力扣加加

努力做西湖区最好的算法题解

目录

Introduction 1.1

算法专题 1.2

- 数据结构 1.2.1
- 基础算法 1.2.2
- 二叉树的遍历 1.2.3
- 动态规划 1.2.4
- 哈夫曼编码和游程编码 1.2.5
- 布隆过滤器 1.2.6
- 字符串问题 1.2.7
- 前缀树专题 1.2.8
- 《贪婪策略》专题 1.2.9
- 《深度优先遍历》专题 1.2.10
- 滑动窗口（思路 + 模板） 1.2.11
- 位运算 1.2.12
- 设计题 1.2.13
- 小岛问题 1.2.14
- 最大公约数 1.2.15
- 并查集 1.2.16
- 前缀和 1.2.17
- 平衡二叉树专题 1.2.18

91 天学算法 1.3

- 第一期讲义-二分法 1.3.1
- 第一期讲义-双指针 1.3.2
- 第二期 1.3.3

精选题解 1.4

- 《日程安排》专题 1.4.1
- 《构造二叉树》专题 1.4.2
- 字典序列删除 1.4.3
- 百度的算法面试题 * 祖玛游戏 1.4.4
- 西法带你学算法】一次搞定前缀和 1.4.5
- 字节跳动的算法面试题是什么难度? 1.4.6
- 字节跳动的算法面试题是什么难度？（第二弹） 1.4.7

《我是你的妈妈呀》 * 第一期 1.4.8
一文带你看懂二叉树的序列化 1.4.9
穿上衣服我就不认识你了？来聊聊最长上升子序列 1.4.10
你的衣服我扒了 * 《最长公共子序列》 1.4.11
一文看懂《最大子序列和问题》 1.4.12
高频考题（简单） 1.5
面试题 17.12. BiNode 1.5.1
0001. 两数之和 1.5.2
0020. 有效的括号 1.5.3
0021. 合并两个有序链表 1.5.4
0026. 删除排序数组中的重复项 1.5.5
0053. 最大子序和 1.5.6
0088. 合并两个有序数组 1.5.7
0101. 对称二叉树 1.5.8
0104. 二叉树的最大深度 1.5.9
0108. 将有序数组转换为二叉搜索树 1.5.10
0121. 买卖股票的最佳时机 1.5.11
0122. 买卖股票的最佳时机 II 1.5.12
0125. 验证回文串 1.5.13
0136.single-number 1.5.14
0155.min-stack 1.5.15
0167.two-sum-ii-input-array-is-sorted 1.5.16
0169.majority-element 1.5.17
0172.factorial-trailing-zeroes 1.5.18
0190.reverse-bits 1.5.19
0191.number-of-1-bits 1.5.20
0198.house-robber 1.5.21
0203.remove-linked-list-elements 1.5.22
0206.reverse-linked-list 1.5.23
0219.contains-duplicate-ii 1.5.24
0226.invert-binary-tree 1.5.25
0232.implement-queue-using-stacks 1.5.26
0263.ugly-number 1.5.27
0283.move-zeroes 1.5.28
0342.power-of-four 1.5.29

0349.intersection-of-two-arrays 1.5.30
0371.sum-of-two-integers 1.5.31
0437.path-sum-iii 1.5.32
0455.AssignCookies 1.5.33
0575.distribute-candies 1.5.34
0874.walking-robot-simulation 1.5.35
1260.shift-2d-grid 1.5.36
1332.remove-palindromic-subsequences 1.5.37
高频考题（中等） 1.6
面试题 17.09. 第 k 个数 1.6.1
0002.add-two-numbers 1.6.2
0003.longest-substring-without-repeating-characters 1.6.3
0005.longest-palindromic-substring 1.6.4
0011.container-with-most-water 1.6.5
0015.3-sum 1.6.6
0017.Letter-Combinations-of-a-Phone-Number 1.6.7
0019. Remove Nth Node From End of List 1.6.8
0022.generate-parentheses.md 1.6.9
0024. Swap Nodes In Pairs 1.6.10
0029.divide-two-integers 1.6.11
0031.next-permutation 1.6.12
0033.search-in-rotated-sorted-array 1.6.13
0039.combination-sum 1.6.14
0040.combination-sum-ii 1.6.15
0046.permutations 1.6.16
0047.permutations-ii 1.6.17
0048.rotate-image 1.6.18
0049.group-anagrams 1.6.19
0050.pow-x-n 1.6.20
0055.jump-game 1.6.21
0056.merge-intervals 1.6.22
0060.permutation-sequence 1.6.23
0062.unique-paths 1.6.24
0073.set-matrix-zeroes 1.6.25
0075.sort-colors 1.6.26

0078.subsets	1.6.27
0079.word-search	1.6.28
0080.remove-duplicates-from-sorted-array-ii	1.6.29
0086.partition-list	1.6.30
0090.subsets-ii	1.6.31
0091.decode-ways	1.6.32
0092.reverse-linked-list-ii	1.6.33
0094.binary-tree-inorder-traversal	1.6.34
0095.unique-binary-search-trees-ii	1.6.35
0096.unique-binary-search-trees	1.6.36
0098.validate-binary-search-tree	1.6.37
0102.binary-tree-level-order-traversal	1.6.38
0103.binary-tree-zigzag-level-order-traversal	1.6.39
105.Construct-Binary-Tree-from-Preorder-and-Inorder-Traversal.md	1.6.40
0113.path-sum-ii	1.6.41
0129.sum-root-to-leaf-numbers	1.6.42
0130.surrounded-regions	1.6.43
0131.palindrome-partitioning	1.6.44
0139.word-break	1.6.45
0144.binary-tree-preorder-traversal	1.6.46
0150.evaluate-reverse-polish-notation	1.6.47
0152.maximum-product-subarray	1.6.48
0199.binary-tree-right-side-view	1.6.49
0200.number-of-islands	1.6.50
0201.bitwise-and-of-numbers-range	1.6.51
0208.implement-trie-prefix-tree	1.6.52
0209.minimum-size-subarray-sum	1.6.53
0211.add-and-search-word-data-structure-design	1.6.54
0215.kth-largest-element-in-an-array	1.6.55
0221.maximal-square	1.6.56
0229.majority-element-ii	1.6.57
0230.kth-smallest-element-in-a-bst	1.6.58
0236.lowest-common-ancestor-of-a-binary-tree	1.6.59
0238product-of-array-except-self	1.6.60
0240.search-a-2-d-matrix-ii	1.6.61

0279.perfect-squares	1.6.62
0309.best-time-to-buy-and-sell-stock-with-cooldown	1.6.63
0322.coin-change	1.6.64
0328.odd-even-linked-list	1.6.65
0334.increasing-triplet-subsequence	1.6.66
0337.house-robber-iii.md	1.6.67
0343.integer-break	1.6.68
0365.water-and-jug-problem	1.6.69
0378.kth-smallest-element-in-a-sorted-matrix	1.6.70
0380.insert-delete-getrandom-o1	1.6.71
0416.partition-equal-subset-sum	1.6.72
0445.add-two-numbers-ii	1.6.73
0454.4-sum-ii	1.6.74
0474.ones-and-zeros	1.6.75
0494.target-sum	1.6.76
0516.longest-palindromic-subsequence	1.6.77
0518.coin-change-2	1.6.78
0547.friend-circles	1.6.79
0560.subarray-sum-equals-k	1.6.80
0609.find-duplicate-file-in-system	1.6.81
0611.valid-triangle-number	1.6.82
0718.maximum-length-of-repeated-subarray	1.6.83
0754.reach-a-number.md	1.6.84
0785.is-graph-bipartite	1.6.85
0820.short-encoding-of-words	1.6.86
0875.koko-eating-bananas	1.6.87
0877.stone-game	1.6.88
0886.possible-bipartition	1.6.89
0900.rle-iterator	1.6.90
0912.sort-an-array	1.6.91
0935.knight-dialer	1.6.92
1011.capacity-to-ship-packages-within-d-days	1.6.93
1014.best-sightseeing-pair	1.6.94
1015.smallest-integer-divisible-by-k	1.6.95
1019.next-greater-node-in-linked-list	1.6.96

1020.number-of-enclaves 1.6.97
1023.camelcase-matching 1.6.98
1031.maximum-sum-of-two-non-overlapping-subarrays 1.6.99
1104.path-in-zigzag-labelled-binary-tree 1.6.100
1131.maximum-of-absolute-value-expression 1.6.101
1186.maximum-subarray-sum-with-one-deletion 1.6.102
1218.longest-arithmetic-subsequence-of-given-difference 1.6.103
1227.airplane-seat-assignment-probability 1.6.104
1261.find-elements-in-a-contaminated-binary-tree 1.6.105
1262.greatest-sum-divisible-by-three 1.6.106
1297.maximum-number-of-occurrences-of-a-substring 1.6.107
1310.xor-queries-of-a-subarray 1.6.108
1334.find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance 1.6.109
1371.find-the-longest-substring-containing-vowels-in-even-counts 1.6.110
高频考题（困难） 1.7
0004.median-of-two-sorted-array 1.7.1
0023.merge-k-sorted-lists 1.7.2
0025.reverse-nodes-in-k-group 1.7.3
0030.substring-with-concatenation-of-all-words 1.7.4
0032.longest-valid-parentheses 1.7.5
0042.trapping-rain-water 1.7.6
0052.N-Queens-II 1.7.7
0084.largest-rectangle-in-histogram 1.7.8
0085.maximal-rectangle 1.7.9
0124.binary-tree-maximum-path-sum 1.7.10
0128.longest-consecutive-sequence 1.7.11
0145.binary-tree-postorder-traversal 1.7.12
0212.word-search-ii 1.7.13
0239.sliding-window-maximum 1.7.14
0295.find-median-from-data-stream 1.7.15
0301.remove-invalid-parentheses 1.7.16
0312.burst-balloons 1.7.17
0335.self-crossPing 1.7.18
0460.lfu-cache 1.7.19

0472.concatenated-words 1.7.20
0488.zuma-game.md 1.7.21
0493.reverse-pairs 1.7.22
0887.super-egg-drop 1.7.23
0895.maximum-frequency-stack 1.7.24
1032.stream-of-characters 1.7.25
1168.optimize-water-distribution-in-a-village 1.7.26
1449.form-largest-integer-with-digits-that-add-up-to-target 1.7.27
后序 1.8

LeetCode

简体中文 | English

- 2019-07-10： 纪念项目 Star 突破 1W 的一个短文， 记录了项目的"兴起"之路，大家有兴趣可以看一下，如果对这个项目感兴趣，请**点击一下 Star**， 项目会**持续更新**，感谢大家的支持。
- 2019-10-08: 纪念 LeetCode 项目 Star 突破 2W，并且 Github 搜索 “LeetCode”，排名第一。
- 2020-04-12: 项目突破三万 Star。
- 2020-04-14: 官网 `力扣加加` 上线啦 💐 💐 💐 💐 💐 ，有专题讲解，每日一题，下载区和视频题解，后续会增加更多内容，还不赶紧收藏起来？地址：http://leetcode-solution.cn/

前言

这是我将我的所有公开的算法资料整理的一个电子书，全部题目信息中文化，以前会有一些英文描述，感谢 @CYL 的中文整理。

由于是电子书，因此阅读体验可能会更好， 但是相应地就不能获得及时的更新，因此你可以收藏一下我的同步电子书的网站 西法带你学算法 - 在线版。后期可能将每日一题， 91 天学算法其他章节的讲义等也整理进来。

介绍

leetcode 题解，记录自己的 leetcode 解题之路。

本仓库目前分为**五个**部分：

- 第一个部分是 leetcode 经典题目的解析，包括思路，关键点和具体的代码实现。
- 第二部分是对于数据结构与算法的总结
- 第三部分是 anki 卡片， 将 leetcode 题目按照一定的方式记录在 anki 中，方便大家记忆。
- 第四部分是每日一题，每日一题是在交流群（包括微信和 qq）里进行的一种活动，大家一起 解一道题，这样讨论问题更加集中，会得到更多的反馈。而且 这些题目可以被记录下来，日后会进行筛选添加到仓库的题解模块。
- 第五部分是计划， 这里会记录将来要加入到以上三个部分内容

> 只有熟练掌握基础的数据结构与算法，才能对复杂问题迎刃有余。

非科学人士看过来

如果是国内的非科学用户，可以使用 https://lucifer.ren/leetcode ，整站做了静态化，速度贼快！但是阅读体验可能一般，大家也可以访问力扣加加（暂时没有静态化）获得更好的阅读体验。

另外需要科学的，我推荐一个工具， 用户体验真的是好，用起来超简单， 提供一站式工具，包括网络检测工具，浏览器插件等，支持多种客户端（还有我最喜欢的 Switch 加速器），价格也不贵，基础套餐折算到月大约 11.2 块/月。它还支持签到送天数，也就是说你可以每天签到无限续期。地址：https://glados.space/landing/M9OHH-Q88JQ-DX72D-R04RN

怎么刷 LeetCode?

- 我是如何刷 LeetCode 的
- 算法小白如何高效、快速刷 leetcode?

刷题插件

- 刷题效率低？或许你就差这么一个插件
- 力扣刷题插件

91 天学算法

- 91 天，遇见不一样的自己

食用指南

- 我对题目难度进行了分类的保留，因此你可以根据自己的情况刷。我推荐大家从简单开始，逐步加大难度，直到困难。
- 这里有一张互联网公司面试中经常考察的问题类型总结的思维导图，我们可以结合图片中的信息分析一下。

（图片来自 leetcode）

其中算法，主要是以下几种：

- 基础技巧：分治、二分、贪心
- 排序算法：快速排序、归并排序、计数排序

- 搜索算法：回溯、递归、深度优先遍历，广度优先遍历，二叉搜索树等
- 图论：最短路径、最小生成树
- 动态规划：背包问题、最长子序列

数据结构，主要有如下几种：

- 数组与链表：单 / 双向链表
- 栈与队列
- 哈希表
- 堆：最大堆 / 最小堆
- 树与图：最近公共祖先、并查集
- 字符串：前缀树（字典树） / 后缀树

精彩预告

0042.trapping-rain-water:

0547.friend-circles:

backtrack problems:

0198.house-robber:

0454.4-sum-ii:

anki 卡片

Anki 主要分为两个部分：一部分是关键点到题目的映射，另一部分是题目到思路，关键点，代码的映射。

全部卡片都在 anki-card

使用方法：

anki - 文件 - 导入 - 下拉格式选择“打包的 anki 集合”，然后选中你下载好的文件，确定即可。

更多关于 anki 使用方法的请查看 anki 官网

目前已更新卡片一览（仅列举正面）：

- 二分法解决问题的关键点是什么，相关问题有哪些?
- 如何用栈的特点来简化操作， 涉及到的题目有哪些?
- 双指针问题的思路以及相关题目有哪些?
- 滑动窗口问题的思路以及相关题目有哪些?
- 回溯法解题的思路以及相关题目有哪些?
- 数论解决问题的关键点是什么，相关问题有哪些?
- 位运算解决问题的关键点是什么，相关问题有哪些?

> 已加入的题目有：#2 #3 #11

每日一题

每日一题是在交流群（包括微信和 qq）里通过 issues 来进行的一种活动，大家一起 解一道题，这样讨论问题更加集中，会得到更多的反馈。而且 这些题目可以被记录下来，日后会进行筛选添加到仓库的题解模块。

- 每日一题汇总

- 每日一题认领区

计划

- LeetCode 换皮题目集锦
- 动态规划完善。最长递增子序列，最长回文子序列，编辑距离等“字符串”题目， 扔鸡蛋问题。 解题模板，滚动数组。
- 堆可以解决的题目。 手写堆
- 单调栈
- BFS & DFS

哪里能找到我?

点关注，不迷路。如果再给 ✚ 个星标就更棒啦！

> 关注加加，星标加加～

关于我

擅长前端工程化，前端性能优化，前端标准化等，做过。net， 搞过 Java，现在是一名前端工程师，我的个人博客：https://lucifer.ren/blog/

我经常会在开源社区进行一些输出和分享，比较受欢迎的有 宇宙最强的前端面试指南 和 我的第一本小书。目前本人正在写一本关于《leetcode 题解》的实体书，感兴趣的可以通过邮箱或者微信联系我，我会在出版的第一时间通知你，并给出首发优惠价。有需要可以直接群里联系我，或者发送到我的个人邮箱 [azl397985856@gmail.com]。 新书详情戳这里：《或许是一本可以彻底改变你刷 LeetCode 效率的题解书》

贡献

- 如果有想法和创意，请提 issue 或者进群提
- 如果想贡献增加题解或者翻译， 可以参考贡献指南

  > 关于如何提交题解，我写了一份 指南

- 如果需要修改项目中图片，这里 存放了项目中绘制图的源代码， 大家可以用 draw.io 打开进行编辑。

鸣谢

感谢为这个项目作出贡献的所有 小伙伴

License

CC BY-NC-ND 4.0

算法专题

以下是一些我总结的类型题目，提前搞懂这些东西对之后的做题很有帮助，强烈建议先掌握。另外我的 91 天学算法也对专题进行了更细粒度的整理，具体参见91 天学算法

首先基础的数据结构大家是必须掌握的，其次就是暴力法。暴力法也是算法，只不过我们追求的肯定是性能更好的算法。因此了解暴力法的算法瓶颈以及各种数据结构的特点就很重要， 这样你可以可以根据这些知识去一步步逼近最优解。

再之后就是必须掌握的算法。比如搜索算法是必须掌握的，搜索算法的范围很广，但是核心就是搜索的，不同的算法在于搜索的方式不同，典型的就是 BFS 和 DFS，当然二分法本质上也是一种搜索算法。

还有就是暴力优化法也是必须掌握的，和搜索一样，范围很广。 有剪枝， 空间换时间等。 其中空间换时间又有很多，比如哈希表， 前缀树等等。

围绕这个思想去学习， 就不会差太多，其他我就不多说，大家慢慢体会。

- 数据结构
- 二叉树的遍历
- 动态规划
- 哈夫曼编码和游程编码
- 布隆过滤器
- 字符串问题
- 前缀树专题
- 《贪婪策略》专题
- 《深度优先遍历》专题
- 滑动窗口（思路 + 模板）
- 位运算
- 设计题
- 小岛问题
- 最大公约数
- 并查集
- 前缀和
- 平衡二叉树专题

基础的数据结构

这篇文章不是讲解数据结构的文章，而是结合现实的场景帮助大家 `理解和复习` 数据结构与算法， 如果你的数据结构基础很差，建议先去看一些基础教程，再转过来看。

本篇文章的定位是侧重于前端的，通过学习前端中实际场景的数据结构，从而加深大家对数据结构的理解和认识。

线性结构

数据结构我们可以从逻辑上分为线性结构和非线性结构。线性结构有 数组，栈，链表等， 非线性结构有树，图等。

> 其实我们可以称树为一种半线性结构。

需要注意的是，线性和非线性不代表存储结构是线性的还是非线性的，这两者没有任何关系，它只是一种逻辑上的划分。 比如我们可以用数组去存储二叉树。

一般而言，有前驱和后继的就是线性数据结构。比如数组和链表。其实一叉树就是链表。

数组

数组是最简单的数据结构了，很多地方都用到它。 比如有一个数据列表等，用它是再合适不过了。 其实后面的数据结构很多都有数组的影子。

我们之后要讲的栈和队列其实都可以看成是一种 `受限` 的数组, 怎么个受限法呢？我们后面讨论。

我们来讲几个有趣的例子来加深大家对数组这种数据结构的理解。

React Hooks

Hooks 的本质就是一个数组， 伪代码：

```
let hooks = null;

export function useHook() {
  hooks.push(hookData);
}

function reactsInternalRenderAComponentMethod(component) {
  hooks = [];
  component();
  let hooksForThisComponent = hooks;
  hooks = null;
}
```

那么为什么 hooks 要用数组？ 我们可以换个角度来解释，如果不用数组会怎么样?

```
function Form() {
  // 1. Use the name state variable
  const [name, setName] = useState("Mary");

  // 2. Use an effect for persisting the form
  useEffect(function persistForm() {
    localStorage.setItem("formData", name);
  });

  // 3. Use the surname state variable
  const [surname, setSurname] = useState("Poppins");

  // 4. Use an effect for updating the title
  useEffect(function updateTitle() {
    document.title = name + " " + surname;
  });

  // ...
}
```

基于数组的方式，Form 的 hooks 就是 [hook1, hook2, hook3, hook4], 我们可以得出这样的关系， hook1 就是[name, setName] 这一对， hook2 就是 persistForm 这个。

如果不用数组实现，比如对象，Form 的 hooks 就是

```
{
  'key1': hook1,
  'key2': hook2,
  'key3': hook3,
  'key4': hook4,
}
```

那么问题是 key1，key2，key3，key4 怎么取呢?

关于 React hooks 的本质研究，更多请查看React hooks: not magic, just arrays

React 将 `如何确保组件内部hooks保存的状态之间的对应关系` 这个工作交给了 开发人员去保证，即你必须保证 HOOKS 的顺序严格一致，具体可以看 React 官网关于 Hooks Rule 部分。

队列

队列是一种受限的序列，它只能够操作队尾和队首，并且只能只能在队尾添加元素，在队首删除元素。

队列作为一种最常见的数据结构同样有着非常广泛的应用， 比如消息队列

> "队列"这个名称,可类比为现实生活中排队（不插队的那种）

在计算机科学中, 一个 队列(queue) 是一种特殊类型的抽象数据类型或集合。集合中的实体按顺序保存。

队列基本操作有两种:

- 向队列的后端位置添加实体，称为入队
- 从队列的前端位置移除实体，称为出队。

队列中元素先进先出 FIFO (first in, first out)的示意：

(图片来自 https://github.com/trekhleb/javascript-algorithms/blob/master/src/data-structures/queue/README.zh-CN.md)

我们前端在做性能优化的时候，很多时候会提到的一点就是“HTTP 1.1 的队头阻塞问题”，具体来说 就是 HTTP2 解决了 HTTP1.1 中的队头阻塞问题，但是为什么 HTTP1.1 有队头阻塞问题，HTTP2 究竟怎么解决的很多人都不清楚。

其实“队头阻塞”是一个专有名词，不仅仅这里有，交换器等其他都有这个问题，引起这个问题的根本原因是使用了 `队列` 这种数据结构。

对于同一个 tcp 连接，所有的 http1.0 请求放入队列中，只有前一个 `请求的响应` 收到了，然后才能发送下一个请求，这个阻塞主要发生在客户端。

这就好像我们在等红绿灯，即使旁边绿灯亮了，你的这个车道是红灯，你还是不能走，还是要等着。

`HTTP/1.0` 和 `HTTP/1.1`：

在 `HTTP/1.0` 中每一次请求都需要建立一个 TCP 连接，请求结束后立即断开连接。 在 `HTTP/1.1` 中，每一个连接都默认是长连接(persistent connection)。对于同一个 tcp 连接，允许一次发送多个 http1.1 请求，也就是说，不必等前一个响应收到，就可以发送下一个请求。这样就解决了 http1.0 的客户端的队头阻塞，而这也就是 `HTTP/1.1` 中 `管道(Pipeline)` 的概念了。 但是， `http1.1规定，服务器端的响应的发送要根据请求被接收的顺序排队` ，也就是说，先接收到的请求的响应也要先发送。这样造成的问题是，如果最先收到的请求的处理时间长的话，响应生成也慢，就会阻塞已经生成了的响应的发送。也会造成队头阻塞。 可见，http1.1 的队首阻塞发生在服务器端。

如果用图来表示的话，过程大概是：

`HTTP/2` 和 `HTTP/1.1`：

为了解决 `HTTP/1.1` 中的服务端队首阻塞， `HTTP/2` 采用了 `二进制分帧` 和 `多路复用` 等方法。

`二进制分帧` 中，帧是 `HTTP/2` 数据通信的最小单位。在 `HTTP/1.1` 数据

包是文本格式，而 `HTTP/2` 的数据包是二进制格式的，也就是二进制帧。采用帧可以将请求和响应的数据分割得更小，且二进制协议可以更高效解析。 `HTTP/2` 中，同域名下所有通信都在单个连接上完成，该连接可以承载任意数量的双向数据流。每个数据流都以消息的形式发送，而消息又由一个或多个帧组成。多个帧之间可以乱序发送，根据帧首部的流标识可以重新组装。 `多路复用` 用以替代原来的序列和拥塞机制。
在 `HTTP/1.1` 中，并发多个请求需要多个 TCP 链接，且单个域名有 6-8 个 TCP 链接请求限制。在 `HTTP/2` 中，同一域名下的所有通信在单个链接完成，仅占用一个 TCP 链接，且在这一个链接上可以并行请求和响应，互不干扰。

> 此网站可以直观感受 `HTTP/1.1` 和 `HTTP/2` 的性能对比。

栈

栈也是一种受限的序列，它只能够操作栈顶，不管入栈还是出栈，都是在栈顶操作。

在计算机科学中, 一个 栈(stack) 是一种抽象数据类型,用作表示元素的集合,具有两种主要操作:

push, 添加元素到栈的顶端(末尾); pop, 移除栈最顶端(末尾)的元素. 以上两种操作可以简单概括为“后进先出(LIFO = last in, first out)”。

此外,应有一个 peek 操作用于访问栈当前顶端(末尾)的元素。（只返回不弹出）

> "栈"这个名称,可类比于一组物体的堆叠(一摞书,一摞盘子之类的)。

栈的 push 和 pop 操作的示意:

(图片来自 https://github.com/trekhleb/javascript-algorithms/blob/master/src/data-structures/stack/README.zh-CN.md)

栈在很多地方都有着应用，比如大家熟悉的浏览器就有很多栈，其实浏览器的执行栈就是一个基本的栈结构，从数据结构上说，它就是一个栈。这也就解释了，我们用递归的解法和用循环+栈的解法本质上是差不多。

比如如下 JS 代码：

```
function bar() {
  const a = 1;
  const b = 2;
  console.log(a, b);
}
function foo() {
  const a = 1;
  bar();
}

foo();
```

真正执行的时候，内部大概是这样的：

> 我画的图没有画出执行上下文中其他部分（this 和 scope 等）， 这部分是闭包的关键，而我这里不是讲闭包的，是为了讲解栈的。
>
> 社区中有很多“执行上下文中的 scope 指的是执行栈中父级声明的变量”说法，这是完全错误的， JS 是词法作用域，scope 指的是函数定义时候的父级，和执行没关系

栈常见的应用有进制转换，括号匹配，栈混洗，中缀表达式（用的很少），后缀表达式（逆波兰表达式）等。

> 合法的栈混洗操作，其实和合法的括号匹配表达式之间存在着一一对应的关系， 也就是说 n 个元素的栈混洗有多少种，n 对括号的合法表达式就有多少种。感兴趣的可以查找相关资料

链表

链表是一种最基本数据结构，熟练掌握链表的结构和常见操作是基础中的基础。

(图片来自： https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/linked-list/traversal)

React Fiber

很多人都说 fiber 是基于链表实现的，但是为什么要基于链表呢，可能很多人并没有答案，那么我觉得可以把这两个点（fiber 和链表）放到一起来讲下。

fiber 出现的目的其实是为了解决 react 在执行的时候是无法停下来的，需要一口气执行完的问题的。

图片来自 Lin Clark 在 ReactConf 2017 分享

上面已经指出了引入 fiber 之前的问题，就是 react 会阻止优先级高的代码（比如用户输入）执行。因此 fiber 打算自己自建一个 `虚拟执行栈` 来解决这个问题，这个虚拟执行栈的实现是链表。

Fiber 的基本原理是将协调过程分成小块，一次执行一块，然乎将运算结果保存起来，并判断是否有时间（react 自己实现了一个类似 requestIdleCallback 的功能）继续执行下一块。 如果有时间，则继续。否则跳出，让浏览器主线程歇一会，执行别的优先级高的代码。

当协调过程完成（所有的小块都运算完毕）， 那么就会进入提交阶段，真正的进行副作用（side effect）操作，比如更新 DOM，这个过程是没有办法取消的，原因就是这部分有副作用。

问题的关键就是将协调的过程划分为一块块的，最后还可以合并到一起，有点像 Map / Reduce。

React 必须重新实现遍历树的算法，从依赖于 `内置堆栈的同步递归模型` ，变为 `具有链表和指针的异步模型` 。

> Andrew 是这么说的： 如果你只依赖于[内置]调用堆栈，它将继续工作直到堆栈为空。。。

如果我们可以随意中断调用堆栈并手动操作堆栈帧，那不是很好吗？ 这就是 React Fiber 的目的。 `Fiber 是堆栈的重新实现，专门用于 React 组件` 。 你可以将单个 Fiber 视为一个 `虚拟堆栈帧` 。

react fiber 大概是这样的：

```
let fiber = {
  tag: HOST_COMPONENT,
  type: "div",
  return: parentFiber,
  children: childFiber,
  sibling: childFiber,
  alternate: currentFiber,
  stateNode: document.createElement("div"),
  props: { children: [], className: "foo" },
  partialState: null,
  effectTag: PLACEMENT,
  effects: []
};
```

从这里可以看出 fiber 本质上是个对象，使用 parent，child，sibling 属性去构建 fiber 树来表示组件的结构树， return, children, sibling 也都是一个 fiber，因此 fiber 看起来就是一个链表。

> 细心的朋友可能已经发现了， alternate 也是一个 fiber， 那么它是用来做什么的呢？ 它其实原理有点像 git， 可以用来执行 git revert ,git commit 等操作，这部分挺有意思，我会在我的《从零开发 git》中讲解

想要了解更多的朋友可以看这个文章

如果可以翻墙， 可以看英文原文

这篇文章也是早期讲述 fiber 架构的优秀文章

我目前也在写关于《从零开发 react 系列教程》中关于 fiber 架构的部分，如果你对具体实现感兴趣，欢迎关注。

非线性结构

那么有了线性结构，我们为什么还需要非线性结构呢？ 答案是为了高效地兼顾静态操作和动态操作。大家可以对照各种数据结构的各种操作的复杂度来直观感受一下。

树

树的应用同样非常广泛，小到文件系统，大到因特网，组织架构等都可以表示为树结构，而在我们前端眼中比较熟悉的 DOM 树也是一种树结构，而 HTML 作为一种 DSL 去描述这种树结构的具体表现形式。如果你接触过 AST，那么 AST 也是一种树，XML 也是树结构。。。树的应用远比大多数人想象的要得多。

树其实是一种特殊的 `图` ，是一种无环连通图，是一种极大无环图，也是一种极小连通图。

从另一个角度看，树是一种递归的数据结构。而且树的不同表示方法，比如不常用的 `长子 + 兄弟` 法，对于 你理解树这种数据结构有着很大用处， 说是一种对树的本质的更深刻的理解也不为过。

树的基本算法有前中后序遍历和层次遍历，有的同学对前中后这三个分别具体表现的访问顺序比较模糊，其实当初我也是一样的，后面我学到了一点，你只需要记住： `所谓的前中后指的是根节点的位置，其他位置按照先左后右排列即可` 。比如前序遍历就是 `根左右` , 中序就是 `左根右` ，后序就是 `左右根` ， 很简单吧?

我刚才提到了树是一种递归的数据结构，因此树的遍历算法使用递归去完成非常简单，幸运的是树的算法基本上都要依赖于树的遍历。 但是递归在计算机中的性能一直都有问题，因此掌握不那么容易理解的"命令式地迭代"遍历算法在某些情况下是有用的。如果你使用迭代式方式去遍历的话，可以借助上面提到的 `栈` 来进行，可以极大减少代码量。

> 如果使用栈来简化运算，由于栈是 FILO 的，因此一定要注意左右子树的推入顺序。

树的重要性质：

- 如果树有 n 个顶点，那么其就有 n - 1 条边，这说明了树的顶点数和边数是同阶的。
- 任何一个节点到根节点存在 `唯一` 路径, 路径的长度为节点所处的深度

实际使用的树有可能会更复杂，比如使用在游戏中的碰撞检测可能会用到四叉树或者八叉树。以及 k 维的树结构 `k-d 树` 等。

（图片来自 https://zh.wikipedia.org/wiki/K-d%E6%A0%91）

二叉树

二叉树是节点度数不超过二的树，是树的一种特殊子集，有趣的是二叉树这种被限制的树结构却能够表示和实现所有的树， 它背后的原理正是 `长子 + 兄弟` 法，用邓老师的话说就是 `二叉树是多叉树的特例，但在有根且有序时，其描述能力却足以覆盖后者` 。

> 实际上， 在你使用 `长子 + 兄弟` 法表示树的同时，进行 45 度角旋转即可。

一个典型的二叉树：

标记为 7 的节点具有两个子节点, 标记为 2 和 6; 一个父节点,标记为 2,作为根节点, 在顶部,没有父节点。

(图片来自 https://github.com/trekhleb/javascript-algorithms/blob/master/src/data-structures/tree/README.zh-CN.md)

对于一般的树，我们通常会去遍历，这里又会有很多变种。

下面我列举一些二叉树遍历的相关算法:

- 94.binary-tree-inorder-traversal
- 102.binary-tree-level-order-traversal
- 103.binary-tree-zigzag-level-order-traversal
- 144.binary-tree-preorder-traversal
- 145.binary-tree-postorder-traversal
- 199.binary-tree-right-side-view

相关概念：

- 真二叉树 （所有节点的度数只能是偶数，即只能为 0 或者 2）

另外我也专门开设了二叉树的遍历章节, 具体细节和算法可以去那里查看。

堆

堆其实是一种优先级队列，在很多语言都有对应的内置数据结构，很遗憾 javascript 没有这种原生的数据结构。 不过这对我们理解和运用不会有影响。

堆的特点：

- 在一个 最小堆(min heap) 中, 如果 P 是 C 的一个父级节点, 那么 P 的 key(或 value)应小于或等于 C 的对应值. 正因为此，堆顶元素一定是最小的，我们会利用这个特点求最小值或者第 k 小的值。

- 在一个 最大堆(max heap) 中, P 的 key(或 value)大于 C 的对应值。

需要注意的是优先队列不仅有堆一种，还有更复杂的，但是通常来说，我们会把两者做等价。

相关算法：

- 295.find-median-from-data-stream

二叉查找树

二叉排序树（Binary Sort Tree），又称二叉查找树（Binary Search Tree），亦称二叉搜索树。

二叉查找树具有下列性质的二叉树：

- 若左子树不空，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；
- 左、右子树也分别为二叉排序树；
- 没有键值相等的节点。

对于一个二叉查找树，常规操作有插入，查找，删除，找父节点，求最大值，求最小值。

二叉查找树，之所以叫查找树就是因为其非常适合查找，举个例子， 如下一颗二叉查找树，我们想找节点值小于且最接近 58 的节点，搜索的流程如图所示：

（图片来自 https://www.geeksforgeeks.org/floor-in-binary-search-tree-bst/）

另外我们二叉查找树有一个性质是：`其中序遍历的结果是一个有序数组`。有时候我们可以利用到这个性质。

相关题目：

- 98.validate-binary-search-tree

二叉平衡树

平衡树是计算机科学中的一类数据结构，为改进的二叉查找树。一般的二叉查找树的查询复杂度取决于目标结点到树根的距离（即深度），因此当结点的深度普遍较大时，查询的均摊复杂度会上升。为了实现更高效的查询，产生了平衡树。

在这里，平衡指所有叶子的深度趋于平衡，更广义的是指在树上所有可能查找的均摊复杂度偏低。

一些数据库引擎内部就是用的这种数据结构，其目标也是将查询的操作降低到 logn（树的深度），可以简单理解为`树在数据结构层面构造了二分查找算法`。

基本操作：

- 旋转
- 插入
- 删除

- 查询前驱
- 查询后继

AVL

是最早被发明的自平衡二叉查找树。在 AVL 树中，任一节点对应的两棵子树的最大高度差为 1，因此它也被称为高度平衡树。查找、插入和删除在平均和最坏情况下的时间复杂度都是 O(logn)。增加和删除元素的操作则可能需要借由一次或多次树旋转，以实现树的重新平衡。AVL 树得名于它的发明者 G. M. Adelson-Velsky 和 Evgenii Landis，他们在 1962 年的论文 An algorithm for the organization of information 中公开了这一数据结构。 节点的平衡因子是它的左子树的高度减去它的右子树的高度（有时相反）。带有平衡因子 1、0 或 -1 的节点被认为是平衡的。带有平衡因子 -2 或 2 的节点被认为是不平衡的，并需要重新平衡这个树。平衡因子可以直接存储在每个节点中，或从可能存储在节点中的子树高度计算出来。

红黑树

在 1972 年由鲁道夫·贝尔发明，被称为"对称二叉 B 树"，它现代的名字源于 Leo J. Guibas 和 Robert Sedgewick 于 1978 年写的一篇论文。红黑树的结构复杂，但它的操作有着良好的最坏情况运行时间，并且在实践中高效：它可以在 O(logn) 时间内完成查找，插入和删除，这里的 n 是树中元素的数目

字典树(前缀树)

又称 Trie 树，是一种树形结构。典型应用是用于统计，排序和保存大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是：利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较，查询效率比哈希树高。

(图来自 https://baike.baidu.com/item/%E5%AD%97%E5%85%B8%E6%A0%91/9825209?fr=aladdin) 它有 3 个基本性质：

- 根节点不包含字符，除根节点外每一个节点都只包含一个字符；
- 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串；
- 每个节点的所有子节点包含的字符都不相同。

immutable 与 字典树

`immutableJS` 的底层就是 `share + tree` . 这样看的话，其实和字典树是一致的。

相关算法：

- 208.implement-trie-prefix-tree
- 211.add-and-search-word-data-structure-design
- 212.word-search-ii

图

前面讲的数据结构都可以看成是图的特例。 前面提到了二叉树完全可以实现其他树结构， 其实有向图也完全可以实现无向图和混合图，因此有向图的研究一直是重点考察对象。

图论〔Graph Theory〕是数学的一个分支。它以图为研究对象。图论中的图是由若干给定的点及连接两点的线所构成的图形，这种图形通常用来描述某些事物之间的某种特定关系，用点代表事物，用连接两点的线表示相应两个事物间具有这种关系。

图的表示方法

- 邻接矩阵(常见)

空间复杂度 O(n^2),n 为顶点个数。

优点：

1. 直观，简单。
2. 适用于稠密图
3. 判断两个顶点是否连接，获取入度和出度以及更新度数，时间复杂度都是 O(1)
4. 关联矩阵
5. 邻接表

对于每个点，存储着一个链表，用来指向所有与该点直接相连的点 对于有权图来说，链表中元素值对应着权重

例如在无向无权图中：

（图片来自 https://zhuanlan.zhihu.com/p/25498681）

可以看出在无向图中，邻接矩阵关于对角线对称，而邻接链表总有两条对称的边 而在有向无权图中：

（图片来自 https://zhuanlan.zhihu.com/p/25498681）

图的遍历

图的遍历就是要找出图中所有的点，一般有以下两种方法：

1. 深度优先遍历：(Depth First Search, DFS)

深度优先遍历图的方法是，从图中某顶点 v 出发， 不断访问邻居， 邻居的邻居直到访问完毕。

1. 广度优先搜索：(Breadth First Search, BFS)

广度优先搜索，可以被形象地描述为 "浅尝辄止"，它也需要一个队列以保持遍历过的顶点顺序，以便按出队的顺序再去访问这些顶点的邻接顶点。

基础算法

时间复杂度分析

算法思想

贪心算法

分治

动态规划

回溯法

枚举法

元算法

排序算法

排序算法是一种元算法，直接考的很少，但是思想还是有用的。

O(n^2)

- 插入排序
- 选择排序
- 希尔排序
- 冒泡排序

O(nlogn)

- 快排
- 归并排序
- 堆排序

O(n)

- 桶排序
- 计数排序
- 基数排序

查找算法

- 线性查找
- 树查找
- 散列表查找

字符串问题

- 朴素
- KMP
- RK
- BM
- trie

其他

- 数论
- 概率论
- 并查集
- 矩阵运算

二叉树的遍历算法

概述

二叉树作为一个基础的数据结构，遍历算法作为一个基础的算法，两者结合当然是经典的组合了。很多题目都会有 ta 的身影，有直接问二叉树的遍历的，有间接问的。比如要你找到树中满足条件的节点，就是间接考察树的遍历，因为你要找到树中满足条件的点，就需要进行遍历。

> 你如果掌握了二叉树的遍历，那么也许其他复杂的树对于你来说也并不遥远了

二叉数的遍历主要有前中后遍历和层次遍历。 前中后属于 DFS，层次遍历属于 BFS。 DFS 和 BFS 都有着自己的应用，比如 leetcode 301 号问题和 609 号问题。

DFS 都可以使用栈来简化操作，并且其实树本身是一种递归的数据结构，因此递归和栈对于 DFS 来说是两个关键点。

DFS 图解：

(图片来自 https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/tree/depth-first-search)

BFS 的关键点在于如何记录每一层次是否遍历完成， 我们可以用一个标识位来表式当前层的结束。

首先不管是前中还是后序遍历，变的只是根节点的位置， 左右节点的顺序永远是先左后右。 比如前序遍历就是根在前面，即根左右。中序就是根在中间，即左根右。后序就是根在后面，即左右根。

下面我们依次讲解：

前序遍历

相关问题144.binary-tree-preorder-traversal

前序遍历的顺序是 `根-左-右`

思路是：

1. 先将根结点入栈
2. 出栈一个元素，将右节点和左节点依次入栈
3. 重复 2 的步骤

总结： 典型的递归数据结构，典型的用栈来简化操作的算法。

其实从宏观上表现为： `自顶向下依次访问左侧链，然后自底向上依次访问右侧链` ， 如果从这个角度出发去写的话，算法就不一样了。从上向下我们可以直接递归访问即可，从下向上我们只需要借助栈也可以轻易做到。 整个过程大概是这样：

这种思路解题有点像我总结过的一个解题思路 `backtrack` - 回溯法。这种思路有一个好处就是 可以 `统一三种遍历的思路` . 这个很重要，如果不了解的朋友，希望能够记住这一点。

中序遍历

相关问题94.binary-tree-inorder-traversal

中序遍历的顺序是 `左–根–右` ，根节点不是先输出，这就有一点点复杂了。

1. 根节点入栈
2. 判断有没有左节点，如果有，则入栈，直到叶子节点

> 此时栈中保存的就是所有的左节点和根节点。

1. 出栈，判断有没有右节点，有则入栈，继续执行 2

值得注意的是，中序遍历一个二叉查找树（BST）的结果是一个有序数组，利用这个性质有些题目可以得到简化， 比如230.kth-smallest-element-in-a-bst， 以及98.validate-binary-search-tree

后序遍历

相关问题145.binary-tree-postorder-traversal

后序遍历的顺序是 `左–右–根`

这个就有点难度了，要不也不会是 leetcode 困难的 难度啊。

其实这个也是属于根节点先不输出，并且根节点是最后输出。 这里可以采用一种讨巧的做法， 就是记录当前节点状态，如果 1. 当前节点是叶子节点或者 2.当前节点的左右子树都已经遍历过了，那么就可以出栈了。

对于 1. 当前节点是叶子节点，这个比较好判断，只要判断 left 和 rigt 是否同时为 null 就好。

对于 2. 当前节点的左右子树都已经遍历过了， 我们只需要用一个变量记录即可。最坏的情况，我们记录每一个节点的访问状况就好了，空间复杂度 O(n) 但是仔细想一下，我们使用了栈的结构，从叶子节点开始输出，我们记录一个当前出栈的元素就好了，空间复杂度 O(1)， 具体请查看上方链接。

层次遍历

层次遍历的关键点在于如何记录每一层次是否遍历完成， 我们可以用一个标识位来表式当前层的结束。

(图片来自 https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/tree/breadth-first-search)

具体做法：

1. 根节点入队列， 并入队列一个特殊的标识位，此处是 null
2. 出队列
3. 判断是不是 null， 如果是则代表本层已经结束。我们再次判断是否当前队列为空，如果不为空继续入队一个 null，否则说明遍历已经完成，我们什么都不不用做
4. 如果不为 null，说明这一层还没完，则将其左右子树依次入队列。

相关问题：

- 102.binary-tree-level-order-traversal
- 117. 填充每个节点的下一个右侧节点指针 II

双色标记法

我们知道垃圾回收算法中，有一种算法叫三色标记法。 即：

- 用白色表示尚未访问
- 灰色表示尚未完全访问子节点

- 黑色表示子节点全部访问

那么我们可以模仿其思想，使用双色标记法来统一三种遍历。

其核心思想如下：

- 使用颜色标记节点的状态，新节点为白色，已访问的节点为灰色。
- 如果遇到的节点为白色，则将其标记为灰色，然后将其右子节点、自身、左子节点依次入栈。
- 如果遇到的节点为灰色，则将节点的值输出。

使用这种方法实现的中序遍历如下：

```
class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]
        WHITE, GRAY = 0, 1
        res = []
        stack = [(WHITE, root)]
        while stack:
            color, node = stack.pop()
            if node is None: continue
            if color == WHITE:
                stack.append((WHITE, node.right))
                stack.append((GRAY, node))
                stack.append((WHITE, node.left))
            else:
                res.append(node.val)
        return res
```

可以看出，实现上 WHITE 就表示的是递归中的第一次进入过程，Gray 则表示递归中的从叶子节点返回的过程。 因此这种迭代的写法更接近递归写法的本质。

如要实现前序、后序遍历，只需要调整左右子节点的入栈顺序即可。可以看出使用三色标记法， 其写法类似递归的形式，因此便于记忆和书写，缺点是使用了额外的内存空间。不过这个额外的空间是线性的，影响倒是不大。

> 虽然递归也是额外的线性时间，但是递归的栈开销还是比一个 0，1 变量开销大的。

Morris 遍历

我们可以使用一种叫做 Morris 遍历的方法，既不使用递归也不借助于栈。从而在 $O(1)$ 空间完成这个过程。

```
def MorrisTraversal(root):
    curr = root

    while curr:
        # If left child is null, print the
        # current node data. And, update
        # the current pointer to right child.
        if curr.left is None:
            print(curr.data, end= " ")
            curr = curr.right

        else:
            # Find the inorder predecessor
            prev = curr.left

            while prev.right is not None and prev.right is
                prev = prev.right

            # If the right child of inorder
            # predecessor already points to
            # the current node, update the
            # current with it's right child
            if prev.right is curr:
                prev.right = None
                curr = curr.right

            # else If right child doesn't point
            # to the current node, then print this
            # node's data and update the right child
            # pointer with the current node and update
            # the current with it's left child
            else:
                print (curr.data, end=" ")
                prev.right = curr
                curr = curr.left
```

参考： what-is-morris-traversal

相关题目

- lowest-common-ancestor-of-a-binary-tree
- binary-tree-level-order-traversal
- binary-tree-zigzag-level-order-traversal
- validate-binary-search-tree
- maximum-depth-of-binary-tree
- balanced-binary-tree

- binary-tree-level-order-traversal-ii
- binary-tree-maximum-path-sum
- insert-into-a-binary-search-tree

递归和动态规划

动态规划可以理解为是**查表的递归（记忆化）**。那么什么是递归？什么是查表（记忆化）？

递归

递归是指在函数的定义中使用函数自身的方法。

算法中使用递归可以很简单地完成一些用循环实现的功能，比如二叉树的左中右序遍历。递归在算法中有非常广泛的使用，包括现在日趋流行的函数式编程。

有意义的递归算法会把问题分解成规模缩小的同类子问题，当子问题缩减到寻常的时候，就可以知道它的解。然后建立递归函数之间的联系即可解决原问题，这也是我们使用递归的意义。准确来说， 递归并不是算法，它是和迭代对应的一种编程方法。只不过，由于隐式地借助了函数调用栈，因此递归写起来更简单。

一个问题要使用递归来解决必须有递归终止条件（算法的有穷性）。虽然以下代码也是递归，但由于其无法结束，因此不是一个有效的算法：

```
def f(n):
  return n + f(n - 1)
```

更多的情况应该是：

```
def f(n):
  if n == 1: return 1
  return n + f(n - 1)
```

练习递归

一个简单练习递归的方式是将你写的迭代全部改成递归形式。比如你写了一个程序，功能是“将一个字符串逆序输出”，那么使用迭代将其写出来会非常容易，那么你是否可以使用递归写出来呢？通过这样的练习，可以让你逐步适应使用递归来写程序。

如果你已经对递归比较熟悉了，那么我们继续往下看。

递归中的重复计算

递归中可能存在这么多的重复计算，为了消除这种重复计算，一种简单的方式就是记忆化递归。即一边递归一边使用“记录表”（比如哈希表或者数组）记录我们已经计算过的情况，当下次再次碰到的时候，如果之前已经计算了，那么直接返回即可，这样就避免了重复计算。而**动态规划中 DP 数组其实和这里“记录表”的作用是一样的**。

递归的时间复杂度分析

敬请期待我的新书。

小结

使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。这里我列举了几道算法题目，这几道算法题目都可以用递归轻松写出来：

- 递归实现 sum
- 二叉树的遍历
- 走楼梯问题
- 汉诺塔问题
- 杨辉三角

当你已经适应了递归的时候，那就让我们继续学习动态规划吧！

动态规划

如果你已经熟悉了递归的技巧，那么使用递归解决问题非常符合人的直觉，代码写起来也比较简单。这个时候我们来关注另一个问题 - **重复计算**。我们可以通过分析（可以尝试画一个递归树），可以看出递归在缩小问题规模的同时**是否可能会重复计算**。 279.perfect-squares 中 我通过递归的方式来解决这个问题，同时内部维护了一个缓存来存储计算过的运算，这么做可以减少很多运算。 这其实和动态规划有着异曲同工的地方。

> 小提示：如果你发现并没有重复计算，那么就没有必要用记忆化递归或者动态规划了。

因此动态规划就是枚举所以可能。不过相比暴力枚举，动态规划不会有重复计算。因此如何保证枚举时不重不漏是关键点之一。 递归由于使用了函数调用栈来存储数据，因此如果栈变得很大，那么会容易爆栈。

爆栈

我们结合求和问题来讲解一下，题目是给定一个数组，求出数组中所有项的和，要求使用递归实现。

代码：

```
function sum(nums) {
  if (nums.length === 0) return 0;
  if (nums.length === 1) return nums[0];

  return nums[0] + sum(nums.slice(1));
}
```

我们用递归树来直观地看一下。

这种做法本身没有问题，但是每次执行一个函数都有一定的开销，拿 JS 引擎执行 JS 来说，每次函数执行都会进行入栈操作，并进行预处理和执行过程，所以内存会有额外的开销，数据量大的时候很容易造成爆栈。

> 浏览器中的 JS 引擎对于代码执行栈的长度是有限制的，超过会爆栈，抛出异常。

重复计算

我们再举一个重复计算的例子，问题描述：

一个人爬楼梯，每次只能爬 1 个或 2 个台阶，假设有 n 个台阶，那么这个人有多少种不同的爬楼梯方法?

> 746. 使用最小花费爬楼梯 是这道题的换皮题， GrowingIO 前端工程师岗位考察过这个题目。

由于上第 n 级台阶一定是从 n - 1 或者 n - 2 来的，因此 上第 n 级台阶的数目就是 `上 n – 1 级台阶的数目加上 n – 1 级台阶的数目` 。

递归代码：

```
function climbStairs(n) {
  if (n === 1) return 1;
  if (n === 2) return 2;
  return climbStairs(n - 1) + climbStairs(n - 2);
}
```

我们继续用一个递归树来直观感受以下：

> 红色表示重复的计算

可以看出这里面有很多重复计算，我们可以使用一个 hashtable 去缓存中间计算结果，从而省去不必要的计算。

那么动态规划是怎么解决这个问题呢？ 答案也是“查表”，不过区别于递归使用函数调用栈，动态规划通常使用的是 dp 数组，数组的索引通常是问题规模，值通常是递归函数的返回值。 递归是从问题的结果倒推，直到问题的规模缩小到寻常。 动态规划是从寻常入手， 逐步扩大规模到最优子结构。

如果上面的爬楼梯问题，使用动态规划，代码是这样的：

```
function climbStairs(n) {
  if (n == 1) return 1;
  const dp = new Array(n);
  dp[0] = 1;
  dp[1] = 2;

  for (let i = 2; i < n; i++) {
    dp[i] = dp[i - 1] + dp[i - 2];
  }
  return dp[dp.length - 1];
}
```

不会也没关系，我们将递归的代码稍微改造一下。其实就是将函数的名字改一下：

```
function dp(n) {
  if (n === 1) return 1;
  if (n === 2) return 2;
  return dp(n - 1) + dp(n - 2);
}
```

> dp[n] 和 dp(n) 对比看，这样是不是有点理解了呢？只不过递归用调用栈枚举状态， 而动态规划使用迭代枚举状态。

动态规划的查表过程如果画成图，就是这样的：

> 虚线代表的是查表过程

这道题目是动态规划中最简单的问题了，因为设计到单个因素的变化，如果涉及到多个因素，就比较复杂了，比如著名的背包问题，挖金矿问题等。

对于单个因素的，我们最多只需要一个一维数组即可，对于如背包问题我们需要二维数组等更高纬度。

爬楼梯我们并没有必要使用一维数组，而是借助两个变量来实现的，空间复杂度是 O(1)。代码：

```
function climbStairs(n) {
  if (n === 1) return 1;
  if (n === 2) return 2;

  let a = 1;
  let b = 2;
  let temp;

  for (let i = 3; i <= n; i++) {
    temp = a + b;
    a = b;
    b = temp;
  }

  return temp;
}
```

之所以能这么做，是因为爬楼梯问题的状态转移方程中**当前状态只和前两个状态有关**，因此只需要存储这两个即可。 动态规划问题有很多这种讨巧的方式，这个技巧叫做滚动数组。

再次强调一下：

- 如果说递归是从问题的结果倒推，直到问题的规模缩小到寻常。 那么动态规划就是从寻常入手， 逐步扩大规模到最优子结构。
- 记忆化递归和动态规划没有本质不同。都是枚举状态，并根据状态直接的联系逐步推导求解。
- 动态规划性能通常更好。 一方面是递归的栈开销，一方面是滚动数组的技巧。

动态规划的三个要素

1. 状态转移方程
2. 临界条件
3. 枚举状态

> 可以看出，用递归解决也是一样的思路

在上面讲解的爬楼梯问题中，如果我们用 f(n) 表示爬 n 级台阶有多少种方法的话，那么：

```
f(1) 与 f(2) 就是【边界】
f(n) = f(n-1) + f(n-2) 就是【状态转移公式】
```

我用动态规划的形式表示一下：

```
dp[0] 与 dp[1] 就是【边界】
dp[n] = dp[n - 1] + dp[n - 2] 就是【状态转移方程】
```

可以看出两者是多么的相似。

实际上临界条件相对简单，大家只有多刷几道题，里面就有感觉。困难的是找到状态转移方程和枚举状态。这两个核心点的都建立在**已经抽象好了状态**的基础上。比如爬楼梯的问题，如果我们用 f(n) 表示爬 n 级台阶有多少种方法的话，那么 f(1), f(2), ... 就是各个**独立的状态**。

不过状态的定义都有特点的套路。 比如一个字符串的状态，通常是 dp[i] 表示字符串 s 以 i 结尾的 ....。 比如两个字符串的状态，通常是 dp[i][j] 表示字符串 s1 以 i 结尾，s2 以 j 结尾的 ....。

当然状态转移方程可能不止一个， 不同的转移方程对应的效率也可能大相径庭，这个就是比较玄学的话题了，需要大家在做题的过程中领悟。

搞定了状态的定义，那么我们来看下状态转移方程。

状态转移方程

爬楼梯问题由于上第 n 级台阶一定是从 n - 1 或者 n - 2 来的，因此 上第 n 级台阶的数目就是 `上 n - 1 级台阶的数目加上 n - 1 级台阶的数目` 。

上面的这个理解是核心， 它就是我们的状态转移方程，用代码表示就是 `f(n) = f(n - 1) + f(n - 2)` 。

实际操作的过程，有可能题目和爬楼梯一样直观，我们不难想到。也可能隐藏很深或者维度过高。 如果你实在想不到，可以尝试画图打开思路，这也是我刚学习动态规划时候的方法。当你做题量上去了，你的题感就会来，那个时候就可以不用画图了。

状态转移方程实在是没有什么灵丹妙药，不同的题目有不同的解法。状态转移方程同时也是解决动态规划问题中最最困难和关键的点，大家一定要多多练习，提高题感。接下来，我们来看下不那么困难，但是新手疑问比较多的问题 - **如何枚举状态**。

如何枚举状态

前面说了如何枚举状态，才能不重不漏是枚举状态的关键所在。

- 如果是一维状态，那么我们使用一层循环可以搞定。
- 如果是两维状态，那么我们使用两层循环可以搞定。
- 。。。

这样可以保证不重不漏。

但是实际操作的过程有很多细节比如:

- 一维状态我是先枚举左边的还是右边的？（从左到右遍历还是从右到左遍历）
- 二维状态我是先枚举左上边的还是右上的，还是左下的还是右下的？
- 里层循环和外层循环的位置关系（可以互换么）
- 。。。

其实这个东西和很多因素有关，很难总结出一个规律，而且我认为也完全没有必要去总结规律。不过这里我还是总结了一个关键点，那就是：

- **如果你没有使用滚动数组的技巧**，那么遍历顺序取决于状态转移方程。比如:

```
for i in range(1, n + 1):
  dp[i] = dp[i - 1] + 1;
```

那么我们就需要从左到右遍历，原因很简单，因为 dp[i] 依赖于 dp[i - 1]，因此计算 dp[i] 的时候， dp[i - 1] 需要已经计算好了。

> 二维的也是一样的，大家可以试试。

- **如果你使用了滚动数组的技巧**，则怎么遍历都可以，但是不同的遍历意义通常不不同的。比如我将二维的压缩到了一维：

```
for i in range(1, n + 1):
  for j in range(1, n + 1):
    dp[j] = dp[j - 1] + 1;
```

这样是可以的。 dp[j - 1] 实际上指的是压缩前的 dp[i][j - 1]

而：

```
for i in range(1, n + 1):
  #  倒着遍历
  for j in range(n, 0, -1):
    dp[j] = dp[j - 1] + 1;
```

这样也是可以的。 但是 dp[j - 1] 实际上指的是压缩前的 dp[i - 1][j - 1]。因此实际中采用怎么样的遍历手段取决于题目。我特意写了一个 【完全背包问题】套路题（1449. 数位成本和为目标值的最大数字 文章，通过一个具体的例子告诉大家不同的遍历有什么实际不同，强烈建议大家看看，并顺手给个三连。

- 关于里外循环的问题，其实和上面原理类似。

这个比较微妙，大家可以参考这篇文章理解一下 0518.coin-change-2。

小结

关于如何确定临界条件通常是比较简单的，多做几个题就可以快速掌握。

关于如何确定状态转移方程，这个其实比较困难。 不过所幸的是，这些套路性比较强， 比如一个字符串的状态，通常是 dp[i] 表示字符串 s 以 i 结尾的 ....。 比如两个字符串的状态，通常是 dp[i][j] 表示字符串 s1 以 i 结尾，s2 以 j 结尾的 ....。 这样遇到新的题目可以往上套， 实在套不出那就先老实画图，不断观察，提高题感。

关于如何枚举状态，如果没有滚动数组， 那么根据转移方程决定如何枚举即可。 如果用了滚动数组，那么要注意压缩后和压缩前的 dp 对应关系即可。

动态规划为什么要画表格

动态规划问题要画表格，但是有的人不知道为什么要画，就觉得这个是必然的，必要要画表格才是动态规划。

其实动态规划本质上是将大问题转化为小问题，然后大问题的解是和小问题有关联的，换句话说大问题可以由小问题进行计算得到。这一点是和用递归解决一样的， 但是动态规划是一种类似查表的方法来缩短时间复杂度和空间复杂度。

画表格的目的就是去不断推导，完成状态转移， 表格中的每一个 cell 都是一个 `小问题` ， 我们填表的过程其实就是在解决问题的过程，

我们先解决规模为寻常的情况，然后根据这个结果逐步推导，通常情况下，表格的右下角是问题的最大的规模，也就是我们想要求解的规模。

比如我们用动态规划解决背包问题， 其实就是在不断根据之前的小问题 `A[i - 1][j] A[i -1][w - wj]` 来询问：

- 应该选择它
- 还是不选择它

至于判断的标准很简单，就是价值最大，因此我们要做的就是对于选择和不选择两种情况分别求价值，然后取最大，最后更新 cell 即可。

其实大部分的动态规划问题套路都是“选择”或者“不选择”，也就是说是一种“选择题”。 并且大多数动态规划题目还伴随着空间的优化（滚动数组），这是动态规划相对于传统的记忆化递归优势的地方。除了这点优势，就是上文提到的使用动态规划可以减少递归产生的函数调用栈，因此性能上更好。

相关问题

- 0091.decode-ways
- 0139.word-break
- 0198.house-robber

- 0309.best-time-to-buy-and-sell-stock-with-cooldown
- 0322.coin-change
- 0416.partition-equal-subset-sum
- 0518.coin-change-2

总结

本篇文章总结了算法中比较常用的两个方法 - 递归和动态规划。递归的话可以拿树的题目练手，动态规划的话则将我上面推荐的刷完，再考虑去刷力扣的动态规划标签即可。

大家前期学习动态规划的时候，可以先尝试使用记忆化递归解决。然后将其改造为动态规划，这样多练习几次就会有感觉。之后大家可以练习一下滚动数组，这个技巧很有用，并且相对来说比较简单。 比较动态规划的难点在于**枚举所以状态（无重复）** 和 **寻找状态转移方程**。

如果你只能记住一句话，那么请记住： 递归是从问题的结果倒推，直到问题的规模缩小到寻常。 动态规划是从寻常入手， 逐步扩大规模到最优子结构。

另外，大家可以去 LeetCode 探索中的 递归 I 中进行互动式学习。

游程编码和哈夫曼编码

Huffman encode(哈夫曼编码)

Huffman 编码的基本思想就是用短的编码表示出现频率高的字符，用长的编码来表示出现频率低的字符，这使得编码之后的字符串的平均长度、长度的期望值降低，从而实现压缩的目的。 因此 Huffman 编码被广泛地应用于无损压缩领域。

Huffman 编码的过程包含两个主要部分：

- 根据输入字符构建 Huffman 树
- 遍历 Huffman 树，并将树的节点分配给字符

上面提到了他的基本原理就是 `用短的编码表示出现频率高的字符，用长的编码来表示出现频率低的字符` ，因此首先要做的就是统计字符的出现频率，然后根据统计的频率来构建 Huffman 树（又叫最优二叉树）。

Huffman 树就像是一个堆。真正执行编码的时候，类似字典树，节点不用来编码，节点的路径用来编码.

> 节点的值只是用来构建 Huffman 树

eg:

我们统计的结果如下：

character	frequency
a	5
b	9
c	12
d	13
e	16
f	45

- 将每个元素构造成一个节点，即只有一个元素的树。并构建一个最小堆，包含所有的节点，该算法用了最小堆来作为优先队列。
- `选取两个权值最小的节点` ，并添加一个权值为5+9=14的节点，作为他们的父节点。并 `更新最小堆` ，现在最小堆包含5个节点，其中4个树还是原来的节点，权值为5和9的节点合并为一个。

结果是这样的：

character	frequency	encoding
a	5	1100
b	9	1101
c	12	100
d	13	101
e	16	111
f	45	0

run-length encode(游程编码)

游程编码是一种比较简单的压缩算法，其基本思想是将重复且连续出现多次的字符使用（连续出现次数，某个字符）来描述。

比如一个字符串：

```
AAAAABBBBCCC
```

使用游程编码可以将其描述为：

```
5A4B3C
```

5A表示这个地方有5个连续的A，同理4B表示有4个连续的B，3C表示有3个连续的C，其它情况以此类推。

但是实际上情况可能会非常复杂， 如何提取子序列有时候没有看的那么简单，还是上面的例子，我们 有时候可以把 `AAAAABBBBCCC` 整体看成一个子序列， 更复杂的情况还有很多，这里不做扩展。

对文件进行压缩比较适合的情况是文件内的二进制有大量的连续重复，一个经典的例子就是具有大面积色块的BMP图像，BMP因为没有压缩，所以看到的是什么样子存储的时候二进制就是什么样子

> 这也是我们图片倾向于纯色的时候，压缩会有很好的效果
>
> 思考一个问题， 如果我们在CDN上存储两个图片，这两个图片几乎完全一样，我们是否可以进行优化呢？ 这虽然是CDN厂商更应该关心的问题，但是这个问题对我们影响依然很大，值得思考

总结

游程编码和Huffman都是无损压缩算法，即解压缩过程不会损失原数据任何内容。 实际情况，我们先用游程编码一遍，然后再用 Huffman 再次编码一次。几乎所有的无损压缩格式都用到了它们，比如PNG，GIF，PDF，ZIP等。

对于有损压缩，通常是去除了人类无法识别的颜色，听力频率范围等。也就是说损失了原来的数据。 但由于人类无法识别这部分信息，因此很多情况下都是值得的。这种删除了人类无法感知内容的编码，我们称之为“感知编码”（也许是一个自创的新名词），比如JPEG，MP3等。关于有损压缩不是本文的讨论范围，感兴趣的可以搜素相关资料。

实际上，视频压缩的原理也是类似，只不过视频压缩会用到一些额外的算法，比如“时间冗余”，即仅存储变化的部分，对于不变的部分，存储一次就够了。

相关题目

900.rle-iterator

场景

假设你现在要处理这样一个问题，你有一个网站并且拥有 `很多` 访客，每当有用户访问时，你想知道这个ip是不是第一次访问你的网站。

hashtable 可以么

一个显而易见的答案是将所有的 IP 用 hashtable 存起来，每次访问都去 hashtable 中取，然后判断即可。但是题目说了网站有 `很多` 访客， 假如有10亿个用户访问过，假设 IP 是 IPV4， 那么每个 IP 的长度是 4 byte，那么你一共需要4 * 1000000000 = 4000000000Bytes = 4G 。

如果是判断 URL 黑名单，由于每个 URL 会更长（可能远大于上面 IPV4 地址的 4 byte），那么需要的空间可能会远远大于你的期望。

bit

另一个稍微难想到的解法是bit， 我们知道bit有 0 和 1 两种状态，那么用来表示**存在**与**不存在**再合适不过了。

假如有 10 亿个 IP，就可以用 10 亿个 bit 来存储，那么你一共需要 1 * 1000000000 = (4000000000 / 8) Bytes = 128M, 变为原来的1/32, 如果是存储URL这种更长的字符串，效率会更高。 问题是，我们怎么把 IPV4 和 bit 的位置关联上呢?

比如 `192.168.1.1` 应该是用第几位表示， `10.18.1.1` 应该是用第几位表示呢? 答案是使用哈希函数。

基于这种想法，我们只需要两个操作，set(ip) 和 has(ip)，以及一个内置函数 hash(ip) 用于将 IP 映射到 bit 表。

这样做有两个非常致命的缺点：

1. 当样本分布极度不均匀的时候，会造成很大空间上的浪费

> 我们可以通过优化散列函数来解决

1. 当元素不是整型（比如URL）的时候，BitSet就不适用了

> 我们还是可以使用散列函数来解决， 甚至可以多hash几次

布隆过滤器

布隆过滤器其实就是 `bit + 多个散列函数` 。k 次 hash(ip) 会生成多个索引，并将其 k 个索引位置的二进制置为 1。 如果经过 k 个索引位置的值都为 1，那么认为其**可能存在**(因为有冲突的可能)。 如果有一个不为1，那

么**一定不存在**（一个值经过散列函数得到的值一定是唯一的），这也是布隆过滤器的一个重要特点。也就是说布隆过滤器回答了：**可能存在** 和 **一定不存在** 的问题。

从上图可以看出， 布隆过滤器本质上是由**一个很长的二进制向量**和**多个哈希函数**组成。

由于没有 hashtable 的100% 可靠性，因此这本质上是一种**可靠性换取空间的做法**。除了可靠性，布隆过滤器删除起来也比较麻烦。

布隆过滤器的应用

1. 网络爬虫

判断某个URL是否已经被爬取过

1. K-V数据库 判断某个key是否存在

比如 Hbase 的每个 Region 中都包含一个 BloomFilter，用于在查询时快速判断某个 key 在该 region 中是否存在。

1. 钓鱼网站识别

浏览器有时候会警告用户，访问的网站很可能是钓鱼网站，用的就是这种技术

> 从这个算法大家可以对 tradeoff(取舍) 有更入的理解。

字符串问题

字符串问题有很多，从简单的实现substr，识别回文，到复杂一点的公共子串/子序列。其实字符串本质上也是字符数组，因此 很多数据的思想和方法也可以用在字符串问题上，并且在有些时候能够发挥很好的作用。

专门处理字符串的算法也很多，比如trie，马拉车算法，游程编码，huffman树等等。

实现字符串的一些原生方法

这类题目应该是最直接的题目了，题目歧义比较小, 难度也是相对较小，因此用于电面等形式也是不错的。

- 28.implement-str-str
- 344.reverse-string

回文

回文串就是一个正读和反读都一样的字符串，比如“level”或者“noon”等等就是回文串。

判断是否回文的通用方法是首尾双指针，具体可以见下方125号题目。 判断最长回文的思路主要是两个字"扩展"， 如果可以充分利用回文的特点，则可以减少很多无谓的计算，典型的是《马拉车算法》。

相关问题

- 5.longest-palindromic-substring
- 125.valid-palindrome
- 131.palindrome-partitioning
- shortest-palindrome
- 516.longest-palindromic-subsequence

前缀问题

前缀树用来处理这种问题是最符合直觉的，但是它也有缺点，比如公共前缀很少的情况下，比较费内存。

相关题目

-14.longest-common-prefix -208.implement-trie-prefix-tree

其他问题

- 139.word-break

前缀树问题

截止目前（2020-02-04） 前缀树（字典树） 在 LeetCode 一共有 17 道题目。其中 2 道简单，8 个中等，7 个困难。

这里总结了四道题，弄懂这几道， 那么前缀树对你应该不是大问题， 希望这个专题可以帮到正在学习前缀树 的你。

前缀树的 api 主要有以下几个：

- `insert(word)` ：插入一个单词
- `search(word)` ： 查找一个单词是否存在
- `startWith(word)` ： 查找是否存在以 word 为前缀的单词

其中 startWith 是前缀树最核心的用法，其名称前缀树就从这里而来。大家可以先拿 208 题开始，熟悉一下前缀树，然后再尝试别的题目。

一个前缀树大概是这个样子：

如图每一个节点存储一个字符，然后外加一个控制信息表示是否是单词结尾，实际使用过程可能会有细微差别，不过变化不大。

以下是本专题的四道题目的题解，内容会持续更新，感谢你的关注～

- 0208.implement-trie-prefix-tree
- 0211.add-and-search-word-data-structure-design
- 0212.word-search-ii
- 0472.concatenated-words
- 0820.short-encoding-of-words
- 1032.stream-of-characters

相关题目

- 648. 单词替换 (换皮题)

贪婪策略

贪婪策略是一种常见的算法思想，具体是指，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，他所做出的是在某种意义上的局部最优解。贪心算法不是对所有问题都能得到整体最优解，关键是贪心策略的选择，选择的贪心策略必须具备无后效性，即某个状态以前的过程不会影响以后的状态，只与当前状态有关，这点和动态规划一样。贪婪策略和动态规划类似，大多数情况也都是用来处理 `极值问题` 。

LeetCode 上对于贪婪策略有 73 道题目。我们将其分成几个类型来讲解，截止目前我们暂时只提供 `覆盖` 问题，其他的可以期待我的新书或者之后的题解文章。

覆盖

我们挑选三道来讲解，这三道题除了使用贪婪法，你也可以尝试动态规划来解决。

- 45. 跳跃游戏 II，困难
- 1024. 视频拼接，中等
- 1326. 灌溉花园的最少水龙头数目，困难

覆盖问题的一大特征，我们可以将其抽象为 `给定数轴上的一个大区间 I 和 n 个小区间 i[0], i[1], ..., i[n - 1]，问最少选择多少个小区间，使得这些小区间的并集可以覆盖整个大区间。`

我们来看下这三道题吧。

45. 跳跃游戏 II

题目描述

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。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

示例:

输入: [2,3,1,1,4] 输出: 2 解释: 跳到最后一个位置的最小跳跃数是 2。 从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。 说明:

假设你总是可以到达数组的最后一个位置。

思路

贪婪策略，即我们每次在可跳范围内选择可以使得跳的更远的位置，由于题目保证了 `你总是可以到达数组的最后一个位置` ,因此这种算法是完备的。

如下图，开始的位置是 2，可跳的范围是橙色的。然后因为 3 可以跳的更远，所以跳到 3 的位置。

如下图，然后现在的位置就是 3 了，能跳的范围是橙色的，然后因为 4 可以跳的更远，所以下次跳到 4 的位置。

写代码的话，我们用 end 表示当前能跳的边界，对于上边第一个图的橙色 1，第二个图中就是橙色的 4，遍历数组的时候，到了边界，我们就重新更新新的边界。

> 图来自 https://leetcode-cn.com/u/windliang/

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def jump(self, nums: List[int]) -> int:
        n, cnt, furthest, end = len(nums), 0, 0, 0
        for i in range(n - 1):
            furthest = max(furthest, nums[i] + i)
            if i == end:
                cnt += 1
                end = furthest

        return cnt
```

复杂度分析

- 时间复杂度：$O(N)$。
- 空间复杂度：$O(1)$。

1024. 视频拼接

题目描述

你将会获得一系列视频片段，这些片段来自于一项持续时长为 T 秒的体育赛事。这些片段可能有所重叠，也可能长度不一。

视频片段 clips[i] 都用区间进行表示：开始于 clips[i][0] 并于 clips[i][1] 结束。我们甚至可以对这些片段自由地再剪辑，例如片段 [0, 7] 可以剪切成 [0, 1] + [1, 3] + [3, 7] 三部分。

我们需要将这些片段进行再剪辑，并将剪辑后的内容拼接成覆盖整个运动过程的片段（[0, T]）。返回所需片段的最小数目，如果无法完成该任务，则返回 -1 。

示例 1：

输入：clips = [[0,2],[4,6],[8,10],[1,9],[1,5],[5,9]], T = 10 输出：3 解释： 我们选中 [0,2], [8,10], [1,9] 这三个片段。 然后，按下面的方案重制比赛片段： 将 [1,9] 再剪辑为 [1,2] + [2,8] + [8,9] 。 现在我们手上有 [0,2] + [2,8] + [8,10]，而这些涵盖了整场比赛 [0, 10]。 示例 2：

输入：clips = [[0,1],[1,2]], T = 5 输出：-1 解释： 我们无法只用 [0,1] 和 [0,2] 覆盖 [0,5] 的整个过程。 示例 3：

输入：clips = [[0,1],[6,8],[0,2],[5,6],[0,4],[0,3],[6,7],[1,3],[4,7],[1,4],[2,5],[2,6],[3,4],[4,5],[5,7],[6,9]], T = 9 输出：3 解释： 我们选取片段 [0,4], [4,7] 和 [6,9] 。 示例 4：

输入：clips = [[0,4],[2,8]], T = 5 输出：2 解释： 注意，你可能录制超过比赛结束时间的视频。

提示：

1 <= clips.length <= 100 0 <= clips[i][0], clips[i][1] <= 100 0 <= T <= 100

思路

贪婪策略，我们选择满足条件的最大值。和上面的不同，这次我们需要手动进行一次排序，实际上贪婪策略经常伴随着排序，我们按照 clip[0]从小到大进行排序。

如图：

- 1 不可以，因此存在断层
- 2 可以
- 3 不行，因为不到 T

我们当前的 clip 开始结束时间分别为 s，e。 上一段 clip 的结束时间是 t1，上上一段 clip 结束时间是 t2。

那么这种情况下 t1 实际上是不需要的，因为 t2 完全可以覆盖它：

那什么样 t1 才是需要的呢？如图：

用代码来说的话就是 `s > t2 and t2 <= t1`

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def videoStitching(self, clips: List[List[int]], T: int
        #  t1 表示选取的上一个clip的结束时间
        #  t2 表示选取的上上一个clip的结束时间
        t2, t1, cnt = -1, 0, 0
        clips.sort(key=lambda a: a[0])
        for s, e in clips:
            # s > t1 已经确定不可以了，  t1 >= T 已经可以了
            if s > t1 or t1 >= T:
                break
            if s > t2 and t2 <= t1:
                cnt += 1
                t2 = t1
            t1 = max(t1,e)
        return cnt if t1 >= T else - 1
```

复杂度分析

- 时间复杂度：由于使用了排序（假设是基于比较的排序），因此时间复杂度为 $O(NlogN)$。
- 空间复杂度：$O(1)$。

1326. 灌溉花园的最少水龙头数目

题目描述

在 x 轴上有一个一维的花园。花园长度为 n，从点 0 开始，到点 n 结束。

花园里总共有 n + 1 个水龙头，分别位于 [0, 1, ..., n] 。

给你一个整数 n 和一个长度为 n + 1 的整数数组 ranges ，其中 ranges[i]（下标从 0 开始）表示：如果打开点 i 处的水龙头，可以灌溉的区域为 [i - ranges[i], i + ranges[i]] 。

请你返回可以灌溉整个花园的 最少水龙头数目 。如果花园始终存在无法灌溉到的地方，请你返回 -1 。

示例 1：

输入：n = 5, ranges = [3,4,1,1,0,0] 输出：1 解释： 点 0 处的水龙头可以灌溉区间 [-3,3] 点 1 处的水龙头可以灌溉区间 [-3,5] 点 2 处的水龙头可以灌溉区间 [1,3] 点 3 处的水龙头可以灌溉区间 [2,4] 点 4 处的水龙头可以灌溉区间 [4,4] 点 5 处的水龙头可以灌溉区间 [5,5] 只需要打开点 1 处的水龙头即可灌溉整个花园 [0,5] 。 示例 2：

输入：n = 3, ranges = [0,0,0,0] 输出：-1 解释：即使打开所有水龙头，你也无法灌溉整个花园。 示例 3：

输入：n = 7, ranges = [1,2,1,0,2,1,0,1] 输出：3 示例 4：

输入：n = 8, ranges = [4,0,0,0,0,0,0,0,4] 输出：2 示例 5：

输入：n = 8, ranges = [4,0,0,0,4,0,0,0,4] 输出：1

提示：

1 <= n <= 10^4 ranges.length == n + 1 0 <= ranges[i] <= 100

思路

贪心策略，我们尽量找到能够覆盖最远（右边）位置的水龙头，并记录它最右覆盖的土地。

- 我们使用 furthest[i] 来记录经过每一个水龙头 i 能够覆盖的最右侧土地。
- 一共有 n+1 个水龙头，我们遍历 n + 1 次。
- 对于每次我们计算水龙头的左右边界，[i - ranges[i], i + ranges[i]]
- 我们更新左右边界范围内的水龙头的 furthest
- 最后从土地 0 开始，一直到土地 n ，记录水龙头数目

代码

代码支持：Python3

Python3 Code:

```python
class Solution:
    def minTaps(self, n: int, ranges: List[int]) -> int:
        furthest, cnt, cur = [0] * n, 0, 0

        for i in range(n + 1):
            l = max(0, i - ranges[i])
            r = min(n, i + ranges[i])
            for j in range(l, r):
                furthest[j] = max(furthest[j], r)
        while cur < n:
            if furthest[cur] == 0: return -1
            cur = furthest[cur]
            cnt += 1
        return cnt
```

复杂度分析

- 时间复杂度：时间复杂度取决 l 和 r，也就是说取决于 ranges 数组的值，假设 ranges 的平均大小为 Size 的话，那么时间复杂度为 $O(N * Size)$。
- 空间复杂度：我们使用了 furthest 数组， 因此空间复杂度为 $O(N)$。

深度优先遍历

深度优先搜索算法（英语：Depth-First-Search，DFS）是一种用于遍历或搜索树或图的算法。沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点 v 的所在边都已被探寻过，搜索将回溯到发现节点 v 的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。属于盲目搜索。

深度优先搜索是图论中的经典算法，利用深度优先搜索算法可以产生目标图的相应拓扑排序表，利用拓扑排序表可以方便的解决很多相关的图论问题，如最大路径问题等等。

因发明「深度优先搜索算法」，约翰 · 霍普克洛夫特与罗伯特 · 塔扬在 1986 年共同获得计算机领域的最高奖：图灵奖。

截止目前（2020-02-21），深度优先遍历在 LeetCode 中的题目是 129 道。在 LeetCode 中的题型绝对是超级大户了。而对于树的题目，我们基本上都可以使用 DFS 来解决，甚至我们可以基于 DFS 来做广度优先遍历。并不一定说 DFS 不可以做 BFS（广度优先遍历）的事情。而且由于 DFS 通常我们可以基于递归去做，因此算法会更简洁。 在对性能有很高邀请的场合，我建议你使用迭代，否则尽量使用递归，不仅写起来简单快速，还不容易出错。

另外深度优先遍历可以结合回溯专题来联系，建议将这两个专题放到一起来学习。

题目

这是我近期总结的几个 DFS 题目，后续会持续更新～

- 200. 岛屿数量 中等
- 695. 岛屿的最大面积 中等
- 979. 在二叉树中分配硬币 中等

滑动窗口（Sliding Window）

笔者最早接触滑动窗口是 `滑动窗口协议` ，滑动窗口协议（Sliding Window Protocol），属于 TCP 协议的一种应用，用于网络数据传输时的流量控制，以避免拥塞的发生。 发送方和接收方分别有一个窗口大小 w1 和 w2。窗口大小可能会根据网络流量的变化而有所不同，但是在更简单的实现中它们是固定的。窗口大小必须大于零才能进行任何操作。

我们算法中的滑动窗口也是类似，只不过包括的情况更加广泛。实际上上面的滑动窗口在某一个时刻就是固定窗口大小的滑动窗口，随着网络流量等因素改变窗口大小也会随着改变。接下来我们讲下算法中的滑动窗口。

介绍

滑动窗口是一种解决问题的思路和方法，通常用来解决一些连续问题。比如 LeetCode 的 209. 长度最小的子数组。更多滑动窗口题目见下方 `题目列表` 。

常见套路

滑动窗口主要用来处理连续问题。比如题目求解“连续子串 xxxx”，“连续子数组 xxxx”，就应该可以想到滑动窗口。能不能解决另说，但是这种敏感性还是要有的。

从类型上说主要有：

- 固定窗口大小
- 窗口大小不固定，求解最大的满足条件的窗口
- 窗口大小不固定，求解最小的满足条件的窗口（上面的 209 题就属于这种）

后面两种我们统称为 `可变窗口` 。当然不管是哪种类型基本的思路都是一样的，不一样的仅仅是代码细节。

固定窗口大小

对于固定窗口，我们只需要固定初始化左右指针 l 和 r，分别表示的窗口的左右顶点，并且保证：

1. l 初始化为 0
2. 初始化 r，使得 r - l + 1 等于窗口大小
3. 同时移动 l 和 r
4. 判断窗口内的连续元素是否满足题目限定的条件

- 4.1 如果满足，再判断是否需要更新最优解，如果需要则更新最优解
- 4.2 如果不满足，则继续。

可变窗口大小

对于可变窗口，我们同样固定初始化左右指针 l 和 r，分别表示的窗口的左右顶点。后面有所不同，我们需要保证：

1. l 和 r 都初始化为 0
2. r 指针移动一步
3. 判断窗口内的连续元素是否满足题目限定的条件
    - 3.1 如果满足，再判断是否需要更新最优解，如果需要则更新最优解。并尝试通过移动 l 指针缩小窗口大小。循环执行 3.1
    - 3.2 如果不满足，则继续。

形象地来看的话，就是 r 指针不停向右移动，l 指针仅仅在窗口满足条件之后才会移动，起到窗口收缩的效果。

模板代码

伪代码

```
初始化慢指针 = 0
初始化 ans

for 快指针 in 可迭代集合
    更新窗口内信息
    while 窗口内不符合题意
        扩展或者收缩窗口
        慢指针移动
返回 ans
```

代码

以下是 209 题目的代码，使用 Python 编写，大家意会即可。

```
class Solution:
    def minSubArrayLen(self, s: int, nums: List[int]) -> in
        l = total = 0
        ans = len(nums) + 1
        for r in range(len(nums)):
            total += nums[r]
            while total >= s:
                ans = min(ans, r - l + 1)
                total -= nums[l]
                l += 1
        return  0 if ans == len(nums) + 1 else ans
```

题目列表（有题解）

以下题目有的信息比较直接，有的题目信息比较隐蔽，需要自己发掘

- 【Python，JavaScript】滑动窗口（3. 无重复字符的最长子串）
- 76. 最小覆盖子串
- 209. 长度最小的子数组
- 【Python】滑动窗口（438. 找到字符串中所有字母异位词）
- 【904. 水果成篮】（Python3）
- 【930. 和相同的二元子数组】（Java，Python）
- 【992. K 个不同整数的子数组】滑动窗口（Python）
- 【1004. 最大连续 1 的个数 III】滑动窗口（Python3）
- 【1234. 替换子串得到平衡字符串】[Java/C++/Python] Sliding Window
- 【1248. 统计「优美子数组」】滑动窗口（Python）

扩展阅读

- LeetCode Sliding Window Series Discussion

位运算

我这里总结了几道位运算的题目分享给大家，分别是 136和137， 260 和 645， 总共加起来四道题。 四道题全部都是位运算的套路，如果你想练习位运算的话，不要错过哦～～

前菜

开始之前我们先了解下异或，后面会用到。

1. 异或的性质

两个数字异或的结果 `a^b` 是将 a 和 b 的二进制每一位进行运算，得出的数字。 运算的逻辑是果同一位的数字相同则为 0，不同则为 1

1. 异或的规律
2. 任何数和本身异或则为 `0`
3. 任何数和 0 异或是 `本身`
4. 异或运算满足交换律，即：

`a ^ b ^ c = a ^ c ^ b`

OK，我们来看下这三道题吧。

136. 只出现一次的数字1

题目大意是除了一个数字出现一次，其他都出现了两次，让我们找到出现一次的数。我们执行一次全员异或即可。

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        single_number = 0
        for num in nums:
            single_number ^= num
        return single_number
```

复杂度分析

- 时间复杂度：$O(N)$，其中N为数组长度。
- 空间复杂度：$O(1)$

137. 只出现一次的数字2

题目大意是除了一个数字出现一次，其他都出现了三次，让我们找到出现一次的数。 灵活运用位运算是本题的关键。

Python3:

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        res = 0
        for i in range(32):
            cnt = 0  # 记录当前 bit 有多少个1
            bit = 1 << i  # 记录当前要操作的 bit
            for num in nums:
                if num & bit != 0:
                    cnt += 1
            if cnt % 3 != 0:
                # 不等于0说明唯一出现的数字在这个 bit 上是1
                res |= bit

        return res - 2 ** 32 if res > 2 ** 31 - 1 else res
```

- 为什么Python最后需要对返回值进行判断?

