;
  x |= x >>> 8;
  x |= x >>> 16;
  //执行完下面这行代码，x相当于把
  //后面的1全部减掉了，只留下最前
  //面的那个1，也是2的n次方，
  //只不过这个是小于原来x的最大的
  //2的n次方
  x -= x >> 1;
  //然后我们再把它往左移一位，就变
  //成了不小于原来x的最小的2的n次方
  return x << 1;
}
```

总结

如果大家经常看源码可能对这个算法比较了解，这是HashMap中专门计算数组长度的，我们知道HashMap是数组加链表的结构，数组的大小就是2的n次方，无论你传入的大小是多少，他都会通过上面的计算。

往期推荐

- 451，回溯和位运算解子集
- 450，什么叫回溯算法，一看就会，一写就废
- 448，组合的几种解决方式
- 445，BFS和DFS两种方式解岛屿数量

425，剑指 Offer-二进制中1的个数

原创 山大王wld 数据结构和算法 8月7日

收录于话题
#剑指offer

27个 >



Success at anything will always come down to this: Focus & Effort, and we control both.

任何事物的成功都取决于两方面：专注和努力，而两者都由我们控制。

问题描述

请实现一个函数，输入一个整数，输出该数二进制表示中 1 的个数。

例如，把 9 表示成二进制是 1001，有 2 位是 1。因此，如果输入 9，则该函数输出 2。

示例 1：

```
输入：00000000000000000000000000001011
输出：3
解释：输入的二进制串 00000000000000000000000000001011 中，共有三位为 '1'。
```

示例 2：

> 输入：00000000000000000000000010000000
> 输出：1
> 解释：输入的二进制串 00000000000000000000000010000000 中，共有一位为 '1'。

示例 3：

> 输入：11111111111111111111111111111101
> 输出：31
> 解释：输入的二进制串 11111111111111111111111111111101 中，共有 31 位为 '1'。

二进制中1的个数

这道题是剑指offer上的一道题，其实比较简单，但要写10种以上的解法估计不容易，之前专门分3个系列讲过二进制中1的个数

364，位1的个数系列（一）

385，位1的个数系列（二）

402，位1的个数系列（三）

前面已经写过了，这里就不在细写了，我把答案全部列出来，因为太多，我只给一些简单的提示，如果不懂的可以看下前面写的那3个系列，或者也可以在下面留言，我来给你解答。

1，把n往右移32次，每次都和1进行与运算

```
public int hammingWeight(int n) {
  int count = 0;
  for (int i = 0; i < 32; i++) {
    if (((n >>> i) & 1) == 1) {
      count++;
    }
  }
  return count;
}
```

2，原理和上面一样，做了一点优化

```
public int hammingWeight(int n) {
  int count = 0;
  while (n != 0) {
    count += n & 1;
    n = n >>> 1;
  }
  return count;
}
```

3，1每次往左移一位，再和n进行与运算

```
public int hammingWeight(int n) {
```

```
    int count = 0;
    for (int i = 0; i < 32; i++) {
      if ((n & (1 << i)) != 0) {
        count++;
      }
    }
    return count;
  }
```

4，1每次往左移一位，把运算的结果在右移判断是否是1

```
  public int hammingWeight(int i) {
    int count = 0;
    for (int j = 0; j < 32; j++) {
      if ((i & (1 << j)) >>> j == 1)
        count++;
    }
    return count;
  }
```

5，这个是最常见的，每次消去最右边的1，直到消完为止

```
  public int hammingWeight(int n) {
    int count = 0;
    while (n != 0) {
      n &= n - 1;
      count++;
    }
    return count;
  }
```

6，把上面的改为递归

```
  public int hammingWeight(int n) {
    return n == 0 ? 0 : 1 + hammingWeight(n & (n - 1));
  }
```

7，查表

```
  public int hammingWeight(int i) {
    //table是0到15转化为二进制时1的个数
    int table[] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    int count = 0;
    while (i != 0) {//通过每4位计算一次，求出包含1的个数
      count += table[i & 0xf];
      i >>>= 4;
    }
    return count;
  }
```

8，每两位存储，使用加法（先运算再移位）

```
  public int hammingWeight(int n) {
    n = ((n & 0xaaaaaaaa) >>> 1) + (n & 0x55555555);
    n = ((n & 0xcccccccc) >>> 2) + (n & 0x33333333);
    n = (((n & 0xf0f0f0f0) >>> 4) + (n & 0x0f0f0f0f));
    n = n + (n >>> 8);
    n = n +  (n >>> 16);
    return n & 63;
  }
```

9，每两位存储，使用加法（先移位再运算）

```
public int hammingWeight(int n) {
    n = ((n >>> 1) & 0x55555555) + (n & 0x55555555);
    n = ((n >>> 2) & 0x33333333) + (n & 0x33333333);
    n = (((n >>> 4) & 0x0f0f0f0f) + (n & 0x0f0f0f0f));
    n = n + (n >>> 8);
    n = n + (n >>> 16);
    return n & 63;
}
```

10，和第8种思路差不多，只不过在最后几行计算的时候过滤的比较干净

```
public int hammingWeight(int n) {
    n = ((n & 0xaaaaaaaa) >>> 1) + (n & 0x55555555);
    n = ((n & 0xcccccccc) >>> 2) + (n & 0x33333333);
    n = (((n & 0xf0f0f0f0) >>> 4) + (n & 0x0f0f0f0f));
    n = (((n & 0xff00ff00) >>> 8) + (n & 0x00ff00ff));
    n = (((n & 0xffff0000) >>> 16) + (n & 0x0000ffff));
    return n;
}
```

11，每4位存储，使用加法

```
public int hammingWeight(int n) {
    n = (n & 0x11111111) + ((n >>> 1) & 0x11111111) + ((n >>> 2) & 0x11111111) + ((n >>> 3) & 0x11111111);
    n = (((n & 0xf0f0f0f0) >>> 4) + (n & 0x0f0f0f0f));
    n = n + (n >>> 8);
    n = n + (n >>> 16);
    return n & 63;
}
```

12，每3位存储，使用加法

```
public int hammingWeight(int n) {
    n = (n & 011111111111) + ((n >>> 1) & 011111111111) + ((n >>> 2) & 011111111111);
    n = ((n + (n >>> 3)) & 030707070707);
    n = ((n + (n >>> 6)) & 07700770077);
    n = ((n + (n >>> 12)) & 037700007777);
    return ((n + (n >>> 24))) & 63;
}
```

13，每5位存储，使用加法

```
public int hammingWeight(int n) {
    n = (n & 0x42108421) + ((n >>> 1) & 0x42108421) + ((n >>> 2) & 0x42108421) + ((n >>> 3) & 0x42108421) +
    n = ((n + (n >>> 5)) & 0xc1f07c1f);
    n = ((n + (n >>> 10) + (n >>> 20) + (n >>> 30)) & 63);
    return n;
}
```

14，每两位存储，使用减法（先运算再移位）

```
public int hammingWeight(int i) {
    i = i - ((i >>> 1) & 0x55555555);
    i = (i & 0x33333333) + ((i >>> 2) & 0x33333333);
    i = (i + (i >>> 4)) & 0x0f0f0f0f;
    i = i + (i >>> 8);
    i = i + (i >>> 16);
```

```
    return i & 0x3f;
}
```

15，每3位存储，使用减法

```
public int hammingWeight(int n) {
    n = n - ((n >>> 1) & 033333333333) - ((n >>> 2) & 011111111111);
    n = ((n + (n >>> 3)) & 030707070707);
    n = ((n + (n >>> 6)) & 07700770077);
    n = ((n + (n >>> 12)) & 037700007777);
    return ((n + (n >>> 24))) & 63;
}
```

16，每4位存储，使用减法

```
public int hammingWeight(int n) {
    int tmp = n - ((n >>> 1) & 0x77777777) - ((n >>> 2) & 0x33333333) - ((n >>> 3) & 0x11111111);
    tmp = ((tmp + (tmp >>> 4)) & 0x0f0f0f0f);
    tmp = ((tmp + (tmp >>> 8)) & 0x00ff00ff);
    return ((tmp + (tmp >>> 16)) & 0x0000ffff) % 63;
}
```

17，每5位存储，使用减法

```
public int hammingWeight(int n) {
    n = n - ((n >>> 1) & 0xdef7bdef) - ((n >>> 2) & 0xce739ce7) - ((n >>> 3) & 0xc6318c63) - ((n >>> 4) & 0x
    n = ((n + (n >>> 5)) & 0xc1f07c1f);
    n = ((n + (n >>> 10) + (n >>> 20) + (n >>> 30)) & 63);
    return n;
}
```

18，每次消去最右边的1，可以参照第5种解法

```
public static int hammingWeight(int num) {
    int total = 0;
    while (num != 0) {
        num -= num & (-num);
        total++;
    }
    return total;
}
```

总结

这题如果一直写下去，再写10种也没问题，如果上面的代码你都能看懂，你也会有和我一样的想法。但解这题的最终思路还是没变，所以再写下去也没有太大价值。上面有些写法其实也很鸡肋，这里只是告诉大家这样写也是可以实现的，虽然可能你永远都不这样去写。

往期推荐

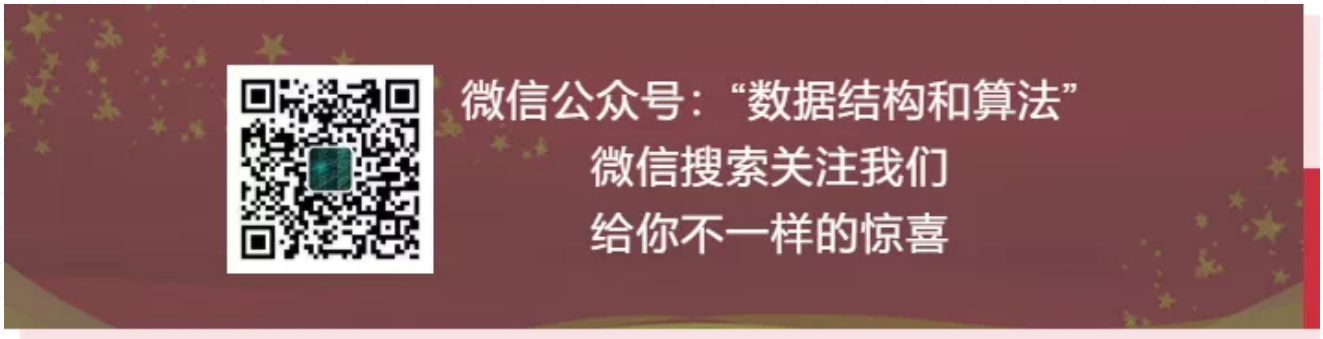
- 364，位1的个数系列（一）
- 385，位1的个数系列（二）
- 402，位1的个数系列（三）
- 361，交替位二进制数

383，不使用“+”,“-”,“×”,“÷”实现四则运算

原创 山大王wld 数据结构和算法 6月13日

收录于话题
#算法图文分析

96个 >



Youth, even in its sorrows, always has a brilliancy of its own.

青春，即使在它的悲哀时也是辉煌的。

基础知识

从我们开始上学的时候就知道，如果要实现加法运算就要使用“+”符号，如果要实现减法运算就要使用“-”符号……，甚至在今天的计算机中也是一样的，我们只知道怎么使用，但很少去关注他的底层是怎么实现的。如果突然哪天给你一道面试题，让你不使用"+"来实现两个数相加，你该怎么做呢，今天我们就来看一下该怎么实现。

一：不使用“+”实现两个数相加

我们先来看一道非常简单的题，在计算机中数字是由二进制位表示的，也就是说是由0和1组成的，如果我们要实现0和1之间的加法该怎么实现呢，他会有4种组合方式

1，0+0=00

2，0+1=01

3，1+0=01

4，1+1=10

我们发现一个很重要的规律，就是只有1+1有进位，其他的都没进位。所以我们判断有没有进位只需要判断a&b是否等于1即可，而a+b的值（不考虑进位）只需要计算a|b即可，看明白了这点，代码就呼之欲出了

```
1   private static int add1(int a, int b) {
2     int c = (a & b) << 1;//进位的值
3     int d = a ^ b;//不考虑进位，相加的值
4     return c | d;//或者 return c ^ d;
5   }
```

a和b要么是1要么是0，所以这里最多也只有一个进位，很好理解。但我们计算二进制的加减法不光只有1个0或1，可能会有好多个1或0，那我们该怎么实现呢。比如a=13（1101），b=9（1001），我们该怎么计算a+b的结果。首先如果我们**不考虑进位**问题，那么a+b的运算会是下面这样

$$\begin{array}{r} 1101 \\ +\ 1001 \\ \hline 0100 \end{array}$$

但实际上最前面和最后面的1都有了进位。

1，我们看到如果不考虑进位，那么a+b的结果其实就是a^b的结果，我们该怎么把进位问题也考虑在内呢，实际上只有1+1的时候才会出现进位，1+0或者0+0都不会出现进位，所以我们首先想到的是&运算

2，这里我们计算一下a&b的结果是1001，我们知道当&运算的结果为1的时候，说明参与&运算的两个都是1，既然两个都是1，那么相加的时候就肯定会有进位，所以他们进位的值实际上是10010（（a&b）<<1），然后在和0100相加就是我们要求的结果，10010+00100=10110，10110实际上就是22，也就是13+9的结果

3，但我们好像忽略了一个问题，就是这道题要求不能使用加减乘除符号，而上面我们分析的时候使用了加号，所以明显不行。通过上面的分析实际上我们已经发现了一个规律，就是a+b通过^和&运算之后又再执行相加操作，所以我们首先想到的是递归，我们来看下代码

```
1   private static int add2(int a, int b) {
2     if (a == 0 || b == 0)
3       return a ^ b;
4     return add2(a ^ b, (a & b) << 1);
5   }
```

第3行表示的是如果a==0就返回b，如果b==0就返回a，这种写法少了一个if语句的判断会更简洁。为了验证代码的准确性我们随便找几个数据测试一下

```
1     int[] array = {1, 1, 1, 0, 0, 1, 0, 0, 13, 9, 1, -1, -Integer.MAX_VALUE, Integer.MAX_VALUE, -8, -9};
```

```
    for (int i = 0; i < array.length / 2; i++) {
      System.out.println(array[i << 1] + "+" + array[(i << 1) + 1] + "=" + add2(array[i << 1], array[(i
    }
```

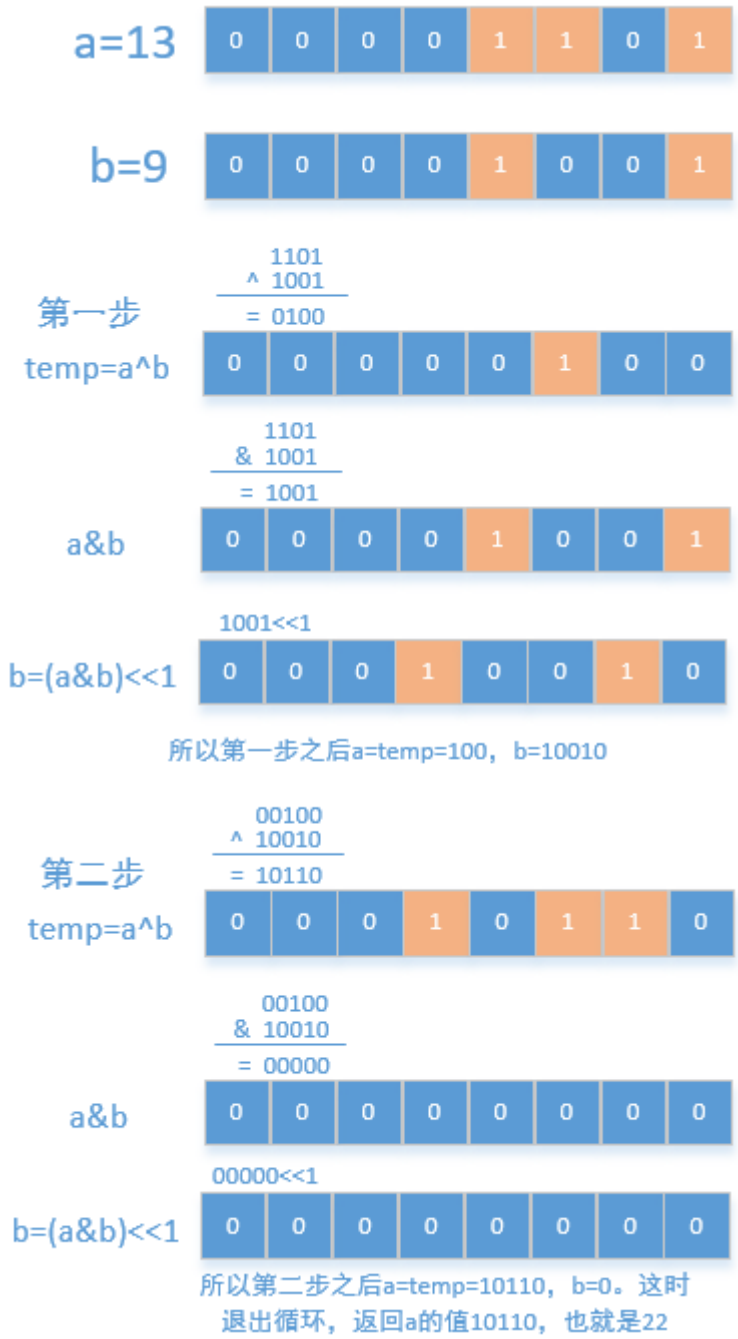
上面的代码可以不用看，我们来看一下运行结果

```
1+1=2
1+0=1
0+1=1
0+0=0
13+9=22
1+-1=0
-2147483647+2147483647=0
-8+-9=-17
```

经过测试，发现我们的代码完全正确，没有使用“+”实现了两个数相加。上面的递归我们还可以改为非递归的方式

```
private static int add3(int a, int b) {
  while (b != 0) {
    int temp = a ^ b;
    b = (a & b) << 1;
    a = temp;
  }
  return a;
}
```

这个也很好理解，每次计算的时候要对a和b进行重新赋值，然后再不断的循环，直到b等于0的时候停止循环，我们知道这里在运算的时候b表示的是进位的值，当b等于0的时候就表示没有进位，没有进位就退出循环，这就是使用位运算来实现加法。我们假设a=13，b=9来画个图加深一下理解



二：不使用“-”实现两个数相减

既然加法都实现了，那么减法就更容易了，a-b，直接改为a+(-b)即可，那么请等一下，我们不是说不能使用“-”吗，这里明显有了“-”符号，肯定不符合规则，那么别着急，在计算机中一个数的相反数还可以用另一种方式来表示，那就是

a的相反数是~a+1

上面“+”我们已经实现了，“~”不属于四则运算符，所以代码也很容易写出

```
1   private static int subtraction(int a, int b) {
2       return add3(a, add3(~b, 1));
3   }
```

这种实现就更简洁了，直接一行代码搞定，代码中add3(~b,1)表示的是-b。如果不使用加法是否能实现两个数相减呢，其实也是可以的，我们这样来思考一下，比如a-b

1，如果b等于0，我们直接返回a即可。如果b不等于0，我们可以先把a和b上同为1的数字给去掉，那么怎么去掉呢，其实很简单，我们先要计算c=a&b，那么c中为1的位置在a和b中相对应的位置上也是1，然后再通过异或运算就可以把它给移除。

2，在经过第一步执行之后，a和b在相同的位置上要么都是0，要么一个0一个1，不可能全是1了，那么下面就要会分为3种情况了（我们**先不考虑因不够减而借位的问题**）

(1)，如果a和b对应的位置上都是0，那么结果对应的位置上也是0。

(2)，如果a对应的位置是1，b对应的位置是0，结果对应的位置是1。

(3)，如果a对应的位置是0，b对应的位置是1，结果对应的位置是1。（向前借一位1）

所以在不考虑借位的情况下，对应位置上的结果其实就是a|b（对应位置都为1的在第一步就已经被踢出了），那么实际计算的时候我们不可能不考虑借位的问题，所以实际结果是(a|b)-(b<<1)，但这里又出现了“-”符号，所以不符合要求，这时我们可以使用递归的方式来解决，代码如下

```
1   private static int subtraction2(int a, int b) {
2     if (b == 0)
3       return a;
4     int c = a & b;
5     //下面两行是把a和b中相同位置为1的都消去
6     a ^= c;
7     b ^= c;
8     return subtraction2(a | b, b << 1);
9   }
```

当然我们还可以把它改为非递归的方式，像下面这样

```
1   private static int subtraction3(int a, int b) {
2     while (b != 0) {
3       int c = a & b;
4       a ^= c;
5       b ^= c;
6       a |= b;
7       b <<= 1;
8     }
9     return a;
10  }
```

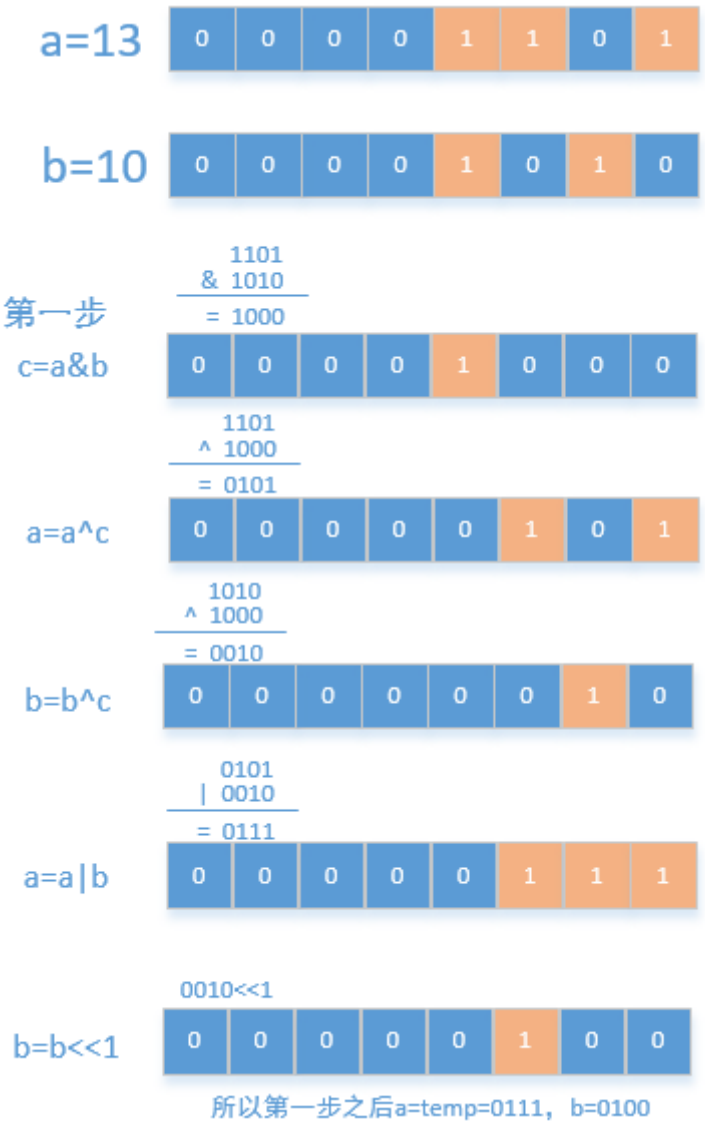
我们还是找几组数据来测试一下吧

```
1     int[] array = {1, 1, 1, 0, 0, 1, 0, 0, 13, 9, 1, -1, Integer.MAX_VALUE, Integer.MAX_VALUE, -8, -9, 10
2     for (int i = 0; i < array.length / 2; i++) {
3       System.out.println(array[i << 1] + "-" + array[(i << 1) + 1] + "=" + subtraction2(array[i << 1], a
4     }
```

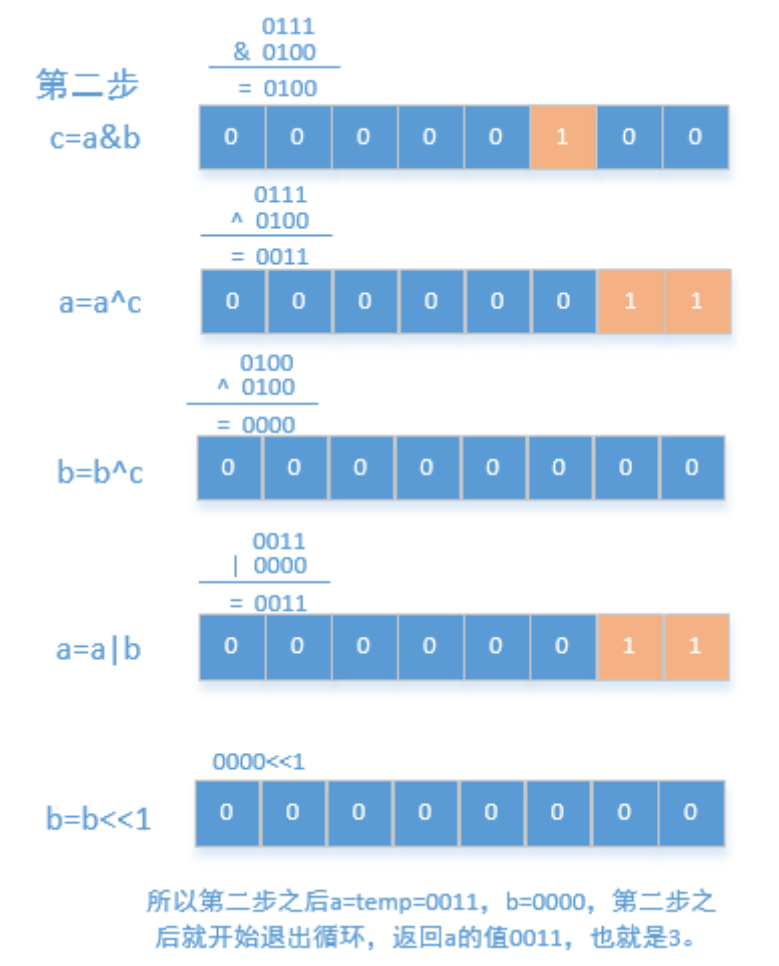
上面的我们可以不用看，直接看运行结果就行了

```
1-1=0
1-0=1
0-1=-1
0-0=0
13-9=4
1--1=2
2147483647-2147483647=0
-8--9=1
100-2147483647=-2147483547
```

我们以a=13，b=10来画个图加深一下理解



所以第一步之后a=temp=0111，b=0100



三：不使用“×”实现两个数相乘

我们先来看个例子，比如13*9，计算方式如下

```
    1101     a=13
×   1001     b=9
-----------------
    1101
1101
-----------------
1110101      结果为117
```

由上面公式我们可以看出只有b的某一位是1的时候和a相乘才有意义，如果b的某一位是0，那么和a相乘则永远都是0，所以我们计算的时候逐步遍历b的每一位，只有当他为1的时候才进行运算，我们来看下代码

```
//求一个数的相反数
private static int negative(int a) {
  return add3(~a, 1);
}

private static int mult(int a, int b) {
  int x = a < 0 ? negative(a) : a;//如果是负数，先转为正数再参与计算
  int y = b < 0 ? negative(b) : b;
  int res = 0;
  while (y != 0) {
    if ((y & 1) == 1)
      res = add3(res, x);
    x <<= 1;
    y >>= 1;
  }
```

```
    return (a ^ b) >= 0 ? res : negative(res);
  }
```

我们还是来找一组数据测试一下，验证我们代码的正确性

```
    int[] array = {1, 1, 1, 0, 0, 1, 0, 0, 13, 9, 1, -1, -8, -9, 100, 99};
    for (int i = 0; i < array.length / 2; i++) {
      System.out.println(array[i << 1] + "x" + array[(i << 1) + 1] + "=" + mult(array[i << 1], array[(i
    }
```

上面代码不用看，我们直接来看一下打印的数据，结果丝毫不差

```
1x1=1
1x0=0
0x1=0
0x0=0
13x9=117
1x-1=-1
-8x-9=72
100x99=9900
```

四：不使用“÷”实现两个数相除

a÷b的含义是a中包含多少个b，比如6÷3=2，7÷3=2，这里我们实现的除法和计算机中两个int类型相除结果是一样的，只记录商的值，余数会被舍去，所以我们想到的一种解法是用a不断的减去b，并记录减了多少次，所以代码很容易想到，我们来看下

```
  private static int div1(int a, int b) {
    int x = a < 0 ? negative(a) : a;
    int y = b < 0 ? negative(b) : b;
    if (x < y)
      return 0;
    return (a ^ b) >= 0 ? div1(subtraction(a, b), b) + 1 : div1(add3(a, b), b) - 1;
  }
```

上面的+1，-1直接改为上面的加法和减法即可，这里我为了方便阅读代码就没写。这种递归的实现效率不是很高，如果a非常大，b又比较小，很容易出现堆栈溢出异常，所以我们还可以把它改为非递归

```
  private static int div2(int a, int b) {
    int x = a < 0 ? negative(a) : a;
    int y = b < 0 ? negative(b) : b;
    int ocunt = 0;
    while (x >= y) {
      x = subtraction3(x, y);
      ocunt++;
    }
    return (a ^ b) >= 0 ? ocunt : -ocunt;
  }
```

这种虽然不会出现堆栈溢出异常了，但如果b是1，a是一个非常非常大的数，这样一直减下去也是非常慢的，我们还可以换种思路，每次减去的不是b，而是b的倍数，我们来看下代码

```
  private static int div3(int a, int b) {
    if (a == 0 || b == 0)
      return 0;//b不能为0，如果b是0我们应该抛异常的，这里简单处理就没抛
    int x = a < 0 ? negative(a) : a;
    int y = b < 0 ? negative(b) : b;
    int result = 0;
    for (int i = 31; i >= 0; i--) {
      if ((x >> i) >= y) {
        result = add3(result, 1 << i);
```

```
            x = subtraction3(x, y << i);
        }
    }
    return (a ^ b) >= 0 ? result : -result;
}
```

我们找一组非常极端的数据来测一下上面两种方法，看一下效率到底相差多少倍

```
int a = Integer.MAX_VALUE;
int b = 1;
long time = System.nanoTime();
System.out.println(a + "÷" + b + "=" + div2(a, b));
System.out.println("优化之前的时间：" + (System.nanoTime() - time));
time = System.nanoTime();
System.out.println(a + "÷" + b + "=" + div3(a, b));
System.out.println("优化之后的时间：" + (System.nanoTime() - time));
```

我们来看一下结果

```
2147483647÷1=2147483647
优化之前的时间：3923926100
2147483647÷1=2147483647
优化之后的时间：24700
```

这个时间相差还是非常大的，一个30多亿纳秒，一个两万多纳秒，相差十几万倍。最后我们再找一组数据测试一下我们的代码是否正确

```
int[] array = {1, 1, 0, 1, 13, 9, 40, 3, 1, -1, -8, -9, 100, 99};
for (int i = 0; i < array.length / 2; i++) {
    System.out.println("div2方法测试：" + array[i << 1] + "÷" + array[(i << 1) + 1] + "=" + div2(array
}
System.out.println("----------------------------------------");
for (int i = 0; i < array.length / 2; i++) {
    System.out.println("div3方法测试：" + array[i << 1] + "÷" + array[(i << 1) + 1] + "=" + div3(array
}
```

我们来看下运行结果

```
div2方法测试：1÷1=1
div2方法测试：0÷1=0
div2方法测试：13÷9=1
div2方法测试：40÷3=13
div2方法测试：1÷-1=-1
div2方法测试：-8÷-9=0
div2方法测试：100÷99=1
--------------------------
div3方法测试：1÷1=1
div3方法测试：0÷1=0
div3方法测试：13÷9=1
div3方法测试：40÷3=13
div3方法测试：1÷-1=-1
div3方法测试：-8÷-9=0
div3方法测试：100÷99=1
```

结果和我们预想的完全一致。

往期推荐

- 380，缺失的第一个正数（中）

361，交替位二进制数

原创 山大王wld 数据结构和算法 5月13日

收录于话题
#算法图文分析
96个 >

给定一个正整数，检查他是否为交替位二进制数：换句话说，就是他的二进制数相邻的两个位数永不相等。

示例 1:

```
输入：5
输出：True
解释：
5的二进制数是：101
```

示例 2:

```
输入：7
输出：False
解释：
7的二进制数是：111
```

示例 3:

```
输入：11
输出：False
解释：
11的二进制数是：1011
```

示例 4:

```
输入：10
输出：True
解释：
10的二进制数是：1010
```

答案：

```java
public boolean hasAlternatingBits(int n) {
    n ^= (n >> 1);
    return (n & (n + 1)) == 0;
}
```

解析：

这道题非常简单，一个数的二进制位如果是0和1交替，那么把这个数往右移一位然后再和原来的数进行异或运算，就会让他全部变为1（注意这里是忽略前面的0的，比如5在java语言中的二进制是32位，我们只考虑后面的101，不用考虑前面的29个0。）。我们来随便举个例子画个图看一下



所以他会把原来的位置上全部变为1，然后我们再加1，让他原来的位置全部变为0，然后在和原来的自己进行与运算，判断结果是否为0。

2，加法实现

上面的异或运算我们还可以改为加法运算，像下面这样

```
1  public boolean hasAlternatingBits(int n) {
2    n += n >> 1;
3    return (n & (n + 1)) == 0;
4  }
```

3，逐个判断

我们还可以使用最原始的方式，从右往左一个个判断0和1是否是交替出现的，代码也很简单，我们来看下

```
1  public boolean hasAlternatingBits(int n) {
2    int pre = n & 1;
3    for (int i = 1; i < 32; i++) {
4      if ((1 << i) > n)
5        break;
6      int cur = (n >> i) & 1;
7      if ((cur ^ pre) == 0)
8        return false;
9      pre = cur;
10   }
11   return true;
12 }
```

第7行如果等于0，说明要么连续出现了0，要么连续出现了1，直接返回false即可。当然第7行的判断我们还可以再改一下，换一种思路

```
1  public boolean hasAlternatingBits(int n) {
2    int pre = n & 1;
3    n >>>= 1;
4    while (n != 0) {
5      if ((n & 1) == pre)
6        return false;
7      pre = n & 1;
8      n >>>= 1;
9    }
10   return true;
11 }
```

第5行如果成立，说明要么连续出现了0，要么连续出现了1，和上面很类似，不过实现方式上有些差别。或者我们还可以不使用任何临时变量，像下面这样每两两前后比较

4，前后两两比较

```
public boolean hasAlternatingBits(int n) {
  while (n != 0 && (n >>> 1) != 0) {
    if (((n & 1) ^ ((n >>> 1) & 1)) == 0)
      return false;
    n = n >>> 1;
  }
  return true;
}
```

或者我们还可以把它给为递归的写法

5，递归实现

```
public boolean hasAlternatingBits(int n) {
  return n < 3 || ((n % 2) != (n / 2 % 2)) && hasAlternatingBits(n / 2);
}
```

6，移两位计算

再来想一下，前面我们把n往右移一位然后在与自己进行异或运算，我们能不能把n先往右移两位在进行异或运算呢，其实也是可以的，我们来画个图分析一下



我们只需要计算异或的结果类似于10000...这种形式的就可以了，所以代码很简单，直接一行搞定

```
public static boolean hasAlternatingBits(int n) {
  return ((n ^= n >> 2) & (n - 1)) == 0;
}
```

我们来找几个数据测试一下

```
  System.out.println("二进制0b111是不是0和1交替出现的：" + hasAlternatingBits(0b111));
  System.out.println("二进制0b101是不是0和1交替出现的：" + hasAlternatingBits(0b101));
  System.out.println("二进制0b1010101010是不是0和1交替出现的：" + hasAlternatingBits(0b1010101010));
  System.out.println("二进制0b1010101011是不是0和1交替出现的：" + hasAlternatingBits(0b1010101011));
```

看一下打印的结果

```
二进制0b111是不是0和1交替出现的：false
二进制0b101是不是0和1交替出现的：true
二进制0b1010101010是不是0和1交替出现的：true
二进制0b1010101011是不是0和1交替出现的：false
```

结果完全正确

7，乘以3计算

我们接着往下思考，如果n是0和1交替出现的，那么会有两种情况，一种是以0结尾的，比如101010，一种是以1结尾的，比如1010101，无论哪种情况我们把它乘以3会有一个奇怪的现象，因为是0和1交替出现，0乘以3还是0，1乘以3是11(二进制)，所以不会出现一直往前进位的问题，我们来看下代码

```
public static boolean hasAlternatingBits(int n) {
  return ((n * 3) & (n * 3 + 1) & (n * 3 + 2)) == 0;
}
```

我们还是来找几组数据来测试一下结果

```
  System.out.println("二进制0b111是不是0和1交替出现的：" + hasAlternatingBits(0b111000));
  System.out.println("二进制0b1010101010是不是0和1交替出现的：" + hasAlternatingBits(0b1010101010));
  System.out.println("二进制0b10101010101是不是0和1交替出现的：" + hasAlternatingBits(0b10101010101));
  System.out.println("二进制0b1010101011是不是0和1交替出现的：" + hasAlternatingBits(0b1010101011));
```

再来看一下打印结果

```
二进制0b111是不是0和1交替出现的：false
二进制0b1010101010是不是0和1交替出现的：true
二进制0b10101010101是不是0和1交替出现的：true
二进制0b1010101011是不是0和1交替出现的：false
```

结果完全在我们的预料之中。这种解法一般我们不太容易想到，但也不提倡使用，会有一些小的问题，因为如果n比较小的话是没问题的，如果n比较大的话，那么n*3就会出现数字溢出，导致结果错误。如果想写这种解法，最好先把n转为long类型就可以了。

8，使用java类库

如果允许的话我们还以使用java类库提供的方法先进行转换，然后再进行判断也是可以的，代码比较简单，我们来看下

```
public boolean hasAlternatingBits(int n) {
  String s = Integer.toBinaryString(n);
  for (int i = 0; i < s.length() - 1; i++) {
    if (s.charAt(i) == s.charAt(i + 1)) {
      return false;
    }
  }
  return true;
}
```

这种就非常好理解了，但对于二进制位的考察相对就弱了很多，或者我们还可以更简洁一些

```
public static boolean hasAlternatingBits(int n) {
  String binary = Integer.toBinaryString(n);
  return !binary.contains("00") && !binary.contains("11");
}
```

9，总结

这道题难度不大，其实解法还是挺多的，主要考察的是对二进制位的熟练使用，如果上面的所有解题思路都能掌握，代码也都能看的懂，那么对二进制位的操作也算是比较熟练的了，但远达不到精通，因为关于二进制位运算的技巧还是非常非常多的，不是这一篇文字就能讲的清楚的，后续也会有一系列对二进制位运算的介绍。

364，位1的个数系列（一）

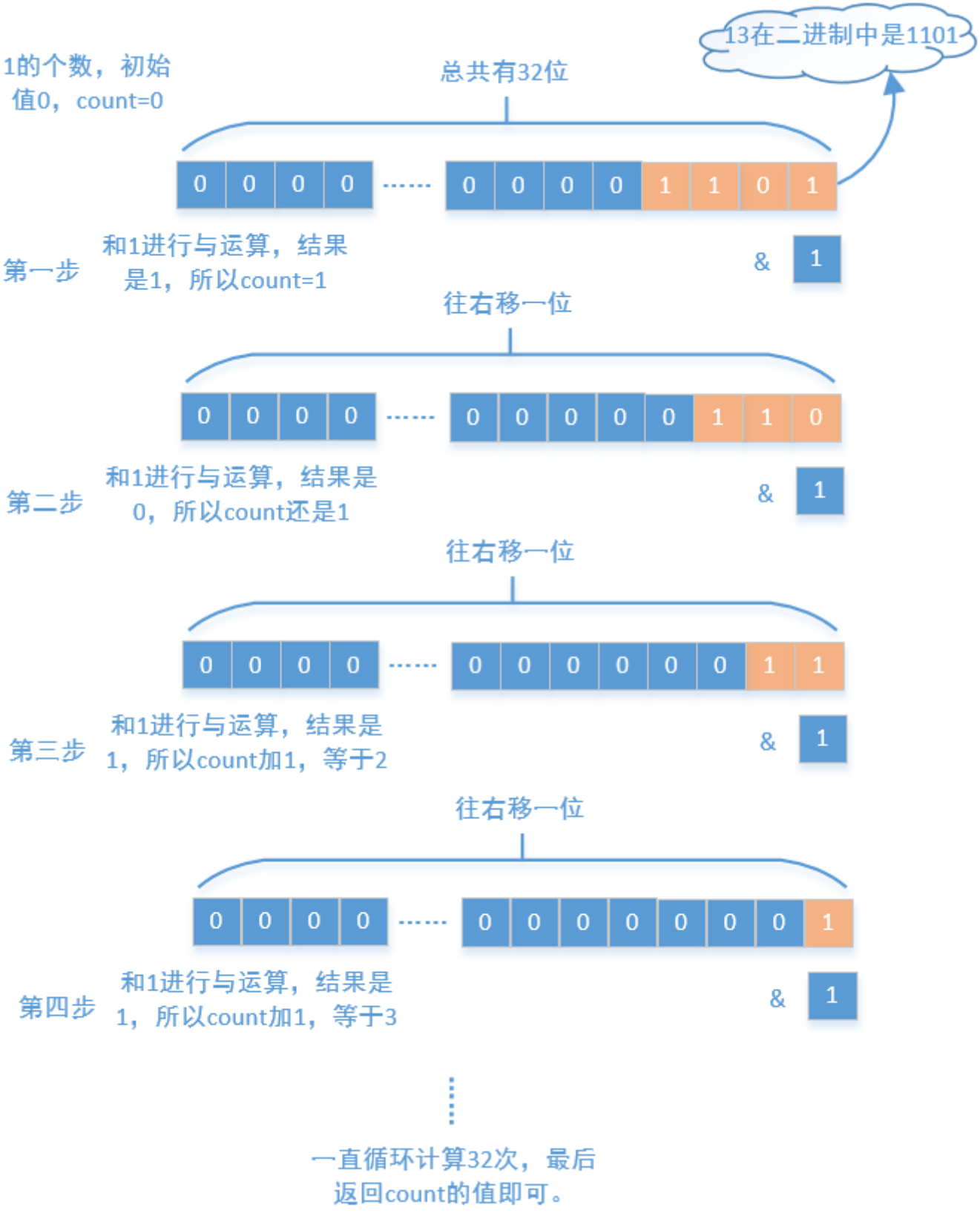
原创 山大王wld 数据结构和算法 5月19日

收录于话题
#算法图文分析
96个 >

我们知道在java语言中一个int类型有32个0或1组成。我们要计算有多少个1，这里主要以int型数据为例来分析。比如15在二进制中表示的是1111，有4个1，所以返回4。再比如16在二进制中表示的是10000，只有一个1，所以返回1。这题解法比较多，我们将会逐个分析。

一，通过移动数字计算

首先想到的是把要求的数字不停的往右移，然后再和1进行与运算，我们就以13为例画个图来分析下



看明白了上面的分析，代码就很容易多了，我们来看下代码

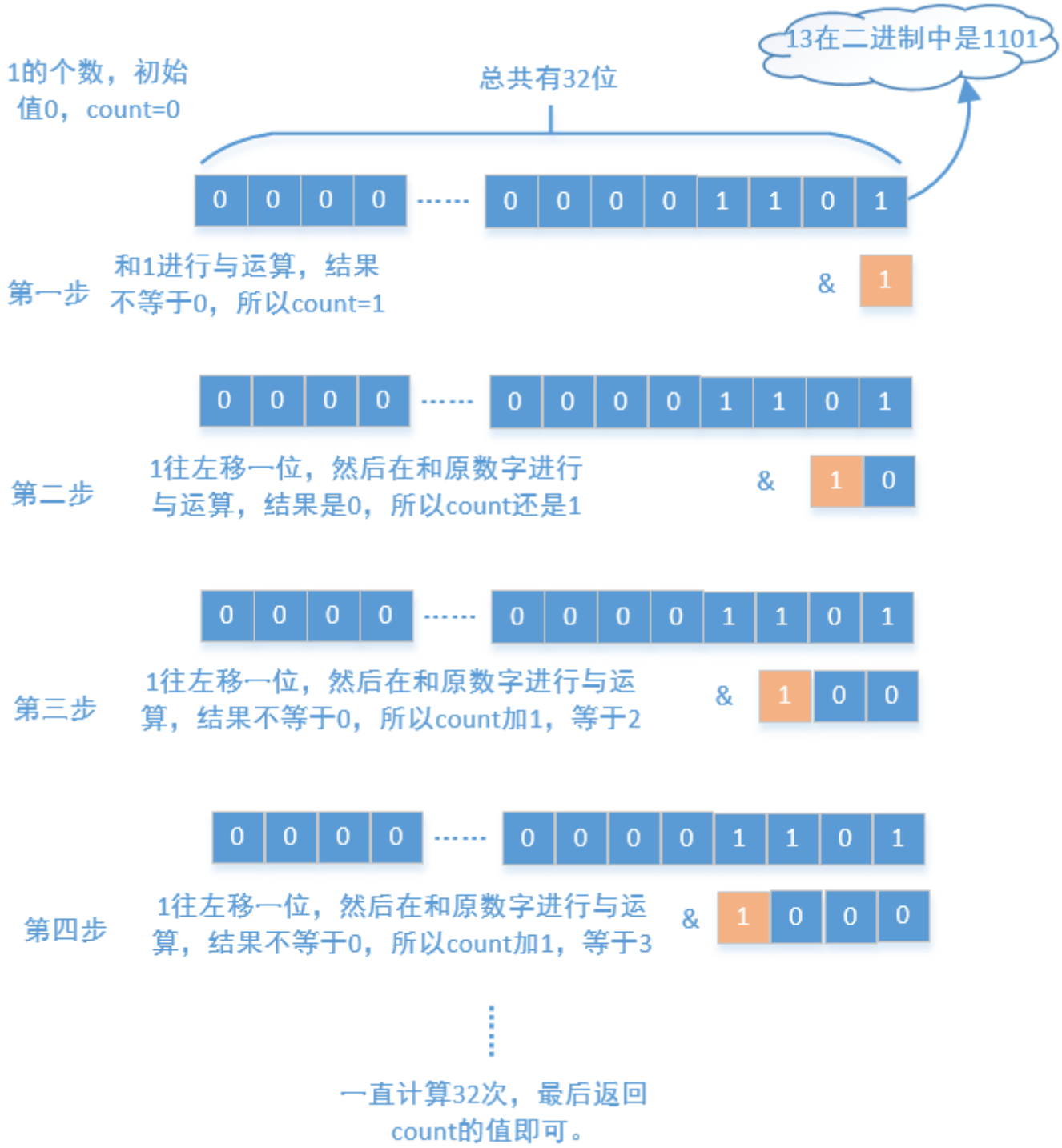
```
1  public int bitCount(int n) {
2    int count = 0;
3    for (int i = 0; i < 32; i++) {
4      if (((n >>> i) & 1) == 1) {
5        count++;
6      }
7    }
8    return count;
9  }
```

上面的分析中我们看到，如果一个数往右移了几步之后结果为0了，就没必要在计算了，所以代码我们还可以在优化一点

```
public int bitCount(int n) {
  int count = 0;
  while (n != 0) {
    count += n & 1;
    n = n >>> 1;
  }
  return count;
}
```

二，通过移动1来计算

上面我们使用的是把一个数字不断的往右移动，其实我们还可以保持原数字不变，用1和他进行与运算，然后通过移动1的位置来计算，这里我们判断的标准不是等于1，而是不等于0。我们还以13为例来画个图分析一下



这次我们移动的是1，我们来看一下代码

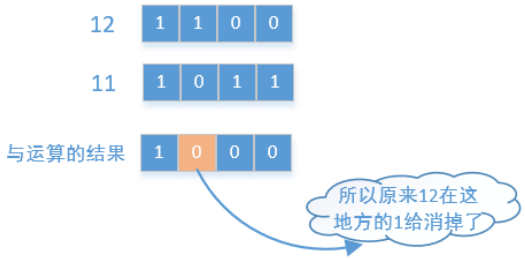
```
public int bitCount(int n) {
  int count = 0;
  for (int i = 0; i < 32; i++) {
    if ((n & (1 << i)) != 0) {
      count++;
    }
  }
  return count;
}
```

当然我们还可以通过运算的结果是否是1来判断也是可以的，我们只需要把往左移的1和n运算完之后再往右移即可，我们来看下代码

```
public int bitCount(int i) {
  int count = 0;
  for (int j = 0; j < 32; j++) {
    if ((i & (1 << j)) >>> j == 1)
      count++;
  }
  return count;
}
```

三，不通过移位计算

前面两种方式要么是移动原数字，要么是移动1，这次我们不移动任何数字来计算。在位运算中有这样一个操作，就是n&(n-1)可以把n最右边的1给消掉。举个例子，当n=12的时候，我们来画个图看一下



明白了这个知识点，代码就很容易写了，我们通过循环计算，不断的把n右边的1给一个个消掉，直到n等于0为止

```
public int bitCount(int n) {
  int count = 0;
  while (n != 0) {
    n &= n - 1;
    count++;
  }
  return count;
}
```

我们还可以把它给为递归的写法，直接一行代码搞定

```
public int bitCount(int n) {
  return n == 0 ? 0 : 1 + bitCount(n & (n - 1));
}
```

四，查表

我们还可以通过查表来计算，我们先要把0到15转化为二进制，记录下每个数字包含1的个数再构成一张表，然后再把数字n每4位进行一次计算，图就不画了，代码中有注释，我们来看下代码

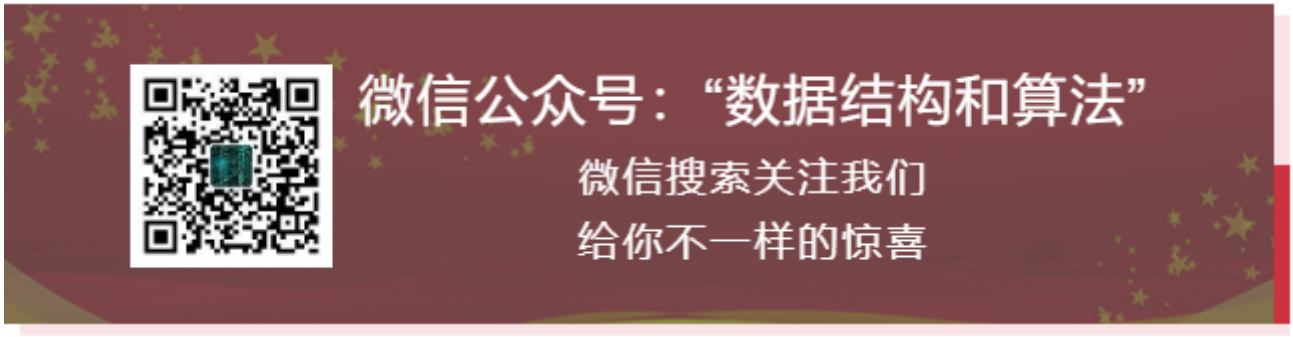
```java
public int bitCount(int i) {
  //table是0到15转化为二进制时1的个数
  int table[] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
  int count = 0;
  while (i != 0) {//通过每4位计算一次，求出包含1的个数
    count += table[i & 0xf];
    i >>>= 4;
  }
  return count;
}
```

这题我感觉还是比较经典的，因为他的解法非常多，上面的几种要么使用循环，要么使用递归，要么是查表，其实我们还可以上面几种方式都不使用，也可以计算1的个数，这个后面会再讲。

385，位1的个数系列（二）

原创 山大王wld 数据结构和算法 6月16日

收录于话题
#算法图文分析
96个 >



The two most important days in your life are the day you are born and the day you find out why.

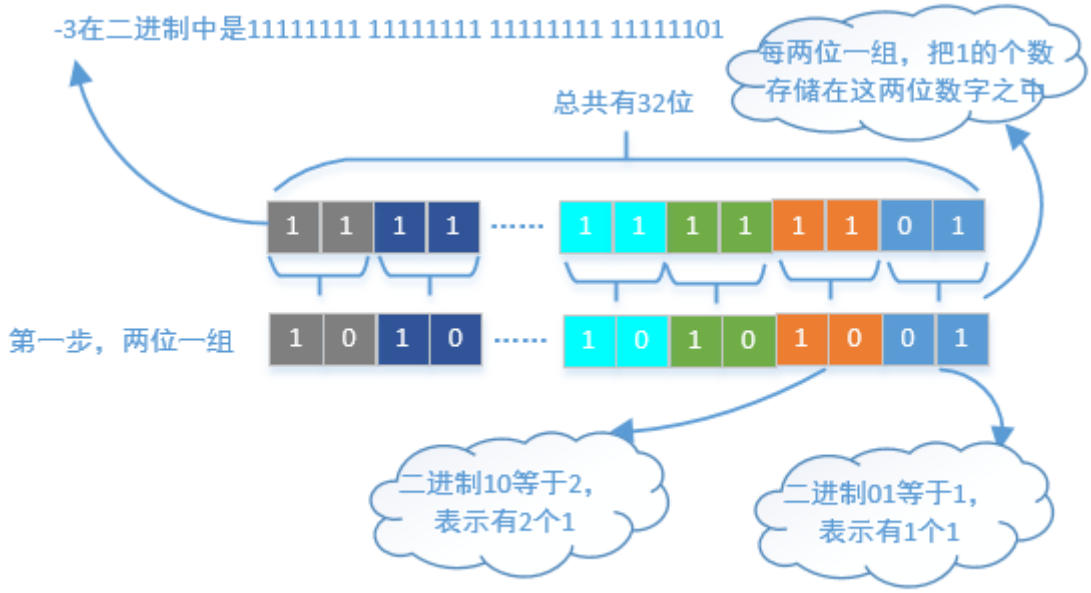
你人生中有两个最重要的日子：一是你出生的日子，二是你发现自己存在意义的日子。

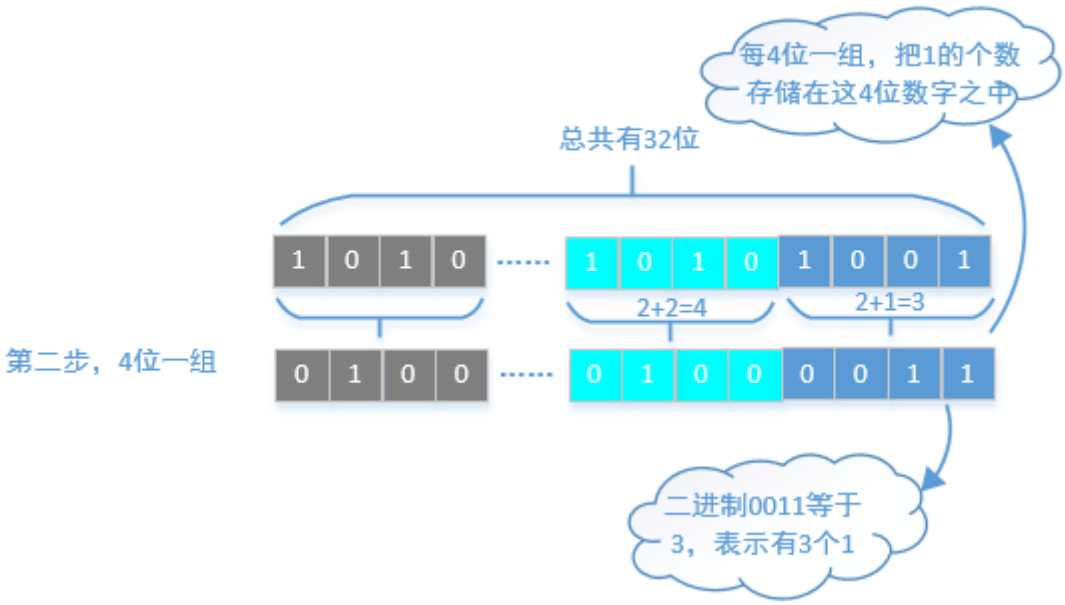
问题描述

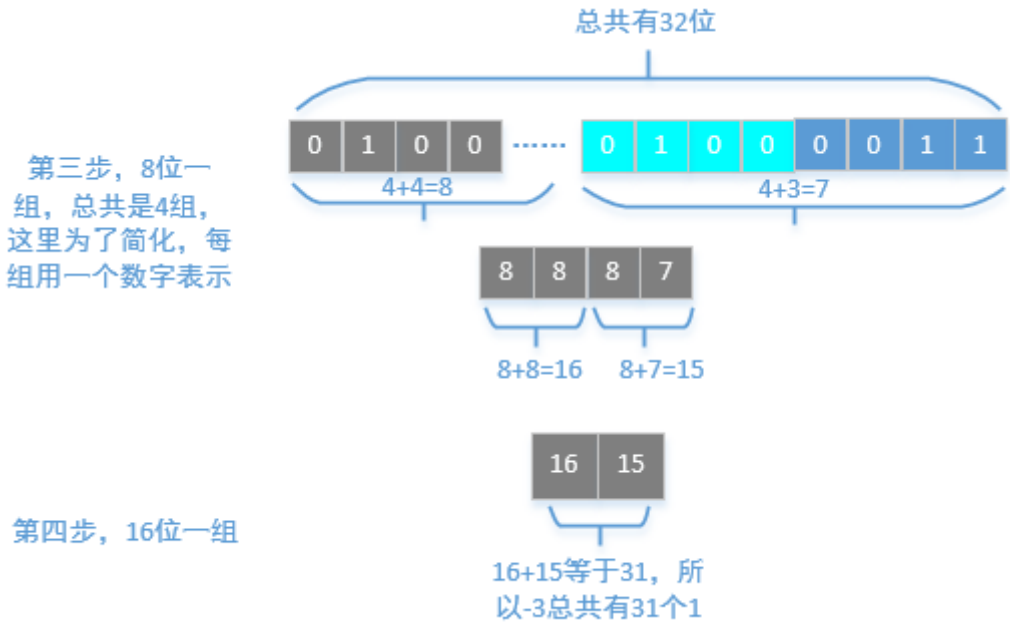
前面我们讲了364，位1的个数系列（一），使用了代码和图形结合的方式，很容易理解，下面再来看另一种解法，如果之前没看过的话，估计不太容易想到。int类型是32位，每位要么是0要么是1，其实我们完全可以把1的个数存储在这二进制位中，我们先来看一个每两位两位存储的方式

1，每两位存储

我们先来看一下，两位二进制位总共有4种表示，分别是00，01，10，11。他们中1的个数分别是0，1，1，2，最大值是2，而两位二进制表示的最大数是3，所以足够存储了，我们就以-3为例来画个图分析一下


-3在二进制中是11111111 11111111 11111111 11111101
每两位一组，把1的个数存储在这两位数字之中
总共有32位
第一步，两位一组
二进制10等于2，表示有2个1
二进制01等于1，表示有1个1


每4位一组，把1的个数存储在这4位数字之中
总共有32位
2+2=4
2+1=3
第二步，4位一组
二进制0011等于3，表示有3个1


总共有32位
4+4=8
4+3=7
第三步，8位一组，总共是4组，这里为了简化，每组用一个数字表示
8+8=16
8+7=15
第四步，16位一组
16+15等于31，所以-3总共有31个1

看明白了上面的图，代码就很好写了，我们来看下

```
public int bitCount(int n) {
    n = ((n & 0xaaaaaaaa) >>> 1) + (n & 0x55555555);
    n = ((n & 0xcccccccc) >>> 2) + (n & 0x33333333);
    n = (((n & 0xf0f0f0f0) >>> 4) + (n & 0x0f0f0f0f));
    n = n + (n >>> 8);
    n = n + (n >>> 16);
    return n & 63;
}
```

这里以0x开头的数字是16进制的，乍一看，上面密密麻麻的数字看起来可能容易犯晕，其实不用怕，我们只需要把上面的16进制改为二进制就很容易看明白了，我们来打印一下

```
    System.out.println("16进制0xaaaaaaaa转化为二进制是----->" + Util.bitInt32(0xaaaaaaaa));
    System.out.println("16进制0x55555555转化为二进制是----->" + Util.bitInt32(0x55555555));
    System.out.println("16进制0xcccccccc转化为二进制是----->" + Util.bitInt32(0xcccccccc));
    System.out.println("16进制0x33333333转化为二进制是----->" + Util.bitInt32(0x33333333));
    System.out.println("16进制0xf0f0f0f0转化为二进制是----->" + Util.bitInt32(0xf0f0f0f0));
    System.out.println("16进制0x0f0f0f0f转化为二进制是----->" + Util.bitInt32(0x0f0f0f0f));
```

再来看一下运行结果

```
      16进制0xaaaaaaaa转化为二进制是----->10101010 10101010 10101010 10101010
      16进制0x55555555转化为二进制是----->01010101 01010101 01010101 01010101
      16进制0xcccccccc转化为二进制是----->11001100 11001100 11001100 11001100
      16进制0x33333333转化为二进制是----->00110011 00110011 00110011 00110011
      16进制0xf0f0f0f0转化为二进制是----->11110000 11110000 11110000 11110000
      16进制0x0f0f0f0f转化为二进制是----->00001111 00001111 00001111 00001111
```

我们很容易发现规律，第一个是10循环，第二个是01循环，第3个是1100循环，第4个是0011循环……也就是上面图中分析的每2个，每4个，每8个……一组进行运算。

上面都是先运算之后再移位进行操作，其实我们还可以先移位之后再进行运算，我们看下代码

```
public int hammingWeight(int n) {
    n = ((n >>> 1) & 0x55555555) + (n & 0x55555555);
    n = ((n >>> 2) & 0x33333333) + (n & 0x33333333);
    n = (((n >>> 4) & 0x0f0f0f0f) + (n & 0x0f0f0f0f));
    n = n + (n >>> 8);
    n = n + (n >>> 16);
    return n & 63;
}
```

写法上虽然有差别，但整体思路还是一样。这里再提到一点，代码的最后为什么要和63进行与运算，这是因为在第5和6行也就是每8位和每16位运算的时候，前面已经计算过的数据没有过滤掉，而63的二进制表示是00111111(6个1)。如果我们把前面计算过的值过滤掉就不用再和63进行与运算了，我们就以第一种写法为例来写一下

```
public int bitCount(int n) {
    n = ((n & 0xaaaaaaaa) >>> 1) + (n & 0x55555555);
    n = ((n & 0xcccccccc) >>> 2) + (n & 0x33333333);
    n = (((n & 0xf0f0f0f0) >>> 4) + (n & 0x0f0f0f0f));
    n = (((n & 0xff00ff00) >>> 8) + (n & 0x00ff00ff));
    n = (((n & 0xffff0000) >>> 16) + (n & 0x0000ffff));
    return n;
}
```

我们来找几个数据测试一下，使用最后一个方法看一下运行结果

```
    int num = 100;
    System.out.println(num + " 的二进制是----->" + Util.bitInt32(num) + "---->1的个数是： " + bitCount(nu
    num = 1024;
    System.out.println(num + "的二进制是----->" + Util.bitInt32(num) + "---->1的个数是： " + bitCount(nu
```

```
    num = 0;
    System.out.println(num + "  的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(n
    num = -1;
    System.out.println(num + "  的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(nu
    num = -2;
    System.out.println(num + "  的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(nu
    num = -100;
    System.out.println(num + "的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(num
```

我们直接看打印结果就行了

```
 100 的二进制是----->00000000 00000000 00000000 01100100 ---->1的个数是：3
 1024的二进制是----->00000000 00000000 00000100 00000000 ---->1的个数是：1
 0  的二进制是----->00000000 00000000 00000000 00000000 ---->1的个数是：0
 -1  的二进制是----->11111111 11111111 11111111 11111111 ---->1的个数是：32
 -2  的二进制是----->11111111 11111111 11111111 11111110 ---->1的个数是：31
 -100的二进制是----->11111111 11111111 11111111 10011100 ---->1的个数是：28
```

结果证明我们的写法是完全正确的。

2，每四位存储

上面我们讲了是每两位存储，那么我们能不能每4位存储呢，其实也是可以的，我们只需要在第一步计算的时候每4位中1的个数存储在这4位中，后面的代码就和上面一样了。明白了上面的解法，这种就非常简单了，图就不再画了，我们来看下代码

```
public int bitCount(int n) {
  n = (n & 0x11111111) + ((n >>> 1) & 0x11111111) + ((n >>> 2) & 0x11111111) + ((n >>> 3) & 0x11111111);
  n = (((n & 0xf0f0f0f0) >>> 4) + (n & 0x0f0f0f0f));
  n = n + (n >>> 8);
  n = n + (n >>> 16);
  return n & 63;
}
```

和上面不同的主要是在第2行，他是先把每4位中包含的1全部都统计一遍，然后再存储在这4位二进制位中，后面再进行计算。

3，每多位存储

上面我们分别采用了每两位存储，每4位存储，我们来头脑风暴一下，我们能不能每8位，每16位呢，当然也是可以的，原理和上面类似，只不过是在第一步统计1的个数的时候可能会麻烦些，代码就不再写了。我们再来思考一下，上面无论是每2位，4位，8位，还是16位，他们都能被32整除。我们能不能使用一个不能被32整除的数来存储呢，比如3，或者5，7……，其实只要不是超过32的正整数都是可以的，只不过使用一个不能被32整除的数计算可能稍微会麻烦些，我们就拿每3位存储来分析一下。

3明显不能被32整除，所以我们不能完全照着上面来做，我们可以这么来做，右边30位每3位分为一组存储1的个数，最左边2个是一组来存储1的个数，我们来看下代码怎么写

```
public static int bitCount(int n) {
  n = (n & 011111111111) + ((n >>> 1) & 011111111111) + ((n >>> 2) & 011111111111);
  n = ((n + (n >>> 3)) & 030707070707);
  n = ((n + (n >>> 6)) & 07700770077);
  n = ((n + (n >>> 12)) & 037700007777);
  return ((n + (n >>> 24))) & 63;
}
```

注意这里以0开头的数字是8进制的，我们来找几组数据再来测试一下

```
    int num = 100;
    System.out.println(num + " 的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(nu
    num = Integer.MAX_VALUE;
    System.out.println(num + "的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(nu
    num = 0;
    System.out.println(num + "  的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(n
    num = -1;
    System.out.println(num + " 的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(nu
    num = -7;
    System.out.println(num + " 的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(nu
    num = Integer.MIN_VALUE;
    System.out.println(num + "的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(nu
```

直接来看运行结果

```
    100 的二进制是----->00000000 00000000 00000000 01100100 ---->1的个数是：3
    2147483647的二进制是----->01111111 11111111 11111111 11111111 ---->1的个数是：31
    0  的二进制是----->00000000 00000000 00000000 00000000 ---->1的个数是：0
    -1 的二进制是----->11111111 11111111 11111111 11111111 ---->1的个数是：32
    -7 的二进制是----->11111111 11111111 11111111 11111001 ---->1的个数是：30
    -2147483648的二进制是----->10000000 00000000 00000000 00000000 ---->1的个数是：1
```

结果完全完全在我们的预料之中。

我们看到每组的数量即使不能被32整除，我们依然可以计算，但上面的计算还有一个巧的地方，就是每3个一组的时候，可以用8进制的3个1来参与运算，如果每5，7，9……个一组能不能计算？当然也是可以的，下面我们以每5个一组来看一下怎么计算的。

```
public static int bitCount(int n) {
  n = (n & 0x42108421) + ((n >>> 1) & 0x42108421) + ((n >>> 2) & 0x42108421) + ((n >>> 3) & 0x42108421) + ((
  n = ((n + (n >>> 5)) & 0xc1f07c1f);
  n = ((n + (n >>> 10) + (n >>> 20) + (n >>> 30)) & 63);
  return n;
}
```

我们再来找几组数据测试一下

```
    int num = 100000000;
    System.out.println(num + " 的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(nu
    num = Integer.MAX_VALUE;
    System.out.println(num + "的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(nu
    num = 0;
    System.out.println(num + "  的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(n
    num = -1;
    System.out.println(num + " 的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(nu
    num = Integer.MAX_VALUE - 10000;
    System.out.println(num + " 的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(nu
    num = Integer.MIN_VALUE;
    System.out.println(num + "的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(nu
```

再来看一下结果

```
    100000000 的二进制是----->00000101 11110101 11100001 00000000 ---->1的个数是：12
    2147483647的二进制是----->01111111 11111111 11111111 11111111 ---->1的个数是：31
    0  的二进制是----->00000000 00000000 00000000 00000000 ---->1的个数是：0
    -1 的二进制是----->11111111 11111111 11111111 11111111 ---->1的个数是：32
    2147473647 的二进制是----->01111111 11111111 11011000 11101111 ---->1的个数是：26
    -2147483648的二进制是----->10000000 00000000 00000000 00000000 ---->1的个数是：1
```

4，总结

我们如果以2，3，4……32分为一组计算，那么这题的解法就非常多了，在加上之前讲的几种解法就更多了。如果搞懂了上面的代码，你会发现之前使用循环的方式统计简直弱爆了，这种写法在面试中我觉得会更有优势，但前提是面试官要懂。其实这题的解法还远远不止这些，后面我们还会再讲对这题的其他的一些算法。

往期推荐

- 383，不使用“+”,“-”,“×”,“÷”实现四则运算
- 364，位1的个数系列（一）
- 361，交替位二进制数

402，位1的个数系列（三）

原创 山大王wld 数据结构和算法 7月17日

收录于话题
#算法图文分析

95个 >



Yesterday is history. Tomorrow is mystery. But today is a gift.

昨日已逝，明天尚远，今天才是老天赐予的礼物。

问题描述

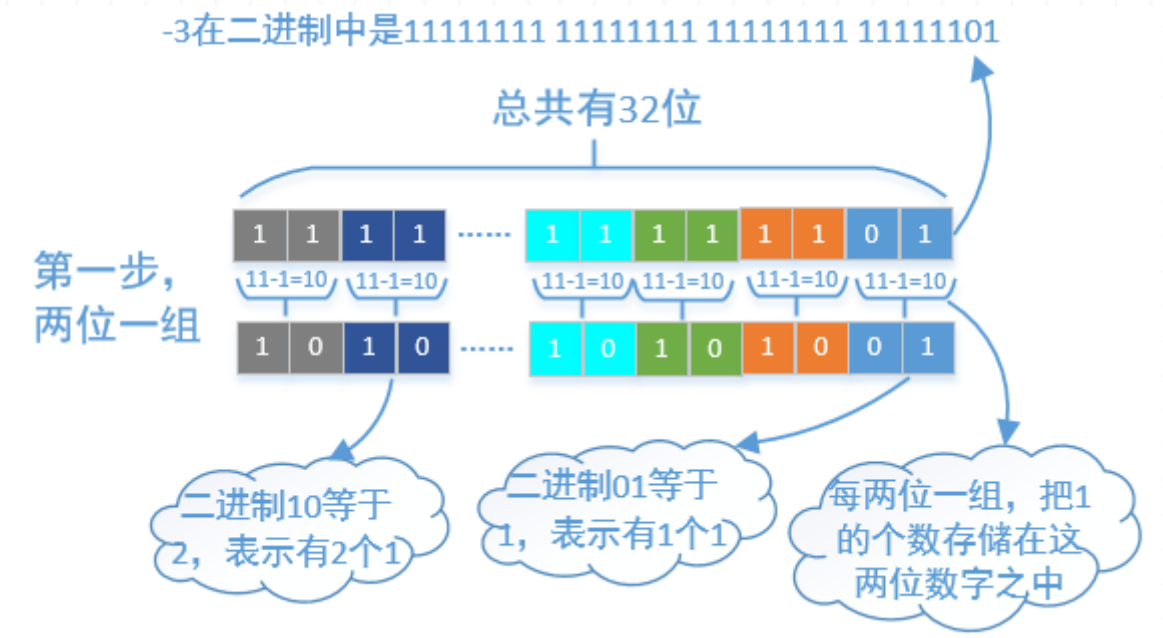
之前我们有两篇是讲了位1的个数，没看过的可以看下，364，位1的个数系列（一），385，位1的个数系列（二）。其中上一篇我们没有使用for循环以及while循环，使用的是相加的方式，今天我们讲的是和上一题非常相似的解法，相减的方式。

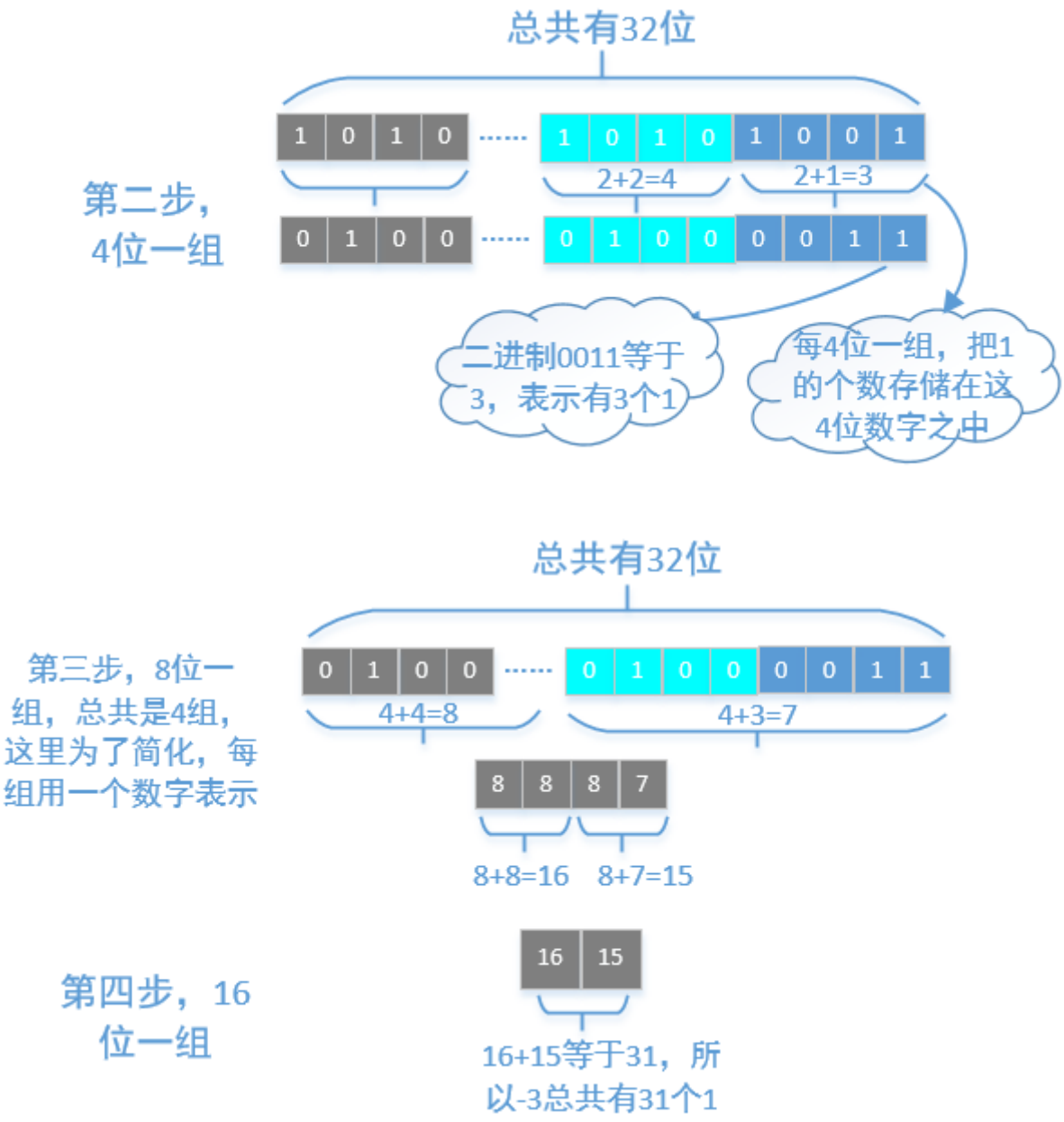
每两位存储

比如我们用两位数字存储的时候，我们只需要用这两位数减去偶数位上的值（从右往左数），比如

- 二进制00：00-0=00（表示有0个1）
- 二进制01：01-0=01（表示有1个1）
- 二进制10：10-1=01（表示有1个1）
- 二进制11：11-1=10（表示有2个1）

我们发现除了第一步和《位1的个数系列（二）》不同以外，其他的我们都可以完全按照上一题的方式来写，我们还是以-3为例来画个图看一下





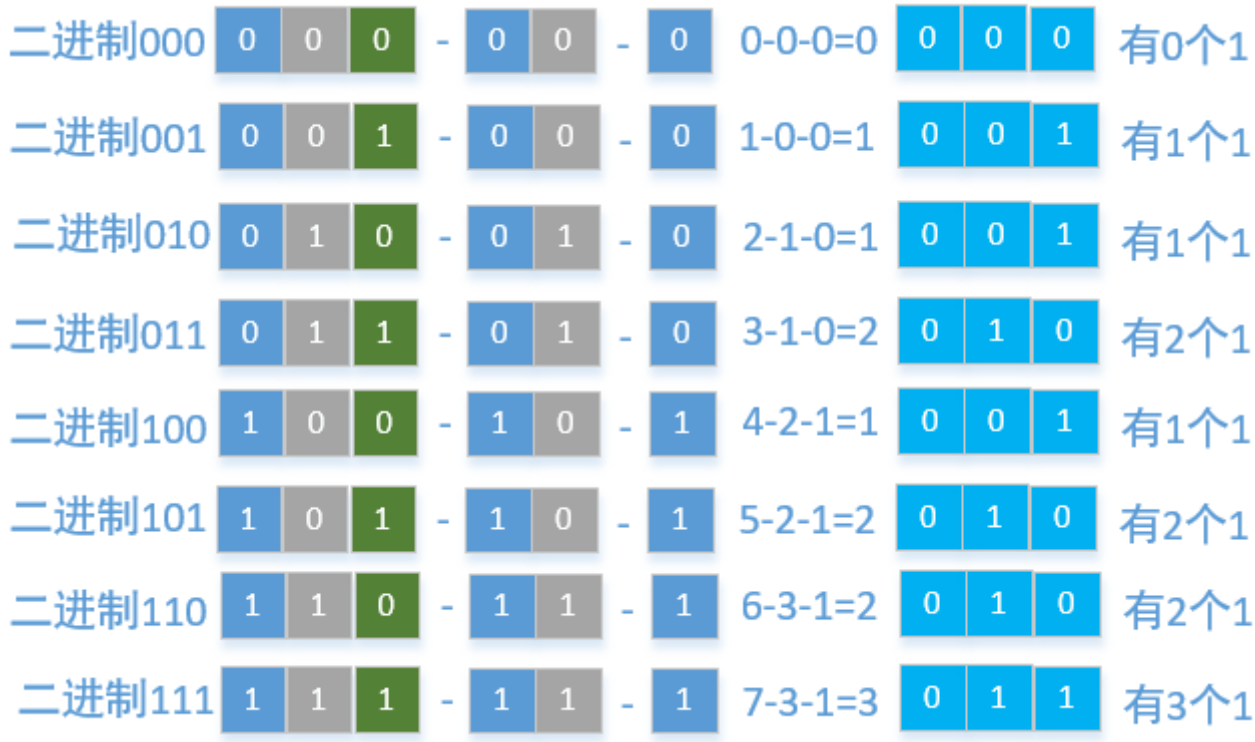
上面的分析如果能够看的懂的话，那么代码就容易多了，我们来看下

```
public static int bitCount(int i) {
  i = i - ((i >>> 1) & 0x55555555);
  i = (i & 0x33333333) + ((i >>> 2) & 0x33333333);
  i = (i + (i >>> 4)) & 0x0f0f0f0f;
  i = i + (i >>> 8);
  i = i + (i >>> 16);
  return i & 0x3f;
}
```

具体细节就不再这里分析了，如果想了解更多可以看《位1的个数系列（二）》

每3位存储

我们两位能存储，那么3位呢，看过《位1的个数系列（二）》的都知道，这也是可以的，之前我们介绍过的是通过一个个相加，也很好理解，那么这里如果使用的是相减应该怎么计算呢，我们来画个图看一下吧



通过上面的图分析我们找到了规律，就是每3位存储的话，只需要执行a=a-(a>>>1)-(a>>>2)就可以把数字存储到这3位数中，原理很简单，我们来看下代码

```
public static int bitCount(int n) {
  n = n - ((n >>> 1) & 033333333333) - ((n >>> 2) & 011111111111);
  n = ((n + (n >>> 3)) & 030707070707);
  n = ((n + (n >>> 6)) & 07700770077);
  n = ((n + (n >>> 12)) & 037700007777);
  return ((n + (n >>> 24))) & 63;
}
```

结果对不对，我们只有找几组数据经过测试之后才具有说服力，我们顺便找几组数据测试一下

```
    int num = 100000000;
    System.out.println(num + " 的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount
    num = Integer.MAX_VALUE;
    System.out.println(num + "的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount
    num = 0;
    System.out.println(num + "  的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCoun
    num = -1;
    System.out.println(num + " 的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCoun
    num = Integer.MAX_VALUE - 10000;
    System.out.println(num + " 的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCoun
    num = Integer.MIN_VALUE;
    System.out.println(num + "的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount
```

我们再来看一下打印结果

```
 100000000 的二进制是----->00000101 11110101 11100001 00000000 ---->1的个数是：12
 2147483647的二进制是----->01111111 11111111 11111111 11111111 ---->1的个数是：31
 0  的二进制是----->00000000 00000000 00000000 00000000 ---->1的个数是：0
 -1 的二进制是----->11111111 11111111 11111111 11111111 ---->1的个数是：32
 2147473647 的二进制是----->01111111 11111111 11011000 11101111 ---->1的个数是：26
 -2147483648的二进制是----->10000000 00000000 00000000 00000000 ---->1的个数是：1
```

结果丝毫不差。

每多位存储

看明白了上面每3位存储的分析过程，那么这题解法就比较多了，我们可以类推每4位存储的时候我们只需要执行a=a-(a>>>1)-(a>>>2)-(a>>>3)，就可以把每4位1的个数存储在这4位数中，同理每5位存储的时候我们只需要执行a=a-(a>>>1)-(a>>>2)-(a>>>3)-(a>>>4)，就可以把每5位1的个数存储在这5位数中。每6位的时候……，那么这样写下去就非常多了。我们就分别以每4位和每5位为例来写一下。首先写的是每4位为一组来存储

```
public static int bitCount(int n) {
    int tmp = n - ((n >>> 1) & 0x77777777) - ((n >>> 2) & 0x33333333) - ((n >>> 3) & 0x11111111);
    tmp = ((tmp + (tmp >>> 4)) & 0x0f0f0f0f);
    tmp = ((tmp + (tmp >>> 8)) & 0x00ff00ff);
    return ((tmp + (tmp >>> 16)) & 0x0000ffff) % 63;
}
```

再来看一个每5位为一组来存储

```
public static int bitCount(int n) {
    n = n - ((n >>> 1) & 0xdef7bdef) - ((n >>> 2) & 0xce739ce7) - ((n >>> 3) & 0xc6318c63) - ((n >>> 4) & 0x
    n = ((n + (n >>> 5)) & 0xc1f07c1f);
    n = ((n + (n >>> 10) + (n >>> 20) + (n >>> 30)) & 63);
    return n;
}
```

我们随便找几组数据测试一下

```
    int num = 1000;
    System.out.println(num + " 的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount
    num = Integer.MAX_VALUE;
    System.out.println(num + "的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(
    num = 0;
    System.out.println(num + "  的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCoun
    num = -1;
    System.out.println(num + "  的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCoun
    num = -8;
    System.out.println(num + "  的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCoun
    num = Integer.MIN_VALUE;
    System.out.println(num + "的二进制是----->" + Util.bitInt32(num) + "---->1的个数是：" + bitCount(
```

我们就用最后一个每5位为一组的来测试一下，来看下结果

```
1000 的二进制是----->00000000 00000000 00000011 11101000 ---->1的个数是：6
2147483647的二进制是----->01111111 11111111 11111111 11111111 ---->1的个数是：31
0  的二进制是----->00000000 00000000 00000000 00000000 ---->1的个数是：0
-1  的二进制是----->11111111 11111111 11111111 11111111 ---->1的个数是：32
-8  的二进制是----->11111111 11111111 11111111 11111000 ---->1的个数是：29
-2147483648的二进制是----->10000000 00000000 00000000 00000000 ---->1的个数是：1
```

结果完全正确

总结

我们通过这3个系列，算是把这道题完全讲透了，这题虽然看似简单，但其中蕴含的解题思路确是非常多的，如果这题的所有解法都看懂了，那么对二进制的掌握程度将会有一个很大的提升。

357，交换两个数字的值

原创 山大王wld 数据结构和算法 5月7日

收录于话题
#算法图文分析
96个 >

1，临时变量实现

一般情况下交换两个数字的值，我们都会使用一个临时变量，像下面这样

```
1   private void swap(int[] array, int i, int j) {
2     int temp = array[i];
3     array[i] = array[j];
4     array[j] = temp;
5   }
```

当然这段代码非常简单，哪怕是刚接触过编程的同学也都能看的懂，我们今天要讲的肯定不是上面这段代码这么简单。那么除了上面这种方法还有没有其他的方法呢，在前面我们大致提到过3种实现方式，但具体细节没有详细讲解，今天我们就来一起看下，首先我们来看加法的实现

2，加法实现

```
1   private void swap2(int[] array, int i, int j) {
2     if (i != j) {
3       array[i] = array[i] + array[j];
4       array[j] = array[i] - array[j];
5       array[i] = array[i] - array[j];
6     }
7   }
```

1，第3行相当于把array[i]和array[j]的和存放到了**array[i]**中，这里用粗体表示。

2，第4行array[j]=**array[i]**-array[j]；相当于array[j]=(array[i]+array[j])-array[j]；也就是array[j]=array[i]，相当于把array[i]的值赋值给了array[j]。

3，第5行array[i]=**array[i]**-array[j]；因为上一步已经把array[i]赋值给了array[j]，所以这里相当于array[i]=(array[i]+array[j])-array[i]；也就是array[i]=array[j]；所以最终实现了array[i]和array[j]的数值交换。

上面其实很好理解，不需要再过多的讨论，下面我们来分析一下当array[i]和array[j]都非常大，相加的时候出现了数字溢出，该怎么办，其实这个不用担心的，因为如果出现了溢出，最多也只能溢出一位，而int类型的最高位是符号位，是1表示的是负数，0表示的是非负数，而这个符号位是可以参与加减运算的。

我们要明白一点，在计算机中所有的加减法其实都是加法，首先是把我们要计算的数转换为二进制，然后再进行运算，负数会以补码的形式存在，当然正数的补码和原码相同。我们先来看一段非常简单的代码

```
1   public static void main(String[] args) {
2     int a = -3;
```

```
    int b = -5;
    int c = a - b;//其实相当于a+(-b);
    System.out.println("a的值是:" + a + "-->二进制: " + Util.bitInt32(a));
    System.out.println("-b的值是:" + -b + "-->二进制: " + Util.bitInt32(-b));
    System.out.println("c的值是:" + c + "--> 二进制: " + Util.bitInt32(c));
  }
```

看一下执行的结果

```
a的值是:-3-->二进制: 11111111 11111111 11111111 11111101
-b的值是:5-->二进制: 00000000 00000000 00000000 00000101
c的值是:2--> 二进制: 00000000 00000000 00000000 00000010
```

我们看到a+（-b）最高位其实已经出现了溢出，但这并不影响最终的结果。

3，减法实现

上面我们使用了加法交换两个数字的值，那么能不能使用减法呢，当然也是可以的，我们来看下代码

```
private void swap3(int[] array, int i, int j) {
  if (i != j) {
    array[i] = array[i] - array[j];
    array[j] = array[i] + array[j];
    array[i] = array[j] - array[i];
  }
}
```

其实原理都是一样的。

4，乘法实现

那我们能不能使用乘法呢，我们知道加法最多往前进一位，而乘法就不一样了，他可能会往前进好多位。这个能不能使用就要看数字大小了，还有一些特殊数字，比如0。如果两个数字比较小且相乘的结果不会出现数字溢出，并且乘数中又没有0，那么是可以的，我们看下代码

```
private void swap4(int[] array, int i, int j) {
  if (i != j) {
    array[i] = array[i] * array[j];
    array[j] = array[i] / array[j];
    array[i] = array[i] / array[j];
  }
}
```

但一般情况下我们不要写出这种代码，因为这样很可能就会出现数字溢出导致结果错误。如果是除法呢，那么这种就更不用再讨论了。

5，逻辑运算符实现

那么除了以上还有没有其他可能呢，比如使用&(与)和|(或)运算符能不能实现，当然也是可以的，我们看下代码

```
private void swap5(int[] array, int i, int j) {
  int temp = (array[i] & array[j]) ^ (array[i] | array[j]);
  array[i] = temp ^ array[i];
  array[j] = temp ^ array[j];
}
```

这种实现很鸡肋，因为第2行其实就是int temp =array[i]^array[j] ，我故意写绕了，只是看大家能不能看的懂，虽然也能实现两个数字的交换，也不用考虑数字溢出问题，但对算法不好的同学来

说也不是很好理解，原理就不在说了，我们可以随便找几个数据测试一下

```
    int[] array = {15, 8, 7, -9, Integer.MAX_VALUE - 3, Integer.MAX_VALUE - 5};
    System.out.println("数据交换之前的值");
    Util.printIntArrays(array);
    System.out.println();
    new Swap().swap5(array, 0, 1);
    new Swap().swap5(array, 2, 3);
    new Swap().swap5(array, 4, 5);
    System.out.println("两两交换之后的值");
    Util.printIntArrays(array);
```

运行结果如下

```
数据交换之前的值
15  8   7   -9  2147483644  2147483642
两两交换之后的值
8   15  -9  7   2147483642  2147483644
```

每两个两个的交换，最终也是实现了数值的交换。

6，异或运算符实现

看了上面的代码我们是不是有点启发，因为异或运算其实还是很强大的，比如0^a=a，a^a=0，同时他还满足交换律，比如a^b^c=a^c^b，^(异或运算)不像&(与)和|(或)那么傻，^(异或运算)有记忆功能，他和+，-，*，÷这些符号一样，如果知道结果和其中的一个值就可以确定另外一个值了，而&(与)和|(或)就不能完全确定了。比如a&1=1或者a&1=0，我们可以确定a的值，但如果a&0=0，我们就无法确定a究竟是0还是1了。那我们这里就仿照+运算符来写一个，代码很简单

```
  private void swap6(int[] array, int i, int j) {
    if (i != j) {
      array[i] ^= array[j];
      array[j] ^= array[i];
      array[i] ^= array[j];
    }
  }
```

所以交换两个数字的值并不是必须要有一个临时变量，写法也不仅仅局限于一种，只要我们多思考，还是有很多种写法的。

喜欢此内容的人还喜欢

483，完全二叉树的节点个数

数据结构和算法





348，数据结构-1,数组

原创 山大王wld 数据结构和算法 4月20日

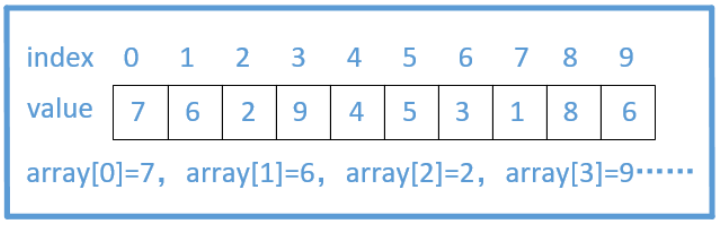
收录于话题
#常见数据结构
7个 >

基础知识

数组是具有相同类型的数据的集合，也就是说数组的所有元素的类型都是相同的，在所有的数据结构中，数组算是最常见也是最简单的一种数据结构，我们最常见的也就是一维数组，当然还有二维，三维……，数组需要先声明才能使用，数组的大小一旦确定就不可以在变了。比如我们声明一个长度为10的数组

```
int[] array = new int[10];
```

数组的下标是从0开始的，比如上面数组的第一个元素是array[0]，最后一个元素是array[9]。



我们还可以在声明的时候直接对他进行初始化，比如

```
int[] array = new int[]{1, 2, 3};
```

上面我们声明了一个长度为3的数组。

源码分析

操作数组的类我们常见的估计也就是ArrayList了，他对数组的操作非常简单，所有的数据都会存放到这个数组中

```
transient Object[] elementData;
```

我们来看一下他常见的几个方法，首先是get方法

```
public E get(int index) {
   if (index >= size)
      throw new IndexOutOfBoundsException(outOfBoundsMsg(index));

   return (E) elementData[index];
}
```

首先判断是否越界，如果越界直接抛异常，否则就根据他的下标从数组中直接返回，在看一下他的set方法

```
public E set(int index, E element) {
   if (index >= size)
      throw new IndexOutOfBoundsException(outOfBoundsMsg(index));

   E oldValue = (E) elementData[index];
   elementData[index] = element;
   return oldValue;
}
```

和get方法一样，也是先判断是否越界，然后再操作，代码比较简单，我们再来看一个add方法

```
public void add(int index, E element) {
```

```
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));

    ensureCapacityInternal(size + 1);  // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1,
                     size - index);
    elementData[index] = element;
    size++;
}
```

这里也是先判断是否越界，然后再判断是否需要扩容，最后在操作，接着我们来看一下ensureCapacityInternal方法

```
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }

    ensureExplicitCapacity(minCapacity);
}

private void ensureExplicitCapacity(int minCapacity) {
    modCount++;

    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

他的默认初始化大小是10

```
private static final int DEFAULT_CAPACITY = 10;
```

上面代码第13行，如果我们需要的空间大于数组长度的时候，说明数组不够用了，要进行扩容，就会执行下面的grow方法，我们来看一下grow方法的代码

```
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

代码也比较简单，扩容的时候在第4行还会增加一半的大小，比如原来数组大小是10，第一次扩容后会是15。在ArrayList中无论使用add还是remove都会使用这样一个方法

```
    System.arraycopy(elementData, index+1, elementData, index,
                     numMoved);
```

这说明对数组的查找是比较方便的，但对数组的增删就没那么方便了，因为数组是一块连续的内存空间，如果在前面增加和删除，都会导致后面元素位置的变动。

ArraList是线程不安全，如果使用线程安全的可以用Vector，还有一个线程安全的类CopyOnWriteArrayList，他只在add和remove的时候，也就是修改数据的时候会先synchronized，在get的时候没有，我们来看一下代码

```
private E get(Object[] a, int index) {
    return (E) a[index];
}
```

我们再来看一下他的add方法

```
1   public boolean add(E e) {
2     synchronized (lock) {
3       Object[] elements = getArray();
4       int len = elements.length;
5       Object[] newElements = Arrays.copyOf(elements, len + 1);
6       newElements[len] = e;
7       setArray(newElements);
8       return true;
9     }
10  }
```

他不像ArrayList每次扩容的时候，size都会增加一半，他是每次add一个元素的时候size只会加1，同理remove的时候size只会减1。

后话

常见的数据结构种类也不是很多，比如，数组，链表，队列，栈，树，图，等，还有数组和链表结合的，比如HashMap。但每一种又会有很多的分类，比如链表有单向的，双向的，环形的，队列又有一般的队列和双端队列，树又分为二叉树，AVL树，红黑树，B+树，2-3树，哈夫曼树，字典树等等。如果细分下去，还是比较多的，今天讲的数组是最简单的一种数据结构，基本上也没什么可说的，剩下的那些数据结构后续都会一一分析。

352，数据结构-2,链表

原创 山大王wld 数据结构和算法 4月27日

收录于话题
#常见数据结构
7个 >

基础知识

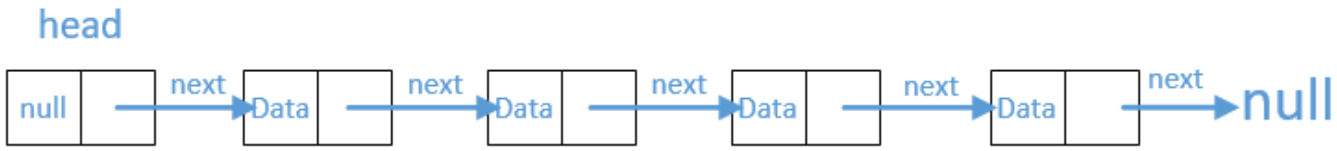
链表是一种物理存储单元上非连续的一种数据结构，看名字我们就知道他是一种链式的结构，就像一群人手牵着手一样。链表有单向的，双向的，还有环形的。

1，单向链表

我们先定义一个最简单的单向链表结点类

```
class Node<E> {
  E data;
  Node<E> next;

  Node(E data, Node<E> next) {
    this.data = data;
    this.next = next;
  }
}
```

这个类非常简单，他只有两个变量，一个是data值，一个是指向下一个结点的指针。我们先来看一下单向的链表是什么样的，



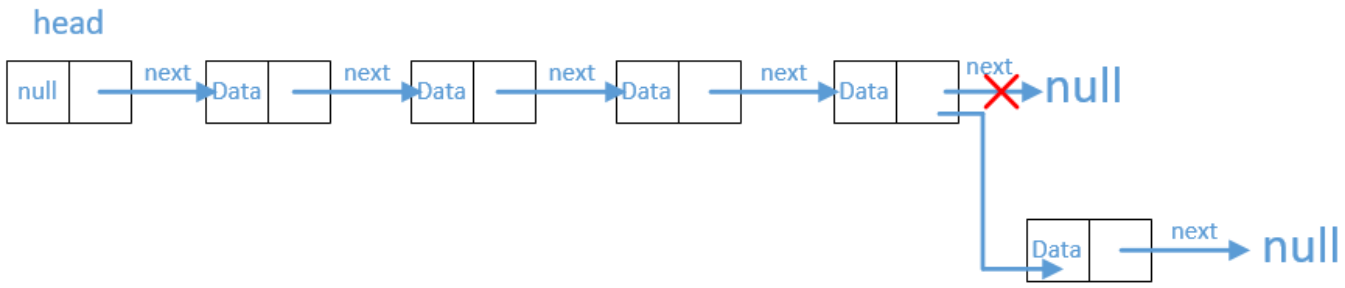
链表头是不存储任何数据的，他只有指向下一个结点的指针，当然如果我们不需要头结点，直接拿第一个结点当做头结点也是可以的，像下面这样



单向链表的增删

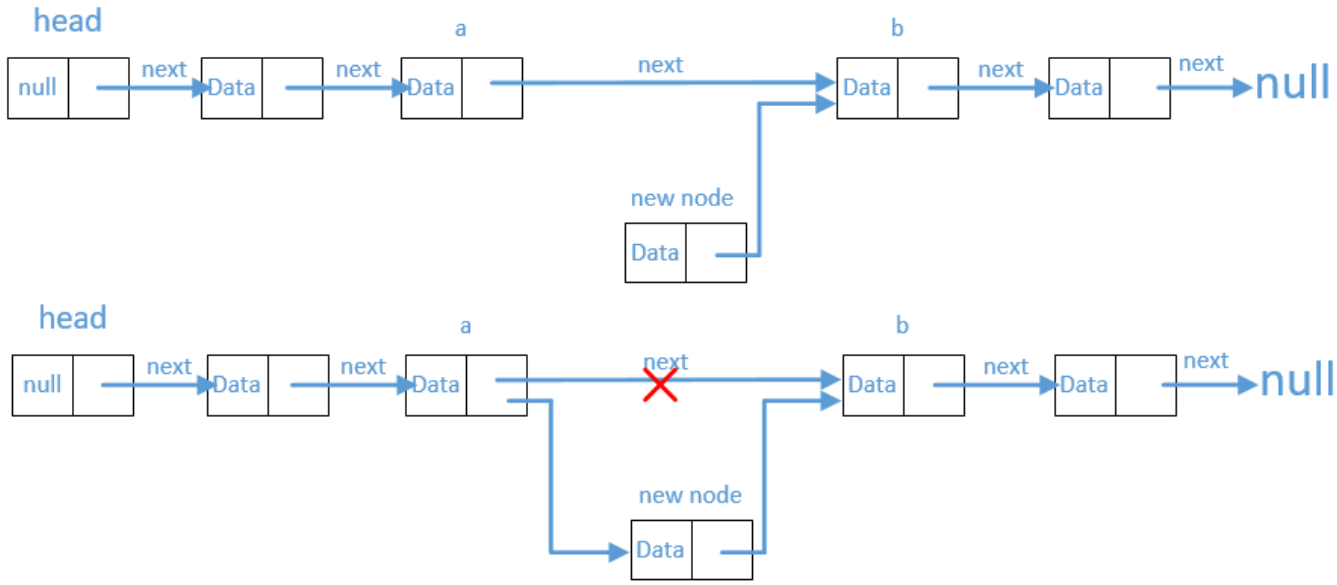
链表不像数组那样，可以通过索引来获取，单向链表查找的时候必须从头开始往后一个个找，而不能从中间找，也不能从后往前找。我们来看一下链表的添加

1，添加到尾结点



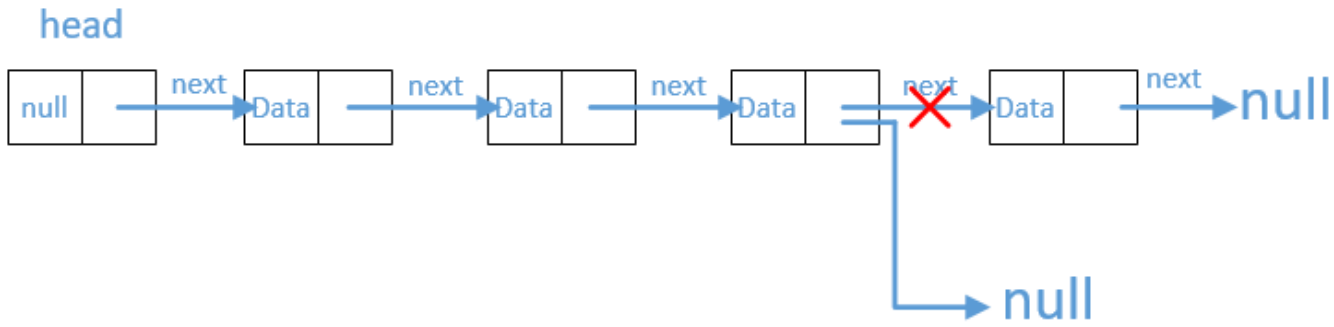
添加到尾结点比较简单，我们只需要找到之前链表的尾结点，然后让他的next指针指向新的结点即可。

2，添加到中间结点



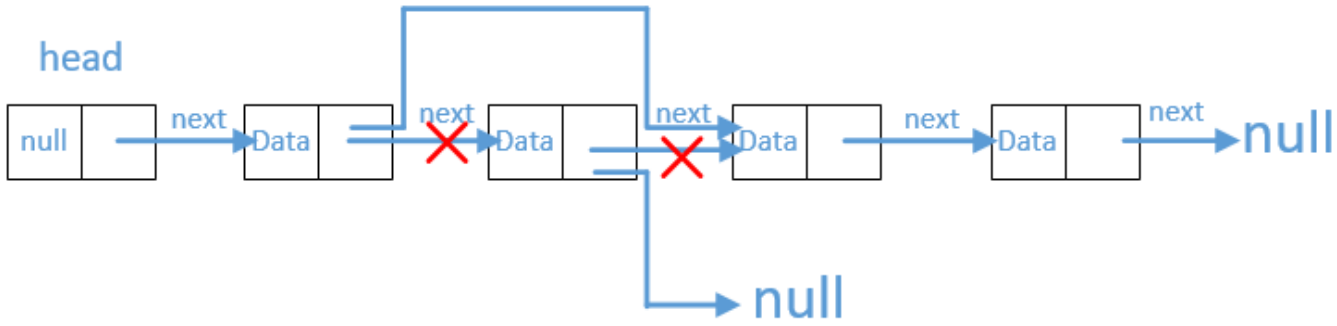
添加到中间结点，分为两步，比如我们要在ab结点之间添加新结点n，第一步让新结点n的指针指向b，然后再让a的指针指向新结点n即可。

3，删除链表的尾结点



只需要让尾结点的上一个结点的指针指向null即可。

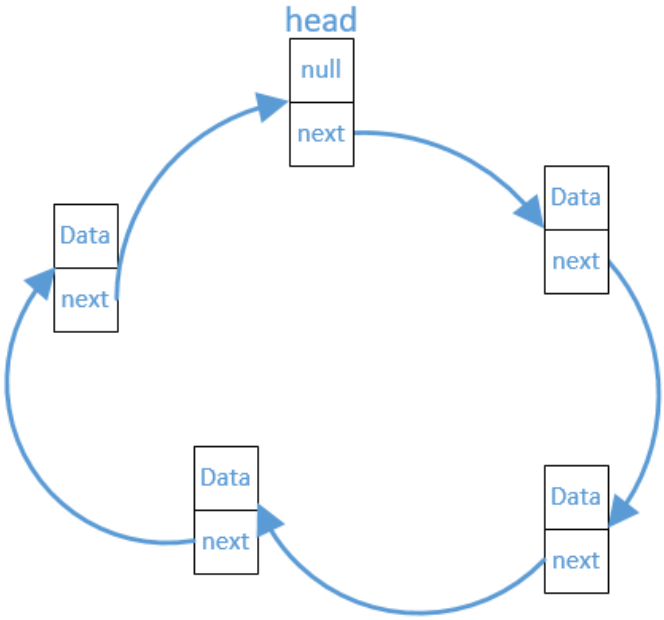
4，删除链表的中间结点



只需要把要删除结点的前一个结点的指针指向要删除结点的下一个结点即可，最好还要把要删除结点的数据清空，并且让他的指针指向null。

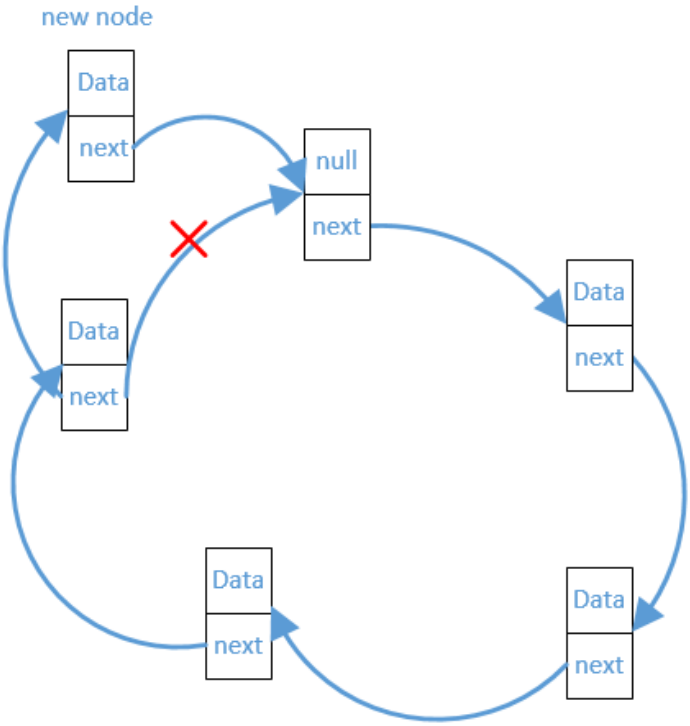
2，单向环形链表

环形链表一般分为两种，一种是单向环形链表，一种是双向环形链表。我们首先来看一下单向的环形链表是什么样的

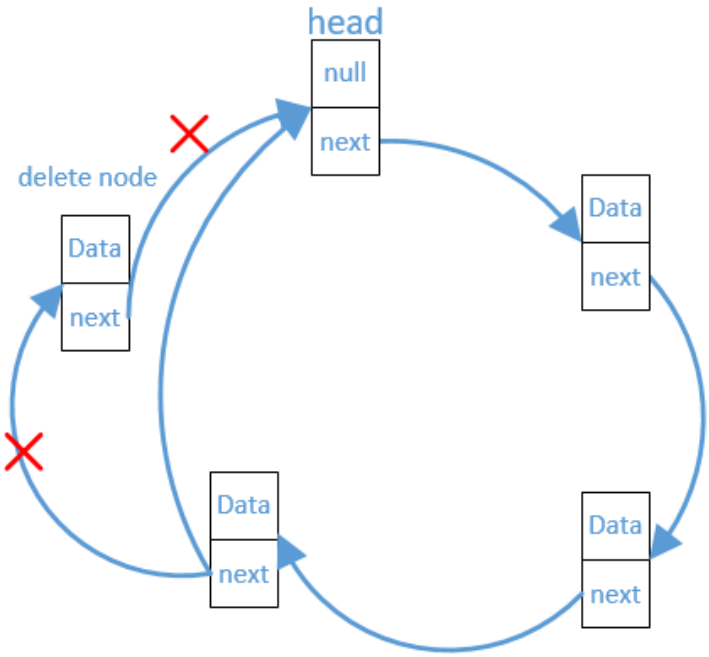


他和单向非环形链表的区就是，单向非环形链表的尾结点的指针是指向null的，而环形的是指向头结点。关于他的增删和单向非环形的差不多，只不过在尾结点的时候会有点区别，我们主要来看下这两种

1，添加到尾结点



2，删除尾结点

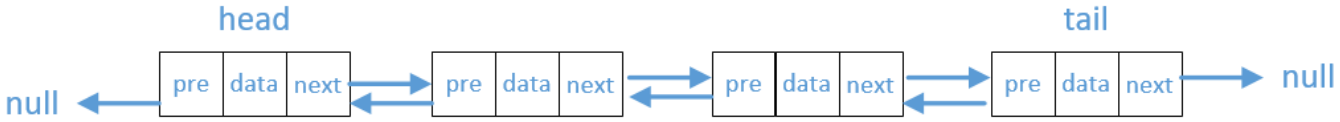


3，双向链表

我们来定义一个双向链表结点的类

```
class Node<E> {
  E data;
  Node<E> next;
  Node<E> prev;

  Node(Node<E> prev, E data, Node<E> next) {
    this.data = data;
    this.next = next;
    this.prev = prev;
  }
}
```

双向链表不光有指向下一个结点的指针，而且还有指向上一个结点的指针，他比单向链表多了一个指向前一个结点的指针，单向链表我们只能从前往后找，而双向链表我们不光可以从前往后找，而且还可以从后往前找。我们来看一下双向链表是什么样的。



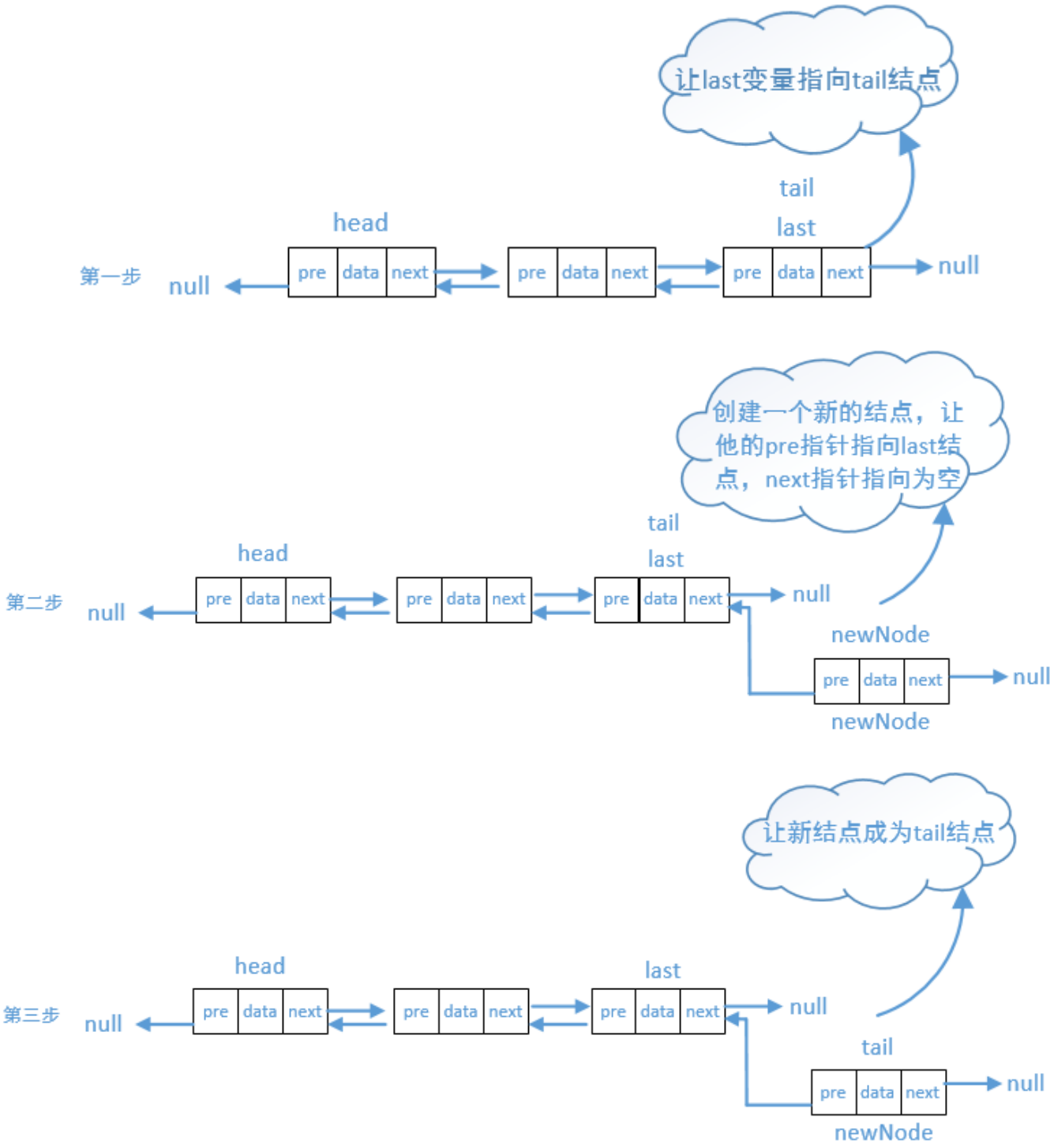
双向链表我们可以从头到尾查找，也可以从尾到头查找，双向链表在代码中最常见的就是LinkedList了（java语言），双向链表的增删和单向链表的增删很类似，只不过双向链表不光要调整他指向的下一个结点，还要调整他指向的上一个结点。这里我们来结合图形和代码的方式来分析一下双向链表的增删。

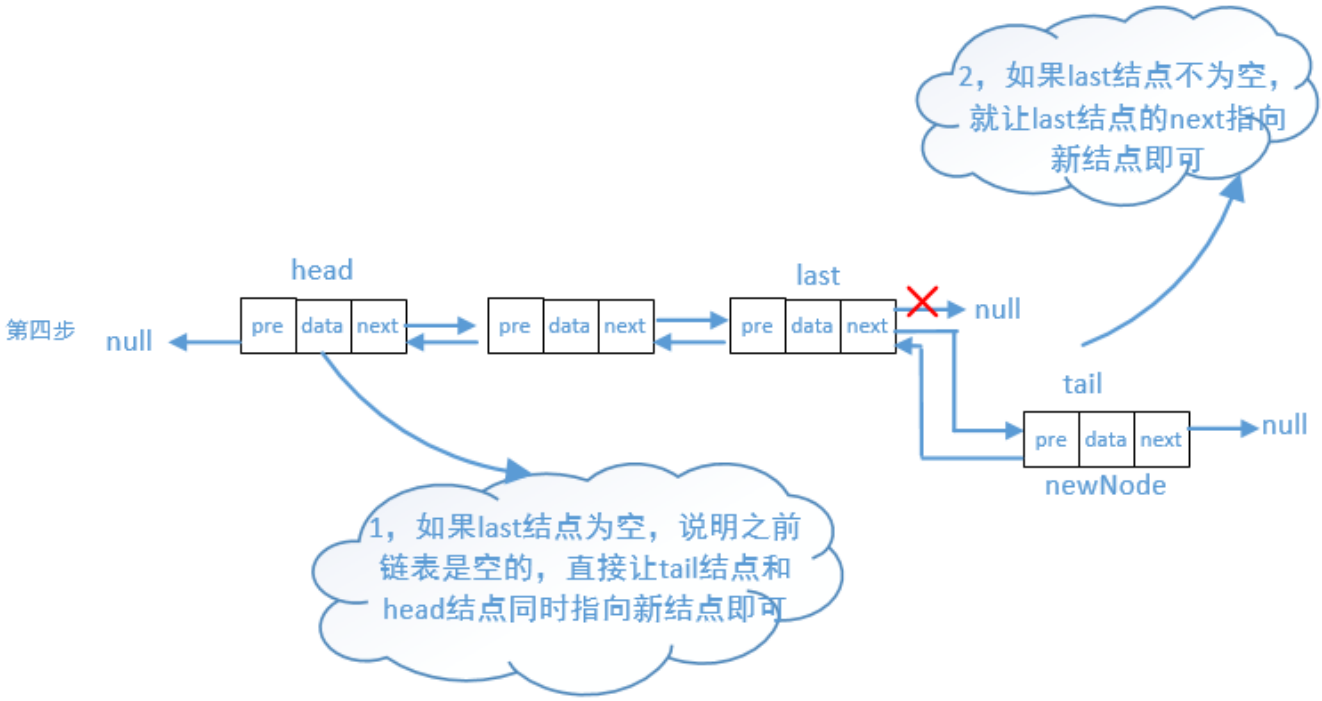
1，添加到尾结点

我们结合着代码来看下

```
void linkLast(Object e) {
  final Node<Object> last = tail;
  final Node<Object> newNode = new Node<>(last, e, null);
  tail = newNode;
  if (last == null)
    head = newNode;
  else
    last.next = newNode;
  size++;
}
```

总共分为4步


让last变量指向tail结点
tail
last
head
第一步
null
pre
data
next
null
创建一个新的结点，让他的pre指针指向last结点，next指针指向为空
tail
last
head
第二步
null
null
newNode
null
newNode
让新结点成为tail结点
head
last
第三步
null
null
tail
null
newNode



2，添加到头结点

```
1   private void linkFirst(Object e) {
2     //用临时变量firt保存head结点
3     final Node<Object> first = head;
4     //创建一个新的结点，让他的pre指针指向null，next指针指向head结点
5     final Node<Object> newNode = new Node<>(null, e, first);
6     //让head指向新的结点
7     head = newNode;
8     //如果之前的first为空，说明之前链表是空的，让head和tial同时指向新结点即可
9     if (first == null)
10      tail = newNode;
11    else//如果之前链表不为空，让之前first结点的pre指向新的结点
12      first.prev = newNode;
13    size++;
14  }
```

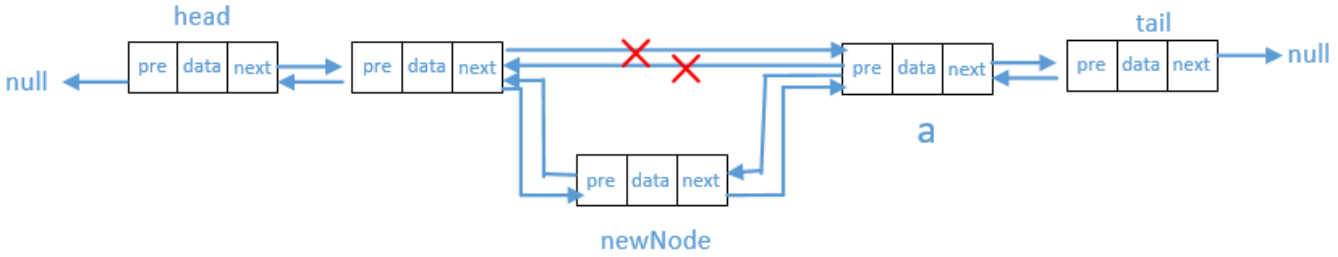
添加到头结点和添加到尾结点很类似，图就不在画了，大家可以看下上面的代码。

3，添加到指定结点之前

比如我们在a结点之前添加一个指定的结点，先来看下代码

```
1   void linkBefore(Object e, Node<Object> a) {
2     final Node<Object> pred = a.prev;
3     final Node<Object> newNode = new Node<>(pred, e, a);
4     a.prev = newNode;
5     if (pred == null)
6       head = newNode;
7     else
8       pred.next = newNode;
9     size++;
10  }
```

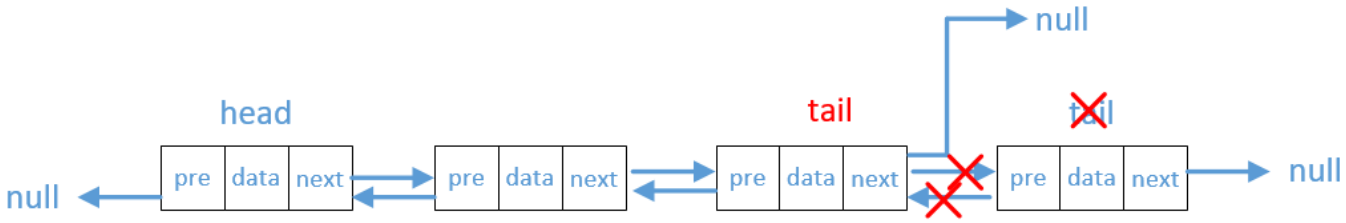
假如我们在a结点之前添加一个结点，图就不在细画，简单看一下即可



4，删除链表的尾结点

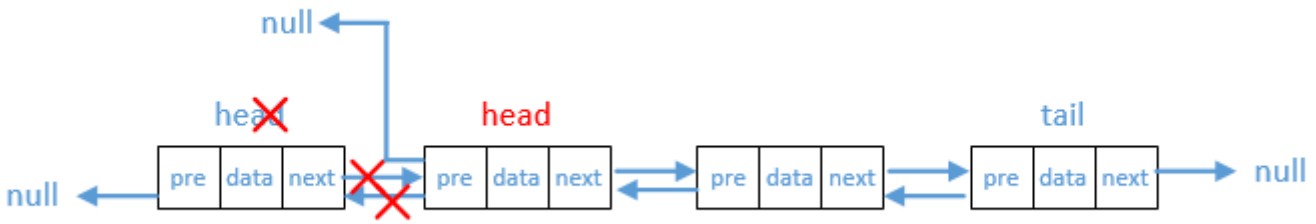
```
1   private Object unlinkLast(Node<Object> last) {
2     final Object element = last.data;
3     final Node<Object> prev = last.prev;
4     last.data = null;
5     last.prev = null;
6     tail = prev;
7     if (prev == null)//如果只有一个结点，把尾结点删除，相当于把链表清空了
8       head = null;
9     else
10      prev.next = null;
11    size--;
12    return element;
13  }
```

如果链表只有一个结点的话，我们把它删除，相当于直接把链表清空了，这种很好理解，就不再画。下面画一个长度大于1的链表，然后删除最后一个结点



5，删除链表的头结点

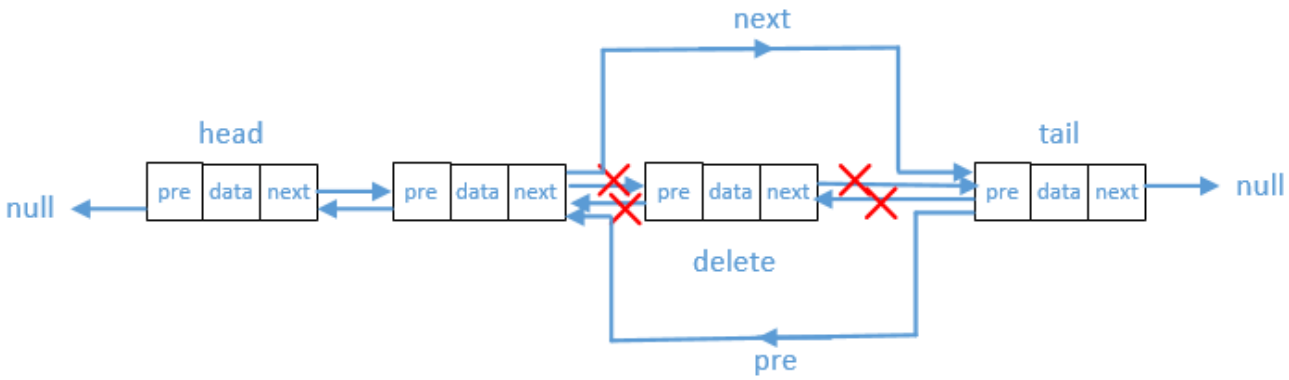
```
1   private Object unlinkFirst(Node<Object> first) {
2     final Object element = first.data;
3     final Node<Object> next = first.next;
4     first.data = null;
5     first.next = null;
6     head = next;
7     if (next == null)
8       tail = null;
9     else
10      next.prev = null;
11    size--;
12    return element;
13  }
```



6，删除链表的中间结点

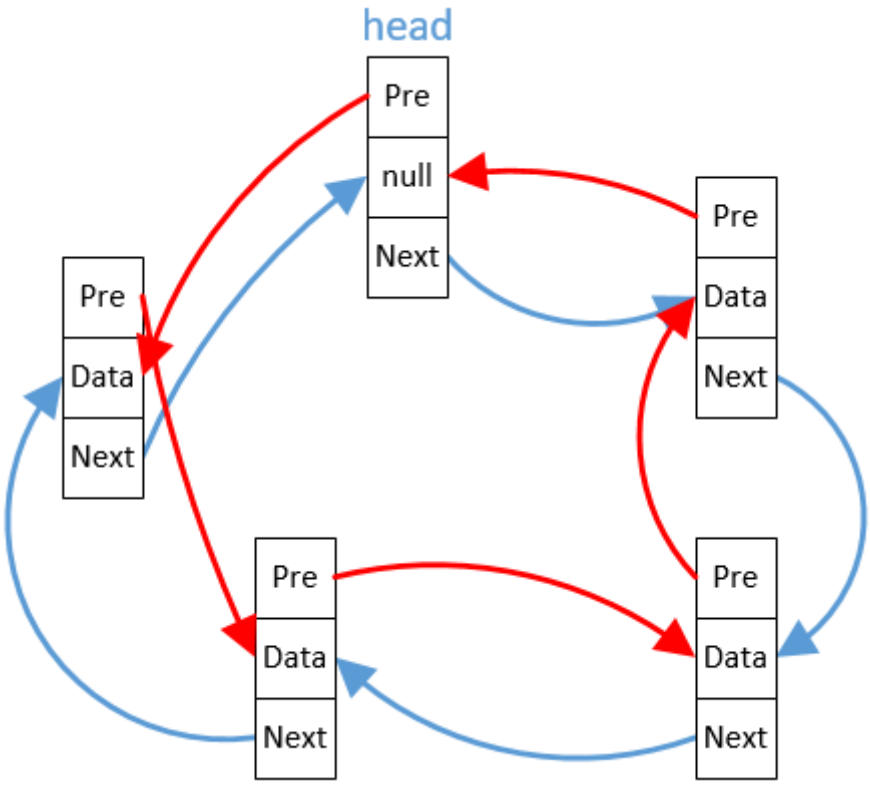
```
1   Object unlink(Node<Object> x) {
2     final Object element = x.data;
3     final Node<Object> next = x.next;//x的前一个结点
4     final Node<Object> prev = x.prev;//x的后一个结点
5
6     if (prev == null) {//前一个结点是空
7       head = next;
8     } else {
9       prev.next = next;
10      x.prev = null;
```

```
    }

    if (next == null) {//后一个结点是空
      tail = prev;
    } else {
      next.prev = prev;
      x.next = null;
    }

    x.data = null;
    size--;
    return element;
  }
```



4，双向环形链表

双向环形链表在代码中最常见的就是LinkedHashMap了，这个一般用于图片缓存的比较多一些，LRUCache这个类里面主要使用的就是LinkedHashMap（java语言中），通过上面的分析，如果对linkedList能理解的话，那么双向环形链表也就不难理解了，其实原理都差不多，这里就不在过多介绍，下面是双向环形链表的图。



359，数据结构-3,队列

原创 山大王wld 数据结构和算法 5月12日

收录于话题
#常见数据结构
7个 >

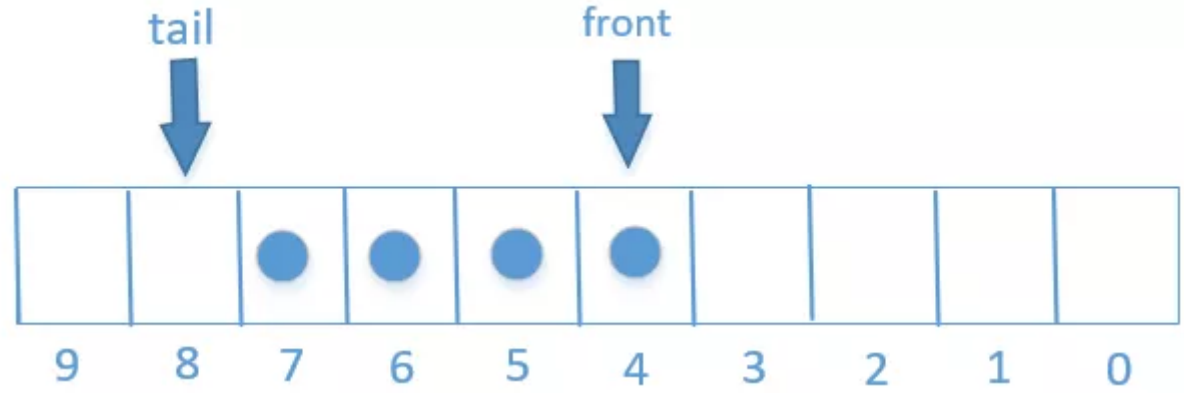
基础知识

队列是一种特殊的线性表，他的特殊性在于我们只能操作他头部和尾部的元素，中间的元素我们操作不了，我们只能在他的头部进行删除，尾部进行添加。就像大家排队到银行取钱一样，先来的肯定要排到前面，后来的只能排在队尾，所有元素都要遵守这个操作，没有VIP会员，所以走后门插队的现象是不可能存在的，他是一种**先进先出**的数据结构。我们来看一下队列的数据结构是什么样的

1，一般队列



他只能从左边进，右边出，队列实现方式一般有两种，一种是基于数组的，还一种是基于链表的，如果基于链表的倒还好说，因为链表的长度是随时都可以变的，这个实现起来比较简单。如果是基于数组的，就会稍微有点不同，因为数组的大小在初始化的时候就已经固定了，我们来看一下基于数组的实现，假如我们初始化一个长度是10的队列



front指向的是队列的头，tail指向的是队列尾的下一个存储空间，最初始的时候front=0，tail=0，每添加一个元素tail就加1，每移除一个元素front就加1，但是这样会有一个问题，如果一个元素不停的加入队列，然后再不停的从队列中移除，会导致tail和front越来越大，最后会导致队列无法再加入数据了，但实际上队列前面全部都是空的，这导致空间的极大浪费。我们自己来写一个简单的队列看一下

```
1   public class MyQueue<E> {
2
```

```
    private final Object[] data;
    private final int maxSize;
    private int size;
    private int front = 0;
    private int tail = 0;

    public MyQueue(int maxSize) {
        if (maxSize <= 0) {
            throw new IllegalArgumentException("队列容量必须大于0：" + maxSize);
        }
        this.maxSize = maxSize;
        data = new Object[this.maxSize];
    }

    public void add(E e) {
        if (isFull()) {//这地方可以扩容也可以抛异常，为了方便这里我们就不在扩容了。
            throw new IllegalStateException("队列已经满了，无法再加入……");
        }
        data[tail++] = e;
        size++;
    }

    public E remove() {
        if (isEmpty()) {
            throw new IllegalStateException("队列是空的，无法移除……");
        }
        E t = (E) data[front];
        data[front++] = null;
        size--;
        return t;
    }

    //队列头和队列尾指向同一空间的时候，并且没到队尾，表示队列是空的
    public boolean isEmpty() {
        return front == tail && !isFull();
    }

    public boolean isFull() {//最后一个位置是不存储数据的
        return tail == maxSize - 1;
    }

    public int getSize() {
        return size;
    }
}
```

代码非常简单，当然队列的实现不一定是这一种方式，比如我们可以让tail指向队尾的元素，或者以链表的形式来实现都是可以的，不同的实现方式会导致上面的方法有所不同。我们来测试一下

```
public static void main(String[] args) {
    MyQueue myQueue = new MyQueue(10);
    System.out.println("isEmpty()=" + myQueue.isEmpty());
    System.out.println("isFull()=" + myQueue.isFull());
    System.out.println("getSize()=" + myQueue.getSize());
    for (int i = 0; i < 9; i++) {
        myQueue.add(i * 100);
        myQueue.remove();
    }
    System.out.println("----------------------------");
    System.out.println("isEmpty()=" + myQueue.isEmpty());
    System.out.println("isFull()=" + myQueue.isFull());
    System.out.println("getSize()=" + myQueue.getSize());
}
```

看一下打印的结果

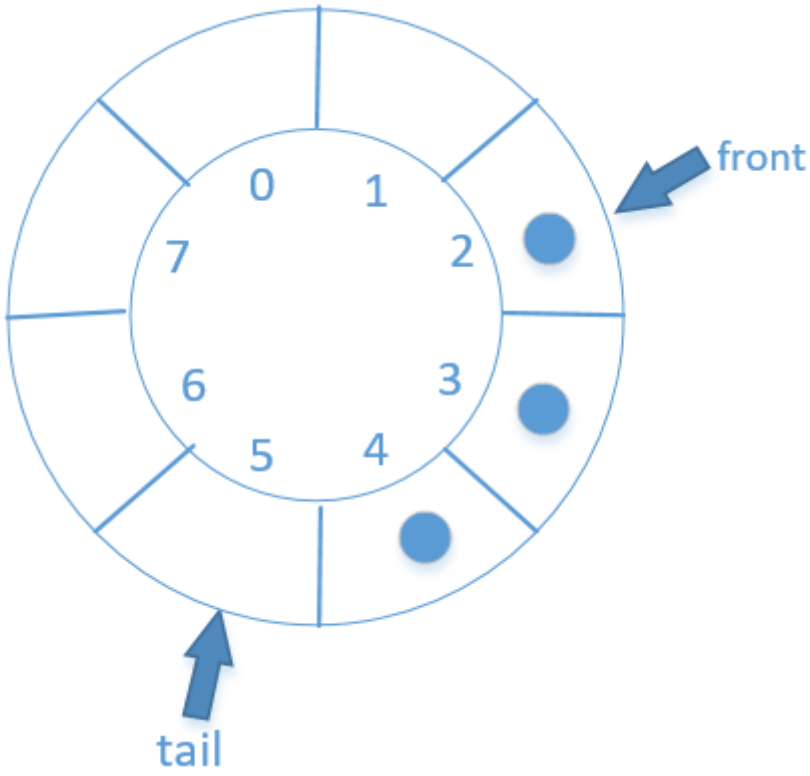
```
isEmpty()=true
isFull()=false
getSize()=0
```

```
4       ----------------------------
5       isEmpty()=false
6       isFull()=true
7       getSize()=0
```

我们添加了9次，然后又移除了9次，结果队列竟然满了，如果我们再添加一次的话肯定会抛异常，但实际上队列的size是0，还是空的，也就是说数组的每个位置只能使用一次，这样就造成了极大的浪费。那么前面使用过的空间还能不能再次利用了呢，实际上是可以的，我们可以把队列看成是环形的，当tail到达数组末尾的时候，如果数组的前面有空位子，我们可以让tail从头开始，这个时候一个新的队列就产生了，那就是双端队列。

2，双端队列

双端队列也是有两个指针，front指向队首，tail指向队尾的下一个存储单元，并且双端队列的队首和队尾都可以添加和删除元素，我们来看一下图



这样空间就可以循环利用了，不会造成浪费，我们来看下代码

```
1   public class MyQueue<E> {
2
3     //存储的元素
4     private Object[] data;
5
6     //指向队列头，这个头并不是数组的第0个元素，如果这样
7     // front就没有意义了，这个从下面的addFirst(E e)方
8     // 法也可以看出，如果当front等于0的时候，在添加到
9     // first，那么会添加到数组的末尾，并且front也指向
10    // 数组的末尾
11    private int front;
12
13    //指向队列尾的下一个空间，可以这样理解，front指向
14    // 的是第一个元素，tail指向的是最后一个元素的下一
15    // 个，指的是空的。
16    private int tail;
17
18
19    public MyQueue(int numElements) {
20      data = new Object[numElements];
```

```
21      }
22
23      //空间扩容，我们这里选择扩大一倍，当然也可以选择其
24      // 他值，仅仅当满的时候才能触发扩容， 这时候front
25      // 和tail都会指向同一个元素
26      private void doubleCapacity() {
27        int p = front;
28        int n = data.length;//数组的长度
29        //关键是r不好理解，举个例子，在数组中，front
30        // 不一定是之前0位置的，他可以指向其他位置，
31        // 比如原来空间大小为16，front为13，也就是第
32        // 14个元素（数组是从0开始的），那么r就是16-13=3，
33        // 也就是从front往后还有多少元素，待会copy的时候
34        // 也是先从最后的r个元素开始
35        int r = n - p;
36        Object[] a = new Object[n << 1];//扩大一倍
37        System.arraycopy(data, p, a, 0, r);//先copy后面的r个
38        System.arraycopy(data, 0, a, r, p);//再copy前面的p个
39        data = a;
40        //重新调整front和tail的值
41        front = 0;
42        tail = n;
43      }
44
45      public void addFirst(E e) {
46        //添加到front的前面，所以front-1
47        front = (front - 1 + data.length) % data.length;
48        data[front] = e;
49        if (front == tail)//判断是否满了
50          doubleCapacity();
51      }
52
53      public void addLast(E e) {
54        //添加到最后一个，这个方法和addFirst有很明显的不同，
55        // addFirst是添加的时候就要计算front的位置，而addLast
56        // 方法是存值之后在计算tail的，/因为tail位置是没有
57        // 存值的，他表示的末端元素的下一个，是空，所以存值之后
58        //要计算tail的值
59        data[tail] = e;
60        tail = (tail + 1) % data.length;
61        if (tail == front)//判断是否满
62          doubleCapacity();
63      }
64
65      public E removeFirst() {//删除第一个
66        if (isEmpty())
67          throw new IllegalStateException("队列是空的，无法移除……");
68        E result = (E) data[front];
69        data[front] = null;
70        // 删除第一个，这里的所有第一我们都认为是front所指的，
71        // 不是数组的0位置，然后在计算front的值
72        front = (front + 1) % data.length;
73        return result;
74      }
75
76      public E removeLast() {//删除最后一个
77        if (isEmpty())
78          throw new IllegalStateException("队列是空的，无法移除……");
79        tail = (tail - 1 + data.length) % data.length;
80        E result = (E) data[tail];
81        data[tail] = null;
82        return result;
83      }
84
85      public E peekFirst() {
86        if (isEmpty())
87          throw new IllegalStateException("队列是空的，无法获取……");
88        return (E) data[front];
89      }
```

```
90
91      public E peekLast() {
92        if (isEmpty())
93          throw new IllegalStateException("队列是空的，无法获取……");
94        return (E) data[(tail - 1 + data.length) % data.length];
95      }
96
97      public int size() {//元素的size
98        return (tail - front + data.length) % data.length;
99      }
100
101     //是否为空，在上面添加元素的时候也可能front==tail，当添加
102     // 元素之后front==tail的时候就认为是满了，然后扩容，重新
103     // 调整front和tail的值,所以扩容之后front就不可能等于tail。
104     //如果没有触发上面添加元素的时候front等于tail我们就认为他是空的。
105     public boolean isEmpty() {
106       return front == tail;
107     }
108   }
```

代码中都有详细的注释，就不在过多介绍。

363，数据结构-4,栈

原创 山大王wld 数据结构和算法 5月18日

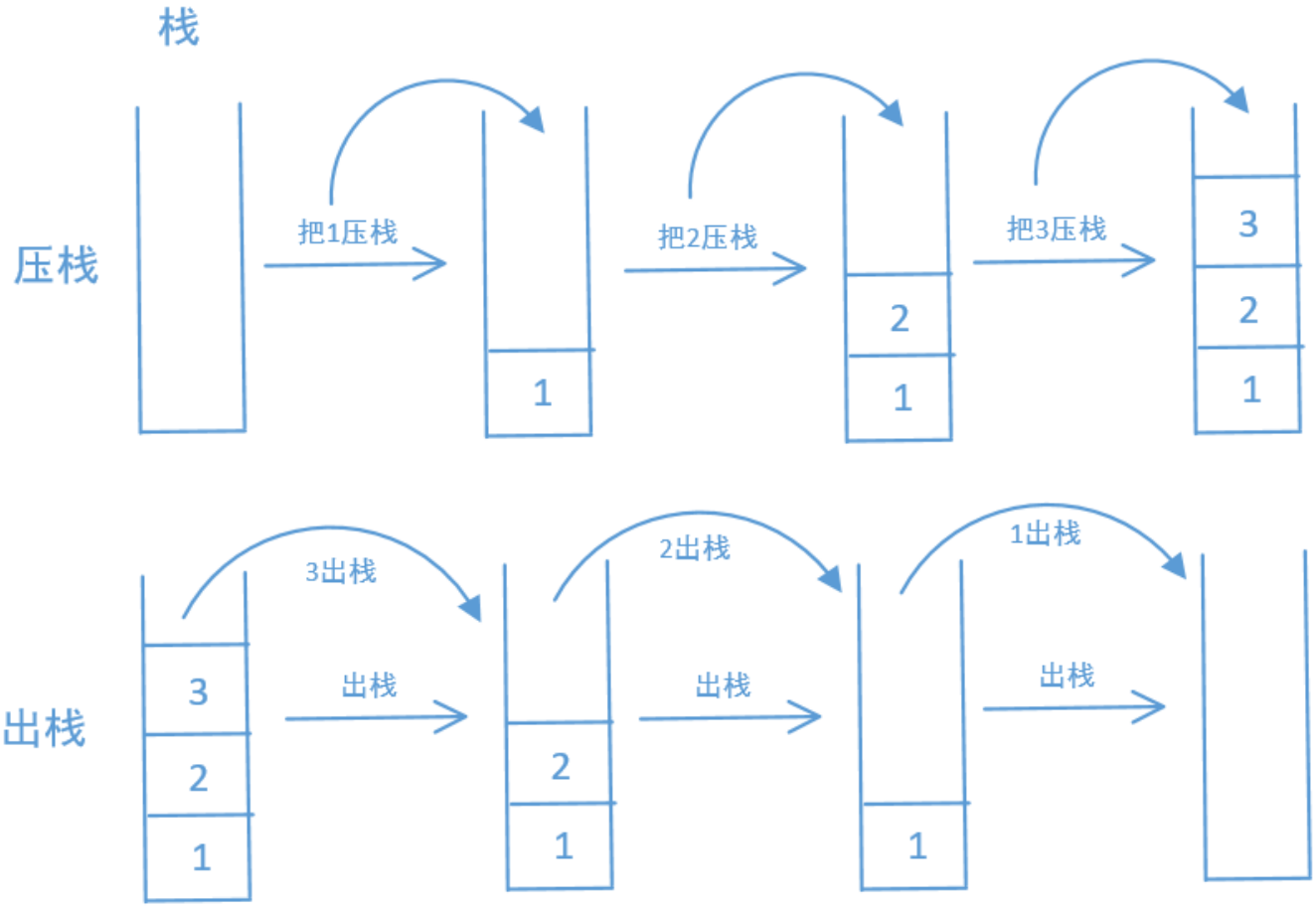
收录于话题
#常见数据结构
7个 >

基础知识

栈也是一种特殊的线性表，他只能对栈顶进行添加和删除元素。栈有入栈和出栈两种操作，他就好像我们把书一本本的摞起来，最先放的书肯定是摞在下边，最后放的书肯定是摞在了最上面，摞的时候不允许从中间放进去，拿书的时候也是先从最上面开始拿，不允许从下边或中间抽出来。

栈的原理图

栈的实现可以使用数组也可以使用链表，我们这里分析的主要是使用数组的形式。



代码实现

栈的实现其实非常简单，常见的就两种操作，一种是压栈一种是出栈，我们来看下代码

```
public class Stack<E> {
  private Object[] data;
  private int size;

  public Stack(int capacity) {
    if (capacity <= 0)
```

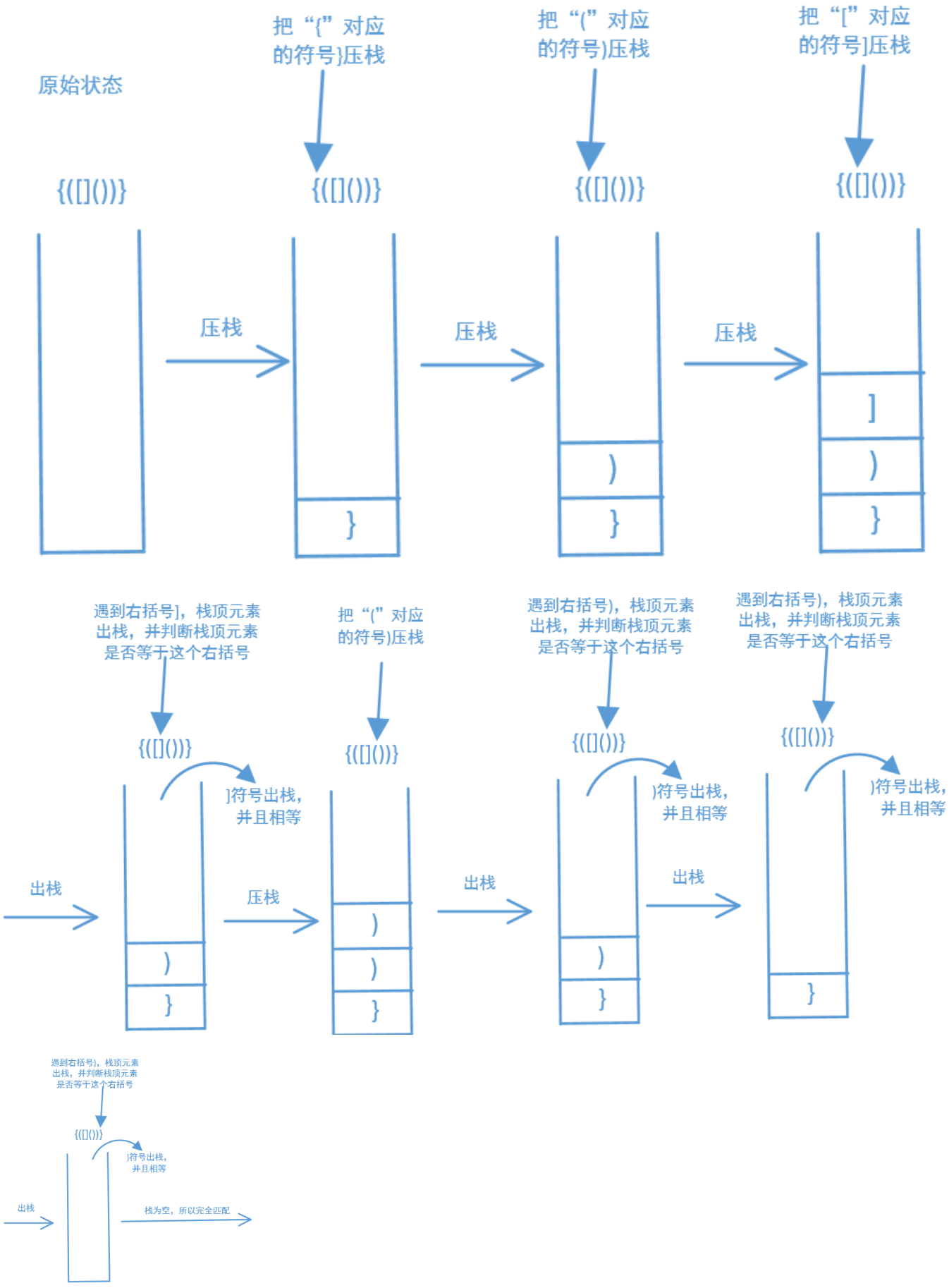
```
7           throw new IllegalArgumentException("栈的大小必须大于0");
8         data = new Object[capacity];
9       }
10
11      public void push(E item) {
12        if (isFull())
13          throw new IllegalArgumentException("栈已经满了");
14        data[size++] = item;
15      }
16
17      public E pop() {
18        if (isEmpty())
19          throw new IllegalArgumentException("栈是空的");
20        E temp = (E) data[--size];
21        data[size] = null;
22        return temp;
23      }
24
25      public E peek() {
26        if (isEmpty())
27          throw new IllegalArgumentException("栈是空的");
28        return (E) data[size - 1];
29      }
30
31      public boolean isEmpty() {
32        return size() == 0;
33      }
34
35      public boolean isFull() {
36        return size() == data.length;
37      }
38
39      public int size() {
40        return size;
41      }
42    }
```

这里为了方便，栈的空间大小在初始化的时候就就已经固定了，并且栈满的时候没有扩容，栈是一个非常有用的数据结构，尤其在算法中用到的还是比较多的。**栈是一种先进后出的数据结构**，他和队列正好相反，**队列是一种先进先出的数据结构。**

例子

我们来看个非常简单的例子，验证括号的有效性，括号只包含"(", ")", "[", "]", "{", "}"这6个字符，比如()，{()}，[](){}都是有效的，而{]，{(]}都是无效的。

我们来分析一下这道题，当我们遍历到括号的左半边的时候，我们把括号的右半边压栈，当我们遍历到括号右半边的时候，我们就把栈顶的元素弹出，然后在和我们遍历的符号比较看是否一样，我们以字符串"{([]())}"为例来画图分析一下，



搞懂了上面的图，写出代码就容易多了，我们来看下代码怎么实现

```
1   public boolean isValid(String s) {
2     Stack<Character> stack = new Stack<>(s.length());
3     for (char c : s.toCharArray()) {
4       if (c == '(')
5         stack.push(')');
6       else if (c == '{')
```

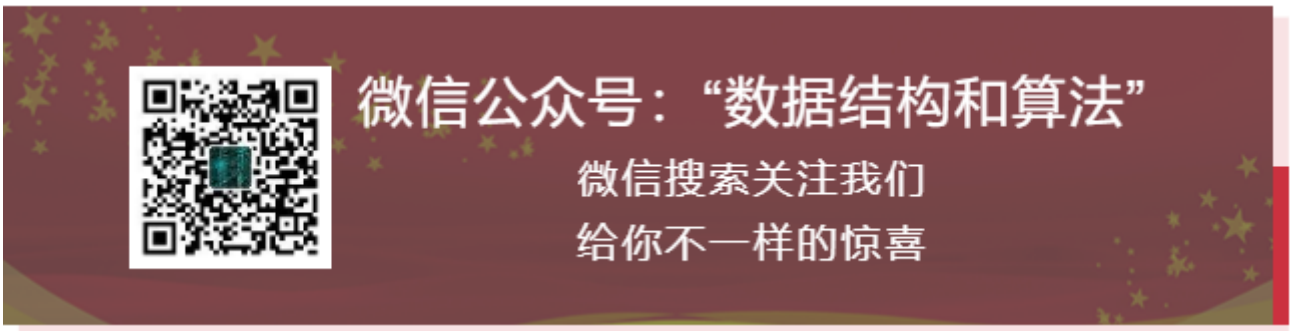
```
7           stack.push('}');
8         else if (c == '[')
9           stack.push(']');
10        else if (stack.isEmpty() || stack.pop() != c)
11          return false;
12      }
13      return stack.isEmpty();
14    }
```

368，数据结构-5,散列表

原创 山大王wld 数据结构和算法 5月25日

收录于话题
#常见数据结构

7个 >

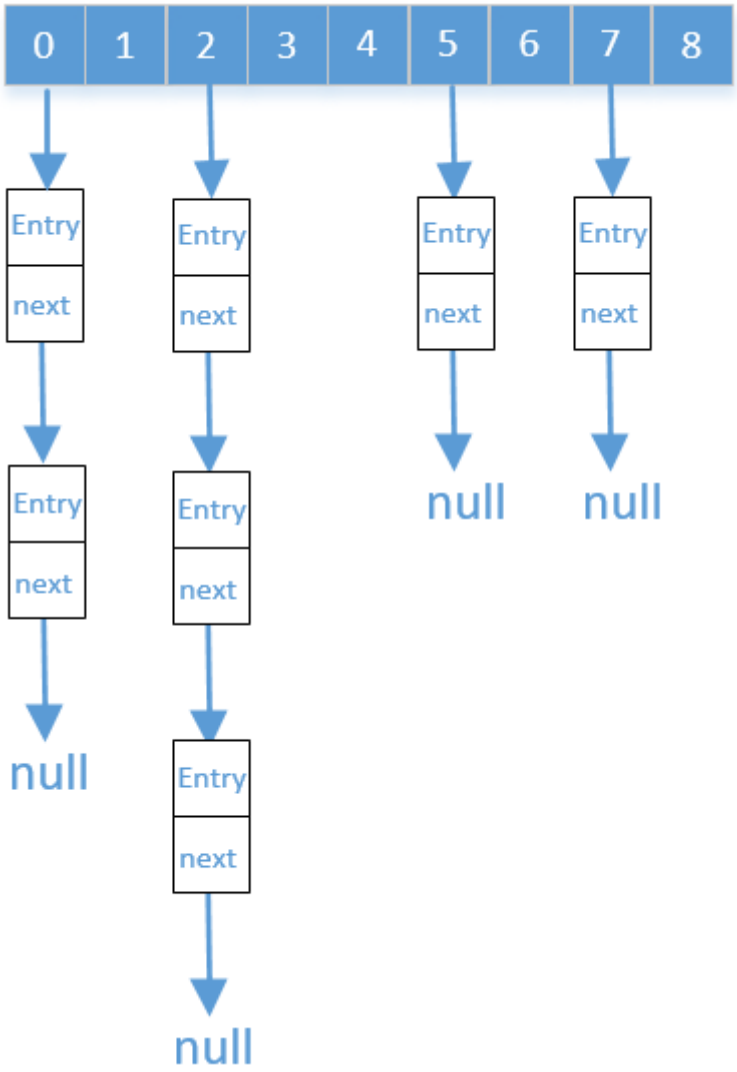


基础知识

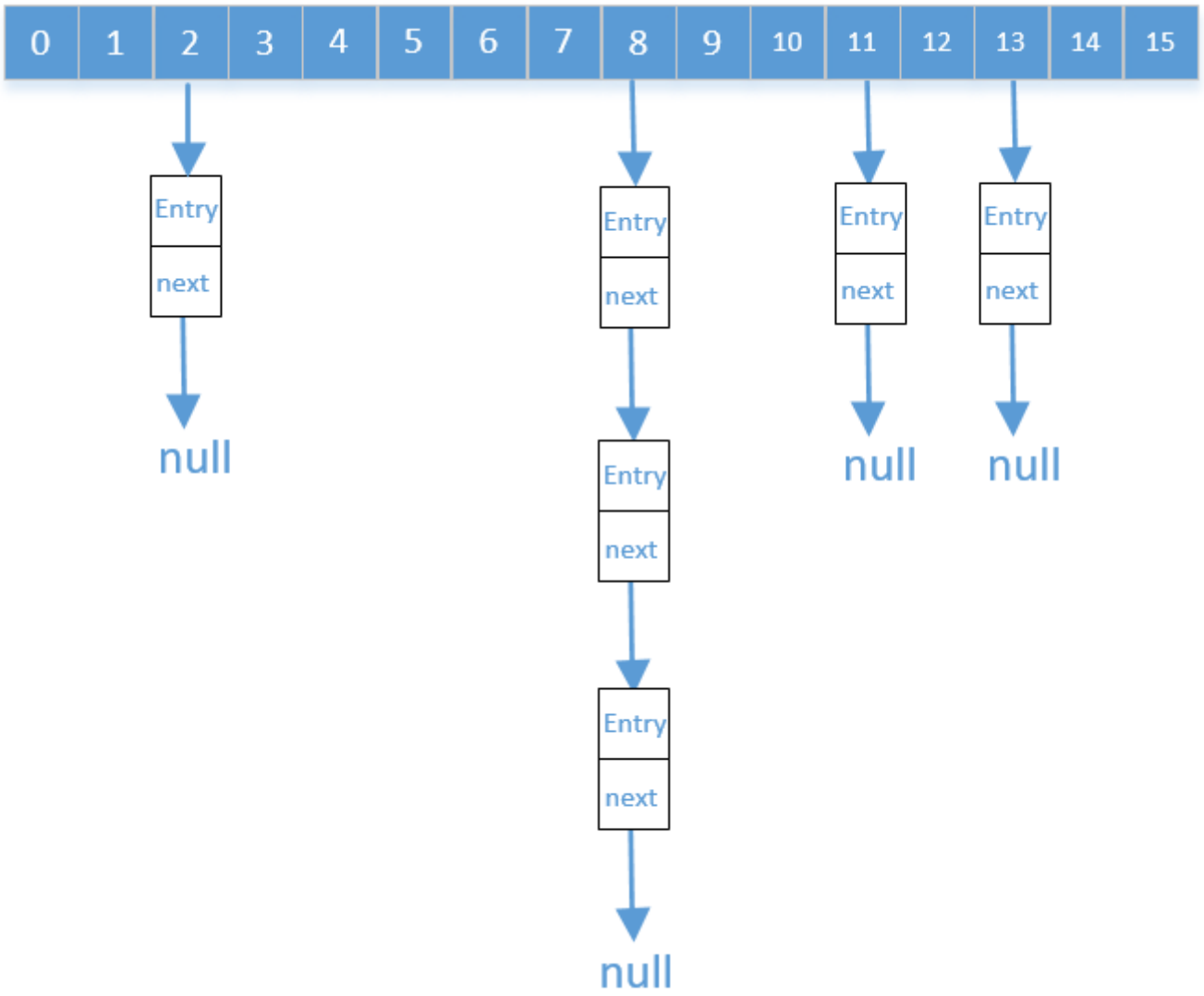
散列表也叫哈希表，是根据键值对（key，value）进行访问的一种数据结构。他是把一对（key，value）通过key的哈希值来映射到数组中的，也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

01 HashMap

散列表中最常见的应该就是HashMap了，HashMap的实现原理非常简单，他其实就是数组加链表的一种数据结构。如果映射在数组中出现了冲突，他会以链表的形式存在。我们来看一下他的数据结构是什么样的。



上面的图有两处非常明显的错误，不知道大家有没有发现，如果对HashMap源码比较熟悉的估计一眼就能看的出来。首先是数组的长度必须是2的n次幂，这里长度是9，明显有错，然后是entry的个数不能大于数组长度的75%，如果大于就会触发扩容机制进行扩容，这里明显是大于75%。我为什么要画这个错误的图呢，因为在网上确实看到过不少这样不严谨的图，希望大家能够看清楚。那么正确的图应该是这样的。



数组的长度即是2的n次幂，而他的size又不大于数组长度的75%。

HashMap的实现原理是先要找到要存放数组的下标，如果是空的就存进去，如果不是空的就判断key值是否一样，如果一样就替换，如果不一样就以链表的形式存在。

在java中1.7及以前的版本如果以链表的形式存在，在插入的时候采用的是头插法。
在1.8是尾插法。并且在java1.8中如果链表的长度大于8的时候会转为红黑树。

在HashMap中，数组的大小是2^n，无论你初始化的时候传的值是多少，他都会初始化为2^n，并且这个2^n是大于等于你初始化值的最小值，比如初始化的时候传的值是17，他会计算得到32。关于怎么计算的，我们有3种方式，第一种就是通过while循环，我们来看下代码

```
1   public static int tableSizeFor(int initialCapacity) {
2     int capacity = 1;
3     while (capacity < initialCapacity)
4       capacity <<= 1;
5     return capacity;
6   }
```

这种解法是最简单的，一眼就能看懂，还有两种解法我们也可以看下

```
1   public static int tableSizeFor(int i) {
2     i--;
```

```
    i |= i >>> 1;
    i |= i >>> 2;
    i |= i >>> 4;
    i |= i >>> 8;
    i |= i >>> 16;
    return i + 1;
  }
```

原理比较简单，就是把最左边的1往右全部铺开，最后在加上1就是我们要求的结果。这里第二行减1的目的是防止i等于2^n的时候结果会放大。比如当i=32的时候如果我们在第2行不减1，会导致结果为64。我们再来看另一种写法

```
1   public static int tableSizeFor(int i) {
2     if ((i & (i - 1)) == 0)
3       return i;
4     i |= (i >> 1);
5     i |= (i >> 2);
6     i |= (i >> 4);
7     i |= (i >> 8);
8     i |= (i >> 16);
9     return (i - (i >>> 1)) << 1;
10  }
```

在第2-3行实现判断是不是2^n，如果是就直接返回，第4-8行也是把i最左边的1往右全部铺开，第9行i-(i>>>1)表示的是把i最左边的1保留，其他的全部置为0，通俗一点也就是他返回的是小于i的最大的2^n，然后再往左移一位就是我们要求的结果。我们就以i等于17为例用最后一个方法来画个图分析一下。

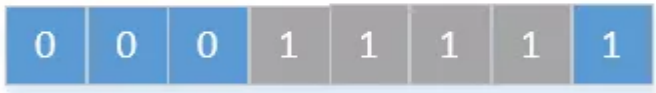
原始状态17



第一步执行
i |= (i >> 1)



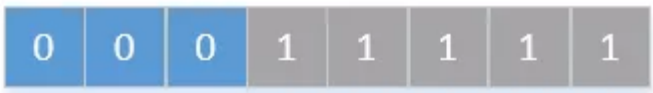
第二步执行
i |= (i >> 2)



第三步执行
i |= (i >> 4)



第四步执行
i |= (i >> 8)



第五步执行
i |= (i >> 16)



执行
i - (i >>> 1)



执行
(i - (i >>> 1)) << 1



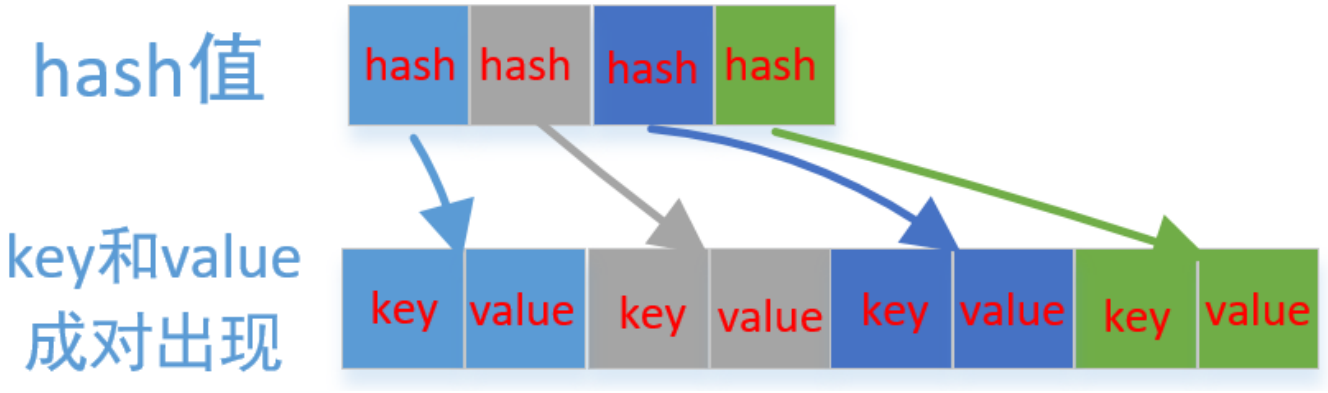
02 ArrayMap

除了使用数组和链表以外，我们能不能只使用一种数据结构呢，比如数组，当然也是可以的。大家可能会怀疑，如果只使用一种数据结构的话，映射出现了冲突怎么办，其实也很好解决。ArrayMap的实现原理是使用两个数组，一个存放hash值，一个存放key和value，其中存放key和value的数组长度是存放hash值数组长度的二倍，其中存放hash值的数组必须是排序的。如果hash值出现了冲突，说明hash值最终的计算是一样的，那么在hash数组中肯定是挨着的，所以查找的时候如果hash值有重复的就会把重复的也查找一遍。我们来看ArrayMap中的一段代码

```
int indexOf(Object key, int hash) {
    final int N = mSize;

    // Important fast case: if nothing is in here, nothing to look for.
    if (N == 0) {
        return ~0;
    }

    int index = binarySearchHashes(mHashes, N, hash);

    // If the hash code wasn't found, then we have no entry for this key.
    if (index < 0) {
        return index;
    }

    // If the key at the returned index matches, that's what we want.
    if (key.equals(mArray[index<<1])) {
        return index;
    }

    // Search for a matching key after the index.
    int end;
    for (end = index + 1; end < N && mHashes[end] == hash; end++) {
        if (key.equals(mArray[end << 1])) return end;
    }

```

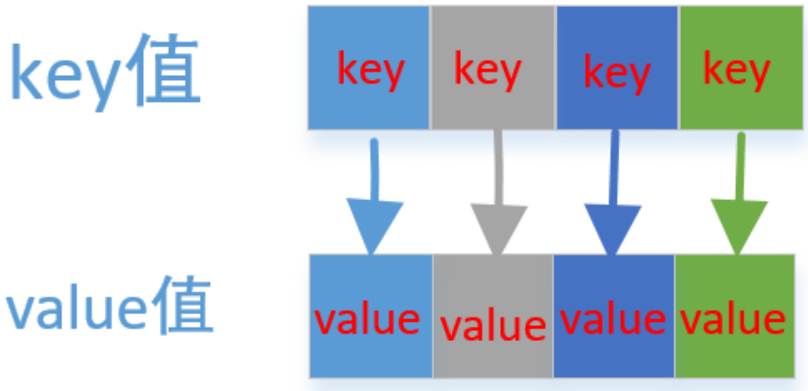
```
27      // Search for a matching key before the index.
28      for (int i = index - 1; i >= 0 && mHashes[i] == hash; i--) {
29          if (key.equals(mArray[i << 1])) return i;
30      }
31
32      // Key not found -- return negative value indicating where a
33      // new entry for this key should go.  We use the end of the
34      // hash chain to reduce the number of array entries that will
35      // need to be copied when inserting.
36      return ~end;
37  }
```

我们看到第23-30行，如果hash值一样，在查找的时候不光往前查找而且还会往后查找。他的数据结构是这样的。



03 SparseArray

在散列表中如果可以确定key值都是int类型，那么又可以简化，直接用key值当hash值存储即可，和ArrayMap一样只需要两个数组即可，一个是存放key的，一个是存放value的，不同的是这两个数组的长度都是一样的。这种情况下就不会出现hash值一样的问题了，因为这个时候如果hash值一样的话，那么他们的key肯定是一样的，而在散列表中是不可能存在了，假如在插入数据的时候有一样的key，那么他的value是要被替换掉的，所以不会出现两个完全一样的key。他的数据结构图是这样的



04 ThreadLocalMap

在java语言中还有一个关于散列表的，那就是ThreadLocalMap，这个类是ThreadLocal的一个静态内部类，一般我们用不到。如果出现hash冲突的时候他的实现原理和上面的几种也都不太一样。存储的时候他首先会根据hash值映射到指定的数组，如果当前位置为空就直接存进去，如果不为空就往后找，找他的下一个，我们来看其中的一段代码

```
1     /**
2      * Increment i modulo len.
3      */
4     private static int nextIndex(int i, int len) {
5       return ((i + 1 < len) ? i + 1 : 0);
6     }
```

总结：

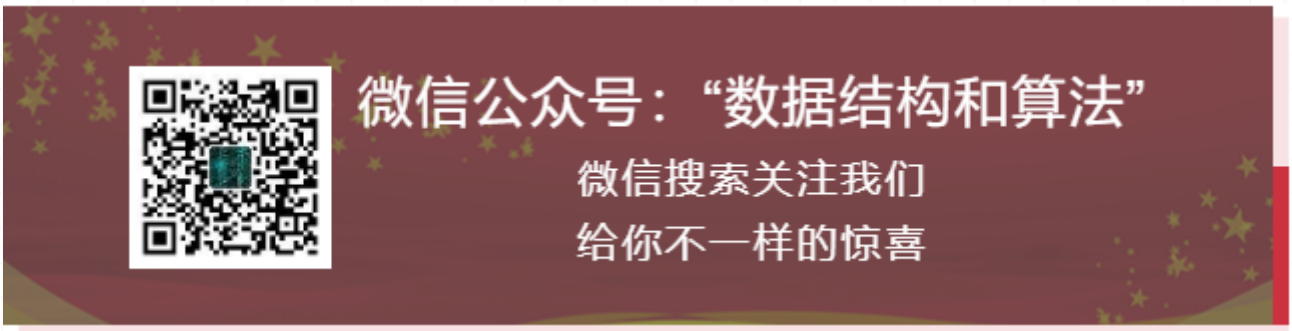
散列表大家第一个想到的就是HashMap，需要数组加链表的方式才能实现，通过我们上面的分析，其实我们不需要链表也能实现。散列表的实现原理其实很简单。他的核心是当我们的hash值出现冲突的时候该怎么解决。

373，数据结构-6,树

原创 山大王wld 数据结构和算法 6月1日

收录于话题
#常见数据结构

7个 >



No matter how good something is, you can ruin it by overthinking it.

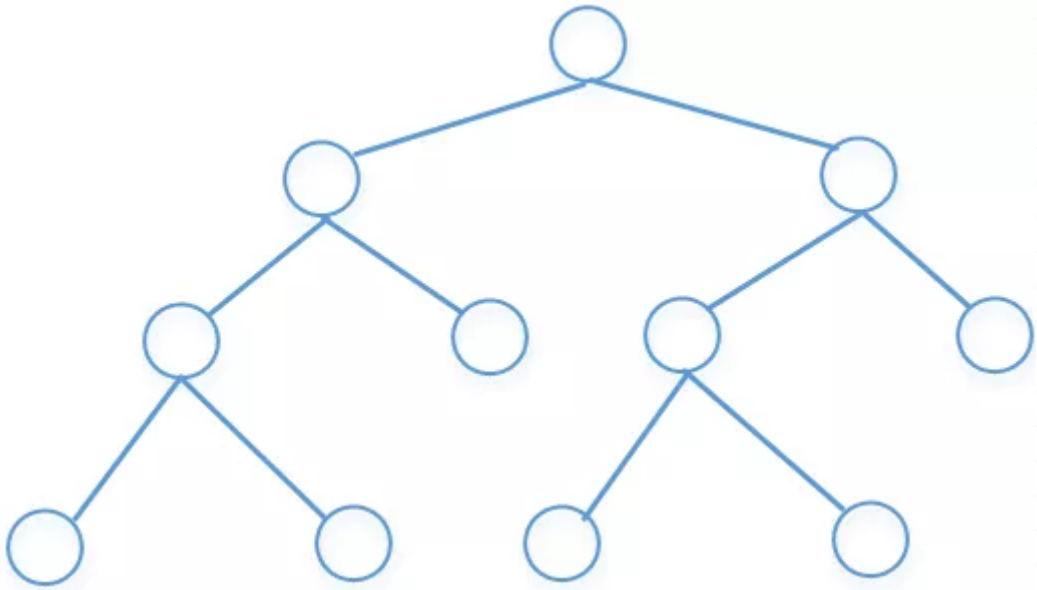
无论一件事有多美好，但若你想的太多反而可能会毁掉它。

基础知识

树是一个有n个有限节点组成一个具有层次关系的集合，每个节点有0个或者多个子节点，没有父节点的节点称为根节点，也就是说除了根节点以外每个节点都有父节点，并且有且只有一个。

树的种类比较多，有二叉树，红黑树，AVL树，B树，哈夫曼树，字典树等等。

甚至堆我们也可以把它看成是一棵树，树的这么多种类中，我们最常见的应该是二叉树了，下面我们来看一下他的结构。



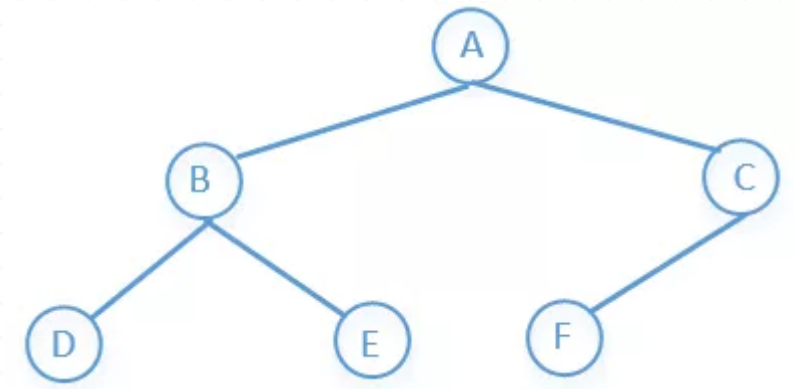
定义：

1. **结点的度**：一个结点含有的子结点的个数称为该结点的度；
2. **叶结点或终端结点**：度为0的结点称为叶结点；
3. **非终端结点或分支结点**：度不为0的结点；
4. **双亲结点或父结点**：若一个结点含有子结点，则这个结点称为其子结点的父结点；
5. **孩子结点或子结点**：一个结点含有的子树的根结点称为该结点的子结点；
6. **兄弟结点**：具有相同父结点的结点互称为兄弟结点；
7. **树的度**：一棵树中，最大的结点的度称为树的度；
8. **结点的层次**：从根开始定义起，根为第1层，根的子结点为第2层，以此类推；
9. **树的高度或深度**：树中结点的最大层次；
10. **堂兄弟结点**：双亲在同一层的结点互为堂兄弟；
11. **结点的祖先**：从根到该结点所经分支上的所有结点；
12. **子孙**：以某结点为根的子树中任一结点都称为该结点的子孙。
13. **森林**：由m（m>=0）棵互不相交的树的集合称为森林；
14. **无序树**：树中任意节点的子结点之间没有顺序关系，这种树称为无序树，也称为自由树；
15. **有序树**：树中任意节点的子结点之间有顺序关系，这种树称为有序树；
16. **二叉树**：每个节点最多含有两个子树的树称为二叉树；
17. **完全二叉树**：若设二叉树的深度为h，除第 h 层外，其它各层（1～h-1）的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树

18. **满二叉树：**除最后一层无任何子节点外，每一层上的所有结点都有两个子结点的二叉树。
19. **哈夫曼树：**带权路径最短的二叉树称为哈夫曼树或最优二叉树；

应用：

树的种类实在是太多，关于树的算法题也是贼多，这一篇文章不可能全部介绍完，我们需要具体问题再具体分析。这里主要介绍的是二叉树，并且只介绍树的一些最基础的几个算法。我们先来看个图

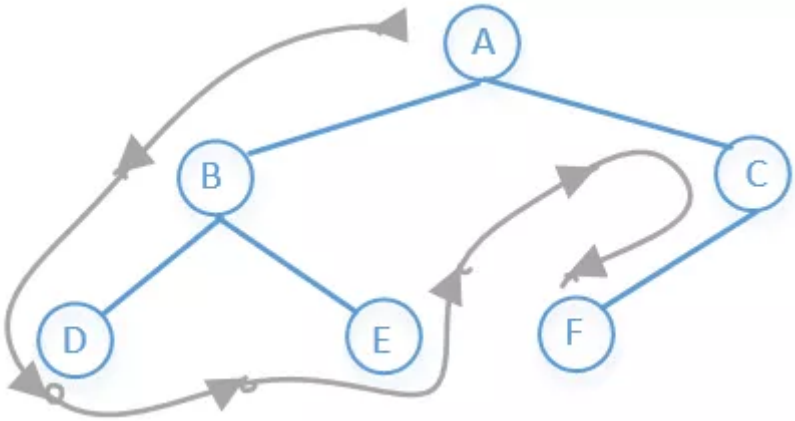


节点类

```java
public class TreeNode {
    public int val;
    public TreeNode left;
    public TreeNode right;

    public TreeNode(int x) {
        val = x;
    }

    public TreeNode() {
    }

    @Override
    public String toString() {
        return "[" + val + "]";
    }
}
```

01 前序遍历

他的访问顺序是：根节点→左子树→右子树
所以上图前序遍历的结果是：A→B→D→E→C→F
访问顺序如下



代码如下

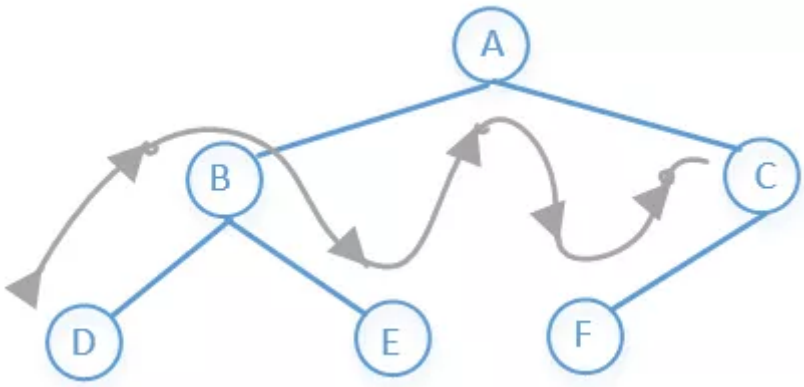
```
1   public static void preOrder(TreeNode tree) {
2     if (tree == null)
3       return;
4     System.out.printf(tree.val + "");
5     preOrder(tree.left);
6     preOrder(tree.right);
7   }
```

非递归的写法

```
1   public static void preOrder(TreeNode tree) {
2     if (tree == null)
3       return;
4     Stack<TreeNode> q1 = new Stack<>();
5     q1.push(tree);//压栈
6     while (!q1.empty()) {
7       TreeNode t1 = q1.pop();//出栈
8       System.out.println(t1.val);
9       if (t1.right != null) {
10        q1.push(t1.right);
11      }
12      if (t1.left != null) {
13        q1.push(t1.left);
14      }
15    }
16  }
```

02 中序遍历

他的访问顺序是：左子树→根节点→右子树
所以上图前序遍历的结果是：D→B→E→A→F→C
访问顺序如下



代码如下

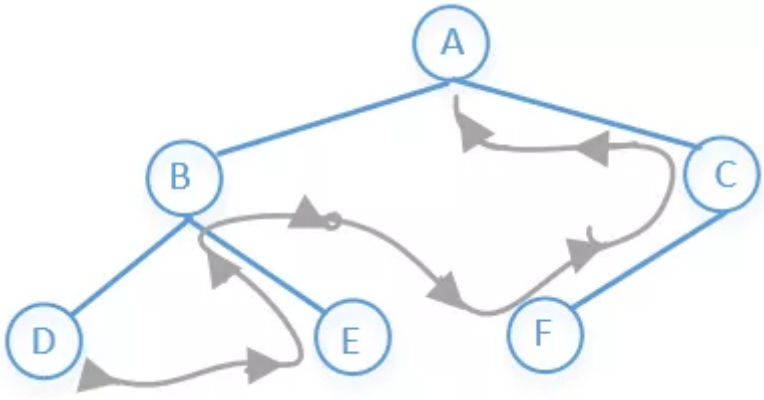
```
1   public static void inOrderTraversal(TreeNode node) {
2     if (node == null)
3       return;
4     inOrderTraversal(node.left);
5     System.out.println(node.val);
6     inOrderTraversal(node.right);
7   }
```

非递归的写法

```
1   public static void inOrderTraversal(TreeNode tree) {
2     Stack<TreeNode> stack = new Stack<>();
3     while (tree != null || !stack.isEmpty()) {
4       while (tree != null) {
5         stack.push(tree);
6         tree = tree.left;
7       }
8       if (!stack.isEmpty()) {
9         tree = stack.pop();
10        System.out.println(tree.val);
11        tree = tree.right;
12      }
13    }
14  }
```

03 后续遍历

他的访问顺序是：左子树→右子树→根节点
所以上图前序遍历的结果是：D→E→B→F→C→A
访问顺序如下



代码如下

```java
public static void postOrder(TreeNode tree) {
  if (tree == null)
    return;
  postOrder(tree.left);
  postOrder(tree.right);
  System.out.println(tree.val);
}
```

非递归的写法

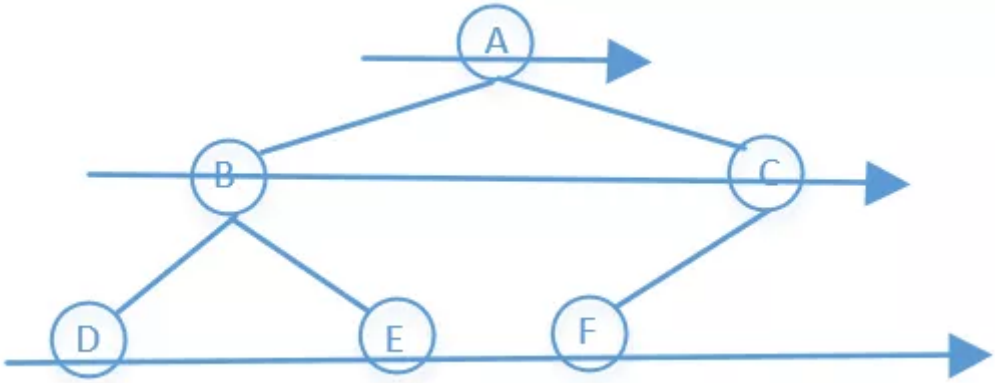
```java
public static void postOrder(TreeNode tree) {
  if (tree == null)
    return;
  Stack<TreeNode> s1 = new Stack<>();
  Stack<TreeNode> s2 = new Stack<>();
  s1.push(tree);
  while (!s1.isEmpty()) {
    tree = s1.pop();
    s2.push(tree);
    if (tree.left != null) {
      s1.push(tree.left);
    }
    if (tree.right != null) {
      s1.push(tree.right);
    }
  }
  while (!s2.isEmpty()) {
    System.out.print(s2.pop().val + " ");
  }
}
```

或者

```java
public static void postOrder(TreeNode tree) {
  if (tree == null)
    return;
  Stack<TreeNode> stack = new Stack<>();
  stack.push(tree);
  TreeNode c;
  while (!stack.isEmpty()) {
    c = stack.peek();
    if (c.left != null && tree != c.left && tree != c.right) {
      stack.push(c.left);
    } else if (c.right != null && tree != c.right) {
      stack.push(c.right);
    } else {
      System.out.print(stack.pop().val + " ");
      tree = c;
    }
  }
}
```

04 BFS(宽度优先搜索(又称广度优先搜索))

> 他的访问顺序是：先访问上一层，在访问下一层，一层一层的往下访问
>
> 所以上图前序遍历的结果是：A→B→C→D→E→F
>
> 访问顺序如下



代码如下

```java
public static void levelOrder(TreeNode tree) {
  if (tree == null)
    return;
  LinkedList<TreeNode> list = new LinkedList<>();//链表，这里我们可以把它看做队列
  list.add(tree);//相当于把数据加入到队列尾部
  while (!list.isEmpty()) {
    TreeNode node = list.poll();//poll方法相当于移除队列头部的元素
    System.out.println(node.val);
    if (node.left != null)
      list.add(node.left);
    if (node.right != null)
      list.add(node.right);
  }
}
```

递归的写法

```java
public static void levelOrder(TreeNode tree) {
  int depth = depth(tree);
  for (int level = 0; level < depth; level++) {
    printLevel(tree, level);
  }
}

private static int depth(TreeNode tree) {
  if (tree == null)
    return 0;
  int leftDepth = depth(tree.left);
  int rightDepth = depth(tree.right);
  return Math.max(leftDepth, rightDepth) + 1;
}


private static void printLevel(TreeNode tree, int level) {
  if (tree == null)
    return;
  if (level == 0) {
    System.out.print(" " + tree.val);
  } else {
    printLevel(tree.left, level - 1);
    printLevel(tree.right, level - 1);
  }
}
```

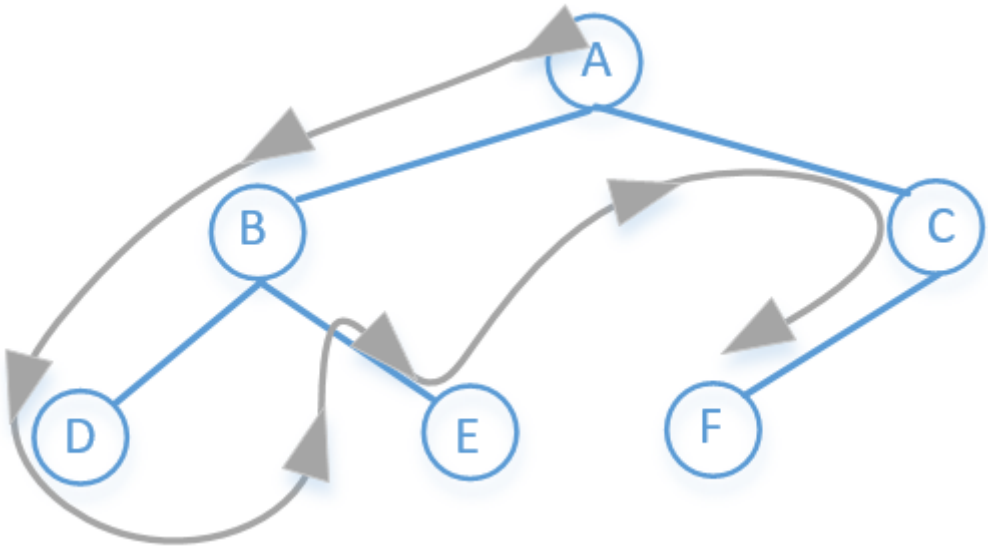
如果想把遍历的结果存放到list中，我们还可以这样写

```java
public static List<List<Integer>> levelOrder(TreeNode tree) {
  if (tree == null)
    return null;
  List<List<Integer>> list = new ArrayList<>();
  bfs(tree, 0, list);
  return list;
}

```

```
  private static void bfs(TreeNode tree, int level, List<List<Integer>> list) {
    if (tree == null)
      return;
    if (level >= list.size()) {
      List<Integer> subList = new ArrayList<>();
      subList.add(tree.val);
      list.add(subList);
    } else {
      list.get(level).add(tree.val);
    }
    bfs(tree.left, level + 1, list);
    bfs(tree.right, level + 1, list);
  }
```

05 DFS(深度优先搜索)

他的访问顺序是：先访根节点，然后左结点，一直往下，直到最左结点没有子节点的时候然后往上退一步到父节点，然后父节点的右子节点在重复上面步骤……
所以上图前序遍历的结果是：A→B→D→E→C→F
访问顺序如下



代码如下

```
  public static void treeDFS(TreeNode root) {
    Stack<TreeNode> stack = new Stack<>();
    stack.add(root);
    while (!stack.empty()) {
      TreeNode node = stack.pop();
      System.out.println(node.val);
      if (node.right != null) {
        stack.push(node.right);
      }
      if (node.left != null) {
        stack.push(node.left);
      }
    }
  }
```

递归的写法

```
  public static void treeDFS(TreeNode root) {
    if (root == null)
      return;
    System.out.println(root.val);
```

```
5       treeDFS(root.left);
6       treeDFS(root.right);
7     }
```

往期推荐

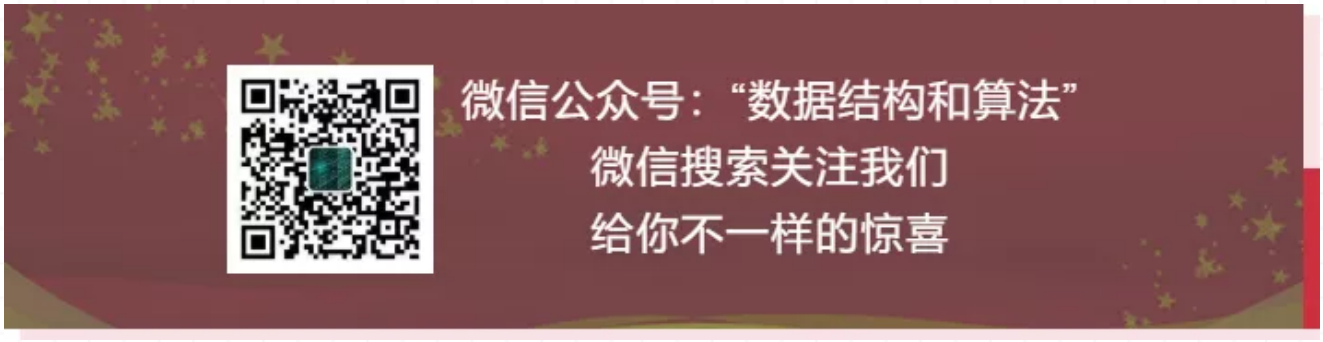
- 367，二叉树的最大深度
- 106，排序-堆排序

378，数据结构-7,堆

原创 山大王wld 数据结构和算法 6月7日

收录于话题
#常见数据结构

7个 >

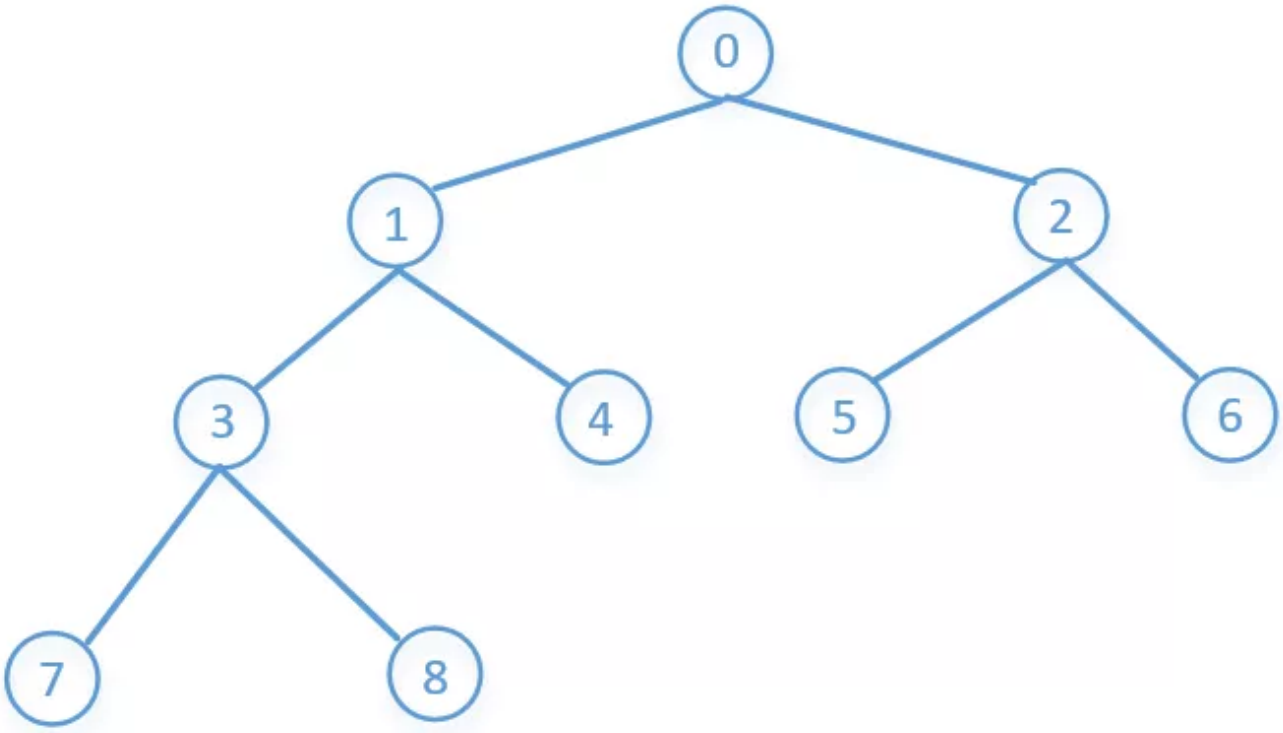


Make magic out of simple things.

在平凡中创造奇迹。

基础知识

通常情况下我们把堆看成是一棵完全二叉树。堆一般分为两种，一种是最大堆，一种是最小堆。最大堆要求根节点的值即大于左子树的值，又大于右子树的值。也就是说最大堆根节点的值是堆中最大的。最小堆根节点的值是堆中最小的，前面我们讲堆排序的时候介绍过堆，实际上他就是数组结构，但因为数组中间有关联，所以我们可以把它当做一棵完全二叉树来看，下面我们再来看一下堆的数据结构是什么样的。



结点中的数字是数组元素的下标，不是数组元素的值。所以如果我们知道父节点的下标我们就可以知道他的两个子节点（如果有子节点），如果知道子节点的下标也一定能找到父节点的下标，他们的关系是：

父节点的下标=（子节点下标-1）>>1；

左子节点下标=父节点下标*2+1；

右子节点下标=父节点下标*2+2；

堆的创建：

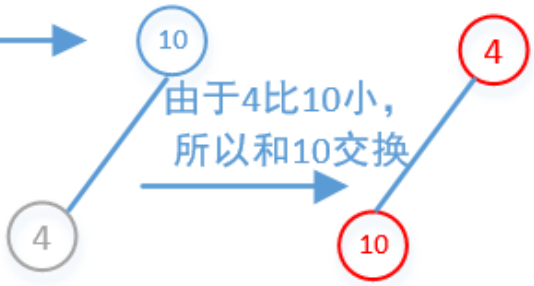
堆有两种，一种是最大堆，一种是最小堆。顾名思义，最大堆就是堆顶的元素是最大的，最小堆就是堆顶的元素是最小的。原理都差不多，我们只分析一个就行了，我们就以数组【10，4，8，3，5，1】为例来画个图分析一下最小堆是怎么创建的。

【10，4，8，3，5，1】

第一步，把10加入堆中

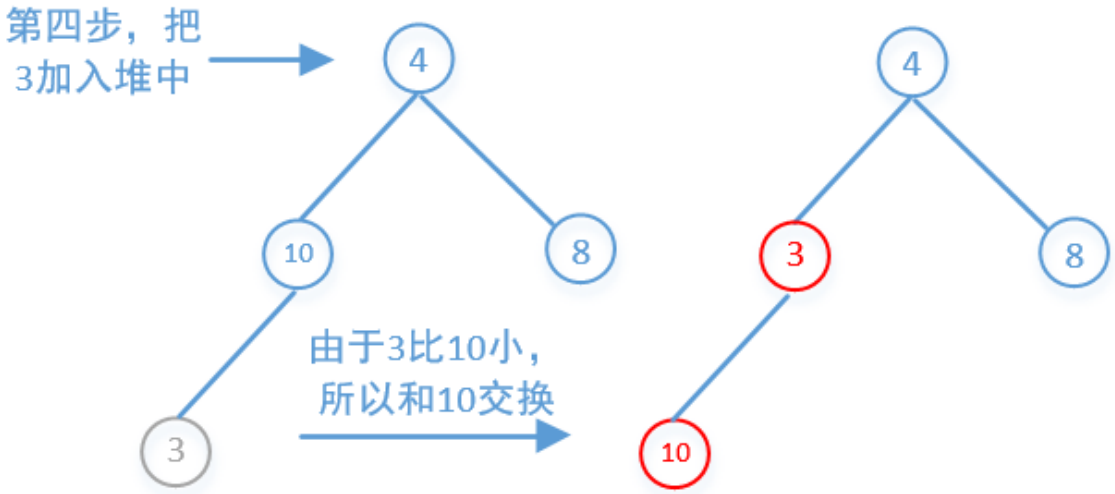


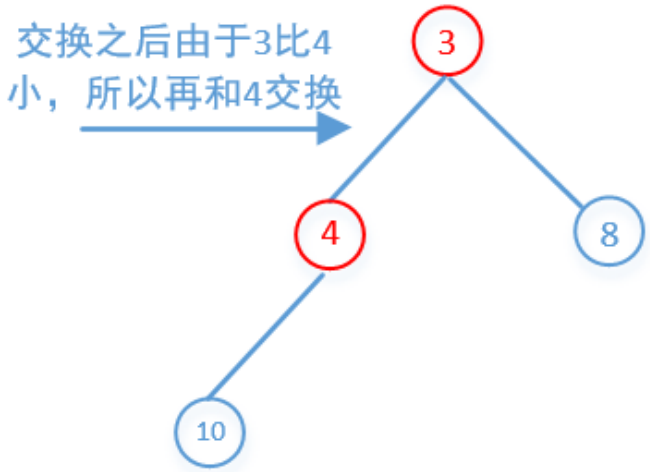
第二步，把4加入堆中

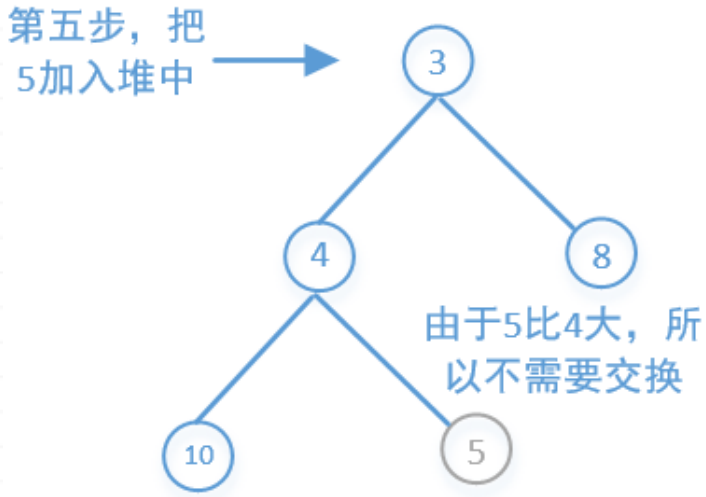


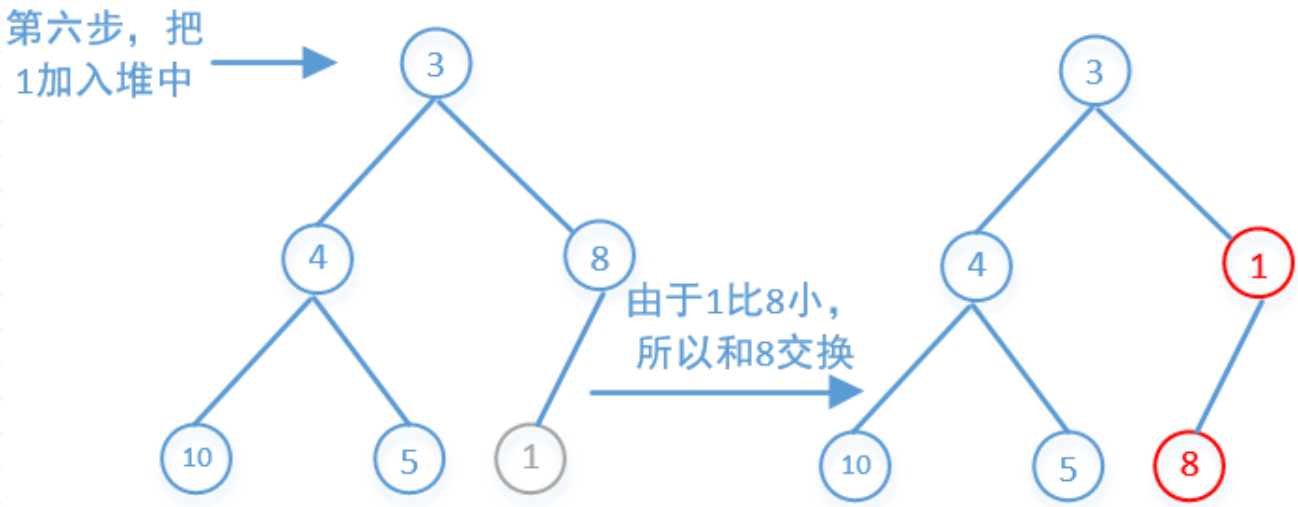
第三步，把8加入堆中

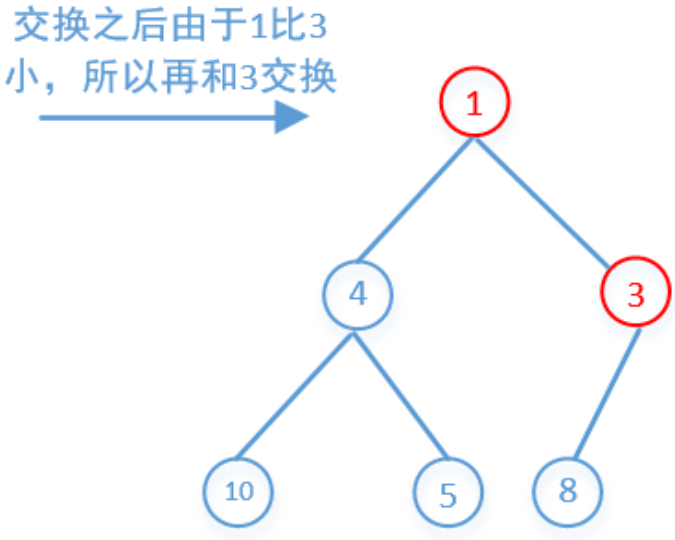



第四步，把
3加入堆中
4
10
8
3
由于3比10小，
所以和10交换
4
3
8
10


交换之后由于3比4
小，所以再和4交换
3
4
8
10





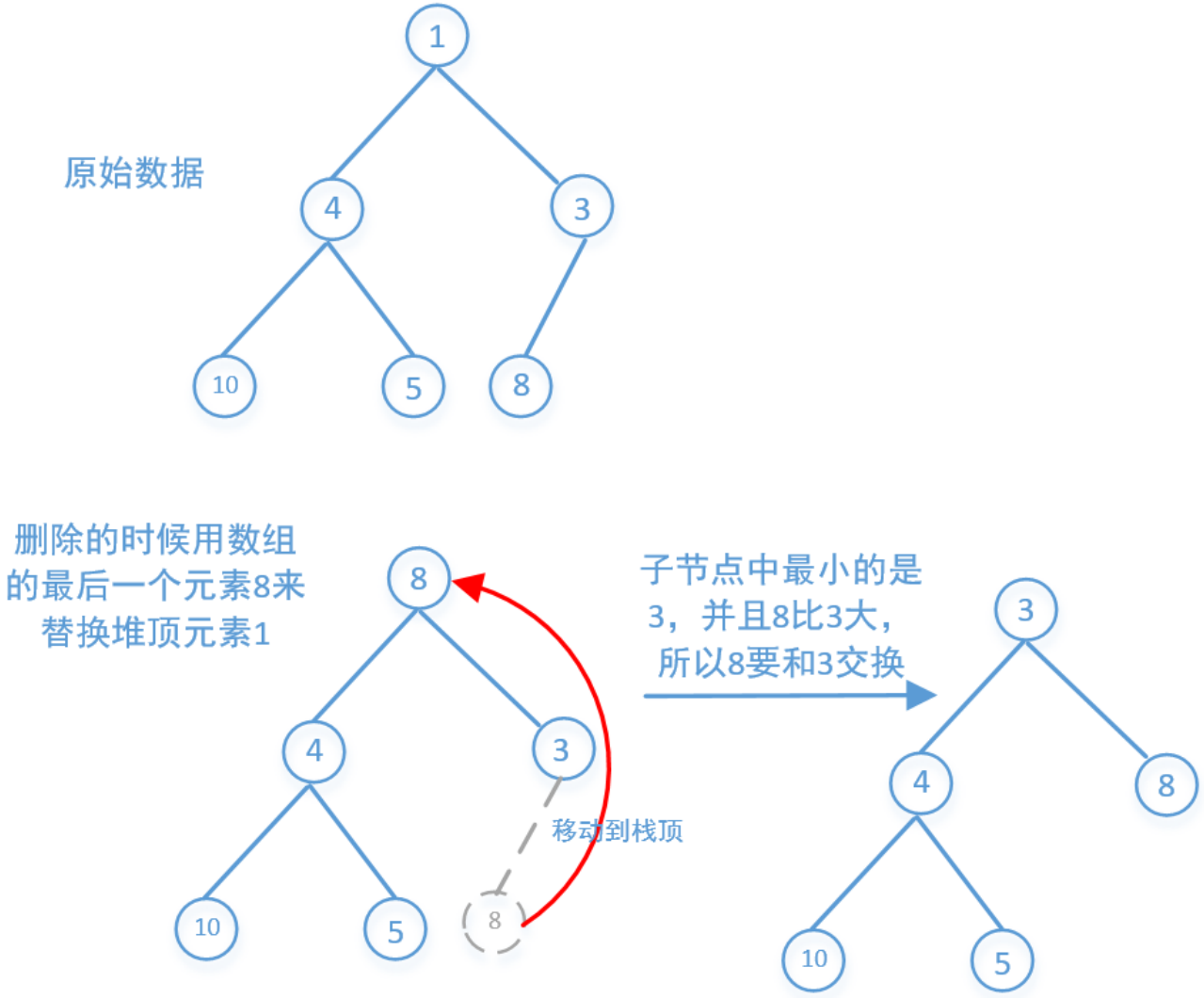


这就是堆的创建，把元素加入到堆中的时候，都要和父节点进行比对，在最小堆中，如果比父节点小，就要和父节点交换，交换之后还要继续和再和新的父节点对比……。同理在最大堆中，如果比父节点大，就要和父节点交换。

我们看到上面堆创建之后数组的元素顺序变为【1，4，3，10，5，8】

堆的删除

堆的添加会往上调整，堆的删除不仅会往下调整而且还有可能往上调整。堆中元素的删除我们可以从堆顶删除，也可以从中间某个位置删除，如果从堆顶删除的话我们只需要往下调整即可，因为堆顶没有父节点，不需要往上调整。如果从中间删除的话，我们先要往下调整，如果往下调整没有成功（比如在最小堆中，他比两个子节点都小），我们还要进行往上的调整。**调整的原理就是把数组最后一个元素放到要删除的元素位置上，然后在删除元素的位置上进行上下调整**，原理其实都差不多，我们来看一下最小堆顶部元素删除之后的调整。



上面是我们把堆顶元素1删除之后堆的调整，如果我们再把堆顶元素3删除之后，只需要用数组的最后一个元素5替换3，然后再往下调整即可……

代码

堆的添加和删除我们都分析过了，如果把前面的分析都搞懂的话，那么堆的代码就很容易写了，我们来看下

```
public class MyHeap<E> {
  private Object[] data;//数据存放区
  private int size;//堆的大小
  private Comparator<? super E> comparator;//比较器

  public MyHeap(int initialCapacity) {
    this(initialCapacity, null);
  }

  public MyHeap(int initialCapacity, Comparator<? super E> comparator) {
    if (initialCapacity < 1)
      throw new IllegalArgumentException("堆的大小必须大于0");
    this.data = new Object[initialCapacity];
    this.comparator = comparator;
  }

  /**
   * @param e 向堆中添加元素
   * @return
   */
  public boolean add(E e) {
    if (e == null)//不能为空
      throw new NullPointerException();
    if (size >= data.length)//如果堆的空间不够了就扩容，这里是扩大2倍
      data = Arrays.copyOf(data, data.length << 1);
    if (size == 0)//如果堆是空的，直接添加就可以了，不需要调整，因为就一个没发调整
      data[0] = e;
    else//如果堆不是空的，就要往上调整。
      siftUp(e);
    size++;//添加完之后size要加1
    return true;
  }

  public int getSize() {
    return size;
  }

  //删除堆顶元素
  public E remove() {
    if (size == 0)
      return null;
    size--;
    E result = (E) data[0];//获取堆顶的元素
    E x = (E) data[size];//取出数组的最后一个元素
    data[size] = null;//然后把最后一个元素的位置置空
    if (size != 0)
      siftDown(x);//这里实际上是把数组的最后一个元素取出放到栈顶，然后再往下调整。
    return result;
  }

  //访问堆顶元素，不删除
  public E peek() {
    return (size == 0) ? null : (E) data[0];
  }

  /**
   * 返回数组的值
```

```
58      *
59      * @param a
60      * @param <T>
61      * @return
62      */
63     public <T> T[] toArray(T[] a) {
64       if (a.length < size)
65         return (T[]) Arrays.copyOf(data, size, a.getClass());
66       System.arraycopy(data, 0, a, 0, size);
67       if (a.length > size)
68         a[size] = null;
69       return a;
70     }
71
72     /**
73      * 往上调整，往上调整只需要和父节点比较即可，如果比父节点大就不需要在调整
74      *
75      * @param e
76      */
77     private void siftUp(E e) {
78       int s = size;
79       while (s > 0) {
80         int parent = (s - 1) >>> 1;//根据子节点的位置可以找到父节点的位置
81         Object pData = data[parent];
82         //和父节点比较，如果比父节点大就退出循环不再调整
83         if (comparator != null) {
84           if (comparator.compare(e, (E) pData) >= 0)
85             break;
86         } else {
87           if (((Comparable<? super E>) e).compareTo((E) pData) >= 0)
88             break;
89         }
90         //如果比父节点小，就和父节点交换，然后再继续往上调整
91         data[s] = pData;
92         s = parent;
93       }
94       //通过上面的往上调整，找到合适的位置，再把e放进去
95       data[s] = e;
96     }
97
98     /**
99      * 往下调整，往下调整需要和他的两个子节点（如果有两个子节点）都要比较，哪个最小就和哪
100     * 个交换，如果比两个子节点都小就不用再交换
101     *
102     * @param e
103     */
104    private void siftDown(E e) {
105      int half = size >>> 1;
106      int index = 0;
107      while (index < half) {
108        int min = (index << 1) + 1;//根据父节点的位置可以找到左子节点的位置
109        Object minChild = data[min];
110        int right = min + 1;//根据左子节点找到右子节点的位置
111        if (right < size) {//如果有右子节点就执行这里的代码
112          //如果有右子节点，肯定会有左子节点。那么就需要左右两个子节点比较，把小的赋值给minChild
113          if (comparator != null) {
114            if (comparator.compare((E) minChild, (E) data[right]) > 0)
115              minChild = data[min = right];
116          } else {
117            if (((Comparable<? super E>) minChild).compareTo((E) data[right]) > 0)
118              minChild = data[min = right];
119          }
120        }
121        //用节点e和他的最小的子节点比较，如果小于他最小的子节点就退出循环，不再往下调整了，
122        if (comparator != null) {
123          if (comparator.compare(e, (E) minChild) <= 0)
124            break;
125        } else {
126          if (((Comparable<? super E>) e).compareTo((E) minChild) <= 0)
```

```
127                 break;
128             }
129             //如果e比它的最小的子节点小，就用最小的子节点和e交换位置，然后再继续往下调整。
130             data[index] = minChild;
131             index = min;
132         }
133         data[index] = e;
134     }
135 }
```

这里的删除和添加操作的都是堆顶的元素，所以添加的时候会调用siftUp进行往上调整，删除的时候会调用siftDown进行往下调整。我把注释都写在代码中了，这里就不再详细介绍。

堆排序

通过上面的分析，我们知道最大堆就是堆顶元素始终保持的是最大值，最小堆就是堆顶元素始终保持的是最小值，假如在最小堆中我们把堆顶元素一个个给移除不就相当于排序了吗。我们来测试一下

```
1     int[] array = {10, 4, 8, 3, 5, 1};
2     System.out.print("数组原始的值：");
3     Util.printIntArrays(array);
4     MyHeap myHeap = new MyHeap(10, new Comparator<Integer>() {
5         @Override
6         public int compare(Integer o, Integer t1) {
7             return (o - t1 > 0) ? 1 : -1;
8         }
9     });
10
11    for (int i = 0; i < array.length; i++) {
12        myHeap.add(array[i]);
13    }
14    System.out.println();
15    System.out.print("堆中元素的值：");
16    Util.printObjectArrays(myHeap.toArray(new Object[myHeap.getSize()]));
17    System.out.println();
18    System.out.print("排序之后的值：");
19    for (int i = 0, length = myHeap.getSize(); i < length; i++) {
20        System.out.print(myHeap.remove() + "\t");
21    }
```

我们来看一下打印结果

```
1   数组原始的值：10  4  8  3  5  1
2   堆中元素的值：1  4  3  10 5  8
3   排序之后的值：1  3  4  5  8  10
```

我们看到堆中元素的值是【1，4，3，10，5，8】和我们最开始分析创建堆的结果完全一致。并且最后一行完成了从小到大的顺序输出

总结：

上面我们主要分析是最小堆，如果是最大堆代码该怎么写呢，其实有两种方式，一种是直接在上面代码MyHeap类中修改，还一种是在创建构造函数的时候重写比较器Comparator，我们这里推荐使用第二种，比如我们想按照从大到小的顺序输出，我们来看下该怎么写

```
1     int[] array = {10, 4, 8, 3, 5, 1};
2     System.out.print("数组原始的值： ");
3     Util.printIntArrays(array);
4     MyHeap myHeap = new MyHeap(10, new Comparator<Integer>() {
5       @Override
6       public int compare(Integer o, Integer t1) {
7         return (o - t1 < 0) ? 1 : -1;
8       }
9     });
10
11    for (int i = 0; i < array.length; i++) {
12      myHeap.add(array[i]);
13    }
14    System.out.println();
15    System.out.print("堆中元素的值： ");
16    Util.printObjectArrays(myHeap.toArray(new Object[myHeap.getSize()]));
17    System.out.println();
18    System.out.print("排序之后的值： ");
19    for (int i = 0, length = myHeap.getSize(); i < length; i++) {
20      System.out.print(myHeap.remove() + "\t");
21    }
```

我们只需修改一个字符即可，就是在上面第7行把之前的大于号改为小于号，我们来看下运行结果

```
1   数组原始的值： 10   4   8   3   5   1
2   堆中元素的值： 10   5   8   3   4   1
3   排序之后的值： 10   8   5   4   3   1
```

最后完全实现了从大到小的输出。

往期推荐

- 106，排序-堆排序
- 367，二叉树的最大深度

101，排序-冒泡排序

原创 山大王wld 数据结构和算法 2018-11-22

从这一节开始就暂时不做题了，先了解一下常用的排序算法，查找算法以及常用的几种数据结构，完了之后再继续做题。
首先第一个常见的排序估计就是冒泡排序了，记得当年学C语言的时候学的第一个排序算法就是它，其实他的原理很简单，就和他的名字一样，先看一下代码

```
public static void bubbleSort(int array[]) {
    int length = array.length;
    for (int i = 0; i < length - 1; i++) {
        for (int j = i + 1; j < length; j++) {
            if (array[j] < array[i]) {
                swap(array, i, j);
            }
        }
    }
}

public static void swap(int[] A, int i, int j) {
    if (i != j) {
        A[i] ^= A[j];
        A[j] ^= A[i];
        A[i] ^= A[j];
    }
}
```

首先拿第一个元素和后面的所有一个个比较，如果比后面的大就交换，所以始终会保证第一个元素是最小的，然后再从第二个第三个，以此类推，swap方法表示交换两个数字的值。我们还可以再改一下

```
public static void bubbleSort(int array[]) {
    int n = array.length;
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < n - i; j++) {
            if (array[j] > array[j + 1]) {
                swap(array, j, j + 1);
            }
        }
    }
}
```

我们看到每次循环的时候j都是从0开始的，并且是相邻两个元素的比较，所以第一轮比完了之后会把最大的值放到数组的最后，第二轮的时候会把第二大的值放到数组的倒数

第二个位置，以此类推。他和上一个的区别是，上一个每次循环都是把小的往前排，而这个每次循环都是把大的往后排。也可以把for改为while循环

```
public static void bubbleSort(int array[]) {
    int count = array.length - 1;
    while (count > 0) {
        for (int i = 0; i < count; i++) {
            if (array[i] > array[i + 1]) {
                swap(array, i, i + 1);
            }
        }
        count--;
    }
}
```

其实效果都是一样的。如果原来数组本来就是排序好的，那么其实这种效率还不是很高，我们还可以再修改一下，当后面的已经排序好的时候其实完全可以终止循环的。

```
public static void bubbleSort(int array[]) {
    boolean flag = true;
    int count = array.length - 1;
    while (flag) {
        flag = false;
        for (int i = 0; i < count; i++) {
            if (array[i] > array[i + 1]) {
                swap(array, i, i + 1);
                flag = true;
            }
        }
        count--;
    }
}
```

当后面的都已经排序好的时候其实是不需要交换的，所以就会终止循环。

```java
public static void bubbleSort(int array[]) {
    int location;
    int count = array.length - 1;// 初始化最后交换位置为最后一个元素
    for (int i = 0; i < array.length - 1; i++) {
        location = count;
        for (int j = 0; j < location; j++) {
            if (array[j] > array[j + 1]) {
                swap(array, j, j + 1);
                count = j; // 记录无序位置的结束 ，有序从j+1位置开始
            }
        }
        if (count == location)    // 没有次序交换，排序完成
            break;
    }
}
```

这个就不用说了，注释已经写的很清楚了，其实无论怎么变形，整体思想还是没变，下面来看最后一种方式，利用递归的方式写冒泡排序

```java
public static void bubbleSort(int[] array, int n) {
    if (n == 1)
        return;
    if (array == null || array.length == 0)
        return;
    //逐渐减少n，每次都是把最大的放在最后面，直到n为1
    for (int i = 0; i < n - 1; i++) {
        if (array[i] > array[i + 1]) {
            swap(array, i, i + 1);
        }
    }
    bubbleSort(array, n - 1);
}
```

上面有注释就不在细说了，这里n第一次传值的时候是数组的长度。其实冒泡排序基本上也就这些东西。

102，排序-选择排序

原创 山大王wld 数据结构和算法 2018-11-23

选择排序和冒泡排序有一点点像，选择排序是默认前面都是已经排序好的，然后从后面选择最小的放在前面排序好的的后面，首先第一轮循环的时候默认的排序好的为空，然后从后面选择最小的放到数组的第一个位置，第二轮循环的时候默认第一个元素是已经排序好的，然后从剩下的找出最小的放到数组的第二个位置，第三轮循环的时候默认前两个都是已经排序好的，然后再从剩下的选择一个最小的放到数组的第三个位置，以此类推。下面看一下代码。

```
private static void selectSort(int[] array) {
    for (int i = 0; i < array.length; i++) {
        int index = i;
        for (int j = i + 1; j < array.length; j++) {
            if (array[index] > array[j]) {
                index = j;
            }
        }
        if (i != index) {
            swap(array, i, index);
        }
    }
}

public static void swap(int[] A, int i, int j) {
    A[i] ^= A[j];
    A[j] ^= A[i];
    A[i] ^= A[j];
}
```

我们看到每轮循环的时候并没有直接交换，而是从他后面的序列中找到最小的记录一下他的index索引，最后再交换，下面看一下他的递归是怎么实现的。

```java
//selectSort(array, 0);调用方式
private static void selectSort(int array[], int index) {
    if (index == array.length)
        return;
    int min = index, i = index + 1;
    for (; i < array.length; i++) {
        if (array[i] < array[min]) min = i;
    }
    if (min != index) {
        swap(array, index, min);
    }
    selectSort(array, ++index);
}

public static void swap(int[] A, int i, int j) {
    A[i] = A[i] + A[j];
    A[j] = A[i] - A[j];
    A[i] = A[i] - A[j];
}
```

代码也很好理解，这里的swap是交换两个数据，我换了种写法，和上面的交换写法不太一样，但其实实现的功能都是一样的。

103，排序-插入排序

原创 山大王wld 数据结构和算法 2018-11-26

插入排序的原理是默认前面的元素都是已经排序好的，然后从后面逐个读取插入到前面排序好的合适的位置，就相当于打扑克的时候每获取一张牌的时候就插入到合适的位置一样。插入排序可以分为两种，一种是直接插入还一种是二分法插入，直接插入的原理比较简单，就是往前逐个查找直到找到合适的位置然后插入，二分法插入是先折半查找，找到合适的位置然后再插入。说到二分法查找，等排序完之后就会介绍查找，有多种包括斐波那契查找，哈希查找，二分法查找等多个，其实这里面也可以使用，我们先看一下简单的直接插入排序代码

```
private static void insertSort(int[] array) {
    for (int i = 1; i < array.length; i++) {
        int j = i;
        int temp = array[i];
        for (; j > 0; j--) {
            if (array[j - 1] > temp) {
                array[j] = array[j - 1];//往后挪
            } else {//如果前面没有比他大则break
                break;
            }
        }
        array[j] = temp;
    }
}
```

可能还不是很严谨，如果array为空的怎么办，那么在修改一下

```java
private static void insertSort(int[] array) {
    if (array == null || array.length < 2) {
        return;
    }
    for (int i = 1; i < array.length; i++) {
        int key = array[i];
        int position = i;
        for (int j = i - 1; j >= 0; j--) {
            if (array[j] > key) {
                array[j + 1] = array[j];
                position--;
            } else {
                break;
            }
        }
        array[position] = key;
    }
}
```

代码原理其实都差不多，只是下面的稍微有点修改，在改变一下，换成while循环的方式

```java
private static void insertSort(int[] array) {
    int length = array.length;
    for (int i = 0; i < length; i++) {
        int j = i;
        int key = array[i];
        while (j > 0 && array[j - 1] > key) {
            array[j] = array[j - 1];
            j--;
        }
        if (i != j) {
            array[j] = key;
        }
    }
}
```

无论怎么变其实原理还是一样的，只是代码的书写方式可能有点区别，我们来看另外一种插入排序方法，二分法插入排序

```
private static void insertSort(int[] array) {
    int length = array.length;
    for (int i = 1; i < length; i++) {
        if (array[i - 1] > array[i]) {
            int key = array[i];
            int low = 0;
            int hight = i - 1;
            while (low <= hight) {
                int mid = (low + hight) >> 1;
                if (array[mid] > key) {
                    hight = mid - 1;
                } else {
                    low = mid + 1;
                }
            }
            for (int j = i; j > low; j--) {
                array[j] = array[j - 1];
            }
            array[low] = key;
        }
    }
}
```

如果数据很大的情况下，二分法插入排序明显比直接插入效率要高。我们再看一下如果使用递归怎么写，看下面代码

```
private static void insertSort(int[] data, int n) {
    if (n < 2) return;
    insertSort(data, --n);//相当于前面n-1个都已经排序好了
    int temp = data[n];//把最后一个元素插入到合适的位置
    int index = n - 1;
    while (index >= 0 && data[index] > temp) {
        data[index + 1] = data[index];
        index--;
    }
    data[index + 1] = temp;
}
```

OK，上面就是所谓的插入算法。

104，排序-快速排序

原创 山大王wld 数据结构和算法 2018-11-27

快速排序原理是首先要找到一个中枢，把小于中枢的值放到他前面，大于中枢的值放到他的右边，然后再以此方法对这两部分数据分别进行快速排序。先看一下代码

```
private static void quickSort(int[] array, int start, int end) {
    if (start < end) {
        int key = array[start];//用待排数组的第一个作为中枢
        int i = start;
        for (int j = start + 1; j <= end; j++) {
            if (key > array[j]) {
                swap(array, j, ++i);
            }
        }
        array[start] = array[i];//先挪，然后再把中枢放到指定位置
        array[i] = key;
        quickSort(array, start, end: i - 1);
        quickSort(array, start: i + 1, end);
    }
}

public static void swap(int[] A, int i, int j) {
    if (i != j) {
        A[i] ^= A[j];
        A[j] ^= A[i];
        A[i] ^= A[j];
    }
}
```

这里是先用待排数组的第一个作为中枢，把数组划分两部分，小于他的往前挪，那大于他的自然就在后面了，然后再把中枢值放到大于和小于他之间的位置。



第一步用2作为中枢，把小于他的往前挪，先看一下每次执行for循环后的结果

[2, 9, -9, 7, -6, 8, 0]
[2, -9, 9, 7, -6, 8, 0]
[2, -9, 9, 7, -6, 8, 0]
[2, -9, -6, 7, 9, 8, 0]
[2, -9, -6, 7, 9, 8, 0]
[2, -9, -6, 0, 9, 8, 7]

所以第一步for循环执行之后的代码是下面这样的。



然后再把中枢值放到指定的位置，结果为下面



所以第一步循环完之后把小于2的都放到了他的前面，大于2的都放到了他的后面，然后再通过递归把前半部分和后半部分别排序，一直分下去，直到不能分为止,我们看一下每次执行的代码

[0, -9, -6, 2, 9, 8, 7]
[-6, -9, 0, 2, 9, 8, 7]
[-9, -6, 0, 2, 9, 8, 7]
[-9, -6, 0, 2, 7, 8, 9]
[-9, -6, 0, 2, 7, 8, 9]
[-9, -6, 0, 2, 7, 8, 9]

快速排序其实有很多变种，我们可以再改一下，代码如下

```java
private static void quickSort(int[] array, int start, int end) {
    if (start < end) {
        int pivot = partition(array, start, end);
        quickSort(array, start, end: pivot - 1);
        quickSort(array, start: pivot + 1, end);
    }
}

private static int partition(int[] array, int start, int end) {
    int pivotElem = array[start];
    int i = start;
    for (int j = start + 1; j <= end; j++) {
        if (array[j] < pivotElem) {
            swap(array, ++i, j);
        }
    }
    swap(array, i, start);
    return i;
}
```

这种实现方式和上面是一样的，只不过是换了种写法，他的原理中枢值先不动，把小于中枢的放到前面，大于中枢的放到后面，最后再把中枢值放到指定的位置。下面在看一种写法，中枢值始终是变动的，一会在前一会在后，循环结束的时候小于中枢的值放到了中枢的前面，大于中枢的值放到中枢值的后面，下面看一下代码

```java
private static void quickSort(int[] array, int start, int end) {
    if (start < end) {
        int pivot = partition(array, start, end);
        quickSort(array, start, end: pivot - 1);
        quickSort(array, start: pivot + 1, end);
    }
}

private static int partition(int[] array, int start, int end) {
    int pivot = start;
    while (start != end) {
        if (pivot != end) {
            /**
             * 第一次循环的时候用第一个元素作为中枢，和最后一个进行对比，
             * 如果小于最后一个元素，执行end--就是和倒数第二个元素进行
             * 对比，以此类推。如果大于最后一个元素那么就和最后一个元素
             * 交换，然后让pivot指向最后一个元素，下一轮循环的时候回指向
             * 下面的else方法和前面的元素进行比较。也即是说这个中枢的位置
             * 始终是在变动的，所以这一轮执行完了之后小于中枢的值就会放到
             * 他的前面，大于中枢的值就会放到他的后面。
             */
            if (array[end] < array[pivot]) {
                swap(array, end, pivot);
                pivot = end;
            } else {
                end--;
            }
        } else {
            if (array[start] > array[pivot]) {
                swap(array, start, pivot);
                pivot = start;
            } else {
                start++;
            }
        }
    }
    return pivot;
}
```

如果看的不是很明白，那么在换一种方式书写，看代码

```
private static void quickSort(int[] array, int start, int end) {
    if (start < end) {
        int pivot = partition(array, start, end);
        quickSort(array, start, end: pivot - 1);
        quickSort(array, start: pivot + 1, end);
    }
}

private static int partition(int[] array, int start, int end) {
    int pivot = array[start];// 采用子序列的第一个元素作为枢纽元素
    while (start < end) {//
        // 从后往前在后半部分中寻找第一个小于枢纽元素的元素
        while (start < end && array[end] >= pivot) {
            --end;
        }
        // 将这个比枢纽元素小的元素交换到前半部分
        swap(array, start, end);
        // 从前往后在前半部分中寻找第一个大于枢纽元素的元素
        while (start < end && array[start] <= pivot) {
            ++start;
        }
        swap(array, start, end);// 将这个枢纽元素大的元素交换到后半部分
    }
    return start;// 返回枢纽元素所在的位置
}
```

下面在看一种方式，就是中枢不变，中枢之后的元素进行最前和最后的交换，最后再把中枢值放到指定位置，其实就相当于上面两种方式的结合，看代码

```
private static void quickSort(int[] array, int start, int end) {
    if (start < end) {
        int pivot = partition(array, start, end);
        quickSort(array, start, end: pivot - 1);
        quickSort(array, start: pivot + 1, end);
    }
}

private static int partition(int[] array, int start, int end) {
    int pivot = start;
    while (start != end) {
        while (start < end && array[start] < array[pivot]) {
            start++;
        }
        while (start < end && array[end] >= array[pivot]) {
            end--;
        }
        swap(array, start, end);
    }
    pivot = start;
    return pivot;
}
```

OK，快速排序基本上也就这些东西，无论怎么变，原理是不变的，都是把小的往前挪大的往后挪，然后重复上面的步骤。我们上面的代码好像都是用了递归，我们来看一下使用非递归的方式怎么写

```
private static void quickSort(int[] a, int start, int end) {
    Stack<Integer> temp = new Stack();
    temp.push(end);
    temp.push(start);
    while (!temp.empty()) {
        int i = temp.pop();//start
        int j = temp.pop();//end
        int k = partition(a, i, j);
        if (k > i) {
            temp.push( e: k - 1);//end
            temp.push(i);//start
        }
        if (j > k) {
            temp.push(j);//end
            temp.push( e: k + 1);//start
        }
    }
}

private static int partition(int[] array, int start, int end) {
    int pivot = array[start];// 采用子序列的第一个元素作为枢纽元素
    while (start < end) {//
        // 从后往前在后半部分中寻找第一个小于枢纽元素的元素
        while (start < end && array[end] >= pivot) {
            --end;
        }
        // 将这个比枢纽元素小的元素交换到前半部分
        swap(array, start, end);
        // 从前往后在前半部分中寻找第一个大于枢纽元素的元素
        while (start < end && array[start] <= pivot) {
            ++start;
        }
        swap(array, start, end);// 将这个枢纽元素大的元素交换到后半部分
    }
    return start;// 返回枢纽元素所在的位置
}
```

105，排序-归并排序

原创 山大王wld 数据结构和算法 2018-11-28

这一部分来分析一下归并排序，归并排序是把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。看一下代码