如果不这么做的话测试用例是[-2,-2,1,1,-3,1,-3,-3,-4,-2] 的时候，就会输出4294967292。 其原因在于Python是动态类型语言，在这种情况下其会将符号位置的1看成了值，而不是当作符号“负数”。 这是不对的。 正确答案应该是 - 4，-4的二进制码是 1111...100，就变成 2^32-4=4294967292，解决办法就是 减去 2 ** 32 。

> 之所以这样不会有问题的原因还在于题目限定的数组范围不会超过 2 ** 32

JavaScript:

```
var singleNumber = function(nums) {
  let res = 0;
  for (let i = 0; i < 32; i++) {
    let cnt = 0;
    let bit = 1 << i;
    for (let j = 0; j < nums.length; j++) {
      if (nums[j] & bit) cnt++;
    }
    if (cnt % 3 != 0) res = res | bit;
  }
  return res;
};
```

复杂度分析

- 时间复杂度：$O(N)$，其中N为数组长度。
- 空间复杂度：$O(1)$

645. 错误的集合

和上面的 `137. 只出现一次的数字2` 思路一样。这题没有限制空间复杂度，因此直接hashmap 存储一下没问题。 不多说了，我们来看一种空间复杂度$O(1)$的解法。

由于和 `137. 只出现一次的数字2` 思路基本一样，我直接复用了代码。具体思路是，将nums的所有索引提取出一个数组idx，那么由idx和nums组成的数组构成singleNumbers的输入，其输出是唯二不同的两个数。

但是我们不知道哪个是缺失的，哪个是重复的，因此我们需要重新进行一次遍历，判断出哪个是缺失的，哪个是重复的。

```
class Solution:
    def singleNumbers(self, nums: List[int]) -> List[int]:
        ret = 0  # 所有数字异或的结果
        a = 0
        b = 0
        for n in nums:
            ret ^= n
        # 找到第一位不是0的
        h = 1
        while(ret & h == 0):
            h <<= 1
        for n in nums:
            # 根据该位是否为0将其分为两组
            if (h & n == 0):
                a ^= n
            else:
                b ^= n

        return [a, b]

    def findErrorNums(self, nums: List[int]) -> List[int]:
        nums = [0] + nums
        idx = []
        for i in range(len(nums)):
            idx.append(i)
        a, b = self.singleNumbers(nums + idx)
        for num in nums:
            if a == num:
                return [a, b]
        return [b, a]
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

260. 只出现一次的数字3

题目大意是除了两个数字出现一次，其他都出现了两次，让我们找到这个两个数。

我们进行一次全员异或操作，得到的结果就是那两个只出现一次的不同的数字的异或结果。

我们刚才讲了异或的规律中有一个 `任何数和本身异或则为0` ， 因此我们的思路是能不能将这两个不同的数字分成两组 A 和 B。 分组需要满足两个条件.

1. 两个独特的的数字分成不同组
2. 相同的数字分成相同组

这样每一组的数据进行异或即可得到那两个数字。

问题的关键点是我们怎么进行分组呢?

由于异或的性质是，同一位相同则为 0，不同则为 1. 我们将所有数字异或的结果一定不是 0，也就是说至少有一位是 1.

我们随便取一个， 分组的依据就来了， 就是你取的那一位是 0 分成 1 组，那一位是 1 的分成一组。 这样肯定能保证 `2. 相同的数字分成相同组` ，不同的数字会被分成不同组么。 很明显当然可以， 因此我们选择是 1，也就是 说 `两个独特的的数字` 在那一位一定是不同的，因此两个独特元素一定会被分成不同组。

```python
class Solution:
    def singleNumbers(self, nums: List[int]) -> List[int]:
        ret = 0  # 所有数字异或的结果
        a = 0
        b = 0
        for n in nums:
            ret ^= n
        # 找到第一位不是0的
        h = 1
        while(ret & h == 0):
            h <<= 1
        for n in nums:
            # 根据该位是否为0将其分为两组
            if (h & n == 0):
                a ^= n
            else:
                b ^= n

        return [a, b]
```

复杂度分析

- 时间复杂度：$O(N)$，其中N为数组长度。
- 空间复杂度：$O(1)$

相关题目

- number-of-1-bits
- counting-bits
- reverse-bits

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经30K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

设计题

系统设计是一个没有标准答案的open-end问题，所以关键在于对于特定问题的设计选择,俗称trade-off。这也是较能考察面试者知识水平的一种题型。

截止目前（2020-03-28），设计题在LeetCode一共58道题目。

其中：

- 简单14道
- 中等32道
- 困难12道

这里精选6道题目进行详细讲解，旨在大家能够对系统设计题的答题技巧和套路有所掌握，喜欢的话别忘了点赞和关注哦。

题目列表

这是我近期总结的几道设计题目，后续会持续更新～

- 0155.min-stack 简单
- 0211.add-and-search-word-data-structure-design 中等
- 0232.implement-queue-using-stacks 简单
- 0460.lfu-cache 困难
- 895.maximum-frequency-stack 困难
- 900.rle-iterator 中等

小岛

LeetCode 上有很多小岛题，虽然官方没有这个标签， 但是在我这里都差不多。不管是思路还是套路都比较类似，大家可以结合起来练习。

- 200. 岛屿数量
- 695. 岛屿的最大面积
- 1162. 地图分析

上面三道题都可以使用常规的 DFS 来做。 并且递归的方向都是上下左右四个方向。更有意思的是，都可以采用原地修改的方式，来减少开辟 visited 的空间。

最大公约数

关于最大公约数有专门的研究。 而在 LeetCode 中虽然没有直接让你求解最大公约数的题目。但是却有一些间接需要你求解最大公约数的题目。

比如：

- 914. 卡牌分组
- 365. 水壶问题
- 1071. 字符串的最大公因子

因此如何求解最大公约数就显得重要了。

定义法

```
def GCD(a: int, b: int) -> int:
    smaller = min(a, b)
    while smaller:
        if a % smaller == 0 and b % smaller == 0:
            return smaller
        smaller -= 1
```

复杂度分析

- 时间复杂度：最好的情况是执行一次循环体，最坏的情况是循环到 smaller 为 1，因此总的时间复杂度为 $O(N)$，其中 N 为 a 和 b 中较小的数。
- 空间复杂度：$O(1)$。

辗转相除法

如果我们需要计算 a 和 b 的最大公约数，运用辗转相除法的话。首先，我们先计算出 a 除以 b 的余数 c，把问题转化成求出 b 和 c 的最大公约数；然后计算出 b 除以 c 的余数 d，把问题转化成求出 c 和 d 的最大公约数；再然后计算出 c 除以 d 的余数 e，把问题转化成求出 d 和 e 的最大公约数。..... 以此类推，逐渐把两个较大整数之间的运算转化为两个较小整数之间的运算，直到两个数可以整除为止。

```
def GCD(a: int, b: int) -> int:
    return a if b == 0 else GCD(b, a % b)
```

复杂度分析

- 时间复杂度：$O(log(max(a,b)))$
- 空间复杂度：空间复杂度取决于递归的深度，因此空间复杂度为 $O(log(max(a,b)))$

下面我们对上面的过程进行一个表形象地讲解，实际上这也是教材里面的讲解方式，我只是照搬过来，增加一下自己的理解罢了。我们来通过一个例子来讲解：

假如我们有一块 1680 米 * 640 米 的土地，我们希望讲起分成若干正方形的土地，且我们想让正方形土地的边长尽可能大，我们应该如何设计算法呢?

实际上这正是一个最大公约数的应用场景，我们的目标就是求解 1680 和 640 的最大公约数。

将 1680 米 * 640 米 的土地分割，相当于对将 400 米 * 640 米 的土地进行分割。 为什么呢？ 假如 400 米 * 640 米分割的正方形边长为 x，那么有 640 % x == 0，那么肯定也满足剩下的两块 640 米 * 640 米的。

我们不断进行上面的分割：

直到边长为 80，没有必要进行下去了。

辗转相除法如果 a 和 b 都很大的时候，a % b 性能会较低。在中国，《九章算术》中提到了一种类似辗转相减法的 更相减损术。它的原理是：`两个正整数 a 和 b（a>b），它们的最大公约数等于 a-b 的差值 c 和较小数 b 的最大公约数。`。

```
def GCD(a: int, b: int) -> int:
    if a == b:
        return a
    if a < b:
        return GCD(b - a, a)
    return GCD(a - b, b)
```

上面的代码会报栈溢出。原因在于如果 a 和 b 相差比较大的话，递归次数会明显增加，要比辗转相除法递归深度增加很多，最坏时间复杂度为 O(max(a, b))。这个时候我们可以将 `辗转相除法` 和 `更相减损术` 做一个结合，从而在各种情况都可以获得较好的性能。

并查集

关于并查集的题目不少，官方给的数据是 30 道（截止 2020-02-20），但是有一些题目虽然官方没有贴 `并查集` 标签，但是使用并查集来说确非常简单。这类题目如果掌握模板，那么刷这种题会非常快，并且犯错的概率会大大降低，这就是模板的好处。

我这里总结了几道并查集的题目：

- 547. 朋友圈
- 721. 账户合并
- 990. 等式方程的可满足性
- 1202. 交换字符串中的元素

看完这里的内容，建议拿上面的题目练下手，检测一下学习成果。

概述

并查集是一种树型的数据结构，用于处理一些不交集（Disjoint Sets）的合并及查询问题。有一个联合-查找算法（Union-find Algorithm）定义了两个用于此数据结构的操作：

- Find：确定元素属于哪一个子集。它可以被用来确定两个元素是否属于同一子集。
- Union：将两个子集合并成同一个集合。

由于支持这两种操作，一个不相交集也常被称为联合-查找数据结构（Union-find Data Structure）或合并-查找集合（Merge-find Set）。为了更加精确的定义这些方法，需要定义如何表示集合。一种常用的策略是为每个集合选定一个固定的元素，称为代表，以表示整个集合。接着，Find(x) 返回 x 所属集合的代表，而 Union 使用两个集合的代表作为参数。

形象解释

比如有两个司令。 司令下有若干军长，军长下有若干师长。。。

我们如何判断某两个师长是否属于同一个司令呢（连通性）？

很简单，我们顺着师长，往上找，找到司令。 如果两个师长找到的是同一个司令，那么就属于同一个司令。我们用 parent[x] = y 表示 x 的 parent 是 y，通过不断沿着搜索 parent 搜索找到 root，然后比较 root 是否相同即可得出结论。

以上过程涉及了两个基本操作 `find` 和 `connnected` 。 并查集除了这两个基本操作，还有一个是 `union` 。即将两个集合合并为同一个。

如图有两个司令：

我们将其合并为一个联通域，最简单的方式就是直接将其中一个司令指向另外一个即可：

以上就是三个核心 API `find` ， `connnected` 和 `union` ， 的形象化解释，下面我们来看下代码实现。

核心 API

find

```
def find(self, x):
    while x != self.parent[x]:
        x = self.parent[x]
    return x
```

connected

```
def connected(self, p, q):
    return self.find(p) == self.find(q)
```

union

```
def union(self, p, q):
    if self.connected(p, q): return
    self.parent[self.find(p)] = self.find(q)
```

完整代码模板

```
class UF:
    parent = {}
    cnt = 0
    def __init__(self, M):
        # 初始化 parent 和 cnt

    def find(self, x):
        while x != self.parent[x]:
            x = self.parent[x]
        return x
    def union(self, p, q):
        if self.connected(p, q): return
        self.parent[self.find(p)] = self.find(q)
        self.cnt -= 1
    def connected(self, p, q):
        return self.find(p) == self.find(q)
```

带路径压缩的代码模板

```
class UF:
    parent = {}
    size = {}
    cnt = 0
    def __init__(self, M):
        # 初始化 parent, size 和 cnt

    def find(self, x):
        while x != self.parent[x]:
            x = self.parent[x]
            # 路径压缩
            self.parent[x] = self.parent[self.parent[x]];
        return x
    def union(self, p, q):
        if self.connected(p, q): return
        # 小的树挂到大的树上， 使树尽量平衡
        leader_p = self.find(p)
        leader_q = self.find(q)
        if self.size[leader_p] < self.size[leader_q]:
            self.parent[leader_p] = leader_q
            self.size[leader_p] += self.size[leader_q]
        else:
            self.parent[leader_q] = leader_p
            self.size[leader_q] += self.size[leader_p]
        self.cnt -= 1
    def connected(self, p, q):
        return self.find(p) == self.find(q)
```

上面是递归的方式进行路径压缩，写起来比较简单。但是有栈溢出的风险。 接下来我们看下迭代的写法：

```python
class UF:
    parent = {}
    def __init__(self, equations):
        # 做一些初始化操作

    def find(self, x):
        # 根节点
        r = x
        while r != parent[r]:
            r = parent[r]
        k = x
        while k != r:
            # 暂存parent[k]的父节点
            j = parent[k]
            parent[k] = r
            k = j
        return r
    def union(self, p, q):
        if self.connected(p, q): return
        self.parent[self.find(p)] = self.find(q)
    def connected(self, p, q):
        return self.find(p) == self.find(q)
```

我花了几天时间，从力扣中精选了五道相同思想的题目，来帮助大家解套，如果觉得文章对你有用，记得点赞分享，让我看到你的认可，有动力继续做下去。

- 467. 环绕字符串中唯一的子字符串(中等)
- 795. 区间子数组个数(中等)
- 904. 水果成篮(中等)
- 992. K 个不同整数的子数组（困难）
- 1109. 航班预订统计(中等)

前四道题都是滑动窗口的子类型，我们知道滑动窗口适合在题目要求连续的情况下使用， 而前缀和也是如此。二者在连续问题中，对于**优化时间复杂度**有着很重要的意义。 因此如果一道题你可以用暴力解决出来，而且题目恰好有连续的限制， 那么滑动窗口和前缀和等技巧就应该被想到。

除了这几道题， 还有很多题目都是类似的套路， 大家可以在学习过程中进行体会。今天我们就来一起学习一下。

前菜

我们从一个简单的问题入手，识别一下这种题的基本形式和套路，为之后的四道题打基础。当你了解了这个套路之后， 之后做这种题就可以直接套。

需要注意的是这四道题的前置知识都是 `滑动窗口` ， 不熟悉的同学可以先看下我之前写的 滑动窗口专题（思路 + 模板）

母题 0

有 N 个的正整数放到数组 A 里，现在要求一个新的数组 B，新数组的第 i 个数 B[i]是原数组 A 第 0 到第 i 个数的和。

这道题可以使用前缀和来解决。 前缀和是一种重要的预处理，能大大降低查询的时间复杂度。我们可以简单理解为“数列的前 n 项的和”。这个概念其实很容易理解，即一个数组中，第 n 位存储的是数组前 n 个数字的和。

对 [1,2,3,4,5,6] 来说，其前缀和可以是 pre=[1,3,6,10,15,21]。我们可以使用公式 pre[i]=pre[$i-1$]+nums[i]得到每一位前缀和的值，从而通过前缀和进行相应的计算和解题。其实前缀和的概念很简单，但困难的是如何在题目中使用前缀和以及如何使用前缀和的关系来进行解题。

题目推荐： 1480. 一维数组的动态和

母题 1

如果让你求一个数组的连续子数组总个数，你会如何求？其中连续指的是数组的索引连续。 比如 [1,3,4]，其连续子数组有： `[1]`, `[3]`, `[4]`, `[1,3]`, `[3,4]` , `[1,3,4]` ，你需要返回 6。

一种思路是总的连续子数组个数等于：**以索引为 0 结尾的子数组个数 + 以索引为 1 结尾的子数组个数 + ... + 以索引为 n - 1 结尾的子数组个数**，这无疑是完备的。

同时**利用母题 0 的前缀和思路， 边遍历边求和。**

参考代码(JS)：

```
function countSubArray(nums) {
  let ans = 0;
  let pre = 0;
  for (_ in nums) {
    pre += 1;
    ans += pre;
  }
  return ans;
}
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$

而由于以索引为 i 结尾的子数组个数就是 i + 1，因此这道题可以直接用等差数列求和公式 `(1 + n) * n / 2` ，其中 n 数组长度。

母题 2

我继续修改下题目， 如果让你求一个数组相邻差为 1 连续子数组的总个数呢？其实就是**索引差 1 的同时，值也差 1。**

和上面思路类似，无非就是增加差值的判断。

参考代码(JS)：

```
function countSubArray(nums) {
  let ans = 1;
  let pre = 1;
  for (let i = 1; i < nums.length; i++) {
    if (nums[i] - nums[i - 1] == 1) {
      pre += 1;
    } else {
      pre = 0;
    }

    ans += pre;
  }
  return ans;
}
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$

如果我值差只要大于 1 就行呢？其实改下符号就行了，这不就是求上升子序列个数么？这里不再继续赘述， 大家可以自己试试。

母题 3

我们继续扩展。

如果我让你求出不大于 k 的子数组的个数呢？不大于 k 指的是子数组的全部元素都不大于 k。 比如 [1,3,4] 子数组有 `[1], [3], [4], [1,3], [3,4] , [1,3,4]` ，不大于 3 的子数组有 `[1], [3], [1,3]` ，那么 [1,3,4] 不大于 3 的子数组个数就是 3。 实现函数 atMostK(k, nums)。

参考代码（JS）：

```
function countSubArray(k, nums) {
  let ans = 0;
  let pre = 0;
  for (let i = 0; i < nums.length; i++) {
    if (nums[i] <= k) {
      pre += 1;
    } else {
      pre = 0;
    }

    ans += pre;
  }
  return ans;
}
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$

母题 4

如果我让你求出子数组最大值刚好是 k 的子数组的个数呢？ 比如 [1,3,4] 子数组有 `[1], [3], [4], [1,3], [3,4] , [1,3,4]` ，子数组最大值刚好是 3 的子数组有 `[3], [1,3]` ，那么 [1,3,4] 子数组最大值刚好是 3 的子数组个数就是 2。实现函数 exactK(k, nums)。

实际上是 exactK 可以直接利用 atMostK，即 atMostK(k) - atMostK(k - 1)，原因见下方母题 5 部分。

母题 5

如果我让你求出子数组最大值刚好是 介于 k1 和 k2 的子数组的个数呢？ 实现函数 betweenK(k1, k2, nums)。

实际上是 betweenK 可以直接利用 atMostK，即 atMostK(k1, nums) - atMostK(k2 - 1, nums)，其中 k1 > k2。前提是值是离散的， 比如上面我出的题都是整数。 因此我可以直接 减 1，因为 **1 是两个整数最小的间隔**。

如上， `小于等于 10 的区域` 减去 `小于 5 的区域` 就是 `大于等于 5 且小于等于 10 的区域` 。

注意我说的是小于 5， 不是小于等于 5。 由于整数是离散的，最小间隔是 1。因此小于 5 在这里就等价于 小于等于 4。这就是 betweenK(k1, k2, nums) = atMostK(k1) - atMostK(k2 - 1) 的原因。

因此不难看出 exactK 其实就是 betweenK 的特殊形式。 当 k1 == k2 的时候， betweenK 等价于 exactK。

因此 atMostK 就是灵魂方法，一定要掌握，不明白建议多看几遍。

有了上面的铺垫， 我们来看下第一道题。

467. 环绕字符串中唯一的子字符串（中等）

题目描述

```
把字符串 s 看作是“abcdefghijklmnopqrstuvwxyz”的无限环绕字符串，所

现在我们有了另一个字符串 p 。你需要的是找出 s 中有多少个唯一的 p 的非

注意: p 仅由小写的英文字母组成，p 的大小可能超过 10000。



示例 1:

输入: "a"
输出: 1
解释: 字符串 S 中只有一个"a"子字符。


示例 2:

输入: "cac"
输出: 2
解释: 字符串 S 中的字符串“cac”只有两个子串“a”、“c”。.


示例 3:

输入: "zab"
输出: 6
解释: 在字符串 S 中有六个子串“z”、“a”、“b”、“za”、“ab”、“zab”。.
```

前置知识

- 滑动窗口

思路

题目是让我们找 p 在 s 中出现的非空子串数目，而 s 是固定的一个无限循环字符串。由于 p 的数据范围是 10^5 ，因此暴力找出所有子串就需要 10^10 次操作了，应该会超时。而且题目很多信息都没用到，肯定不对。

仔细看下题目发现，这不就是母题 2 的变种么？话不多说， 直接上代码，看看有多像。

> 为了减少判断， 我这里用了一个黑科技， p 前面加了个 `^` 。

```
class Solution:
    def findSubstringInWraproundString(self, p: str) -> int
        p = '^' + p
        w = 1
        ans = 0
        for i in range(1,len(p)):
            if ord(p[i])-ord(p[i-1]) == 1 or ord(p[i])-ord(
                w += 1
            else:
                w = 1
            ans += w
        return ans
```

如上代码是有问题。 比如 `cac` 会被计算为 3，实际上应该是 2。根本原因在于 c 被错误地计算了两次。因此一个简单的思路就是用 set 记录一下访问过的子字符串即可。比如：

```
{
    c,
    abc,
    ab,
    abcd
}
```

而由于 set 中的元素一定是连续的，因此上面的数据也可以用 hashmap 存：

```
{
    c: 3
    d: 4
    b: 1
}
```

含义是：

- 以 b 结尾的子串最大长度为 1，也就是 b。
- 以 c 结尾的子串最大长度为 3，也就是 abc。
- 以 d 结尾的子串最大长度为 4，也就是 abcd。

至于 c ，是没有必要存的。我们可以通过母题 2 的方式算出来。

具体算法：

- 定义一个 len_mapper。key 是 字母， value 是 长度。 含义是以 key 结尾的最长连续子串的长度。

> 关键字是：最长

- 用一个变量 w 记录连续子串的长度，遍历过程根据 w 的值更新 len_mapper
- 返回 len_mapper 中所有 value 的和。

比如: abc，此时的 len_mapper 为:

```
{
    c: 3
    b: 2
    a: 1
}
```

再比如：abcab，此时的 len_mapper 依旧。

再比如: abcazabc，此时的 len_mapper：

```
{
    c: 4
    b: 3
    a: 2
    z: 1
}
```

这就得到了去重的目的。这种算法是不重不漏的，因为最长的连续子串一定是包含了比它短的连续子串，这个思想和 1297. 子串的最大出现次数 剪枝的方法有异曲同工之妙。

代码（Python）

```
class Solution:
    def findSubstringInWraproundString(self, p: str) -> int
        p = '^' + p
        len_mapper = collections.defaultdict(lambda: 0)
        w = 1
        for i in range(1,len(p)):
            if ord(p[i])-ord(p[i-1]) == 1 or ord(p[i])-ord(
                w += 1
            else:
                w = 1
            len_mapper[p[i]] = max(len_mapper[p[i]], w)
        return sum(len_mapper.values())
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为字符串 p 的长度。

- 空间复杂度：由于最多存储 26 个字母， 因此空间实际上是常数，故空间复杂度为 $O(1)$。

795. 区间子数组个数（中等）

题目描述

```
给定一个元素都是正整数的数组 A ，正整数 L  以及  R (L <= R)。

求连续、非空且其中最大元素满足大于等于 L  小于等于 R 的子数组个数。

例如 :
输入:
A = [2, 1, 4, 3]
L = 2
R = 3
输出: 3
解释: 满足条件的子数组: [2], [2, 1], [3].
注意:

L, R  和  A[i] 都是整数，范围在  [0, 10^9]。
数组  A  的长度范围在[1, 50000]。
```

前置知识

- 滑动窗口

思路

由母题 5，我们知道 **betweenK 可以直接利用 atMostK，即 atMostK(k1) - atMostK(k2 - 1)，其中 k1 > k2**。

由母题 2，我们知道如何求满足一定条件（这里是元素都小于等于 R）子数组的个数。

这两个结合一下， 就可以解决。

代码（Python）

> 代码是不是很像

```
class Solution:
    def numSubarrayBoundedMax(self, A: List[int], L: int, R
        def notGreater(R):
            ans = cnt = 0
            for a in A:
                if a <= R: cnt += 1
                else: cnt = 0
                ans += cnt
            return  ans

        return notGreater(R) - notGreater(L - 1)
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$。

904. 水果成篮（中等）

题目描述

```
在一排树中，第 i 棵树产生 tree[i] 型的水果。
你可以从你选择的任何树开始，然后重复执行以下步骤：

把这棵树上的水果放进你的篮子里。如果你做不到，就停下来。
移动到当前树右侧的下一棵树。如果右边没有树，就停下来。
请注意，在选择一颗树后，你没有任何选择：你必须执行步骤 1，然后执行步骤

你有两个篮子，每个篮子可以携带任何数量的水果，但你希望每个篮子只携带一

用这个程序你能收集的水果树的最大总量是多少？



示例 1：

输入：[1,2,1]
输出：3
解释：我们可以收集 [1,2,1]。
示例 2：

输入：[0,1,2,2]
输出：3
解释：我们可以收集 [1,2,2]
如果我们从第一棵树开始，我们将只能收集到 [0, 1]。
示例 3：

输入：[1,2,3,2,2]
输出：4
解释：我们可以收集 [2,3,2,2]
如果我们从第一棵树开始，我们将只能收集到 [1, 2]。
示例 4：

输入：[3,3,3,1,2,1,1,2,3,3,4]
输出：5
解释：我们可以收集 [1,2,1,1,2]
如果我们从第一棵树或第八棵树开始，我们将只能收集到 4 棵水果树。



提示：

1 <= tree.length <= 40000
0 <= tree[i] < tree.length
```

前置知识

- 滑动窗口

思路

题目花里胡哨的。我们来抽象一下，就是给你一个数组， 让你**选定一个子数组， 这个子数组最多只有两种数字**，这个选定的子数组最大可以是多少。

这不就和母题 3 一样么？只不过 k 变成了固定值 2。另外由于题目要求整个窗口最多两种数字，我们用哈希表存一下不就好了吗？

> set 是不行了的。 因此我们不但需要知道几个数字在窗口， 我们还要知道每个数字出现的次数，这样才可以使用滑动窗口优化时间复杂度。

代码（Python）

```
class Solution:
    def totalFruit(self, tree: List[int]) -> int:
        def atMostK(k, nums):
            i = ans = 0
            win = defaultdict(lambda: 0)
            for j in range(len(nums)):
                if win[nums[j]] == 0: k -= 1
                win[nums[j]] += 1
                while k < 0:
                    win[nums[i]] -= 1
                    if win[nums[i]] == 0: k += 1
                    i += 1
                ans = max(ans, j - i + 1)
            return ans

        return atMostK(2, tree)
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(k)$。

992. K 个不同整数的子数组（困难）

题目描述

```
给定一个正整数数组 A，如果 A 的某个子数组中不同整数的个数恰好为 K，则

（例如，[1,2,3,1,2] 中有 3 个不同的整数：1，2，以及 3。）

返回 A 中好子数组的数目。



示例 1：

输入：A = [1,2,1,2,3], K = 2
输出：7
解释：恰好由 2 个不同整数组成的子数组：[1,2], [2,1], [1,2], [2,3]
示例 2：

输入：A = [1,2,1,3,4], K = 3
输出：3
解释：恰好由 3 个不同整数组成的子数组：[1,2,1,3], [2,1,3], [1,3,4


提示：

1 <= A.length <= 20000
1 <= A[i] <= A.length
1 <= K <= A.length
```

前置知识

- 滑动窗口

思路

由母题 5，知：exactK = atMostK(k) - atMostK(k - 1)， 因此答案便呼之欲出了。其他部分和上面的题目 `904. 水果成篮` 一样。

> 实际上和所有的滑动窗口题目都差不多。

代码（Python）

```python
class Solution:
    def subarraysWithKDistinct(self, A, K):
        return self.atMostK(A, K) - self.atMostK(A, K - 1)

    def atMostK(self, A, K):
        counter = collections.Counter()
        res = i = 0
        for j in range(len(A)):
            if counter[A[j]] == 0:
                K -= 1
            counter[A[j]] += 1
            while K < 0:
                counter[A[i]] -= 1
                if counter[A[i]] == 0:
                    K += 1
                i += 1
            res += j - i + 1
        return res
```

复杂度分析

- 时间复杂度：$O(N)$，中 N 为数组长度。
- 空间复杂度：$O(k)$。

1109. 航班预订统计（中等）

题目描述

```
这里有 n 个航班，它们分别从 1 到 n 进行编号。

我们这儿有一份航班预订表，表中第 i 条预订记录 bookings[i] = [i,

请你返回一个长度为 n 的数组 answer，按航班编号顺序返回每个航班上预订



示例：

输入：bookings = [[1,2,10],[2,3,20],[2,5,25]], n = 5
输出：[10,55,45,25,25]



提示：

1 <= bookings.length <= 20000
1 <= bookings[i][0] <= bookings[i][1] <= n <= 20000
1 <= bookings[i][2] <= 10000
```

前置知识

- 前缀和

思路

这道题的题目描述不是很清楚。我简单分析一下题目：

[i, j, k] 其实代表的是 第 i 站上来了 k 个人， 一直到 第 j 站都在飞机上，到第 j + 1 就不在飞机上了。所以第 i 站到第 j 站的**每一站**都会因此多 k 个人。

理解了题目只会不难写出下面的代码。

```python
class Solution:
    def corpFlightBookings(self, bookings: List[List[int]],
        counter = [0] * n

        for i, j, k in bookings:
            while i <= j:
                counter[i - 1] += k
                i += 1
        return counter
```

如上的代码复杂度太高，无法通过全部的测试用例。

注意到里层的 while 循环是连续的数组全部加上一个数字，不难想到可以利用母题 0 的前缀和思路优化。

一种思路就是在 i 的位置 + k， 然后利用前缀和的技巧给 i 到 n 的元素都加上 k。但是题目需要加的是一个区间， j + 1 及其之后的元素会被多加一个 k。一个简单的技巧就是给 j + 1 的元素减去 k，这样正负就可以抵消。

代码（Python）

```
class Solution:
    def corpFlightBookings(self, bookings: List[List[int]],
        counter = [0] * (n + 1)

        for i, j, k in bookings:
            counter[i - 1] += k
            if j < n: counter[j] -= k
        for i in range(n + 1):
            counter[i] += counter[i - 1]
        return counter[:-1]
```

复杂度分析

- 时间复杂度：$O(N)$，中 N 为数组长度。
- 空间复杂度：$O(N)$。

总结

这几道题都是滑动窗口和前缀和的思路。力扣类似的题目还真不少，大家只有多留心，就会发现这个套路。

前缀和的技巧以及滑动窗口的技巧都比较固定，且有模板可套。 难点就在于我怎么才能想到可以用这个技巧呢?

我这里总结了两点：

1. 找关键字。比如题目中有连续，就应该条件反射想到滑动窗口和前缀和。比如题目求最大最小就想到动态规划和贪心等等。想到之后，就可以和题目信息对比快速排除错误的算法，找到可行解。这个思考的时间会随着你的题感增加而降低。
2. 先写出暴力解，然后找暴力解的瓶颈， 根据瓶颈就很容易知道应该用什么数据结构和算法去优化。

最后推荐几道类似的题目， 供大家练习，一定要自己写出来才行哦。

- 303. 区域和检索 - 数组不可变
- 1186.删除一次得到子数组最大和
- 1310. 子数组异或查询
- 1371. 每个元音包含偶数次的最长子字符串

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。

更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 36K star 啦。

大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

衡二叉树专题

力扣关于平衡二叉树的题目还是有一些的，并且都非常经典，推荐大家练习。今天给大家精选了 4 道题，如果你彻底搞明白了这几道题，碰到其他的平衡二叉树的题目应该不至于没有思路。当你领会了我的思路之后，建议再找几个题目练手，巩固一下学习成果。

110. 平衡二叉树（简单）

最简单的莫过于判断一个树是否为平衡二叉树了，我们来看下。

题目描述

```
给定一个二叉树，判断它是否是高度平衡的二叉树。

本题中，一棵高度平衡二叉树定义为：

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 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
  / \
 4   4
返回 false
```

思路

由于平衡二叉树定义为就是**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**。用伪代码描述就是：

```
if abs(高度(root.left) - 高度(root.right)) <= 1 and root.lef
    print('是平衡二叉树')
else:
    print('不是平衡二叉树')
```

而 root.left 和 root.right **如何判断是否是二叉平衡树就和 root 是一样的了**，可以看出这个问题有明显的递归性。

因此我们首先需要知道如何计算一个子树的高度。这个可以通过递归的方式轻松地计算出来。计算子树高度的 Python 代码如下：

```
def dfs(node, depth):
    if not node: return 0
    l = dfs(node.left, depth + 1)
    r = dfs(node.right, depth + 1)
    return max(l, r) + 1
```

代码

代码支持： Python3

Python3 Code:

```
class Solution:
    def isBalanced(self, root: TreeNode) -> bool:
        def dfs(node, depth):
            if not node: return 0
            l = dfs(node.left, depth + 1)
            r = dfs(node.right, depth + 1)
            return max(l, r)  + 1
        if not root: return True
        if abs(dfs(root.left, 0) -  dfs(root.right, 0)) > 1
        return self.isBalanced(root.left) and self.isBalanc
```

复杂度分析

- 时间复杂度：对于 isBalanced 来说，由于每个节点最多被访问一次，这部分的时间复杂度为 $O(N)$，而 dfs 函数 每次被调用的次数不超过 $logN$，因此总的时间复杂度为 $O(NlogN)$，其中 N 为 树的节点总数。

- 空间复杂度：由于使用了递归，这里的空间复杂度的瓶颈在栈空间，因此空间复杂度为 $O(h)$，其中 h 为树的高度。

108. 将有序数组转换为二叉搜索树（简单）

108 和 109 基本是一样的，只不过数据结构不一样，109 变成了链表了而已。由于链表操作比数组需要考虑更多的因素，因此 109 是 中等难度。

题目描述

```
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

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

      0
     / \
   -3   9
   /   /
 -10  5
```

思路

对于这个问题或者 `给定一个二叉搜索树，将其改为平衡（后面会讲）` 基本思路都是一样的。

题目的要求是将有序数组转化为：

1. 高度平衡的二叉树
2. 二叉搜索树

由于平衡二叉树是左右两个子树的高度差的绝对值不超过 1。因此一种简单的方法是**选择中点作为根节点，根节点左侧的作为左子树，右侧的作为右子树即可。**原因很简单，这样分配可以保证左右子树的节点数目差不超过 1。因此高度差自然也不会超过 1 了。

上面的操作同时也满足了二叉搜索树，原因就是题目给的数组是有序的。

> 你也可以选择别的数作为根节点，而不是中点，这也可以看出答案是不唯一的。

代码

代码支持： Python3

Python3 Code:

```
class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> TreeNode
        if not nums: return None
        mid = (len(nums) - 1) // 2
        root = TreeNode(nums[mid])
        root.left = self.sortedArrayToBST(nums[:mid])
        root.right = self.sortedArrayToBST(nums[mid + 1:])
        return root
```

复杂度分析

- 时间复杂度：由于每个节点最多被访问一次，因此总的时间复杂度为 $O(N)$，其中 N 为数组长度。
- 空间复杂度：由于使用了递归，这里的空间复杂度的瓶颈在栈空间，因此空间复杂度为 $O(h)$，其中 h 为树的高度。同时由于是平衡二叉树，因此 h 就是 $logN$。

109. 有序链表转换二叉搜索树（中等）

题目描述

```
`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

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

      0
     / \
   -3   9
   /   /
 -10  5
```

思路

和 108 思路一样。 不同的是数据结构的不同，因此我们需要关注的是链表和数组的操作差异。

（数组的情况）

我们再来看下链表：

（链表的情况）

找到中点，只需要使用经典的快慢指针即可。同时为了防止环的出现，我们需要斩断指向 mid 的 next 指针，因此需要记录一下中点前的一个节点，这只需要用一个变量 pre 记录即可。

代码

代码支持： Python3

Python3 Code:

```
class Solution:
    def sortedListToBST(self, head: ListNode) -> TreeNode:
        if not head:
            return head
        pre, slow, fast = None, head, head

        while fast and fast.next:
            fast = fast.next.next
            pre = slow
            slow = slow.next
        if pre:
            pre.next = None
        node = TreeNode(slow.val)
        if slow == fast:
            return node
        node.left = self.sortedListToBST(head)
        node.right = self.sortedListToBST(slow.next)
        return node
```

复杂度分析

- 时间复杂度：由于每个节点最多被访问一次，因此总的时间复杂度为 $O(N)$，其中 N 为链表长度。
- 空间复杂度：由于使用了递归，这里的空间复杂度的瓶颈在栈空间，因此空间复杂度为 $O(h)$，其中 h 为树的高度。同时由于是平衡二叉树，因此 h 就是 $logN$。

1382. 将二叉搜索树变平衡（中等）

题目描述

```
给你一棵二叉搜索树，请你返回一棵 平衡后 的二叉搜索树，新生成的树应该与

如果一棵二叉搜索树中，每个节点的两棵子树高度差不超过 1 ，我们就称这棵二

如果有多种构造方法，请你返回任意一种。



示例：
```

```
输入：root = [1,null,2,null,3,null,4,null,null]
输出：[2,1,3,null,null,null,4]
解释：这不是唯一的正确答案，[3,1,4,null,2,null,null] 也是一个可行

提示：

树节点的数目在 1 到 10^4 之间。
树节点的值互不相同，且在 1 到 10^5 之间。
```

思路

由于 `二叉搜索树的中序遍历是一个有序数组` ，因此问题很容易就转化为 `108. 将有序数组转换为二叉搜索树（简单）` 。

代码

代码支持： Python3

Python3 Code:

```
class Solution:
    def inorder(self, node):
        if not node: return []
        return self.inorder(node.left) + [node.val] + self.
    def balanceBST(self, root: TreeNode) -> TreeNode:
        nums = self.inorder(root)
        def dfs(start, end):
            if start == end: return TreeNode(nums[start])
            if start > end: return None
            mid = (start + end) // 2
            root = TreeNode(nums[mid])
            root.left = dfs(start, mid - 1)
            root.right = dfs(mid + 1, end)
            return root
        return dfs(0, len(nums) - 1)
```

复杂度分析

- 时间复杂度：由于每个节点最多被访问一次，因此总的时间复杂度为 $O(N)$，其中 N 为链表长度。
- 空间复杂度：虽然使用了递归，但是瓶颈不在栈空间，而是开辟的长度为 N 的 nums 数组，因此空间复杂度为 $O(N)$，其中 N 为树的节点总数。

总结

本文通过四道关于二叉平衡树的题帮助大家识别此类型题目背后的思维逻辑，我们来总结一下学到的知识。

平衡二叉树指的是：`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。`

如果需要让你判断一个树是否是平衡二叉树，只需要死扣定义，然后用递归即可轻松解决。

如果需要你将一个数组或者链表（逻辑上都是线性的数据结构）转化为平衡二叉树，只需要随便选一个节点，并分配一半到左子树，另一半到右子树即可。

同时，如果要求你转化为平衡二叉搜索树，则可以选择排序数组(或链表)的中点，左边的元素为左子树， 右边的元素为右子树即可。

> 小提示 1： 如果不需要是二叉搜索树则不需要排序，否则需要排序。
>
> 小提示 2： 你也可以不选择中点， 算法需要相应调整，感兴趣的同学可以试试。
>
> 小提示 3： 链表的操作需要特别注意环的存在。

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

91 天学算法

- 第一期讲义-二分法
- 第一期讲义-双指针
- 第二期

二分查找

二分查找又称 `折半搜索算法` 。 狭义地来讲，二分查找是一种在有序数组查找某一特定元素的搜索算法。这同时也是大多数人所知道的一种说法。实际上， 广义的二分查找是将问题的规模缩小到原有的一半。类似的，三分法就是将问题规模缩小为原来的 1/3。

本文给大家带来的内容则是 `狭义地二分查找` ，如果想了解其他广义上的二分查找可以查看我之前写的一篇博文 从老鼠试毒问题来看二分法

> 尽管二分查找的基本思想相对简单，但细节可以令人难以招架 ... —高德纳

当乔恩·本特利将二分搜索问题布置给专业编程课的学生时，百分之 90 的学生在花费数小时后还是无法给出正确的解答，主要因为这些错误程序在面对边界值的时候无法运行，或返回错误结果。1988 年开展的一项研究显示，20 本教科书里只有 5 本正确实现了二分搜索。不仅如此，本特利自己 1986 年出版的《编程珠玑》一书中的二分搜索算法存在整数溢出的问题，二十多年来无人发现。Java 语言的库所实现的二分搜索算法中同样的溢出问题存在了九年多才被修复。

可见二分查找并不简单， 本文就试图带你走近 ta，明白 ta 的底层逻辑，并提供模板帮助大家写书 bug free 的二分查找代码。

大家可以看完讲义结合 LeetCode Book 二分查找练习一下

问题定义

给定一个由数字组成的有序数组 nums，并给你一个数字 target。问 nums 中是否存在 target。如果存在， 则返回其在 nums 中的索引。如果不存在，则返回 - 1。

这是二分查找中最简单的一种形式。当然二分查找也有很多的变形，这也是二分查找容易出错，难以掌握的原因。

常见变体有：

- 如果存在多个满足条件的元素，返回最左边满足条件的索引。
- 如果存在多个满足条件的元素，返回最右边满足条件的索引。
- 数组不是整体有序的。 比如先升序再降序，或者先降序再升序。
- 将一维数组变成二维数组。
- 。。。

接下来，我们逐个进行查看。

前提

- 数组是有序的（如果无序，我们也可以考虑排序，不过要注意排序的复杂度）

术语

二分查找中使用的术语：

- target —— 要查找的值
- index —— 当前位置
- l 和 r —— 左右指针
- mid —— 左右指针的中点，用来确定我们应该向左查找还是向右查找的索引

常见题型

查找一个数

算法描述：

- 先从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜索过程结束；
- 如果目标元素大于中间元素，则在数组大于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。
- 如果目标元素小于中间元素，则在数组小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。
- 如果在某一步骤数组为空，则代表找不到。

复杂度分析

- 平均时间复杂度： $O(logN)$
- 最坏时间复杂度： $O(logN)$
- 最优时间复杂度： $O(1)$
- 空间复杂度
  - 迭代: $O(1)$
  - 递归： $O(logN)$（无尾调用消除）

> 后面的复杂度也是类似的，不再赘述。

这种搜索算法每一次比较都使搜索范围缩小一半，是典型的二分查找。

这个是二分查找中最简答的一种类型了，我们先来搞定它。 我们来一个具体的例子， 这样方便大家增加代入感。假设 nums 为 `[1,3,4,6,7,8,10,13,14]` ， target 为 4·。

- 刚开始数组中间的元素为 7
- 7 > 4 ，由于 7 右边的数字都大于 7 ，因此不可能是答案。我们将范围缩写到了 7 的左侧。

- 此时中间元素为 3
- 3 < 4，由于 3 左边的数字都小于 3 ，因此不可能是答案。我们将范围缩写到了 3 的右侧。
- 此时中间元素为 4，正好是我们要找的，返回其索引 2 即可。

如何将上面的算法转换为容易理解的可执行代码呢？就算是这样一个简简单单，朴实无华的二分查找， 不同的人写出来的差别也是很大的。 如果没有一个思维框架指导你，那么你在不同的时间可能会写出差异很大的代码。这样的话，你犯错的几率会大大增加。

这里给大家介绍一个我经常使用的思维框架和代码模板。

思维框架

首先定义搜索区间为 [left, right]，注意是左右都闭合，之后会用到这个点

> 你可以定义别的搜索区间形式，不过后面的代码也相应要调整，感兴趣的可以试试别的搜索区间。

- 由于定义的搜索区间为 [left, right]，因此当 left <= right 的时候，搜索区间都不为空，此时我们都需要继续搜索。 也就是说终止搜索条件应该为 left <= right。

> 举个例子容易明白一点。 比如对于区间 [4,4]，其包含了一个元素 4，因此搜索区间不为空，需要继续搜索（试想 4 恰好是我们要找的 target，如果不继续搜索， 会错过正确答案）。而当搜索区间为 [left, right) 的时候，同样对于 [4,4]，这个时候搜索区间却是空的，因为这样的一个区间不存在任何数字·。

- 循环体内，我们不断计算 mid ，并将 nums[mid] 与 目标值比对。
  - 如果 nums[mid] 等于目标值， 则提前返回 mid（只需要找到一个满足条件的即可）
  - 如果 nums[mid] 小于目标值， 说明目标值在 mid 右侧，这个时候搜索区间可缩小为 [mid + 1, right] （mid 以及 mid 左侧的数字被我们排除在外）
  - 如果 nums[mid] 大于目标值， 说明目标值在 mid 左侧，这个时候搜索区间可缩小为 [left, mid - 1] （mid 以及 mid 右侧的数字被我们排除在外）
- 循环结束都没有找到，则说明找不到，返回 -1 表示未找到。

代码模板

Java

```java
public int binarySearch(int[] nums, int target) {
    // 左右都闭合的区间 [l, r]
    int left = 0;
    int right = nums.length - 1;

    while(left <= right) {
        int mid = left + (right - left) / 2;
        if(nums[mid] == target)
            return mid;
        if (nums[mid] < target)
                  // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        if (nums[mid] > target)
            // 搜索区间变为 [left, mid - 1]
            right = mid - 1;
    }
    return -1;
}
```

Python

```python
def binarySearch(nums, target):
    # 左右都闭合的区间 [l, r]
    l, r = 0, len(nums) - 1
    while l <= r:
        mid = (left + right) >> 1
        if nums[mid] == target: return mid
        # 搜索区间变为 [mid+1, right]
        if nums[mid] < target: l = mid + 1
        # 搜索区间变为 [left, mid - 1]
        if nums[mid] > target: r = mid - 1
    return -1
```

JavaScript

```
function binarySearch(nums, target) {
  let left = 0;
  let right = nums.length - 1;
  while (left <= right) {
    const mid = Math.floor(left + (right - left) / 2);
    if (nums[mid] == target) return mid;
    if (nums[mid] < target)
      // 搜索区间变为 [mid+1, right]
      left = mid + 1;
    if (nums[mid] > target)
      // 搜索区间变为 [left, mid - 1]
      right = mid - 1;
  }
  return -1;
}
```

C++

暂时空缺，欢迎 PR

```

```

寻找最左边的满足条件的值

和 `查找一个数` 类似， 我们仍然套用 `查找一个数` 的思维框架和代码模板。

思维框架

- 首先定义搜索区间为 [left, right]，注意是左右都闭合，之后会用到这个点。
- 终止搜索条件为 left <= right。
- 循环体内，我们不断计算 mid ，并将 nums[mid] 与 目标值比对。
  - 如果 nums[mid] 等于目标值， 则收缩右边界，我们找到了一个备胎，继续看看左边还有没有了（**注意这里不一样**）
  - 如果 nums[mid] 小于目标值， 说明目标值在 mid 右侧，这个时候搜索区间可缩小为 [mid + 1, right]
  - 如果 nums[mid] 大于目标值， 说明目标值在 mid 左侧，这个时候搜索区间可缩小为 [left, mid - 1]
- 由于不会提前返回，因此我们需要检查最终的 left，看 nums[left]是否等于 target。
  - 如果不等于 target，或者 left 出了右边边界了，说明至死都没有找到一个备胎，则返回 -1.
  - 否则返回 left 即可，备胎转正。

代码模板

> 实际上 nums[mid] > target 和 nums[mid] == target 是可以合并的。
> 我这里为了清晰，就没有合并，大家熟悉之后合并起来即可。

Java

```
public int binarySearchLeft(int[] nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0;
    int right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        }
        if (nums[mid] > target) {
            // 搜索区间变为 [left, mid-1]
            right = mid - 1;
        }
        if (nums[mid] == target) {
            // 收缩右边界
            right = mid - 1;
        }
    }
    // 检查是否越界
    if (left >= nums.length || nums[left] != target)
        return -1;
    return left;
}
```

Python

```python
def binarySearchLeft(nums, target):
    # 左右都闭合的区间 [l, r]
    l, r = 0, len(nums) - 1
    while l <= r:
        mid = (l + r) >> 1
        if nums[mid] == target:
            # 收缩右边界
            r = mid - 1;
        # 搜索区间变为 [mid+1, right]
        if nums[mid] < target: l = mid + 1
        # 搜索区间变为 [left, mid - 1]
        if nums[mid] > target: r = mid - 1
    if l >= len(nums) or nums[l] != target: return -1
    return l
```

JavaScript

```javascript
function binarySearchLeft(nums, target) {
  let left = 0;
  let right = nums.length - 1;
  while (left <= right) {
    const mid = Math.floor(left + (right - left) / 2);
    if (nums[mid] == target)
      // 收缩右边界
      right = mid - 1;
    if (nums[mid] < target)
      // 搜索区间变为 [mid+1, right]
      left = mid + 1;
    if (nums[mid] > target)
      // 搜索区间变为 [left, mid - 1]
      right = mid - 1;
  }
  // 检查是否越界
  if (left >= nums.length || nums[left] != target) return -
  return left;
}
```

C++

暂时空缺，欢迎 PR

```
```

寻找最右边的满足条件的值

和 `查找一个数` 类似， 我们仍然套用 `查找一个数` 的思维框架和代码模板。

> 有没有感受到框架和模板的力量?

思维框架

- 首先定义搜索区间为 [left, right]，注意是左右都闭合，之后会用到这个点。

> 你可以定义别的搜索区间形式，不过后面的代码也相应要调整，感兴趣的可以试试别的搜索区间。

- 由于我们定义的搜索区间为 [left, right]，因此当 left <= right 的时候，搜索区间都不为空。 也就是说我们的终止搜索条件为 left <= right。

> 举个例子容易明白一点。 比如对于区间 [4,4]，其包含了一个元素 4，因此搜索区间不为空。而当搜索区间为 [left, right) 的时候，同样对于 [4,4]，这个时候搜索区间却是空的。

- 循环体内，我们不断计算 mid ，并将 nums[mid] 与 目标值比对。
  - 如果 nums[mid] 等于目标值， 则收缩左边界，我们找到了一个备胎，继续看看右边还有没有了
  - 如果 nums[mid] 小于目标值， 说明目标值在 mid 右侧，这个时候搜索区间可缩小为 [mid + 1, right]
  - 如果 nums[mid] 大于目标值， 说明目标值在 mid 左侧，这个时候搜索区间可缩小为 [left, mid - 1]
- 由于不会提前返回，因此我们需要检查最终的 right，看 nums[right] 是否等于 target。
  - 如果不等于 target，或者 right 出了左边边界了，说明至死都没有找到一个备胎，则返回 -1.
  - 否则返回 right 即可，备胎转正。

代码模板

> 实际上 nums[mid] < target 和 nums[mid] == target 是可以合并的。我这里为了清晰，就没有合并，大家熟悉之后合并起来即可。

Java

```java
public int binarySearchRight(int[] nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0
    int right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        }
        if (nums[mid] > target) {
            // 搜索区间变为 [left, mid-1]
            right = mid - 1;
        }
        if (nums[mid] == target) {
            // 收缩左边界
            left = mid + 1;
        }
    }
    // 检查是否越界
    if (right < 0 || nums[right] != target)
        return -1;
    return right;
}
```

Python

```python
def binarySearchRight(nums, target):
    # 左右都闭合的区间 [l, r]
    l, r = 0, len(nums) - 1
    while l <= r:
        mid = (l + r) >> 1
        if nums[mid] == target:
            # 收缩左边界
            l = mid + 1;
        # 搜索区间变为 [mid+1, right]
        if nums[mid] < target: l = mid + 1
        # 搜索区间变为 [left, mid - 1]
        if nums[mid] > target: r = mid - 1
    if r < 0 or nums[r] != target: return -1
    return r
```

JavaScript

```
function binarySearchRight(nums, target) {
  let left = 0;
  let right = nums.length - 1;
  while (left <= right) {
    const mid = Math.floor(left + (right - left) / 2);
    if (nums[mid] == target)
      // 收缩左边界
      left = mid + 1;
    if (nums[mid] < target)
      // 搜索区间变为 [mid+1, right]
      left = mid + 1;
    if (nums[mid] > target)
      // 搜索区间变为 [left, mid - 1]
      right = mid - 1;
  }
  // 检查是否越界
  if (right < 0 || nums[right] != target) return -1;
  return right;
}
```

C++

暂时空缺，欢迎 PR

```

```

局部有序（先降后升或先升后降）

LeetCode 有原题 33. 搜索旋转排序数组 和 81. 搜索旋转排序数组 II， 我们直接拿过来讲解好了。

其中 81 题是在 33 题的基础上增加了 `包含重复元素` 的可能，实际上 33 题的进阶就是 81 题。通过这道题，大家可以感受到"包含重复与否对我们算法的影响"。 我们直接上最复杂的 81 题，这个会了，可以直接 AC 第 33 题。

81. 搜索旋转排序数组 II

题目描述

```
假设按照升序排序的数组在预先未知的某个点上进行了旋转。

( 例如，数组 [0,0,1,2,2,5,6] 可能变为 [2,5,6,0,0,1,2] )。

编写一个函数来判断给定的目标值是否存在于数组中。若存在返回 true，否则

示例 1:

输入: nums = [2,5,6,0,0,1,2], target = 0
输出: true
示例 2:

输入: nums = [2,5,6,0,0,1,2], target = 3
输出: false
进阶:

这是 搜索旋转排序数组 的延伸题目，本题中的 nums  可能包含重复元素。
这会影响到程序的时间复杂度吗？会有怎样的影响，为什么？
```

思路

这是一个我在网上看到的前端头条技术终面的一个算法题。我们先不考虑重复元素。

题目要求时间复杂度为 logn，因此基本就是二分法了。 这道题目不是直接的有序数组，不然就是 easy 了。

首先要知道，我们随便选择一个点，将数组分为前后两部分，其中一部分一定是有序的。

具体步骤：

- 我们可以先找出 mid，然后根据 mid 来判断，mid 是在有序的部分还是无序的部分

假如 mid 小于 start，则 mid 一定在右边有序部分。 假如 mid 大于 start，则 mid 一定在左边有序部分。

> 注意我没有考虑等号，之后我会讲。

- 然后我们继续判断 target 在哪一部分， 我们就可以舍弃另一部分了

我们只需要比较 target 和有序部分的边界关系就行了。 比如 mid 在右侧有序部分，即[mid, end] 那么我们只需要判断 target >= mid && target <= end 就能知道 target 在右侧有序部分，我们就 可以舍弃左边部分了(start = mid + 1)， 反之亦然。

我们以([6,7,8,1,2,3,4,5], 4)为例讲解一下：

接下来，我们考虑重复元素的问题。就会发生 nums[mid] == nums[start] 了，比如 30333 。这个时候，可以选择 l 右移一位。有的同学会担心”会不会错失目标元素？“。其实这个担心是多余的，前面我们已经介绍了”搜索区间“。由于搜索区间同时包含 l 和 mid ，因此去除一个 l ，我们还有 mid。假如 3 是我们要找的元素， 这样进行下去绝对不会错过，而是收缩”搜索区间“到一个元素 3 ，我们就可以心安理得地返回 3 了。

代码（Python）

```
class Solution:
    def search(self, nums, target):
        l, r = 0, len(nums)-1
        while l <= r:
            mid = l + (r-l)//2
            if nums[mid] == target:
                return True
            while l < mid and nums[l] == nums[mid]:  # tri
                l += 1
            # the first half is ordered
            if nums[l] <= nums[mid]:
                # target is in the first half
                if nums[l] <= target < nums[mid]:
                    r = mid - 1
                else:
                    l = mid + 1
            # the second half is ordered
            else:
                # target is in the second half
                if nums[mid] < target <= nums[r]:
                    l = mid + 1
                else:
                    r = mid - 1
        return False
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(1)$

二维数组

二维数组的二分查找和一维没有本质区别， 我们通过两个题来进行说明。

74. 搜索二维矩阵

题目地址

https://leetcode-cn.com/problems/search-a-2d-matrix/

题目描述

```
编写一个高效的算法来判断 m x n 矩阵中，是否存在一个目标值。该矩阵具有

每行中的整数从左到右按升序排列。
每行的第一个整数大于前一行的最后一个整数。
示例 1:

输入:
matrix = [
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
target = 3
输出: true
示例 2:

输入:
matrix = [
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
target = 13
输出: false
```

思路

简单来说就是将一个一维有序数组切成若干长度相同的段，然后将这些段拼接成一个二维数组。你的任务就是在这个拼接成的二维数组中找到target。

需要注意的是，数组是不存在重复元素的。

> 如果有重复元素，我们该怎么办?

算法：

- 选择矩阵左下角作为起始元素 Q
- 如果 Q > target，右方和下方的元素没有必要看了（相对于一维数组的右边元素）
- 如果 Q < target，左方和上方的元素没有必要看了（相对于一维数组的左边元素）
- 如果 Q == target ，直接 返回 True
- 交回了都找不到，返回 False

代码(Python)

```
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target:
        m = len(matrix)
        if m == 0:
            return False
        n = len(matrix[0])

        x = m - 1
        y = 0
        while x >= 0 and y < n:
            if matrix[x][y] > target:
                x -= 1
            elif matrix[x][y] < target:
                y += 1
            else:
                return True
        return False
```

复杂度分析

- 时间复杂度：最坏的情况是只有一行或者只有一列，此时时间复杂度为 $O(M*N)$。更多的情况下时间复杂度为 $O(M+N)$
- 空间复杂度：$O(1)$

力扣 240. 搜索二维矩阵 II 发生了一点变化，不再是 `每行的第一个整数大于前一行的最后一个整数` ，而是 `每列的元素从上到下升序排列` 。我们仍然可以选择左下进行二分。

寻找最值(改进的二分)

上面全部都是找到给定值，这次我们试图寻找最值（最小或者最大）。我们以最小为例，讲解一下这种题如何切入。

153. 寻找旋转排序数组中的最小值

题目地址

https://leetcode-cn.com/problems/find-minimum-in-rotated-sorted-array/

题目描述

```
假设按照升序排序的数组在预先未知的某个点上进行了旋转。

( 例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2] )。

请找出其中最小的元素。

你可以假设数组中不存在重复元素。

示例 1:

输入: [3,4,5,1,2]
输出: 1
示例 2:

输入: [4,5,6,7,0,1,2]
输出: 0
```

二分法

思路

和查找指定值得思路一样。我们还是：

- 初始化首尾指针 l 和 r
- 如果 nums[mid] 大于 nums[r]，说明 mid 在左侧有序部分，由于最小的一定在右侧，因此可以收缩左区间，即 l = mid + 1
- 否则收缩右侧，即 r = mid（不可以 r = mid - 1）

> 这里多判断等号没有意义，因为题目没有让我们找指定值

- 当 l >= r 或者 nums[l] < nums[r] 的时候退出循环

> nums[l] < nums[r]，说明区间 [l, r] 已经是整体有序了，因此 nums[l] 就是我们想要找的

代码（Python）

```python
class Solution:
    def findMin(self, nums: List[int]) -> int:
        l, r = 0, len(nums) - 1

        while l < r:
            # important
            if nums[l] < nums[r]:
                return nums[l]
            mid = (l + r) // 2
            # left part
            if nums[mid] > nums[r]:
                l = mid + 1
            else:
                # right part
                r = mid
        # l or r is not important
        return nums[l]
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(1)$

另一种二分法

思路

我们当然也也可以和 nums[l] 比较，而不是上面的 nums[r]，我们发现：

- 旋转点左侧元素**都大于**数组第一个元素
- 旋转点右侧元素**都小于**数组第一个元素

这样就建立了 nums[mid] 和 nums[0] 的联系。

具体算法：

1. 找到数组的中间元素 mid。
2. 如果中间元素 > 数组第一个元素，我们需要在 mid 右边搜索。

- 如果中间元素 <= 数组第一个元素，我们需要在 mid 左边搜索。

上面的例子中，中间元素 6 比第一个元素 4 大，因此在中间点右侧继续搜索。

1. 当我们找到旋转点时停止搜索，当以下条件满足任意一个即可：
2. nums[mid] > nums[mid + 1]，因此 mid+1 是最小值。
3. nums[mid - 1] > nums[mid]，因此 mid 是最小值。

代码（Python）

```
class Solution:
    def findMin(self, nums):
        # If the list has just one element then return that
        if len(nums) == 1:
            return nums[0]

        # left pointer
        left = 0
        # right pointer
        right = len(nums) - 1

        # if the last element is greater than the first ele
        # e.g. 1 < 2 < 3 < 4 < 5 < 7. Already sorted array.
        # Hence the smallest element is first element. A[0]
        if nums[right] > nums[0]:
            return nums[0]

        # Binary search way
        while right >= left:
            # Find the mid element
            mid = left + (right - left) / 2
            # if the mid element is greater than its next e
            # This point would be the point of change. From
            if nums[mid] > nums[mid + 1]:
                return nums[mid + 1]
            # if the mid element is lesser than its previou
            if nums[mid - 1] > nums[mid]:
                return nums[mid]

            # if the mid elements value is greater than the
            # the least value is still somewhere to the rig
            if nums[mid] > nums[0]:
                left = mid + 1
            # if nums[0] is greater than the mid value then
            else:
                right = mid - 1
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(1)$

二叉树

对于一个给定的二叉树，其任意节点最多只有两个子节点。 从这个定义，我们似乎可以嗅出一点二分法的味道， 但是这并不是二分。但是，二叉树中却和二分有很多联系，我们来看一下。

最简单的，如果这个二叉树是一个二叉搜索树（BST）。 那么实际上，在一个二叉搜索树种进行搜索的过程就是二分法。

如上图，我们需要在这样一个二叉搜索树中搜索 7。那么我们的搜索路径则会是 8 -> 3 -> 6 -> 7，这也是一种二分法。只不过相比于普通的**有序序列查找给定值**二分， 其时间复杂度的下界更差，原因在于二叉搜索树并不一定是二叉平衡树。

上面讲了二叉搜索树，我们再来看一种同样特殊的树 - 完全二叉树。 如果我们给一颗完全二叉树的所有节点进行编号（二进制），依次为 01,10,11, ...。

那么实际上，最后一行的编号就是从根节点到该节点的路径。 其中 0 表示向左， 1 表示向右。(第一位数字不用)。 我们以最后一行的 101 为例，我们需要执行一次左，然后一次右。

其实原理也不难，如果你用数组表示过完全二叉树，那么就很容易理解。我们可以发现， 父节点的编号都是左节点的二倍，并且都是右节点的二倍 + 1。从二进制的角度来看就是：**父节点的编号左移一位就是左节点的编号，左移一位 + 1 就是右节点的编号**。 因此反过来， 知道了子节点的最后一位，我们就能知道它是父节点的左节点还是右节点啦。

题目推荐

- 875. 爱吃香蕉的珂珂
- 300. 最长上升子序列
- 354. 俄罗斯套娃信封问题
- 面试题 17.08. 马戏团人塔

> 后面三个题建议一起做

总结

二分查找是一种非常重要且难以掌握的核心算法，大家一定要好好领会。有的题目直接二分就可以了，有的题目二分只是其中一个环节。不管是哪种，都需要我们对二分的思想和代码模板非常熟悉才可以。

二分查找的基本题型有：

- 查找满足条件的元素，返回对应索引
- 如果存在多个满足条件的元素，返回最左边满足条件的索引。
- 如果存在多个满足条件的元素，返回最右边满足条件的索引。
- 数组不是整体有序的。 比如先升序再降序，或者先降序再升序。
- 将一维数组变成二维数组。
- 局部有序查找最大（最小）元素
- 。。。

不管是哪一种类型，我们的思维框架都是类似的，都是：

- 先定义**搜索区间**（非常重要）

- 根据搜索区间定义循环结束条件
- 取中间元素和目标元素做对比（目标元素可能是需要找的元素或者是数组第一个，最后一个元素等）（非常重要）
- 根据比较的结果收缩区间，舍弃非法解（也就是二分）

> 如果是整体有序通常只需要 nums[mid] 和 target 比较即可。如果是局部有序，则可能需要与其周围的特定元素进行比较。

大家可以使用这个思维框架并结合本文介绍的几种题型进行练习，必要的情况可以使用我提供的解题模板，提供解题速度的同时，有效地降低出错的概率。

特别需要注意的是**有无重复元素对二分算法影响很大**，我们需要小心对待。

【91算法-基础篇】05.双指针

力扣加加，一个努力做西湖区最好的算法题解的团队。就在今天它给大家带来了《91 天学算法》，帮助大家摆脱困境，征服算法。

什么是双指针

顾名思议，双指针就是**两个指针**，但是不同于 C，C++中的指针， 其是一种**算法思想**。

如果说，我们迭代一个数组，并输出数组每一项，我们需要一个指针来记录当前遍历的项，这个过程我们叫单指针（index）的话。

```
for(int i = 0;i < nums.size(); i++) {
  输出(nums[i]);
}
```

（图 1）

那么双指针实际上就是有两个这样的指针，最为经典的就是二分法中的左右双指针啦。

```
int l = 0;
int r = nums.size() - 1;

while (l < r) {
    if(一定条件) return 合适的值，一般是 l 和 r 的中点
    if(一定条件) l++
    if(一定条件) r--
}
// 因为 l == r，因此返回 l 和 r 都是一样的
return l
```

（图 2）

读到这里，你发现双指针是一个很宽泛的概念，就好像数组，链表一样，其类型会有很多很多， 比如二分法经常用到 `左右端点双指针` 。滑动窗口会用到 `快慢指针和固定间距指针` 。 因此双指针其实是一种综合性很强的类型，类似于数组，栈等。 但是我们这里所讲述的双指针，往往指的是某几种类型的双指针，而不是“只要有两个指针就是双指针了”。

> 有了这样一个**算法框架**，或者算法思维，有很大的好处。它能帮助你理清思路，当你碰到新的问题，在脑海里进行搜索的时候，**双指针**这个词就会在你脑海里闪过，闪过的同时你可以根据**双指针**的所有套路和这道题进行**穷举匹配**，这个思考解题过程本来就像是算法，我会在进阶篇《搜索算法》中详细阐述。

那么究竟我们算法中提到的双指针指的是什么呢？我们一起来看下算法中双指针的常见题型吧。

常见题型有哪些?

这里我将其分为三种类类型，分别是：

1. 快慢指针（两个指针步长不同）
2. 左右端点指针（两个指针分别指向头尾，并往中间移动，步长不确定）
3. 固定间距指针（两个指针间距相同，步长相同）

> 上面是我自己的分类，没有参考别人。可以发现我的分类标准已经覆盖了几乎所有常见的情况。 大家在平时做题的时候一定要养成这样的习惯，将题目类型进行总结，当然这个总结可以是别人总结好的，也可以是自己独立总结的。不管是哪一种，都要进行一定的消化吸收，把它们变成真正属于自己的知识。

不管是哪一种双指针，只考虑双指针部分的话，由于最多还是会遍历整个数组一次，因此时间复杂度取决于步长，如果步长是 1，2 这种常数的话，那么时间复杂度就是 O(N)，如果步长是和数据规模有关（比如二分法），其时间复杂度就是 O(logN)。并且由于不管规模多大，我们都只需要最多两个指针，因此空间复杂度是 O(1)。下面我们就来看看双指针的常见套路有哪些。

常见套路

快慢指针

1. 判断链表是否有环

这里给大家推荐两个非常经典的题目，一个是力扣 287 题，一个是 142 题。其中 142 题我在我的 LeetCode 题解仓库中的每日一题板块出过，并且给了很详细的证明和解答。而 287 题相对不直观，比较难以想到，这道题曾被官方选定为每日一题，也是相当经典的。

- 287. 寻找重复数
- 【每日一题】- 2020-01-14 - 142. 环形链表 II · Issue #274 · azl397985856/leetcode
- 读写指针。典型的是 `删除重复元素`

这里推荐我仓库中的一道题， 我这里写了一个题解，横向对比了几个相似题目，并剖析了这种题目的本质是什么，让你看透题目本质，推荐阅读。

- 80.删除排序数组中的重复项 II

左右端点指针

1. 二分查找。

二分查找会在专题篇展开，这里不多说，大家先知道就行了。

1. 暴力枚举中“从大到小枚举”（剪枝）

一个典型的题目是我之前参加官方每日一题的时候给的一个解法，大家可以看下。这种解法是可以 AC 的。同样地，这道题我也给出了三种方法，帮助大家从多个纬度看清这个题目。强烈推荐大家做到一题多解。这对于

你做题很多帮助。除了一题多解，还有一个大招是多题同解，这部分我们放在专题篇介绍。

find-the-longest-substring-containing-vowels-in-even

1. 有序数组。

区别于上面的二分查找，这种算法指针移动是连续的，而不是跳跃性的，典型的是 LeetCode 的 `两数和` ，以及 `N数和` 系列问题。

固定间距指针

1. 一次遍历（One Pass）求链表的中点
2. 一次遍历（One Pass）求链表的倒数第 k 个元素
3. 固定窗口大小的滑动窗口

模板(伪代码)

我们来看下上面三种题目的算法框架是什么样的。这个时候我们没必要纠结具体的语言，这里我直接使用了伪代码，就是防止你掉进细节。

当你掌握了这种算法的细节，就应该找几个题目试试。一方面是检测自己是否真的掌握了，另一方面是“细节”，”细节“是人类，尤其是软件工程师最大的敌人，毕竟我们都是 `差不多先生` 。

1. 快慢指针

```
l = 0
r = 0
while 没有遍历完
  if 一定条件
    l += 1
  r += 1
return 合适的值
```

1. 左右端点指针

```
l = 0
r = n - 1
while l < r
  if 找到了
    return 找到的值
  if 一定条件1
    l += 1
  else if  一定条件2
    r -= 1
return 没找到
```

1. 固定间距指针

```
l = 0
r = k
while 没有遍历完
  自定义逻辑
  l += 1
  r += 1
return 合适的值
```

题目推荐

如果你 `差不多` 理解了上面的东西，那么可以拿下面的题练练手。Let's Go!

左右端点指针

- 16.3Sum Closest (Medium)
- 713.Subarray Product Less Than K (Medium)
- 977.Squares of a Sorted Array (Easy)
- Dutch National Flag Problem

> 下面是二分类型

- 33.Search in Rotated Sorted Array (Medium)
- 875.Koko Eating Bananas（Medium）
- 881.Boats to Save People（Medium）

快慢指针

- 26.Remove Duplicates from Sorted Array（Easy）
- 141.Linked List Cycle (Easy)
- 142.Linked List Cycle II（Medium）

- 287.Find the Duplicate Number（Medium）
- 202.Happy Number (Easy)

固定间距指针

- 1456.Maximum Number of Vowels in a Substring of Given Length（Medium）

> 固定窗口大小的滑动窗口见专题篇的滑动窗口专题（暂未发布）

其他

有时候也不能太思维定式，比如 https://leetcode-cn.com/problems/consecutive-characters/ 这道题根本就没必要双指针什么的。

再比如：https://lucifer.ren/blog/2020/05/31/101.symmetric-tree/

回炉重铸， 91 天见证不一样的自己（第二期）

力扣加加，一个努力做西湖区最好的算法题解的团队。就在今天它给大家带来了《91 天学算法》，帮助大家摆脱困境，征服算法。

力扣加加

努力做西湖区最好的算法题解

初衷

为了让想学习的人能够真正学习到东西， 我打算新开一个栏目《91 天学算法》，在 91 天内来帮助那些想要学习算法，提升自己算法能力的同学，帮助大家建立完整的算法知识体系。

群里每天都会有题目，推荐大家讨论当天的题目。我们会帮助大家规划学习路线，91 天见证不一样的自己。群里会有专门的资深算法竞赛大佬坐阵解答大家的问题和疑问，并且会对前一天的题目进行讲解。

91天学算法

活动时间

`2020-10-01` 至 `2020-12-30`

> lucifer 我刚好 12.31 生日， 好巧！

你能够得到什么？

1. 显著提高你的刷题效率，让你少走弯路
2. 掌握常见面试题的思路和解法
3. 掌握常见套路，了解常见算法的本质，横向对比各种题目
4. 纵向剖析一道题，多种方法不同角度解决同一题目

要求

- 🈲 不允许经常闲聊
- 🈲 不允许发广告，软文（只能发算法相关的技术文章）
- ✅ 一周至少参与一次打卡

> 违反上述条件的人员会被强制清退

课程大纲

> **基础篇**
>
> - 【91 算法-基础篇】01.数组，栈，队列
> - 【91 算法-基础篇】02.链表
> - 【91 算法-基础篇】03.树
> - 【91 算法-基础篇】04.哈希表
> - 【91 算法-基础篇】05.双指针
> - 【91 算法-基础篇】06.图（TODO）
>
> **进阶篇**
>
> - 【91 算法-进阶篇】01.并查集
> - 【91 算法-进阶篇】02.Trie（PDF）
>
> > 【91 算法-进阶篇】02.Trie（Markdown）
>
> - 【91 算法-进阶篇】03.KMP & RK
> - 【91 算法-进阶篇】04.跳表
> - 【91 算法-进阶篇】05.剪枝
> - 【91 算法-进阶篇】07.高频面试题
> - 【91 算法-进阶篇】08.堆
>
> **专题**
>
> - 【91 算法-专题篇】01.二分法
> - 【91 算法-专题篇】02.滑动窗口
> - 【91 算法-专题篇】03.位运算
> - 【91 算法-专题篇】04.搜索
> - 【91 算法-专题篇】05.背包问题
> - 【91 算法-专题篇】06.动态规划
> - 【91 算法-专题篇】07.分治
> - 【91 算法-专题篇】07.贪心
>
> **参考答案**
>
> - 基础篇
> - 进阶篇
> - 专题篇

部分公开的讲义：

- 【91 算法-基础篇】05.双指针
- 动态规划问题为什么要画表格?

基础篇（30 天）

1. 数组，队列，栈
2. 链表
3. 树与递归
4. 哈希表
5. 双指针

进阶篇（30 天）

1. 堆

2. 前缀树
3. 并查集
4. 跳表
5. 剪枝技巧
6. RK 和 KMP
7. 高频面试题

...

专题篇（31 天）

1. 二分法
2. 滑动窗口
3. 位运算
4. 背包问题
5. 搜索（BFS，DFS，回溯）
6. 动态规划
7. 分治
8. 贪心

...