```
public static void mergeSort(int array[], int left, int right) {
    if (left < right) {
        int center = (left + right) >> 1;
        mergeSort(array, left, center);
        mergeSort(array, left: center + 1, right);
        merge(array, left, center, right);
    }
}

public static void merge(int[] data, int left, int center, int right) {
    int[] tmp = new int[data.length];
    int tempIndex = left;
    //_left是前半部分开始的位置，_right是后半部分开始的位置
    int _left = left;
    int _right = center + 1;
    while (_left <= center && _right <= right) {
        if (data[_left] <= data[_right]) {
            tmp[tempIndex++] = data[_left++];
        } else {
            tmp[tempIndex++] = data[_right++];
        }
    }
    while (_right <= right) {
        tmp[tempIndex++] = data[_right++];
    }
    while (_left <= center) {
        tmp[tempIndex++] = data[_left++];
    }
    while (left <= right) {
        data[left] = tmp[left++];
    }
}
```

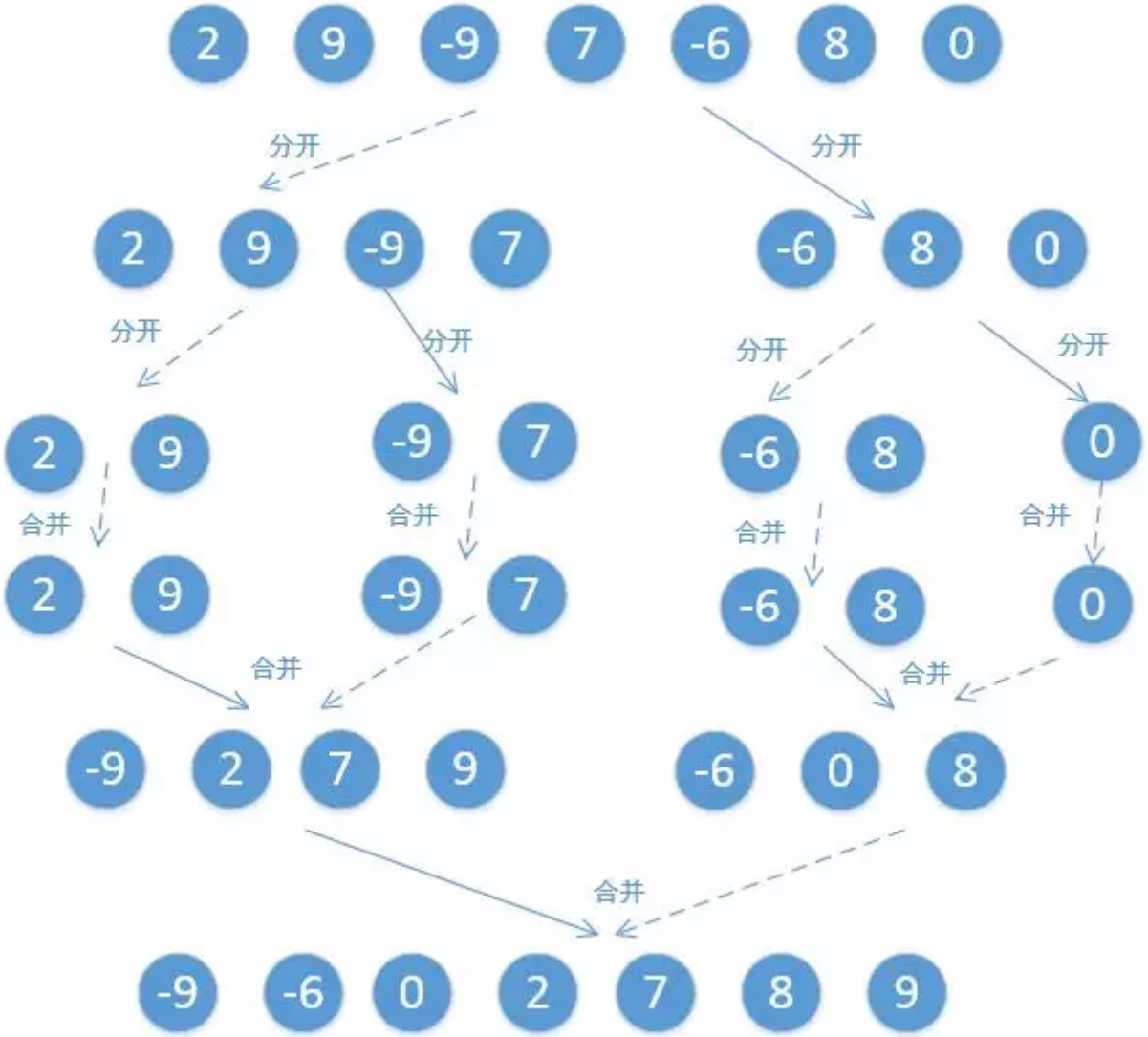
先把待排序列分为两部分，然后各部分再分为两部分，一直分下去，直到不能再分为止，然后在两两合并两个有序数组，直到合并完为止。有序数组的合并也很好理解，代码可以参考上面。上面代码在合并的时候都会创建一个临时数组tmp，如果排序的数组很大的话，每次merge的时候都会浪费大量的空间，不是最好的解决方式，这里可以优化一下，看代码

```java
public static void mergeSort(int array[], int left, int right) {
    if (left < right) {
        int center = (left + right) >> 1;
        mergeSort(array, left, center);
        mergeSort(array, left: center + 1, right);
        merge(array, left, center, right);
    }
}

public static void merge(int[] data, int left, int center, int right) {
    int length = right - left + 1;
    int[] tmp = new int[length];
    int tempIndex = 0;
    //_left是前半部分开始的位置，_right是后半部分开始的位置
    int _left = left;
    int _right = center + 1;
    while (_left <= center && _right <= right) {
        if (data[_left] <= data[_right]) {
            tmp[tempIndex++] = data[_left++];
        } else {
            tmp[tempIndex++] = data[_right++];
        }
    }
    while (_right <= right) {
        tmp[tempIndex++] = data[_right++];
    }
    while (_left <= center) {
        tmp[tempIndex++] = data[_left++];
    }
    tempIndex = 0;
    while (tempIndex < length) {
        data[left + tempIndex] = tmp[tempIndex++];
    }
}
```

上面的代码都是递归实现的，下面看一个非递归实现的

```java
public static void mergeSort(int[] data) {
    int i = 1;
    while (i < data.length) {
        //原理很简单，就是先两个两个合并，然后4个，然后8个……
        for (int j = 0; j + i < data.length; j += 2 * i) {
            merge(data, j, center: j + i - 1, Math.min(j + 2 * i - 1, data.length - 1));
        }
        i = i << 1;
    }
}

public static void merge(int[] data, int left, int center, int right) {
    int length = right - left + 1;
    int[] tmp = new int[length];
    int tempIndex = 0;
    //_left是前半部分开始的位置，_right是后半部分开始的位置
    int _left = left;
    int _right = center + 1;
    while (_left <= center && _right <= right) {
        if (data[_left] <= data[_right]) {
            tmp[tempIndex++] = data[_left++];
        } else {
            tmp[tempIndex++] = data[_right++];
        }
    }
    while (_right <= right) {
        tmp[tempIndex++] = data[_right++];
    }
    while (_left <= center) {
        tmp[tempIndex++] = data[_left++];
    }
    tempIndex = 0;
    while (tempIndex < length) {
        data[left + tempIndex] = tmp[tempIndex++];
    }
}
```

他说先合并两个然后在合并4个，然后在合并8个……直到合并完为止，下面有一幅图来简单了解下什么叫归并排序。



上面的图表示的很明白，这就是归并排序的原理

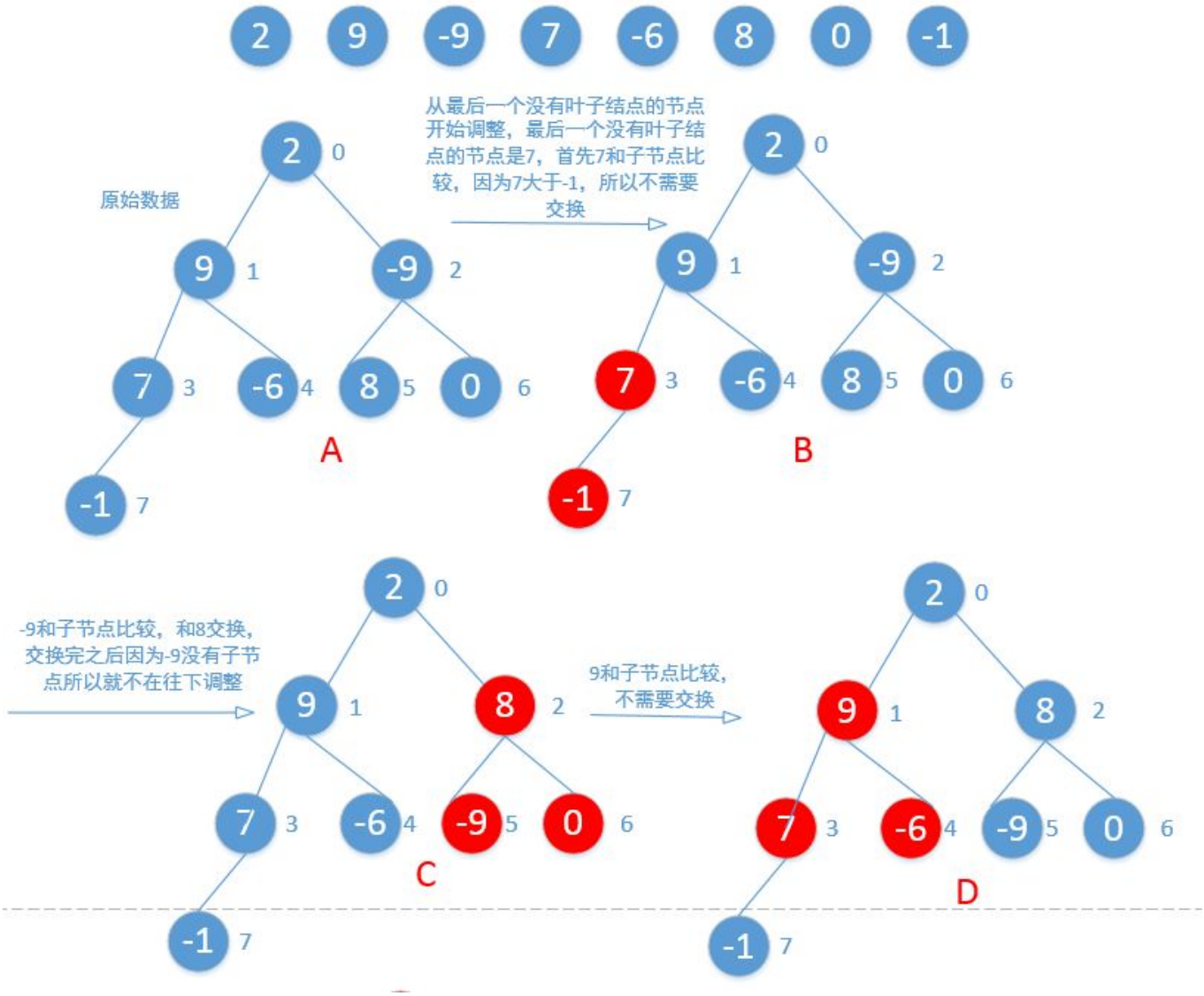
106，排序-堆排序

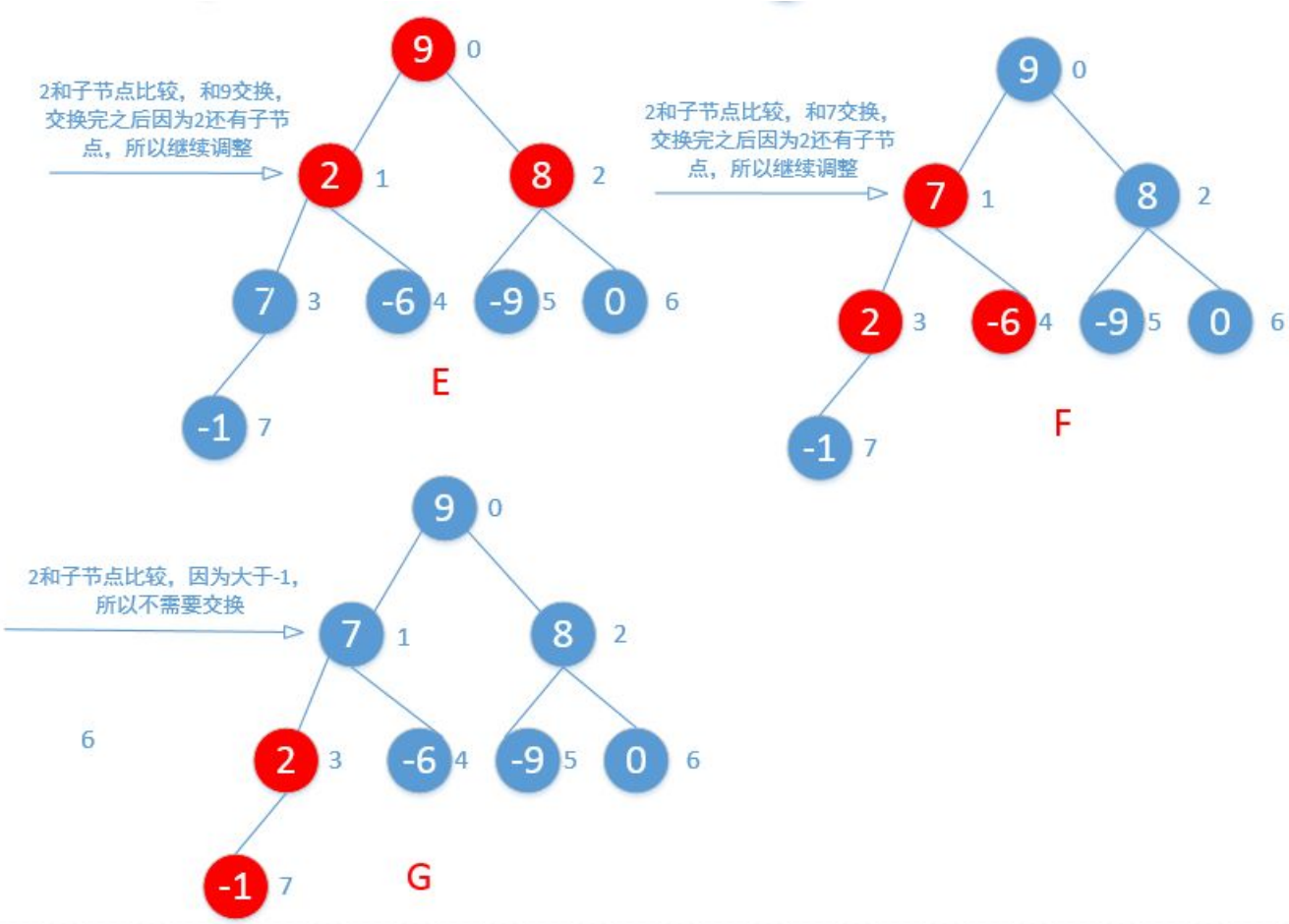
原创 山大王wld 数据结构和算法 2018-11-29

这一部分来分析一下堆排序，也可以理解为二叉树排序，这里的堆分为两种，一种是大顶堆，一种是小顶堆，我们所有的排序方法都以升序为主，其实倒序原理也都差不多，所以这里我们主要分析的是大顶堆。大顶堆就是根节点不小于他的两个子节点，先看一下代码

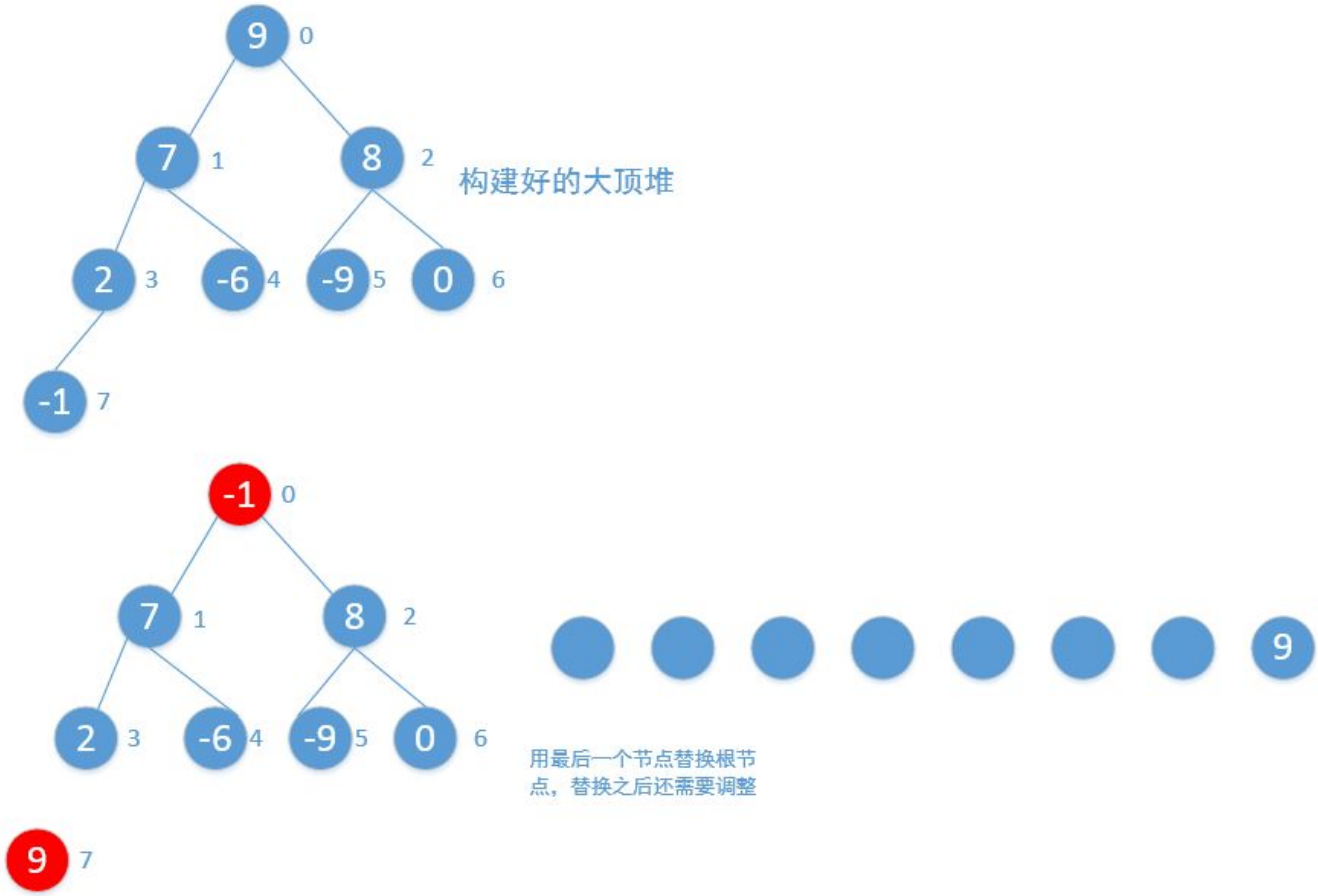
```
private static void heapSort(int[] array) {
    int length = array.length;
    buildMaxHeap(array, length);
    for (int i = 0; i < length; i++) {
        swap(array, i: 0, j: length - 1 - i);
        maxHeapfy(array, i: 0, heapSize: length - i - 1);
    }
}

private static void maxHeapfy(int[] array, int i, int heapSize) {
    int left = i * 2 + 1;
    int right = i * 2 + 2;
    int largest = i;
    if (left < heapSize && array[left] > array[largest]) {
        largest = left;
    }
    if (right < heapSize && array[right] > array[largest]) {
        largest = right;
    }
    if (largest != i) {//把最大值给父节点
        swap(array, largest, i);
        maxHeapfy(array, largest, heapSize);
    }
}

private static void buildMaxHeap(int[] array, int heapSize) {
    //从最后一个非叶子节点开始循环
    for (int i = (heapSize - 2) >> 1; i >= 0; i--) {
        maxHeapfy(array, i, heapSize);
    }
}

public static void swap(int[] A, int i, int j) {
    if (i != j) {
        A[i] ^= A[j];
        A[j] ^= A[i];
        A[i] ^= A[j];
    }
}
```

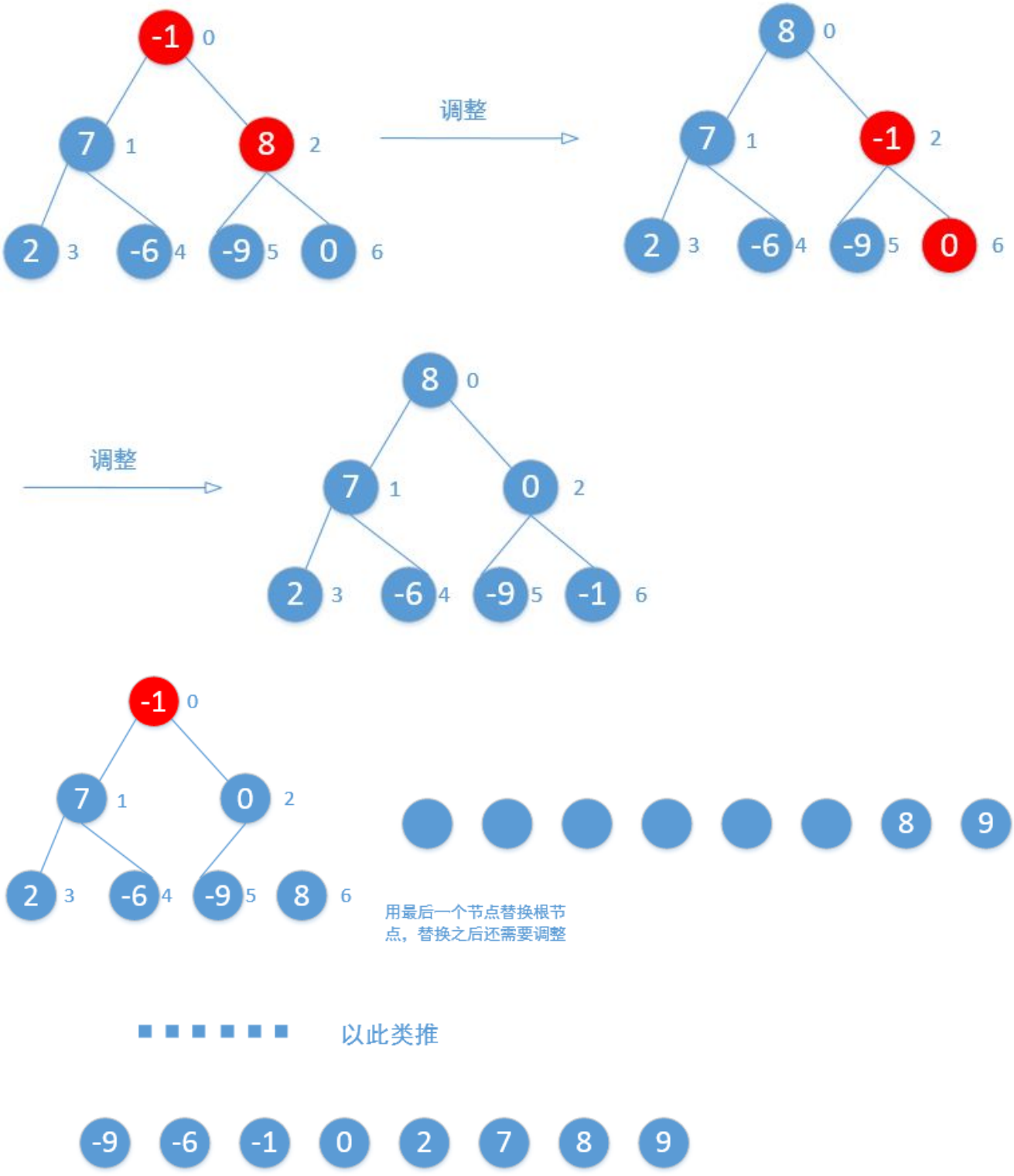
上面代码中heapSort方法表示对数组进行排序，buildMaxHeap表示堆的构建，maxHeapfy表示堆的调整，包括每次截取数据的时候也都需要调整，截取数据就相当于把root节点截取，然后用最后的一个节点替换到root的位置，然后再进行调整。看代码可能不是很直观，下面看一下图有助于理解





这是堆的构建的过程，其实也就是代码中的buildMaxHeap方法执行的过程，下面再来看一下是怎么排序的。





其实就是每次截取root节点然后存放到数组中，存放数组的位置是从后往前，然后把最后一个节点截取放到原来的root节点的位置，因为最后一个节点放到root节点，打破了二叉树的平衡，所以要从root节点开始进行调整。然后通过不断的循环不断的截取不断的再调整，直到截取完为止。

107，排序-桶排序

原创 山大王wld 数据结构和算法 2018-11-30

桶排序是将数组分散到有限的桶中，然后每个桶再分别排序，而每个桶的排序又可以使用其他排序方式进行排序，可以是桶排序也可以是其他排序。桶的大小可以随便定，如果桶的数量足够多就会变成我们后面介绍的计数排序，其实我们完全可以把桶固定在一个数量，根据数组的大小来确定，也可以自己定，比如3个或者5个7个等，桶的大小确定之后，下一步就需要把数组中的值一一存放到桶里，小的值就会放到前面的桶里，大的值就会放到后面的桶里，中间的值就会放到中间的桶里，然后再分别对每个桶进行单独排序，最后再把所有桶的数据都合并到一起就会得到排序好的数组，看代码

```
public static void bucketSort2(int[] array, int bucketSize) {
    int arrayLength = array.length;
    int max = array[0];
    int min = array[0];
    for (int i = 0; i < arrayLength; i++) {
        if (array[i] > max)
            max = array[i];
        else if (array[i] < min)
            min = array[i];
    }
    //bucketSize表示每个桶存放数据的大小，bucketCount总共桶的数量
    int bucketCount = (max - min) / bucketSize + 1;
    List<List<Integer>> buckets = new ArrayList<>(bucketCount);
    for (int i = 0; i < bucketCount; i++) {
        buckets.add(new ArrayList<Integer>());
    }

    for (int i = 0; i < arrayLength; i++) {
        //根据value的大小存放到不同的桶里，最终的结果是小的出现在前面的桶里，
        //大的出现在后面的桶里吗，中间的也就在中间的桶里了，然后再对每个桶分别
        //进行排序。
        buckets.get((array[i] - min) / bucketSize).add(array[i]);
    }

    int currentIndex = 0;
    for (int i = 0; i < buckets.size(); i++) {
        //取出每个桶的数据
        Integer[] bucketArray = new Integer[buckets.get(i).size()];
        bucketArray = buckets.get(i).toArray(bucketArray);
        //每一个桶进行排序,这里面可以选择其他排序算法进行排序
        Arrays.sort(bucketArray);
        for (int j = 0; j < bucketArray.length; j++) {
            array[currentIndex++] = bucketArray[j];
        }
    }
}
```

我们用一组数据来测试一下

```
public static void main(String src[]) {
    int array[] = {2, 6, 9, 3, 5, 1, -9, 7, -3, -1, -6, 8, 0};
    bucketSort2(array, bucketSize: 3);
    System.out.println(Arrays.toString(array));
}
```

看一下运行结果

```
[-9, -6, -3, -1, 0, 1, 2, 3, 5, 6, 7, 8, 9]
```

结果完成正确，这就是所谓的桶排序，首先要找到他的最大值和最小值，然后计算桶的数量，找出最小值是因为存放的时候要让当前值减去最小值，否则当排序中有负数的时候存放到桶里会报异常，代码中也都有注释，这里就不在详细介绍。

108，排序-基数排序

原创 山大王wld 数据结构和算法 2018-12-03

基数排序的方式可以采用最低位优先LSD（Least sgnificant digital）法或最高位优先MSD（Most sgnificant digital）法，LSD的排序方式由键值的最右边开始，而MSD则相反，由键值的最左边开始。我们这里使用LSD法，原理就是一个数组我们首先根据他的个位进行排序，然后在根据十位，百位……，这里最多排到多少位是根据他的最大值确定的，如果最大值有千位，我们必须要计算到千位，如果最多只有十位，我们就计算到十位就可以了，每一位都排序完了之后，数组也就排序成功了，来看一下代码

```
private static void radixSort(int[] array) {
    int digitCount = 10;//从0到9最多10位数
    int maxCount = getBitCount(getMaxNum(array));
    int radix = 1;
    int[][] tempArray = new int[digitCount][array.length];
    for (int i = 0; i < maxCount; i++) {
        int[] count = new int[digitCount];
        for (int j = 0; j < array.length; j++) {
            int temp = ((array[j] / radix) % 10);
            tempArray[temp][count[temp]++] = array[j];
        }
        int index = 0;
        for (int j = 0; j < digitCount; j++) {
            if (count[j] == 0)
                continue;
            for (int k = 0; k < count[j]; k++) {
                array[index++] = tempArray[j][k];
            }
        }
        radix *= 10;
    }
}
private static int getBitCount(int num) {
    int count = 1;
    int temp = num / 10;
    while (temp != 0) {
        count++;
        temp /= 10;
    }
    return count;
}
private static int getMaxNum(int array[]) {
    int max = array[0];
    for (int i = 1, length = array.length; i < length; i++) {
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max;
}
```

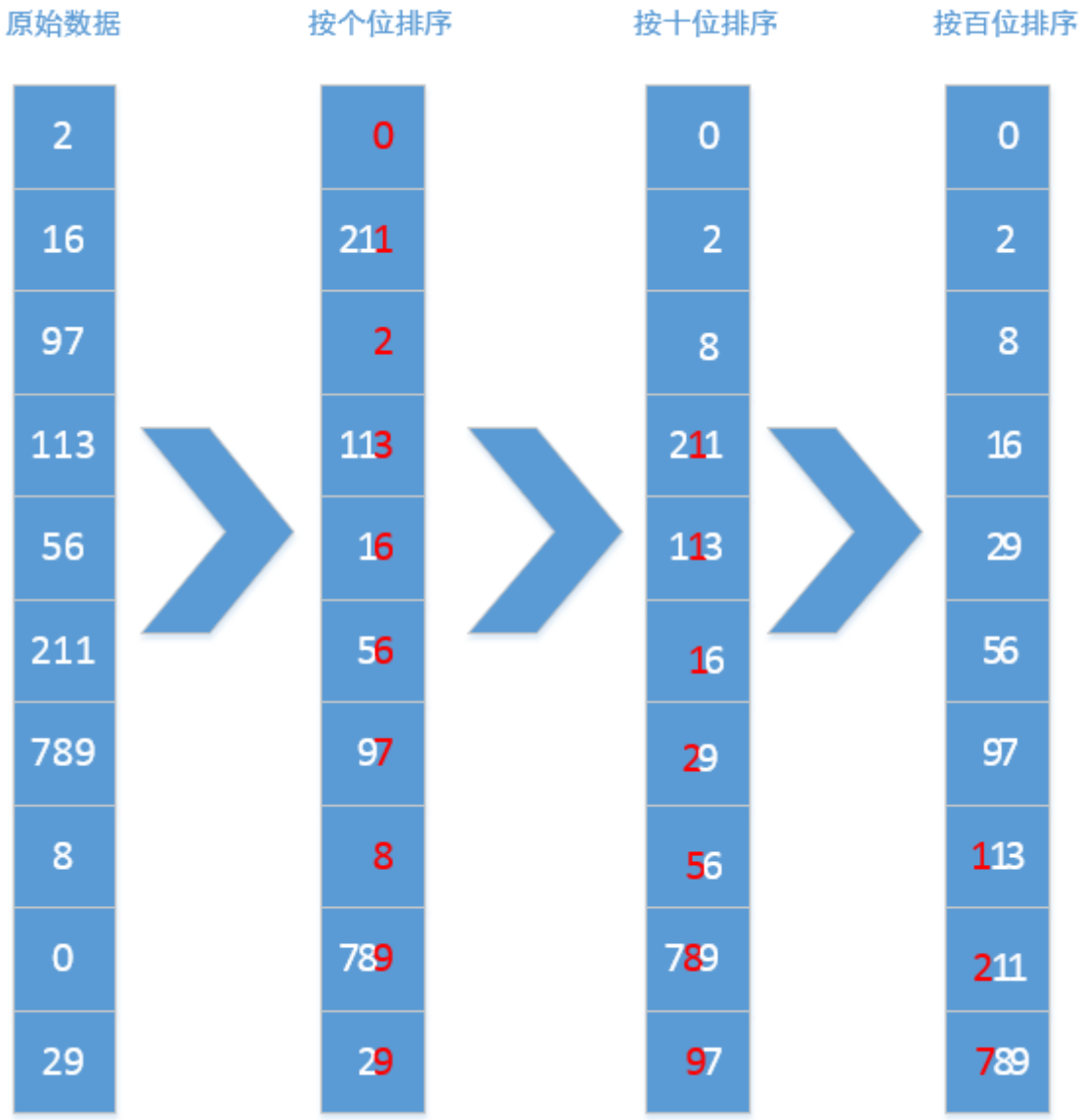
如果了解基数排序的上面代码可能很容易理解，我们随便找一组代码来测试一下结果吧

```
int array[] = {2, 16, 97, 113, 56, 211, 789, 8,  0, 29};
radixSort(array);
System.out.println(Arrays.toString(array));
```

看一下运行结果

```
[0, 2, 8, 16, 29, 56, 97, 113, 211, 789]
```

结果完成正确，看代码不知很直观，我们还是画一个图来帮助我们理解上面的代码吧。



排序的时候先根据个位，然后十位，然后百位，等每个位上的都排序完了之后整个数组也就排序完成了。但是上面代码还不是很完美，因为当出现负数的时候上面代码就没法排序了，我们来想一下当出现负数的时候应该怎么办，话不多说，直接上代码，我们来看一下

```
private static void radixSort1(int[] array) {
    int digitCount = 19;//从-9到9最多19位数
    int maxCount = getBitCount(getMaxNumbit(array));
    int radix = 1;
    int[][] tempArray = new int[digitCount][array.length];
    for (int i = 0; i < maxCount; i++) {
        int[] count = new int[digitCount];
        for (int j = 0; j < array.length; j++) {
            int temp = ((array[j] / radix) % 10) + 9;
            tempArray[temp][count[temp]++] = array[j];
        }
        int index = 0;
        for (int j = 0; j < digitCount; j++) {
            if (count[j] == 0)
                continue;
            for (int k = 0; k < count[j]; k++) {
                array[index++] = tempArray[j][k];
            }
        }
        radix *= 10;
    }
}

private static int getMaxNumbit(int array[]) {
    int max = array[0];
    int min = array[0];
    for (int i = 1, length = array.length; i < length; i++) {
        if (array[i] > max) {
            max = array[i];
        } else if (array[i] < min) {
            min = array[i];
        }
    }
    return max < -min ? -min : max;
}

private static int getBitCount(int num) {
    int count = 1;
    int temp = num / 10;
    while (temp != 0) {
        count++;
        temp /= 10;
    }
    return count;
}
```

其中getMaxNumbit表示返回位数最多的值，也可以说是返回绝对值最大的值，我们看到上面temp加了9，所以如果出现负数就不会报错了，因为每一位只能是从-9~9,总共

19个数，当最小-9的时候，再加上9就变为0，下标从0开始，所以也不会出现数组越界异常。

我们找一组数据来测试一下

```
public static void main(String src[]) {
    int array[] = {2, 6, 9, -394, 3, 5, 1, -9, 7, -3, -1, -6, 8, 0, -121};
    radixSort1(array);
    System.out.println(Arrays.toString(array));
}
```

来看一下运行结果

```
[-394, -121, -9, -6, -3, -1, 0, 1, 2, 3, 5, 6, 7, 8, 9]
```

结果完全正确。

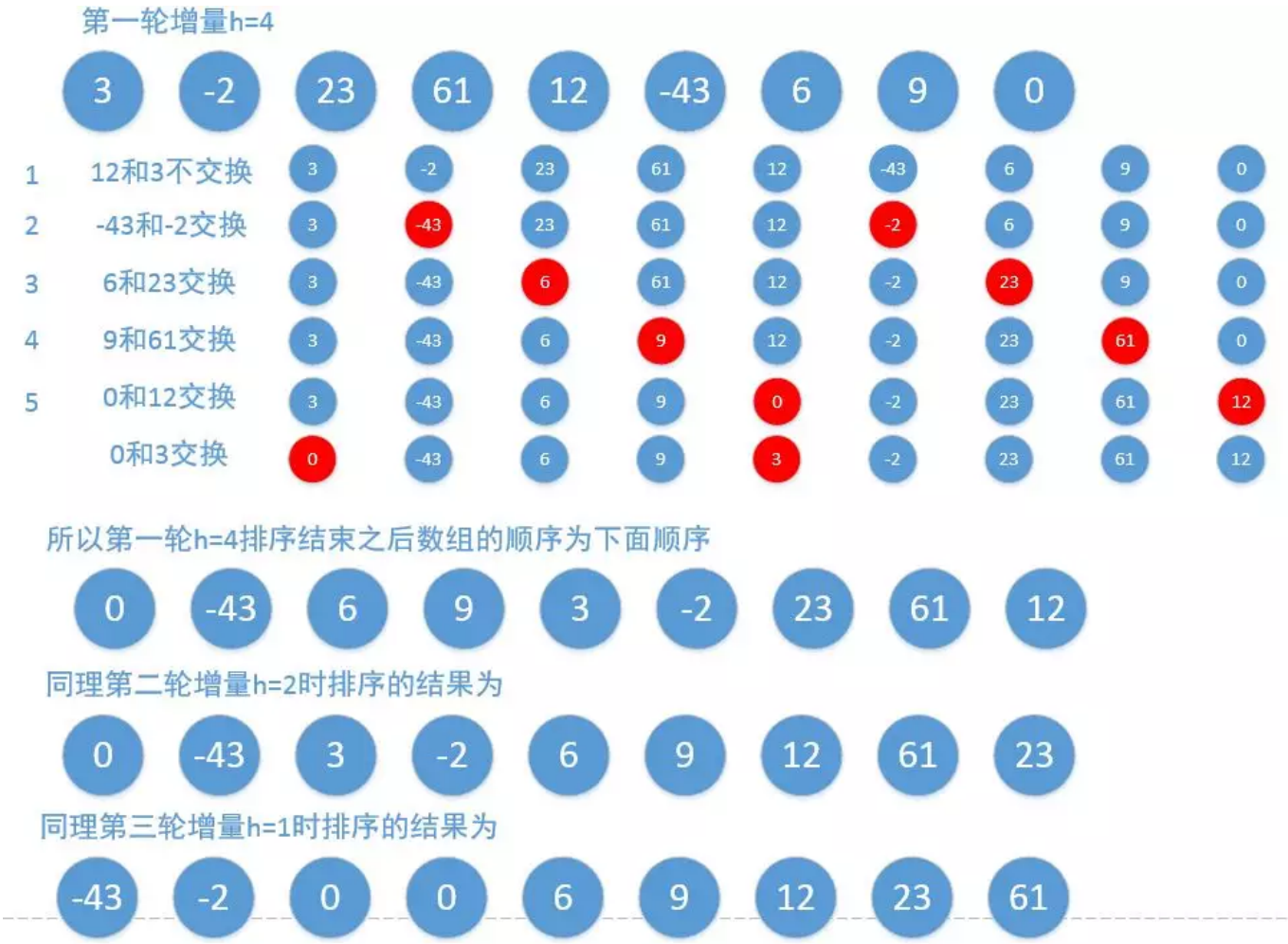
109，排序-希尔排序

原创 山大王wld 数据结构和算法 2018-12-04

希尔排序也成缩小增量排序，原理是将待排序列划分为若干组，每组都是不连续的，有间隔step，step可以自己定，但间隔step最后的值一定是1，也就说最后一步是前后两两比较。间隔为step的默认划分为一组，先在每一组内进行排序，以使整个序列基本有序，然后再减小间隔step的值，重新分组再排序……不断重复，直到间隔step小于1则停止。还是先看代码

```
public static void shellSort1(int[] array) {
    int length = array.length;
    int step = length >> 1;
    while (step >= 1) {
        for (int i = step; i < length; i++) {
            for (int j = i; j >= step; j -= step) {
                if (array[j] < array[j - step]) {
                    swap(array, j, j - step);
                } else {
                    //如果大于，则不用继续往前比较了，
                    // 因为前面的元素已经排好序
                    break;
                }
            }
        }
        step >>= 1;
    }
}

public static void swap(int[] A, int i, int j) {
    if (i != j) {
        A[i] ^= A[j];
        A[j] ^= A[i];
        A[i] ^= A[j];
    }
}
```

代码还是比较简单的，我们画个图来说明一下



上面的排序其实和冒泡排序很像，只不过冒泡排序是每次都是间隔为1相邻的两个之间进行比较，但希尔排序是间隔为step，还是有一定区别的，我们再看另一种写法

```
private static void ShellSort2(int[] arr) {
    int j;
    int len = arr.length;
    for (int step = len >> 1; step > 0; step >>= 1) {
        for (int i = step; i < len; i++) {
            int temp = arr[i];
            for (j = i; j >= step && temp < arr[j - step]; j -= step) {
                arr[j] = arr[j - step];
            }
            arr[j] = temp;
        }
    }
}
```

首先它是把待比较的变量保存到temp中，然后往前找，如果前面的比他大，就会把前面的值挪到当前位置，然后再往前找，如果还比当前大那么还挪，直到循环完为止，然后再把当前保存的temp值放到最前面挪动的那个值。这个挪动和前面介绍的插入排序的挪动有点类似，只不过插入排序的挪动是一个个往前比较（二分法插入例外），而这个挪动是每间隔step进行比较然后确定是否挪动。我们上面使用的间隔是数组长度的一半，这个间隔实际上是可以自己定的，但一定要保证间隔最后的一步是1即可，希尔排序中大家都比较认可的一种计算间隔的公式是step = step * 3 + 1;我们再来看一下代码

```java
public static void ShellSort3(int[] data) {
    int step = 1;
    while (step <= data.length / 3) {
        step = step * 3 + 1;
    }
    while (step > 0) {
        for (int i = step; i < data.length; i += step) {
            if (data[i] < data[i - step]) {
                int tmp = data[i];
                int j = i - step;
                while (j >= 0 && data[j] > tmp) {
                    data[j + step] = data[j];
                    j -= step;
                }
                data[j + step] = tmp;
            }
        }
        step = (step - 1) / 3;
    }
}
```

110，排序-计数排序

原创 山大王wld 数据结构和算法 2018-12-05

计数排序是一个非基于比较的排序算法，他首先要找到数组的最大值和最小值然后再根据最大值和最小值申请频率表，其实就是个数组，每个数在数组中出现的频率，我们这里暂且以桶来表示，每个桶对应一个数在原数组中出现的频率，如果一个桶为1就表示和这个桶对应的这个数在原数组中只出现一次，如果为2就表示出现两次……，我们直接看代码

```
public static void bucketSort1(int[] array) {
    int arrayLength = array.length;
    int max = array[0];
    int min = array[0];
    for (int i = 0; i < arrayLength; i++) {
        if (array[i] > max)
            max = array[i];
        else if (array[i] < min)
            min = array[i];
    }

    int bucketLength = max - min + 1;//桶的数量
    int[] tmp = new int[arrayLength];
    int[] buckets = new int[bucketLength];//桶
    for (int i = 0; i < arrayLength; i++) {
        buckets[array[i] - min]++;//落在某个桶里就加1
    }
    // 从小到大排序
    for (int i = 1; i < bucketLength; i++) {
        //后面桶对前面桶的累加
        buckets[i] = buckets[i] + buckets[i - 1];
    }
    // 从大到小排序
      for (int i = bucketLength - 1; i > 0; i--) {
          buckets[i - 1] = buckets[i] + buckets[i - 1];
      }
    //把原数组保存在临时数组中。
    System.arraycopy(array, i: 0, tmp, i1: 0, arrayLength);
    for (int k = 0; k < arrayLength; k++) {
        //根据每个数值在桶中的顺序重新存储
        array[--buckets[tmp[k] - min]] = tmp[k];
    }
}
```

原理就是把对应的数放到对应的桶里，如果桶里已经有值了，说明出现了两个重复的数，桶的值要累加。当每个桶对应值的个数确定以后就把前面的个数不断的累加然后放到后面的桶中，因为当一个数对应后面的桶的时候，他要确定前面有多少比他小的数，然后排序的时候再存放到原数组对应的位置。这里我们是以升序讲解的，比如，当一个数和一个桶对应的时候他需要知道前面究竟有几个比他小的他才能找准在原数组排序的位置，所以前面桶的值要累加到当前桶中，还是看一下图吧



我们随便找一组数据，测试一下运行结果

```
int array[] = {2, 6, 9, 3, 5, 1, -9, 7, -3, -1, -6, 8, 0};
bucketSort1(array);
System.out.println(Arrays.toString(array));
```

看一下运行结果

```
[-9, -6, -3, -1, 0, 1, 2, 3, 5, 6, 7, 8, 9]
```

结果正是我们所求的，这就是计数排序。

111，排序-位图排序

原创 山大王wld 数据结构和算法 2018-12-06

位图排序也称为bitmap排序，它主要用于海量数据去重和海量数据排序，假如说有10亿个int类型且全部不相同的数据，给1G内存让你排序，你怎么排，如果全部加载到内存中，相当于40亿个字节，大概约等于4G内存。所以全部加载到内存肯定不行，如果我们使用位图排序的话，我们用long类型表示，一个long占用8个字节也就是64位，所以如果我们使用位图排序的话只会占用约0.125G内存,内存占用大大减少。但位图排序有个缺点就是数据不能有重复的，如果有重复的会覆盖掉，这也是位图能在海量数据中去重的原因，我们看下位图排序的代码

```
private static int[] bitmapSort1(int[] array) {
    int max = getMaxNumbit1(array);
    int N = max / 64 + 1;
    long[] bitmap = new long[N];
    for (int i = 0; i < array.length; i++)
        bitmap[array[i] / 64] |= 1L << (array[i] % 64);
    int k = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < 64; j++) {
            if ((bitmap[i] & (1L << j)) != 0) {
                array[k++] = i * 64 + j;
            }
        }
    }
    if (k < array.length)
        return Arrays.copyOfRange(array, 0, k);
    return array;
}

private static int getMaxNumbit1(int array[]) {
    int max = array[0];
    for (int i = 1, length = array.length; i < length; i++) {
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max;
}
```

我们看到这是使用的是位表示，一个long类型占8个字节，但他可以表示64个数字，所以内存占用会大大减少。最后有个k < array.length的判断，是因为如果有重复的数据会覆盖掉重复的，导致数组变小。但这里面还有个问题就是不能有负数出现，如果出现负数会报异常，我们也可以改一下让负数也可以排序，看代码。

```
private static int[] bitmapSort2(int[] array) {
    int[] value = getMaxNumbit2(array);
    int N = (value[0] - value[1])  / 64 + 1;
    long[] bitmap = new long[N];
    for (int i = 0; i < array.length; i++)
        bitmap[(array[i] - value[1]) / 64] |= 1L << ((array[i] - value[1]) % 64);
    int k = 0;
    int[] temp = new int[array.length];
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < 64; j++) {
            if ((bitmap[i] & (1L << j)) != 0) {
                temp[k++] = i * 64 + j + value[1];
            }
        }
    }
    if (k < temp.length)
        return Arrays.copyOfRange(temp,  i: 0, k);
    return temp;
}

private static int[] getMaxNumbit2(int array[]) {
    int max = array[0];
    int min = array[0];
    for (int i = 1, length = array.length; i < length; i++) {
        if (array[i] > max) {
            max = array[i];
        } else if (array[i] < min) {
            min = array[i];
        }
    }
    return new int[]{max, min};
}
```

我们来找几行数据测试一下

```
int[] array2 = new int[20];
Random random = new Random();
for (int i = 0; i < array2.length; i++) {
    array2[i] = random.nextInt( i: 1000) - 500;
}
System.out.println("排序前：" + Arrays.toString(array2));
int temp2[] = bitmapSort2(array2);
System.out.println("排序后：" + Arrays.toString(temp2));
```

再来看一下运行的结果

```
排序前：[-196, 369, -67, 39, -244, 437, -160, -1, -340, 94, 132, 189, 188, -241, 117, 492, -402, -104, -461, 3]
排序后：[-461, -402, -340, -244, -241, -196, -160, -104, -67, -1, 3, 39, 94, 117, 132, 188, 189, 369, 437, 492]
```

112，排序-其他排序

原创 山大王wld 数据结构和算法 2018-12-07

常见的11种排序算法我们都已经分析完了，实际上排序算法并不止我们之前所介绍的那11种，只不过这些算法我们并不常见，有些感觉像闹着玩是的，比如BogoSort，我们来看一下

```
private static final Random random = new Random();

private static void bogoSort(int[] array) {
    while (!isOrder(array)) {
        for (int i = 0; i < array.length; i++) {
            int randomPosition = random.nextInt(array.length);
            int temp = array[i];
            array[i] = array[randomPosition];
            array[randomPosition] = temp;
        }
    }
}

private static boolean isOrder(int[] array) {
    for (int i = 0; i < array.length - 1; i++) {
        if (array[i] > array[i + 1]) return false;
    }
    return true;
}
```

这种排序基本上没人会用的，他的原理就是每次随机打乱顺序，然后在验证一下是否已经排序好了，如果没有排序好就继续打乱再验证，直到验证排序好为止。并且如果数组的长度超过10的时候，基本上就非常慢了。

除了这种排序以外，还有一种排序叫CocktailSort，简称鸡尾酒排序，应该说它是冒泡排序的一种，他和冒泡排序的不同之处在于，冒泡排序是往一个方向排的，而鸡尾酒排序是往两个方向排的，它是先从左往右把大的排到右边，然后再从右往左把小的排到左边，我们看下代码

```
public static void cocktailSort(int[] array) {
    for (int i = 0; i < array.length / 2; i++) {
        //把大的挪到右边
        for (int j = i; j < array.length - i - 1; j++) {
            if (array[j] > array[j + 1]) {
                swap(array, j, j + 1);
            }
        }
        //把小的挪到前面
        for (int j = array.length - 1 - (i + 1); j > i; j--) {
            if (array[j] < array[j - 1]) {
                swap(array, j, j - 1);
            }
        }
    }
}

public static void swap(int[] A, int i, int j) {
    A[i] = A[i] + A[j];
    A[j] = A[i] - A[j];
    A[i] = A[i] - A[j];
}
```

每次循环的时候都是把剩余最大的挪到右边，剩余最小的挪到左边。我们也可以换一种写法，分别用两个指针表示，一个指向前面一个指向后面，然后两个指针分别都往中间移，指针走过的地方都是已经排序好的，只要左边指针小于右边指针就一直走下去，看下代码

```
public static void cocktailSort1(int[] array) {
    int left = 0, right = array.length - 1;
    while (left < right) {
        for (int i = left; i < right; i++)
            if (array[i] > array[i + 1]) {
                swap(array, i, i + 1);
            }
        right--;
        for (int i = right; i > left; i--)
            if (array[i - 1] > array[i]) {
                swap(array, i, i - 1);
            }
        left++;
    }
}
```

除了上面两种排序之外还有一种叫鸽巢排序，他和计数排序有点像，我们来看一下代码

```java
private static void pigeonholeSort(int[] array) {
    int max = max(array);
    int bucket[] = new int[max + 1];
    for (int i = 0; i < array.length; ++i)
        bucket[array[i]]++;
    int j = 0;
    for (int i = 0; i <= max; ++i)
        for (int k = 0; k < bucket[i]; ++k)
            array[j++] = i;
}

private static int max(int[] array) {
    int max = array[0];
    for (int i = 0; i < array.length; i++) {
        if (array[i] > max)
            max = array[i];
    }
    return max;
}
```

除了上面介绍的3种排序之外，还有其他排序，基本上也都用不到，并且也很少见到，这里就不在一一介绍

201，查找-顺序查找

原创 山大王wld 数据结构和算法 2019-05-09

查找算法中顺序查找算是最简单的了，无论是有序的还是无序的都可以，也不需要排序，只需要一个个对比即可，但其实效率很低。我们来看下代码

```
public static int search1(int[] a, int key) {
  for (int i = 0, length = a.length; i < length; i++) {
    if (a[i] == key)
      return i;
  }
  return -1;
}
```

如果找到就返回查找的数所在数组中的下标，如果没找到就返回-1。还有说上面的代码可以优化，使用一个哨兵，免去了每次都要越界的判断，但通过实际测试运行效率并没有提高，无论测试的数据是多还是少运行的时间都差不多，我们来看下代码。

```
public static int search2(int[] a, int key) {
  int index = a.length - 1;
  if (key == a[index])
    return index;
  a[index] = key;
  int i = 0;
  while (a[i++] != key) ;
  return i == index + 1 ? -1 : i - 1;
}
```

顺序查找是最简单的一种查找算法，对数据的要求也很随意，不需要排序即可查找。后面会介绍二分法查找，插值查找和斐波那契查找都是基于已经排序过的数据。

202，查找-二分法查找

原创 山大王wld 数据结构和算法 2019-05-10

二分法查找适用于大的数据，但前提条件是数据必须是有序的，他的原理是先和中间的比较，如果等于就直接返回，如果小于就在前半部分继续使用二分法进行查找，如果大于则在后半部分继续使用二分法进行查找.....我们来看下代码

```
public static int binarySearch(int[] array, int value) {
  int low = 0;
  int high = array.length - 1;
  while (low <= high) {
    int middle = low + ((high - low) >> 1);
    // int middle = (low + high) >> 1;
    if (value == array[middle])
      return middle;
    if (value > array[middle])
      low = middle + 1;
    else
      high = middle - 1;
  }
  return -1;
}
```

上面的求middle的方法最好不要使用注释的部分，因为当数组比较大的时候，low+high可能会溢出。上面是通过while循环的方式，我们也可以不使用循环，使用递归的方式来求，看下代码

```
public static int binarySearch(int[] array, int value) {
  int low = 0;
  int high = array.length - 1;
  return searchmy(array, low, high, value);
}

private static int searchmy(int array[], int low, int high, int value) {
  if (low > high)
    return -1;
  int mid = low + ((high - low) >> 1);
  if (value == array[mid])
    return mid;
  if (value < array[mid])
    return searchmy(array, low, mid - 1, value);
  return searchmy(array, mid + 1, high, value);
}
```

203，查找-插值查找

原创 山大王wld 数据结构和算法 2019-05-13

二分法查然效率很高，但我们为什么要和中间的值进行比较，如果我们和数组1/4或者3/4部分的值进行比较可不可以呢，对于一个要查找的数我们不知道他大概在数组的什么位置的时候我们可以使用二分法进行查找。但如果我们知道要查找的值大概在数组的最前面或最后面的时候使用二分法查找显然是不明智的。比如我们查字典的时候如果是a或者b开头的我们一般会在前面找，如果是y或者z开头的我们一般偏向于往后面找，这个时候如果我们使用二分法从中间开始找显然是不合适的。之前二分法查找的时候我们比较的是中间值，mid=low+1/2*(high-low);但插值查找的时候我们比较的不是中间值，是mid=low+(key-a[low])/(a[high]-a[low])*(high-low)，我们来看下插值查找的代码。

```
public static int insertSearch(int[] array, int key) {
  return search(array, key, 0, array.length - 1);
}

private static int search(int[] array, int key, int left, int right) {
  while (left <= right) {
    if (array[right] == array[left]) {
      if (array[right] == key)
        return right;
      else return -1;
    }
    int middle = left + ((key - array[left]) / (array[right] - array[left])) * (right - left);
    if (array[middle] == key) {
      return middle;
    }
    if (key < array[middle]) {
      right = middle - 1;
    } else {
      left = middle + 1;
    }
  }
  return -1;
}
```

他和二分法查找代码很相似，只不过计算middle的方式不一样。再来看一下递归的版本

```
public static int insertSearch(int[] array, int key) {
  return search2(array, key, 0, array.length - 1);
}

private static int search2(int array[], int key, int left, int right) {
  if (left > right)
    return -1;
  if (array[right] == array[left]) {
    if (array[right] == key)
      return right;
    else return -1;
  }
  int mid = left + (key - array[left]) / (array[right] - array[left]) * (right - left);
  if (array[mid] == key)
    return mid;
  if (array[mid] > key)
    return search2(array, key, left, mid - 1);
  return search2(array, key, mid + 1, right);
}
```

204，查找-斐波那契查找

原创 山大王wld 数据结构和算法 2019-05-14

斐波那契数列我们都知道{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 }，前后的比值会越来越接近0.618，也就是黄金分割点。0.618也被公认为最具有审美意义的比例数字。斐波那契查找原理其实和二分法查找原理差不多，只不过计算中间值mid的方式不同，还有一点就是斐波那契查找的数组长度必须是f（k）-1，这样我们就可以把斐波那契数列进行划分f(k)-1=f(k-1)+f(k-2)-1=[f(k-1)-1]+1+[f(k-2)-1]；然后前面部分和后面部分都还可以继续进行划分。但实际上我们要查找的数组长度不可能都是f（k）-1，所以我们要补齐最后的部分，让数组的长度等于f（k）-1，让数组的最后一位数字把后面铺满。比如我们查找的数组长度是21，而f（8）-1=21-1=20；小于21，所以f（8）-1是不行的，我们需要把数组长度变为f（9）-1=34-1=33，后面多余的我们都用原数组最后的那个值填充。我们来看下代码

```
1   public static int fibonacciSearch(int[] array, int key) {
2     if (array == null || array.length == 0)
3       return -1;
4     int length = array.length;
5     int k = 0;
6     while (length > fibonacci(k) - 1 || fibonacci(k) - 1 < 5) {
7       k++;
8     }
9     int[] fb = makeFbArray(fibonacci(k) - 1);
10    int[] temp = Arrays.copyOf(array, fb[k] - 1);
11    for (int i = length; i < temp.length; i++) {
12      temp[i] = array[length - 1];//用原数组最后的值填充
13    }
14    int low = 0;
15    int hight = length - 1;
16    while (low <= hight) {
17      int middle = low + fb[k - 1] - 1;
18      if (temp[middle] > key) {//要查找的值在前半部分
19        hight = middle - 1;
20        k = k - 1;
21      } else if (temp[middle] < key) {//要查找的值在后半部分
22        low = middle + 1;
23        k = k - 2;
24      } else {
25        if (middle <= hight) {
26          return middle;
27        } else {
28          return hight;
29        }
30      }
31    }
32    return -1;
33  }
34
35  private static int fibonacci(int n) {
36    if (n == 0 || n == 1)
37      return n;
38    return fibonacci(n - 1) + fibonacci(n - 2);
39  }
40
41  public static int[] makeFbArray(int length) {
42    int[] array = new int[length];
43    array[0] = 0;
44    array[1] = 1;
45    for (int i = 2; i < length; i++)
```

```
        array[i] = array[i - 1] + array[i - 2];
    return array;
}
```

其实经过测试发现斐波那契查找效率并没有那么高，我们再来看一下斐波那契查找的递归实现

```
public static int search(int[] array, int value) {
    if (array == null || array.length == 0) return -1;
    int length = array.length;
    int k = 0;
    while (length > fibonacci(k) - 1 || fibonacci(k) - 1 < 5) {
        k++;
    }
    int[] fb = makeFbArray(fibonacci(k) - 1);
    int[] temp = Arrays.copyOf(array, fb[k] - 1);
    for (int i = length; i < temp.length; i++) {
        temp[i] = array[length - 1];//用原数组最后的值填充
    }
    return fibonacciSearch(temp, fb, value, 0, length - 1, k);
}

public static int fibonacciSearch(int[] array, int[] fb, int value, int low, int hight, int k) {
    if (value < array[low] || value > array[hight] || low > hight) return -1;
    int middle = low + fb[k - 1] - 1;
    if (value < array[middle]) {
        return fibonacciSearch(array, fb, value, low, middle - 1, k - 1);
    } else if (value > array[middle]) {
        return fibonacciSearch(array, fb, value, middle + 1, hight, k - 2);
    } else {
        if (middle <= hight) {
            return middle;
        } else {
            return hight;
        }
    }
}

private static int fibonacci(int n) {
    if (n == 0 || n == 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

public static int[] makeFbArray(int length) {
    int[] array = new int[length];
    array[0] = 0;
    array[1] = 1;
    for (int i = 2; i < length; i++) array[i] = array[i - 1] + array[i - 2];
    return array;
}
```

上面的两个斐波那契查找有一个缺点，就是数组长度必须是斐波那契数减1，否则数组就要增大，浪费空间。我们可以优化一下，不需要扩大数组的长度，当查找的位置大于原数组的长度的时候，我们让比较的值等于数组的最后一个元素即可。

```
public static int fibonacciSearch1(int[] array, int key) {
    if (array == null || array.length == 0)
        return -1;
    int length = array.length;
    int k = 0;
    while (length > fibonacci(k) - 1 || fibonacci(k) - 1 < 5) {
        k++;
    }
    int[] fb = makeFbArray(fibonacci(k) - 1);
    int low = 0;
    int hight = length - 1;
    while (low <= hight) {
        int middle = low + fb[k - 1] - 1;
```

```
14        int midvalue;
15        if (middle >= length)
16          midvalue = array[length - 1];
17        else
18          midvalue = array[middle];
19        if (midvalue > key) {//要查找的值在前半部分
20          hight = middle - 1;
21          k = k - 1;
22        } else if (midvalue < key) {//要查找的值在后半部分
23          low = middle + 1;
24          k = k - 2;
25        } else {
26          if (middle <= hight) {
27            return middle;
28          } else {
29            return hight;
30          }
31        }
32      }
33      return -1;
34    }
```

205，查找-分块查找

原创 山大王wld 数据结构和算法 2019-05-15

分块查找是折半查找和顺序查找的一种改进方法，分块查找由于只要求索引表是有序的，对块内节点没有排序要求，因此特别适合于节点动态变化的情况。
分块查找要求把一个数据分为若干块，每一块里面的元素可以是无序的，但是块与块之间的元素需要是有序的。即第1块中任一元素的关键字都必须小于第2块中任一元素的关键字；而第2块中任一元素又都必须小于第3块中的任一元素，……。我们来看下代码

```
public static void main(String args[]) {
  int index[] = {22, 48, 86};
  int st2[] = {22, 12, 13, 8, 9, 20, 33, 42, 44, 38, 24, 48, 60, 58, 74, 49, 86, 53};
  for (int i = 0; i < 100; i++) {
    System.out.println(blocksearch(index, st2, i, 6));
  }
}

//index每个元素代表的是每块的最大值，
// array代表的是要查找的数组，
// key代表要查找的元素，m代表每个块大小
public static int blocksearch(int[] index, int[] array, int key, int m) {
  int i = search(index, key);
  if (i < 0)
    return -1;
  for (int j = m * i, length = j + m; j < length; j++) {
    if (array[j] == key)
      return j;
  }
  return -1;
}

private static int search(int[] index, int key) {
  int start = 0;
  int end = index.length - 1;
  if (key > index[end])
    return -1;
  while (start <= end) {
    int mid = start + ((end - start) >> 1);
    if (index[mid] >= key) {
      end = mid - 1;
    } else {
      start = mid + 1;
    }
  }
  return start;
}
```

search方法表示查找的在哪个块中，确定某个块之后，然后再在那个块中进行查找。

206，查找-哈希查找

原创 山大王wld 数据结构和算法 2019-05-16

说到哈希我们很容易想到HashMap和HashSet，其中HashSet封装的就是HashMap，HashMap的原理很简单，就是数组加链表的形式。如果有做Android开发的，可能比较熟悉，Android中有个类ArrayMap，他就是以纯数组的形式进行存储的，其中Hash值是排序的，相同的Hash值会挨着，存放数据的数组是hash数组的两倍，因为存放数据的key和value是成对出现的，查找的时候先找到hash值的位置，然后在数据数组中相对应位置2倍的地方进行查找，如果没有再往前或往后找，这个原理很简单。hash查找首先要构造hash值，hash值的构造方式非常多，什么平均取中法，折叠法，平方法，求模法，等等等……。然后就是碰撞问题，HashMap解决碰撞问题是以链表的形式存在，ArrayMap出现碰撞时以数组的形式存在，并且出现碰撞都会挨着的。我们这里来写一个简单的查找，严格意义上来说不能说是查找，因为他不能确定要查找的元素在原数组中发位置，只能确定有没有，我们这里解决碰撞和上面两种都不一样，是以数组形式但是不挨着。

```
1   public static void main(String args[]) {
2     int array[] = {2, 8, 3, 7, 1, 21, 36, 17};
3     int hashLength = 30;
4     int hash[] = new int[hashLength];
5     for (int i = 0; i < array.length; i++) {
6       insertHash(hash, array[i]);
7     }
8     for (int i = 0; i < 50; i++) {
9       int index = searchHash(hash, hashLength, i);
10      if (index != -1)
11        System.out.println("原数组中有" + i + "这个元素");
12    }
13  }
14
15  public static int searchHash(int[] hash, int hashLength, int key) {
16    int index = key % hashLength;
17    while (hash[index] != 0 && hash[index] != key) {
18      index = (++index) % hashLength;
19    }
20    if (hash[index] == 0)
21      return -1;
22    return index;
23  }
24
25  public static void insertHash(int[] hash, int data) {
26    int index = data % hash.length;
27    while (hash[index] != 0) {
28      index = (++index) % hash.length;
29    }
30    hash[index] = data;
31  }
```

这里hashLength的长度不能小于原数组的长度，否则会出现死循环。其中searchHash函数返回的是数据在hash数组中存放的下标，不是原数组的下标。当然上面代码也有缺点，就是原数组中不能包含0，因为hash数组没赋初值，默认值也是0，所以解决方式就是给hash数组中每一个元素都赋一个原数组中不可能出现的初值，比如Integer.MIN_VALUE。下面我们来看下运行结果

原数组中有1这个元素
原数组中有2这个元素
原数组中有3这个元素
原数组中有7这个元素
原数组中有8这个元素
原数组中有17这个元素
原数组中有21这个元素
原数组中有36这个元素

Process finished with exit code 0

207，查找-其他查找

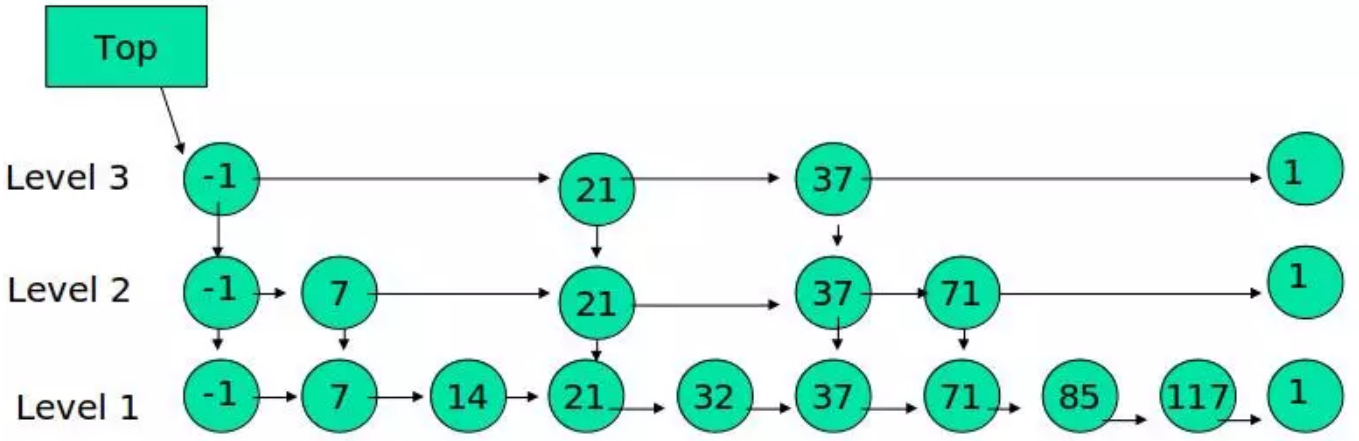
原创 山大王wld 数据结构和算法 2019-05-17

一，二叉树查找

除了前面介绍的几个查找算法以外，还有一种叫二叉树查找，二叉树查找比较简单，我们知道二叉树的节点有两种常见的遍历方式，一种是BFS（广度优先搜索），一种是DFS（深度优先搜索）。如果二叉树是排序好的，我们使用DFS，查找的时候先找根节点，如果找到就返回，如果没找到，判断是大于根节点还是小于根节点，如果小于根节点就在左半部分找，如果大于根节点就在右半部分找……一直递归下去。如果二叉树没有排序我们也可以一层一层的找，使用BFS，当然这个比较费劲，但也没办法，因为没有排过序，也只能这样干了。

二，跳表查找

我们知道链表添加和删除比较方便，但查找不是很方便，对于排过序的单向链表来说如果我们要查找，每次我们都要从头开始查，如果链表比较长的话，这样效率很慢。但使用跳表效率可以大大改善，虽然跳表每次也是从头开始查，但他每次可以跳很多步，不像链表每次只能走一步，java中有ConcurrentSkipListMap 这个类，是基于跳表的，代码就不在贴了，大家有兴趣可以自己看，关于跳表在网上找了一张图，可以参照下



三，其他查找

当然还有一些其他的比如2-3树，红黑树，B树、B+树。其中2-3树是最简单的B-树（或-树）结构，其每个非叶节点都有两个或三个子节点。红黑树也算是个平衡二叉树，他会根据插入的节点进行左旋或右旋来达到平衡。B树也称为B-树，首先把根结点取来，在根结点所包含的关键字K1,…,Kn查找给定的关键字（可用顺序查找或二分查找法），若找到等于给定值的关键字，则查找成功；否则，一定可以确定要查找的关键字在Ki与Ki+1之间，Pi为指向子树根节点的指针，此时取指针Pi所指的结点继续查找，直至找到，或指针Pi为空时查找失败。B+ 树元素自底向上插入，这与二叉树恰好相反。

601，下一个排列

原创 博哥 数据结构和算法 今天

问题描述

来源：LeetCode第31题
难度：中等

实现获取下一个排列的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列（即，组合出下一个更大的整数）。

如果不存在下一个更大的排列，则将数字重新排列成最小的排列（即升序排列）。

必须原地修改，只允许使用额外常数空间。

示例 1：

> **输入**：nums = [1,2,3]
> **输出**：[1,3,2]

示例 2：

> **输入**：nums = [3,2,1]
> **输出**：[1,2,3]

示例 3：

> **输入**：nums = [1,1,5]
> **输出**：[1,5,1]

示例 4：

> **输入**：nums = [1]
> **输出**：[1]

提示：

- 1 <= nums.length <= 100
- 0 <= nums[i] <= 100

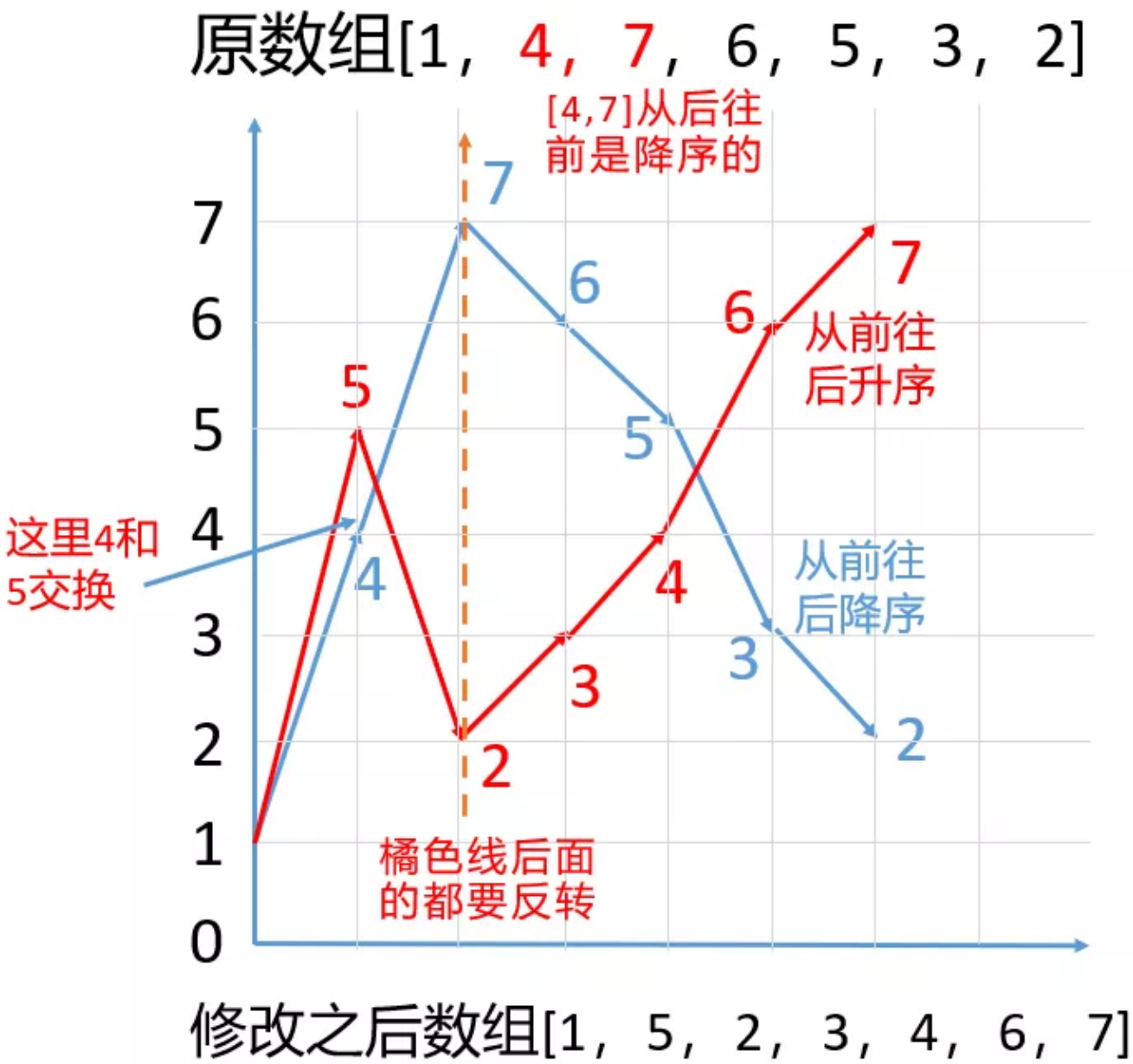
问题分析

这题说的是找出数字序列重新排列成字典序中下一个更大的排列。举个例子，比如数字213的下一个排列是231，231的下一个排列是312。

那么这题的规律该怎么找呢，我们来看这样一组数字
[7,5,4,3,2]
这些数字从后往前都是升序的，无论怎么调换位置都不可能获得更大的排列。

再来看一组数字
[1,4,7,6,5,3,2]
从后往前看2→3→5→6→7是升序的，但7→4是降序的，我们只需要把4和7交换一下就可以获得比原来更大的排列。但这里要等一下，题中要求的是找出比原来大的最小的排列。交换4和7虽然比原来大，但不是最小的。实际上用5和4交换要比7和4交换更小。

所以这里当我们从后往前找到第一个降序的数字之后（比如上面的4），我们还要从后往前找到第一个比降序数字大的值（比如上面的5），然后这两个数字交换（比如上面的4和5交换）。交换完之后（比如上面的交换之后是[1,5,7,6,4,3,2]），这个排列肯定是比原来的大，因为5比4大，我们只需要让5后面的排列数字最小即可。因为5后面的数字[7,6,4,3,2]从后往前是升序的，我们只需要把他反转即可，所以[1,4,7,6,5,3,2]的下一个排列是[1,5,2,3,4,6,7]，上面很啰嗦，画个图来加深一下理解。



最后再来看下代码

```
public void nextPermutation(int[] nums) {
    int left = nums.length - 2;
    //两两比较，从后面往前找第一个降序的
    while (left >= 0 && nums[left] >= nums[left + 1])
        left--;
    //如果数组nums中的元素都是倒叙，那么left就等于-1
    if (left >= 0) {
        int right = nums.length - 1;
        //从后面查找第一个比nums[left]大的值
        while (nums[right] <= nums[left])
            right--;
        swap(nums, left, right);
    }
    //反转后面升序的数字
    reverse(nums, left + 1, nums.length - 1);
}

//反转子数组[left，right]中的元素
private void reverse(int[] nums, int left, int right) {
```

```
    while (left < right)
        swap(nums, left++, right--);
}

//交换数组中的两个数字
private void swap(int[] nums, int left, int right) {
    int tmp = nums[left];
    nums[left] = nums[right];
    nums[right] = tmp;
}
```

时间复杂度：O（N）。

空间复杂度：O（1）。

往期推荐

- 599，统计全 1 子矩形
- 598，动态规划解目标和
- 593，经典回溯算法题-全排列
- 557，动态规划解戳气球

599，统计全 1 子矩形

原创 博哥 数据结构和算法 5天前

问题描述

来源：LeetCode第1504题
难度：中等

给你一个只包含0和1的rows*columns矩阵mat，请你返回有多少个子矩形的元素全部都是1。

示例 1：

> **输入**：mat = [
> [1,0,1],
> [1,1,0],
> [1,1,0]]
> **输出**：13
> **解释**：
> 有 6 个 1x1 的矩形。
> 有 2 个 1x2 的矩形。
> 有 3 个 2x1 的矩形。
> 有 1 个 2x2 的矩形。
> 有 1 个 3x1 的矩形。
> 矩形数目总共 = 6 + 2 + 3 + 1 + 1 = 13 。

示例 2：

> **输入**：mat = [
> [0,1,1,0],
> [0,1,1,1],
> [1,1,1,0]]
> **输出**：24
> **解释**：
> 有 8 个 1x1 的子矩形。
> 有 5 个 1x2 的子矩形。
> 有 2 个 1x3 的子矩形。
> 有 4 个 2x1 的子矩形。
> 有 2 个 2x2 的子矩形。
> 有 2 个 3x1 的子矩形。

> 有 1 个 3x2 的子矩形。
> 矩形数目总共 = 8 + 5 + 2 + 4 + 2 + 2 + 1 = 24 。

示例 3：

> **输入**：mat = [[1,1,1,1,1,1]]
> **输出**：21

示例 4：

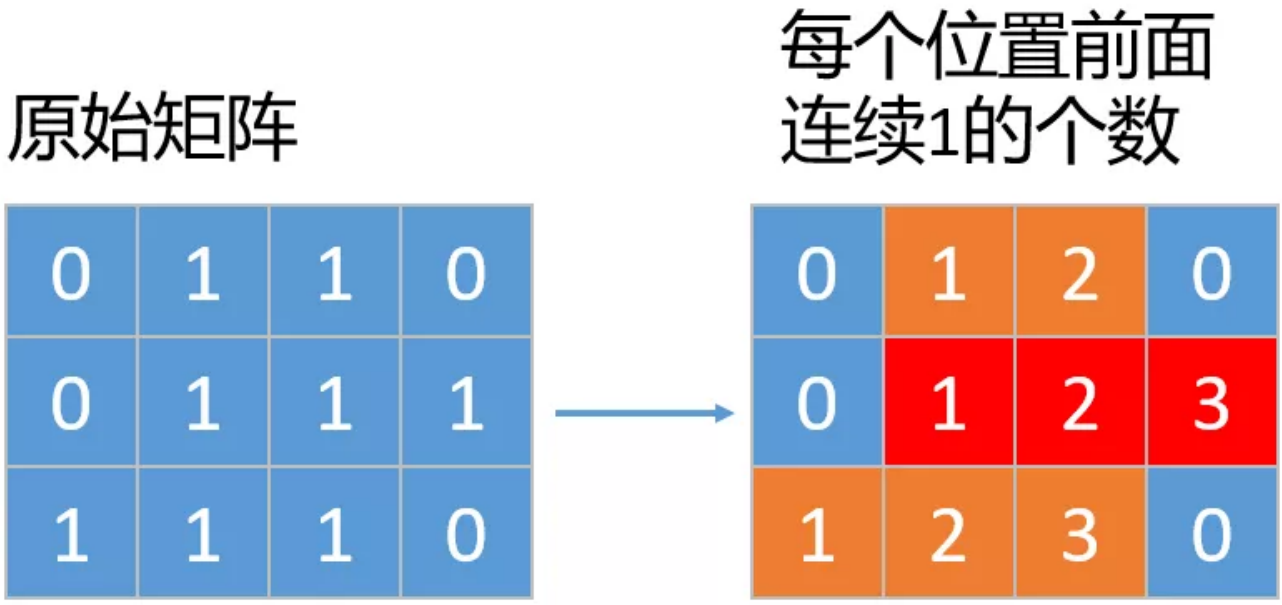
> **输入**：mat = [
> [1,0,1],
> [0,1,0],
> [1,0,1]]
> **输出**：5

提示：

- 1 <= rows <= 150
- 1 <= columns <= 150
- 0 <= mat[i][j] <= 1

问题分析

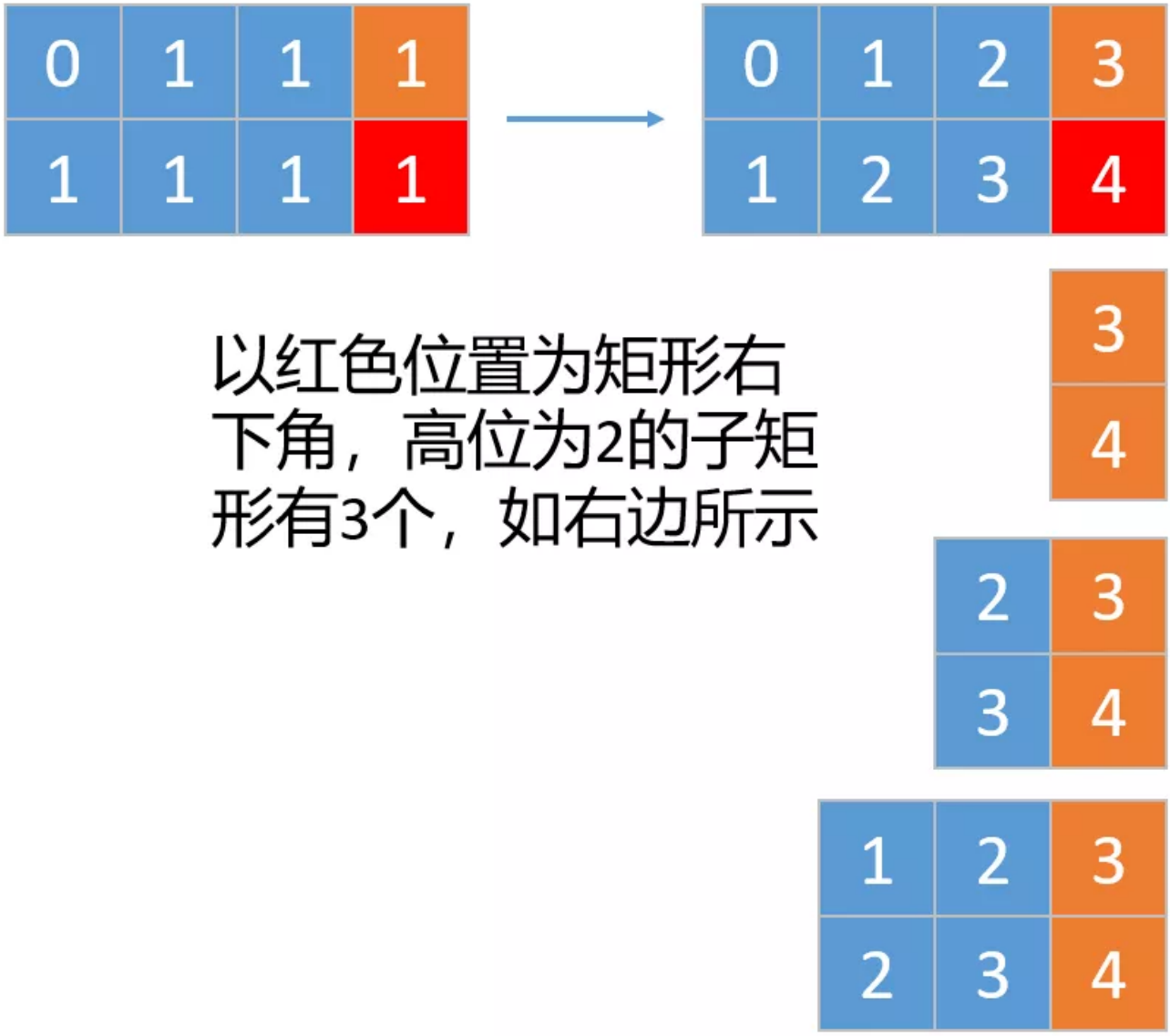
题中说了矩阵中的数字只有0和1，我们可以申请一个二维数组temp，其中temp[i][j]表示坐标（i，j）左边连续1的个数。很明显如果坐标（i，j）位置是0，那么temp[i][j]=0。如果坐标（i，j）位置是1，那么temp[i][j]=temp[i][j-1]+1，这里要注意边界条件的判断。我们以示例2为例画个图来看下



假设是一维数组，也就是说矩形的高度是1，我们可以发现，以当前位置为矩形的最右边总共有**temp[i][j]**个子矩形，如下图所示。

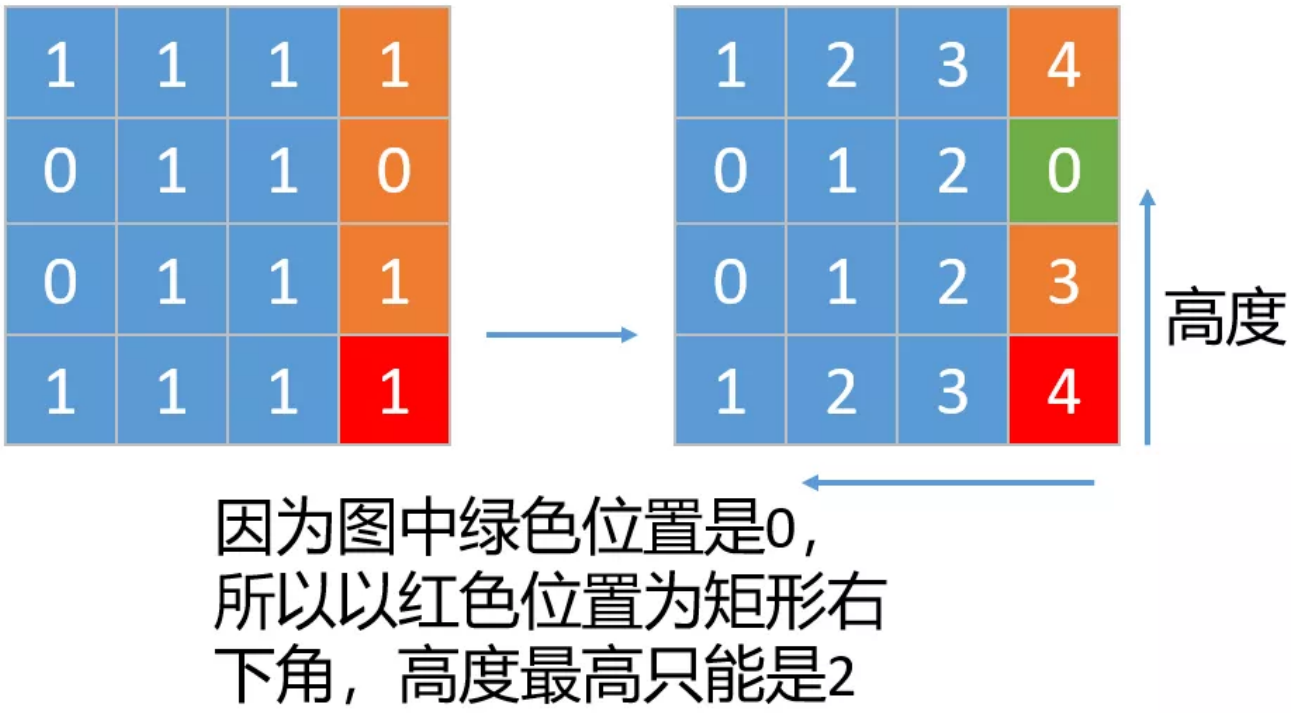


如果矩形的高度都是1，我们只需要把二维数组temp中的所有值相加即可，但实际上矩形的高度不可能都是1。如果当前位置左边连续1的个数不为0，我们计算完高度为1的子矩形之后还要计算高度为2，3……的子矩形，直到不能构成矩形为止，如下图所示。



上面高度为2的子矩形个数是怎么确定的呢，其实就是以红色位置往上找左边连续1的最小值，由图中我们可以看到3比4小，所以高度为2的子矩形宽度最大是3，也就是高度为2的子矩形个数是3。

那么什么情况下构不成矩形呢，就是遇到了0，如下图所示，即使0上面还有更大的数字，他也不可能和下面红色位置构成矩形。



搞懂了上面的分析过程，代码就很容易写了。来看下代码

```
public int numSubmat(int[][] mat) {
    int m = mat.length;//矩阵的宽
    int n = mat[0].length;//矩阵的高
    //数组temp[i][j]表示坐标(i,j)左边连续1的个数
    int[][] temp = new int[m][n];
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (j == 0) {
                //如果是第一列，连续1的个数就是当前数字
                temp[i][j] = mat[i][j];
            } else if (mat[i][j] == 1) {
                //如果不是第1列，并且当前位置是1，
                // 那么左边连续1的个数就
                //是temp[i][j - 1] + 1
                temp[i][j] = temp[i][j - 1] + 1;
            } else {
                //当前位置是0，连续1的个数就是0
                temp[i][j] = 0;
            }
        }
    }
    //记录矩形的个数
    int res = 0;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            //temp[i][j]表示的是左边连续1的个数，我们
            //可以把它看做是矩形的宽度
            int width = temp[i][j];
            //往上找矩形的高度，直到遇到0为止
            for (int k = i; k >= 0; k--) {
                //最小的宽度作为矩形的宽度
                width = Math.min(width, temp[k][j]);
                res += width;
                //如果宽度为0，就不能往上查找
                if (width == 0)
```

```
                    break;
                }
            }
        }
        return res;
    }
```

时间复杂度：O（n*m^2）,最上面计算temp值的时间复杂度是O（m*n），下面统计矩形个数的时间复杂度是O（m*n*m），最差情况下矩阵全是1，k每次都会从i到0。所以整体时间复杂度是O（n*m^2）。

空间复杂度：O（m*n）,使用一个m*n的二维数组

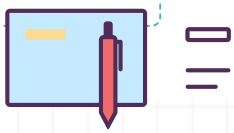
往期推荐

- 587，最大的以1为边界的正方形
- 530，动态规划解最大正方形
- 520，回溯算法解火柴拼正方形
- 379，柱状图中最大的矩形（难）

585，最大升序子数组和

原创 博哥 数据结构和算法 1周前

> A real loser is someone so afraid of not wining, they don't even try.
>
> 真正的失败者是那些害怕失败不敢尝试的人。



问题描述

来源：LeetCode第1800题
难度：简单

给你一个正整数组成的数组nums，返回nums中一个升序子数组的最大可能元素和。

子数组是数组中的一个连续数字序列。

已知子数组[numsl,numsl+1,...,numsr-1,numsr]，若对所有i（l<=i<r），numsi<numsi+1都成立，则称这一子数组为升序子数组。注意，大小为1的子数组也视作升序子数组。

示例 1：

> **输入**：nums = [10,20,30,5,10,50]
> **输出**：65
> **解释**：[5,10,50] 是元素和最大的升序子数组，最大元素和为 65 。

示例 2：

> **输入**：nums = [10,20,30,40,50]
> **输出**：150
> **解释**：[10,20,30,40,50] 是元素和最大的升序子数组，最大元素和为 150 。

示例 3：

> **输入**：nums = [12,17,15,13,10,11,12]
> **输出**：33
> **解释**：[10,11,12] 是元素和最大的升序子数组，最大元素和为 33 。

示例 4：

> **输入**：nums = [100,10,1]
> **输出**：100

提示：

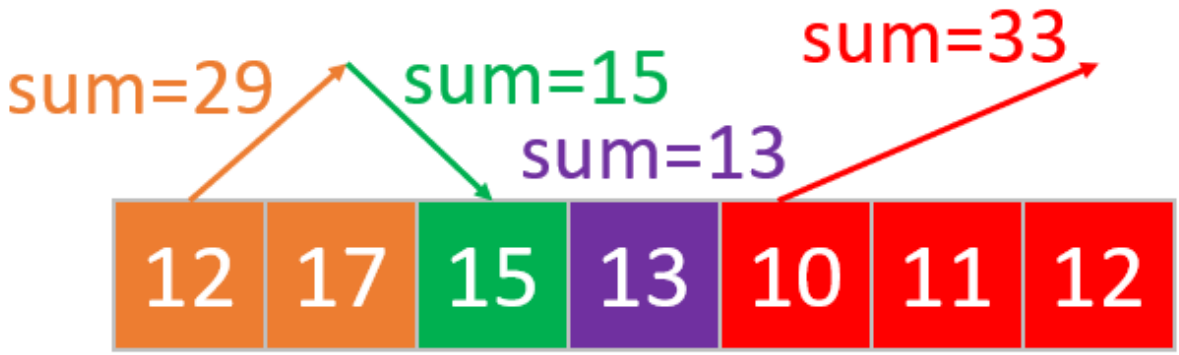
- 1<=nums.length<=100
- 1<=nums[i]<=100

问题分析

这题可以参照《486，动态规划解最大子序和》使用动态规划来解决，但实际上不需要dp数组也可以解决。这题让求的是升序子数组的最大和，解题思路如下

- 使用一个变量sum记录子数组的和
- 当递增的时候，sum就累加
- 当递减的时候，把当前元素的值赋值给sum，也就是重新开始统计
- 使用一个变量max记录遍历过的升序子数组的最大和

如下图所示



原理比较简单，来看下代码

```
public int maxAscendingSum(int[] nums) {
  int sum = nums[0];
  int max = sum;
  for (int i = 1; i < nums.length; i++) {
    //如果是升序的，就一直累加
    if (nums[i] > nums[i - 1]) {
      sum += nums[i];
```

```
    } else {
      //如果是降序，sum就重新赋值
      sum = nums[i];
    }
    //记录最大的连续子数组的和
    max = Math.max(max, sum);
  }
  return max;
}
```

时间复杂度：O(n)

空间复杂度：O(1)

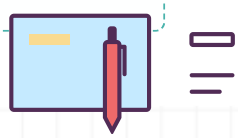
往期推荐

- 577，数组中的最长连续子序列
- 539，双指针解删除有序数组中的重复项
- 527，两个数组的交集 II
- 509，数组中的第K个最大元素

584，前缀和解和为K的子数组

原创 博哥 数据结构和算法 1周前

> It hurts to remember, but it would be worse to forget.
>
> 铭记虽痛苦，但遗忘更糟糕。



问题描述

来源：LeetCode第560题
难度：中等

给定一个整数数组和一个整数k，你需要找到该数组中和为k的连续的子数组的个数。

示例 1：

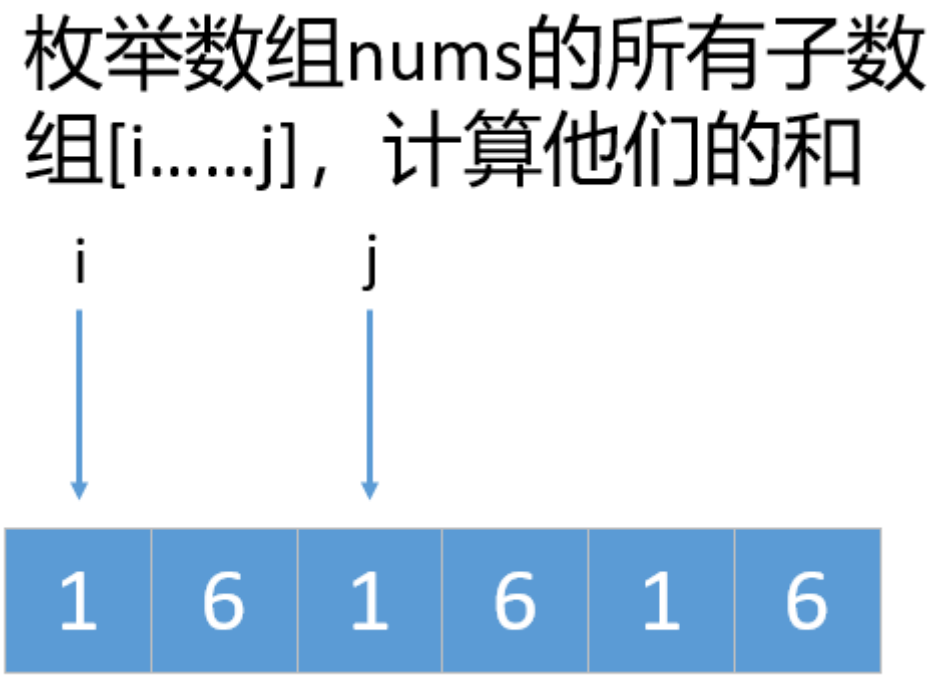
```
输入:nums = [1,1,1], k = 2
输出: 2 , [1,1] 与 [1,1] 为两种不同的情况。
```

说明：

- 数组的长度为[1,20,000]。
- 数组中元素的范围是[-1000,1000]，且整数k的范围是[-1e7, 1e7]。

暴力求解

暴力求解是最容易想到的，枚举数组nuns的所有子数组，然后统计所有子数组和等于k的个数



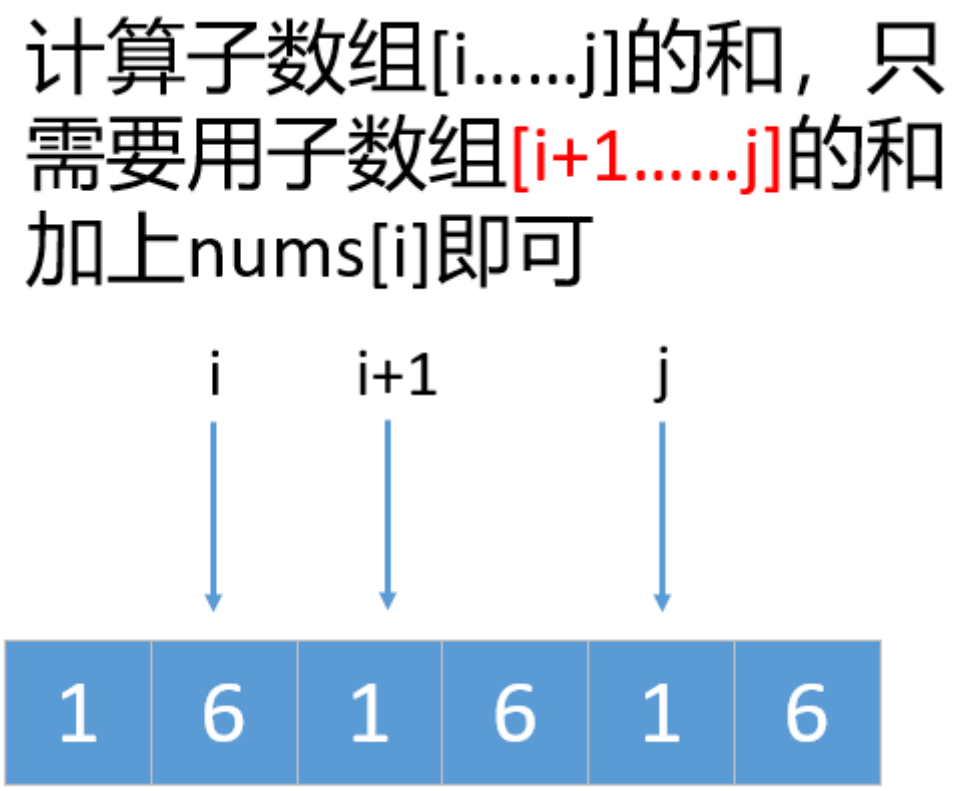
来看下代码

```
1   public int subarraySum(int[] nums, int k) {
2     int count = 0;
3     for (int j = 0; j < nums.length; j++) {
4       for (int i = 0; i <= j; i++) {
5         int sum = 0;
6         //计算子数组[i……j]中所有数字的和
7         for (int m = i; m <= j; m++) {
8           sum += nums[m];
9         }
10        //如果子数组[i……j]中所有数字
11        //的和等于k，count加1
12        if (sum == k)
13          count++;
14      }
15    }
16    return count;
17  }
```

时间复杂度：O(n^3)。

空间复杂度：O(1)。

这种时间复杂度太高，当数据量比较大的时候，很容易超时，我们再来优化一下。当我们以nums[j]为子数组最后一个元素的时候，不用每次都枚举子数组[i……j]之间所有元素的和，只需要以nums[j]为最后一个元素，从后往前累加，即可计算以nums[j]为最后一个元素的连续子数组。比较绕，来看个图



来看下代码

```java
public int subarraySum(int[] nums, int k) {
    int count = 0;
    for (int j = 0; j < nums.length; j++) {
        //sum是以nums[j]为最后一个元素，
        //从后往前累加的值
        int sum = 0;
        for (int i = j; i >= 0; i--) {
            sum += nums[i];
            //如果子数组的和等于k，count就加1
            if (sum == k) {
                count++;
            }
        }
    }
    return count;
}
```

时间复杂度：O(n^2)。

空间复杂度：O(1)。

时间复杂度从n^3降到了n^2，我们再来看一个时间复杂度为n的解决方式，就是前缀和。

前缀和解决

所谓前缀和就是数组中前面n个元素的和，比如前缀和pre[i]的值是：

pre[i]=nums[0]+nums[1]+.....+nums[i];

前缀和pre[j]的值是：

pre[j]=nums[0]+nums[1]+……+nums[i]+nums[i+1]+……+nums[j];

如果我们要求子数组[i……j]之间所有元素的和，也就是
nums[i]+nums[i+1]+……+nums[j]=pre[j]-pre[i-1];

也就是说如果pre[j]-pre[i-1]等于k，说明我们找到了一个和为k个连续子数组，这就变成了**两数之和问题**。因为k的值是固定的，如果枚举pre[j]，我们只需要统计pre[i-1]的个数即可，这个统计方式可以使用map来解决，看下代码（这里为了方便计算，pre长度增加1，pre[0]=0）

```
public int subarraySum(int[] nums, int k) {
  //先计算前缀和，pre[i]表示数组nums中前i个元素的和
  int[] pre = new int[nums.length + 1];
  for (int i = 0; i < nums.length; i++) {
    pre[i + 1] = pre[i] + nums[i];
  }

  int count = 0;
  Map<Integer, Integer> map = new HashMap<>();
  for (int j = 0; j <= nums.length; j++) {
    //计算pre[i-1]+pre[j]=k，我们只需要找出pre[i-1]
    //的个数即可，这个可以通过map来查找
    int other = pre[j] - k;
    if (map.containsKey(other)) {
      //如果map中存在pre[i-1]，把他的个数进行累加
      count += map.get(other);
    }
    //pre[j]的个数加1在放到map中
    map.put(pre[j], map.getOrDefault(pre[j], 0) + 1);
  }
  return count;
}
```

时间复杂度：O(n)。
空间复杂度：O(n)。

往期推荐

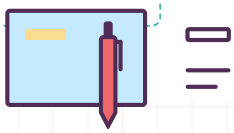
- 572，动态规划解分割回文串 III
- 570，动态规划解回文串分割 IV
- 558，最长回文串
- 553，动态规划解分割回文串 II

583，字符串中的最大奇数

原创 博哥 数据结构和算法 1周前

If you want to understand today , you have to search yesterday.

如果你想参透今天，就必须探究昨天。



问题描述

来源：LeetCode第1903题

难度：简单

给你一个字符串num，表示一个大整数。请你在字符串num的所有非空子字符串中找出值最大的奇数，并以字符串形式返回。如果不存在奇数，则返回一个空字符串""。

子字符串是字符串中的一个连续的字符序列。

示例 1

> **输入**：num = "52"
> **输出**："5"
> **解释**：非空子字符串仅有 "5"、"2" 和 "52" 。"5" 是其中唯一的奇数。

示例 2：

> **输入**：num = "4206"
> **输出**：""
> **解释**：在 "4206" 中不存在奇数。

示例 3：

> **输入**：num = "35427"
> **输出**："35427"
> **解释**："35427" 本身就是一个奇数。

提示：

- 1 <= num.length <= 10^5
- num 仅由数字组成且不含前导零

问题分析

这题是让在字符串num的所有非空子串中找出最大的奇数。我们知道只有个位数是奇数（比如1，3，5，7，9），这个数才可能是奇数，如果个位数是偶数，前面无论怎么截取，最终还是偶数。所以如果想把一个数字变为奇数，唯一能改变的就是他的个位数，所以这题思路就很简单了
先判断字符串num最右边的数字是否是奇数：

- **如果是奇数直接返回**
- **如果不是奇数在判断他的倒数第2位是不是奇数，如果是奇数就从前面截取，如果不是就继续判断倒数第3位……**



看下代码

```
1   public String largestOddNumber(String num) {
2     for (int i = num.length() - 1; i >= 0; i--) {
3       //判断最后一个数字是否是奇数，如果是奇数直接截取返回
4       if (((num.charAt(i) - '0') & 1) == 1)
5         return num.substring(0, i + 1);
```

```
    }
    return "";
}
```

时间复杂度：O(n)，n是字符串的长度，最差情况下都是偶数，遍历所有字符。

空间复杂度：O(1)。

往期推荐

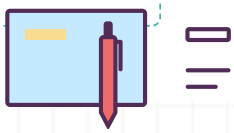
- 577，数组中的最长连续子序列
- 571，山脉数组的峰顶索引
- 562，数组中的最长山脉
- 539，双指针解删除有序数组中的重复项

581，所有蚂蚁掉下来前的最后一刻

原创 博哥 数据结构和算法 7月14日

Our job is improving the quality of life, not just delaying death.

我们要做的是提升生活品质，而非仅仅延缓死亡。



问题描述

来源：LeetCode第1503题

难度：中等

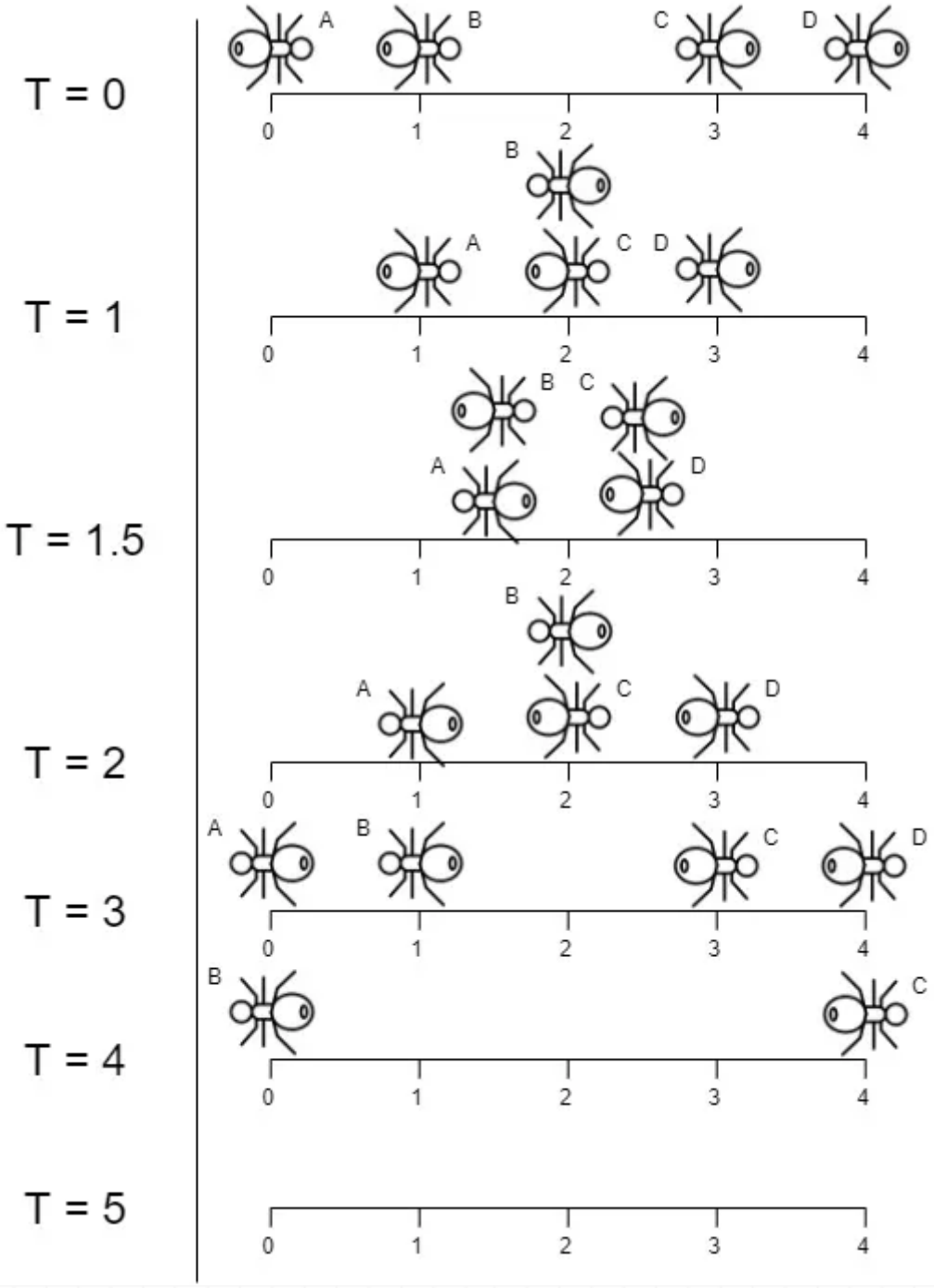
有一块木板，长度为n个单位 。一些蚂蚁在木板上移动，每只蚂蚁都以每秒一个单位的速度移动。其中，一部分蚂蚁向左移动，其他蚂蚁向右移动。

当两只向不同方向移动的蚂蚁在某个点相遇时，它们会同时改变移动方向并继续移动。假设更改方向不会花费任何额外时间。

而当蚂蚁在某一时刻t到达木板的一端时，它立即从木板上掉下来。

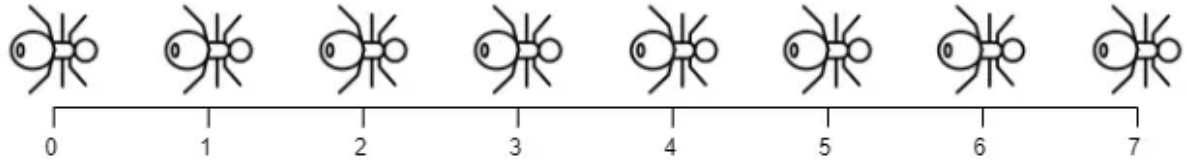
给你一个整数n和两个整数数组left以及right 。两个数组分别标识向左或者向右移动的蚂蚁在t=0时的位置。请你返回最后一只蚂蚁从木板上掉下来的时刻。

示例 1：



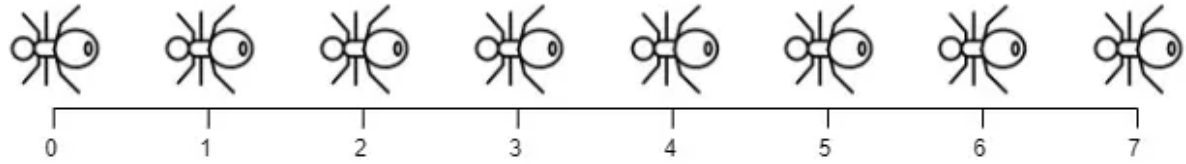
输入：n = 4, left = [4,3], right = [0,1]
输出：4
解释：如上图所示：
-下标 0 处的蚂蚁命名为 A 并向右移动。
-下标 1 处的蚂蚁命名为 B 并向右移动。
-下标 3 处的蚂蚁命名为 C 并向左移动。
-下标 4 处的蚂蚁命名为 D 并向左移动。
请注意，蚂蚁在木板上的最后时刻是 t = 4 秒，之后蚂蚁立即从木板上掉下来。（也就是说在 t = 4.0000000001 时，木板上没有蚂蚁）。

示例 2：



> **输入**：n = 7, left = [], right = [0,1,2,3,4,5,6,7]
> **输出**：7
> **解释**：所有蚂蚁都向右移动，下标为 0 的蚂蚁需要 7 秒才能从木板上掉落。

示例 3：



> **输入**：n = 7, left = [0,1,2,3,4,5,6,7], right = []
> **输出**：7
> **解释**：所有蚂蚁都向左移动，下标为 7 的蚂蚁需要 7 秒才能从木板上掉落。

示例 4：

> **输入**：n = 9, left = [5], right = [4]
> **输出**：5
> **解释**：t = 1 秒时，两只蚂蚁将回到初始位置，但移动方向与之前相反。

示例 5：

> **输入**：n = 6, left = [6], right = [0]
> **输出**：6

提示：

- 1<=n<=10^4
- 0<=left.length<=n+1
- 0<=left[i]<=n
- 0<=right.length<=n+1
- 0<=right[i]<=n
- 1<=left.length+right.length<=n+1
- left和right中的所有值都是唯一的，并且每个值只能出现在二者之一中。

问题分析

这题具有很大的迷惑性，当两只蚂蚁相遇的时候需要计算这两只蚂蚁的位置以及方向，当蚂蚁比较多的时候这样计算非常复杂。

题中说了当两只蚂蚁相遇的时候他们同时改变方向，但速度不变。我们可以这样来思考，假设所有蚂蚁都是一样的，并且具有穿透功能。当两只蚂蚁相遇的时候我们可以认为这两只蚂蚁穿透了，依然沿着原来的方向往前走，最终他们会从木板上掉下来，这个时间就是他们各自距离木板边缘的距离（往左边走的蚂蚁距离木板左边的距离，往右边走的蚂蚁距离木板右边的距离）。

如果还不能明白，我们再来这样思考一下。假设有一只蚂蚁往右走，如果没有遇到其他蚂蚁，他会一直往右走。如果遇到其他蚂蚁，在相遇的那一刻他就会改变方向，往左走，原来往左走的改为往右走。我们假设在这一刻他俩交换身体，那么往右走的那个蚂蚁身体还一直往右走，相当于直接穿过去了。后面如果还遇到蚂蚁还可以按照上面的思路……，所以他就会一直往右走。



搞懂了这个过程，这题就简单多了。这题让求的是最后掉落的时间，我们只需要找出往左走的最大时间和往右走的最大时间，取他俩的最大值即可，来看下代码。