游戏规则

- 每天会根据课程大纲的规划，出一道相关题目。
- 大家可以在指定 https://91.leetcode-solution.cn 中打卡（不可以抄作业哦），对于不会做的题目可以在群里提问。
- 第二天会对前一天的题目进行讲解。

奖励

- 对于坚持打卡满一个月的同学，可以参加抽奖，奖品包括 `算法模拟面试` ，算法相关的图书，科学上网兑换码等
- 连续打卡七天可以获得补签卡一张哦

冲鸭

报名开始时间待定。

采用 微信群的方式进行， `前 50 个进群的小伙伴免费哦 ~` ，50 名之后的小伙伴采取阶梯收费的形式。

收费标准：

- 前 50 人免费

- 51 - 100 收费 5 元
- 101 - 500 收费 10 元

想要参与的小伙伴加我，发红包拉你进群。

- 微信号：DevelopeEngineer

精选题解

这里是我以前写的题解。 这里的题解一般都是多个题目，而不是针对某一具体题目的。熟悉我的朋友应该知道了， 这是我所说的**第二阶段** 。

第一阶段按照 tag 去刷， 第二阶段则要一题多解，多题同解，挖掘题目背后的东西。而这个系列大多数就是做了这个事情。其他不是**一题多解，多题同解**的，基本就是**大厂真题解析**。

- 《日程安排》专题
- 《构造二叉树》专题
- 字典序列删除
- 百度的算法面试题 - 祖玛游戏
- 西法带你学算法】一次搞定前缀和
- 字节跳动的算法面试题是什么难度?
- 字节跳动的算法面试题是什么难度？（第二弹）
- 《我是你的妈妈呀》 - 第一期
- 一文带你看懂二叉树的序列化
- 穿上衣服我就不认识你了？来聊聊最长上升子序列
- 你的衣服我扒了 - 《最长公共子序列》
- 一文看懂《最大子序列和问题》

《我的日程安排表》截止目前（2020-02-03）在 LeetCode 上一共有三道题，其中两个中等难度，一个困难难度,分别是：

- 729. 我的日程安排表 I
- 731. 我的日程安排表 II
- 732. 我的日程安排表 III

另外 LeetCode 上有一个类似的系列《会议室》，截止目前（2020-02-03）有两道题目。其中一个简单一个中等，分别是：

- 252. 会议室
- 253. 会议室 II

今天我们就来攻克它们。

729. 我的日程安排表 I

题目地址

https://leetcode-cn.com/problems/my-calendar-i

题目描述

实现一个 MyCalendar 类来存放你的日程安排。如果要添加的时间内没有其他安排，则可以存储这个新的日程安排。

MyCalendar 有一个 book(int start, int end)方法。它意味着在 start 到 end 时间内增加一个日程安排，注意，这里的时间是半开区间，即 [start, end), 实数 x 的范围为， start <= x < end。

当两个日程安排有一些时间上的交叉时（例如两个日程安排都在同一时间内），就会产生重复预订。

每次调用 MyCalendar.book 方法时，如果可以将日程安排成功添加到日历中而不会导致重复预订，返回 true。否则，返回 false 并且不要将该日程安排添加到日历中。

请按照以下步骤调用 MyCalendar 类: MyCalendar cal = new MyCalendar(); MyCalendar.book(start, end)

示例 1:

MyCalendar(); MyCalendar.book(10, 20); // returns true MyCalendar.book(15, 25); // returns false MyCalendar.book(20, 30); // returns true 解释: 第一个日程安排可以添加到日历中. 第二个日程安排不能添加到日历中，因为时间 15 已经被第一个日程安排预定了。 第三个日程安排可以添加到日历中，因为第一个日程安排并不包含时间 20 。 说明:

每个测试用例，调用 MyCalendar.book 函数最多不超过 100 次。 调用函数 MyCalendar.book(start, end)时， start 和 end 的取值范围为 [0, 10^9]。

暴力法

思路

首先我们考虑暴力法。每插入一个元素我们都判断其是否和已有的 `所有` 课程重叠。

我们定一个函数 `intersected(calendar, calendars)` ，其中 calendar 是即将要插入的课程，calendars 是已经插入的课程。 只要 calendar 和 calendars 中的任何一个课程有交叉，我们就返回 True，否则返回 False。

对于两个 calendar，我们的判断逻辑都是一样的。假设连个 calendar 分别是 `[s1, e1]` 和 `[s2, e2]` 。那么如果 `s1 >= e2 or s2 <= e1` , 则两个课程没有交叉，可以预定，否则不可以。如图，1，2，3 可以预定，剩下的不可以。

代码是这样的：

```
    def intersected(calendar, calendars):
        for [start, end] in calendars:
            if calendar[0] >= end or calendar[1] <= start:
                continue
            else:
                return True

        return False
```

复杂度分析：

- 时间复杂度：$O(N^2)$。N 指的是日常安排的数量，对于每个新的日常安排，我们检查新的日常安排是否发生冲突来决定是否可以预订新的日常安排。
- 空间复杂度: $O(N)$。

这个代码写出来之后整体代码就呼之欲出了，全部代码见下方代码部分。

代码

代码支持 Python3:

Python3 Code:

```
#
# @lc app=leetcode.cn id=729 lang=python3
#
# [729] 我的日程安排表 I
#

# @lc code=start


class MyCalendar:

    def __init__(self):
        self.calendars = []

    def book(self, start: int, end: int) -> bool:
        def intersected(calendar, calendars):
            for [start, end] in calendars:
                if calendar[0] >= end or calendar[1] <= sta
                    continue
                else:
                    return True

            return False
        if intersected([start, end], self.calendars):
            return False
        self.calendars.append([start, end])
        return True

        # Your MyCalendar object will be instantiated and c
        # obj = MyCalendar()
        # param_1 = obj.book(start,end)
        # @lc code=end
```

实际上我们还可以换个角度，上面的思路判断交叉部分我们考虑的是“如何不交叉”，剩下的就是交叉。我们也可以直接考虑交叉。还是上面的例子，如果两个课程交叉，那么一定满足 `s1 < e2 and e1 > s2` 。基于此，我们写出下面的代码。

代码支持 Python3:

Python3 Code:

```python
#
# @lc app=leetcode.cn id=729 lang=python3
#
# [729] 我的日程安排表 I
#

# @lc code=start


class MyCalendar:

    def __init__(self):
        self.calendars = []

    def book(self, start: int, end: int) -> bool:
        for s, e in self.calendars:
            if start < e and end > s:
                return False
        self.calendars.append([start, end])
        return True

        # Your MyCalendar object will be instantiated and c
        # obj = MyCalendar()
        # param_1 = obj.book(start,end)
        # @lc code=end
```

二叉查找树法

思路

和上面思路类似，只不过我们每次都对 calendars 进行排序，那么我们可以通过二分查找日程安排的情况来检查新日常安排是否可以预订。如果每次插入之前都进行一次排序，那么时间复杂度会很高。如图，我们的 [s1,e1], [s2,e2], [s3,e3] 是按照时间顺序排好的日程安排。我们现在要插入[s,e],我们使用二分查找，找到要插入的位置，然后和插入位置的课程进行一次比对即可，这部分的时间复杂度是 $O(logN)$。

我们考虑使用平衡二叉树来维护这种动态的变化，在最差的情况时间复杂度会退化到上述的$O(N^2)$，平均情况是$O(NlogN)$，其中 N 是已预订的日常安排数。

代码

代码支持 Python3:

Python3 Code:

```
class Node:
    def __init__(self, start, end):
        self.start = start
        self.end = end
        self.left = self.right = None

    def insert(self, node):
        if node.start >= self.end:
            if not self.right:
                self.right = node
                return True
            return self.right.insert(node)
        elif node.end <= self.start:
            if not self.left:
                self.left = node
                return True
            return self.left.insert(node)
        else:
            return False

class MyCalendar(object):
    def __init__(self):
        self.root = None

    def book(self, start, end):
        if self.root is None:
            self.root = Node(start, end)
            return True
        return self.root.insert(Node(start, end))
```

731. 我的日程安排表 II

题目地址

https://leetcode-cn.com/problems/my-calendar-ii

题目描述

实现一个 MyCalendar 类来存放你的日程安排。如果要添加的时间内不会导致三重预订时，则可以存储这个新的日程安排。

MyCalendar 有一个 book(int start, int end)方法。它意味着在 start 到 end 时间内增加一个日程安排，注意，这里的时间是半开区间，即 [start, end), 实数 x 的范围为， start <= x < end。

当三个日程安排有一些时间上的交叉时（例如三个日程安排都在同一时间内），就会产生三重预订。

每次调用 MyCalendar.book 方法时，如果可以将日程安排成功添加到日历中而不会导致三重预订，返回 true。否则，返回 false 并且不要将该日程安排添加到日历中。

请按照以下步骤调用 MyCalendar 类: MyCalendar cal = new MyCalendar(); MyCalendar.book(start, end)

示例：

MyCalendar(); MyCalendar.book(10, 20); // returns true MyCalendar.book(50, 60); // returns true MyCalendar.book(10, 40); // returns true MyCalendar.book(5, 15); // returns false MyCalendar.book(5, 10); // returns true MyCalendar.book(25, 55); // returns true 解释： 前两个日程安排可以添加至日历中。 第三个日程安排会导致双重预订，但可以添加至日历中。 第四个日程安排活动（5,15）不能添加至日历中，因为它会导致三重预订。 第五个日程安排（5,10）可以添加至日历中，因为它未使用已经双重预订的时间 10。 第六个日程安排（25,55）可以添加至日历中，因为时间 [25,40] 将和第三个日程安排双重预订； 时间 [40,50] 将单独预订，时间 [50,55）将和第二个日程安排双重预订。

提示：

每个测试用例，调用 MyCalendar.book 函数最多不超过 1000 次。 调用函数 MyCalendar.book(start, end)时， start 和 end 的取值范围为 [0, 10^9]。

暴力法

思路

暴力法和上述思路类似。但是我们多维护一个数组 intersectedCalendars 用来存储**二次预定**的日程安排。如果课程第一次冲突，我们将其加入 intersectedCalendars，如果和 intersectedCalendars 也冲突了，说明出现了三次预定，我们直接返回 False。

代码

代码支持 Python3:

Python3 Code:

```
class MyCalendarTwo:

    def __init__(self):
        self.calendars = []
        self.intersectedCalendars = []

    def book(self, start: int, end: int) -> bool:
        for [s, e] in self.intersectedCalendars:
            if start < e and end > s:
                return False
        for [s, e] in self.calendars:
            if start < e and end > s:
                self.intersectedCalendars.append([max(start
        self.calendars.append([start, end])
        return True
```

二叉查找树法

和上面的题目类似，我们仍然可以使用平衡二叉树来简化查找逻辑。具体可以参考这个 discussion>)

每次插入之前我们都需要进行一次判断，判断是否可以插入。如果不可以插入，直接返回 False，否则我们进行一次插入。 插入的时候，如果和已有的相交了，我们判断是否之前已经相交了一次，如果是返回 False，否则返回 True。关于**如何判断是否和已有的相交**，我们可以在 node 节点增加一个字段的方式来标记，在这里我们使用 single_overlap，True 表示产生了二次预定，False 则表示没有产生过两次及以上的预定。

代码

代码支持 Python3:

Python3 Code:

```
class Node:
    def __init__(self, start, end):
        self.start = start
        self.end = end
        self.left = None
        self.right = None
        self.single_overlap = False

class MyCalendarTwo:

    def __init__(self):
        self.root = None

    def book(self, start, end):
        if not self.canInsert(start, end, self.root):
            return False

        self.root = self.insert(start, end, self.root)
        return True


    def canInsert(self, start, end, root):
        if not root:
            return True

        if start >= end:
            return True

        if end <= root.start:
            return self.canInsert(start, end, root.left)

        elif start >= root.end:
            return self.canInsert(start, end, root.right)

        else:
            if root.single_overlap:
                return False
            elif start >= root.start and end <= root.end:
                return True
            else:
                return self.canInsert(start, root.start, r


    def insert(self, start, end, root):
        if not root:
            root = Node(start, end)
            return root
```

```
        if start >= end:
            return root

        if start >= root.end:
            root.right = self.insert(start, end, root.righ

        elif end <= root.start:
            root.left = self.insert(start, end, root.left)

        else:
            root.single_overlap = True
            a = min(root.start, start)
            b = max(root.start, start)
            c = min(root.end, end)
            d = max(root.end, end)
            root.start, root.end = b, c
            root.left, root.right = self.insert(a, b, root.

        return root

# Your MyCalendarTwo object will be instantiated and calle
# obj = MyCalendarTwo()
# param_1 = obj.book(start,end)
```

732. 我的日程安排表 III

题目地址

https://leetcode-cn.com/problems/my-calendar-iii/

题目描述

实现一个 MyCalendar 类来存放你的日程安排，你可以一直添加新的日程安排。

MyCalendar 有一个 book(int start, int end)方法。它意味着在 start 到 end 时间内增加一个日程安排，注意，这里的时间是半开区间，即 [start, end), 实数 x 的范围为， start <= x < end。

当 K 个日程安排有一些时间上的交叉时（例如 K 个日程安排都在同一时间内），就会产生 K 次预订。

每次调用 MyCalendar.book 方法时，返回一个整数 K ，表示最大的 K 次预订。

请按照以下步骤调用 MyCalendar 类: MyCalendar cal = new MyCalendar(); MyCalendar.book(start, end)

示例 1:

MyCalendarThree(); MyCalendarThree.book(10, 20); // returns 1 MyCalendarThree.book(50, 60); // returns 1 MyCalendarThree.book(10, 40); // returns 2 MyCalendarThree.book(5, 15); // returns 3 MyCalendarThree.book(5, 10); // returns 3 MyCalendarThree.book(25, 55); // returns 3 解释: 前两个日程安排可以预订并且不相交，所以最大的 K 次预订是 1。 第三个日程安排[10,40]与第一个日程安排相交，最高的 K 次预订为 2。 其余的日程安排的最高 K 次预订仅为 3。 请注意，最后一次日程安排可能会导致局部最高 K 次预订为 2，但答案仍然是 3，原因是从开始到最后，时间[10,20]，[10,40]和[5,15]仍然会导致 3 次预订。 说明:

每个测试用例，调用 MyCalendar.book 函数最多不超过 400 次。 调用函数 MyCalendar.book(start, end)时， start 和 end 的取值范围为 [0, 10^9]。

二叉查找树法

思路

我们仍然可以使用上述的平衡二叉树的做法。只不过我们需要额外维护一个全局的最大值“k”，表示需要多少个预定。最终我们返回 k。 同时每一个 node 我们都增加一个属性 k，用来表示局部的最大值，对于每次插入，我们将 node 的 k 和全部的 k 进行比较，取出最大值即可。

代码

代码支持 Python3:

Python3 Code:

```
class Node(object):
    def __init__(self, start, end, ktime=1):
        self.k = ktime
        self.s = start
        self.e = end
        self.right = None
        self.left = None

class MyCalendarThree(object):

    def __init__(self):
        self.root = None
        self.k = 0

    def book(self, start, end):
        self.root = self.insert(self.root, start, end, 1)
        return self.k
    def insert(self, root, start, end, k):
        if start >= end:
            return root
        if not root:
            self.k = max(self.k, k)
            return Node(start, end, k)
        else:
            if start >= root.e:
                root.right = self.insert(root.right, start,
                return root
            elif end <= root.s:
                root.left = self.insert(root.left, start, e
                return root
            else:

                a = min(root.s, start)
                b = max(root.s, start)
                c = min(root.e, end)
                d = max(root.e, end)

                root.left = self.insert(root.left, a, b, a
                root.right = self.insert(root.right, c,d, c
                root.k += k
                root.s = b
                root.e = c
                self.k = max(root.k, self.k)
                return root
```

Count Map 法

思路

这个是我在看了 Discussion [C++] Map Solution, beats 95%+ 之后写的解法，解法非常巧妙。

我们使用一个 count map 来存储所有的预定，对于每次插入，我们执行 `count[start] += 1` 和 `count[end] -= 1` 。 count[t] 表示从 t 开始到下一个 t 我们有几个预定。因此我们需要对 count 进行排序才行。 我们维护一个最大值来 cnt 来表示需要的预定数。

比如预定[1,3]和[5,7]，我们产生一个预定即可：

再比如预定[1,5]和[3,7]，我们需要两个预定：

我们可以使用红黑树来简化时间复杂度，如果你使用的是 Java，可以直接使用现成的数据结构 TreeMap。我这里偷懒，每次都排序，时间复杂度会很高，但是可以 AC。

读到这里，你可能会发现： 这个解法似乎更具有通用型。对于第一题我们可以判断 cnt 是否小于等于 1，对于第二题我们可以判断 cnt 是否小于等于 2。

> 如果你不借助红黑树等数据结构直接使用 count-map 法，即每次都进行一次排序，第一题和第二题可能会直接超时。

代码

代码支持 Python3:

Python3 Code:

```
class MyCalendarThree:

    def __init__(self):
        self.count = dict()

    def book(self, start: int, end: int) -> int:
        self.count[start] = self.count.get(start, 0) + 1
        self.count[end] = self.count.get(end, 0) - 1
        cnt = 0
        cur = 0

        for k in sorted(self.count):
            cur += self.count[k]
            cnt = max(cnt, cur)
        return cnt

        # Your MyCalendarThree object will be instantiated
        # obj = MyCalendarThree()
        # param_1 = obj.book(start,end)
```

相关题目

LeetCode 上有一个类似的系列《会议室》，截止目前（2020-02-03）有两道题目。其中一个简单一个中等，解题思路非常类似，大家用这个解题思路尝试一下，检测一下自己是否已经掌握。两道题分别是：

- 252. 会议室
- 253. 会议室 II

总结

我们对 LeetCode 上的专题《我的日程安排》的三道题进行了汇总。对于区间判断是否重叠，我们可以反向判断，也可以正向判断。 暴力的方法是每次对所有的课程进行判断是否重叠，这种解法可以 AC。我们也可以进一步优化，使用二叉查找树来简化时间复杂度。最后我们介绍了一种 Count-Map 方法来通用解决所有的问题，不仅可以完美解决这三道题，还可以扩展到《会议室》系列的两道题。

构造二叉树系列

构造二叉树是一个常见的二叉树考点，相比于直接考察二叉树的遍历，这种题目的难度会更大。截止到目前(2020-02-08) LeetCode 关于构造二叉树一共有三道题目，分别是：

- 105. 从前序与中序遍历序列构造二叉树
- 106. 从中序与后序遍历序列构造二叉树
- 889. 根据前序和后序遍历构造二叉树

今天就让我们用一个套路一举攻破他们。

105. 从前序与中序遍历序列构造二叉树

题目描述

```
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
```

思路

我们以题目给出的测试用例来讲解：

前序遍历是 `根左右` ，因此 preorder 第一个元素一定整个树的根。由于题目说明了没有重复元素，因此我们可以通过 val 去 inorder 找到根在 inorder 中的索引 i。 而由于中序遍历是 `左根右` ，我们容易找到 i 左边的都是左子树，i 右边都是右子树。

我使用红色表示根，蓝色表示左子树，绿色表示右子树。

根据此时的信息，我们能构造的树是这样的：

我们 preorder 继续向后移动一位，这个时候我们得到了第二个根节点”9“，实际上就是左子树的根节点。

我们 preorder 继续向后移动一位，这个时候我们得到了第二个根节点”20“，实际上就是右子树的根节点。其中右子树由于个数大于 1，我们无法确定，我们继续执行上述逻辑。

根据此时的信息，我们能构造的树是这样的：

我们不断执行上述逻辑即可。简单起见，递归的时候每次我都开辟了新的数组，这个其实是没有必要的，我们可以通过四个变量来记录 inorder 和 preorder 的起始位置即可。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def buildTree(self, preorder: List[int], inorder: List
        # 实际上inorder 和 postorder一定是同时为空的，因此你无论
        if not preorder:
            return None
        root = TreeNode(preorder[0])

        i = inorder.index(root.val)
        root.left = self.buildTree(preorder[1:i + 1], inor
        root.right = self.buildTree(preorder[i + 1:], inor

        return root
```

复杂度分析

- 时间复杂度：由于每次递归我们的 inorder 和 preorder 的总数都会减 1，因此我们要递归 N 次，故时间复杂度为 $O(N)$，其中 N 为节点个数。
- 空间复杂度：我们使用了递归，也就是借助了额外的栈空间来完成，由于栈的深度为 N，因此总的空间复杂度为 $O(N)$，其中 N 为节点个数。

> 空间复杂度忽略了开辟数组的内存消耗。

106. 从中序与后序遍历序列构造二叉树

如果你会了上面的题目，那么这个题目对你来说也不是难事，我们来看下。

题目描述

```
根据一棵树的中序遍历与后序遍历构造二叉树。

注意:
你可以假设树中没有重复的元素。

例如，给出

中序遍历 inorder = [9,3,15,20,7]
后序遍历 postorder = [9,15,7,20,3]
返回如下的二叉树:

    3
   / \
  9  20
    /  \
   15   7
```

思路

我们以题目给出的测试用例来讲解：

后序遍历是 `左右根` ，因此 postorder 最后一个元素一定整个树的根。由于题目说明了没有重复元素，因此我们可以通过 val 去 inorder 找到根在 inorder 中的索引 i。 而由于中序遍历是 `左根右` ，我们容易找到 i 左边的都是左子树，i 右边都是右子树。

我使用红色表示根，蓝色表示左子树，绿色表示右子树。

根据此时的信息，我们能构造的树是这样的：

其中右子树由于个数大于 1，我们无法确定，我们继续执行上述逻辑。我们 postorder 继续向前移动一位，这个时候我们得到了第二个根节点”20“，实际上就是右子树的根节点。

根据此时的信息，我们能构造的树是这样的：

我们不断执行上述逻辑即可。简单起见，递归的时候每次我都开辟了新的数组，这个其实是没有必要的，我们可以通过四个变量来记录 inorder 和 postorder 的起始位置即可。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def buildTree(self, inorder: List[int], postorder: List
        # 实际上inorder 和 postorder一定是同时为空的，因此你无论
        if not inorder:
            return None
        root = TreeNode(postorder[-1])
        i = inorder.index(root.val)
        root.left = self.buildTree(inorder[:i], postorder[
        root.right = self.buildTree(inorder[i+1:], postorde

        return root
```

复杂度分析

- 时间复杂度：由于每次递归我们的 inorder 和 postorder 的总数都会减 1，因此我们要递归 N 次，故时间复杂度为 $O(N)$，其中 N 为节点个数。
- 空间复杂度：我们使用了递归，也就是借助了额外的栈空间来完成，由于栈的深度为 N，因此总的空间复杂度为 $O(N)$，其中 N 为节点个数。

> 空间复杂度忽略了开辟数组的内存消耗。

889. 根据前序和后序遍历构造二叉树

题目描述

```
返回与给定的前序和后序遍历匹配的任何二叉树。

 pre 和 post 遍历中的值是不同的正整数。



示例：

输入：pre = [1,2,4,5,3,6,7], post = [4,5,2,6,7,3,1]
输出：[1,2,3,4,5,6,7]


提示：

1 <= pre.length == post.length <= 30
pre[] 和 post[] 都是 1, 2, ..., pre.length 的排列
每个输入保证至少有一个答案。如果有多个答案，可以返回其中一个。
```

思路

我们以题目给出的测试用例来讲解：

前序遍历是 `根左右` ，因此 preorder 第一个元素一定整个树的根，preorder 第二个元素（如果存在的话）一定是左子树。由于题目说明了没有重复元素，因此我们可以通过 val 去 postorder 找到 pre[1]在 postorder 中的索引 i。 而由于后序遍历是 `左右根` ，因此我们容易得出。 postorder 中的 0 到 i(包含)是左子树，preorder 的 1 到 i+1（包含）也是左子树。

其他部分可以参考上面两题。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def constructFromPrePost(self, pre: List[int], post: L
        # 实际上pre 和 post一定是同时为空的，因此你无论判断哪个都行
        if not pre:
            return None
        node = TreeNode(pre[0])
        if len(pre) == 1:
            return node
        i = post.index(pre[1])

        node.left = self.constructFromPrePost(pre[1:i + 2],
        node.right = self.constructFromPrePost(pre[i + 2:],

        return node
```

复杂度分析

- 时间复杂度：由于每次递归我们的 postorder 和 preorder 的总数都会减 1，因此我们要递归 N 次，故时间复杂度为 $O(N)$，其中 N 为节点个数。

- 空间复杂度：我们使用了递归，也就是借助了额外的栈空间来完成，由于栈的深度为 N，因此总的空间复杂度为 $O(N)$，其中 N 为节点个数。

> 空间复杂度忽略了开辟数组的内存消耗。

总结

如果你仔细对比一下的话，会发现我们的思路和代码几乎一模一样。注意到每次递归我们的两个数组个数都会减去 1，因此我们递归终止条件不难写出，并且递归问题规模如何缩小也很容易，那就是数组总长度减去 1。

我们拿最后一个题目来说：

```
node.left = self.constructFromPrePost(pre[1:i + 2], post[:
node.right = self.constructFromPrePost(pre[i + 2:], post[i
```

我们发现 pre 被拆分为两份，pre[1:i + 2]和 pre[i + 2:]。很明显总数少了 1，那就是 pre 的第一个元素。 也就是说如果你写出一个，其他一个不用思考也能写出来。

而对于 post 也一样，post[:i + 1] 和 post[i + 1:-1]，很明显总数少了 1，那就是 post 最后一个元素。

这个解题模板足够简洁，并且逻辑清晰，大家可以用我的模板试试～

关注我

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

大家也可以关注我的公众号《力扣加加》获取更多更新鲜的 LeetCode 题解

一招吃遍力扣四道题，妈妈再也不用担心我被套路啦～

我花了几天时间，从力扣中精选了四道相同思想的题目，来帮助大家解套，如果觉得文章对你有用，记得点赞分享，让我看到你的认可，有动力继续做下去。

这就是接下来要给大家讲的四个题，其中 1081 和 316 题只是换了说法而已。

- 316. 去除重复字母(困难)
- 321. 拼接最大数(困难)
- 402. 移掉 K 位数字(中等)
- 1081. 不同字符的最小子序列（中等）

402. 移掉 K 位数字（中等）

我们从一个简单的问题入手，识别一下这种题的基本形式和套路，为之后的三道题打基础。

题目描述

```
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

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。
```

前置知识

- 数学

思路

这道题让我们从一个字符串数字中删除 k 个数字，使得剩下的数最小。也就说，我们要保持原来的数字的相对位置不变。

以题目中的 `num = 1432219, k = 3` 为例，我们需要返回一个长度为 4 的字符串，问题在于： 我们怎么才能求出这四个位置依次是什么呢?

（图 1）

暴力法的话，我们需要枚举 `C_n^(n - k)` 种序列（其中 n 为数字长度），并逐个比较最大。这个时间复杂度是指数级别的，必须进行优化。

一个思路是：

- 从左到右遍历
- 对于每一个遍历到的元素，我们决定是**丢弃**还是**保留**

问题的关键是：我们怎么知道，一个元素是应该保留还是丢弃呢？

这里有一个前置知识：**对于两个数 123a456 和 123b456，如果 a > b，那么数字 123a456 大于 数字 123b456，否则数字 123a456 小于等于数字 123b456**。也就说，两个**相同位数**的数字大小关系取决于第一个不同的数的大小。

因此我们的思路就是：

- 从左到右遍历
- 对于遍历到的元素，我们选择保留。
- 但是我们可以选择性丢弃前面相邻的元素。
- 丢弃与否的依据如上面的前置知识中阐述中的方法。

以题目中的 `num = 1432219, k = 3` 为例的图解过程如下：

（图 2）

由于没有左侧相邻元素，因此**没办法丢弃**。

（图 3）

由于 4 比左侧相邻的 1 大。如果选择丢弃左侧的 1，那么会使得剩下的数字更大（开头的数从 1 变成了 4）。因此我们仍然选择**不丢弃**。

（图 4）

由于 3 比左侧相邻的 4 小。 如果选择丢弃左侧的 4，那么会使得剩下的数字更小（开头的数从 4 变成了 3）。因此我们选择**丢弃**。

。。。

后面的思路类似，我就不继续分析啦。

然而需要注意的是，如果给定的数字是一个单调递增的数字，那么我们的算法会永远**选择不丢弃**。这个题目中要求的，我们要永远确保**丢弃** k 个矛盾。

一个简单的思路就是：

- 每次丢弃一次，k 减去 1。当 k 减到 0 ，我们可以提前终止遍历。
- 而当遍历完成，如果 k 仍然大于 0。不妨假设最终还剩下 x 个需要丢弃，那么我们需要选择删除末尾 x 个元素。

上面的思路可行，但是稍显复杂。

（图 5）

我们需要把思路逆转过来。刚才我的关注点一直是**丢弃**，题目要求我们丢弃 k 个。反过来说，不就是让我们保留 $n-k$ 个元素么？其中 n 为数字长度。 那么我们只需要按照上面的方法遍历完成之后，再截取前**n - k**个元素即可。

按照上面的思路，我们来选择数据结构。由于我们需要**保留**和**丢弃相邻**的元素，因此使用栈这种在一端进行添加和删除的数据结构是再合适不过了，我们来看下代码实现。

代码（Python）

```
class Solution(object):
    def removeKdigits(self, num, k):
        stack = []
        remain = len(num) - k
        for digit in num:
            while k and stack and stack[-1] > digit:
                stack.pop()
                k -= 1
            stack.append(digit)
        return ''.join(stack[:remain]).lstrip('0') or '0'
```

复杂度分析

- 时间复杂度：虽然内层还有一个 while 循环，但是由于每个数字最多仅会入栈出栈一次，因此时间复杂度仍然为 $O(N)$，其中 N 为数字长度。
- 空间复杂度：我们使用了额外的栈来存储数字，因此空间复杂度为 $O(N)$，其中 N 为数字长度。

> 提示： 如果题目改成求删除 k 个字符之后的最大数，我们只需要将 stack[-1] > digit 中的大于号改成小于号即可。

316. 去除重复字母（困难）

题目描述

```
给你一个仅包含小写字母的字符串，请你去除字符串中重复的字母，使得每个字

示例 1:

输入: "bcabc"
输出: "abc"
示例 2:

输入: "cbacdcbc"
输出: "acdb"
```

前置知识

- 字典序
- 数学

思路

与上面题目不同，这道题没有一个全局的删除次数 k。而是对于每一个在字符串 s 中出现的字母 c 都有一个 k 值。这个 k 是 c 出现次数 - 1。

沿用上面的知识的话，我们首先要做的就是计算每一个字符的 k，可以用一个字典来描述这种关系，其中 key 为 字符 c，value 为其出现的次数。

具体算法：

- 建立一个字典。其中 key 为 字符 c，value 为其出现的剩余次数。
- 从左往右遍历字符串，每次遍历到一个字符，其剩余出现次数 - 1.
- 对于每一个字符，如果其对应的剩余出现次数大于 1，我们**可以**选择丢弃（也可以选择不丢弃），否则不可以丢弃。
- 是否丢弃的标准和上面题目类似。如果栈中相邻的元素字典序更大，那么我们选择丢弃相邻的栈中的元素。

还记得上面题目的边界条件么？如果栈中剩下的元素大于 $n-k$，我们选择截取前 $n-k$ 个数字。然而本题中的 k 是分散在各个字符中的，因此这种思路不可行的。

不过不必担心。由于题目是要求只出现一次。我们可以在遍历的时候简单地判断其是否在栈上即可。

代码：

```
class Solution:
    def removeDuplicateLetters(self, s) -> int:
        stack = []
        remain_counter = collections.Counter(s)

        for c in s:
            if c not in stack:
                while stack and c < stack[-1] and  remain_c
                    stack.pop()
                stack.append(c)
            remain_counter[c] -= 1
        return ''.join(stack)
```

复杂度分析

- 时间复杂度：由于判断当前字符是否在栈上存在需要 $O(N)$ 的时间，因此总的时间复杂度就是 $O(N^2)$，其中 N 为字符串长度。
- 空间复杂度：我们使用了额外的栈来存储数字，因此空间复杂度为 $O(N)$，其中 N 为字符串长度。

查询给定字符是否在一个序列中存在的方法。根本上来说，有两种可能：

- 有序序列： 可以二分法，时间复杂度大致是 $O(N)$。
- 无序序列： 可以使用遍历的方式，最坏的情况下时间复杂度为 $O(N)$。我们也可以使用空间换时间的方式，使用 N的空间 换取 $O(1)$ 的时间复杂度。

由于本题中的 stack 并不是有序的，因此我们的优化点考虑空间换时间。而由于每种字符仅可以出现一次，这里使用 hashset 即可。

代码（Python）

```
class Solution:
    def removeDuplicateLetters(self, s) -> int:
        stack = []
        seen = set()
        remain_counter = collections.Counter(s)

        for c in s:
            if c not in seen:
                while stack and c < stack[-1] and  remain_c
                    seen.discard(stack.pop())
                seen.add(c)
                stack.append(c)
            remain_counter[c] -= 1
        return ''.join(stack)
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为字符串长度。
- 空间复杂度：我们使用了额外的栈和 hashset，因此空间复杂度为 $O(N)$，其中 N 为字符串长度。

> LeetCode 《1081. 不同字符的最小子序列》 和本题一样，不再赘述。

321. 拼接最大数（困难）

题目描述

```
给定长度分别为  m  和  n  的两个数组，其元素由  0-9  构成，表示两个

求满足该条件的最大数。结果返回一个表示该最大数的长度为  k  的数组。

说明: 请尽可能地优化你算法的时间和空间复杂度。

示例  1:

输入:
nums1 = [3, 4, 6, 5]
nums2 = [9, 1, 2, 5, 8, 3]
k = 5
输出:
[9, 8, 6, 5, 3]
示例 2:

输入:
nums1 = [6, 7]
nums2 = [6, 0, 4]
k = 5
输出:
[6, 7, 6, 0, 4]
示例 3:

输入:
nums1 = [3, 9]
nums2 = [8, 9]
k = 3
输出:
[9, 8, 9]
```

前置知识

- 分治
- 数学

思路

和第一道题类似，只不不过这一次是两个**数组**，而不是一个，并且是求最大数。

最大最小是无关紧要的，关键在于是两个数组，并且要求从两个数组选取的元素个数加起来一共是 k。

然而在一个数组中取 k 个数字，并保持其最小（或者最大），我们已经会了。但是如果问题扩展到两个，会有什么变化呢?

实际上，问题本质并没有发生变化。 假设我们从 nums1 中取了 k1 个，从 num2 中取了 k2 个，其中 k1 + k2 = k。而 k1 和 k2 这 两个子问题我们是会解决的。由于这两个子问题是相互独立的，因此我们只需要分别求解，然后将结果合并即可。

假如 k1 和 k2 个数字，已经取出来了。那么剩下要做的就是将这个长度分别为 k1 和 k2 的数字，合并成一个长度为 k 的数组合并成一个最大的数组。

以题目的 `nums1 = [3, 4, 6, 5] nums2 = [9, 1, 2, 5, 8, 3] k = 5` 为例。 假如我们从 num1 中取出 1 个数字，那么就要从 nums2 中取出 4 个数字。

运用第一题的方法，我们计算出应该取 nums1 的 [6]，并取 nums2 的 [9,5,8,3]。 如何将 [6] 和 [9,5,8,3]，使得数字尽可能大，并且保持相对位置不变呢?

实际上这个过程有点类似 `归并排序` 中的**治**，而上面我们分别计算 num1 和 num2 的最大数的过程类似 `归并排序` 中的**分**。

（图 6）

代码:

> 我们将从 num1 中挑选的 k1 个数组成的数组称之为 A，将从 num2 中挑选的 k2 个数组成的数组称之为 B，

```
def merge(A, B):
    ans = []
    while A or B:
        bigger = A if A > B else B
        ans.append(bigger[0])
        bigger.pop(0)
    return ans
```

这里需要说明一下。 在很多编程语言中：**如果 A 和 B 是两个数组，当前仅当 A 的首个元素字典序大于 B 的首个元素，A > B 返回 true，否则返回 false**。

比如：

```
A = [1,2]
B = [2]
A < B # True

A = [1,2]
B = [1,2,3]
A < B # False
```

以合并 [6] 和 [9,5,8,3] 为例，图解过程如下：

（图 7）

具体算法：

- 从 nums1 中 取 $min(i, len(nums1))$ 个数形成新的数组 A（取的逻辑同第一题），其中 i 等于 0,1,2, ... k。
- 从 nums2 中 对应取 $min(j, len(nums2))$ 个数形成新的数组 B（取的逻辑同第一题），其中 j 等于 k - i。
- 将 A 和 B 按照上面的 merge 方法合并

- 上面我们暴力了 k 种组合情况，我们只需要将 k 种情况取出最大值即可。

代码（Python）

```
class Solution:
    def maxNumber(self, nums1, nums2, k):

        def pick_max(nums, k):
            stack = []
            drop = len(nums) - k
            for num in nums:
                while drop and stack and stack[-1] < num:
                    stack.pop()
                    drop -= 1
                stack.append(num)
            return stack[:k]

        def merge(A, B):
            ans = []
            while A or B:
                bigger = A if A > B else B
                ans.append(bigger[0])
                bigger.pop(0)
            return ans

        return max(merge(pick_max(nums1, i), pick_max(nums2
```

复杂度分析

- 时间复杂度：pick_max 的时间复杂度为 $O(M+N)$ ，其中 M 为 nums1 的长度，N 为 nums2 的长度。 merge 的时间复杂度为 $O(k)$，再加上外层遍历所有的 k 中可能性。因此总的时间复杂度为 $O(k^2*(M+N))$。
- 空间复杂度：我们使用了额外的 stack 和 ans 数组，因此空间复杂度为 $O(max(M,N,k))$，其中 M 为 nums1 的长度，N 为 nums2 的长度。

总结

这四道题都是删除或者保留若干个字符，使得剩下的数字最小（或最大）或者字典序最小（或最大）。而解决问题的前提是要有一定**数学前提**。而基于这个数学前提，我们贪心地删除栈中相邻的字符。如果你会了这个套路，那么这四个题目应该都可以轻松解决。

`316. 去除重复字母（困难）` ，我们使用 hashmap 代替了数组的遍历查找，属于典型的空间换时间方式，可以认识到数据结构的灵活使用是多么的重要。背后的思路是怎么样的？为什么想到空间换时间的方式，我在文中也进行了详细的说明，这都是值得大家思考的问题。然而实际上，这些题目中使用的栈也都是空间换时间的思想。大家下次碰到**需要空间换取时间**的场景，是否能够想到本文给大家介绍的**栈**和**哈希表**呢？

`321. 拼接最大数（困难）` 则需要我们能够对问题进行分解，这绝对不是一件简单的事情。但是对难以解决的问题进行分解是一种很重要的技能，希望大家能够通过这道题加深这种**分治**思想的理解。 大家可以结合我之前写过的几个题解练习一下，它们分别是：

- 【简单易懂】归并排序（Python）
- 一文看懂《最大子序列和问题》

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

大家也可以关注我的公众号《力扣加加》获取更多更新鲜的 LeetCode 题解

百度的算法面试题 - 祖玛游戏

这是一道百度的算法面试题， 让我来拷拷你~。

题目地址（488. 祖玛游戏）

https://leetcode-cn.com/problems/zuma-game/

题目描述

```
回忆一下祖玛游戏。现在桌上有一串球，颜色有红色(R)，黄色(Y)，蓝色(B)，

每一次，你可以从手里的球选一个，然后把这个球插入到一串球中的某个位置上

找到插入并可以移除掉桌上所有球所需的最少的球数。如果不能移除桌上所有的

示例:
输入: "WRRBBW", "RB"
输出: -1
解释: WRRBBW -> WRR[R]BBW -> WBBW -> WBB[B]W -> WW （翻译者标

输入: "WWRRBBWW", "WRBRW"
输出: 2
解释: WWRRBBWW -> WWRR[R]BBWW -> WWBBWW -> WWBB[B]WW -> WWW

输入:"G", "GGGGG"
输出: 2
解释: G -> G[G] -> GG[G] -> empty

输入: "RBYYBBRRB", "YRBGB"
输出: 3
解释: RBYYBBRRB -> RBYY[Y]BBRRB -> RBBBRRB -> RRRB -> B ->
标注:

你可以假设桌上一开始的球中，不会有三个及三个以上颜色相同且连着的球。
桌上的球不会超过20个，输入的数据中代表这些球的字符串的名字是 "board"
你手中的球不会超过5个，输入的数据中代表这些球的字符串的名字是 "hand"。
输入的两个字符串均为非空字符串，且只包含字符 'R','Y','B','G','W'。
```

前置知识

- 回溯
- 哈希表

- 双指针

公司

- 百度

思路

面试题困难难度的题目常见的题型有：

- DP
- 设计题
- 图
- 游戏

本题就是游戏类题目。 如果你是一个前端， 说不定还会考察你如何实现一个 zuma 游戏。这种游戏类的题目，可以简单可以困难， 比如力扣经典的石子游戏，宝石游戏等。这类题目没有固定的解法。我做这种题目的思路就是先暴力模拟，再尝试优化算法瓶颈。

注意下数据范围球的数目 <= 5，因此暴力法就变得可行。基本思路是暴力枚举手上的球可以消除的地方， 我们可以使用回溯法来完成暴力枚举的过程，在回溯过程记录最小值即可。由于回溯树的深度不会超过 5，因此这种解法应该可以 AC。

上面提到的 `可以消除的地方` ，指的是**连续相同颜色 + 手上相同颜色的球大于等于 3**，这也是题目说明的消除条件。

因此我们只需要两个指针记录连续相同颜色球的位置，如果可以消除，消除即可。

如图，我们记录了连续红球的位置， 如果手上有红球， 则可以尝试将其清除，这一次决策就是回溯树（决策树）的一个分支。之后我们会撤回到这个决策分支， 尝试其他可行的决策分支。

以 board = RRBBRR ， hand 为 RRBB 为例，其决策树为：

其中虚线表示无需手动干预，系统自动消除。叶子节点末尾的黄色表示全部消除需要的手球个数。路径上的文字后面的数字表示此次消除需要的手球个数

> 如果你对回溯不熟悉，可以参考下我之前写的几篇题解：比如 46.permutations。

可以看出， 如果选择先消除中间的蓝色，则只需要一步即可完成。

关于计算连续球位置的核心代码(Python3):

```
i = 0
while i < len(board):
    j = i + 1
    while j < len(board) and board[i] == board[j]: j += 1
    # 其他逻辑

    # 更新左指针
    i = j
```

具体算法：

1. 用哈希表存储手上的球的种类和个数，这么做是为了后面**快速判断连续的球是否可以被消除**。由于题目限制手上求不会超过 5，因此哈希表的最大容量就是 5，可以认为这是一个常数的空间。
2. 回溯。

   2.1 确认可以消除的位置，算法参考上面的代码。

   2.2 判断手上是否有足够相同颜色的球可以消除。

   2.3 回溯的过程记录全局最小值。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def findMinStep(self, board: str, hand: str) -> int:
        def backtrack(board):
            if not board: return 0
            i = 0
            ans = 6
            while i < len(board):
                j = i + 1
                while j < len(board) and board[i] == board
                balls = 3 - (j - i)
                if counter[board[i]] >= balls:
                    balls = max(0, balls)
                    counter[board[i]] -= balls
                    ans = min(ans, balls + backtrack(board
                    counter[board[i]] += balls
                i = j
            return ans

        counter = collections.Counter(hand)
        ans = backtrack(board)
        return -1 if ans > 5 else ans
```

复杂度分析

- 时间复杂度：$O(2^{(}min(C,5)))$，其中 C 为连续相同颜色球的次数，比如 WWRRRR， C 就是 2， WRBDD， C 就是 4。min(C, 5) 是因为题目限定了手上球的个数不大于 5。
- 空间复杂度：$O(min(C,5)*Board)$，其中 C 为连续相同颜色球的次数，Board 为 Board 的长度。

关键点解析

- 回溯模板
- 双指针写法

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 36K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

【西法带你学算法】一次搞定前缀和

我花了几天时间，从力扣中精选了五道相同思想的题目，来帮助大家解套，如果觉得文章对你有用，记得点赞分享，让我看到你的认可，有动力继续做下去。

- 467. 环绕字符串中唯一的子字符串(中等)
- 795. 区间子数组个数(中等)
- 904. 水果成篮(中等)
- 992. K 个不同整数的子数组（困难）
- 1109. 航班预订统计(中等)

前四道题都是滑动窗口的子类型，我们知道滑动窗口适合在题目要求连续的情况下使用， 而前缀和也是如此。二者在连续问题中，对于**优化时间复杂度**有着很重要的意义。 因此如果一道题你可以用暴力解决出来，而且题目恰好有连续的限制， 那么滑动窗口和前缀和等技巧就应该被想到。

除了这几道题， 还有很多题目都是类似的套路， 大家可以在学习过程中进行体会。今天我们就来一起学习一下。

前菜

我们从一个简单的问题入手，识别一下这种题的基本形式和套路，为之后的四道题打基础。当你了解了这个套路之后， 之后做这种题就可以直接套。

需要注意的是这四道题的前置知识都是 `滑动窗口` ， 不熟悉的同学可以先看下我之前写的 滑动窗口专题（思路 + 模板）

母题 0

有 N 个的正整数放到数组 A 里，现在要求一个新的数组 B，新数组的第 i 个数 B[i]是原数组 A 第 0 到第 i 个数的和。

这道题可以使用前缀和来解决。 前缀和是一种重要的预处理，能大大降低查询的时间复杂度。我们可以简单理解为“数列的前 n 项的和”。这个概念其实很容易理解，即一个数组中，第 n 位存储的是数组前 n 个数字的和。

对 [1,2,3,4,5,6] 来说，其前缀和可以是 pre=[1,3,6,10,15,21]。我们可以使用公式 pre[i]=pre[$i-1$]+nums[i]得到每一位前缀和的值，从而通过前缀和进行相应的计算和解题。其实前缀和的概念很简单，但困难的是如何在题目中使用前缀和以及如何使用前缀和的关系来进行解题。

母题 1

如果让你求一个数组的连续子数组总个数，你会如何求？其中连续指的是数组的索引连续。 比如 [1,3,4]，其连续子数组有： `[1], [3], [4], [1,3], [3,4] , [1,3,4]` ，你需要返回 6。

一种思路是总的连续子数组个数等于：**以索引为 0 结尾的子数组个数 + 以索引为 1 结尾的子数组个数 + ... + 以索引为 n - 1 结尾的子数组个数**，这无疑是完备的。

同时**利用母题 0 的前缀和思路， 边遍历边求和。**

参考代码(JS)：

```
function countSubArray(nums) {
  let ans = 0;
  let pre = 0;
  for (_ in nums) {
    pre += 1;
    ans += pre;
  }
  return ans;
}
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$

而由于以索引为 i 结尾的子数组个数就是 i + 1，因此这道题可以直接用等差数列求和公式 `(1 + n) * n / 2` ，其中 n 数组长度。

母题 2

我继续修改下题目， 如果让你求一个数组相邻差为 1 连续子数组的总个数呢？其实就是**索引差 1 的同时，值也差 1。**

和上面思路类似，无非就是增加差值的判断。

参考代码(JS)：

```
function countSubArray(nums) {
  let ans = 1;
  let pre = 1;
  for (let i = 1; i < nums.length; i++) {
    if (nums[i] - nums[i - 1] == 1) {
      pre += 1;
    } else {
      pre = 0;
    }

    ans += pre;
  }
  return ans;
}
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$

如果我值差只要大于 1 就行呢？其实改下符号就行了，这不就是求上升子序列个数么？这里不再继续赘述， 大家可以自己试试。

母题 3

我们继续扩展。

如果我让你求出不大于 k 的子数组的个数呢？不大于 k 指的是子数组的全部元素都不大于 k。 比如 [1,3,4] 子数组有 `[1], [3], [4], [1,3], [3,4] , [1,3,4]` ，不大于 3 的子数组有 `[1], [3], [1,3]` ，那么 [1,3,4] 不大于 3 的子数组个数就是 3。 实现函数 atMostK(k, nums)。

参考代码（JS）：

```
function countSubArray(k, nums) {
  let ans = 0;
  let pre = 0;
  for (let i = 0; i < nums.length; i++) {
    if (nums[i] <= k) {
      pre += 1;
    } else {
      pre = 0;
    }

    ans += pre;
  }
  return ans;
}
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$

母题 4

如果我让你求出子数组最大值刚好是 k 的子数组的个数呢？ 比如 [1,3,4] 子数组有 `[1], [3], [4], [1,3], [3,4] , [1,3,4]` ，子数组最大值刚好是 3 的子数组有 `[3], [1,3]` ，那么 [1,3,4] 子数组最大值刚好是 3 的子数组个数就是 2。实现函数 exactK(k, nums)。

实际上是 exactK 可以直接利用 atMostK，即 atMostK(k) - atMostK(k - 1)，原因见下方母题 5 部分。

母题 5

如果我让你求出子数组最大值刚好是 介于 k1 和 k2 的子数组的个数呢？ 实现函数 betweenK(k1, k2, nums)。

实际上是 betweenK 可以直接利用 atMostK，即 atMostK(k1, nums) - atMostK(k2 - 1, nums)，其中 k1 > k2。前提是值是离散的， 比如上面我出的题都是整数。 因此我可以直接 减 1，因为 **1 是两个整数最小的间隔**。

如上， `小于等于 10 的区域` 减去 `小于 5 的区域` 就是 `大于等于 5 且小于等于 10 的区域` 。

注意我说的是小于 5， 不是小于等于 5。 由于整数是离散的，最小间隔是 1。因此小于 5 在这里就等价于 小于等于 4。这就是 betweenK(k1, k2, nums) = atMostK(k1) - atMostK(k2 - 1) 的原因。

因此不难看出 exactK 其实就是 betweenK 的特殊形式。 当 k1 == k2 的时候， betweenK 等价于 exactK。

因此 atMostK 就是灵魂方法，一定要掌握，不明白建议多看几遍。

有了上面的铺垫， 我们来看下第一道题。

467. 环绕字符串中唯一的子字符串（中等）

题目描述

```
把字符串 s 看作是“abcdefghijklmnopqrstuvwxyz”的无限环绕字符串，所

现在我们有了另一个字符串 p 。你需要的是找出 s 中有多少个唯一的 p 的非

注意: p 仅由小写的英文字母组成，p 的大小可能超过 10000。



示例 1:

输入: "a"
输出: 1
解释: 字符串 S 中只有一个"a"子字符。


示例 2:

输入: "cac"
输出: 2
解释: 字符串 S 中的字符串“cac”只有两个子串“a”、“c”。.


示例 3:

输入: "zab"
输出: 6
解释: 在字符串 S 中有六个子串“z”、“a”、“b”、“za”、“ab”、“zab”。.
```

前置知识

- 滑动窗口

思路

题目是让我们找 p 在 s 中出现的非空子串数目，而 s 是固定的一个无限循环字符串。由于 p 的数据范围是 10^5 ，因此暴力找出所有子串就需要 10^10 次操作了，应该会超时。而且题目很多信息都没用到，肯定不对。

仔细看下题目发现，这不就是母题 2 的变种么？话不多说， 直接上代码，看看有多像。

> 为了减少判断， 我这里用了一个黑科技， p 前面加了个 `^` 。

```
class Solution:
    def findSubstringInWraproundString(self, p: str) -> int
        p = '^' + p
        w = 1
        ans = 0
        for i in range(1,len(p)):
            if ord(p[i])-ord(p[i-1]) == 1 or ord(p[i])-ord(
                w += 1
            else:
                w = 1
            ans += w
        return ans
```

如上代码是有问题。 比如 `cac` 会被计算为 3，实际上应该是 2。根本原因在于 c 被错误地计算了两次。因此一个简单的思路就是用 set 记录一下访问过的子字符串即可。比如：

```
{
    c,
    abc,
    ab,
    abcd
}
```

而由于 set 中的元素一定是连续的，因此上面的数据也可以用 hashmap 存：

```
{
    c: 3
    d: 4
    b: 1
}
```

含义是：

- 以 b 结尾的子串最大长度为 1，也就是 b。
- 以 c 结尾的子串最大长度为 3，也就是 abc。
- 以 d 结尾的子串最大长度为 4，也就是 abcd。

至于 c ，是没有必要存的。我们可以通过母题 2 的方式算出来。

具体算法：

- 定义一个 len_mapper。key 是 字母， value 是 长度。 含义是以 key 结尾的最长连续子串的长度。

> 关键字是：最长

- 用一个变量 w 记录连续子串的长度，遍历过程根据 w 的值更新 len_mapper
- 返回 len_mapper 中所有 value 的和。

比如: abc，此时的 len_mapper 为:

```
{
    c: 3
    b: 2
    a: 1
}
```

再比如：abcab，此时的 len_mapper 依旧。

再比如: abcazabc，此时的 len_mapper：

```
{
    c: 4
    b: 3
    a: 2
    z: 1
}
```

这就得到了去重的目的。这种算法是不重不漏的，因为最长的连续子串一定是包含了比它短的连续子串，这个思想和 1297. 子串的最大出现次数 剪枝的方法有异曲同工之妙。

代码（Python）

```
class Solution:
    def findSubstringInWraproundString(self, p: str) -> int
        p = '^' + p
        len_mapper = collections.defaultdict(lambda: 0)
        w = 1
        for i in range(1,len(p)):
            if ord(p[i])-ord(p[i-1]) == 1 or ord(p[i])-ord(
                w += 1
            else:
                w = 1
            len_mapper[p[i]] = max(len_mapper[p[i]], w)
        return sum(len_mapper.values())
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为字符串 p 的长度。

- 空间复杂度：由于最多存储 26 个字母， 因此空间实际上是常数，故空间复杂度为 $O(1)$。

795. 区间子数组个数（中等）

题目描述

```
给定一个元素都是正整数的数组 A ，正整数 L  以及  R (L <= R)。

求连续、非空且其中最大元素满足大于等于 L  小于等于 R 的子数组个数。

例如 :
输入:
A = [2, 1, 4, 3]
L = 2
R = 3
输出: 3
解释: 满足条件的子数组: [2], [2, 1], [3].
注意:

L, R  和  A[i] 都是整数，范围在  [0, 10^9]。
数组  A  的长度范围在[1, 50000]。
```

前置知识

- 滑动窗口

思路

由母题 5，我们知道 **betweenK 可以直接利用 atMostK，即 atMostK(k1) - atMostK(k2 - 1)，其中 k1 > k2**。

由母题 2，我们知道如何求满足一定条件（这里是元素都小于等于 R）子数组的个数。

这两个结合一下， 就可以解决。

代码（Python）

> 代码是不是很像

```
class Solution:
    def numSubarrayBoundedMax(self, A: List[int], L: int, R
        def notGreater(R):
            ans = cnt = 0
            for a in A:
                if a <= R: cnt += 1
                else: cnt = 0
                ans += cnt
            return  ans

        return notGreater(R) - notGreater(L - 1)
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$。

904. 水果成篮（中等）

题目描述

```
在一排树中，第 i 棵树产生 tree[i] 型的水果。
你可以从你选择的任何树开始，然后重复执行以下步骤：

把这棵树上的水果放进你的篮子里。如果你做不到，就停下来。
移动到当前树右侧的下一棵树。如果右边没有树，就停下来。
请注意，在选择一颗树后，你没有任何选择：你必须执行步骤 1，然后执行步骤

你有两个篮子，每个篮子可以携带任何数量的水果，但你希望每个篮子只携带一

用这个程序你能收集的水果树的最大总量是多少？



示例 1：

输入：[1,2,1]
输出：3
解释：我们可以收集 [1,2,1]。
示例 2：

输入：[0,1,2,2]
输出：3
解释：我们可以收集 [1,2,2]
如果我们从第一棵树开始，我们将只能收集到 [0, 1]。
示例 3：

输入：[1,2,3,2,2]
输出：4
解释：我们可以收集 [2,3,2,2]
如果我们从第一棵树开始，我们将只能收集到 [1, 2]。
示例 4：

输入：[3,3,3,1,2,1,1,2,3,3,4]
输出：5
解释：我们可以收集 [1,2,1,1,2]
如果我们从第一棵树或第八棵树开始，我们将只能收集到 4 棵水果树。



提示：

1 <= tree.length <= 40000
0 <= tree[i] < tree.length
```

前置知识

- 滑动窗口

思路

题目花里胡哨的。我们来抽象一下，就是给你一个数组， 让你**选定一个子数组， 这个子数组最多只有两种数字**，这个选定的子数组最大可以是多少。

这不就和母题 3 一样么？只不过 k 变成了固定值 2。另外由于题目要求整个窗口最多两种数字，我们用哈希表存一下不就好了吗？

> set 是不行了的。 因此我们不但需要知道几个数字在窗口， 我们还要知道每个数字出现的次数，这样才可以使用滑动窗口优化时间复杂度。

代码（Python）

```
class Solution:
    def totalFruit(self, tree: List[int]) -> int:
        def atMostK(k, nums):
            i = ans = 0
            win = defaultdict(lambda: 0)
            for j in range(len(nums)):
                if win[nums[j]] == 0: k -= 1
                win[nums[j]] += 1
                while k < 0:
                    win[nums[i]] -= 1
                    if win[nums[i]] == 0: k += 1
                    i += 1
                ans = max(ans, j - i + 1)
            return ans

        return atMostK(2, tree)
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(k)$。

992. K 个不同整数的子数组（困难）

题目描述

```
给定一个正整数数组 A，如果 A 的某个子数组中不同整数的个数恰好为 K，则

（例如，[1,2,3,1,2] 中有 3 个不同的整数：1，2，以及 3。）

返回 A 中好子数组的数目。



示例 1：

输入：A = [1,2,1,2,3], K = 2
输出：7
解释：恰好由 2 个不同整数组成的子数组：[1,2], [2,1], [1,2], [2,3]
示例 2：

输入：A = [1,2,1,3,4], K = 3
输出：3
解释：恰好由 3 个不同整数组成的子数组：[1,2,1,3], [2,1,3], [1,3,4


提示：

1 <= A.length <= 20000
1 <= A[i] <= A.length
1 <= K <= A.length
```

前置知识

- 滑动窗口

思路

由母题 5，知：exactK = atMostK(k) - atMostK(k - 1)， 因此答案便呼之欲出了。其他部分和上面的题目 `904. 水果成篮` 一样。

> 实际上和所有的滑动窗口题目都差不多。

代码（Python）

```python
class Solution:
    def subarraysWithKDistinct(self, A, K):
        return self.atMostK(A, K) - self.atMostK(A, K - 1)

    def atMostK(self, A, K):
        counter = collections.Counter()
        res = i = 0
        for j in range(len(A)):
            if counter[A[j]] == 0:
                K -= 1
            counter[A[j]] += 1
            while K < 0:
                counter[A[i]] -= 1
                if counter[A[i]] == 0:
                    K += 1
                i += 1
            res += j - i + 1
        return res
```

复杂度分析

- 时间复杂度：$O(N)$，中 N 为数组长度。
- 空间复杂度：$O(k)$。

1109. 航班预订统计（中等）

题目描述

```
这里有 n 个航班，它们分别从 1 到 n 进行编号。

我们这儿有一份航班预订表，表中第 i 条预订记录 bookings[i] = [i,

请你返回一个长度为 n 的数组 answer，按航班编号顺序返回每个航班上预订



示例：

输入：bookings = [[1,2,10],[2,3,20],[2,5,25]], n = 5
输出：[10,55,45,25,25]



提示：

1 <= bookings.length <= 20000
1 <= bookings[i][0] <= bookings[i][1] <= n <= 20000
1 <= bookings[i][2] <= 10000
```

前置知识

- 前缀和

思路

这道题的题目描述不是很清楚。我简单分析一下题目：

[i, j, k] 其实代表的是 第 i 站上来了 k 个人， 一直到 第 j 站都在飞机上，到第 j + 1 就不在飞机上了。所以第 i 站到第 j 站的**每一站**都会因此多 k 个人。

理解了题目只会不难写出下面的代码。

```python
class Solution:
    def corpFlightBookings(self, bookings: List[List[int]],
        counter = [0] * n

        for i, j, k in bookings:
            while i <= j:
                counter[i - 1] += k
                i += 1
        return counter
```

如上的代码复杂度太高，无法通过全部的测试用例。

注意到里层的 while 循环是连续的数组全部加上一个数字，不难想到可以利用母题 0 的前缀和思路优化。

一种思路就是在 i 的位置 + k， 然后利用前缀和的技巧给 i 到 n 的元素都加上 k。但是题目需要加的是一个区间， j + 1 及其之后的元素会被多加一个 k。一个简单的技巧就是给 j + 1 的元素减去 k，这样正负就可以抵消。

代码（Python）

```
class Solution:
    def corpFlightBookings(self, bookings: List[List[int]],
        counter = [0] * (n + 1)

        for i, j, k in bookings:
            counter[i - 1] += k
            if j < n: counter[j] -= k
        for i in range(n + 1):
            counter[i] += counter[i - 1]
        return counter[:-1]
```

复杂度分析

- 时间复杂度：$O(N)$，中 N 为数组长度。
- 空间复杂度：$O(N)$。

总结

这几道题都是滑动窗口和前缀和的思路。力扣类似的题目还真不少，大家只有多留心，就会发现这个套路。

前缀和的技巧以及滑动窗口的技巧都比较固定，且有模板可套。 难点就在于我怎么才能想到可以用这个技巧呢?

我这里总结了两点：

1. 找关键字。比如题目中有连续，就应该条件反射想到滑动窗口和前缀和。比如题目求最大最小就想到动态规划和贪心等等。想到之后，就可以和题目信息对比快速排除错误的算法，找到可行解。这个思考的时间会随着你的题感增加而降低。
2. 先写出暴力解，然后找暴力解的瓶颈， 根据瓶颈就很容易知道应该用什么数据结构和算法去优化。

最后推荐几道类似的题目， 供大家练习，一定要自己写出来才行哦。

- 303. 区域和检索 - 数组不可变
- 1186.删除一次得到子数组最大和
- 1310. 子数组异或查询
- 1371. 每个元音包含偶数次的最长子字符串

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。

更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 36K star 啦。

大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

字节跳动的算法面试题是什么难度?

由于 lucifer 我是一个小前端， 最近也在准备写一个《前端如何搞定算法面试》的专栏，因此最近没少看各大公司的面试题。都说字节跳动算法题比较难，我就先拿 ta 下手，做了几套 。这次我们就拿一套 `2018 年的前端校招（第四批）` 来看下字节的算法笔试题的难度几何。地址：https://www.nowcoder.com/test/8536639/summary

> 实际上，这套字节的前端岗位笔试题和后端以及算法岗位的笔试题也只有一道题目（红包的设计题被换成了另外一个设计题）不一样而已，因此也不需要担心你不是前端，题目类型和难度和你的岗位不匹配。

这套题一共四道题， 两道问答题， 两道编程题。

其中一道问答题是 LeetCode 426 的原题，只不过题型变成了找茬（改错）。可惜的是 LeetCode 的 426 题是一个会员题目，没有会员的就看不来了。不过，剑指 Offer 正好也有这个题，并且力扣将剑指 Offer 全部的题目都 OJ 化了。 这道题大家可以去 https://leetcode-cn.com/problems/er-cha-sou-suo-shu-yu-shuang-xiang-lian-biao-lcof 提交答案。简单说一下这个题目的思路，我们只需要中序遍历即可得到一个有序的数列，同时在中序遍历过程中将 pre 和 cur 节点通过指针串起来即可。

另一个问答是红包题目，这里不多说了。我们重点看一下剩下两个算法编程题。

> 两个问答题由于不能在线判题，我没有做，只做了剩下两个编程题。

球队比赛

第一个编程题是一个球队比赛的题目。

题目描述

有三只球队，每只球队编号分别为球队 1，球队 2，球队 3，这三只球队一共需要进行 n 场比赛。现在已经踢完了 k 场比赛，每场比赛不能打平，踢赢一场比赛得一分，输了不得分不减分。已知球队 1 和球队 2 的比分相差 d1 分，球队 2 和球队 3 的比分相差 d2 分，每场比赛可以任意选择两只队伍进行。求如果打完最后的 (n-k) 场比赛，有没有可能三只球队的分数打平。

思路

假设球队 1，球队 2，球队 3 此时的胜利次数分别为 a，b，c，球队 1，球队 2，球队 3 总的胜利次数分别为 n1，n2，n3。

我一开始的想法是只要保证 n1，n2，n3 相等且都小于等于 n / 3 即可。如果题目给了 n1，n2，n3 的值就直接：

```
print(n1 == n2 == n3 == n / 3)
```

可是不仅 n1，n2，n3 没给， a，b，c 也没有给。

实际上此时我们的信息仅仅是：

```
① a + b + c = k
② a - b = d1 or b - a = d1
③ b - c = d2 or c - b = d2
```

其中 k 和 d1，d2 是已知的。a ，b，c 是未知的。 也就是说我们需要枚举所有的 a，b，c 可能性，解方程求出合法的 a，b，c，并且 合法的 a，b，c 都小于等于 n / 3 即可。

> 这个 a,b,c 的求解数学方程就是中学数学难度， 三个等式化简一下即可，具体见下方代码区域。

- a 只需要再次赢得 n / 3 - a 次
- b 只需要再次赢得 n / 3 - b 次
- c 只需要再次赢得 n / 3 - c 次

```
n1 = a + n / 3 - a = n / 3
n2 = b + (n / 3 - b) = n / 3
n3 = c + (n / 3 - c) = n / 3
```

代码(Python)

> 牛客有点让人不爽， 需要 print 而不是 return

```
t = int(input())
for i in range(t):
    n, k, d1, d2 = map(int, input().split(" "))
    if n % 3 != 0:
        print('no')
        continue
    abcs = []
    for r1 in [-1, 1]:
        for r2 in [-1, 1]:
            a = (k + 2 * r1 * d1 + r2 * d2) / 3
            b = (k + -1 * r1 * d1 + r2 * d2) / 3
            c = (k + -1 * r1 * d1 + -2 * r2 * d2) / 3
            a + r1
            if  0 <= a <= k and 0 <= b <= k and 0 <= c <= k
                abcs.append([a, b, c])
    flag = False
    for abc in abcs:
        if len(abc) > 0 and max(abc) <= n / 3:
            flag = True
            break
    if flag:
        print('yes')
    else:
        print('no')
```

复杂度分析

- 时间复杂度：$O(t)$
- 空间复杂度：$O(t)$

小结

感觉这个难度也就是力扣中等水平吧，力扣也有一些数学等式转换的题目， 比如 494.target-sum

转换字符串

题目描述

有一个仅包含'a'和'b'两种字符的字符串 s，长度为 n，每次操作可以把一个字符做一次转换（把一个'a'设置为'b'，或者把一个'b'置成'a'）；但是操作的次数有上限 m，问在有限的操作数范围内，能够得到最大连续的相同字符的子串的长度是多少。

思路

看完题我就有种似曾相识的感觉。

> 每次对妹子说出这句话的时候，她们都会觉得好假 ^_^

不过这次是真的。 ”哦，不！每次都是真的“。 这道题其实就是我之前写的滑动窗口的一道题【1004. 最大连续 1 的个数 III】滑动窗口（Python3）的换皮题。 专题地址：https://github.com/azl397985856/leetcode/blob/master/thinkings/slide-window.md

所以说，如果这道题你完全没有思路的话。说明：

- 抽象能力不够。
- 滑动窗口问题理解不到位。

第二个问题可以看我上面贴的地址，仔细读读，并完成课后练习即可解决。

第一个问题就比较困难了， 不过多看我的题解也可以慢慢提升的。比如：

- 《割绳子》 实际上就是 343. 整数拆分 的换皮题。
- 力扣 230 和 力扣 645 就是换皮题，详情参考位运算专题
- 以及 你的衣服我扒了 - 《最长公共子序列》
- 以及 穿上衣服我就不认识你了？来聊聊最长上升子序列
- 以及 一招吃遍力扣四道题，妈妈再也不用担心我被套路啦～
- 等等

回归这道题。其实我们只需要稍微抽象一下， 就是一个纯算法题。 抽象的另外一个好处则是将很多不同的题目返璞归真，从而可以在茫茫题海中逃脱。这也是我开启《我是你的妈妈呀》 的原因之一。

如果我们把 a 看成是 0 ， b 看成是 1。或者将 b 看成 1， a 看成 0。不就抽象成了：

```
给定一个由若干 0 和 1 组成的数组 A，我们最多可以将 m 个值从 0 变成

返回仅包含 1 的最长（连续）子数组的长度。
```

这就是 力扣 1004. 最大连续 1 的个数 III 原题。

因此实际上我们要求的是上面两种情况:

1. a 表示 0， b 表示 1
2. a 表示 1， b 表示 0

的较大值。

> lucifer 小提示： 其实我们也可以仅仅考虑一种情况，比如 a 看成是 0 ， b 看成是 1。这个时候， 我们操作变成了两种情况，0 变成 1 或者 1 变成 0，同时求解的也变成了最长连续 0 或者 最长连续 1 。由于这种抽象操作起来更麻烦， 我们不考虑。

问题得到了抽象就好解决了。我们只需要记录下加入窗口的是 0 还是 1：

- 如果是 1，我们什么都不用做
- 如果是 0，我们将 m 减 1

相应地，我们需要记录移除窗口的是 0 还是 1:

- 如果是 1，我们什么都不做
- 如果是 0，说明加进来的时候就是 1，加进来的时候我们 m 减去了 1，这个时候我们再加 1。

> lucifer 小提示： 实际上题目中是求**连续** a 或者 b 的长度。看到连续，大家也应该有滑动窗口的敏感度， 别管行不行， 想到总该有的。

我们拿 A = [1, 1, 0, 1, 0, 1], m = 1 来说。看下算法的具体过程：

> lucifer 小提示： 左侧的数字表示此时窗口大小，黄色格子表示修补的墙，黑色方框表示的是窗口。

这里我形象地将 0 看成是洞，1 看成是墙， 我们的目标就是补洞，使得连续的墙最长。

每次碰到一个洞，我们都去不加选择地修补。由于 m 等于 1， 也就是说我们最多补一个洞。因此需要在修补超过一个洞的时候，我们需要调整窗口范围，使得窗口内最多修补一个墙。由于窗口表示的就是连续的墙（已有的或者修补的），因此最终我们返回窗口的最大值即可。

> 由于下面的图窗口内有两个洞，这和”最多补一个洞“冲突， 我们需要收缩窗口使得满足“最多补一个洞”的先决条件。

因此最大的窗口就是 max(2, 3, 4, ...) = 4。

> lucifer 小提示： 可以看出我们不加选择地修补了所有的洞，并调整窗口，使得窗口内最多有 m 个修补的洞，因此窗口的最大值就是答案。然而实际上，我们并不需要真的”修补“（0 变成 1），而是仅仅修改 m 的值即可。

我们先来看下抽象之后的**其中一种情况**的代码：

```
class Solution:
    def longestOnes(self, A: List[int], m: int) -> int:
        i = 0
        for j in range(len(A)):
            m -= 1 - A[j]
            if m < 0:
                m += 1 - A[i]
                i += 1
        return j - i + 1
```

因此**完整代码**就是：

```
class Solution:
    def longestOnes(self, A: List[int], m: int) -> int:
        i = 0
        for j in range(len(A)):
            m -= 1 - A[j]
            if m < 0:
                m += 1 - A[i]
                i += 1
        return j - i + 1
    def longestAorB(self, A:List[int], m: int) -> int:
        return max(self.longestOnes(map(lambda x: 0 if x ==
```

这里的两个 map 会生成两个不同的数组。 我只是为了方便大家理解才新建的两个数组， 实际上根本不需要，具体见后面的代码.

代码(Python)

```
i = 0
n, m = map(int, input().split(" "))
s = input()
ans = 0
k = m #   存一下，后面也要用这个初始值
# 修补 b
for j in range(n):
    m -= ord(s[j]) - ord('a')
    if m < 0:
        m += ord(s[i]) - ord('a')
        i += 1
ans = j - i + 1
i = 0
# 修补 a
for j in range(n):
    k += ord(s[j]) - ord('b')
    if k < 0:
        k -= ord(s[i]) - ord('b')
        i += 1
print(max(ans, j - i + 1))
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

小结

这道题就是一道换了皮的力扣题，难度中等。如果你能将问题抽象，同时又懂得滑动窗口，那这道题就很容易。我看了题解区的参考答案， 内容比较混乱，不够清晰。这也是我写下这篇文章的原因之一。

总结

这一套字节跳动的题目一共四道，一道设计题，三道算法题。

其中三道算法题从难度上来说，基本都是中等难度。从内容来看，基本都是力扣的换皮题。但是如果我不说他们是换皮题， 你们能发现么？ 如果你可以的话，说明你的抽象能力已经略有小成了。如果看不出来也没有关系，关注我。 手把手扒皮给你们看，扒多了慢慢就会了。切记，不要盲目做题！如果你做了很多题， 这几道题还是看不出套路，说明你该缓缓，改变下刷题方式了。

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 36K+ star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

字节跳动的算法面试题是什么难度？（第二弹）

由于 lucifer 我是一个小前端， 最近也在准备写一个《前端如何搞定算法面试》的专栏，因此最近没少看各大公司的面试题。都说字节跳动算法题比较难，我就先拿 ta 下手，做了几套 。这次我们就拿一套 `字节跳动2017秋招编程题汇总` 来看下字节的算法笔试题的难度几何。地址：https://www.nowcoder.com/test/6035789/summary

这套题一共 11 道题， 三道编程题， 八道问答题。本次给大家带来的就是这三道编程题。更多精彩内容，请期待我的搞定算法面试专栏。

其中有一道题《异或》我没有通过所有的测试用例， 小伙伴可以找找茬，第一个找到并在公众号力扣加加留言的小伙伴奖励现金红包 10 元。

1. 头条校招

题目描述

```
头条的 2017 校招开始了！为了这次校招，我们组织了一个规模宏大的出题团队
a<=b<=c
b-a<=10
c-b<=10
所有出题人一共出了 n 道开放性题目。现在我们想把这 n 道题分布到若干场考

输入描述:
输入的第一行包含一个整数 n，表示目前已经出好的题目数量。

第二行给出每道题目的难度系数 d1,d2,...,dn。

数据范围

对于 30%的数据，1 ≤ n,di ≤ 5;

对于 100%的数据，1 ≤ n ≤ 10^5,1 ≤ di ≤ 100。

在样例中，一种可行的方案是添加 2 个难度分别为 20 和 50 的题目，这样可

输出描述:
输出只包括一行，即所求的答案。
示例 1
输入
4
20 35 23 40
输出
2
```

思路

这道题看起来很复杂， 你需要考虑很多的情况。，属于那种没有技术含量，但是考验编程能力的题目，需要思维足够严密。这种**模拟的题目**，就是题目让我干什么我干什么。 类似之前写的囚徒房间问题，约瑟夫环也是模拟，只不过模拟之后需要你剪枝优化。

这道题的情况其实很多， 我们需要考虑每一套题中的难度情况， 而不需要考虑不同套题的难度情况。题目要求我们满足： `a<=b<=c b-a<=10 c-b<=10` ，也就是题目难度从小到大排序之后，相邻的难度不能大于 10 。

因此我们的思路就是先排序，之后从小到大遍历，如果满足相邻的难度不大于 10 ，则继续。如果不满足， 我们就只能让字节的老师出一道题使得满足条件。

由于只需要比较同一套题目的难度，因此我的想法就是**比较同一套题目的第二个和第一个，以及第三个和第二个的 diff**。

- 如果 diff 小于 10，什么都不做，继续。

- 如果 diff 大于 10，我们必须补充题目。

这里有几个点需要注意。

对于第二题来说：

- 比如 1 **30** 40 这样的难度。 我可以在 1，30 之间加一个 21，这样 1，21，30 就可以组成一一套。
- 比如 1 **50** 60 这样的难度。 我可以在 1，50 之间加 21， 41 才可以组成一套，自身（50）是无论如何都没办法组到这套题中的。

不难看出， 第二道题的临界点是 diff = 20 。 小于等于 20 都可以将自身组到套题，增加一道即可，否则需要增加两个，并且自身不能组到当前套题。

对于第三题来说：

- 比如 1 20 **40**。 我可以在 20，40 之间加一个 30，这样 1，20，30 就可以组成一一套，自身（40）是无法组到这套题的。
- 比如 1 20 **60**。 也是一样的，我可以在 20，60 之间加一个 30，自身（60）同样是没办法组到这套题中的。

不难看出， 第三道题的临界点是 diff = 10 。 小于等于 10 都可以将自身组到套题，否则需要增加一个，并且自身不能组到当前套题。

这就是所有的情况了。

有的同学比较好奇，我是怎么思考的。 我是怎么**保障不重不漏**的。

实际上，这道题就是一个决策树， 我画个决策树出来你就明白了。

> 图中红色边框表示自身可以组成套题的一部分， 我也用文字进行了说明。#2 代表第二题， #3 代表第三题。

从图中可以看出， 我已经考虑了所有情况。如果你能够像我一样画出这个决策图，我想你也不会漏的。当然我的解法并不一定是最优的，不过确实是一个非常好用，具有普适性的思维框架。

需要特别注意的是，由于需要凑整， 因此你需要使得题目的总数是 3 的倍数向上取整。

代码

```
n = int(input())
nums = list(map(int, input().split()))
cnt = 0
cur = 1
nums.sort()
for i in range(1, n):
    if cur == 3:
        cur = 1
        continue
    diff = nums[i] - nums[i - 1]
    if diff <= 10:
        cur += 1
    if 10 < diff <= 20:
        if cur == 1:
            cur = 3
        if cur == 2:
            cur = 1
        cnt += 1
    if diff > 20:
        if cur == 1:
            cnt += 2
        if cur == 2:
            cnt += 1
        cur = 1
print(cnt + 3 - cur)
```

复杂度分析

- 时间复杂度：由于使用了排序， 因此时间复杂度为 $O(NlogN)$。（假设使用了基于比较的排序）
- 空间复杂度：$O(1)$

2. 异或

题目描述

```
给定整数 m 以及 n 各数字 A1,A2,..An，将数列 A 中所有元素两两异或，

输入描述:
第一行包含两个整数 n,m.

第二行给出 n 个整数 A1，A2，...，An。

数据范围

对于 30%的数据，1 <= n, m <= 1000

对于 100%的数据，1 <= n, m, Ai <= 10^5

输出描述:
输出仅包括一行，即所求的答案

输入例子 1:
3 10
6 5 10

输出例子 1:
2
```

前置知识

- 异或运算的性质
- 如何高效比较两个数的大小（从高位到低位）

首先普及一下前置知识。 第一个是异或运算：

异或的性质：两个数字异或的结果 a^b 是将 a 和 b 的二进制每一位进行运算，得出的数字。 运算的逻辑是如果同一位的数字相同则为 0，不同则为 1

异或的规律：

1. 任何数和本身异或则为 0
2. 任何数和 0 异或是本身
3. 异或运算满足交换律，即： a ^ b ^ c = a ^ c ^ b

同时建议大家去看下我总结的几道位运算的经典题目。 位运算系列

其次要知道一个常识， 即比较两个数的大小， 我们是从高位到低位比较，这样才比较高效。

比如：

```
123
456
1234
```

这三个数比较大小， 为了方便我们先补 0 ，使得大家的位数保持一致。

```
0123
0456
1234
```

先比较第一位，1 比较 0 大， 因此 1234 最大。再比较第二位， 4 比 1 大， 因此 456 大于 123，后面位不需要比较了。这其实就是剪枝的思想。

有了这两个前提，我们来试下暴力法解决这道题。

思路

暴力法就是枚举 $N^2/2$ 中组合， 让其两两按位异或，将得到的结果和 m 进行比较， 如果比 m 大， 则计数器 + 1， 最后返回计数器的值即可。

暴力的方法就如同题目描述的那样， 复杂度为 N^2。 一定过不了所有的测试用例， 不过大家实在没有好的解法的情况可以兜底。不管是牛客笔试还是实际的面试都是可行的。

接下来，让我们来**分析一下暴力为什么低效，以及如何选取数据结构和算法能够使得这个过程变得高效。** 记住这句话， 几乎所有的优化都是基于这种思维产生的，除非你开启了上帝模式，直接看了答案。 只不过等你熟悉了之后，这个思维过程会非常短， 以至于变成条件反射， 你感觉不到有这个过程， 这就是**有了题感。**

其实我刚才说的第二个前置知识就是我们优化的关键之一。

我举个例子， 比如 3 和 5 按位异或。

3 的二进制是 011， 5 的二进制是 101，

```
011
101
```

按照我前面讲的异或知识， 不难得出其异或结构就是 110。

上面我进行了三次异或：

1. 第一次是最高位的 0 和 1 的异或， 结果为 1。
2. 第二次是次高位的 1 和 0 的异或， 结果为 1。
3. 第三次是最低位的 1 和 1 的异或， 结果为 0。

那如何 m 是 1 呢？ 我们有必要进行三次异或么？ 实际上进行第一次异或的时候已经知道了一定比 m（m 是 1） 大。因为第一次异或的结构导致其最高位为 1，也就是说其最小也不过是 100，也就是 4，一定是大于 1 的。这就是**剪枝**， 这就是算法优化的关键。

> 看出我一步一步的思维过程了么？所有的算法优化都需要经过类似的过程。

因此我的算法就是从高位开始两两异或，并且异或的结果和 m 对应的二进制位比较大小。

- 如果比 m 对应的二进制位大或者小，我们提前退出即可。
- 如果相等，我们继续往低位移动重复这个过程。

这虽然已经剪枝了，但是极端情况下，性能还是很差。比如：

```
m: 1111
a: 1010
b: 0101
```

a，b 表示两个数，我们比较到最后才发现，其异或的值和 m 相等。因此极端情况，算法效率没有得到改进。

这里我想到了一点，就是如果一个数 a 的前缀和另外一个数 b 的前缀是一样的，那么 c 和 a 或者 c 和 b 的异或的结构前缀部分一定也是一样的。比如：

```
a: 111000
b: 111101
c: 101011
```

a 和 b 有共同的前缀 111，c 和 a 异或过了，当再次和 b 异或的时候，实际上前三位是没有必要进行的，这也是重复的部分。这就是算法可以优化的部分， 这就是剪枝。

分析算法，找到算法的瓶颈部分，然后选取合适的数据结构和算法来优化到。 这句话很重要， 请务必记住。

在这里，我们用的就是剪枝技术，关于剪枝，91 天学算法也有详细的介绍。

回到前面讲到的算法瓶颈， 多个数是有共同前缀的， 前缀部分就是我们浪费的运算次数， 说到前缀大家应该可以想到前缀树。如果不熟悉前缀树的话，看下我的这个前缀树专题，里面的题全部手写一遍就差不多了。

因此一种想法就是建立一个前缀树， **树的根就是最高的位。** 由于题目要求异或， 我们知道异或是二进制的位运算， 因此这棵树要存二进制才比较好。

反手看了一眼数据范围：m, n<=10^5 。 10^5 = 2 ^ x，我们的目标是求出满足条件的 x 的 ceil（向上取整），因此 x 应该是 17。

树的每一个节点存储的是：**n 个数中，从根节点到当前节点形成的前缀有多少个是一样的**，即多少个数的前缀是一样的。这样可以剪枝，提前退出的时候，就直接取出来用了。比如异或的结果是 1， m 当前二进制位是 0 ，那么这个前缀有 10 个，我都不需要比较了， 计数器直接 + 10 。

> 我用 17 直接复杂度过高，目前仅仅通过了 70 % - 80 % 测试用例，希望大家可以帮我找找毛病，我猜测是语言的锅。

代码

```
class TreeNode:
    def __init__(self):
        self.cnt = 1
        self.children = [None] * 2
def solve(num, i, cur):
    if cur == None or i == -1: return 0
    bit = (num >> i) & 1
    mbit = (m >> i) & 1
    if bit == 0 and mbit == 0:
        return (cur.children[1].cnt if cur.children[1] else
    if bit == 1 and mbit == 0:
        return (cur.children[0].cnt if cur.children[0] else
    if bit == 0 and mbit == 1:
        return solve(num, i - 1, cur.children[1])
    if bit == 1 and mbit == 1:
        return solve(num, i - 1, cur.children[0])

def preprocess(nums, root):
    for num in nums:
        cur = root
        for i in range(16, -1, -1):
            bit = (num >> i) & 1
            if cur.children[bit]:
                cur.children[bit].cnt += 1
            else:
                cur.children[bit] = TreeNode()
            cur = cur.children[bit]

n, m = map(int, input().split())
nums = list(map(int, input().split()))
root = TreeNode()
preprocess(nums, root)
ans = 0
for num in nums:
    ans += solve(num, 16, root)
print(ans // 2)
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

3. 字典序

题目描述

```
给定整数 n 和 m，将 1 到 n 的这 n 个整数按字典序排列之后，求其中的第
对于 n=11，m=4，按字典序排列依次为 1，10，11，2，3，4，5，6，7
对于 n=200，m=25，按字典序排列依次为 1 10 100 101 102 103 104

输入描述：
输入仅包含两个整数 n 和 m。

数据范围：

对于 20%的数据，1 <= m <= n <= 5 ；

对于 80%的数据，1 <= m <= n <= 10^7 ；

对于 100%的数据，1 <= m <= n <= 10^18.

输出描述：
输出仅包括一行，即所求排列中的第 m 个数字.
示例 1
输入
11 4
输出
2
```

前置知识

- 十叉树
- 完全十叉树
- 计算完全十叉树的节点个数
- 字典树

思路

和上面题目思路一样， 先从暴力解法开始，尝试打开思路。

暴力兜底的思路是直接生成一个长度为 n 的数组， 排序，选第 m 个即可。代码：

```
n, m = map(int, input().split())

nums  = [str(i) for i in range(1, n + 1)]
print(sorted(nums)[m - 1])
```

复杂度分析

- 时间复杂度：取决于排序算法， 不妨认为是 $O(NlogN)$
- 空间复杂度: $O(N)$

这种算法可以 pass 50 % case。

上面算法低效的原因是开辟了 N 的空间，并对整 N 个 元素进行了排序。

一种简单的优化方法是将排序换成堆，利用堆的特性求第 k 大的数， 这样时间复杂度可以减低到 $mlogN$。

我们继续优化。实际上，你如果把字典序的排序结构画出来， 可以发现他本质就是一个十叉树，并且是一个完全十叉树。

接下来，我带你继续分析。

如图， 红色表示根节点。节点表示一个十进制数， **树的路径存储真正的数字**，比如图上的 100，109 等。 这不就是上面讲的前缀树么?

如图黄色部分， 表示字典序的顺序，注意箭头的方向。因此本质上，**求字典序第 m 个数， 就是求这棵树的前序遍历的第 m 个节点。**

因此一种优化思路就是构建一颗这样的树，然后去遍历。 构建的复杂度是 $O(N)$，遍历的复杂度是 $O(M)$。因此这种算法的复杂度可以达到 $O(max(m,n))$ ，由于 n >= m，因此就是 $O(N)$。

实际上， 这样的优化算法依然是无法 AC 全部测试用例的，会超内存限制。 因此我们的思路只能是不使用 N 的空间去构造树。想想也知道， 由于 N 最大可能为 10^18，一个数按照 4 字节来算， 那么这就有 400000000 字节，大约是 381 M，这是不能接受的。

上面提到这道题就是一个完全十叉树的前序遍历，问题转化为求完全十叉树的前序遍历的第 m 个数。

> 十叉树和二叉树没有本质不同， 我在二叉树专题部分， 也提到了 N 叉树都可以用二叉树来表示。

对于一个节点来说，第 m 个节点：

- 要么就是它本身
- 要么其孩子节点中
- 要么在其兄弟节点
- 要么在兄弟节点的孩子节点中

究竟在上面的四个部分的哪，取决于其孩子节点的个数。

- count > m ， m 在其孩子节点中，我们需要深入到子节点。
- count <= m ， m 不在自身和孩子节点, 我们应该跳过所有孩子节点，直接到兄弟节点。

这本质就是一个递归的过程。

需要注意的是，我们并不会真正的在树上走，因此上面提到的**深入到子节点**， 以及 **跳过所有孩子节点，直接到兄弟节点**如何操作呢?

你仔细观察会发现： 如果当前节点的前缀是 x ，那么其第一个子节点（就是最小的子节点）是 x * 10，第二个就是 x * 10 + 1，以此类推。因此：

- 深入到子节点就是 x * 10。
- 跳过所有孩子节点，直接到兄弟节点就是 x + 1。

ok，铺垫地差不多了。

接下来，我们的重点是**如何计算给定节点的孩子节点的个数**。

这个过程和完全二叉树计算节点个数并无二致，这个算法的时间复杂度应该是 $O(logN * logN)$。 如果不会的同学，可以参考力扣原题： 222. 完全二叉树的节点个数 ，这是一个难度为中等的题目。

> 因此这道题本身被划分为 hard，一点都不为过。

这里简单说下，计算给定节点的孩子节点的个数的思路， 我的 91 天学算法里出过这道题。

一种简单但非最优的思路是分别计算左右子树的深度。

- 如果当前节点的左右子树高度相同，那么左子树是一个满二叉树，右子树是一个完全二叉树。
- 否则（左边的高度大于右边），那么左子树是一个完全二叉树，右子树是一个满二叉树。

如果是满二叉树，当前节点数 是 2 ^ depth，而对于完全二叉树，我们继续递归即可。

```python
class Solution:
    def countNodes(self, root):
        if not root:
            return 0
        ld = self.getDepth(root.left)
        rd = self.getDepth(root.right)
        if ld == rd:
            return 2 ** ld + self.countNodes(root.right)
        else:
            return 2 ** rd + self.countNodes(root.left)

    def getDepth(self, root):
        if not root:
            return 0
        return 1 + self.getDepth(root.left)
```

复杂度分析

- 时间复杂度：$O(logN * logN)$
- 空间复杂度：$O(logN)$

而这道题， 我们可以更简单和高效。

比如我们要计算 1 号节点的子节点个数。

- 它的孩子节点个数是 。。。
- 它的孙子节点个数是 。。。
- 。。。

全部加起来即可。

它的孩子节点个数是 `20 - 10 = 10` 。 也就是它的**右边的兄弟节点的第一个子节点** 减去 它的**第一个子节点**。

由于是完全十叉树，而不是满十叉树 。因此你需要考虑边界情况，比如题目的 n 是 15。 那么 1 的子节点个数就不是 20 - 10 = 10 了， 而是 15 - 10 + 1 = 16。

其他也是类似的过程， 我们只要：

- Go deeper and do the same thing

或者：

- Move to next neighbor and do the same thing

不断重复，直到 m 降低到 0 。

代码

```
def count(c1, c2, n):
    steps = 0
    while c1 <= n:
        steps += min(n + 1, c2) - c1
        c1 *= 10
        c2 *= 10
    return steps
def findKthNumber(n: int, k: int) -> int:
    cur = 1
    k = k - 1
    while k > 0:
        steps = count(cur, cur + 1, n)
        if steps <= k:
            cur += 1
            k -= steps
        else:
            cur *= 10
            k -= 1
    return cur
n, m = map(int, input().split())
print(findKthNumber(n, m))
```

复杂度分析

- 时间复杂度：$O(logM * logN)$
- 空间复杂度：$O(1)$

总结

其中三道算法题从难度上来说，基本都是困难难度。从内容来看，基本都是力扣的换皮题，且都或多或少和树有关。如果大家一开始没有思路，建议大家先给出暴力的解法兜底，再画图或举简单例子打开思路。

我也刷了很多字节的题了，还有一些难度比较大的题。如果你第一次做，那么需要你思考比较久才能想出来。加上面试紧张，很可能做不出来。这个时候就更需要你冷静分析，先暴力打底，慢慢优化。有时候即使给不了最优解，让面试官看出你的思路也很重要。 比如小兔的棋盘 想出最优解难度就不低，不过你可以先暴力 DFS 解决，再 DP 优化会慢慢帮你打开思路。有时候面试官也会引导你，给你提示， 加上你刚才“发挥不错”，说不定一下子就做出最优解了，这个我深有体会。

另外要提醒大家的是， 刷题要适量，不要贪多。要完全理清一道题的来龙去脉。多问几个为什么。 这道题暴力法怎么做？暴力法哪有问题？怎么优化？为什么选了这个算法就可以优化？为什么这种算法要用这种数据结构来实现？

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 36K+ star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

《我是你的妈妈呀》 - 第一期

记得我初中的时候，学校发的一个小册子的名字就是母题啥的。

大概意思是市面上的题（尤其是中考题）都是这些母题生的，都是它们的儿子。

熟悉我的朋友应该知道，我有一个风格：”喜欢用通俗易懂的语言以及图片，还原解题过程“。包括我是如何抽象的，如何与其他题目建立联系的等。比如：

- 一招吃遍力扣四道题，妈妈再也不用担心我被套路啦～
- 超级详细记忆化递归，图解，带你一次攻克三道 Hard 套路题（44. 通配符匹配）
- 穿上衣服我就不认识你了？来聊聊最长上升子序列
- 扒一扒这种题的外套（343. 整数拆分）

如果把这个思考过程称之为自顶向下的话，那么实际上能写出来取决于你：

- 是否有良好的抽象能力
- 是否有足够的基础知识
- 是否能与学过的基础知识建立联系

如果反着呢？我先把所有抽象之后的纯粹的东西掌握，也就是母题。那么遇到新的题，我就往上套呗？这就是我在《LeetCode 题解仓库》中所说的**只有熟练掌握基础的数据结构与算法，才能对复杂问题迎刃有余。** 这种思路就是**自底向上**。（有点像动态规划?） 市面上的题那么多，但是题目类型就是那几种。甚至出题人出题的时候都是根据以前的题目变个条件，变个说法从而搞出一个“新”的题。

这个专题的目标就是从反的方向来，我们先学习和记忆底层的被抽象过的经典的题目。遇到新的题目，就往这些母题上套即可。

那让我们来自底向上看下第一期的这八道母题吧~

母题 1

题目描述

给你两个有序的非空数组 nums1 和 nums2，让你从每个数组中分别挑一个，使得二者差的绝对值最小。

思路

- 初始化 ans 为无限大
- 使用两个指针，一个指针指向数组 1，一个指针指向数组 2
- 比较两个指针指向的数字的大小，并更新较小的那个的指针，使其向后移动一位。更新的过程顺便计算 ans
- 最后返回 ans

代码

```
def f(nums1, nums2):
    i = j = 0
    ans = float('inf')
    while i < len(nums1) and j < len(nums2):
        ans = min(ans, abs(nums1[i] - nums2[j]))
        if  nums1[i] < nums2[j]:
            i += 1
        else:
            j += 1
    return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

母题 2

题目描述

给你两个非空数组 nums1 和 nums2，让你从每个数组中分别挑一个，使得二者差的绝对值最小。

思路

数组没有说明是有序的，可以选择暴力。两两计算绝对值，返回最小的即可。

代码：

```
def f(nums1, nums2):
    ans = float('inf')
    for num1 in nums1:
        for num2 in nums2:
            ans = min(ans, abs(num1 - num2))
    return ans
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(1)$

由于暴力的时间复杂度是 $O(N^2)$，因此其实也可以先排序将问题转换为母题 1，然后用母题 1 的解法求解。

复杂度分析

- 时间复杂度：$O(NlogN)$
- 空间复杂度：$O(1)$

母题 3

题目描述

给你 k 个有序的非空数组，让你从每个数组中分别挑一个，使得二者差的绝对值最小。

思路

继续使用母题 1 的思路，使用 k 个 指针即可。

复杂度分析

- 时间复杂度：$O(klogM)$，其中 M 为 k 个非空数组的长度的最小值。
- 空间复杂度：$O(1)$

我们也可以使用堆来处理，代码更简单，逻辑更清晰。这里我们使用小顶堆，作用就是选出最小值。

代码

```
def f(matrix):
    ans = float('inf')
    max_value = max(nums[0] for nums in matrix)
    heap = [(nums[0], i, 0) for i, nums in enumerate(nums)]
    heapq.heapify(heap)

    while True:
        min_value, row, idx = heapq.heappop(heap)
        if max_value - min_value < ans:
            ans = max_value - min_value
        if idx == len(matrix[row]) - 1:
            break
        max_value = max(max_value, matrix[row][idx + 1])
        heapq.heappush(heap, (matrix[row][idx + 1], row, id

    return ans
```

复杂度分析

建堆的时间和空间复杂度为 $O(k)$。

while 循环会执行 M 次，其中 M 为 k 个非空数组的长度的最小值。heappop 和 heappush 的时间复杂度都是 logk。因此 while 循环总的时间复杂度为 $O(Mlogk)$。

- 时间复杂度：$O(max(Mlogk, k))$，其中 M 为 k 个非空数组的长度的最小值。
- 空间复杂度：$O(k)$

母题 4

题目描述

给你 k 个非空数组，让你从每个数组中分别挑一个，使得二者差的绝对值最小。

思路

先排序，然后转换为母题 3

母题 5

题目描述

给你两个有序的非空数组 nums1 和 nums2，让你将两个数组合并，使得新的数组有序。

LeetCode 地址： https://leetcode-cn.com/problems/merge-sorted-array/

思路

和母题 1 类似。

代码

```
def f(nums1, nums2):
    i = j = 0
    ans = []
    while i < len(nums1) and j < len(nums2):
        if nums1[i] < nums2[j]:
            ans.append(nums1[i])
            i += 1
        else:
            ans.append(nums2[j])
            j += 1
    if nums1:
        ans += nums2[j:]
    else:
        ans += nums1[i:]
    return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

母题 6

题目描述

给你 k 个有序的非空数组 nums1 和 nums2，让你将 k 个数组合并，使得新的数组有序。

思路

和母题 5 类似。 只不过不是两个，而是多个。我们继续套用堆的思路。

代码

```
import heapq


def f(matrix):
    ans = []
    heap = []
    for row in matrix:
        heap += row
    heapq.heapify(heap)

    while heap:
        cur = heapq.heappop(heap)
        ans.append(cur)

    return ans
```

复杂度分析

建堆的时间和空间复杂度为 $O(N)$。

heappop 的时间复杂度为 $O(logN)$。

- 时间复杂度：$O(NlogN)$，其中 N 是矩阵中的数字总数。
- 空间复杂度：$O(N)$，其中 N 是矩阵中的数字总数。

母题 7

题目描述

给你两个有序的链表 root1 和 root2，让你将两个链表合并，使得新的链表有序。

LeetCode 地址：https://leetcode-cn.com/problems/merge-two-sorted-lists/

思路

和母题 5 类似。 不同的地方在于数据结构从数组变成了链表，我们只需要注意链表的操作即可。

这里我使用了迭代和递归两种方式。

> 大家可以把母题 5 使用递归写一下。

代码

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) ->
        if not l1: return l2
        if not l2: return l1
        if l1.val < l2.val:
            l1.next = self.mergeTwoLists(l1.next, l2)
            return l1
        else:
            l2.next = self.mergeTwoLists(l1, l2.next)
            return l2
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为两个链表中较短的那个的长度。
- 空间复杂度：$O(N)$，其中 N 为两个链表中较短的那个的长度。

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) ->
        if not l1: return l2
        if not l2: return l1
        ans = cur = ListNode(0)
        while l1 and l2:
            if l1.val < l2.val:
                cur.next = l1
                cur = cur.next
                l1 = l1.next
            else:
                cur.next = l2
                cur = cur.next
                l2 = l2.next


        if l1:
            cur.next = l1
        else:
            cur.next = l2
        return ans.next
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为两个链表中较短的那个的长度。
- 空间复杂度：$O(1)$

母题 8

题目描述

给你 k 个有序的链表，让你将 k 个链表合并，使得新的链表有序。

LeetCode 地址：https://leetcode-cn.com/problems/merge-k-sorted-lists/

思路

和母题 7 类似，我们使用递归可以轻松解决。其实本质上就是

代码

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) ->
        if not l1: return l2
        if not l2: return l1
        if l1.val < l2.val:
            l1.next = self.mergeTwoLists(l1.next, l2)
            return l1
        else:
            l2.next = self.mergeTwoLists(l1, l2.next)
            return l2
    def mergeKLists(self, lists: List[ListNode]) -> ListNo
        if not lists: return None
        if len(lists) == 1: return lists[0]
        return self.mergeTwoLists(lists[0], self.mergeKLis
```

复杂度分析

mergeKLists 执行了 k 次，每次都执行一次 mergeTwoLists，mergeTwoLists 的时间复杂度前面已经分析过了，为 $O(N)$，其中 N 为两个链表中较短的那个的长度。

- 时间复杂度：$O(k*N)$，其中 N 为两个链表中较短的那个的长度
- 空间复杂度：$O(max(k,N))$

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) ->
        if not l1: return l2
        if not l2: return l1
        if l1.val < l2.val:
            l1.next = self.mergeTwoLists(l1.next, l2)
            return l1
        else:
            l2.next = self.mergeTwoLists(l1, l2.next)
            return l2
    def mergeKLists(self, lists: List[ListNode]) -> ListNo
        if not lists: return None
        if len(lists) == 1: return lists[0]
        return self.mergeTwoLists(self.mergeKLists(lists[:
```

复杂度分析

mergeKLists 执行了 logk 次，每次都执行一次 mergeTwoLists，mergeTwoLists 的时间复杂度前面已经分析过了，为 $O(N)$，其中 N 为两个链表中较短的那个的长度。

- 时间复杂度：$O(Nlogk)$，其中 N 为两个链表中较短的那个的长度
- 空间复杂度：$O(max(logk, N))$，其中 N 为两个链表中较短的那个的长度

全家福

最后送大家一张全家福：

子题

实际子题数量有很多，这里提供几个供大家练习。一定要练习，不能眼高手低。多看我的题解，多练习，多总结，你也可以的。

- 面试题 17.14. 最小 K 个数
- 1200. 最小绝对差
- 632. 最小区间
- 两数和，三数和，四数和。。。 k 数和

总结

母题就是**抽象之后的纯粹的东西**。如果你掌握了母题，即使没有掌握抽象的能力，依然有可能套出来。但是随着题目做的变多，“抽象能力”也会越来越强。因为你知道这些题背后是怎么产生的。

本期给大家介绍了八道母题， 大家可以在之后的刷题过程中尝试使用母题来套模板。之后会给大家带来更多的母题。

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 35K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

文带你看懂二叉树的序列化

我们先来看下什么是序列化，以下定义来自维基百科：

> 序列化（serialization）在计算机科学的数据处理中，是指将数据结构或对象状态转换成可取用格式（例如存成文件，存于缓冲，或经由网络中发送），以留待后续在相同或另一台计算机环境中，能恢复原先状态的过程。依照序列化格式重新获取字节的结果时，可以利用它来产生与原始对象相同语义的副本。对于许多对象，像是使用大量引用的复杂对象，这种序列化重建的过程并不容易。面向对象中的对象序列化，并不概括之前原始对象所关系的函数。这种过程也称为对象编组（marshalling）。从一系列字节提取数据结构的反向操作，是反序列化（也称为解编组、deserialization、unmarshalling）。

可见，序列化和反序列化在计算机科学中的应用还是非常广泛的。就拿 LeetCode 平台来说，其允许用户输入形如：

```
[1,2,3,null,null,4,5]
```

这样的数据结构来描述一颗树：

([1,2,3,null,null,4,5] 对应的二叉树)

其实序列化和反序列化只是一个概念，不是一种具体的算法，而是很多的算法。并且针对不同的数据结构，算法也会不一样。本文主要讲述的是二叉树的序列化和反序列化。看完本文之后，你就可以放心大胆地去 AC 以下两道题：

- 449. 序列化和反序列化二叉搜索树(中等)
- 297. 二叉树的序列化与反序列化(困难)

前置知识

阅读本文之前，需要你对树的遍历以及 BFS 和 DFS 比较熟悉。如果你还不熟悉，推荐阅读一下相关文章之后再来看。或者我这边也写了一个总结性的文章二叉树的遍历，你也可以看看。

前言

我们知道：二叉树的深度优先遍历，根据访问根节点的顺序不同，可以将其分为 `前序遍历` ， `中序遍历` ， `后序遍历` 。即如果先访问根节点就是前序遍历，最后访问根节点就是后续遍历，其它则是中序遍历。而左右节点的相对顺序是不会变的，一定是先左后右。

> 当然也可以设定为先右后左。

并且知道了三种遍历结果中的任意两种即可还原出原有的树结构。这不就是序列化和反序列化么？如果对这个比较陌生的同学建议看看我之前写的《构造二叉树系列》

有了这样一个前提之后算法就自然而然了。即先对二叉树进行两次不同的遍历，不妨假设按照前序和中序进行两次遍历。然后将两次遍历结果序列化，比如将两次遍历结果以逗号“,” join 成一个字符串。 之后将字符串反序列即可，比如将其以逗号“,” split 成一个数组。

序列化：

```
class Solution:
    def preorder(self, root: TreeNode):
        if not root: return []
        return [str(root.val)] +self. preorder(root.left) -
    def inorder(self, root: TreeNode):
        if not root: return []
        return  self.inorder(root.left) + [str(root.val)] -
    def serialize(self, root):
        ans = ''
        ans += ','.join(self.preorder(root))
        ans += '{% math %}'
        ans += ','.join(self.inorder(root))

        return ans
```

反序列化：

这里我直接用了力扣 `105. 从前序与中序遍历序列构造二叉树` 的解法，一行代码都不改。

```
class Solution:
    def deserialize(self, data: str):
        preorder, inorder = data.split('{% endmath %}')
        if not preorder: return None
        return self.buildTree(preorder.split(','), inorder.

    def buildTree(self, preorder: List[int], inorder: List
        # 实际上inorder 和 preorder 一定是同时为空的，因此你无论
        if not preorder:
            return None
        root = TreeNode(preorder[0])

        i = inorder.index(root.val)
        root.left = self.buildTree(preorder[1:i + 1], inorc
        root.right = self.buildTree(preorder[i + 1:], inorc

        return root
```

实际上这个算法是不一定成立的，原因在于树的节点可能存在重复元素。也就是说我前面说的 `知道了三种遍历结果中的任意两种即可还原出原有的树结构` 是不对的，严格来说应该是**如果树中不存在重复的元素，那么知道了三种遍历结果中的任意两种即可还原出原有的树结构**。

聪明的你应该发现了，上面我的代码用了 `i = inorder.index(root.val)` ，如果存在重复元素，那么得到的索引 i 就可能不是准确的。但是，如果题目限定了没有重复元素则可以用这种算

法。但是现实中不出现重复元素不太现实，因此需要考虑其他方法。那究竟是什么样的方法呢? 接下来进入正题。

DFS

序列化

我们来模仿一下力扣的记法。 比如： `[1,2,3,null,null,4,5]` (本质上是 BFS 层次遍历)，对应的树如下：

> 选择这种记法，而不是 DFS 的记法的原因是看起来比较直观

序列化的代码非常简单， 我们只需要在普通的遍历基础上，增加对空节点的输出即可（普通的遍历是不处理空节点的）。

比如我们都树进行一次前序遍历的同时增加空节点的处理。选择前序遍历的原因是容易知道根节点的位置，并且代码好写，不信你可以试试。

因此序列化就仅仅是普通的 DFS 而已，直接给大家看看代码。

Python 代码：

```
class Codec:
    def serialize_dfs(self, root, ans):
        # 空节点也需要序列化，否则无法唯一确定一棵树，后不赘述。
        if not root: return ans + '#,'
        # 节点之间通过逗号（,）分割
        ans += str(root.val) + ','
        ans = self.serialize_dfs(root.left, ans)
        ans = self.serialize_dfs(root.right, ans)
        return ans
    def serialize(self, root):
        # 由于最后会添加一个额外的逗号，因此需要去除最后一个字符，后
        return self.serialize_dfs(root, '')[:-1]
```

Java 代码：

```
public class Codec {
    public String serialize_dfs(TreeNode root, String str)
        if (root == null) {
            str += "None,";
        } else {
            str += str.valueOf(root.val) + ",";
            str = serialize_dfs(root.left, str);
            str = serialize_dfs(root.right, str);
        }
        return str;
    }

    public String serialize(TreeNode root) {
        return serialize_dfs(root, "");
    }
}
```

`[1,2,3,null,null,4,5]` 会被处理为 `1,2,#,#,3,4,#,#,5,#,#`

我们先看一个短视频：

（动画来自力扣）

反序列化

反序列化的第一步就是将其展开。以上面的例子来说，则会变成数组：`[1,2,#,#,3,4,#,#,5,#,#]`，然后我们同样执行一次前序遍历，每次处理一个元素，重建即可。由于我们采用的前序遍历，因此第一个是根元素，下一个是其左子节点，下下一个是其右子节点。

Python 代码：

```
    def deserialize_dfs(self, nodes):
        if nodes:
            if nodes[0] == '#':
                nodes.pop(0)
                return None
            root = TreeNode(nodes.pop(0))
            root.left = self.deserialize_dfs(nodes)
            root.right = self.deserialize_dfs(nodes)
            return root
        return None

    def deserialize(self, data: str):
        nodes = data.split(',')
        return self.deserialize_dfs(nodes)
```

Java 代码：

```
    public TreeNode deserialize_dfs(List<String> l) {
        if (l.get(0).equals("None")) {
            l.remove(0);
            return null;
        }

        TreeNode root = new TreeNode(Integer.valueOf(l.get
        l.remove(0);
        root.left = deserialize_dfs(l);
        root.right = deserialize_dfs(l);

        return root;
    }

    public TreeNode deserialize(String data) {
        String[] data_array = data.split(",");
        List<String> data_list = new LinkedList<String>(Ar
        return deserialize_dfs(data_list);
    }
```

复杂度分析

- 时间复杂度：每个节点都会被处理一次，因此时间复杂度为 $O(N)$，其中 N 为节点的总数。
- 空间复杂度：空间复杂度取决于栈深度，因此空间复杂度为 $O(h)$，其中 h 为树的深度。

BFS

序列化

实际上我们也可以使用 BFS 的方式来表示一棵树。在这一点上其实就和力扣的记法是一致的了。

我们知道层次遍历的时候实际上是有层次的。只不过有的题目需要你记录每一个节点的层次信息，有些则不需要。

这其实就是一个朴实无华的 BFS，唯一不同则是增加了空节点。

Python 代码：

```
class Codec:
    def serialize(self, root):
        ans = ''
        queue = [root]
        while queue:
            node = queue.pop(0)
            if node:
                ans += str(node.val) + ','
                queue.append(node.left)
                queue.append(node.right)
            else:
                ans += '#,'
        return ans[:-1]
```

反序列化

如图有这样一棵树：

那么其层次遍历为 [1,2,3,#,#, 4, 5]。我们根据此层次遍历的结果来看下如何还原二叉树，如下是我画的一个示意图：

容易看出：

- level x 的节点一定指向 level x + 1 的节点，如何找到 level + 1 呢？这很容易通过层次遍历来做到。
- 对于给的的 level x，从左到右依次对应 level x + 1 的节点，即第 1 个节点的左右子节点对应下一层的第 1 个和第 2 个节点，第 2 个节点的左右子节点对应下一层的第 3 个和第 4 个节点。。。
- 接上，其实如果你仔细观察的话，实际上 level x 和 level x + 1 的判断是无需特别判断的。我们可以把思路逆转过来：`即第 1 个节点的左右子节点对应第 1 个和第 2 个节点，第 2 个节点的左右子节点对应第 3 个和第 4 个节点。。。`（注意，没了下一层三个字）

因此我们的思路也是同样的 BFS，并依次连接左右节点。

Python 代码：

```
    def deserialize(self, data: str):
        if data == '#': return None
        # 数据准备
        nodes = data.split(',')
        if not nodes: return None
        # BFS
        root = TreeNode(nodes[0])
        queue = [root]
        # 已经有 root 了，因此从 1 开始
        i = 1

        while i < len(nodes) - 1:
            node = queue.pop(0)
            #
            lv = nodes[i]
            rv = nodes[i + 1]
            i += 2
            # 对于给的的 level x，从左到右依次对应 level x + 1
            # node 是 level x 的节点，l 和 r 则是 level x + 1
            if lv != '#':
                l = TreeNode(lv)
                node.left = l
                queue.append(l)

            if rv != '#':
                r = TreeNode(rv)
                node.right = r
                queue.append(r)
        return root
```

复杂度分析

- 时间复杂度：每个节点都会被处理一次，因此时间复杂度为 $O(N)$，其中 N 为节点的总数。
- 空间复杂度：$O(N)$，其中 N 为节点的总数。

总结

除了这种方法还有很多方案， 比如括号表示法。 关于这个可以参考力扣 606. 根据二叉树创建字符串，这里就不再赘述了。

本文从 BFS 和 DFS 角度来思考如何序列化和反序列化一棵树。 如果用 BFS 来序列化，那么相应地也需要 BFS 来反序列化。如果用 DFS 来序列化，那么就需要用 DFS 来反序列化。

我们从马后炮的角度来说，实际上对于序列化来说，BFS 和 DFS 都比较常规。对于反序列化，大家可以像我这样举个例子，画一个图。可以先在纸上，电脑上，如果你熟悉了之后，也可以画在脑子里。

（Like This）

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

穿上衣服我就不认识你了？来聊聊最长上升子序列

最长上升子序列是一个很经典的算法题。有的会直接让你求最长上升子序列，有的则会换个说法，但最终考察的还是最长上升子序列。那么问题来了，它穿上衣服你还看得出来是么？

如果你完全看不出来了，说明抽象思维还不到火候。经常看我的题解的同学应该会知道，我经常强调 `抽象思维` 。没有抽象思维，所有的题目对你来说都是新题。你无法将之前做题的经验迁移到这道题，那你做的题意义何在？

虽然抽象思维很难练成，但是幸好算法套路是有限的，经常考察的题型更是有限的。从这些入手，或许可以让你轻松一些。本文就从一个经典到不行的题型《最长上升子序列》，来帮你进一步理解 `抽象思维` 。

> 注意。 本文是帮助你识别套路，从横向上理清解题的思维框架，并没有采用最优解，所有的题目给的解法都不是最优的，但是都可以通过所有的测试用例。如果你想看最优解，可以直接去讨论区看。或者期待我的 `深入剖析系列` 。

300. 最长上升子序列

题目地址

https://leetcode-cn.com/problems/longest-increasing-subsequence

题目描述

```
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。
说明:

可能会有多种最长上升子序列的组合，你只需要输出对应的长度即可。
你算法的时间复杂度应该为 O(n2) 。
进阶: 你能将算法的时间复杂度降低到 O(n log n) 吗?
```

思路

> 美团和华为都考了这个题。

题目的意思是让我们从给定数组中挑选若干数字，这些数字满足： `如果 i < j 则 nums[i] < nums[j]` 。问：一次可以挑选最多满足条件的数字是多少个。

最长上升子序列 @lucifer

这种子序列求极值的题目，应该要考虑到贪心或者动态规划。这道题贪心是不可以的，我们考虑动态规划。

按照动态规划定义状态的套路，我们有**两种常见**的定义状态的方式：

- dp[i] : 以 i 结尾（一定包括 i）所能形成的最长上升子序列长度, 答案是 max(dp[i])，其中 i = 0,1,2, ..., n - 1
- dp[i] : 以 i 结尾（可能包括 i）所能形成的最长上升子序列长度，答案是 dp[-1] （-1 表示最后一个元素）

容易看出第二种定义方式由于无需比较不同的 dp[i] 就可以获得答案，因此更加方便。但是想了下，状态转移方程会很不好写，因为 dp[i] 的末尾数字（最大的）可能是 任意 j < i 的位置。

第一种定义方式虽然需要比较不同的 dp[i] 从而获得结果，但是我们可以在循环的时候顺便得出，对复杂度不会有影响，只是代码多了一点而已。因此我们**选择第一种建模方式**。

dp[i] ： 以 i 结尾（一定包括 i）所能形成的最长上升子序列

ans 不一定在 dp[-1] 产生
而是 max(dp[i]), 其中 i = 0, 1, 2, n - 1

由于 dp[j] 中一定会包括 j，且以 j 结尾， 那么 nums[j] 一定是其所形成的序列中最大的元素，那么如果位于其后（意味着 i > j）的 nums[i] > nums[j]，那么 nums[i] 一定能够融入 dp[j] 从而形成更大的序列，这个序列的长度是 dp[j] + 1。因此状态转移方程就有了： `dp[i] = dp[j] + 1 (其中 i > j, nums[i] > nums[j])`

以 `[10, 9, 2, 5, 3, 7, 101, 18]` 为例，当我们计算到 dp[5]的时候，我们需要往回和 0，1，2，3，4 进行比较。

具体的比较内容是：

```
nums[5] > nums[0] no
nums[5] > nums[1] no
nums[5] > nums[2] yes
nums[5] > nums[3] yes
nums[5] > nums[4] yes
```

最后从三个中选一个最大的 + 1 赋给 dp[5]即可。

```
max(dp[2], dp[3], dp[4])+ 1
```

记住这个状态转移方程，后面我们还会频繁用到。

代码

```python
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        n = len(nums)
        if n == 0: return 0
        dp = [1] * n
        ans = 1
        for i in range(n):
            for j in range(i):
                if nums[i] > nums[j]:
                    dp[i] = max(dp[i], dp[j] + 1)
                    ans = max(ans, dp[i])
        return  ans
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N)$

435. 无重叠区间

题目地址

https://leetcode-cn.com/problems/non-overlapping-intervals/

题目描述

```
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:

可以认为区间的终点总是大于它的起点。
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。
```

思路

我们先来看下最终**剩下**的区间。由于剩下的区间都是不重叠的，因此剩下的**相邻区间的后一个区间的开始时间一定是不小于前一个区间的结束时间的**。 比如我们剩下的区间是 `[ [1,2], [2,3], [3,4] ]` 。就是第一个区间的 2 小于等于 第二个区间的 2，第二个区间的 3 小于等于第三个区间的 3。

不难发现如果我们将 `前面区间的结束` 和 `后面区间的开始` 结合起来看，其就是一个**非严格递增序列**。而我们的目标就是删除若干区间，从而**剩下最长的非严格递增子序列**。这不就是上面的题么？只不过上面是严格递增，这不重要，就是改个符号的事情。 上面的题你可以看成是删除了若干数字，然后剩下**剩下最长的严格递增子序列**。 **这就是抽象的力量，这就是套路。**

如果对区间按照起点或者终点进行排序，那么就转化为上面的最长递增子序列问题了。和上面问题不同的是，由于是一个区间。因此实际上，我们是需要拿**后面的开始时间**和**前面的结束时间**进行比较。

而由于：

- 题目求的是需要移除的区间，因此最后 return 的时候需要做一个转化。
- 题目不是要求严格递增，而是可以相等，因此我们的判断条件要加上等号。

> 这道题还有一种贪心的解法，其效率要比动态规划更好，但由于和本文的主题不一致，就不在这里讲了。

代码

你看代码多像

```
class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[in
        n = len(intervals)
        if n == 0: return 0
        dp = [1] * n
        ans = 1
        intervals.sort(key=lambda a: a[1])

        for i in range(len(intervals)):
            for j in range(i - 1, -1, -1):
                if intervals[i][0] >= intervals[j][1]:
                    dp[i] = max(dp[i], dp[j] + 1)
                    # 由于我事先进行了排序，因此倒着找的时候，找到
                    # 这也是为什么我按照结束时间排序的原因。
                    break
            dp[i] = max(dp[i], dp[i - 1])
            ans = max(ans, dp[i])

        return n - ans
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N)$

646. 最长数对链

题目地址

https://leetcode-cn.com/problems/maximum-length-of-pair-chain/

题目描述

```
给出 n 个数对。 在每一个数对中，第一个数字总是比第二个数字小。

现在，我们定义一种跟随关系，当且仅当 b < c 时，数对(c, d) 才可以跟在

给定一个对数集合，找出能够形成的最长数对链的长度。你不需要用到所有的数

示例 :

输入: [[1,2], [2,3], [3,4]]
输出: 2
解释: 最长的数对链是 [1,2] -> [3,4]
注意:

给出数对的个数在 [1, 1000] 范围内。
```

思路

和上面的 `435. 无重叠区间` 是换皮题，唯一的区别这里又变成了严格增加。没关系，我们把等号去掉就行了。并且由于这道题求解的是最长的长度，因此转化也不需要了。

> 当然，这道题也有一种贪心的解法，其效率要比动态规划更好，但由于和本文的主题不一致，就不在这里讲了。

代码

这代码更像了！

```
class Solution:
    def findLongestChain(self, pairs: List[List[int]]) -> 
        n = len(pairs)
        dp = [1] * n
        ans = 1
        pairs.sort(key=lambda a: a[0])
        for i in range(n):
            for j in range(i):
                if pairs[i][0] > pairs[j][1]:
                    dp[i] = max(dp[i], dp[j] + 1)
                    ans = max(ans, dp[i])
        return ans
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N)$

452. 用最少数量的箭引爆气球

题目地址

https://leetcode-cn.com/problems/minimum-number-of-arrows-to-burst-balloons/

题目描述

```
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

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

Example:

输入:
[[10,16], [2,8], [1,6], [7,12]]

输出:
2

解释:
对于该样例，我们可以在x = 6（射爆[2,8],[1,6]两个气球）和 x = 11（
```

思路

把气球看成区间，几个箭可以全部射爆，意思就是有多少不重叠的区间。注意这里重叠的情况也可以射爆。这么一抽象，就和上面的 `646. 最长数对链` 一模一样了，不用我多说了吧？

> 当然，这道题也有一种贪心的解法，其效率要比动态规划更好，但由于和本文的主题不一致，就不在这里讲了。

代码

代码像不像？

```
class Solution:
    def findMinArrowShots(self, points: List[List[int]]) ->
        n = len(points)
        if n == 0: return 0
        dp = [1] * n
        cnt = 1
        points.sort(key=lambda a:a[1])

        for i in range(n):
            for j in range(0, i):
                if points[i][0] > points[j][1]:
                    dp[i] = max(dp[i], dp[j] + 1)
                    cnt = max(cnt, dp[i])
        return cnt
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N)$

More

其他的我就不一一说了。比如 673. 最长递增子序列的个数 （滴滴面试题）。 不就是求出最长序列，之后再循环比对一次就可以得出答案了么？

491. 递增子序列 由于需要找到所有的递增子序列，因此动态规划就不行了，妥妥回溯就行了，套一个模板就出来了。回溯的模板可以看我之前写的回溯专题。

大家把我讲的思路搞懂，这几个题一写，还怕碰到类似的题不会么？**只有熟练掌握基础的数据结构与算法，才能对复杂问题迎刃有余。** 最长上升子序列就是一个非常经典的基础算法，把它彻底搞懂，再去面对出题人的各种换皮就不怕了。相反，如果你不去思考题目背后的逻辑，就会刷地很痛苦。题目稍微一变化你就不会了，这也是为什么很多人说**刷了很多题，**

但是碰到新的题目还是不会做的原因之一。关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

欢迎长按关注

你的衣服我扒了 - 《最长公共子序列》

之前出了一篇穿上衣服我就不认识你了？来聊聊最长上升子序列，收到了大家的一致好评。今天给大家带来的依然是换皮题 - 最长公共子序列系列。

最长公共子序列是一个很经典的算法题。有的会直接让你求最长上升子序列，有的则会换个说法，但最终考察的还是最长公共子序列。那么问题来了，它穿上衣服你还看得出来是么？

如果你完全看不出来了，说明抽象思维还不到火候。经常看我的题解的同学应该会知道，我经常强调 `抽象思维` 。没有抽象思维，所有的题目对你来说都是新题。你无法将之前做题的经验迁移到这道题，那你做的题意义何在？

虽然抽象思维很难练成，但是幸好算法套路是有限的，经常考察的题型更是有限的。从这些入手，或许可以让你轻松一些。本文就从一个经典到不行的题型《最长公共子序列》，来帮你进一步理解 `抽象思维` 。

> 注意。 本文是帮助你识别套路，从横向上理清解题的思维框架，并没有采用最优解，所有的题目给的解法可能不是最优的，但是都可以通过所有的测试用例。如果你想看最优解，可以直接去讨论区看。或者期待我的 `深入剖析系列` 。

718. 最长重复子数组

题目地址

https://leetcode-cn.com/problems/maximum-length-of-repeated-subarray/

题目描述

```
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

示例 1:

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]。
说明:

1 <= len(A), len(B) <= 1000
0 <= A[i], B[i] < 100
```

前置知识

- 哈希表
- 数组
- 二分查找
- 动态规划

思路

这就是最经典的最长公共子序列问题。一般这种求解**两个数组或者字符串求最大或者最小**的题目都可以考虑动态规划，并且通常都定义 dp[i][j] 为 `以 A[i], B[j] 结尾的 xxx` 。这道题就是： `以 A[i], B[j] 结尾的两个数组中公共的、长度最长的子数组的长度` 。

> 关于状态转移方程的选择可以参考： 穿上衣服我就不认识你了？来聊聊最长上升子序列

算法很简单：

- 双层循环找出所有的 i, j 组合，时间复杂度 $O(m*n)$，其中 m 和 n 分别为 A 和 B 的 长度。
  - 如果 A[i] == B[j]，dp[i][j] = dp[i - 1][j - 1] + 1
  - 否则，dp[i][j] = 0
- 循环过程记录最大值即可。

记住这个状态转移方程，后面我们还会频繁用到。

关键点解析

- dp 建模套路

代码

代码支持：Python

Python Code:

```
class Solution:
    def findLength(self, A, B):
        m, n = len(A), len(B)
        ans = 0
        dp = [[0 for _ in range(n + 1)] for _ in range(m +
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if A[i - 1] == B[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                    ans = max(ans, dp[i][j])
        return ans
```

复杂度分析

- 时间复杂度：$O(m*n)$，其中 m 和 n 分别为 A 和 B 的 长度。
- 空间复杂度：$O(m*n)$，其中 m 和 n 分别为 A 和 B 的 长度。

> 二分查找也是可以的，不过并不容易想到，大家可以试试。

1143.最长公共子序列

题目地址

https://leetcode-cn.com/problems/longest-common-subsequence

题目描述

给定两个字符串 text1 和 text2，返回这两个字符串的最长公共子序列的长度。

一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。 例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。两个字符串的「公共子序列」是这两个字符串所共同拥有的子序列。

若这两个字符串没有公共子序列，则返回 0。

示例 1:

输入：text1 = "abcde", text2 = "ace" 输出：3
解释：最长公共子序列是 "ace"，它的长度为 3。 示例 2:

输入：text1 = "abc", text2 = "abc" 输出：3 解释：最长公共子序列是 "abc"，它的长度为 3。 示例 3:

输入：text1 = "abc", text2 = "def" 输出：0 解释：两个字符串没有公共子序列，返回 0。

提示:

1 <= text1.length <= 1000 1 <= text2.length <= 1000 输入的字符串只含有小写英文字符。

前置知识

- 数组
- 动态规划

思路

和上面的题目类似，只不过数组变成了字符串（这个无所谓），子数组（连续）变成了子序列 （非连续）。

算法只需要一点小的微调： `如果 A[i] != B[j]，那么 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])`

关键点解析

- dp 建模套路

代码

> 你看代码多像

代码支持：Python

Python Code:

```
class Solution:
    def longestCommonSubsequence(self, A: str, B: str) -> 
        m, n = len(A), len(B)
        ans = 0
        dp = [[0 for _ in range(n + 1)] for _ in range(m +
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if A[i - 1] == B[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                    ans = max(ans, dp[i][j])
                else:
                    dp[i][j] = max(dp[i - 1][j], dp[i][j -
        return ans
```

复杂度分析

- 时间复杂度：$O(m*n)$，其中 m 和 n 分别为 A 和 B 的 长度。
- 空间复杂度：$O(m*n)$，其中 m 和 n 分别为 A 和 B 的 长度。

1035. 不相交的线

题目地址

https://leetcode-cn.com/problems/uncrossed-lines/description/

题目描述

我们在两条独立的水平线上按给定的顺序写下 A 和 B 中的整数。

现在，我们可以绘制一些连接两个数字 A[i] 和 B[j] 的直线，只要 A[i] == B[j]，且我们绘制的直线不与任何其他连线（非水平线）相交。

以这种方法绘制线条，并返回我们可以绘制的最大连线数。

示例 1：

输入：A = [1,4,2], B = [1,2,4] 输出：2 解释： 我们可以画出两条不交叉的线，如上图所示。 我们无法画出第三条不相交的直线，因为从 A[1]=4 到 B[2]=4 的直线将与从 A[2]=2 到 B[1]=2 的直线相交。 示例 2：

输入：A = [2,5,1,2,5], B = [10,5,2,1,5,2] 输出：3 示例 3：

输入：A = [1,3,7,1,7,5], B = [1,9,2,5,1] 输出：2

提示：

1 <= A.length <= 500 1 <= B.length <= 500 1 <= A[i], B[i] <= 2000

前置知识

- 数组
- 动态规划

思路

从图中可以看出，如果想要不相交，则必然相对位置要一致，换句话说就是：**公共子序列**。因此和上面的 `1143.最长公共子序列` 一样，属于换皮题，代码也是一模一样。

关键点解析

- dp 建模套路

代码

> 你看代码多像

代码支持：Python

Python Code:

```
class Solution:
    def longestCommonSubsequence(self, A: str, B: str) -> 
        m, n = len(A), len(B)
        ans = 0
        dp = [[0 for _ in range(n + 1)] for _ in range(m +
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if A[i - 1] == B[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                    ans = max(ans, dp[i][j])
                else:
                    dp[i][j] = max(dp[i - 1][j], dp[i][j -
        return ans
```

复杂度分析

- 时间复杂度：$O(m*n)$，其中 m 和 n 分别为 A 和 B 的 长度。
- 空间复杂度：$O(m*n)$，其中 m 和 n 分别为 A 和 B 的 长度。

总结

第一道是“子串”题型，第二和第三则是“子序列”。不管是“子串”还是“子序列”，状态定义都是一样的，不同的只是一点小细节。

只有熟练掌握基础的数据结构与算法，才能对复杂问题迎刃有余。 基础算法，把它彻底搞懂，再去面对出题人的各种换皮就不怕了。相反，如果你不去思考题目背后的逻辑，就会刷地很痛苦。题目稍微一变化你就不会了，这也是为什么很多人说**刷了很多题，但是碰到新的题目还是不会做**的原因之一。关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

欢迎长按关注

一文看懂《最大子序列和问题》

最大子序列和是一道经典的算法题， leetcode 也有原题《53.maximum-sum-subarray》，今天我们就来彻底攻克它。

题目描述

求取数组中最大连续子序列和，例如给定数组为 A = [1， 3， -2， 4， -5]， 则最大连续子序列和为 6，即 1 + 3 +（-2）+ 4 = 6。 去

首先我们来明确一下题意。

- 题目说的子数组是连续的
- 题目只需要求和，不需要返回子数组的具体位置。
- 数组中的元素是整数，但是可能是正数，负数和 0。
- 子序列的最小长度为 1。

比如：

- 对于数组 [1, -2, 3, 5, -3, 2], 应该返回 3 + 5 = 8
- 对于数组 [0, -2, 3, 5, -1, 2], 应该返回 3 + 5 + -1 + 2 = 9
- 对于数组 [-9, -2, -3, -5, -3], 应该返回 -2

解法一 - 暴力法（超时法）

一般情况下，先从暴力解分析，然后再进行一步步的优化。

思路

我们来试下最直接的方法，就是计算所有的子序列的和，然后取出最大值。 记 Sum[i,....,j]为数组 A 中第 i 个元素到第 j 个元素的和，其中 0 <= i <= j < n， 遍历所有可能的 Sum[i,....,j] 即可。

我们去枚举以 0,1,2...n-1 开头的所有子序列即可， 对于每一个开头的子序列，我们都去枚举从当前开始到 n-1 的所有情况。

这种做法的时间复杂度为 O(N^2), 空间复杂度为 O(1)。

代码

JavaScript:

```
function LSS(list) {
  const len = list.length;
  let max = -Number.MAX_VALUE;
  let sum = 0;
  for (let i = 0; i < len; i++) {
    sum = 0;
    for (let j = i; j < len; j++) {
      sum += list[j];
      if (sum > max) {
        max = sum;
      }
    }
  }

  return max;
}
```

Java：

```
class MaximumSubarrayPrefixSum {
  public int maxSubArray(int[] nums) {
      int len = nums.length;
      int maxSum = Integer.MIN_VALUE;
      int sum = 0;
      for (int i = 0; i < len; i++) {
        sum = 0;
        for (int j = i; j < len; j++) {
          sum += nums[j];
          maxSum = Math.max(maxSum, sum);
        }
      }
      return maxSum;
  }
}
```

Python 3:

```
import sys
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        maxSum = -sys.maxsize
        sum = 0
        for i in range(n):
            sum = 0
            for j in range(i, n):
                sum += nums[j]
                maxSum = max(maxSum, sum)

        return maxSum
```

空间复杂度非常理想，但是时间复杂度有点高。怎么优化呢？我们来看下下一个解法。

解法二 - 分治法

思路

我们来分析一下这个问题， 我们先把数组平均分成左右两部分。

此时有三种情况：

- 最大子序列全部在数组左部分
- 最大子序列全部在数组右部分
- 最大子序列横跨左右数组

对于前两种情况，我们相当于将原问题转化为了规模更小的同样问题。

对于第三种情况，由于已知循环的起点（即中点），我们只需要进行一次循环，分别找出 左边和右边的最大子序列即可。

所以一个思路就是我们每次都对数组分成左右两部分，然后分别计算上面三种情况的最大子序列和， 取出最大的即可。

举例说明，如下图：

(by snowan)

这种做法的时间复杂度为 O(N*logN), 空间复杂度为 O(1)。

代码

JavaScript:

```
function helper(list, m, n) {
  if (m === n) return list[m];
  let sum = 0;
  let lmax = -Number.MAX_VALUE;
  let rmax = -Number.MAX_VALUE;
  const mid = ((n - m) >> 1) + m;
  const l = helper(list, m, mid);
  const r = helper(list, mid + 1, n);
  for (let i = mid; i >= m; i--) {
    sum += list[i];
    if (sum > lmax) lmax = sum;
  }

  sum = 0;

  for (let i = mid + 1; i <= n; i++) {
    sum += list[i];
    if (sum > rmax) rmax = sum;
  }

  return Math.max(l, r, lmax + rmax);
}

function LSS(list) {
  return helper(list, 0, list.length - 1);
}
```

Java:

```
class MaximumSubarrayDivideConquer {
  public int maxSubArrayDividConquer(int[] nums) {
      if (nums == null || nums.length == 0) return 0;
      return helper(nums, 0, nums.length - 1);
    }
    private int helper(int[] nums, int l, int r) {
      if (l > r) return Integer.MIN_VALUE;
      int mid = (l + r) >>> 1;
      int left = helper(nums, l, mid - 1);
      int right = helper(nums, mid + 1, r);
      int leftMaxSum = 0;
      int sum = 0;
      // left surfix maxSum start from index mid - 1 to l
      for (int i = mid - 1; i >= l; i--) {
        sum += nums[i];
        leftMaxSum = Math.max(leftMaxSum, sum);
      }
      int rightMaxSum = 0;
      sum = 0;
      // right prefix maxSum start from index mid + 1 to r
      for (int i = mid + 1; i <= r; i++) {
        sum += nums[i];
        rightMaxSum = Math.max(sum, rightMaxSum);
      }
      // max(left, right, crossSum)
      return Math.max(leftMaxSum + rightMaxSum + nums[mid],
    }
}
```

Python 3 :

```
import sys
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        return self.helper(nums, 0, len(nums) - 1)
    def helper(self, nums, l, r):
        if l > r:
            return -sys.maxsize
        mid = (l + r) // 2
        left = self.helper(nums, l, mid - 1)
        right = self.helper(nums, mid + 1, r)
        left_suffix_max_sum = right_prefix_max_sum = 0
        sum = 0
        for i in reversed(range(l, mid)):
            sum += nums[i]
            left_suffix_max_sum = max(left_suffix_max_sum,
        sum = 0
        for i in range(mid + 1, r + 1):
            sum += nums[i]
            right_prefix_max_sum = max(right_prefix_max_sur
        cross_max_sum = left_suffix_max_sum + right_prefix_
        return max(cross_max_sum, left, right)
```

解法三 - 动态规划

思路

我们来思考一下这个问题， 看能不能将其拆解为规模更小的同样问题，并且能找出 递推关系。

我们不妨假设问题 Q(list, i) 表示 list 中以索引 i 结尾的情况下最大子序列和， 那么原问题就转化为 Q(list, i), 其中 i = 0,1,2...n-1 中的最大值。

我们继续来看下递归关系，即 Q(list, i)和 Q(list, i - 1)的关系， 即如何根据 Q(list, i - 1) 推导出 Q(list, i)。

如果已知 Q(list, i - 1)， 我们可以将问题分为两种情况，即以索引为 i 的元素终止， 或者只有一个索引为 i 的元素。

- 如果以索引为 i 的元素终止， 那么就是 Q(list, i - 1) + list[i]
- 如果只有一个索引为 i 的元素，那么就是 list[i]

分析到这里，递推关系就很明朗了，即 `Q(list, i) = Math.max(0, Q(list, i - 1)) + list[i]`

举例说明，如下图：

(by snowan)

这种算法的时间复杂度 O(N), 空间复杂度为 O(1)

代码

JavaScript:

```
function LSS(list) {
  const len = list.length;
  let max = list[0];
  for (let i = 1; i < len; i++) {
    list[i] = Math.max(0, list[i - 1]) + list[i];
    if (list[i] > max) max = list[i];
  }

  return max;
}
```

Java:

```
class MaximumSubarrayDP {
  public int maxSubArray(int[] nums) {
     int currMaxSum = nums[0];
     int maxSum = nums[0];
     for (int i = 1; i < nums.length; i++) {
       currMaxSum = Math.max(currMaxSum + nums[i], nums[i]
       maxSum = Math.max(maxSum, currMaxSum);
     }
     return maxSum;
  }
}
```

Python 3:

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        max_sum_ending_curr_index = max_sum = nums[0]
        for i in range(1, n):
            max_sum_ending_curr_index = max(max_sum_ending_
            max_sum = max(max_sum_ending_curr_index, max_su

        return max_sum
```

解法四 - 数学分析

思路

我们来通过数学分析来看一下这个题目。

我们定义函数 S(i) ， 它的功能是计算以 0（包括 0）开始加到 i（包括 i）的值。

那么 S(j) - S(i - 1) 就等于 从 i 开始（包括 i）加到 j（包括 j）的值。

我们进一步分析，实际上我们只需要遍历一次计算出所有的 S(i), 其中 i 等于 0,1,2....,n-1。 然后我们再减去之前的 S(k),其中 k 等于 0，1，i - 1，中的最小值即可。 因此我们需要 用一个变量来维护这个最小值，还需要一个变量维护最大值。

这种算法的时间复杂度 O(N), 空间复杂度为 O(1)。

其实很多题目，都有这样的思想， 比如之前的《每日一题 - 电梯问题》。

代码

JavaScript:

```
function LSS(list) {
  const len = list.length;
  let max = list[0];
  let min = 0;
  let sum = 0;
  for (let i = 0; i < len; i++) {
    sum += list[i];
    if (sum - min > max) max = sum - min;
    if (sum < min) {
      min = sum;
    }
  }

  return max;
}
```

Java:

```
class MaxSumSubarray {
  public int maxSubArray3(int[] nums) {
      int maxSum = nums[0];
      int sum = 0;
      int minSum = 0;
      for (int num : nums) {
        // prefix Sum
        sum += num;
        // update maxSum
        maxSum = Math.max(maxSum, sum - minSum);
        // update minSum
        minSum = Math.min(minSum, sum);
      }
      return maxSum;
  }
}
```

Python 3:

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        maxSum = nums[0]
        minSum = sum = 0
        for i in range(n):
            sum += nums[i]
            maxSum = max(maxSum, sum - minSum)
            minSum = min(minSum, sum)

        return maxSum
```

总结

我们使用四种方法解决了 《最大子序列和问题》，并详细分析了各个解法的思路以及复杂度，相信下次你碰到相同或者类似的问题 的时候也能够发散思维，做到 一题多解，多题一解 。

实际上，我们只是求出了最大的和，如果题目进一步要求出最大子序列和的子序列呢？ 如果要题目允许不连续呢？ 我们又该如何思考和变通？如何将数组改成二维，求解最大矩阵和怎么计算？ 这些问题留给读者自己来思考。

简单难度题目合集

这里的题目难度比较小， 大多是模拟题，或者是很容易看出解法的题目，另外简单题目一般使用暴力法都是可以解决的。 这个时候只有看一下数据范围，思考下你的算法复杂度就行了。

当然也不排除很多 hard 题目也可以暴力模拟，大家平时多注意数据范围即可。

- 面试题 17.12. BiNode 🆕
- 0001.two-sum
- 0020.Valid Parentheses
- 0021.MergeTwoSortedLists
- 0026.remove-duplicates-from-sorted-array
- 0053.maximum-sum-subarray
- 0088.merge-sorted-array
- 0101.symmetric-tree🆕
- 0104.maximum-depth-of-binary-tree
- 0108.convert-sorted-array-to-binary-search-tree
- 0121.best-time-to-buy-and-sell-stock
- 0122.best-time-to-buy-and-sell-stock-ii
- 0125.valid-palindrome
- 0136.single-number
- 0155.min-stack 🆕
- 0167.two-sum-ii-input-array-is-sorted
- 0169.majority-element
- 0172.factorial-trailing-zeroes
- 0190.reverse-bits
- 0191.number-of-1-bits
- 0198.house-robber
- 0203.remove-linked-list-elements
- 0206.reverse-linked-list
- 0219.contains-duplicate-ii
- 0226.invert-binary-tree
- 0232.implement-queue-using-stacks 🆕
- 0263.ugly-number
- 0283.move-zeroes
- 0342.power-of-four
- 0349.intersection-of-two-arrays
- 0371.sum-of-two-integers
- 0437.path-sum-iii 🆕
- 0455.AssignCookies 🆕

- 0575.distribute-candies
- 0874.walking-robot-simulation 🆕
- 1260.shift-2d-grid 🆕
- 1332.remove-palindromic-subsequences 🆕

题目地址（面试题 17.12. BiNode）

https://leetcode-cn.com/problems/binode-lcci/

题目描述

```
二叉树数据结构TreeNode可用来表示单向链表（其中left置空，right为下一

返回转换后的单向链表的头节点。

注意：本题相对原题稍作改动



示例：

输入：  [4,2,5,1,3,null,6,0]
输出：  [0,null,1,null,2,null,3,null,4,null,5,null,6]
提示：

节点数量不会超过 100000。
```

前置知识

- 二叉查找树
- 递归
- 二叉树的遍历

公司

- 暂无

思路

实际上这就是一个考察二叉树遍历 + 二叉查找树性质的题目。需要注意的是指针操作，这一点和链表反转系列题目是一样的。

首先我们要知道一个性质： 对于一个二叉查找树来说，其中序遍历结果是一个有序数组。 而题目要求你输出的恰好就是有序数组（虽然没有明说， 不过从测试用例也可以看出）。

因此一个思路就是中序遍历， 边遍历边改变指针即可。 这里有两个注意点:

1. 指针操作小心死循环
2. 你需要返回的是最左下角的节点，而不是题目给的 root

对于第一个问题， 其实只要注意操作指针的顺序即可。对于第二个问题，我用了一个黑科技，让代码看起来简洁又高效。如果不懂的话， 你也可以换个朴素的写法。

让我们进入正题。

其中绿色是我们要增加的连线，而黑色是是原本的连线。

我们再来看一个复杂一点的：

实际上，不管多么复杂。 我们只需要进行一次**中序遍历**，同时记录前驱节点。然后操作修改前驱节点和当前节点的指针即可，整个过程就好像是链表反转。

核心代码（假设 pre 我们已经正确计算出了）：

```
cur.left = None
pre.right = cur
pre = cur
```

剩下的就是如何计算 pre，这个也不难，直接看代码：

```
self.pre = None
def dfs(root):
    dfs(root.left)
    # 上面的指针改变逻辑写到这里
    self.pre = root
    dfs(root.right)
```

问题得以解决。

这里还有最后一个问题就是返回值，题目要返回的实际上是最左下角的值。而我用了一个黑科技的方法（注意看注释）：

```
self.pre = self.ans = TreeNode(-1)
def dfs(root):
   if not root: return
   dfs(root.left)
   root.left = None
   self.pre.right = root
   # 当第一次执行到下面这一行代码，恰好是在最左下角，  这个时候 self.
   # 之后 self.pre 的变化都不会体现到 self.ans 上。
   # 直观上来说就是 self.ans 在遍历到最左下角的时候下车了
   # 因此最后返回 self.ans 即可
   self.pre = root
   dfs(root.right)
dfs(root)
return self.ans.right
```

关键点

- 指针操作
- 返回值的处理

代码

```python
class Solution:
    def convertBiNode(self, root):
        self.pre = self.ans = TreeNode(-1)
        def dfs(root):
            if not root: return
            dfs(root.left)
            root.left = None
            self.pre.right = root
            self.pre = root
            dfs(root.right)
        dfs(root)
        return self.ans.right
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为树的节点总数。
- 空间复杂度：$O(h)$，其中 h 为树的高度。

相关题目

- 206.reverse-linked-list
- 92.reverse-linked-list-ii
- 25.reverse-nodes-in-k-groups-cn

大家也可以关注我的公众号《力扣加加》获取更多更新鲜的 LeetCode 题解

题目地址(1. 两数之和)

https://leetcode-cn.com/problems/two-sum

题目描述

```
给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的

示例:

给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9
所以返回 [0, 1]
```

前置知识

- 哈希表

公司

- 字节
- 百度
- 腾讯
- adobe
- airbnb
- amazon
- apple
- bloomberg
- dropbox
- facebook
- linkedin
- microsoft
- uber
- yahoo
- yelp

思路

最容易想到的就是暴力枚举，我们可以利用两层 for 循环来遍历每个元素，并查找满足条件的目标元素。不过这样时间复杂度为 O(N^2)，空间复杂度为 O(1)，时间复杂度较高，我们要想办法进行优化。我们可以增加一个 Map 记录已经遍历过的数字及其对应的索引值。这样当遍历一个新数字的时候去 Map 里查询，target 与该数的差值是否已经在前面的数字中出现过。如果出现过，那么已经得出答案，就不必再往下执行了。

关键点

- 求和转换为求差
- 借助 Map 结构将数组中每个元素及其索引相互对应
- 以空间换时间，将查找时间从 O(N) 降低到 O(1)

代码

- 语言支持：JS

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
const twoSum = function (nums, target) {
  const map = new Map();
  for (let i = 0; i < nums.length; i++) {
    const diff = target - nums[i];
    if (map.has(diff)) {
      return [map.get(diff), i];
    }
    map.set(nums[i], i);
  }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(20. 有效的括号)

https://leetcode-cn.com/problems/valid-parentheses/description

题目描述

```
给定一个只包括 '(', ')', '{', '}', '[', ']' 的字符串，判断字符串是

有效字符串需满足：

左括号必须用相同类型的右括号闭合。
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
```

前置知识

- 栈

公司

- 阿里
- 百度

- 腾讯
- 字节
- airbnb
- amazon
- bloomberg
- facebook
- google
- microsoft
- twitter
- zenefits

栈

思路

关于这道题的思路，邓俊辉讲的非常好，没有看过的同学可以看一下，视频地址。

使用栈，遍历输入字符串

如果当前字符为左半边括号时，则将其压入栈中

如果遇到右半边括号时，分类讨论：

1）如栈不为空且为对应的左半边括号，则取出栈顶元素，继续循环

2）若此时栈为空，则直接返回 false

3）若不为对应的左半边括号，反之返回 false

（图片来自： https://github.com/MisterBooo/LeetCodeAnimation）

> 值得注意的是，如果题目要求只有一种括号，那么我们其实可以使用更简洁，更省内存的方式 - 计数器来进行求解，而 不必要使用栈。
>
> 事实上，这类问题还可以进一步扩展，我们可以去解析类似 HTML 等标记语法， 比如

关键点解析

1. 栈的基本特点和操作
2. 如果你用的是 JS 没有现成的栈，可以用数组来模拟 入： push 出： pop

> 入： push 出 shift 就是队列

代码

代码支持：JS，Python

Javascript Code:

```
/**
 * @param {string} s
 * @return {boolean}
 */
var isValid = function (s) {
  let valid = true;
  const stack = [];
  const mapper = {
    "{": "}",
    "[": "]",
    "(": ")",
  };

  for (let i in s) {
    const v = s[i];
    if (["(", "[", "{"].indexOf(v) > -1) {
      stack.push(v);
    } else {
      const peak = stack.pop();
      if (v !== mapper[peak]) {
        return false;
      }
    }
  }

  if (stack.length > 0) return false;

  return valid;
};
```

Python Code:

```
class Solution:
    def isValid(self,s):
      stack = []
      map = {
        "{":"}",
        "[":"]",
        "(":")"
      }
      for x in s:
        if x in map:
          stack.append(map[x])
        else:
          if len(stack)!=0:
            top_element = stack.pop()
            if x != top_element:
              return False
            else:
              continue
          else:
            return False
      return len(stack) == 0
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

O(1) 空间

思路

基本思路是修改参数，将参数作为我们的栈。 随着我们不断遍历， s 慢慢变成了一个栈。

因此 Python，Java，JS 等**字符串不可变**的语言无法使用此方法达到 $O(1)$ 。

具体参考： No stack O(1) space complexity O(n) time complexity solution in C++-space-complexity-O(n)-time-complexity-solution-in-C++/244061>)

代码

代码支持：C++

C++:

```
class Solution {
public:
    bool isValid(string s) {
        int top = -1;
        for(int i =0;i<s.length();++i){
            if(top<0 || !isMatch(s[top], s[i])){
                ++top;
                s[top] = s[i];
            }else{
                --top;
            }
        }
        return top == -1;
    }
    bool isMatch(char c1, char c2){
        if(c1 == '(' && c2 == ')') return true;
        if(c1 == '[' && c2 == ']') return true;
        if(c1 == '{' && c2 == '}') return true;
        return false;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

正则匹配

思路

我们不断通过消除 '[]' ， '()', '{}' ， 最后判断剩下的是否是空串即可，就像开心消消乐一样。

代码

代码支持：Python，JavaScript

Python:

```
class Solution:
     def isValid(self, s):

        while '[]' in s or '()' in s or '{}' in s:
            s = s.replace('[]','').replace('()','').replace
        return not len(s)
```

JavaScript:

```
var isValid = function (s) {
  while (s.includes("[]") || s.includes("()") || s.includes
    s = s.replace("[]", "").replace("()", "").replace("{}",
  }
  s = s.replace("[]", "").replace("()", "").replace("{}", "
  return s.length === 0;
};
```

复杂度分析

- 时间复杂度：取决于正则引擎的实现
- 空间复杂度：取决于正则引擎的实现

相关题目

- 32. 最长有效括号

扩展

- 如果让你检查 XML 标签是否闭合如何检查， 更进一步如果要你实现一个简单的 XML 的解析器，应该怎么实现?

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 35K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(21. 合并两个有序链表)

https://leetcode-cn.com/problems/merge-two-sorted-lists

题目描述

```
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

示例：

输入：1->2->4, 1->3->4
输出：1->1->2->3->4->4
```

前置知识

- 递归
- 链表

公司

- 阿里
- 字节
- 腾讯
- amazon
- apple
- linkedin
- microsoft

公司

- 阿里、字节、腾讯

思路

使用递归来解题，将两个链表头部较小的一个与剩下的元素合并，并返回排好序的链表头，当两条链表中的一条为空时终止递归。

关键点

- 掌握链表数据结构

- 考虑边界情况 2

代码

- 语言支持：JS

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} l1
 * @param {ListNode} l2
 * @return {ListNode}
 */
const mergeTwoLists = function (l1, l2) {
  if (l1 === null) {
    return l2;
  }
  if (l2 === null) {
    return l1;
  }
  if (l1.val < l2.val) {
    l1.next = mergeTwoLists(l1.next, l2);
    return l1;
  } else {
    l2.next = mergeTwoLists(l1, l2.next);
    return l2;
  }
};
```

复杂度分析

M、N 是两条链表 l1、l2 的长度

- 时间复杂度：$O(M+N)$
- 空间复杂度：$O(M+N)$

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 35K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(26. 删除排序数组中的重复项)

https://leetcode-cn.com/problems/remove-duplicates-from-sorted-array/description/

题目描述

给定一个排序数组，你需要在 原地 删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

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:

给定数组 nums = [1,1,2],

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。

你不需要考虑数组中超出新长度后面的元素。 示例 2:

给定 nums = [0,0,1,1,1,2,2,3,3,4],

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。

你不需要考虑数组中超出新长度后面的元素。

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以「引用」方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下:

```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

前置知识

- 数组
- 双指针

公司

- 阿里
- 腾讯
- 百度
- 字节
- bloomberg
- facebook
- microsoft

思路

使用快慢指针来记录遍历的坐标。

- 开始时这两个指针都指向第一个数字
- 如果两个指针指的数字相同，则快指针向前走一步
- 如果不同，则两个指针都向前走一步
- 当快指针走完整个数组后，慢指针当前的坐标加 1 就是数组中不同数字的个数

（图片来自： https://github.com/MisterBooo/LeetCodeAnimation）

> 实际上这就是双指针中的快慢指针。在这里快指针是读指针， 慢指针是写指针。

关键点解析

- 双指针

这道题如果不要求，O(n) 的时间复杂度， O(1) 的空间复杂度的话，会很简单。 但是这道题是要求的，这种题的思路一般都是采用双指针

- 如果是数据是无序的，就不可以用这种方式了，从这里也可以看出排序在算法中的基础性和重要性。
- 注意 nums 为空时的边界条件。

代码

- 语言支持：JS，Python，C++

Javascript Code:

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var removeDuplicates = function (nums) {
  const size = nums.length;
  if (size == 0) return 0;
  let slowP = 0;
  for (let fastP = 0; fastP < size; fastP++) {
    if (nums[fastP] !== nums[slowP]) {
      slowP++;
      nums[slowP] = nums[fastP];
    }
  }
  return slowP + 1;
};
```

Python Code:

```
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        if nums:
            slow = 0
            for fast in range(1, len(nums)):
                if nums[fast] != nums[slow]:
                    slow += 1
                    nums[slow] = nums[fast]
            return slow + 1
        else:
            return 0
```

C++ Code:

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if(nums.empty()) return 0;
        int fast,slow;
        fast=slow=0;
        while(fast!=nums.size()){
            if(nums[fast]==nums[slow]) fast++;
            else {
                slow++;
                nums[slow]=nums[fast];
                fast++;
            }
        }
        return slow+1;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(53. 最大子序和)

https://leetcode-cn.com/problems/maximum-subarray/

题目描述

```
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

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。
进阶:

如果你已经实现复杂度为 O(n) 的解法，尝试使用更为精妙的分治法求解。
```

前置知识

- 滑动窗口
- 动态规划

公司

- 阿里
- 百度
- 字节
- 腾讯
- bloomberg
- linkedin
- microsoft

公司

- 阿里、百度、字节、腾讯

思路

这道题求解连续最大子序列和，以下从时间复杂度角度分析不同的解题思路。

解法一 - 暴力解 （暴力出奇迹， 噢耶！）

一般情况下，先从暴力解分析，然后再进行一步步的优化。

原始暴力解： （超时）

求子序列和，那么我们要知道子序列的首尾位置，然后计算首尾之间的序列和。用 2 个 for 循环可以枚举所有子序列的首尾位置。 然后用一个 for 循环求解序列和。这里时间复杂度太高， `O(n^3)` .

复杂度分析

- 时间复杂度：$O(N^3)$, 其中 N 是数组长度
- 空间复杂度：$O(1)$

解法二 - 前缀和 + 暴力解

优化暴力解： (震惊，居然 AC 了）

在暴力解的基础上，用前缀和我们可以优化到暴力解 `O(n^2)` , 这里以空间换时间。 这里可以使用原数组表示 `prefixSum` , 省空间。

求序列和可以用前缀和（ `prefixSum`) 来优化，给定子序列的首尾位置 `(l, r),` 那么序列和 `subarraySum=prefixSum[r] - prefixSum[l - 1];` 用一个全局变量 `maxSum` , 比较每次求解的子序列和， `maxSum = max(maxSum, subarraySum)` .

复杂度分析

- 时间复杂度：$O(N^2)$, 其中 N 是数组长度
- 空间复杂度：$O(N)$

> 如果用更改原数组表示前缀和数组，空间复杂度降为 `O(1)`

但是时间复杂度还是太高，还能不能更优化。答案是可以，前缀和还可以优化到 `O(n)` .