```
public int getLastMoment(int n, int[] left, int[] right) {
  int leftMax = 0;
  //找出往左边走的最大距离
  for (int num : left) {
    leftMax = Math.max(leftMax, num);
  }
  int rightMax = 0;
  //找出往右边走的最大距离
  for (int num : right) {
    rightMax = Math.max(rightMax, n - num);
```

```
    }
    return Math.max(leftMax, rightMax);
}
```

往期推荐

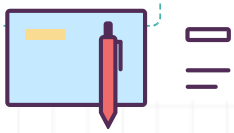
- 579，摩尔投票算法解主要元素
- 557，动态规划解戳气球
- Manacher(马拉车)算法
- 488，二叉树的Morris中序和前序遍历

579，摩尔投票算法解主要元素

原创 博哥 数据结构和算法 7月10日

This world may be rough around the edges, but it's got its charms.

这个世界可能很粗野，但有其魅力所在。



问题描述

来源：LeetCode面试题 17.10
难度：简单

数组中占比超过一半的元素称之为**主要元素**。给你一个整数数组，找出其中的主要元素。若没有，返回-1。请设计时间复杂度为O(N)、空间复杂度为O(1)的解决方案。

示例 1：

> **输入**：[1,2,5,9,5,9,5,5,5]
> **输出**：5

示例 2：

> **输入**：[3,2]
> **输出**：-1

示例 3：

> **输入**：[2,2,1,1,1,2,2]
> **输出**：2

摩尔投票算法解决

这题是让求主要元素，主要元素就是在数组中占比超过一半的元素。咋一看这题很简单，我们可以通过排序或者使用HashMap都是可以解决的。但这题要求时间复杂度为O(N)，空间复杂度为O(1)，很明显上面两种方式都不符合要求。

除了上面两种方式我们可以使用摩尔投票算法来解，维基百科对摩尔投票算法是这样解释的。

摩尔投票算法是在集合中寻找可能存在的多数元素，这一元素在输入的序列重复出现并占到了序列元素的一半以上；在第一遍遍历之后应该再进行一个遍历以统计第一次算法遍历的结果出现次数，确定其是否为众数；如果一个序列中没有占到多数的元素，那么第一次的结果就可能是无效的随机元素。对于数据流而言，则不太可能在亚线性空间复杂度的情况下中就寻找到出现频率最高的元素；而对于序列，其元素的重复次数也有可能很低。

上面说了一大堆，主要说了两层意思：

- **第一就是找出主要元素（不一定有）**
- **第二判断这个数是否是主要元素。**

摩尔投票算法的原理就是使用两个变量，一个major记录当前数字，一个count记录当前数字出现的次数，遇到相同的count就加1，遇到不同的就减1，当count小于0的时候，说明前面的都相互抵消完了，major和count都要重新赋值……，最后再判断major是否是主要元素即可。我们来举个例子

假设数组中每个不同的数字就代表一个国家，而数字的个数就代表这个国家的人数，他们在一起混战，就是不同国家每两个人都同归于尽。我们就可以知道那个人数大于数组长度一半的肯定会获胜（假设主要元素存在）。

就算退一万步来说，其他的所有人都来攻击这个人数最多的国家，他们每两个两个同归于尽，最终剩下的也是那个主要元素，（这里有个前提，就是主要元素必须存在），来看下代码

```
1   public int majorityElement(int[] nums) {
2     //边界条件判断，如果数组为空就返回-1
3     if (nums == null || nums.length == 0)
4       return -1;
5     //先找出主要元素
6     int major = nums[0];
7     int count = 1;
8     for (int i = 1; i < nums.length; i++) {
9       if (major == nums[i]) {
10        //如果当前元素等于major，count就加1
11        count++;
12      } else {
13        //否则count就减1，
14        count--;
15        if (count < 0) {
```

```
            //如果count小于0，说明前面的
            //全部抵消了，这里在重新赋值
            major = nums[i];
            count = 1;
          }
        }
      }
      //下面是判断主要元素是否存在
      count = 0;
      int half = nums.length >> 1;
      for (int num : nums) {
        if (major == num)
          if (++count > half)
            return major;
      }
      return -1;
    }
```

时间复杂度：O(N)，两个for循环，不是嵌套的。

空间复杂度：O(1)，使用了两个变量。

往期推荐

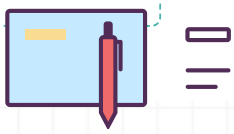
- 577，数组中的最长连续子序列
- 571，山脉数组的峰顶索引
- 556，位运算解形成两个异或相等数组的三元组数目
- 539，双指针解删除有序数组中的重复项

578，计数质数

原创 博哥 数据结构和算法 7月8日

A man is like a novel: until the very last page you don't know how it will end.

人就像一部小说：除非翻到最后一页，否则你不知道TA有怎样的结局。



问题描述

来源：LeetCode第204题

难度：简单

统计所有小于非负整数n的质数的数量。

示例 1：

> **输入**：n = 10
> **输出**：4
> **解释**：小于 10 的质数一共有 4 个，它们是 2，3，5，7 。

示例 2：

> **输入**：n = 0
> **输出**：0

示例 3：

> **输入**：n = 1
> **输出**：0

提示：

- 0 <= n <= 5 * 10^6

解题思路

这题让求质数的个数，质数是指在大于1的自然数中，除了1和它本身以外不再有其他因数的自然数。

我们知道任何合数都可以分解成m个质数的乘积，比如

- **12=2*2*3;**
- **100=2*2*5*5**
- **……**

我们反过来想，任何一个质数比如a，他的n（n>=2）倍一定不是质数，也就是a*2，a*3……都不在是质数。我们可以申请一个长度为length的数组用来存储对应的数是不是质数。然后在用一个变量count来统计质数的个数，如果是合数就不需要统计，如果是质数，count就加1，然后再把质数的2倍，3倍……都标记为非质数，原理比较简单，我们来看下代码

```
public int countPrimes(int n) {
  //标记合数
  boolean[] composite = new boolean[n];
  int count = 0;//统计质数的个数
  for (int i = 2; i < n; i++) {
    //如果是合数就不需要统计
    if (composite[i])
      continue;
    count++;
    //到这一步说明是质数，直接让他的2倍，
    // 3倍……都标记为合数
    for (int j = i; j < n; j += i)
      composite[j] = true;
  }
  return count;
}
```


往期推荐

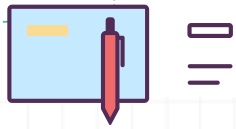
- 575，回溯算法和DFS解单词拆分 II
- 551，回溯算法解分割回文串
- 572，动态规划解分割回文串 III
- 570，动态规划解回文串分割 IV

577，数组中的最长连续子序列

原创 博哥 数据结构和算法 今天

There's room for sentiment but not sentimentality.

可以有感情，但不能感情用事。



问题描述

来源：牛客题霸第95题

难度：中等

给定无序数组arr，返回其中最长的连续序列的长度(要求值连续，位置可以不连续,例如3,4,5,6为连续的自然数)

示例1

> **输入**：[100,4,200,1,3,2]
> **返回值**：4

示例2

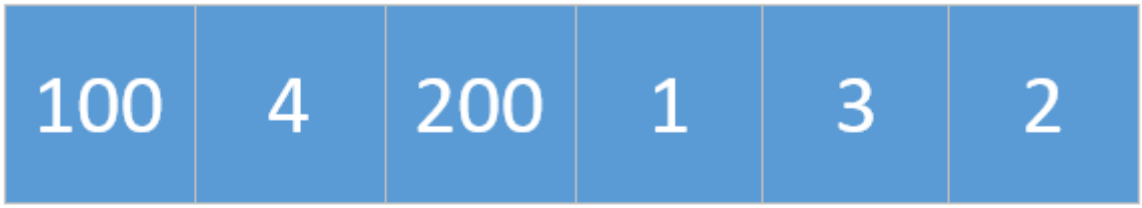
> **输入**：[1,1,1]
> **返回值**：1

提示：

- 0 <= nums.length <= 10^4
- -10^9 <= nums[i] <= 10^9

先排序

因为数组是无序的，如果要想找出最长的连续序列（这里序列的顺序可以打乱），我们最容易想到的就是先对数组进行排序，然后再查找。



排序后



使用一个变量count来记录当前有序序列的长度。

- 如果当前元素比前一个大1，说明他们可以构成连续的序列，count就加1。
- 如果相等就跳过。
- 否则就不能构成连续的序列，count要重置为1，要重新统计。

原理比较简单，我们来看下代码

```
public int MLS(int[] arr) {
    if (arr == null || arr.length == 0)
        return 0;
    int longest = 1;//记录最长的有序序列
    int count = 1;//目前有序序列的长度
    //先对数组进行排序
    Arrays.sort(arr);
    for (int i = 1; i < arr.length; i++) {
        //跳过重复的
        if (arr[i] == arr[i - 1])
            continue;
        //比前一个大1，可以构成连续的序列，count++
        if ((arr[i] - arr[i - 1]) == 1) {
            count++;
        } else {
            //没有比前一个大1，不可能构成连续的，
            //count重置为1
            count = 1;
        }
        //记录最长的序列长度
        longest = Math.max(longest, count);
    }
    return longest;
}
```

时间复杂度：O（nlog(n)），排序的复杂度是nlog(n)，for循环是n，相加是nlog(n)+n，所以时间复杂度是nlog(n)。

空间复杂度：O（1），就使用两个变量

使用集合Set解决

如果不排序的话，我们可以先把数组中的元素全部放到集合set中，然后再查找。假如有最长连续序列

x，x+1，x+2……x+n

我们只有从x往后查找才能找出最长的序列，因为从x+1往后查找的有序序列长度肯定是小于从x往后查找的有序序列长度的。

明白了这点，代码就容易写了，我们需要从有序序列的最小值开始计算即可，来看下代码

```
public int MLS(int[] arr) {
  //先把数组放到集合set中
  Set<Integer> set = new HashSet<>();
  for (int num : arr)
    set.add(num);
  int longest = 0;//记录最长的有序序列
  for (int num : arr) {
    //这里要找有序序列最小的元素（不一定是最长
    //有序序列的）。如果还有更小的，说明当前元素
    //不是最小的，直接跳过
    if (set.contains(num - 1))
      continue;
    //说明当前元素num是当前序列中最小的元素（这里
    //的当前序列不一定是最长的有序序列）
    int currentNum = num;
    //统计当前序列的长度
    int count = 1;
    while (set.contains(currentNum + 1)) {
      currentNum++;
      count++;
    }
    //保存最长的值
    longest = Math.max(longest, count);
  }
  return longest;
}
```

时间复杂度：O（n），for循环是n，只有遇到有序序列最小元素的时候才会执行while里面的循环。

空间复杂度：O（n），使用集合set存储数组中的所有元素

往期推荐

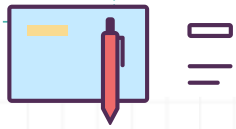
- 548，动态规划解最长的斐波那契子序列的长度
- 538，剑指 Offer-和为s的连续正数序列
- 529，动态规划解最长回文子序列
- 413，动态规划求最长上升子序列

571，山脉数组的峰顶索引

原创 博哥 数据结构和算法 今天

This is the best summer I've ever had.

这是我度过最好的一个夏天。



问题描述

来源：LeetCode第852题

难度：简单

符合下列属性的数组arr称为山脉数组 ：

1，arr.length>=3

2，存在i（0<i<arr.length-1）使得：

- **arr[0]<arr[1]<...arr[i-1]<arr[i]**
- **arr[i]>arr[i+1]>...>arr[arr.length-1]**

给你由整数组成的山脉数组arr，返回任何满足arr[0]<arr[1]<...arr[i-1]<arr[i]>arr[i+1]>...>arr[arr.length-1]的下标i。

示例 1：

> **输入**：arr = [0,1,0]
> **输出**：1

示例 2：

> **输入**：arr = [0,2,1,0]
> **输出**：1

示例 3：

> **输入**：arr = [0,10,5,2]
> **输出**：1

示例 4：

> **输入**：arr = [3,4,5,1]
> **输出**：2

示例 5：

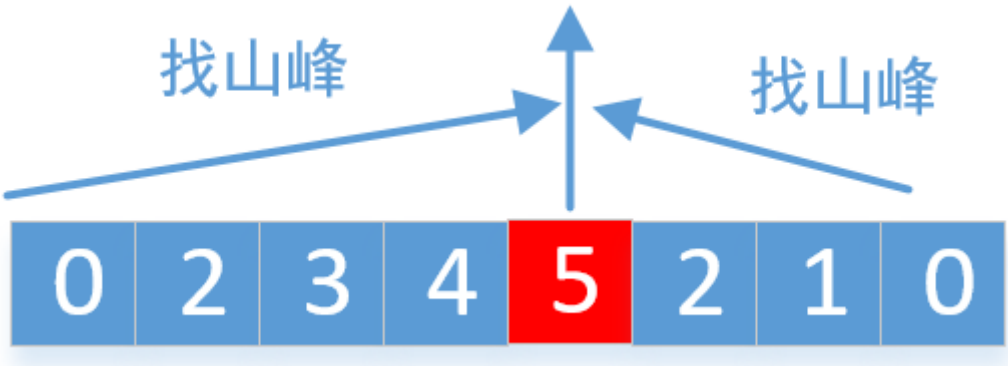
> **输入**：arr = [24,69,100,99,79,78,67,36,26,19]
> **输出**：2

提示：

- 3 <= arr.length <= 10^4
- 0 <= arr[i] <= 10^6
- 题目数据保证arr是一个山脉数组

直接查找

之前讲过475，有效的山脉数组，而这题是让找出山脉数组的峰顶索引，我们看提示的最后一条是题目数据保证arr是一个山脉数组，也就是山脉数组一定是存在的，所以不需要判断，直接查找即可。



山脉数组是先上升然后在下降，如果当前元素比右边挨着的小，说明开始下降了，那么当前元素就是山脉数组的峰顶，只需要返回他的索引即可

```
1   public int peakIndexInMountainArray(int[] arr) {
2     for (int i = 0; i < arr.length - 1; ++i)
3       if (arr[i] > arr[i + 1])
4         return i;
5     return 0;
6   }
```

二分法解决

山脉数组前面部分是升序的，后面部分是降序的，所以也可以使用二分法，用中间的值和他的下一个比较（根据题中对山脉数组的定义，以及提示的最后一条，所以数组中挨着的两个数字不可能相同，要么比下一个小，要么比下一个大）

- 如果比下一个小，说明还处在上升阶段，缩小查找的范围到[mid+1，right]
- 如果比下一个大，说明处在下降阶段，缩小查找范围到[left，mid]

```
public int peakIndexInMountainArray(int[] arr) {
  int left = 0;
  int right = arr.length - 1;
  while (left < right) {
    int mid = left + (right - left) / 2;
    //如果mid比他后面的小，说明是在上升，缩小范围
    if (arr[mid] < arr[mid + 1])
      left = mid + 1;
    else
      right = mid;
  }
  return left;
}
```

或者还可以写成递归的形式

```
public int peakIndexInMountainArray(int[] arr) {
  return helper(arr, 0, arr.length - 1);
}

public int helper(int[] arr, int left, int right) {
  int mid = left + (right - left) / 2;
  //比左右两个都大，说明是峰顶，直接返回
  if (arr[mid] > arr[mid - 1] && arr[mid] > arr[mid + 1])
    return mid;
  //mid指向的值处在爬升阶段
  if (arr[mid] < arr[mid + 1])
    return helper(arr, mid + 1, right);
  return helper(arr, left, mid);
}
```

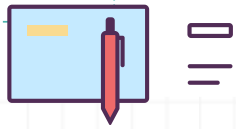

往期推荐

- 562，数组中的最长山脉
- 539，双指针解删除有序数组中的重复项
- 527，两个数组的交集 II
- 516，贪心算法解按要求补齐数组

569，多种方式解4的幂

原创 博哥 数据结构和算法 6天前

> Things have a way of working themselves out.
>
> 船到桥头自然直。



问题描述

来源：LeetCode第342题
难度：简单

给定一个整数，写一个函数来判断它是否是4的幂次方。如果是，返回true；否则，返回false。

整数n是4的幂次方需满足：存在整数x使得n==4^x

示例 1：

> **输入**：n = 16
> **输出**：true

示例 2：

> **输入**：n = 5
> **输出**：false

示例 3：

> **输入**：n = 1
> **输出**：true

提示：

- -2^31 <= n <= 2^31 - 1

递归方式解决

判断一个数是否是4的幂，最简单的一种方式就是不断的除以4，如果最后等于1则是4的幂，否则则不是4的幂，原理比较简单，来直接看下代码

```
public boolean isPowerOfFour(int num) {
  //负数不可能是4的幂
  if (num <= 0)
    return false;
  //1是4的0次幂
  if (num == 1)
    return true;
  //如果不能够被4整除，肯定不是4的幂
  if (num % 4 != 0)
    return false;
  //如果能被4整除，除以4然后递归调用
  return isPowerOfFour(num / 4);
}
```

当然还可以一行代码搞定，但这种可读性不是太好

```
public boolean isPowerOfFour(int num) {
  return num > 0 && (num == 1 || (num % 4 == 0 && isPowerOfFour(num / 4)));
}
```

位运算解决

先不看4的幂，我们先来观察一下2的幂，在32位的二进制中，只要有一个位置是1（不能是符号位），其他位置都是0，那么这个数就是2的幂，比如

```
0b 00000000 00000000 00000000 00010000（是2的幂）
0b 00000000 00000010 00000000 00000000（是2的幂）
0b 00010000 00000000 00000000 00000000（是2的幂）
0b 10000000 00000000 00000000 00000000（不是2的幂，因为他是负数了）
```

我们可以看到在int类型中，是2的幂的只有31个，再来观察一组数据

```
0b 00000000 00000000 00000000 00000001（是2的幂也是4的幂）
0b 00000000 00000000 00000000 00000010（只是2的幂但不是4的幂）
0b 00000000 00000000 00000000 00000100（是2的幂也是4的幂）
0b 00000000 00000000 00000000 00001000（只是2的幂但不是4的幂）
0b 00000000 00000000 00000000 00010000（是2的幂也是4的幂）
0b 00000000 00000000 00000000 00100000（只是2的幂但不是4的幂）
0b 00000000 00000000 00000000 01000000（是2的幂也是4的幂）
0b 00000000 00000000 00000000 10000000（只是2的幂但不是4的幂）
……

0b 00100000 00000000 00000000 00000000（只是2的幂但不是4的幂）
```

```
0b 01000000 00000000 00000000 00000000（是2的幂也是4的幂）
```

通过上面我们可以看到如果一个数是2的幂，并且二进制从右边数奇数位是1的一定是4的幂。判断是2的幂，我们只需要判断二进制中1的个数即可，这里可以参照425，剑指Offer-二进制中1的个数，实际上还有一种更简单的方式，就是判断(num & (num - 1)) == 0，并且还要保证num>0;

最终代码如下

```
public boolean isPowerOfFour(int num) {
    return num > 0 && (num & (num - 1)) == 0 && (num & 0x55555555) == num;
}
```

注意这里0x55555555的二进制是

```
01010101 01010101 01010101 01010101
```

实际上还可以换一种方式

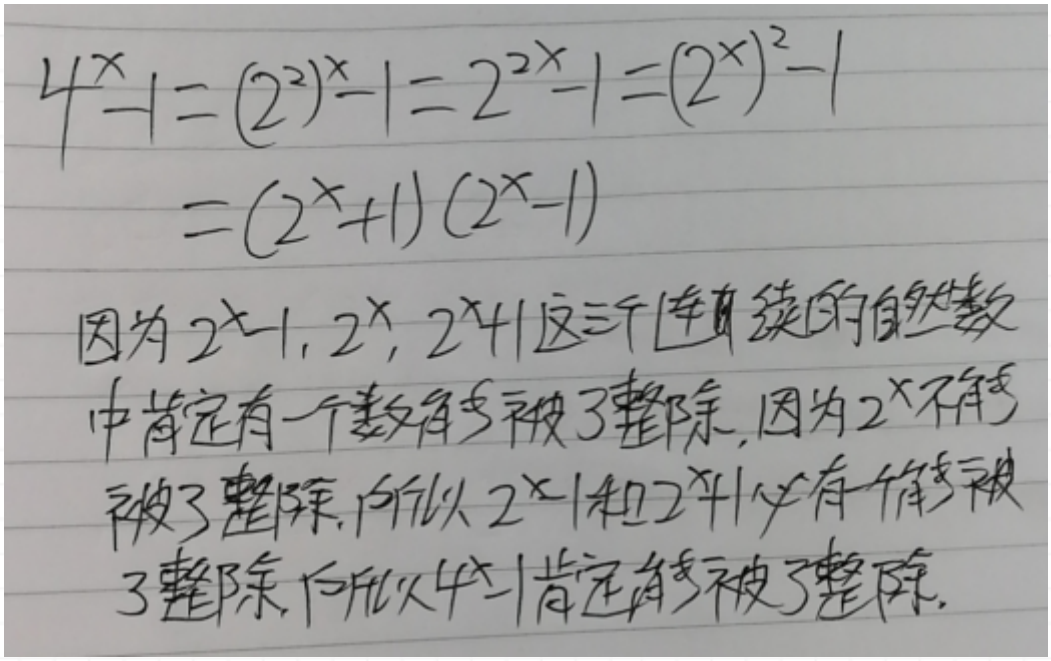
```
public boolean isPowerOfFour(int num) {
    return num > 0 && ((num & (num - 1)) == 0) && (num & 0xaaaaaaaa) == 0;
}
```

这里0xaaaaaaaa的二进制是

```
10101010 10101010 10101010 10101010
```

公式计算

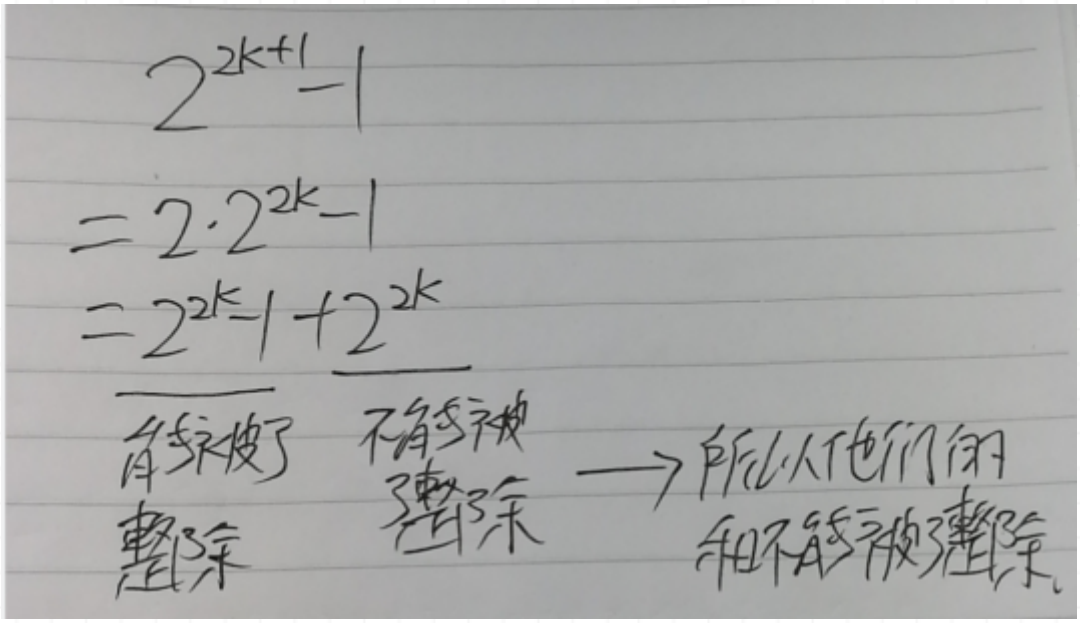
我们来观察一下4的幂次方的一些特点，4的幂次方不好观察，我们来研究一下4的幂次方减1，研究这个特点之前一定要明白这样一条定律：
任何连续的n个自然数的乘积一定能被n整除。



上面图中2^x是2的幂，他是不能被3整除的，所以**如果一个数是2的幂，并且减1还能被3整除，那么这个数一定是4的幂**，代码如下

```
1   public boolean isPowerOfFour(int num) {
2     return num > 0 && (num & (num - 1)) == 0 && (num - 1) % 3 == 0;
3   }
```

有没有一种可能就是一个数num是2的幂，但不是4的幂而且减去1还能被3整除呢，其实是没有这种可能的，如果一个数是2的幂但不是4的幂，那么这个数一定是2的奇次幂，类似于2^(2k+1)，我们来证明一下



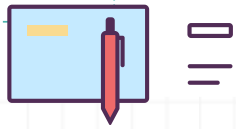
往期推荐

- 565，多种方式解2的幂
- 560，位运算解只出现一次的数字 II
- 556，位运算解形成两个异或相等数组的三元组数目

567，最后一块石头的重量

原创 博哥 数据结构和算法 4天前

> Never give up, always have hope in frontwaiting for.
>
> 永不放弃，总有希望在前面等待。



问题描述

来源：LeetCode第1046题
难度：简单

有一堆石头，每块石头的重量都是正整数。

每一回合，从中选出两块最重的石头，然后将它们一起粉碎。假设石头的重量分别为x和y，且x<=y。那么粉碎的可能结果如下：

如果x==y，那么两块石头都会被完全粉碎；
如果x!=y，那么重量为x的石头将会完全粉碎，而重量为y的石头新重量为y-x。
最后，最多只会剩下一块石头。返回此石头的重量。如果没有石头剩下，就返回0。

示例：

> **输入**：[2,7,4,1,8,1]
> **输出**：1
> **解释**：
> 先选出 7 和 8，得到 1，所以数组转换为 [2,4,1,1,1]，
> 再选出 2 和 4，得到 2，所以数组转换为 [2,1,1,1]，
> 接着是 2 和 1，得到 1，所以数组转换为 [1,1,1]，
> 最后选出 1 和 1，得到 0，最终数组转换为 [1]，这就是最后剩下那块石头的重量。

提示：

- 1 <= stones.length <= 30
- 1 <= stones[i] <= 1000

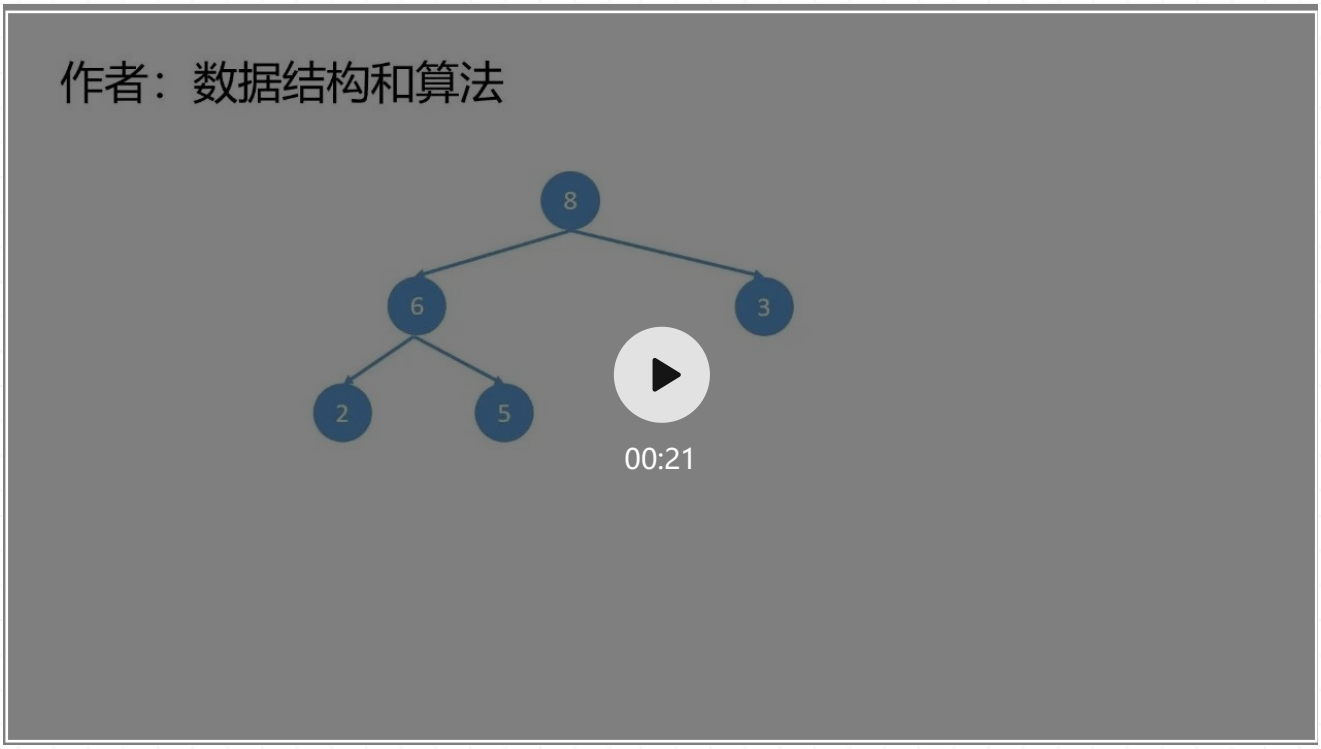
使用最大堆解决

这题要求每次取出最重的两块石头让他们俩相互销毁，如果重量一样则全部销毁，否则销毁之后的重量是大的减小的……，直到全部销毁，或者只剩一个石头为止。

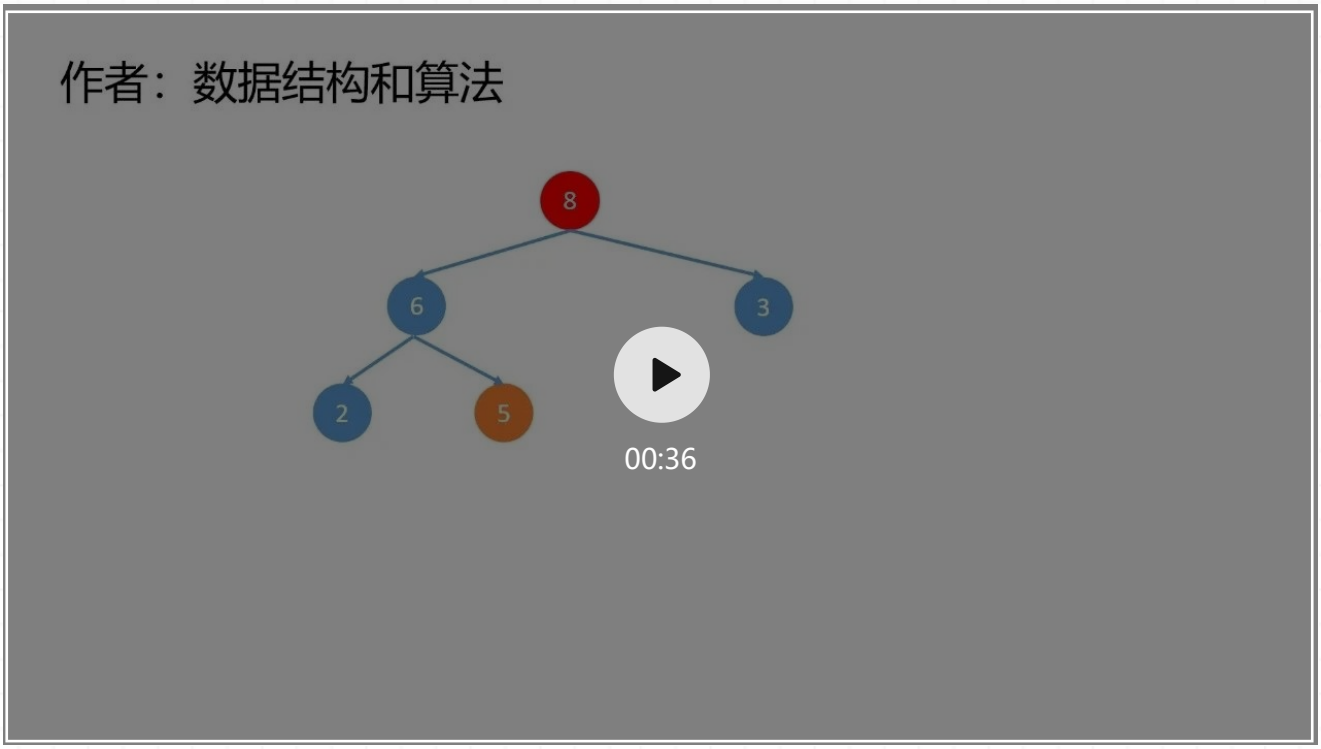
因为每次都要取出最大的两块，如果先排序的话，销毁之后还要在重新排序，这样效率很差，我们可以使用最大堆来解决。最大堆就是堆顶元素始终是堆中所有元素中最大的，我们每次从堆中取出元素或者往堆中添加元素都会导致堆的调整，也就是堆顶元素始终是最大的，来看下代码。

```
public int lastStoneWeight(int[] stones) {
  //最大堆，也就是元素最大的在堆顶
  PriorityQueue<Integer> pq = new PriorityQueue<>((a, b) -> b - a);
  //把数组中的元素全部放入堆中
  for (int num : stones)
    pq.offer(num);
  while (pq.size() > 1) {
    //分别取出堆中最大的值和第二大的值
    int largest = pq.poll();
    int large = pq.poll();
    //如果largest和large一样大，相当于他俩玉石俱焚了，
    //否则就把他俩的差值放到堆中
    if (largest > large)
      pq.offer(largest - large);
  }
  //最后如果堆是空的，说明他们全部都玉石俱焚了，否则就返回
  //堆中仅有的那个值
  return pq.isEmpty() ? 0 : pq.poll();
}
```

前面也讲过378，数据结构-7,堆，具体可以看下，堆中元素的**添加**和**删除**涉及到**往上调整**和**往下调整**，堆也可以看作是一棵完全二叉树，这里我随便举个例子做个视频来看一下**堆的添加**



堆的删除



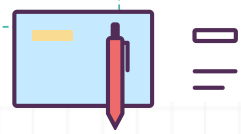
往期推荐

- 563，N叉树的最大深度
- 562，数组中的最长山脉
- 557，动态规划解戳气球
- Manacher(马拉车)算法

562，数组中的最长山脉

原创 博哥 数据结构和算法 5月31日

> To suffer without complaining is the only lesson that has to be learned in this life.
>
> 默默承受，是人生唯一必须懂得的道理。



问题描述

来源：LeetCode第845题
难度：中等

我们把数组A中符合下列属性的任意连续子数组B称为“山脉”：

- B.length >= 3
- 存在0 < i < B.length - 1使得
  B[0] < B[1] < ... B[i-1] < B[i] > B[i+1] > ... > B[B.length-1]

（注意：B可以是A的任意子数组，包括整个数组A。）
给出一个整数数组A，返回最长“山脉”的长度。
如果不含有“山脉”则返回0。

示例 1：

> **输入**：[2,1,4,7,3,2,5]
> **输出**：5
> **解释**：最长的“山脉”是 [1,4,7,3,2]，长度为 5。

示例 2：

> **输入**：[2,2,2]
> **输出**：0
> **解释**：不含“山脉”。

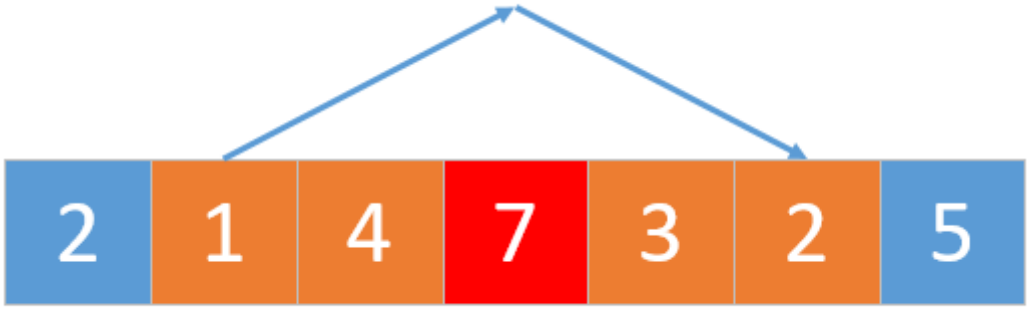
提示：

- 0 <= A.length <= 10000
- 0 <= A[i] <= 10000

解法一

山脉数组就是数组的前部分都是上升的，剩下的部分都是下降的，这样的数组称为山脉数组，之前专门讲过山脉数组，具体可以看下475，有效的山脉数组。

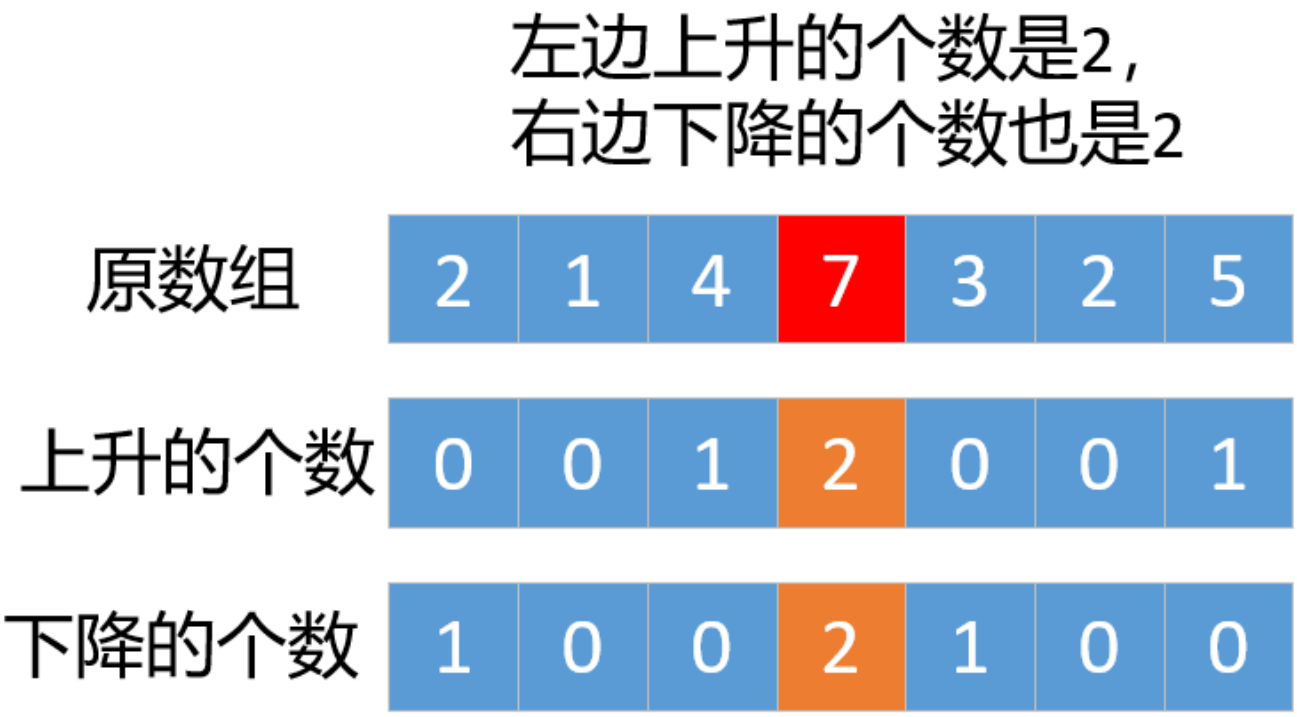
这里我们先找到上升的数组元素个数，到最高点之后再找下降的元素个数，他们相加再加上最顶端的元素个数（1）就是山脉数组的长度，我们只需要保留最长的即可。这里就以示例一为例来画个图看一下



来看下代码

```
public int longestMountain(int[] A) {
  int length = A.length;
  int max = 0;//保存最长的山脉长度
  int index = 1;
  while (index < length) {
    int up = 0;//上升的个数
    int down = 0;//下降的个数

    //既不上升也不下降的要过滤掉
    while (index < length && A[index - 1] == A[index])
      index++;
    //统计上升的个数
    while (index < length && A[index - 1] < A[index]) {
      index++;
      up++;
    }
    //统计下降的个数
    while (index < length && A[index - 1] > A[index]) {
      index++;
      down++;
    }
    //上升和下降的个数必须都大于0，才能称为山脉，计算山脉的长度，
    //保留最大的即可
    if (up > 0 && down > 0)
      max = Math.max(max, up + down + 1);
  }
  return max;
}
```

其实我们还可以提前计算好每一个元素左边上升的个数和右边下降的个数，如果某个元素左边是上升的并且右边是下降的，那么这个元素就是山脉数组中最大的元素，也就是山顶，我们需要计算这个山脉数组的长度。如下图所示



来看下代码。

```
public int longestMountain(int[] A) {
  int length = A.length;
  //up表示上升元素的个数
  int[] up = new int[length];
  for (int i = 1; i < length; i++) {
    if (A[i] > A[i - 1])
      up[i] = up[i - 1] + 1;
  }
  //down表示下降元素的个数
  int[] down = new int[length];
  for (int i = length - 1; i > 0; i--) {
    if (A[i - 1] > A[i])
      down[i - 1] = down[i] + 1;
  }
  //保留最大的长度
  int max = 0;
  for (int i = 0; i < length; i++) {
    if (up[i] == 0 || down[i] == 0)
      continue;
    max = Math.max(max, up[i] + down[i] + 1);
  }
  return max;
}
```

或者还可以合并一起，这样代码就越来越简洁了。

```
public int longestMountain(int[] A) {
  int max = 0, up = 0, down = 0;
  for (int i = 1; i < A.length; ++i) {
    //到山脚的拐点了，或者既没有上升也没有下降，把up和down
    //重新赋值
    if (down > 0 && A[i - 1] < A[i] || A[i - 1] == A[i])
      up = down = 0;
    //计算上升的长度
    if (A[i - 1] < A[i])
      up++;
```

```
        //计算下降的长度
        if (A[i - 1] > A[i])
          down++;
        //既有上升又有下降，说明是山脉数组，他是他的长度，
        //保留最长的即可
        if (up > 0 && down > 0)
          max = Math.max(max, up + down + 1);
      }
      return max;
    }
```

往期推荐

- 539，双指针解删除有序数组中的重复项
- 536，剑指 Offer-构建乘积数组
- 491，回溯算法解将数组拆分成斐波那契序列
- 475，有效的山脉数组

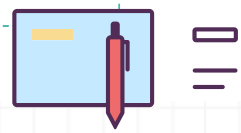
558，最长回文串

原创 博哥 数据结构和算法 5月26日

收录于话题
#算法图文分析

161个 >

Although come back to normal life, we still admire those who see the daylight in fearless years.
尽管最终又回到平凡的生活，人们还是钟爱那些在无畏岁月里看到曙光的人。



问题描述

来源：LeetCode第409题
难度：简单

给定一个包含大写字母和小写字母的字符串，找到通过这些字母构造成的最长的回文串。

在构造过程中，请注意区分大小写。比如"Aa"不能当做一个回文字符串。

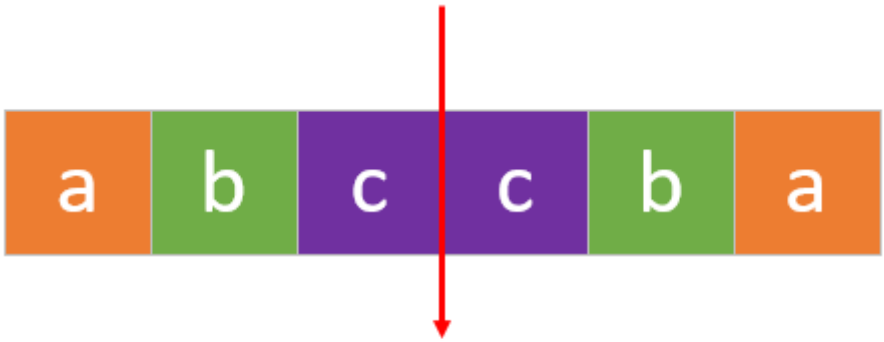
注意:
假设字符串的长度不会超过1010。

示例 1:

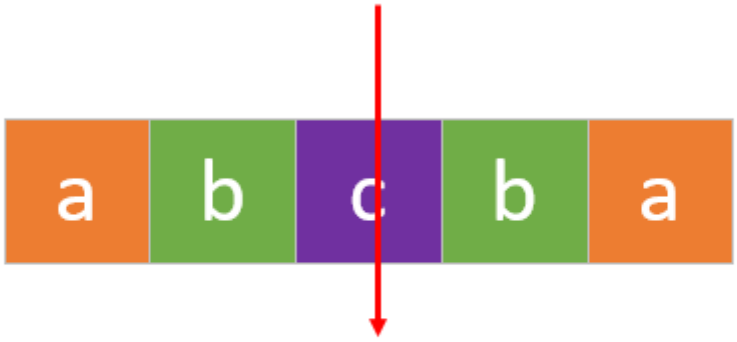
> **输入:**
> "abccccdd"
>
> **输出:**
> 7
>
> **解释:**
> 我们可以构造的最长的回文串是"dccaccd", 它的长度是 7。

解法一

回文串有两种组成形式，**一种是偶数的**比如"abba"，**一种是奇数的**比如"abcba"。所以回文串中每个字符要么都是偶数个，要么只有一个字符的个数是奇数，其他都是偶数。



左右两边对称



左右两边对称

因为这题让求的是可以构造的最长回文串，我们只需要把所有字符截取最大的偶数个，统计他们的和，如果还有剩下的字符，最后再加1，如果没有，最后不用再加了。

比如字符串"aaaaabbbcc"长度是10，a的最大偶数是4，b的最大偶数是2，c的最大偶数也是2，他们的和是4+2+2=8，小于字符串长度10，所以最后要加1，也就是字符串"aaaaabbbcc"可以构造的最长回文串长度是9。

比如字符串"aaaabbcc"长度是8，a的最大偶数是4，b的最大偶数是2，c的最大偶数也是2，他们的和是4+2+2=8，等于字符串长度8，所以最后不需要再加1，也就是字符串"aaaabbcc"可以构造的最长回文串长度是8。

搞懂了上面的分析，代码就简单多了，来看下

```
1   public int longestPalindrome(String s) {
```

```
    int[] map = new int[256];
    //统计每个字符的个数
    for (char ch : s.toCharArray())
      map[ch]++;
    int res = 0;
    int mask = -2;
    for (int count : map) {
      //每个字符的个数取最大偶数，然后相加
      res += count & mask;
    }
    //如果相加的和小于字符串的长度，最后还要加1
    return res < s.length() ? res + 1 : res;
  }
```

上面可能不容易理解的是这样一行代码

```
res += count & mask;
```

因为mask是-2，count&-2的意思就是如果count是偶数，计算的结果还是count，如果count是奇数，计算的结果是count-1。直接看可能不直观，把-2转化为二进制就明白了。

```
11111111 11111111 11111111 11111110
```

除了上面的统计方式以外，我们还可以使用集合Set来统计，代码如下

```
  public int longestPalindrome(String s) {
    int res = 0;
    Set<Character> set = new HashSet<>();
    for (char ch : s.toCharArray()) {
      //如果set中有字符ch，remove方法会
      //返回true，否则返回false
      if (set.remove(ch)) {
        //当前字符ch，出现了2次
        res += 2;
      } else {
        //当前字符ch，只出现1次
        set.add(ch);
      }
    }
    return res + (set.isEmpty() ? 0 : 1);
  }
```

或者还可以这样写，但不管怎么写，原理还是不变的，换汤不换药。

```
  public int longestPalindrome(String s) {
    boolean[] map = new boolean[256];
    int res = 0;
    for (int i = 0; i < s.length(); i++) {
      res += map[s.charAt(i)] ? 2 : 0;
      map[s.charAt(i)] = !map[s.charAt(i)];
    }
    return res < s.length() ? res + 1 : res;
  }
```

550，旋转图像

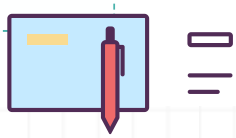
原创 博哥 数据结构和算法 5月10日

收录于话题
#算法图文分析

161个 >

It is never too late to change your life.

任何时候想改变生活都不会太迟。

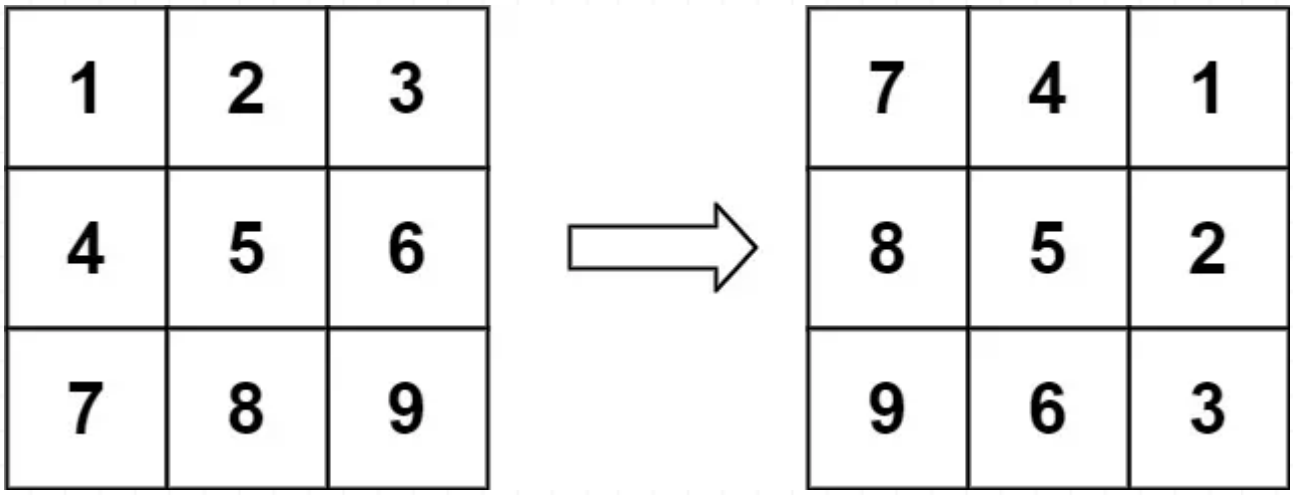


问题描述

给定一个n×n的二维矩阵matrix表示一个图像。请你将图像顺时针旋转90度。

你必须在原地旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要使用另一个矩阵来旋转图像。

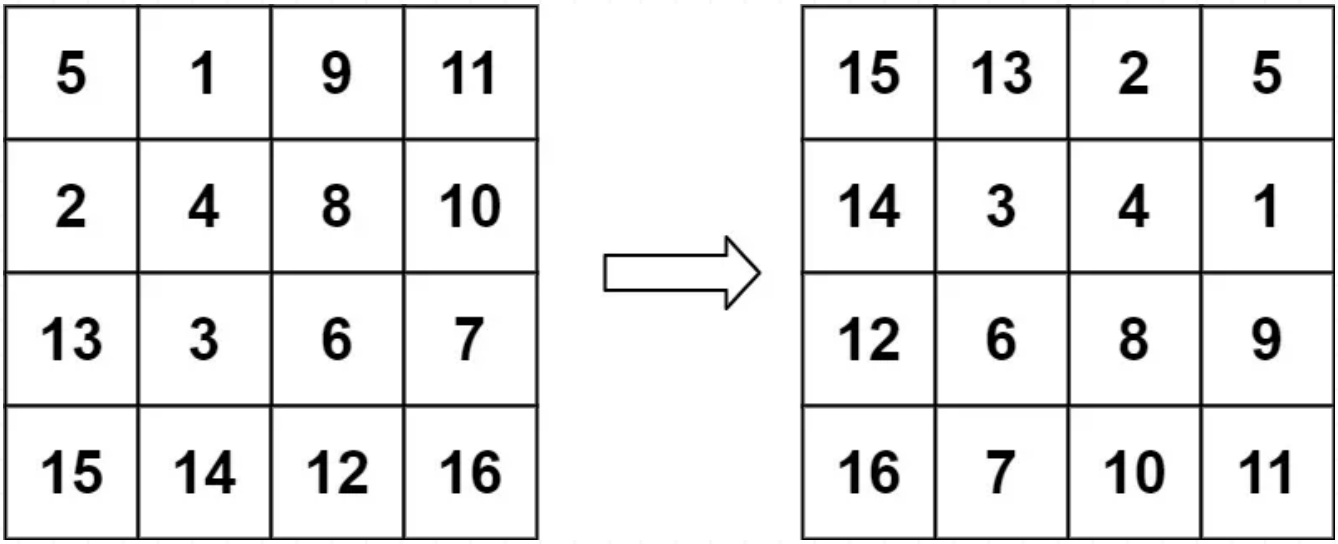
示例 1：



> **输入**：matrix = [
> [1,2,3],
> [4,5,6],
> [7,8,9]]
>
> **输出**：[

[7,4,1],
[8,5,2],
[9,6,3]]

示例 2：



输入：matrix = [
[5,1,9,11],
[2,4,8,10],
[13,3,6,7],
[15,14,12,16]]

输出：[
[15,13,2,5],
[14,3,4,1],
[12,6,8,9],
[16,7,10,11]]

示例 3：

输入：matrix = [[1]]

输出：[[1]]

示例 4：

输入：matrix = [
[1,2],
[3,4]]

输出：[

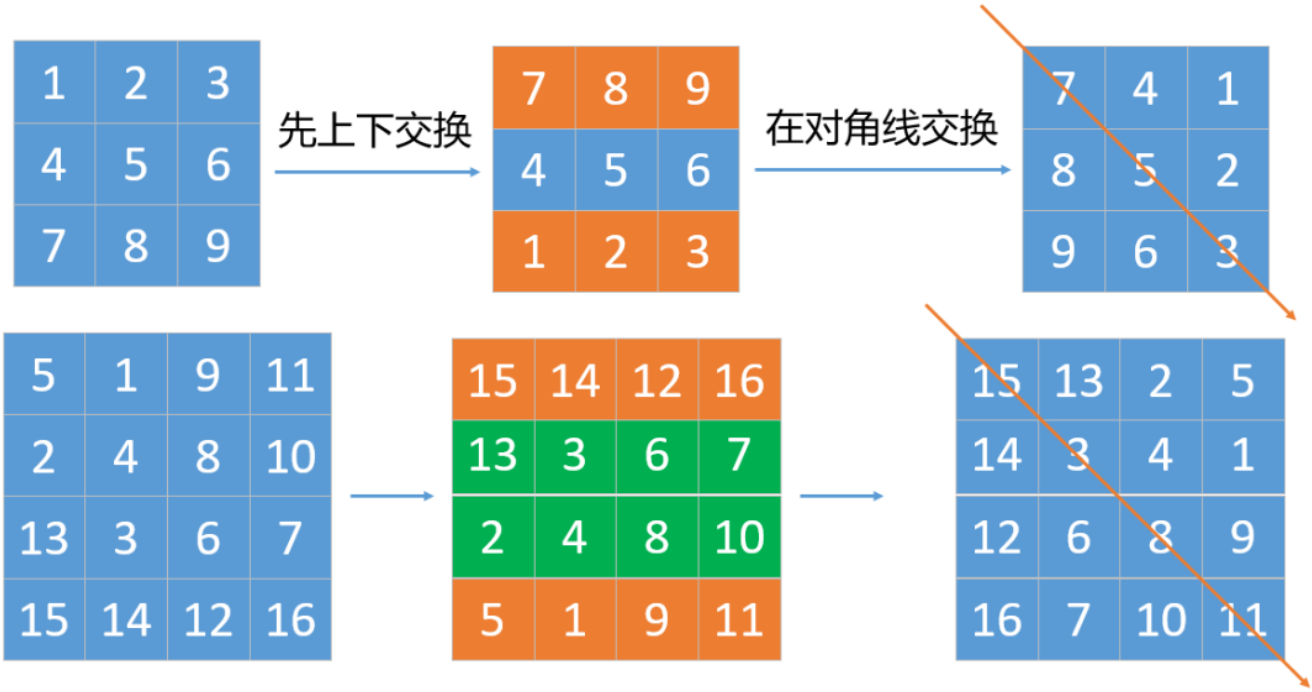
```
[3,1],
[4,2]]
```

提示：

- matrix.length==n
- matrix[i].length==n
- 1<=n<=20
- -1000<=matrix[i][j]<=1000

先上下交换，在对角线交换

这题是让把矩阵顺时针旋转90度，最简单的一种方式就是先上下关于中心线翻转，然后再对角线翻转，具体看下图形分析



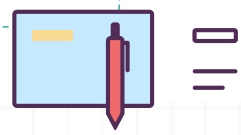
原理比较简单，来直接看下代码

```java
public void rotate(int[][] matrix) {
    int length = matrix.length;
    //先上下交换
    for (int i = 0; i < length / 2; i++) {
        int temp[] = matrix[i];
        matrix[i] = matrix[length - i - 1];
        matrix[length - i - 1] = temp;
    }
    //在按照对角线交换
    for (int i = 0; i < length; ++i) {
        for (int j = i + 1; j < length; ++j) {
            int temp = matrix[i][j];
            matrix[i][j] = matrix[j][i];
            matrix[j][i] = temp;
        }
    }
}
```

546，砖墙，哈希表解决

原创 博哥 数据结构和算法 5月4日

> The closer you think you are, the less you'll actually see.
>
> 离得越近，其实看见的越少。



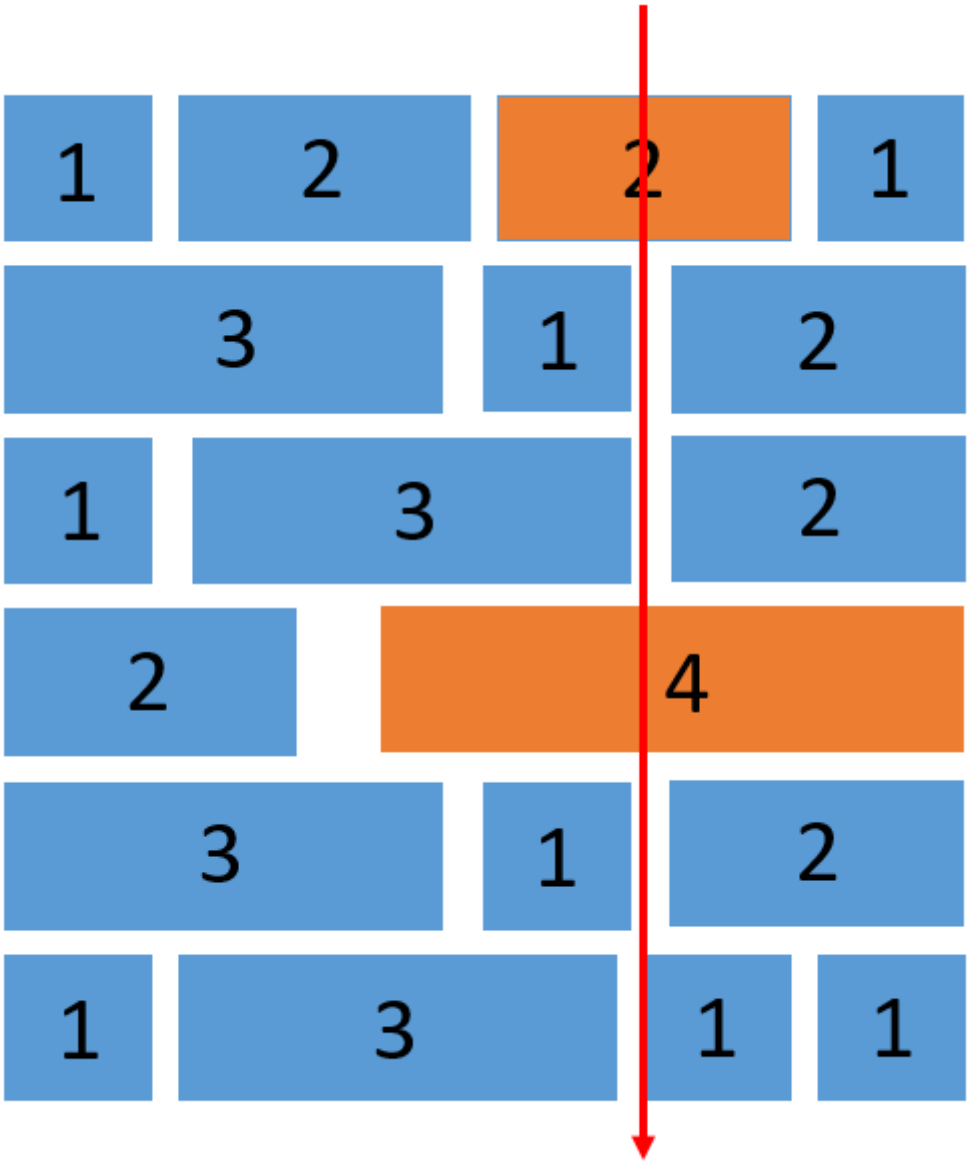
问题描述

你的面前有一堵矩形的、由n行砖块组成的砖墙。这些砖块高度相同（也就是一个单位高）但是宽度不同。每一行砖块的宽度之和应该相等。

你现在要画一条自顶向下的、穿过最少砖块的垂线。如果你画的线只是从砖块的边缘经过，就不算穿过这块砖。你不能沿着墙的两个垂直边缘之一画线，这样显然是没有穿过一块砖的。

给你一个二维数组wall，该数组包含这堵墙的相关信息。其中，wall[i]是一个代表从左至右每块砖的宽度的数组。你需要找出怎样画才能使这条线 穿过的砖块数量最少，并且返回穿过的砖块数量 。

示例 1：



> **输入**：wall = [[1,2,2,1],[3,1,2],[1,3,2],[2,4],[3,1,2],[1,3,1,1]]
>
> **输出**：2

示例 2：

> **输入**：wall = [[1],[1],[1]]
>
> **输出**：3

提示：

- n == wall.length
- 1 <= n <= 10^4
- 1 <= wall[i].length <= 10^4

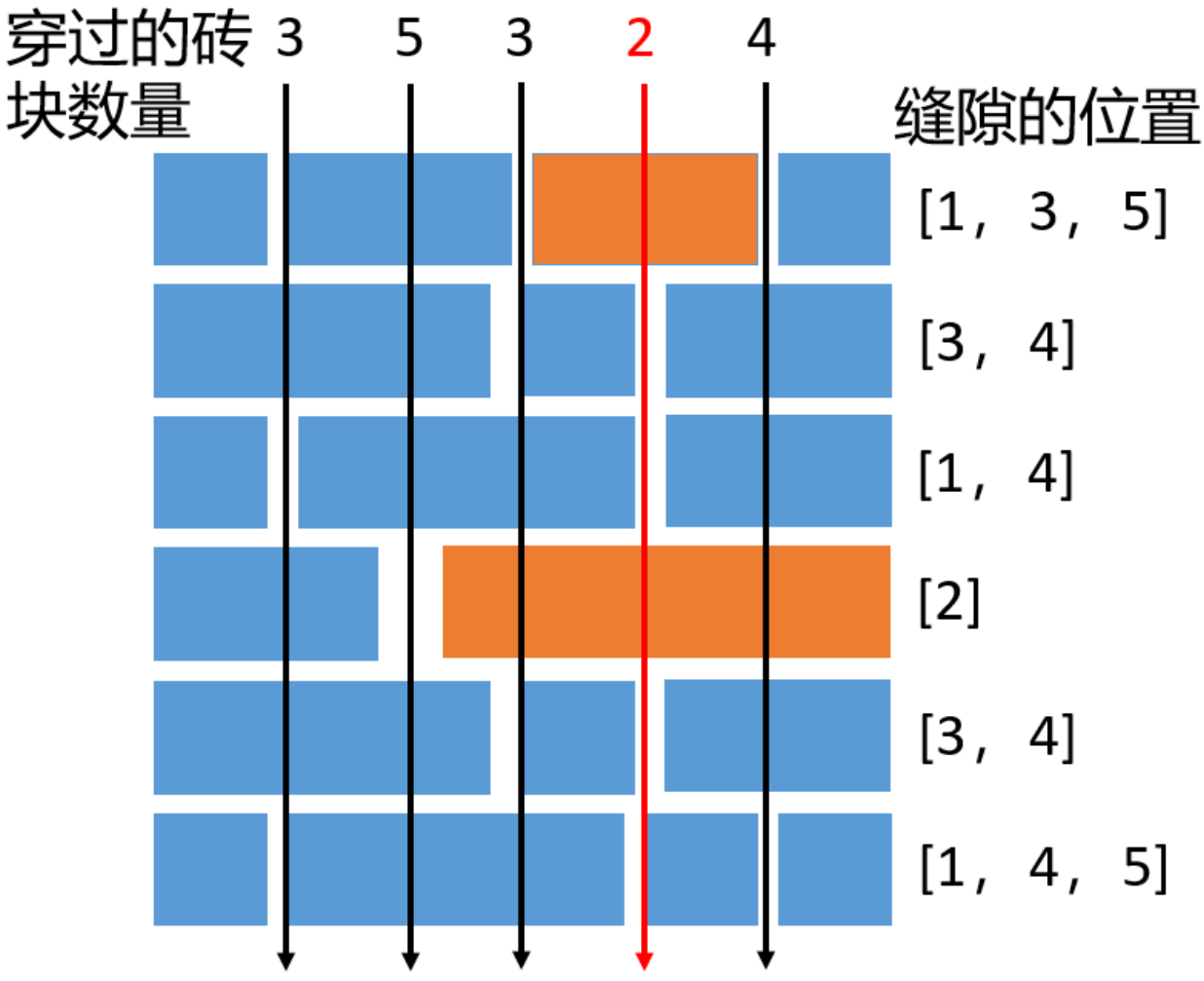
- 1 <= sum(wall[i].length) <= 2 * 10^4
- 对于每一行 i ，sum(wall[i]) 应当是相同的
- 1 <= wall[i][j] <= 2^31 - 1

哈希表解决

这题要求从最上面到最下面画一条线，所穿过的砖块最少，从两砖块之间的缝隙穿过不算穿过砖块。因为高度是一定的，要想穿过砖块最少，必须是穿过缝隙最多，所以我们可以先统计每行缝隙的位置，最后再统计缝隙出现次数最多的位置即可。我们就以示例一为例来看下

第1行的缝隙位置有**[1，3，5]**
第2行的缝隙位置有**[3，4]**
第3行的缝隙位置有**[1，4]**
第4行的缝隙位置有**[2]**
第5行的缝隙位置有**[3，4]**
第6行的缝隙位置有**[1，4，5]**

如下图所示



这就是一个前缀和的问题，因为缝隙是不穿过砖块的，我们求出每行缝隙的位置，然后计算缝隙最多的位置即可，从上面我们可以看出，缝隙最多的位置是4，也就是上面图中所画的，总共是穿过了2个砖块。搞懂了上面的分析，代码就简单多了，我们来看下

```
public int leastBricks(List<List<Integer>> wall) {
    //map中的key存储的是每块砖缝隙的位置，value是每个不同的
    //位置出现的次数
    Map<Integer, Integer> map = new HashMap();
    int maxGap = 0;//保存缝隙出现的最大值即可
    for (List<Integer> row : wall) {
        //每行缝隙的位置，也就当前行中每块砖右边的位置
        int gap = 0;
        //因为最后一块砖的右边是墙的边缘，根据题的要求不能沿
        //着这里画垂线，所以最后一块砖的位置就不要计算了
        for (int i = 0; i < row.size() - 1; i++) {
            //计算当前砖右边的位置，也是当前砖右边缝隙的位置
            gap += row.get(i);
            //key是缝隙的位置，value是这个位置出现的次数
            map.put(gap, map.getOrDefault(gap, 0) + 1);
            //保存缝隙出现的最大值
            maxGap = Math.max(maxGap, map.get(gap));
        }
    }
    //穿过砖的数量是墙的高度减去穿过缝隙的数量
    return wall.size() - maxGap;
}
```


往期推荐

- 540，动态规划和中心扩散法解回文子串
- 538，剑指 Offer-和为s的连续正数序列
- 535，剑指 Offer-扑克牌中的顺子
- 447，双指针解旋转链表

541，字符串压缩，视频演示

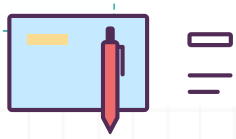
原创 博哥 数据结构和算法 今天

收录于话题
#算法图文分析

146个 >

Live life to the fullest.

尽情地享受生活吧。



问题描述

字符串压缩。利用字符重复出现的次数，编写一种方法，实现基本的字符串压缩功能。比如，字符串aabcccccaaa会变为a2b1c5a3。若“压缩”后的字符串没有变短，则返回原先的字符串。你可以假设字符串中只包含大小写英文字母（a至z）。

示例1：

> **输入**："aabcccccaaa"
> **输出**："a2b1c5a3"

示例2：

> **输入**："abbccd"
> **输出**："abbccd"
> **解释**："abbccd"压缩后为"a1b2c2d1"，比原字符串长度更长。

提示：

1. 字符串长度在[0, 50000]范围内。

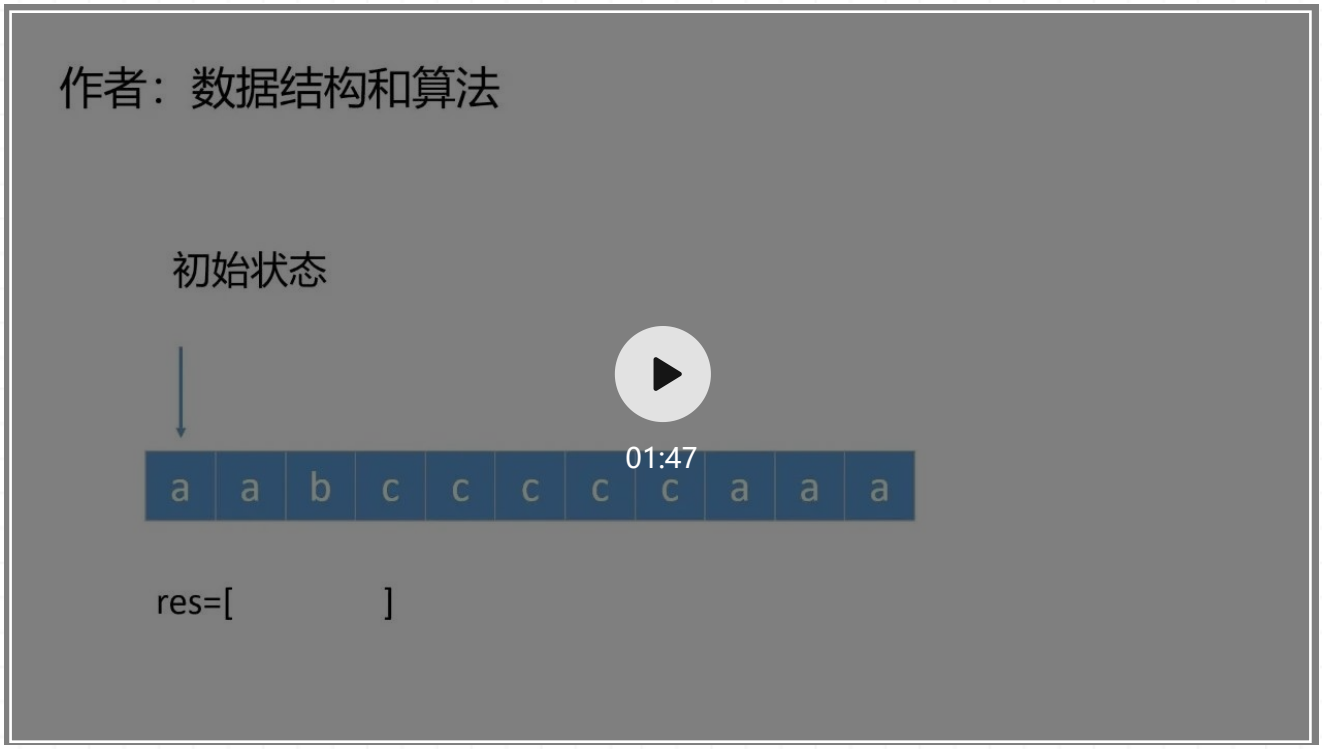
问题分析

这题是让统计连续相同字符的数量，非常简单的一道题，我们来看下解题思路

- **使用一个变量curChar记录当前字符，一个变量curCharCount记录当前字符出现的次数**
- **当遇到新字符的时候，就把curChar和curCharCount加入到结果res中，然后再对curChar和curCharCount重新初始化。**

重复上面的重复，直到把所有的字符遍历完为止

我们就以示例1为例来看下视频演示



再来看下代码

```
1   public String compressString(String S) {
2     //边界条件判断
3     if (S == null || S.length() == 0)
4       return S;
5
6     StringBuilder res = new StringBuilder();
7     //当前字符
8     char curChar = S.charAt(0);
9     //当前字符的数量
10    int curCharCount = 1;
11    for (int i = 1; i < S.length(); i++) {
12      //如果当前字符有重复的，统计当前字符的数量
13      if (S.charAt(i) == curChar) {
14        curCharCount++;
15        continue;
16      }
17      //走到这里，说明遇到了新的字符，
18      //这里先把当前字符和他的数量加入到res中
19      res.append(curChar).append(curCharCount);
20      //然后让当前字符指向新的字符，并且数量也要
21      //重新赋值为1
22      curChar = S.charAt(i);
23      curCharCount = 1;
24    }
25    //因为上面计算的时候会遗漏最后一个字符和他的数量，
26    //这要添加到res中
27    res.append(curChar).append(curCharCount);
28
```

```
    //根据题的要求，若“压缩”后的字符串没有变短，
    // 则返回原先的字符串
    return res.length() >= S.length() ? S : res.toString();
  }
```

还可以换另一种方式，先把当前字符加入到res中，当遇到新的字符的时候再把当前字符的数量加进来，其实原理都差不多，我们来看下

```
public String compressString(String S) {
    //边界条件判断
    if (S == null || S.length() == 0)
      return S;
    StringBuilder res = new StringBuilder();
    //先把第一个字符添加到res中
    res.append(S.charAt(0));
    int count = 1;
    for (int i = 1; i < S.length(); i++) {
      //判断重复字符的数量
      if (S.charAt(i) == S.charAt(i - 1)) {
        count++;
        continue;
      }
      //走到这里，说明遇到了新的字符，先把前面字符
      //的数量添加到res中，然后再添加这个新的字符
      res.append(count).append(S.charAt(i));
      count = 1;
    }
    //上面的计算会遗漏最后一个字符的数量，这里加上
    res.append(count);
    return res.length() >= S.length() ? S : res.toString();
  }
```

总结

非常简单的一道题，只需要从前往后一个个遍历，遇到相同的就累加，遇到不同的就重新初始化……

往期推荐

- 537，剑指 Offer-字符串的排列
- 526，删除字符串中的所有相邻重复项
- 496，字符串中的第一个唯一字符
- 487，重构字符串

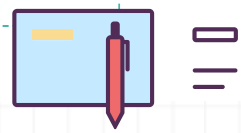
536，剑指 Offer-构建乘积数组

原创 博哥 数据结构和算法 6天前

收录于话题
#剑指offer

32个 >

> By its very nature, history is always a one-sided account.
>
> 就其本质而言，历史始终是一面之词。



问题描述

给定一个数组 A[0,1,…,n-1]，请构建一个数组 B[0,1,…,n-1]，其中 B[i] 的值是数组 A 中除了下标 i 以外的元素的积，即 B[i]=A[0]×A[1]×…×A[i-1]×A[i+1]×…×A[n-1]。不能使用除法。

示例：

> **输入**：[1,2,3,4,5]
> **输出**：[120,60,40,30,24]

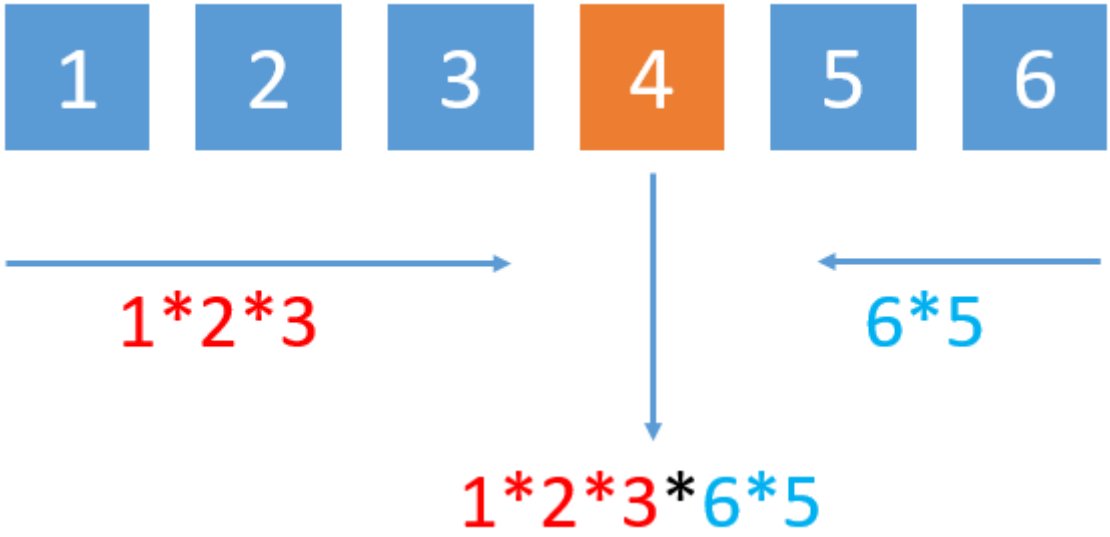
提示：

- 所有元素乘积之和不会溢出32位整数
- a.length<=100000

两边分别相乘

这题要求的是每个元素的值是除自己以外其他所有元素的乘积。最简单的一种方式就是把所有元素都相乘，然后再用这个乘积除以每一个元素即可。但题中要求的是不能使用除法，所以这种方式是行不通的。

如果我们能计算每个元素左边所有元素的乘积和右边所有元素的乘积，只需要把他们相乘就可以满足这题的要求，就像下面这样，如果我们要求元素4的值



代码如下

```
public int[] constructArr(int[] a) {
    //边界条件判断
    if (a == null || a.length == 0)
        return a;
    int length = a.length;
    //每个元素左边所有元素的乘积
    int[] resLeft = new int[length];
    //每个元素右边所有元素的乘积
    int[] resRight = new int[length];
    //两个默认值
    resLeft[0] = 1;
    resRight[length - 1] = 1;

    //当前元素左边的所有元素乘积（不包含当前元素）
    for (int i = 1; i < length; i++) {
        resLeft[i] = resLeft[i - 1] * a[i - 1];
    }
    //当前元素右边的所有元素乘积（不包含当前元素）
    for (int i = length - 2; i >= 0; i--) {
        resRight[i] = resRight[i + 1] * a[i + 1];
    }
    //左边乘以右边就是我们要求的结果
    int[] res = new int[length];
    for (int i = 0; i < length; i++) {
        res[i] = resLeft[i] * resRight[i];
    }
    return res;
}
```

代码优化

上面代码中有3个for循环，其中第2个可以和第3个合并，来看下代码。

```
public int[] constructArr(int[] a) {
    //边界条件的判断
    if (a == null || a.length == 0)
        return a;
    int length = a.length;
    int[] res = new int[length];
    res[0] = 1;
    //当前元素左边的所有元素乘积（不包含当前元素）
```

```
    for (int i = 1; i < length; i++) {
      res[i] = res[i - 1] * a[i - 1];
    }
    int right = 1;
    //right表示当前元素右边所有元素的乘积（不包含当前元素），
    //res[i]表示的是左边的乘积，他俩相乘就是
    //除了自己以外数组的乘积
    for (int i = length - 1; i >= 0; i--) {
      res[i] *= right;
      right *= a[i];
    }
    return res;
  }
```

往期推荐

- 440，剑指 Offer-从上到下打印二叉树 II
- 436，剑指 Offer-顺时针打印矩阵
- 432，剑指 Offer-反转链表的3种方式
- 422，剑指 Offer-使用DFS和BFS解机器人的运动范围

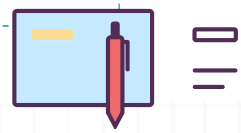
535，剑指 Offer-扑克牌中的顺子

原创 博哥 数据结构和算法 1周前

收录于话题
#剑指offer

32个 >

> The real voyage of discovery consists not in seeking new landscapes, but in having new eyes.
>
> 真正的发现之旅不在于找寻新的天地，而在于拥有新的眼光。



问题描述

从扑克牌中随机抽5张牌，判断是不是一个顺子，即这5张牌是不是连续的。2～10为数字本身，A为1，J为11，Q为12，K为13，而大、小王为 0 ，可以看成任意数字。A 不能视为 14。

示例 1:

> **输入**: [1,2,3,4,5]
> **输出**: True

示例 2:

> **输入**: [0,0,1,2,5]
> **输出**: True

限制:

- 数组长度为 5
- 数组的数取值为 [0, 13] .

先排序

这题要求很简单

- 假如没有大小王的情况下，只需要这5个数字**不能有重复的**，并且这5个数字中的**最大值减去最小值等于4**就行了。类似于[a,a+1,a+2,a+3,a+4]。
- 假如有大小王的情况下，无论有一个还是有两个，我们只需要让大小王替换上面数组[a,a+1,a+2,a+3,a+4]中的任意元素，也能构成顺子。比如替换a,类似于[0，a+1,a+2,a+3,a+4]，那么这个数组中的最大值减去最小值就是3了。

所以我们可以得出结论

- **只要数组中的最大值减去最小值小于等于4**
- **数组中的元素不能有重复的**

只要满足上面两个条件就是顺子

我们先对数组进行排序，然后再来求解，来看下代码

```
public boolean isStraight(int[] nums) {
  //先对数组进行排序
  Arrays.sort(nums);
  //记录大小王的数量
  int zero = 0;
  for (int i = 0; i < 5; i++) {
    //统计大小王的数量
    if (nums[i] == 0) {
      zero++;
      continue;
    }
    //如果不是大小王，不能有重复的
    if (i != 0 && nums[i] == nums[i - 1])
      return false;
  }
  //最大牌和最小牌的差值小于等于4（这里zero是大小王是数量，
  // nums[zero]表示排序后第一个非大小王的牌）
  return nums[nums.length - 1] - nums[zero] <= 4;
}
```

位运算解决

因为题中的最大数字是14，小于int的32位，所以我们可以用位运算来标记是否有重复的。原理和上面一样，只需要满足上面的两个条件即可。

来看下代码

```
public boolean isStraight(int[] nums) {
  int bit = 0;//记录每个数字是否出现过
  //记录数组中的最小数字
  int min = Integer.MAX_VALUE;
  //记录数组中的最大数字
  int max = Integer.MIN_VALUE;
  for (int num : nums) {
    //如果是大小王则跳过
    if (num == 0)
      continue;
    //判断相应的位置是否有数字，如果有数字
    //说明之前出现过，也就是有重复的，
    //直接返回false
    if ((bit & (1 << num)) != 0)
      return false;
    //把相应的位置标记为有数字
    bit |= 1 << num;
```

```
18          //记录遍历过的最大值和最小值
19          min = min > num ? num : min;
20          max = max < num ? num : max;
21      }
22      //最大牌和最小牌的差值小于等于4
23      return max - min <= 4;
24  }
```

往期推荐

- 533，剑指 Offer-最小的k个数
- 442，剑指 Offer-回溯算法解二叉树中和为某一值的路径
- 441，剑指 Offer-二叉搜索树的后序遍历序列
- 434，剑指 Offer-二叉树的镜像

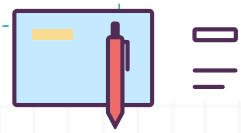
533，剑指 Offer-最小的k个数

原创 博哥 数据结构和算法 4月1日

收录于话题
#算法图文分析

143个 >

> Optimists are right. Pessimists are right. It's up to you to choose which you will be.
>
> 乐观者是对的，悲观者也没错，你自己决定你想成为哪种人。



问题描述

输入整数数组arr，找出其中最小的k个数。例如，输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。

示例 1：

> **输入**：arr = [3,2,1], k = 2
> **输出**：[1,2] 或者 [2,1]

示例 2：

> **输入**：arr = [0,1,2,1], k = 1
> **输出**：[0]

限制：

- 0 <= k <= arr.length <= 10000
- 0 <= arr[i] <= 10000

先排序

这题是让求数组中最小的k个数，很容易想到的一种实现方式就是，先对数组进行排序，然后取前k个即可

```java
public int[] getLeastNumbers(int[] arr, int k) {
  //先排序，然后选择前k个即可
  Arrays.sort(arr);
  int[] res = new int[k];
  for (int i = 0; i < k; ++i) {
    res[i] = arr[i];
  }
  return res;
}
```

使用最大堆

关于堆不熟悉的可以看下《378，数据结构-7,堆》，堆分为最大堆和最小堆，最大堆的堆顶元素是堆中最大的，最小堆的堆顶元素是堆中最小的。这里可以使用最大堆，我们把堆看做是一个容器，最大只能容纳k个元素，具体实现如下：

遍历数组中的每一个元素，

1，如果堆的size小于k，直接把遍历的元素加入到堆中。

2，如果堆的size大于等于k，就要判断当前遍历的元素是否比堆顶元素小，

- 如果比堆顶元素小，就把堆顶元素给移除，把当前遍历的元素加入到堆中。
- 如果比堆顶元素大，就跳过。

原理比较简单，我们来看下代码

```java
public int[] getLeastNumbers(int[] arr, int k) {
  //加个边界条件的判断
  if (k == 0) {
    return new int[0];
  }
  //创建最大堆
  PriorityQueue<Integer> queue = new PriorityQueue<>((num1, num2) -> num2 - num1);
  //先在堆中放数组的前k个元素
  for (int i = 0; i < k; ++i) {
    queue.offer(arr[i]);
  }
  //因为是最大堆，也就是堆顶的元素是堆中最大的，遍历数组后面元素的时候，
  //如果当前元素比堆顶元素大，就把堆顶元素给移除，然后再把当前元素放到堆中，
  for (int i = k; i < arr.length; ++i) {
    if (queue.peek() > arr[i]) {
      queue.poll();
      queue.offer(arr[i]);
    }
  }
  //最后再把堆中元素转化为数组
  int[] res = new int[k];
  for (int i = 0; i < k; ++i) {
    res[i] = queue.poll();
  }
  return res;
}
```

如果对堆不熟悉，还可以换种方式，使用TreeMap，原理都是一样的。

```java
public int[] getLeastNumbers(int[] arr, int k) {
  //加个边界条件的判断
  if (k == 0) {
    return new int[0];
  }
  //map中key存放数组中元素，value存放这个元素的个数
```

```
    TreeMap<Integer, Integer> map = new TreeMap<>();
    int count = 0;
    for (int i = 0; i < arr.length; i++) {
      //map中先存放k个元素，之后map中元素始终维持在k个
      if (count < k) {
        map.put(arr[i], map.getOrDefault(arr[i], 0) + 1);
        count++;
        continue;
      }
      Map.Entry<Integer, Integer> entry = map.lastEntry();
      //从第k+1个元素开始，每次存放的时候都要和map中最大的那个比较，如果比map中最大的小，
      //就把map中最大的给移除，然后把当前元素加入到map中
      if (entry.getKey() > arr[i]) {
        //移除map中最大的元素，如果只有一个直接移除。如果有多个（数组中会有重复的元素），移除一个就行
        if (entry.getValue() == 1) {
          map.pollLastEntry();
        } else {
          map.put(entry.getKey(), entry.getValue() - 1);
        }
        //把当前元素加入到map中
        map.put(arr[i], map.getOrDefault(arr[i], 0) + 1);
      }
    }

    //把map中key存放到集合list中
    int[] res = new int[k];
    int index = 0;
    for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
      int keyCount = entry.getValue();
      while (keyCount-- > 0) {
        res[index++] = entry.getKey();
      }
    }
    return res;
  }
```

参考快速排序

快速排序是通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。--来自百度百科

快速排序说简单一点就是我们找到一个中枢值，这个可以使任何值，我们一般默认是排序范围内的第一个，然后扫描后面的元素，把小于中枢值的往前挪，大于中枢值的往后挪，然后再把中枢值放到合适的位置，这样一轮排序下来，中枢值在他前面的都是比他小的，在他后面的都是比他大的。如果对快排不熟悉的也可以看下《104，排序-快速排序》。我们随便举个例子来看个视频



所以这题的解题思路就是，每次确定中枢值的位置之后，我们要判断这个位置是否等于k（数组的下标是从0开始的）。

- 如果等于k，那么他前面的k个元素都是小于中枢值的，后面的都是大于中枢值的，也就是说他前面的k个元素正是我们要找的。
- 如果小于k，说明他前面的元素都是我们要找的，但还不够，我们还要继续往后找剩下的。
- 如果大于k，说明他前面的元素够k个了，我们只需要在他前面找即可，他后面的就不需要了。

最后再来看下代码

```
public int[] getLeastNumbers(int[] arr, int k) {
    int[] res = new int[k];
    quickSort(arr, res, k, 0, arr.length - 1);
    return res;
}

private void quickSort(int[] arr, int[] res, int k, int left, int right) {
    //快排的实现方式有多种，我们选择了其中的一种
    int start = left;
    int end = right;
    while (left < right) {
        while (left < right && arr[right] >= arr[start]) {
            right--;
        }
        while (left < right && arr[left] <= arr[start]) {
            left++;
        }
        swap(arr, left, right);
    }
    swap(arr, left, start);
    //注意这里，start是数组中元素的下标。在start之前的元素都是比start指向的元素小，
    //后面的都是比他大。如果k==start，正好start之前的k个元素是我们要找的，也就是
    //数组中最小的k个，如果k>start，说明前k个元素不够，我们还要往后再找找。如果
    //k<start，说明前k个足够了，我们只需要在start之前找k个即可。
```

```
    if (left > k) {
      quickSort(arr, res, k, start, left - 1);
    } else if (left < k) {
      quickSort(arr, res, k, left + 1, end);
    } else {
      //取前面的k个即可
      for (int m = 0; m < k; ++m) {
        res[m] = arr[m];
      }
    }
  }

  //交换数组中两个元素的值
  private void swap(int[] arr, int i, int j) {
    if (i == j)
      return;
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
  }
```

总结

这题使用快排的思路相对于其他两种实现方式要复杂一点，但效率要比前面两种高很多。

往期推荐

- 521，滑动窗口解最大连续1的个数 III
- 443，滑动窗口最大值
- 407，动态规划和滑动窗口解决最长重复子数组
- 398，双指针求无重复字符的最长子串

525，最富有客户的资产总量

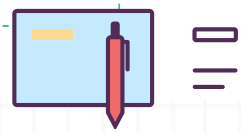
原创 博哥 数据结构和算法 4天前

收录于话题
#算法图文分析

137个 >

There is no development physically or intellectually without effort, and effort means work.

没有努力，就不会有身体或智能上的成长，而努力意谓干活。



问题描述

给你一个mxn的整数网格accounts，其中accounts[i][j]是第i位客户在第j家银行托管的资产数量。返回最富有客户所拥有的资产总量 。

客户的资产总量就是他们在各家银行托管的资产数量之和。最富有客户就是资产总量最大的客户。

示例 1：

> **输入**：accounts = [[1,2,3],[3,2,1]]
> **输出**：6
> **解释**：
> 第 1 位客户的资产总量 = 1 + 2 + 3 = 6
> 第 2 位客户的资产总量 = 3 + 2 + 1 = 6
> 两位客户都是最富有的，资产总量都是 6 ，所以返回 6 。

示例 2：

> **输入**：accounts = [[1,5],[7,3],[3,5]]
> **输出**：10
> **解释**：
> 第 1 位客户的资产总量 = 6
> 第 2 位客户的资产总量 = 10

> 第 3 位客户的资产总量 = 8
> 第 2 位客户是最富有的，资产总量是 10

示例 3：

> **输入**：accounts = [[2,8,7],[7,1,3],[1,9,5]]
> **输出**：17

提示：

- m==accounts.length
- n==accounts[i].length
- 1<=m,n<=50
- 1<=accounts[i][j]<=100

问题分析

写了那么多题解，这应该是所有题解中最简单的一道题了。只需要计算每一个客户的所有资产，然后保留最大的即可。

```
public int maximumWealth(int[][] accounts) {
  int max = 0;
  for (int i = 0; i < accounts.length; i++) {
    int temp = 0;//统计每一个客户的所有资产
    for (int j = 0; j < accounts[i].length; j++) {
      temp += accounts[i][j];
    }
    //保留最大值
    max = Math.max(max, temp);
  }
  return max;
}
```


往期推荐

- 486，动态规划解最大子序和
- 451，回溯和位运算解子集
- 398，双指针求无重复字符的最长子串
- 443，滑动窗口最大值

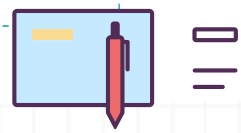
524，爱生气的书店老板

原创 博哥 数据结构和算法 5天前

收录于话题
#算法图文分析

137个 >

> Some people choose to see the ugliness in this world.
>
> The disarray. I choose to see the beauty.
>
> 一些人选择去看见这个世界的丑陋，混乱，我选择去发现美好。



问题描述

今天，书店老板有一家店打算试营业customers.length分钟。每分钟都有一些顾客（customers[i]）会进入书店，所有这些顾客都会在那一分钟结束后离开。

在某些时候，书店老板会生气。如果书店老板在第i分钟生气，那么 grumpy[i]=1，否则grumpy[i]=0。当书店老板生气时，那一分钟的顾客就会不满意，不生气则他们是满意的。

书店老板知道一个秘密技巧，能抑制自己的情绪，可以让自己连续X分钟不生气，但却只能使用一次。

请你返回这一天营业下来，最多有多少客户能够感到满意的数量。

示例：

> **输入**：customers=[1,0,1,2,1,1,7,5],
> grumpy=[0,1,0,1,0,1,0,1],X=3
>
> **输出**：16
> **解释**：
> 书店老板在最后3分钟保持冷静。
> 感到满意的最大客户数量=1+1+1+1+7+5=16.

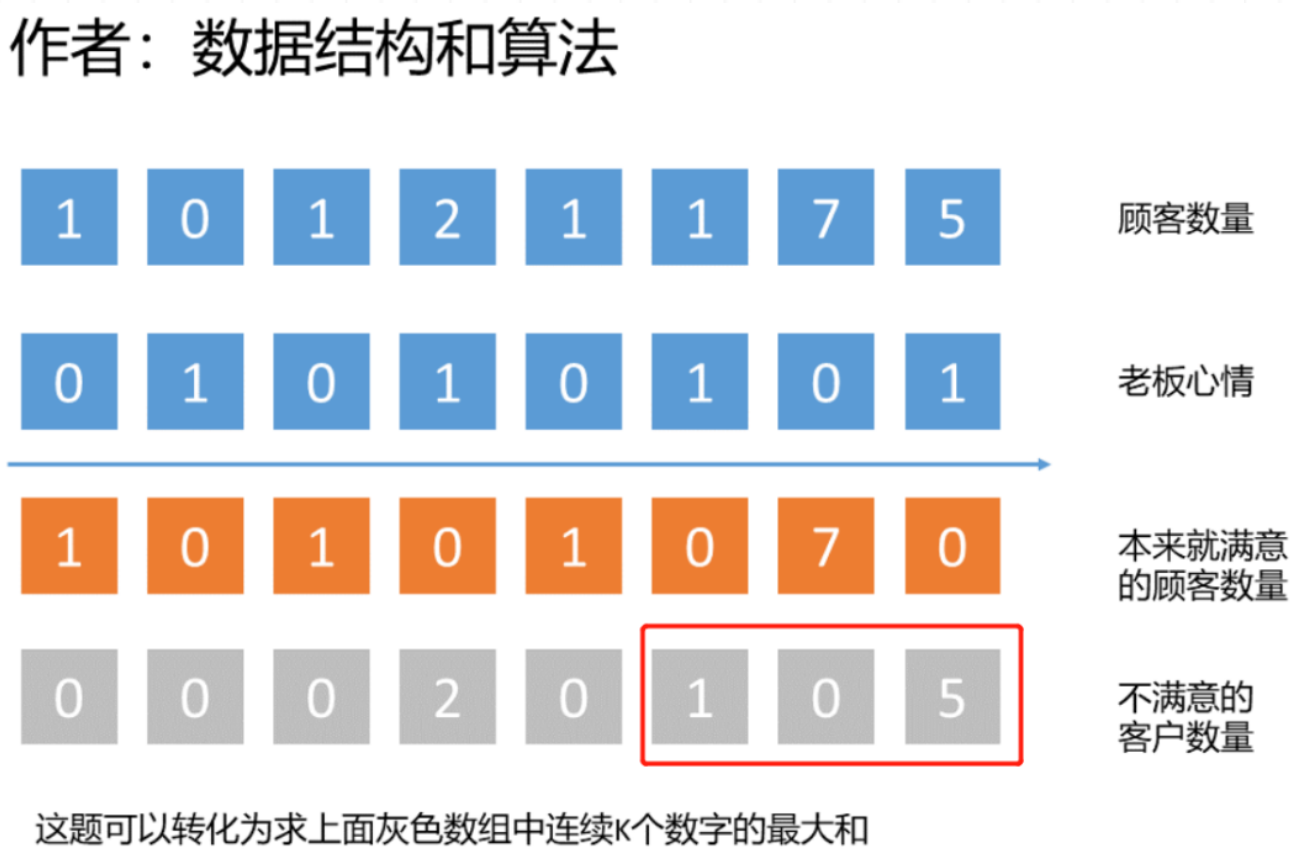
提示：

- 1<=X<=customers.length==grumpy.length<=20000
- 0<=customers[i]<=1000
- 0<=grumpy[i]<=1

窗口内的最大值

数组customers中的每个值表示每分钟内进来的客户量，数组grumpy中的每个值表示每分钟内老板是否生气。

- 如果老板不生气，那么顾客肯定是满意的，我们先计算所有满意的数量。
- 如果老板生气，那么顾客是不满意的，这些不满意的可以构成一个新的数组。而老板可以控制K分钟不生气，这个K分钟我们可以把它当做一个窗口，题目就转化为求这个新的数组中连续K个数字的最大和。



来看下代码

```
public int maxSatisfied(int[] customers, int[] grumpy, int X) {
  int satisfied = 0;//先统计本来就满意的
  int length = grumpy.length;
  //新的数组，统计不满意的
  int[] noSatisfied = new int[length];
  for (int i = 0; i < length; i++) {
    if (grumpy[i] == 0)
      satisfied += customers[i];
    else
```

```
            noSatisfied[i] = customers[i] * grumpy[i];
        }
        //使用两个指针，类似于窗口的左边和右边
        int left = 0;
        int right = 0;
        int max = 0;//记录窗口内的最大值
        int sum = 0;//记录当前窗口内的值
        for (; right < length; right++) {
            sum += noSatisfied[right];
            //如果窗口长度超过K，要减去窗口左边的值，同时
            //窗口左边要往右移一步
            if (right - left >= X) {
                sum -= noSatisfied[left++];
            }
            //保存最大值
            max = Math.max(max, sum);
        }
        //本来就满意的+老板控制情绪让顾客满意的
        return satisfied + max;
    }
```

上面计算的时候我们使用了两个for循环，实际上我们还可以把他们合并成一个

```
public int maxSatisfied(int[] customers, int[] grumpy, int X) {
    int satisfied = 0;//本来就满意的
    int maxPretendSatisfied = 0;//最大抑制情绪满意的
    int pretendSatisfied = 0;//窗口内抑制情绪满意的
    for (int i = 0; i < grumpy.length; ++i) {
        //如果grumpy[i]是0，表示顾客是满意的
        if (grumpy[i] == 0) {
            satisfied += customers[i];
        } else {
            //如果不等于0，表示顾客是不满意的，但老板可以控制自己的
            //情绪，顾客表示假装满意
            pretendSatisfied += customers[i];
        }
        //老板控制自己的情绪是有限的，这个范围我们可以把它看做是一个窗口，
        //这个窗口是一直往右移动的，如果移除窗口的有不满意的，要减去
        if (i >= X && grumpy[i - X] == 1) {
            pretendSatisfied -= customers[i - X];
        }
        //保存通过抑制情绪使顾客满意的最大数量
        maxPretendSatisfied = Math.max(maxPretendSatisfied, pretendSatisfied);
    }
    //最后返回本来就使顾客满意的数量+抑制情绪使顾客满意的数量
    return satisfied + maxPretendSatisfied;
}
```

总结

顾客本来就满意的只需要累加即可，而顾客不满意的，老板可以通过控制自己的情绪让顾客满意，我们只需要求这连续K个不满意的最大值，问题就很容易解决了。

往期推荐

- 443，滑动窗口最大值
- 407，动态规划和滑动窗口解决最长重复子数组

521，滑动窗口解最大连续1的个数 III

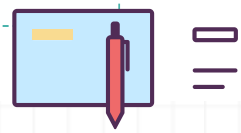
原创 博哥 数据结构和算法 1周前

收录于话题
#算法图文分析

137个 >

There is no pressure when you are making a dream come true.

当你是在为梦想成真努力时，就不会有压力。



问题描述

给定一个由若干0和1组成的数组A，我们最多可以将K个值从0变成1。
返回仅包含1的最长（连续）子数组的长度。

示例 1：

输入：
A=[1,1,1,0,0,0,1,1,1,1,0],K=2

输出：6
解释：
[1,1,1,0,0,1,1,1,1,1,1]
粗体数字从0翻转到1，最长的子数组长度为6。

示例 2：

输入：
A=[0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1],
K=3

输出：10
解释：
[0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1]
粗体数字从0翻转到1，最长的子数组长度为10。

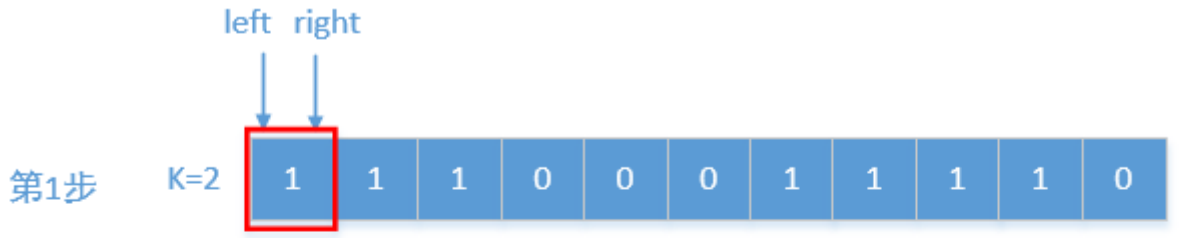
提示：

1. 1<=A.length<=20000
2. 0<=K<=A.length
3. A[i]为0或1

滑动窗口解决

这题让求的是仅包含1的最长连续子数组，并且我们还有魔法，可以把K个0变为1。这题使用滑动窗口解决应该是最容易理解的。

我们可以使用两个指针，一个指向窗口的左边，一个指向窗口的右边，每次遍历数组的时候窗口左边的指针先不动，窗口右边的指针始终都会往右移动，然后顺便统计窗口内0的个数，如果0的个数大于K的时候，说明我们即使使用魔法，也不能把窗口内的所有数字都变为1，这个时候我们在移动窗口左边的指针，直到窗口内0的个数不大于K为止……，具体可以参照下图



窗口大小是1，0的个数是0，不大于K，满足条件，左边不动，右边往右移，扩大窗口

maxWindow=1

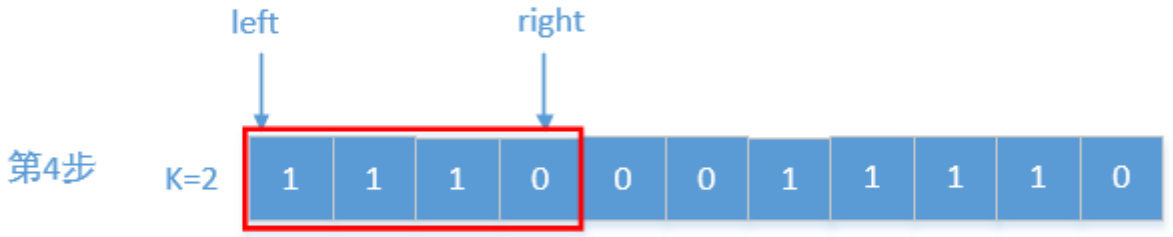
第2步略 ……

maxWindow=2



窗口大小是3，0的个数是0，不大于K，满足条件，左边不动，右边往右移，扩大窗口

maxWindow=3



窗口大小是4，0的个数是1，不大于K，满足条件，左边不动，右边往右移，扩大窗口

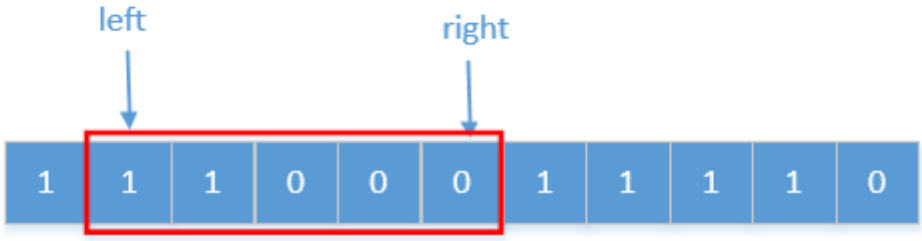
maxWindow=4

第5步略 ……

maxWindow=5

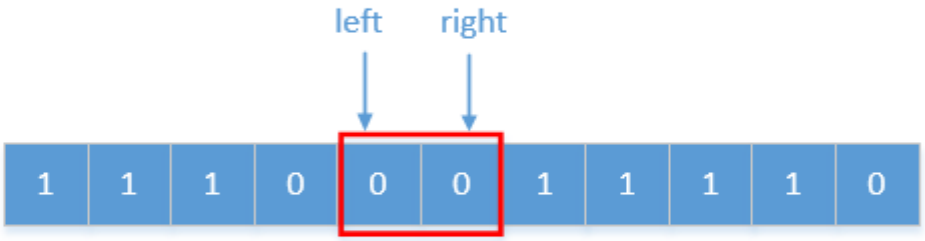


窗口大小是6，0的个数是3，大于K，不满足条件，
要缩小窗口的大小，直到0的个数不大于K为止



窗口大小是5，0的个数是3，大于K，不满足条件，
继续缩小窗口

.....



窗口大小是2，0的个数是2，不大于K，满足条件

..... 继续扩大窗口，后面就不在画了

来看下代码

```
public int longestOnes(int[] A, int K) {
  int left = 0;//窗口左边的位置
  int maxWindow = 0;//窗口的最大值
  int zeroCount = 0;//窗口中0的个数
  for (int right = 0; right < A.length; right++) {
    if (A[right] == 0) {
      zeroCount++;
    }
    //如果窗口中0的个数超过了K，要缩小窗口的大小，直到0的个数
    //不大于K位置
    while (zeroCount > K) {
      if (A[left++] == 0)
        zeroCount--;
    }
```

```
15        //记录最大的窗口
16        maxWindow = Math.max(maxWindow, right - left + 1);
17      }
18      return maxWindow;
19    }
```

其实还可以换种思路，当窗口内0的个数刚好大于K的时候（也就是zeroCount+1==K），说明这个时候right指向的肯定是0，那么目前为止最大的窗口大小是(right-1)-left，因为窗口的右指针是一直往右滑动的，我们可以通过改变左指针的位置来缩小窗口。

所以right-left(注意这里的left已经执行++了，在下面的第10行)始终指向的是最大窗口的值，最后我们只需要返回right-left即可，不需要while循环，来看下代码

```
1   public int longestOnes(int[] A, int K) {
2     int left = 0;//窗口左边的位置
3     int right = 0;//窗口右边的位置
4     int zeroCount = 0;//窗口中0的个数
5     for (; right < A.length; right++) {
6       if (A[right] == 0) {
7         zeroCount++;
8       }
9       //如果窗口中0的个数超过了K，要缩小窗口的大小
10      if (zeroCount > K && A[left++] == 0)
11        zeroCount--;
12    }
13    return right - left;
14  }
```

或者还可以更简洁一些，其实原理都一样，换汤不换药。

```
1   public int longestOnes(int[] A, int K) {
2     int left = 0;//窗口左边的位置
3     int right = 0;//窗口右边的位置
4     int zeroCount = 0;//窗口中0的个数
5     for (; right < A.length; right++) {
6       zeroCount += 1 - A[right];
7       if (zeroCount > K)
8         zeroCount -= 1 - A[left++];
9     }
10    return right - left;
11  }
```

总结

滑动窗口和回溯算法其实都有一个经典的模板，对于回溯算法可以看下450，什么叫回溯算法，一看就会，一写就废。而滑动窗口问题，首先要使用两个指针，一个确定窗口的左边界，一个确定窗口的右边界，其中左边界不动，右边界往右移动，每移动一步都要判断窗口内的值是否满足条件，如果满足，要记录下最优值。如果不满足，左边界在开始移动，相当于缩小窗口……。滑动窗口的题有很多，有时间再对滑动窗口做个总结。

518，托普利茨矩阵

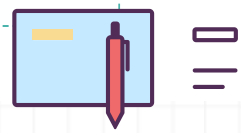
原创 博哥 数据结构和算法 2月23日

收录于话题
#算法图文分析

137个 >

Sometimes it is better to lose and do the right thing than to win and do the wrong thing.

有时候做对的事而输，比做错的事而赢还要好。



问题描述

给你一个mxn的矩阵matrix。如果这个矩阵是托普利茨矩阵，返回true；否则，返回false 。

如果矩阵上每一条由左上到右下的对角线上的元素都相同，那么这个矩阵是托普利茨矩阵。

示例 1：

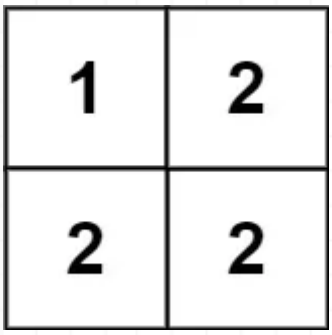
1	2	3	4
5	1	2	3
9	5	1	2

输入：
matrix = [[1,2,3,4],[5,1,2,3],[9,5,1,2]]

输出：true

> **解释**：
> 在上述矩阵中，其对角线为：
> "[9]"，"[5，5]"，"[1，1，1]"，"[2，2，2]"，"[3，3]"，"[4]"。
> 各条对角线上的所有元素均相同，因此答案是 True 。

示例 2：



> **输入**：matrix = [[1,2],[2,2]]
> **输出**：false
> **解释**：
> 对角线 "[1，2]" 上的元素不同。

提示：

- m == matrix.length
- n == matrix[i].length
- 1 <= m, n <= 20
- 0 <= matrix[i][j] <= 99

问题分析

这题比较简单，可以使用两层循环，除了最后一行和最后一列外，其他的每个元素都要和右下角的比较，如果不相同直接返回false。最后一行和最后一列因为没有右下角的元素，所以不需要比较，看下视频



再来看下代码

```java
public boolean isToeplitzMatrix(int[][] matrix) {
  //最后一行和最后一列因为没有右下角的元素，不需要比较
  int row = matrix.length - 1, column = matrix[0].length - 1;
  for (int i = 0; i < row; i++) {
    for (int j = 0; j < column; j++) {
      //当前元素和右下角比较，如果不一样直接返回false
      if (matrix[i][j] != matrix[i + 1][j + 1]) {
        return false;
      }
    }
  }
  return true;
}
```

总结

这题没什么难度，代码也比较简单。

往期推荐

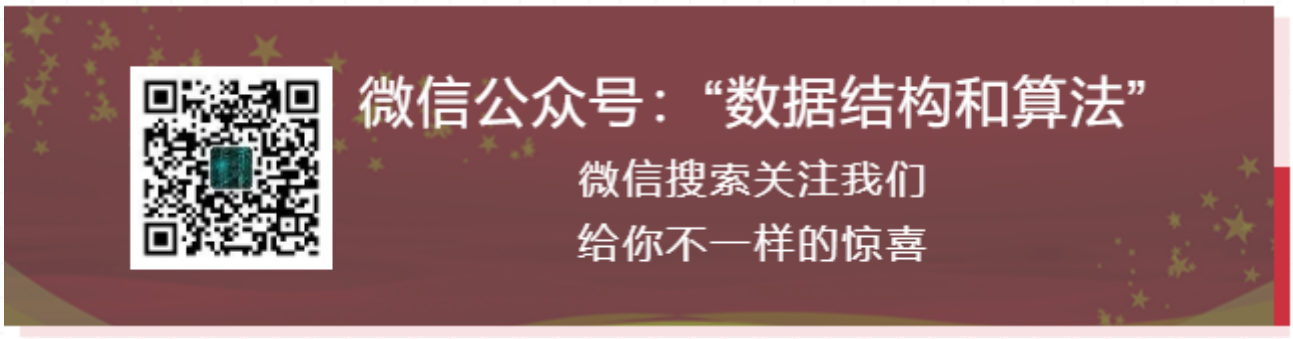
- 482，上升下降字符串
- 478，回溯算法解单词搜索
- 447，双指针解旋转链表
- 374，二叉树的最小深度

511，独一无二的出现次数

原创 山大王wld 数据结构和算法 1月23日

收录于话题
#算法图文分析

137个 >



The size of your dreams must always exceed your current capacity to achieve them.

你的梦想应该总是比当前的能力要大。

问题描述

给你一个整数数组arr，请你帮忙统计数组中每个数的出现次数。

如果每个数的出现次数都是独一无二的，就返回 true；否则返回 false。

示例 1：

> **输入**：arr = [1,2,2,1,1,3]
> **输出**：true
> **解释**：在该数组中，1 出现了 3 次，2 出现了 2 次，3 只出现了 1 次。没有两个数的出现次数相同。

示例 2：

> **输入**：arr = [1,2]
> **输出**：false

示例 3：

> **输入：** arr = [-3,0,1,-3,1,1,1,-3,10,0]
> **输出：** true

提示：

- 1 <= arr.length <= 1000
- -1000 <= arr[i] <= 1000

使用Map解决

这题让判断每个数字出现的次数是否都不一样。如果都不一样，返回true，否则返回false。所以第一步应该先统计每个数字出现的次数，然后再判断这些次数是否有重复的。统计的时候可以使用Map，判断是否有重复的可以使用集合Set。

我们知道集合set是不能有重复元素的，如果有就会替换掉，我们可以把出现次数的数组放到集合set中，如果有重复的就会被替换掉，那么set的大小肯定和出现次数的数组长度不一样。否则如果没有重复的，他们的长度肯定是一样的，看下代码。

```
public boolean uniqueOccurrences(int[] arr) {
  //map统计每个数字出现的次数
  Map<Integer, Integer> map = new HashMap<>();
  for (int i = 0; i < arr.length; i++) {
    map.put(arr[i], map.getOrDefault(arr[i], 0) + 1);
  }
  //map中的key是数字，value是数字出现的次数
  return map.size() == new HashSet<>(map.values()).size();
}
```

上面的方式还可以稍微修改一下，在set集合中如果有相同的元素，就会存储失败，返回false，每次存储的时候我们只要判断是否存储成功即可，代码如下

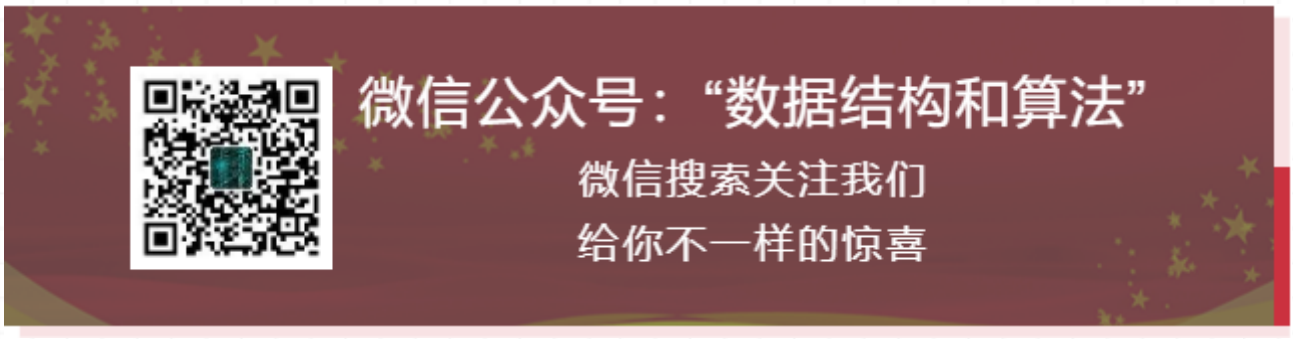
```
public boolean uniqueOccurrences(int[] arr) {
  Map<Integer, Integer> map = new HashMap<>();
  for (int i = 0; i < arr.length; i++) {
    map.put(arr[i], map.getOrDefault(arr[i], 0) + 1);
  }
  Set<Integer> set = new HashSet<>();
  for (int value : map.values()) {
    //如果存储失败，说明有重复的
    if (!set.add(value))
      return false;
  }
  return true;
}
```

509，数组中的第K个最大元素

原创 山大王wld 数据结构和算法 1月18日

收录于话题
#算法图文分析

137个 >



The old dreams were good dreams. They didn't work out, but I'm glad I had them.

曾经的梦都是美梦，虽未成真，但庆幸我曾拥有过。

问题描述

在未排序的数组中找到第**k**个最大的元素。请注意，你需要找的是数组排序后的第k个最大的元素，而不是第k个不同的元素。

示例 1:

> **输入**: [3,2,1,5,6,4] 和 k = 2
> **输出**: 5

示例 2:

> **输入**: [3,2,3,1,2,4,5,5,6] 和 k = 4
> **输出**: 4

说明:

你可以假设 k 总是有效的，且 1 ≤ k ≤ 数组的长度。

先排序再查找

这题是让找出排序后的第k个最大的元素，所以最简单的一种方式就是先对数组进行排序，然后再查找。关于排序，我公众号之前介绍了有十几种排序算法，具体可以在公众号的目录中查看。代码比较简单，我们来看一下

```
public int findKthLargest(int[] nums, int k) {
  Arrays.sort(nums);//先排序
  return nums[nums.length - k];//在查找
}
```

使用最小堆

这题只让找出最大的第k个元素即可，没说一定要对数组进行排序，所以我们还可以使用最小堆来解决。解决方式就是一个个遍历原数组的值，添加到堆中，添加之后如果堆中元素个数大于k的时候，我们就把最顶端的元素给移除掉，因为是最小堆，所以移除的就是堆中最小的值。

```
public int findKthLargest(int[] nums, int k) {
  final PriorityQueue<Integer> queue = new PriorityQueue<>();
  for (int val : nums) {
    queue.add(val);//加入堆中
    //如果堆中元素大于k，则把堆顶元素给移除
    if (queue.size() > k)
      queue.poll();
  }
  return queue.peek();//返回堆顶元素
}
```

参考快速排序

快速排序是先选择一个中枢（一般我们选第一个），然后遍历后面的元素，最终会把数组分为两部分，前面部分比中枢值小，后面部分大于或等于中枢值。

1. 分开之后中枢值所在的位置如果从后面数是第k个，我们直接返回中枢值即可。
2. 如果从后面数大于k，说明要找的值还在后面这部分，我们只需按照同样的方式从后面部分开始找即可。
3. 如果从后面数小于k，说明要找的值在前面部分，我们同样从前面部分开始查找。

原理比较简单，我们来看下代码

```
public int findKthLargest(int[] nums, int k) {
  k = nums.length - k;//注意这里的k已经变了
  int lo = 0, hi = nums.length - 1;
  while (lo <= hi) {
    int i = lo;
    //这里把数组以A[lo]的大小分为两部分，
```

```
      //一部分是小于A[lo]的，一部分是大于A[lo]的
      // [lo,i]<A[lo]，[i+1,j)>=A[lo]
      for (int j = lo + 1; j <= hi; j++)
        if (nums[j] < nums[lo])
          swap(nums, j, ++i);
      swap(nums, lo, i);
      if (k == i)
        return nums[i];
      else if (k < i)
        hi = i - 1;
      else
        lo = i + 1;
    }
    return -1;
  }

  //交换两个元素的值
  private void swap(int[] nums, int i, int j) {
    if (i != j) {
      nums[i] ^= nums[j];
      nums[j] ^= nums[i];
      nums[i] ^= nums[j];
    }
  }
```

上面swap方法是交换两个数字的值，一般情况下我们会使用一个临时变量temp来解决，哪种方式都是可以了，具体可以看下357，交换两个数字的值，这里介绍了几种交换的方式。

往期推荐

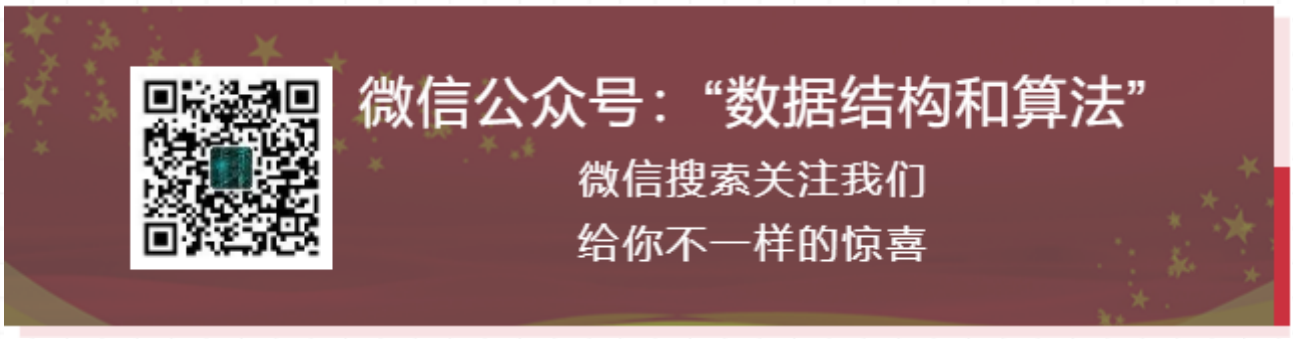
- 498，回溯算法解活字印刷
- 451，回溯和位运算解子集
- 450，什么叫回溯算法，一看就会，一写就废
- 446，回溯算法解黄金矿工问题

506，无重叠区间

原创 山大王wld 数据结构和算法 1月12日

收录于话题
#算法图文分析

137个 >



You deserve someone who loves you with every beat of his heart.

你值得拥有一个全心全意爱你的人。

问题描述

给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。

注意:

1. 可以认为区间的终点总是大于它的起点。
2. 区间 [1,2] 和 [2,3] 的边界相互“接触”，但没有相互重叠。

示例 1:

> **输入**: [[1,2], [2,3], [3,4], [1,3]]
> **输出**: 1
> **解释**: 移除 [1,3] 后，剩下的区间没有重叠。

示例 2:

> **输入**: [[1,2], [1,2], [1,2]]
> **输出**: 2
> **解释**: 你需要移除两个 [1,2] 来使剩下的区间没有重叠。

示例 3:

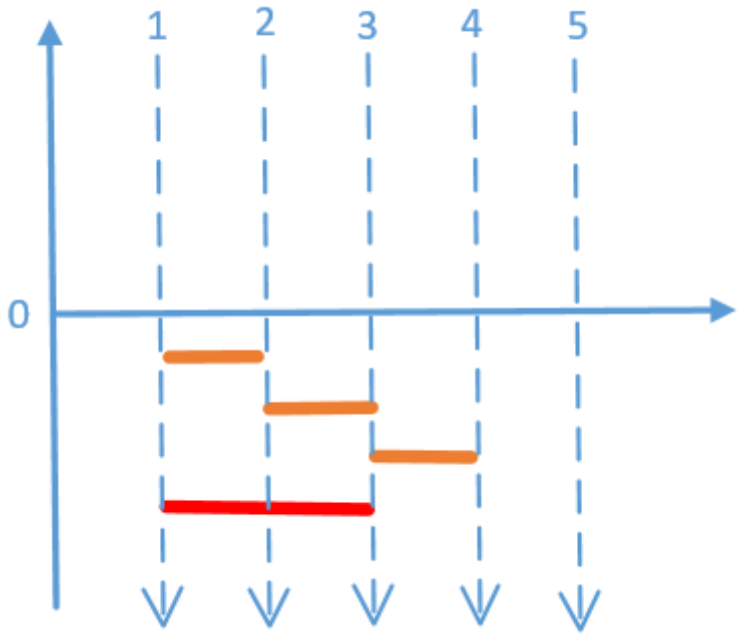
> **输入**: [[1,2], [2,3]]
> **输出**: 0
> **解释**: 你不需要移除任何区间，因为它们已经是无重叠的了。

问题分析

这题是让移除最少的区间，然后使剩下的区间不重合。

首先要对区间进行排序，这里先以区间的头来排序，然后在遍历区间。
1，如果后面区间的头小于当前区间的尾，
比如当前区间是[3,6]，后面区间是[4,5]或者是[5,9]
说明这两个区间有重叠，必须要移除一个，那么要移除哪个呢，为了防止在下一个区间和现有区间有重叠，我们应该让现有区间越短越好，所以应该移除尾部比较大的，保留尾部比较小的。
2，如果后面区间的头不小于当前区间的尾，说明他们没有重叠，不需要移除

如下图区间[1,2]和[1,3]有了重叠，我们要移除尾部比较大的，也就是红色的[1,3]区间



代码如下

```java
public int eraseOverlapIntervals(int[][] intervals) {
    if (intervals.length == 0)
        return 0;
    //先排序
    Arrays.sort(intervals, (a, b) -> a[0] - b[0]);
    //记录区间尾部的位置
    int end = intervals[0][1];
    //需要移除的数量
    int count = 0;
    for (int i = 1; i < intervals.length; i++) {
        if (intervals[i][0] < end) {
            //如果重叠了，必须要移除一个，所以count要加1，
            //然后更新尾部的位置，我们取尾部比较小的
            end = Math.min(end, intervals[i][1]);
            count++;
        } else {
            //如果没有重叠，就不需要移除，只需要更新尾部的位置即可
            end = intervals[i][1];
        }
    }
    return count;
}
```

总结

这题关键点在于排序之后要怎么确定两个区间是否相交，确定之后就简单了，如果相交我们只需要移除尾部较大的即可。

往期推荐

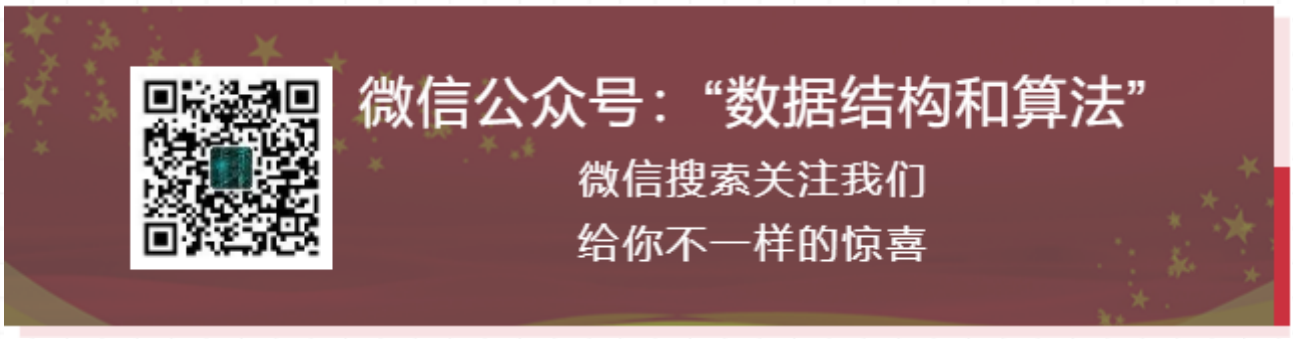
- 486，动态规划解最大子序和
- 498，回溯算法解活字印刷
- 445，BFS和DFS两种方式解岛屿数量
- 469，位运算求最小的2的n次方

504，旋转数组的3种解决方式

原创 山大王wld 数据结构和算法 1月10日

收录于话题
#算法图文分析

137个 >



There are many things that seem impossible only so long as one does not attempt them.

很多事情看起来不可能只是因为没有人尝试过。

问题描述

给定一个数组，将数组中的元素向右移动k个位置，其中k是非负数。

示例 1:

> **输入**: nums = [1,2,3,4,5,6,7], k = 3
> **输出**: [5,6,7,1,2,3,4]
> **解释**:
> 向右旋转 1 步: [7,1,2,3,4,5,6]
> 向右旋转 2 步: [6,7,1,2,3,4,5]
> 向右旋转 3 步: [5,6,7,1,2,3,4]

示例 2:

> **输入**: nums = [-1,-100,3,99], k = 2

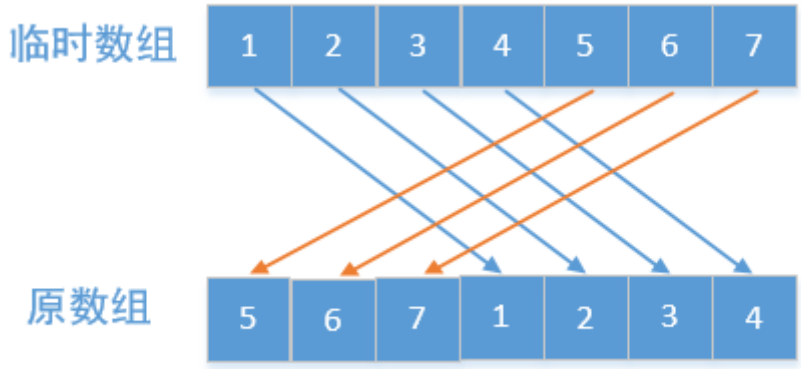
> **输出**：[3,99,-1,-100]
> **解释**：
> 向右旋转 1 步：[99,-1,-100,3]
> 向右旋转 2 步：[3,99,-1,-100]

提示：

- 1<=nums.length<=2*10^4
- -2^31<=nums[i]<=2^31-1
- 0<=k<=10^5

使用临时数组解决

这题是让把数组中的每个元素都往右移动k位。最简单的一种解决方式就是使用一个临时数组解决，先把原数组的值存放到一个临时数组中，然后再把临时数组的值重新赋给原数组，重新赋值的时候要保证每个元素都要往后移k位，如果超过数组的长度就从头开始，所以这里可以使用(i + k) % length来计算重新赋值的元素下标




```java
public void rotate(int nums[], int k) {
  int length = nums.length;
  int temp[] = new int[length];
  // 把原数组值放到一个临时数组中，
  for (int i = 0; i < length; i++) {
    temp[i] = nums[i];
  }
  // 然后在把临时数组的值重新放到原数组，
  // 并且往右移动k位
  for (int i = 0; i < length; i++) {
    nums[(i + k) % length] = temp[i];
  }
}
```

部分元素多次反转

还有一种方式就是先反转全部数组，在反转前k个，最后在反转剩余的，如下所示



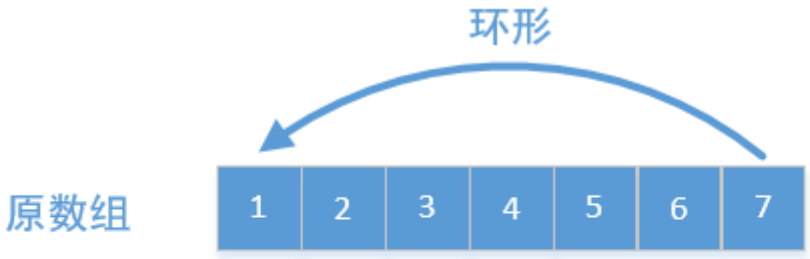

```java
public void rotate(int[] nums, int k) {
    int length = nums.length;
    k %= length;
    //先反转全部的元素
    reverse(nums, 0, length - 1);
    //在反转前k个元素
    reverse(nums, 0, k - 1);
    //接着反转剩余的
    reverse(nums, k, length - 1);
}

//把数组中从[start，end]之间的元素两两交换,也就是反转
public void reverse(int[] nums, int start, int end) {
    while (start < end) {
        int temp = nums[start];
        nums[start++] = nums[end];
        nums[end--] = temp;
    }
}
```

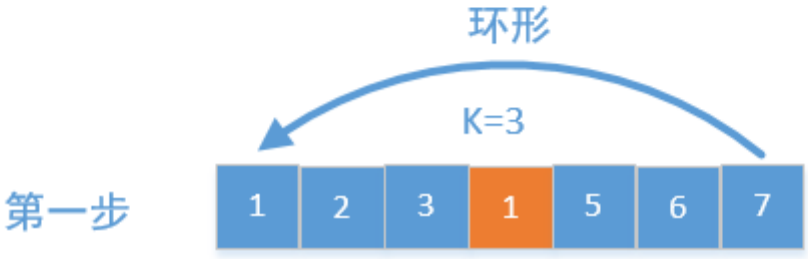
其实还可以在调整下，先反转前面的，接着反转后面的k个，最后在反转全部，原理都一样

```java
public void rotate(int[] nums, int k) {
    int length = nums.length;
    k %= length;
    //先反转前面的
    reverse(nums, 0, length - k - 1);
    //接着反转后面k个
    reverse(nums, length - k, length - 1);
    //最后在反转全部的元素
    reverse(nums, 0, length - 1);
}

//把数组中从[start，end]之间的元素两两交换,也就是反转
public void reverse(int[] nums, int start, int end) {
    while (start < end) {
        int temp = nums[start];
        nums[start++] = nums[end];
        nums[end--] = temp;
    }
}
```

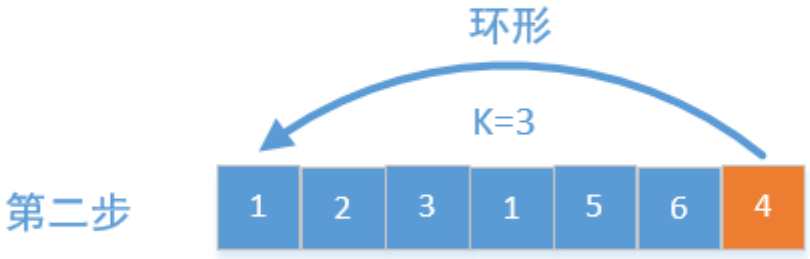
环形旋转

类似约瑟夫环一样，把数组看作是环形的，每一个都往后移动k位，这个很好理解，画个图来看一下





把位置4的值保存下来，
hold=4，把1放到4的位置



把位置7的值保存下来，
hold=7，把4放到7的位置

第三步



把位置3的值保存下来，
hold=3，把7放到3的位置

第四步



把位置6的值保存下来，
hold=6，把3放到6的位置

第五步

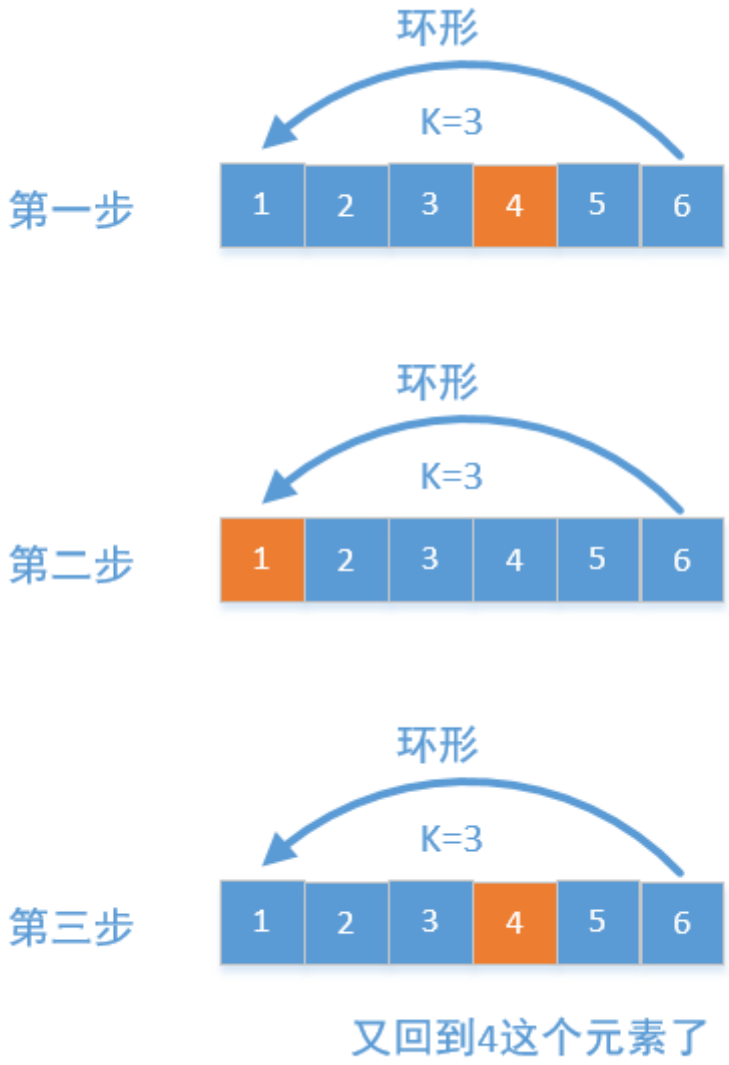


把位置2的值保存下来，
hold=2，把6放到2的位置





但这里有一个坑，如果 `nums.length%k=0` ，也就是数组长度为k的倍数，这个会原地打转，如下图所示



对于这个问题我们可以使用一个数组visited表示这个元素有没有被访问过，如果被访问过就从他的下一个开始，防止原地打转。

```java
public static void rotate(int[] nums, int k) {
    int hold = nums[0];
    int index = 0;
    int length = nums.length;
    boolean[] visited = new boolean[length];
    for (int i = 0; i < length; i++) {
        index = (index + k) % length;
        if (visited[index]) {
            //如果访问过，再次访问的话，会出现原地打转的现象，
            //不能再访问当前元素了，我们直接从他的下一个元素开始
            index = (index + 1) % length;
            hold = nums[index];
            i--;
        } else {
            //把当前值保存在下一个位置，保存之前要把下一个位置的
            //值给记录下来
            visited[index] = true;
            int temp = nums[index];
            nums[index] = hold;
            hold = temp;
        }
    }
}
```

总结

这题使用前两种方式是最容易想到的，也是比较简单的，第3种方式也容易想到，但操作起来可能稍微有点难度。

往期推荐

- 491，回溯算法解将数组拆分成斐波那契序列
- 475，有效的山脉数组
- 419，剑指 Offer-旋转数组的最小数字
- 407，动态规划和滑动窗口解决最长重复子数组

496，字符串中的第一个唯一字符

原创 山大王wld 数据结构和算法 1周前

收录于话题
#算法图文分析

111个 >



The only real failure is the failure to try.

真正的失败是未曾尝试。

问题描述

给定一个字符串，找到它的第一个不重复的字符，并返回它的索引。如果不存在，则返回 -1。

示例：

```
s = "leetcode"
返回 0

s = "loveleetcode"
返回 2
```

提示：你可以假定该字符串只包含小写字母。

两次遍历解决

这题让找出第一个不重复的字符，可能最简单的方式就是暴力查找，类似于冒泡排序一样，使用两个嵌套的for循环，但这种效率很差。

除此之外还有另一种方式，也是使用两个for循环，但不是嵌套的，第一个for循环先统计每个字符出现的次数，第二个for循环再从前往后遍历字符串s中的每个字符，如果某个字符出现一次直接返回，原理比较简单，看下代码

```
public int firstUniqChar(String s) {
    int count[] = new int[26];
    char[] chars = s.toCharArray();
    //先统计每个字符出现的次数
    for (int i = 0; i < s.length(); i++)
        count[chars[i] - 'a']++;
    //然后在遍历字符串s中的字符，如果出现次数是1就直接返回
    for (int i = 0; i < s.length(); i++)
        if (count[chars[i] - 'a'] == 1)
            return i;
    return -1;
}
```

使用HashMap解决

也是换汤不换药，原理和上面的一样，先统计每个字符的数量，然后在查找。

```
public int firstUniqChar(String s) {
    Map<Character, Integer> map = new HashMap();
    char[] chars = s.toCharArray();
    //先统计每个字符的数量
    for (char ch : chars) {
        map.put(ch, map.getOrDefault(ch, 0) + 1);
    }
    //然后在遍历字符串s中的字符，如果出现次数是1就直接返回
    for (int i = 0; i < s.length(); i++) {
        if (map.get(chars[i]) == 1) {
            return i;
        }
    }
    return -1;
}
```

使用indexOf方法

在Java中String有这样两个方法，一个是indexOf，表示的是从前面查找字符在字符串中第一次出现的位置。一个是lastIndexOf，从后面查找字符在字符串中第一次出现的位置。一个从前查找，一个从后查找，如果位置相等，说明只出现了一次。

```
public int firstUniqChar(String s) {
    for (int i = 0; i < s.length(); i++)
        if (s.indexOf(s.charAt(i)) == s.lastIndexOf(s.charAt(i)))
            return i;
    return -1;
}
```

总结

今天这道题算是比较简单的一道题，基本上没什么难度。

往期推荐

- 486，动态规划解最大子序和
- 464. BFS和DFS解二叉树的所有路径
- 446，回溯算法解黄金矿工问题
- 469，位运算求最小的2的n次方

487，重构字符串

原创 山大王wld 数据结构和算法 今天

收录于话题
#算法图文分析

97个 >



You can nearly always enjoy something if you make up your mind firmly that you will.

只要你下定决心做某件事，总能从中找到乐趣。

问题描述

给定一个字符串 S ，检查是否能重新排布其中的字母，使得两相邻的字符不同。

若可行，输出任意可行的结果。若不可行，返回空字符串。

示例 1:

> **输入**: S = "aab"
> **输出**: "aba"

示例 2:

> **输入**: S = "aaab"
> **输出**: ""

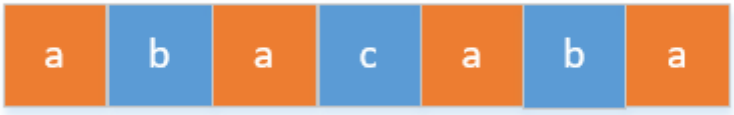
注意:

- S 只包含小写字母并且长度在[1, 500]区间内。

问题分析

这题是让重新排布字符串S中的字符，让任何两个相邻的字符都不相同，如果能做到就返回排布后的字符串，如果做不到就返回空字符串。

如果要使得两相邻的字符不同，那么出现次数最多的那个数的数量必须满足下面条件，如下图所示，比如下面的a是出现次数最多的



这个时候a的数量已经达到了临界值，如果再多一个a，那么至少有两个a是相邻的。所以这里出现次数最多的那个字符数量的临界值是threshold = (length + 1) >> 1（其中 length 是字符串的长度）

如果能使得两相邻的字符不同，我们可以先把出现次数最多的那个字符放到新数组下标为偶数的位置上，也就是从数组的第一个位置开始放，放完之后在用其他的字符填充数组剩下的下标为偶数的位置，如果下标为偶数的位置都填满了，我们就从下标为1开始，也就是数组的第2个位置开始，填下标为奇数的位置。

注意这里能不能先把出现次数最多的字符放到字符串下标为奇数的位置呢，当然是不可以的。比如我们上面举的例子abacaba本来是可以满足的，如果放到下标为奇数的位置，最后一个 a 就没法放了，除非放到最前面，那又变成了放到下标为偶数的位置了。





代码如下

```
1   public String reorganizeString(String S) {
2     //把字符串S转化为字符数组
3     char[] alphabetArr = S.toCharArray();
4     //记录每个字符出现的次数
5     int[] alphabetCount = new int[26];
```

```
    //字符串的长度
    int length = S.length();
    int max = 0, alphabet = 0, threshold = (length + 1) >> 1;
    //找出出现次数最多的那个字符
    for (int i = 0; i < length; i++) {
      alphabetCount[alphabetArr[i] - 'a']++;
      if (alphabetCount[alphabetArr[i] - 'a'] > max) {
        max = alphabetCount[alphabetArr[i] - 'a'];
        alphabet = alphabetArr[i] - 'a';
        //如果出现次数最多的那个字符的数量大于阈值，
        // 说明他不能使得两相邻的字符不同，
        // 直接返回空字符串即可
        if (max > threshold)
          return "";
      }
    }
    //到这一步说明他可以使得两相邻的字符不同，
    // 我们随便返回一个结果，res就是返回
    //结果的数组形式，最后会再转化为字符串的
    char[] res = new char[length];
    int index = 0;
    //先把出现次数最多的字符存储在数组下标为偶数的位置上
    while (alphabetCount[alphabet]-- > 0) {
      res[index] = (char) (alphabet + 'a');
      index += 2;
    }
    //然后再把剩下的字符存储在其他位置上
    for (int i = 0; i < alphabetCount.length; i++) {
      while (alphabetCount[i]-- > 0) {
        //如果偶数位置填完了，我们就让index从1开始，
        // 填充下标为奇数的位置
        if (index >= res.length) {
          index = 1;
        }
        res[index] = (char) (i + 'a');
        index += 2;
      }
    }
    return new String(res);
  }
```

总结

这题直接判断比较简单，我们只需要统计出出现次数最多的字符即可。但这题还要返回结果，所以最简单的方式就是把出现次数最多的字符从数组的第一个位置开始放，每隔一个放一次。放完之后再用其他的字符从后面接着放，也是每隔一个，如果超出数组之后再从数组的第2个位置开始放，也是每隔一个，这样就能保证结果一定不会出错，并且也少了很多的判断。

往期推荐

- 482，上升下降字符串
- 451，回溯和位运算解子集
- 413，动态规划求最长上升子序列
- 398，双指针求无重复字符的最长子串

484，打家劫舍 II

原创 山大王wld 数据结构和算法 昨天

收录于话题
#算法图文分析

95个 >



Sometimes it's about doing the right thing, even if it's painful inside.

有时候就是要做对的事，哪怕内心万分痛苦。

问题描述

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都围成一圈 ，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警 。

给定一个代表每个房屋存放金额的非负整数数组，计算你在不触动警报装置的情况下 ，能够偷窃到的最高金额。

示例 1：

> **输入：** nums = [2,3,2]
> **输出：** 3

> **解释**：你不能先偷窃 1 号房屋（金额 = 2），然后偷窃 3 号房屋（金额 = 2），因为他们是相邻的。

示例 2：

> **输入**：nums = [1,2,3,1]
> **输出**：4
> **解释**：你可以先偷窃 1 号房屋（金额 = 1），然后偷窃 3 号房屋（金额 = 3）。
> 偷窃到的最高金额 = 1 + 3 = 4 。

示例 3：

> **输入**：nums = [0]
> **输出**：0

提示：

- 1 <= nums.length <= 100
- 0 <= nums[i] <= 1000

动态规划解决

我们先来考虑这样一个问题，假如所有的房屋没有围成一个圈，也就是说第一个房屋和最后一个房屋不是挨着的，那么这道题就回退到479，递归方式解打家劫舍和477，动态规划解按摩师的最长预约时间这两道题的解了。

这里来定义一个二维数组dp[length][2]，其中length是房屋的数量。

- dp[i][0]表示不偷当前房屋时能偷到的最高金额
- dp[i][1]表示偷当前房屋时能偷到的最高金额

如果不偷当前房屋，那么前一家偷不偷都是可以的，我们取最大值即可
dp[i][0]=max(dp[i-1][0]，dp[i-1][1]);

如果偷当前房屋，那么前一家肯定是不能偷的，所以
dp[i][1]=dp[i-1][0]+nums[i];

所以递推公式很容易找出来，和第477题完全一样。
边界条件是

- dp[0][0]=0，第一家没偷
- dp[0][1]=nums[0]，第一家偷了

代码如下

```
public int robHelper(int[] nums) {
  //边界条件判断
  if (nums == null || nums.length == 0)
    return 0;
  int length = nums.length;
  int[][] dp = new int[length][2];
  dp[0][0] = 0;//第1家没偷
  dp[0][1] = nums[0];//第1家偷了
  //从第2个开始判断
  for (int i = 1; i < length; i++) {
    //下面两行是递推公式
    dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1]);
    dp[i][1] = dp[i - 1][0] + nums[i];
  }
  //最后取最大值即可
  return Math.max(dp[length - 1][0], dp[length - 1][1]);
}
```

当然我们还可以进一步优化，因为上面的二维数组中每次计算当前值的时候只和前面的两个值有关，其他的都不会在用到了，所以这里可以使用两个变量即可，不需要申请一个二维数组。

```
private int robHelper(int[] num) {
  int steal = 0, noSteal = 0;
  for (int j = 0; j < num.length; j++) {
    int temp = steal;
    steal = noSteal + num[j];
    noSteal = Math.max(noSteal, temp);
  }
  return Math.max(steal, noSteal);
}
```

到这里还没完，上面我们假设所有的房屋没有构成一个环，但这道题中所以的房屋是围成一个环形的。

- 如果偷第1家，就不能偷最后一家，所以可偷的范围是[0，length-2]。
- 如果不偷第1家，那么就可以偷最后一家，可偷的范围是[1，length-1]。

所以最终代码如下

```
public int rob(int[] nums) {
  if (nums.length == 1)
    return nums[0];
  //可以偷第一家，但不能偷最后一家
  int robFirst = robHelper(nums, 0, nums.length - 2);
  //可以偷最后一家，但不能偷第一家
  int robLast = robHelper(nums, 1, nums.length - 1);
  //选择偷第1家和不偷第1家结果的最大值
  return Math.max(robFirst, robLast);
}

private int robHelper(int[] num, int start, int end) {
  int steal = 0, noSteal = 0;
  for (int j = start; j <= end; j++) {
    int temp = steal;
    steal = noSteal + num[j];
    noSteal = Math.max(noSteal, temp);
  }
```

```
19      return Math.max(steal, noSteal);
20  }
```

总结

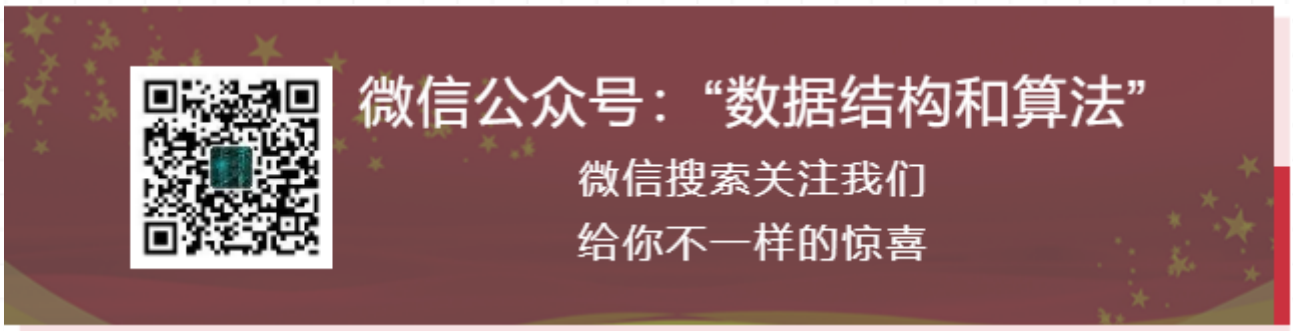
这题与479，递归方式解打家劫舍和477，动态规划解按摩师的最长预约时间其实很相似，代码完全可以照搬，然后再稍加修改即可。这里稍微麻烦一点的是数组是构成一个环，要避免第一个和最后一个同时选择。



- 465. 递归和动态规划解三角形最小路径和
- 430，剑指 Offer-动态规划求正则表达式匹配
- 423，动态规划和递归解最小路径和
- 413，动态规划求最长上升子序列

482，上升下降字符串

原创 山大王wld 数据结构和算法 3天前



Freeing yourself up for something better in the future.

释放自己，为了更好的未来。

问题描述

给你一个字符串s，请你根据下面的算法重新构造字符串：

1. 从s中选出**最小**的字符，将它**接在**结果字符串的后面。
2. 从s剩余字符中选出**最小**的字符，且该字符比上一个添加的字符大，将它**接在**结果字符串后面。
3. 重复步骤2，直到你没法从s中选择字符。
4. 从s中选出**最大**的字符，将它**接在**结果字符串的后面。
5. 从s剩余字符中选出**最大**的字符，且该字符比上一个添加的字符小，将它**接在**结果字符串后面。
6. 重复步骤5，直到你没法从s中选择字符。
7. 重复步骤1到6，直到s中所有字符都已经被选过。

在任何一步中，如果最小或者最大字符不止一个，你可以选择其中任意一个，并将其添加到结果字符串。

请你返回将s中字符重新排序后的**结果字符串** 。

示例 1：

> **输入**：s = "aaaabbbbcccc"

> **输出**：`"abccbaabccba"`
> **解释**：第一轮的步骤 1，2，3 后，结果字符串为 result = "abc"
> 第一轮的步骤 4，5，6 后，结果字符串为 result = "abccba"
> 第一轮结束，现在 s = "aabbcc" ，我们再次回到步骤 1
> 第二轮的步骤 1，2，3 后，结果字符串为 result = "abccbaabc"
> 第二轮的步骤 4，5，6 后，结果字符串为 result = "abccbaabccba"

示例 2：

> **输入**：s = "rat"
> **输出**："art"
> **解释**：单词 "rat" 在上述算法重排序以后变成 "art"

示例 3：

> **输入**：s = "leetcode"
> **输出**："cdelotee"

示例 4：

> **输入**：s = "ggggggg"
> **输出**："ggggggg"

示例 5：

> **输入**：s = "spo"
> **输出**："ops"

提示：

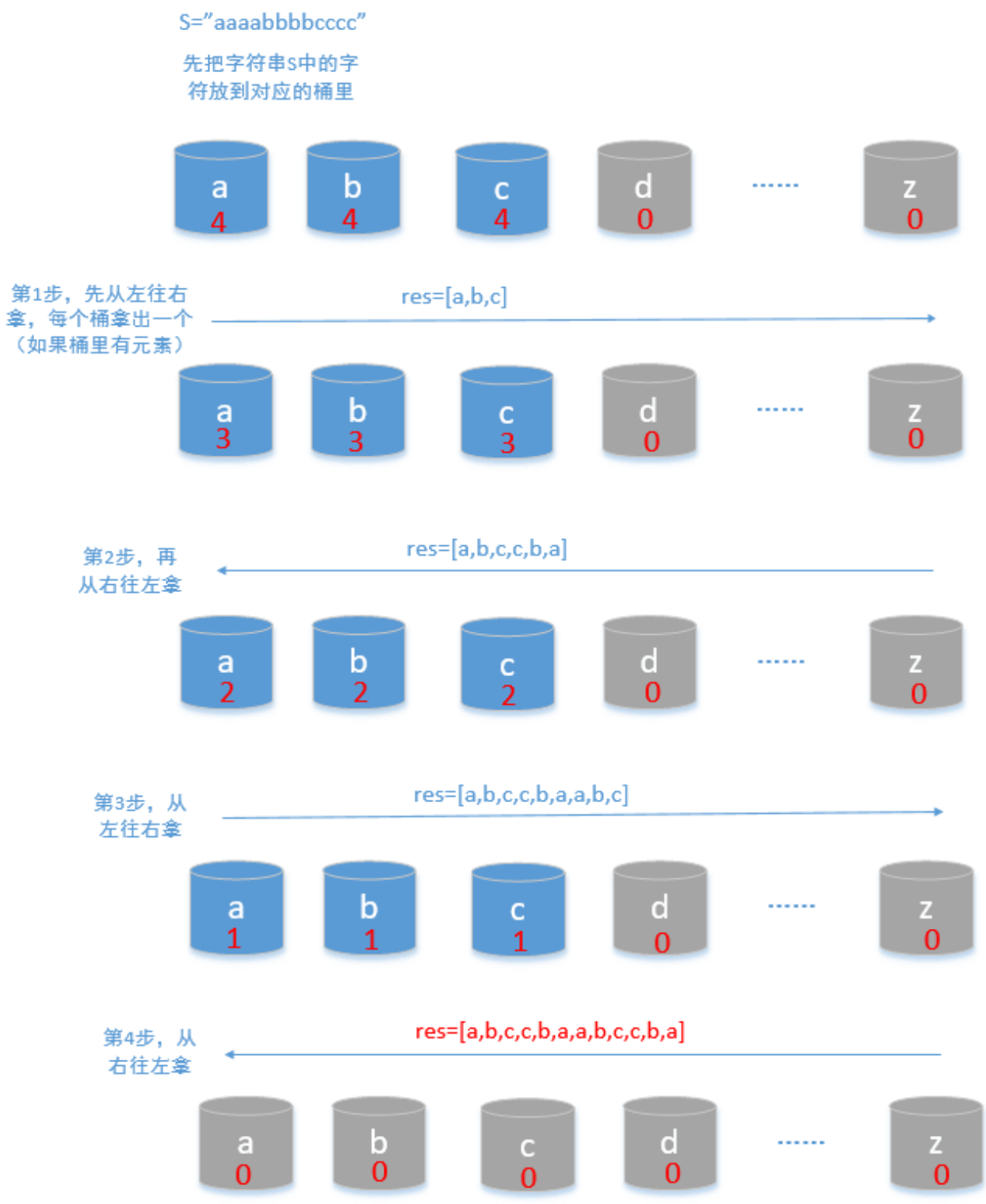
- `1 <= s.length <= 500`
- `s` 只包含小写英文字母。

问题分析

这道题是让从字符串s中先选出升序的字符，然后再选出降序字符……一直这样循环，直到选完为止。因为题中的提示中说了s只包含小写英文字符，我们可以申请一个大小为26的数组，相当于26个桶。

- 把s中的每个字符分别放到对应的桶里，比如a放到第一个桶里，b放到第2个桶里……。
- 第1次从左往右遍历26个桶，从每个桶里拿出一个字符(如果没有就不用拿)
- 第2次从右往左遍历26个桶，从每个桶里拿出一个字符(如果没有就不用拿)
- ……
- 一直这样循环下去，直到所有的桶里的元素都拿完为止。

这里以示例为例，来画个图看下



原理比较简单，来看下代码

```java
public String sortString(String s) {
    //相当于26个桶
    int[] bucket = new int[26];
    char[] charArr = s.toCharArray();
    //把s中的字符分别放到对应的桶里
    for (char c : charArr) {
        bucket[c - 'a']++;
    }
    //存储计算的结果
    char[] res = new char[s.length()];
    int index = 0;
    while (index < s.length()) {
        //先从左往右找，遍历26个桶,如果当前桶不为空，
        //就从当前桶里拿出一个元素出来
        for (int i = 0; i < 26; i++) {
            if (bucket[i] != 0) {
                res[index++] = (char) (i + 'a');
                bucket[i]--;//拿出之后桶中元素的个数要减1
            }
        }
        //从右往左拿，同上
        for (int i = 25; i >= 0; i--) {
            if (bucket[i] != 0) {
                res[index++] = (char) (i + 'a');
                bucket[i]--;
            }
        }
    }
    //把结果转化为字符串
    return new String(res);
}
```

总结

这题算是比较简单，只需要把每个字符都放到对应的桶里，然后每次从左往右把26个桶过一遍，接着在从右往左把26个桶过一遍，直到所有桶里的元素都空为止。

往期推荐

- 481，用最少数量的箭引爆气球
- 464. BFS和DFS解二叉树的所有路径
- 457，二叉搜索树的最近公共祖先
- 451，回溯和位运算解子集

481，用最少数量的箭引爆气球

原创 山大王wld 数据结构和算法 1周前

收录于话题
#算法图文分析

95个 >



A wise man gets more from his enemies than a fool from his friends.

智者从敌人身上学到的比愚者从朋友身上学到的还要多。

问题描述

在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。由于它是水平的，所以纵坐标并不重要，因此只要知道开始和结束的横坐标就足够了。开始坐标总是小于结束坐标。

一支弓箭可以沿着 x 轴从不同点完全垂直地射出。在坐标 x 处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart，xend， 且满足 xstart ≤ x ≤ xend，则该气球会被引爆。可以射出的弓箭的数量没有限制。弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。

给你一个数组points ，其中points[i]=[xstart,xend]，返回引爆所有气球所必须射出的最小弓箭数。

示例 1：

> **输入**：
> points = [[10,16],[2,8],[1,6],[7,12]]
>
> **输出**：2
> **解释**：对于该样例，x = 6 可以射爆 [2,8],[1,6] 两个气球，以及 x = 11 射爆另外两个气球

示例 2：

> **输入**：points = [[1,2],[3,4],[5,6],[7,8]]
> **输出**：4

示例 3：

> **输入**：points = [[1,2],[2,3],[3,4],[4,5]]
> **输出**：2

示例 4：

> **输入**：points = [[1,2]]
> **输出**：1

示例 5：

> **输入**：points = [[2,3],[2,3]]
> **输出**：1

提示：

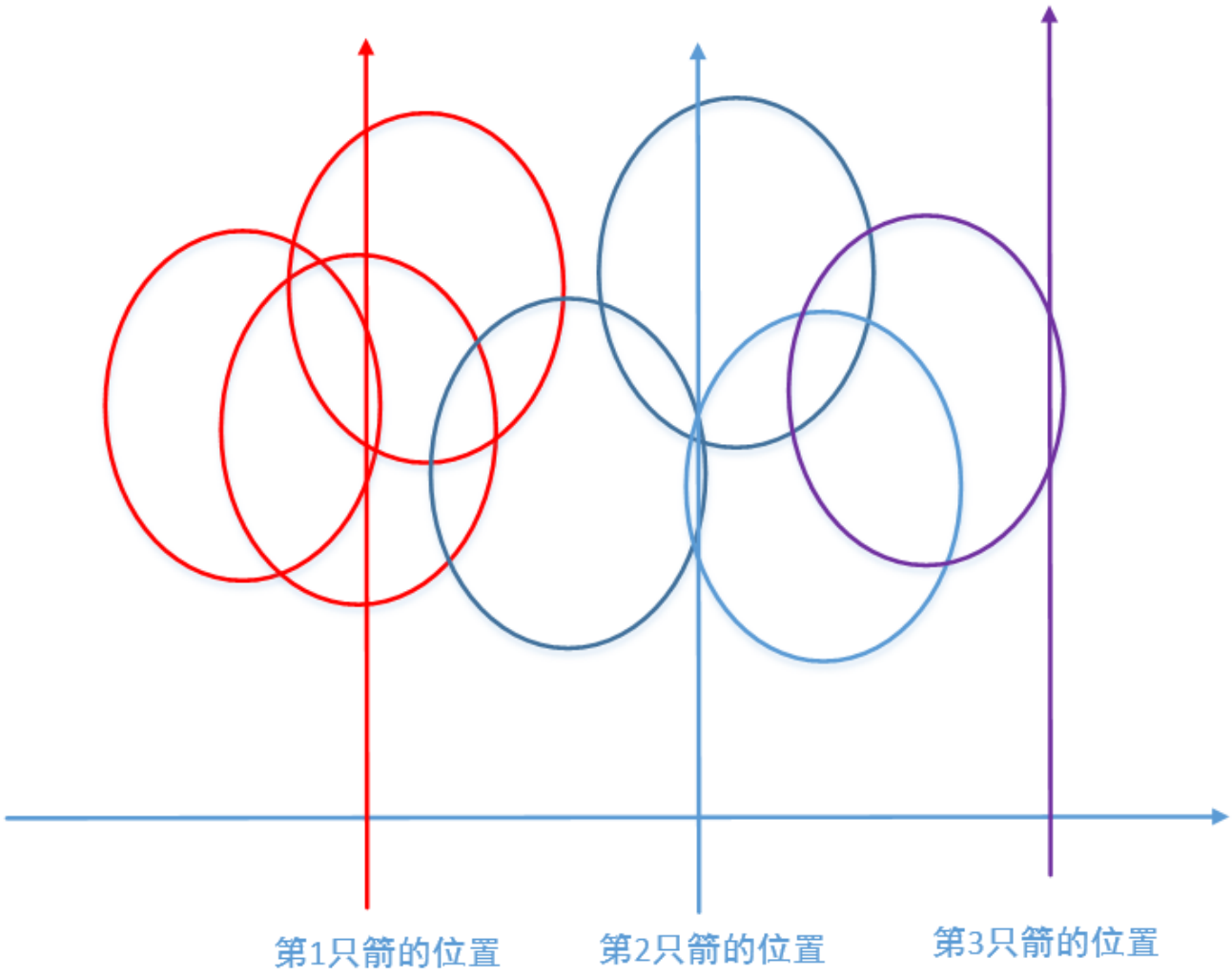
- 0 <= points.length <= 10^4
- points[i].length == 2
- -2^31 <= xstart < xend <= 2^31 - 1

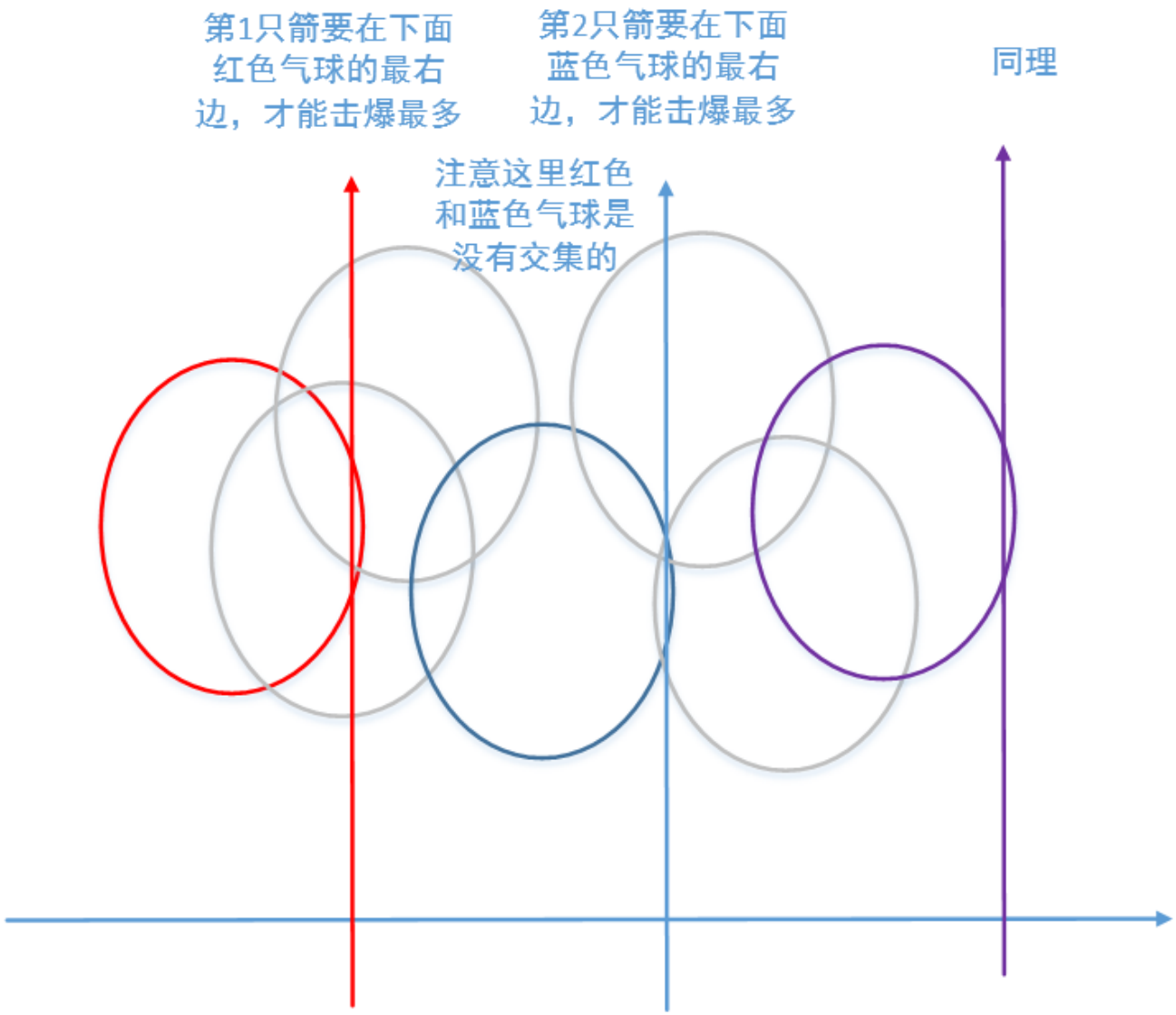
气球按照右边界排序

这题是让求用最少的箭把所有的气球全部击爆。每个气球都是有宽度的，分别是左边界和右边界，这里只需要把所有的气球按照右边界的大小进行排序。然后把第一支箭尽可能的往第一个气球的右边靠，这样第一支箭引爆气球的数量就是最多的。同理，第二支

箭要尽可能的往第二个气球（这里不是排序后的第二个气球，这里的第二个气球是和第一个气球坐标没有交集的那个气球）的右边靠……。

下面来画个图看下，假如下面是已经按照右边界排序好的气球，





再来看下代码

```java
public int findMinArrowShots(int[][] points) {
    //边界条件判断
    if (points == null || points.length == 0)
        return 0;
    //按照每个气球的右边界排序
    Arrays.sort(points, (a, b) -> a[1] > b[1] ? 1 : -1);
    //获取排序后第一个气球右边界的位置，我们可以认为是箭射入的位置
    int last = points[0][1];
    //统计箭的数量
    int count = 1;
    for (int i = 1; i < points.length; i++) {
        //如果箭射入的位置小于下标为i这个气球的左边位置，说明这支箭不能
        //击爆下标为i的这个气球，需要再拿出一支箭，并且要更新这支箭射入的
        //位置
        if (last < points[i][0]) {
            last = points[i][1];
            count++;
        }
    }
    return count;
}
```

气球按照左边界排序

上面气球是按照右边界排序，其实还可以按照左边界排序，原理都是一样的，看下代码

```java
public int findMinArrowShots(int[][] points) {
    //边界条件判断
    if (points == null || points.length == 0)
        return 0;
    //按照每个气球的左边界排序
    Arrays.sort(points, (a, b) -> a[0] > b[0] ? 1 : -1);
    //获取排序后最后一个气球左边界的位置，我们可以认为是箭射入的位置
    int last = points[points.length - 1][0];
    //统计箭的数量
    int count = 1;
    for (int i = points.length - 1; i >= 0; i--) {
        //如果箭射入的位置大于下标为i这个气球的右边位置，说明这支箭不能
        //击爆下标为i的这个气球，需要再拿出一支箭，并且要更新这支箭射入的
        //位置
        if (last > points[i][1]) {
            last = points[i][0];
            count++;
        }
    }
    return count;
}
```

总结

这题和472，插入区间有点类似，但又不同。这题首先要对数组进行排序，无论是按照气球的左边还是右边排序，实现原理都是一样的。

往期推荐

- 472，插入区间
- 463. 判断回文链表的3种方式
- 457，二叉搜索树的最近公共祖先
- 455，DFS和BFS解被围绕的区域

480，移动零，通过一个精彩的故事告诉你怎么解

原创 山大王wld 数据结构和算法 1周前

收录于话题
#算法图文分析
95个 >



You never really knew a man until you stood in his shoes and walked around in them.

只有设身处地为他人着想，你才能真正了解对方。

问题描述

给定一个数组 `nums` ，编写一个函数将所有 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。

示例:

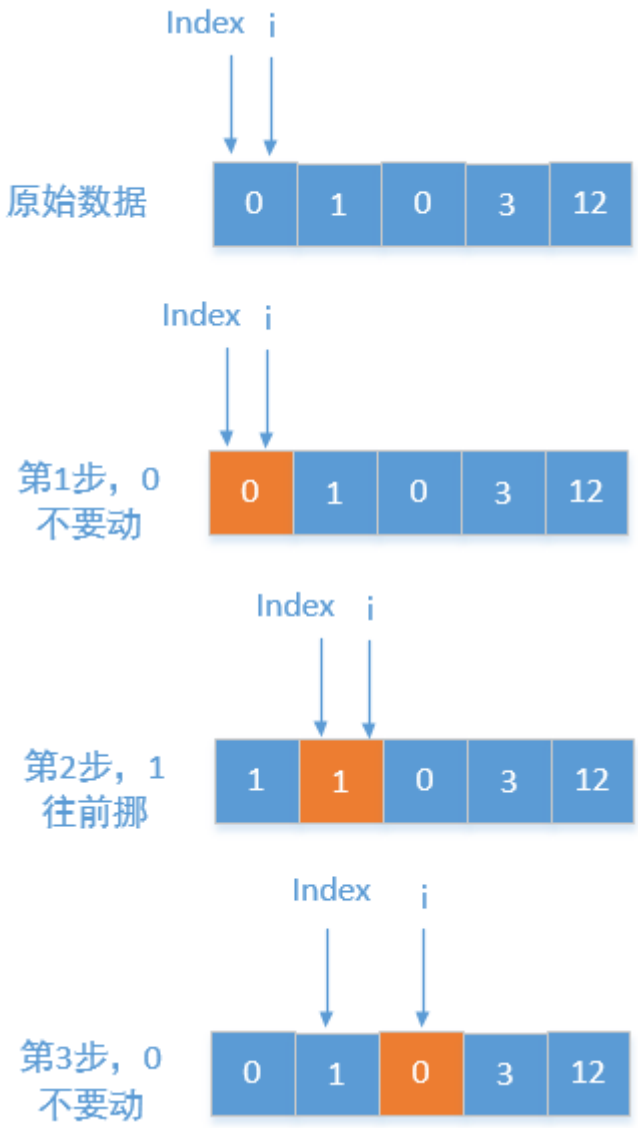
> **输入**: [0,1,0,3,12]
> **输出**: [1,3,12,0,0]

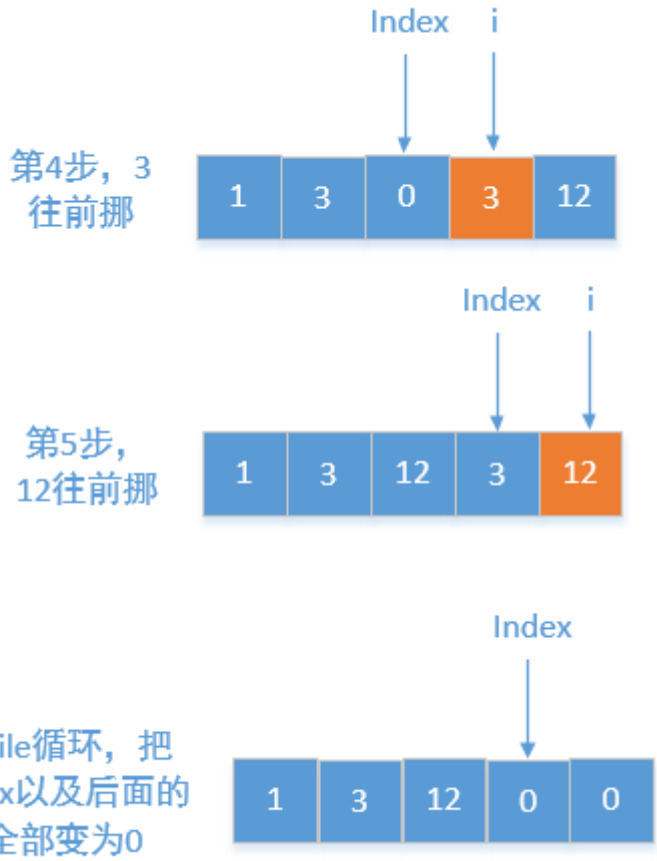
说明:

1. 必须在原数组上操作，不能拷贝额外的数组。
2. 尽量减少操作次数。

把非0的往前挪

这题很容易理解，就是把0移动到数组的末尾，然后前面就都是非0的元素了，并且移完之后还要保证之前非0的顺序不要变。所以一种最简单的方式就是把非0的往前挪，挪完之后，后面的就都是0了，然后在用0覆盖后面的元素。这种是最容易理解也是最容易想到的，代码比较简单，这里就以示例为例画个图来看下





再来看下代码

```
public void moveZeroes(int[] nums) {
  if (nums == null || nums.length == 0)
    return;
  int index = 0;
  //一次遍历，把非零的都往前挪
  for (int i = 0; i < nums.length; i++) {
    if (nums[i] != 0)
      nums[index++] = nums[i];
  }
  //后面的都是0，
  while (index < nums.length) {
    nums[index++] = 0;
  }
}
```

参照双指针解决

这里可以参照双指针的思路解决，指针j是一直往后移动的，如果指向的值不等于0才对他进行操作。而i统计的是前面0的个数，我们可以把j-i看做另一个指针，它是指向前面第一个0的位置，然后我们让j指向的值和j-i指向的值交换，代码比较简单，来直接看下

```
public void moveZeroes(int[] nums) {
  int i = 0;//统计前面0的个数
  for (int j = 0; j < nums.length; j++) {
    if (nums[j] == 0) {//如果当前数字是0就不操作
      i++;
    } else if (i != 0) {
      //否则，把当前数字放到最前面那个0的位置，然后再把
      //当前位置设为0
      nums[j - i] = nums[j];
      nums[j] = 0;
    }
```

```
    }
  }
```

如果觉得绕，还可以在换种写法

```
public void moveZeroes(int[] nums) {
  int i = 0;
  for (int j = 0; j < nums.length; j++) {
    //只要不为0就往前挪
    if (nums[j] != 0) {
      //i指向的值和j指向的值交换
      int temp = nums[i];
      nums[i] = nums[j];
      nums[j] = temp;
      i++;
    }
  }
}
```

上面的解法都不难，如果还理解不了，我给你讲个故事吧。有两个人A和B，其中A有特异功能，水路，陆路他都能走，而B只能走陆路，不能走水路。题中数组为0的我们把它看做是水路，不为0的我们可以把它看做是陆路，A和B同时出发，走的时候，无论是水路还是陆路，A都会往前走一步。而B只能遇到陆路才能走，遇到水路就掉到水里，走不动了，这个时候A可以继续走水路，当A往前走找到陆路的时候就把陆路踢到B的面前，然后B就可以继续走了。

这里就以示例 [0,1,0,3,12]为例，来看下A和B的对话。

初始状态：咱哥俩一起走

第1步

- B：兄弟救我，我掉进水里了。
- A：兄弟别急，我也在水里，我到前面找块陆地给你

第1步之后数组的值是[0,1,0,3,12]

第2步

- A：兄弟，我找到陆地1了，踢给你，你接着（把1踢给B）
- B：好嘞

第2步之后数组的值是[1,0,0,3,12]

第3步

- B：兄弟，我又掉进水里了
- A：别急，我也在水了，我在到前面看看有没有陆地，找块给你

第3步之后数组的值是[1,0,0,3,12]

第4步

- A：兄弟，我找到陆地3了，踢给你，你接着（把3踢给B）
- B：好嘞

第4步之后数组的值是[1,3,0,0,12]

第5步

- B：兄弟，我又掉水里了，快来救救我
- A：我现在在陆地12上面，我把它踢给你吧
- B：好嘞，谢了，兄弟

第5步之后数组的值是[1,3,12,0,0]

往期推荐

- 477，动态规划解按摩师的最长预约时间
- 474，翻转二叉树的多种解决方式
- 470，DFS和BFS解合并二叉树
- 已经写了400多道算法了，够你看一段时间了

479，递归方式解打家劫舍

原创 山大王wld 数据结构和算法 1周前

收录于话题
#算法图文分析

95个 >



No man understands a deep book until he has seen and lived at least part of its contents.

人们只有在切身体会过某些情节后，才能理解那些深奥的书。

问题描述

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，**如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。**

给定一个代表每个房屋存放金额的非负整数数组，计算你**不触动警报装置的情况下**，一夜之内能够偷窃到的最高金额。

示例 1：

> **输入**：[1,2,3,1]
> **输出**：4
> **解释**：偷窃 1 号房屋（金额 =1），然后偷窃 3 号房屋（金额 =3）。偷窃到的最高金额 =1+3=4。

示例 2：

> **输入**：[2,7,9,3,1]
> **输出**：12
> **解释**：偷窃1号房屋(金额=2),偷窃3号房屋(金额=9)，接着偷窃5号房屋(金额=1)。偷窃到的最高金额=2+9+1=12。

提示：

- 0 <= nums.length <= 100
- 0 <= nums[i] <= 400

递归方式解决

这题让计算小偷偷窃的最大金额，并且不能偷窃相邻的两家，这题一看就知道，不就是前几天讲的477，动态规划解按摩师的最长预约时间吗，只不过是换汤不换药，基本原理还是一样的。这题最简单的一种解决方式就是使用动态规划，可以参照第477题，就不在过多介绍。这里来看另一种解决方式，**递归**。

之前讲426，什么是递归，通过这篇文章，让你彻底搞懂递归的时候提到递归的两个重要要素，**一个是调用自己，一个是必须要有终止条件**，我们先来定义一个函数

```
1   private int robHelper(int[] nums, int i) {
2
3   }
```

他表示的是前i+1（i是从0开始的，0表示的是第1个房屋）个房屋所能偷窃到的最大值，那么很明显

```
1   private int robHelper(nums, i-1)
```

表示的是前i个房屋所能偷窃到的最大值

```
1   private int robHelper(nums, i-2)
```

表示的就是前i-1个房屋所能偷窃到的最大值

因为第i-1个房屋和第i+1个房屋是不挨着的，所以如果偷完前i-1个房屋之后是可以再偷第i+1个房屋的。所以我们可以找到一种关系就是

```
1   private int robHelper(int[] nums, int i) {
```

```
    //偷上上家之前所能得到的最大值
    int lastLast = robHelper(nums, i - 2);
    //偷上家之前所能得到的最大值
    int last = robHelper(nums, i - 1);
    //偷上上家之前的还可以再偷当前这一家
    int cur = lastLast + nums[i];
    //然后返回偷当前这一家和不偷当前这一家的最大值
    return Math.max(last, cur);
  }
```

问题结束了吗，当然没有，因为我们知道递归必须要有终止条件，那么终止条件是什么呢，就是i小于0，也就是说没有房屋可偷，最终代码如下

```
public int rob(int[] nums) {
  return robHelper(nums, nums.length - 1);
}

private int robHelper(int[] nums, int i) {
  //终止条件
  if (i < 0)
    return 0;
  //偷上上家之前所能得到的最大值
  int lastLast = robHelper(nums, i - 2);
  //偷上家之前所能得到的最大值
  int last = robHelper(nums, i - 1);
  //偷上上家之前的还可以再偷当前这一家
  int cur = lastLast + nums[i];
  //然后返回偷当前这一家和不偷当前这一家的最大值
  return Math.max(last, cur);
}
```

代码优化

之前讲418，剑指 Offer-斐波那契数列和356，青蛙跳台阶相关问题的时候都提到过斐波那契数列的递归计算方式，递归的时候效率是很差的，因为代码中包含大量的重复计算。这题也一样，如果非要使用递归的方式解决，可以把计算的值先存起来，下次用的时候如果有就直接去取，如果没有，再计算。

```
public int rob(int[] nums) {
  return robHelper(nums, nums.length - 1, new HashMap<>());
}

private int robHelper(int[] nums, int i, Map<Integer, Integer> map) {
  //终止条件
  if (i < 0)
    return 0;

  int lastLast = 0;
  int last = 0;

  //查看map中是否存在，如果存在就从map中取，不用再计算了
  if (map.containsKey(i - 2))
    lastLast = map.get(i - 2);
  else {
    //偷上上家之前所能得到的最大值
    lastLast = robHelper(nums, i - 2, map);
    //如果map中不存在就计算，计算完之后要存储在map中，下次用的
    //时候直接从map中取，不用再计算了。
    map.put(i - 2, lastLast);
  }

  //原理同时
  if (map.containsKey(i - 1))
    last = map.get(i - 1);
```

```
27      else {
28        //偷上家之前所能得到的最大值
29        last = robHelper(nums, i - 1, map);
30        map.put(i - 1, last);
31      }
32
33      //偷上上家之前的还可以再偷当前这一家
34      int cur = lastLast + nums[i];
35      //然后返回偷当前这一家和不偷当前这一家的最大值
36      return Math.max(last, cur);
37    }
```

总结

这题解法比较多，使用动态规划和递归都是可以解决的，如果使用递归要注意代码优化，否则当数据稍微多点，执行时间就会很长。

往期推荐

- 467. 递归和非递归解路径总和问题
- 465. 递归和动态规划解三角形最小路径和
- 426，什么是递归，通过这篇文章，让你彻底搞懂递归
- 423，动态规划和递归解最小路径和

475，有效的山脉数组

原创 山大王wld 数据结构和算法 11月12日

收录于话题
#算法图文分析

95个 >



The strongest person is the person who isn't scared to be alone.

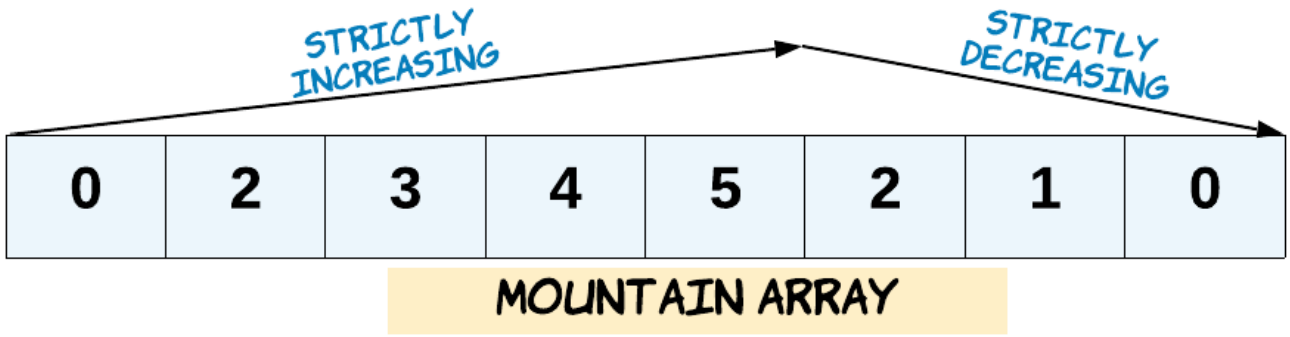
强大的人不会惧怕孤独。

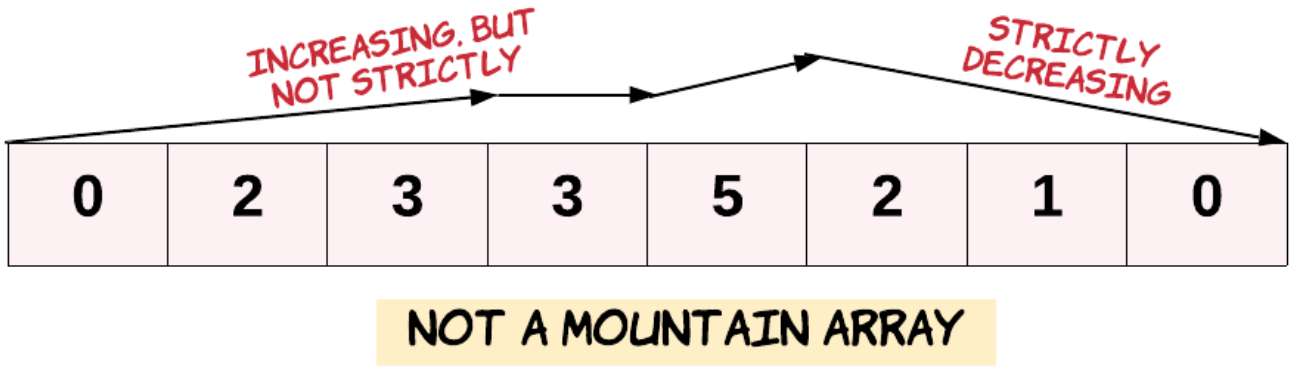
问题描述

给定一个整数数组 `A`，如果它是有效的山脉数组就返回 `true`，否则返回 `false`。

让我们回顾一下，如果A满足下述条件，那么它是一个山脉数组：

- **A.length >= 3**
- **在0 < i < A.length-1条件下，存在i使得：**
  - **A[0] < A[1] < ... A[i-1] < A[i]**
  - **A[i] > A[i+1] > ... > A[A.length - 1]**





示例 1：

> **输入：** [2,1]
> **输出：** false

示例 2：

> **输入：** [3,5,5]
> **输出：** false

示例 3：

> **输入：** [0,3,2,1]
> **输出：** true

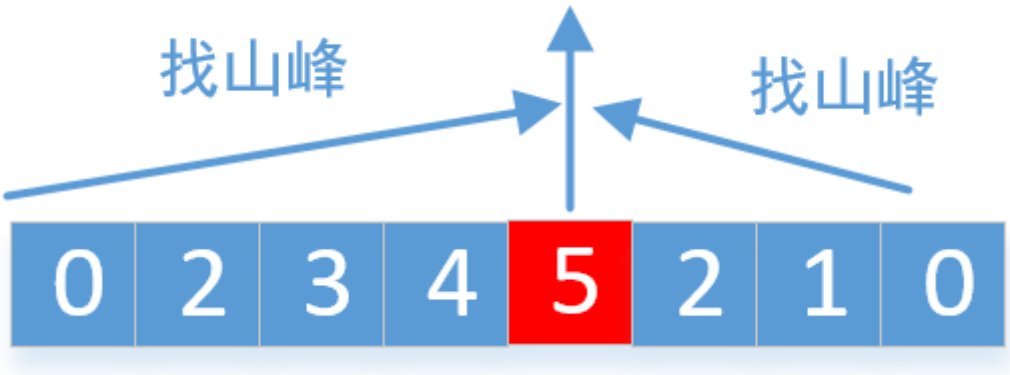
提示：

1. 0 <= A.length <= 10000
2. 0 <= A[i] <= 10000

从两边找

这题让判断一个数组是否是有效山峰数组，所谓有效山峰数组就是在数组中有且仅有一个最大值，并且最大值往前走是降序的，往后走也是降序的。

一种简单的解决方式就是使用两个变量left和right，我们也可以把它看做是两个指针，left从数组的前面开始，如果是升序的就一直找，直到遇到降序的时候停止，right从数组后面往前找，如果是升序的就一直找，直到遇到降序的时候停止。然后再判断left和right是否相等。如下图所示



来看下代码

```
1   public boolean validMountainArray(int[] A) {
2     int len = A.length;
3     int left = 0;
4     int right = len - 1;
5     //从左边往右边找，一直找到山峰为止
6     while (left + 1 < len && A[left] < A[left + 1])
7       left++;
8     //从右边往左边找，一直找到山峰为止
9     while (right > 0 && A[right - 1] > A[right])
10      right--;
11    //判断从左边和从右边找的山峰是不是同一个
12    return left > 0 && right < len - 1 && left == right;
13  }
```

从一边找

从一边找的思路就是，先从左边开始找到山峰，然后再从山峰开始往右边下山，如果能走到数组的最后一个元素，说明是有效山峰。

```
1   public boolean validMountainArray(int[] A) {
2     int len = A.length;
3     int left = 0;
4     //从左边往右边找，一直找到山峰为止
5     while (left + 1 < len && A[left] < A[left + 1])
6       left++;
7     //如果数组一开始就是降序,比如[4,3,2,1],或者一直是升序的,
8     //比如[1,2,3,4]都不能称为有效山峰
9     if (left == 0 || left == len - 1)
10      return false;
11    //找到山峰之后然后下山
12    while (left + 1 < len && A[left] > A[left + 1])
13      left++;
14    //如果能走到山脚下，也就是走到数组的最后一个元素，就表示
15    //是有效山峰
16    return left == A.length - 1;
17  }
```

总结

这题只要找到数组中的最大值，然后判断最大值前面部分是升序的（从数组的第一个元素开始到最大值），后面部分是降序的即是有效的山脉数组。

往期推荐

- 407，动态规划和滑动窗口解决最长重复子数组
- 406，剑指 Offer-二维数组中的查找
- 404，剑指 Offer-数组中重复的数字
- 394，经典的八皇后问题和N皇后问题

472，插入区间

原创 山大王wld 数据结构和算法 11月5日

收录于话题
#算法图文分析

95个 >



If we cease to believe in love, why would we want to live?

如果我们不相信爱，那又为何而活呢？

问题描述

给出一个无重叠的 ， 按照区间起始端点排序的区间列表。

在列表中插入一个新的区间，你需要确保列表中的区间仍然有序且不重叠（如果有必要的话，可以合并区间）。

示例 1：

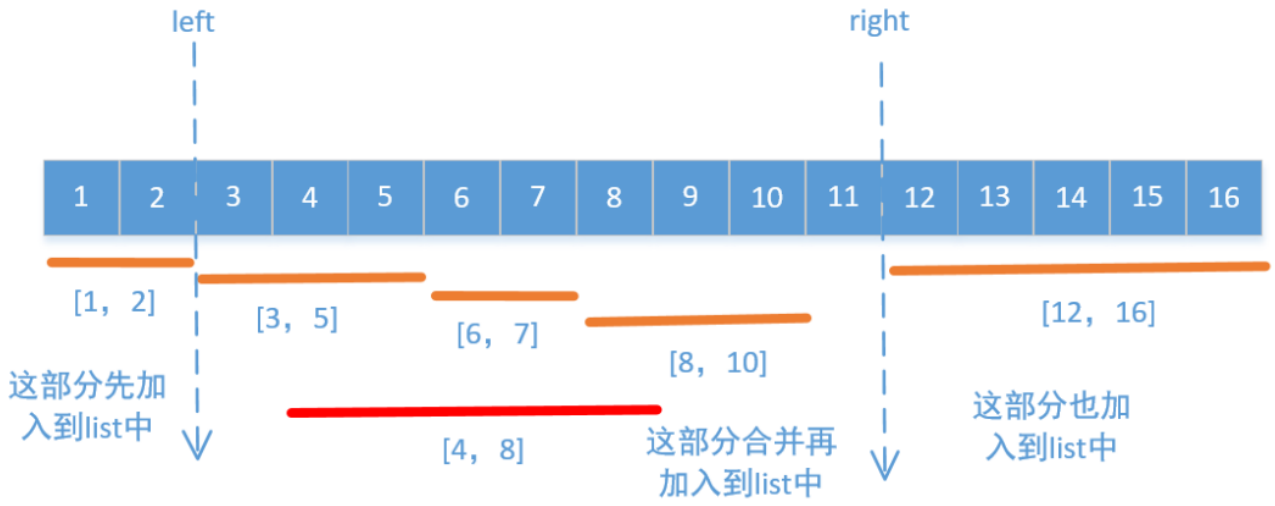
> **输入**：
> intervals = [[1,3],[6,9]]
> newInterval = [2,5]
>
> **输出**：[[1,5],[6,9]]

示例 2：

> **输入**：
> intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]]
> newInterval = [4,8]
>
> **输出**：[[1,2],[3,10],[12,16]]
> **解释**：这是因为新的区间 [4,8] 与 [3,5],[6,7],[8,10] 重叠。

先计算两边再计算中间

这里我们人为把数组分为3部分，左边不重合的（如果有）添加到集合list中，右边不重合的（如果有）也添加到集合list中，然后再合并中间的，这里以示例2为例画个图看一下



原理很简单，来看下代码

```
public int[][] insert(int[][] intervals, int[] newInterval) {
    //边界条件判断
    if (intervals.length == 0)
        return new int[][]{newInterval};
    List<int[]> resList = new ArrayList<>();

    //一个从左边开始找不重合的
    int left = 0;
    //一个从右边开始找不重合的
    int right = intervals.length - 1;

    //左边不重合的添加到list中
    while (left < intervals.length && intervals[left][1] < newInterval[0]) {
        resList.add(intervals[left++]);
    }

    //右边不重合的添加到list中
    while (right >= 0 && intervals[right][0] > newInterval[1]) {
        resList.add(left, intervals[right--]);
    }

    //下面一大坨是合并中间重合的，注意一些边界条件的判断
    int[] newArr = new int[2];
    newArr[0] = left >= intervals.length ? newInterval[0] : Math.min(intervals[left][0], newInterva
    newArr[1] = right < 0 ? newInterval[1] : Math.max(intervals[right][1], newInterval[1]);
    resList.add(left, newArr);
```

```
27
28      //这一大坨是把list转二维数组
29      int[][] resArr = new int[resList.size()][2];
30      for (int i = 0; i < resList.size(); i++) {
31          resArr[i] = resList.get(i);
32      }
33      return resArr;
34  }
35
```

逐步合并

上面一种方式是先把两边不重合的添加到集合list中，之后在合并中间的。这里还可以从左边开始把不重合的（如果有不重合的）添加到集合list中，如果遇到重合的就找出重合的范围然后再添加到集合中，最后再把后面不重合的（如果有）添加到集合list中。

```
1   public int[][] insert(int[][] intervals, int[] newInterval) {
2       List<int[]> resList = new ArrayList<>();
3       int i = 0;
4       //先把前面不重合的添加到list中
5       while (i < intervals.length && intervals[i][1] < newInterval[0])
6           resList.add(intervals[i++]);
7
8       int mergeStar = newInterval[0];
9       int mergeEnd = newInterval[1];
10      //前面不重合的都添加到集合list中了，从这里开始就出现重合了，我们要找到重合的开始和结束值
11      while (i < intervals.length && intervals[i][0] <= newInterval[1]) {
12          mergeStar = Math.min(mergeStar, intervals[i][0]);
13          mergeEnd = Math.max(mergeEnd, intervals[i][1]);
14          i++;
15      }
16      //然后再把重合的添加到list中
17      resList.add(new int[]{mergeStar, mergeEnd});
18
19      //把剩下的在添加到集合list中
20      while (i < intervals.length)
21          resList.add(intervals[i++]);
22
23      //这一大坨是把list转二维数组
24      int[][] resArr = new int[resList.size()][2];
25      for (int j = 0; j < resList.size(); j++) {
26          resArr[j] = resList.get(j);
27      }
28      return resArr;
29  }
```

总结

这题难度不是很大，但一次写出来不出错比较困难，因为他有很多边界条件的判断。

468，提莫攻击的两种解决方式

原创 山大王wld 数据结构和算法 10月26日

收录于话题
#算法图文分析

95个 >



If you only do what you can do, you'll never be more than you are now.

如果你只做力所能及的事，你就没法进步。

问题描述

在《英雄联盟》的世界中，有一个叫“提莫”的英雄，他的攻击可以让敌方英雄艾希（编者注：寒冰射手）进入中毒状态。现在，给出提莫对艾希的攻击时间序列和提莫攻击的中毒持续时间，你需要输出艾希的中毒状态总时长。

你可以认为提莫在给定的时间点进行攻击，并立即使艾希处于中毒状态。

输入：[1,4]，2
输出：4

原因：第1秒初，提莫开始对艾希进行攻击并使其立即中毒。中毒状态会维持2秒钟，直到第2秒末结束。
第4秒初，提莫再次攻击艾希，使得艾希获得另外2秒中毒时间。

> 所以最终输出4秒。

示例2：

> **输入**：[1,2]，2
> **输出**：3
>
> **原因**：第1秒初，提莫开始对艾希进行攻击并使其立即中毒。中毒状态会维持2秒钟，直到第2秒末结束。
> 但是第2秒初，提莫再次攻击了已经处于中毒状态的艾希。
> 由于中毒状态不可叠加，提莫在第2秒初的这次攻击会在第3秒末结束。
>
> 所以最终输出 3 。

提示：

1. 你可以假定时间序列数组的总长度不超过 10000。
2. 你可以假定提莫攻击时间序列中的数字和提莫攻击的中毒持续时间都是非负整数，并且不超过 10,000,000。

问题分析

假设中毒的持续时间是t，只要数组中每个元素的间隔都大于t，那么总时间就是**数组的长度***t。因为下一个攻击的时间还没到，中毒的持续时间就已经完成了，到下一个攻击的时间还可以继续攻击。比如中毒持续时间是2，攻击时间序列是[1，3，6，8]，那么中毒的总时间就是2*4=8。

如果数组的间隔小于中毒的持续时间，下次攻击的时候时间上就会出现重叠，我们要做的就是减去这个重叠的时间。这道题比较简单，我们来看下代码。

```
1   public int findPoisonedDuration(int[] timeSeries, int duration) {
2     //边界条件判断
3     if (timeSeries.length == 0 || duration == 0)
4       return 0;
5     //res表示总的中毒持续时间
6     int res = duration;
7     for (int i = 1; i < timeSeries.length; i++) {
8       //两次攻击的时间差和中毒持续的时间比较，选择小的
9       res += Math.min(timeSeries[i] - timeSeries[i - 1], duration);
10    }
11    return res;
12  }
```

其实还有一种思路就是先计算所有的中毒持续时间，然后再减去重叠的时间段，可以看下

```java
public int findPoisonedDuration(int[] timeSeries, int duration) {
    //计算总的中毒时间，不考虑重叠的
    int res = duration * timeSeries.length;
    for (int i = 1; i < timeSeries.length; i++) {
        //然后再减去重叠的时间段
        res -= Math.max(0, duration - (timeSeries[i] - timeSeries[i - 1]));
    }
    return res;
}
```

总结

这题算是比较简单的，计算的时候主要判断有没有重叠的时间段，如果有重叠，减去就行了。

往期推荐

- 467. 递归和非递归解路径总和问题
- 465. 递归和动态规划解三角形最小路径和
- 464. BFS和DFS解二叉树的所有路径
- 457，二叉搜索树的最近公共祖先

467. 递归和非递归解路径总和问题

原创 山大王wld 数据结构和算法 10月23日

收录于话题
#算法图文分析

95个 >



A weak man has doubts before a decision. A strong man has them afterwards.

弱者在决策前迟疑，强者则反之。

问题描述

给定一个二叉树和一个目标和，判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和。

说明: 叶子节点是指没有子节点的节点。

示例:
给定如下二叉树，以及目标和 sum = 22，

```
              5
             / \
            4   8
           /   / \
          11  13  4
```

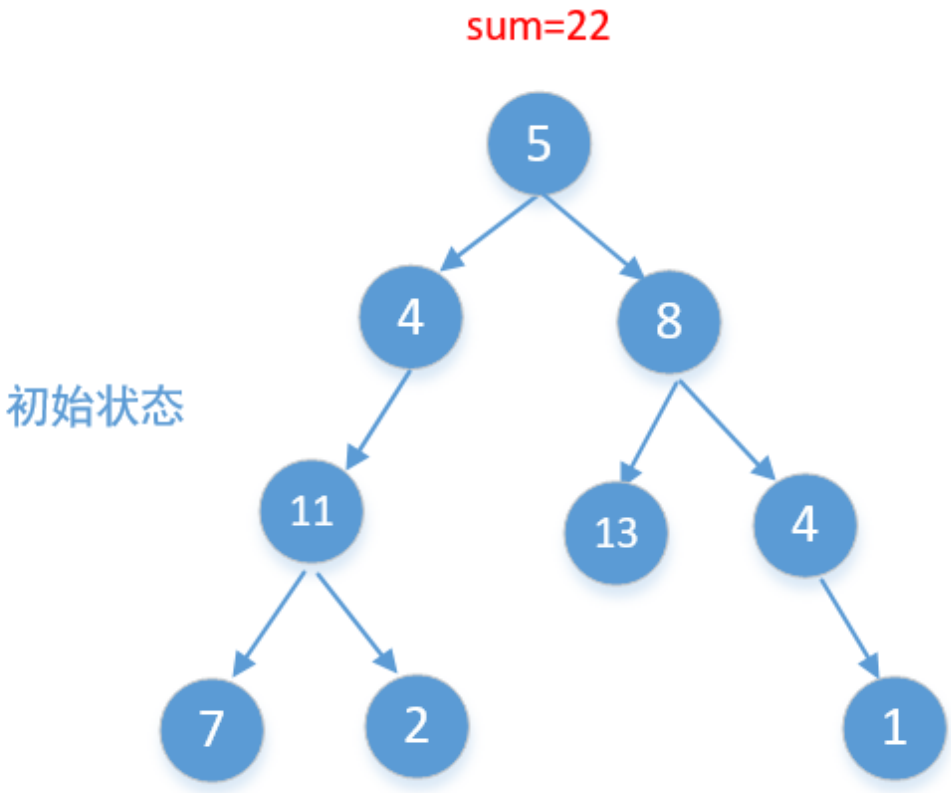
```
        /  \      \
       7    2      1
```

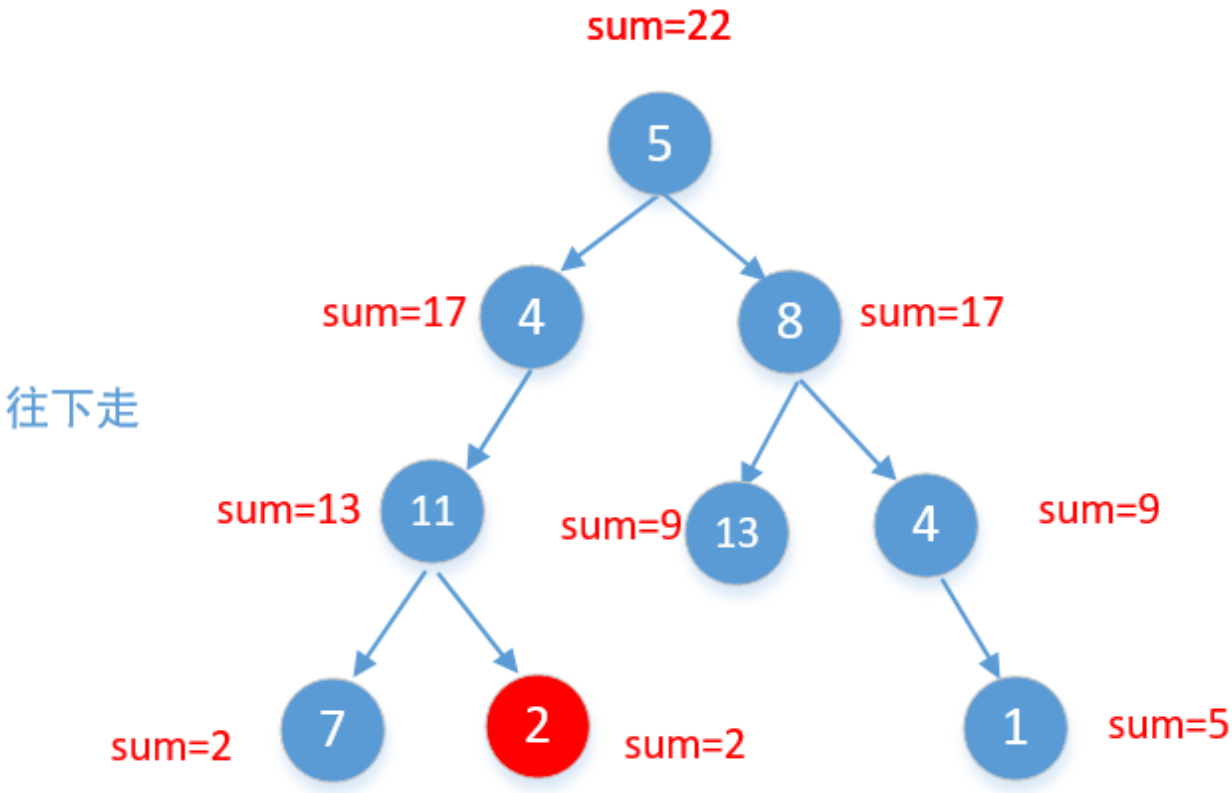
返回 `true`，因为存在目标和为 22 的根节点到叶子节点的路径 `5->4->11->2`。

递归求解

这题让判断从根节点到叶子节点的所有路径中，有没有和等于sum的，如果看过之前讲的442，剑指 Offer-回溯算法解二叉树中和为某一值的路径，再来看这一题就觉得这题有点简单了。第442题要求的是把所有的和等于sum的路径都打印出来，而这题只要判断有一个路径的和等于sum即可。

我们可以从根节点往下走，走的时候减去当前节点的值，一直到叶子节点，如果减到最后，值等于叶子节点的值，说明存在这样的结果，直接返回true，否则如果把所有节点都遍历完了也不存在这样的结果，就返回false。我们就以示例为例画个图来看一下





再来看下代码

```
1   public boolean hasPathSum(TreeNode root, int sum) {
2     //如果根节点为空，或者叶子节点也遍历完了也没找到这样的结果，就返回false
3     if (root == null)
4       return false;
5     //如果到叶子节点了，并且剩余值等于叶子节点的值，说明找到了这样的结果，直接返回true
6     if (root.left == null && root.right == null && sum - root.val == 0)
7       return true;
8     //分别沿着左右子节点走下去，然后顺便把当前节点的值减掉，左右子节点只要有一个返回true，
9     //说明存在这样的结果
```

```
    return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum - root.val);
}
```

非递归解决

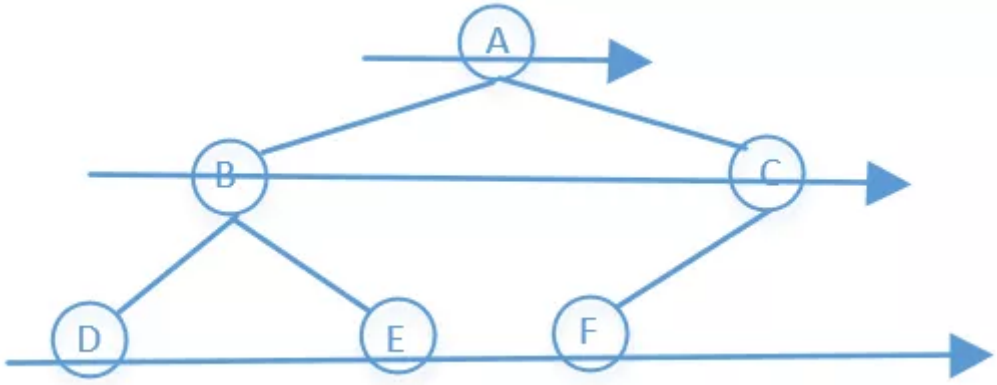
上面使用的是递归的方式，我们还可以使用非递归的方式，在遍历的时候有两种方式，一种是从0开始累加，到叶子节点的时候如果累加的值等于sum，说明存在这样的一条路径。还一种是减，从根节点一直减下去，如果到叶子节点的时候，值等于叶子节点的值，说明也存在这样的一条路径。原理都一样，这里就以加的方式来看下代码该怎么写

```
public boolean hasPathSum(TreeNode root, int sum) {
  if (root == null)
    return false;
  Stack<TreeNode> stack = new Stack<>();
  stack.push(root);//根节点入栈
  while (!stack.isEmpty()) {
    TreeNode cur = stack.pop();//出栈
    //累加到叶子节点之后，结果等于sum，说明存在这样的一条路径
    if (cur.left == null && cur.right == null) {
      if (cur.val == sum)
        return true;
    }
    //右子节点累加，然后入栈
    if (cur.right != null) {
      cur.right.val = cur.val + cur.right.val;
      stack.push(cur.right);
    }
    //左子节点累加，然后入栈
    if (cur.left != null) {
      cur.left.val = cur.val + cur.left.val;
      stack.push(cur.left);
    }
  }
  return false;
}
```

如果大家看过464. BFS和DFS解二叉树的所有路径，还可以不直接操作节点的值，可以再使用一个额外的栈，专门存放累加或者往下减的值，这个值是和节点一一对应的，他们会同时出栈，以及同时入栈，实现过程比较简单，这里就不在介绍。

BFS解决

之前在讲373，数据结构-6,树的时候，讲到树的BFS，就是一层一层的往下打印，像下面这样



他的代码如下

```
1   public void levelOrder(TreeNode tree) {
2     if (tree == null)
3       return;
4     Queue<TreeNode> queue = new LinkedList<>();
5     queue.add(tree);//相当于把数据加入到队列尾部
6     while (!queue.isEmpty()) {
7       //poll方法相当于移除队列头部的元素
8       TreeNode node = queue.poll();
9       System.out.println(node.val);
10      if (node.left != null)
11        queue.add(node.left);
12      if (node.right != null)
13        queue.add(node.right);
14    }
15  }
```

在一层一层打印的时候，我们可以把值累加或累减都可以，这里使用累减的方式来看下代码

```
1   public boolean hasPathSum(TreeNode root, int sum) {
2     if (root == null)
3       return false;
4     Queue<TreeNode> queue = new LinkedList<>();
5     root.val = sum - root.val;
6     queue.add(root);
7     while (!queue.isEmpty()) {
8       TreeNode node = queue.poll();
9       //累减到根节点之后，结果为0，说明存在这样一条路径，直接返回true
10      if (node.left == null && node.right == null && node.val == 0)
11        return true;
12      //左子节点累减
13      if (node.left != null) {
14        node.left.val = node.val - node.left.val;
15        queue.add(node.left);
16      }
17      //右子节点累减
18      if (node.right != null) {
19        node.right.val = node.val - node.right.val;
20        queue.add(node.right);
21      }
22    }
23    return false;
24  }
```

总结

如果对二叉树的各种遍历比较熟悉的话，这题还算是比较简单的，这题比较灵活，解法比较多，如果想写还可以继续写下去。

454，字母异位词分组

原创 山大王wld 数据结构和算法 9月22日

收录于话题
#算法图文分析

95个 >



Boredom is for lazy people who have no imagination.

好吃懒做又脑袋空空的人才会觉得生活无聊。

问题描述

给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。

示例:

> **输入**:
> ["eat", "tea", "tan", "ate", "nat", "bat"]
>
> **输出**:
> [
> ["ate","eat","tea"],
> ["nat","tan"],
> ["bat"]
>]

说明:

- 所有输入均为小写字母。
- 不考虑答案输出的顺序。

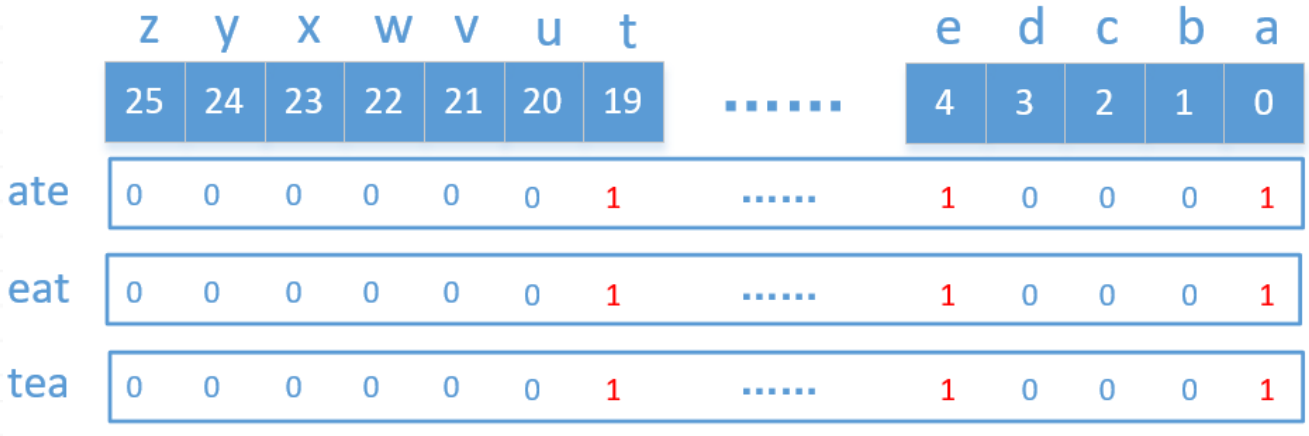
先排序再判断

今天这道题比较简单，字母异位词就是两个字符串中的字母都是一样的，只不过顺序被打乱了，这里要把他们找出来，然后放到一起。既然字母异位词的字母都是一样的，可以对字符串中的字符进行排序，生成一个新的字符串，如果生成新的字符串相同，那么他们就是字母异位词。代码比较简单，来看下

```
public List<List<String>> groupAnagrams(String[] strs) {
    //边界条件判断
    if (strs == null || strs.length == 0)
        return new ArrayList<>();
    //map中key存储的是字符串中字母排序后新的字符串
    Map<String, List<String>> map = new HashMap<>();
    for (int i = 0; i < strs.length; i++) {
        //取出字符串，然后把它转化为字符数组
        char[] c = strs[i].toCharArray();
        //对字符数组进行排序
        Arrays.sort(c);
        //排序之后再把它转化为一个字符串
        String keyStr = String.valueOf(c);
        //判断map中有没有这个字符串，如果没有这个字符串，
        //说明还没有出现和这个字符串一样的字母异位词，
        //要新建一个list，把它存放到map中
        if (!map.containsKey(keyStr))
            map.put(keyStr, new ArrayList<>());
        //把字符串存放到对应的list中
        map.get(keyStr).add(strs[i]);
    }
    //最后返回
    return new ArrayList<>(map.values());
}
```

统计每个字母的个数

题目中说了所有输入均为小写字母，所以还可以只用一个数组，统计字符串中每个字符的个数，最终会生成一个新的字符串，如果生成新的字符串相同，说明他们是字母异位词，画个图看一下



可以看到，只要是字母异位词，通过上面的方式转换，他们生成的字符串都是一样的

```java
public List<List<String>> groupAnagrams(String[] strs) {
    //边界条件判断
    if (strs == null || strs.length == 0)
        return new ArrayList<>();
    Map<String, List<String>> map = new HashMap<>();
    for (String s : strs) {
        char[] ca = new char[26];
        //统计字符串中每个字符串出现的次数
        for (char c : s.toCharArray())
            ca[c - 'a']++;
        //统计每个字符出现次数的数组转化为字符串
        String keyStr = String.valueOf(ca);
        if (!map.containsKey(keyStr))
            map.put(keyStr, new ArrayList<>());
        map.get(keyStr).add(s);
    }
    return new ArrayList<>(map.values());
}
```

总结

由相同的字母组成的字符串才是字母异位词，既然有相同的字母，直接把他们排序是最容易想到的，所以第一种方式一般都能想到。第二种方式通过统计每个字符出现的次数，然后再把统计的结果转化为字符串也是可以实现的。

往期推荐

- 452，跳跃游戏
- 451，回溯和位运算解子集
- 450，什么叫回溯算法，一看就会，一写就废
- 446，回溯算法解黄金矿工问题

452，跳跃游戏

原创 山大王wld 数据结构和算法 9月19日

收录于话题
#算法图文分析

95个 >



One often meets his destiny on the road he takes to avoid it.

人永远无法逃避自己的宿命。

问题描述

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

示例 1:

> **输入**: [2,3,1,1,4]
> **输出**: true
> **解释**: 我们可以先跳 1 步，从位置 0 到达位置 1，然后再从位置 1 跳 3 步到达最后一个位置。

示例 2:

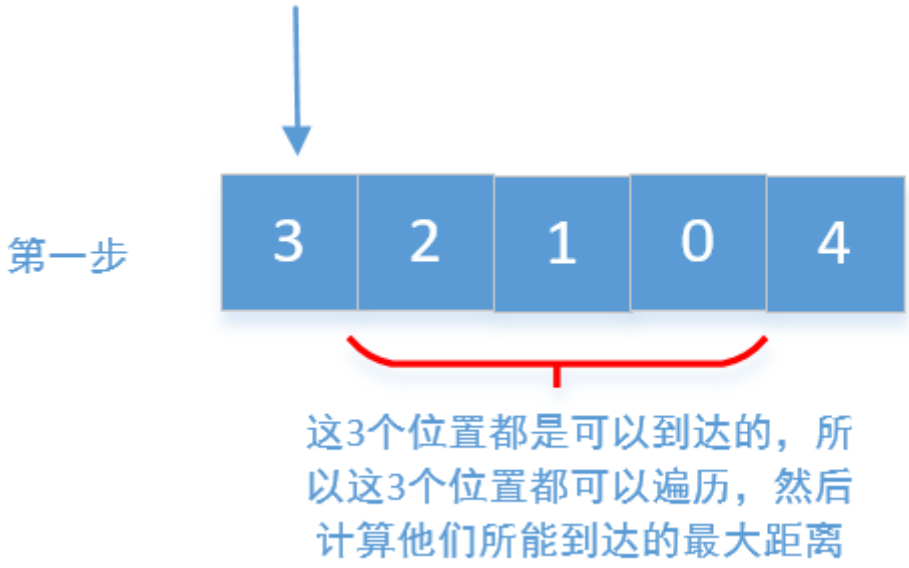
> **输入**: [3,2,1,0,4]
> **输出**: false
> **解释**: 无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0 ， 所以你永远不可能到达最后一个位置。

保存每步所能到达的最大距离

这题让求的是能否到达最后一个位置，我们先遍历数组的数字，然后保存下来他所能跳到的最大距离，如果能到达最后一个位置，直接返回true，如果不能到达就继续遍历，如果最大距离连下一步都到不了，就直接返回false。

比如第一步能跳到的最大距离是3，也就是说接下来的3个位置都是可以到达的，我们就要遍历接下来的3个位置，并记录这3个位置所能到达的最大距离，如果这3个位置的任何一个位置的最大距离能到达最后一个位置，直接返回true。

以示例2为例画个图来看下，第1个元素的值是3，所以接下来的3个位置都能到达，因为前3个位置所能跳到的最大距离是第4个位置，然后到第4个位置的时候，他能跳到的最大距离是0，不能到下一步了，直接返回false。




```java
public boolean canJump(int[] nums) {
    //maxStep表示能到达的距离
    int maxStep = 0;
    int length = nums.length;
    for (int i = 0; i < length; i++) {
        //如果跳不到位置i，直接返回false
        if (i > maxStep)
            return false;
        //如果能跳到位置i，就更新所能跳的最大距离
        maxStep = Math.max(maxStep, i + nums[i]);
        //如果能跳到最后的位置，说明能够成功，直接终止循环
        if (maxStep >= length)
            return true;
```

```
    }
    return true;
}
```

从后往前判断

可以逆向思维，这题说的是从前往后跳的，我们也可以从后往前来推断，从数组的最后第二位开始计算，如果当前的位置加上当前所能跳转的最大距离大于等于last，说明这一步跳转是没问题的，是可以到达last这一步（last初值是数组的最后一个元素）。能走到第一步，即last等于0的时候，说明是可以从位置0跳到最后一位的。

```
public boolean canJump(int[] nums) {
    //last表示的是能不能到达last这个位置
    int last = nums.length - 1;
    for (int i = nums.length - 2; i >= 0; i--) {
        //从倒数第2个位置往前遍历，如果当前位置能够跳
        //到last这个位置，就更新last，如果从当前位置
        //不能到达last这个位置就继续往前遍历
        if (i + nums[i] >= last)
            last = i;
    }
    //如果last等于0，说明可以从第一个位置跳到最后
    return last == 0;
}
```

总结

这题没有什么难度，第2种方式不太容易想到，一般更容易想到的是第一种解决方式，就是每走一步都要判断所能跳的最大距离，如果能够到达最后就直接返回，如果连下一步都到不了，那么就不可能到达最后了，直接返回false，否则就在当前位置所能到达最大位置前的元素都要遍历一遍，然后记录下他能跳的最大距离。

往期推荐

- 448，组合的几种解决方式
- 446，回溯算法解黄金矿工问题
- 445，BFS和DFS两种方式解岛屿数量
- 440，剑指 Offer-从上到下打印二叉树 II

443，滑动窗口最大值

原创 山大王wld 数据结构和算法 8月25日

收录于话题
#算法图文分析

95个 >



A major advantage of the remarkable people is: perseverance in the adverse and difficult encounter.

卓越的人的一大优点是：在不利和艰难的遭遇里百折不挠。

问题描述

给定一个数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

示例：

```
输入: nums = [1,3,-1,-3,5,3,6,7], 和 k = 3
输出: [3,3,5,5,6,7]
解释:

  滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
```

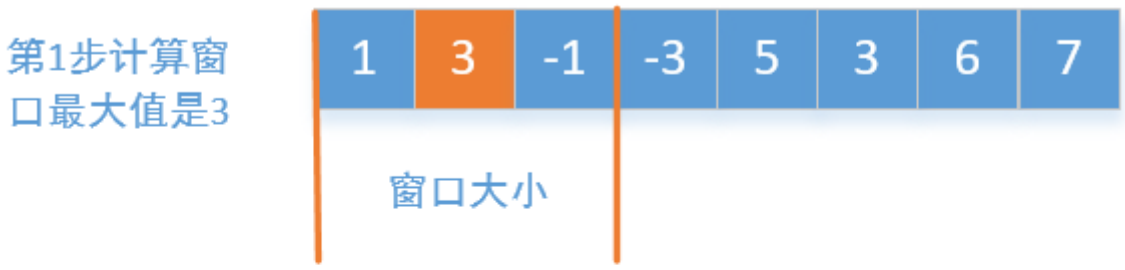
```
1 [3  -1  -3] 5  3  6  7       3
1  3 [-1  -3  5] 3  6  7       5
1  3  -1 [-3  5  3] 6  7       5
1  3  -1  -3 [5  3  6] 7       6
1  3  -1  -3  5 [3  6  7]      7
```

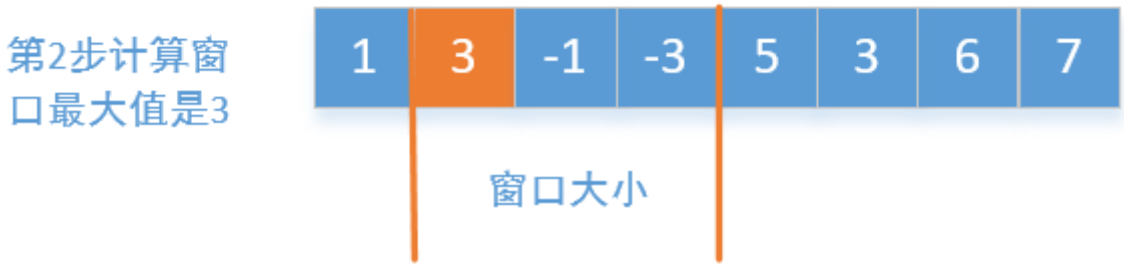
提示：

- 1 <= nums.length <= 10^5
- -10^4 <= nums[i] <= 10^4
- 1 <= k <= nums.length

暴力求解

最简单的一种方式就是暴力求解，原理其实很简单，就是窗口在往右滑动的过程中，每滑动一步就计算窗口内最大的值，就以上面的数据画个图来看下







代码比较简单，直接看下

```java
public int[] maxSlidingWindow(int[] nums, int k) {
    //边界条件判断
    if (nums == null || nums.length == 0)
        return new int[0];
    int res[] = new int[nums.length - k + 1];
    for (int i = 0; i < res.length; i++) {
        int max = nums[i];
        //在每个窗口内找到最大值
        for (int j = 1; j < k; j++) {
            max = Math.max(max, nums[i + j]);
        }
        res[i] = max;
    }
    return res;
}
```

如果看过之前讲的[378，数据结构-7,堆](#)我们还可以使用堆来解决，这里可以使用最大堆，堆顶的元素是最大的，因为这题求的就是窗口内的最大值，堆的大小就是窗口的大小。因为堆的每次删除和添加都会涉及到往下调整和往上调整，所以效率一般不是很高，也可以看下，这里就是用PriorityQueue来代替堆