解法三 - 优化前缀和 - from @lucifer

我们定义函数 `S(i)` ，它的功能是计算以 `0（包括 0）` 开始加到 `i（包括 i）` 的值。

那么 `S(j) - S(i - 1)` 就等于 从 `i` 开始（包括 i）加到 `j` （包括 j）的值。

我们进一步分析，实际上我们只需要遍历一次计算出所有的 `S(i)` , 其中 `i = 0,1,2....,n-1。` 然后我们再减去之前的 `S(k)` ,其中 `k = 0, 1, i - 1` ，中的最小值即可。 因此我们需要 用一个变量来维护这个最小值，还需要一个变量维护最大值。

复杂度分析

- 时间复杂度：$O(N)$, 其中 N 是数组长度
- 空间复杂度：$O(1)$

解法四 - 分治法

我们把数组 `nums` 以中间位置（ `m`)分为左（ `left`)右(`right`)两部分. 那么有， `left = nums[0]...nums[m - 1]` 和 `right = nums[m + 1]...nums[n-1]`

最大子序列和的位置有以下三种情况：

1. 考虑中间元素 `nums[m]` , 跨越左右两部分，这里从中间元素开始，往左求出后缀最大，往右求出前缀最大, 保持连续性。
2. 不考虑中间元素，最大子序列和出现在左半部分，递归求解左边部分最大子序列和
3. 不考虑中间元素，最大子序列和出现在右半部分，递归求解右边部分最大子序列和

分别求出三种情况下最大子序列和，三者中最大值即为最大子序列和。

举例说明，如下图：

复杂度分析

- 时间复杂度：$O(NlogN)$, 其中 N 是数组长度
- 空间复杂度：$O(logN)$

解法五 - 动态规划

动态规划的难点在于找到状态转移方程，

`dp[i] - 表示到当前位置 i 的最大子序列和`

状态转移方程为： `dp[i] = max(dp[i - 1] + nums[i], nums[i])`

初始化： `dp[0] = nums[0]`

从状态转移方程中，我们只关注前一个状态的值，所以不需要开一个数组记录位置所有子序列和，只需要两个变量，

`currMaxSum - 累计最大和到当前位置i`

`maxSum - 全局最大子序列和` ：

- `currMaxSum = max(currMaxSum + nums[i], nums[i])`
- `maxSum = max(currMaxSum, maxSum)`

如图：

复杂度分析

- 时间复杂度：$O(N)$, 其中 N 是数组长度
- 空间复杂度：$O(1)$

关键点分析

1. 暴力解，列举所有组合子序列首尾位置的组合，求解最大的子序列和, 优化可以预先处理，得到前缀和
2. 分治法，每次从中间位置把数组分为左右中三部分， 分别求出左右中（这里中是包括中间元素的子序列）最大和。对左右分别深度递归，三者中最大值即为当前最大子序列和。
3. 动态规划，找到状态转移方程，求到当前位置最大和。

代码 (`Java/Python3/Javascript`)

解法二 - 前缀和 + 暴力

Java code

```java
class MaximumSubarrayPrefixSum {
  public int maxSubArray(int[] nums) {
      int len = nums.length;
      int maxSum = Integer.MIN_VALUE;
      int sum = 0;
      for (int i = 0; i < len; i++) {
        sum = 0;
        for (int j = i; j < len; j++) {
          sum += nums[j];
          maxSum = Math.max(maxSum, sum);
        }
      }
      return maxSum;
  }
}
```

Python3 code `(TLE)`

```python
import sys
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        maxSum = -sys.maxsize
        sum = 0
        for i in range(n):
            sum = 0
            for j in range(i, n):
                sum += nums[j]
                maxSum = max(maxSum, sum)

        return maxSum
```

Javascript code from **@lucifer**

```
function LSS(list) {
  const len = list.length;
  let max = -Number.MAX_VALUE;
  let sum = 0;
  for (let i = 0; i < len; i++) {
    sum = 0;
    for (let j = i; j < len; j++) {
      sum += list[j];
      if (sum > max) {
        max = sum;
      }
    }
  }

  return max;
}
```

解法三 - 优化前缀和

Java code

```
class MaxSumSubarray {
  public int maxSubArray3(int[] nums) {
      int maxSum = nums[0];
      int sum = 0;
      int minSum = 0;
      for (int num : nums) {
        // prefix Sum
        sum += num;
        // update maxSum
        maxSum = Math.max(maxSum, sum - minSum);
        // update minSum
        minSum = Math.min(minSum, sum);
      }
      return maxSum;
  }
}
```

Python3 code

```python
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        maxSum = nums[0]
        minSum = sum = 0
        for i in range(n):
            sum += nums[i]
            maxSum = max(maxSum, sum - minSum)
            minSum = min(minSum, sum)

        return maxSum
```

Javascript code from **@lucifer**

```javascript
function LSS(list) {
  const len = list.length;
  let max = list[0];
  let min = 0;
  let sum = 0;
  for (let i = 0; i < len; i++) {
    sum += list[i];
    if (sum - min > max) max = sum - min;
    if (sum < min) {
      min = sum;
    }
  }

  return max;
}
```

解法四 - 分治法

Java code

```
class MaximumSubarrayDivideConquer {
  public int maxSubArrayDividConquer(int[] nums) {
      if (nums == null || nums.length == 0) return 0;
      return helper(nums, 0, nums.length - 1);
    }
    private int helper(int[] nums, int l, int r) {
      if (l > r) return Integer.MIN_VALUE;
      int mid = (l + r) >>> 1;
      int left = helper(nums, l, mid - 1);
      int right = helper(nums, mid + 1, r);
      int leftMaxSum = 0;
      int sum = 0;
      // left surfix maxSum start from index mid - 1 to l
      for (int i = mid - 1; i >= l; i--) {
        sum += nums[i];
        leftMaxSum = Math.max(leftMaxSum, sum);
      }
      int rightMaxSum = 0;
      sum = 0;
      // right prefix maxSum start from index mid + 1 to r
      for (int i = mid + 1; i <= r; i++) {
        sum += nums[i];
        rightMaxSum = Math.max(sum, rightMaxSum);
      }
      // max(left, right, crossSum)
      return Math.max(leftMaxSum + rightMaxSum + nums[mid],
    }
}
```

Python3 code

```
import sys
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        return self.helper(nums, 0, len(nums) - 1)
    def helper(self, nums, l, r):
        if l > r:
            return -sys.maxsize
        mid = (l + r) // 2
        left = self.helper(nums, l, mid - 1)
        right = self.helper(nums, mid + 1, r)
        left_suffix_max_sum = right_prefix_max_sum = 0
        sum = 0
        for i in reversed(range(l, mid)):
            sum += nums[i]
            left_suffix_max_sum = max(left_suffix_max_sum,
        sum = 0
        for i in range(mid + 1, r + 1):
            sum += nums[i]
            right_prefix_max_sum = max(right_prefix_max_sur
        cross_max_sum = left_suffix_max_sum + right_prefix_
        return max(cross_max_sum, left, right)
```

Javascript code from **@lucifer**

```
function helper(list, m, n) {
  if (m === n) return list[m];
  let sum = 0;
  let lmax = -Number.MAX_VALUE;
  let rmax = -Number.MAX_VALUE;
  const mid = ((n - m) >> 1) + m;
  const l = helper(list, m, mid);
  const r = helper(list, mid + 1, n);
  for (let i = mid; i >= m; i--) {
    sum += list[i];
    if (sum > lmax) lmax = sum;
  }

  sum = 0;

  for (let i = mid + 1; i <= n; i++) {
    sum += list[i];
    if (sum > rmax) rmax = sum;
  }

  return Math.max(l, r, lmax + rmax);
}

function LSS(list) {
  return helper(list, 0, list.length - 1);
}
```

解法五 - 动态规划

Java code

```
class MaximumSubarrayDP {
  public int maxSubArray(int[] nums) {
     int currMaxSum = nums[0];
     int maxSum = nums[0];
     for (int i = 1; i < nums.length; i++) {
       currMaxSum = Math.max(currMaxSum + nums[i], nums[i]
       maxSum = Math.max(maxSum, currMaxSum);
     }
     return maxSum;
  }
}
```

Python3 code

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        max_sum_ending_curr_index = max_sum = nums[0]
        for i in range(1, n):
            max_sum_ending_curr_index = max(max_sum_ending_
            max_sum = max(max_sum_ending_curr_index, max_su

        return max_sum
```

Javascript code from **@lucifer**

```
function LSS(list) {
  const len = list.length;
  let max = list[0];
  for (let i = 1; i < len; i++) {
    list[i] = Math.max(0, list[i - 1]) + list[i];
    if (list[i] > max) max = list[i];
  }

  return max;
}
```

扩展

- 如果数组是二维数组，求最大子数组的和?
- 如果要求最大子序列的乘积?

相似题

- Maximum Product Subarray
- Longest Turbulent Subarray

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 35K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址（88. 合并两个有序数组）

https://leetcode-cn.com/problems/merge-sorted-array/

题目描述

```
给定两个有序整数数组 nums1 和 nums2，将 nums2 合并到 nums1 中，使

说明：

初始化 nums1 和 nums2 的元素数量分别为 m 和 n。
你可以假设 nums1 有足够的空间（空间大小大于或等于 m + n）来保存 num
示例：

输入：
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6],       n = 3

输出：[1,2,2,3,5,6]
```

公司

- 阿里
- 腾讯
- 百度
- 字节
- loomberg
- facebook
- microsoft

前置知识

- 归并排序

思路

符合直觉的做法是 `将nums2插到num1的末尾，然后排序`

具体代码：

```
// 这种解法连m都用不到
// 这显然不是出题人的意思
if (n === 0) return;
let current2 = 0;
for (let i = nums1.length - 1; i >= nums1.length - n; i--)
  nums1[i] = nums2[current2++];
}
nums1.sort((a, b) => a - b); // 当然你可以自己写排序，这里懒得写
```

这道题目其实和基本排序算法中的 `merge sort` 非常像，但是 merge sort 很多时候，合并的时候我们通常是 新建一个数组，这样就很简单。但是这道题目要求的是 `原地修改` .

这就和 merge sort 的 merge 过程有点不同，我们先来回顾一下 merge sort 的 merge 过程。

merge 的过程 `可以` 是先比较两个数组的头元素，然后将较小的推到最终的数组中，并将其从原数组中出队列。 循环直到两个数组都为空。

具体代码如下：

```
// 将nums1 和 nums2 合并
function merge(nums1, nums2) {
  let ret = [];
  while (nums1.length || nums2.length) {
    // 为了方便大家理解，这里代码有点赘余
    if (nums1.length === 0) {
      ret.push(nums2.shift());
      continue;
    }

    if (nums2.length === 0) {
      ret.push(nums1.shift());
      continue;
    }
    const a = nums1[0];
    const b = nums2[0];
    if (a > b) {
      ret.push(nums2.shift());
    } else {
      ret.push(nums1.shift());
    }
  }
  return ret;
}
```

这里要求原地修改，其实我们能只要从后往前比较，并从后往前插入即可。

我们需要三个指针：

1. current 用于记录当前填补到那个位置了
2. m 用于记录 nums1 数组处理到哪个元素了
3. n 用于记录 nums2 数组处理到哪个元素了

如图所示：

- 灰色代表 num2 数组已经处理过的元素
- 红色代表当前正在进行比较的元素
- 绿色代表已经就位的元素

current
1 2 3 0 0 0
2 5 6
current
1 2 3 0 0 6
2 5 6
current
1 2 3 0 5 6
2 5 6
current
1 2 3 3 5 6
2 5 6
current

关键点解析

- 从后往前比较，并从后往前插入

代码

代码支持：Python3, C++, Java, JavaScript

JavaSCript Code:

```
var merge = function (nums1, m, nums2, n) {
  // 设置一个指针，指针初始化指向nums1的末尾（根据#62，应该是index
  // 然后不断左移指针更新元素
  let current = m + n - 1;

  while (current >= 0) {
    // 没必要继续了
    if (n === 0) return;

    // 为了方便大家理解，这里代码有点赘余
    if (m < 1) {
      nums1[current--] = nums2[--n];
      continue;
    }

    if (n < 1) {
      nums1[current--] = nums1[--m];
      continue;
    }
    // 取大的填充 nums1的末尾
    // 然后更新 m 或者 n
    if (nums1[m - 1] > nums2[n - 1]) {
      nums1[current--] = nums1[--m];
    } else {
      nums1[current--] = nums2[--n];
    }
  }
};
```

C++ code:

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums
        int current = m + n - 1;
        while (current >= 0) {
            if (n == 0) return;
            if (m < 1) {
                nums1[current--] = nums2[--n];
                continue;
            }
            if (n < 1) {
                nums1[current--] = nums1[--m];
                continue;
            }
            if (nums1[m - 1] > nums2[n - 1]) nums1[current-
            else nums1[current--] = nums2[--n];
        }
    }
};
```

Java Code:

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int
        int i=m-1, j=n-1, k=m+n-1;
        // 合并
        while(i>=0 && j>=0)
        {
            if(nums1[i] > nums2[j])
            {
                nums1[k--] = nums1[i--];
            }
            else
            {
                nums1[k--] = nums2[j--];
            }
        }
        // 合并剩余的nums2
        while(j>=0)
        {
            nums1[k--] = nums2[j--];
        }
    }
}
```

Python Code:

```
class Solution:
    def merge(self, nums1: List[int], m: int, nums2: List[
        """
        Do not return anything, modify nums1 in-place inste
        """
        # 整体思路相似，只不过没有使用 current 指针记录当前填补位置
        while m > 0 and n > 0:
            if nums1[m-1] <= nums2[n-1]:
                nums1[m+n-1] = nums2[n-1]
                n -= 1
            else:
                nums1[m+n-1] = nums1[m-1]
                m -=1
        """
        由于没有使用 current，第一步比较结束后有两种情况:
            1. 指针 m>0，n=0，此时不需要做任何处理
            2. 指针 n>0，m=0，此时需要将 nums2 指针左侧元素全部拷
        """
        if n > 0:
            nums1[:n] = nums2[:n]
```

复杂度分析

- 时间复杂度：$O(M+N)$
- 空间复杂度：$O(1)$

欢迎关注我的公众号《脑洞前端》获取更多更新鲜的 LeetCode 题解

题目地址(101. 对称二叉树)

https://leetcode-cn.com/problems/symmetric-tree/

题目描述

```
给定一个二叉树，检查它是否是镜像对称的。



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


进阶:

你可以运用递归和迭代两种方法解决这个问题吗?
```

公司

- 阿里
- 腾讯
- 百度
- 字节
- bloomberg
- linkedin
- microsoft

前置知识

- 二叉树
- 递归

思路

看到这题的时候，我的第一直觉是 DFS。然后我就想：`如果左子树是镜像，并且右子树也是镜像，是不是就说明整体是镜像？` 。经过几秒的思考， 这显然是不对的，不符合题意。

很明显其中左子树中的节点会和右子树中的节点进行比较，我把比较的元素进行了颜色区分，方便大家看。

这里我的想法是： `遍历每一个节点的时候，我都可以通过某种方法知道它对应的对称节点是谁。这样的话我直接比较两者是否一致就行了。`

最初我的想法是两次遍历，第一次遍历的同时将遍历结果存储到哈希表中，然后第二次遍历去哈希表取。

这种方法可行，但是需要 N 的空间（N 为节点总数）。我想到如果两者可以同时进行遍历，是不是就省去了哈希表的开销。

如果不明白的话，我举个简单例子：

```
给定一个数组，检查它是否是镜像对称的。例如，数组 [1,2,2,3,2,2,1] 是
```

如果用哈希表的话大概是：

```
seen = dict()
for i, num in enumerate(nums):
    seen[i] = num
for i, num in enumerate(nums):
    if  seen[len(nums) - 1 - i] != num:
      return False
return True
```

而同时遍历的话大概是这样的：

```
l = 0
r = len(nums) - 1

while l < r:
   if nums[l] != nums[r]: return False
   l += 1
   r -= 1
return True
```

> 其实更像本题一点的话应该是从中间分别向两边扩展 😂

代码

```
class Solution:
    def isSymmetric(self, root: TreeNode) -> bool:
        def dfs(root1, root2):
            if root1 == root2: return True
            if not root1 or not root2: return False
            if root1.val != root2.val: return False
            return dfs(root1.left, root2.right) and dfs(roo
        if not root: return True
        return dfs(root.left, root.right)
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为节点数。
- 空间复杂度：递归的深度最高为节点数，因此空间复杂度是 $O(N)$，其中 N 为节点数。

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

大家也可以关注我的公众号《脑洞前端》获取更多更新鲜的 LeetCode 题解

题目地址（104. 二叉树的最大深度）

https://leetcode-cn.com/problems/maximum-depth-of-binary-tree/description/

题目描述

```
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 。
```

前置知识

- 递归

公司

- 阿里
- 腾讯
- 百度
- 字节
- apple
- linkedin
- uber
- yahoo

思路

由于树是一种递归的数据结构，因此用递归去解决的时候往往非常容易，这道题恰巧也是如此， 用递归实现的代码如下：

```
var maxDepth = function (root) {
  if (!root) return 0;
  if (!root.left && !root.right) return 1;
  return 1 + Math.max(maxDepth(root.left), maxDepth(root.r
};
```

如果使用迭代呢？ 我们首先应该想到的是树的各种遍历，由于我们求的是深度，因此 使用层次遍历（BFS）是非常合适的。 我们只需要记录有多少层即可。相关思路请查看binary-tree-traversal

关键点解析

- 队列
- 队列中用 Null(一个特殊元素)来划分每层，或者在对每层进行迭代之前保存当前队列元素的个数（即当前层所含元素个数）
- 树的基本操作- 遍历 - 层次遍历（BFS）

代码

- 语言支持：JS，C++，Python

JavaScript Code:

```javascript
/*
 * @lc app=leetcode id=104 lang=javascript
 *
 * [104] Maximum Depth of Binary Tree
 */
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
var maxDepth = function (root) {
  if (!root) return 0;
  if (!root.left && !root.right) return 1;

  // 层次遍历 BFS
  let cur = root;
  const queue = [root, null];
  let depth = 1;

  while ((cur = queue.shift()) !== undefined) {
    if (cur === null) {
      // 注意⚠: 不处理会无限循环，进而堆栈溢出
      if (queue.length === 0) return depth;
      depth++;
      queue.push(null);
      continue;
    }
    const l = cur.left;
    const r = cur.right;

    if (l) queue.push(l);
    if (r) queue.push(r);
  }

  return depth;
};
```

C++ Code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {
 * };
 */
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (root == nullptr) return 0;
        auto q = vector<TreeNode*>();
        auto d = 0;
        q.push_back(root);
        while (!q.empty())
        {
            ++d;
            auto sz = q.size();
            for (auto i = 0; i < sz; ++i)
            {
                auto t = q.front();
                q.erase(q.begin());
                if (t->left != nullptr) q.push_back(t->left
                if (t->right != nullptr) q.push_back(t->rig
            }
        }
        return d;
    }
};
```

Python Code:

```python
class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        if not root: return 0
        q, depth = [root, None], 1
        while q:
            node = q.pop(0)
            if node:
                if node.left: q.append(node.left)
                if node.right: q.append(node.right)
            elif q:
                q.append(None)
                depth += 1
        return depth
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

相关题目

- 102.binary-tree-level-order-traversal
- 103.binary-tree-zigzag-level-order-traversal

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 35K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(108. 将有序数组转换为二叉搜索树)

https://leetcode-cn.com/problems/convert-sorted-array-to-binary-search-tree/

题目描述

```
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

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

      0
     / \
   -3   9
   /   /
 -10  5
```

前置知识

- 二叉搜索树
- 平衡二叉树
- 递归

公司

- 阿里
- 腾讯
- 百度
- 字节
- airbnb

思路

由于输入是一个**升序排列的有序数组**。因此任意选择一点，将其作为根节点，其左部分左节点，其右部分右节点即可。 因此我们很容易写出递归代码。

而题目要求是**高度平衡**的二叉搜索树，因此我们必须要取中点。 不难证明： `由于是中点，因此左右两部分差不会大于 1，也就是说其形成的左右子树节点数最多相差 1，因此左右子树高度差的绝对值不超过 1` 。

形象一点来看就像你提起一根绳子，从中点提的话才能使得两边绳子长度相差最小。

关键点

- 找中点

代码

代码支持：JS，C++，Java，Python

JS Code:

```
var sortedArrayToBST = function (nums) {
  // 由于数组是排序好的，因此一个思路就是将数组分成两半，一半是左子树
  // 然后运用“树的递归性质”递归完成操作即可。
  if (nums.length === 0) return null;
  const mid = nums.length >> 1;
  const root = new TreeNode(nums[mid]);

  root.left = sortedArrayToBST(nums.slice(0, mid));
  root.right = sortedArrayToBST(nums.slice(mid + 1));
  return root;
};
```

Python Code:

```
class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> TreeNode
        if not nums: return None
        mid = (len(nums) - 1) // 2
        root = TreeNode(nums[mid])
        root.left = self.sortedArrayToBST(nums[:mid])
        root.right = self.sortedArrayToBST(nums[mid + 1:])
        return root
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：每次递归都 copy 了 N 的 空间，因此空间复杂度为 $O(N^2)$

然而，实际上没必要开辟新的空间：

C++ Code:

```
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return reBuild(nums, 0, nums.size()-1);
    }

    TreeNode* reBuild(vector<int>& nums, int left, int rigl
    {
        // 终止条件：中序遍历为空
        if(left > right)
        {
            return NULL;
        }
        // 建立当前子树的根节点
        int mid = (left+right)/2;
        TreeNode * root = new TreeNode(nums[mid]);

        // 左子树的下层递归
        root->left = reBuild(nums, left, mid-1);
        // 右子树的下层递归
        root->right = reBuild(nums, mid+1, right);
        // 返回根节点
        return root;
    }
};
```

Java Code:

```
class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        return dfs(nums, 0, nums.length - 1);
    }

    private TreeNode dfs(int[] nums, int lo, int hi) {
        if (lo > hi) {
            return null;
        }
        int mid = lo + (hi - lo) / 2;
        TreeNode root = new TreeNode(nums[mid]);
        root.left = dfs(nums, lo, mid - 1);
        root.right = dfs(nums, mid + 1, hi);
        return root;
    }
}
```

Python Code:

```python
class Solution(object):
    def sortedArrayToBST(self, nums):
        """
        :type nums: List[int]
        :rtype: TreeNode
        """
        return self.reBuild(nums, 0, len(nums)-1)

    def reBuild(self, nums, left, right):
         # 终止条件:
        if left > right:
            return
        # 建立当前子树的根节点
        mid = (left + right)//2
        root = TreeNode(nums[mid])
        # 左右子树的下层递归
        root.left = self.reBuild(nums, left, mid-1)
        root.right = self.reBuild(nums, mid+1, right)

        return root
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：隐式调用栈的开销为 $O(N)$

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

公众号【 力扣加加】 知乎专栏【 Lucifer - 知乎】

点关注，不迷路！

题目地址(121. 买卖股票的最佳时机)

https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock/description/

题目描述

```
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

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）
     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
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。
```

前置知识

- 数组

公司

- 阿里
- 腾讯
- 百度
- 字节
- amazon
- bloomberg
- facebook
- microsoft
- uber

思路

由于我们是想获取到最大的利润，我们的策略应该是低点买入，高点卖出。

由于题目对于交易次数有限制，只能交易一次，因此问题的本质其实就是求波峰浪谷的差值的最大值。

用图表示的话就是这样：

关键点解析

- 这类题只要你在心中（或者别的地方）画出上面这种图就很容易解决

代码

语言支持：JS，C++，Java，Python

JS Code:

```
/**
 * @param {number[]} prices
 * @return {number}
 */
var maxProfit = function (prices) {
  let min = prices[0];
  let profit = 0;
  // 7 1 5 3 6 4
  for (let i = 1; i < prices.length; i++) {
    if (prices[i] > prices[i - 1]) {
      profit = Math.max(profit, prices[i] - min);
    } else {
      min = Math.min(min, prices[i]);
    }
  }

  return profit;
};
```

C++ Code:

```
/**
 * 系统上C++的测试用例中的输入有[]，因此需要加一个判断
 */
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if (prices.empty()) return 0;
        auto min = prices[0];
        auto profit = 0;
        for (auto i = 1; i < prices.size(); ++i) {
            if (prices[i] > prices[i -1]) {
                profit = max(profit, prices[i] - min);
            } else {
                min = std::min(min, prices[i]);;
            }
        }
        return profit;
    }
};
```

Java Code:

```
class Solution {
    public int maxProfit(int[] prices) {
        int minprice = Integer.MAX_VALUE;
        int maxprofit = 0;
        for (int price: prices) {
            maxprofit = Math.max(maxprofit, price - minpric
             minprice = Math.min(price, minprice);
        }
        return maxprofit;
    }
}
```

Python Code:

```
class Solution:
    def maxProfit(self, prices: 'List[int]') -> int:
        if not prices: return 0

        min_price = float('inf')
        max_profit = 0

        for price in prices:
            if price < min_price:
                min_price = price
            elif max_profit < price - min_price:
                max_profit = price - min_price
        return max_profit
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

相关题目

- 122.best-time-to-buy-and-sell-stock-ii
- 309.best-time-to-buy-and-sell-stock-with-cooldown

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 35K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(122. 买卖股票的最佳时机 II)

https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-ii/description/

题目描述

```
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（

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）
      随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格
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
      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
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
```

前置知识

- 数组

公司

- 阿里
- 腾讯
- 百度
- 字节
- bloomberg

思路

由于我们是想获取到最大的利润，我们的策略应该是低点买入，高点卖出。

由于题目对于交易次数没有限制，因此只要能够赚钱的机会我们都不应该放过。

> 如下图，我们只需要求出加粗部分的总和即可

用图表示的话就是这样：

关键点解析

- 这类题只要你在心中（或者别的地方）画出上面这种图就很容易解决

代码

语言支持：JS，C++，Java

JS Code:

```
/**
 * @param {number[]} prices
 * @return {number}
 */
var maxProfit = function (prices) {
  let profit = 0;

  for (let i = 1; i < prices.length; i++) {
    if (prices[i] > prices[i - 1]) {
      profit = profit + prices[i] - prices[i - 1];
    }
  }

  return profit;
};
```

C++ Code:

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int res = 0;
        for(int i=1;i<prices.size();i++)
        {
            if(prices[i] > prices[i-1])
            {
                res += prices[i] - prices[i-1];
            }
        }
        return res;
    }
};
```

Java Code:

```
class Solution {
    public int maxProfit(int[] prices) {
        int res = 0;
        for(int i=1;i<prices.length;i++)
        {
            if(prices[i] > prices[i-1])
            {
                res += prices[i] - prices[i-1];
            }
        }
        return res;
    }
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

相关题目

- 121.best-time-to-buy-and-sell-stock
- 309.best-time-to-buy-and-sell-stock-with-cooldown

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 35K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(125. 验证回文串)

https://leetcode-cn.com/problems/valid-palindrome/description/

题目描述

```
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

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
```

前置知识

- 回文
- 双指针

公司

- 阿里
- 腾讯
- 百度
- 字节
- facebook
- microsoft
- uber
- zenefits

思路

这是一道考察回文的题目，而且是最简单的形式，即判断一个字符串是否是回文。

针对这个问题，我们可以使用头尾双指针，

- 如果两个指针的元素不相同，则直接返回 false，
- 如果两个指针的元素相同，我们同时更新头尾指针，循环。 直到头尾指针相遇。

时间复杂度为 O(n).

拿“noon”这样一个回文串来说，我们的判断过程是这样的：

拿“abaa”这样一个不是回文的字符串来说，我们的判断过程是这样的：

关键点解析

- 双指针

代码

- 语言支持：JS，C++，Python

JavaScript Code:

```javascript
/*
 * @lc app=leetcode id=125 lang=javascript
 *
 * [125] Valid Palindrome
 */
// 只处理英文字符（题目忽略大小写，我们前面全部转化成了小写， 因此这里
function isValid(c) {
  const charCode = c.charCodeAt(0);
  const isDigit =
    charCode >= "0".charCodeAt(0) && charCode <= "9".charC
  const isChar = charCode >= "a".charCodeAt(0) && charCode

  return isDigit || isChar;
}
/**
 * @param {string} s
 * @return {boolean}
 */
var isPalindrome = function (s) {
  s = s.toLowerCase();
  let left = 0;
  let right = s.length - 1;

  while (left < right) {
    if (!isValid(s[left])) {
      left++;
      continue;
    }
    if (!isValid(s[right])) {
      right--;
      continue;
    }

    if (s[left] === s[right]) {
      left++;
      right--;
    } else {
      break;
    }
  }

  return right <= left;
};
```

C++ Code:

```
class Solution {
public:
    bool isPalindrome(string s) {
        if (s.empty())
            return true;
        const char* s1 = s.c_str();
        const char* e = s1 + s.length() - 1;
        while (e > s1) {
            if (!isalnum(*s1)) {++s1; continue;}
            if (!isalnum(*e)) {--e; continue;}
            if (tolower(*s1) != tolower(*e)) return false;
            else {--e; ++s1;}
        }
        return true;
    }
};
```

Python Code:

```
class Solution:
    def isPalindrome(self, s: str) -> bool:
        left, right = 0, len(s) - 1
        while left < right:
            if not s[left].isalnum():
                left += 1
                continue
            if not s[right].isalnum():
                right -= 1
                continue
            if s[left].lower() == s[right].lower():
                left += 1
                right -= 1
            else:
                break
        return right <= left

    def isPalindrome2(self, s: str) -> bool:
        """
        使用语言特性进行求解
        """
        s = ''.join(i for i in s if i.isalnum()).lower()
        return s == s[::-1]
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 35K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址

https://leetcode.com/problems/single-number/description/

题目描述

```
Given a non-empty array of integers, every element appears

Note:

Your algorithm should have a linear runtime complexity. Cou
```

前置知识

- 位运算

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

根据题目描述，由于加上了时间复杂度必须是 O(n)，并且空间复杂度为 O(1)的条件，因此不能用排序方法，也不能使用 map 数据结构。

我们可以利用二进制异或的性质来完成，将所有数字异或即得到唯一出现的数字。

关键点

1. 异或的性质 两个数字异或的结果 `a^b` 是将 a 和 b 的二进制每一位进行运算，得出的数字。 运算的逻辑是 如果同一位的数字相同则为 0，不同则为 1
2. 异或的规律
3. 任何数和本身异或则为 `0`
4. 任何数和 0 异或是 `本身`

5. 很多人只是记得异或的性质和规律，但是缺乏对其本质的理解，导致很难想到这种解法（我本人也没想到）
6. bit 运算

代码

- 语言支持：JS，C，C++，Java，Python

JavaScrip Code：

```
/*
 * @lc app=leetcode id=136 lang=javascript
 *
 * [136] Single Number
 *
 * https://leetcode.com/problems/single-number/description/
 *
 * algorithms
 * Easy (59.13%)
 * Total Accepted:    429.3K
 * Total Submissions: 724.1K
 * Testcase Example:  '[2,2,1]'
 *
 * Given a non-empty array of integers, every element appea
 * one. Find that single one.
 *
 * Note:
 *
 * Your algorithm should have a linear runtime complexity.
 * it without using extra memory?
 *
 * Example 1:
 *
 *
 * Input: [2,2,1]
 * Output: 1
 *
 *
 * Example 2:
 *
 *
 * Input: [4,1,2,1,2]
 * Output: 4
 *
 *
 */
/**
 * @param {number[]} nums
 * @return {number}
 */
var singleNumber = function(nums) {
  let ret = 0;
  for (let index = 0; index < nums.length; index++) {
    const element = nums[index];
    ret = ret ^ element;
  }
  return ret;
};
```

C Code：

```
int singleNumber(int* nums, int numsSize){
    int res=0;
    for(int i=0;i<numsSize;i++)
    {
        res ^= nums[i];
    }

    return res;
}
```

C++ Code：

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        auto ret = 0;
        for (auto i : nums) ret ^= i;
        return ret;
    }
};

// C++ one-liner
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        return accumulate(nums.cbegin(), nums.cend(), 0, b
    }
};
```

Java Code：

```
class Solution {
    public int singleNumber(int[] nums) {
        int res = 0;
        for(int n:nums)
        {
            // 异或
            res ^= n;
        }
        return res;
    }
}
```

Python Code:

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        single_number = 0
        for num in nums:
            single_number ^= num
        return single_number
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

延伸

有一个 n 个元素的数组，除了两个数只出现一次外，其余元素都出现两次，让你找出这两个只出现一次的数分别是几，要求时间复杂度为 O(n) 且再开辟的内存空间固定(与 n 无关)。

和上面一样，只是这次不是一个数字，而是两个数字。还是按照上面的思路，我们进行一次全员异或操作， 得到的结果就是那两个只出现一次的不同的数字的异或结果。

我们刚才讲了异或的规律中有一个 `任何数和本身异或则为0` ， 因此我们的思路是能不能将这两个不同的数字分成两组 A 和 B。 分组需要满足两个条件.

1. 两个独特的的数字分成不同组
2. 相同的数字分成相同组

这样每一组的数据进行异或即可得到那两个数字。

问题的关键点是我们怎么进行分组呢?

由于异或的性质是，同一位相同则为 0，不同则为 1. 我们将所有数字异或的结果一定不是 0，也就是说至少有一位是 1.

我们随便取一个， 分组的依据就来了， 就是你取的那一位是 0 分成 1 组，那一位是 1 的分成一组。 这样肯定能保证 `2. 相同的数字分成相同组` ，不同的数字会被分成不同组么。 很明显当然可以， 因此我们选择是 1，也就是 说 `两个独特的的数字` 在那一位一定是不同的，因此两个独特元素一定会被分成不同组。

Done!

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址

https://leetcode.com/problems/min-stack/description/

题目描述

```
Design a stack that supports push, pop, top, and retrieving

push(x) -- Push element x onto stack.
pop() -- Removes the element on top of the stack.
top() -- Get the top element.
getMin() -- Retrieve the minimum element in the stack.
Example:
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> Returns -3.
minStack.pop();
minStack.top();      --> Returns 0.
minStack.getMin();   --> Returns -2.
```

前置知识

- 栈

公司

- amazon
- bloomberg
- google
- snapchat
- uber
- zenefits

两个栈

公司

- 阿里

- 腾讯
- 百度
- 字节

思路

我们使用两个栈：

- 一个栈存放全部的元素，push，pop都是正常操作这个正常栈。
- 另一个存放最小栈。 每次push，如果比最小栈的栈顶还小，我们就push进最小栈，否则不操作
- 每次pop的时候，我们都判断其是否和最小栈栈顶元素相同，如果相同，那么我们pop掉最小栈的栈顶元素即可

关键点

- 往minstack中 push的判断条件。 应该是stack为空或者x小于等于minstack栈顶元素

代码

- 语言支持：JS，C++，Java，Python

JavaScript Code:

```
/**
 * initialize your data structure here.
 */
var MinStack = function() {
    this.stack = []
    this.minStack = []
};

/**
 * @param {number} x
 * @return {void}
 */
MinStack.prototype.push = function(x) {
    this.stack.push(x)
    if (this.minStack.length == 0 ||  x <= this.minStack[th
        this.minStack.push(x)
    }
};

/**
 * @return {void}
 */
MinStack.prototype.pop = function() {
    const x = this.stack.pop()
    if (x !== void 0 &&  x === this.minStack[this.minStack.
        this.minStack.pop()
    }
};

/**
 * @return {number}
 */
MinStack.prototype.top = function() {
    return this.stack[this.stack.length - 1]
};

/**
 * @return {number}
 */
MinStack.prototype.min = function() {
    return this.minStack[this.minStack.length - 1]
};

/**
 * Your MinStack object will be instantiated and called as
 * var obj = new MinStack()
 * obj.push(x)
 * obj.pop()
```

```
 * var param_3 = obj.top()
 * var param_4 = obj.min()
 */
```

C++ Code:

```
class MinStack {
    stack<int> data;
    stack<int> helper;
public:
    /** initialize your data structure here. */
    MinStack() {

    }

    void push(int x) {
        data.push(x);
        if(helper.empty() || helper.top() >= x)
        {
            helper.push(x);
        }

    }

    void pop() {
        int top = data.top();
        data.pop();
        if(top == helper.top())
        {
            helper.pop();
        }

    }

    int top() {
        return data.top();
    }

    int getMin() {
        return helper.top();
    }
};

/**
 * Your MinStack object will be instantiated and called as
 * MinStack* obj = new MinStack();
 * obj->push(x);
 * obj->pop();
 * int param_3 = obj->top();
 * int param_4 = obj->getMin();
 */
```

Java Code:

```
public class MinStack {

    // 数据栈
    private Stack<Integer> data;
    // 辅助栈
    private Stack<Integer> helper;

    /**
     * initialize your data structure here.
     */
    public MinStack() {
        data = new Stack<>();
        helper = new Stack<>();
    }

    public void push(int x) {
        // 辅助栈在必要的时候才增加
        data.add(x);
        if (helper.isEmpty() || helper.peek() >= x) {
            helper.add(x);
        }
    }

    public void pop() {
        // 关键 3：data 一定得 pop()
        if (!data.isEmpty()) {
            // 注意：声明成 int 类型，这里完成了自动拆箱，从 Inte
            // 因此下面的比较可以使用 "==" 运算符
            int top = data.pop();
            if(top == helper.peek()){
                helper.pop();
            }
        }
    }

    public int top() {
        if(!data.isEmpty()){
            return data.peek();
        }
    }

    public int getMin() {
        if(!helper.isEmpty()){
            return helper.peek();
        }
    }
}
```

Python3 Code:

```
class MinStack:

    def __init__(self):
        """
        initialize your data structure here.
        """
        self.stack = []
        self.minstack = []

    def push(self, x: int) -> None:
        self.stack.append(x)
        if not self.minstack or x <= self.minstack[-1]:
            self.minstack.append(x)

    def pop(self) -> None:
        tmp = self.stack.pop()
        if tmp == self.minstack[-1]:
            self.minstack.pop()

    def top(self) -> int:
        return self.stack[-1]

    def min(self) -> int:
        return self.minstack[-1]


# Your MinStack object will be instantiated and called as s
# obj = MinStack()
# obj.push(x)
# obj.pop()
# param_3 = obj.top()
# param_4 = obj.min()
```

复杂度分析

- 时间复杂度：O(1)
- 空间复杂度：O(1)

一个栈

思路

符合直觉的方法是，每次对栈进行修改操作（push和pop）的时候更新最小值。 然后getMin只需要返回我们计算的最小值即可， top也是直接返回栈顶元素即可。 这种做法每次修改栈都需要更新最小值，因此时间复杂度是O(n).

是否有更高效的算法呢？答案是有的。

我们每次入栈的时候，保存的不再是真正的数字，而是它与当前最小值的差（当前元素没有入栈的时候的最小值）。 这样我们pop和top的时候拿到栈顶元素再加上**上一个**最小值即可。 另外我们在push和pop的时候去更新min，这样getMin的时候就简单了，直接返回min。

> 注意上面加粗的“上一个”，不是“当前的最小值”

经过上面的分析，问题的关键转化为“如何求得上一个最小值”，解决这个的关键点在于利用min。

pop或者top的时候：

- 如果栈顶元素小于0，说明栈顶是当前最小的元素，它出栈会对min造成影响，我们需要去更新min。 上一个最小的是“min - 栈顶元素”,我们需要将上一个最小值更新为当前的最小值

  > 因为栈顶元素入栈的时候的通过 `栈顶元素 = 真实值 - 上一个最小的元素` 得到的， 而真实值 = min， 因此可以得出 `上一个最小的元素 = 真实值 -栈顶元素`

- 如果栈顶元素大于0，说明它对最小值 `没有影响` ，上一个最小值就是上上个最小值。

关键点

- 最小栈存储的不应该是真实值，而是真实值和min的差值
- top的时候涉及到对数据的还原，这里千万注意是**上一个**最小值

代码

- 语言支持：JS，C++，Java，Python

Javascript Code:

```javascript
/*
 * @lc app=leetcode id=155 lang=javascript
 *
 * [155] Min Stack
 */
/**
 * initialize your data structure here.
 */
var MinStack = function() {
  this.stack = [];
  this.minV = Number.MAX_VALUE;
};

/**
 * @param {number} x
 * @return {void}
 */
MinStack.prototype.push = function(x) {
  // update 'min'
  const minV = this.minV;
  if (x < this.minV) {
    this.minV = x;
  }
  return this.stack.push(x - minV);
};

/**
 * @return {void}
 */
MinStack.prototype.pop = function() {
  const item = this.stack.pop();
  const minV = this.minV;

  if (item < 0) {
    this.minV = minV - item;
    return minV;
  }
  return item + minV;
};

/**
 * @return {number}
 */
MinStack.prototype.top = function() {
  const item = this.stack[this.stack.length - 1];
  const minV = this.minV;

  if (item < 0) {
```

```
    return minV;
  }
  return item + minV;
};

/**
 * @return {number}
 */
MinStack.prototype.min = function() {
  return this.minV;
};

/**
 * Your MinStack object will be instantiated and called as
 * var obj = new MinStack()
 * obj.push(x)
 * obj.pop()
 * var param_3 = obj.top()
 * var param_4 = obj.min()
 */
```

C++ Code:

```
class MinStack {
    stack<long> data;
    long min = INT_MAX;
public:
    /** initialize your data structure here. */
    MinStack() {

    }

    void push(int x) {
        data.push(x - min);
        if(x < min)
        {
            min = x;
        }

    }

    void pop() {
        long top = data.top();
        data.pop();
        // 更新最小值
        if(top < 0)
        {
            min -= top;
        }

    }

    int top() {
        long top = data.top();
        // 最小值为 min
        if (top < 0)
        {
            return min;
        }
        else{
            return min+top;
        }
    }

    int getMin() {
        return min;
    }
};

/**
 * Your MinStack object will be instantiated and called as
```

```
 * MinStack* obj = new MinStack();
 * obj->push(x);
 * obj->pop();
 * int param_3 = obj->top();
 * int param_4 = obj->getMin();
 */
```

Java Code:

```
class MinStack {
    long min;
    Stack<Long> stack;

    /** initialize your data structure here. */
    public MinStack() {
        stack = new Stack<>();
    }

    public void push(int x) {
        if (stack.isEmpty()) {
            stack.push(0L);
            min = x;
        }
        else {
            stack.push(x - min);
            if (x < min)
                min = x;
        }
    }

    public void pop() {
        long p = stack.pop();

        if (p < 0) {
            // if (p < 0), the popped value is the min
            // Recall p is added by this statement: stack.p
            // So, p = x - old_min
            // old_min = x - p
            // again, if (p < 0), x is the min so:
            // old_min = min - p
            min = min - p;
        }
    }

    public int top() {
        long p = stack.peek();

        if (p < 0) {
            return (int) min;
        }
        else {
            // p = x - min
            // x = p + min
            return (int) (p + min);
        }
    }
```

```
    public int getMin() {
        return (int) min;
    }
}
```

Python Code:

```python
class MinStack:

    def __init__(self):
        """
        initialize your data structure here.
        """
        self.minV = float('inf')
        self.stack = []

    def push(self, x: int) -> None:
        self.stack.append(x - self.minV)
        if x < self.minV:
            self.minV = x

    def pop(self) -> None:
        if not self.stack:
            return
        tmp = self.stack.pop()
        if tmp < 0:
            self.minV -= tmp

    def top(self) -> int:
        if not self.stack:
            return
        tmp = self.stack[-1]
        if tmp < 0:
            return self.minV
        else:
            return self.minV + tmp

    def min(self) -> int:
        return self.minV



# Your MinStack object will be instantiated and called as s
# obj = MinStack()
# obj.push(x)
# obj.pop()
# param_3 = obj.top()
# param_4 = obj.min()
```

复杂度分析

- 时间复杂度：O(1)
- 空间复杂度：O(1)

更多题解可以访问我的LeetCode题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址

https://leetcode.com/problems/two-sum-ii-input-array-is-sorted/description/

题目描述

这是leetcode头号题目 `two sum` 的第二个版本，难度简单。

```
Given an array of integers that is already sorted in ascend

The function twoSum should return indices of the two number

Note:

Your returned answers (both index1 and index2) are not zero
You may assume that each input would have exactly one solut
Example:

Input: numbers = [2,7,11,15], target = 9
Output: [1,2]
Explanation: The sum of 2 and 7 is 9. Therefore index1 = 1,
```

前置知识

- 双指针

公司

- 阿里
- 腾讯
- 百度
- 字节
- amazon

思路

由于题目没有对空间复杂度有求，用一个hashmap 存储已经访问过的数字即可。

假如题目空间复杂度有要求，由于数组是有序的，只需要双指针即可。一个left指针，一个right指针， 如果left + right 值 大于target 则 right左移动， 否则left右移，代码见下方python code。

> 如果数组无序，需要先排序（从这里也可以看出排序是多么重要的操作）

关键点解析

- 由于是有序的，因此双指针更好

代码

- 语言支持：JS，C++，Java，Python

Javascript Code:

```
/**
 * @param {number[]} numbers
 * @param {number} target
 * @return {number[]}
 */
var twoSum = function(numbers, target) {
    const visited = {} // 记录出现的数字，  空间复杂度N

    for (let index = 0; index < numbers.length; index++) {
        const element = numbers[index];
        if (visited[target - element] !== void 0) {
            return [visited[target - element], index + 1]
        }
        visited[element] = index + 1;
    }
    return [];
};
```

C++ Code:

```
class Solution {
public:
    vector<int> twoSum(vector<int>& numbers, int target) {
        int n = numbers.size();
        int left = 0;
        int right = n-1;
        while(left <= right)
        {
            if(numbers[left] + numbers[right] == target)
            {
                return {left + 1, right + 1};
            }
            else if (numbers[left] + numbers[right] > targe
            {
                right--;
            }
            else
            {
                left++;
            }
        }
        return {-1, -1};
    }
};
```

Java Code:

```
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        int n = numbers.length;
        int left = 0;
        int right = n-1;
        while(left <= right)
        {
            if(numbers[left] + numbers[right] == target)
            {
                return new int[]{left + 1, right + 1};
            }
            else if (numbers[left] + numbers[right] > targe
            {
                right--;
            }
            else
            {
                left++;
            }
        }

        return new int[]{-1, -1};
    }
}
```

Python Code:

```
class Solution:
    def twoSum(self, numbers: List[int], target: int) -> L
        visited = {}
        for index, number in enumerate(numbers):
            if target - number in visited:
                return [visited[target-number], index+1]
            else:
                visited[number] = index + 1

# 双指针思路实现
class Solution:
    def twoSum(self, numbers: List[int], target: int) -> L
        left, right = 0, len(numbers) - 1
        while left < right:
            if numbers[left] + numbers[right] < target:
                left += 1
            if numbers[left] + numbers[right] > target:
                right -= 1
            if numbers[left] + numbers[right] == target:
                return [left+1, right+1]
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经30K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode.com/problems/majority-element/description/

题目描述

```
Given an array of size n, find the majority element. The ma

You may assume that the array is non-empty and the majority

Example 1:

Input: [3,2,3]
Output: 3
Example 2:

Input: [2,2,1,1,1,2,2]
Output: 2
```

前置知识

- 投票算法

公司

- 阿里
- 腾讯
- 百度
- 字节
- adobe
- zenefits

思路

符合直觉的做法是利用额外的空间去记录每个元素出现的次数，并用一个单独的变量记录当前出现次数最多的元素。

但是这种做法空间复杂度较高，有没有可能进行优化呢？ 答案就是用"投票算法"。

投票算法的原理是通过不断消除不同元素直到没有不同元素，剩下的元素就是我们要找的元素。

关键点解析

- 投票算法

代码

- 语言支持：JS，Python

Javascript Code:

```
var majorityElement = function(nums) {
    let count = 1;
    let majority = nums[0];
    for(let i = 1; i < nums.length; i++) {
        if (count === 0) {
            majority = nums[i];
        }
        if (nums[i] === majority) {
            count ++;
        } else {
            count --;
        }
    }
    return majority;
};
```

Python Code:

```python
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        count, majority = 1, nums[0]
        for num in nums[1:]:
            if count == 0:
                majority = num
            if num == majority:
                count += 1
            else:
                count -= 1
        return majority
```

复杂度分析

- 时间复杂度：$O(N)$，其中N为数组长度
- 空间复杂度：$O(1)$

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode.com/problems/factorial-trailing-zeroes/description/

题目描述

```
Given an integer n, return the number of trailing zeroes in

Example 1:

Input: 3
Output: 0
Explanation: 3! = 6, no trailing zero.
Example 2:

Input: 5
Output: 1
Explanation: 5! = 120, one trailing zero.
Note: Your solution should be in logarithmic time complexit
```

前置知识

- 递归

公司

- 阿里
- 腾讯
- 百度
- bloomberg

思路

我们需要求解这n个数字相乘的结果末尾有多少个0，由于题目要求log的复杂度，因此暴力求解是不行的。

通过观察，我们发现如果想要结果末尾是0，必须是分解质因数之后，2 和 5 相乘才行，同时因数分解之后发现5的个数远小于2， 因此我们只需要求解这n数字分解质因数之后一共有多少个5即可.

如图如果n为30，那么结果应该是图中红色5的个数，即7。

我们的结果并不是直接f(n) = n / 5, 比如n为30， 25中是有两个5的。 类似，n为150，会有7个这样的数字，通过观察我们发现规律 `f(n) = n/5 + n/5^2 + n/5^3 + n/5^4 + n/5^5+..`

如果可以发现上面的规律，用递归还是循环实现这个算式就看你的了。

关键点解析

- 数论

代码

- 语言支持：JS，Python

Javascript Code:

```
/*
 * @lc app=leetcode id=172 lang=javascript
 *
 * [172] Factorial Trailing Zeroes
 */
/**
 * @param {number} n
 * @return {number}
 */
var trailingZeroes = function(n) {
  // tag: 数论

  // if (n === 0) return n;

  // 递归:  f(n) = n / 5 + f(n / 5)
  // return Math.floor(n / 5)  + trailingZeroes(Math.floor
  let count = 0;
  while (n >= 5) {
    count += Math.floor(n / 5);
    n = Math.floor(n / 5);
  }
  return count;
};
```

Python Code:

```python
class Solution:
    def trailingZeroes(self, n: int) -> int:
        count = 0
        while n >= 5:
            n = n // 5
            count += n
        return count


# 递归
class Solution:
    def trailingZeroes(self, n: int) -> int:
        if n == 0: return 0
        return n // 5 + self.trailingZeroes(n // 5)
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(1)$

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode.com/problems/reverse-bits/description/

题目描述

```
Reverse bits of a given 32 bits unsigned integer.



Example 1:

Input: 00000010100101000001111010011100
Output: 00111001011110000010100101000000
Explanation: The input binary string 00000010100101000001111
Example 2:

Input: 11111111111111111111111111111101
Output: 10111111111111111111111111111111
Explanation: The input binary string 11111111111111111111


Note:

Note that in some languages such as Java, there is no unsig
In Java, the compiler represents the signed integers using
```

前置知识

- 双指针

公司

- 阿里
- 腾讯
- 百度
- airbnb
- apple

思路

这道题是给定一个32位的无符号整型，让你按位翻转， 第一位变成最后一位， 第二位变成倒数第二位。。。

那么思路就是 `双指针`

> 这个指针可以加引号

- n从高位开始逐步左， res从低位（0）开始逐步右移
- 逐步判断，如果该位是1，就res + 1 , 如果是该位是0， 就res + 0
- 32位全部遍历完，则遍历结束

关键点解析

1. 可以用任何数字和1进行位运算的结果都取决于该数字最后一位的特性简化操作和提高性能

eg：

- n & 1 === 1, 说明n的最后一位是1
- n & 1 === 0, 说明n的最后一位是0

- 对于JS，ES规范在之前很多版本都是没有无符号整形的， 转化为无符号，可以用一个trick `n >>> 0`

- 双"指针" 模型

- bit 运算

代码

- 语言支持：JS，C++，Python

JavaScript Code：

```
/*
 * @lc app=leetcode id=190 lang=javascript
 *
 * [190] Reverse Bits
 *
 * https://leetcode.com/problems/reverse-bits/description/
 *
 * algorithms
 * Easy (30.30%)
 * Total Accepted:    173.7K
 * Total Submissions: 568.2K
 * Testcase Example:  '00000010100101000001111010011100'
 *
 * Reverse bits of a given 32 bits unsigned integer.
 *
 *
 *
 * Example 1:
 *
 *
 * Input: 00000010100101000001111010011100
 * Output: 00111001011110000010100101000000
 * Explanation: The input binary string 00000010100101000001
 * represents the unsigned integer 43261596, so return 964
 * binary representation is 00111001011110000010100101000000
 *
 *
 * Example 2:
 *
 *
 * Input: 11111111111111111111111111111101
 * Output: 10111111111111111111111111111111
 * Explanation: The input binary string 11111111111111111111
 * represents the unsigned integer 4294967293, so return 32
 * binary representation is 10111111110010110010011101101001
 *
 *
 *
 * Note:
 *
 *
 * Note that in some languages such as Java, there is no u
 * In this case, both input and output will be given as sig
 * should not affect your implementation, as the internal b
 * of the integer is the same whether it is signed or unsig
 * In Java, the compiler represents the signed integers usi
 * notation. Therefore, in Example 2 above the input repres
 * integer -3 and the output represents the signed integer
```

```
 *
 *
 *
 *
 * Follow up:
 *
 * If this function is called many times, how would you opt
 *
 */
/**
 * @param {number} n - a positive integer
 * @return {number} - a positive integer
 */
var reverseBits = function(n) {
  let res = 0;
  for (let i = 0; i < 32; i++) {
    res = (res << 1) + (n & 1);
    n = n >>> 1;
  }

  return res >>> 0;
};
```

C++ Code：

```
class Solution {
public:
    uint32_t reverseBits(uint32_t n) {
        auto ret = 0;
        for (auto i = 0; i < 32; ++i) {
            ret = (ret << 1) + (n & 1);
            n >>= 1;
        }
        return ret;
    }
};
```

Python Code:

```
class Solution:
    # @param n, an integer
    # @return an integer
    def reverseBits(self, n):
        result = 0
        for i in range(32):
            result = (result << 1) + (n & 1)
            n >>= 1
        return result
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(1)$

拓展

不使用迭代也可以完成相同的操作：

1. 两两相邻的1位对调
2. 两两相邻的2位对调
3. 两两相邻的4位对调
4. 两两相邻的8位对调
5. 两两相邻的16位对调

C++代码如下：

```
class Solution {
public:
    uint32_t reverseBits(uint32_t n) {
        auto ret = ((n & 0xaaaaaaaa) >> 1) | ((n & 0x55555
        ret = ((ret & 0xcccccccc) >> 2) | ((ret & 0x3333333
        ret = ((ret & 0xf0f0f0f0) >> 4) | ((ret & 0x0f0f0f0
        ret = ((ret & 0xff00ff00) >> 8) | ((ret & 0x00ff00
        return ((ret & 0xffff0000) >> 16) | ((ret & 0x0000
    }
};
```

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址

https://leetcode.com/problems/number-of-1-bits/description/

题目描述

```
Write a function that takes an unsigned integer and return




Example 1:

Input: 00000000000000000000000000001011
Output: 3
Explanation: The input binary string 00000000000000000000
Example 2:

Input: 00000000000000000000000010000000
Output: 1
Explanation: The input binary string 00000000000000000000
Example 3:

Input: 11111111111111111111111111111101
Output: 31
Explanation: The input binary string 11111111111111111111


Note:

Note that in some languages such as Java, there is no unsig
In Java, the compiler represents the signed integers using
```

前置知识

- 位运算

公司

- 阿里
- 腾讯
- 百度
- 字节
- apple

- microsoft

思路

这个题目的大意是： 给定一个无符号的整数， 返回其用二进制表式的时候的1的个数。

这里用一个trick， 可以轻松求出。 就是 `n & (n - 1)` 可以 `消除` n 最后的一个1的原理。

> 为什么能消除最后一个1， 其实也比较简单，大家自己想一下

这样我们可以不断进行 `n = n & (n - 1)` 直到n === 0 ， 说明没有一个1了。 这个时候 `我们消除了多少1变成一个1都没有了， 就说明n有多少个1了` 。

关键点解析

1. `n & (n - 1)` 可以 `消除` n 最后的一个1的原理 简化操作
2. bit 运算

代码

语言支持：JS, C++，Python

JavaScript Code:

```
/*
 * @lc app=leetcode id=191 lang=javascript
 *
 */
/**
 * @param {number} n - a positive integer
 * @return {number}
 */
var hammingWeight = function(n) {
  let count = 0;
  while (n !== 0) {
    n = n & (n - 1);
    count++;
  }

  return count;
};
```

C++ code:

```
class Solution {
public:
    int hammingWeight(uint32_t v) {
        auto count = 0;
        while (v != 0) {
            v &= (v - 1);
            ++count;
        }
        return count;
    }
};
```

Python Code:

```
class Solution(object):
    def hammingWeight(self, n):
        """
        :type n: int
        :rtype: int
        """
        count = 0
        while n:
            n &= n - 1
            count += 1
        return count
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(N)$

扩展

可以使用位操作来达到目的。例如8位的整数21:

C++ Code：

```
const uint32_t ODD_BIT_MASK = 0xAAAAAAAA;
const uint32_t EVEN_BIT_MASK = 0x55555555;
const uint32_t ODD_2BIT_MASK = 0xCCCCCCCC;
const uint32_t EVEN_2BIT_MASK = 0x33333333;
const uint32_t ODD_4BIT_MASK = 0xF0F0F0F0;
const uint32_t EVEN_4BIT_MASK = 0x0F0F0F0F;
const uint32_t ODD_8BIT_MASK = 0xFF00FF00;
const uint32_t EVEN_8BIT_MASK = 0x00FF00FF;
const uint32_t ODD_16BIT_MASK = 0xFFFF0000;
const uint32_t EVEN_16BIT_MASK = 0x0000FFFF;

class Solution {
public:

    int hammingWeight(uint32_t v) {
        v = (v & EVEN_BIT_MASK) + ((v & ODD_BIT_MASK) >> 1
        v = (v & EVEN_2BIT_MASK) + ((v & ODD_2BIT_MASK) >>
        v = (v & EVEN_4BIT_MASK) + ((v & ODD_4BIT_MASK) >>
        v = (v & EVEN_8BIT_MASK) + ((v & ODD_8BIT_MASK) >>
        return (v & EVEN_16BIT_MASK) + ((v & ODD_16BIT_MASK
    }
};
```

更多题解可以访问我的LeetCode题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址

https://leetcode.com/problems/house-robber/description/

题目描述

```
You are a professional robber planning to rob houses along

Given a list of non-negative integers representing the amou

Example 1:

Input: [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (
             Total amount you can rob = 1 + 3 = 4.
Example 2:

Input: [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money =
             Total amount you can rob = 2 + 9 + 1 = 12.
```

前置知识

- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节
- airbnb
- linkedin

思路

这是一道非常典型且简单的动态规划问题，但是在这里我希望通过这个例子， 让大家对动态规划问题有一点认识。

为什么别人的动态规划可以那么写，为什么没有用 dp 数组就搞定了。 比如别人的爬楼梯问题怎么就用 fibnacci 搞定了？为什么？在这里我们就来看下。

思路还是和其他简单的动态规划问题一样，我们本质上在解决 `对于第[i]个房子，我们抢还是不抢。` 的问题。

判断的标准就是总价值哪个更大， 那么对于抢的话 `就是当前的房子可以抢的价值 + dp[i - 2]`

> i - 1 不能抢，否则会触发警铃

如果不抢的话，就是 `dp[i - 1]` .

> 这里的 dp 其实就是 `子问题` .

状态转移方程也不难写 `dp[i] = Math.max(dp[i - 2] + nums[i - 2], dp[i - 1]);` （注：这里为了方便计算，令 `dp[0]` 和 `dp[1]` 都等于 0，所以 `dp[i]` 对应的是 `nums[i - 2]` ）

上述过程用图来表示的话，是这样的：

我们仔细观察的话，其实我们只需要保证前一个 dp[i - 1] 和 dp[i - 2] 两个变量就好了， 比如我们计算到 i = 6 的时候，即需要计算 dp[6]的时候，我们需要 dp[5], dp[4]，但是我们 不需要 dp[3], dp[2] ...

因此代码可以简化为：

```
let a = 0;
let b = 0;

for (let i = 0; i < nums.length; i++) {
  const temp = b;
  b = Math.max(a + nums[i], b);
  a = temp;
}

return b;
```

如上的代码，我们可以将空间复杂度进行优化，从 O(n)降低到 O(1), 类似的优化在 DP 问题中不在少数。

> 动态规划问题是递归问题查表，避免重复计算，从而节省时间。 如果我们对问题加以分析和抽象，有可能对空间上进一步优化

关键点解析

代码

- 语言支持：JS，C++，Python

JavaScript Code：

```
/*
 * @lc app=leetcode id=198 lang=javascript
 *
 * [198] House Robber
 *
 * https://leetcode.com/problems/house-robber/description/
 *
 * algorithms
 * Easy (40.80%)
 * Total Accepted:    312.1K
 * Total Submissions: 762.4K
 * Testcase Example:  '[1,2,3,1]'
 *
 * You are a professional robber planning to rob houses alo
 * house has a certain amount of money stashed, the only co
 * you from robbing each of them is that adjacent houses ha
 * connected and it will automatically contact the police i
 * houses were broken into on the same night.
 *
 * Given a list of non-negative integers representing the a
 * each house, determine the maximum amount of money you ca
 * without alerting the police.
 *
 * Example 1:
 *
 *
 * Input: [1,2,3,1]
 * Output: 4
 * Explanation: Rob house 1 (money = 1) and then rob house
 * 3).
 * Total amount you can rob = 1 + 3 = 4.
 *
 * Example 2:
 *
 *
 * Input: [2,7,9,3,1]
 * Output: 12
 * Explanation: Rob house 1 (money = 2), rob house 3 (money
 * 5 (money = 1).
 * Total amount you can rob = 2 + 9 + 1 = 12.
 *
 *
 */
/**
 * @param {number[]} nums
 * @return {number}
 */
var rob = function (nums) {
```

```
  // Tag: DP
  const dp = [];
  dp[0] = 0;
  dp[1] = 0;

  for (let i = 2; i < nums.length + 2; i++) {
    dp[i] = Math.max(dp[i - 2] + nums[i - 2], dp[i - 1]);
  }

  return dp[nums.length + 1];
};
```

C++ Code：

> 与 JavaScript 代码略有差异，但状态迁移方程是一样的。

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.empty()) return 0;
        auto sz = nums.size();
        if (sz == 1) return nums[0];
        auto prev = nums[0];
        auto cur = max(prev, nums[1]);
        for (auto i = 2; i < sz; ++i) {
            auto tmp = cur;
            cur = max(nums[i] + prev, cur);
            prev = tmp;
        }
        return cur;
    }
};
```

Python Code:

```python
class Solution:
    def rob(self, nums: List[int]) -> int:
        if not nums:
            return 0

        length = len(nums)
        if length == 1:
            return nums[0]
        else:
            prev = nums[0]
            cur = max(prev, nums[1])
            for i in range(2, length):
                cur, prev = max(prev + nums[i], cur), cur
            return cur
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

相关题目

- 337.house-robber-iii

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 35K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode.com/problems/remove-linked-list-elements/description/

题目描述

```
Remove all elements from a linked list of integers that hav

Example:

Input:  1->2->6->3->4->5->6, val = 6
Output: 1->2->3->4->5
```

前置知识

- 链表

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这个一个链表基本操作的题目，思路就不多说了。

关键点解析

- 链表的基本操作（删除指定节点）
- 虚拟节点 dummy 简化操作

> 其实设置 dummy 节点就是为了处理特殊位置（头节点），这这道题就是如果头节点是给定的需要删除的节点呢？ 为了保证代码逻辑的一致性，即不需要为头节点特殊定制逻辑，才采用的虚拟节点。

- 如果连续两个节点都是要删除的节点，这个情况容易被忽略。 eg:

```
// 只有下个节点不是要删除的节点才更新 current
if (!next || next.val !== val) {
    current =  next;
}
```

代码

- 语言支持：JS，Python

Javascript Code:

```
/**
 * @param {ListNode} head
 * @param {number} val
 * @return {ListNode}
 */
var removeElements = function(head, val) {
    const dummy = {
        next: head
    }
    let current = dummy;

    while(current && current.next) {
        let next = current.next;
        if (next.val === val) {
            current.next = next.next;
            next = next.next;
        }

        if (!next || next.val !== val) {
            current =  next;
        }
    }

    return dummy.next;
};
```

Python Code:

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def removeElements(self, head: ListNode, val: int) -> L
        prev = ListNode(0)
        prev.next = head
        cur = prev
        while cur.next:
            if cur.next.val == val:
                cur.next = cur.next.next
            else:
                cur = cur.next
        return prev.next
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode.com/problems/reverse-linked-list/description/

题目描述

Reverse a singly linked list.

Example:

Input: 1->2->3->4->5->NULL Output: 5->4->3->2->1->NULL Follow up:

A linked list can be reversed either iteratively or recursively. Could you implement both?

前置知识

- 链表

公司

- 阿里
- 百度
- 腾讯
- adobe
- amazon
- apple
- bloomberg
- facebook
- microsoft
- snapchat
- twitter
- uber
- yahoo
- yelp
- zenefits

思路

这个就是常规操作了，使用一个变量记录前驱 pre，一个变量记录后继 next.

不断更新 `current.next = pre` 就好了

关键点解析

- 链表的基本操作（交换）
- 虚拟节点 dummy 简化操作
- 注意更新 current 和 pre 的位置， 否则有可能出现溢出

代码

语言支持：JS, C++, Python,Java

JavaScript Code：

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */
var reverseList = function (head) {
  if (!head || !head.next) return head;

  let cur = head;
  let pre = null;

  while (cur) {
    const next = cur.next;
    cur.next = pre;
    pre = cur;
    cur = next;
  }

  return pre;
};
```

C++ Code：

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = NULL;
        ListNode* cur = head;
        ListNode* next = NULL;
        while (cur != NULL) {
            next = cur->next;
            cur->next = prev;
            prev = cur;
            cur = next;
        }
        return prev;
    }
};
```

Python Code:

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        if not head: return None
        prev = None
        cur = head
        while cur:
            cur.next, prev, cur = prev, cur, cur.next
        return prev
```

Java Code:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode pre = null, cur = head;

        while (cur != null) {
            ListNode next = cur.next;
            cur.next = pre;
            pre = cur;
            cur = next;
        }

        return pre;
    }
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

拓展

通过单链表的定义可以得知，单链表也是递归结构，因此，也可以使用递归的方式来进行 reverse 操作。

> 由于单链表是线性的，使用递归方式将导致栈的使用也是线性的，当链表长度达到一定程度时，递归会导致爆栈，因此，现实中并不推荐使用递归方式来操作链表。

1. 除第一个节点外，递归将链表 reverse
2. 将第一个节点添加到已 reverse 的链表之后

> 这里需要注意的是，每次需要保存已经 reverse 的链表的头节点和尾节点

C++实现

```
// 普通递归
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* tail = nullptr;
        return reverseRecursive(head, tail);
    }

    ListNode* reverseRecursive(ListNode *head, ListNode *&t
        if (head == nullptr) {
            tail = nullptr;
            return head;
        }
        if (head->next == nullptr) {
            tail = head;
            return head;
        }
        auto h = reverseRecursive(head->next, tail);
        if (tail != nullptr) {
            tail->next = head;
            tail = head;
            head->next = nullptr;
        }
        return h;
    }
};

//  (类似) 尾递归
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if (head == nullptr) return head;
        return reverseRecursive(nullptr, head, head->next);
    }

    ListNode* reverseRecursive(ListNode *prev, ListNode *he
    {
        if (next == nullptr) return head;
        auto n = next->next;
        next->next = head;
        head->next = prev;
        return reverseRecursive(head, next, n);
    }
};
```

JavaScript 实现

```
var reverseList = function (head) {
  // 递归结束条件
  if (head === null || head.next === null) {
    return head;
  }

  // 递归反转 子链表
  let newReverseList = reverseList(head.next);
  // 获取原来链表的第 2 个节点 newReverseListTail
  let newReverseListTail = head.next;
  // 调整原来头结点和第 2 个节点的指向
  newReverseListTail.next = head;
  head.next = null;

  // 将调整后的链表返回
  return newReverseList;
};
```

Python 实现

```
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        if not head or not head.next: return head
        ans = self.reverseList(head.next)
        head.next.next = head
        head.next = None
        return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

相关题目

- 92.reverse-linked-list-ii
- 25.reverse-nodes-in-k-groups

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址

https://leetcode.com/problems/contains-duplicate-ii/description/

题目描述

```
Given an array of integers and an integer k, find out whet

Example 1:

Input: nums = [1,2,3,1], k = 3
Output: true
Example 2:

Input: nums = [1,0,1,1], k = 1
Output: true
Example 3:

Input: nums = [1,2,3,1,2,3], k = 2
Output: false
```

前置知识

- hashmap

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

由于题目没有对空间复杂度有求，用一个hashmap 存储已经访问过的数字即可，每次访问都会看hashmap中是否有这个元素，有的话拿出索引进行比对，是否满足条件（相隔不大于k），如果满足返回true即可。

公司

- airbnb

- palantir

关键点解析

- 空间换时间

代码

- 语言支持：JS，Python，C++

Javascript Code:

```
/*
 * @lc app=leetcode id=219 lang=javascript
 *
 * [219] Contains Duplicate II
 *
 * https://leetcode.com/problems/contains-duplicate-ii/desc
 *
 * algorithms
 * Easy (34.75%)
 * Total Accepted:    187.3K
 * Total Submissions: 537.5K
 * Testcase Example:  '[1,2,3,1]\n3'
 *
 * Given an array of integers and an integer k, find out wh
 * distinct indices i and j in the array such that nums[i]
 * absolute difference between i and j is at most k.
 *
 *
 * Example 1:
 *
 *
 * Input: nums = [1,2,3,1], k = 3
 * Output: true
 *
 *
 *
 * Example 2:
 *
 *
 * Input: nums = [1,0,1,1], k = 1
 * Output: true
 *
 *
 *
 * Example 3:
 *
 *
 * Input: nums = [1,2,3,1,2,3], k = 2
 * Output: false
 *
 *
 *
 *
 *
 */
/**
 * @param {number[]} nums
 * @param {number} k
```

```
 * @return {boolean}
 */
var containsNearbyDuplicate = function(nums, k) {
    const visited = {};
    for(let i = 0; i < nums.length; i++) {
        const num = nums[i];
        if (visited[num] !== undefined && i - visited[num]
            return true;
        }
        visited[num] = i;
    }
    return false
};
```

Python Code:

```
class Solution:
    def containsNearbyDuplicate(self, nums: List[int], k: 
        d = {}
        for index, num in enumerate(nums):
            if num in d and index - d[num] <= k:
                return True
            d[num] = index
        return False
```

C++ Code：

```
class Solution {
public:
    bool containsNearbyDuplicate(vector<int>& nums, int k)
        auto m = unordered_map<int, int>();
        for (int i = 0; i < nums.size(); ++i) {
            auto iter = m.find(nums[i]);
            if (iter != m.end()) {
                if (i - m[nums[i]] <= k) {
                    return true;
                }
            }
            m[nums[i]] = i;
        }
        return false;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$

- 空间复杂度：$O(N)$

更多题解可以访问我的LeetCode题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode.com/problems/invert-binary-tree/description/

题目描述

```
Invert a binary tree.

Example:

Input:

     4
   /   \
  2     7
 / \   / \
1   3 6   9
Output:

     4
   /   \
  7     2
 / \   / \
9   6 3   1
Trivia:
This problem was inspired by this original tweet by Max Hov

Google: 90% of our engineers use the software you wrote (Ho
```

前置知识

- 递归

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

遍历树（随便怎么遍历），然后将左右子树交换位置。

关键点解析

- 递归简化操作
- 如果树很高，建议使用栈来代替递归
- 这道题目对顺序没要求的，因此队列数组操作都是一样的，无任何区别

代码

- 语言支持：JS，Python，C++

Javascript Code:

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {TreeNode}
 */
var invertTree = function(root) {
  if (!root) return root;
  // 递归
  //   const left = root.left;
  //   const right = root.right;
  //   root.right = invertTree(left);
  //   root.left = invertTree(right);
  // 我们用stack来模拟递归
  // 本质上递归是利用了执行栈，执行栈也是一种栈
  // 其实这里使用队列也是一样的，因为这里顺序不重要

  const stack = [root];
  let current = null;
  while ((current = stack.shift())) {
    const left = current.left;
    const right = current.right;
    current.right = left;
    current.left = right;
    if (left) {
      stack.push(left);
    }
    if (right) {
      stack.push(right);
    }
  }
  return root;
};
```

Python Code:

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def invertTree(self, root: TreeNode) -> TreeNode:
        if not root:
            return None
        stack = [root]
        while stack:
            node = stack.pop(0)
            node.left, node.right = node.right, node.left
            if node.left:
                stack.append(node.left)
            if node.right:
                stack.append(node.right)
        return root
```

C++ Code：

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {
 * };
 */
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if (root == NULL) return root;
        auto q = queue<TreeNode*>();
        q.push(root);
        while (!q.empty()) {
            auto n = q.front(); q.pop();
            swap(n->left, n->right);
            if (n->left != nullptr) {
                q.push(n->left);
            }
            if (n->right != nullptr) {
                q.push(n->right);
            }
        }
        return root;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址

https://leetcode.com/problems/implement-queue-using-stacks/description/

题目描述

```
Implement the following operations of a queue using stacks.

push(x) -- Push element x to the back of queue.
pop() -- Removes the element from in front of queue.
peek() -- Get the front element.
empty() -- Return whether the queue is empty.
Example:

MyQueue queue = new MyQueue();

queue.push(1);
queue.push(2);
queue.peek();  // returns 1
queue.pop();   // returns 1
queue.empty(); // returns false
Notes:

You must use only standard operations of a stack -- which r
Depending on your language, stack may not be supported nati
You may assume that all operations are valid (for example,
```

前置知识

- 栈

公司

- 阿里
- 腾讯
- 百度
- 字节
- bloomberg
- microsoft

思路

这道题目是让我们用栈来模拟实现队列。 我们知道栈和队列都是一种受限的数据结构。 栈的特点是只能在一端进行所有操作，队列的特点是只能在一端入队，另一端出队。

在这里我们可以借助另外一个栈，也就是说用两个栈来实现队列的效果。这种做法的时间复杂度和空间复杂度都是O(n)。

由于栈只能操作一端，因此我们peek或者pop的时候也只去操作顶部元素，要达到目的 我们需要在push的时候将队头的元素放到栈顶即可。

因此我们只需要在push的时候，用一下辅助栈即可。 具体做法是先将栈清空并依次放到另一个辅助栈中，辅助栈中的元素再次放回栈中，最后将新的元素push进去即可。

比如我们现在栈中已经是1，2，3，4了。 我们现在要push一个5.

push之前是这样的：

然后我们将栈中的元素转移到辅助栈：

最后将新的元素添加到栈顶。

整个过程是这样的：

关键点解析

- 在push的时候利用辅助栈(双栈)

代码

- 语言支持：JS, Python, Java

Javascript Code:

```
/*
 * @lc app=leetcode id=232 lang=javascript
 *
 * [232] Implement Queue using Stacks
 */
/**
 * Initialize your data structure here.
 */
var MyQueue = function() {
  // tag: queue stack array
  this.stack = [];
  this.helperStack = [];
};

/**
 * Push element x to the back of queue.
 * @param {number} x
 * @return {void}
 */
MyQueue.prototype.push = function(x) {
  let cur = null;
  while ((cur = this.stack.pop())) {
    this.helperStack.push(cur);
  }
  this.helperStack.push(x);

  while ((cur = this.helperStack.pop())) {
    this.stack.push(cur);
  }
};

/**
 * Removes the element from in front of queue and returns t
 * @return {number}
 */
MyQueue.prototype.pop = function() {
  return this.stack.pop();
};

/**
 * Get the front element.
 * @return {number}
 */
MyQueue.prototype.peek = function() {
  return this.stack[this.stack.length - 1];
};

/**
```

```
 * Returns whether the queue is empty.
 * @return {boolean}
 */
MyQueue.prototype.empty = function() {
  return this.stack.length === 0;
};

/**
 * Your MyQueue object will be instantiated and called as s
 * var obj = new MyQueue()
 * obj.push(x)
 * var param_2 = obj.pop()
 * var param_3 = obj.peek()
 * var param_4 = obj.empty()
 */
```

Python Code:

```python
class MyQueue:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.stack = []
        self.help_stack = []

    def push(self, x: int) -> None:
        """
        Push element x to the back of queue.
        """
        while self.stack:
            self.help_stack.append(self.stack.pop())
        self.help_stack.append(x)
        while self.help_stack:
            self.stack.append(self.help_stack.pop())

    def pop(self) -> int:
        """
        Removes the element from in front of queue and retu
        """
        return self.stack.pop()

    def peek(self) -> int:
        """
        Get the front element.
        """
        return self.stack[-1]

    def empty(self) -> bool:
        """
        Returns whether the queue is empty.
        """
        return not bool(self.stack)


# Your MyQueue object will be instantiated and called as su
# obj = MyQueue()
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.peek()
# param_4 = obj.empty()
```

Java Code

```
class MyQueue {
    Stack<Integer> pushStack = new Stack<> ();
    Stack<Integer> popStack = new Stack<> ();

    /** Initialize your data structure here. */
    public MyQueue() {

    }

    /** Push element x to the back of queue. */
    public void push(int x) {
        while (!popStack.isEmpty()) {
            pushStack.push(popStack.pop());
        }
        pushStack.push(x);
    }

    /** Removes the element from in front of queue and retu
    public int pop() {
        while (!pushStack.isEmpty()) {
            popStack.push(pushStack.pop());
        }
        return popStack.pop();
    }

    /** Get the front element. */
    public int peek() {
        while (!pushStack.isEmpty()) {
            popStack.push(pushStack.pop());
        }
        return popStack.peek();
    }

    /** Returns whether the queue is empty. */
    public boolean empty() {
        return pushStack.isEmpty() && popStack.isEmpty();
    }
}

/**
 * Your MyQueue object will be instantiated and called as s
 * MyQueue obj = new MyQueue();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.peek();
 * boolean param_4 = obj.empty();
 */
`
```

复杂度分析

- 时间复杂度：$O(1)$
- 空间复杂度：$O(1)$

扩展

- 类似的题目有用队列实现栈，思路是完全一样的，大家有兴趣可以试一下。
- 栈混洗也是借助另外一个栈来完成的，从这点来看，两者有相似之处。

延伸阅读

实际上现实中也有使用两个栈来实现队列的情况，那么为什么我们要用两个stack来实现一个queue?

其实使用两个栈来替代一个队列的实现是为了在多进程中分开对同一个队列对读写操作。一个栈是用来读的，另一个是用来写的。当且仅当读栈满时或者写栈为空时，读写操作才会发生冲突。

当只有一个线程对栈进行读写操作的时候，总有一个栈是空的。在多线程应用中，如果我们只有一个队列，为了线程安全，我们在读或者写队列的时候都需要锁住整个队列。而在两个栈的实现中，只要写入栈不为空，那么 `push` 操作的锁就不会影响到 `pop` 。

- reference
- further reading

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址

https://leetcode.com/problems/ugly-number/description/

题目描述

```
Write a program to check whether a given number is an ugly

Ugly numbers are positive numbers whose prime factors only

Example 1:

Input: 6
Output: true
Explanation: 6 = 2 × 3
Example 2:

Input: 8
Output: true
Explanation: 8 = 2 × 2 × 2
Example 3:

Input: 14
Output: false
Explanation: 14 is not ugly since it includes another prime
Note:

1 is typically treated as an ugly number.
Input is within the 32-bit signed integer range: [−231,  23
```

前置知识

- 数学
- 因数分解

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

题目要求给定一个数字，判断是否为“丑陋数”(ugly number), 丑陋数是指只包含质因子2, 3, 5的正整数。

根据定义，我们将给定数字除以2、3、5(顺序无所谓)，直到无法整除。如果得到1，说明是所有因子都是2或3或5，如果不是1，则不是丑陋数。

这就好像我们判断一个数字是否为n(n为大于1的正整数)的幂次方一样，我们只需要 不断除以n，直到无法整除，如果得到1，那么就是n的幂次方。 这道题的不同在于 它不再是某一个数字的幂次方，而是三个数字（2，3，5），不过解题思路还是一样的。

转化为代码可以是：

```
while(num % 2 === 0)   num = num / 2;
while(num % 3 === 0)   num = num / 3;
while(num % 5 === 0)   num = num / 5;

return num === 1;
```

> 我下方给出的代码是用了递归实现，只是给大家看下不同的写法而已。

关键点

- 数论
- 因数分解

代码

- 语言支持：JS, C++, Java, Python

Javascript Code:

```
/*
 * @lc app=leetcode id=263 lang=javascript
 *
 * [263] Ugly Number
 */
/**
 * @param {number} num
 * @return {boolean}
 */
var isUgly = function(num) {
  // TAG: 数论
  if (num <= 0) return false;
  if (num === 1) return true;

  const list = [2, 3, 5];

  if (list.includes(num)) return true;

  for (let i of list) {
    if (num % i === 0) return isUgly(Math.floor(num / i));
  }
  return false;
};
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(logN)$

C++ Code:

```
class Solution {
public:
    bool isUgly(int num) {
        int ugly[] = {2,3,5};
        for(int u : ugly)
        {
            while(num%u==0 && num%u < num)
            {
                num/=u;
            }
        }
        return num == 1;
    }
};
```

Java Code:

```
class Solution {
    public boolean isUgly(int num) {
        int [] ugly = {2,3,5};
        for(int u : ugly)
        {
            while(num%u==0 && num%u < num)
            {
                num/=u;
            }
        }
        return num == 1;
    }
}
```

Python Code:

```
# 非递归写法
class Solution:
    def isUgly(self, num: int) -> bool:
        if num <= 0:
            return False
        for i in (2, 3, 5):
            while num % i == 0:
                num /= i
        return num == 1
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(1)$

更多题解可以访问我的LeetCode题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址

https://leetcode.com/problems/move-zeroes/description/

题目描述

```
Given an array nums, write a function to move all 0's to th

Example:

Input: [0,1,0,3,12]
Output: [1,3,12,0,0]
Note:

You must do this in-place without making a copy of the arra
Minimize the total number of operations.
```

前置知识

- 数组
- 双指针

公司

- 阿里
- 腾讯
- 百度
- 字节
- bloomberg
- facebook

思路

如果题目没有要求 modify in-place 的话，我们可以先遍历一遍将包含 0 的和不包含 0 的存到两个数组， 然后拼接两个数组即可。 但是题目要求 modify in-place， 也就是不需要借助额外的存储空间，刚才的方法 空间复杂度是 O(n).

那么如果 modify in-place 呢？ 空间复杂度降低为 1。

我们可以借助一个游标记录位置，然后遍历一次，将非 0 的原地修改，最后补 0 即可。

关键点解析

- 双指针

代码

- 语言支持：JS, C++, Java，Python

JavaScript Code:

```
/**
 * @param {number[]} nums
 * @return {void} Do not return anything, modify nums in-p
 */
var moveZeroes = function(nums) {
    let index = 0;
    for(let i = 0; i < nums.length; i++) {
        const num = nums[i];
        if (num !== 0) {
            nums[index++] = num;
        }
    }

    for(let i = index; i < nums.length; i++) {
        nums[index++] = 0;
    }
};
```

C++ Code：

> 解题思想与上面 JavaScript 一致，做了少许代码优化（非性能优化，因为时间复杂度都是 O(n)）： 增加一个游标来记录下一个待处理的元素的位置，这样只需要写一次循环即可。

```
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        vector<int>::size_type nonZero = 0;
        vector<int>::size_type next = 0;
        while (next < nums.size()) {
            if (nums[next] != 0) {
                // 使用 std::swap() 会带来 8ms 的性能损失
                // swap(nums[next], nums[nonZero]);
                auto tmp = nums[next];
                nums[next] = nums[nonZero];
                nums[nonZero] = tmp;
                ++nonZero;
            }
            ++next;
        }
    }
};
```

Java Code:

```
class Solution {
    public void moveZeroes(int[] nums) {
        // 双指针
        int i = 0;
        for(int j=0; j<nums.length; j++)
        {
            // 不为0，前移
            if(nums[j] != 0)
            {
                int temp = nums[i];
                nums[i] = nums[j];
                nums[j] = temp;
                i++;
            }
        }
    }
}
```

Python Code:

```
class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instea
        """
        slow = fast = 0
        while fast < len(nums):
            if nums[fast] != 0:
                nums[fast], nums[slow] = nums[slow], nums[f
                slow += 1
            fast += 1
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode.com/problems/power-of-four/description/

题目描述

```
Given an integer (signed 32 bits), write a function to chec

Example 1:

Input: 16
Output: true
Example 2:

Input: 5
Output: false
Follow up: Could you solve it without loops/recursion?
```

前置知识

- 数论

公司

- 百度
- twosigma

思路

符合直觉的做法是不停除以 4 直到不能整除，然后判断是否为 1 即可。代码如下：

```
while (num && num % 4 == 0) {
  num /= 4;
}
return num == 1;
```

但是这道题目有一个 follow up: “你是否可以不使用循环/递归完成”。因此我们需要换种思路。

我们先来看下，4 的幂次方用 2 进制表示是什么样的.

发现规律： 4 的幂次方的二进制表示 1 的位置都是在奇数位（且不在最低位），其他位置都为 0

我们还可以发现： 2 的幂次方的特点是最低位之外，其他位置有且仅有一个 1（1 可以在任意位置）

我们进一步分析，如果一个数字是四的幂次方，那么只需要满足：

1. 是 2 的幂次方， 就能保证最低位之外，其他位置有且仅有一个 1
2. 这个 1 不在偶数位置，一定在奇数位置

对于第一点，如果保证一个数字是 2 的幂次方呢？ 显然不能不停除以 2，看结果是否等于 1，这样就循环了。 我们可以使用一个 trick， 如果一个数字 n 是 2 的幂次方，那么 n & (n - 1) 一定等于 0， 这个可以作为思考题，大家思考一下。

对于第二点，我们可以取一个特殊数字，这个特殊数字，奇数位置都是 1，偶数位置都是 0，然后和这个特殊数字 `求与` ， 如果等于本身，那么毫无疑问，这个 1 不再偶数位置，一定在奇数位置，因为如果在偶数位置， `求与` 的结果就是 0 了 题目要求 n 是 32 位有符号整形，那么我们的特殊数字就应该是 `01010101010101010101010101010101` (不用数了，一共 32 位)。

如上图，64和这个特殊数字求与，得到的是本身。 8 是 2的次方，但是不是4的次方，我们求与结果就是0了。

为了体现自己的逼格，我们可以使用计算器，来找一个逼格比较高的数字，这里我选了十六进制，结果是 `0x55555555` 。

代码见下方代码区。

说实话，这种做法不容易想到，其实还有一种方法。 如果一个数字是 4 的幂次方，那么只需要满足：

1. 是二的倍数
2. 减去 1 是三的倍数

代码如下：

```
return num > 0 && (num & (num - 1)) === 0 && (num - 1) % 3
```

关键点

- 数论
- 2的幂次方特点（数学性质以及二进制表示）
- 4的幂次方特点（数学性质以及二进制表示）

代码

语言支持：JS, Python

JavaScript Code：

```
/*
 * @lc app=leetcode id=342 lang=javascript
 *
 * [342] Power of Four
 */
/**
 * @param {number} num
 * @return {boolean}
 */
var isPowerOfFour = function(num) {
  // tag: 数论

  if (num === 1) return true;
  if (num < 4) return false;

  if ((num & (num - 1)) !== 0) return false;

  return (num & 0x55555555) === num;
};
```

Python Code:

```
class Solution:
    def isPowerOfFour(self, num: int) -> bool:
        if num == 1:
            return True
        elif num < 4:
            return False
        else:
            if not num & (num-1) == 0:
                return False
            else:
                return num & 0x55555555 == num

    # 另一种解法：将数字转化为二进制表示的字符串，利用字符串的相关操作
    def isPowerOfFour(self, num: int) -> bool:
        binary_num = bin(num)[2:]
        return binary_num.strip('0') == '1' and len(binary_
```

复杂度分析

- 时间复杂度：$O(1)$
- 空间复杂度：$O(1)$

更多题解可以访问我的LeetCode题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode.com/problems/intersection-of-two-arrays/description/

题目描述

```
Given two arrays, write a function to compute their interse

Example 1:

Input: nums1 = [1,2,2,1], nums2 = [2,2]
Output: [2]
Example 2:

Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
Output: [9,4]
Note:

Each element in the result must be unique.
The result can be in any order.
```

前置知识

- hashtable

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

先遍历第一个数组，将其存到hashtable中， 然后遍历第二个数组，如果在hashtable中存在就push到return，然后清空hashtable即可。

关键点解析

- 空间换时间

代码

- 语言支持：JS, Python

Javascript Code:

```
/**
 * @param {number[]} nums1
 * @param {number[]} nums2
 * @return {number[]}
 */
var intersection = function(nums1, nums2) {
    const visited = {};
    const ret = [];
    for(let i = 0; i < nums1.length; i++) {
        const num = nums1[i];

        visited[num] = num;
    }

    for(let i = 0; i < nums2.length; i++) {
        const num = nums2[i];

        if (visited[num] !== undefined) {
            ret.push(num);
            visited[num] = undefined;
        }
    }

    return ret;

};
```

Python Code:

```
class Solution:
    def intersection(self, nums1: List[int], nums2: List[in
        visited, result = {}, []
        for num in nums1:
            visited[num] = num
        for num in nums2:
            if num in visited:
                result.append(num)
                visited.pop(num)
        return result

    # 另一种解法：利用 Python 中的集合进行计算
    def intersection(self, nums1: List[int], nums2: List[in
        return set(nums1) & set(nums2)
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

更多题解可以访问我的LeetCode题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode.com/problems/sum-of-two-integers/description/

题目描述

```
Calculate the sum of two integers a and b, but you are not

Example 1:

Input: a = 1, b = 2
Output: 3
Example 2:

Input: a = -2, b = 3
Output: 1
```

前置知识

- 位运算

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

不能使用加减法来求加法。 我们只能朝着位元算的角度来思考了。

由于 `异或` 是 `相同则位0，不同则位1` ，因此我们可以把异或看成是一种不进位的加减法。

由于 `与` 是 `全部位1则位1，否则位0` ，因此我们可以求与之后左移一位来表示进位。

然后我们对上述两个元算结果递归求解即可。 递归的结束条件就是其中一个为0，我们直接返回另一个。

关键点解析

- 位运算
- 异或是一种不进位的加减法
- 求与之后左移一位来可以表示进位

代码

代码支持：JS，C++，Java，Python
Javascript Code:

```
/*
 * @lc app=leetcode id=371 lang=javascript
 *
 * [371] Sum of Two Integers
 */
/**
 * @param {number} a
 * @param {number} b
 * @return {number}
 */
var getSum = function(a, b) {
    if (a === 0) return b;

    if (b === 0) return a;

    return getSum(a ^ b, (a & b) << 1);
};
```

C++ Code:

```
class Solution {
public:
    int getSum(int a, int b) {
        if(a==0) return b;
        if(b==0) return a;

        while(b!=0)
        {
            // 防止 AddressSanitizer 对有符号左移的溢出保护处理
            auto carry = ((unsigned int ) (a & b))<<1;
            // 计算无进位的结果
            a = a^b;
            //将存在进位的位置置1
            b =carry;
        }
        return a;
    }
};
```

Java Code:

```
class Solution {
    public int getSum(int a, int b) {
        if(a==0) return b;
        if(b==0) return a;

        while(b!=0)
        {
            int carry = a&b;
            // 计算无进位的结果
            a = a^b;
            //将存在进位的位置置1
            b =carry<<1;
        }
        return a;
    }
}
```

Python Code:

```python
# python整数类型为Unifying Long Integers, 即无限长整数类型.
# 模拟 32bit 有符号整型加法
class Solution:
    def getSum(self, a: int, b: int) -> int:
        a &= 0xFFFFFFFF
        b &= 0xFFFFFFFF
        while b:
            carry = a & b
            a ^= b
            b = ((carry) << 1) & 0xFFFFFFFF
            # print((a, b))
        return a if a < 0x80000000 else ~(a^0xFFFFFFFF)
```

复杂度分析

- 时间复杂度：$O(1)$
- 空间复杂度：$O(1)$

> 由于题目数据规模不会变化，因此其实复杂度分析是没有意义的。

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode.com/problems/path-sum-iii/description/

题目描述

```
You are given a binary tree in which each node contains an

Find the number of paths that sum to a given value.

The path does not need to start or end at the root or a lea

The tree has no more than 1,000 nodes and the values are in

Example:

root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

      10
     /  \
    5   -3
   / \    \
  3   2   11
 / \   \
3  -2   1

Return 3. The paths that sum to 8 are:

1.  5 -> 3
2.  5 -> 2 -> 1
3. -3 -> 11
```

前置知识

- hashmap

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是要我们求解出任何一个节点出发到子孙节点的路径中和为指定值。 注意这里，不一定是从根节点出发，也不一定在叶子节点结束。

一种简单的思路就是直接递归解决，空间复杂度O(n) 时间复杂度介于 O(nlogn) 和 O(n^2)， 具体代码：

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
// the number of the paths starting from self
function helper(root, sum) {
  if (root === null) return 0;
  const l = helper(root.left, sum - root.val);
  const r = helper(root.right, sum - root.val);

  return l + r + (root.val === sum ? 1 : 0);
}
/**
 * @param {TreeNode} root
 * @param {number} sum
 * @return {number}
 */
var pathSum = function(root, sum) {
// 空间复杂度O(n) 时间复杂度介于O(nlogn) 和 O(n^2)
  // tag: dfs tree
  if (root === null) return 0;
  // the number of the paths starting from self
  const self = helper(root, sum);
  // we don't know the answer, so we just pass it down
  const l = pathSum(root.left, sum);
  // we don't know the answer, so we just pass it down
  const r = pathSum(root.right, sum);

  return self + l + r;
};
```

但是还有一种空间复杂度更加优秀的算法，利用hashmap来避免重复计算，时间复杂度和空间复杂度都是O(n)。 这种思路是 `subarray-sum-equals-k` 的升级版本，如果那道题目你可以O(n)解决，这道题目难度就不会很大， 只是将数组换成了二叉树。关于具体的思路可以看这道题目

这里有一个不一样的地方，这里我说明一下，就是为什么要有 `hashmap[acc] = hashmap[acc] - 1;` ， 原因很简单，就是我们DFS的时候，从底部往上回溯的时候，map的值应该也回溯。如果你对回溯法比较熟悉的话， 应该很容易理解，如果不熟悉可以参考这道题目，这道题目就是通过 `tempList.pop()` 来完成的。

另外我画了一个图，相信看完你就明白了。

当我们执行到底部的时候：

接着往上回溯：

很容易看出，我们的hashmap不应该有第一张图的那个记录了，因此需要减去。

具体实现见下方代码区。

关键点解析

- 通过hashmap，以时间换空间
- 对于这种连续的元素求和问题，有一个共同的思路，可以参考这道题目

代码

- 语言支持：JS

```
/*
 * @lc app=leetcode id=437 lang=javascript
 *
 * [437] Path Sum III
 */
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
function helper(root, acc, target, hashmap) {
  // see also : https://leetcode.com/problems/subarray-sum-

  if (root === null) return 0;
  let count = 0;
  acc += root.val;
  if (acc === target) count++;
  if (hashmap[acc - target] !== void 0) {
    count += hashmap[acc - target];
  }
  if (hashmap[acc] === void 0) {
    hashmap[acc] = 1;
  } else {
    hashmap[acc] += 1;
  }
  const res =
    count +
    helper(root.left, acc, target, hashmap) +
    helper(root.right, acc, target, hashmap);

    // 这里要注意别忘记了
    hashmap[acc] = hashmap[acc] - 1;

    return res;
}

var pathSum = function(root, sum) {
  const hashmap = {};
  return helper(root, 0, sum, hashmap);
};
```

复杂度分析

- 时间复杂度：$O(N)$

- 空间复杂度：$O(N)$

更多题解可以访问我的LeetCode题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode-cn.com/problems/assign-cookies

题目描述

```
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

注意：

你可以假设胃口值为正。
一个小朋友最多只能拥有一块饼干。

示例 1:

输入: [1,2,3], [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: [1,2], [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.
```

前置知识

- 贪心算法
- 双指针

公司

- 阿里
- 腾讯
- 字节

思路

贪心算法+双指针求解

给一个孩子的饼干应当尽量小并且能满足孩子，大的留来满足胃口大的孩子。因为胃口小的孩子最容易得到满足，所以优先满足胃口小的孩子需求。按照从小到大的顺序使用饼干尝试是否可满足某个孩子。

关键点

将需求因子 g 和 s 分别从小到大进行排序，使用贪心思想，配合双指针，每个饼干只尝试一次，成功则换下一个孩子来尝试。

代码

- 语言支持：JS

```
/**
 * @param {number[]} g
 * @param {number[]} s
 * @return {number}
 */
const findContentChildren = function (g, s) {
    g = g.sort((a, b) => a - b);
    s = s.sort((a, b) => a - b);
    let gi = 0; // 胃口值
    let sj = 0; // 饼干尺寸
    let res = 0;
    while (gi < g.length && sj < s.length) {
        // 当饼干 sj >= 胃口 gi 时，饼干满足胃口，更新满足的孩子数
        if (s[sj] >= g[gi]) {
            gi++;
            sj++;
            res++;
        } else {
            // 当饼干 sj < 胃口 gi 时，饼干不能满足胃口，需要换大
            sj++;
        }
    }
    return res;
};
```

复杂度分析

- 时间复杂度：由于使用了排序，因此时间复杂度为 O(NlogN)
- 空间复杂度：O(1)

更多题解可以访问我的LeetCode题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode.com/problems/distribute-candies/description/

题目描述

```
Given an integer array with even length, where different nu
Example 1:
Input: candies = [1,1,2,2,3,3]
Output: 3
Explanation:
There are three different kinds of candies (1, 2 and 3), ar
Optimal distribution: The sister has candies [1,2,3] and th
The sister has three different kinds of candies.
Example 2:
Input: candies = [1,1,2,3]
Output: 2
Explanation: For example, the sister has candies [2,3] and
The sister has two different kinds of candies, the brother
Note:

The length of the given array is in range [2, 10,000], and
The number in given array is in range [-100,000, 100,000].
```

前置知识

- 数组

公司

- 阿里
- 字节

思路

由于糖果是偶数，并且我们只需要做到两个人糖果数量一样即可。

考虑两种情况:

- 如果糖果种类大于n / 2（糖果种类数为n），妹妹最多可以获得的糖果种类应该是 `n / 2` (因为妹妹只有n / 2个糖).
- 糖果种类数小于n / 2, 妹妹能够得到的糖果种类可以是糖果的种类数（糖果种类本身就这么多）.

因此我们发现，妹妹能够获得的糖果种类的制约因素其实是糖果种类数。

关键点解析

- 这是一道逻辑题目，因此如果逻辑分析清楚了，代码是自然而然的

代码

- 语言支持：JS, Python

Javascript Code:

```
/*
 * @lc app=leetcode id=575 lang=javascript
 *
 * [575] Distribute Candies
 */
/**
 * @param {number[]} candies
 * @return {number}
 */
var distributeCandies = function(candies) {
    const count = new Set(candies);
    return Math.min(count.size, candies.length >> 1);
};
```

Python Code:

```
class Solution:
    def distributeCandies(self, candies: List[int]) -> int:
        return min(len(set(candies)), len(candies) >> 1)
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

更多题解可以访问我的LeetCode题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址（874. 模拟行走机器人）

https://leetcode-cn.com/problems/walking-robot-simulation/submissions/

题目描述

```
机器人在一个无限大小的网格上行走，从点 (0, 0) 处开始出发，面向北方。i

-2：向左转 90 度
-1：向右转 90 度
1 <= x <= 9：向前移动 x 个单位长度
在网格上有一些格子被视为障碍物。

第 i 个障碍物位于网格点  (obstacles[i][0], obstacles[i][1])

如果机器人试图走到障碍物上方，那么它将停留在障碍物的前一个网格方块上，

返回从原点到机器人的最大欧式距离的平方。



示例 1：

输入: commands = [4,-1,3], obstacles = []
输出: 25
解释: 机器人将会到达 (3, 4)
示例 2：

输入: commands = [4,-1,4,-2,4], obstacles = [[2,4]]
输出: 65
解释: 机器人在左转走到 (1, 8) 之前将被困在 (1, 4) 处


提示：

0 <= commands.length <= 10000
0 <= obstacles.length <= 10000
-30000 <= obstacle[i][0] <= 30000
-30000 <= obstacle[i][1] <= 30000
答案保证小于 2 ^ 31
```

前置知识

- hashtable

公司

- 暂无

思路

这道题之所以是简单难度，是因为其没有什么技巧。你只需要看懂题目描述，然后把题目描述转化为代码即可。

唯一需要注意的是查找障碍物的时候如果你采用的是 `线形查找` 会很慢，很可能会超时。

> 我实际测试了一下，确实会超时

- 一种方式是使用排序，然后二分查找，如果采用基于比较的排序算法，那么这种算法的瓶颈在于排序本身，也就是$O(NlogN)$。
- 另一种方式是使用集合，将 obstacles 放入集合，然后需要的时候进行查询，查询的时候的时间复杂度为$O(1)$。

这里我们采用第二种方式。

接下来我们来“翻译”一下题目。

- 由于机器人只能往前走。因此机器人往东西南北哪个方向走取决于它的 `朝向` 。
- 我们使用枚举来表示当前机器人的 `朝向` 。
- 题目只有两种方式改变 `朝向` ，一种是左转（-2），另一种是右转（-1）。
- 题目要求的是机器人在 `运动过程中距离原点的最大值` ，而不是最终位置距离原点的距离。

为了代码书写简单，我建立了一个直角坐标系。用 `机器人的朝向和 x 轴正方向的夹角度数` 来作为枚举值，并且这个度数是 `0 <= deg < 360` 。我们不难知道，其实这个取值就是 `0` ，`90` ，`180` ，`270` 四个值。那么当 0 度的时候，我们只需要不断地 x+1，90 度的时候我们不断地 y + 1 等等。

关键点解析

- 理解题意，这道题容易理解错题意，求解为 `最终位置距离原点的距离`
- 建立坐标系
- 空间换时间

代码

代码支持： Python3

Python3 Code:

```
class Solution:
    def robotSim(self, commands: List[int], obstacles: List
        pos = [0, 0]
        deg = 90
        ans = 0
        obstaclesSet = set(map(tuple, obstacles))

        for command in commands:
            if command == -1:
                deg = (deg + 270) % 360
            elif command == -2:
                deg = (deg + 90) % 360
            else:
                if deg == 0:
                    i = 0
                    while i < command and not (pos[0] + 1,
                        pos[0] += 1
                        i += 1
                if deg == 90:
                    i = 0
                    while i < command and not (pos[0], pos
                        pos[1] += 1
                        i += 1
                if deg == 180:
                    i = 0
                    while i < command and not (pos[0] - 1,
                        pos[0] -= 1
                        i += 1
                if deg == 270:
                    i = 0
                    while i < command and not (pos[0], pos
                        pos[1] -= 1
                        i += 1
                ans = max(ans, pos[0] ** 2 + pos[1] ** 2)
        return ans
```

复杂度分析

- 时间复杂度：$O(N*M)$, 其中 N 为 commands 的长度， M 为 commands 数组的平均值。
- 空间复杂度：$O(obstacles)$

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址（1260. 二维网格迁移）

https://leetcode-cn.com/problems/shift-2d-grid/description/

题目描述

```
给你一个 n 行 m 列的二维网格 grid 和一个整数 k。你需要将 grid 迁移

每次「迁移」操作将会引发下述活动：

位于 grid[i][j] 的元素将会移动到 grid[i][j + 1]。
位于 grid[i][m - 1] 的元素将会移动到 grid[i + 1][0]。
位于 grid[n - 1][m - 1] 的元素将会移动到 grid[0][0]。
请你返回 k 次迁移操作后最终得到的 二维网格。



示例 1：



输入：grid = [[1,2,3],[4,5,6],[7,8,9]], k = 1
输出：[[9,1,2],[3,4,5],[6,7,8]]
示例 2：



输入：grid = [[3,8,1,9],[19,7,2,5],[4,6,11,10],[12,0,21,13]]
输出：[[12,0,21,13],[3,8,1,9],[19,7,2,5],[4,6,11,10]]
示例 3：

输入：grid = [[1,2,3],[4,5,6],[7,8,9]], k = 9
输出：[[1,2,3],[4,5,6],[7,8,9]]


提示：

1 <= grid.length <= 50
1 <= grid[i].length <= 50
-1000 <= grid[i][j] <= 1000
0 <= k <= 100
```

前置知识

- 数组
- 数学

暴力法

公司

- 字节

思路

我们直接翻译题目，没有任何 hack 的做法。

代码

```
from copy import deepcopy

class Solution:
    def shiftGrid(self, grid: List[List[int]], k: int) -> L
        n = len(grid)
        m = len(grid[0])
        for _ in range(k):
            old = deepcopy(grid)
            for i in range(n):
                for j in range(m):
                    if j == m - 1:
                            grid[(i + 1) % n][0] = old[i][j
                    elif i == n - 1 and j == m - 1:
                        grid[0][0] = old[i][j]
                    else:
                        grid[i][j + 1] = old[i][j]
        return grid
```

由于是 easy，上述做法勉强可以过，我们考虑优化。

数学分析

思路

我们仔细观察矩阵会发现，其实这样的矩阵迁移是有规律的。 如图：

因此这个问题就转化为我们一直的一维矩阵转移问题，LeetCode 也有原题189. 旋转数组，同时我也写了一篇文章文科生都能看懂的循环移位算法专门讨论这个，最终我们使用的是三次旋转法，相关数学证明也有写，很详细，这里不再赘述。

LeetCode 真的是喜欢换汤不换药呀 😂

代码

Python 代码：

```python
#
# @lc app=leetcode.cn id=1260 lang=python3
#
# [1260] 二维网格迁移
#

# @lc code=start


class Solution:
    def shiftGrid(self, grid: List[List[int]], k: int) -> L
        n = len(grid)
        m = len(grid[0])
        # 二维到一维
        arr = [grid[i][j] for i in range(n) for j in range(
        # 取模，缩小k的范围，避免无意义的运算
        k %= m * n
        res = []
        # 首尾交换法

        def reverse(l, r):
            while l < r:
                t = arr[l]
                arr[l] = arr[r]
                arr[r] = t
                l += 1
                r -= 1
        # 三次旋转
        reverse(0, m * n - k - 1)
        reverse(m * n - k, m * n - 1)
        reverse(0, m * n - 1)
        # 一维到二维
        row = []
        for i in range(m * n):
            if i > 0 and i % m == 0:
                res.append(row)
                row = []
            row.append(arr[i])
        res.append(row)

        return res

# @lc code=end
```

复杂度分析

- 时间复杂度：$O(N)$

- 空间复杂度：$O(1)$

相关题目

- 189. 旋转数组

参考

- 文科生都能看懂的循环移位算法

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址（1332. 删除回文子序列）

https://leetcode-cn.com/problems/remove-palindromic-subsequences/

题目描述

```
给你一个字符串 s，它仅由字母 'a' 和 'b' 组成。每一次删除操作都可以从

返回删除给定字符串中所有字符（字符串为空）的最小删除次数。

「子序列」定义：如果一个字符串可以通过删除原字符串某些字符而不改变原字

「回文」定义：如果一个字符串向后和向前读是一致的，那么这个字符串就是一



示例 1：

输入：s = "ababa"
输出：1
解释：字符串本身就是回文序列，只需要删除一次。
示例 2：

输入：s = "abb"
输出：2
解释："abb" -> "bb" -> "".
先删除回文子序列 "a"，然后再删除 "bb"。
示例 3：

输入：s = "baabb"
输出：2
解释："baabb" -> "b" -> "".
先删除回文子序列 "baab"，然后再删除 "b"。
示例 4：

输入：s = ""
输出：0


提示：

0 <= s.length <= 1000
s 仅包含字母 'a'  和 'b'
在真实的面试中遇到过这道题?
```

前置知识

- 回文

公司

- 暂无

思路

这又是一道“抖机灵”的题目，类似的题目有1297.maximum-number-of-occurrences-of-a-substring

由于只有 a 和 b 两个字符。其实最多的消除次数就是 2。因为我们无论如何都可以先消除全部的 1 再消除全部的 2（先消除 2 也一样），这样只需要两次即可完成。 我们再看一下题目给的一次消除的情况，题目给的例子是“ababa”，我们发现其实它本身就是一个回文串，所以才可以一次全部消除。那么思路就有了：

- 如果 s 是回文，则我们需要一次消除
- 否则需要两次
- 一定要注意特殊情况， 对于空字符串，我们需要 0 次

关键点解析

- 注意审题目，一定要利用题目条件“只含有 a 和 b 两个字符”否则容易做的很麻烦

代码

代码支持：Python3

Python3 Code:

```python
class Solution:
    def removePalindromeSub(self, s: str) -> int:
        if s == '':
            return 0
        def isPalindrome(s):
            l = 0
            r = len(s) - 1
            while l < r:
                if s[l] != s[r]:
                    return False
                l += 1
                r -= 1
            return True
        return 1 if isPalindrome(s) else 2
```

如果你觉得判断回文不是本题重点，也可以简单实现：

Python3 Code:

```python
class Solution:
    def removePalindromeSub(self, s: str) -> int:
        if s == '':
            return 0
        return 1 if s == s[::-1] else 2
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

中等难度题目合集

中等题目是力扣比例最大的部分，因此这部分我的题解也是最多的。 大家不要太过追求难题，先把中等难度题目做熟了再说。

这部分的题目要不需要我们挖掘题目的内含信息， 将其抽象成简单题目。 要么是一些写起来比较麻烦的题目， 一些人编码能力不行就挂了。 因此大家一定要自己做， 即使看了题解”会了“，也要自己码一遍。自己不亲自写一遍，里面的细节永远不知道。

- 面试题 17.09. 第 k 个数
- 0002.add-two-numbers
- 0003.longest-substring-without-repeating-characters
- 0005.longest-palindromic-substring
- 0011.container-with-most-water
- 0015.3-sum
- 0017.Letter-Combinations-of-a-Phone-Number 🆕
- 0019. Remove Nth Node From End of List
- 0022.generate-parentheses.md 🆕
- 0024. Swap Nodes In Pairs
- 0029.divide-two-integers
- 0031.next-permutation
- 0033.search-in-rotated-sorted-array
- 0039.combination-sum
- 0040.combination-sum-ii
- 0046.permutations
- 0047.permutations-ii
- 0048.rotate-image
- 0049.group-anagrams
- 0050.pow-x-n 🆕
- 0055.jump-game
- 0056.merge-intervals
- 0060.permutation-sequence 🆕
- 0062.unique-paths 🖊️
- 0073.set-matrix-zeroes
- 0075.sort-colors
- 0078.subsets
- 0079.word-search
- 0080.remove-duplicates-from-sorted-array-ii 🆕
- 0086.partition-list
- 0090.subsets-ii
- 0091.decode-ways

- 0092.reverse-linked-list-ii 🖊
- 0094.binary-tree-inorder-traversal
- 0095.unique-binary-search-trees-ii 🆕
- 0096.unique-binary-search-trees 🆕
- 0098.validate-binary-search-tree
- 0102.binary-tree-level-order-traversal
- 0103.binary-tree-zigzag-level-order-traversal
- 105.Construct-Binary-Tree-from-Preorder-and-Inorder-Traversal.md
- 0113.path-sum-ii
- 0129.sum-root-to-leaf-numbers
- 0130.surrounded-regions
- 0131.palindrome-partitioning
- 0139.word-break
- 0144.binary-tree-preorder-traversal
- 0150.evaluate-reverse-polish-notation
- 0152.maximum-product-subarray 🖊
- 0199.binary-tree-right-side-view
- 0200.number-of-islands 🆕
- 0201.bitwise-and-of-numbers-range 🖊
- 0208.implement-trie-prefix-tree
- 0209.minimum-size-subarray-sum
- 0211.add-and-search-word-data-structure-design 🆕
- 0215.kth-largest-element-in-an-array 🆕
- 0221.maximal-square
- 0229.majority-element-ii 🆕
- 0230.kth-smallest-element-in-a-bst
- 0236.lowest-common-ancestor-of-a-binary-tree
- 0238.product-of-array-except-self
- 0240.search-a-2-d-matrix-ii
- 0279.perfect-squares
- 0309.best-time-to-buy-and-sell-stock-with-cooldown
- 0322.coin-change
- 0328.odd-even-linked-list
- 0334.increasing-triplet-subsequence
- 0337.house-robber-iii.md
- 0343.integer-break🆕
- 0365.water-and-jug-problem
- 0378.kth-smallest-element-in-a-sorted-matrix
- 0380.insert-delete-getrandom-o1
- 0416.partition-equal-subset-sum🖊
- 0445.add-two-numbers-ii
- 0454.4-sum-ii
- 0474.ones-and-zeros
- 0494.target-sum

- 0516.longest-palindromic-subsequence
- 0518.coin-change-2
- 0547.friend-circles
- 0560.subarray-sum-equals-k
- 0609.find-duplicate-file-in-system
- 0611.valid-triangle-number
- 0718.maximum-length-of-repeated-subarray
- 0754.reach-a-number.md 🆕
- 0785.is-graph-bipartite 🆕
- 0820.short-encoding-of-words 🖊
- 0875.koko-eating-bananas
- 0877.stone-game
- 0886.possible-bipartition 🆕
- 0900.rle-iterator
- 0912.sort-an-array
- 0935.knight-dialer 🆕
- 1011.capacity-to-ship-packages-within-d-days
- 1014.best-sightseeing-pair 🆕
- 1015.smallest-integer-divisible-by-k
- 1019.next-greater-node-in-linked-list 🆕
- 1020.number-of-enclaves
- 1023.camelcase-matching
- 1031.maximum-sum-of-two-non-overlapping-subarrays
- 1104.path-in-zigzag-labelled-binary-tree
- 1131.maximum-of-absolute-value-expression
- 1186.maximum-subarray-sum-with-one-deletion 🆕
- 1218.longest-arithmetic-subsequence-of-given-difference
- 1227.airplane-seat-assignment-probability
- 1261.find-elements-in-a-contaminated-binary-tree
- 1262.greatest-sum-divisible-by-three
- 1297.maximum-number-of-occurrences-of-a-substring
- 1310.xor-queries-of-a-subarray 🆕
- 1334.find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance 🆕
- 1371.find-the-longest-substring-containing-vowels-in-even-counts 🆕

题目地址（面试题 17.09. 第 k 个数）

https://leetcode-cn.com/problems/get-kth-magic-number-lcci/

题目描述

```
有些数的素因子只有 3，5，7，请设计一个算法找出第 k 个数。注意，不是必

示例 1:

输入: k = 5

输出: 9
```

前置知识

- 堆
- 状态机
- 动态规划

公司

- 暂无

堆

思路

思路很简单。 就是使用一个小顶堆，然后每次从小顶堆取一个， 取 k 次即可。

唯一需要注意的是重复数字，比如 `3 * 5 = 15` ， `5 * 3 = 15` 。关于去重， 用 set 就对了。

关键点

- 去重

代码(Python)

```
from heapq import heappop, heappush
class Solution:
    def getKthMagicNumber(self, k: int) -> int:
        heap = [1]
        numbers = set()
        # 每次从小顶堆取一个， 取 k 次即可
        while k:
            cur = heappop(heap)
            if cur not in numbers:
                k -= 1
                heappush(heap, cur * 3)
                heappush(heap, cur * 5)
                heappush(heap, cur * 7)
            numbers.add(cur)
        return cur
```

状态机

思路

这个思路力扣的题目还是蛮多的， 比如西法我之前写的一个 《原来状态机也可以用来刷 LeetCode? 》.

思路很简单， 就是用三个指针记录因子是 3，5，7 的最小值的索引。

代码（Python）

```
class Solution:
    def getKthMagicNumber(self, k: int) -> int:
        p3 = p5 = p7 = 0
        state = [1] + [0] * (k - 1)

        for i in range(1, k):
            state[i] = min(state[p3] * 3, state[p5] * 5, st
            if 3 * state[p3] == state[i]: p3 += 1
            if 5 * state[p5] == state[i]: p5 += 1
            if 7 * state[p7] == state[i]: p7 += 1
        return state[-1]
```

复杂度分析

- 时间复杂度：$O(k)$
- 空间复杂度：$O(k)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 36K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址

https://leetcode.com/problems/add-two-numbers/description/

题目描述

```
You are given two non-empty linked lists representing two r

You may assume the two numbers do not contain any leading z

Example

Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)
Output: 7 -> 0 -> 8
Explanation: 342 + 465 = 807.
```

前置知识

- 链表

公司

- 阿里
- 百度
- 腾讯

思路

设立一个表示进位的变量 carried，建立一个新链表， 把输入的两个链表从头往后同时处理，每两个相加，将结果加上 carried 后的值作为一个新节点到新链表后面。

(图片来自： https://github.com/MisterBooo/LeetCodeAnimation)

关键点解析

1. 链表这种数据结构的特点和使用
2. 用一个 carried 变量来实现进位的功能，每次相加之后计算 carried，并用于下一位的计算

代码

- 语言支持：JS，C++

JavaScript:

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} l1
 * @param {ListNode} l2
 * @return {ListNode}
 */
var addTwoNumbers = function (l1, l2) {
  if (l1 === null || l2 === null) return null;

  // 使用dummyHead可以简化对链表的处理，dummyHead.next指向新链表
  let dummyHead = new ListNode(0);
  let cur1 = l1;
  let cur2 = l2;
  let cur = dummyHead; // cur用于计算新链表
  let carry = 0; // 进位标志

  while (cur1 !== null || cur2 !== null) {
    let val1 = cur1 !== null ? cur1.val : 0;
    let val2 = cur2 !== null ? cur2.val : 0;
    let sum = val1 + val2 + carry;
    let newNode = new ListNode(sum % 10); // sum%10取模结果
    carry = sum >= 10 ? 1 : 0; // sum>=10，carry=1，表示有进
    cur.next = newNode;
    cur = cur.next;

    if (cur1 !== null) {
      cur1 = cur1.next;
    }

    if (cur2 !== null) {
      cur2 = cur2.next;
    }
  }

  if (carry > 0) {
    // 如果最后还有进位，新加一个节点
    cur.next = new ListNode(carry);
  }

  return dummyHead.next;
};
```

C++

> C++代码与上面的 JavaScript 代码略有不同：将 carry 是否为 0 的判断放到了 while 循环中

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* ret = nullptr;
        ListNode* cur = nullptr;
        int carry = 0;
        while (l1 != nullptr || l2 != nullptr || carry != 0
            carry += (l1 == nullptr ? 0 : l1->val) + (l2 ==
            auto temp = new ListNode(carry % 10);
            carry /= 10;
            if (ret == nullptr) {
                ret = temp;
                cur = ret;
            }
            else {
                cur->next = temp;
                cur = cur->next;
            }
            l1 = l1 == nullptr ? nullptr : l1->next;
            l2 = l2 == nullptr ? nullptr : l2->next;
        }
        return ret;
    }
};
```

拓展

通过单链表的定义可以得知，单链表也是递归结构，因此，也可以使用递归的方式来进行 reverse 操作。

> 由于单链表是线性的，使用递归方式将导致栈的使用也是线性的，当链表长度达到一定程度时，递归会导致爆栈，因此，现实中并不推荐使用递归方式来操作链表。

描述

1. 将两个链表的第一个节点值相加，结果转为 0-10 之间的个位数，并设置进位信息
2. 将两个链表第一个节点以后的链表做带进位的递归相加
3. 将第一步得到的头节点的 next 指向第二步返回的链表

C++实现

```
// 普通递归
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        return addTwoNumbers(l1, l2, 0);
    }

private:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2, int
        if (l1 == nullptr && l2 == nullptr && carry == 0) r
        carry += (l1 == nullptr ? 0 : l1->val) + (l2 == nu
        auto ret = new ListNode(carry % 10);
        ret->next = addTwoNumbers(l1 == nullptr ? l1 : l1-
                                  l2 == nullptr ? l2 : l2->r
                                  carry / 10);
        return ret;
    }
};
//  (类似) 尾递归
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* head = nullptr;
        addTwoNumbers(head, nullptr, l1, l2, 0);
        return head;
    }

private:
    void addTwoNumbers(ListNode*& head, ListNode* cur, List
        if (l1 == nullptr && l2 == nullptr && carry == 0) r
        carry += (l1 == nullptr ? 0 : l1->val) + (l2 == nu
        auto temp = new ListNode(carry % 10);
        if (cur == nullptr) {
            head = temp;
            cur = head;
        } else {
            cur->next = temp;
            cur = cur->next;
        }
        addTwoNumbers(head, cur, l1 == nullptr ? l1 : l1->r
    }
};
```

题目地址

https://leetcode.com/problems/longest-substring-without-repeating-characters/description/

题目描述

Given a string, find the length of the longest substring without repeating characters.

Examples:

```
Given "abcabcbb", the answer is "abc", which the length is

Given "bbbbb", the answer is "b", with the length of 1.

Given "pwwkew", the answer is "wke", with the length of 3.
```

前置知识

- 哈希表
- 滑动窗口

公司

- 阿里
- 字节
- 腾讯

思路

用一个 hashmap 来建立字符和其出现位置之间的映射。同时维护一个滑动窗口，窗口内的都是没有重复的字符，去尽可能的扩大窗口的大小，窗口不停的向右滑动。

（1）如果当前遍历到的字符从未出现过，那么直接扩大右边界；

（2）如果当前遍历到的字符出现过，则缩小窗口（左边索引向右移动），然后继续观察当前遍历到的字符；

（3）重复（1）（2），直到左边索引无法再移动；

（4）维护一个结果 res，每次用出现过的窗口大小来更新结果 res，最后返回 res 获取结果。

(图片来自： https://github.com/MisterBooo/LeetCodeAnimation)

关键点

1. 用一个 mapper 记录出现过并且没有被删除的字符
2. 用一个滑动窗口记录当前 index 开始的最大的不重复的字符序列
3. 用 res 去记录目前位置最大的长度，每次滑动窗口更新就去决定是否需要更新 res

代码

代码支持：Python3

Python3 Code:

```python
from collections import defaultdict


class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        l = 0
        ans = 0
        counter = defaultdict(lambda: 0)

        for r in range(len(s)):
            while counter.get(s[r], 0) != 0:
                counter[s[l]] = counter.get(s[l], 0) - 1
                l += 1
            counter[s[r]] += 1
            ans = max(ans, r - l + 1)

        return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

大家也可以关注我的公众号《力扣加加》获取更多更新鲜的 LeetCode 题解

题目地址（5. 最长回文子串）

https://leetcode-cn.com/problems/longest-palindromic-substring/

题目描述

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

示例 1：

输入: "babad" 输出: "bab" 注意: "aba" 也是一个有效答案。 示例 2：

输入: "cbbd" 输出: "bb"

前置知识

- 回文

公司

- 阿里
- 百度
- 腾讯

思路

这是一道最长回文的题目，要我们求出给定字符串的最大回文子串。

解决这类问题的核心思想就是两个字“延伸”，具体来说

- 如果在一个不是回文字符串的字符串两端添加任何字符，或者在回文串左右分别加不同的字符，得到的一定不是回文串
- 如果一个字符串不是回文串，或者在回文串左右分别加不同的字符，得到的一定不是回文串

事实上，上面的分析已经建立了大问题和小问题之间的关联， 基于此，我们可以建立动态规划模型。

我们可以用 dp[i][j] 表示 s 中从 i 到 j（包括 i 和 j）是否可以形成回文， 状态转移方程只是将上面的描述转化为代码即可：

```
if (s[i] === s[j] && dp[i + 1][j - 1]) {
  dp[i][j] = true;
}
```

base case 就是一个字符（轴对称点是本身），或者两个字符（轴对称点是介于两者之间的虚拟点）。

关键点

- ”延伸“（extend）

代码

代码支持：Python，JavaScript：

Python Code：

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        n = len(s)
        if n == 0:
            return ""
        res = s[0]
        def extend(i, j, s):
            while(i >= 0 and j < len(s) and s[i] == s[j]):
                i -= 1
                j += 1
            return s[i + 1:j]

        for i in range(n - 1):
            e1 = extend(i, i, s)
            e2 = extend(i, i + 1, s)
            if max(len(e1), len(e2)) > len(res):
                res = e1 if len(e1) > len(e2) else e2
        return res
```

JavaScript Code：

```
/*
 * @lc app=leetcode id=5 lang=javascript
 *
 * [5] Longest Palindromic Substring
 */
/**
 * @param {string} s
 * @return {string}
 */
var longestPalindrome = function (s) {
  // babad
  // tag : dp
  if (!s || s.length === 0) return "";
  let res = s[0];

  const dp = [];

  // 倒着遍历简化操作， 这么做的原因是dp[i][..]依赖于dp[i + 1][..
  for (let i = s.length - 1; i >= 0; i--) {
    dp[i] = [];
    for (let j = i; j < s.length; j++) {
      if (j - i === 0) dp[i][j] = true;
      // specail case 1
      else if (j - i === 1 && s[i] === s[j]) dp[i][j] = tru
      // specail case 2
      else if (s[i] === s[j] && dp[i + 1][j - 1]) {
        // state transition
        dp[i][j] = true;
      }

      if (dp[i][j] && j - i + 1 > res.length) {
        // update res
        res = s.slice(i, j + 1);
      }
    }
  }

  return res;
};
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N^2)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

大家也可以关注我的公众号《脑洞前端》获取更多更新鲜的 LeetCode 题解

相关题目

- 516.longest-palindromic-subsequence

题目地址

https://leetcode-cn.com/problems/container-with-most-water/description/

题目描述

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

前置知识

- 双指针

公司

- 字节
- 腾讯
- 百度
- 阿里

思路

题目中说 `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。` ，因此符合直觉的解法就是固定两个端点，计算可以承载的水量，然后不断更新最大值，最后返回最大值即可。这种算法，需要两层循环，时间复杂度是 $O(n^2)$。

代码（JS）：

```
let max = 0;
for (let i = 0; i < height.length; i++) {
  for (let j = i + 1; j < height.length; j++) {
    const currentArea = Math.abs(i - j) * Math.min(height[
    if (currentArea > max) {
      max = currentArea;
    }
  }
}
return max;
```

虽然解法效率不高，但是可以通过（JS可以通过，Python 不可以，其他语言没有尝试）。那么有没有更优的解法呢？

我们来换个角度来思考这个问题，上述的解法是通过两两组合，这无疑是完备的。我们换个角度思考，是否可以先计算长度为 n 的面积，然后计算长度为 n-1 的面积，... 计算长度为 1 的面积。 这样去不断更新最大值呢？很显然这种解法也是完备的，但是似乎时间复杂度还是 $O(n^2)$, 不要着急。

考虑一下，如果我们计算 n-1 长度的面积的时候,是可以直接排除一半的结果的。

如图：

比如我们计算 n 面积的时候，假如左侧的线段高度比右侧的高度低，那么我们通过左移右指针来将长度缩短为 n-1 的做法是没有意义的， 因为 `新的形成的面积变成了(n-1) * heightOfLeft 这个面积一定比刚才的长度为 n 的面积 (n * heightOfLeft) 小` 。

也就是说**最大面积一定是当前的面积或者通过移动短的端点得到。**

关键点解析

- 双指针优化时间复杂度

代码

- 语言支持：JS，C++,Python

JavaScript Code:

```
/**
 * @param {number[]} height
 * @return {number}
 */
var maxArea = function (height) {
  if (!height || height.length <= 1) return 0;

  let leftPos = 0;
  let rightPos = height.length - 1;
  let max = 0;
  while (leftPos < rightPos) {
    const currentArea =
      Math.abs(leftPos - rightPos) *
      Math.min(height[leftPos], height[rightPos]);
    if (currentArea > max) {
      max = currentArea;
    }
    // 更新小的
    if (height[leftPos] < height[rightPos]) {
      leftPos++;
    } else {
      // 如果相等就随便了
      rightPos--;
    }
  }

  return max;
};
```

C++ Code:

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        auto ret = 0ul, leftPos = 0ul, rightPos = height.s
        while( leftPos < rightPos)
        {
            ret = std::max(ret, std::min(height[leftPos], 
            if (height[leftPos] < height[rightPos]) ++left
            else --rightPos;
        }
        return ret;
    }
};
```

Python Code:

```
class Solution:
    def maxArea(self, heights):
        l, r =  0, len(heights) - 1
        ans = 0
        while l < r:
            ans = max(ans, (r - l) * min(heights[l], height
            if heights[r] > heights[l]:
                l += 1
            else:
                r -= 1
        return ans
```

复杂度分析

- 时间复杂度：由于左右指针移动的次数加起来正好是 n， 因此时间复杂度为 $O(N)$。
- 空间复杂度：$O(1)$。

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经30K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode.com/problems/3sum/description/

题目描述

```
Given an array nums of n integers, are there elements a, b,

Note:

The solution set must not contain duplicate triplets.

Example:

Given array nums = [-1, 0, 1, 2, -1, -4],

A solution set is:
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

前置知识

- 排序
- 双指针
- 分治

公司

- 阿里
- 字节

思路

我们采用 `分治` 的思想. 想要找出三个数相加等于 0，我们可以数组依次遍历， 每一项 a[i]我们都认为它是最终能够用组成 0 中的一个数字，那么我们的目标就是找到 剩下的元素（除 a[i]） `两个` 相加等于-a[i].

通过上面的思路，我们的问题转化为了 `给定一个数组，找出其中两个相加等于给定值` ， 这个问题是比较简单的， 我们只需要对数组进行排序，然后双指针解决即可。 加上我们需要外层遍历依次数组，因此总的时间复杂

度应该是 O(N^2)。

思路如图所示：

> 在这里之所以要排序解决是因为， 我们算法的瓶颈在这里不在于排序，而在于 O(N^2)，如果我们瓶颈是排序，就可以考虑别的方式了
>
> 如果找某一个特定元素，一个指针就够了。如果是找两个元素满足一定关系（比如求和等于特定值），需要双指针， 当然前提是数组有序。

关键点解析

- 排序之后，用双指针
- 分治

代码

```
/*
 * @lc app=leetcode id=15 lang=javascript
 *
 * [15] 3Sum
 *
 * https://leetcode.com/problems/3sum/description/
 *
 * algorithms
 * Medium (23.51%)
 * Total Accepted:    531.5K
 * Total Submissions: 2.2M
 * Testcase Example:  '[-1,0,1,2,-1,-4]'
 *
 * Given an array nums of n integers, are there elements a,
 * that a + b + c = 0? Find all unique triplets in the arra
 * sum of zero.
 *
 * Note:
 *
 * The solution set must not contain duplicate triplets.
 *
 * Example:
 *
 *
 * Given array nums = [-1, 0, 1, 2, -1, -4],
 *
 * A solution set is:
 * [
 *  [-1, 0, 1],
 *  [-1, -1, 2]
 * ]
 *
 *
 */
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var threeSum = function (nums) {
  if (nums.length < 3) return [];
  const list = [];
  nums.sort((a, b) => a - b);
  for (let i = 0; i < nums.length; i++) {
    //nums is sorted,so it's impossible to have a sum = 0
    if (nums[i] > 0) break;
    // skip duplicated result without set
    if (i > 0 && nums[i] === nums[i - 1]) continue;
    let left = i + 1;
```

```
    let right = nums.length - 1;

    // for each index i
    // we want to find the triplet [i, left, right] which s
    while (left < right) {
      // since left < right, and left > i, no need to compa
      if (nums[left] + nums[right] + nums[i] === 0) {
        list.push([nums[left], nums[right], nums[i]]);
        // skip duplicated result without set
        while (nums[left] === nums[left + 1]) {
          left++;
        }
        left++;
        // skip duplicated result without set
        while (nums[right] === nums[right - 1]) {
          right--;
        }
        right--;
        continue;
      } else if (nums[left] + nums[right] + nums[i] > 0) {
        right--;
      } else {
        left++;
      }
    }
  }
  return list;
};
```

题目地址

https://leetcode-cn.com/problems/letter-combinations-of-a-phone-number

题目描述

```
给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。

示例:

输入："23"
输出：["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]

说明:
尽管上面的答案是按字典序排列的，但是你可以任意选择答案输出的顺序。
```

前置知识

- 回溯

公司

- 阿里
- 百度
- 字节
- 腾讯

思路

使用回溯法进行求解，回溯是一种通过穷举所有可能情况来找到所有解的算法。如果一个候选解最后被发现并不是可行解，回溯算法会舍弃它，并在前面的一些步骤做出一些修改，并重新尝试找到可行解。究其本质，其实就是枚举。

如果没有更多的数字需要被输入，说明当前的组合已经产生。

如果还有数字需要被输入：

- 遍历下一个数字所对应的所有映射的字母

- 将当前的字母添加到组合最后，也就是 str + tmp[r]

关键点

- 回溯
- 回溯模板

代码

- 语言支持：JS

```
/**
 * @param {string} digits
 * @return {string[]}
 */
const letterCombinations = function (digits) {
  if (!digits) {
    return [];
  }
  const len = digits.length;
  const map = new Map();
  map.set("2", "abc");
  map.set("3", "def");
  map.set("4", "ghi");
  map.set("5", "jkl");
  map.set("6", "mno");
  map.set("7", "pqrs");
  map.set("8", "tuv");
  map.set("9", "wxyz");
  const result = [];

  function generate(i, str) {
    if (i == len) {
      result.push(str);
      return;
    }
    const tmp = map.get(digits[i]);
    for (let r = 0; r < tmp.length; r++) {
      generate(i + 1, str + tmp[r]);
    }
  }
  generate(0, "");
  return result;
};
```

复杂度分析

N + M 是输入数字的总数

- 时间复杂度：O(3^N * 4^M)
- 空间复杂度：O(3^N * 4^M)

题目地址

https://leetcode.com/problems/remove-nth-node-from-end-of-list/description

题目描述

Given a linked list, remove the n-th node from the end of list and return its head.

Example:

Given linked list: 1->2->3->4->5, and n = 2.

After removing the second node from the end, the linked list becomes 1->2->3->5. Note:

Given n will always be valid.

Follow up:

Could you do this in one pass?

前置知识

- 链表
- 双指针

公司

- 阿里
- 百度
- 腾讯
- 字节

思路

双指针，指针 A 先移动 n 次， 指针 B 再开始移动。当 A 到达 null 的时候， 指针 b 的位置正好是倒数 n

我们可以设想假设设定了双指针 p 和 q 的话，当 q 指向末尾的 NULL，p 与 q 之间相隔的元素个数为 n 时，那么删除掉 p 的下一个指针就完成了要求。

设置虚拟节点 dummyHead 指向 head

设定双指针 p 和 q，初始都指向虚拟节点 dummyHead

移动 q，直到 p 与 q 之间相隔的元素个数为 n

同时移动 p 与 q，直到 q 指向的为 NULL

将 p 的下一个节点指向下下个节点

(图片来自： https://github.com/MisterBooo/LeetCodeAnimation)

关键点解析

1. 链表这种数据结构的特点和使用
2. 使用双指针
3. 使用一个 dummyHead 简化操作

代码

Support: JS, Java

- Javascript Implementation

```
/*
 * @lc app=leetcode id=19 lang=javascript
 *
 * [19] Remove Nth Node From End of List
 *
 * https://leetcode.com/problems/remove-nth-node-from-end-c
 *
 * algorithms
 * Medium (34.03%)
 * Total Accepted:    360.1K
 * Total Submissions: 1.1M
 * Testcase Example:  '[1,2,3,4,5]\n2'
 *
 * Given a linked list, remove the n-th node from the end c
 * its head.
 *
 * Example:
 *
 *
 * Given linked list: 1->2->3->4->5, and n = 2.
 *
 * After removing the second node from the end, the linked
 * 1->2->3->5.
 *
 *
 * Note:
 *
 * Given n will always be valid.
 *
 * Follow up:
 *
 * Could you do this in one pass?
 *
 */
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @param {number} n
 * @return {ListNode}
 */
var removeNthFromEnd = function (head, n) {
  let i = -1;
```

```
  const noop = {
    next: null,
  };

  const dummyHead = new ListNode(); // 增加一个dummyHead 简化
  dummyHead.next = head;

  let currentP1 = dummyHead;
  let currentP2 = dummyHead;

  while (currentP1) {
    if (i === n) {
      currentP2 = currentP2.next;
    }

    if (i !== n) {
      i++;
    }

    currentP1 = currentP1.next;
  }

  currentP2.next = ((currentP2 || noop).next || noop).next;

  return dummyHead.next;
};
```

- Java Code

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n)
        TreeNode dummy = new TreeNode(0);
        dummy.next = head;
        TreeNode first = dummy;
        TreeNode second = dummy;

        if (int i=0; i<=n; i++) {
            first = first.next;
        }

        while (first != null) {
            first = first.next;
            second = second.next;
        }

        second.next = second.next.next;

        return dummy.next;
    }
}
```

题目地址

https://leetcode-cn.com/problems/generate-parentheses

题目描述

```
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

示例：

输入：n = 3
输出：[
       "((())))",
       "(()())",
       "(())()",
       "()(())",
       "()()()"
     ]
```

前置知识

- DFS
- 回溯法

公司

- 阿里
- 百度
- 腾讯
- 字节

思路

深度优先搜索(回溯思想)，从空字符串开始构造，做加法。

关键点

- 当 l < r 时记得剪枝

代码

- 语言支持：JS

```
/**
 * @param {number} n
 * @return {string[]}
 * @param l 左括号已经用了几个
 * @param r 右括号已经用了几个
 * @param str 当前递归得到的拼接字符串结果
 * @param res 结果集
 */
const generateParenthesis = function (n) {
  const res = [];

  function dfs(l, r, str) {
    if (l == n && r == n) {
      return res.push(str);
    }
    // l 小于 r 时不满足条件 剪枝
    if (l < r) {
      return;
    }
    // l 小于 n 时可以插入左括号，最多可以插入 n 个
    if (l < n) {
      dfs(l + 1, r, str + "(");
    }
    // r < l 时 可以插入右括号
    if (r < l) {
      dfs(l, r + 1, str + ")");
    }
  }
  dfs(0, 0, "");
  return res;
};
```

复杂度分析

- 时间复杂度：O(2^N)
- 空间复杂度：O(2^N)

题目地址

https://leetcode.com/problems/swap-nodes-in-pairs/description/

题目描述

Given a linked list, swap every two adjacent nodes and return its head.

You may not modify the values in the list's nodes, only nodes itself may be changed.

Example:

Given 1->2->3->4, you should return the list as 2->1->4->3.

前置知识

- 链表

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

设置一个 dummy 节点简化操作，dummy next 指向 head。

1. 初始化 first 为第一个节点
2. 初始化 second 为第二个节点
3. 初始化 current 为 dummy
4. first.next = second.next
5. second.next = first
6. current.next = second
7. current 移动两格
8. 重复

（图片来自： https://github.com/MisterBooo/LeetCodeAnimation）

关键点解析

1. 链表这种数据结构的特点和使用
2. dummyHead 简化操作

代码

- 语言支持：JS，Python3

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */
var swapPairs = function(head) {
  const dummy = new ListNode(0);
  dummy.next = head;
  let current = dummy;
  while (current.next != null && current.next.next != null
    // 初始化双指针
    const first = current.next;
    const second = current.next.next;

    // 更新双指针和 current 指针
    first.next = second.next;
    second.next = first;
    current.next = second;

    // 更新指针
    current = current.next.next;
  }
  return dummy.next;
};
```

Python3 Code:

```python
class Solution:
    def swapPairs(self, head: ListNode) -> ListNode:
        """
        用递归实现链表相邻互换：
        第一个节点的 next 是第三、第四个节点交换的结果，第二个节点的
        第三个节点的 next 是第五、第六个节点交换的结果，第四个节点的
        以此类推
        :param ListNode head
        :return ListNode
        """
        # 如果为 None 或 next 为 None，则直接返回
        if not head or not head.next:
            return head

        _next = head.next
        head.next = self.swapPairs(_next.next)
        _next.next = head
        return _next
```

题目地址

https://leetcode.com/problems/divide-two-integers/description/

题目描述

```
Given two integers dividend and divisor, divide two integer

Return the quotient after dividing dividend by divisor.

The integer division should truncate toward zero.

Example 1:

Input: dividend = 10, divisor = 3
Output: 3
Example 2:

Input: dividend = 7, divisor = -3
Output: -2
Note:

Both dividend and divisor will be 32-bit signed integers.
The divisor will never be 0.
Assume we are dealing with an environment which could only
```

前置知识

- 二分法

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

符合直觉的做法是，减数一次一次减去被减数，不断更新差，直到差小于0，我们减了多少次，结果就是多少。

核心代码：

```
let acc = divisor;
let count = 0;

while (dividend - acc >= 0) {
  acc += divisor;
  count++;
}

return count;
```

这种做法简单直观，但是性能却比较差. 下面来介绍一种性能更好的方法。

通过上面这样的分析，我们直到可以使用二分法来解决，性能有很大的提升。

关键点解析

- 二分查找
- 正负数的判断中，这样判断更简单。

```
const isNegative = dividend > 0 !== divisor > 0;
```

代码

- 语言支持：JS，Python3

```javascript
/*
 * @lc app=leetcode id=29 lang=javascript
 *
 * [29] Divide Two Integers
 */
/**
 * @param {number} dividend
 * @param {number} divisor
 * @return {number}
 */
var divide = function(dividend, divisor) {
  if (divisor === 1) return dividend;

  // 这种方法很巧妙，即符号相同则为正，不同则为负
  const isNegative = dividend > 0 !== divisor > 0;

  const MAX_INTERGER = Math.pow(2, 31);

  const res = helper(Math.abs(dividend), Math.abs(divisor)

  // overflow
  if (res > MAX_INTERGER - 1 || res < -1 * MAX_INTERGER) {
    return MAX_INTERGER - 1;
  }

  return isNegative ? -1 * res : res;
};

function helper(dividend, divisor) {
  // 二分法
  if (dividend <= 0) return 0;
  if (dividend < divisor) return 0;
  if (divisor === 1) return dividend;

  let acc = 2 * divisor;
  let count = 1;

  while (dividend - acc > 0) {
    acc += acc;
    count += count;
  }
  // 直接使用位移运算，比如acc >> 1会有问题
  const last = dividend - Math.floor(acc / 2);

  return count + helper(last, divisor);
}
```

Python3 Code:

```
class Solution:
    def divide(self, dividend: int, divisor: int) -> int:
        """
        二分法
        :param int divisor
        :param int dividend
        :return int
        """
        # 错误处理
        if divisor == 0:
            raise ZeroDivisionError
        if abs(divisor) == 1:
            result = dividend if 1 == divisor else -divider
            return min(2**31-1, max(-2**31, result))

        # 确定结果的符号
        sign = ((dividend >= 0) == (divisor >= 0))

        result = 0
        # abs也可以直接写在while条件中，不过可能会多计算几次
        _divisor = abs(divisor)
        _dividend = abs(dividend)

        while _divisor <= _dividend:
            r, _dividend = self._multi_divide(_divisor, _d
            result += r

        result = result if sign else -result

        # 注意返回值不能超过32位有符号数的表示范围
        return min(2**31-1, max(-2**31, result))

    def _multi_divide(self, divisor, dividend):
        """
        翻倍除法，如果可以被除，则下一步除数翻倍，直至除数大于被除数，
        返回商加总的结果与被除数的剩余值；
        这里就不做异常处理了；
        :param int divisor
        :param int dividend
        :return tuple result, left_dividend
        """
        result = 0
        times_count = 1
        while divisor <= dividend:
            dividend -= divisor
            result += times_count
            times_count += times_count
```

```
            divisor += divisor
        return result, dividend
```

相关题目

- 875.koko-eating-bananas

题目地址

https://leetcode.com/problems/next-permutation/description/

题目描述

```
Implement next permutation, which rearranges numbers into t

If such arrangement is not possible, it must rearrange it a

The replacement must be in-place and use only constant extr

Here are some examples. Inputs are in the left-hand column

1,2,3 → 1,3,2
3,2,1 → 1,2,3
1,1,5 → 1,5,1
```

前置知识

- 回溯法

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

符合直觉的方法是我们按顺序求出所有的排列，如果当前排列等于 nums，那么我直接取下一个 但是这种做法不符合 constant space 要求（题目要求直接修改原数组）,时间复杂度也太高，为 O(n!),肯定不是合适的解。

这种题目比较抽象，写几个例子通常会帮助理解问题的规律。我找了几个例子，其中蓝色背景表示的是当前数字找下一个更大排列的时候 `需要改变的元素` .

我们不难发现，蓝色的数字都是从后往前第一个不递增的元素，并且我们的下一个更大的排列 只需要改变蓝色的以及之后部分即可，前面的不需要变。

那么怎么改变蓝色的以及后面部分呢？为了使增量最小， 由于前面我们观察发现，其实剩下的元素从左到右是递减的，而我们想要变成递增的，我们只需要不断交换首尾元素即可。

另外我们也可以以回溯的角度来思考这个问题，让我们先回溯一次：

这个时候可以选择的元素只有2，我们无法组成更大的排列，我们继续回溯，直到如图：

我们发现我们可以交换4或者2实现变大的效果，但是要保证变大的幅度最小（下一个更大）， 我们需要选择最小的，由于之前我们发现后面是从左到右递减的，显然就是交换最右面大于1的。

之后就是不断交换使之幅度最小：

关键点解析

- 写几个例子通常会帮助理解问题的规律
- 在有序数组中首尾指针不断交换位置即可实现reverse
- 找到从右边起 `第一个大于nums[i]的` ，并将其和nums[i]进行交换

代码

- 语言支持: Javascript，Python3

```javascript
/*
 * @lc app=leetcode id=31 lang=javascript
 *
 * [31] Next Permutation
 */

function reverseRange(A, i, j) {
  while (i < j) {
    const temp = A[i];
    A[i] = A[j];
    A[j] = temp;
    i++;
    j--;
  }
}
/**
 * @param {number[]} nums
 * @return {void} Do not return anything, modify nums in-p
 */
var nextPermutation = function(nums) {
  // 时间复杂度O(n) 空间复杂度O(1)
  if (nums == null || nums.length <= 1) return;

  let i = nums.length - 2;
  // 从后往前找到第一个降序的,相当于找到了我们的回溯点
  while (i > -1 && nums[i + 1] <= nums[i]) i--;

  // 如果找了就swap
  if (i > -1) {
    let j = nums.length - 1;
    // 找到从右边起第一个大于nums[i]的，并将其和nums[i]进行交换
    // 因为如果交换的数字比nums[i]还要小肯定不符合题意
    while (nums[j] <= nums[i]) j--;
    const temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
  }

  // 最后我们只需要将剩下的元素从左到右，依次填入当前最小的元素就可以
  // [i + 1, A.length -1]的元素进行反转

  reverseRange(nums, i + 1, nums.length - 1);
};
```

Python3 Code:

```
class Solution:
    def nextPermutation(self, nums):
        """
        Do not return anything, modify nums in-place instea
        :param list nums
        """
        # 第一步，从后往前，找到下降点
        down_index = None
        for i in range(len(nums)-2, -1, -1):
            if nums[i] < nums[i+1]:
                down_index = i
                break
        # 如果没有下降点，重新排列
        if down_index is None:
            nums.reverse()
        # 如果有下降点
        else:
            # 第二步，从后往前，找到比下降点大的数，对换位置
            for i in range(len(nums)-1, i, -1):
                if nums[down_index] < nums[i]:
                    nums[down_index], nums[i] = nums[i], nu
                    break
            # 第三部，重新排列下降点之后的数
            i, j = down_index+1, len(nums)-1
            while i < j:
                nums[i], nums[j] = nums[j], nums[i]
                i += 1
                j -= 1
```

Python3 Code:

```
class Solution:
    def nextPermutation(self, nums):
        """
        Do not return anything, modify nums in-place instea
        :param list nums
        """
        # 第一步，从后往前，找到下降点
        down_index = None
        for i in range(len(nums)-2, -1, -1):
            if nums[i] < nums[i+1]:
                down_index = i
                break
        # 如果没有下降点，重新排列
        if down_index is None:
            nums.reverse()
        # 如果有下降点
        else:
            # 第二步，从后往前，找到比下降点大的数，对换位置
            for i in range(len(nums)-1, i, -1):
                if nums[down_index] < nums[i]:
                    nums[down_index], nums[i] = nums[i], nu
                    break
            # 第三步，重新排列下降点之后的数
            i, j = down_index+1, len(nums)-1
            while i < j:
                nums[i], nums[j] = nums[j], nums[i]
                i += 1
                j -= 1
```

相关题目

- 46.next-permutation
- 47.permutations-ii
- 60.permutation-sequence(TODO)

题目地址

https://leetcode.com/problems/search-in-rotated-sorted-array/

题目描述

```
Suppose an array sorted in ascending order is rotated at sc

(i.e., [0,1,2,4,5,6,7] might become [4,5,6,7,0,1,2]).

You are given a target value to search. If found in the arr

You may assume no duplicate exists in the array.

Your algorithm's runtime complexity must be in the order of

Example 1:

Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4
Example 2:

Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1
```

前置知识

- 数组
- 二分法

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这是一个我在网上看到的前端头条技术终面的一个算法题。

题目要求时间复杂度为logn，因此基本就是二分法了。 这道题目不是直接的有序数组，不然就是easy了。

首先要知道，我们随便选择一个点，将数组分为前后两部分，其中一部分一定是有序的。

具体步骤：

- 我们可以先找出mid，然后根据mid来判断，mid是在有序的部分还是无序的部分

假如mid小于start，则mid一定在右边有序部分。 假如mid大于等于start，则mid一定在左边有序部分。

> 注意等号的考虑

- 然后我们继续判断target在哪一部分， 我们就可以舍弃另一部分了

我们只需要比较target和有序部分的边界关系就行了。 比如mid在右侧有序部分，即[mid, end] 那么我们只需要判断 target >= mid && target <= end 就能知道target在右侧有序部分，我们就 可以舍弃左边部分了(start = mid + 1)， 反之亦然。

我们以([6,7,8,1,2,3,4,5], 4)为例讲解一下：

关键点解析

- 二分法
- 找出有序区间，然后根据target是否在有序区间舍弃一半元素

代码

- 语言支持: Javascript，Python3

```javascript
/*
 * @lc app=leetcode id=33 lang=javascript
 *
 * [33] Search in Rotated Sorted Array
 */
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number}
 */
var search = function(nums, target) {
  // 时间复杂度: O(logn)
  // 空间复杂度: O(1)
  // [6,7,8,1,2,3,4,5]
  let start = 0;
  let end = nums.length - 1;

  while (start <= end) {
    const mid = start + ((end - start) >> 1);
    if (nums[mid] === target) return mid;

    // [start, mid]有序

    // ⚠注意这里的等号
    if (nums[mid] >= nums[start]) {
      //target 在 [start, mid] 之间

      // 其实target不可能等于nums[mid], 但是为了对称, 我还是加上
      if (target >= nums[start] && target <= nums[mid]) {
        end = mid - 1;
      } else {
        //target 不在 [start, mid] 之间
        start = mid + 1;
      }
    } else {
      // [mid, end]有序

      // target 在 [mid, end] 之间
      if (target >= nums[mid] && target <= nums[end]) {
        start = mid + 1;
      } else {
        // target 不在 [mid, end] 之间
        end = mid - 1;
      }
    }
  }
```

```
    return -1;
};
```

Python3 Code:

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        """用二分法，先判断左右两边哪一边是有序的，再判断是否在有序的
        if len(nums) <= 0:
            return -1

        left = 0
        right = len(nums) - 1
        while left < right:
            mid = (right - left) // 2 + left
            if nums[mid] == target:
                return mid

            # 如果中间的值大于最左边的值，说明左边有序
            if nums[mid] > nums[left]:
                if nums[left] <= target <= nums[mid]:
                    right = mid
                else:
                    # 这里 +1，因为上面是 <= 符号
                    left = mid + 1
            # 否则右边有序
            else:
                # 注意：这里必须是 mid+1，因为根据我们的比较方式，
                if nums[mid+1] <= target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid

        return left if nums[left] == target else -1
```

题目地址

https://leetcode.com/problems/combination-sum/description/

题目描述

```
Given a set of candidate numbers (candidates) (without dup

The same repeated number may be chosen from candidates unl

Note:

All numbers (including target) will be positive integers.
The solution set must not contain duplicate combinations.
Example 1:

Input: candidates = [2,3,6,7], target = 7,
A solution set is:
[
  [7],
  [2,2,3]
]
Example 2:

Input: candidates = [2,3,5], target = 8,
A solution set is:
[
  [2,2,2,2],
  [2,3,3],
  [3,5]
]
```

前置知识

- 回溯法

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是求集合，并不是 `求极值` ，因此动态规划不是特别切合，因此我们需要考虑别的方法。

这种题目其实有一个通用的解法，就是回溯法。 网上也有大神给出了这种回溯法解题的 通用写法)，这里的所有的解法使用通用方法解答。 除了这道题目还有很多其他题目可以用这种通用解法，具体的题目见后方相关题目部分。

我们先来看下通用解法的解题思路，我画了一张图：

> 图是 78.subsets，都差不多，仅做参考。

通用写法的具体代码见下方代码区。

关键点解析

- 回溯法
- backtrack 解题公式

代码

- 语言支持: Javascript，Python3

```
/*
 * @lc app=leetcode id=39 lang=javascript
 *
 * [39] Combination Sum
 *
 * https://leetcode.com/problems/combination-sum/descriptio
 *
 * algorithms
 * Medium (46.89%)
 * Total Accepted:    326.7K
 * Total Submissions: 684.2K
 * Testcase Example:  '[2,3,6,7]\n7'
 *
 * Given a set of candidate numbers (candidates) (without d
 * target number (target), find all unique combinations in
 * candidate numbers sums to target.
 *
 * The same repeated number may be chosen from candidates u
 * times.
 *
 * Note:
 *
 *
 * All numbers (including target) will be positive integers
 * The solution set must not contain duplicate combinations
 *
 *
 * Example 1:
 *
 *
 * Input: candidates = [2,3,6,7], target = 7,
 * A solution set is:
 * [
 *  [7],
 *  [2,2,3]
 * ]
 *
 *
 * Example 2:
 *
 *
 * Input: candidates = [2,3,5], target = 8,
 * A solution set is:
 * [
 * [2,2,2,2],
 * [2,3,3],
 * [3,5]
 * ]
```

```
 *
 */

function backtrack(list, tempList, nums, remain, start) {
  if (remain < 0) return;
  else if (remain === 0) return list.push([...tempList]);
  for (let i = start; i < nums.length; i++) {
    tempList.push(nums[i]);
    backtrack(list, tempList, nums, remain - nums[i], i);
    tempList.pop();
  }
}
/**
 * @param {number[]} candidates
 * @param {number} target
 * @return {number[][]}
 */
var combinationSum = function(candidates, target) {
  const list = [];
  backtrack(list, [], candidates.sort((a, b) => a - b), tar
  return list;
};
```

Python3 Code:

```
class Solution:
    def combinationSum(self, candidates: List[int], target:
        """
        回溯法，层层递减，得到符合条件的路径就加入结果集中，超出则剪枝
        主要是要注意一些细节，避免重复等；
        """
        size = len(candidates)
        if size <= 0:
            return []

        # 先排序，便于后面剪枝
        candidates.sort()

        path = []
        res = []
        self._find_path(target, path, res, candidates, 0, s

        return res

    def _find_path(self, target, path, res, candidates, beg
        """沿着路径往下走"""
        if target == 0:
            res.append(path.copy())
        else:
            for i in range(begin, size):
                left_num = target - candidates[i]
                # 如果剩余值为负数，说明超过了，剪枝
                if left_num < 0:
                    break
                # 否则把当前值加入路径
                path.append(candidates[i])
                # 为避免重复解，我们把比当前值小的参数也从下一次寻找
                # 因为根据他们得出的解一定在之前就找到过了
                self._find_path(left_num, path, res, candid
                # 记得把当前值移出路径，才能进入下一个值的路径
                path.pop()
```

相关题目

- 40.combination-sum-ii
- 46.permutations
- 47.permutations-ii
- 78.subsets
- 90.subsets-ii
- 113.path-sum-ii
- 131.palindrome-partitioning

题目地址

https://leetcode.com/problems/combination-sum-ii/description/

题目描述

```
Given a collection of candidate numbers (candidates) and a

Each number in candidates may only be used once in the comb

Note:

All numbers (including target) will be positive integers.
The solution set must not contain duplicate combinations.
Example 1:

Input: candidates = [10,1,2,7,6,1,5], target = 8,
A solution set is:
[
  [1, 7],
  [1, 2, 5],
  [2, 6],
  [1, 1, 6]
]
Example 2:

Input: candidates = [2,5,2,1,2], target = 5,
A solution set is:
[
  [1,2,2],
  [5]
]
```

前置知识

- 回溯法

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是求集合，并不是 `求极值` ，因此动态规划不是特别切合，因此我们需要考虑别的方法。

这种题目其实有一个通用的解法，就是回溯法。 网上也有大神给出了这种回溯法解题的 通用写法)，这里的所有的解法使用通用方法解答。 除了这道题目还有很多其他题目可以用这种通用解法，具体的题目见后方相关题目部分。

我们先来看下通用解法的解题思路，我画了一张图：

> 图是 78.subsets，都差不多，仅做参考。

通用写法的具体代码见下方代码区。

关键点解析

- 回溯法
- backtrack 解题公式

代码

- 语言支持: Javascript，Python3

```
function backtrack(list, tempList, nums, remain, start) {
    if (remain < 0) return;
    else if (remain === 0) return list.push([...tempList]);
    for (let i = start; i < nums.length; i++) {
      // 和39.combination-sum 的其中一个区别就是这道题candidate
      // 代码表示就是下面这一行
      if(i > start && nums[i] == nums[i-1]) continue; // sk
      tempList.push(nums[i]);
      backtrack(list, tempList, nums, remain - nums[i], i +
      tempList.pop();
    }
  }
/**
 * @param {number[]} candidates
 * @param {number} target
 * @return {number[][]}
 */
var combinationSum2 = function(candidates, target) {
    const list = [];
    backtrack(list, [], candidates.sort((a, b) => a - b), t
    return list;
};
```

Python3 Code:

```
class Solution:
    def combinationSum2(self, candidates: List[int], target
        """
        与39题的区别是不能重用元素，而元素可能有重复；
        不能重用好解决，回溯的index往下一个就行；
        元素可能有重复，就让结果的去重麻烦一些；
        """
        size = len(candidates)
        if size == 0:
            return []

        # 还是先排序，主要是方便去重
        candidates.sort()

        path = []
        res = []
        self._find_path(candidates, path, res, target, 0, s

        return res

    def _find_path(self, candidates, path, res, target, beg
        if target == 0:
            res.append(path.copy())
        else:
            for i in range(begin, size):
                left_num = target - candidates[i]
                if left_num < 0:
                    break
                # 如果存在重复的元素，前一个元素已经遍历了后一个元素
                if i > begin and candidates[i] == candidate
                    continue
                path.append(candidates[i])
                # 开始的 index 往后移了一格
                self._find_path(candidates, path, res, left
                path.pop()
```

相关题目

- 39.combination-sum
- 46.permutations
- 47.permutations-ii
- 78.subsets
- 90.subsets-ii
- 113.path-sum-ii
- 131.palindrome-partitioning

题目地址

https://leetcode.com/problems/permutations/description/

题目描述

```
Given a collection of distinct integers, return all possib

Example:

Input: [1,2,3]
Output:
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

前置知识

- 回溯法

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是求集合，并不是 `求极值` ，因此动态规划不是特别切合，因此我们需要考虑别的方法。

这种题目其实有一个通用的解法，就是回溯法。 网上也有大神给出了这种回溯法解题的 通用写法>)，这里的所有的解法使用通用方法解答。 除了这道题目还有很多其他题目可以用这种通用解法，具体的题目见后方相关题目部分。

我们先来看下通用解法的解题思路，我画了一张图：

backtrack @lucifer

> 图是 78.subsets，都差不多，仅做参考。

通用写法的具体代码见下方代码区。

关键点解析

- 回溯法
- backtrack 解题公式

代码

- 语言支持: Javascript, Python3

Javascript Code:

```
/*
 * @lc app=leetcode id=46 lang=javascript
 *
 * [46] Permutations
 *
 * https://leetcode.com/problems/permutations/description/
 *
 * algorithms
 * Medium (53.60%)
 * Total Accepted:    344.6K
 * Total Submissions: 642.9K
 * Testcase Example:  '[1,2,3]'
 *
 * Given a collection of distinct integers, return all poss
 *
 * Example:
 *
 *
 * Input: [1,2,3]
 * Output:
 * [
 *  [1,2,3],
 *  [1,3,2],
 *  [2,1,3],
 *  [2,3,1],
 *  [3,1,2],
 *  [3,2,1]
 * ]
 *
 *
 */
function backtrack(list, tempList, nums) {
  if (tempList.length === nums.length) return list.push([..
  for (let i = 0; i < nums.length; i++) {
    if (tempList.includes(nums[i])) continue;
    tempList.push(nums[i]);
    backtrack(list, tempList, nums);
    tempList.pop();
  }
}
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var permute = function (nums) {
  const list = [];
  backtrack(list, [], nums);
```

```
    return list;
};
```

Python3 Code:

```
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        """itertools库内置了这个函数"""
        return itertools.permutations(nums)

    def permute2(self, nums: List[int]) -> List[List[int]]:
        """自己写回溯法"""
        res = []
        def _backtrace(nums, pre_list):
            if len(nums) <= 0:
                res.append(pre_list)
            else:
                for i in nums:
                    # 注意copy一份新的调用，否则无法正常循环
                    p_list = pre_list.copy()
                    p_list.append(i)
                    left_nums = nums.copy()
                    left_nums.remove(i)
                    _backtrace(left_nums, p_list)
        _backtrace(nums, [])
        return res
```

Python Code:

```
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        """itertools库内置了这个函数"""
        import itertools
        return itertools.permutations(nums)

    def permute2(self, nums: List[int]) -> List[List[int]]:
        """自己写回溯法"""
        res = []
        def _backtrace(nums, pre_list):
            if len(nums) <= 0:
                res.append(pre_list)
            else:
                for i in nums:
                    # 注意copy一份新的调用，否则无法正常循环
                    p_list = pre_list.copy()
                    p_list.append(i)
                    left_nums = nums.copy()
                    left_nums.remove(i)
                    _backtrace(left_nums, p_list)
        _backtrace(nums, [])
        return res

    def permute3(self, nums: List[int]) -> List[List[int]]:
        """回溯的另一种写法"""
        res = []
        length = len(nums)
        def _backtrack(start=0):
            if start == length:
                # nums[:] 返回 nums 的一个副本，指向新的引用，这
                res.append(nums[:])
            for i in range(start, length):
                nums[start], nums[i] = nums[i], nums[start]
                _backtrack(start+1)
                nums[start], nums[i] = nums[i], nums[start]
        _backtrack()
        return res
```

相关题目

- 31.next-permutation
- 39.combination-sum
- 40.combination-sum-ii
- 47.permutations-ii
- 60.permutation-sequence
- 78.subsets

- 90.subsets-ii
- 113.path-sum-ii
- 131.palindrome-partitioning

题目地址

https://leetcode.com/problems/permutations-ii/description/

题目描述

```
Given a collection of numbers that might contain duplicates

Example:

Input: [1,1,2]
Output:
[
  [1,1,2],
  [1,2,1],
  [2,1,1]
]
```

前置知识

- 回溯法

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是求集合，并不是 `求极值` ，因此动态规划不是特别切合，因此我们需要考虑别的方法。

这种题目其实有一个通用的解法，就是回溯法。 网上也有大神给出了这种回溯法解题的 通用写法>)，这里的所有的解法使用通用方法解答。 除了这道题目还有很多其他题目可以用这种通用解法，具体的题目见后方相关题目部分。

我们先来看下通用解法的解题思路，我画了一张图：

backtrack @lucifer

> 图是 78.subsets，都差不多，仅做参考。

通用写法的具体代码见下方代码区。

关键点解析

- 回溯法
- backtrack 解题公式

代码

- 语言支持: Javascript，Python3

```
/*
 * @lc app=leetcode id=47 lang=javascript
 *
 * [47] Permutations II
 *
 * https://leetcode.com/problems/permutations-ii/descriptio
 *
 * algorithms
 * Medium (39.29%)
 * Total Accepted:    234.1K
 * Total Submissions: 586.2K
 * Testcase Example:  '[1,1,2]'
 *
 * Given a collection of numbers that might contain duplica
 * possible unique permutations.
 *
 * Example:
 *
 *
 * Input: [1,1,2]
 * Output:
 * [
 *  [1,1,2],
 *  [1,2,1],
 *  [2,1,1]
 * ]
 *
 *
 */
function backtrack(list, nums, tempList, visited) {
  if (tempList.length === nums.length) return list.push([..
  for (let i = 0; i < nums.length; i++) {
    // 和46.permutations的区别是这道题的nums是可以重复的
    // 我们需要过滤这种情况
    if (visited[i]) continue; // 不能用tempList.includes(nu
    // visited[i - 1] 这个判断容易忽略
    if (i > 0 && nums[i] === nums[i - 1] && visited[i - 1]

    visited[i] = true;
    tempList.push(nums[i]);
    backtrack(list, nums, tempList, visited);
    visited[i] = false;
    tempList.pop();
  }
}
/**
 * @param {number[]} nums
 * @return {number[][]}
```

```
 */
var permuteUnique = function (nums) {
  const list = [];
  backtrack(
    list,
    nums.sort((a, b) => a - b),
    [],
    []
  );
  return list;
};
```

Python3 code:

```
class Solution:
    def permuteUnique(self, nums: List[int]) -> List[List[:
        """与46题一样，当然也可以直接调用itertools的函数，然后去重
        return list(set(itertools.permutations(nums)))

    def permuteUnique(self, nums: List[int]) -> List[List[:
        """自己写回溯法，与46题相比，需要去重"""
        # 排序是为了去重
        nums.sort()
        res = []
        def _backtrace(nums, pre_list):
            if len(nums) <= 0:
                res.append(pre_list)
            else:
                for i in range(len(nums)):
                    # 如果是同样的数字，则之前一定已经生成了对应可
                    if i > 0 and nums[i] == nums[i-1]:
                        continue
                    p_list = pre_list.copy()
                    p_list.append(nums[i])
                    left_nums = nums.copy()
                    left_nums.pop(i)
                    _backtrace(left_nums, p_list)
        _backtrace(nums, [])
        return res
```

相关题目

- 31.next-permutation
- 39.combination-sum
- 40.combination-sum-ii
- 46.permutations

- 60.permutation-sequence(TODO)
- 78.subsets
- 90.subsets-ii
- 113.path-sum-ii
- 131.palindrome-partitioning

题目地址

https://leetcode.com/problems/rotate-image/description/

题目描述

```
You are given an n x n 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Note:

You have to rotate the image in-place, which means you have

Example 1:

Given input matrix =
[
  [1,2,3],
  [4,5,6],
  [7,8,9]
],

rotate the input matrix in-place such that it becomes:
[
  [7,4,1],
  [8,5,2],
  [9,6,3]
]
Example 2:

Given input matrix =
[
  [ 5, 1, 9,11],
  [ 2, 4, 8,10],
  [13, 3, 6, 7],
  [15,14,12,16]
],

rotate the input matrix in-place such that it becomes:
[
  [15,13, 2, 5],
  [14, 3, 4, 1],
  [12, 6, 8, 9],
  [16, 7,10,11]
]
```

前置知识

- 原地算法
- 矩阵

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目让我们 in-place，也就说空间复杂度要求 O(1)，如果没有这个限制的话，很简单。

通过观察发现，我们只需要将第 i 行变成第 n - i - 1 列， 因此我们只需要保存一个原有矩阵，然后按照这个规律一个个更新即可。

代码：

```
var rotate = function (matrix) {
  // 时间复杂度O(n^2) 空间复杂度O(n)
  const oMatrix = JSON.parse(JSON.stringify(matrix)); // c
  const n = oMatrix.length;
  for (let i = 0; i < n; i++) {
    for (let j = 0; j < n; j++) {
      matrix[j][n - i - 1] = oMatrix[i][j];
    }
  }
};
```

如果要求空间复杂度是 O(1)的话，我们可以用一个 temp 记录即可，这个时候就不能逐个遍历了。 比如遍历到 1 的时候，我们把 1 存到 temp，然后更新 1 的值为 7。 1 被换到了 3 的位置，我们再将 3 存到 temp，依次类推。 但是这种解法写起来比较麻烦，这里我就不写了。

事实上有一个更加巧妙的做法，我们可以巧妙地利用对称轴旋转达到我们的目的，如图，我们先进行一次以对角线为轴的翻转，然后 再进行一次以水平轴心线为轴的翻转即可。

这种做法的时间复杂度是 O(n^2)，空间复杂度是 O(1)

关键点解析

- 矩阵旋转操作

代码

- 语言支持: Javascript，Python3

```javascript
/*
 * @lc app=leetcode id=48 lang=javascript
 *
 * [48] Rotate Image
 */
/**
 * @param {number[][]} matrix
 * @return {void} Do not return anything, modify matrix in-
 */
var rotate = function (matrix) {
  // 时间复杂度O(n^2) 空间复杂度O(1)

  // 做法： 先沿着对角线翻转，然后沿着水平线翻转
  const n = matrix.length;
  function swap(arr, [i, j], [m, n]) {
    const temp = arr[i][j];
    arr[i][j] = arr[m][n];
    arr[m][n] = temp;
  }
  for (let i = 0; i < n - 1; i++) {
    for (let j = 0; j < n - i; j++) {
      swap(matrix, [i, j], [n - j - 1, n - i - 1]);
    }
  }

  for (let i = 0; i < n / 2; i++) {
    for (let j = 0; j < n; j++) {
      swap(matrix, [i, j], [n - i - 1, j]);
    }
  }
};
```

Python3 Code:

```
class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place inst
        先做矩阵转置（即沿着对角线翻转），然后每个列表翻转；
        """
        n = len(matrix)
        for i in range(n):
            for j in range(i, n):
                matrix[i][j], matrix[j][i] = matrix[j][i],
        for m in matrix:
            m.reverse()

    def rotate2(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place inst
        通过内置函数zip，可以简单实现矩阵转置，下面的代码等于先整体翻
        不过这种写法的空间复杂度其实是O(n);
        """
        matrix[:] = map(list, zip(*matrix[::-1]))
```

题目地址

https://leetcode.com/problems/group-anagrams/description/

题目描述

```
Given an array of strings, group anagrams together.

Example:

Input: ["eat", "tea", "tan", "ate", "nat", "bat"],
Output:
[
  ["ate","eat","tea"],
  ["nat","tan"],
  ["bat"]
]
Note:

All inputs will be in lowercase.
The order of your output does not matter.
```

前置知识

- 哈希表
- 排序

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

一个简单的解法就是遍历数组，然后对每一项都进行排序，然后将其添加到 hashTable 中，最后输出 hashTable 中保存的值即可。

这种做法空间复杂度 O(n)， 假设排序算法用的快排，那么时间复杂度为 O(n * klogk), n 为数组长度，k 为字符串的平均长度

代码：

```
var groupAnagrams = function (strs) {
  const hashTable = {};

  function sort(str) {
    return str.split("").sort().join("");
  }

  // 这个方法需要排序，因此不是很优，但是很直观，容易想到
  for (let i = 0; i < strs.length; i++) {
    const str = strs[i];
    const key = sort(str);
    if (!hashTable[key]) {
      hashTable[key] = [str];
    } else {
      hashTable[key].push(str);
    }
  }

  return Object.values(hashTable);
};
```

下面我们介绍另外一种方法，我们建立一个 26 长度的 counts 数组（如果区分大小写，我们可以建立 52 个，如果支持其他字符依次类推）。 然后我们给每一个字符一个固定的数组下标，然后我们只需要更新每个字符出现的次数。 最后形成的 counts 数组如果一致，则说明他们可以通过 交换顺序得到。这种算法空间复杂度 O(n), 时间复杂度 O(n * k), n 为数组长度，k 为字符串的平均长度.