```java
public int[] maxSlidingWindow(int[] nums, int k) {
    //边界条件的判断
    if (nums == null || k <= 0)
        return new int[0];
    int[] res = new int[nums.length - k + 1];
    int index = 0;
    //优先队列
    PriorityQueue<Integer> queue = new PriorityQueue<>((t1, t2) -> t2 - t1);
    for (int i = 0; i < nums.length; i++) {
        //元素添加到堆中
        queue.add(nums[i]);
        //如果堆的大小大于k，把最先加入的元素给移除
        if (queue.size() > k)
            queue.remove(nums[i - k]);
        if (i >= k - 1) {
            //把堆顶元素加入到数组中
            res[index++] = queue.peek();
        }
    }
    return res;
}
```

双端队列求解

我们知道一般的队列都是先进先出的，但双端队列两端都可以进出，如果对双端队列不熟悉的可以看下之前写的[359，数据结构-3,队列](#)。

使用双端队列首先要搞懂一个问题，就是在双端队列中，要始终保证队头是队列中最大的值。那怎么保证呢，就是在添加一个值之前，比他小的都要被移除掉，然后再添加这个值。我们举个例子，比如窗口大小是3，双端队列中依次添加3个值[4,2,5]，在添加5之前我们要把4和2给移除，让队列中只有一个5，因为窗口是往由滑动的，当添加5的时候，4和2都不可能再成为最大值了，并且4和2要比5还先出队列，搞懂了上面的过程我们随便画个图看下



在添加4之前要把-1和-3都给移除掉，因为有4在，-1和-3都永远不可能成为最大值，并且窗口是往右滑动的，-1和-3也会比4先出窗口



在添加9之前要把9前面所有的值都给移除掉，因为有9在，前面的都永远不可能成为最大值，并且窗口是往右滑动的，前面的值也都会比9先出窗口，这样队头的元素也就是最大值9了

搞懂了上面的过程代码就很容易写了，再看代码之前先来看一下双端队列常用的几个函数

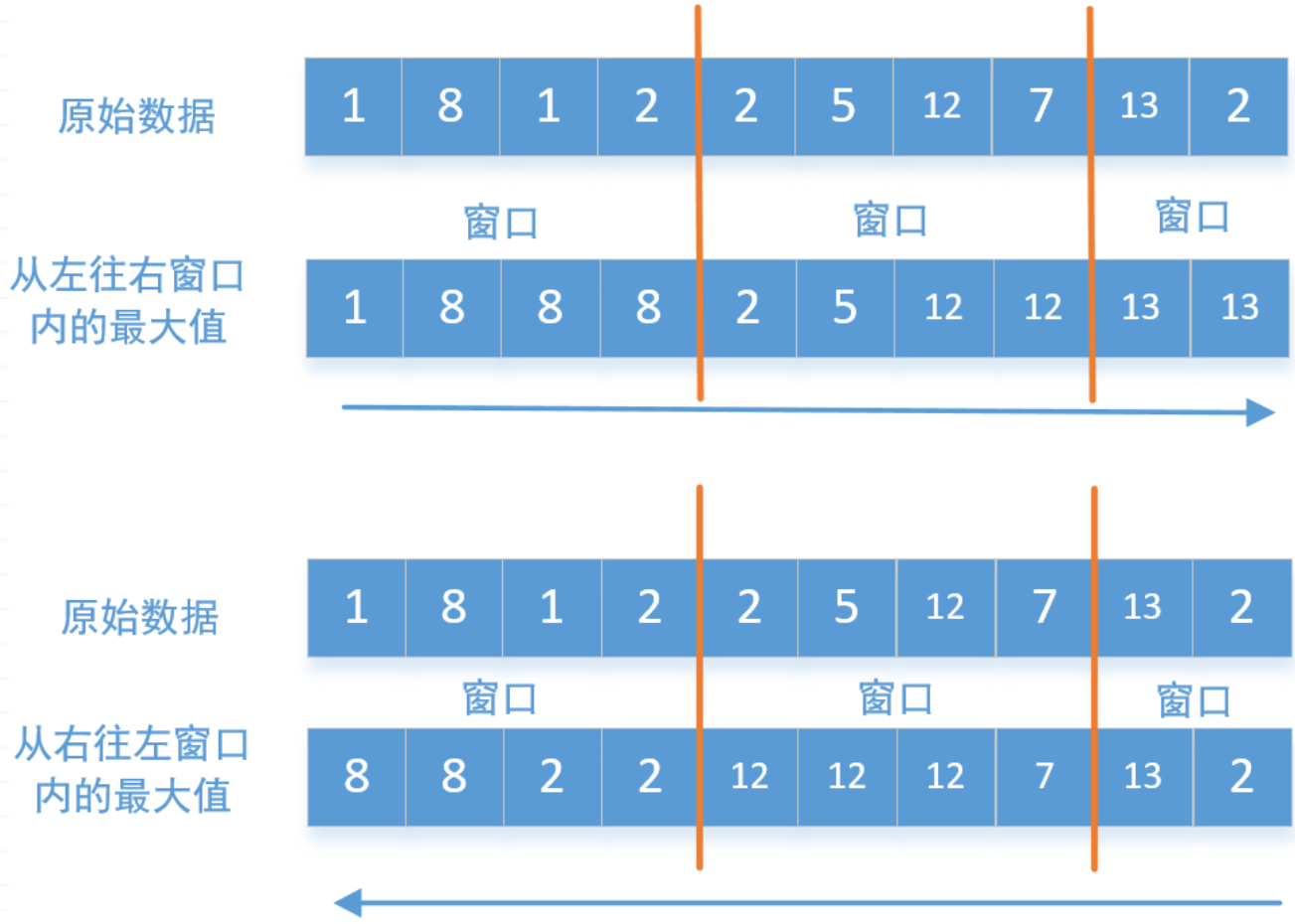


代码如下

```java
public int[] maxSlidingWindow(int[] nums, int k) {
    //边界条件的判断
    if (nums == null || k <= 0)
        return new int[0];
    int[] res = new int[nums.length - k + 1];
    int index = 0;
    //双端队列，就是两边都可以插入和删除数据的队列，注意这里存储
    //的是元素在数组中的下标，不是元素的值
    Deque<Integer> qeque = new ArrayDeque<>();
    for (int i = 0; i < nums.length; i++) {
        //如果队列中队头元素和当前元素位置相差i-k，相当于队头元素要
        //出窗口了，就把队头元素给移除，注意队列中存储
        //的是元素的下标（函数peekFirst()表示的是获取队头的下标，函数
        //pollFirst()表示的是移除队头元素的下标）
        if (!qeque.isEmpty() && qeque.peekFirst() <= i - k) {
            qeque.pollFirst();
        }
        //在添加一个值之前，前面比他小的都要被移除掉，并且还要保证窗口
        //中队列头部元素永远是队列中最大的
        while (!qeque.isEmpty() && nums[qeque.peekLast()] < nums[i]) {
            qeque.pollLast();
        }
        //当前元素的下标加入到队列的尾部
        qeque.addLast(i);
        //当窗口的长度大于等于k个的时候才开始计算（注意这里的i是从0开始的）
        if (i >= k - 1) {
            //队头元素是队列中最大的，把队列头部的元素加入到数组中
            res[index++] = nums[qeque.peekFirst()];
        }
    }
    return res;
}
```

两端扫描解决

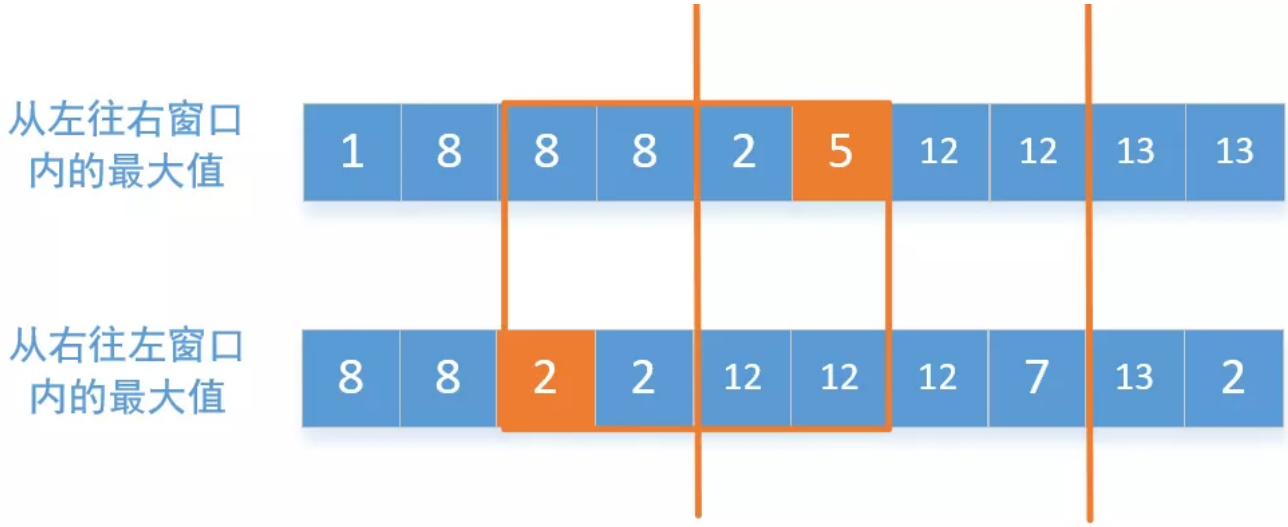
这个不太容易想到，就是根据窗口大小把数组分成n个窗口，每个窗口分别从左往右和从右往左扫描，记录扫描的最大值，就像下面这样



窗口分好之后一个从前往后扫描一个从后往前扫描，记录每个窗口扫描的最大值。我们取窗口内的最大值的时候，如果窗口在原数组中开始的下标正好是k的倍数，比如下面这样，他的最大值很容易找



但如果窗口滑动到下面这种情况下



如果要找这个窗口的最大值，我们就要选窗口内从左边扫描最后一个和从右边扫描最后一个（窗口内从左边数第一个）的最大值，也就是下面这样

```
res[j] = Math.max(maxRight[i], maxLeft[i + k - 1]);
```

为什么要这样选，大家可以想一下，因为如果选择从左边扫描的第一个值的话，那么这个值可能不是当前窗口内的值，同理从右边扫描的也一样。

搞懂了上面的分析过程代码就很容易写了

```
public int[] maxSlidingWindow(int[] nums, int k) {
    int len = nums.length;
    int[] maxLeft = new int[len];
    int[] maxRight = new int[len];
    //从左往右窗口的第一个最大值默认是数组第一个值
    maxLeft[0] = nums[0];
    //从右往左窗口的最后一个最大值是数组的最后一个值
    maxRight[len - 1] = nums[len - 1];

    for (int i = 1; i < len; i++) {
        //这里分别计算从前往后窗口的最大值和从后往前窗口的最大值。要搞懂这里的判断，如果
        //i % k == 0，表示到了下一个窗口
        maxLeft[i] = (i % k == 0) ? nums[i] : Math.max(maxLeft[i - 1], nums[i]);
        int j = len - i - 1;
        maxRight[j] = ((j + 1) % k == 0) ? nums[j] : Math.max(maxRight[j + 1], nums[j]);
    }
    //返回的结果值
    int[] res = new int[len - k + 1];
    for (int i = 0, j = 0; i < res.length; i++) {
        //取每个窗口内从左往右扫描的最后一个值和从右往左扫描的最后
        //一个值(如果从左边数是第一个)的最大值
        res[j++] = Math.max(maxRight[i], maxLeft[i + k - 1]);
    }
    return res;
}
```

总结

滑动窗口题，第一种暴力求解一般都能想到，但效率很差，最常见的就是第2种使用双端队列，第3种方式效率也挺高的，但一般不太容易想到。

436，剑指 Offer-顺时针打印矩阵

原创 山大王wld 数据结构和算法 8月19日

收录于话题
#剑指offer

27个 >



All over the place was six pence, but he looked up at the moon.

满地都是六便士，他却抬头看见了月亮。

问题描述

输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字。

示例 1：

```
输入：matrix = [[1,2,3],[4,5,6],[7,8,9]]
输出：[1,2,3,6,9,8,7,4,5]
```

示例 2：

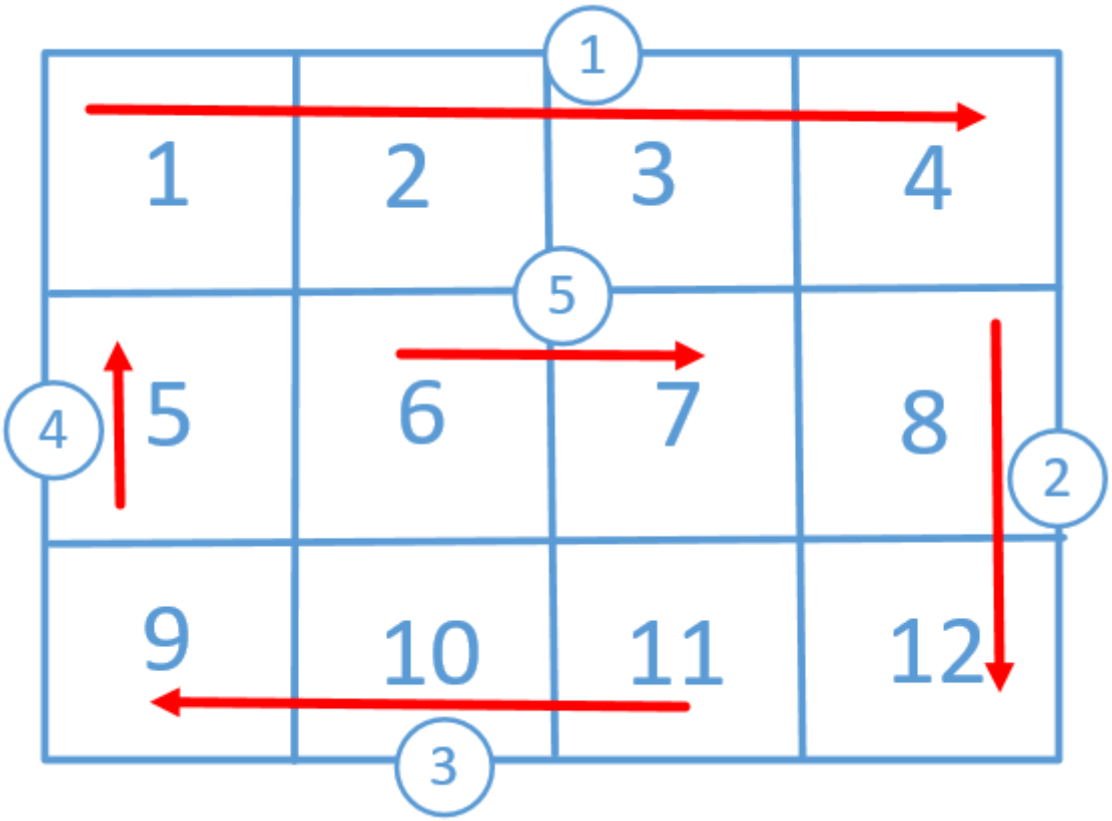
```
输入：matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
输出：[1,2,3,4,8,12,11,10,9,5,6,7]
```

限制：

- 0 <= matrix.length <= 100
- 0 <= matrix[i].length <= 100

问题分析

逆时针打印，也就是下面这张图这样

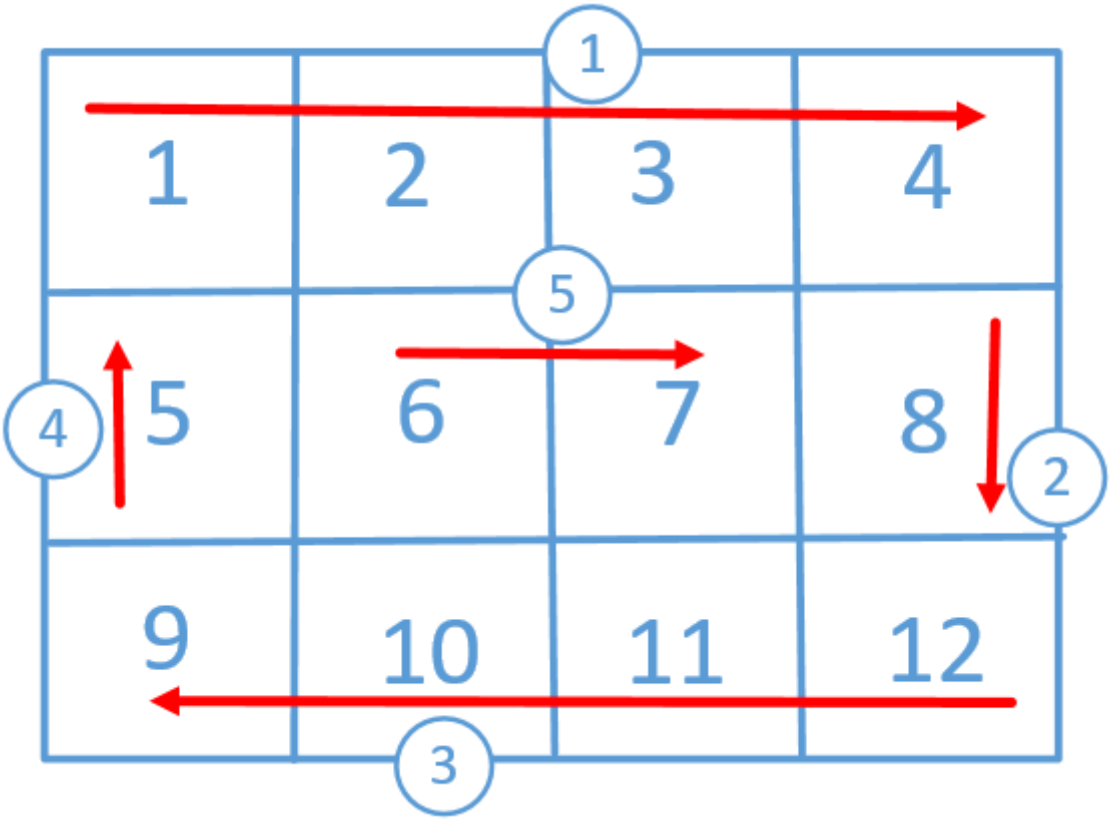


代码没什么难度，主要是在打印的时候做一些边界的判断，看下代码

```
public int[] spiralOrder(int[][] matrix) {
  if (matrix == null || matrix.length == 0)
    return new int[0];
  int m = matrix.length, n = matrix[0].length;
  int[] res = new int[m * n];
  int up = 0, down = m - 1, left = 0, right = n - 1, index = 0;
  while (true) {
    // 上面行，从左往右打印（行不变，改变列的下标）
    for (int col = left; col <= right; col++)
      res[index++] = matrix[up][col];
    if (++up > down)
      break;

    // 右边列，从上往下打印（列不变，改变行的下标）
    for (int row = up; row <= down; row++)
      res[index++] = matrix[row][right];
    if (--right < left)
      break;

    // 下面行，从右往左打印（行不变，改变列的下标）
    for (int col = right; col >= left; col--)
      res[index++] = matrix[down][col];
    if (--down < up)
      break;

    // 左边列，从下往上打印（列不变，改变行的下标）
```

```
      for (int row = down; row >= up; row--)
        res[index++] = matrix[row][left];
      if (++left > right)
        break;
    }
    return res;
  }
```

再来看一种方式，就是每次打印的时候上面一行和下面一行都是完整打印，左边一列和右边一列打印的值是夹在上下两行之间的，打印一圈之后，再缩小圈的范围。和上面有一点点区别，但原理还是没变。




```
public int[] spiralOrder(int[][] matrix) {
  if (matrix == null || matrix.length == 0)
    return new int[0];
  int n = matrix.length, m = matrix[0].length;
  int[] res = new int[m * n];
  int up = 0, down = n - 1;
  int left = 0, right = m - 1;
  int total = m * n;
  int index = 0;
  while (index < total) {
    //上面，从左往右打印
    for (int j = left; j <= right && index < total; j++)
      res[index++] = matrix[up][j];
    //右边，从上往下打印(注意这里i的取值范围)
    for (int i = up + 1; i <= down - 1 && index < total; i++)
      res[index++] = matrix[i][right];
    //下边，从右往左打印
    for (int j = right; j >= left && index < total; j--)
      res[index++] = matrix[down][j];
    //左边，从下往上打印(注意这里i的取值范围)
    for (int i = down - 1; i >= up + 1 && index < total; i--)
      res[index++] = matrix[i][left];
    left++;
    right--;
    up++;
```

```
26          down--;
27      }
28      return res;
29  }
```

总结

难度不大，控制好边界条件沿着上右下左的方向打印就行了。

往期推荐

- 420，剑指 Offer-回溯算法解矩阵中的路径
- 419，剑指 Offer-旋转数组的最小数字
- 406，剑指 Offer-二维数组中的查找
- 350，有序矩阵中第K小的元素

428，剑指 Offer-打印从1到最大的n位数

原创 山大王wld 数据结构和算法 8月11日

收录于话题
#剑指offer

27个 >



Life is a gift. We must celebrate it.

生活就是一份赠礼，每一天都值得我们庆祝。

问题描述

输入数字 n，按顺序打印出从 1 到最大的 n 位十进制数。比如输入 3，则打印出 1、2、3 一直到最大的 3 位数 999。

示例 1:

```
输入: n = 1
输出: [1,2,3,4,5,6,7,8,9]
```

说明：

- 用返回一个整数列表来代替打印
- n 为正整数

问题分析

今天的两道题都是剑指offer上的，应该是有史以来最简单的两道题了。这道题是剑指offer上的第17题。直接求出n位数的最大值，然后从1开始打印即可，没什么难度，看下代码

```
public int[] printNumbers(int n) {
  //统计总共需要打印多少个数字
  int size = (int) Math.pow(10, n) - 1;
  int[] res = new int[size];
  for (int i = 0; i < size; i++) {
    res[i] = i + 1;
  }
  return res;
}
```

往期推荐

- 423，动态规划和递归解最小路径和
- 417，BFS和DFS两种方式求岛屿的最大面积
- 413，动态规划求最长上升子序列
- 397，双指针求接雨水问题

427，剑指 Offer-数值的整数次方

原创 山大王wld 数据结构和算法 8月10日

收录于话题
#剑指offer

27个 >



Do or do not. There is no try.

要么勇敢去做，要么果断放手，没有只是试试这一说。

问题描述

实现函数double Power(double base, int exponent)，求base的exponent次方。不得使用库函数，同时不需要考虑大数问题。

示例 1:

```
输入: 2.00000, 10
输出: 1024.00000
```

示例 2:

```
输入: 2.10000, 3
输出: 9.26100
```

示例 3:

> 输入: 2.00000, -2
> 输出: 0.25000
> 解释: 2^(-2) = 1/(2^2) = 1/4 = 0.25

说明:

- -100.0 < x < 100.0
- n 是 32 位有符号整数，其数值范围是 [−2^31, 2^31 − 1] 。

问题分析

这题看起来很简单，但能一步写成功很不容易，我们先来分析下。

- 当exponent是0的时候，直接返回1即可，
- 当exponent小于0的时候，需要把它转化为正数才能更方便计算，同时base要变为1/base。
- 当exponent大于0的时候要分为两种情况，一种是偶数，一种是奇数。

1，如果exponent是偶数我们只需要计算Power(base*base, exponent/2)。举个例子，比如我们要计算Power（3，8），我们可以改为Power（3*3，8/2），也就是Power（9，4）。

2，如果exponent是奇数，我们只需要计算base*Power(base*base, exponent/2)，比如Power（3，9），我们只需要计算3*Power（3*3，9/2），也就是3*Power（9，4）。

所以代码很容易写，我们来看下

```
1   public double myPow(double x, int n) {
2     //如果n等于0，直接返回1
3     if (n == 0)
4       return 1;
5     //如果n小于0，把它改为正数
6     if (n < 0)
7       return myPow(1 / x, -n);
8     //根据n是奇数还是偶数来做不同的处理
9     return (n % 2 == 0) ? myPow(x * x, n / 2) : x * myPow(x * x, n / 2);
10  }
```

上面代码有一点瑕疵，就是如果当n=Integer.MIN_VALUE的时候，第7行代码就会出问题，这是因为数字溢出问题，导致Integer.MIN_VALUE的相反数还是他自己，所以会一直调用，直到最后出现堆栈溢出异常。所以有一种方式就是把1/x提取出来一个，代码如下

```
1   public double myPow(double x, int n) {
2     if (n == 0)
3       return 1;
4     //如果n小于0，把它改为正数，并且把1/x提取出来一个
5     if (n < 0)
6       return 1 / x * myPow(1 / x, -n - 1);
7     return (n % 2 == 0) ? myPow(x * x, n / 2) : x * myPow(x * x, n / 2);
8   }
```

这样代码就不会有问题了。我们还可以把他改为非递归的写法，这样在计算的时候就不需要关心exponent究竟是正数还是负数了，只需要在最后返回的时候判断一下即可，代码也很简单

```
public double myPow(double x, int n) {
  double result = 1.0;
  for (int i = n; i != 0; i /= 2, x *= x) {
    if (i % 2 != 0) {
      //i是奇数
      result *= x;
    }
  }
  return n < 0 ? 1.0 / result : result;
}
```

总结

这题也可以一个一个相乘，这样运算量是比较大的，所以我们可以根据指数是偶数还是奇数来减少计算。

往期推荐

- 425，剑指 Offer-二进制中1的个数
- 424，剑指 Offer-剪绳子
- 419，剑指 Offer-旋转数组的最小数字
- 418，剑指 Offer-斐波那契数列

424，剑指 Offer-剪绳子

原创 山大王wld 数据结构和算法 8月6日

收录于话题
#剑指offer

27个 >



Smile and maybe tomorrow you'll see sun come shinning through.

微笑吧，或许明天你就会看到太阳照耀着你。

问题描述

给你一根长度为 n 的绳子，请把绳子剪成整数长度的 m 段（m、n都是整数，n>1并且m>1），每段绳子的长度记为 k[0],k[1]...k[m-1] 。

请问 k[0]*k[1]*...*k[m-1] 可能的最大乘积是多少？

例如，当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到的最大乘积是18。

示例 1：

```
输入: 2
输出: 1
解释: 2 = 1 + 1, 1 × 1 = 1
```

示例 2：

> 输入：10
> 输出：36
> 解释：10 = 3 + 3 + 4，3 × 3 × 4 = 36

提示：

- `2 <= n <= 58`

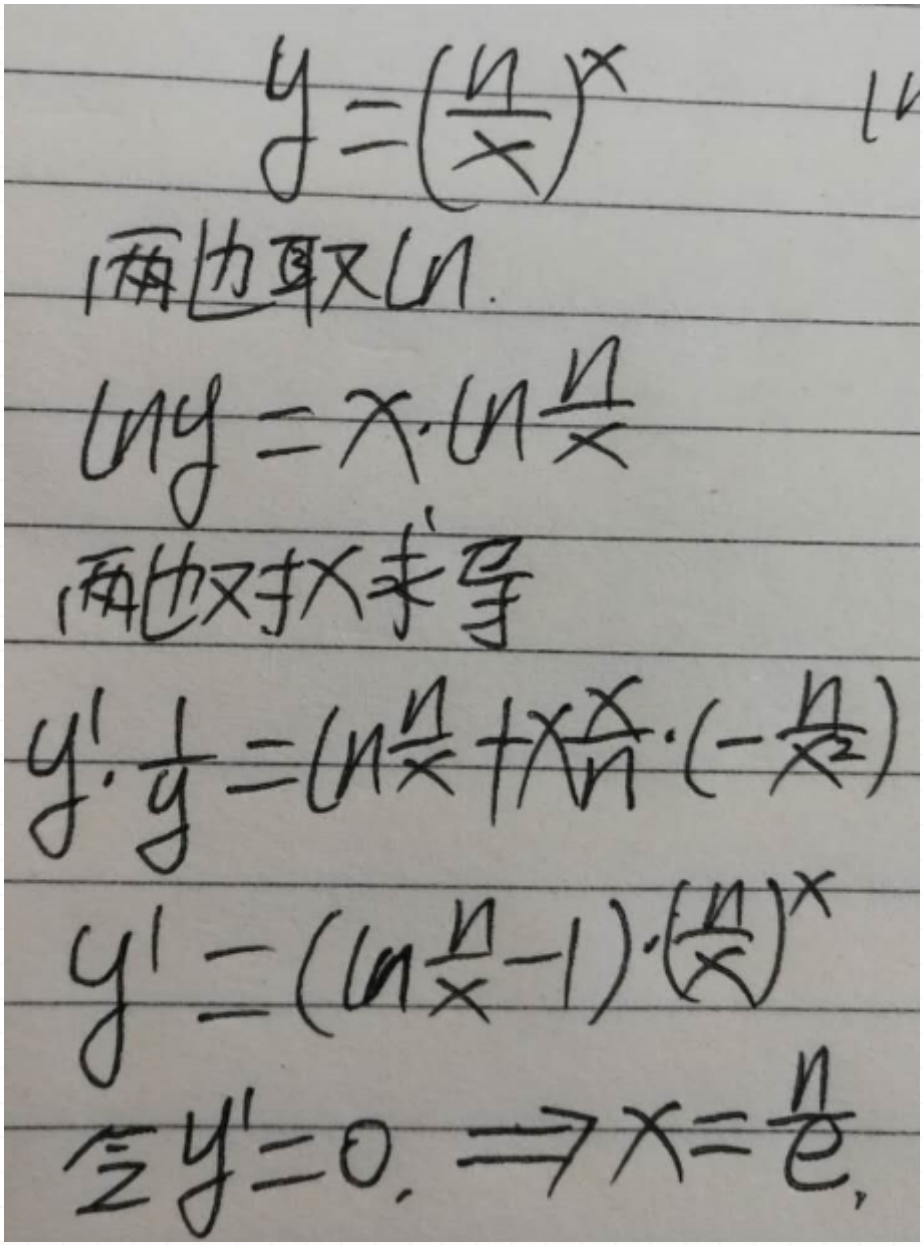
数学方式解决

在做这题之前我们先来看这样一个问题，一个整数先把他分成两部分，x+y=n（假设x>=y并且x-y<=1,也就是说x和y非常接近）那么乘积是x*y。然后我们再把这两部分的差放大(x+1)+(y-1)=n(假设x>=y)；他们的乘积是（x+1）*(y-1)=x*y-(x-y)-1，很明显是小于x*y的，所以我们得出结论，如果把整数n分为两部分，那么这两部分的值相差越小乘积越大。

同理还可以证明如果分成3部分，4部分……也是相差越小乘积会越大。

$$\left(\frac{a_1+a_2+.....+a_n}{n}\right)^n \geqslant a_1a_2.....a_n$$，当且仅当$a_1=a_2=.....=a_n$时取等号。

根据上面的证明，如果我们把长度为n的绳子分为x段，则每段只有在长度相等的时候乘积最大，那么每段的长度是n/x。所以他们的乘积是(n/x)^x。我们来对这个函数求导



通过对函数求导我们发现，当x=n/e的时候，也就是每段绳子的长度是n/x=n/(n/e)=**e**的时候乘积最大。我们知道e=2.718281828459。而题中我们的绳子剪的长度都是整数，所以不可能取e，我们只能取接近e的值，也就是3的时候乘积最大。

但也有例外，当n<=4的时候会有特殊情况，因为2*2>1*3。明白了这点代码就容易多了，如果n大于4，我们不停的把绳子减去3，来看下代码

```
public int cuttingRope(int n) {
  if (n == 2 || n == 3)
    return n - 1;
  int res = 1;
  while (n > 4) {
    //如果n大于4，我们不停的让他减去3
    n = n - 3;
    //计算每段的乘积
    res = res * 3;
  }
  return n * res;
}
```

或者如果不想使用循环，我们还可以使用公式

```
public int cuttingRope(int n) {
    if (n == 2 || n == 3)
        return n - 1;
    else if (n % 3 == 0) {
        //如果n是3的倍数，绳子全部剪为3
        return (int) Math.pow(3, n / 3);
    } else if (n % 3 == 1) {
        //如果n对3求余等于1，我们剪出一个长度为4的，其他长度都是3
        return 4 * (int) Math.pow(3, (n - 4) / 3);
    } else {
        //如果n对3求余等于2，我们剪出一个长度为2的，其他长度都是3
        return 2 * (int) Math.pow(3, n / 3);
    }
}
```

动态规划解决

定义一个数组dp，其中dp[i]表示的是长度为i的绳子能得到的最大乘积。我们先把长度为i的绳子拆成两部分，一部分是j，另一部分是i-j，那么会有下面4种情况

1，j和i-j都不能再拆了
- dp[i]=j*(i-j);

2，j能拆，i-j不能拆
- dp[i]=dp[j]*(i-j);

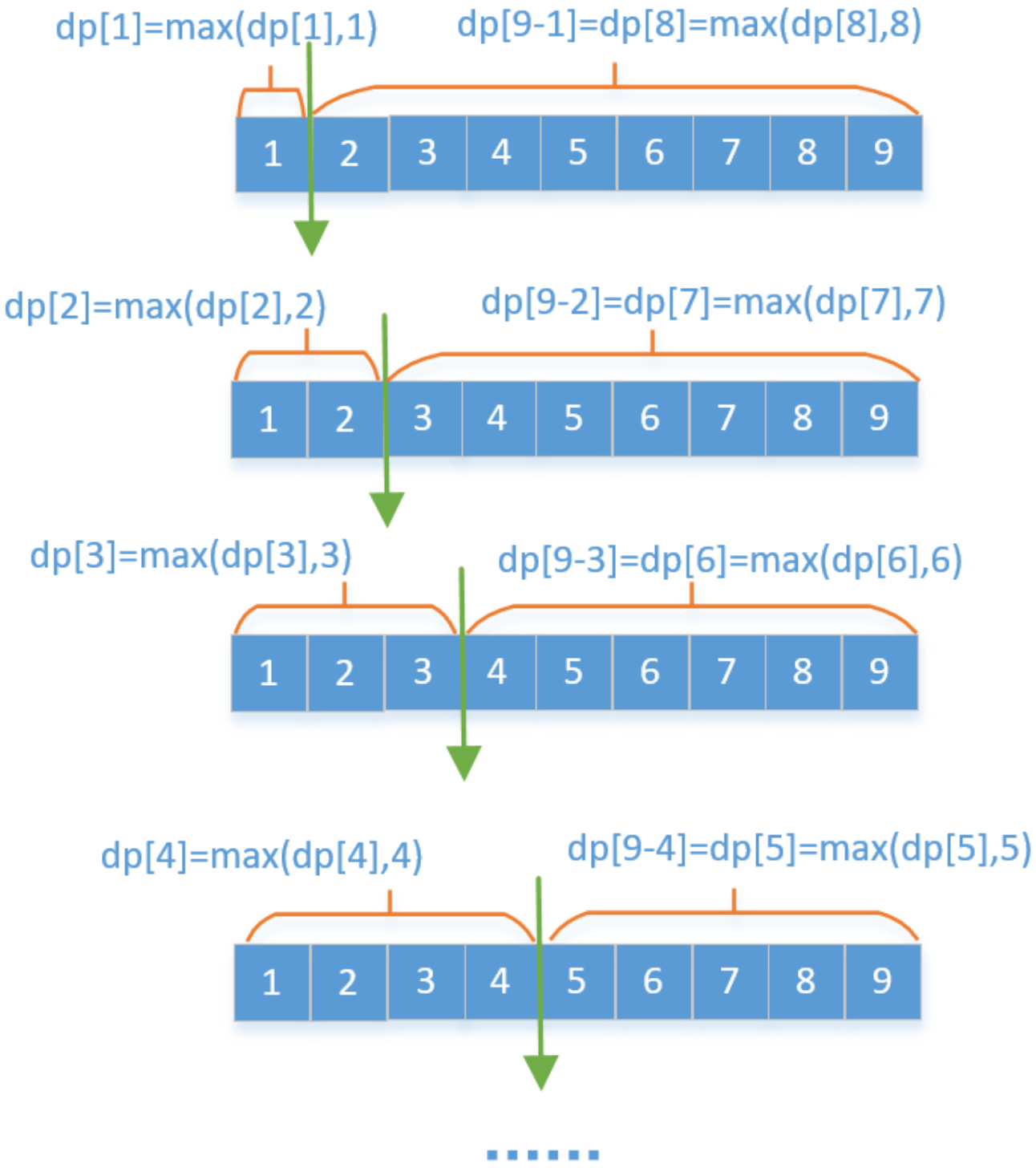
3，j不能拆，i-j能拆
- dp[i]=j*dp[i-j];

4，j和i-j都能拆
- dp[i]=dp[j]*dp[i-j];

我们取上面4种情况的最大值即可。我们把它整理一下，得到递推公式如下

dp[i] = max(dp[i], (max(j, dp[j])) * (max(i - j, dp[i - j])));

比如我们想计算长度为9的绳子，画个图来看一下



计算长度为9的绳子之前，我们必须要先计算长度为8的绳子。对于长度为9的绳子我们可以先分为两部分，每一部分都取最大值，然后相乘。

最后再来看下代码

```
public int cuttingRope(int n) {
    int[] dp = new int[n + 1];
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        for (int j = 1; j < i; j++) {
            dp[i] = Math.max(dp[i], (Math.max(j, dp[j])) * (Math.max(i - j, dp[i - j])));
        }
    }
    return dp[n];
}
```

总结

这题应该说更像是一道数学题，使用数学的方式很容易解决，动态规划的递推公式可能不太容易想到。

往期推荐

- 422，剑指 Offer-使用DFS和BFS解机器人的运动范围
- 419，剑指 Offer-旋转数组的最小数字
- 418，剑指 Offer-斐波那契数列
- 416，剑指 Offer-用两个栈实现队列

421，在排序数组中查找元素的第一个和最后一个位置-对二分法查找的改造

原创 山大王wld 数据结构和算法 8月5日

收录于话题
#算法图文分析

95个 >



Being born in a duck yard does not matter, if only you are hatched from a swan's egg.

只要是从天鹅蛋孵出来的，即使生在养鸭场也没有关系。

问题描述

给定一个按照升序排列的整数数组 nums，和一个目标值 target。找出给定目标值在数组中的开始位置和结束位置。

你的算法时间复杂度必须是 O(log n) 级别。
如果数组中不存在目标值，返回 [-1, -1]。

示例 1:

```
输入: nums = [5,7,7,8,8,10],
target = 8

输出: [3,4]
```

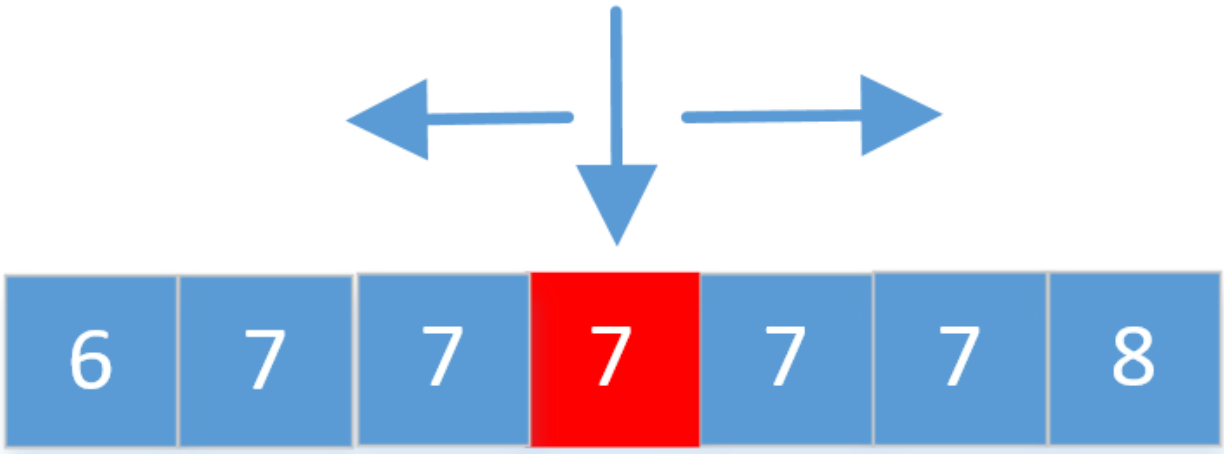
示例 2:

> 输入: nums = [5,7,7,8,8,10],
> target = 6
>
> 输出: [-1,-1]

二分法查找

题中说了是升序的数组，也就是排过序的，对于排过序的数组查找我们很容易想到的就是二分法。这里要返回的是目标值在数组中的开始位置和结束位置，因为相同的数字在排序数组中肯定是挨着的，所以我们通过二分法查到之后，还要往前和往后继续查找，直到不等于目标值为止。

如果是做Android的并且经常看源码的可能知道，Android中有个类ArrayMap，他存储的时候hash值是排过序的，查找的时候也是通过二分法查找，但有可能hash值会有冲突，所以他查找之后也是分别往前和往后继续查找然后再比较key值，和这题解法很相似。我们来画个简图看一下



比如我们通过二分法查找7，然后还要往他的前面和后面继续查找，目的是要找到最开始7和最后7的位置，来看下代码

```
1   public int[] searchRange(int[] nums, int target) {
2     int find = searchRangeHelper(nums, target);
3     //如果没找到，返回{-1, -1}
4     if (find == -1)
5       return new int[]{-1, -1};
6     int left = find - 1;
7     int right = find + 1;
8     //查看前面是否还有target
9     while (left >= 0 && nums[left] == target)
10      left--;
11    //查看后面是否还有target
12    while (right < nums.length && nums[right] == target)
13      right++;
14    return new int[]{left + 1, right - 1};
```

```
}

//二分法查找
public int searchRangeHelper(int[] nums, int target) {
  int low = 0;
  int high = nums.length - 1;
  while (low <= high) {
    int mid = low + (high - low) / 2;
    int midVal = nums[mid];
    if (midVal < target) {
      low = mid + 1;
    } else if (midVal > target) {
      high = mid - 1;
    } else {
      return mid;
    }
  }
  return -1;
}
```

二分法的另一种写法

二分查找一般我们找到某个值之后会直接返回。其实我们有时候还可以对二分法进行改造，当查找某个值的时候不直接返回，而是要继续查找，直到左右两个指针相遇为止。像下面这样，代码中有详细的注释，可以看一下

```
public int[] searchRange(int[] nums, int target) {
  //查找第一个出现的target
  int first = searchRangeHelper(nums, target);
  //判断有没有查找到
  if (first < nums.length && nums[first] == target) {
    //如果查找到了，说明有这个值，我们再来查(target + 1)，如果有这个值，
    //那么查找的结果也是第一次出现的(target + 1)的下标，我们减去1，就是target
    //最后一次出现的下标。如果没有(target + 1)这个值，那么查找的结果就是第一个
    // 大于target的下标，我们减去1也是target最后一次出现的下标。所以这里
    // 无论有没有(target + 1)这个值，都不影响
    int last = searchRangeHelper(nums, target + 1) - 1;
    return new int[]{first, last};
  } else {
    //没有找到这个值，直接返回{-1, -1}
    return new int[]{-1, -1};
  }
}

//如果数组nums中有多个target，那么返回值就是第一个出现的target的下标
//如果数组nums中没有target，那么返回的就是第一个大于target的下标
public static int searchRangeHelper(int[] nums, double target) {
  int low = 0, high = nums.length - 1;
  while (low <= high) {
    int m = low + (high - low) / 2;
    if (target > nums[m])
      low = m + 1;
    else
      high = m - 1;
  }
  return low;
}
```

看到这里可能有的同学灵光乍现，通过二分法能找到target第一次出现的位置，那么通过二分法能不能找到target最后一次出现的位置。当然也是可以的，代码在下面给你准备好了

```
public int[] searchRange(int[] nums, int target) {
  int first = searchFirst(nums, target);
```

```
    //判断有没有查找到
    if (first < nums.length && nums[first] == target) {
      int last = searchLast(nums, target);
      return new int[]{first, last};
    } else {
      //没有找到这个值，直接返回{-1, -1}
      return new int[]{-1, -1};
    }
  }

  //如果数组nums中有多个target，那么返回值就是第一个出现的target的下标
  //如果数组nums中没有target，那么返回的就是第一个大于target的下标
  public static int searchFirst(int[] nums, double target) {
    int low = 0, high = nums.length - 1;
    while (low <= high) {
      int m = low + (high - low) / 2;
      if (target > nums[m])
        low = m + 1;
      else
        high = m - 1;
    }
    return low;
  }

  public static int searchLast(int[] nums, double target) {
    int low = 0, high = nums.length - 1;
    while (low <= high) {
      int m = low + (high - low) / 2;
      if (target >= nums[m])
        low = m + 1;
      else
        high = m - 1;
    }
    return high;
  }
```

总结

以前对二分法的查找，我们是找到之后就返回。但如果有多个重复的值，我们是没法确定返回的是哪个值的下标。今天这里我们对二分法进行了改造，如果有多个重复的值，你想返回第一个值或者最后一个值的下标都是可以的。

往期推荐

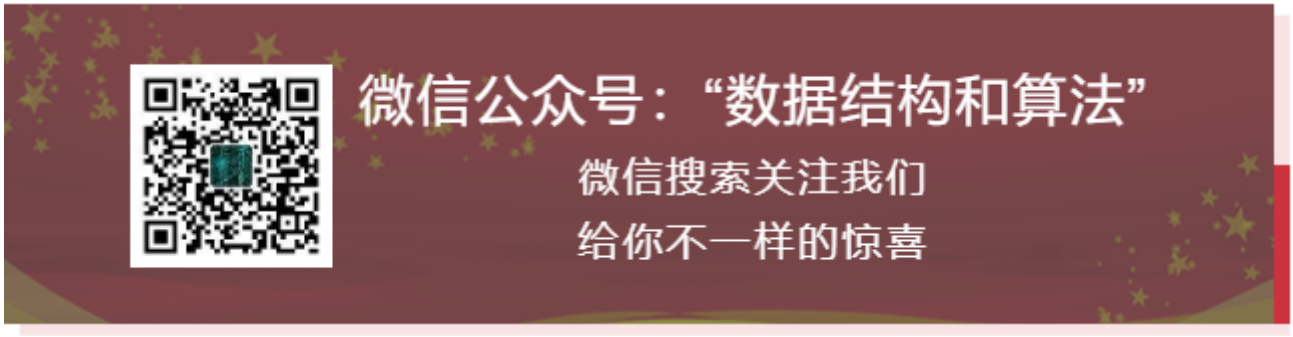
- 397，双指针求接雨水问题
- 396，双指针求盛最多水的容器
- 393，括号生成
- 391，回溯算法求组合问题

419，剑指 Offer-旋转数组的最小数字

原创 山大王wld 数据结构和算法 8月3日

收录于话题
#剑指offer

27个 >



You can never replace anyone. What is lost is lost.

每个人都是无可替代的，失去了便是失去了。

问题描述

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个递增排序的数组的一个旋转，输出旋转数组的最小元素。例如，数组 [3,4,5,1,2] 为 [1,2,3,4,5] 的一个旋转，该数组的最小值为1。

示例 1：

```
输入：[3,4,5,1,2]
输出：1
```

示例 2：

```
输入：[2,2,2,0,1]
输出：0
```

逐个查找和排序查找

这题要求的是数组中的最小元素，所以最简单的两种方式，一个是逐个查找，一个是排序之后再查找，先看一下逐个查找，就是一个个查找，把数组的元素全部都遍历一遍，即可找到最小值

```
public int minArray(int[] numbers) {
  int min = numbers[0];
  for (int i = 1; i < numbers.length; i++) {
    if (min > numbers[i])
      min = numbers[i];
  }
  return min;
}
```

再来看一下排序查找，我们使用从小到大的顺序对数组进行升序排列，排序完之后直接返回数组的第一个元素即可。

```
public int minArray(int[] numbers) {
  Arrays.sort(numbers);
  return numbers[0];
}
```

二分法查找

首先来说下上面两种方式肯定是都能解决的，如果来解决这道题总觉得不是很合适。因为这道题说了，他本来是一个递增排序的数组，然后经过了一次旋转。一想到排序数组的查找，第一个应该想到的是二分法。前面我们讲查找的时候提到过二分法查找，当然还有插值查找，有兴趣的可以看下

202，查找-二分法查找

203，查找-插值查找

204，查找-斐波那契查找

我们看下二分法查找的正常代码是什么样的

```
public int search(int[] nums, int target) {
  int left = 0, right = nums.length - 1;
  while (left <= right) {
    int mid = left + (right - left) / 2;
    if (nums[mid] == target)
      return mid;
    if (target < nums[mid])
      right = mid - 1;
    else
      left = mid + 1;
  }
  return -1;
}
```

但这道题的数组在中间有一次旋转，所以不能直接用以前的那种二分法来查找，我们可以参照上面代码换种思路，我们不断的缩小查找范围，当查找范围的长度为1的时候返回，下面代码中也就是left等于right的时候。

```
public int minArray(int[] numbers) {
  int left = 0, right = numbers.length - 1;
```

```
    while (left < right) {
      //找出left和right中间值的索引
      int mid = left + (right - left) / 2;
      if (numbers[mid] > numbers[right]) {
        //如果中间值大于最右边的值，说明旋转之后最小的
        //数字肯定在mid的右边，比如[3, 4, 5, 6, 7, 1, 2]
        left = mid + 1;
      } else if (numbers[mid] < numbers[right]) {
        //如果中间值小于最右边的值，说明旋转之后最小的
        //数字肯定在mid的前面，比如[6, 7, 1, 2, 3, 4, 5],
        //注意这里mid是不能减1的，比如[3，1，3]，我们这里只是
        //证明了numbers[mid]比numbers[right]小，但有可能
        //numbers[mid]是最小的，所以我们不能把它给排除掉
        right = mid;
      } else {
        //如果中间值等于最后一个元素的值，我们是没法确定最小值是
        // 在mid的前面还是后面，但我们可以缩小查找范围，让right
        // 减1，因为即使right指向的是最小值，但因为他的值和mid
        // 指向的一样，我们这里并没有排除mid，所以结果是不会有影响的。
        //比如[3，1，3，3，3，3，3]和[3，3，3，3，3，1，3],中间的值
        //等于最右边的值，但我们没法确定最小值是在左边还是右边
        right--;
      }
    }
    return numbers[left];
  }
```

总结

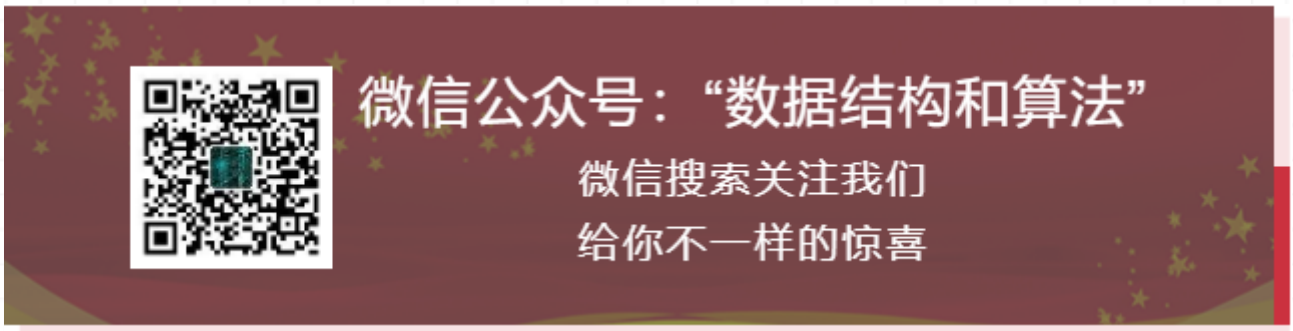
只要是排过序的数组，我们最应该想到的就是二分法，然后再考虑其他的

往期推荐

- 418，剑指 Offer-斐波那契数列
- 416，剑指 Offer-用两个栈实现队列
- 410，剑指 Offer-从尾到头打印链表
- 406，剑指 Offer-二维数组中的查找

418，剑指 Offer-斐波那契数列

原创 山大王wld 数据结构和算法 7月31日



We don't get a chance to do that many things, and every one should be really excellent.

我们这一生能做的事情有限，所以要把每件事都做到完美。

问题描述

写一个函数，输入 n ，求斐波那契（Fibonacci）数列的第 n 项。斐波那契数列的定义如下：

> F(0) = 0, F(1) = 1
> F(N) = F(N - 1) + F(N - 2),
> 其中 N > 1.

斐波那契数列由 0 和 1 开始，之后的斐波那契数就是由之前的两数相加而得出。

答案需要取模 1e9+7（1000000007），如计算初始结果为：1000000008，请返回 1。

示例 1：

> 输入：n = 2
> 输出：1

示例 2：

> 输入：n = 5
> 输出：5

提示：

- 0 <= n <= 100

递归解法

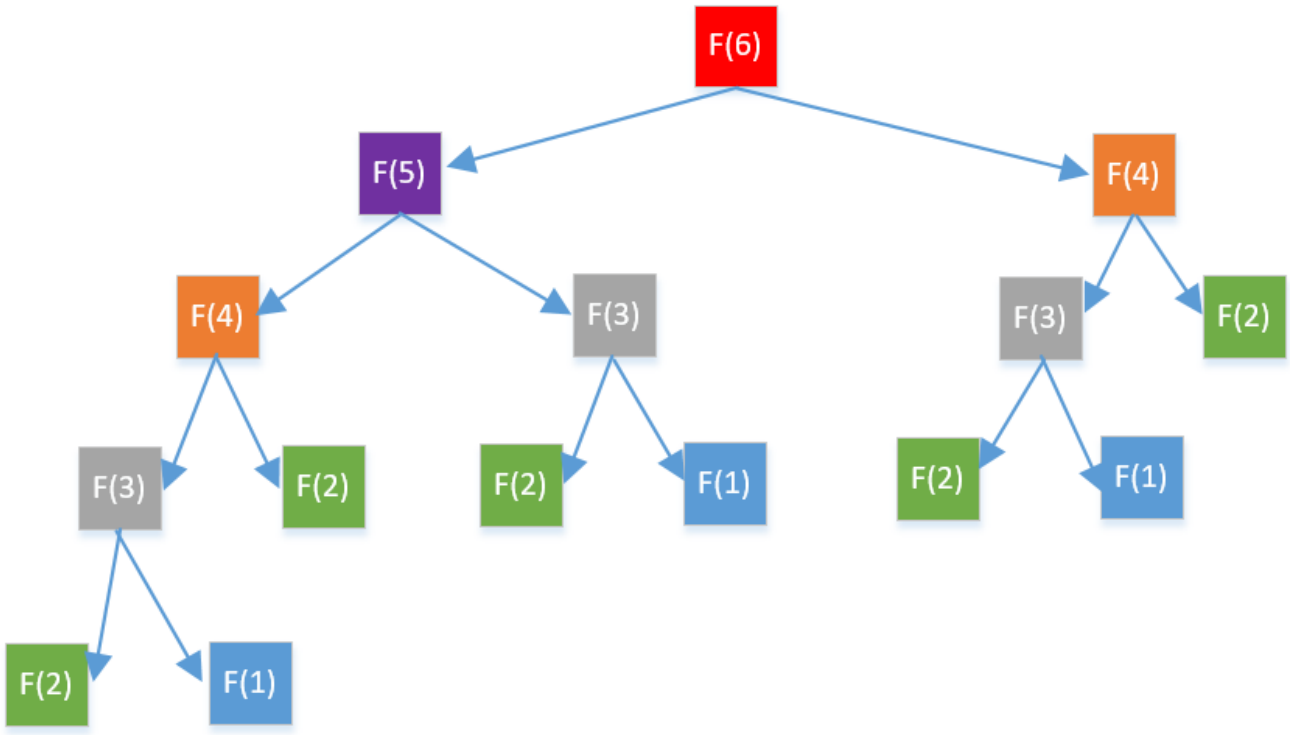
前面讲[356，青蛙跳台阶相关问题](#)的时候提到过斐波那契数列，代码比较简单

```
1  public int fib(int n) {
2    if (n < 2)
3      return n;
4    return fib(n - 1) + fib(n - 2);
5  }
```

当n很大的时候可能会出现数字溢出，所以我们需要用结果对1000000007求余，但实际上可能还没有执行到最后一步就已经溢出了，所以我们需要对每一步的计算都要对1000000007求余，代码如下

```
1  int constant = 1000000007;
2
3  public int fib(int n) {
4    if (n < 2)
5      return n;
6    int first = fib(n - 1) % constant;
7    int second = fib(n - 2) % constant;
8    return (first + second) % constant;
9  }
```

之前讲过斐波那契数列递归的时候会造成大量的重复计算，比如就计算fib(6)为例来看下



我们看到上面相同颜色的都是重复计算，当n越大，重复的越多，所以我们可以使用一个map把计算过的值存起来，每次计算的时候先看map中有没有，如果有就表示计算过，直接从map中取，如果没有就先计算，计算完之后再把结果存到map中。

```
int constant = 1000000007;

public int fib(int n) {
  return fib(n, new HashMap());
}

public int fib(int n, Map<Integer, Integer> map) {
  if (n < 2)
    return n;
  if (map.containsKey(n))
    return map.get(n);
  int first = fib(n - 1, map) % constant;
  map.put(n - 1, first);
  int second = fib(n - 2, map) % constant;
  map.put(n - 2, second);
  int res = (first + second) % constant;
  map.put(n, res);
  return res;
}
```

非递归解法

我们还可以把斐波那契递归改为非递归，代码很简单，可以看下

```
public int fib(int n) {
  int constant = 1000000007;
  int first = 0;
  int second = 1;
  while (n-- > 0) {
    int temp = first + second;
    first = second % constant;
    second = temp % constant;
  }
  return first;
}
```

总结

斐波那契数列又称黄金分割数列，常见的比如兔子的繁殖，青蛙跳台阶等问题。递归方式解决是最容易理解的，但递归会包含大量的重复计算，效率很差，一般还是改为非递归为好。如果n不是很大的话，我们还可以使用公式来算，他的通项公式如下

$$a_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

415，最佳观光组合

原创 山大王wld 数据结构和算法 7月30日

收录于话题
#算法图文分析

95个 >



You must practice being stupid, dumb, unthinking, empty.

你得学着痴一点，钝一些，少想一些，彻底放空自己。

问题描述

给定正整数数组 A，A[i] 表示第 i 个观光景点的评分，并且两个景点 i 和 j 之间的距离为 j - i。

一对景点（i < j）组成的观光组合的得分为（A[i] + A[j] + i - j）：景点的评分之和减去它们两者之间的距离。

返回一对观光景点能取得的最高分。

示例：

> 输入：[8,1,5,2,6]
> 输出：11
> 解释：i = 0, j = 2,

> A[i] + A[j] + i - j = 8 + 5 + 0 - 2 = 11

提示：

1. 2 <= A.length <= 50000
2. 1 <= A[i] <= 1000

暴力求解

这题其实最容易想到的就是暴力求解，每两个两个计算，记录下最大值，最后再返回，代码比较简单

```
public int maxScoreSightseeingPair(int[] A) {
    int res = 0;
    for (int i = 0; i < A.length; i++) {
        for (int j = i + 1; j < A.length; j++) {
            res = Math.max(res, A[i] + i + A[j] - j);
        }
    }
    return res;
}
```

根据公式求解

暴力求解毕竟效率不高，我们还可以根据上面的计算公式来找规律。根据公式

res = A[i] + A[j] + i - j （i < j）

我们求每个景点j的时候，他的A[j] - j 实际上是固定的，要想让res最大，我们就要想办法让A[i] + i尽可能大。所以我们可以使用一个变量cur记录下遍历过的最大值（当前值+当前下标），所以代码很容易想到

```
public int maxScoreSightseeingPair(int[] A) {
    int res = 0, cur = A[0] + 0;
    for (int j = 1; j < A.length; ++j) {
        res = Math.max(res, cur + A[j] - j);
        cur = Math.max(cur, A[j] + j);
    }
    return res;
}
```

总结

这题没什么难度，主要还是要搞懂这个公式。

往期推荐

- 413，动态规划求最长上升子序列

412，判断子序列

原创 山大王wld 数据结构和算法 7月28日

收录于话题
#算法图文分析

95个 >



Tough time don't last, tough people do.

没有过不去的坎，只有打不倒的人。

问题描述

给定字符串 s 和 t ，判断 s 是否为 t 的子序列。

示例 1:

s = "abc", t = "ahbgdc"

返回 true.

示例 2:

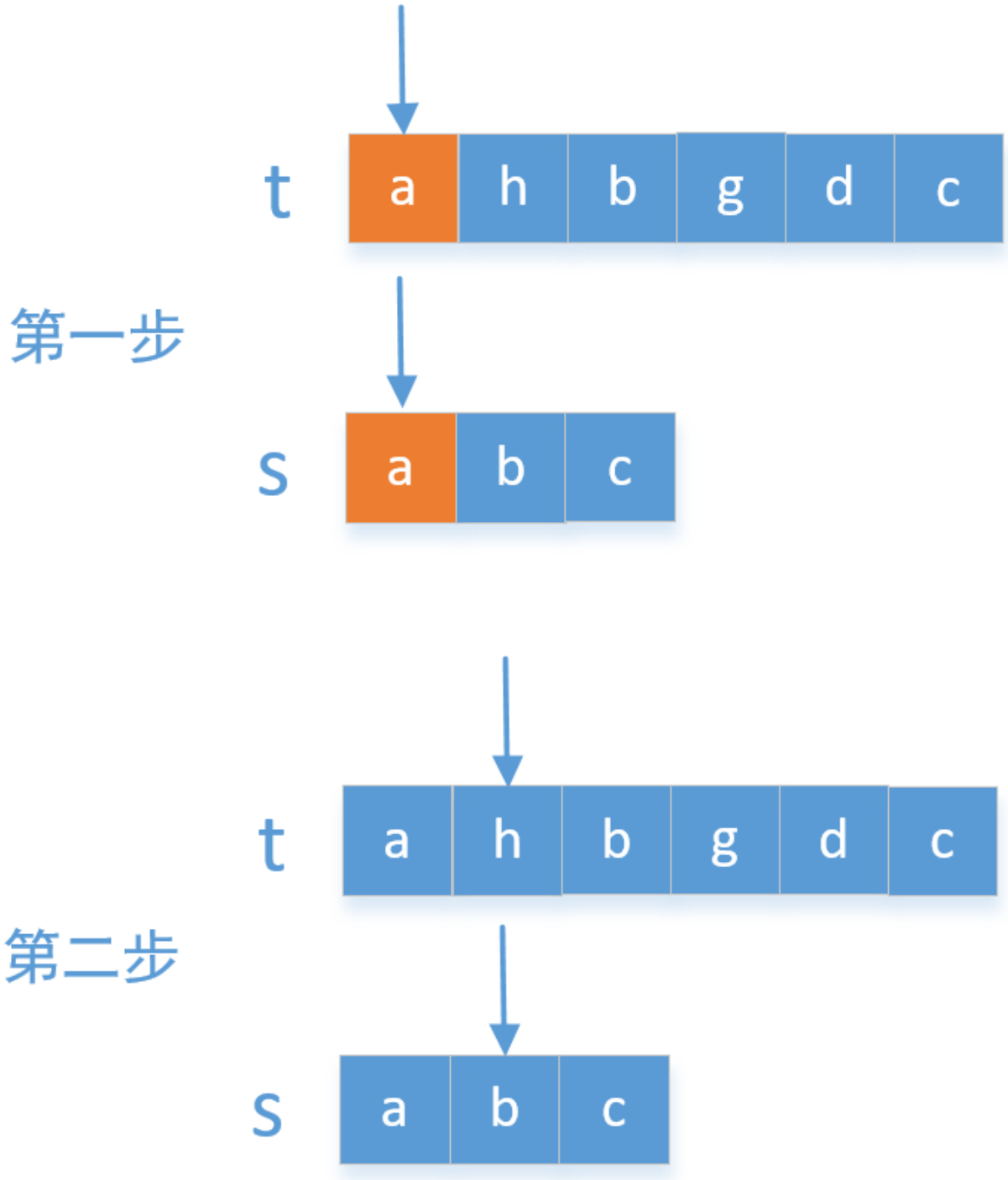
s = "axc", t = "ahbgdc"

返回 false.

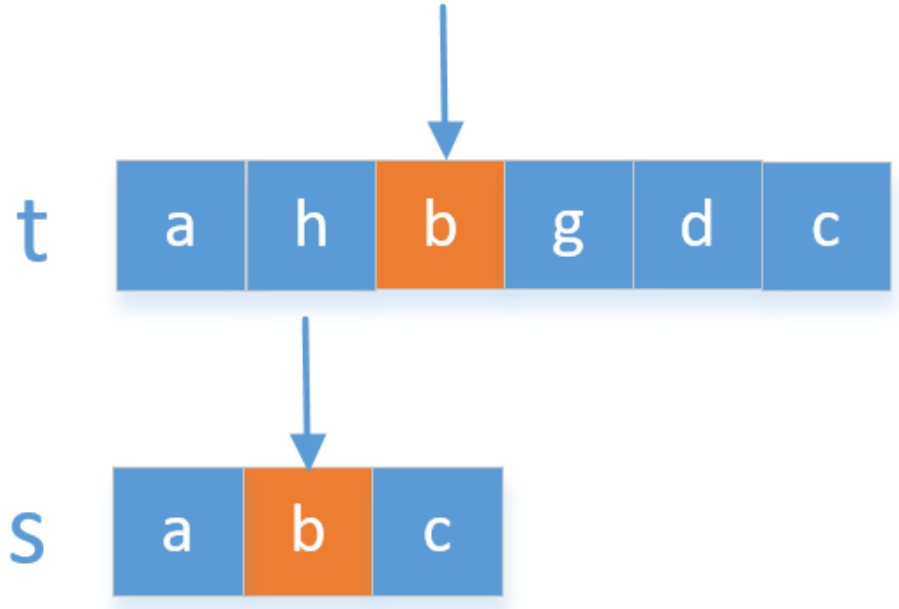
双指针求解

这题让求的是s是否是t的子序列，我们可以使用两个指针，一个指向s的某个字符，一个指向t的某个字符，其中指向t的指针每次都会往右移一位，指向s的指针每次和指向t的

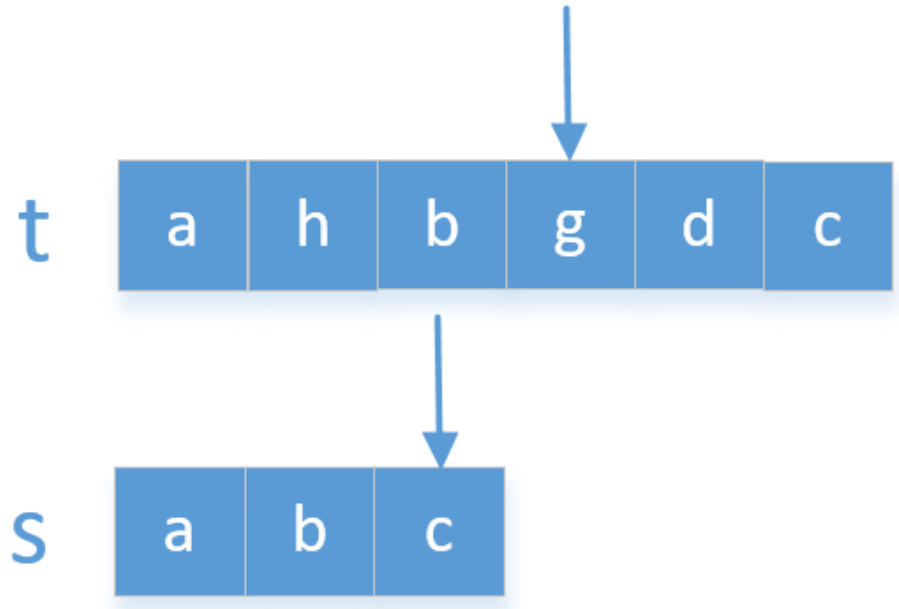
指针所对应的字符相同时才会往右移，否则就不移动，当指向s的指针指向s的末尾的时候，返回true。这里以示例1为例来画个图看一下

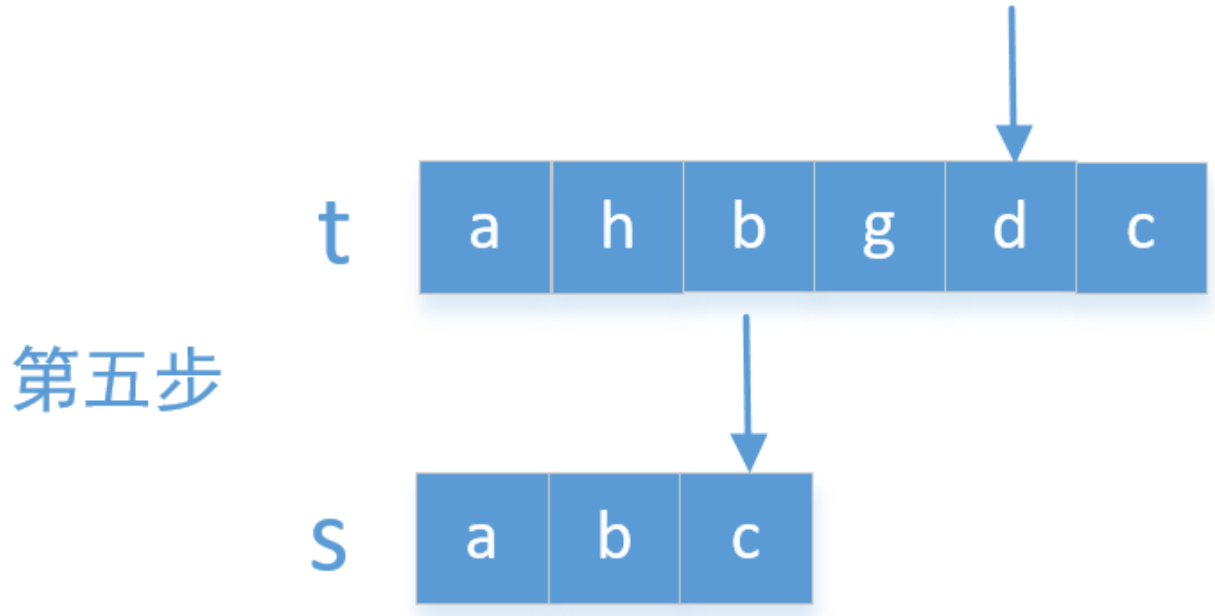


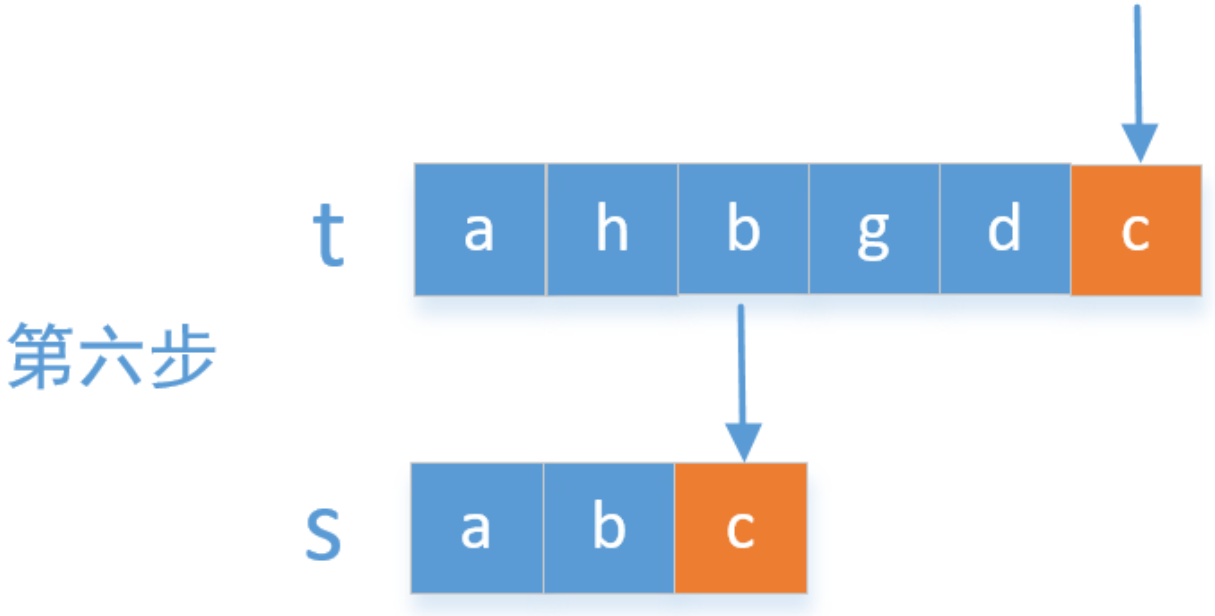
第三步



第四步







原理很简单，我们来直接看下代码

```java
public boolean isSubsequence(String s, String t) {
  if (s.length() == 0)
    return true;
  int indexS = 0, indexT = 0;
  while (indexT < t.length()) {
    if (t.charAt(indexT) == s.charAt(indexS)) {
      //指向s的指针只有在两个字符串相同时才会往右移
      indexS++;
      if (indexS == s.length())
        return true;
    }
    //指向t的指针每次都会往右移一位
    indexT++;
  }
  return false;
}
```

动态规划

我们用dp[i][j]表示字符串t的前j个字符包含s的前i个字符
所以递归公式是

1，s.charAt(i - 1) == t.charAt(j - 1)
dp[i][j] = dp[i - 1][j - 1]

2，s.charAt(i - 1) != t.charAt(j - 1)
dp[i][j] = dp[i][j - 1];

那么边界条件是什么呢，当s为空的时候，我们默认t是包含s的，所以当s为空的时候，返回true。有了递推公式和边界条件，代码就很容易写了，来看下

```
public boolean isSubsequence(String s, String t) {
  if (s.length() == 0)
    return true;
  boolean[][] dp = new boolean[s.length() + 1][t.length() + 1];
  //边界条件
  for (int i = 0; i < t.length(); i++) {
    dp[0][i] = true;
  }
  for (int i = 1; i <= s.length(); i++) {
    for (int j = 1; j <= t.length(); j++) {
      //递推公式
      if (s.charAt(i - 1) == t.charAt(j - 1)) {
        dp[i][j] = dp[i - 1][j - 1];
      } else {
        dp[i][j] = dp[i][j - 1];
      }
    }
  }
  return dp[s.length()][t.length()];
}

```

逐个查找

如果熟悉java语言的都知道，在java中String类有这样一个方法
public int indexOf(int ch, int fromIndex)
他表示的是在字符串中是否存在一个字符ch，并且是从字符串的下标fromIndex开始查找的。我们要做的是在t字符串中查找s中的每一个字符，如果没查到，直接返回false。如果查到，就从t的下一个位置继续开始查

```
public boolean isSubsequence(String s, String t) {
  int index = -1;
  for (char c : s.toCharArray()) {
    //index表示上一次查找的位置(第一次查找的时候为-1)，
    // 所以这里要从t的下标(index+1)开始查找
    index = t.indexOf(c, index + 1);
    //没找到，返回false
    if (index == -1)
      return false;
  }
  return true;
}
```

参照最长公共子序列

看到这道题我们还可以参照之前写的370，最长公共子串和子序列，我们只要求出s和t的最长公共子序列的长度即可，如果长度等于s的长度，说明s就是t的子序列

```java
public boolean isSubsequence(String s, String t) {
    int[] dp = new int[t.length() + 1];
    int last = 0;
    for (int i = 1; i <= s.length(); i++) {
        for (int j = 1; j <= t.length(); j++) {
            int temp = dp[j];
            if (s.charAt(i - 1) == t.charAt(j - 1))
                dp[j] = last + 1;
            else
                dp[j] = Math.max(dp[j], dp[j - 1]);
            last = temp;
        }
    }
    return dp[t.length()] == s.length();
}
```

总结

这题比较简单，但解法比较多，最容易想到的估计就是双指针了。

往期推荐

- 398，双指针求无重复字符的最长子串
- 397，双指针求接雨水问题
- 396，双指针求盛最多水的容器
- 390，长度最小的子数组

408，剑指 Offer-替换空格

原创 山大王wld 数据结构和算法 7月21日

收录于话题
#剑指offer

27个 >



Every strike brings me closer to the next home run.

每一次挥棒落空都让我更接近下一个全垒打。

问题描述

请实现一个函数，把字符串 s 中的每个空格替换成"%20"。

示例 1：

> 输入：s = "We are happy."
> 输出："We%20are%20happy."

限制：

0 <= s 的长度 <= 10000

先把字符串转换为单个字符

这里让求的是把字符串中的空格替换成%20，其中一种实现方式就是申请一个临时数组，然后再遍历这个字符串的每个字符，如果不是空格就把遍历的字符添加到临时数组中，如果是空格就添加3个字符'%'，'2'，'0'分别到临时数组中，最后再把临时数组转化为字符串即可。

```
public String replaceSpace(String s) {
    int length = s.length();
    char[] array = new char[length * 3];
    int index = 0;
    for (int i = 0; i < length; i++) {
        char c = s.charAt(i);
        if (c == ' ') {
            array[index++] = '%';
            array[index++] = '2';
            array[index++] = '0';
        } else {
            array[index++] = c;
        }
    }
    String newStr = new String(array, 0, index);
    return newStr;
}
```

使用StringBuilder

还有一种方式和上面差不多，就是把字符串中的每个字符一个个添加到StringBuilder中，如果遇到空格就把他换成%20。

```
public String replaceSpace(String s) {
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) == ' ')
            stringBuilder.append("%20");
        else
            stringBuilder.append(s.charAt(i));
    }
    return stringBuilder.toString();
}
```

如果还想要更简单的，直接调用API，一行代码搞定

```
public String replaceSpace(String s) {
    return s.replace(" ", "%20");
}
```

总结

这题应该是非常简单的一道题了，没什么好说的。

406，剑指 Offer-二维数组中的查找

原创 山大王wld 数据结构和算法 7月20日

收录于话题
#剑指offer

27个 >



Having dreams is what makes life tolerable.

梦想使生活得以忍受。

问题描述

在一个 n * m 的二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

示例:

现有矩阵 matrix 如下：

```
[
  [1,  4,  7, 11, 15],
  [2,  5,  8, 12, 19],
  [3,  6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]
```

给定 target = 5，返回 true。

给定 target = 20，返回 false。

限制：

- 0 <= n <= 1000
- 0 <= m <= 1000

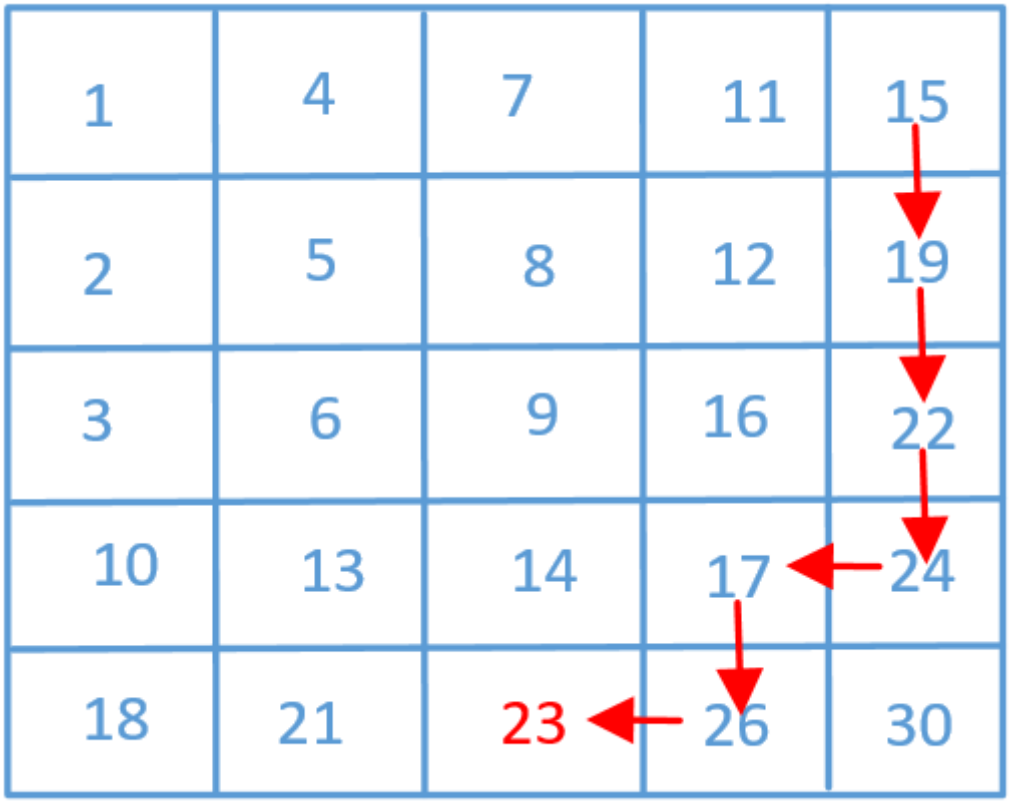
暴力求解

当然最容易想到的是暴力求解，就是一个个查找，如果找到就返回true，没找到就返回false，代码很简单，没什么可说的。

```
public boolean findNumberIn2DArray(int[][] matrix, int target) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
        return false;
    }
    int rows = matrix.length, columns = matrix[0].length;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            if (matrix[i][j] == target) {
                return true;
            }
        }
    }
    return false;
}
```

线性查找

题中说了每行都是递增的，每列也是递增的。所以我们查找的时候可以利用这个特性，如果我们从左上角开始找，当目标值target大于当前值的时候，我们需要往更大的找，但这个时候无论往右找还是往下找都是比当前值大，所以我们无法确定该往哪个方向找。同理右下角也一样，所以我们只能从右上角或者左下角开始找。我们就用上面的数据当target等于23的时候从右上角开始找，来画个图看一下



从右上角开始找有个方便的地方就是他左边的都是比他小的，他下边的都是比他大的，如果target大于当前值我们就往下边找，如果target小于当前值我们就往左边找，来看下代码。

```
public boolean findNumberIn2DArray(int[][] matrix, int target) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
        return false;
    }
    int rows = matrix.length, col = matrix[0].length;
    //从第0行col - 1列开始查找，也就是第1行最后一列的那个数字开始
    int row = 0;
    int column = col - 1;
    while (row < rows && column >= 0) {
        //num表示当前值
        int num = matrix[row][column];
        if (num == target) {
            //如果找到直接返回
            return true;
        } else if (num > target) {
            //到前面查找
            column--;
        } else {
            //到下面查找
            row++;
        }
    }
    return false;
}
```

当然从左下角查找也是可以的，因为左下角右边的值是比他大的，上边的值是比他小的，也能区分，代码和上面差不多，来看下

```
public boolean findNumberIn2DArray(int[][] matrix, int target) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
        return false;
    }
    int rows = matrix.length, col = matrix[0].length;
```

```
    int row = rows - 1;
    int column = 0;
    while (row >= 0 && column < col) {
      int num = matrix[row][column];
      if (num == target) {
        //如果找到直接返回
        return true;
      } else if (num > target) {
        //往上面找
        row--;
      } else {
        //往右边找
        column++;
      }
    }
    return false;
  }
```

总结

这题说了行和列都是递增的，所以我们可以根据这个特性来查找，只有从右上角或左下角查找的时候才能确定下一步该往哪个方向走，从左上角或右下角都没法确定。

往期推荐

- 404，剑指 Offer-数组中重复的数字
- 398，双指针求无重复字符的最长子串
- 397，双指针求接雨水问题
- 396，双指针求盛最多水的容器

长按上图，识别图中二维码之后即可关注。

如果喜欢这篇文章就点个"赞"吧

405，换酒问题

原创 山大王wld 数据结构和算法 7月20日

收录于话题
#算法图文分析

95个 >



It's not the altitude, it's the attitude.

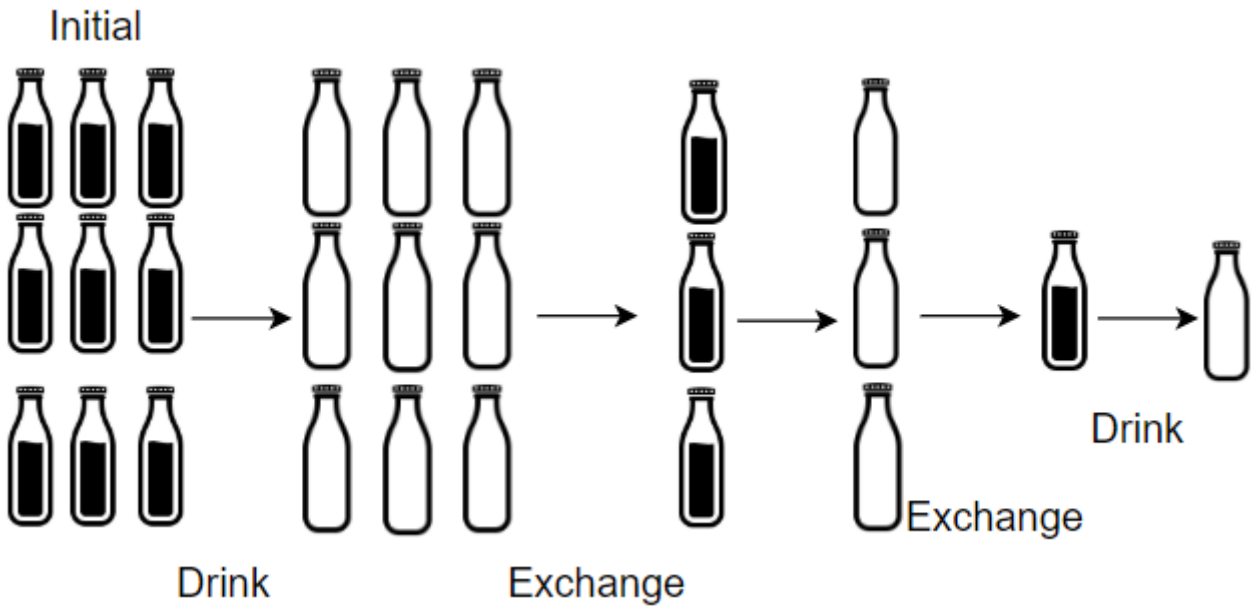
决定一切的不是高度而是态度。

问题描述

小区便利店正在促销，用 numExchange 个空酒瓶可以兑换一瓶新酒。你购入了 numBottles 瓶酒。

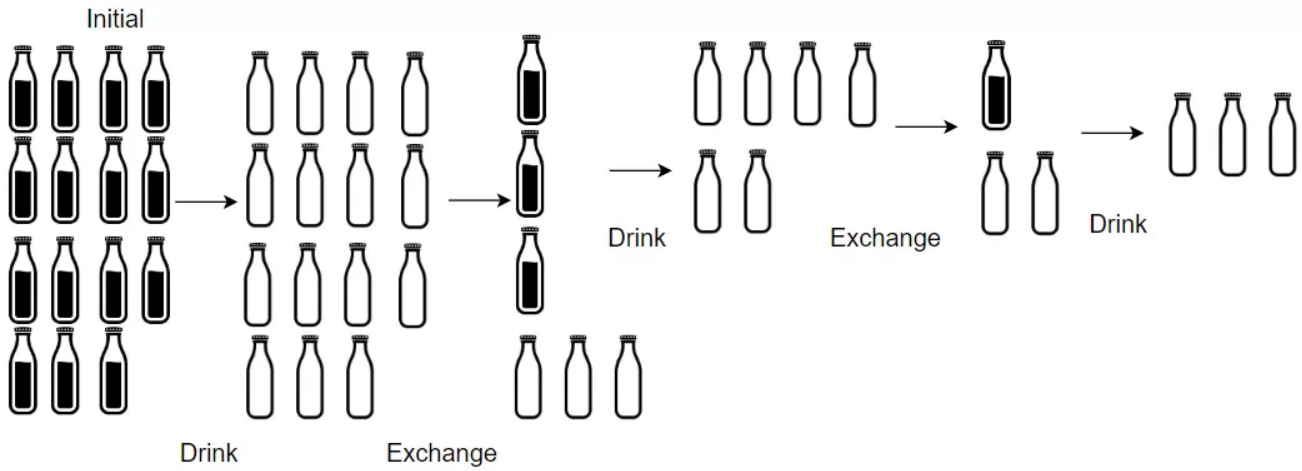
如果喝掉了酒瓶中的酒，那么酒瓶就会变成空的。
请你计算最多能喝到多少瓶酒。

示例 1：



输入：numBottles = 9, numExchange = 3
输出：13
解释：你可以用 3 个空酒瓶兑换 1 瓶酒。
所以最多能喝到 9 + 3 + 1 = 13 瓶酒。

示例 2：



输入：numBottles = 15, numExchange = 4
输出：19
解释：你可以用 4 个空酒瓶兑换 1 瓶酒。
所以最多能喝到 15 + 3 + 1 = 19 瓶酒。

示例 3：

输入：numBottles = 5, numExchange = 5

> 输出：6

示例 4：

> 输入：numBottles = 2，numExchange = 3
> 输出：2

提示：

- 1 <= numBottles <= 100
- 2 <= numExchange <= 100

问题分析

类似这种题估计很多人都见过，一般是作为一道脑筋急转弯的题出现，我们就以示例4为例来说下，如果3个空瓶子可以换一瓶酒的话，我们只有两瓶酒，喝完之后最多只有两个空瓶子，所以是换不成一瓶酒的。但如果我们找别人借一个空瓶子的话，正好是3个空瓶子，可以换一瓶酒，喝完之后再把这个空瓶子还给别人，所以我们可以喝3瓶酒，作为一道脑筋急转弯题这是正确的，但这道题答案返回的是2，也就是说我们不能找别人借。

我们假设一个空瓶子的价值是1，那么一瓶酒（不包含瓶子）的价值就是numExchange-1。我们最初所拥有的总价值就是numBottles* numExchange，因为不能找别人借，所以最后我们至少会有一个空瓶子，也就是说最后我们所能喝到的最大价值<=numBottles* numExchange-1，我们能喝的到酒的数量就很容易算出来了。

```
public int numWaterBottles(int numBottles, int numExchange) {
  return (numBottles * numExchange - 1) / (numExchange - 1);
}
```

另一种解法

每次喝完之后如果空瓶子数量大于等于numExchange，我们就把他兑换成酒，这个时候我们所拥有的瓶子数量就是我们兑换成酒的瓶子数量加上没有兑换成酒的瓶子数量，如果还大于numExchange就继续循环……，直到不能兑换为止。

```
public int numWaterBottles(int numBottles, int numExchange) {
  int total = numBottles;
  while (numBottles >= numExchange) {
    //change表示的是兑换成酒的数量
    int change = numBottles / numExchange;
    //total再加上兑换的酒
    total += change;
    //瓶子数量变为兑换成酒的数量加上没有兑换成酒的数量
    numBottles = change + numBottles % numExchange;
  }
```

```
11      return total;
12  }
```

总结

这道题和脑筋急转弯题型还是有点区别的，如果空瓶子数量不够的情况下我们是不能借的，这两种解法都比较简单。

往期推荐

- 403，验证二叉搜索树
- 364，位1的个数系列（一）
- 385，位1的个数系列（二）
- 402，位1的个数系列（三）

404，剑指 Offer-数组中重复的数字

原创 山大王wld 数据结构和算法 7月18日

收录于话题
#剑指offer

27个 >



All over the place was six pence, but he looked up at the moon.

满地都是六便士，他却抬头看见了月亮。

问题描述

找出数组中重复的数字。

在一个长度为 n 的数组 nums 里的所有数字都在 0～n-1 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。

示例 1：

> 输入：
> [2, 3, 1, 0, 2, 5, 3]
> 输出：2 或 3

限制：

2 <= n <= 100000

使用集合set

题中说的是让我们找出重复的数字，重复的数字可能会有多个，我们随便返回一个即可。最简单的方式就是把数组中的元素一个个加入到集合set中，加入的时候如果set集合中有这个数据了，就说明有了重复的，我们直接返回这个值，我们看下代码

```
public int findRepeatNumber(int[] nums) {
    Set<Integer> set = new HashSet<>();
    for (int num : nums) {
        //如果添加的时候返回false表示有重复的数据，
        //我们直接返回
        if (!set.add(num))
            return num;
    }
    return -1;
}
```

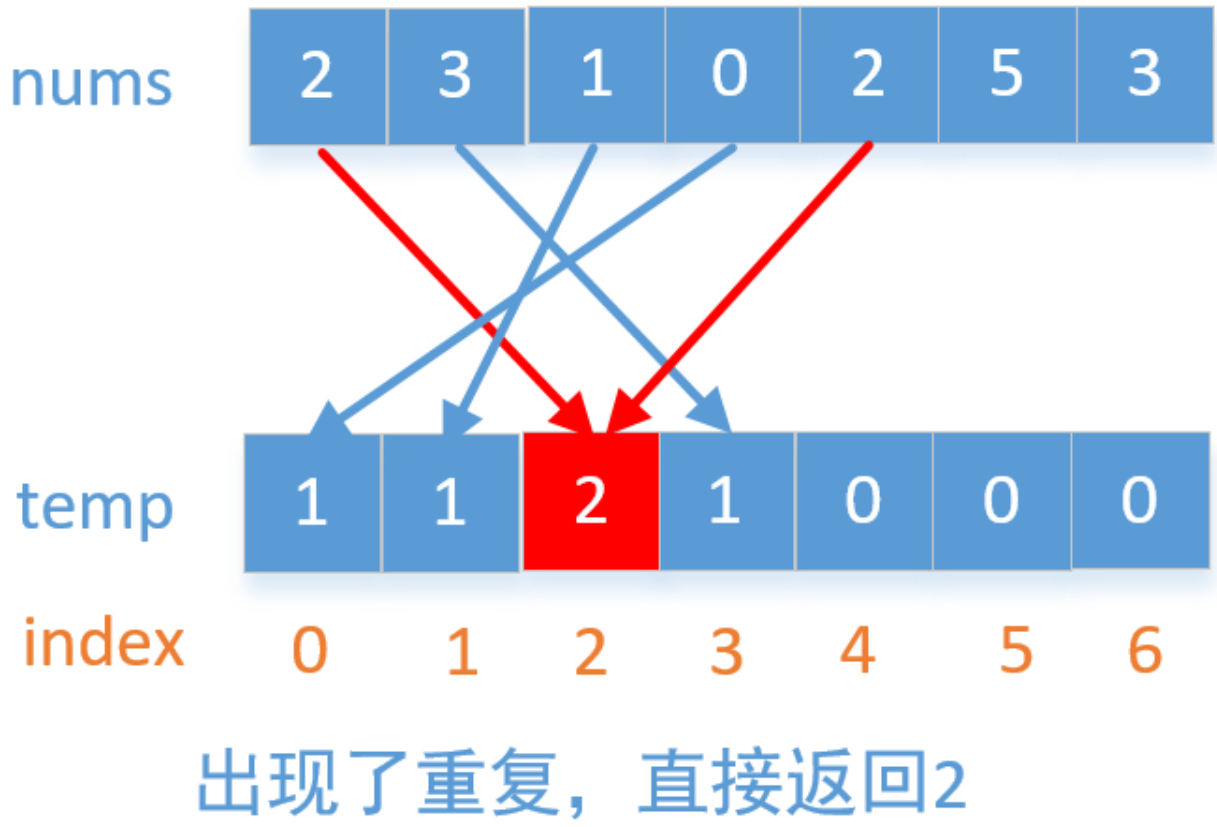
先排序再查找

其实还有一种简单的方式就是先排序再查找，如果有重复的数据，排序之后他们肯定是挨着的。排序之后我们只需要前后两两判断是否有相同的，如果有则直接返回

```
public int findRepeatNumber(int[] nums) {
    //先排序
    Arrays.sort(nums);
    //再前后两两判断是否有相同的
    for (int i = 1; i < nums.length; i++) {
        if (nums[i] == nums[i - 1])
            return nums[i];
    }
    return -1;
}
```

使用临时数组

上面两种解法非常简单，但这道题有个很明显的特点，就是数字的大小在0~n-1之间，所以使用上面两种方式肯定不是最好的选择。这里我们可以申请一个临时数组temp，因为nums元素中的每个元素的大小都在0~n-1之间，所以我们可以把nums中元素的值和临时数组temp建立映射关系，就是nums中元素的值是几，我们就把temp中对应的位置值加1，当temp某个位置的值大于1的时候，就表示出现了重复，我们直接返回即可



上面图中我们看到红色的部分，2出现了两次，我们可以直接返回

```
public int findRepeatNumber(int[] nums) {
    int length = nums.length;
    //申请一个临时数组
    int[] temp = new int[length];
    for (int i = 0; i < length; i++) {
        //每个数字出现的次数加1
        temp[nums[i]]++;
        //如果大于1，说明出现了重复的，直接返回
        if (temp[nums[i]] > 1)
            return nums[i];
    }
    return -1;
}
```

元素放到指定的位置

我们还可以不使用临时数组，我们在遍历的时候把数组nums中的值放到对应的位置上，比如某个元素是5，我们就把他放到nums[5]中，每次放入的时候查看一下这个位置是否放入了正确的值，如果已经放入了正确的值，就说明重复了，我们直接返回即可。

```
public int findRepeatNumber(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        //位置正确，先不用管
        if (i == nums[i])
            continue;
        //出现了重复，直接返回
        if (nums[i] == nums[nums[i]])
            return nums[i];
        //交换
        int temp = nums[nums[i]];
        nums[nums[i]] = nums[i];
        nums[i] = temp;
```

```
        //这里的i--是为了抵消掉上面的i++,
        //交换之后需要原地再比较
        i--;
    }
    return -1;
}
```

总结

这题是剑指offer上的一道题，比较简单，解法也比较多，我们从题中给出的数字大小的范围可知最后两种解法应该是最适合这道题的。

往期推荐

- 398，双指针求无重复字符的最长子串
- 397，双指针求接雨水问题
- 396，双指针求盛最多水的容器
- 384，整数反转

393，括号生成

原创 山大王wld 数据结构和算法 7月3日

收录于话题
#算法图文分析

96个 >



Nothing that has meaning is easy. Easy doesn't enter into grown-up life.

凡是有意义的事都不会容易，成年人的生活里没有容易二字。

问题描述

数字n代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且有效的括号组合。

示例：

```
输入：n = 3
输出：[
       "((()))",
       "(()())",
       "(())()",
       "()(())",
       "()()()"
     ]
```

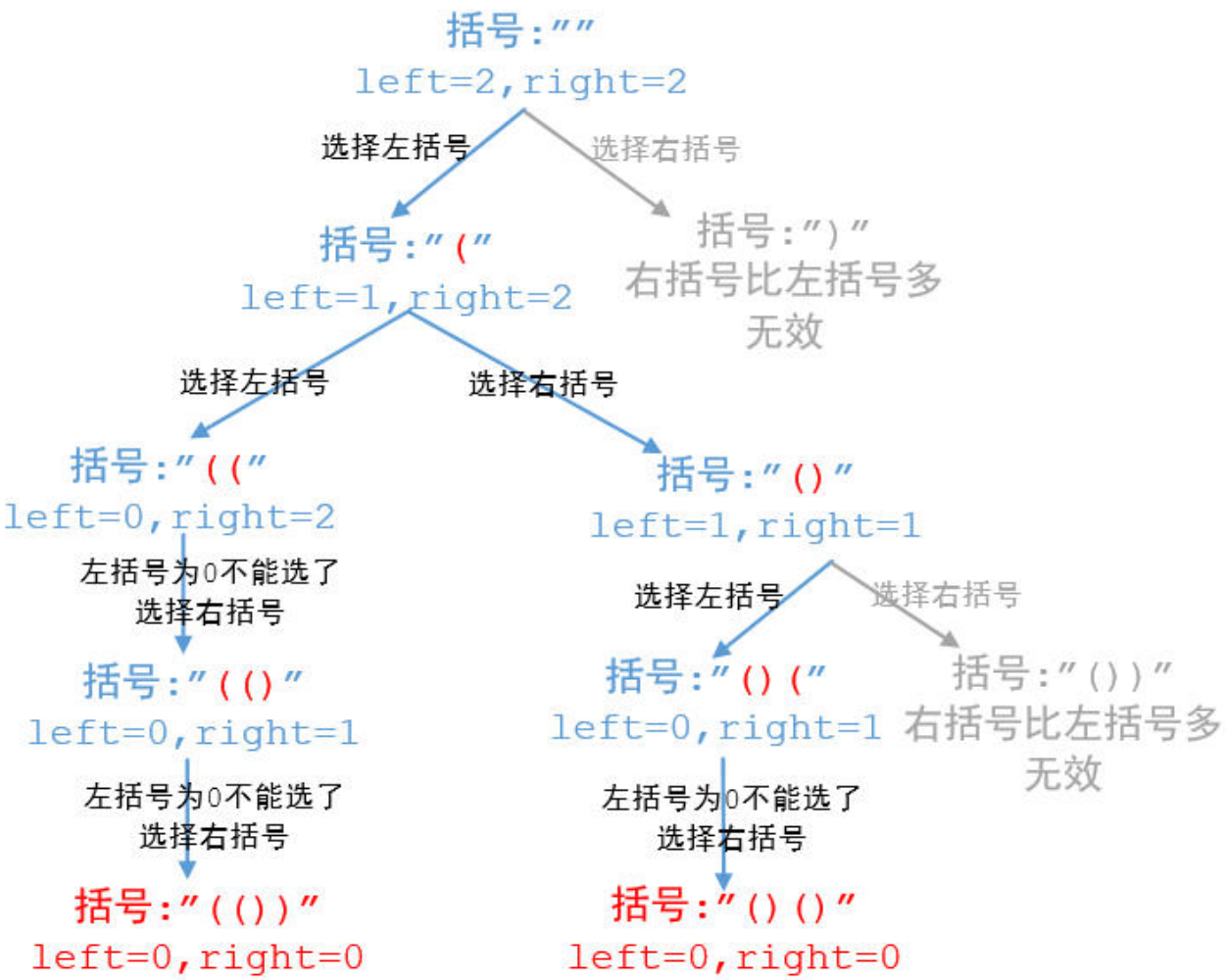
问题分析

通过观察我们可以发现，生成的任何括号组合中都有两个规律：

1，括号组合中左括号的数量等于右括号的数量

2，括号组合中任何位置左括号的数量都是大于等于右括号的数量

第一条很容易理解，我们来看第二条，比如有效括号"(())()"，在任何一个位置右括号的数量都不大于左括号的数量，所以他是有效的。如果像这样"())()"第3个位置的是右括号，那么他前面只有一个左括号，而他和他前面的右括号有2个，所以无论如何都不能组合成有效的括号。搞懂了上面的原理，我们就以n等于2为例来画个图看一下



看到上面的图我们很容易想到二叉树的前序遍历，可以看下之前写的373，数据结构-6，树，所以这里我们可以参考一下，二叉树的前序遍历代码如下

```
public static void preOrder(TreeNode tree) {
    if (tree == null)
        return;
    System.out.printf(tree.val + "");
    preOrder(tree.left);
    preOrder(tree.right);
```

```
}
```

使用的是递归的方式，有一个终止条件，然后后面是两个递归的调用，所以这题的参考代码如下

```
public List<String> generateParenthesis(int n) {
    List<String> res = new ArrayList<>();
    dfs(res, n, n, "");
    return res;
}

private void dfs(List<String> res, int left, int right, String curStr) {
    /**
     * 这里有终止条件
     *  return
     */
    //选择左括号
    dfs(res, left - 1, right, curStr + "(");
    // 选择右括号
    dfs(res, left, right - 1, curStr + ")");
}
```

其中left是左括号剩余的数量，right是右括号剩余的数量。代码的大致轮廓已经出来了，关键是终止条件。根据上面的分析，我们知道如果左括号和右括号剩余可选数量都等于0的时候，说明找到了有效的括号组合。如果左括号剩余可选数量为0的时候，我们不能再选择左括号了，但可以选择右括号。如果左括号剩余数量大于右括号剩余数量说明之前选择的是无效的。所以终止条件就呼之欲出了，最终代码如下

```
public List<String> generateParenthesis(int n) {
    List<String> res = new ArrayList<>();
    dfs(res, n, n, "");
    return res;
}

private void dfs(List<String> res, int left, int right, String curStr) {
    if (left == 0 && right == 0) { // 左右括号都不剩余了，说明找到了有效的括号
        res.add(curStr);
        return;
    }
    //左括号只有剩余的时候才可以选，如果左括号的数量已经选完了，是不能再选左括号了。
    //如果选完了左括号我们是还可以选择右括号的。
    if (left < 0)
        return;
    // 如果右括号剩余数量小于左括号剩余的数量，说明之前选择的无效
    if (right < left)
        return;
    //选择左括号
    dfs(res, left - 1, right, curStr + "(");
    //选择右括号
    dfs(res, left, right - 1, curStr + ")");
}
```

动态规划

我们用dp[i]表示的是n等于i的时候生成的有效括号组合，那么递推公式就是

dp[i]="("+dp[m]+")"+dp[k]

其中m+k=i-1

因为他再加上我们添加的一对括号正好是i，（其中m是从0到i-1）所以这里我们需要枚举m的所有值。主要代码如下

```
for (int m = 0; m < i; m++) {
    int k = i - 1 - m;
    List<String> str1 = dp[m];
    List<String> str2 = dp[k];
    for (String s1 : str1) {
        for (String s2 : str2) {
            cur.add("(" + s1 + ")" + s2);
        }
    }
}
```

这题的边界条件是dp[0]=""，因为0的时候是没有括号的。所以完整代码如下

```
public static List<String> generateParenthesis(int n) {
  List<String>[] dp = new List[n + 1];
  List<String> dp0 = new ArrayList<>();
  dp0.add("");
  dp[0] = dp0;
  for (int i = 1; i <= n; i++) {
    List<String> cur = new ArrayList<>();
    for (int m = 0; m < i; m++) {
      int k = i - 1 - m;
      List<String> str1 = dp[m];
      List<String> str2 = dp[k];
      for (String s1 : str1) {
        for (String s2 : str2) {
          cur.add("(" + s1 + ")" + s2);
        }
      }
    }
    dp[i] = cur;
  }
  return dp[n];
}
```

我们就用n等于3来测试一下打印的结果

```
public static void main(String args[]) {
    System.out.println(Arrays.toString(generateParenthesis(3).toArray()));
}
```

运行结果如下

```
[()()(), ()(()), (())(), (()()), ((()))]
```

动态规划改递归

我们看到上面动态规划中核心代码是dp[m]和dp[k]的组合，而dp[m]和dp[k]分别表示的是n等于m和k的时候有效括号的组合，所以如果函数

`List<String> generateParenthesis(int n)`

表示的是n对有效括号的组合，那么

`List<String> generateParenthesis(int m)`

和

`List<String> generateParenthesis(int k)`

分别表示的是m对和k对有效括号的组合，所以上面的核心代码我们可以这样改

```
for (int m = 0; m < n; m++) {
    int k = n - m - 1;
    List<String> first = generateParenthesis(m);
    List<String> second = generateParenthesis(k);
    for (String left : first) {
        for (String right : second) {
            list.add("(" + left + ")" + right);
        }
    }
}
```

所以完整代码如下

```
public static List<String> generateParenthesis(int n) {
  List<String> list = new ArrayList<>();
  if (n == 0) {//边界条件的判断
    list.add("");
    return list;
  }
  for (int m = 0; m < n; m++) {
    int k = n - m - 1;
    List<String> first = generateParenthesis(m);
    List<String> second = generateParenthesis(k);
    for (String left : first) {
      for (String right : second) {
        list.add("(" + left + ")" + right);
      }
    }
  }
  return list;
}
```

总结

这题可能最容易想到的是暴力求解，就是生成所有的组合，然后再判断这些组合哪些是有效的，但这种效率很差，所以这里没写。上面第一种解法很好的利用了有效括号的特

性，无效括号直接舍去，达到剪枝的目的。下面两种解法原理都是一样的，只不过一个使用的是动态规划，一个使用的是递归，都是根据已经生成的长度为i-1的有效括号，然后推出长度为i的有效括号。

往期推荐

- 392，检查数组对是否可以被 k 整除
- 391，回溯算法求组合问题
- 376，动态规划之编辑距离
- 371，背包问题系列之-基础背包问题

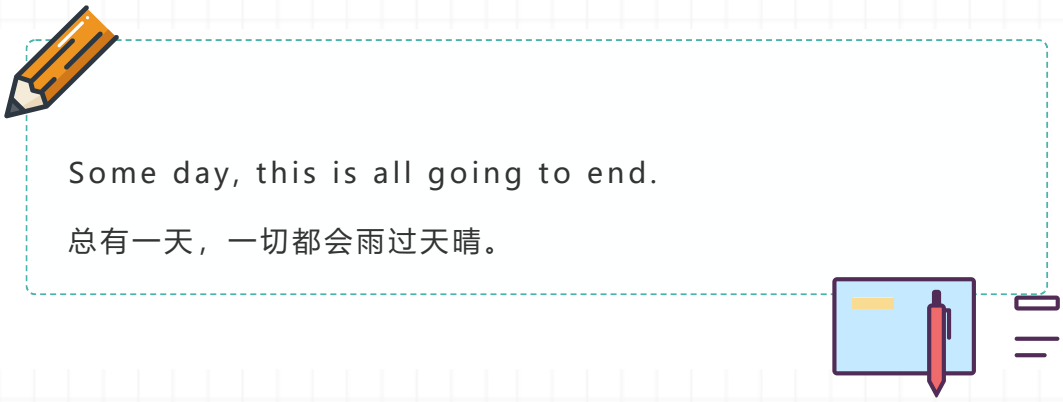
392，检查数组对是否可以被 k 整除

原创 山大王wld 数据结构和算法 7月2日

收录于话题
#算法图文分析

96个 >





问题描述

给你一个整数数组 arr 和一个整数 k ，其中数组长度是偶数，值为 n 。
现在需要把数组恰好分成 n / 2 对，以使每对数字的和都能够被 k 整除。
如果存在这样的分法，请返回 true ；否则，返回 false 。

示例 1：

```
输入：arr = [1,2,3,4,5,10,6,7,8,9], k = 5
输出：true
解释：划分后的数字对为 (1,9),(2,8),(3,7),(4,6) 以及 (5,10) 。
```

示例 2：

```
输入：arr = [1,2,3,4,5,6], k = 7
输出：true
解释：划分后的数字对为 (1,6),(2,5) 以及 (3,4) 。
```

示例 3：

```
输入：arr = [1,2,3,4,5,6], k = 10
输出：false
解释：无法在将数组中的数字分为三对的同时满足每对数字和能够被10整除的条件。
```

示例 4：

```
输入：arr = [-10,10], k = 2
输出：true
```

示例 5：

```
输入：arr = [-1,1,-2,2,-3,3,-4,4], k = 3
输出：true
```

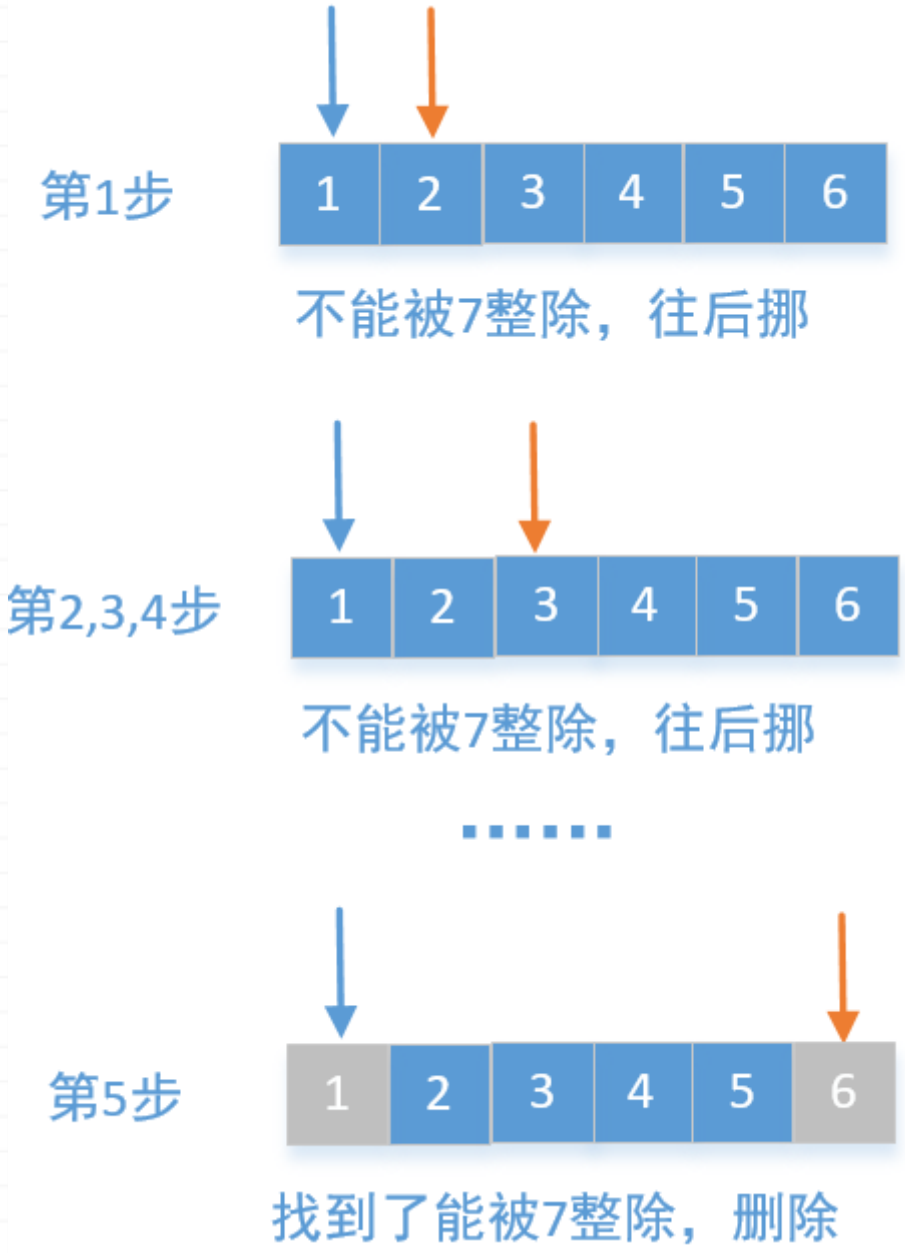
问题分析

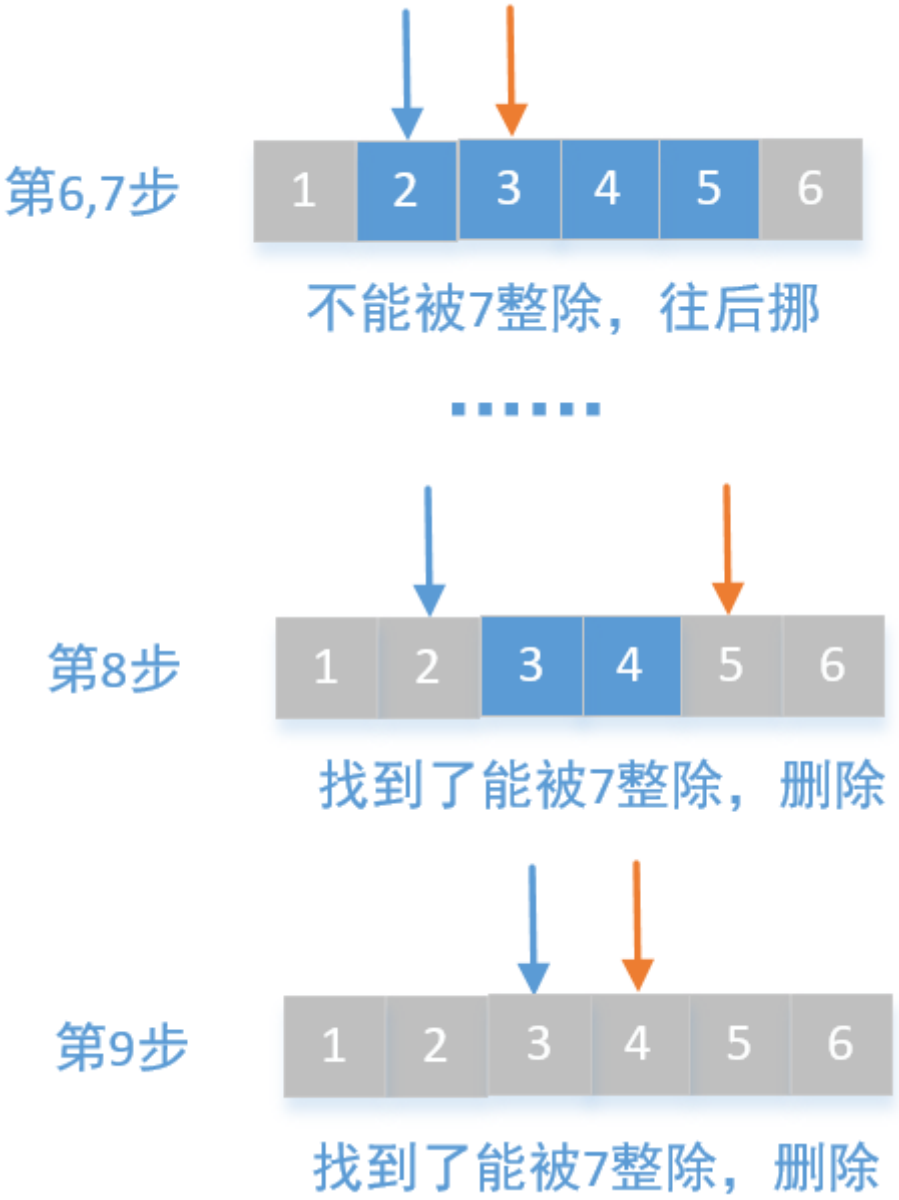
这道题问的实际上是把数组中的元素每两个分为一组，总共分为n/2组，然后确保每组都能被k整除，这样结果才会返回true，否则返回false。

其实这里面有个数学问题，假如有两组数据（a，b）和(c，d)他们都能被k整除，也就是说(a+b)%k=0，并且(c+d)%k=0；如果(a+c)%k=0，那么(b+d)%k=0肯定也是成立的。（这里的所有字母都是整数）

我们可以证明一下
假如a+b=m*k，并且c+d=n*k。
如果a+c=t*k;
那么b+d=(m*k-a)+(n*k-c)
=(m+n)*k-(a+c)
=(m+n)*k-t*k
=(m+n-t)*k（这里能被k整除）
所以我们可以证明b+d也一定是可以被k整除的。
举个例子，比如(3，5)，(7，9)都能被4整除，如果（3+9）能被4整除，那么（5+7）也一定能被4整除。

有了上面的证明我们再来看这道题，所以我们很容易想到暴力求解，我们使用两个指针，一个指针指向一个固定的元素，另一个指针从这个固定的元素下一个开始查找，如果找到就把这两个元素标记为删除，然后再继续查找……。如果没找到就直接返回false，我们以示例2为例来画个图看一下





最后我们再来看下代码部分