关键点解析

- 桶排序

代码

- 语言支持: Javascript，Python3

```javascript
/*
 * @lc app=leetcode id=49 lang=javascript
 *
 * [49] Group Anagrams
 */
/**
 * @param {string[]} strs
 * @return {string[][]}
 */
var groupAnagrams = function (strs) {
  // 类似桶排序

  let counts = [];
  const hashTable = {};
  for (let i = 0; i < strs.length; i++) {
    const str = strs[i];
    counts = Array(26).fill(0);
    for (let j = 0; j < str.length; j++) {
      counts[str[j].charCodeAt(0) - "a".charCodeAt(0)]++;
    }
    const key = counts.join("");
    if (!hashTable[key]) {
      hashTable[key] = [str];
    } else {
      hashTable[key].push(str);
    }
  }

  return Object.values(hashTable);
};
```

Python3 Code:

```
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[s
        """
        思路同上，在Python中，这里涉及到3个知识点：
        1. 使用内置的 defaultdict 字典设置默认值；
        2. 内置的 ord 函数，计算ASCII值（等于chr）或Unicode值(等
        3. 列表不可哈希，不能作为字典的键，因此这里转为元组；
        """
        str_dict = collections.defaultdict(list)
        for s in strs:
          s_key = [0] * 26
          for c in s:
            s_key[ord(c)-ord('a')] += 1
          str_dict[tuple(s_key)].append(s)
        return str_dict.values()
```

题目地址（50. Pow(x, n)）

https://leetcode-cn.com/problems/powx-n/description/

题目描述

```
实现 pow(x, n) ，即计算 x 的 n 次幂函数。

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 = 1/22 = 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 [-231, 231 - 1] 。
```

前置知识

- 递归
- 位运算

解法零 - 遍历法

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题是让我们实现数学函数 `幂` ，因此直接调用系统内置函数是不被允许的。

符合直觉的做法是 `将x乘以n次` ，这种做法的时间复杂度是$O(N)$。

经实际测试，这种做法果然超时了。测试用例通过 291/304，在 `0.00001\n2147483647` 这个测试用例挂掉了。如果是面试，这个解法可以作为一种兜底解法。

代码

语言支持: Python3

Python3 Code:

```
class Solution:
    def myPow(self, x: float, n: int) -> float:
        if n == 0:
            return 1
        if n < 0:
            return 1 / self.myPow(x, -n)
        res = 1
        for _ in range(n):
            res *= x
        return res
```

解法一 - 普通递归（超时法）

思路

首先我们要知道：

- 如果想要求 x ^ 4，那么我们可以求 (x^2)^2
- 如果是奇数，会有一点不同。 比如 x ^ 5 就等价于 x * (x^2)^2。

> 当然 x ^ 5 可以等价于 (x ^ 2) ^ 2.5, 但是这不相当于直接调用了幂函数了么。对于整数，我们可以很方便的模拟，但是小数就不方便了。

我们的思路就是：

- 将 n 地板除 2，我们不妨设结果为 a
- 那么 myPow(x, n) 就等价于 `myPow(x, a) * myPow(x, n - a)`

很可惜这种算法也会超时，原因在于重复计算会比较多，你可以试一下缓存一下计算看能不能通过。

> 如果你搞不清楚有哪些重复计算，建议画图理解一下。

代码

语言支持: Python3

Python3 Code:

```
class Solution:
    def myPow(self, x: float, n: int) -> float:
        if n == 0:
            return 1
        if n == 1:
            return x
        if n < 0:
            return 1 / self.myPow(x, -n)
        return self.myPow(x, n // 2) * self.myPow(x, n - n
```

解法二 - 优化递归

思路

上面的解法每次直接 myPow 都会调用两次自己。我们不从缓存计算角度，而是从减少这种调用的角度来优化。

我们考虑 myPow 只调用一次自身可以么？ 没错，是可以的。

我们的思路就是：

- 如果 n 是偶数，我们将 n 折半，底数变为 x^2
- 如果 n 是奇数， 我们将 n 减去 1 ，底数不变，得到的结果再乘上底数 x

这样终于可以 AC。

代码

语言支持: Python3

Python3 Code:

```
class Solution:
    def myPow(self, x: float, n: int) -> float:
        if n == 0:
            return 1
        if n == 1:
            return x
        if n < 0:
            return 1 / self.myPow(x, -n)
        return self.myPow(x _ x, n // 2) if n % 2 == 0 else
```

解法三 - 位运算

思路

我们来从位（bit）的角度来看一下这道题。如果你经常看我的题解和文章的话，可能知道我之前写过几次相关的“从位的角度思考分治法”，比如 LeetCode 458.可怜的小猪。

以 x 的 10 次方举例。10 的 2 进制是 1010，然后用 2 进制转 10 进制的方法把它展成 2 的幂次的和。

$$x^{10} = x^{(1010)_2} = x^{1*2^3+0*2^2+1*2^1+0*2^0} = x^{1*2^3} * x^{0*2^2} x^{1*2^1} * x^{0*2^0}$$

因此我们的算法就是：

- 不断的求解 x 的 2^0 次方，x 的 2^1 次方，x 的 2^2 次方等等。
- 将 n 转化为二进制表示
- 将 n 的二进制表示中 `1的位置` pick 出来。比如 n 的第 i 位为 1，那么就将 x^i pick 出来。
- 将 pick 出来的结果相乘

这里有两个问题：

第一个问题是 `似乎我们需要存储 x^i 以便后续相乘的时候用到` 。实际上，我们并不需要这么做。我们可以采取一次遍历的方式来完成，具体看代码。

第二个问题是，如果我们从低位到高位计算的时候，我们如何判断最高位置是否为 1？我们需要一个 bitmask 来完成，这种算法我们甚至需要借助一个额外的变量。 然而我们可以 hack 一下，直接从高位到低位进行计算，这个时候我们只需要判断最后一位是否为 1 就可以了，这个就简单了，我们直接和 1 进行一次 `与运算` 即可。

代码

语言支持: Python3

Python3 Code:

```python
class Solution:
    def myPow(self, x: float, n: int) -> float:
        if n < 0:
            return 1 / self.myPow(x, -n)
        res = 1
        while n:
            if n & 1 == 1:
                res *= x
            x *= x
            n >>= 1
        return res
```

关键点解析

- 超时分析
- hashtable
- 数学分析
- 位运算
- 二进制转十进制

相关题目

- 458.可怜的小猪

题目地址

https://leetcode.com/problems/jump-game/description/

题目描述

```
Given an array of non-negative integers, you are initially

Each element in the array represents your maximum jump leng

Determine if you are able to reach the last index.

Example 1:

Input: [2,3,1,1,4]
Output: true
Explanation: Jump 1 step from index 0 to 1, then 3 steps to
Example 2:

Input: [3,2,1,0,4]
Output: false
Explanation: You will always arrive at index 3 no matter wh
             jump length is 0, which makes it impossible to
```

前置知识

- 贪心

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是一道典型的 `贪心` 类型题目。思路就是用一个变量记录当前能够到达的最大的索引，我们逐个遍历数组中的元素去更新这个索引。变量完成判断这个索引是否大于数组下表即可。

关键点解析

- 建模 (记录和更新当前位置能够到达的最大的索引即可)

代码

- 语言支持: Javascript,C++,Java,Python3 Javascript Code:

```
/**
* @param {number[]} nums
* @return {boolean}
*/
var canJump = function (nums) {
let max = 0; // 能够走到的数组下标

for (let i = 0; i < nums.length; i++) {
  if (max < i) return false; // 当前这一步都走不到，后面更走
  max = Math.max(nums[i] + i, max);
}

return max >= nums.length - 1;
};
```

C++ Code:

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int n=nums.size();
        int k=0;
        for(int i=0;i<n;i++)
        {
            if(i>k){
                return false;
            }
            // 能跳到最后一个位置
            if(k>=n-1){
                return true;
            }
            // 从当前位置能跳的最远的位置
            k = max(k, i+nums[i]);
        }
        return k >= n-1;
    }
};
```

Java Code:

```
class Solution {
    public boolean canJump(int[] nums) {
        int n=nums.length;
        int k=0;
        for(int i=0;i<n;i++)
        {
            if(i>k){
                return false;
            }
            // 能跳到最后一个位置
            if(k>=n-1){
                return true;
            }
            // 从当前位置能跳的最远的位置
            k = Math.max(k, i+nums[i]);
        }
        return k >= n-1;
    }
}
```

Python3 Code:

```
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        """思路同上"""
        _max = 0
        _len = len(nums)
        for i in range(_len-1):
            if _max < i:
                return False
            _max = max(_max, nums[i] + i)
            # 下面这个判断可有可无，但提交的时候数据会好看点
            if _max >= _len - 1:
                return True
        return _max >= _len - 1
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

大家也可以关注我的公众号《脑洞前端》获取更多更新鲜的 LeetCode 题解

题目地址

https://leetcode.com/problems/merge-intervals/description/

题目描述

```
Given a collection of intervals, merge all overlapping inte

Example 1:

Input: [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlaps, merg
Example 2:

Input: [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are considered over
NOTE: input types have been changed on April 15, 2019. Plea
```

前置知识

- 排序

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

- 先对数组进行排序，排序的依据就是每一项的第一个元素的大小。
- 然后我们对数组进行遍历，遍历的时候两两运算（具体运算逻辑见下）
- 判断是否相交，如果不相交，则跳过
- 如果相交，则合并两项

关键点解析

- 对数组进行排序简化操作
- 如果不排序，需要借助一些 hack,这里不介绍了

代码

- 语言支持: Javascript，Python3

```
/*
 * @lc app=leetcode id=56 lang=javascript
 *
 * [56] Merge Intervals
 */
/**
 * @param {number[][]} intervals
 * @return {number[][]}
 */

function intersected(a, b) {
  if (a[0] > b[1] || a[1] < b[0]) return false;
  return true;
}

function mergeTwo(a, b) {
  return [Math.min(a[0], b[0]), Math.max(a[1], b[1])];
}
var merge = function (intervals) {
  // 这种算法需要先排序
  intervals.sort((a, b) => a[0] - b[0]);
  for (let i = 0; i < intervals.length - 1; i++) {
    const cur = intervals[i];
    const next = intervals[i + 1];

    if (intersected(cur, next)) {
      intervals[i] = undefined;
      intervals[i + 1] = mergeTwo(cur, next);
    }
  }
  return intervals.filter((q) => q);
};
```

Python3 Code:

```
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[Lis
        """先排序，后合并"""
        if len(intervals) <= 1:
            return intervals

        # 排序
        def get_first(a_list):
            return a_list[0]
        intervals.sort(key=get_first)

        # 合并
        res = [intervals[0]]
        for i in range(1, len(intervals)):
            if intervals[i][0] <= res[-1][1]:
                res[-1] = [res[-1][0], max(res[-1][1], inte
            else:
                res.append(intervals[i])

        return res
```

复杂度分析

- 时间复杂度：由于采用了排序，因此复杂度大概为 $O(NlogN)$
- 空间复杂度：$O(N)$

题目地址（第 K 个排列）

https://leetcode-cn.com/problems/permutation-sequence/description/

前置知识

- 数学
- 回溯
- factorial

公司

- 阿里
- 百度
- 字节
- Twitter

题目描述

```
给出集合 [1,2,3,…,n]，其所有元素共有 n! 种排列。

按大小顺序列出所有排列情况，并一一标记，当 n = 3 时，所有排列如下：

"123"
"132"
"213"
"231"
"312"
"321"
给定 n 和 k，返回第 k 个排列。

说明：

给定 n 的范围是 [1, 9]。
给定 k 的范围是[1, n!]。
示例 1:

输入: n = 3, k = 3
输出: "213"
示例 2:

输入: n = 4, k = 9
输出: "2314"
```

思路

LeetCode 上关于排列的题目截止目前（2020-01-06）主要有三种类型：

- 生成全排列
- 生成下一个排列
- 生成第 k 个排列（我们的题目就是这种）

我们不可能求出所有的排列，然后找到第 k 个之后返回。因为排列的组合是 N！，要比 2^n 还要高很多，非常有可能超时。我们必须使用一些巧妙的方法。

我们以题目中的 n= 3 k = 3 为例：

- "123"
- "132"
- "213"
- "231"
- "312"
- "321"

可以看出 1xx，2xx 和 3xx 都有两个，如果你知道阶乘的话，实际上是 2！个。 我们想要找的是第 3 个。那么我们可以直接跳到 2 开头，我们排除了以 1 开头的排列，问题缩小了，我们将 2 加入到结果集，我们不断重复上述的逻辑，知道结果集的元素为 n 即可。

关键点解析

- 找规律
- 排列组合

代码

- 语言支持: Python3

```
import math

class Solution:
    def getPermutation(self, n: int, k: int) -> str:
        res = ""
        candidates = [str(i) for i in range(1, n + 1)]

        while n != 0:
            facto = math.factorial(n - 1)
            # i 表示前面被我们排除的组数，也就是k所在的组的下标
            # k // facto 是不行的， 比如在 k % facto == 0的情况
            i = math.ceil(k / facto) - 1
            # 我们把candidates[i]加入到结果集，然后将其弹出candi
            res += candidates[i]
            candidates.pop(i)
            # k 缩小了 facto *  i
            k -= facto * i
            # 每次迭代我们实际上就处理了一个元素，n 减去 1，当n ==
            n -= 1
        return res
```

题目地址

https://leetcode.com/problems/unique-paths/description/

题目描述

```
A robot is located at the top-left corner of a m x n grid (

The robot can only move either down or right at any point i

How many possible unique paths are there?
```

```
Above is a 7 x 3 grid. How many possible unique paths are t

Note: m and n will be at most 100.

Example 1:

Input: m = 3, n = 2
Output: 3
Explanation:
From the top-left corner, there are a total of 3 ways to re
1. Right -> Right -> Down
2. Right -> Down -> Right
3. Down -> Right -> Right
Example 2:

Input: m = 7, n = 3
Output: 28
```

前置知识

- 动态规划
- 排列组合

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

首先这道题可以用排列组合的解法来解，需要一点高中的知识。

$$C_n^r = \frac{n!}{r!(n-r)!}$$

而这道题我们用动态规划来解。

这是一道典型的适合使用动态规划解决的题目，它和爬楼梯等都属于动态规划中最简单的题目，因此也经常会被用于面试之中。

读完题目你就能想到动态规划的话，建立模型并解决恐怕不是难事。其实我们很容易看出，由于机器人只能右移动和下移动， 因此第[i, j]个格子的总数应该等于[i - 1, j] + [i, j -1]， 因为第[i,j]个格子一定是从左边或者上面移动过来的。

代码大概是：

Python Code:

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        d = [[1] * n for _ in range(m)]

        for col in range(1, m):
            for row in range(1, n):
                d[col][row] = d[col - 1][row] + d[col][row

        return d[m - 1][n - 1]
```

复杂度分析

- 时间复杂度：$O(M * N)$
- 空间复杂度：$O(M * N)$

由于 dp[i][j] 只依赖于左边的元素和上面的元素，因此空间复杂度可以进一步优化， 优化到 O(n).

具体代码请查看代码区。

当然你也可以使用记忆化递归的方式来进行，由于递归深度的原因，性能比上面的方法差不少：

> 直接暴力递归的话可能会超时。

Python3 Code:

```
class Solution:

    @lru_cache
    def uniquePaths(self, m: int, n: int) -> int:
        if m == 1 or n == 1:
            return 1
        cnt = self.uniquePaths(m - 1, n) + self.uniquePaths
        return cnt
```

关键点

- 排列组合原理
- 记忆化递归
- 基本动态规划问题
- 空间复杂度可以进一步优化到 O(n), 这会是一个考点

代码

代码支持 JavaScript，Python3

JavaScript Code:

```
/*
 * @lc app=leetcode id=62 lang=javascript
 *
 * [62] Unique Paths
 *
 * https://leetcode.com/problems/unique-paths/description/
 */
/**
 * @param {number} m
 * @param {number} n
 * @return {number}
 */
var uniquePaths = function (m, n) {
  const dp = Array(n).fill(1);

  for (let i = 1; i < m; i++) {
    for (let j = 1; j < n; j++) {
      dp[j] = dp[j] + dp[j - 1];
    }
  }

  return dp[n - 1];
};
```

Python3 Code:

```
class Solution:

    def uniquePaths(self, m: int, n: int) -> int:
        dp = [1] * n
        for _ in range(1, m):
            for j in range(1, n):
                dp[j] += dp[j - 1]
        return dp[n - 1]
```

复杂度分析

- 时间复杂度：$O(M * N)$
- 空间复杂度：$O(N)$

扩展

你可以做到比$O(M * N)$更快，比$O(N)$更省内存的算法么？这里有一份资料可供参考。

> 提示： 考虑数学

相关题目

- 70. 爬楼梯
- 63. 不同路径 II

题目地址

https://leetcode.com/problems/set-matrix-zeroes/description/

题目描述

```
Given a m x n matrix, if an element is 0, set its entire ro

Example 1:

Input:
[
  [1,1,1],
  [1,0,1],
  [1,1,1]
]
Output:
[
  [1,0,1],
  [0,0,0],
  [1,0,1]
]
Example 2:

Input:
[
  [0,1,2,0],
  [3,4,5,2],
  [1,3,1,5]
]
Output:
[
  [0,0,0,0],
  [0,4,5,0],
  [0,3,1,0]
]
Follow up:

- A straight forward solution using O(mn) space is probably
- A simple improvement uses O(m + n) space, but still not t
- Could you devise a constant space solution?
```

前置知识

- 状态压缩

公司

- 阿里
- 百度
- 字节

思路

符合直觉的想法是，使用一个 m + n 的数组来表示每一行每一列是否”全部是 0“， 先遍历一遍去构建这样的 m + n 数组，然后根据这个 m + n 数组去修改 matrix 即可。

这样的时间复杂度 O(m * n), 空间复杂度 O(m + n).

代码如下：

```
var setZeroes = function (matrix) {
  if (matrix.length === 0) return matrix;
  const m = matrix.length;
  const n = matrix[0].length;
  const zeroes = Array(m + n).fill(false);

  for (let i = 0; i < m; i++) {
    for (let j = 0; j < n; j++) {
      const item = matrix[i][j];

      if (item === 0) {
        zeroes[i] = true;
        zeroes[m + j] = true;
      }
    }
  }

  for (let i = 0; i < m; i++) {
    if (zeroes[i]) {
      matrix[i] = Array(n).fill(0);
    }
  }

  for (let i = 0; i < n; i++) {
    if (zeroes[m + i]) {
      for (let j = 0; j < m; j++) {
        matrix[j][i] = 0;
      }
    }
  }

  return matrix;
};
```

但是这道题目还有一个 follow up， 要求使用 O(1)的时间复杂度。因此上述的方法就不行了。 但是我们要怎么去存取这些信息（哪一行哪一列应该全部为 0）呢?

一种思路是使用第一行第一列的数据来代替上述的 zeros 数组。 这样我们就不必借助额外的存储空间，空间复杂度自然就是 O(1)了。

由于我们不能先操作第一行和第一列， 因此我们需要记录下”第一行和第一列是否全是 0“这样的一个数据，最后根据这个信息去 修改第一行和第一列。

具体步骤如下：

- 记录下”第一行和第一列是否全是 0“这样的一个数据

- 遍历除了第一行和第一列之外的所有的数据，如果是 0，那就更新第一行第一列中对应的元素为 0

  > 你可以把第一行第一列看成我们上面那种解法使用的 m + n 数组。

- 根据第一行第一列的数据，更新 matrix
- 最后根据我们最开始记录的”第一行和第一列是否全是 0“去更新第一行和第一列即可

关键点

- 使用第一行和第一列来替代我们 m + n 数组
- 先记录下”第一行和第一列是否全是 0“这样的一个数据，否则会因为后续对第一行第一列的更新造成数据丢失
- 最后更新第一行第一列

代码

- 语言支持：JS，Python3

```
/*
 * @lc app=leetcode id=73 lang=javascript
 *
 * [73] Set Matrix Zeroes
 */
/**
 * @param {number[][]} matrix
 * @return {void} Do not return anything, modify matrix in-
 */
var setZeroes = function (matrix) {
  if (matrix.length === 0) return matrix;
  const m = matrix.length;
  const n = matrix[0].length;

  // 时间复杂度 O(m * n), 空间复杂度 O(1)
  let firstRow = false; // 第一行是否应该全部为0
  let firstCol = false; // 第一列是否应该全部为0

  for (let i = 0; i < m; i++) {
    for (let j = 0; j < n; j++) {
      const item = matrix[i][j];
      if (item === 0) {
        if (i === 0) {
          firstRow = true;
        }
        if (j === 0) {
          firstCol = true;
        }
        matrix[0][j] = 0;
        matrix[i][0] = 0;
      }
    }
  }

  for (let i = 1; i < m; i++) {
    for (let j = 1; j < n; j++) {
      const item = matrix[i][j];
      if (matrix[0][j] == 0 || matrix[i][0] == 0) {
        matrix[i][j] = 0;
      }
    }
  }

  // 最后处理第一行和第一列

  if (firstRow) {
    for (let i = 0; i < n; i++) {
      matrix[0][i] = 0;
```

```
    }
  }

  if (firstCol) {
    for (let i = 0; i < m; i++) {
      matrix[i][0] = 0;
    }
  }

  return matrix;
};
```

Python3 Code:

直接修改第一行和第一列为 0 的解法：

```
class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place inst
        """
        def setRowZeros(matrix: List[List[int]], i:int) ->
            C = len(matrix[0])
            matrix[i] = [0] * C

        def setColZeros(matrix: List[List[int]], j:int) ->
            R = len(matrix)
            for i in range(R):
                matrix[i][j] = 0

        isCol = False
        R = len(matrix)
        C = len(matrix[0])

        for i in range(R):
            if matrix[i][0] == 0:
                isCol = True
            for j in range(1, C):
                if matrix[i][j] == 0:
                    matrix[i][0] = 0
                    matrix[0][j] = 0
        for j in range(1, C):
            if matrix[0][j] == 0:
                setColZeros(matrix, j)

        for i in range(R):
            if matrix[i][0] == 0:
                setRowZeros(matrix, i)

        if isCol:
            setColZeros(matrix, 0)
```

另一种方法是用一个特殊符合标记需要改变的结果，只要这个特殊标记不在我们的题目数据范围（0 和 1）即可，这里用 None。

```
class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        """
        这题要解决的问题是，必须有个地方记录判断结果，但又不能影响下-
        直接改为0的话，会影响下一步的判断条件；
        因此，有一种思路是先改为None，最后再将None改为0；
        从条件上看，如果可以将第一行、第二行作为记录空间，那么，用No
        """
        rows = len(matrix)
        cols = len(matrix[0])
        # 遍历矩阵，用None记录要改的地方，注意如果是0则要保留，否则
        for r in range(rows):
            for c in range(cols):
                if matrix[r][c] is not None and matrix[r][
                    # 改值
                    for i in range(rows):
                        matrix[i][c] = None if matrix[i][c
                    for j in range(cols):
                        matrix[r][j] = None if matrix[r][j
        # 再次遍历，将None改为0
        for r in range(rows):
            for c in range(cols):
                if matrix[r][c] is None:
                    matrix[r][c] = 0
```

扩展

为什么选择第一行第一列，选择其他行和列可以么？为什么？

题目地址

https://leetcode.com/problems/sort-colors/description/

题目描述

Given an array with n objects colored red, white or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Note: You are not suppose to use the library's sort function for this problem.

Example:

Input: [2,0,2,1,1,0] Output: [0,0,1,1,2,2] Follow up:

A rather straight forward solution is a two-pass algorithm using counting sort. First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's. Could you come up with a one-pass algorithm using only constant space?

前置知识

- 荷兰国旗问题
- 排序

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这个问题是典型的荷兰国旗问题（https://en.wikipedia.org/wiki/Dutch_national_flag_problem）。因为我们可以将红白蓝三色小球想象成条状物，有序排列后正好组成荷兰国旗。

有两种解决思路。

解法一

- 遍历数组，统计红白蓝三色球（0，1，2）的个数
- 根据红白蓝三色球（0，1，2）的个数重排数组

这种思路的时间复杂度：$O(n)$，需要遍历数组两次。

解法二

我们可以把数组分成三部分，前部（全部是 0），中部（全部是 1）和后部（全部是 2）三个部分。每一个元素（红白蓝分别对应 0、1、2）必属于其中之一。将前部和后部各排在数组的前边和后边，中部自然就排好了。

我们用三个指针，设置两个指针 begin 指向前部的末尾的下一个元素（刚开始默认前部无 0，所以指向第一个位置），end 指向后部开头的前一个位置（刚开始默认后部无 2，所以指向最后一个位置），然后设置一个遍历指针 current，从头开始进行遍历。

这种思路的时间复杂度也是$O(n)$, 只需要遍历数组一次。

关键点解析

- 荷兰国旗问题
- counting sort

代码

代码支持： Python3

Python3 Code:

```
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instea
        """
        p0 = cur = 0
        p2 = len(nums) - 1

        while cur <= p2:
            if nums[cur] == 0:
                nums[cur], nums[p0] = nums[p0], nums[cur]
                p0 += 1
                cur += 1
            elif nums[cur] == 2:
                nums[cur], nums[p2] = nums[p2], nums[cur]
                p2 -= 1
            else:
                cur += 1
```

题目地址

https://leetcode.com/problems/subsets/description/

题目描述

```
Given a set of distinct integers, nums, return all possible

Note: The solution set must not contain duplicate subsets.

Example:

Input: nums = [1,2,3]
Output:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

前置知识

- 回溯

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是求集合，并不是 `求极值` ，因此动态规划不是特别切合，因此我们需要考虑别的方法。

这种题目其实有一个通用的解法，就是回溯法。 网上也有大神给出了这种回溯法解题的 通用写法>)，这里的所有的解法使用通用方法解答。 除了这道题目还有很多其他题目可以用这种通用解法，具体的题目见后方相关题目部分。

我们先来看下通用解法的解题思路，我画了一张图：

backtrack @lucifer

> 每一层灰色的部分，表示当前有哪些节点是可以选择的， 红色部分则是选择路径。1，2，3，4，5，6 则分别表示我们的 6 个子集。

通用写法的具体代码见下方代码区。

关键点解析

- 回溯法
- backtrack 解题公式

代码

- 语言支持：JS，C++

JavaScript Code:

```
/*
 * @lc app=leetcode id=78 lang=javascript
 *
 * [78] Subsets
 *
 * https://leetcode.com/problems/subsets/description/
 *
 * algorithms
 * Medium (51.19%)
 * Total Accepted:    351.6K
 * Total Submissions: 674.8K
 * Testcase Example:  '[1,2,3]'
 *
 * Given a set of distinct integers, nums, return all possi
 * power set).
 *
 * Note: The solution set must not contain duplicate subset
 *
 * Example:
 *
 *
 * Input: nums = [1,2,3]
 * Output:
 * [
 *  [3],
 * [1],
 * [2],
 * [1,2,3],
 * [1,3],
 * [2,3],
 * [1,2],
 * []
 * ]
 *
 */
function backtrack(list, tempList, nums, start) {
  list.push([...tempList]);
  for (let i = start; i < nums.length; i++) {
    tempList.push(nums[i]);
    backtrack(list, tempList, nums, i + 1);
    tempList.pop();
  }
}
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var subsets = function (nums) {
```

```
  const list = [];
  backtrack(list, [], nums, 0);
  return list;
};
```

C++ Code：

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        auto ret = vector<vector<int>>();
        auto tmp = vector<int>();
        backtrack(ret, tmp, nums, 0);
        return ret;
    }

    void backtrack(vector<vector<int>>& list, vector<int>&
        list.push_back(tempList);
        for (auto i = start; i < nums.size(); ++i) {
            tempList.push_back(nums[i]);
            backtrack(list, tempList, nums, i + 1);
            tempList.pop_back();
        }
    }
};
```

相关题目

- 39.combination-sum
- 40.combination-sum-ii
- 46.permutations
- 47.permutations-ii
- 90.subsets-ii
- 113.path-sum-ii
- 131.palindrome-partitioning

Problem

https://leetcode.com/problems/word-search/

Problem Description

```
Given a 2D board and a word, find if the word exists in the

The word can be constructed from letters of sequentially ac

Example:

board =
[
  ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']
]

Given word = "ABCCED", return true.
Given word = "SEE", return true.
Given word = "ABCB", return false.
```

Solution

This problem does not give start position, or direction restriction, so

1. Scan board, find starting position with matching word first letter
2. From starting position, DFS (4 (up, down, left, right 4 directions) match word's rest letters
3. For each visited letter, mark it as visited, here use `board[i][j] = '*'` to represent visited.
4. If one direction cannot continue, backtracking, mark start position unvisited, mark `board[i][j] = word[start]`
5. If found any matching, terminate
6. Otherwise, no matching found, return false.

For example:

board, word： `SEE` as below pic:

```
1. Scan board, found board[1,0] = word[0], match word first
2. DFS (up, down, left, right 4 directions)

as below pic:
```

Staring position （1， 0), check whether adjacent cells match word next letter `E` .

```
1. mark current position (1, 0) as visited, board[1][0] = '*
2. Up (0, 0) letter='A' not match,
3. Down (2, 0) letter='A', not match,
4. Left (-1, 0) out of board boundry, not match,
5. right (1, 1) letter='F', not match

as below pic:
```

Didn't find matching from starting position, so

```
1. backtracking, mart start position (1, 0) as unvisited, bo
2. scan board, find next start position (1, 3) which match w

as below pic:
```

New starting position（1，3），check whether adjacent cells match word next letter `E`.

```
1. mark current position (1, 3) as already visited, board[1]
2. Up (0, 3) letter='E', match, continue DFS search, refer p
3. Down (2, 3) letter='E', match, since #2 DFS didn't find w
4. Left (1, 2) letter='C', not match,
5. Right (1, 4) out of board boundry, not match

as below pic:
```

Start position（0，3), DFS，check whether adjacent cells match word next letter `E`

```
1. marck current position (0, 3) already visited, board[0][3
2. Up  (-1, 3) out of board boundry, not match
3. Down (1, 3) already visited,
4. Left (0, 2) letter='C', not match
5. Right (1, 4) out of board boundry, not match

as below pic:
```

Start from position（0, 3）not matching word, start position (2, 3) DFS search:

```
1. Backtracking, mark (0, 3) as unvisited。board[0][3] = 'E'
2. Backtracking to next position (2, 3) , DFS, check whether
3. Up (1, 3) visited, continue
4. Down (3, 3) out of board boundry, not match
5. Left (2, 2) letter='E', match
6. Right (2, 4) out of board boundry, not match

as below pic:
```

Found match with word, return `True` .

Complexity Analysis

- *Time Complexity:* `O(m*n) - m is number of board rows, n is number of board columns`
- *Space Complexity:* `O(1) - no extra space`

> **Note**：if use Set or boolean[][] mark position visited，need extra space `O(m*n)`.

Key Points

- Scan board, find start position which match word first letter, DFS
- Remember visited letter
- Backtracking if not found matching

Code (`Java/Javascript/Python3`)

Java Code

```
public class LC79WordSearch {
  public boolean exist(char[][] board, String word) {
    if (board == null || word == null) return false;
    if (word.length() == 0) return true;
    if (board.length == 0) return false;
    int rows = board.length;
    int cols = board[0].length;
    for (int r = 0; r < rows; r++) {
      for (int c = 0; c < cols; c++) {
        // scan board, start with word first character
        if (board[r][c] == word.charAt(0)) {
          if (helper(board, word, r, c, 0)) {
            return true;
          }
        }
      }
    }
    return false;
  }

  private boolean helper(char[][] board, String word, int r
    // already match word all characters, return true
    if (start == word.length()) return true;
    if (!isValid(board, r, c) ||
        board[r][c] != word.charAt(start)) return false;
    // mark visited
    board[r][c] = '*';
    boolean res = helper(board, word, r + 1, c, start + 1)
        ||  helper(board, word, r, c + 1, start + 1)
        ||  helper(board, word, r - 1, c, start + 1)
        ||  helper(board, word, r, c - 1, start + 1);
    // backtracking to start position
    board[r][c] = word.charAt(start);
    return res;
  }

  private boolean isValid(char[][] board, int r, int c) {
    return r >= 0 && r < board.length && c >= 0 && c < boar
  }
}
```

Python3 Code

```
class Solution:
    def exist(self, board: List[List[str]], word: str) -> b
        m = len(board)
        n = len(board[0])

        def dfs(board, r, c, word, index):
            if index == len(word):
                return True
            if r < 0 or r >= m or c < 0 or c >= n or board
                return False
            board[r][c] = '*'
            res = dfs(board, r - 1, c, word, index + 1) or
            board[r][c] = word[index]
            return res

        for r in range(m):
            for c in range(n):
                if board[r][c] == word[0]:
                    if dfs(board, r, c, word, 0):
                        return True
```

Javascript Code from **@lucifer**

```
/*
 * @lc app=leetcode id=79 lang=javascript
 *
 * [79] Word Search
 */
function DFS(board, row, col, rows, cols, word, cur) {
  // 边界检查
  if (row >= rows || row < 0) return false;
  if (col >= cols || col < 0) return false;

  const item = board[row][col];

  if (item !== word[cur]) return false;

  if (cur + 1 === word.length) return true;

  // If use HashMap keep track visited letters, then need r
  // here we use a little trick
  board[row][col] = null;

  // UP, DOWN, LEFT, RIGHT
  const res =
    DFS(board, row + 1, col, rows, cols, word, cur + 1) ||
    DFS(board, row - 1, col, rows, cols, word, cur + 1) ||
    DFS(board, row, col - 1, rows, cols, word, cur + 1) ||
    DFS(board, row, col + 1, rows, cols, word, cur + 1);

  board[row][col] = item;

  return res;
}
/**
 * @param {character[][]} board
 * @param {string} word
 * @return {boolean}
 */
var exist = function(board, word) {
  if (word.length === 0) return true;
  if (board.length === 0) return false;

  const rows = board.length;
  const cols = board[0].length;

  for (let i = 0; i < rows; i++) {
    for (let j = 0; j < cols; j++) {
      const hit = DFS(board, i, j, rows, cols, word, 0);
      if (hit) return true;
    }
```

```
  }
  return false;
};
```

References

1. Backtracking Wiki

题目地址（删除排序数组中的重复项 II）

https://leetcode-cn.com/problems/remove-duplicates-from-sorted-array-ii/description/

题目描述

```
给定一个排序数组，你需要在原地删除重复出现的元素，使得每个元素最多出现

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

示例 1:

给定 nums = [1,1,1,2,2,3],

函数应返回新长度 length = 5, 并且原数组的前五个元素被修改为 1, 1,

你不需要考虑数组中超出新长度后面的元素。
示例 2:

给定 nums = [0,0,1,1,1,1,2,3,3],

函数应返回新长度 length = 7, 并且原数组的前五个元素被修改为 0, 0,

你不需要考虑数组中超出新长度后面的元素。
说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以“引用”方式传递的，这意味着在函数里修改输入数组对于

你可以想象内部操作如下:

// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

前置知识

- 双指针

公司

- 阿里
- 百度
- 字节

思路

”删除排序“类题目截止到现在（2020-1-15）一共有四道题：

这道题是26.remove-duplicates-from-sorted-array 的进阶版本，唯一的不同是不再是全部元素唯一，而是全部元素不超过 2 次。实际上这种问题可以更抽象一步，即“删除排序数组中的重复项，使得相同数字最多出现 k 次” 。 那么这道题 k 就是 2， 26.remove-duplicates-from-sorted-array 的 k 就是 1。

上一题我们使用了快慢指针来实现，这道题也是一样，只不过逻辑稍有不同。 其实快慢指针本质是读写指针，在这里我们的快指针实际上就是读指针，而慢指针恰好相当于写指针。”快慢指针的说法“便于描述和记忆，“读写指针”的说法更便于理解本质。本文中，以下内容均描述为快慢指针。

- 初始化快慢指针 slow ， fast ，全部指向索引为 0 的元素。
- fast 每次移动一格
- 慢指针选择性移动，即只有写入数据之后才移动。是否写入数据取决于 slow - 2 对应的数字和 fast 对应的数字是否一致。
- 如果一致，我们不应该写。 否则我们就得到了三个相同的数字，不符合题意
- 如果不一致，我们需要将 fast 指针的数据写入到 slow 指针。
- 重复这个过程，直到 fast 走到头，说明我们已无数字可写。

图解（红色的两个数字，表示我们需要比较的两个数字）：

关键点分析

- 快慢指针
- 读写指针
- 删除排序问题

代码

代码支持： Python

Python Code：

```python
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        # 写指针
        i = 0
        K = 2
        for num in nums:
            if i < K or num != nums[i-K]:
                nums[i] = num
                i += 1
        return i
```

基于这套代码，你可以轻易地实现 k 为任意正整数的算法。

相关题目

正如上面所说，相关题目一共有三道（排除自己）。其中一道我们仓库已经讲到了。剩下两道原理类似，但是实际代码和细节有很大不同，原因就在于数组可以随机访问，而链表不行。 感兴趣的可以做一下剩下的两道链表题。

- 1. 删除排序链表中的重复元素 II

- 1. 删除排序链表中的重复元素

83.删除排序链表中的重复元素.py — 常用 (Workspace)

sorted-array-ii.md | 83.删除排序链表中的重复元素.py

ode > 83.删除排序链表中的重复元素.py > Solution > deleteDuplicates

```python
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None


class Solution:
    def deleteDuplicates(self, head: ListNode) -> L
        if not head:
            return head
        slow = head
        fast = head

        while fast:
            if fast.val != slow.val:
                slow = slow.next
                slow.val = fast.val
            fast = fast.next
        slow.next = None
        return head
```

Description | Submission

Accepted

- 165/165 cases passed (52 ms)
- Your runtime beats 53.4 % of python3 submissions
- Your memory usage beats 80.53 % of python3 submissions (13.1 MB)

题目地址

https://leetcode.com/problems/partition-list/description/

题目描述

Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.

You should preserve the original relative order of the nodes in each of the two partitions.

Example:

Input: head = 1->4->3->2->5->2, x = 3 Output: 1->2->2->4->3->5

前置知识

- 链表

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

- 设定两个虚拟节点，dummyHead1 用来保存小于该值的链表，dummyHead2 来保存大于等于该值的链表
- 遍历整个原始链表，将小于该值的放于 dummyHead1 中，其余的放置在 dummyHead2 中

遍历结束后，将 dummyHead2 插入到 dummyHead1 后面

(图片来自： https://github.com/MisterBooo/LeetCodeAnimation)

关键点解析

- 链表的基本操作（遍历）
- 虚拟节点 dummy 简化操作
- 遍历完成之后记得 `currentL1.next = null;` 否则会内存溢出

> 如果单纯的遍历是不需要上面操作的，但是我们的遍历会导致 currentL1.next 和 currentL2.next 中有且仅有一个不是 null， 如果不这么操作的话会导致两个链表成环，造成溢出。

代码

- 语言支持: Javascript，Python3

```
/*
 * @lc app=leetcode id=86 lang=javascript
 *
 * [86] Partition List
 *
 * https://leetcode.com/problems/partition-list/descriptio
 *
 * algorithms
 * Medium (36.41%)
 * Total Accepted:    155.1K
 * Total Submissions: 425.1K
 * Testcase Example:  '[1,4,3,2,5,2]\n3'
 *
 * Given a linked list and a value x, partition it such tha
 * than x come before nodes greater than or equal to x.
 *
 * You should preserve the original relative order of the n
 * two partitions.
 *
 * Example:
 *
 *
 * Input: head = 1->4->3->2->5->2, x = 3
 * Output: 1->2->2->4->3->5
 *
 *
 */
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @param {number} x
 * @return {ListNode}
 */
var partition = function (head, x) {
  const dummyHead1 = {
    next: null,
  };
  const dummyHead2 = {
    next: null,
  };

  let current = {
```

```
    next: head,
  };
  let currentL1 = dummyHead1;
  let currentL2 = dummyHead2;
  while (current.next) {
    current = current.next;
    if (current.val < x) {
      currentL1.next = current;
      currentL1 = current;
    } else {
      currentL2.next = current;
      currentL2 = current;
    }
  }

  currentL2.next = null;

  currentL1.next = dummyHead2.next;

  return dummyHead1.next;
};
```

Python3 Code:

```python
class Solution:
    def partition(self, head: ListNode, x: int) -> ListNode
        """在原链表操作，思路基本一致，只是通过指针进行区分而已"""
        # 在链表最前面设定一个初始node作为锚点，方便返回最后的结果
        first_node = ListNode(0)
        first_node.next = head
        # 设计三个指针，一个指向小于x的最后一个节点，即前后分离点
        # 一个指向当前遍历节点的前一个节点
        # 一个指向当前遍历的节点
        sep_node = first_node
        pre_node = first_node
        current_node = head

        while current_node is not None:
            if current_node.val < x:
                # 注意有可能出现前一个节点就是分离节点的情况
                if pre_node is sep_node:
                    pre_node = current_node
                    sep_node = current_node
                    current_node = current_node.next
                else:
                    # 这段次序比较烧脑
                    pre_node.next = current_node.next
                    current_node.next = sep_node.next
                    sep_node.next = current_node
                    sep_node = current_node
                    current_node = pre_node.next
            else:
                pre_node = current_node
                current_node = pre_node.next

        return first_node.next
```

题目地址

https://leetcode.com/problems/subsets-ii/description/

题目描述

```
Given a collection of integers that might contain duplicate

Note: The solution set must not contain duplicate subsets.

Example:

Input: [1,2,2]
Output:
[
  [2],
  [1],
  [1,2,2],
  [2,2],
  [1,2],
  []
]
```

前置知识

- 回溯

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是求集合，并不是 `求极值` ，因此动态规划不是特别切合，因此我们需要考虑别的方法。

这种题目其实有一个通用的解法，就是回溯法。 网上也有大神给出了这种回溯法解题的 通用写法>)，这里的所有的解法使用通用方法解答。 除了这道题目还有很多其他题目可以用这种通用解法，具体的题目见后方相

关题目部分。

我们先来看下通用解法的解题思路，我画了一张图：

> 图是 78.subsets，都差不多，仅做参考。

通用写法的具体代码见下方代码区。

关键点解析

- 回溯法
- backtrack 解题公式

代码

- 语言支持：JS，C++，Python3

JavaScript Code：

```
function backtrack(list, tempList, nums, start) {
  list.push([...tempList]);
  for (let i = start; i < nums.length; i++) {
    // 和78.subsets的区别在于这道题nums可以有重复
    // 因此需要过滤这种情况
    if (i > start && nums[i] === nums[i - 1]) continue;
    tempList.push(nums[i]);
    backtrack(list, tempList, nums, i + 1);
    tempList.pop();
  }
}
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var subsetsWithDup = function (nums) {
  const list = [];
  backtrack(
    list,
    [],
    nums.sort((a, b) => a - b),
    0,
    []
  );
  return list;
};
```

C++ Code：

```
class Solution {
private:
    void subsetsWithDup(vector<int>& nums, size_t start, ve
        res.push_back(tmp);
        for (auto i = start; i < nums.size(); ++i) {
            if (i > start && nums[i] == nums[i - 1]) contin
            tmp.push_back(nums[i]);
            subsetsWithDup(nums, i + 1, tmp, res);
            tmp.pop_back();
        }
    }
public:
    vector<vector<int>> subsetsWithDup(vector<int>& nums) -
        auto tmp = vector<int>();
        auto res = vector<vector<int>>();
        sort(nums.begin(), nums.end());
        subsetsWithDup(nums, 0, tmp, res);
        return res;
    }
};
```

Python Code:

```
class Solution:
    def subsetsWithDup(self, nums: List[int], sorted: bool=
        """回溯法，通过排序参数避免重复排序"""
        if not nums:
            return [[]]
        elif len(nums) == 1:
            return [[], nums]
        else:
            # 先排序，以便去重
            # 注意，这道题排序花的时间比较多
            # 因此，增加一个参数，使后续过程不用重复排序，可以大幅提
            if not sorted:
                nums.sort()
            # 回溯法
            pre_lists = self.subsetsWithDup(nums[:-1], sort
            all_lists = [i+[nums[-1]] for i in pre_lists] -
            # 去重
            result = []
            for i in all_lists:
                if i not in result:
                    result.append(i)
            return result
```

相关题目

- 39.combination-sum
- 40.combination-sum-ii
- 46.permutations
- 47.permutations-ii
- 78.subsets
- 113.path-sum-ii
- 131.palindrome-partitioning

题目地址

https://leetcode.com/problems/decode-ways/description/

题目描述

```
A message containing letters from A-Z is being encoded to r

'A' -> 1
'B' -> 2
...
'Z' -> 26
Given a non-empty string containing only digits, determine

Example 1:

Input: "12"
Output: 2
Explanation: It could be decoded as "AB" (1 2) or "L" (12).
Example 2:

Input: "226"
Output: 3
Explanation: It could be decoded as "BZ" (2 26), "VF" (22 6
```

前置知识

- 爬楼梯问题
- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目和爬楼梯问题有异曲同工之妙。

这也是一道典型的动态规划题目。我们来思考：

- 对于一个数字来说[1,9]这九个数字能够被识别为一种编码方式
- 对于两个数字来说[10, 26]这几个数字能被识别为一种编码方式

我们考虑用 dp[i]来切分子问题， 那么 dp[i]表示的意思是当前字符串的以索引 i 结尾的子问题。 这样的话，我们最后只需要取 dp[s.length] 就可以解决问题了。

关于递归公式，让我们 `先局部后整体` 。对于局部，我们遍历到一个元素的时候， 我们有两种方式来组成编码方式，第一种是这个元素本身（需要自身是[1,9]）, 第二种是它和前一个元素组成[10, 26]。 用伪代码来表示的话就是： `dp[i] = 以自身去编码（一位） + 以前面的元素和自身去编码（两位）` .这显然是完备的， 这样我们通过层层推导就可以得到结果。

关键点解析

- 爬楼梯问题（我把这种题目统称为爬楼梯问题）

代码

```
/*
 * @lc app=leetcode id=91 lang=javascript
 *
 * [91] Decode Ways
 *
 * https://leetcode.com/problems/decode-ways/description/
 *
 * algorithms
 * Medium (21.93%)
 * Total Accepted:    254.4K
 * Total Submissions: 1.1M
 * Testcase Example:  '"12"'
 *
 * A message containing letters from A-Z is being encoded t
 * following mapping:
 *
 *
 * 'A' -> 1
 * 'B' -> 2
 * ...
 * 'Z' -> 26
 *
 *
 * Given a non-empty string containing only digits, determi
 * of ways to decode it.
 *
 * Example 1:
 *
 *
 * Input: "12"
 * Output: 2
 * Explanation: It could be decoded as "AB" (1 2) or "L" (1
 *
 *
 * Example 2:
 *
 *
 * Input: "226"
 * Output: 3
 * Explanation: It could be decoded as "BZ" (2 26), "VF" (2
 * 6).
 *
 */
/**
 * @param {string} s
 * @return {number}
 */
var numDecodings = function (s) {
```

```
  if (s == null || s.length == 0) {
    return 0;
  }
  const dp = Array(s.length + 1).fill(0);
  dp[0] = 1;
  dp[1] = s[0] !== "0" ? 1 : 0;
  for (let i = 2; i < s.length + 1; i++) {
    const one = +s.slice(i - 1, i);
    const two = +s.slice(i - 2, i);

    if (two >= 10 && two <= 26) {
      dp[i] = dp[i - 2];
    }

    if (one >= 1 && one <= 9) {
      dp[i] += dp[i - 1];
    }
  }

  return dp[dp.length - 1];
};
```

扩展

如果编码的范围不再是 1-26，而是三位的话怎么办？

题目地址

https://leetcode.com/problems/reverse-linked-list-ii/description/

题目描述

Reverse a linked list from position m to n. Do it in one-pass.

Note: 1 ≤ m ≤ n ≤ length of list.

Example:

Input: 1->2->3->4->5->NULL, m = 2, n = 4 Output: 1->4->3->2->5->NULL

前置知识

- 链表

公司

- 阿里
- 腾讯
- 百度
- 字节

思路（四点法）

这道题和206.reverse-linked-list 有点类似，并且这道题是 206 的升级版。 让我们反转某一个区间，而不是整个链表，我们可以将 206 看作本题的特殊情况（special case）。

核心在于**取出需要反转的这一小段链表，反转完后再插入到原先的链表中。**

以本题为例：

反转的是 2,3,4 这三个点，那么我们可以先取出 2，用 cur 指针指向 2，然后当取出 3 的时候，我们将 3 指向 2 的，把 cur 指针前移到 3，依次类推，到 4 后停止，这样我们得到一个新链表 4->3->2, cur 指针指向 4。

对于原链表来说，有两个点的位置很重要，需要用指针记录下来，分别是 1 和 5，把新链表插入的时候需要这两个点的位置。用 pre 指针记录 1 的位置当 4 结点被取走后，5 的位置需要记下来

这样我们就可以把反转后的那一小段链表加入到原链表中

(图片来自网络)

- 首先我们直接返回 head 是不行的。 当 m 不等于 1 的时候是没有问题的，但只要 m 为 1，就会有问题。
- 其次如果链表商都小于 4 的时候，p1，p2，p3，p4 就有可能为空。为了防止 NPE，我们也要充分地判空。

```
class Solution:
    def reverseBetween(self, head: ListNode, m: int, n: int
        pre = None
        cur = head
        i = 0
        p1 = p2 = p3 = p4 = None
        # 一坨逻辑
        if p1:
            p1.next = p3
        else:
            dummy.next = p3
        if p2:
            p2.next = p4
        return head
```

如上代码是不可以的，我们考虑使用 dummy 节点。

```
class Solution:
    def reverseBetween(self, head: ListNode, m: int, n: int)
        pre = None
        cur = head
        i = 0
        p1 = p2 = p3 = p4 = None
        dummy = ListNode(0)
        dummy.next = head
        # 一坨逻辑
        if p1:
            p1.next = p3
        else:
            dummy.next = p3
        if p2:
            p2.next = p4

        return dummy.next
```

关于链表反转部分, 顺序比较重要，我们需要：

- 先 cur.next = pre
- 再 更新 p2 和 p2.next(其中要设置 p2.next = None，否则会互相应用，造成无限循环)
- 最后更新 pre 和 cur

上述的顺序不能错，不然会有问题。原因就在于 `p2.next = None` ，如果这个放在最后，那么我们的 cur 会提前断开。

```
    while cur:
            i += 1
            if i == m - 1:
                p1 = cur
            next = cur.next
            if m < i <= n:
                cur.next = pre

            if i == m:
                p2 = cur
                p2.next = None

            if i == n:
                p3 = cur

            if i == n + 1:
                p4 = cur

            pre = cur
            cur = next
```

关键点解析

- 四点法
- 链表的基本操作
- 考虑特殊情况 m 是 1 或者 n 是链表长度的情况，我们可以采用虚拟节点 dummy 简化操作
- 用四个变量记录特殊节点， 然后操作这四个节点使之按照一定方式连接即可。
- 注意更新 current 和 pre 的位置， 否则有可能出现溢出

代码

我把这个方法称为 `四点法`

语言支持：JS, Python3

JavaScript Code:

```
/*
 * @lc app=leetcode id=92 lang=javascript
 *
 * [92] Reverse Linked List II
 *
 * https://leetcode.com/problems/reverse-linked-list-ii/des
 */
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @param {number} m
 * @param {number} n
 * @return {ListNode}
 */
var reverseBetween = function (head, m, n) {
  // 虚拟节点，简化操作
  const dummyHead = {
    next: head,
  };

  let cur = dummyHead.next; // 当前遍历的节点
  let pre = cur; // 因为要反转，因此我们需要记住前一个节点
  let index = 0; // 链表索引，用来判断是否是特殊位置（头尾位置）

  // 上面提到的四个特殊节点
  let p1 = (p2 = p3 = p4 = null);

  while (cur) {
    const next = cur.next;
    index++;

    // 对（m - n）范围内的节点进行反转
    if (index > m && index <= n) {
      cur.next = pre;
    }

    // 下面四个if都是边界，用于更新四个特殊节点的值
    if (index === m - 1) {
      p1 = cur;
    }
    if (index === m) {
      p2 = cur;
```

```
    }

    if (index === n) {
      p3 = cur;
    }

    if (index === n + 1) {
      p4 = cur;
    }

    pre = cur;

    cur = next;
  }

  // 两个链表合并起来
  (p1 || dummyHead).next = p3; // 特殊情况需要考虑
  p2.next = p4;

  return dummyHead.next;
};
```

Python Code:

```
class Solution:
    def reverseBetween(self, head: ListNode, m: int, n: int
        if not head.next or n == 1:
            return head
        dummy = ListNode()
        dummy.next = head
        pre = None
        cur = head
        i = 0
        p1 = p2 = p3 = p4 = None
        while cur:
            i += 1
            next = cur.next
            if m < i <= n:
                cur.next = pre
            if i == m - 1:
                p1 = cur
            if i == m:
                p2 = cur
            if i == n:
                p3 = cur
            if i == n + 1:
                p4 = cur
            pre = cur
            cur = next
        if not p1:
            dummy.next = p3
        else:
            p1.next = p3
        p2.next = p4
        return dummy.next
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

相关题目

- 25.reverse-nodes-in-k-groups
- 206.reverse-linked-list

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址

https://leetcode.com/problems/binary-tree-inorder-traversal/description/

题目描述

```
Given a binary tree, return the inorder traversal of its no

Example:

Input: [1,null,2,3]
   1
    \
     2
    /
   3

Output: [1,3,2]
Follow up: Recursive solution is trivial, could you do it i
```

前置知识

- 二叉树
- 递归

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

递归的方式相对简单，非递归的方式借助栈这种数据结构实现起来会相对轻松。

如果采用非递归，可以用栈(Stack)的思路来处理问题。

中序遍历的顺序为左-根-右，具体算法为：

- 从根节点开始，先将根节点压入栈

- 然后再将其所有左子结点压入栈，取出栈顶节点，保存节点值
- 再将当前指针移到其右子节点上，若存在右子节点，则在下次循环时又可将其所有左子结点压入栈中， 重复上步骤

(图片来自： https://github.com/MisterBooo/LeetCodeAnimation)

关键点解析

- 二叉树的基本操作（遍历）
  > 不同的遍历算法差异还是蛮大的
- 如果非递归的话利用栈来简化操作
- 如果数据规模不大的话，建议使用递归
- 递归的问题需要注意两点，一个是终止条件，一个如何缩小规模
- 终止条件，自然是当前这个元素是 null（链表也是一样）
- 由于二叉树本身就是一个递归结构， 每次处理一个子树其实就是缩小了规模， 难点在于如何合并结果，这里的合并结果其实就是 `left.concat(mid).concat(right)` , mid 是一个具体的节点，left 和 right `递归求出即可`

代码

- 语言支持：JS，C++，Python3, Java

JavaScript Code：

```
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var inorderTraversal = function (root) {
  // 1. Recursive solution
  // if (!root) return [];
  // const left = root.left ? inorderTraversal(root.left) :
  // const right = root.right ? inorderTraversal(root.right
  // return left.concat([root.val]).concat(right);

  // 2. iterative solutuon
  if (!root) return [];
  const stack = [root];
  const ret = [];
  let left = root.left;

  let item = null; // stack 中弹出的当前项

  while (left) {
    stack.push(left);
    left = left.left;
  }

  while ((item = stack.pop())) {
    ret.push(item.val);
    let t = item.right;

    while (t) {
      stack.push(t);
      t = t.left;
    }
  }

  return ret;
};
```

C++ Code：

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {
 * };
 */
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<TreeNode*> s;
        vector<int> v;
        while (root != NULL || !s.empty()) {
            for (; root != NULL; root = root->left)
                s.push_back(root);
            v.push_back(s.back()->val);
            root = s.back()->right;
            s.pop_back();
        }
        return v;
    }
};
```

Python Code:

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]
        """
        1. 递归法可以一行代码完成，无需讨论；
        2. 迭代法一般需要通过一个栈保存节点顺序，我们这里直接使用列表
          - 首先，我要按照中序遍历的顺序存入栈，这边用的逆序，方便从
          - 在存入栈时加入一个是否需要深化的参数
          - 在回头取值时，这个参数应该是否，即直接取值
          - 简单调整顺序，即可实现前序和后序遍历
        """
        # 递归法
        # if root is None:
        #     return []
        # return self.inorderTraversal(root.left)\
        #     + [root.val]\
        #     + self.inorderTraversal(root.right)
        # 迭代法
        result = []
        stack = [(1, root)]
        while stack:
            go_deeper, node = stack.pop()
            if node is None:
                continue
            if go_deeper:
                # 左右节点还需继续深化，并且入栈是先右后左
                stack.append((1, node.right))
                # 节点自身已遍历，回头可以直接取值
                stack.append((0, node))
                stack.append((1, node.left))
            else:
                result.append(node.val)
        return result
```

Java Code:

- recursion

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    List<Integer> res = new LinkedList<>();
    public List<Integer> inorderTraversal(TreeNode root) {
        inorder(root);
        return res;
    }

    public void inorder (TreeNode root) {
        if (root == null) return;

        inorder(root.left);

        res.add(root.val);

        inorder(root.right);
    }
}
```

- iteration

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<> ();
        Stack<TreeNode> stack = new Stack<> ();

        while (root != null || !stack.isEmpty()) {
            while (root != null) {
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
            res.add(root.val);
            root = root.right;
        }
        return res;
    }
}
```

相关专题

- 二叉树的遍历

题目地址（95. 不同的二叉搜索树 II）

https://leetcode-cn.com/problems/unique-binary-search-trees-ii/description/

题目描述

```
给定一个整数 n，生成所有由 1 ... n 为节点所组成的二叉搜索树。

示例：

输入：3
输出：
[
  [1,null,3,2],
  [3,2,null,1],
  [3,1,null,null,2],
  [2,1,3],
  [1,null,2,null,3]
]
解释：
以上的输出对应以下 5 种不同结构的二叉搜索树：

   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

前置知识

- 二叉搜索树
- 分治

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这是一个经典的使用分治思路的题目。基本思路和96.unique-binary-search-trees一样。

只是我们需要求解的不仅仅是数字，而是要求解所有的组合。我们假设问题 f(1, n) 是求解 1 到 n（两端包含）的所有二叉树。那么我们的目标就是求解 f(1, n)。

我们将问题进一步划分为子问题，假如左侧和右侧的树分别求好了，我们是不是只要运用组合的原理，将左右两者进行做和就好了，我们需要两层循环来完成这个过程。

关键点解析

- 分治法

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def generateTrees(self, n: int) -> List[TreeNode]:
        if not n:
            return []

        def generateTree(start, end):
            if start > end:
                return [None]
            res = []
            for i in range(start, end + 1):
                ls = generateTree(start, i - 1)
                rs = generateTree(i + 1, end)
                for l in ls:
                    for r in rs:
                        node = TreeNode(i)
                        node.left = l
                        node.right = r
                        res.append(node)

            return res

        return generateTree(1, n)
```

相关题目

- 96.unique-binary-search-trees

题目地址（96. 不同的二叉搜索树）

https://leetcode-cn.com/problems/unique-binary-search-trees-ii/description/

题目描述

```
给定一个整数 n，求以 1 ... n 为节点组成的二叉搜索树有多少种?

示例:

输入: 3
输出: 5
解释:
给定 n = 3, 一共有 5 种不同结构的二叉搜索树:

   1          3     3      2      1
    \        /     /      / \      \
     3      2     1      1   3      2
    /      /       \                 \
   2      1         2                 3
```

前置知识

- 二叉搜索树
- 分治

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这是一个经典的使用分治思路的题目。

对于数字 n ，我们可以 1- n 这样的离散整数分成左右两部分。我们不妨设其分别为 A 和 B。那么问题转化为 A 和 B 所能组成的 BST 的数量的笛卡尔积。而对于 A 和 B 以及原问题除了规模，没有不同，这不就是分治

思路么？至于此，我们只需要考虑边界即可，边界很简单就是 n 小于等于 1 的时候，我们返回 1。

具体来说：

- 生成一个[1:n + 1] 的数组
- 我们遍历一次数组，对于每一个数组项，我们执行以下逻辑
- 对于每一项，我们都假设其是断点。断点左侧的是 A，断点右侧的是 B。
- 那么 A 就是 i - 1 个数， 那么 B 就是 n - i 个数
- 我们递归，并将 A 和 B 的结果相乘即可。

> 其实我们发现，题目的答案只和 n 有关，和具体 n 个数的具体组成，只要是有序数组即可。

题目没有明确 n 的取值范围，我们试一下暴力递归。

代码（Python3）：

```
class Solution:
    def numTrees(self, n: int) -> int:
        if n <= 1:
            return 1
        res = 0
        for i in range(1, n + 1):
            res += self.numTrees(i - 1) * self.numTrees(n -
        return res
```

上面的代码会超时，并没有栈溢出，因此我们考虑使用 hashmap 来优化，代码见下方代码区。

关键点解析

- 分治法
- 笛卡尔积
- 记忆化递归

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    visited = dict()

    def numTrees(self, n: int) -> int:
        if n in self.visited:
            return self.visited.get(n)
        if n <= 1:
            return 1
        res = 0
        for i in range(1, n + 1):
            res += self.numTrees(i - 1) * self.numTrees(n -
        self.visited[n] = res
        return res
```

复杂度分析

- 时间复杂度：一层循环是 N，另外递归深度是 N，因此总的时间复杂度是 $O(N^2)$
- 空间复杂度：递归的栈深度和visited 的大小都是 N，因此总的空间复杂度为 $O(N)$

相关题目

- 95.unique-binary-search-trees-ii

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经30K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode.com/problems/validate-binary-search-tree/description/

题目描述

```
Given a binary tree, determine if it is a valid binary sear

Assume a BST is defined as follows:

The left subtree of a node contains only nodes with keys le
The right subtree of a node contains only nodes with keys g
Both the left and right subtrees must also be binary searc


Example 1:

    2
   / \
  1   3

Input: [2,1,3]
Output: true
Example 2:

    5
   / \
  1   4
     / \
    3   6

Input: [5,1,4,null,null,3,6]
Output: false
Explanation: The root node's value is 5 but its right chil
```

前置知识

- 中序遍历

公司

- 阿里
- 腾讯

- 百度
- 字节

思路

中序遍历

这道题是让你验证一棵树是否为二叉查找树（BST）。 由于中序遍历的性质 `如果一个树遍历的结果是有序数组，那么他也是一个二叉查找树(BST)`，我们只需要中序遍历，然后两两判断是否有逆序的元素对即可，如果有，则不是 BST，否则即为一个 BST。

定义法

根据定义，一个结点若是在根的左子树上，那它应该小于根结点的值而大于左子树最小值；若是在根的右子树上，那它应该大于根结点的值而小于右子树最大值。也就是说，每一个结点必须落在某个取值范围：

1. 根结点的取值范围为（考虑某个结点为最大或最小整数的情况）：(long_min, long_max)
2. 左子树的取值范围为：(current_min, root.value)
3. 右子树的取值范围为：(root.value, current_max)

关键点解析

- 二叉树的基本操作（遍历）
- 中序遍历一个二叉查找树（BST）的结果是一个有序数组
- 如果一个树遍历的结果是有序数组，那么他也是一个二叉查找树(BST)

代码

中序遍历

- 语言支持：JS，C++, Java

JavaScript Code：

```javascript
/*
 * @lc app=leetcode id=98 lang=javascript
 *
 * [98] Validate Binary Search Tree
 */
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {boolean}
 */
var isValidBST = function (root) {
  if (root === null) return true;
  if (root.left === null && root.right === null) return tru
  const stack = [root];
  let cur = root;
  let pre = null;

  function insertAllLefts(cur) {
    while (cur && cur.left) {
      const l = cur.left;
      stack.push(l);
      cur = l;
    }
  }
  insertAllLefts(cur);

  while ((cur = stack.pop())) {
    if (pre && cur.val <= pre.val) return false;
    const r = cur.right;

    if (r) {
      stack.push(r);
      insertAllLefts(r);
    }
    pre = cur;
  }

  return true;
};
```

C++ Code：

```
// 递归
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        TreeNode* prev = nullptr;
        return validateBstInorder(root, prev);
    }

private:
    bool validateBstInorder(TreeNode* root, TreeNode*& prev
        if (root == nullptr) return true;
        if (!validateBstInorder(root->left, prev)) return f
        if (prev != nullptr && prev->val >= root->val) retu
        prev = root;
        return validateBstInorder(root->right, prev);
    }
};

// 迭代
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        auto s = vector<TreeNode*>();
        TreeNode* prev = nullptr;
        while (root != nullptr || !s.empty()) {
            while (root != nullptr) {
                s.push_back(root);
                root = root->left;
            }
            root = s.back();
            s.pop_back();
            if (prev != nullptr && prev->val >= root->val)
            prev = root;
            root = root->right;
        }
        return true;
    }
};
```

Java Code:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public boolean isValidBST(TreeNode root) {
        Stack<Integer> stack = new Stack<> ();
        TreeNode previous = null;

        while (root != null || !stack.isEmpty()) {
            while (root != null) {
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
            if (previous != null && root.val <= previous.va
            previous = root;
            root = root.right;
        }
        return true;
    }
}
```

定义法

- 语言支持：C++，Python3, Java, JS

C++ Code：

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {
 * };
 */
// 递归
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        return helper(root, LONG_MIN, LONG_MAX);
    }
private:
    bool helper(const TreeNode* root, long min, long max) -
        if (root == nullptr) return true;
        if (root->val >= max || root->val <= min) return fa
        return helper(root->left, min, root->val) && helpe
    }
};

// 循环
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        if (root == nullptr) return true;
        auto ranges = queue<pair<long, long>>();
        ranges.push(make_pair(LONG_MIN, LONG_MAX));
        auto nodes = queue<const TreeNode*>();
        nodes.push(root);
        while (!nodes.empty()) {
            auto sz = nodes.size();
            for (auto i = 0; i < sz; ++i) {
                auto range = ranges.front();
                ranges.pop();
                auto n = nodes.front();
                nodes.pop();
                if (n->val >= range.second || n->val <= ra
                    return false;
                }
                if (n->left != nullptr) {
                    ranges.push(make_pair(range.first, n->
                    nodes.push(n->left);
                }
                if (n->right != nullptr) {
                    ranges.push(make_pair(n->val, range.se
```

```
                    nodes.push(n->right);
                }
            }
        }
        return true;
    }
};
```

Python Code:

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def isValidBST(self, root: TreeNode, area: tuple=(-floa
        """思路如上面的分析，用Python表达会非常直白
        :param root TreeNode 节点
        :param area tuple 取值区间
        """
        if root is None:
            return True

        is_valid_left = root.left is None or\
                    (root.left.val < root.val and area[0] <
        is_valid_right = root.right is None or\
                    (root.right.val > root.val and area[0] <

        # 左右节点都符合，说明本节点符合要求
        is_valid = is_valid_left and is_valid_right

        # 递归下去
        return is_valid\
            and self.isValidBST(root.left, (area[0], root.v
            and self.isValidBST(root.right, (root.val, area
```

Java Code:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public boolean isValidBST(TreeNode root) {
        return helper(root, null, null);
    }

    private boolean helper(TreeNode root, Integer lower, In
        if (root == null) return true;

        if (lower != null && root.val <= lower) return fals
        if (higher != null && root.val >= higher) return fa

        if (!helper(root.left, lower, root.val)) return fal
        if (!helper(root.right, root.val, higher)) return t

        return true;
    }
}
```

JavaScript Code:

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {boolean}
 */
var isValidBST = function (root) {
  if (!root) return true;
  return valid(root);
};

function valid(root, min = -Infinity, max = Infinity) {
  if (!root) return true;
  const val = root.val;
  if (val <= min) return false;
  if (val >= max) return false;
  return valid(root.left, min, val) && valid(root.right, va
}
```

相关题目

230.kth-smallest-element-in-a-bst

题目地址

https://leetcode.com/problems/binary-tree-level-order-traversal/description/

题目描述

```
给你一个二叉树，请你返回其按 层序遍历 得到的节点值。 （即逐层地，从左到




示例：
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]
]
```

前置知识

- 队列

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这是一个典型的二叉树遍历问题， 关于二叉树遍历，我总结了一个专题，大家可以先去看下那个，然后再来刷这道题。

这道题可以借助 `队列` 实现，首先把root入队，然后入队一个特殊元素Null(来表示每层的结束)。

然后就是while(queue.length), 每次处理一个节点，都将其子节点（在这里是left和right）放到队列中。

然后不断的出队， 如果出队的是null，则表式这一层已经结束了，我们就继续push一个null。

如果不入队特殊元素Null来表示每层的结束，则在while循环开始时保存当前队列的长度，以保证每次只遍历一层（参考下面的C++ Code）。

> 如果采用递归方式，则需要将当前节点，当前所在的level以及结果数组传递给递归函数。在递归函数中，取出节点的值，添加到level参数对应结果数组元素中（参考下面的C++ Code 或 Python Code）。

关键点解析

- 队列
- 队列中用Null(一个特殊元素)来划分每层
- 树的基本操作- 遍历 - 层次遍历（BFS）
- 注意塞入null的时候，判断一下当前队列是否为空，不然会无限循环

代码

- 语言支持：JS，C++，Python3

Javascript Code:

```
/**
 * @param {TreeNode} root
 * @return {number[][]}
 */
var levelOrder = function(root) {
  if (!root) return [];
  const items = []; // 存放所有节点
  const queue = [root, null]; // null 简化操作
  let levelNodes = []; // 存放每一层的节点

  while (queue.length > 0) {
    const t = queue.shift();

    if (t) {
      levelNodes.push(t.val)
      if (t.left) {
        queue.push(t.left);
      }
      if (t.right) {
        queue.push(t.right);
      }
    } else { // 一层已经遍历完了
      items.push(levelNodes);
      levelNodes = [];
      if (queue.length > 0) {
        queue.push(null)
      }
    }
  }

  return items;
};
```

C++ Code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {
 * };
 */

 // 迭代
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        auto ret = vector<vector<int>>();
        if (root == nullptr) return ret;
        auto q = vector<TreeNode*>();
        q.push_back(root);
        auto level = 0;
        while (!q.empty())
        {
            auto sz = q.size();
            ret.push_back(vector<int>());
            for (auto i = 0; i < sz; ++i)
            {
                auto t = q.front();
                q.erase(q.begin());
                ret[level].push_back(t->val);
                if (t->left != nullptr) q.push_back(t->left
                if (t->right != nullptr) q.push_back(t->rig
            }
            ++level;
        }
        return ret;
    }
};

// 递归
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> v;
        levelOrder(root, 0, v);
        return v;
    }
private:
    void levelOrder(TreeNode* root, int level, vector<vecto
        if (root == NULL) return;
```

```
        if (v.size() < level + 1) v.resize(level + 1);
        v[level].push_back(root->val);
        levelOrder(root->left, level + 1, v);
        levelOrder(root->right, level + 1, v);
    }
};
```

Python Code：

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]
        """递归法"""
        if root is None:
            return []

        result = []

        def add_to_result(level, node):
            """递归函数
            :param level int 当前在二叉树的层次
            :param node TreeNode 当前节点
            """
            if level > len(result) - 1:
                result.append([])

            result[level].append(node.val)
            if node.left:
                add_to_result(level+1, node.left)
            if node.right:
                add_to_result(level+1, node.right)

        add_to_result(0, root)
        return result
```

复杂度分析

- 时间复杂度：$O(N)$，其中N为树中节点总数。
- 空间复杂度：$O(N)$，其中N为树中节点总数。

更多题解可以访问我的LeetCode题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经30K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

扩展

实际上这道题方法很多， 比如经典的三色标记法。

相关题目

- 103.binary-tree-zigzag-level-order-traversal
- 104.maximum-depth-of-binary-tree

相关专题

- 二叉树的遍历

题目地址

https://leetcode.com/problems/binary-tree-zigzag-level-order-traversal/description/

题目描述

和leetcode 102 基本是一样的，思路是完全一样的。

```
Given a binary tree, return the zigzag level order traversa

For example:
Given binary tree [3,9,20,null,null,15,7],
    3
   / \
  9  20
    /  \
   15   7
return its zigzag level order traversal as:
[
  [3],
  [20,9],
  [15,7]
]
```

前置知识

- 队列

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题可以借助 `队列` 实现，首先把root入队，然后入队一个特殊元素Null(来表示每层的结束)。

然后就是while(queue.length), 每次处理一个节点，都将其子节点（在这里是left和right）放到队列中。

然后不断的出队， 如果出队的是null，则表式这一层已经结束了，我们就继续push一个null。

关键点解析

- 队列
- 队列中用Null(一个特殊元素)来划分每层
- 树的基本操作- 遍历 - 层次遍历（BFS）

代码

- 语言支持：JS，C++

JavaScript Code：

```
/*
 * @lc app=leetcode id=103 lang=javascript
 *
 * [103] Binary Tree Zigzag Level Order Traversal
 *
 * https://leetcode.com/problems/binary-tree-zigzag-level-o
 *
 * algorithms
 * Medium (40.57%)
 * Total Accepted:    201.2K
 * Total Submissions: 493.7K
 * Testcase Example:  '[3,9,20,null,null,15,7]'
 *
 * Given a binary tree, return the zigzag level order trave
 * values. (ie, from left to right, then right to left for
 * alternate between).
 *
 *
 * For example:
 * Given binary tree [3,9,20,null,null,15,7],
 *
 *    3
 *   / \
 *  9  20
 *    /  \
 *   15   7
 *
 *
 *
 * return its zigzag level order traversal as:
 *
 * [
 *  [3],
 *  [20,9],
 *  [15,7]
 * ]
 *
 *
 */
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
```

```
 * @return {number[][]}
 */
var zigzagLevelOrder = function(root) {
  if (!root) return [];
  const items = [];
  let isOdd = true;
  let levelNodes = [];

  const queue = [root, null];


  while(queue.length > 0) {
      const t = queue.shift();

      if (t) {
          levelNodes.push(t.val)
          if (t.left) {
            queue.push(t.left)
          }
          if (t.right) {
            queue.push(t.right)
          }
      } else {
        if (!isOdd) {
          levelNodes = levelNodes.reverse();
        }
        items.push(levelNodes)
        levelNodes = [];
        isOdd = !isOdd;
        if (queue.length > 0) {
            queue.push(null);
        }
      }
  }

  return items

};
```

C++ Code：

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {
 * };
 */
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        auto ret = vector<vector<int>>();
        if (root == nullptr) return ret;
        auto queue = vector<const TreeNode*>{root};
        auto isOdd = true;
        while (!queue.empty()) {
            auto sz = queue.size();
            auto level = vector<int>();
            for (auto i = 0; i < sz; ++i) {
                auto n = queue.front();
                queue.erase(queue.begin());
                if (isOdd) level.push_back(n->val);
                else level.insert(level.begin(), n->val);
                if (n->left != nullptr) queue.push_back(n-
                if (n->right != nullptr) queue.push_back(n-
            }
            isOdd = !isOdd;
            ret.push_back(level);
        }
        return ret;
    }
};
```

拓展

由于二叉树是递归结构，因此，可以采用递归的方式来处理。在递归时需要保留当前的层次信息（从0开始），作为参数传递给下一次递归调用。

描述

1. 当前层次为偶数时，将当前节点放到当前层的结果数组尾部
2. 当前层次为奇数时，将当前节点放到当前层的结果数组头部
3. 递归对左子树进行之字形遍历，层数参数为当前层数+1
4. 递归对右子树进行之字形遍历，层数参数为当前层数+1

C++实现

```
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        auto ret = vector<vector<int>>();
        zigzagLevelOrder(root, 0, ret);
        return ret;
    }
private:
    void zigzagLevelOrder(const TreeNode* root, int level,
        if (root == nullptr || level < 0) return;
        if (ret.size() <= level) {
            ret.push_back(vector<int>());
        }
        if (level % 2 == 0) ret[level].push_back(root->val
        else ret[level].insert(ret[level].begin(), root->va
        zigzagLevelOrder(root->left, level + 1, ret);
        zigzagLevelOrder(root->right, level + 1, ret);
    }
};
```

相关题目

- 102.binary-tree-level-order-traversal
- 104.maximum-depth-of-binary-tree

问题地址/Problem URL

https://leetcode.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/

问题介绍/Problem Description

Given preorder and inorder traversal of a tree, construct the binary tree.