```java
public boolean canArrange(int[] arr, int k) {
    int length = arr.length;
    boolean[] visit = new boolean[arr.length];
    for (int i = 0; i < length - 1; i++) {
        if (visit[i])//数字被访问过了，就不能再用了
            continue;
        for (int j = i + 1; j < length; j++) {
            if (visit[j])//数字被访问过了，就不能再用了
                continue;
            if ((arr[i] + arr[j]) % k == 0) {
                //如果被找到了，我们就把他标记为已使用，
                //下次就不会再用它了
                visit[i] = visit[j] = true;
                break;
            }
        }
        if (!visit[i])//没找到匹配的直接返回false
            return false;
    }
    return true;
}
```

代码优化

我们来思考这样一个问题，如果a+b能被k整除，那么a和b分别对k求余的结果相加也一定能被k整除，即(a%k+b%k)%k=0。所以我们可以对上面数组中的元素分别对k求余。

即num=num%k，因为数组中可能会有负数，所以求余的结果也可能为负，这里为了计算方便，我们把求余的结果全部转化为非负数，大小在[0，k-1]中，包含0和k-1。所以计算公式是num=（num%k+k)%k

这样我们只需要计算余数相对应位置上的个数是否相等就可以了，举个例子，比如k是5，那么余数中1的个数必须和4的个数一样多，2的个数必须和3的个数一样多，这样才能匹配成功，否则直接返回false。还有一点是0的个数必须是偶数

比如余数中[1,2,1,3,4,1]由于2和3的个数都是1所以能组合成一组，但1的个数和4的个数不一致，所以只有一个能组合成功，另一对组合失败。最后我们再来看下代码

```
1  public boolean canArrange(int[] arr, int k) {
2    int[] mod = new int[k];
3    //统计求余之后各余数的个数
4    for (int num : arr)
5      mod[(num % k + k) % k]++;
6    for (int i = 1; i <= k / 2; ++i)
7      //如果对应的个数不匹配，直接返回false
8      if (mod[i] != mod[k - i])
9        return false;
10   //余数中0的个数必须是偶数
11   return mod[0] % 2 == 0;
12 }
```


往期推荐

- 391，回溯算法求组合问题
- 390，长度最小的子数组
- 381，合并两个有序链表（易）
- 372，二叉树的最近公共祖先

390，长度最小的子数组

原创 山大王wld 数据结构和算法 6月28日

收录于话题
#算法图文分析

96个 >



Success, real success, is being willing to do the things that other people are not.

成功，真正的成功，是愿意做别人不愿意做的事情。

请输入标题

给定一个含有 n 个正整数的数组和一个正整数 s ，找出该数组中满足其和 ≥ s 的长度最小的连续子数组，并返回其长度。如果不存在符合条件的连续子数组，返回 0。

示例:

```
输入: s = 7, nums = [2,3,1,2,4,3]
输出: 2
解释: 子数组 [4,3] 是该条件下的长度最小的连续子数组。
```

暴力求解

首先这题最容易想到的是暴力求解，使用两个for循环，一个for循环固定一个数字比如m，另一个for循环从m的下一个元素开始累加，当和大于等于s的时候终止内层循环，顺便记录下最小长度

```
public int minSubArrayLen(int s, int[] nums) {
  int min = Integer.MAX_VALUE;
  for (int i = 0; i < nums.length; i++) {
    int sum = nums[i];
    if (sum >= s)
      return 1;
    for (int j = i + 1; j < nums.length; j++) {
      sum += nums[j];
      if (sum >= s) {
        min = Math.min(min, j - i + 1);
        break;
      }
    }
  }
  return min == Integer.MAX_VALUE ? 0 : min;
}
```

暴力求解虽然也能解出来，但毕竟效率很差，我们来看下其他的几种解题方法

使用队列（一）

实际上我们也可以把它称作是滑动窗口，这里的队列其实就相当于一个窗口。我们把数组中的元素不停的入队，直到总和大于等于s为止，接着记录下队列中元素的个数，然后再不停的出队，直到队列中元素的和小于s为止（如果不小于s，也要记录下队列中元素的个数，这个个数其实就是不小于s的连续子数组长度，我们要记录最小的即可）。接着再把数组中的元素添加到队列中……重复上面的操作，直到数组中的元素全部使用完为止。

这里以[2,3,1,2,4,3]举例画个图来看下

队列



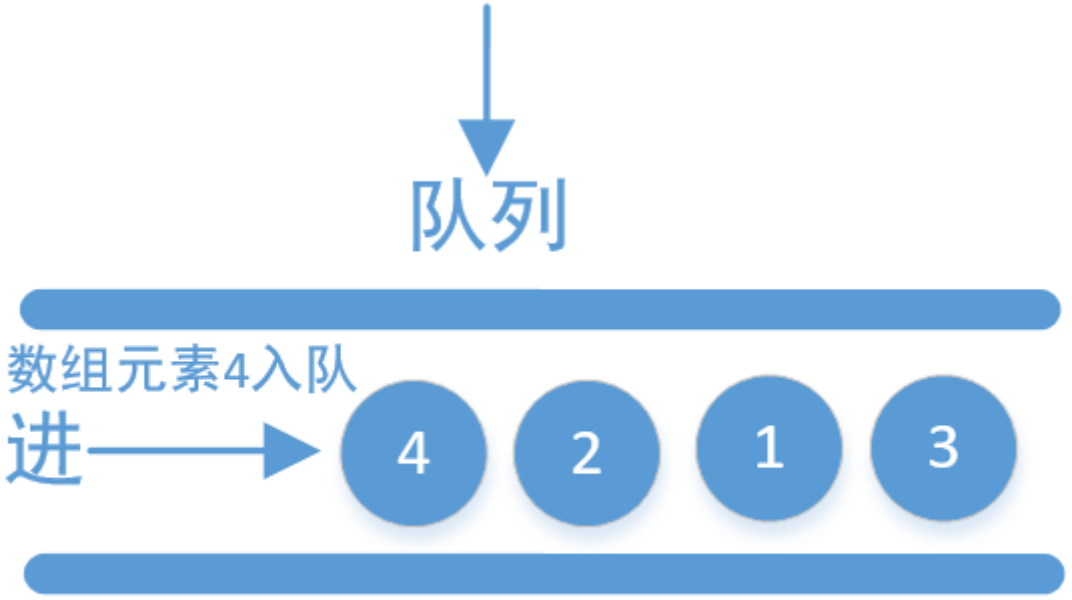
2，3，1，2分别入队，由于队列元素的总和大于7,所以记录下长度4，也就是队列中元素的个数。然后只要队列中元素的和大于等于7，队列的头就不停的出队，顺便记录下队列元素的个数，直到小于7，队列头就不在出队



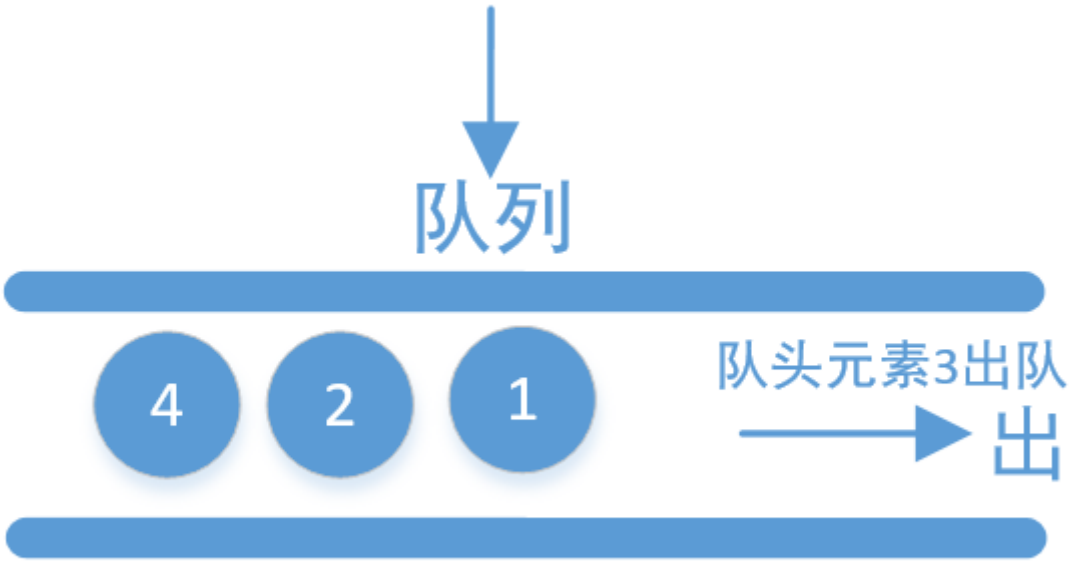
队列



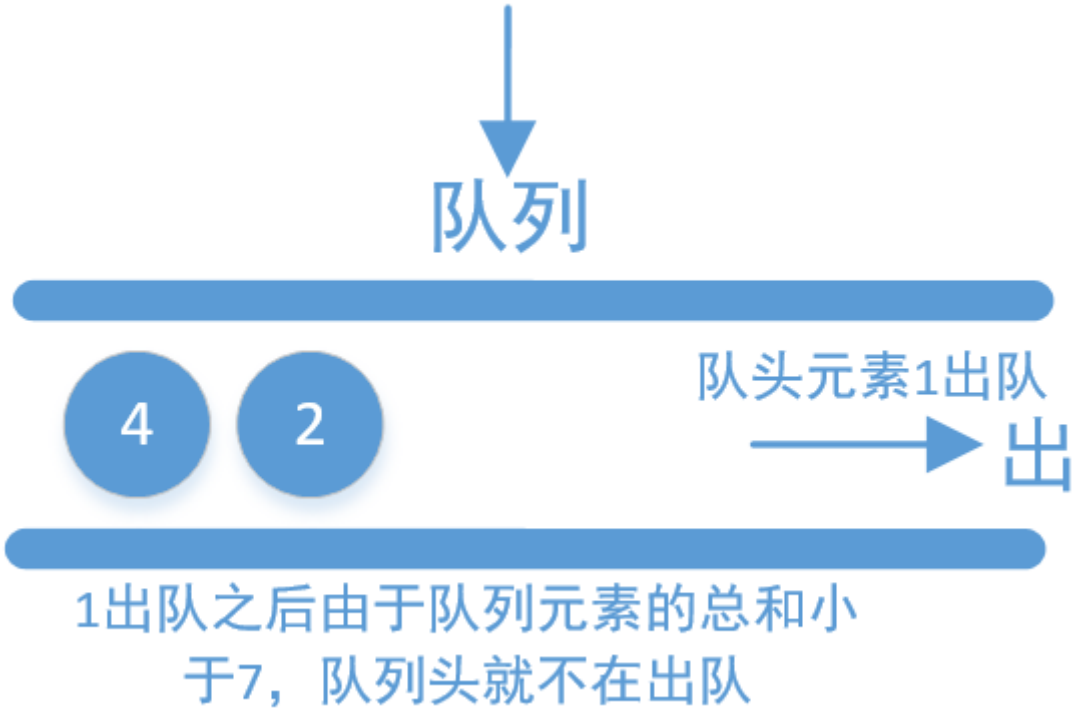
队头元素2出队之后，由于总和是小于7的，所以队头元素3就不在出队



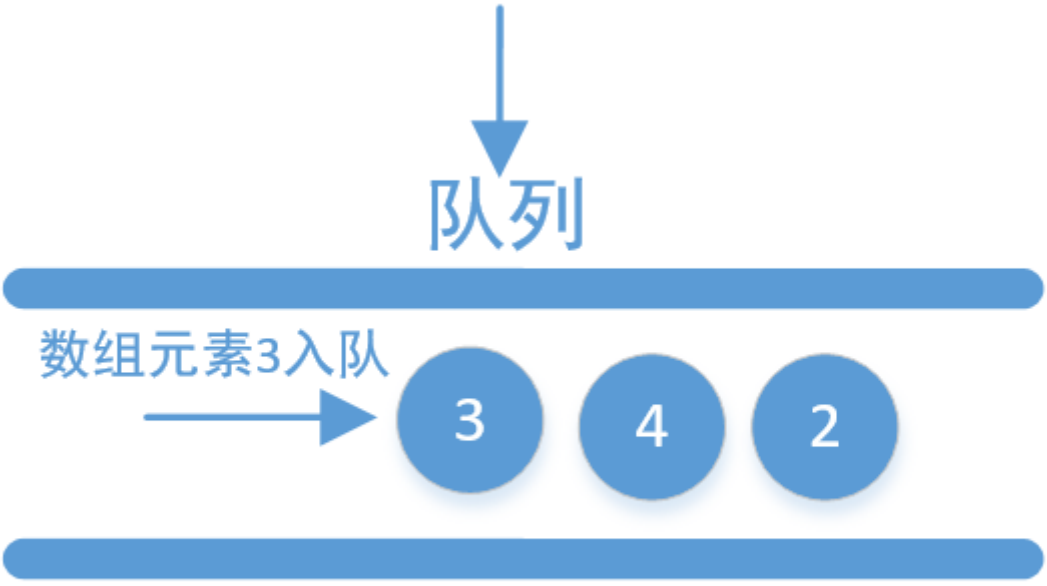
4加入队列之后，由于队列元素的总和是10>7，所以记录下长度4，也就是元素的个数。然后队列头3出队



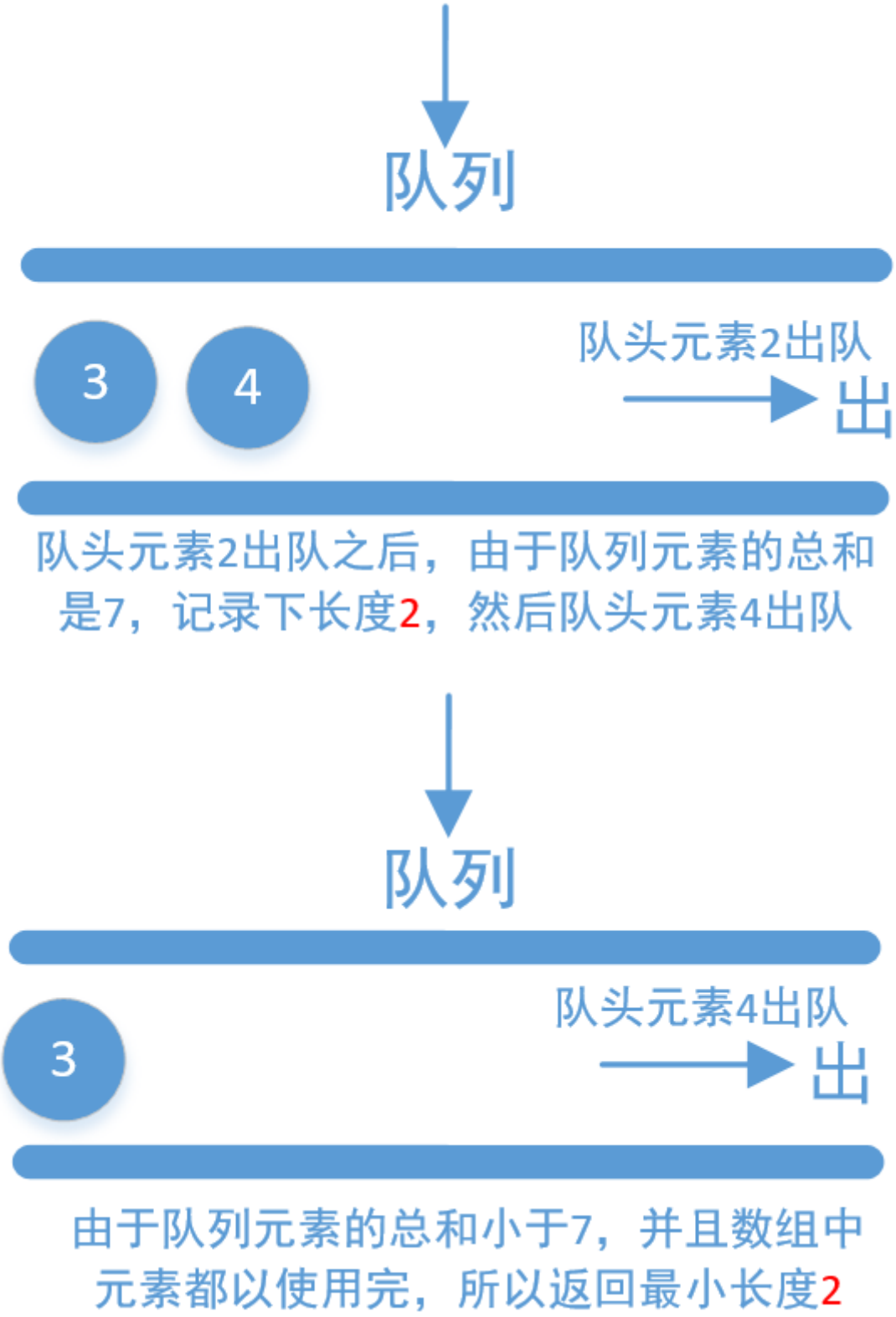
3出队之后和等于7，记录下长度3，队头1继续出队



1出队之后由于队列元素的总和小于7，队列头就不在出队



由于队列元素的总和大于7，记录下长度3，然后队头元素2出队



上面画的图是使用队列，但在代码中我们不直接使用队列，我们使用两个指针，一个指向队头一个指向队尾，和使用队列类似，我们来看下代码

```
1   public int minSubArrayLen(int s, int[] nums) {
2     int lo = 0, hi = 0, sum = 0, min = Integer.MAX_VALUE;
3     while (hi < nums.length) {
4       sum += nums[hi++];
5       while (sum >= s) {
6         min = Math.min(min, hi - lo);
7         sum -= nums[lo++];
8       }
9     }
10    return min == Integer.MAX_VALUE ? 0 : min;
11  }
```

使用队列（二）

上面使用的是相加的方式，也就是说队列中（或者是窗口中）的元素相加，然后判断是否大于等于s。其实我们还可以改为相减的方式，判断s是否小于等于0，其实基本原理和上面差不多，我们来看下

```
public int minSubArrayLen(int s, int[] nums) {
    int lo = 0, hi = 0, min = Integer.MAX_VALUE;
    while (hi < nums.length) {
        s -= nums[hi++];
        while (s <= 0) {
            min = Math.min(min, hi - lo);
            s += nums[lo++];
        }
    }
    return min == Integer.MAX_VALUE ? 0 : min;
}
```

二分法查找

我们还可以申请一个临时数组sums，其中sums[i]表示的是原数组nums中前i个元素的和，题中说了“给定一个含有 n 个正整数的数组”，既然是正整数，那么相加的和会越来越大，也就是sums数组中的元素是递增的。我们只需要找到sums[k]-sums[j]>=s，那么k-j就是满足的连续子数组，但不一定是最小的，所以我们要继续找，直到找到最小的为止。

怎么找呢，我们可以使用两个for循环来枚举，但这又和第一种暴力求解一样了，所以我们可以换种思路，求sums[k]-sums[j]>=s我们可以求sums[j]+s<=sums[k]，那这样就好办了，因为数组sums中的元素是递增的，也就是排序的，我们只需要求出sum[j]+s的值，然后使用二分法查找即可找到这个k值。

```
public int minSubArrayLen(int s, int[] nums) {
    int length = nums.length;
    int min = Integer.MAX_VALUE;
    int[] sums = new int[length + 1];
    for (int i = 1; i <= length; i++) {
        sums[i] = sums[i - 1] + nums[i - 1];
    }
    for (int i = 0; i <= length; i++) {
        int target = s + sums[i];
        int index = Arrays.binarySearch(sums, target);
        if (index < 0)
            index = ~index;
        if (index <= length) {
            min = Math.min(min, index - i);
        }
    }
    return min == Integer.MAX_VALUE ? 0 : min;
}
```

注意这里的查找函数

Arrays.binarySearch(sums, target);

如果找到就会返回值的下标，如果没找到就会返回一个负数，这个负数取反之后就是查找的值应该在数组中的位置

举个例子，比如排序数组[2，5，7，10，15，18，20]如果我们查找18，因为数组中有这个数，所以会返回18的下标5，如果我们查找9，因为数组中没这个数，所以会返回-4（至于这个是怎么得到的，大家可以看下源码，这里不再过多展开讨论），我们对他取反之后就是3，也就是说如果我们在数组中添加一个9，他在数组的下标是3，也就是第4个位置（也可以这么理解，只要取反之后不是数组的长度，那么他就是原数组中第一个比他大的值的下标）

直接使用窗口

上面第2种解法我们使用的是使用两个指针，我们也可以把它看做是一个窗口，每次往窗口中添加元素来判断是否满足。其实我们可以逆向思维，先固定一个窗口大小比如leng，然后遍历数组，查看在数组中leng个元素的和是否有满足的，如果没有满足的我们就扩大窗口的大小继续查找，如果有满足的我们就记录下窗口的大小leng，因为这个leng不一定是最小的，我们要缩小窗口的大小再继续找……

```
1   public int minSubArrayLen(int s, int[] nums) {
2     int lo = 1, hi = nums.length, min = 0;
3     while (lo <= hi) {
4       int mid = (lo + hi) >> 1;
5       if (windowExist(mid, nums, s)) {
6         hi = mid - 1;//找到就缩小窗口的大小
7         min = mid;//如果找到就记录下来
8       } else
9         lo = mid + 1;//没找到就扩大窗口的大小
10    }
11    return min;
12  }
13
14  //size窗口的大小
15  private boolean windowExist(int size, int[] nums, int s) {
16    int sum = 0;
17    for (int i = 0; i < nums.length; i++) {
18      if (i >= size)
19        sum -= nums[i - size];
20      sum += nums[i];
21      if (sum >= s)
22        return true;
23    }
24    return false;
25  }
```


往期推荐

- 388，先序遍历构造二叉树
- 379，柱状图中最大的矩形（难）

384，整数反转

原创 山大王wld 数据结构和算法 6月15日

收录于话题
#算法图文分析

96个 >



Take the sourest lemon that life has to offer and turn it into something resembling lemonade.

接过生活中酸涩的柠檬，把它变成酸甜可口的柠檬汽水。

问题描述

给出一个 32 位的有符号整数，你需要将这个整数中每位上的数字进行反转。

示例 1:

```
输入: 123
输出: 321
```

示例 2:

> 输入：-123
> 输出：-321

示例 3：

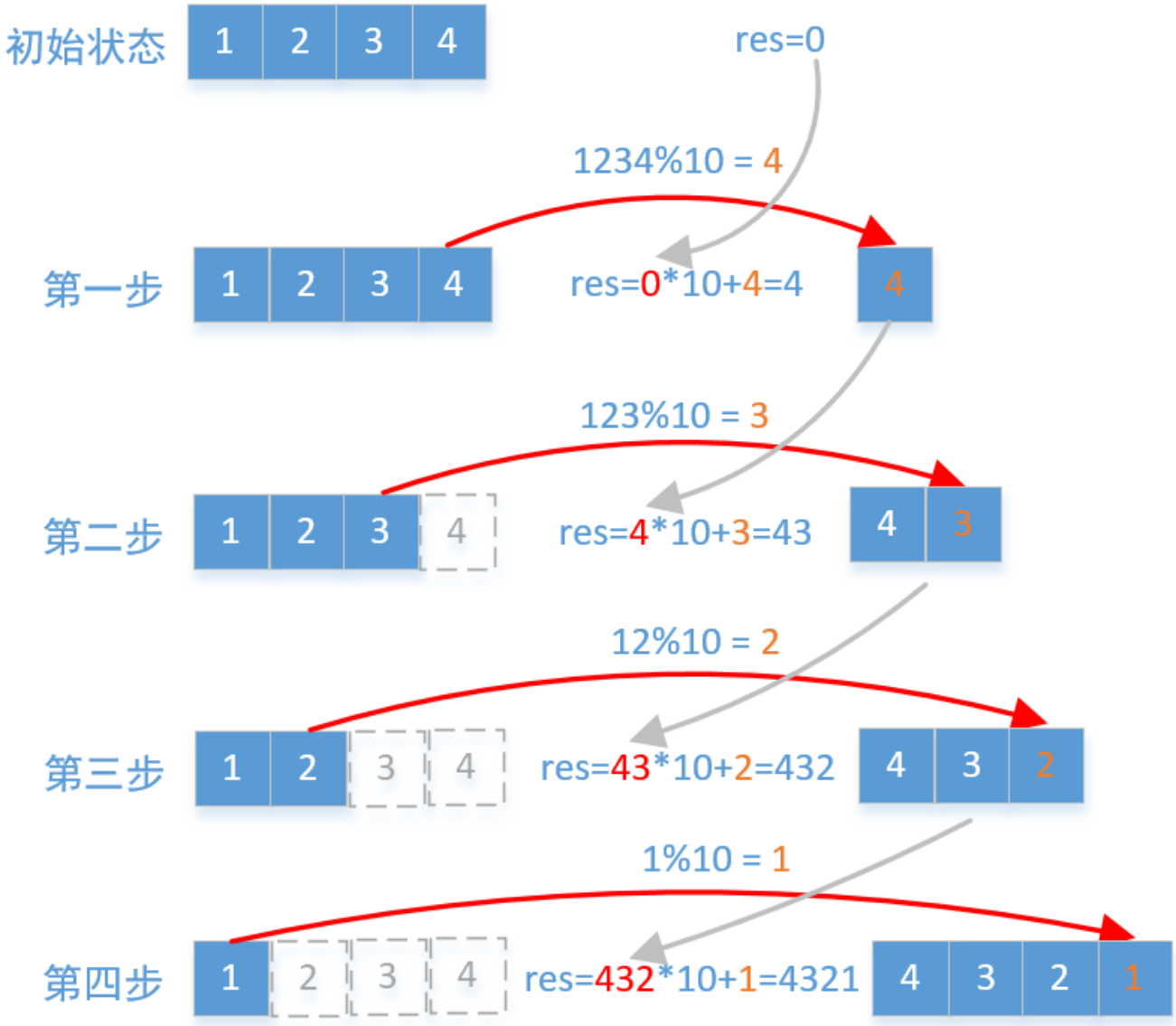
> 输入：120
> 输出：21

问题分析

> 看到这道题可能我们最容易想到的是先把他转化为一个字符串，然后再进行反转，代码如下

```
1   public int reverse(int x) {
2     boolean negative = x < 0;
3     StringBuilder stringBuilder = new StringBuilder(x + "");
4     if (negative)
5       stringBuilder.deleteCharAt(0);
6     stringBuilder.reverse();
7     long reverseDigit = Long.parseLong(stringBuilder.toString());
8     if (reverseDigit > Integer.MAX_VALUE || reverseDigit < Integer.MIN_VALUE)
9       return 0;
10    if (negative)
11      return (int) -reverseDigit;
12    return (int) reverseDigit;
13  }
```

> 第3行是先把他转化为字符串；
> 第6行再对字符串进行反转；
> 第8-9行如果反转之后大于int表示的范围就返回0；
> 第10-11行是对符号的处理；
>
> 这种也能实现，但效率实在是太低。下面我们就以数字1234为例画个图来看一下，如果不转化为字符串该怎么实现



我们看到上一步的结果在下一步都会先乘以10，然后在加一个个位数就是当前的值，一直这样循环下去，直到全部反转为止。大家可能会怀疑，上面图中分析的是正数，如果是负数该怎么办，其实负数也是一样，大家可以自己画个图看一下。上面的图很容易理解，我们来看下代码

```
1  public int reverse(int x) {
2    long res = 0;
3    while (x != 0) {
4      res = res * 10 + x % 10;
5      x /= 10;
6    }
7    return (int) res == res ? (int) res : 0;
8  }
```

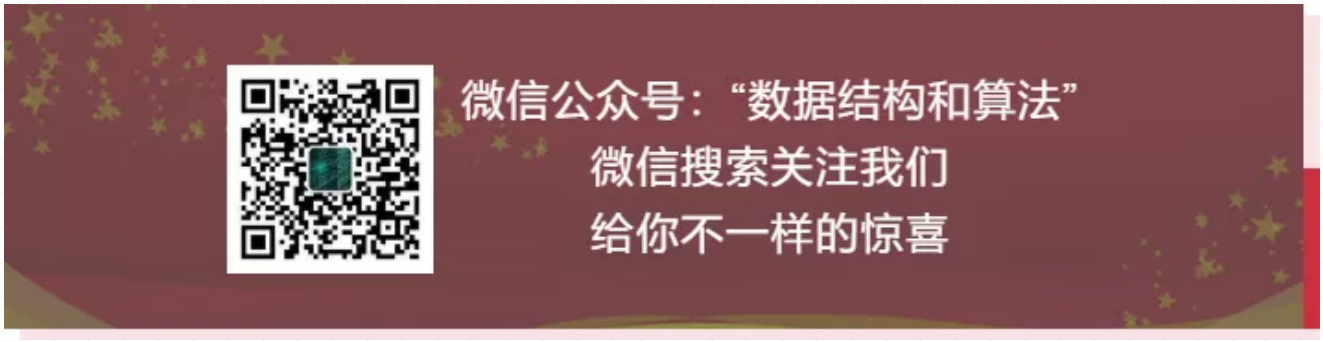
注意这里的res是long类型，在第7行的时候，会把它转化为int类型，如果res的范围大于int类型表示的范围，转化之后是不相等的，直接返回0，如果在int类型表示的范围内，转化之后是相等的，返回转化后的值即可。

382，每日温度的5种解题思路

原创 山大王wld 数据结构和算法 6月12日

收录于话题
#算法图文分析

96个 >



People make mistakes. That's the very reason why they put rubbers on the ends of pencils.

人们都会犯错，这便是铅笔需要橡皮的原因。

问题描述

请根据每日气温列表，重新生成一个列表。对应位置的输出为：要想观测到更高的气温，至少需要等待的天数。如果气温在这之后都不会升高，请在该位置用 0 来代替。

例如

给定一个列表
[73, 74, 75, 71, 69, 72, 76, 73]

你的输出应该是

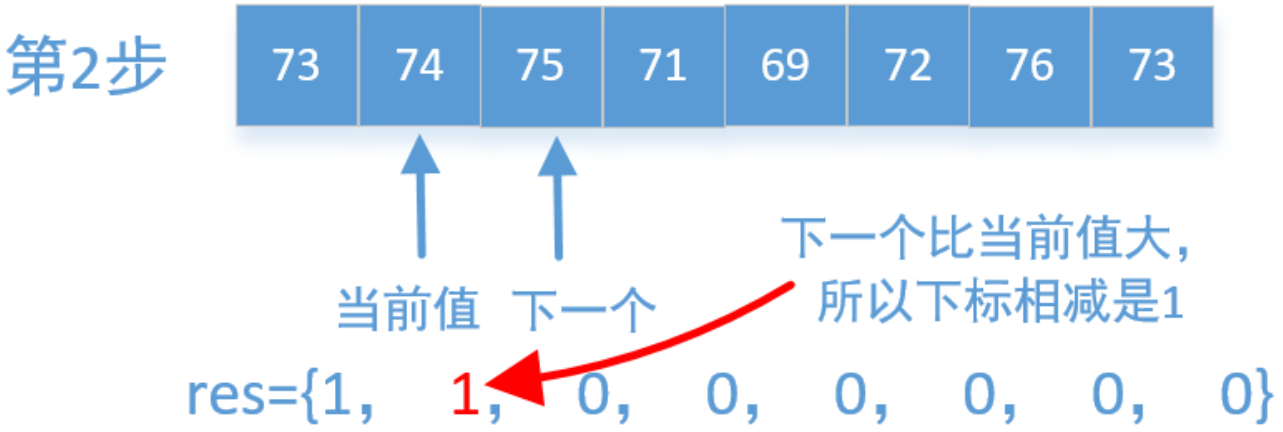
[1，1，4，2，1，1，0，0]。

说明：
73的时候只需要等1天，温度是74比73大。
74的时候只需要等1天，温度是75比74大。
75的时候只需要等4天，温度是76比75大。
……

暴力求解

看到这道题我们首先想到的是暴力求解。他的原理是遍历每一个元素，然后再从当前元素往后找比它大的，找到之后记录下他俩位置的差值，然后停止内层循环，如果没找到默认为0。我们画个图来看一下





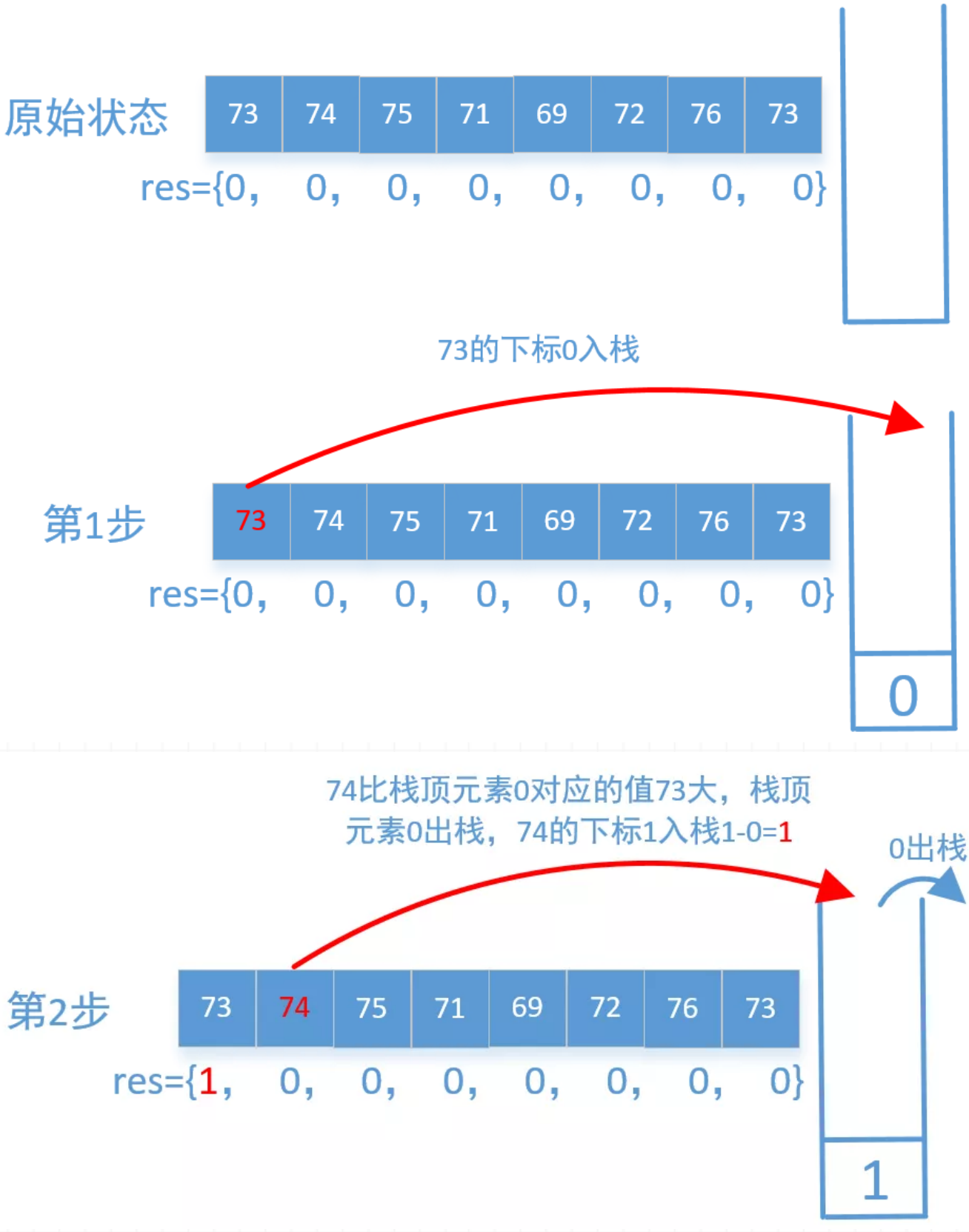


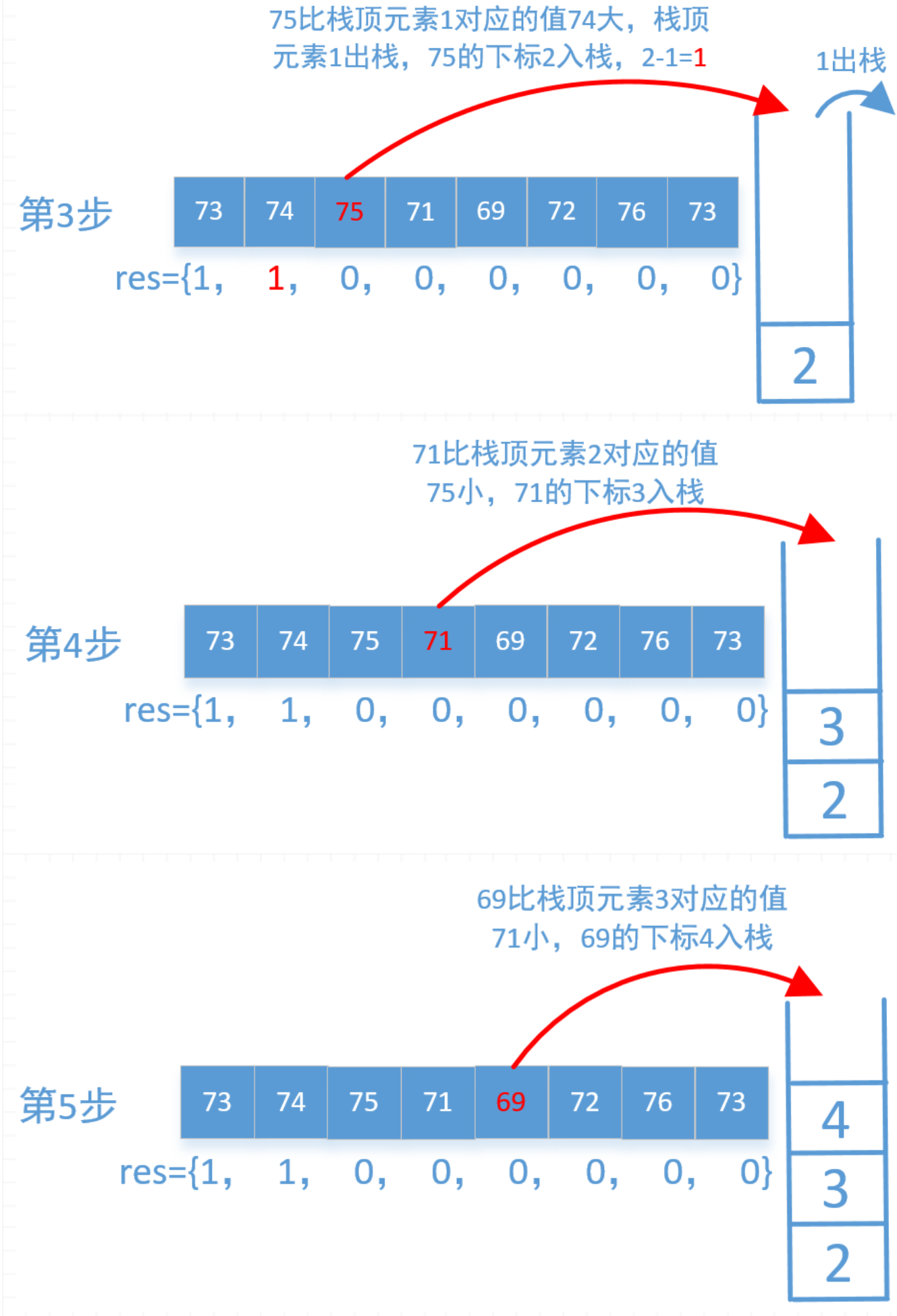
> 看明白了上面的分析过程，代码就容易多了，我们来看下

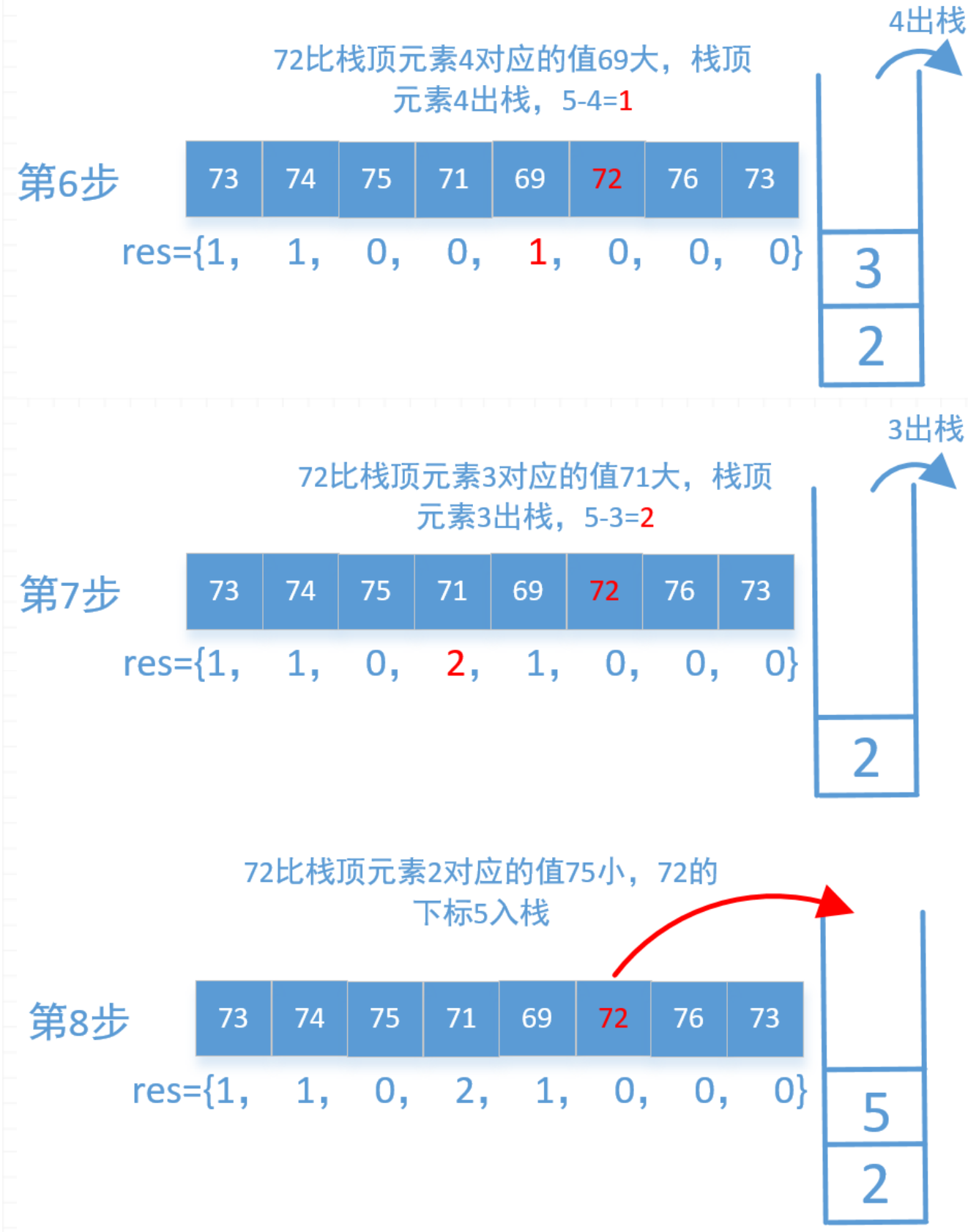
```
public int[] dailyTemperatures(int[] T) {
    int length = T.length;
    int[] res = new int[length];
    for (int i = 0; i < length; i++) {
        for (int j = i + 1; j < length; j++) {
            if (T[j] > T[i]) {
                res[i] = j - i;
                break;
            }
        }
    }
    return res;
}
```

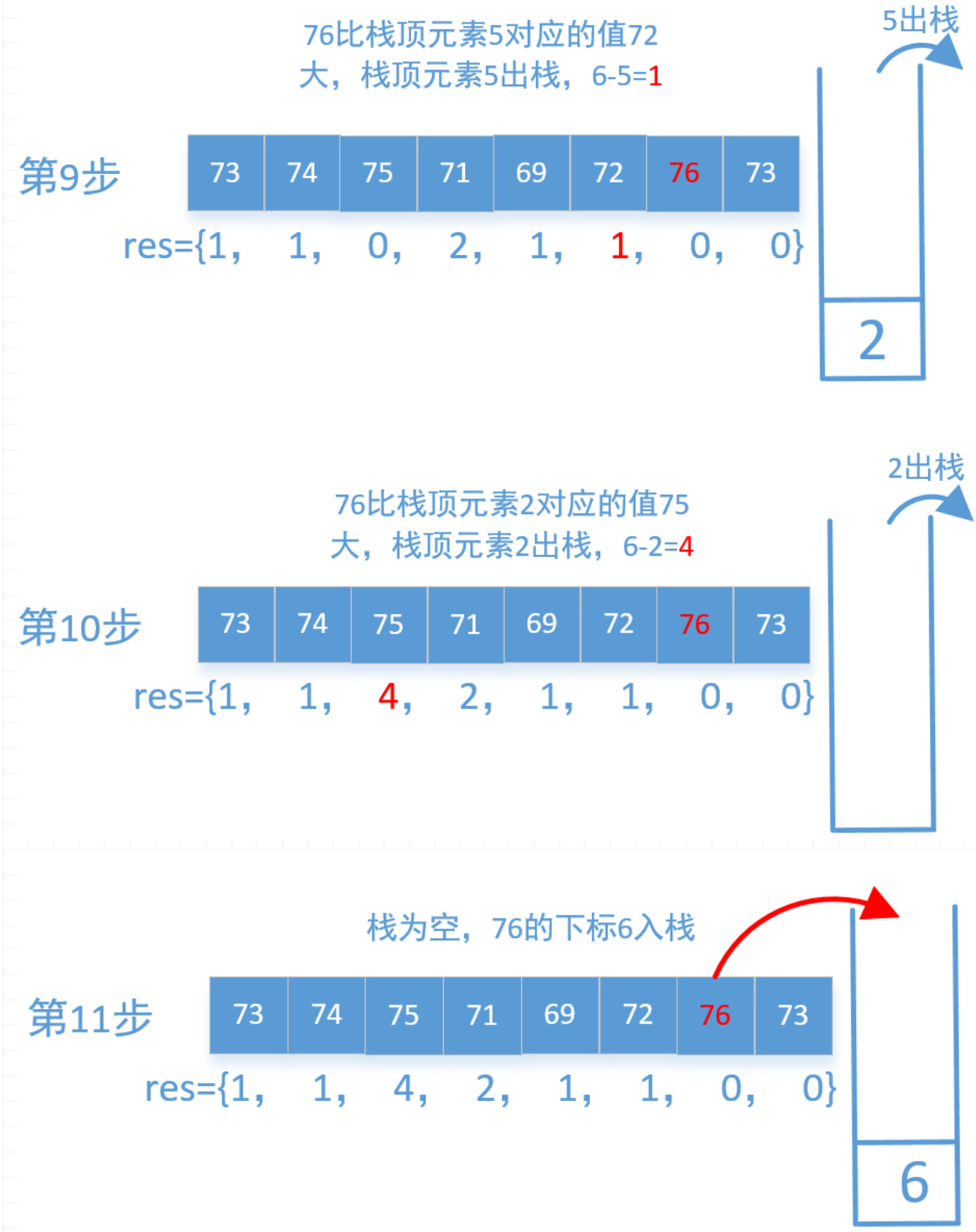
使用栈解决

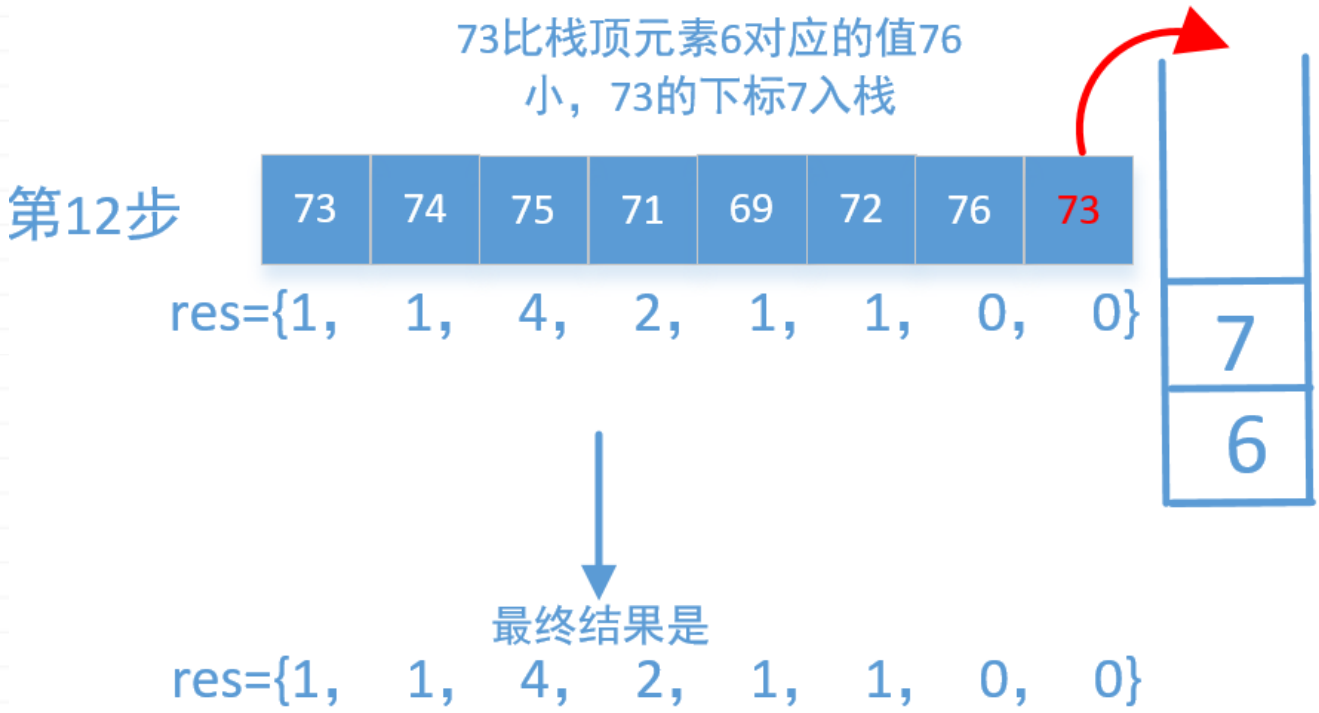
暴力求解，效率并不高，我们还可以使用栈来解决，栈存储的是元素的下标，不是元素的值。他的原理就是我们遍历到每个元素的时候用它和栈顶（栈不为空，如果为空直接入栈）元素比较，如果比栈顶元素小就把它对应的下标压栈，如果比栈顶元素大，说明栈顶元素遇到了右边比它大的，然后栈顶元素出栈，在计算下标的差值……重复这样计算。我们就还用上面的数据[73，74，75，71，69，72，76，73]画个图来看下




75比栈顶元素1对应的值74大，栈顶元素1出栈，75的下标2入栈，2-1=1
1出栈
第3步
73 74 75 71 69 72 76 73
res={1, 1, 0, 0, 0, 0, 0, 0}
2
71比栈顶元素2对应的值75小，71的下标3入栈
第4步
73 74 75 71 69 72 76 73
res={1, 1, 0, 0, 0, 0, 0, 0}
3
2
69比栈顶元素3对应的值71小，69的下标4入栈
第5步
73 74 75 71 69 72 76 73
res={1, 1, 0, 0, 0, 0, 0, 0}
4
3
2


4出栈
72比栈顶元素4对应的值69大，栈顶元素4出栈，5-4=1
第6步
73 74 75 71 69 72 76 73
res={1, 1, 0, 0, 1, 0, 0, 0}
3
2
3出栈
72比栈顶元素3对应的值71大，栈顶元素3出栈，5-3=2
第7步
73 74 75 71 69 72 76 73
res={1, 1, 0, 2, 1, 0, 0, 0}
2
72比栈顶元素2对应的值75小，72的下标5入栈
第8步
73 74 75 71 69 72 76 73
res={1, 1, 0, 2, 1, 0, 0, 0}
5
2


76比栈顶元素5对应的值72
大，栈顶元素5出栈，6-5=1
5出栈
第9步
73 74 75 71 69 72 76 73
res={1, 1, 0, 2, 1, 1, 0, 0}
2
76比栈顶元素2对应的值75
大，栈顶元素2出栈，6-2=4
2出栈
第10步
73 74 75 71 69 72 76 73
res={1, 1, 4, 2, 1, 1, 0, 0}
栈为空，76的下标6入栈
第11步
73 74 75 71 69 72 76 73
res={1, 1, 4, 2, 1, 1, 0, 0}
6



> 搞懂了上面的分析过程我们再来看代码

```java
public int[] dailyTemperatures(int[] T) {
    Stack<Integer> stack = new Stack<>();
    int[] ret = new int[T.length];
    for (int i = 0; i < T.length; i++) {
        while (!stack.isEmpty() && T[i] > T[stack.peek()]) {
            int idx = stack.pop();
            ret[idx] = i - idx;
        }
        stack.push(i);
    }
    return ret;
}
```

> 我们还可以用数组替换栈

```java
public int[] dailyTemperatures(int[] T) {
    int[] stack = new int[T.length];
    int top = -1;
    int[] res = new int[T.length];
    for (int i = 0; i < T.length; i++) {
        while (top >= 0 && T[i] > T[stack[top]]) {
            int idx = stack[top--];
            res[idx] = i - idx;
        }
        stack[++top] = i;
    }
    return res;
}
```

参照第379题

这题和第379题有非常相似的地方，如果看过379，柱状图中最大的矩形（难），这题就非常容易了，代码也非常相似，这个栈中元素所对应值的顺序从栈底到栈顶是递减的，和379题正好相反，我们来看下代码

```
public int[] dailyTemperatures(int[] T) {
  int length = T.length;
  Stack<Integer> stack = new Stack<>();
  int[] res = new int[length];
  for (int i = 0; i < length; i++) {
    int h = T[i];
    if (stack.isEmpty() || h <= T[stack.peek()]) {
      stack.push(i);
    } else {
      int top = stack.pop();
      res[top] = i - top;
      i--;
    }
  }
  return res;
}
```

剪枝

最后一种解法效率也是非常高的，代码中有注释，就不在过多解释，有兴趣的可以看下

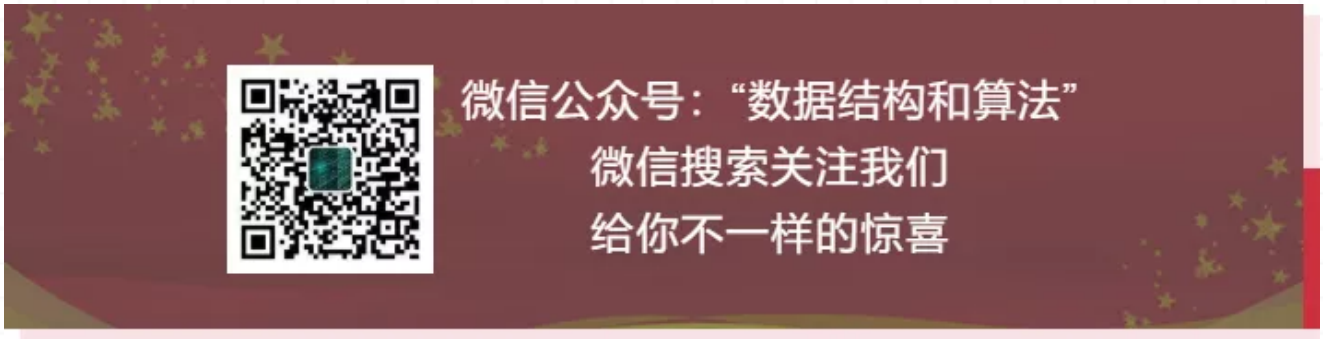
```
public int[] dailyTemperatures(int[] T) {
  int[] res = new int[T.length];
  //从后面开始查找
  for (int i = res.length - 1; i >= 0; i--) {
    int j = i + 1;
    while (j < res.length) {
      if (T[j] > T[i]) {
        //如果找到就停止while循环
        res[i] = j - i;
        break;
      } else if (res[j] == 0) {
        //如果没找到，并且res[j]==0。说明第j个元素后面没有
        //比第j个元素大的值，因为这一步是第i个元素大于第j个元素的值，
        //那么很明显这后面就更没有大于第i个元素的值。直接终止while循环。
        break;
      } else {
        //如果没找到，并且res[j]！=0说明第j个元素后面有比第j个元素大的值，
        //然后我们让j往后挪res[j]个单位，找到那个值，再和第i个元素比较
        j += res[j];
      }
    }
  }
  return res;
}
```

380，缺失的第一个正数（中）

原创 山大王wld 数据结构和算法 6月9日

收录于话题
#算法图文分析

96个 >



You have to decide whether to trust your own eyes and ears, or what other people say.

你需要决定是相信自己的眼睛和耳朵，还是相信别人的话。

问题描述

> 给你一个未排序的整数数组，请你找出其中没有出现的最小的正整数。

示例 1：

```
输入：[1,2,0]
输出：3
```

示例 2：

```
输入: [3,4,-1,1]
输出: 2
```

示例 3:

```
输入: [7,8,9,11,12]
输出: 1
```

问题分析

这道题最容易想到的就是暴力求解，我们从1到数组的长度一个个找，如果找到了就继续寻找下一个，如果没找到就直接返回，代码很简单，我们来看下

01 暴力求解

```java
public int firstMissingPositive(int[] nums) {
    for (int i = 1; i <= nums.length; i++) {
        boolean has = false;
        for (int j = 0; j < nums.length; j++) {
            if (nums[j] == i) {
                has = true;
                break;
            }
        }
        if (!has) {
            //没有找到这个数，直接返回
            return i;
        }
    }
    return nums.length + 1;
}
```

在算法中我们只要听到暴力二字，就知道这种答案效率肯定不会很高，一般这种答案在面试中非常不占优势

02 排序之后再查找

```java
public int firstMissingPositive(int[] nums) {
    int len = nums.length;
    Arrays.sort(nums);//先排序
    for (int i = 1; i <= len; i++) {
        int res = binarySearch(nums, i);
        //一个个查找，如果没找到就返回
        if (res == -1)
            return i;
```

```
    }
    return len + 1;
  }

  //二分法查找
  private int binarySearch(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1;
    while (left <= right) {
      int mid = left + ((right - left) >> 1);
      if (nums[mid] == target) {
        return mid;
      } else if (nums[mid] < target) {
        left = mid + 1;
      } else {
        right = mid - 1;
      }
    }
    return -1;
  }
```

这种我们可以对数组排序之后再一个个查找，因为排序之后我们可以使用二分法查找，代码也很简单。我们再来看下一种，通过集合来一个个判断

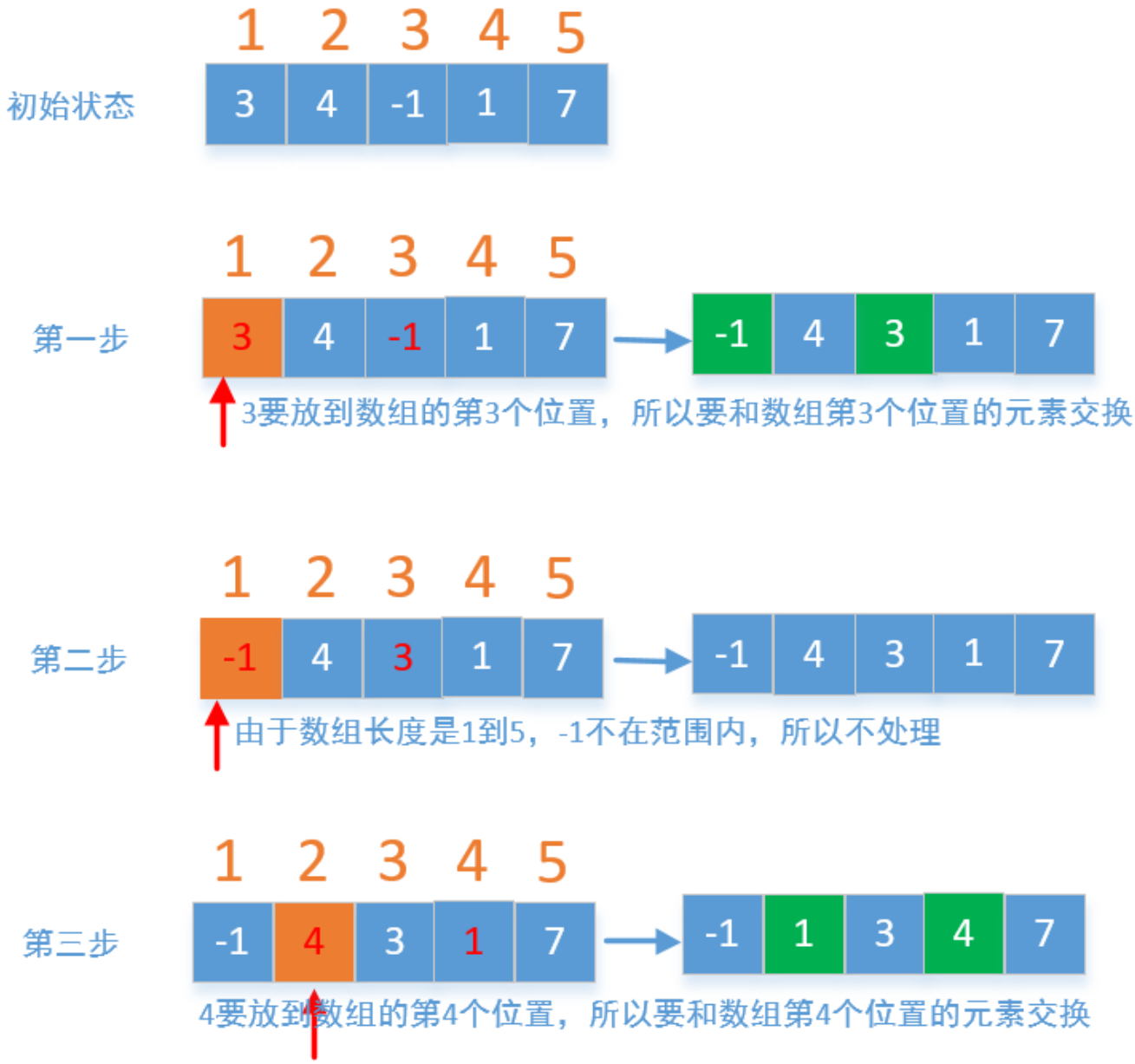
03 存放到集合中再查找

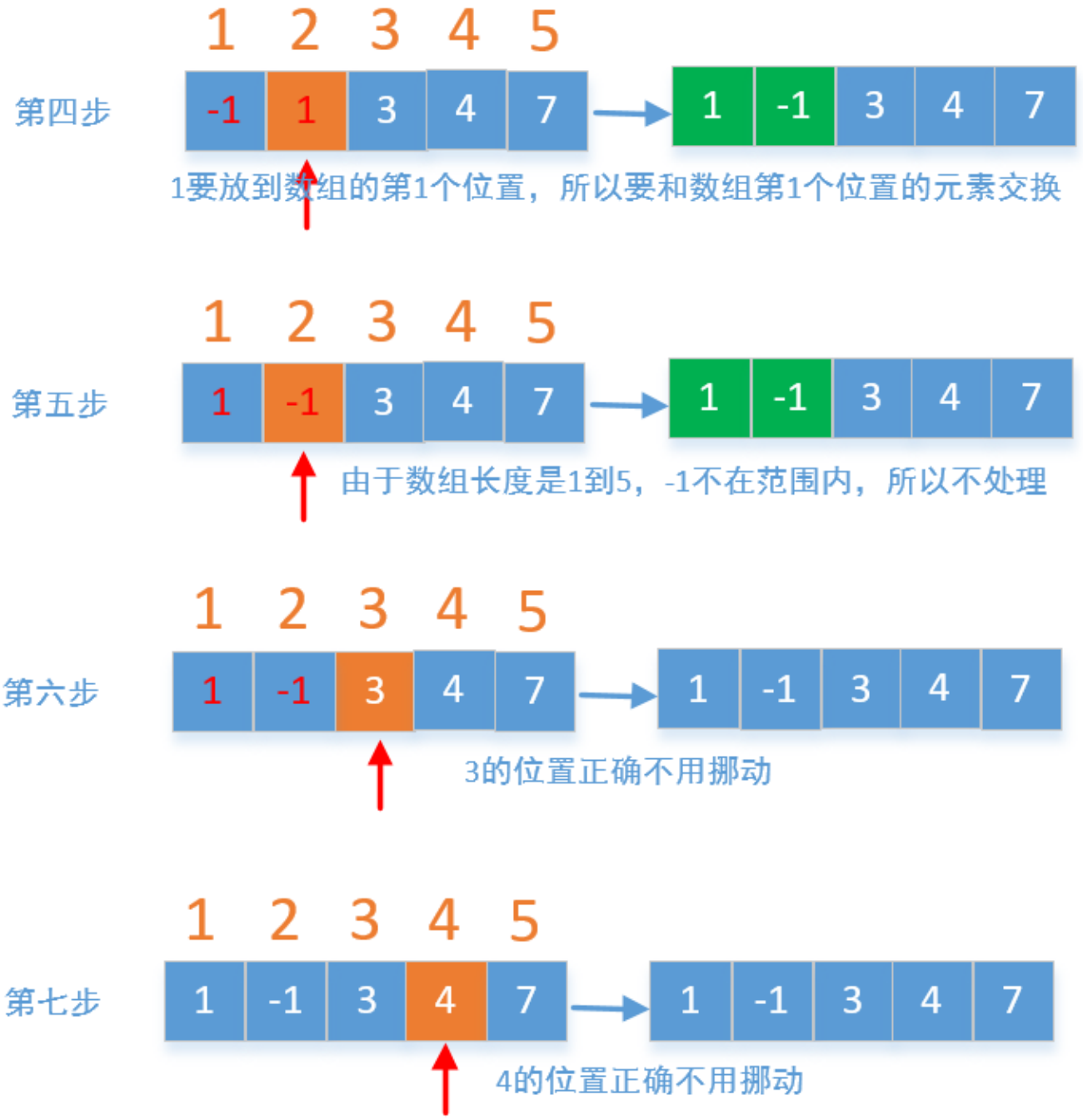
```
1  public int firstMissingPositive(int[] nums) {
2    int len = nums.length;
3    Set<Integer> hashSet = new HashSet<>();
4    for (int num : nums) {
5      hashSet.add(num);
6    }
7    for (int i = 1; i <= len; i++) {
8      if (!hashSet.contains(i))
9        return i;
10   }
11   return len + 1;
12 }
```

我们可以把数组中的元素全部放到集合set中，然后在第7到10行，从1到数组长度一个个判断集合set中是否包含这个值，如果没有直接返回。这个也很好理解，下面我们再来看第四种解法

04 存放到对应的位置判断

我们还可以把数组中的数字放到对应下标的位置，然后再循环一遍，查看每个位置和对应的下标是否一致，如果不一致直接返回即可，我们就以[3，4，-1，1，7]为例来画个图分析一下


初始状态
1 2 3 4 5
3 4 -1 1 7
第一步
1 2 3 4 5
3 4 -1 1 7
-1 4 3 1 7
3要放到数组的第3个位置，所以要和数组第3个位置的元素交换
第二步
1 2 3 4 5
-1 4 3 1 7
-1 4 3 1 7
由于数组长度是1到5，-1不在范围内，所以不处理
第三步
1 2 3 4 5
-1 4 3 1 7
-1 1 3 4 7
4要放到数组的第4个位置，所以要和数组第4个位置的元素交换


第四步
1 2 3 4 5
-1 1 3 4 7
1 -1 3 4 7
1要放到数组的第1个位置，所以要和数组第1个位置的元素交换
第五步
1 2 3 4 5
1 -1 3 4 7
1 -1 3 4 7
由于数组长度是1到5，-1不在范围内，所以不处理
第六步
1 2 3 4 5
1 -1 3 4 7
1 -1 3 4 7
3的位置正确不用挪动
第七步
1 2 3 4 5
1 -1 3 4 7
1 -1 3 4 7
4的位置正确不用挪动



放好之后最后在执行一遍循环，发现2的位置存放的是-1，直接返回2即可。




```java
public int firstMissingPositive(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        //如果在指定的位置就不需要修改
        if (i + 1 == nums[i])
            continue;
        int x = nums[i];
        if (x >= 1 && x <= nums.length && x != nums[x - 1]) {
            swap(nums, i, x - 1);
            i--;//抵消上面的i++，如果交换之后就不++;
        }
    }
    //最后在执行一遍循环，查看对应位置的元素是否正确，如果不正确直接返回
    for (int i = 0; i < nums.length; i++) {
        if (i + 1 != nums[i])
            return i + 1;
    }
    return nums.length + 1;
}

//交换两个数的值
public void swap(int[] A, int i, int j) {
    if (i != j) {
        A[i] ^= A[j];
        A[j] ^= A[i];
        A[i] ^= A[j];
    }
}
```

05 位运算求解

位运算的实现原理和上面第4种答案类似，只不过这里使用的是位运算的方式来解决的。我们知道在java中一个int类型占4个字节是32位，我们可以申请一个数组，1到32我们可以存放到数组的第一个元素中，33到64可以存放到第2个元素中……，有的同学可能好奇，一个数字怎么可能存放32个数呢。因为一个int类型

数字是32位的，也就是由32个0和1组成，我们只要统计1在存储中的位置即可，我们来看下代码。

```
public int firstMissingPositive(int[] nums) {
    int length = nums.length;
    int bit[] = new int[(length - 1) / 32 + 1];
    for (int i = 0; i < nums.length; i++) {
        int digit = nums[i];
        //数组必须在1到length之间才有效
        if (digit >= 1 && digit <= length) {
            int index = (digit - 1) / 32;
            bit[index] = bit[index] | (1 << ((digit - 1) % 32));
        }
    }
    //最后在执行一遍循环，查看对应位置的元素是否正确，如果不正确直接返回
    for (int i = 0; i < nums.length; i++) {
        if ((bit[i / 32] & (1 << (i % 32))) == 0)
            return i + 1;
    }
    return length + 1;
}
```

06 填补替换

下面再来看最后一种解法，也是比较经典的，如果某个元素的值是无效的，他会让数组后面的元素填补过来，然后再判断。注释写在代码中了，有兴趣的也可以看下。

```
public int firstMissingPositive(int[] nums) {
    int start = 0;
    int end = nums.length - 1;
    while (start <= end) {
        int index = nums[start] - 1;
        if (index == start) {
            //存放的位置正确
            start++;
        } else if (index < 0 || index > end || nums[start] == nums[index]) {
            //前面两个表示数字不在存放位置范围内，就让数组end位置的元素把这个无效的
            // 元素覆盖掉，后面一个表示的是index这个位置存放正确了就不需要在存放了
            nums[start] = nums[end--];
        } else {
            //把start对应的元素存放到正确的位置
            nums[start] = nums[index];
            nums[index] = index + 1;
        }
    }
    return start + 1;
}
```

往期推荐

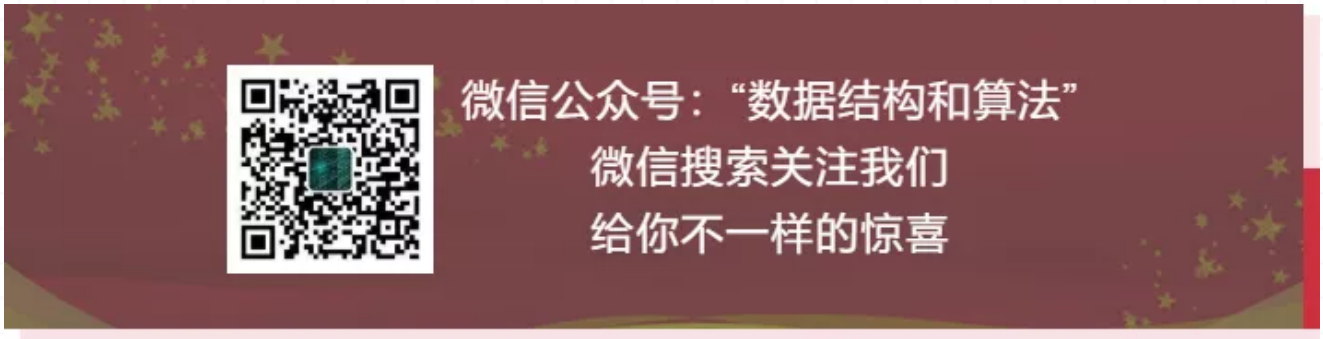
- 377，调整数组顺序使奇数位于偶数前面

379，柱状图中最大的矩形（难）

原创 山大王wld 数据结构和算法 6月9日

收录于话题
#算法图文分析

96个 >



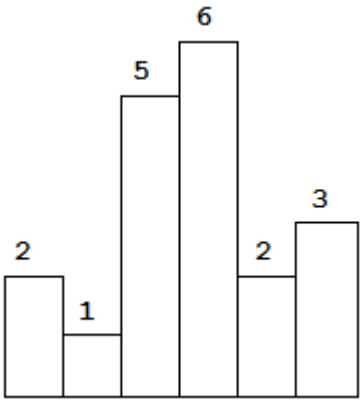
Remember, quitters never win...and winners never quit.

记住，放弃者难以成功，成功者决不放弃。

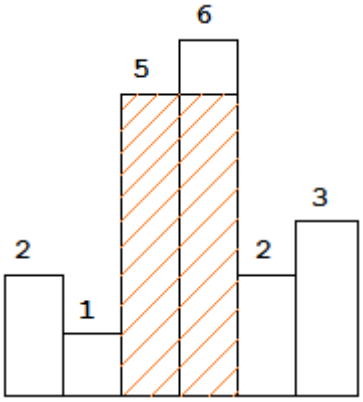
问题描述

给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。

求在该柱状图中，能够勾勒出来的矩形的最大面积。



以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 [2,1,5,6,2,3]。



图中阴影部分为所能勾勒出的最大矩形面积，其面积为 10 个单位。

示例：

```
输入: [2,1,5,6,2,3]
输出: 10
```

问题分析

01 暴力求解

最简单的方式就是暴力求解，我们都知道暴力求解的效率很差，但不妨碍我们做出来。暴力求解有两种方式。

一种是从左边确定一根柱子，然后从左往右扫描，确定以当前柱子的高为最大高度所围成的最大矩形（这个矩形的高度不能超过当前柱子的高度），记录下最大面积。

还一种是确定一根柱子以后分别从他的前后两个方向扫描，确定以当前柱子高度为矩形的高所围成的最大矩形（这个矩形的高度就是当前这个柱子的高度），记录下最大面积。

我们来分别看下这两种写法的代码

```java
public int largestRectangleArea(int[] heights) {
    int length = heights.length;
    int area = 0;
    // 枚举左边界
    for (int left = 0; left < length; ++left) {
      int minHeight = Integer.MAX_VALUE;
      // 枚举右边界
      for (int right = left; right < length; ++right) {
        // 确定高度，我们要最小的高度
        minHeight = Math.min(minHeight, heights[right]);
        // 计算面积，我们要保留计算过的最大的面积
        area = Math.max(area, (right - left + 1) * minHeight);
      }
    }
    return area;
  }
```

暴力解法的另一种写法

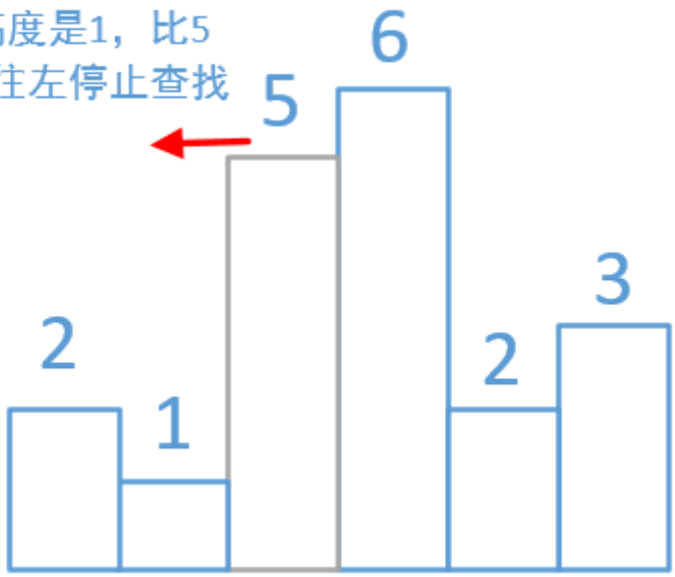
```java
public int largestRectangleArea(int[] heights) {
    int area = 0, length = heights.length;
    // 遍历每个柱子，以当前柱子的高度作为矩形的高 h，
    // 从当前柱子向左右遍历，找到矩形的宽度 w。
    for (int i = 0; i < length; i++) {
      int w = 1, h = heights[i], j = i;
      //往左边找
      while (--j >= 0 && heights[j] >= h) {
        w++;
      }
      j = i;
      //往右边找
      while (++j < length && heights[j] >= h) {
        w++;
      }
      //记录最大面积
      area = Math.max(area, w * h);
    }
    return area;
  }
```

02 使用栈求解

我们看一下暴力求解的第二种方式，他是每遍历一根柱子就会往左和往右查找，直到找到比他小的为止，然后以当前柱子的高度为矩形的高，以不低于当前柱子的数量（必须是和当前柱子挨着的）为矩形的宽来计算矩形的面积，我们就用上面的示例以当前高度为5的柱子为例来画个图看一下。

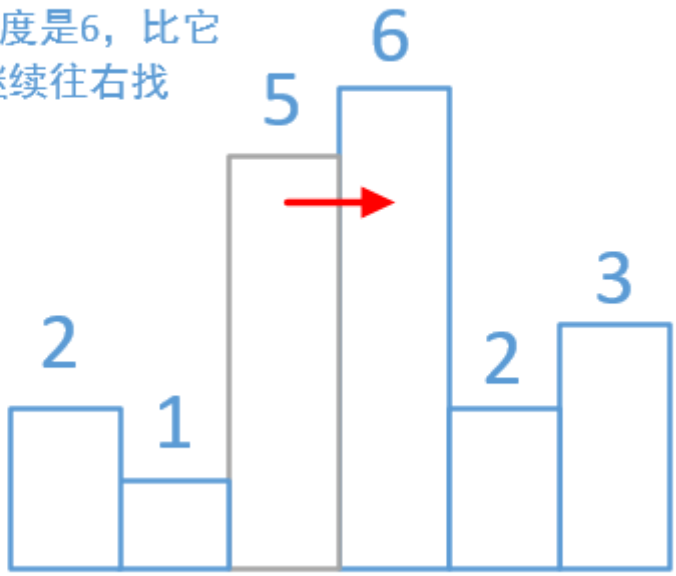
当前柱子高度是5，往左找，高度是1，比5小，所以往左停止查找

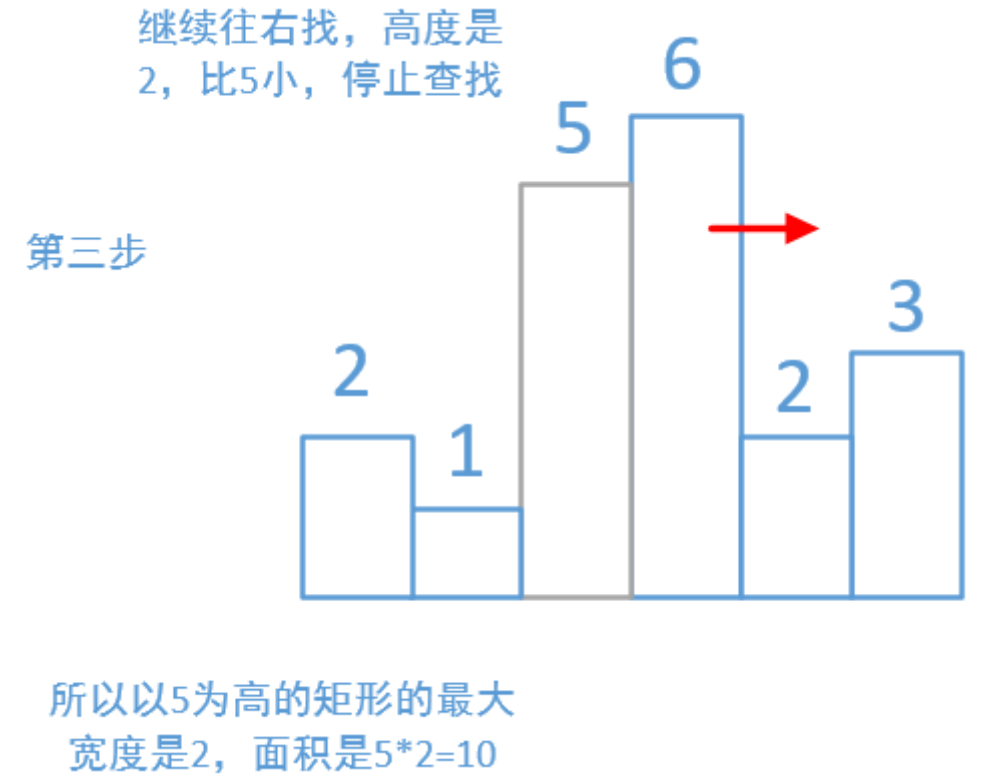
第一步



当前柱子高度是5，往右找，高度是6，比它高，继续往右找

第二步





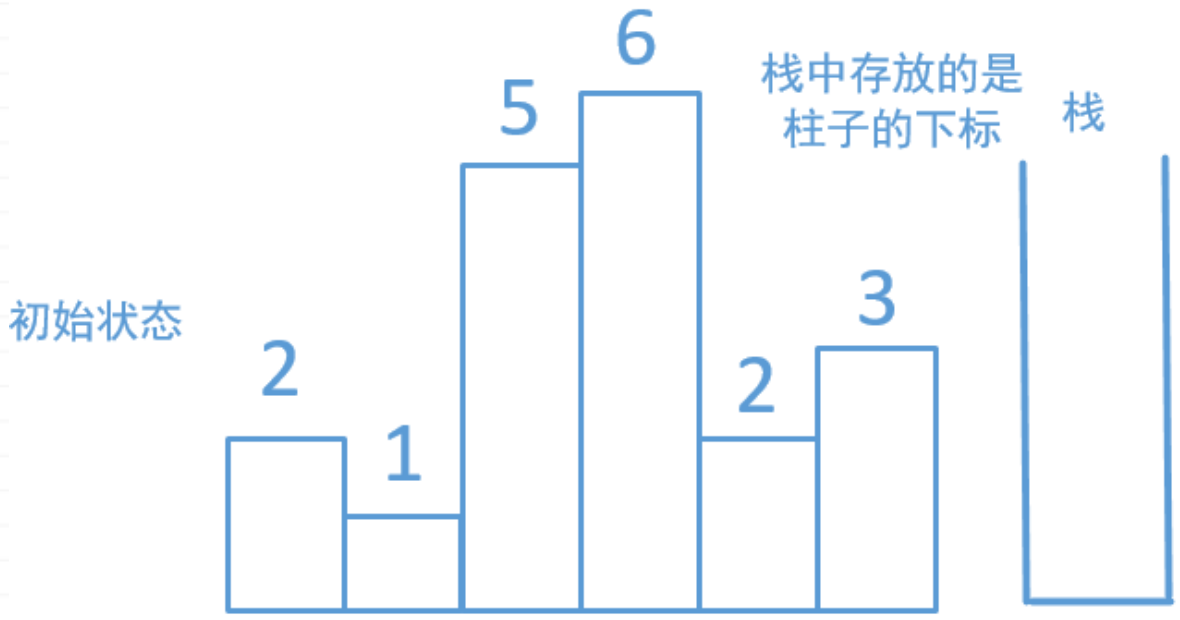
看明白了上面的分析，我们是不是会有点启发，我们如果以当前柱子的高度为矩形的高，我们只需要往左和往右找到小于当前的柱子，就可以确定矩形的宽度。知道宽和高面积自然就求出来了。

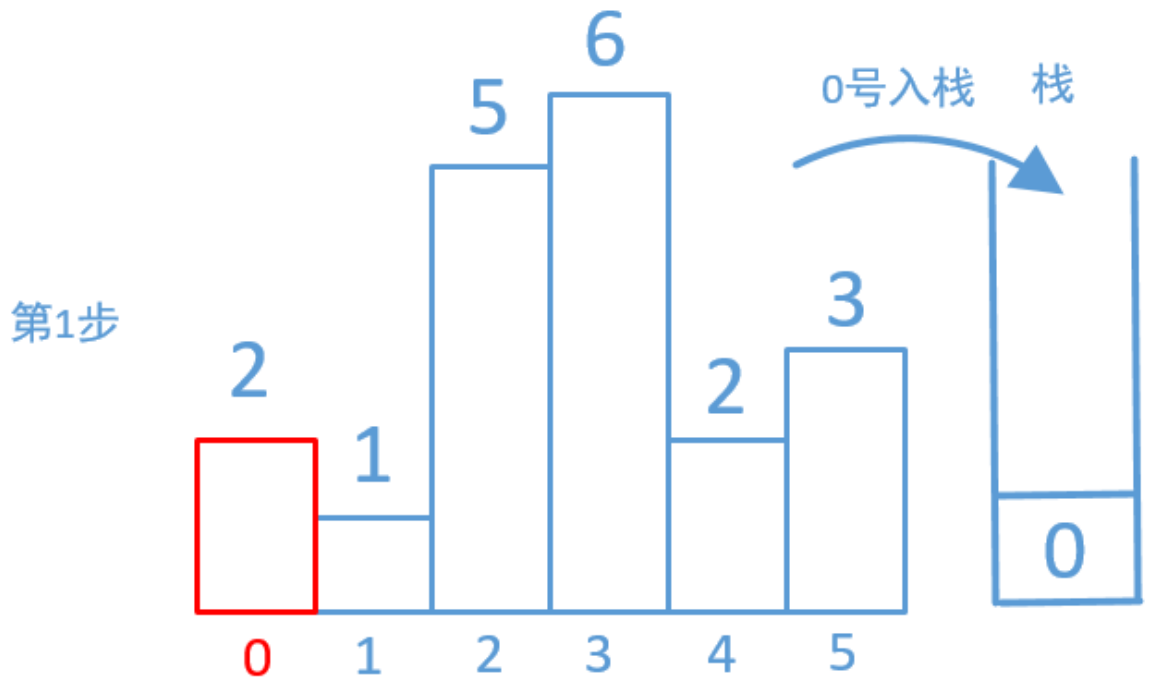
但是矩形的宽度怎么求呢，我们这里并不是直接求，我们要维护一个递增的栈（从栈底到栈顶的元素所对应柱子的高度是递增的），注意栈中存放的是柱子的下标，不是柱子的高度。

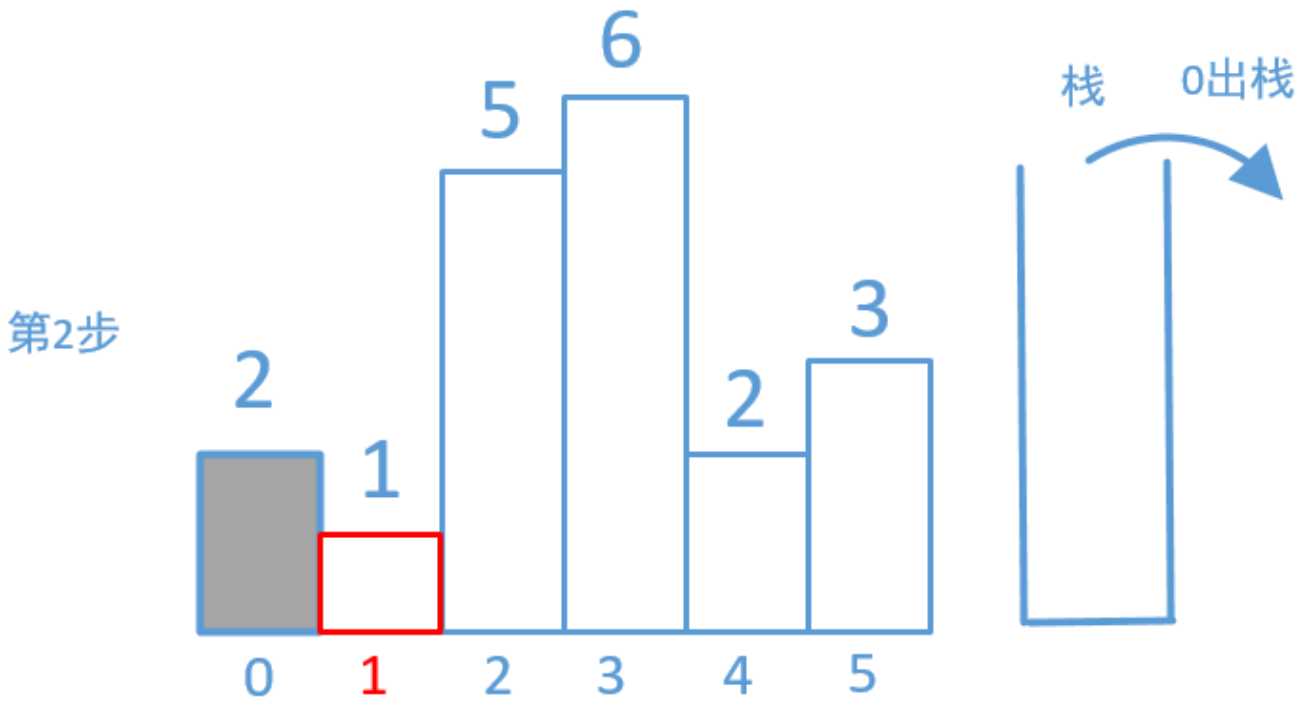
我们每遍历一个柱子的时候如果当前柱子i的值大于等于栈顶元素对应柱子的高度，我们就把当前柱子的下标压入到栈顶中。

如果当前柱子i的值小于栈顶元素柱子k的高度，说明栈顶元素对应的柱子k遇到了右边比它小的柱子，我们只需要弹出栈顶柱子k。那么怎么确定柱子k他左边比它小的柱子呢，很明显因为栈从栈底到栈顶是递增的，柱子k已经出栈了，现在栈顶元素w对应柱子的高度就是柱子k遇到的左边比他小的值（有可能这时候栈顶元素w对应柱子的高度和柱子k对应的高度一样大，但没关系，因为下一步我们还会在继续计算）。根据上面的暴力求解，我们知道一个柱子左边和右边比它小的值，就可以以当前柱子的高度为矩形的高，计算出矩形的面积。然后我们在用栈顶元素w对应的值和柱子i对应的值比较，重复上面的步骤……直到柱子i对应的值大于栈顶元素对应的值（或栈为空）为止。（注意这里的比较是栈中元素对应柱子高度的比较，不是栈中元素的比较）

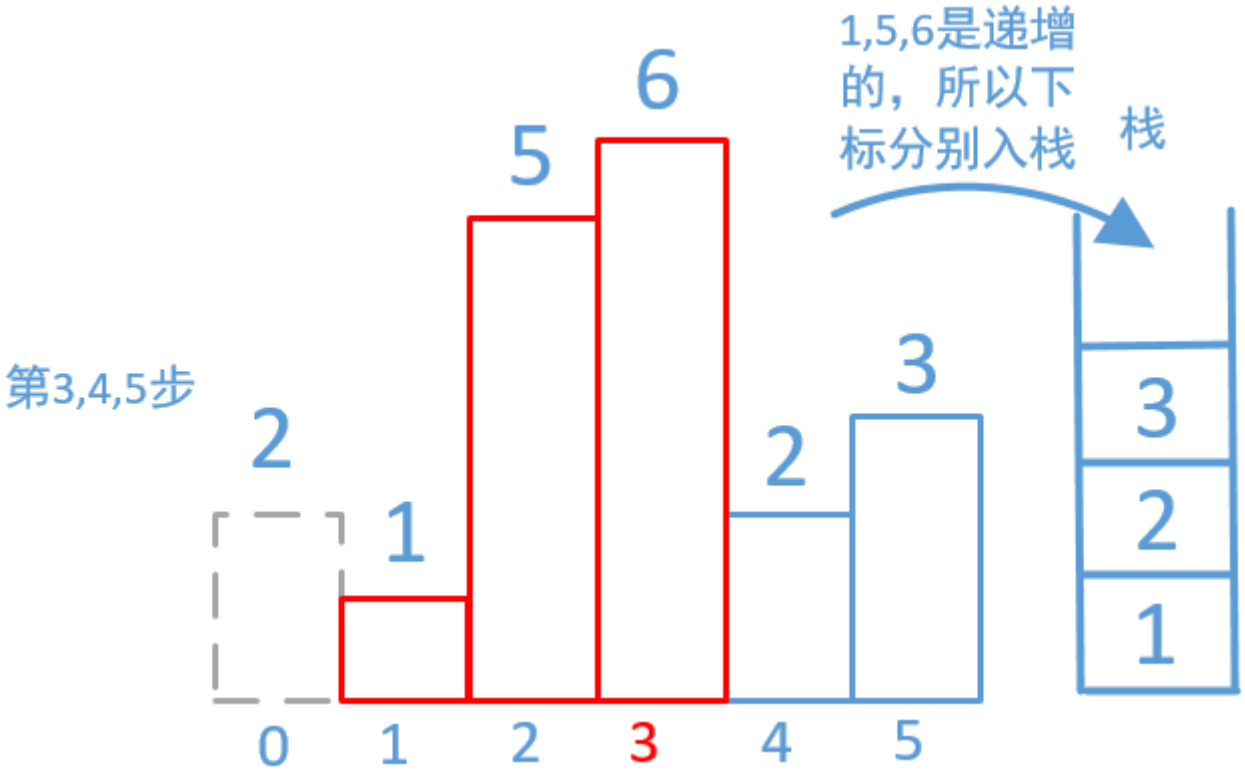
上面的解说比较绕，看不明白可以多读几遍，我们来画个图看一下


初始状态
2
1
5
6
2
3
栈中存放的是
柱子的下标
栈

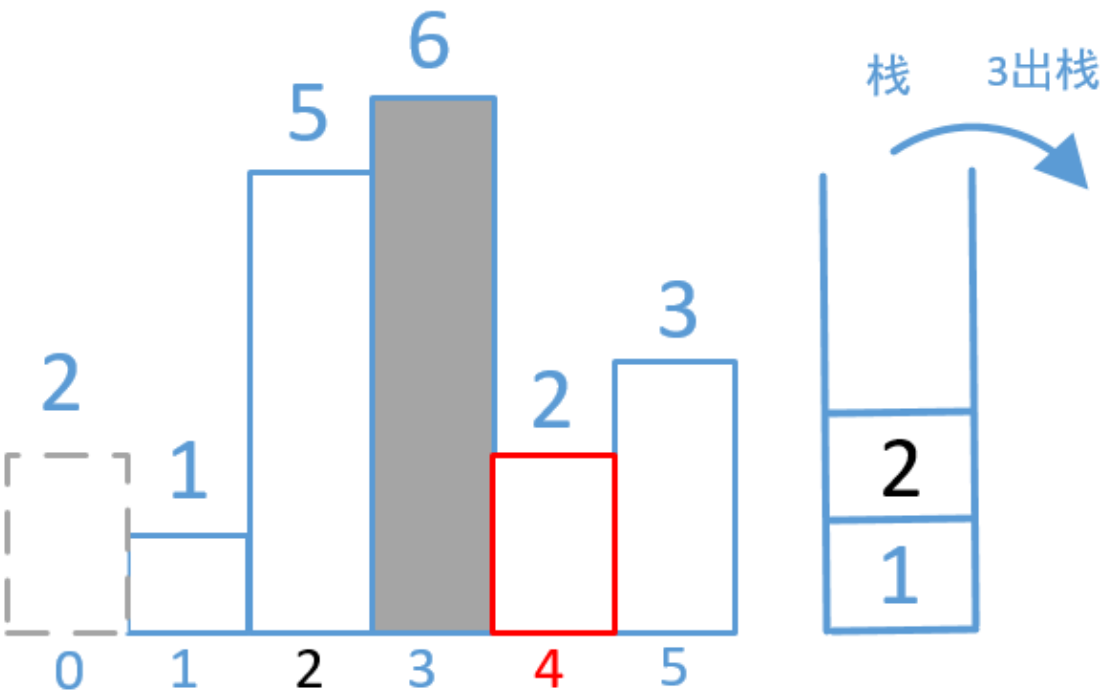

第1步
2
1
5
6
2
3
0号入栈
栈
0
0
1
2
3
4
5



由于1比栈顶元素0号柱子2小，所以0出栈，最大面积是2*1=2
当栈为空的时候，矩形宽就是遍历到的柱子下标。2表示的是出栈的下标对应的柱子的高度

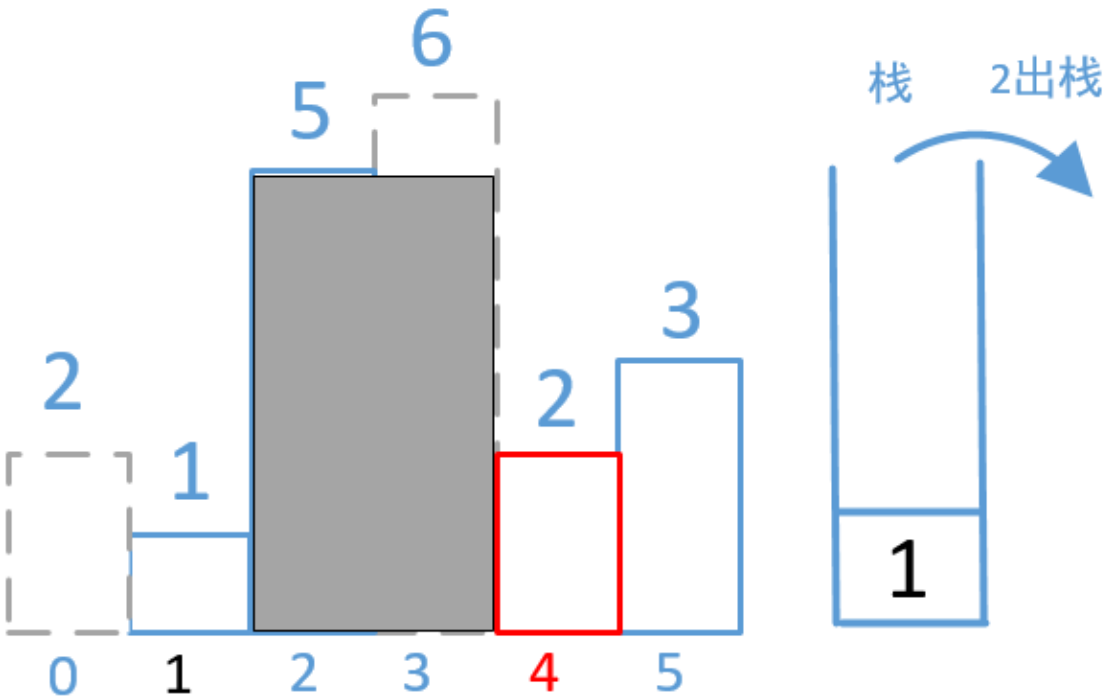


第6步



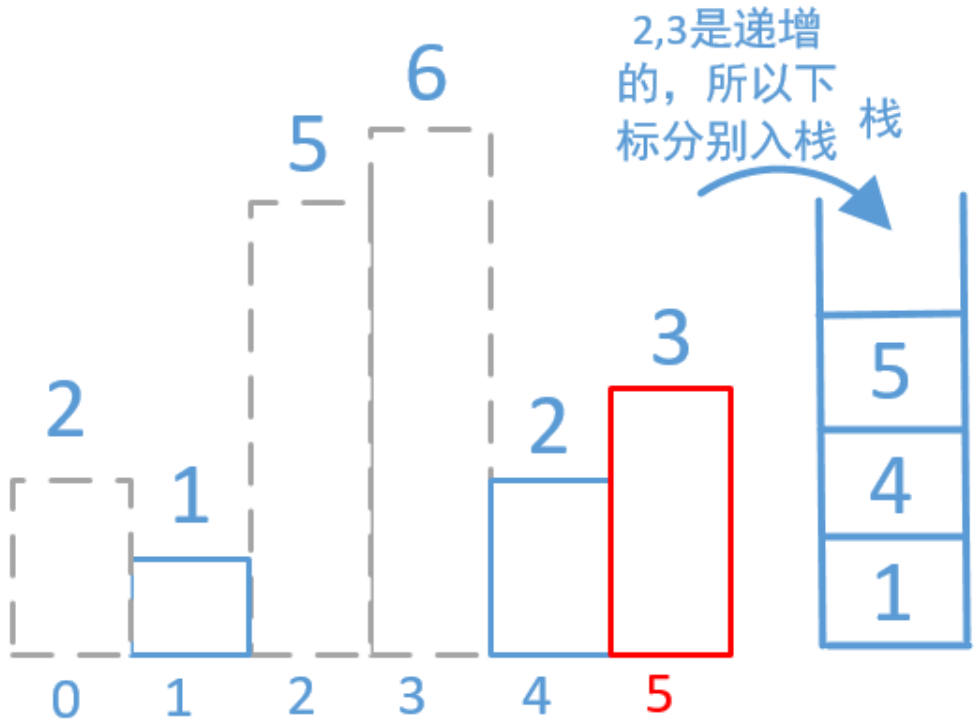
由于4号柱子高度是2比栈顶元素3号柱子6小，所以3出栈，最大面积是6*(4-1-2)=6，6表示的是出栈的下标所对应的柱子高度，4是遍历到的柱子的下标，2是栈顶元素

第7步



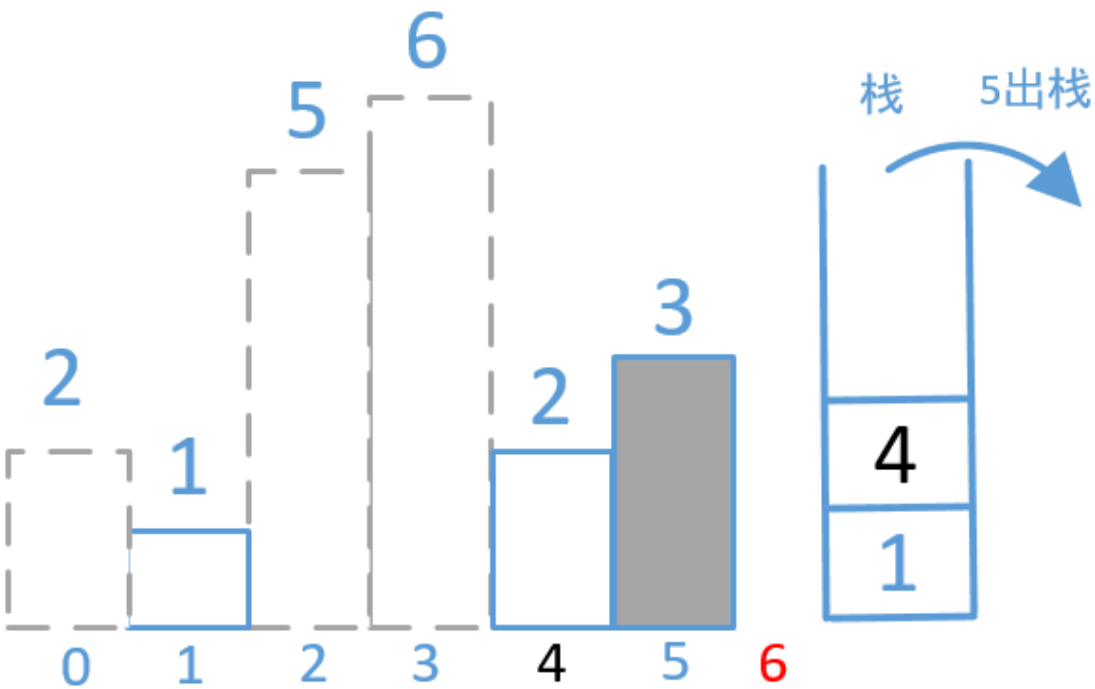
由于4号柱子高度是2比栈顶元素2号柱子5小，所以栈顶元素2出栈，最大面积是5*(4-1-1)=10，5是出栈的下标所对应的柱子高度，1就是栈顶元素（也就是左边比它小的柱子的下标）。在通俗一点就是，2是右边比5小的，1是左边比5小的，知道左右两边比它小的，就可以确定高度是5的矩形的宽度了。

第8，9步



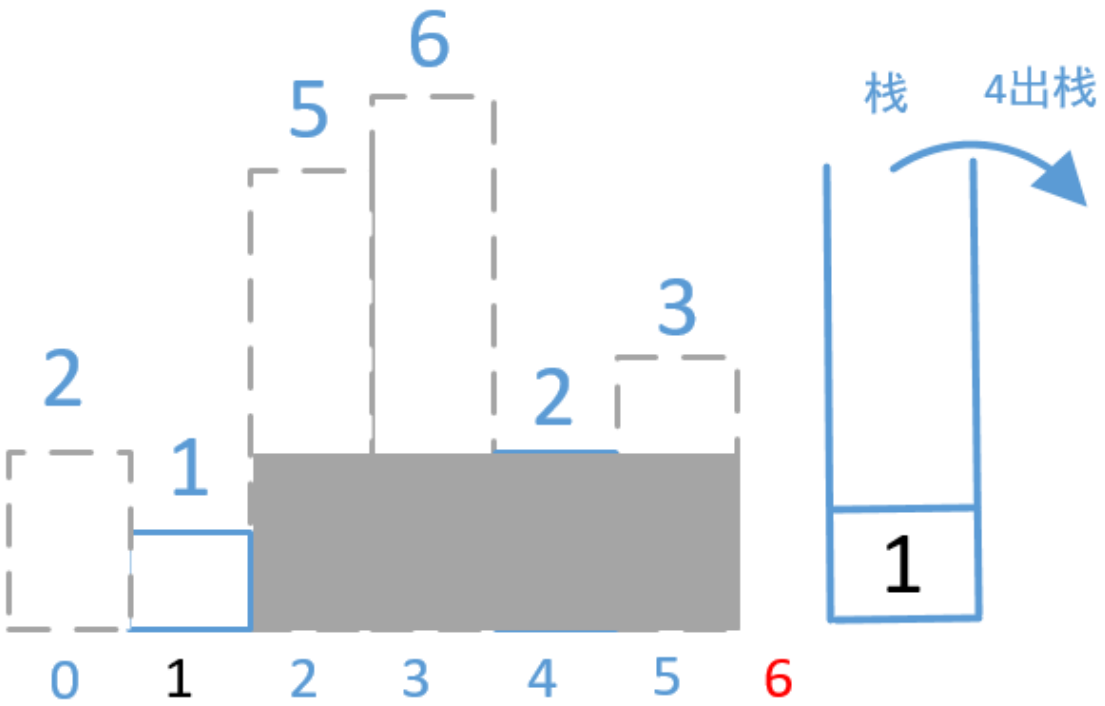
这个时候就头疼了，因为5号柱子后面没值了，但栈中的值还没有使用完。所以这个时候我们默认会在柱子的最后一列添加一个高度为0的柱子，这个时候我们就可以进行后面的计算了

第10步



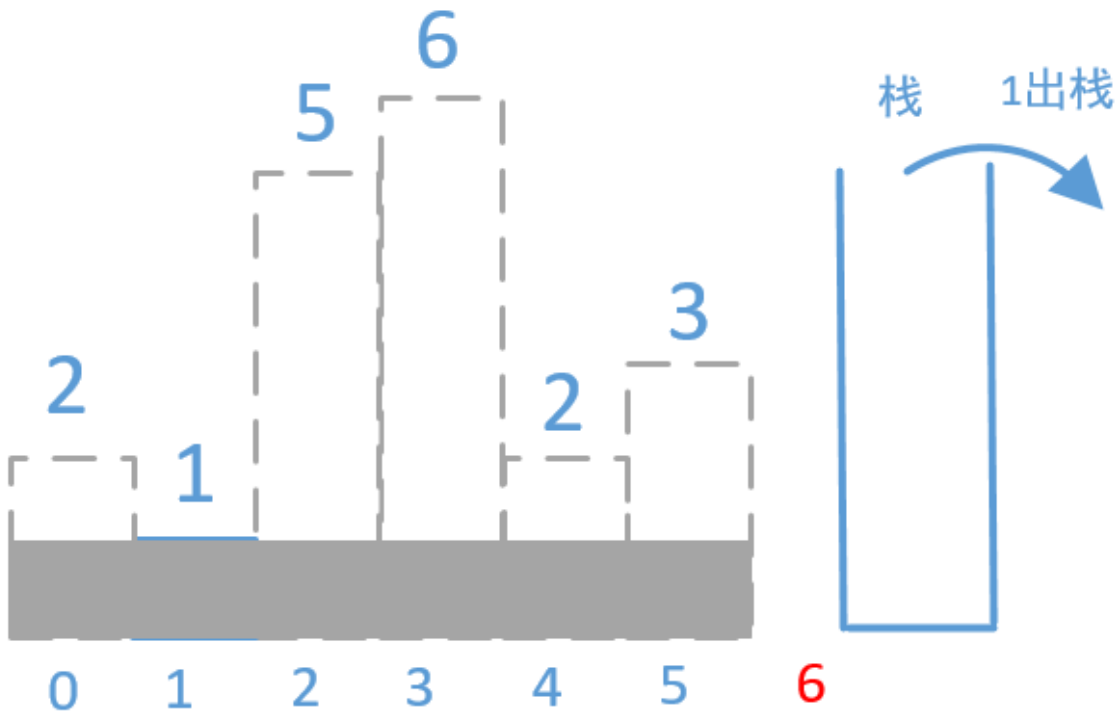
由于6号柱子0比栈顶元素5号柱子3小，所以5出栈，最大面积是3*(6-1-4)=3，黑色4是栈顶元素

第11步



由于6号柱子0比栈顶元素4号柱子2小，所以4出栈，最大面积是2*(6-1-1)=8，黑色1是栈顶元素

第12步



由于6号柱子0比栈顶元素1号柱子1小，所以1出栈，最大面积是1*6=6，这个时候栈是空的，当栈为空的时候，矩形宽就是遍历到的柱子下标

通过上面的计算我们可以找到最大面积是10

03 使用栈求解代码

```java
public int largestRectangleArea(int[] heights) {
    int length = heights.length;
    Stack<Integer> stack = new Stack<>();
    int maxArea = 0;
    for (int i = 0; i <= length; i++) {
        int h = (i == length ? 0 : heights[i]);
        //如果栈是空的，或者当前柱子的高度大于等于栈顶元素所对应柱子的高度，
        //直接把当前元素压栈
        if (stack.isEmpty() || h >= heights[stack.peek()]) {
            stack.push(i);
        } else {
            int top = stack.pop();
            int area = heights[top] * (stack.isEmpty() ? i : i - 1 - stack.peek());
            maxArea = Math.max(maxArea, area);
            i--;
        }
    }
    return maxArea;
}
```

这题标注是**难**，确实有一定的难度，如果上面图看懂了，上面代码也就不难理解了。其实我们还可以换种思路，在柱状图的最左边和最右边分别增加一个高度为0的柱子，这样代码写起来也比较容易理解，图就不再画了，代码中有详细的注释，我们直接看代码

```java
public int largestRectangleArea(int[] heights) {
    //申请一个比heights长度大2的临时数组
    int[] tmp = new int[heights.length + 2];
    //把数组heights的值复制到数组tmp中，并且tmpd第一个元素
    // 和最后一个元素都是0，表示高度为0的柱子
    System.arraycopy(heights, 0, tmp, 1, heights.length);
    Stack<Integer> stack = new Stack<>();//栈
    int maxArea = 0;
    for (int i = 0; i < tmp.length; i++) {
        //如果当前值tem[i]比栈顶元素对应的柱子高度小，说明栈顶元素的柱子遇到
        // 了右边比它小的柱子。那么他左边比它小的就是栈顶元素所对应的柱子高度
        // (因为栈中元素从栈底到栈顶对应柱子的高度是递增的)，知道左右两边比
        // 它小的就可以确定矩形的面积了，但这个矩形不一定是最大的，所以我们要保存下来
        while (!stack.isEmpty() && tmp[i] < tmp[stack.peek()]) {
            int h = tmp[stack.pop()];
            //计算矩形的面积
            int area = (i - 1 - stack.peek()) * h;
            //哪个大留哪个
            maxArea = Math.max(maxArea, area);
        }
        //注意这里入栈的是柱子的下标，不是柱子的高度
        stack.push(i);
    }
    return maxArea;
}
```

04 通过两边的临界值求解

根据上面的分析，我们知道对于第i根柱子所围成的最大矩形是

s=(right-left-1)*height[i]
其中right是右边比它小的柱子的下标，left是左边比它小的柱子的下标，height[i]是当前柱子的高度。

如果我们知道每根柱子左右两边比它小的值，我们就可以求出最大面积

```
    int maxArea = 0;
    for (int i = 0; i < height.length; i++) {
      maxArea = Math.max(maxArea, height[i] * (rightLess[i] - leftLess[i] - 1));
    }
```

但问题是我们怎么求出左右两边比它小的值呢？比如我们想求左边比它小的值，我们可以这样来计算

```
    for (int i = 1; i < height.length; i++) {
      int p = i - 1;
      while (p >= 0 && height[p] >= height[i]) {
        p--;
      }
      leftLess[i] = p;
    }
```

代码很简单，就是从他的左边挨着的那个一直往左找，直到找到为止。如果没找到p就会为-1，比如一直递减的柱子每一个p都是-1，-1符合上面的公式。同理右边的也一样。

但我们看到上面的查找效率真的不是很高，实际上代码我们还可以再优化一下，如果左边的柱子i比当前柱子k高，那么柱子i左边比柱子i高的肯定也比当前柱子k高，这种我们就不需要在找了，我们要找柱子i左边比柱子i矮的柱子再和当前柱子k对比，我们来看下

```
    for (int i = 1; i < height.length; i++) {
      int p = i - 1;
      while (p >= 0 && height[p] >= height[i]) {
        p = leftLess[p];
      }
      leftLess[i] = p;
    }
```

看明白了上面的分析，代码就容易多了，我们再来看下

```
  public static int largestRectangleArea(int[] height) {
    if (height == null || height.length == 0) {
      return 0;
    }
    //存放左边比它小的下标
    int[] leftLess = new int[height.length];
    //存放右边比它小的下标
    int[] rightLess = new int[height.length];
    rightLess[height.length - 1] = height.length;
    leftLess[0] = -1;
```

```
11
12      //计算每个柱子左边比它小的柱子的下标
13      for (int i = 1; i < height.length; i++) {
14          int p = i - 1;
15          while (p >= 0 && height[p] >= height[i]) {
16              p = leftLess[p];
17          }
18          leftLess[i] = p;
19      }
20      //计算每个柱子右边比它小的柱子的下标
21      for (int i = height.length - 2; i >= 0; i--) {
22          int p = i + 1;
23          while (p < height.length && height[p] >= height[i]) {
24              p = rightLess[p];
25          }
26          rightLess[i] = p;
27      }
28      int maxArea = 0;
29      //以每个柱子的高度为矩形的高，计算矩形的面积。
30      for (int i = 0; i < height.length; i++) {
31          maxArea = Math.max(maxArea, height[i] * (rightLess[i] - leftLess[i] - 1));
32      }
33      return maxArea;
34  }
```

05 总结

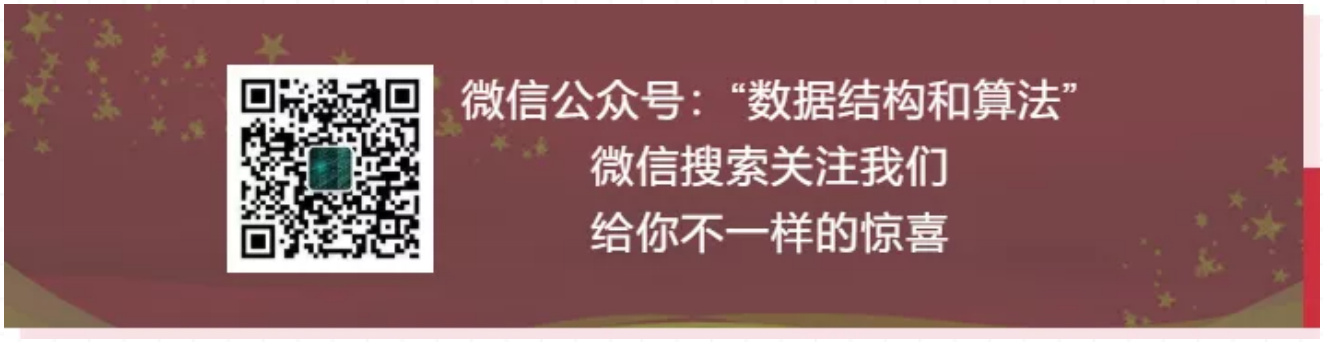
这题如果单从暴力破解的方式上来看不是很难，但我们都知道暴力二字是什么意思，在面试中暴力求解往往不占优势。如果不使用暴力破解这题还是有一定的难度的。

往期推荐

- 377，调整数组顺序使奇数位于偶数前面
- 376，动态规划之编辑距离

377，调整数组顺序使奇数位于偶数前面

原创 山大王wld 数据结构和算法 6月5日



The human voice can never reach the distance that is covered by the still small voice of conscience.

良心之声寂静微小，但它传递的距离是人声永远达不到的。

问题描述

给一个整数数组，让它的奇数和偶数分开并且奇数在数组的前面，偶数在数组的后面。

问题分析

01 临时数组求解

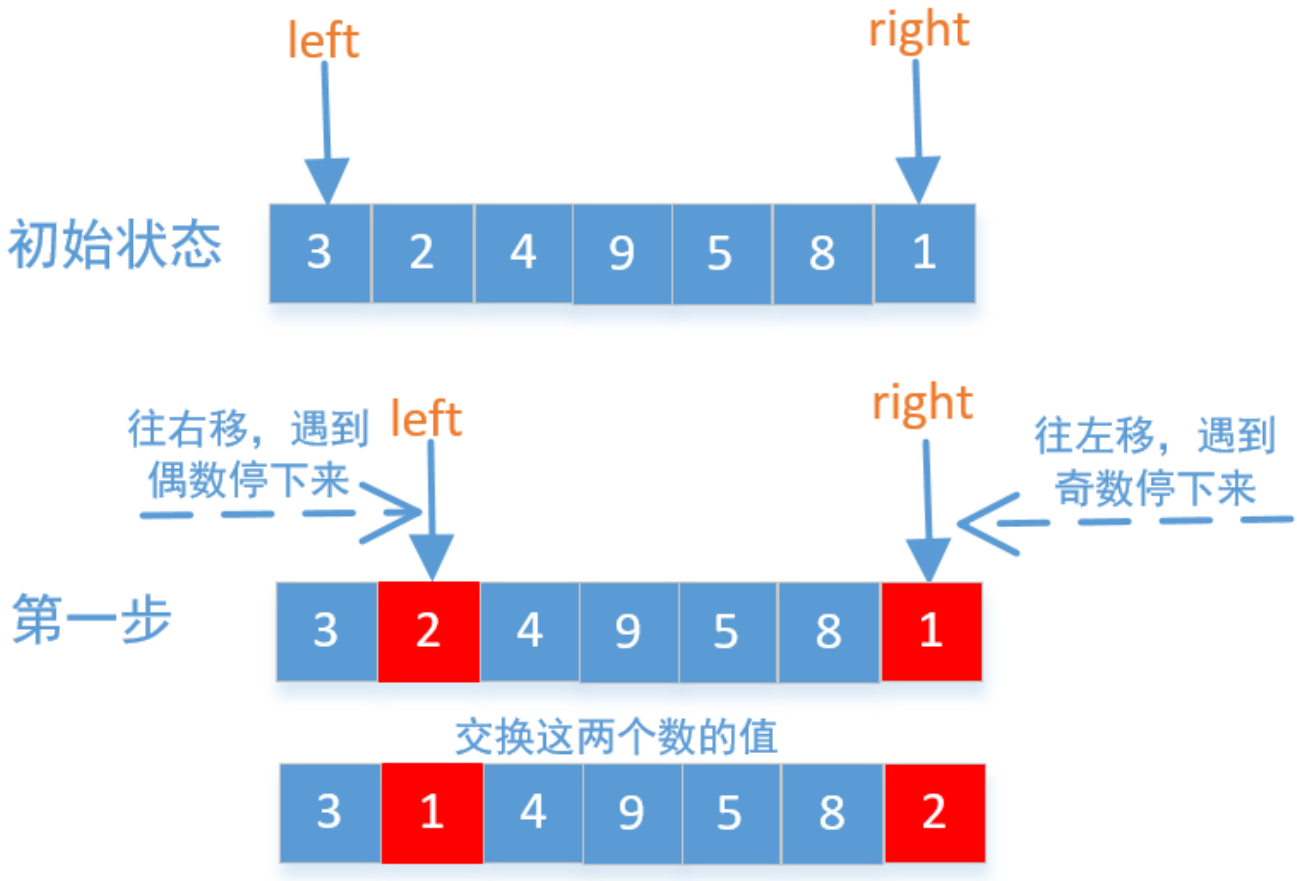
这题没什么难度，首先最容易想到的是申请一个同样大小的临时数组，把原数组的值放到临时数组中，奇数从前面放，偶数从后面放，我们来看下代码

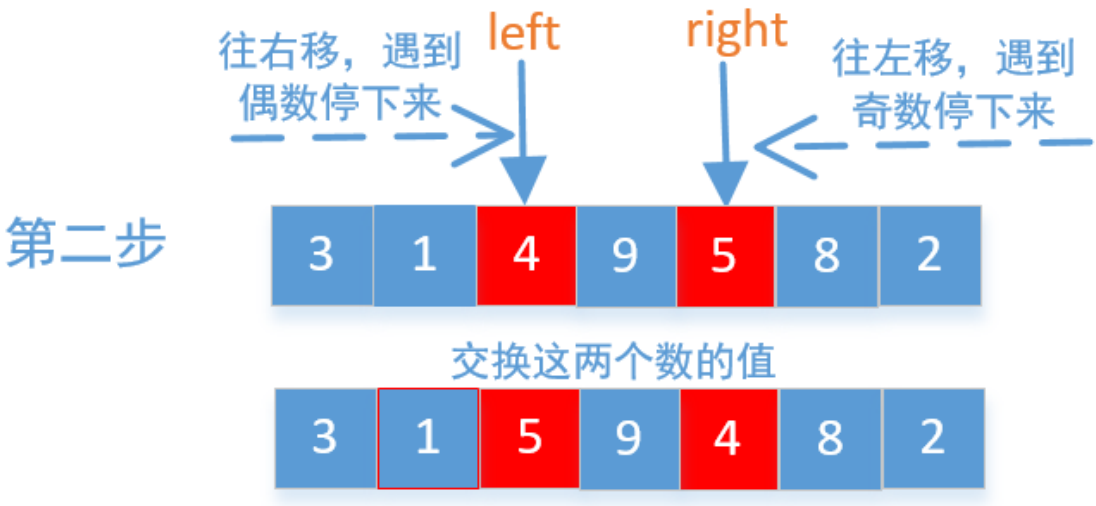
```
1   public int[] exchange(int[] nums) {
2     if (nums == null || nums.length == 0)
```

```
        return nums;
    int left = 0;
    int right = nums.length - 1;
    int temp[] = new int[nums.length];
    for (int i = 0; i < nums.length; i++) {
      if ((nums[i] & 1) == 0) {
        //偶数从后面放
        temp[right--] = nums[i];
      } else {
        //奇数从前面放
        temp[left++] = nums[i];
      }
    }
    return temp;
  }
```

02 双指针求解

我们可以使用两个指针left和right。left从左边开始扫描，如果是奇数就往右走，如果遇到偶数就停下来（此时left指向的是偶数），right从右边开始扫描，如果是偶数就往左走，如果是奇数就停下来（此时right指向的是奇数），交换left和right指向的值。继续循环，直到left==right为止。我们就以数组[3，2，4，9，5，8，1]为例来画个图看一下





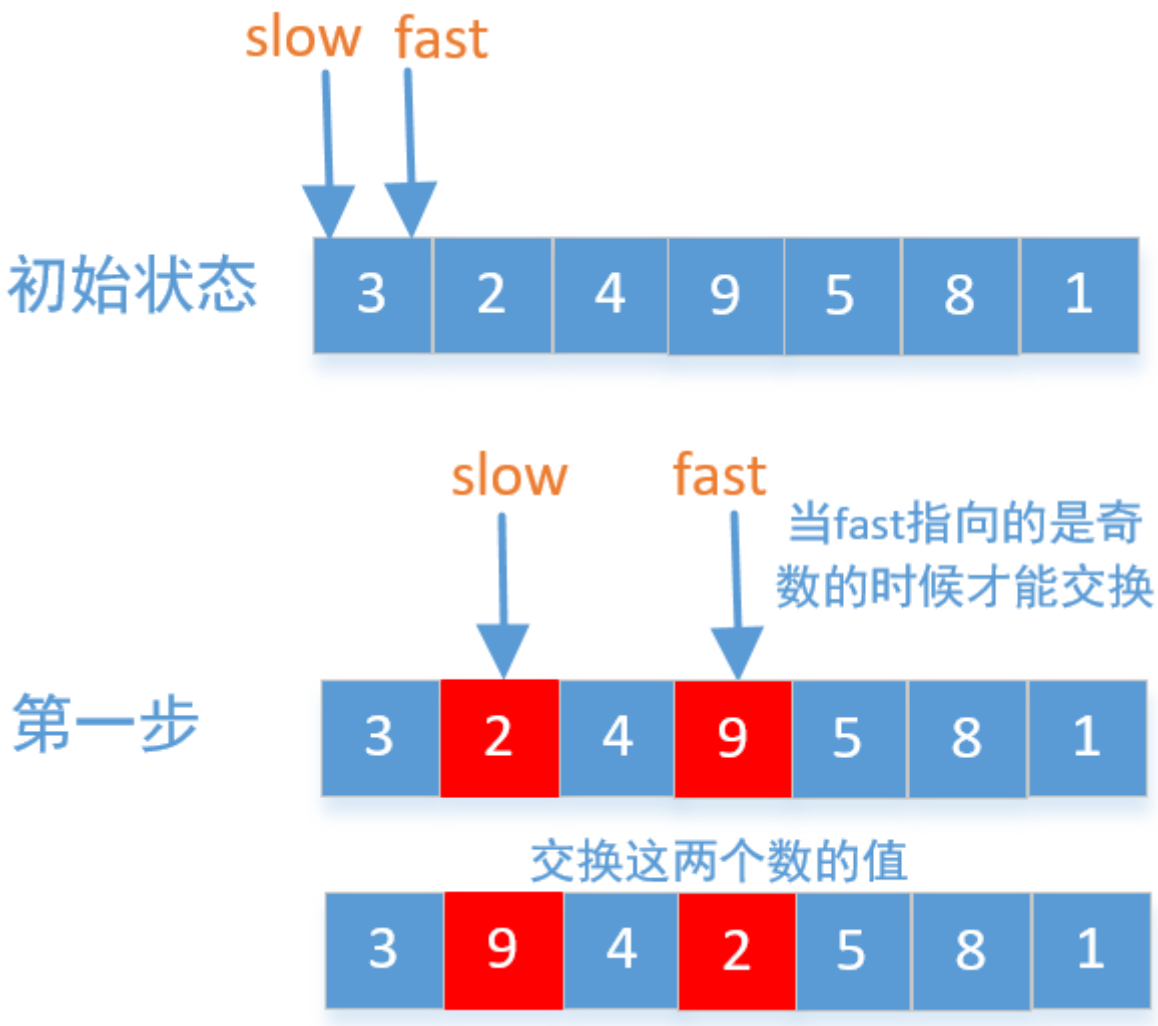


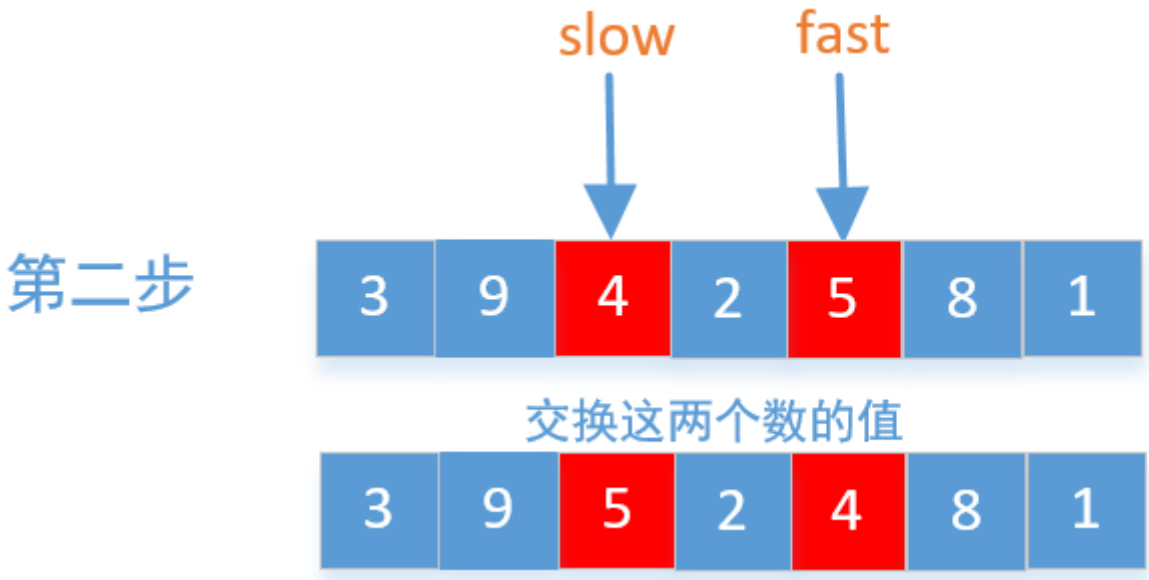

```java
public static int[] exchange(int[] nums) {
  if (nums == null || nums.length == 0)
    return nums;
  int left = 0;
  int right = nums.length - 1;
  while (left < right) {
    //如果是奇数，就往后挪，直到遇到偶数为止
    while (left < right && (nums[left] & 1) == 1) {
      left++;
    }
    //如果是偶数，就往前挪，直到遇到奇数为止
    while (left < right && (nums[right] & 1) == 0) {
      right--;
    }
    //交换两个值
    if (left < right) {
      nums[left] ^= nums[right];
      nums[right] ^= nums[left];
      nums[left] ^= nums[right];
    }
  }
  return nums;
}
```

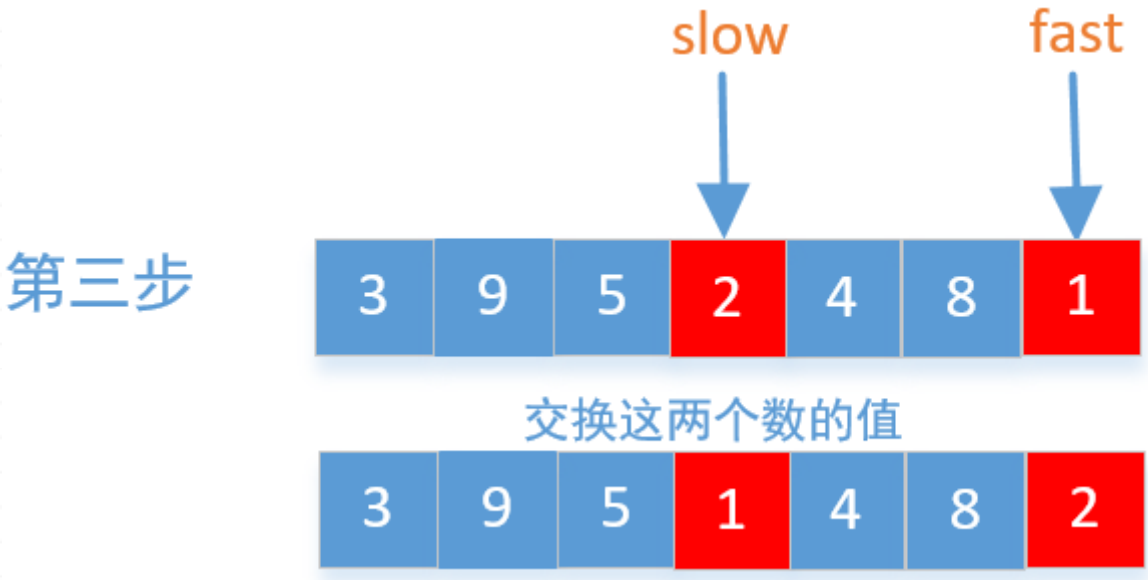
代码16到20行是交换两个数字的值，交换两个数的值有多种方式，这里选择的是通过异或来交换，如果看不明白可以看一下下面往期推荐中的第357题。

03 快慢指针求解

第三种方式使用的是快慢指针，和上一种解决方式有一点区别，上一种是一前一后扫描。我们这里使用的快慢指针都是从头开始扫描。我们使用两个指针，一个快指针fast，一个慢指针slow。慢指针slow存放下一个奇数应该存放的位置，快指针fast往前搜索奇数，搜索到之后然后就和slow指向的值交换，我们还以上面的数据为例画个图来分析下








```java
public static int[] exchange(int[] nums) {
  int slow = 0, fast = 0;
  while (fast < nums.length) {
    if ((nums[fast] & 1) == 1) {//奇数
      if (slow != fast) {
        nums[slow] ^= nums[fast];
        nums[fast] ^= nums[slow];
        nums[slow] ^= nums[fast];
      }
      slow++;
    }
    fast++;
  }
  return nums;
}
```

往期推荐

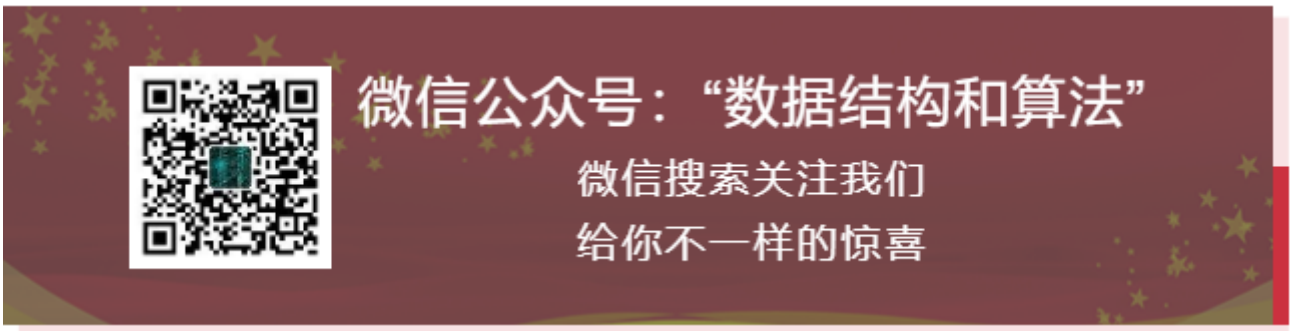
- 364，位1的个数系列（一）
- 357，交换两个数字的值

369，整数替换

原创 山大王wld 数据结构和算法 5月26日

收录于话题
#算法图文分析

96个 >



题目：

给定一个正整数 n，你可以做如下操作：
1. 如果 n 是偶数，则用 n / 2替换 n。
2. 如果 n 是奇数，则可以用 n + 1或n - 1替换 n。
n 变为 1 所需的最小替换次数是多少?

示例 1:

输入:
8

输出:
3

解释:
8 -> 4 -> 2 -> 1

示例 2:

输入:
7

输出:
4

解释:
7 -> 8 -> 4 -> 2 -> 1
或
7 -> 6 -> 3 -> 2 -> 1

01 答案

```
public int integerReplacement(int n) {
  if (n == Integer.MAX_VALUE)
    return 32;
  if (n <= 3)
    return n - 1;
  if (n % 2 == 0)
    return integerReplacement(n / 2) + 1;
  else
    return Math.min(integerReplacement(n - 1), integerReplacement(n + 1)) + 1;
}
```

解析：

使用递归的方式很容易理解，当n是偶数的时候非常简单，我们只需要让n/2代替n即可。当n是奇数的时候，我们取n-1，和n+1计算次数的最小值即可。

其实这道题我们还可以换种思路。当n是奇数的时候，比如n=2k+1；无论是加1还是减1，结果都会是偶数，这个偶数有可能是4的倍数，有可能只是2的倍数(比如6,10等)。我们为了减少计算次数要尽可能多的往4的倍数上靠。所以当n%4=3的时候我们让n加1，当n%4=1的时候我们让n减1。当n等于3的时候是个例外，因为
3→2→1要比3→4→2→1替换次数少。所以我们计算的时候要把n=3的情况单独处理，我们来看下代码

02 另一种思路的递归写法

```
public int integerReplacement(int n) {
  if (n == Integer.MAX_VALUE)
    return 32;
  if (n <= 3)
    return n - 1;
  if (n % 2 == 0)
    return integerReplacement(n / 2) + 1;
  else
```

```
        return (n & 2) == 0 ? integerReplacement(n - 1) + 1 : integerReplacement(n + 1) + 1;
    }
```

03 另一种思路的非递归写法

```
public int integerReplacement(int n) {
    int count = 0;
    while (n > 1) {
        if ((n & 1) == 0) {//是偶数
            n /= 2;
        } else {//是奇数
            if (((n + 1) & 3) == 0 && n != 3) {//对3求余为0
                n = n / 2 + 1;
                count++;
            } else {
                n--;
            }
        }
        count++;
    }
    return count;
}
```



上面的3种实现方式其实都很简单，我们看到在判断n%4=3的时候有多种实现方式，我们还可以来列举一下（下面的n首先是奇数，判断才没问题），如果判断结果为true，那么n对4求余的结果就是3。

1， n%4==3;

2， (n+1)&3==0;

3， (n&2)!=0;

4， Integer.bitCount(n + 1) <= Integer.bitCount(n - 1)（Integer.bitCount是判断二进制中1的个数，也可以参照前面讲的364，位1的个数系列（一））

5， ((n >>1) & 1) != 0

365，消除游戏

原创 山大王wld 数据结构和算法 5月20日

收录于话题
#算法图文分析
96个 >

给定一个从1 到 n 排序的整数列表。
首先，从左到右，从第一个数字开始，每隔一个数字进行删除，直到列表的末尾。

第二步，在剩下的数字中，从右到左，从倒数第一个数字开始，每隔一个数字进行删除，直到列表开头。

我们不断重复这两步，从左到右和从右到左交替进行，直到只剩下一个数字。
返回长度为 n 的列表中，最后剩下的数字。

示例：

```
输入：
n = 9,
1 2 3 4 5 6 7 8 9 （1,3,5,7,9被删除）
2 4 6 8 （8,4被删除）
2 6 （2被删除）
6 （剩余6）

输出：
6
```

答案：

```java
public int lastRemaining(int n) {
    boolean left = true;
    int remaining = n;
    int step = 1;
    int head = 1;
    while (remaining > 1) {
        if (left || ((remaining & 1) == 1)) {
            head = head + step;
        }
        remaining = remaining >> 1;
        step = step << 1;
        left = !left;
    }
    return head;
}
```

解析：

题描述的很清晰，就是先从左往右每隔一个就删除一个数字，然后再从右往左每隔一个删除一个数字……一直这样循环下去，直到最后剩下一个数字为止。

在计算机编程中有个非常著名的算法题就是“约瑟夫环问题”，也称“丢手绢问题”，如果对约瑟夫环问题比较熟悉的话，那么今天的这道题也就很容易理解了。如果不熟悉的话也没关系，我们今天就详细分析一下这道题。关于约瑟夫环问题不在今天所讲的范围之内，后续有时间我们在单独讲解。

这题如果使用双向链表或者是双端队列很好解决，因为双向链表既可以从前往后删除也可以从后往前删除，当然这两种方式都需要先初始化，今天我们讲的这种方式是既没有使用链表也没有使用数组。

我们来看下上面的代码，直接看可能不太直观，我们可以把n想象成一个长度为n的数组，数组的元素是1,2,3,4,5.....n，我们只需要记录下每次删除一遍之后数组的第一个元素即可，当remaining==1的时候就会退出while循环，最后返回数组的仅有的一个元素即可（这只是我们的想象，实际上操作的并不是数组，也没有删除，只是记录，但原理都类似）。

```
boolean left = true;
```

代码left判断是否是从左往右删除，如果为true表示的是从左往右删除，如果为false表示的是从右往左删除。

```
int remaining = n;
int step = 1;
int head = 1;
```

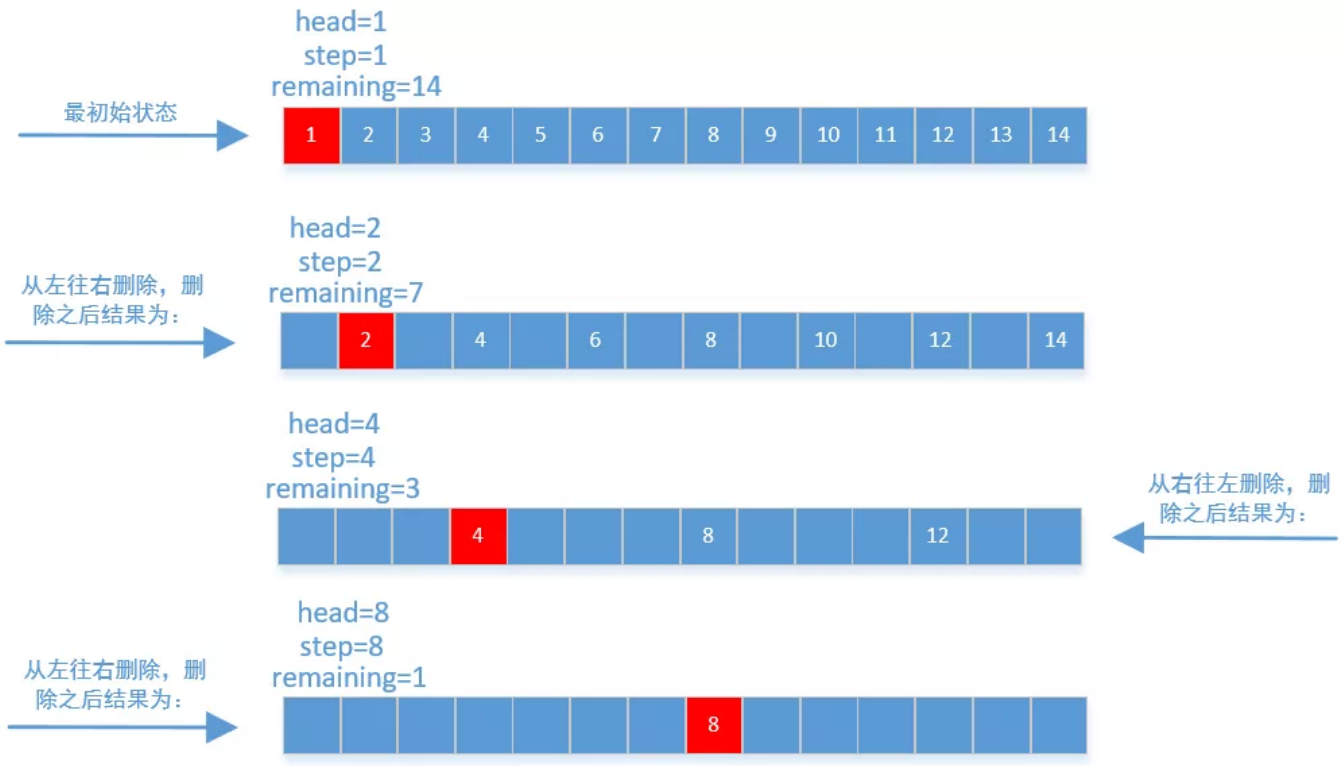
代码remaining表示剩余的个数。step表示每次删除的间隔的值，不是间隔的数量，比如1,2,3,4,5,6,7,8。第一次从左往右删除的时候间隔值是1，删除之后结果为2,4,6,8。第二次从右往左删除间隔值就变为2了，删除之后结果是2,6。然后第3次就变成4了。head表示的是记录的剩余数字从左边数第一个的值。

```
1     while (remaining > 1) {
2       if (left || ((remaining & 1) == 1)) {
3         head = head + step;
4       }
5       remaining = remaining >> 1;
6       step = step << 1;
7       left = !left;
8     }
```

第5-7行代码很好理解，remaining表示的是剩余个数，每次删除的时候都会剩余一半，所以除以2，也可以表示为往右移一位。step上面说了表示的是间隔值，每次循环之后都会扩大一倍，left就不在说了，一次往左一次往右.....一种这样循环。

我们主要来看下第2-4行代码，如果是从这边循环，那么第一个肯定是会被删除的，第二个会成为head，而第二个值就是head+step；如果从右边开始循环，如果数组的长度是奇数，那么第一个元素head也是要被删除的，所以head值也要更新，代码remaining&1==1判断remaining是否是奇数。

我们以n=14为例，画个图来看下会更直观一些



上面代码变量比较多，实际上我们还可以改的更简洁一些

```
public int lastRemaining(int n) {
  int first = 1;
  for (int step = 0; n != 1; n = n >> 1, step++) {
    if (step % 2 == 0 || n % 2 == 1)
      first += 1 << step;
  }
  return first;
}
```

注意这里的step不是删除的间隔值，他是表示的是每删除一遍就会加1，比如最开始从左往右删除的时候step是0，然后再从右往左删除的时候是1，然后再从左往右删除的时候是2，然后再从右往左删除的时候是3……，一直这样累加。代码很好理解，就不在过多解释。

下面我们再来换种思路想一下

1，当我们从左往右消除的时候，比如**[1,2,3,4]**第一次从左往右消除的时候结果是**[2,4]**，也就是2*[1,2]

或者**[1,2,3,4,5]**第一次从左往右消除的时候结果也是**[2,4]**，也就是2*[1,2]

所以我们只需要计算数组前面一半的结果然后再乘以2即可。

2，当我们从右往左消除的时候，如果数组是偶数，比如**[1,2,3,4,5,6]**消除的结果是**[1,3,5]**，也就是2*[1,2,3]-1，如果数组是奇数的话，比如**[1,2,3,4,5,6,7]**消除的结果是**[2,4,6]**，也就是2*[1,2,3]。所以明白了这点，代码就很容易想到了

```
public int lastRemaining(int n) {
  return leftToRight(n);
}

private static int leftToRight(int n) {
  if (n <= 2)
    return n;
  return 2 * rightToLeft(n / 2);
}
```

```
10
11    private static int rightToLeft(int n) {
12      if (n <= 2)
13        return 1;
14      if (n % 2 == 1)
15        return 2 * leftToRight(n / 2);
16      return 2 * leftToRight(n / 2) - 1;
17    }
```

我们再来思考一个问题，可以找一下规律

1，当n个数的时候，假设我们从左往右执行，剩下的数字记为f1(n)（从数组[1,2,.....n]开始），从右往左执行，剩下的数字是f2(n)（从数组[n,n-1,.....1]开始）。

2，如果我们记f1(n)在数组[1,2,.....n]中的下标为k，那么f2(n)在数组中[n,n-1,.....1]的下标也一定是k。所以我们可以得到f1(n)+f2(n)=n+1。

3，对于n个元素，执行一次从左往右之后，剩下的[2,4,.....n/2]就应该从右往左了，我们记他执行，剩下的数字是f3(n/2)，所以我们可以得到f1(n)=f3(n/2)，f3(n/2)=2*f2(n/2)；

4，根据上面的3个公式

(1)：f1(n)+f2(n)=n+1

(2)：f1(n)=f3(n/2)

(3)：f3(n/2)=2*f2(n/2)

我们可以得出f1(n)=2*(n/2+1-f1(n/2))；并且当n等于1的时候结果就是1，所以代码如下，非常简单

```
1   public int lastRemaining(int n) {
2     return n == 1 ? 1 : 2 * (1 + n / 2 - lastRemaining(n / 2));
3   }
```

对于这道题的理解我们还可以来举个例子，比如[1,2,3.....n]，如果从左开始结果是k，那么从右开始结果就是n+1-k。比如[1,2,3,4,5,6,7,8,9,10]第一遍从左到右运算之后是[2,4,6,8,10]，假如[1,2,3,4,5]从左到右的结果是f(5)，那么他从右到左的结果就是5+1-f(5)，也就是6-f(5)，所以[2,4,6,8,10]从右到左的结果就是2*(6-f(5))，所以我们可以得出f(10)=2*(6-f(5))，所以递推公式就是f(n)=2*(n/2+1-f(n/2))。

我们还可以把上面递归的代码改为非递归，这个稍微有一定的难度

```
1   public int lastRemaining(int n) {
2     Stack<Integer> stack = new Stack<>();
3     while (n > 1) {
4       n >>= 1;
5       stack.push(n);
6     }
7     int result = 1;
8     while (!stack.isEmpty()) {
9       result = (1 + stack.pop() - result) << 1;
10    }
11    return result;
12  }
```

下面再来思考一下，看能不能再优化一下，我们让left(n)=left[1,2,3,.....n]表示从左往右执行之后，剩下的数字，right(n)=right[1,2,3,.....,n]表示从右往左执行之后，剩下的数字，所以我们可以得出一个结论

1，left(1)=right(1)=1;

2，left(2k)=left[1,2,3,.....2k]=right[2,4,6,.....2k]=2*right(k);

3，left(2k+1)=left[1,2,3,.....2k,2k+1]=right[2,4,6,.....2k]=2*right(k);

4，right(2k)=right[1,2,3,.....2k]=left[1,3,5,.....2k-1]=left[2,4,6,.....2k]-1=2*left(k)-1

5，right(2k+1)=right[1,2,3,.....2k,2k+1]=left[2,4,6,.....2k]=2*left(k)。

6，left(4k)=left(4k+1)=4*left(k)-2;

7，left(4k+2)=left(4k+3)=4*left(k);

搞懂了上面的规律，代码就呼之欲出了，下面我们来看下代码

```
1   public int lastRemaining(int n) {
2     if (n < 4)
3       return (n == 1) ? 1 : 2;
4     return (lastRemaining(n / 4) * 4) - (~n & 2);
5   }
```

我们再来看最后一种解法，也是一行代码搞定

```
1   public int lastRemaining(int n) {
2     return ((Integer.highestOneBit(n) - 1) & (n | 0x55555555)) + 1;
3   }
```

如果对约瑟夫环问题比较熟练的话，那么这种解法就比较好理解了，其实他就是约瑟夫环中k=2的一个问题。

360，等差数列划分

原创 山大王wld 数据结构和算法 5月12日

收录于话题
#算法图文分析

96个 >

如果一个数列至少有三个元素，并且任意两个相邻元素之差相同，则称该数列为等差数列。

例如，以下数列为等差数列:

```
1, 3, 5, 7, 9
7, 7, 7, 7
3, -1, -5, -9
```

以下数列不是等差数列。

```
1, 1, 2, 5, 7
```

求函数中返回数组 A 中所有为等差数组的子数组个数。

示例:

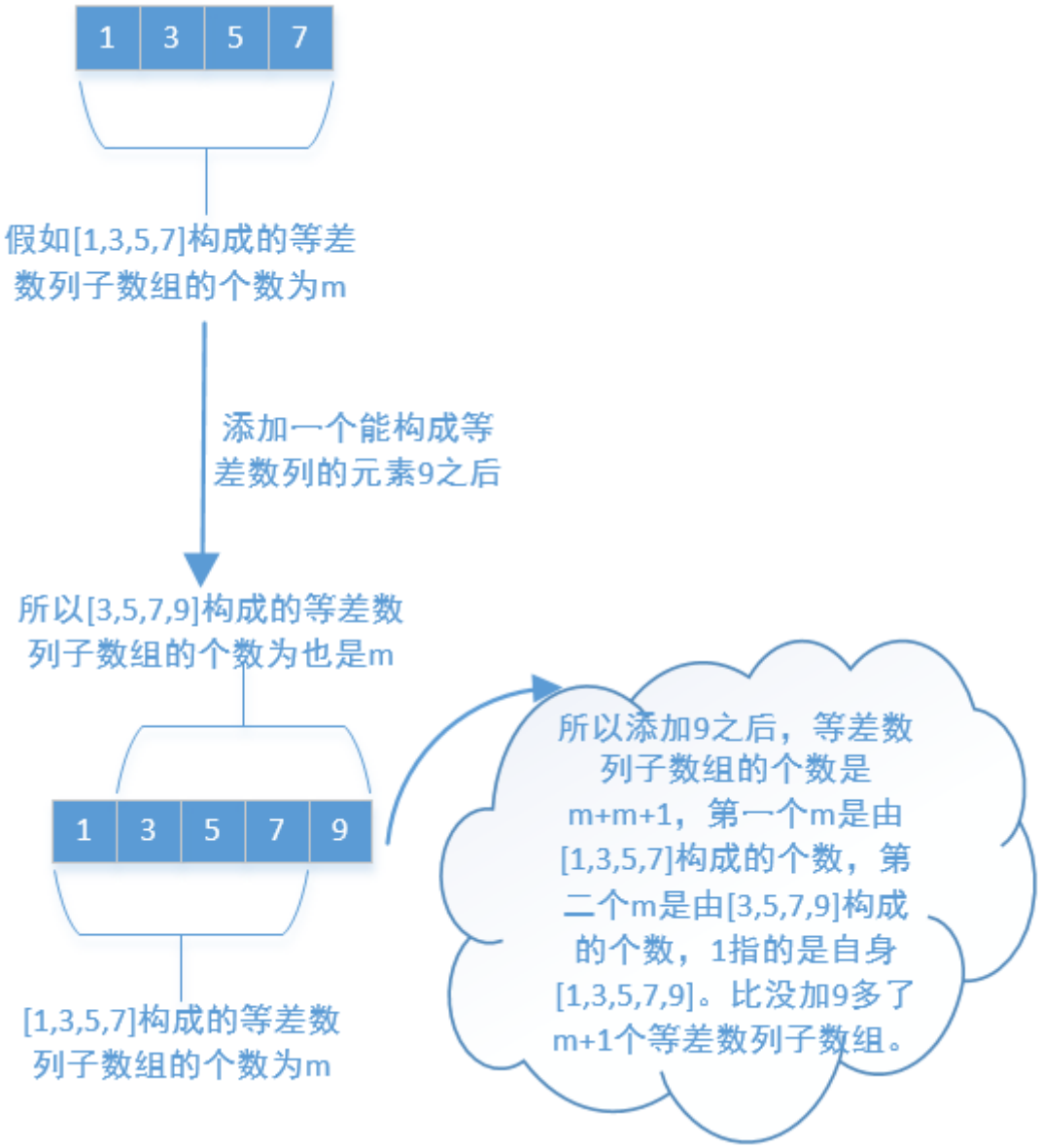
```
A = [1, 2, 3, 4]

返回: 3,
A 中有三个子等差数组: [1, 2, 3], [2, 3, 4] 以及自身 [1, 2, 3, 4]。
```

答案:

```
1   public int numberOfArithmeticSlices(int[] A) {
2     int curr = 0, sum = 0;
3     for (int i = 2; i < A.length; i++)
4       if (A[i] - A[i - 1] == A[i - 1] - A[i - 2]) {
5         curr += 1;
6         sum += curr;
7       } else {
8         curr = 0;
9       }
10    return sum;
11  }
```

解析:

要想构成等差数列，至少要3个元素，所以i是从2(也就是第三个元素)开始判断，curr表示的是已经构成的等差数列的子数组的个数，如果不能构成等差数列就让curr等于0，就不再计入总数sum了。如果能构成等差数列就让curr加1，然后再加入到总数sum中，这里curr为什么是要加1，我们画个图来分析一下



或者代码也可以更简洁一些，如下

```
public int numberOfArithmeticSlices(int[] A) {
  int curr = 0, sum = 0;
  for (int i = 2; i < A.length; i++) {
    curr = (A[i] - A[i - 1] == A[i - 1] - A[i - 2]) ? curr + 1 : 0;
    sum += curr;
  }
  return sum;
}
```

写法上虽然有点差别，但思路其实都是一样的，下面我们再来把它改为递归的写法

```
public int numberOfArithmeticSlices(int[] A) {
  return slices(A, A.length - 1);
}

public int slices(int[] A, int i) {
  if (i < 2)
    return 0;
  if (A[i] - A[i - 1] == A[i - 1] - A[i - 2]) {
    return slices(A, i - 1) * 2 - slices(A, i - 2) + 1;
  } else {
    return slices(A, i - 1);
  }
}
```

这种执行效率太差，多次重复计算，很容易超时，我们还可以在优化一下

```
int sum = 0;
```

```java

  public int numberOfArithmeticSlices(int[] A) {
    slices(A, A.length - 1);
    return sum;
  }

  public int slices(int[] A, int i) {
    if (i < 2)
      return 0;
    int count = 0;
    if (A[i] - A[i - 1] == A[i - 1] - A[i - 2]) {
      count = 1 + slices(A, i - 1);
      sum += count;
    } else {
      slices(A, i - 1);
    }
    return count;
  }
```

358，移掉K位数字

原创 山大王wld 数据结构和算法 5月8日

收录于话题
#算法图文分析

96个 >

给定一个以字符串表示的非负整数 *num*，移除这个数中的 *k* 位数字，使得剩下的数字最小。

注意:

- num 的长度小于 10002 且 ≥ k。
- num 不会包含任何前导零。

示例 1 :

```
输入: num = "1432219", k = 3
输出: "1219"
解释: 移除掉三个数字 4, 3, 和 2 形成一个新的最小的数字 1219。
```

示例 2 :

```
输入: num = "10200", k = 1
输出: "200"
解释: 移掉首位的 1 剩下的数字为 200. 注意输出不能有任何前导零。
```

示例 **3 :**

```
输入: num = "10", k = 2
输出: "0"
解释: 从原数字移除所有的数字，剩余为空就是0。
```

答案:

```java
public String removeKdigits(String num, int k) {
  if (num.length() <= k)
    return "0";
  int digits = num.length() - k;
  char[] stk = new char[digits];
  int top = 0;
  for (int i = 0; i < num.length(); i++) {
    char c = num.charAt(i);
    while (top > 0 && stk[top - 1] > c && k > 0) {
      k--;
      top--;
    }
    if (top < digits) {
      stk[top++] = c;
    } else {
      k--;
    }
  }
  int idx = 0;
  while (idx < digits && stk[idx] == '0')
    idx++;
  return idx == digits ? "0" : new String(stk, idx, digits - idx);
}
```

解析：

我们可以先不用看上面的代码，等我们分析完之后再看，否则直接看容易懵。上面的题说的是从一个字符串num中移除k个数字，使剩下的数字最小。我们可以先从移除一个数字开始看



所以我们找出了一个规律就是，就是从左往右删除第一个降序开始的值。

比如第一个3156，31是降序，所以删除3；

比如第二个1563，63是降序，所以删除6；

比如第二个3756，75是降序，所以删除7。

如果没有降序的，比如1234，我们就删除最后一个，结果才能是最小。

我们再来举一个移除两个数字的例子



所以规律很容易发现了，就是从左往右删除开始降序的数字，直到删除k个为止。或者也可以这样理解，就是**每遍历一个数字就要判断他前面有没有比他大的数字，如果有就删除，直到累计删除k个为止。**

1，那么这时候问题就来了，如果遍历完了删除的还不到k个怎么办呢，就像上面说的1234没有降序的这种，我们就删除最后的几个数字，凑够k个为止。

2，这个时候还会有一个问题，比如2014，如果我们删除一个数字让剩下的数字最小，我们删除的是2，那么结果就会变成014，所以我们还要把前导0去掉。
所以原理就这么简单。再来看上面的代码是不是如醍醐灌顶般豁然开朗。

而上面的代码稍微有点不同的是，他没有在原始字符串上操作，而是使用了一个临时数组stk，他会把原始字符串逐个放到stk数组中，放的时候如果发现前面有比他大的，就把前面比他大的移除掉，第13行有个判断，如果个数达到了那么当前值就不在添加了，在下一轮的for循环中还会进行判断，第20行通过循环去掉前导0。

355，两数相加 II

原创 山大王wld 数据结构和算法 4月30日

收录于话题
#算法图文分析
96个 >

给定两个**非空**链表来代表两个非负整数。数字最高位位于链表开始位置。它们的每个节点只存储单个数字。将这两数相加会返回一个新的链表。

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

示例:

```
输入: (7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)
输出: 7 -> 8 -> 0 -> 7
```

结点类如下

```
public class ListNode {
  int val;
  ListNode next;

  ListNode(int x) {
    val = x;
  }
}
```

答案:

```
public ListNode addTwoNumbers(ListNode list1, ListNode list2) {
  Stack<Integer> s1 = new Stack<>();
  Stack<Integer> s2 = new Stack<>();
  while (list1 != null) {
    s1.push(list1.val);
    list1 = list1.next;
  }
  while (list2 != null) {
    s2.push(list2.val);
    list2 = list2.next;
  }
  int sum = 0;
  ListNode head = new ListNode(0);
  while (!s1.empty() || !s2.empty()) {
    if (!s1.empty())
      sum += s1.pop();
    if (!s2.empty())
      sum += s2.pop();
    head.val = sum % 10;
    ListNode node = new ListNode(sum / 10);
    node.next = head;
    head = node;
    sum /= 10;
  }
  return head.val == 0 ? head.next : head;
}
```

解析：

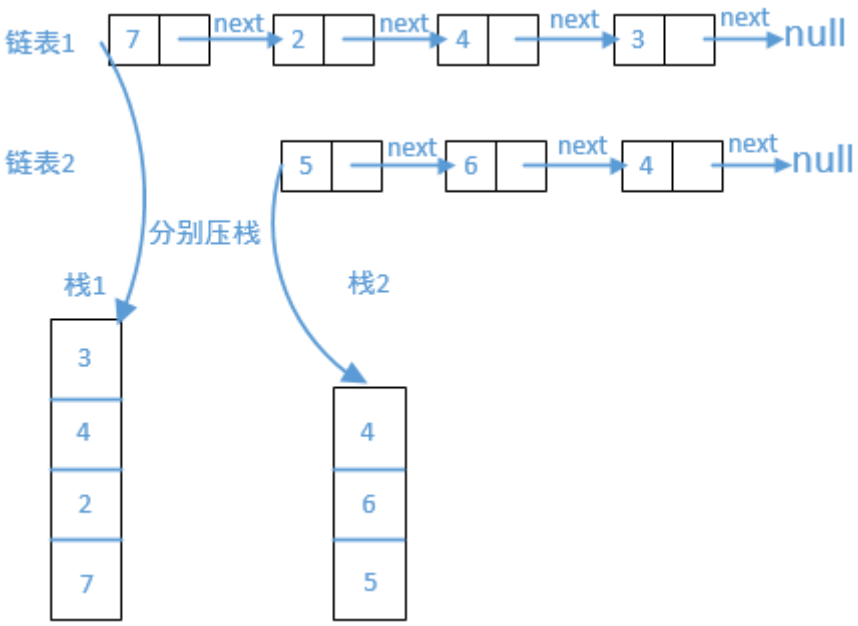
思路其实很简单，就是分别把两个链表的结点先存放到两个栈中，因为链表的最高位是在链表的最开始的位置，所以存放到栈中之后，栈底是高位，栈顶是个位（也是低位），然后两个栈中的元素再相加，因为栈是先进后出的，最先出来的肯定是个位（也是低位），最后出来的肯定是高位，也就是这两个数是从个位开始相加，这也符合加法的运算规律。

1，代码中第19行我们只保存相加的个位数，因为链表的每个结点只能保存一位数，如果有进位就会在下一步进行保存。

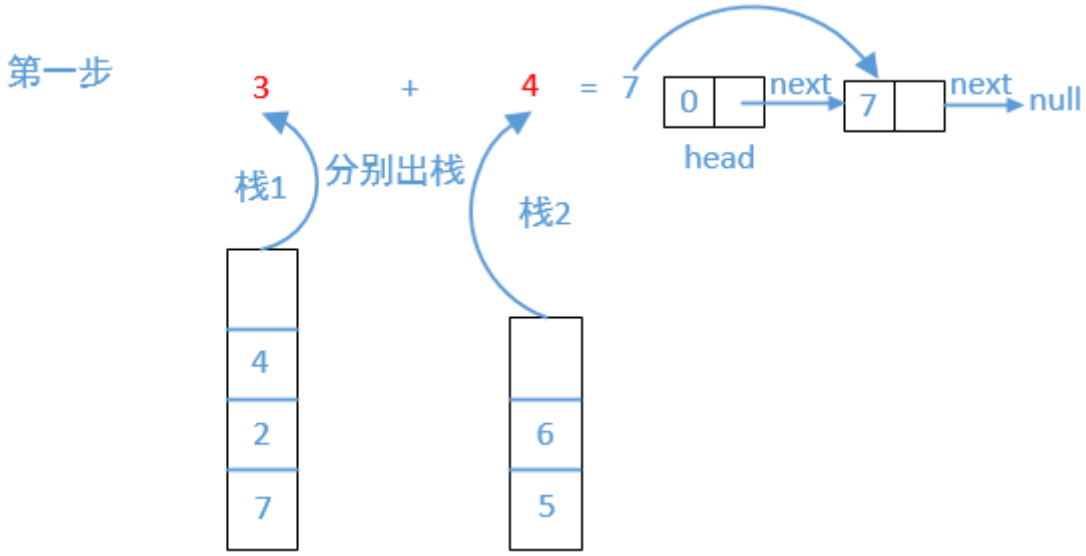
2，第20行在保留进位的值。其中第20到22行涉及到链表的插入，这个使用的是头插法，在链表节点的头部插入，比较简单，如果看不懂的，还可以看下前面刚讲的352，数据结构-2,链表。

3，代码第25行先判断链表相加之后的最高位是否有进位，如果有就直接返回，如果没有就返回头结点head的下一个结点即可。

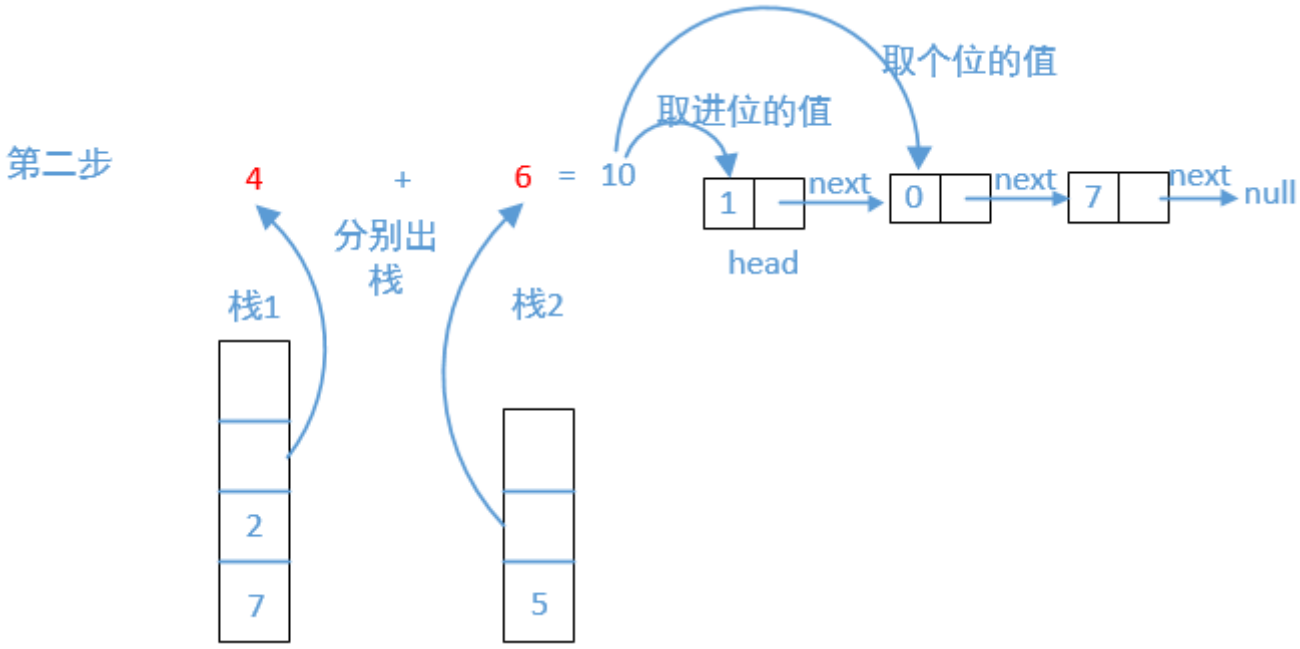
代码比较简单，我们就以上面的例子来画一个图加深一下理解，



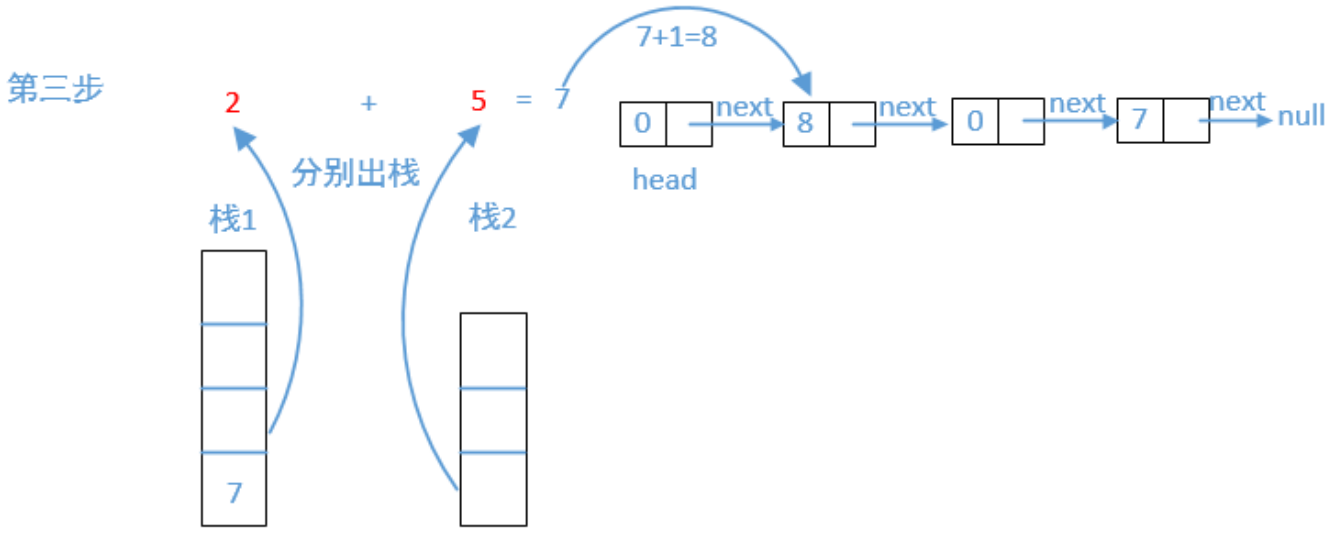
第一步

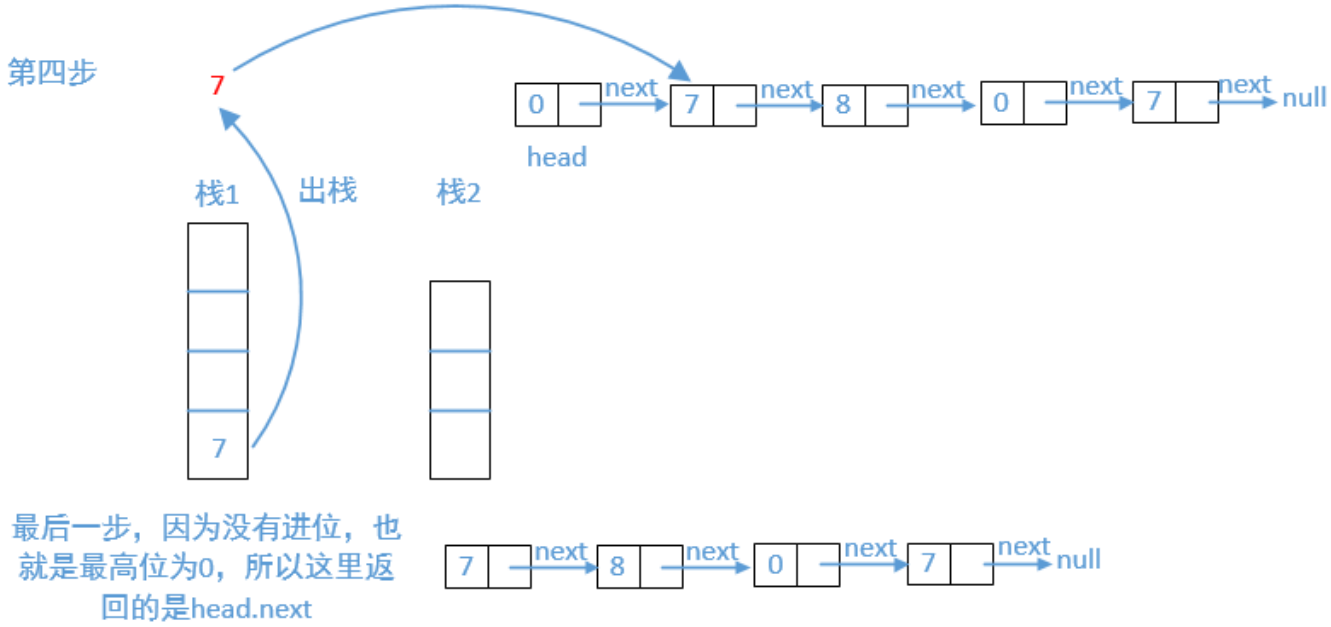

3
+
4
=
7
0
next
7
next
null
head
栈1
分别出栈
栈2
4
2
7
6
5

第二步


取个位的值
取进位的值
4
+
6
=
10
1
next
0
next
7
next
null
head
分别出栈
栈1
栈2
2
7
5

第三步


7+1=8
2
+
5
=
7
0
next
8
next
0
next
7
next
null
head
分别出栈
栈1
栈2
7



对于加法运算这块，我们还可以再来换一种写法

```
public ListNode addTwoNumbers(ListNode list1, ListNode list2) {
  Stack<ListNode> stk1 = new Stack();
  Stack<ListNode> stk2 = new Stack();
  while (list1 != null) {
    stk1.push(list1);
    list1 = list1.next;
  }
  while (list2 != null) {
    stk2.push(list2);
    list2 = list2.next;
  }
  int carry = 0;
  ListNode node = null, prev = null;
  while (!stk1.isEmpty() || !stk2.isEmpty()) {
    int no1 = !stk1.isEmpty() ? stk1.pop().val : 0;
    int no2 = !stk2.isEmpty() ? stk2.pop().val : 0;
    node = new ListNode((no1 + no2 + carry) % 10);
    carry = (no1 + no2 + carry) / 10;
    node.next = prev;
    prev = node;
  }
  if (carry != 0) {
    node = new ListNode(carry);
    node.next = prev;
  }
  return node;
}
```

carry表示的是进位的值，上面两种方式虽然写法上有一点差别，但整体思路还是没变，下面我们再来换种思路，使用递归的方式来解决

```
public ListNode addTwoNumbers(ListNode list1, ListNode list2) {
  int size1 = getLength(list1);
  int size2 = getLength(list2);
  ListNode head = new ListNode(1);
  head.next = size1 < size2 ? helper(list2, list1, size2 - size1) : helper(list1, list2, size1 - size2);
  if (head.next.val > 9) {
    head.next.val = head.next.val % 10;
    return head;
  }
  return head.next;
}

//这里链表list1的长度是大于等于list2的长度的
public ListNode helper(ListNode list1, ListNode list2, int offset) {
```

```
15      if (list1 == null)
16        return null;
17      ListNode result = offset == 0 ? new ListNode(list1.val + list2.val) : new ListNode(list1.val);
18      ListNode post = offset == 0 ? helper(list1.next, list2.next, 0) : helper(list1.next, list2, offset -
19      if (post != null && post.val > 9) {
20        result.val += 1;
21        post.val = post.val % 10;
22      }
23      result.next = post;
24      return result;
25    }
26
27    public int getLength(ListNode list) {
28      int count = 0;
29      while (list != null) {
30        list = list.next;
31        count++;
32      }
33      return count;
34    }
```

getLength表示的是计算链表的长度，代码很容易理解。这题解法思路奇特的地方在第4行，他先默认两个链表相加，最高位会有进位，然后在第6行进行判断，如果确实有进位，修改一下head.next的值，然后返回head，如果没有进位，直接在第10行返回head.next即可。这段代码的核心是在helper方法中，我们来仔细分析一下

1，list1的长度是大于等于list2的长度的，offset表示的是list1的长度与list2长度的差值，如果list1长度大于list2的长度，那么offset是正数，如果list1长度等于list2的长度，那么offset就是0，offset不可能是负数，因为list1长度不可能小于list2的长度。

2，第17行是创建一个新的结点，如果两个链表长度相等，这个新的结点的值就是这两个链表的头结点相加（注意这里的头结点是一直变的），否则新结点的值就是list1的头结点的值。

3，然后第18行进行这种递归的操作后续结点

4，19行判断后续结点是否有进位，如果有进位再处理进位的问题

5，第23行是链表的连接，连接之后在24行直接返回。

往期精彩回顾

352，数据结构-2,链表

333，奇偶链表

280，排序链表

276，重排链表

275，环形链表 II

232，旋转链表

354，字典序排数

山大王wld 数据结构和算法 4月29日

给定一个整数 n, 返回从 1 到 n 的字典顺序。

例如，

给定 n =13，返回 [1,10,11,12,13,2,3,4,5,6,7,8,9] 。

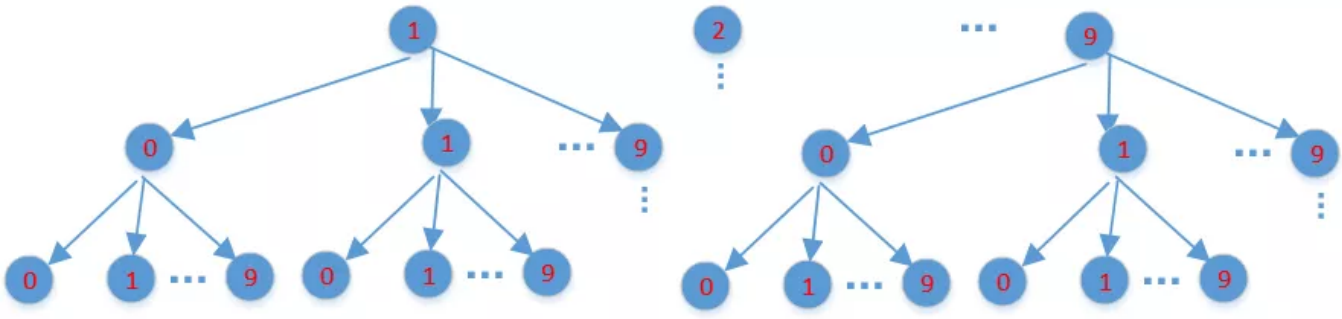
请尽可能的优化算法的时间复杂度和空间复杂度。输入的数据 n 小于等于 5,000,000。

答案：

```
1  public List<Integer> lexicalOrder(int n) {
2    List<Integer> res = new ArrayList<>();
3    for (int i = 1; i < 10; ++i) {
4      dfs(i, n, res);
5    }
6    return res;
7  }
8
9  public void dfs(int cur, int n, List<Integer> res) {
10   if (cur > n)
11     return;
12   res.add(cur);
13   for (int i = 0; i < 10; ++i) {
14     dfs(10 * cur + i, n, res);
15   }
16 }
```

解析：

解这题之前实现要明白什么是字典序，其实就是类似于字典一样，根据字母的顺序进行排列，我们先来看下面的图

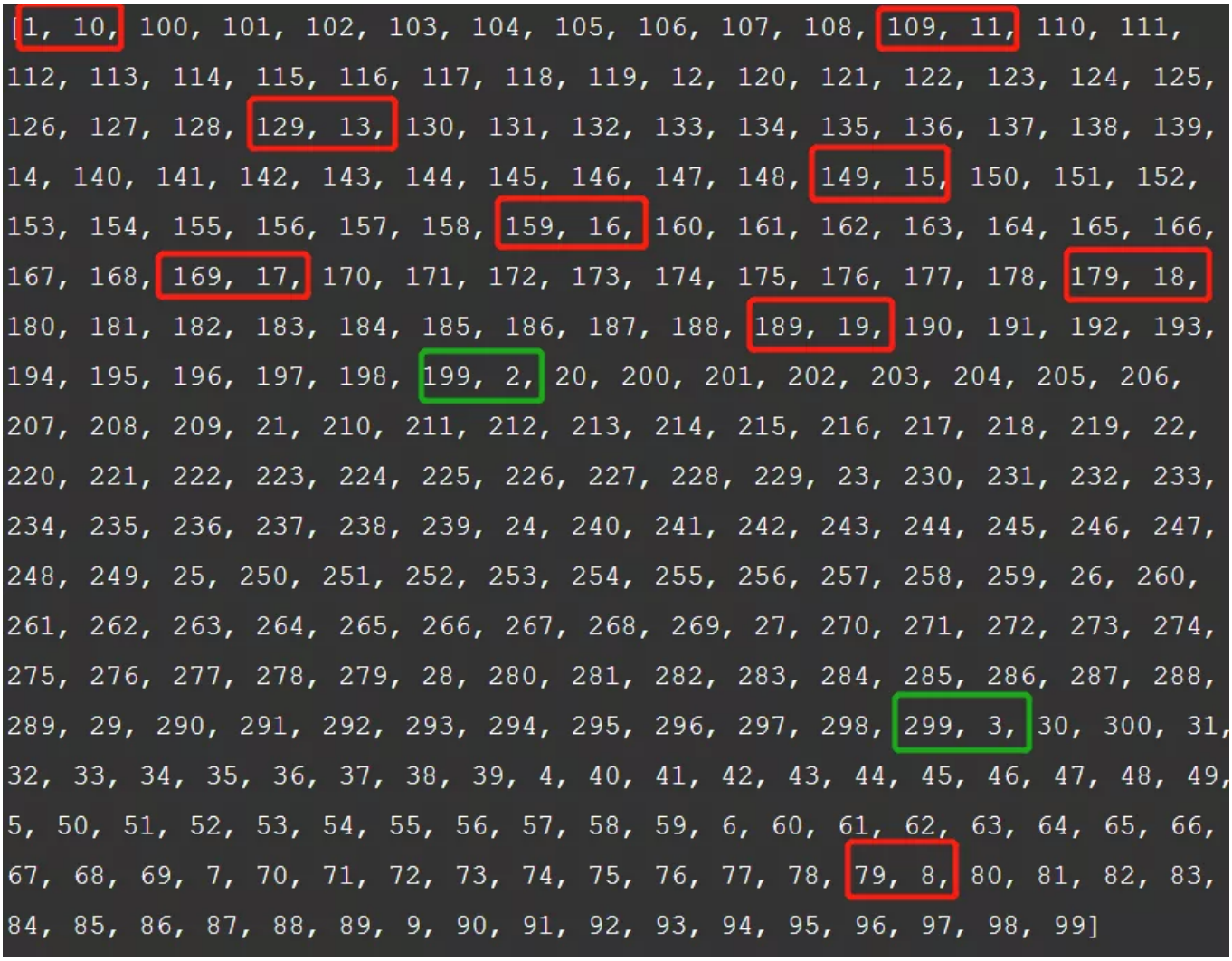


我们可以把它看成有9棵树，每棵树的根节点的值分别是从1到9，并且每棵树都有10个子节点，并且每个子节点又会有10个子节点.....

1，代码3到5行分别遍历这9棵树。

2，方法dfs对每棵树执行dfs（深度优先搜索），关于树的深度优先搜索可以参考前面介绍过的304，完全二叉树的节点个数这道题第二种解法使用的就是深度优先搜索（dfs）

我们仔细观察上面的图，字典序排数有一个规律，比如当n等于300的时候，结果是下面这样的



我们可以观察上面的数字，也可以查看最上面的图，就会发现这样一个规律。当数字curr小于n的时候，只要curr的个位数是9，那么他的下一个数就是(curr/10)+1（但要保证curr/10的个位数不能是9，如果是9就继续执行curr/10，直到curr/10的个位数不是9为止，比如199的下一个数是2），比如109的下一个是11，129的下一个是13一样。如果curr等于n，那么他的下一个数也是和上面同样的操作。理解了这点，代码就很容易写出来了

```java
public List<Integer> lexicalOrder(int n) {
  List<Integer> ans = new ArrayList<>(n);
  int curr = 1;
  for (int i = 1; i <= n; ++i) {
    ans.add(curr);
    if (curr * 10 <= n) {
      curr *= 10;
    } else {
      while (curr % 10 == 9 || curr == n)
        curr /= 10;
      curr++;
    }
  }
  return ans;
}
```

重点是在第9到10行，如果curr的个位数是9或者curr等于n就要执行curr/10这步操作，直到curr的个位数不是9为止。

往期精彩回顾

327，最小高度树

353，打乱数组

山大王wld 数据结构和算法 4月28日

打乱一个没有重复元素的数组。

示例:

```
// 以数字集合 1, 2 和 3 初始化数组。
int[] nums = {1,2,3};
Solution solution = new Solution(nums);

// 打乱数组 [1,2,3] 并返回结果。任何 [1,2,3]的排列返回的概率应该相同。
solution.shuffle();

// 重设数组到它的初始状态[1,2,3]。
solution.reset();

// 随机返回数组[1,2,3]打乱后的结果。
solution.shuffle();
```

答案:

```
1  public class Solution {
2    private int[] nums;
3    private Random random;
4
5    public Solution(int[] nums) {
6      this.nums = nums;
7      random = new Random();
8    }
9
10   //重置数组，就是返回之前的数组
11   public int[] reset() {
12     return nums;
13   }
14
15
16   //打乱数组
17   public int[] shuffle() {
18     if (nums == null)
19       return null;
20     int[] a = nums.clone();//clone一个新的数组
21     for (int j = 1; j < a.length; j++) {
22       int i = random.nextInt(j + 1);
23       swap(a, i, j);
24     }
25     return a;
26   }
27
28   //交换两个数字的值
29   private void swap(int[] a, int i, int j) {
30     if (i != j) {
31       a[i] ^= a[j];
32       a[j] ^= a[i];
33       a[i] ^= a[j];
34     }
35   }
36 }
```

解析：

代码很简单，就不再过多叙述。上面的swap方法是叫交换两个数的值，如果不这样写，我们还可以通过一个临时变量来交换，比如下面这样

```
1     private void swap(int[] a, int i, int j) {
2       int t = a[i];
3       a[i] = a[j];
4       a[j] = t;
5     }
```

这种写法估计大家都能看的懂，非常简单，除了上面两种写法以外，我们还可以再换一种写法

```
1     private void swap(int[] a, int i, int j) {
2       if (i != j) {
3         a[i] = a[i] + a[j];
4         a[j] = a[i] - a[j];
5         a[i] = a[i] - a[j];
6       }
7     }
```

这种写法估计大家会有疑惑，第3行两个数相加会不会出现数字溢出，这种考虑是对的，但这种担忧是多余的，如果a[i]，和a[j]都比较大的话，相加会出现溢出，但不影响最终的结果。因为a[i]和a[j]无论多大，他们相加的结果最多只会往前进一位。举个例子，一个3位数和一个3位数相加最多只能是4位数，不可能是5位数。而二进制的最高位是符号位，1表示的是负数，0表示的是非负数，他们是可以参与运算的。我们可以写一段代码来测试一下

```
1   public static void main(String args[]) {
2     int a = Integer.MAX_VALUE - 5;
3     int b = Integer.MAX_VALUE - 20;
4     System.out.println("交换之前--》：a的值是：" + a + "：---->b的值是：" + b);
5     a = a + b;
6     System.out.println("相加之后a的值：" + a + "\t\t二进制是：" + Util.bitInt32(a));
7     b = a - b;
8     System.out.println("相减之后b的值：" + b + "\t二进制是：" + Util.bitInt32(b));
9     a = a - b;
10    System.out.println("相减之后a的值：" + a + "\t二进制是：" + Util.bitInt32(a));
11    System.out.println("交换之后--》：a的值是：" + a + "：---->b的值是：" + b);
12  }
```

我们看到第5行肯定会出现数字溢出，我们来看一下打印的结果

```
1     交换之前--》：a的值是：2147483642：---->b的值是：2147483627
2     相加之后a的值：-27          二进制是：11111111 11111111 11111111 11100101
3     相减之后b的值：2147483642   二进制是：01111111 11111111 11111111 11111010
4     相减之后a的值：2147483627   二进制是：01111111 11111111 11111111 11101011
5     交换之后--》：a的值是：2147483627：---->b的值是：2147483642
```

尽管a和b非常大，但最终还是实现了a和b的交换。

351，最少移动次数使数组元素相等 II

山大王wld 数据结构和算法 4月23日

给定一个非空整数数组，找到使所有数组元素相等所需的最小移动数，其中每次移动可将选定的一个元素加1或减1。您可以假设数组的长度最多为10000。

例如:

```
输入：
[1,2,3]

输出：
2

说明：
只有两个动作是必要的（记得每一步仅可使其中一个元素加1或减1）：

[1,2,3]  =>  [2,2,3]  =>  [2,2,2]
```

答案:

```java
public int minMoves2(int[] nums) {
  Arrays.sort(nums);
  int i = 0, j = nums.length - 1;
  int count = 0;
  while (i < j) {
    count += nums[j--] - nums[i++];
  }
  return count;
}
```

解析:

这题其实更像是一道数学题，先来分析一下，我们假设把两个数a，b（a<=b）经过最少的转换最终变为x。这里有3种情况

1，x<=a，转变次数为(a-x)+(b-x)=(a+b)-2x;

2，a<=x<=b，转变次数为(x-a)+(b-x)=b-a;

3，x>=b，转变次数为(x-a)+(x-b)=2x-(a+b);

所以很明显第二种情况转换的次数是最少的，也就是说x在a和b之间的时候，转换的次数是最少的。我们可以使用极限法证明

1，当x=a时取最小值，最小值是(a+b)-2a=b-a;

3，当x=b时取最小值，最小值是2b-(a+b)=b-a;

所以这个问题就很好解决了，我们只需要把数组排好序，当数组长度为2的时候，只需要大的减小的即可，当数组长度为3的时候，我们只需要计算中间值与那两个数差的绝对值的和即可（或者是最后一个减去第一个），当数组长度为n的时候，我们只需要找到最中间值，然后每一个数与他的差的绝对值相加即可。

这里的排序使用的是Arrays的sort方法，当然我们还可以自己写，如果对排序算法不熟悉的，还可以看一下我之前写的十几种排序算法

101，排序-冒泡排序
102，排序-选择排序
103，排序-插入排序
104，排序-快速排序
105，排序-归并排序
106，排序-堆排序
107，排序-桶排序
108，排序-基数排序
109，排序-希尔排序
110，排序-计数排序
111，排序-位图排序
112，排序-其他排序

350，有序矩阵中第K小的元素

山大王wld 数据结构和算法 4月22日

给定一个 *n* x *n* 矩阵，其中每行和每列元素均按升序排序，找到矩阵中第k小的元素。

请注意，它是排序后的第k小元素，而不是第k个元素。

示例:

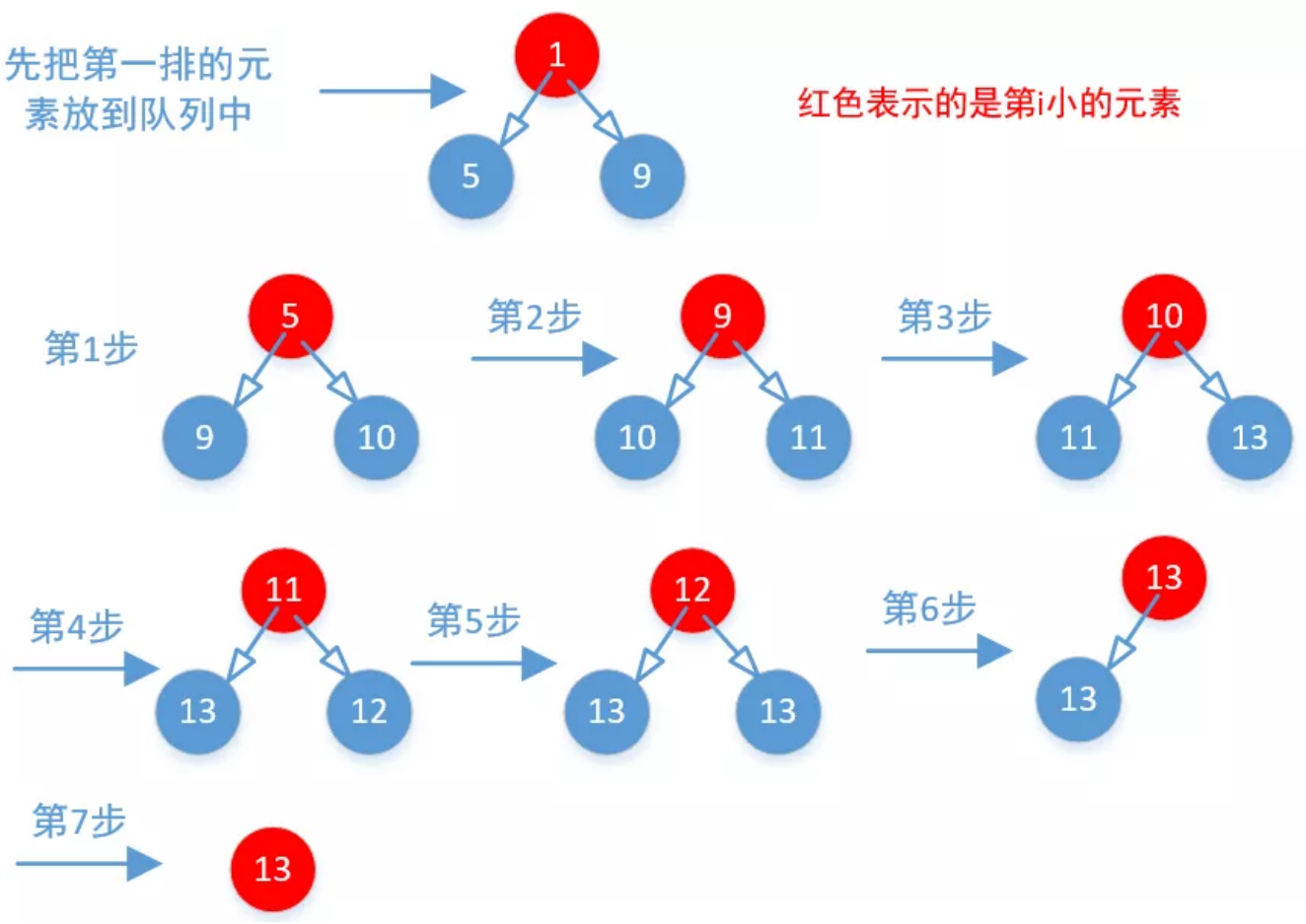
```
matrix = [
   [ 1,  5,  9],
   [10, 11, 13],
   [12, 13, 15]
],
k = 8,

返回 13。
```

答案：

```
1   public int kthSmallest(int[][] matrix, int k) {
2     int n = matrix.length;
3     PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> a[2] - b[2]);
4     for (int j = 0; j < n; j++)
5       pq.offer(new int[]{0, j, matrix[0][j]});
6     for (int i = 0; i < k - 1; i++) {
7       int[] cur = pq.poll();
8       if (cur[0] == n - 1)
9         continue;
10      pq.offer(new int[]{cur[0] + 1, cur[1], matrix[cur[0] + 1][cur[1]]});
11    }
12    return pq.poll()[2];
13  }
```

解析：

PriorityQueue会对添加进去的数据进行排序，其实他就是一个堆，在这里他是个最小堆，也就是最顶端的元素是最小的（虽然他是数组结构，但数组的位置是有关联的），每添加一个元素他都会往上调整，删除的时候往下调整，并且他有两个最重要的函数一个是siftDown，一个是siftUp。上面的代码中它添加的是个数组，数组的最后一个元素是我们添加的值，前两个元素是这个值在矩阵中（x，y）的坐标。他是先把矩阵中第一行的元素全部添加进去，后面再进行k-1次循环。每次循环的时候都会把最小的给移除掉，然后把它紧挨着的下一个元素添加进去。因为是最小堆，所以当我们移除了k-1次的时候，这时堆的顶端就是第k小的元素。我们还以上面的例子来画个图加深一下理解



题中矩阵的每行每列都是排过序的，所以我们还可以想到另一种方法，使用二分法查找，之前介绍过二分法查找，不会的可以看下202，查找-二分法查找

```
public int kthSmallest(int[][] matrix, int k) {
  int low = matrix[0][0], high = matrix[matrix.length - 1][matrix[0].length - 1];
  while (low < high) {
    int mid = low + (high - low) / 2;
    int count = 0, j = matrix[0].length - 1;
    for (int i = 0; i < matrix.length; i++) {
      while (j >= 0 && matrix[i][j] > mid)
        j--;
      count += (j + 1);
    }
    if (count < k)
      low = mid + 1;
    else
      high = mid;
  }
  return low;
}

```

因为矩阵的行和列都是排过序的，这里先找到最中间的值，count表示的是比mid小的值有多少个，每次都是用每行的最右边的一个值和mid比较，直到比mid大，才会执行上面的while循环，然后再往前找，这里的第count个值是大于mid的最小值，所以我们不能在count==k的时候直接return。

往期精彩回顾

238，搜索二维矩阵

312，搜索二维矩阵II

349，组合总和 IV

山大王wld 数据结构和算法 4月21日

给定一个由正整数组成且不存在重复数字的数组，找出和为给定目标正整数的组合的个数。

示例:

```
nums = [1, 2, 3]
target = 4

所有可能的组合为:
(1, 1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 3)
(2, 1, 1)
(2, 2)
(3, 1)

请注意，顺序不同的序列被视作不同的组合。

因此输出为 7。
```

答案:

```java
public int combinationSum4(int[] nums, int target) {
  int[] count = new int[1];
  helper(nums, new int[]{target}, count);
  return count[0];
}

private void helper(int[] nums, int[] target, int[] count) {
  if (target[0] == 0) {
    count[0]++;
    return;
  }
  if (target[0] > 0)
    for (int i = 0; i < nums.length; i++) {
      target[0] -= nums[i];
      helper(nums, target, count);
      target[0] += nums[i];
    }
}
```

解析:

这种是递归加回溯的方式，前面也讲过很多这种类似的题的，也很容易理解，但这种递归效率实在是很差，我们还可以在改进一下，减少重复计算

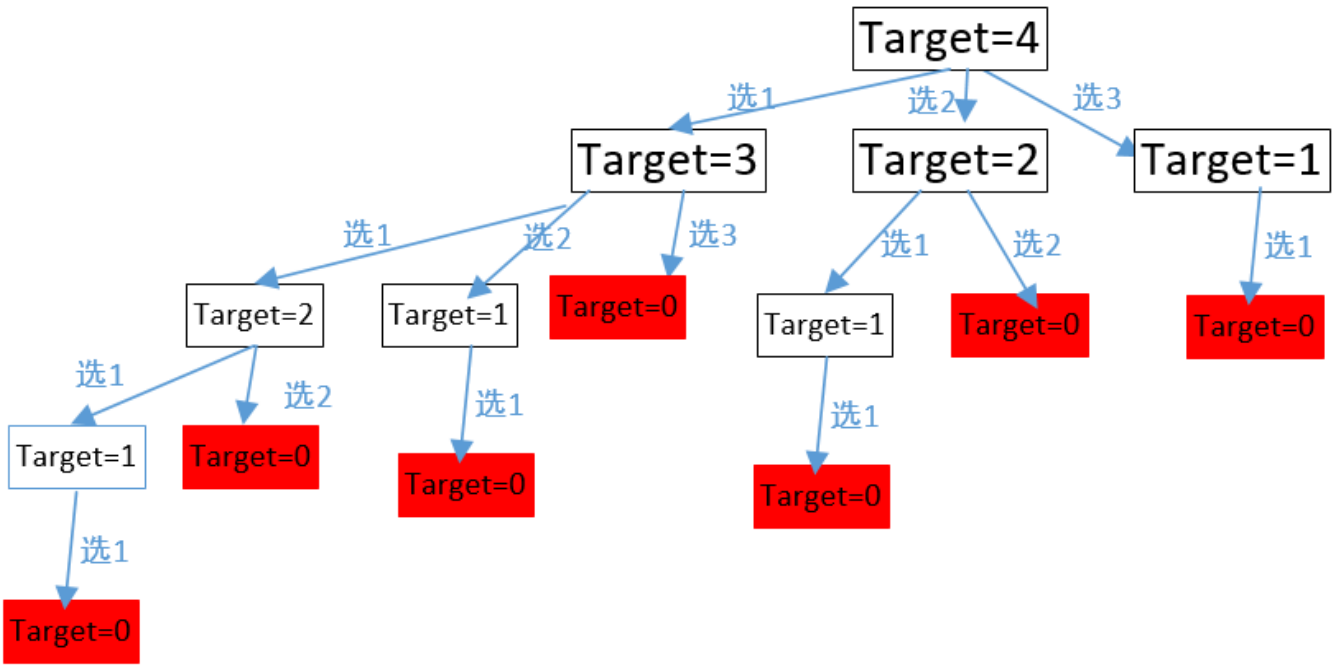
```java
public int combinationSum4(int[] nums, int target) {
  return helper(nums, target, new HashMap<Integer, Integer>());
}

private int helper(int[] nums, int target, Map<Integer, Integer> map) {
  if (target < 0)
    return 0;
  if (target == 0)
```

```
      return 1;
    if (map.containsKey(target))
      return map.get(target);
    int res = 0;
    for (int i = 0; i < nums.length; i++) {
      int cnt = helper(nums, target - nums[i], map);
      if (target >= nums[i])
        map.put(target - nums[i], cnt);
      res += cnt;
    }
    return res;
  }
```

即便是这样，但因为递归的使用，导致运行效率还不是很高，我们可以考虑一下动态规划的使用

```
public int combinationSum4(int[] nums, int target) {
  int[] dp = new int[target + 1];
  dp[0] = 1;
  for (int i = 1; i <= target; i++) {
    for (int j = 0; j < nums.length; j++) {
      if (i - nums[j] >= 0) {
        dp[i] += dp[i - nums[j]];
      }
    }
  }
  return dp[target];
}
```

这个非常类似于背包问题，但背包问题有数量上的限制，而这道题没有。我们还以上面举的例子，画个图来加深一下理解，我们可以把它当成一棵树，每个节点最多有nums.length个子节点。



我们看到树中有大量的子树重复，上面代码是按照从树的底往上进行计算的，如果想要从上往下计算我们就要使用递归的方式了，也就是前面两种解法的答案。

喜欢此内容的人还喜欢

483，完全二叉树的节点个数

数据结构和算法



347，猜数字大小 II

山大王wld 数据结构和算法 4月16日

我们正在玩一个猜数游戏，游戏规则如下：

我从 1 到 n 之间选择一个数字，你来猜我选了哪个数字。

每次你猜错了，我都会告诉你，我选的数字比你的大了或者小了。

然而，当你猜了数字 x 并且猜错了的时候，你需要支付金额为 x 的现金。直到你猜到我选的数字，你才算赢得了这个游戏。

示例:

```
n = 10, 我选择了8.
第一轮: 你猜我选择的数字是5，我会告诉你，我的数字更大一些，然后你需要支付5块。
第二轮: 你猜是7，我告诉你，我的数字更大一些，你支付7块。
第三轮: 你猜是9，我告诉你，我的数字更小一些，你支付9块。
游戏结束。8 就是我选的数字。
你最终要支付 5 + 7 + 9 = 21 块钱。
```

给定 **n ≥ 1**，计算你至少需要拥有多少现金才能确保你能赢得这个游戏。

答案：

```
1  public static int getMoneyAmount(int n) {
2    return DP(1, n);
3  }
4
5  public static int DP(int left, int right) {
6    if (left >= right)
7      return 0;
8    int res = Integer.MAX_VALUE;
9    for (int i = left; i <= right; i++) {
10     int tmp = i + Math.max(DP(left, i - 1), DP(i + 1, right));
11     res = Math.min(res, tmp);
12   }
13   return res;
14 }
```

解析：

在数字1-n之间，假如我们选择了x，数组就会分成3个部分，[1,x-1]，x，[x+1,n]。那么会有3种情况，第一种我们选中了，所以这时候花费现金最少，第二种是我们选大了，第三种是我们选小了，用函数f（m,n）表示从数字m到n中所花费的最小金额，如果选择了在范围（i，j）中保证所花费最少（i<=x<=j），我们有下面这样一个公式。money=x+max(f(i，x-1)+f(x+1，j))；代码第10行我们找的是最大值，因为题目说了至少花费多钱现金才能赢得游戏，我们只需要找到所有的花费最大值中的最小值即可。但是上面代码效率明显不是很高，因为递归的原因会出现重复计算，我们只需要用一个临时数组存储每次计算的结果，防止重复计算即可，我们来优化一下

```
1  public static int getMoneyAmount(int n) {
2    int[][] table = new int[n + 1][n + 1];
3    return DP(table, 1, n);
4  }
5
6  public static int DP(int[][] table, int left, int right) {
7    if (left >= right)
```

```
8         return 0;
9       if (table[left][right] != 0)
10        return table[left][right];
11      int res = Integer.MAX_VALUE;
12      for (int i = left; i <= right; i++) {
13        int tmp = i + Math.max(DP(table, left, i - 1), DP(table, i + 1, right));
14        res = Math.min(res, tmp);
15      }
16      table[left][right] = res;
17      return res;
18    }
19
```

除了递归的写法以外，我们还可以使用动态规划的方式解决，先看一下代码

```
1   public int getMoneyAmount(int n) {
2     int[][] table = new int[n + 1][n + 1];
3     for (int i = 2; i <= n; i++) {
4       for (int j = i - 1; j > 0; j--) {
5         int globalMin = Integer.MAX_VALUE;
6         for (int k = j + 1; k < i; k++) {
7           int localMax = k + Math.max(table[j][k - 1], table[k + 1][i]);
8           globalMin = Math.min(globalMin, localMax);
9         }
10        table[j][i] = j + 1 == i ? j : globalMin;
11      }
12    }
13    return table[1][n];
14  }
15
```

前两个for循环会组成一个封闭的空间[j，i]，然后第3个for循环再从这个封闭的空间中找出所有花费最大值的最小值即可，这个最小值也只不过是在区间[j，i]之间的，然后通过外面两层的for循环最终会找出区间[1，n]的值。第10行表示如果j和i仅仅相差1的话，那么第3个for循环根本就不会执行，我们要猜最小的才能花费最少，所以选择j。下面来画个图加深一下理解

	1	2	3	4	5
1	0	1	2	4	6
2		0	2	3	6
3			0	3	4
4				0	4
5					0

1，这里如果只有一个数字，我们只有一个选择，所以花费为0，图中的绿色

2，如果有两个数字，我们只需要选择最小的即可，就会花费最少，图中金色

3，如果有三个数字，我们只需要选择中间的即可，就会花费最少，图中蓝色

3，如果超过3个数字就是图中的红色部分了，我们就需要通过计算才能得到，我这里举个例子，比如我们要计算（2,5）方格的值，我们只需要计算

2+（3,5）=2+4=6；

3+max((2,2),(4,5))=3+max(0+4)=7；

4+max((2,3),(5,5))=4+max(2,0)=6；

5+（2，4）=5+3=8;

我们只需要取他们的最小值即可，min(6,7,6,8)=6,所以（2,5）这个方格是6，同理（1,4）和（1,5）都可以通过这种方式计算出来

再举个例子，比如我们计算（1,5）方格内的值

1+（2,5）=1+6=7

2+max((1,1),(3,5))=2+max(0,4)=6;

3+max((1,2),(4,5))=3+max(1,4)=7;

4+max((1,3),(5,5))=4+max(2,0)=6;

5+(1,4)=5+4=9;

所以min(7,6,7,6,9)=6，即（1,5）=6；

这里我们来举个例子简单说下，比如n等于5，为什么只需要6块钱就一定能赢得游戏，结合上面的分析，我们知道当我们先猜2，或者4的时候结果都是6，我们以先猜2来分析一下，

1，如果选择的是1，我们猜2，说明大了，下一步直接猜1就行了，所以只花了2块钱。

2，如果选择的是2，我们猜2，说明猜对了，一分没花。

3，如果选择的是3，我们猜2，说明猜小了，下一步猜4，说明大了，在下一步直接猜3，猜对了，所以总共花了2+4=6块钱。

4，如果选择的是4，我们猜2，说明猜小了，下一步猜4，猜中了，只花了2块钱。

5，如果选择的是5，我们猜2，说明猜小了，下一步猜4，又小了，在下一步直接猜5，猜对了，所以总共花了2+4=6块钱。

综上所述，当n等于5的时候，我们只需要6块钱就一定能赢，

思考：

这题我估计很多人都听说过，或者看过类似的这种题，很多时候我们猜这样的题都喜欢从中间来猜，显然通过上面的分析，如果我们从中间猜不一定会有最优的结果，比如当n=5的时候，当我们选择4，或者5的

时候，如果我们从中间来猜，先猜3，小了，再猜4，这时候无论是猜对了还是猜小了，所花费的肯定是大于6的，很明显不是最优解。

往期精彩回顾

185，猜数字大小

186，赎金信

346，查找和最小的K对数字

山大王wld 数据结构和算法 4月15日

给定两个以升序排列的整形数组 nums1 和 nums2, 以及一个整数 k。

定义一对值 (u,v)，其中第一个元素来自 nums1，第二个元素来自 nums2。

找到和最小的 k 对数字 (u1,v1), (u2,v2) ... (uk,vk)。

示例 1:

```
输入: nums1 = [1,7,11], nums2 = [2,4,6], k = 3
输出: [1,2],[1,4],[1,6]
解释: 返回序列中的前 3 对数:
     [1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]
```

示例 2:

```
输入: nums1 = [1,1,2], nums2 = [1,2,3], k = 2
输出: [1,1],[1,1]
解释: 返回序列中的前 2 对数:
     [1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]
```

示例 3:

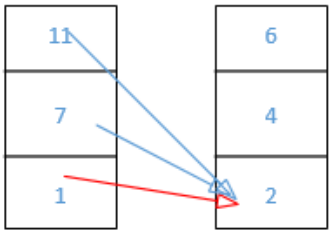
```
输入: nums1 = [1,2], nums2 = [3], k = 3
输出: [1,3],[2,3]
解释: 也可能序列中所有的数对都被返回:[1,3],[2,3]
```

答案:

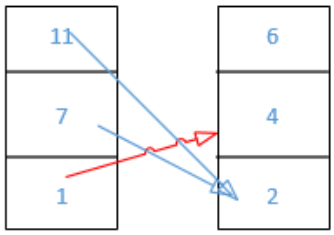
```java
public List<List<Integer>> kSmallestPairs(int[] nums1, int[] nums2, int k) {
    PriorityQueue<int[]> que = new PriorityQueue<>((a, b) -> a[0] + a[1] - b[0] - b[1]);
    List<List<Integer>> res = new ArrayList<>();
    if (nums1.length == 0 || nums2.length == 0 || k == 0)
        return res;
    for (int i = 0; i < nums1.length && i < k; i++)
        que.offer(new int[]{nums1[i], nums2[0], 0});
    while (k-- > 0 && !que.isEmpty()) {
        int[] cur = que.poll();
        res.add(Arrays.asList(cur[0], cur[1]));
        if (cur[2] == nums2.length - 1)
            continue;
        que.offer(new int[]{cur[0], nums2[cur[2] + 1], cur[2] + 1});
    }
    return res;
}
```

解析:

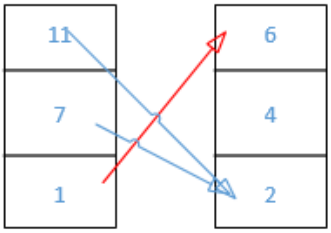
题目说了nums1和nums2是排过序的，首先把nums1中的每个元素和nums2中的第一个元素进行组合，然后存放到队列中，队列que会根据每个组合的和的大小对他们进行排序，所以第一个最小的组合毫无疑问是[nums1[0]，nums2[0]]，在while循环中是第一个出队的，然后再把[nums1[0]，nums2[1]]放入到队列中（加入到队列中的组合都会进行排序的），然后这样进行不停的循环，直到找出k个为止。我们可以参照上面的数据画个图来分析一下



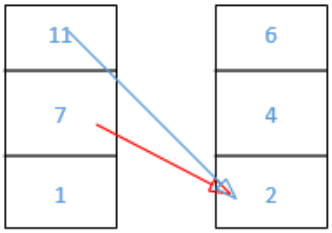
第一步



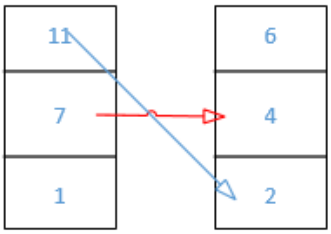
第二步



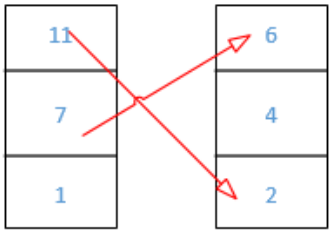
第三步



第四步



第五步



第六步

往期精彩试题

312，搜索二维矩阵II

238，搜索二维矩阵

345，超级次方

山大王wld 数据结构和算法 4月14日

你的任务是计算 *a^b* 对 1337 取模，*a* 是一个正整数，*b* 是一个非常大的正整数且会以数组形式给出。

示例 1:

```
输入: a = 2, b = [3]
输出: 8
```

示例 2:

```
输入: a = 2, b = [1,0]
输出: 1024
```

答案:

```
1   public int superPow1(int a, int[] b) {
2     int res = 1;
3     for (int i : b) {
4       res = pow(res, 10) * pow(a, i) % 1337;
5     }
6     return res;
7   }
8
9   int pow(int x, int y) {
10    if (y == 0)
11      return 1;
12    if (y == 1)
13      return x % 1337;
14    return pow(x % 1337, y / 2) * pow(x % 1337, y - y / 2) % 1337;
15  }
```

解析:

这题其实更像是一道数学题，要想解这题我们要明白这样一个公式(a*b)%k=(a%k)*(b%k)%k，很好证明，这里就不在过多解释。上面pow函数使用递归的方式求解x^y，并且每次运算都会对1337求余，举个例子，如果求3^18，我们只需要求3^9然后再相乘即可，如果求3^9，我们只需要求3^4和3^5的乘积即可。代码第4行我们把它想象成一个数组转换为数字这样一个过程，就很容易理解了。下面我们再来看种解法

```
1   public int superPow(int a, int[] b) {
2     int res = 1;
3     int p = a;
4     for (int i = b.length - 1; i >= 0; i--) {
5       res = res * pow(p, b[i], 1337) % 1337;
6       p = pow(p, 10, 1337);
7     }
8     return res;
9   }
10
11  public int pow(int a, int b, int c) {
12    long res = 1;
13    long p = a;
14    while (b > 0) {
15      if ((b & 1) == 1) {
16        res = (res * p) % c;
17      }
18      p = (p * p) % c;
```

```
19        b >>= 1;
20      }
21      return (int) (res % c);
22    }
```

函数pow代码很好理解，第15行先判断b是否是奇数，然后再计算。superPow函数中我们首先要明白a^(bc)=(a^b)^c这样一个公式才能看懂上面的代码，比如3^20=(3^10)^2。下面再来看最后一种解法

```
1   public int superPow2(int a, int[] b) {
2     if (a % 1337 == 0)
3       return 0;
4     int p = 0;
5     for (int i : b)
6       p = (p * 10 + i) % 1140;
7     if (p == 0)
8       p += 1440;
9     return power(a, p, 1337);
10  }
11
12  public int power(int a, int b, int c) {
13    long res = 1;
14    long p = a;
15    while (b > 0) {
16      if ((b & 1) == 1) {
17        res = (res * p) % c;
18      }
19      p = (p * p) % c;
20      b >>= 1;
21    }
22    return (int) (res % c);
23  }
```

这种解法如果看不懂的话，可以忽略。我估计有部分同学是看不懂的，因为这里涉及到一个定理，叫欧拉定理，也叫费马-欧拉定理。下面简单提示一下

1337的因数中除了1和他本身以外，还可以分解为1337=7*191，并且7和191都是质数，也称为素数，

φ(7)=6，φ(191)=190，

所以φ(1337)=φ(7)*φ(191)=6*190=1140；

φ(a)表示的是比a小的正整数中与a互素的数的个数。

往期精彩试题

297，找出不小于x的2的n次方的最小值

294，计算2的n次方

178，4的幂

177，3的幂

162，2的幂

50，幂的数字和

344，最大整除子集

山大王wld 数据结构和算法 4月13日

给出一个由无重复的正整数组成的集合，找出其中最大的整除子集，子集中任意一对 (Si，Sj) 都要满足：Si % Sj = 0 或 Sj % Si = 0。

如果有多个目标子集，返回其中任何一个均可。

示例 1:

```
输入：[1,2,3]
输出：[1,2] (当然，[1,3] 也正确)
```

示例 2:

```
输入：[1,2,4,8]
输出：[1,2,4,8]
```

答案：

```java
public List<Integer> largestDivisibleSubset1(int[] nums) {
  int n = nums.length;
  int[] dp = new int[n];
  int[] pre = new int[n];
  Arrays.sort(nums);
  Arrays.fill(dp,1);
  Arrays.fill(pre,-1);
  int max = 0, index = -1;
  for (int i = 0; i < n; i++) {
    for (int j = i - 1; j >= 0; j--) {
      if (nums[i] % nums[j] == 0) {
        if (1 + dp[j] > dp[i]) {
          dp[i] = dp[j] + 1;
          pre[i] = j;
        }
      }
    }
    if (dp[i] > max) {
      max = dp[i];
      index = i;
    }
  }
  List<Integer> res = new ArrayList<>();
  while (index != -1) {
    res.add(nums[index]);
    index = pre[index];
  }
  return res;
}
```

解析：

这题实际上是求最长等比数列，我们可以通过动态规划来求解。dp[i]表示是数组中前i个元素组成的最大整除子集的个数，首先第5行对数组nums进行排序，如果nums[i]%nums[j]==0，则表示nums[i]能被nums[j]整除，所以dp[i]=max{dp[i]，dp[j]+1},但这里求的不是最大整除子集的长度，而是把最大整除子集的元素全部列出来，所以这里要使用一个临时数组pre来存储最大整除子集元素的下标。代码中第8-

17行主要是计算dp和pre的值，第18-21行主要是为了记录最大整除子集的最后一个元素的下标，然后在第24-27行通过最大整除子集的下标把元素找出来，存放到res集合中。当然我们还可以在换一种写法

```
1   public List<Integer> largestDivisibleSubset2(int[] nums) {
2     List<Integer> res = new ArrayList<>();
3     Arrays.sort(nums);
4     List<Integer>[] lists = new List[nums.length];
5     for (int i = 0; i < nums.length; i++) {
6       lists[i] = new ArrayList<>();
7     }
8     for (int i = nums.length - 1; i >= 0; i--) {
9       List<Integer> largest = new ArrayList<>();
10      for (int j = i + 1; j < nums.length; j++) {
11        if (nums[j] % nums[i] == 0) {
12          if (largest.size() < lists[j].size())
13            largest = lists[j];
14        }
15      }
16      lists[i].add(nums[i]);
17      lists[i].addAll(largest);
18      if (res.size() < lists[i].size())
19        res = lists[i];
20    }
21    return res;
22  }
```

这种写法没有像第一种那样有一个临时数组存储最大整除子集的下标，他是每次计算找到最大的都会赋值给res。下面再来看最后一种解法，使用递归的方式解决

```
1   public List<Integer> largestDivisibleSubset(int[] nums) {
2     Arrays.sort(nums);
3     return helper(nums, 0, new HashMap<>());
4   }
5
6   private List<Integer> helper(int[] nums, int index, HashMap<Integer, List<Integer>> map) {
7     if (map.containsKey(index))
8       return map.get(index);
9     List<Integer> maxLenLst = new ArrayList<>();
10    int div = index == 0 ? 1 : nums[index - 1];//div除数
11    for (int k = index; k < nums.length; k++) {
12      if (nums[k] % div == 0) {
13        List<Integer> lst = new ArrayList<>(helper(nums, k + 1, map));
14        lst.add(nums[k]);
15        if (lst.size() > maxLenLst.size())
16          maxLenLst = lst;
17      }
18    }
19    map.put(index, maxLenLst);
20    return maxLenLst;
21  }
```

map只是为了减少重复计算而引入的，helper函数表示从数组下标的index到数组的最后一个元素所能构成的最大整除子集。

往期精彩试题

329，最大单词长度乘积

303，最大正方形

300，数组中的第K个最大元素

221，最大回文数乘积

119，最大子序和

343，水壶问题

山大王wld 数据结构和算法 4月9日

有两个容量分别为 x升 和 y升 的水壶以及无限多的水。请判断能否通过使用这两个水壶，从而可以得到恰好 z升 的水？

如果可以，最后请用以上水壶中的一或两个来盛放取得的 z升 水。

你允许：

- 装满任意一个水壶
- 清空任意一个水壶
- 从一个水壶向另外一个水壶倒水，直到装满或者倒空

示例 1:

```
输入: x = 3, y = 5, z = 4
输出: True
```

示例 2:

```
输入: x = 2, y = 6, z = 5
输出: False
```

答案：

```
public boolean canMeasureWater1(int x, int y, int z) {
  return z == 0 || (long) x + y >= z && z % gcd(x, y) == 0;
}

public int gcd(int x, int y) {//求x，y的最大公约数
  return y == 0 ? x : gcd(y, x % y);
}
```

解析：

这题估计大家都遇到过好多次了，即使没在面试中遇到过，但至少在书上也看到过。这题如果单从代码上来看基本上没什么难度，难的是对这题的理解，其实这里面涉及到一个定理叫**裴蜀定理。**需要理解他，这题才能看明白。下面再来看一种解法，

```
public boolean canMeasureWater(int x, int y, int z) {
  if (z < 0 || z > x + y) {
    return false;
  }
  Set<Integer> set = new HashSet<>();
  Queue<Integer> q = new LinkedList<>();
  q.offer(0);
  while (!q.isEmpty()) {
    int n = q.poll();

    int top = n - y;
    if (top >= 0 && set.add(top)) {
      q.offer(top);
    }
    int down = n + y;
    if (down <= x + y && set.add(down)) {
      q.offer(down);
    }
    int left = n - x;
```

```
20        if (left >= 0 && set.add(left)) {
21          q.offer(left);
22        }
23        int right = n + x;
24        if (right <= x + y && set.add(right)) {
25          q.offer(right);
26        }
27        if (set.contains(z)) {
28          return true;
29        }
30      }
31      return false;
32    }
```

以原点0为中心，向他的上下左右4个方向发散，所以最终会满足一个方程ax+by=z，并且a，b都是整数，如果x，y，z都不为0的情况下，当且仅当x+y=z的时候，a=b=1，否则如果满足条件，a和b肯定是一个为正数一个为负数，也就是一个总共装了几桶水，一个总共倒了几桶水。

喜欢此内容的人还喜欢

483，完全二叉树的节点个数

数据结构和算法



342，计算各个位数不同的数字个数

山大王wld 数据结构和算法 4月8日

给定一个**非负**整数 n，计算各位数字都不同的数字 x 的个数，其中 0 ≤ x < 10^n 。

示例:

```
输入: 2
输出: 91
解释: 答案应为除去 11,22,33,44,55,66,77,88,99 外，在 [0,100) 区间内的所有数字。
```

答案:

```
public int countNumbersWithUniqueDigits1(int n) {
  if (n == 0)
    return 1;
  int res = 10;
  int uniqueDigits = 9;
  int availableNumber = 9;
  while (n-- > 1 && availableNumber > 0) {
    uniqueDigits = uniqueDigits * availableNumber;
    res += uniqueDigits;
    availableNumber--;
  }
  return res;
}
```

解析:

这题没什么难度，只要上过高中，学过排列组合的估计都能看懂，我简单介绍一下，当n=1的时候也就是从0-9有多少个数各位数字都不同，很明显是10，当n等于2的时候也就是从10-99有多少个数各位数字都不同，根据排序组合先从1-9中选择一个数字x作为十位数（十位数不能是0），然后再从0-9中选择一个数字y作为个位数，组成一个新的数，其中x!=y，选择x有9种方式，选择y也有9种方式，所以有81种，再加上前面的10种，总共91种。同理当n=3的时候从100-999中选择有9*9*8种，当n=4的时候从1000-9999中选择有9*9*8*7种，所以规律很好发现，明白了这点，代码就容易理解多了，我们还可以再来修改一下

```
public int countNumbersWithUniqueDigits(int n) {
  int res[] = new int[n + 1];
  res[0] = 1;
  int sum = 1;
  int k = 9;
  for (int i = 1; i <= n && k > 0; i++) {
    if (i == 1)
      res[i] += res[i - 1] * 9;
    else
      res[i] += res[i - 1] * k--;
    sum += res[i];
  }
  return sum;
}
```

上面两种虽然写法有点区别，但思路都是一样的。下面再来看一种递归的方式。

```
public int countNumbersWithUniqueDigits3(int n) {
  return doCount(n, new boolean[10], 0);
}

private int doCount(int n, boolean[] used, int d) {
  if (d == n)
    return 1;
```

```
    int total = 1;
    for (int i = (d == 0) ? 1 : 0; i <= 9; i++) {
      if (!used[i]) {
        used[i] = true;
        total += doCount(n, used, d + 1);
        used[i] = false;
      }
    }
    return total;
  }
```

这种解法稍微有一点难度，需要有一定的算法基础知识，否则不容易看懂。他使用递归加回溯的方式，used[i]表示i这个数字被使用过了，当然这种效率很差，但也是一种思路。