Note: You may assume that duplicates do not exist in the tree.

For example, given

```
preorder = [3,9,20,15,7]
inorder = [9,3,15,20,7]
```

Return the following binary tree:

```
    3
   / \
  9  20
    /  \
   15   7
```

前置知识

- 二叉树

公司

- 阿里
- 腾讯
- 百度
- 字节

思路/Thinking Path

目标是构造二叉树。

构造二叉树需要根的值、左子树和右子树。

此问题可被抽象为：从前序遍历和中序遍历中找到根节点、左子树和右子树。

先找根： 由前序遍历的性质，第 `0` 个节点为当前树的根节点。 再找左右子树： 在中序遍历中找到这个根节点，设其下标为 `i` 。由中序遍历的性质， `0 ~ i-1` 是左子树的中序遍历， `i+1 ~ inorder.length-1` 是右子树的中序遍历。

然后递归求解，终止条件是左右子树为 `null` 。

We are going to construct a binary tree from its preorder and inorder traversal.

To build a binary tree, it requires us to creact a new `TreeNode` as the root with filling in the root value. And then, find its left child and right child recursively until the left or right child is `null` .

Now this problem is abstracted as how to find the root node, left child and right child from the preorder traversal and inorder traversal.

In preorder traversal, the first node (`preorder[0]`) is the root of current binary tree. In inorder traversal, find the location of this root which is `i` . The left sub-tree is `0 to i-1` and the right sub-tree is `i+1 to inorder.length-1` in inorder traversal.

Then applying the previous operation to the left and right sub-trees.

关键解析/Key Points

如何在前序遍历的数组里找到左右子树的：

- 根据前序遍历的定义可知，每个当前数组的第一个元素就是当前子树的根节点的值
- 在中序遍历数组中找到这个值，设其下标为 `inRoot`
  - 当前中序遍历数组的起点 `inStart` 到 `inRoot` 之间就是左子树，其长度 `leftChldLen` 为 `inRoot-inStart`
  - 当前中序遍历数组的终点 `inEnd` 和 `inRoot` 之间就是右子树
- 前序遍历和中序遍历中左子树的长度是相等的，所以在前序遍历数组中，根节点下标 `preStart` 往后数 `leftChldLen` 即为左子树的最后一个节点，其下标为 `preStart+leftChldLen` ，右子树的第一个节点下标为 `preStart+leftChldLen+1` 。

PLEASE READ THE CODE BEFORE READING THIS PART

If you can't figure out how to get the index of the left and right child, please read this.

- index of current node in preorder array is preStart(or whatever your call it), it's the root of a subtree.
- according to the properties of preoder traversal, all right sub-tree nodes are behine all left sub-tree nodes. The length of left sub-tree

can help us to divide left and right sub-trees.

- the length of left sub-tree can be find in the inorder traversal. The location of current node is `inRoot` (or whatever your call it). The start index of current inorder array is `inStart` (or whatever your call it). So, the lenght of the left sub-tree is `leftChldLen = inRoot - inStart`.

代码/Code

- Java

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
  public TreeNode buildTree(int[] preorder, int[] inorder)
    if (preorder.length != inorder.length) return null;

    HashMap<Integer, Integer> map = new HashMap<> ();

    for (int i=0; i<inorder.length; i++) {
      map.put(inorder[i], i);
    }

    return helper(preorder, 0, preorder.length-1, inorder,
  }

  public TreeNode helper(int[] preorder, int preStart, int
    if (preStart>preEnd || inStart>inEnd) return null;

    TreeNode root = new TreeNode(preorder[prestart]);
    int inRoot = map.get(preorder[preStart]);
    int leftChldLen = inRoot - inStart;

    root.left = helper(preorder, preStart+1, preStart+left(
    root.left = helper(preorder, preStart+leftChldLen+1, pr

    return root;
  }
}
```

题目地址

https://leetcode.com/problems/path-sum-ii/description/

题目描述

```
Given a binary tree and a sum, find all root-to-leaf paths

Note: A leaf is a node with no children.

Example:

Given the below binary tree and sum = 22,

      5
     / \
    4   8
   /   / \
  11  13  4
 /  \    / \
7    2  5   1
Return:

[
   [5,4,11,2],
   [5,8,4,5]
]
```

前置知识

- 回溯法

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是求集合，并不是 `求值` ，而是枚举所有可能，因此动态规划不是特别切合，因此我们需要考虑别的方法。

这种题目其实有一个通用的解法，就是回溯法。 网上也有大神给出了这种回溯法解题的 通用写法)，这里的所有的解法使用通用方法解答。 除了这道题目还有很多其他题目可以用这种通用解法，具体的题目见后方相关题目部分。

我们先来看下通用解法的解题思路，我画了一张图：

backtrack @lucifer

> 图是 78.subsets，都差不多，仅做参考。

通用写法的具体代码见下方代码区。

关键点解析

- 回溯法
- backtrack 解题公式

代码

- 语言支持：JS，C++，Python3

JavaScript Code：

```
/*
 * @lc app=leetcode id=113 lang=javascript
 *
 * [113] Path Sum II
 */
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
function backtrack(root, sum, res, tempList) {
  if (root === null) return;
  if (root.left === null && root.right === null && sum ===
    return res.push([...tempList, root.val]);

  tempList.push(root.val);
  backtrack(root.left, sum - root.val, res, tempList);

  backtrack(root.right, sum - root.val, res, tempList);
  tempList.pop();
}
/**
 * @param {TreeNode} root
 * @param {number} sum
 * @return {number[][]}
 */
var pathSum = function(root, sum) {
  if (root === null) return [];
  const res = [];
  backtrack(root, sum, res, []);
  return res;
};
```

C++ Code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {
 * };
 */
class Solution {
public:
    vector<vector<int>> pathSum(TreeNode* root, int sum) {
        auto ret = vector<vector<int>>();
        auto temp = vector<int>();
        backtrack(root, sum, ret, temp);
        return ret;
    }
private:
    void backtrack(const TreeNode* root, int sum, vector<ve
        if (root == nullptr) return;
        tempList.push_back(root->val);
        if (root->val == sum && root->left == nullptr && ro
            ret.push_back(tempList);
        } else {
            backtrack(root->left, sum - root->val, ret, tem
            backtrack(root->right, sum - root->val, ret, te
        }
        tempList.pop_back();
    }
};
```

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def pathSum(self, root: TreeNode, sum: int) -> List[Lis
        if not root:
            return []

        result = []

        def trace_node(pre_list, left_sum, node):
            new_list = pre_list.copy()
            new_list.append(node.val)
            if not node.left and not node.right:
                # 这个判断可以和上面的合并，但分开写会快几毫秒，可
                if left_sum == node.val:
                    result.append(new_list)
            else:
                if node.left:
                    trace_node(new_list, left_sum-node.val,
                if node.right:
                    trace_node(new_list, left_sum-node.val,

        trace_node([], sum, root)
        return result
```

相关题目

- 39.combination-sum
- 40.combination-sum-ii
- 46.permutations
- 47.permutations-ii
- 78.subsets
- 90.subsets-ii
- 131.palindrome-partitioning

题目地址

https://leetcode.com/problems/sum-root-to-leaf-numbers/description/

题目描述

```
Given a binary tree containing digits from 0-9 only, each r

An example is the root-to-leaf path 1->2->3 which represent

Find the total sum of all root-to-leaf numbers.

Note: A leaf is a node with no children.

Example:

Input: [1,2,3]
    1
   / \
  2   3
Output: 25
Explanation:
The root-to-leaf path 1->2 represents the number 12.
The root-to-leaf path 1->3 represents the number 13.
Therefore, sum = 12 + 13 = 25.
Example 2:

Input: [4,9,0,5,1]
    4
   / \
  9   0
 / \
5   1
Output: 1026
Explanation:
The root-to-leaf path 4->9->5 represents the number 495.
The root-to-leaf path 4->9->1 represents the number 491.
The root-to-leaf path 4->0 represents the number 40.
Therefore, sum = 495 + 491 + 40 = 1026.
```

前置知识

- 递归

公司

- 阿里
- 百度
- 字节

思路

这是一道非常适合训练递归的题目。虽然题目不难，但是要想一次写正确，并且代码要足够优雅却不是很容易。

这里我们的思路是定一个递归的helper函数，用来帮助我们完成递归操作。 递归函数的功能是将它的左右子树相加，注意这里不包括这个节点本身，否则会多加， 我们其实关注的就是叶子节点的值，然后通过层层回溯到root，返回即可。

整个过程如图所示：

那么数字具体的计算逻辑，如图所示，相信大家通过这个不难发现规律：

关键点解析

- 递归分析

代码

- 语言支持：JS，C++，Python

JavaScipt Code：

```
/*
 * @lc app=leetcode id=129 lang=javascript
 *
 * [129] Sum Root to Leaf Numbers
 */
function helper(node, cur) {
  if (node === null) return 0;
  const next = node.val + cur * 10;

  if (node.left === null && node.right === null) return nex

  const l = helper(node.left, next);
  const r = helper(node.right, next);

  return l + r;
}
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
var sumNumbers = function(root) {
  // tag: `tree` `dfs` `math`
  return helper(root, 0);
};
```

C++ Code：

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {
 * };
 */
class Solution {
public:
    int sumNumbers(TreeNode* root) {
        return helper(root, 0);
    }
private:
    int helper(const TreeNode* root, int val) {
        if (root == nullptr) return 0;
        auto ret = root->val + val * 10;
        if (root->left == nullptr && root->right == nullptr
            return ret;
        auto l = helper(root->left, ret);
        auto r = helper(root->right, ret);
        return l + r;
    }
};
```

Python Code:

```python
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def sumNumbers(self, root: TreeNode) -> int:

        def helper(node, cur_val):
            if not node: return 0
            next_val = cur_val * 10 + node.val

            if not (node.left or node.right):
                return next_val

            left_val = helper(node.left, next_val)
            right_val = helper(node.right, next_val)

            return left_val + right_val

        return helper(root, 0)
```

拓展

通常来说，可以利用队列、栈等数据结构将递归算法转为递推算法。

描述

使用两个队列：

1. 当前和队列：保存上一层每个结点的当前和（比如49和40）
2. 结点队列：保存当前层所有的非空结点

每次循环按层处理结点队列。处理步骤：

1. 从结点队列取出一个结点
2. 从当前和队列将上一层对应的当前和取出来
3. 若左子树非空，则将该值乘以10加上左子树的值，并添加到当前和队列中
4. 若右子树非空，则将该值乘以10加上右子树的值，并添加到当前和队列中
5. 若左右子树均为空时，将该节点的当前和加到返回值中

实现

- 语言支持：C++，Python

C++ Code：

```
class Solution {
public:
    int sumNumbers(TreeNode* root) {
        if (root == nullptr) return 0;
        auto ret = 0;
        auto runningSum = vector<int>{root->val};
        auto queue = vector<const TreeNode*>{root};
        while (!queue.empty()) {
            auto sz = queue.size();
            for (auto i = 0; i < sz; ++i) {
                auto n = queue.front();
                queue.erase(queue.begin());
                auto tmp = runningSum.front();
                runningSum.erase(runningSum.begin());
                if (n->left != nullptr) {
                    runningSum.push_back(tmp * 10 + n->left
                    queue.push_back(n->left);
                }
                if (n->right != nullptr) {
                    runningSum.push_back(tmp * 10 + n->righ
                    queue.push_back(n->right);
                }
                if (n->left == nullptr && n->right == nullp
                    ret += tmp;
                }
            }
        }
        return ret;
    }
};
```

Python Code:

```
class Solution:
    def sumNumbers(self, root: TreeNode) -> int:
        if not root: return 0
        result = 0
        node_queue, sum_queue = [root], [root.val]
        while node_queue:
            for i in node_queue:
                cur_node = node_queue.pop(0)
                cur_val = sum_queue.pop(0)
                if cur_node.left:
                    node_queue.append(cur_node.left)
                    sum_queue.append(cur_val * 10 + cur_noc
                if cur_node.right:
                    node_queue.append(cur_node.right)
                    sum_queue.append(cur_val * 10 + cur_noc
                if not (cur_node.left or cur_node.right):
                    result += cur_val
        return result
```

相关题目

- sum-of-root-to-leaf-binary-numbers

> 这道题和本题太像了，跟一道题没啥区别

题目地址

https://leetcode.com/problems/surrounded-regions/description/

题目描述

```
Given a 2D board containing 'X' and 'O' (the letter O), cap

A region is captured by flipping all 'O's into 'X's in that

Example:

X X X X
X O O X
X X O X
X O X X
After running your function, the board should be:

X X X X
X X X X
X X X X
X O X X
Explanation:

Surrounded regions shouldn’t be on the border, which means
```

前置知识

- DFS

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

我们需要将所有被X包围的O变成X，并且题目明确说了边缘的所有O都是不可以变成X的。

其实我们观察会发现，我们除了边缘的O以及和边缘O连通的O是不需要变成X的，其他都要变成X。

经过上面的思考，问题转化为连通区域问题。 这里我们需要标记一下 `边缘的0以及和边缘0连通的0` 。 我们当然可以用额外的空间去存，但是对于这道题目而言，我们完全可以mutate。这样就空间复杂度会好一点。

整个过程如图所示：

> 我将 `边缘的0以及和边缘0连通的0` 标记为了 "A"

关键点解析

- 二维数组DFS解题模板
- 转化问题为 `连通区域问题`
- 直接mutate原数组，节省空间

代码

- 语言支持：JS，Python3

```
/*
 * @lc app=leetcode id=130 lang=javascript
 *
 * [130] Surrounded Regions
 */
// 将O以及周边的O转化为A
function mark(board, i, j, rows, cols) {
  if (i < 0 || i > rows - 1 || j < 0 || j > cols - 1 || boa
    return;

  board[i][j] = "A";
  mark(board, i + 1, j, rows, cols);
  mark(board, i - 1, j, rows, cols);
  mark(board, i, j + 1, rows, cols);
  mark(board, i, j - 1, rows, cols);
}
/**
 * @param {character[][]} board
 * @return {void} Do not return anything, modify board in-p
 */
var solve = function(board) {
  const rows = board.length;
  if (rows === 0) return [];
  const cols = board[0].length;

  for (let i = 0; i < rows; i++) {
    for (let j = 0; j < cols; j++) {
      if (i === 0 || i == rows - 1 || j === 0 || j === cols
        mark(board, i, j, rows, cols);
      }
    }
  }

  for (let i = 0; i < rows; i++) {
    for (let j = 0; j < cols; j++) {
      if (board[i][j] === "O") {
        board[i][j] = "X";
      } else if (board[i][j] === "A") {
        board[i][j] = "O";
      }
    }
  }

  return board;
};
```

Python Code：

```
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place inste
        """
        # 如果数组长或宽小于等于2，则不需要替换
        if len(board) <= 2 or len(board[0]) <= 2:
            return

        row, col = len(board), len(board[0])

        def dfs(i, j):
            """
            深度优先算法，如果符合条件，替换为A并进一步测试，否则停止
            """
            if i < 0 or j < 0 or i >= row or j >= col or bo
                return
            board[i][j] = 'A'

            dfs(i - 1, j)
            dfs(i + 1, j)
            dfs(i, j - 1)
            dfs(i, j + 1)

        # 从外围开始
        for i in range(row):
            dfs(i, 0)
            dfs(i, col-1)

        for j in range(col):
            dfs(0, j)
            dfs(row-1, j)

        # 最后完成替换
        for i in range(row):
            for j in range(col):
                if board[i][j] == 'O':
                    board[i][j] = 'X'
                elif board[i][j] == 'A':
                    board[i][j] = 'O'
```

相关题目

- 200.number-of-islands

> 解题模板是一样的

复杂度分析

- 时间复杂度：$O(row * col)$
- 空间复杂度：$O(row * col)$

更多题解可以访问我的LeetCode题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经35K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode.com/problems/palindrome-partitioning/description/

题目描述

```
Given a string s, partition s such that every substring of

Return all possible palindrome partitioning of s.

Example:

Input: "aab"
Output:
[
  ["aa","b"],
  ["a","a","b"]
]
```

前置知识

- 回溯法

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这是一道求解所有可能性的题目， 这时候可以考虑使用回溯法。 回溯法解题的模板我们已经在很多题目中用过了， 这里就不多说了。大家可以结合其他几道题目加深一下理解。

这里我画了一个图：

> 图是 78.subsets，都差不多，仅做参考。

关键点解析

- 回溯法

代码

- 语言支持：JS，Python3

```
/*
 * @lc app=leetcode id=131 lang=javascript
 *
 * [131] Palindrome Partitioning
 */

function isPalindrom(s) {
    let left = 0;
    let right = s.length - 1;

    while(left < right && s[left] === s[right]) {
        left++;
        right--;
    }

    return left >= right;
}
 function backtrack(s, list, tempList, start) {
    const sliced = s.slice(start);

    if (isPalindrom(sliced) && (tempList.join("").length ==

    for(let i = 0; i < sliced.length; i++) {
        const sub = sliced.slice(0, i + 1);
        if (isPalindrom(sub)) {
            tempList.push(sub);
        } else {
            continue;
        }
        backtrack(s, list, tempList, start + i + 1);
        tempList.pop();
    }
 }
/**
 * @param {string} s
 * @return {string[][]}
 */
var partition = function(s) {
    // "aab"
    // ["aa", "b"]
    // ["a", "a", "b"]
    const list = [];
    backtrack(s, list, [], 0);
    return list;
};
```

```python
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        """回溯法"""

        res = []

        def helper(s, tmp):
            """
            如果是空字符串，说明已经处理完毕
            否则逐个字符往前测试，判断是否是回文
            如果是，则处理剩余字符串，并将已经得到的列表作为参数
            """
            if not s:
                res.append(tmp)
            for i in range(1, len(s) + 1):
                if s[:i] == s[:i][::-1]:
                    helper(s[i:], tmp + [s[:i]])

        helper(s, [])
        return res
```

相关题目

- 39.combination-sum
- 40.combination-sum-ii
- 46.permutations
- 47.permutations-ii
- 78.subsets
- 90.subsets-ii
- 113.path-sum-ii

题目地址

https://leetcode.com/problems/word-break/description/

题目描述

```
Given a non-empty string s and a dictionary wordDict contai

Note:

The same word in the dictionary may be reused multiple time
You may assume the dictionary does not contain duplicate wo
Example 1:

Input: s = "leetcode", wordDict = ["leet", "code"]
Output: true
Explanation: Return true because "leetcode" can be segmente
Example 2:

Input: s = "applepenapple", wordDict = ["apple", "pen"]
Output: true
Explanation: Return true because "applepenapple" can be seg
             Note that you are allowed to reuse a dictionar
Example 3:

Input: s = "catsandog", wordDict = ["cats", "dog", "sand",
Output: false
```

前置知识

- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题是给定一个字典和一个句子，判断该句子是否可以由字典里面的单词组出来，一个单词可以用多次。

暴力的方法是无解的，复杂度极其高。 我们考虑其是否可以拆分为小问题来解决。 对于问题 `(s, wordDict)` 我们是否可以用(s', wordDict) 来解决。 其中 s' 是 s 的子序列， 当 s'变成寻常(长度为 0)的时候问题就解决了。 我们状态转移方程变成了这道题的难点。

我们可以建立一个数组 dp, dp[i]代表 字符串 s.substring(0, i) 能否由字典里面的单词组成， 值得注意的是，这里我们无法建立 dp[i] 和 dp[i - 1] 的关系， 我们可以建立的是 dp[i - word.length] 和 dp[i] 的关系。

我们用图来感受一下：

没有明白也没有关系，我们分步骤解读一下：

（以下的图左边都代表 s，右边都是 dict，灰色代表没有处理的字符，绿色代表匹配成功，红色代表匹配失败）

l
e
e
t
c
o
d
e
放不下
l c
e o
e d
t e

l
e
e
t
c
o
d
e
放不下
l c
e o
e d
t e

上面分步解释了算法的基本过程，下面我们感性认识下这道题，我把它比喻为 你正在 往一个老式手电筒🔦 中装电池

代码

```
/*
 * @lc app=leetcode id=139 lang=javascript
 *
 * [139] Word Break
 *
 * https://leetcode.com/problems/word-break/description/
 *
 * algorithms
 * Medium (34.45%)
 * Total Accepted:    317.8K
 * Total Submissions: 913.9K
 * Testcase Example:  '"leetcode"\n["leet","code"]'
 *
 * Given a non-empty string s and a dictionary wordDict con
 * non-empty words, determine if s can be segmented into a
 * sequence of one or more dictionary words.
 *
 * Note:
 *
 *
 * The same word in the dictionary may be reused multiple t
 * segmentation.
 * You may assume the dictionary does not contain duplicate
 *
 *
 * Example 1:
 *
 *
 * Input: s = "leetcode", wordDict = ["leet", "code"]
 * Output: true
 * Explanation: Return true because "leetcode" can be segme
 * code".
 *
 *
 * Example 2:
 *
 *
 * Input: s = "applepenapple", wordDict = ["apple", "pen"]
 * Output: true
 * Explanation: Return true because "applepenapple" can be
 * pen apple".
 * Note that you are allowed to reuse a dictionary word.
 *
 *
 * Example 3:
 *
 *
 * Input: s = "catsandog", wordDict = ["cats", "dog", "sand
```

```
 * Output: false
 *
 *
 */
/**
 * @param {string} s
 * @param {string[]} wordDict
 * @return {boolean}
 */
var wordBreak = function (s, wordDict) {
  const dp = Array(s.length + 1);
  dp[0] = true;
  for (let i = 0; i < s.length + 1; i++) {
    for (let word of wordDict) {
      if (word.length <= i && dp[i - word.length]) {
        if (s.substring(i - word.length, i) === word) {
          dp[i] = true;
        }
      }
    }
  }

  return dp[s.length] || false;
};
```

题目地址

https://leetcode.com/problems/binary-tree-preorder-traversal/description/

题目描述

```
Given a binary tree, return the preorder traversal of its r

Example:

Input: [1,null,2,3]
   1
    \
     2
    /
   3

Output: [1,2,3]
Follow up: Recursive solution is trivial, could you do it i
```

前置知识

- 递归
- 栈

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是前序遍历，这个和之前的 `leetcode 94 号问题 - 中序遍历` 完全不一回事。

前序遍历是 `根左右` 的顺序，注意是 `根` 开始，那么就很简单。直接先将根节点入栈，然后 看有没有右节点，有则入栈，再看有没有左节点，有则入栈。 然后出栈一个元素，重复即可。

> 其他树的非递归遍历可没这么简单

（迭代 VS 递归）

关键点解析

- 二叉树的基本操作（遍历）
  > 不同的遍历算法差异还是蛮大的
- 如果非递归的话利用栈来简化操作
- 如果数据规模不大的话，建议使用递归
- 递归的问题需要注意两点，一个是终止条件，一个如何缩小规模
- 终止条件，自然是当前这个元素是 null（链表也是一样）
- 由于二叉树本身就是一个递归结构， 每次处理一个子树其实就是缩小了规模， 难点在于如何合并结果，这里的合并结果其实就是 `mid.concat(left).concat(right)` , mid 是一个具体的节点，left 和 right 递归求出即可

代码

- 语言支持：JS，C++

JavaScript Code:

```
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var preorderTraversal = function(root) {
  // 1. Recursive solution

  // if (!root) return [];

  // return [root.val].concat(preorderTraversal(root.left)

  // 2. iterative solutuon

  if (!root) return [];
  const ret = [];
  const stack = [root];
  let t = stack.pop();

  while (t) {
    if (t.right) {
      stack.push(t.right);
    }
    if (t.left) {
      stack.push(t.left);
    }
    ret.push(t.val);
    t = stack.pop();
  }

  return ret;
};
```

C++ Code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {
 * };
 */
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> v;
        vector<TreeNode*> s;
        while (root != NULL || !s.empty()) {
            while (root != NULL) {
                v.push_back(root->val);
                s.push_back(root);
                root = root->left;
            }
            root = s.back()->right;
            s.pop_back();
        }
        return v;
    }
};
```

相关专题

- 二叉树的遍历

题目地址

https://leetcode.com/problems/evaluate-reverse-polish-notation/description/

题目描述

```
Evaluate the value of an arithmetic expression in Reverse P

Valid operators are +, -, *, /. Each operand may be an inte

Note:

Division between two integers should truncate toward zero.
The given RPN expression is always valid. That means the ex
```

前置知识

- 栈

公司

- 阿里
- 腾讯

思路

逆波兰表达式又叫做后缀表达式。在通常的表达式中，二元运算符总是置于与之相关的两个运算对象之间，这种表示法也称为 `中缀表示` 。

波兰逻辑学家J.Lukasiewicz于1929年提出了另一种表示表达式的方法，按此方法，每一运算符都置于其运算对象之后，故称为 `后缀表示` 。

> 逆波兰表达式是一种十分有用的表达式，它将复杂表达式转换为可以依靠简单的操作得到计算结果的表达式。例如(a+b)*(c+d)转换为ab+cd+*

关键点

1. 栈的基本用法

2. 如果你用的是JS的话，需要注意/ 和 其他很多语言是不一样的

3. 如果你用的是JS的话，需要先将字符串转化为数字。否则有很多意想不到的结果
4. 操作符的顺序应该是 先出栈的是第二位，后出栈的是第一位。 这在不符合交换律的操作中很重要， 比如减法和除法。

代码

```
/*
 * @lc app=leetcode id=150 lang=javascript
 *
 * [150] Evaluate Reverse Polish Notation
 *
 * https://leetcode.com/problems/evaluate-reverse-polish-no
 *
 * algorithms
 * Medium (31.43%)
 * Total Accepted:    153.3K
 * Total Submissions: 485.8K
 * Testcase Example:  '["2","1","+","3","*"]'
 *
 * Evaluate the value of an arithmetic expression in Revers
 *
 * Valid operators are +, -, *, /. Each operand may be an i
 * expression.
 *
 * Note:
 *
 *
 * Division between two integers should truncate toward zer
 * The given RPN expression is always valid. That means the
 * always evaluate to a result and there won't be any divio
 * operation.
 *
 *
 * Example 1:
 *
 *
 * Input: ["2", "1", "+", "3", "*"]
 * Output: 9
 * Explanation: ((2 + 1) * 3) = 9
 *
 *
 * Example 2:
 *
 *
 * Input: ["4", "13", "5", "/", "+"]
 * Output: 6
 * Explanation: (4 + (13 / 5)) = 6
 *
 *
 * Example 3:
 *
 *
 * Input: ["10", "6", "9", "3", "+", "-11", "*", "/", "*",
 * Output: 22
```

```
 * Explanation:
 *  ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
 * = ((10 * (6 / (12 * -11))) + 17) + 5
 * = ((10 * (6 / -132)) + 17) + 5
 * = ((10 * 0) + 17) + 5
 * = (0 + 17) + 5
 * = 17 + 5
 * = 22
 *
 *
 */
/**
 * @param {string[]} tokens
 * @return {number}
 */
var evalRPN = function(tokens) {
  // 这种算法的前提是 tokens是有效的,
  // 当然这由算法来保证
  const stack = [];

  for (let index = 0; index < tokens.length; index++) {
    const token = tokens[index];
    // 对于运算数,  我们直接入栈
    if (!Number.isNaN(Number(token))) {
      stack.push(token);
    } else {
      // 遇到操作符, 我们直接大胆运算, 不用考虑算术优先级
      // 然后将运算结果入栈即可

      // 当然如果题目进一步扩展, 允许使用单目等其他运算符, 我们的算法
      const a = Number(stack.pop());
      const b = Number(stack.pop());
      if (token === "*") {
        stack.push(b * a);
      } else if (token === "/") {
        stack.push(b / a >> 0);
      } else if (token === "+") {
        stack.push(b + a);
      } else if (token === "-") {
        stack.push(b - a);
      }
    }
  }

  return stack.pop();
};
```

扩展

逆波兰表达式中只改变运算符的顺序，并不会改变操作数的相对顺序，这是一个重要的性质。 另外逆波兰表达式完全不关心操作符的优先级，这在中缀表达式中是做不到的，这很有趣，感兴趣的可以私下查找资料研究下为什么会这样。

题目地址

https://leetcode.com/problems/maximum-product-subarray/description/

题目描述

```
给你一个整数数组 nums ，请你找出数组中乘积最大的连续子数组（该子数组中
示例 1:

输入: [2,3,-2,4]
输出: 6
解释: 子数组 [2,3] 有最大乘积 6。
示例 2:

输入: [-2,0,-1]
输出: 0
解释: 结果不能为 2, 因为 [-2,-1] 不是子数组。
```

前置知识

- 滑动窗口

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目要我们求解连续的 n 个数中乘积最大的积是多少。这里提到了连续，笔者首先想到的就是滑动窗口，但是这里比较特殊，我们不能仅仅维护一个最大值，因此最小值（比如-20）乘以一个比较小的数（比如-10）可能就会很大。 因此这种思路并不方便。

首先来暴力求解,我们使用两层循环来枚举所有可能项，这种解法的时间复杂度是O(n^2), 代码如下：

```
var maxProduct = function(nums) {
  let max = nums[0];
  let temp = null;
  for (let i = 0; i < nums.length; i++) {
    temp = nums[i];
    for (let j = i + 1; j < nums.length; j++) {
      temp *= nums[j];
      max = Math.max(temp, max);
    }
  }

  return max;
};
```

前面说了 `最小值（比如-20）乘以一个比较小的数（比如-10）可能就会很大` 。因此我们需要同时记录乘积最大值和乘积最小值，然后比较元素和这两个的乘积，去不断更新最大值。当然，我们也可以选择只取当前元素。因此实际上我们的选择有三种，而如何选择就取决于哪个选择带来的价值最大（乘积最大或者最小）。

这种思路的解法由于只需要遍历一次，其时间复杂度是O(n)，代码见下方代码区。

关键点

- 同时记录乘积最大值和乘积最小值

代码

代码支持：Python3，JavaScript

Python3 Code:

```
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        n = len(nums)
        max__dp = [1] * (n + 1)
        min_dp = [1] * (n + 1)
        ans = float('-inf')

        for i in range(1, n + 1):
            max__dp[i] = max(max__dp[i - 1] * nums[i - 1],
                             min_dp[i - 1] * nums[i - 1], n
            min_dp[i] = min(max__dp[i - 1] * nums[i - 1],
                            min_dp[i - 1] * nums[i - 1], nu
            ans = max(ans, max__dp[i])
        return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

当我们知道动态转移方程的时候，其实应该发现了。我们的dp[i] 只和 dp[i - 1]有关，这是一个空间优化的信号，告诉我们 `可以借助两个额外变量记录即可` 。

Python3 Code:

```
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        n = len(nums)
        a = b = 1
        ans = float('-inf')

        for i in range(1, n + 1):
            temp = a
            a = max(a * nums[i - 1],
                    b * nums[i - 1], nums[i - 1])
            b = min(temp * nums[i - 1],
                    b * nums[i - 1], nums[i - 1])
            ans = max(ans, a)
        return ans
```

JavaScript Code:

```
var maxProduct = function(nums) {
  let max = nums[0];
  let min = nums[0];
  let res = nums[0];

  for (let i = 1; i < nums.length; i++) {
    let tmp = min;
    min = Math.min(nums[i], Math.min(max * nums[i], min * r
    max = Math.max(nums[i], Math.max(max * nums[i], tmp * r
    res = Math.max(res, max);
  }
  return res;
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经30K star啦。

大家也可以关注我的公众号《脑洞前端》获取更多更新鲜的LeetCode题解

题目地址

https://leetcode.com/problems/binary-tree-right-side-view/description/

题目描述

```
Given a binary tree, imagine yourself standing on the righ

Example:

Input: [1,2,3,null,5,null,4]
Output: [1, 3, 4]
Explanation:

   1            <---
 /   \
2     3         <---
 \     \
  5     4       <---
```

前置知识

- 队列

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

> 这道题和 leetcode 102 号问题《102.binary-tree-level-order-traversal》很像

这道题可以借助 `队列` 实现，首先把 root 入队，然后入队一个特殊元素 Null(来表示每层的结束)。

然后就是 while(queue.length), 每次处理一个节点，都将其子节点（在这里是 left 和 right）放到队列中。

然后不断的出队， 如果出队的是 null，则表式这一层已经结束了，我们就继续 push 一个 null。

关键点解析

- 队列
- 队列中用 Null(一个特殊元素)来划分每层
- 树的基本操作- 遍历 - 层次遍历（BFS）
- 二叉树的右视图可以看作是层次遍历每次只取每一层的最右边的元素

代码

- 语言支持：JS，C++

Javascript Code:

```
/*
 * @lc app=leetcode id=199 lang=javascript
 *
 * [199] Binary Tree Right Side View
 *
 * https://leetcode.com/problems/binary-tree-right-side-vie
 *
 * algorithms
 * Medium (46.74%)
 * Total Accepted:    156.1K
 * Total Submissions: 332.3K
 * Testcase Example:  '[1,2,3,null,5,null,4]'
 *
 * Given a binary tree, imagine yourself standing on the ri
 * return the values of the nodes you can see ordered from
 *
 * Example:
 *
 *
 * Input: [1,2,3,null,5,null,4]
 * Output: [1, 3, 4]
 * Explanation:
 *
 *   1            <---
 * /   \
 * 2     3         <---
 * \     \
 *  5     4       <---
 *
 */
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var rightSideView = function(root) {
  if (!root) return [];

  const ret = [];
  const queue = [root, null];

  let levelNodes = [];
```

```
  while (queue.length > 0) {
    const node = queue.shift();
    if (node !== null) {
      levelNodes.push(node.val);
      if (node.right) {
        queue.push(node.right);
      }
      if (node.left) {
        queue.push(node.left);
      }
    } else {
      // 一层遍历已经结束
      ret.push(levelNodes[0]);
      if (queue.length > 0) {
        queue.push(null);
      }
      levelNodes = [];
    }
  }

  return ret;
};
```

C++ Code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {
 * };
 */
class Solution {
public:
    vector<int> rightSideView(TreeNode* root) {
        auto ret = vector<int>();
        if (root == nullptr) return ret;
        auto q = queue<const TreeNode*>();
        q.push(root);
        while (!q.empty()) {
            auto sz = q.size();
            for (auto i = 0; i < sz; ++i) {
                auto n = q.front();
                q.pop();
                if (n->left != nullptr ) q.push(n->left);
                if (n->right != nullptr ) q.push(n->right);
                if (i == sz - 1) ret.push_back(n->val);
            }
        }
        return ret;
    }
};
```

扩展

假如题目变成求二叉树的左视图呢?

很简单我们只需要取 queue 的最后一个元素即可。 或者存的时候反着来也行

> 其实我们没必要存储 levelNodes，而是只存储每一层最右的元素，这样空间复杂度就不是 n 了， 就是 logn 了。

题目地址

https://leetcode.com/problems/number-of-islands/description/

题目描述

```
Given a 2d grid map of '1's (land) and '0's (water), count

Example 1:

Input:
11110
11010
11000
00000

Output: 1
Example 2:

Input:
11000
11000
00100
00011

Output: 3
```

前置知识

- DFS

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

如图，我们其实就是要求红色区域的个数，换句话说就是求连续区域的个数。

符合直觉的做法是用DFS来解：

- 我们需要建立一个 visited 数组用来记录某个位置是否被访问过。
- 对于一个为 `1` 且未被访问过的位置，我们递归进入其上下左右位置上为 `1` 的数，将其 visited 变成 true。
- 重复上述过程
- 找完相邻区域后，我们将结果 res 自增1，然后我们在继续找下一个为 `1` 且未被访问过的位置，直至遍历完.

但是这道题目只是让我们求连通区域的个数，因此我们其实不需要额外的空间去存储visited信息。 注意到上面的过程，我们对于数字为0的其实不会进行操作的，也就是对我们“没用”。 因此对于已经访问的元素， 我们可以将其置为0即可。

关键点解析

- 二维数组DFS解题模板
- 将已经访问的元素置为0，省去visited的空间开销

代码

- 语言支持：C++, Java, JS, python3

C++ Code：

```
class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        int res = 0;
        for(int i=0;i<grid.size();i++)
        {
            for(int j=0;j<grid[0].size();j++)
            {
                if(grid[i][j] == '1')
                {
                    dfs(grid, i, j);
                    res += 1;
                }
            }
        }
        return res;

    }
    void dfs(vector<vector<char>>& grid, int i, int j)
    {
        // edge
        if(i<0 || i>= grid.size() || j<0 || j>= grid[0].siz
        {
            return;
        }
        grid[i][j] = '0';
        dfs(grid, i+1, j);
        dfs(grid, i-1, j);
        dfs(grid, i, j+1);
        dfs(grid, i, j-1);
    }
};
```

Java Code：

```
    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0 || grid[0].le

        int count = 0;
        for (int row = 0; row < grid.length; row++) {
            for (int col = 0; col < grid[0].length; col++)
                if (grid[row][col] == '1') {
                    dfs(grid, row, col);
                    count++;
                }
            }
        }
        return count;
    }

    private void dfs(char[][] grid,int row,int col) {
        if (row<0||row== grid.length||col<0||col==grid[0].
            return;
        }
        grid[row][col] = '0';
        dfs(grid, row-1, col);
        dfs(grid, row+1, col);
        dfs(grid, row, col+1);
        dfs(grid, row, col-1);
    }
```

Javascript Code:

```javascript
/*
 * @lc app=leetcode id=200 lang=javascript
 *
 * [200] Number of Islands
 */
function helper(grid, i, j, rows, cols) {
  if (i < 0 || j < 0 || i > rows - 1 || j > cols - 1 || gri
    return;

  grid[i][j] = "0";

  helper(grid, i + 1, j, rows, cols);
  helper(grid, i, j + 1, rows, cols);
  helper(grid, i - 1, j, rows, cols);
  helper(grid, i, j - 1, rows, cols);
}
/**
 * @param {character[][]} grid
 * @return {number}
 */
var numIslands = function(grid) {
  let res = 0;
  const rows = grid.length;
  if (rows === 0) return 0;
  const cols = grid[0].length;
  for (let i = 0; i < rows; i++) {
    for (let j = 0; j < cols; j++) {
      if (grid[i][j] === "1") {
        helper(grid, i, j, rows, cols);
        res++;
      }
    }
  }
  return res;
};
```

python code:

```
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        if not grid: return 0

        count = 0
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j] == '1':
                    self.dfs(grid, i, j)
                    count += 1

        return count

    def dfs(self, grid, i, j):
        if i < 0 or j < 0 or i >= len(grid) or j >= len(gri
            return
        grid[i][j] = '0'
        self.dfs(grid, i + 1, j)
        self.dfs(grid, i - 1, j)
        self.dfs(grid, i, j + 1)
        self.dfs(grid, i, j - 1)
```

复杂度分析

- 时间复杂度：$O(m * n)$
- 空间复杂度：$O(m * n)$

欢迎关注我的公众号《脑洞前端》获取更多更新鲜的LeetCode题解

相关题目

- 695. 岛屿的最大面积

题目地址

https://leetcode.com/problems/bitwise-and-of-numbers-range/description/

题目描述

```
Given a range [m, n] where 0 <= m <= n <= 2147483647, retu

Example 1:

Input: [5,7]
Output: 4
Example 2:

Input: [0,1]
Output: 0
```

前置知识

- 位运算

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

一个显而易见的解法是， 从m到n依次进行 `求与` 的操作。

```
    let res = m;
    for (let i = m + 1; i <= n; i++) {
      res = res & i;
    }
    return res;
```

但是， 如果你把这个solution提交的话，很显然不会通过， 会超时。

我们依旧还是用trick来简化操作。 我们利用的性质是， n个连续数字求与的时候，前m位都是1.

举题目给的例子：[5,7] 共 5， 6，7三个数字， 用二进制表示 101, 110,111, 这三个数字特点是第一位都是1，后面几位求与一定是0.

再来一个明显的例子：[20, 24], 共 20， 21， 22， 23，24五个数字，二进制表示就是

```
0001 0100
0001 0101
0001 0110
0001 0111
0001 1000
```

这五个数字特点是第四位都是1，后面几位求与一定是0.

因此我们的思路就是， 求出这个数字区间的数字前多少位都是1了，那么他们求与的结果一定是前几位数字，然后后面都是0.

关键点解析

- n个连续数字求与的时候，前m位都是1
- 可以用递归实现， 个人认为比较难想到
- bit 运算

代码：

```
(n > m) ? (rangeBitwiseAnd(m/2, n/2) << 1) : m;
```

> 每次问题规模缩小一半， 这是二分法吗?

代码

语言支持：JavaSCript，Python3

JavaScript Code:

```javascript
/*
 * @lc app=leetcode id=201 lang=javascript
 *
 * [201] Bitwise AND of Numbers Range
 *
 */
/**
 * @param {number} m
 * @param {number} n
 * @return {number}
 */
var rangeBitwiseAnd = function(m, n) {
  let count = 0;
  while (m !== n) {
    m = m >> 1;
    n = n >> 1;
    count++;
  }

  return n << count;
};
```

Python Code:

```python
class Solution:
    def rangeBitwiseAnd(self, m: int, n: int) -> int:
        cnt = 0
        while m != n:
            m >>= 1
            n >>= 1
            cnt += 1

        return m << cnt
```

复杂度分析

- 时间复杂度：最坏的情况我们需要循环N次，最好的情况是一次都不需要， 因此时间复杂度取决于我们移动的位数，具体移动的次数取决于我们的输入，平均来说时间复杂度为 $O(N)$，其中N为M和N的二进制表示的位数。
- 空间复杂度：$O(1)$

题目地址

https://leetcode.com/problems/implement-trie-prefix-tree/description/

题目描述

```
Implement a trie with insert, search, and startsWith methoc

Example:

Trie trie = new Trie();

trie.insert("apple");
trie.search("apple");   // returns true
trie.search("app");     // returns false
trie.startsWith("app"); // returns true
trie.insert("app");
trie.search("app");     // returns true
Note:

You may assume that all inputs are consist of lowercase let
All inputs are guaranteed to be non-empty strings.
```

前置知识

- 前缀树

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这是一道很直接的题目，上来就让你实现 `前缀树（字典树）` 。这算是基础数据结构中的 知识了，不清楚什么是字典树的可以查阅相关资料。

我们看到题目给出的使用方法 `new Trie` ，`insert` ，`search` 和 `startWith` .

为了区分 `search` 和 `startWith` 我们需要增加一个标示来区分当前节点是否是某个单词的结尾。 因此节点的数据结构应该是:

```
function TrieNode(val) {
  this.val = val; // 当前的字母
  this.children = []; // 题目要求字典仅有a-z，那么其长度最大为26
  this.isWord = false;
}
```

每次 insert 我们其实都是从根节点出发，一个一个找到我们需要添加的节点，修改 children 的值.

我们应该修改哪一个 child 呢？ 我们需要一个函数来计算索引

```
function computeIndex(c) {
  return c.charCodeAt(0) - "a".charCodeAt(0);
}
```

其实不管 insert， search 和 startWith 的逻辑都是差不多的，都是从 root 出发， 找到我们需要操作的 child， 然后进行相应操作（添加，修改，返回）。

关键点解析

- 前缀树

代码

```
/*
 * @lc app=leetcode id=208 lang=javascript
 *
 * [208] Implement Trie (Prefix Tree)
 *
 * https://leetcode.com/problems/implement-trie-prefix-tree
 *
 * algorithms
 * Medium (36.93%)
 * Total Accepted:    172K
 * Total Submissions: 455.5K
 * Testcase Example:  '["Trie","insert","search","search","
 *
 * Implement a trie with insert, search, and startsWith met
 *
 * Example:
 *
 *
 * Trie trie = new Trie();
 *
 * trie.insert("apple");
 * trie.search("apple");   // returns true
 * trie.search("app");     // returns false
 * trie.startsWith("app"); // returns true
 * trie.insert("app");
 * trie.search("app");     // returns true
 *
 *
 * Note:
 *
 *
 * You may assume that all inputs are consist of lowercase
 * All inputs are guaranteed to be non-empty strings.
 *
 *
 */
function TrieNode(val) {
  this.val = val;
  this.children = [];
  this.isWord = false;
}

function computeIndex(c) {
  return c.charCodeAt(0) - "a".charCodeAt(0);
}
/**
 * Initialize your data structure here.
 */
```

```
var Trie = function () {
  this.root = new TrieNode(null);
};

/**
 * Inserts a word into the trie.
 * @param {string} word
 * @return {void}
 */
Trie.prototype.insert = function (word) {
  let ws = this.root;
  for (let i = 0; i < word.length; i++) {
    const c = word[i];
    const current = computeIndex(c);
    if (!ws.children[current]) {
      ws.children[current] = new TrieNode(c);
    }
    ws = ws.children[current];
  }
  ws.isWord = true;
};

/**
 * Returns if the word is in the trie.
 * @param {string} word
 * @return {boolean}
 */
Trie.prototype.search = function (word) {
  let ws = this.root;
  for (let i = 0; i < word.length; i++) {
    const c = word[i];
    const current = computeIndex(c);
    if (!ws.children[current]) return false;
    ws = ws.children[current];
  }
  return ws.isWord;
};

/**
 * Returns if there is any word in the trie that starts wi
 * @param {string} prefix
 * @return {boolean}
 */
Trie.prototype.startsWith = function (prefix) {
  let ws = this.root;
  for (let i = 0; i < prefix.length; i++) {
    const c = prefix[i];
    const current = computeIndex(c);
```

```
    if (!ws.children[current]) return false;
    ws = ws.children[current];
  }
  return true;
};

/**
 * Your Trie object will be instantiated and called as such
 * var obj = new Trie()
 * obj.insert(word)
 * var param_2 = obj.search(word)
 * var param_3 = obj.startsWith(prefix)
 */
```

相关题目

- 0211.add-and-search-word-data-structure-design
- 0212.word-search-ii
- 0472.concatenated-words
- 0820.short-encoding-of-words
- 1032.stream-of-characters

题目地址

https://leetcode.com/problems/minimum-size-subarray-sum/description/

题目描述

```
Given an array of n positive integers and a positive intege

Example:

Input: s = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: the subarray [4,3] has the minimal length unde
Follow up:
If you have figured out the O(n) solution, try coding anoth
```

前置知识

- 滑动窗口

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

用滑动窗口来记录序列， 每当滑动窗口中的 sum 超过 s， 就去更新最小值，并根据先进先出的原则更新滑动窗口，直至 sum 刚好小于 s

> 这道题目和 leetcode 3 号题目有点像，都可以用滑动窗口的思路来解决

关键点

- 滑动窗口简化操作(滑窗口适合用于求解这种要求 `连续` 的题目)

代码

- 语言支持：JS，C++，Python

Python Code：

```
class Solution:
    def minSubArrayLen(self, s: int, nums: List[int]) -> in
        l = total = 0
        ans = len(nums) + 1
        for r in range(len(nums)):
            total += nums[r]
            while total >= s:
                ans = min(ans, r - l + 1)
                total -= nums[l]
                l += 1
        return  0 if ans == len(nums) + 1 else ans
```

JavaScript Code：

```
/*
 * @lc app=leetcode id=209 lang=javascript
 *
 * [209] Minimum Size Subarray Sum
 *
 */
/**
 * @param {number} s
 * @param {number[]} nums
 * @return {number}
 */
var minSubArrayLen = function(s, nums) {
  if (nums.length === 0) return 0;
  const slideWindow = [];
  let acc = 0;
  let min = null;

  for (let i = 0; i < nums.length + 1; i++) {
    const num = nums[i];

    while (acc >= s) {
      if (min === null || slideWindow.length < min) {
        min = slideWindow.length;
      }
      acc = acc - slideWindow.shift();
    }

    slideWindow.push(num);

    acc = slideWindow.reduce((a, b) => a + b, 0);
  }

  return min || 0;
};
```

C++ Code：

```
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        int num_len= nums.size();
        int left=0, right=0, total=0, min_len= num_len+1;
        while (right < num_len) {
            do {
                total += nums[right++];
            } while (right < num_len && total < s);
            while (left < right && total - nums[left] >= s)
            if (total >=s && min_len > right - left)
                min_len = right- left;
        }
        return min_len <= num_len ? min_len: 0;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组大小。
- 空间复杂度：$O(1)$

欢迎关注我的公众号《脑洞前端》获取更多更新鲜的LeetCode题解

扩展

如果题目要求是 sum = s, 而不是 sum >= s 呢?

eg:

```
var minSubArrayLen = function(s, nums) {
  if (nums.length === 0) return 0;
  const slideWindow = [];
  let acc = 0;
  let min = null;

  for (let i = 0; i < nums.length + 1; i++) {
    const num = nums[i];

    while (acc > s) {
      acc = acc - slideWindow.shift();
    }
    if (acc === s) {
      if (min === null || slideWindow.length < min) {
        min = slideWindow.length;
      }
      slideWindow.shift();
    }

    slideWindow.push(num);

    acc = slideWindow.reduce((a, b) => a + b, 0);
  }

  return min || 0;
};
```

题目地址（211. 添加与搜索单词 - 数据结构设计）

https://leetcode-cn.com/problems/add-and-search-word-data-structure-design/description/

题目描述

```
设计一个支持以下两种操作的数据结构：

void addWord(word)
bool search(word)
search(word) 可以搜索文字或正则表达式字符串，字符串只包含字母 . 或

示例：

addWord("bad")
addWord("dad")
addWord("mad")
search("pad") -> false
search("bad") -> true
search(".ad") -> true
search("b..") -> true
说明：

你可以假设所有单词都是由小写字母 a-z 组成的。
```

前置知识

- 前缀树

公司

- 阿里
- 腾讯

思路

我们首先不考虑字符"."的情况。这种情况比较简单，我们 addWord 直接添加到数组尾部，search 则线性查找即可。

接下来我们考虑特殊字符“.”，其实也不难，只不过 search 的时候，判断如果是“.”, 我们认为匹配到了，继续往后匹配即可。

上面的代码复杂度会比较高，我们考虑优化。如果你熟悉前缀树的话，应该注意到这可以使用前缀树来进行优化。前缀树优化之后每次查找复杂度是$O(h)$, 其中 h 是前缀树深度，也就是最长的字符串长度。

关于前缀树，LeetCode 有很多题目。有的是直接考察，让你实现一个前缀树，有的是间接考察，比如本题。前缀树代码见下方，大家之后可以直接当成前缀树的解题模板使用。

由于我们这道题需要考虑特殊字符".", 因此我们需要对标准前缀树做一点改造，insert 不做改变，我们只需要改变 search 即可，代码(Python 3)：

```
def search(self, word):
        """
        Returns if the word is in the trie.
        :type word: str
        :rtype: bool
        """
        curr = self.Trie
        for i, w in enumerate(word):
            if w == '.':
                wizards = []
                for k in curr.keys():
                    if k == '#':
                        continue
                    wizards.append(self.search(word[:i] + k
                return any(wizards)
            if w not in curr:
                return False
            curr = curr[w]
        return "#" in curr
```

标准的前缀树搜索我也贴一下代码，大家可以对比一下：

```
def search(self, word):
        """
        Returns if the word is in the trie.
        :type word: str
        :rtype: bool
        """
        curr = self.Trie
        for w in word:
            if w not in curr:
                return False
            curr = curr[w]
        return "#" in curr
```

关键点

- 前缀树（也叫字典树），英文名 Trie（读作 tree 或者 try）

代码

- 语言支持：Python3

Python3 Code：

关于 Trie 的代码:

```
class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.Trie = {}

    def insert(self, word):
        """
        Inserts a word into the trie.
        :type word: str
        :rtype: void
        """
        curr = self.Trie
        for w in word:
            if w not in curr:
                curr[w] = {}
            curr = curr[w]
        curr['#'] = 1

    def search(self, word):
        """
        Returns if the word is in the trie.
        :type word: str
        :rtype: bool
        """
        curr = self.Trie
        for i, w in enumerate(word):
            if w == '.':
                wizards = []
                for k in curr.keys():
                    if k == '#':
                        continue
                    wizards.append(self.search(word[:i] + k
                return any(wizards)
            if w not in curr:
                return False
            curr = curr[w]
        return "#" in curr
```

主逻辑代码:

```
class WordDictionary:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.trie = Trie()

    def addWord(self, word: str) -> None:
        """
        Adds a word into the data structure.
        """
        self.trie.insert(word)

    def search(self, word: str) -> bool:
        """
        Returns if the word is in the data structure. A wo
        """
        return self.trie.search(word)


# Your WordDictionary object will be instantiated and calle
# obj = WordDictionary()
# obj.addWord(word)
# param_2 = obj.search(word)
```

相关题目

- 0208.implement-trie-prefix-tree
- 0212.word-search-ii
- 0472.concatenated-words
- 0820.short-encoding-of-words

题目地址

https://leetcode.com/problems/kth-largest-element-in-an-array/

题目描述

```
Find the kth largest element in an unsorted array. Note tha

Example 1:

Input: [3,2,1,5,6,4] and k = 2
Output: 5
Example 2:

Input: [3,2,3,1,2,4,5,5,6] and k = 4
Output: 4
Note:
You may assume k is always valid, 1 ≤ k ≤ array's length.
```

前置知识

- 堆
- Quick Select

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题要求在一个无序的数组中，返回第K大的数。根据时间复杂度不同，这题有3种不同的解法。

解法一 （排序）

很直观的解法就是给数组排序，这样求解第 `K` 大的数，就等于是从小到大排好序的数组的第 `（n-K）` 小的数 (n 是数组的长度）。

例如：

```
[3,2,1,5,6,4], k = 2
1. 数组排序:
 [1,2,3,4,5,6],
2. 找第（n-k）小的数
 n-k=4, nums[4]=5 （第2大的数）
```

时间复杂度: `O(nlogn) - n 是数组长度。`

解法二 - 小顶堆（Heap）

可以维护一个大小为 `K` 的小顶堆，堆顶是最小元素，当堆的 `size > K` 的时候，删除堆顶元素. 扫描一遍数组，最后堆顶就是第 `K` 大的元素。直接返回。

例如:

时间复杂度: `O(n * logk) , n is array length` *空间复杂度:* `O(k)`

跟排序相比，以空间换时间。

解法三 - Quick Select

Quick Select 类似快排，选取pivot，把小于pivot的元素都移到pivot之前，这样pivot所在位置就是第pivot index 小的元素。 但是不需要完全给数组排序，只要找到当前pivot的位置是否是在第(n-k)小的位置，如果是，找到第k大的数直接返回。

具体步骤:

```
1. 在数组区间随机取`pivot index = left + random[right-left]`.
2. 根据pivot 做 partition, 在数组区间, 把小于pivot的数都移到pivot
3. 得到pivot的位置 index, `compare(index, (n-k))`.
    a. index == n-k -> 找到第`k`大元素, 直接返回结果。
    b. index < n-k -> 说明在`index`右边, 继续找数组区间`[index+
    c. index > n-k -> 那么第`k`大数在`index`左边, 继续查找数组区

例子, 【3,2,3,1,2,4,5,5,6], k = 4

如下图:
```

时间复杂度:

- 平均是: `O(n)`
- 最坏的情况是: `O(n * n)`

关键点分析

1. 直接排序很简单
2. 堆（Heap）主要是要维护一个K大小的小顶堆，扫描一遍数组，最后堆顶元素即是所求。
3. Quick Select, 关键是是取pivot，对数组区间做partition，比较pivot的位置，类似二分，取pivot左边或右边继续递归查找。

代码（Java code）

解法一 - 排序

```
class KthLargestElementSort {
 public int findKthlargest2(int[] nums, int k) {
    Arrays.sort(nums);
    return nums[nums.length - k];
  }
}
```

解法二 - Heap （PriorityQueue）

```
class KthLargestElementHeap {
  public int findKthLargest(int[] nums, int k) {
      PriorityQueue<Integer> pq = new PriorityQueue<>();
      for (int num : nums) {
        pq.offer(num);
        if (pq.size() > k) {
          pq.poll();
        }
      }
      return pq.poll();
  }
}
```

解法三 - Quick Select

```
class KthLargestElementQuickSelect {
    static Random random = new Random();
    public int findKthLargest3(int[] nums, int k) {
      int len = nums.length;
      return select(nums, 0, len - 1, len - k);
    }

    private int select(int[] nums, int left, int right, int
      if (left == right) return nums[left];
      // random select pivotIndex between left and right
      int pivotIndex = left + random.nextInt(right - left);
      // do partition, move smaller than pivot number into
      int pos = partition(nums, left, right, pivotIndex);
      if (pos == k) {
        return nums[pos];
      } else if (pos > k) {
        return select(nums, left, pos - 1, k);
      } else {
        return select(nums, pos + 1, right, k);
      }
    }

    private int partition(int[] nums, int left, int right,
      int pivot = nums[pivotIndex];
      // move pivot to end
      swap(nums, right, pivotIndex);
      int pos = left;
      // move smaller num to pivot left
      for (int i = left; i <= right; i++) {
        if (nums[i] < pivot) {
          swap(nums, pos++, i);
        }
      }
      // move pivot to original place
      swap(nums, right, pos);
      return pos;
    }

    private void swap(int[] nums, int i, int j) {
      int tmp = nums[i];
      nums[i] = nums[j];
      nums[j] = tmp;
    }
}
```

参考（References）

1. Quick Select Wiki

题目地址

https://leetcode.com/problems/maximal-square/

题目描述

```
Given a 2D binary matrix filled with 0's and 1's, find the

Example:

Input:

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

Output: 4
```

前置知识

- 动态规划
- 递归

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

符合直觉的做法是暴力求解处所有的正方形，逐一计算面积，然后记录最大的。这种时间复杂度很高。

我们考虑使用动态规划，我们使用dp[i][j]表示以matrix[i][j]为右下角的顶点的可以组成的最大正方形的边长。 那么我们只需要计算所有的i，j组合，然后求出最大值即可。

我们来看下dp[i][j] 怎么推导。 首先我们要看matrix[i][j], 如果matrix[i][j]等于0，那么就不用看了，直接等于0。 如果matrix[i][j]等于1，那么我们将matrix[[i][j]分别往上和往左进行延伸，直到碰到一个0为止。

如图 dp[3][3] 的计算。 matrix[3][3]等于1，我们分别往上和往左进行延伸，直到碰到一个0为止，上面长度为1，左边为3。 dp[2][2]等于1（之前已经计算好了），那么其实这里的瓶颈在于三者的最小值, 即 `Min(1, 1, 3)` , 也就是 `1` 。 那么dp[3][3] 就等于 `Min(1, 1, 3) + 1` 。

dp[i - 1][j - 1]我们直接拿到，关键是 `往上和往左进行延伸` ，最直观的做法是我们内层加一个循环去做就好了。 但是我们仔细观察一下，其实我们根本不需要这样算。 我们可以直接用dp[i - 1][j]和dp[i][j -1]。 具体就是 `Min(dp[i - 1][j - 1], dp[i][j - 1], dp[i - 1][j]) + 1` 。

事实上，这道题还有空间复杂度O(N)的解法，其中N指的是列数。 大家可以去这个leetcode讨论看一下。

关键点解析

- DP
- 递归公式可以利用dp[i - 1][j]和dp[i][j -1]的计算结果，而不用重新计算
- 空间复杂度可以降低到O(n), n为列数

代码

代码支持：Python，JavaScript：

Python Code：

```
class Solution:
    def maximalSquare(self, matrix: List[List[str]]) -> int
        res = 0
        m = len(matrix)
        if m == 0:
            return 0
        n = len(matrix[0])
        dp = [[0] * (n + 1) for _ in range(m + 1)]

        for i in range(1, m + 1):
            for j in range(1, n + 1):
                dp[i][j] = min(dp[i - 1][j], dp[i][j - 1],
                res = max(res, dp[i][j])
        return res ** 2
```

JavaScript Code：

```
/*
 * @lc app=leetcode id=221 lang=javascript
 *
 * [221] Maximal Square
 */
/**
 * @param {character[][]} matrix
 * @return {number}
 */
var maximalSquare = function(matrix) {
  if (matrix.length === 0) return 0;
  const dp = [];
  const rows = matrix.length;
  const cols = matrix[0].length;
  let max = Number.MIN_VALUE;

  for (let i = 0; i < rows + 1; i++) {
    if (i === 0) {
      dp[i] = Array(cols + 1).fill(0);
    } else {
      dp[i] = [0];
    }
  }

  for (let i = 1; i < rows + 1; i++) {
    for (let j = 1; j < cols + 1; j++) {
      if (matrix[i - 1][j - 1] === "1") {
        dp[i][j] = Math.min(dp[i - 1][j - 1], dp[i - 1][j],
        max = Math.max(max, dp[i][j]);
      } else {
        dp[i][j] = 0;
      }
    }
  }

  return max * max;
};
```

复杂度分析

- 时间复杂度：$O(M*N)$，其中M为行数，N为列数。
- 空间复杂度：$O(M*N)$，其中M为行数，N为列数。

题目地址

https://leetcode.com/problems/majority-element-ii/description/

题目描述

```
Given an integer array of size n, find all elements that ap

Note: The algorithm should run in linear time and in O(1) s

Example 1:

Input: [3,2,3]
Output: [3]
Example 2:

Input: [1,1,1,3,3,2,2,2]
Output: [1,2]
```

前置知识

- 摩尔投票法

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目和169.majority-element 很像。

我们仍然可以采取同样的方法 - “摩尔投票法”， 具体的思路可以参考上面的题目。

但是这里有一个不同的是这里的众数不再是超过 `1 / 2` ,而是超过 `1 / 3` 。 题目也说明了，超过三分之一的有可能有多个（实际上就是0，1，2三种可能）。

因此我们不能只用一个counter来解决了。 我们的思路是同时使用两个counter，其他思路和上一道题目一样。

最后需要注意的是两个counter不一定都满足条件，这两个counter只是出现次数最多的两个数字。 有可能不满足出现次数大于1/3， 因此最后我们需要进行过滤筛选。

这里画了一个图，大家可以感受一下：

关键点解析

- 摩尔投票法
- 两个counter
- 最后得到的只是出现次数最多的两个数字，有可能不满足出现次数大于1/3

代码

JavaScript代码：

```javascript
/*
 * @lc app=leetcode id=229 lang=javascript
 *
 * [229] Majority Element II
 */
/**
 * @param {number[]} nums
 * @return {number[]}
 */
var majorityElement = function(nums) {
  const res = [];
  const len = nums.length;
  let n1 = null,
    n2 = null,
    cnt1 = 0,
    cnt2 = 0;

  for (let i = 0; i < len; i++) {
    if (n1 === nums[i]) {
      cnt1++;
    } else if (n2 === nums[i]) {
      cnt2++;
    } else if (cnt1 === 0) {
      n1 = nums[i];
      cnt1++;
    } else if (cnt2 === 0) {
      n2 = nums[i];
      cnt2++;
    } else {
      cnt1--;
      cnt2--;
    }
  }

  cnt1 = 0;
  cnt2 = 0;

  for (let i = 0; i < len; i++) {
    if (n1 === nums[i]) {
      cnt1++;
    } else if (n2 === nums[i]) {
      cnt2++;
    }
  }

  if (cnt1 > (len / 3) >>> 0) {
    res.push(n1);
  }
```

```
    if (cnt2 > (len / 3) >>> 0) {
      res.push(n2);
    }

    return res;
  };
```

Java代码：

```java
/*
 * @lc app=leetcode id=229 lang=java
 *
 * [229] Majority Element II
 */
class Solution {
    public List<Integer> majorityElement(int[] nums) {
        List<Integer> res = new ArrayList<Integer>();
        if (nums == null || nums.length == 0)
            return res;
        int n1 = nums[0], n2 = nums[0], cnt1 = 0, cnt2 = 0,
        for (int i = 0; i < len; i++) {
            if (nums[i] == n1)
                cnt1++;
            else if (nums[i] == n2)
                cnt2++;
            else if (cnt1 == 0) {
                n1 = nums[i];
                cnt1 = 1;
            } else if (cnt2 == 0) {
                n2 = nums[i];
                cnt2 = 1;
            } else {
                cnt1--;
                cnt2--;
            }
        }
        cnt1 = 0;
        cnt2 = 0;
        for (int i = 0; i < len; i++) {
            if (nums[i] == n1)
                cnt1++;
            else if (nums[i] == n2)
                cnt2++;
        }
        if (cnt1 > len / 3)
            res.add(n1);
        if (cnt2 > len / 3 && n1 != n2)
            res.add(n2);
        return res;
    }
}
```

扩展

如果题目中3变成了k，怎么解决?

大家可以自己思考一下，我这里给一个参考链接：
https://leetcode.com/problems/majority-element-ii/discuss/63500/JAVA-Easy-Version-To-Understand!!!!!!!!!!!!/64925

这个实现说实话不是很好，大家可以优化一下。

题目地址

https://leetcode.com/problems/kth-smallest-element-in-a-bst/description/

题目描述

```
Given a binary search tree, write a function kthSmallest to

Note:
You may assume k is always valid, 1 ≤ k ≤ BST's total eleme

Example 1:

Input: root = [3,1,4,null,2], k = 1
   3
  / \
 1   4
  \
   2
Output: 1
Example 2:

Input: root = [5,3,6,2,4,null,null,1], k = 3
       5
      / \
     3   6
    / \
   2   4
  /
 1
Output: 3
Follow up:
What if the BST is modified (insert/delete operations) ofte
```

前置知识

- 中序遍历

公司

- 阿里
- 腾讯
- 百度

- 字节

思路

解法一：

由于‘中序遍历一个二叉查找树（BST）的结果是一个有序数组’，因此我们只需要在遍历到第k个，返回当前元素即可。 中序遍历相关思路请查看 binary-tree-traversal

解法二：

联想到二叉搜索树的性质，root 大于左子树，小于右子树，如果左子树的节点数目等于 K-1，那么 root 就是结果，否则如果左子树节点数目小于 K-1，那么结果必然在右子树，否则就在左子树。 因此在搜索的时候同时返回节点数目，跟 K 做对比，就能得出结果了。

关键点解析

- 中序遍历

代码

解法一：

JavaScript Code:

```javascript
/*
 * @lc app=leetcode id=230 lang=javascript
 *
 * [230] Kth Smallest Element in a BST
 */
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @param {number} k
 * @return {number}
 */
var kthSmallest = function(root, k) {
    const stack = [root];
    let cur = root;
    let i = 0;

    function insertAllLefts(cur) {
        while(cur && cur.left) {
            const l = cur.left;
            stack.push(l);
            cur = l;
        }
    }
    insertAllLefts(cur);

    while(cur = stack.pop()) {
        i++;
        if (i === k) return cur.val;
        const r = cur.right;

        if (r) {
            stack.push(r);
            insertAllLefts(r);
        }
    }

    return -1;


};
```

Java Code:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
private int count = 1;
private int res;

public int KthSmallest (TreeNode root, int k) {
    inorder(root, k);
    return res;
}

public void inorder (TreeNode root, int k) {
    if (root == null) return;

    inorder(root.left, k);

    if (count++ == k) {
        res = root.val;
        return;
    }

    inorder(root.right, k);
}
```

解法二:

JavaScript Code:

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
function nodeCount(node) {
    if (node === null) return 0;

    const l = nodeCount(node.left);
    const r = nodeCount(node.right);

    return 1 + l + r;
}
/**
 * @param {TreeNode} root
 * @param {number} k
 * @return {number}
 */
var kthSmallest = function(root, k) {
    const c = nodeCount(root.left);
    if (c === k - 1) return root.val;
    else if (c < k - 1) return kthSmallest(root.right, k -
    return kthSmallest(root.left, k)
};
```

扩展

这道题有一个follow up：

`What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?`

大家可以思考一下。

题目地址

https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/description/

题目描述

```
Given a binary tree, find the lowest common ancestor (LCA)

According to the definition of LCA on Wikipedia: “The lowes

Given the following binary tree:  root = [3,5,1,6,2,0,8,nu
```

```
Example 1:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
Output: 3
Explanation: The LCA of nodes 5 and 1 is 3.
Example 2:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
Output: 5
Explanation: The LCA of nodes 5 and 4 is 5, since a node ca


Note:

All of the nodes' values will be unique.
p and q are different and both values will exist in the bi
```

前置知识

- 递归

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是求解二叉树中，两个给定节点的最近的公共祖先。是一道非常经典的二叉树题目。

我们之前说过树是一种递归的数据结构，因此使用递归方法解决二叉树问题从写法上来看是最简单的，这道题目也不例外。

用递归的思路去思考树是一种非常重要的能力。

如果大家这样去思考的话，问题就会得到简化，我们的目标就是分别在左右子树进行查找p和q。 如果p没有在左子树，那么它一定在右子树（题目限定p一定在树中）， 反之亦然。

对于具体的代码而言就是，我们假设这个树就一个结构，然后尝试去解决，然后在适当地方去递归自身即可。 如下图所示：

我们来看下核心代码：

```
// 如果我们找到了p，直接进行返回，那如果下面就是q呢？ 其实这没有影响
if (!root || root === p || root === q) return root;
const left = lowestCommonAncestor(root.left, p, q); // 去
const right = lowestCommonAncestor(root.right, p, q);// 
if (!left) return right; // 左子树找不到，返回右子树
if (!right) return left; // 右子树找不到，返回左子树
return root; // 左右子树分别有一个，则返回root
```

> 如果没有明白的话，请多花时间消化一下

关键点解析

- 用递归的思路去思考树

代码

代码支持： JavaScript， Python3

- JavaScript Code:

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @param {TreeNode} p
 * @param {TreeNode} q
 * @return {TreeNode}
 */
var lowestCommonAncestor = function(root, p, q) {
  if (!root || root === p || root === q) return root;
  const left = lowestCommonAncestor(root.left, p, q);
  const right = lowestCommonAncestor(root.right, p, q);
  if (!left) return right; // 左子树找不到，返回右子树
  if (!right) return left; // 右子树找不到，返回左子树
  return root; // 左右子树分别有一个，则返回root
};
```

- Python Code:

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'Tr
        if not root or root == p or root == q:
            return root
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q

        if not left:
            return right
        if not right:
            return left
        else:
            return root
```

扩展

如果递归的结束条件改为 `if (!root || root.left === p || root.right === q) return root;` 代表的是什么意思，对结果有什么样的影响？

题目地址

https://leetcode.com/problems/search-a-2d-matrix-ii/description/

题目描述

```
Write an efficient algorithm that searches for a value in a

Integers in each row are sorted in ascending from left to r
Integers in each column are sorted in ascending from top to
Example:

Consider the following matrix:

[
  [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]
]
Given target = 5, return true.

Given target = 20, return false.
```

前置知识

- 数组

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

符合直觉的做法是两层循环遍历，时间复杂度是O(m * n), 有没有时间复杂度更好的做法呢？ 答案是有，那就是充分运用矩阵的特性（横向纵向都递增）， 我们可以从角落（左下或者右上）开始遍历，这样时间复杂度是O(m + n).

其中蓝色代表我们选择的起点元素， 红色代表目标元素。

关键点解析

- 从角落开始遍历，利用递增的特性简化时间复杂度

代码

代码支持：JavaScript, Python3

JavaScript Code:

```
/*
 * @lc app=leetcode id=240 lang=javascript
 *
 * [240] Search a 2D Matrix II
 *
 * https://leetcode.com/problems/search-a-2d-matrix-ii/desc
 *
 *
 */
/**
 * @param {number[][]} matrix
 * @param {number} target
 * @return {boolean}
 */
var searchMatrix = function(matrix, target) {
    if (!matrix || matrix.length === 0) return false;

    let colIndex = 0;
    let rowIndex = matrix.length - 1;
    while(rowIndex > 0 && target < matrix[rowIndex][colInde
        rowIndex --;
    }

    while(colIndex < matrix[0].length) {
        if (target === matrix[rowIndex][colIndex]) return t
        if (target > matrix[rowIndex][colIndex]) {
            colIndex ++;
        } else if (rowIndex > 0){
            rowIndex --;
        } else {
            return false;
        }
    }

    return  false;
};
```

Python Code:

```python
class Solution:
    def searchMatrix(self, matrix, target):
        m = len(matrix)
        if m == 0:
            return False
        n = len(matrix[0])
        i = m - 1
        j = 0

        while i >= 0 and j < n:
            if matrix[i][j] == target:
                return True
            if matrix[i][j] > target:
                i -= 1
            else:
                j += 1
        return False
```

题目地址

https://leetcode.com/problems/perfect-squares/description/

题目描述

```
Given a positive integer n, find the least number of perfe

Example 1:

Input: n = 12
Output: 3
Explanation: 12 = 4 + 4 + 4.
Example 2:

Input: n = 13
Output: 2
Explanation: 13 = 4 + 9.
```

前置知识

- 递归
- 动态规划

公司

- 阿里
- 百度
- 字节

思路

直接递归处理即可，但是这种暴力的解法很容易超时。如果你把递归的过程化成一棵树的话（其实就是递归树）， 可以看出中间有很多重复的计算。

如果能将重复的计算缓存下来，说不定能够解决时间复杂度太高的问题。

> 递归对内存的要求也很高， 如果数字非常大，也会面临爆栈的风险，将递归转化为循环可以解决。

递归 + 缓存的方式代码如下：

```
const mapper = {};

function d(n, level) {
  if (n === 0) return level;

  let i = 1;
  const arr = [];

  while (n - i * i >= 0) {
    const hit = mapper[n - i * i];
    if (hit) {
      arr.push(hit + level);
    } else {
      const depth = d(n - i * i, level + 1) - level;
      mapper[n - i * i] = depth;
      arr.push(depth + level);
    }
    i++;
  }

  return Math.min(...arr);
}
/**
 * @param {number} n
 * @return {number}
 */
var numSquares = function(n) {
  return d(n, 0);
};
```

如果使用 DP，其实本质上和递归 + 缓存 差不多。

DP 的代码见代码区。

关键点解析

- 如果用递归 + 缓存， 缓存的设计很重要 我的做法是 key 就是 n，value 是以 n 为起点，到达底端的深度。 下次取出缓存的时候用当前的 level + 存的深度 就是我们想要的 level.
- 使用动态规划的核心点还是选和不选的问题

```
for (let i = 1; i <= n; i++) {
  for (let j = 1; j * j <= i; j++) {
    // 不选 (dp[i])  还是  选 (dp[i - j * j])
    dp[i] = Math.min(dp[i], dp[i - j * j] + 1);
  }
}
```

代码

```
/*
 * @lc app=leetcode id=279 lang=javascript
 *
 * [279] Perfect Squares
 *
 * https://leetcode.com/problems/perfect-squares/descriptio
 *
 * algorithms
 * Medium (40.98%)
 * Total Accepted:    168.2K
 * Total Submissions: 408.5K
 * Testcase Example:  '12'
 *
 * Given a positive integer n, find the least number of per
 * (for example, 1, 4, 9, 16, ...) which sum to n.
 *
 * Example 1:
 *
 *
 * Input: n = 12
 * Output: 3
 * Explanation: 12 = 4 + 4 + 4.
 *
 * Example 2:
 *
 *
 * Input: n = 13
 * Output: 2
 * Explanation: 13 = 4 + 9.
 */
/**
 * @param {number} n
 * @return {number}
 */
var numSquares = function(n) {
  if (n <= 0) {
    return 0;
  }

  const dp = Array(n + 1).fill(Number.MAX_VALUE);
  dp[0] = 0;
  for (let i = 1; i <= n; i++) {
    for (let j = 1; j * j <= i; j++) {
      // 不选 (dp[i])  还是  选 (dp[i - j * j])
      dp[i] = Math.min(dp[i], dp[i - j * j] + 1);
    }
  }
```

```
  return dp[n];
};
```

题目地址

https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/description/

题目描述

```
Say you have an array for which the ith element is the pric

Design an algorithm to find the maximum profit. You may cor

You may not engage in multiple transactions at the same tir
After you sell your stock, you cannot buy stock on next day
Example:

Input: [1,2,3,0,2]
Output: 3
Explanation: transactions = [buy, sell, cooldown, buy, sel
```

前置知识

- 动态规划

公司

- 阿里
- 腾讯
- 字节

思路

这是一道典型的 DP 问题， DP 问题的核心是找到状态和状态转移方程。

这道题目的状态似乎比我们常见的那种 DP 问题要多，这里的状态有 buy sell cooldown 三种， 我们可以用三个数组来表示这这三个状态，buy,sell, cooldown.

- buy[i]表示第 i 天，且以 buy 结尾的最大利润
- sell[i]表示第 i 天，且以 sell 结尾的最大利润
- cooldown[i]表示第 i 天，且以 sell 结尾的最大利润

我们思考一下，其实 cooldown 这个状态数组似乎没有什么用，因此 cooldown 不会对 `profit` 产生 任何影响。 我们可以进一步缩小为两种状态。

- buy[i] 表示第 i 天，且以 buy 或者 coolwown 结尾的最大利润
- sell[i] 表示第 i 天，且以 sell 或者 cooldown 结尾的最大利润

对应的状态转移方程如下：

> 这个需要花点时间来理解

```
buy[i] = Math.max(buy[i - 1], sell[i - 2] - prices[i]);
sell[i] = Math.max(sell[i - 1], buy[i - 1] + prices[i]);
```

我们来分析一下，buy[i]对应第 i 的 action 只能是 buy 或者 cooldown。

- 如果是 cooldown，那么 profit 就是 buy[i - 1]
- 如果是 buy，那么就是 `前一个卖的profit减去今天买股票花的钱` ，即 sell[i -2] - prices[i]

> 注意这里是 i - 2，不是 i-1 ，因为有 cooldown 的限制

sell[i]对应第 i 的 action 只能是 sell 或者 cooldown。

- 如果是 cooldown，那么 profit 就是 sell[i - 1]
- 如果是 sell，那么就是 `前一次买的时候获取的利润加上这次卖的钱` ，即 buy[i - 1] + prices[i]

关键点解析

- 多状态动态规划

代码

```
/*
 * @lc app=leetcode id=309 lang=javascript
 *
 * [309] Best Time to Buy and Sell Stock with Cooldown
 *
 */
/**
 * @param {number[]} prices
 * @return {number}
 */
var maxProfit = function (prices) {
  if (prices == null || prices.length <= 1) return 0;

  // 定义状态变量
  const buy = [];
  const sell = [];
  // 寻常
  buy[0] = -prices[0];
  buy[1] = Math.max(-prices[0], -prices[1]);
  sell[0] = 0;
  sell[1] = Math.max(0, prices[1] - prices[0]);
  for (let i = 2; i < prices.length; i++) {
    // 状态转移方程
    // 第i天只能是买或者cooldown
    // 如果买利润就是sell[i - 2] - prices[i], 注意这里是i - 2,
    // cooldown就是buy[i -1]
    buy[i] = Math.max(buy[i - 1], sell[i - 2] - prices[i]);
    // 第i天只能是卖或者cooldown
    // 如果卖利润就是buy[i  -1] + prices[i]
    // cooldown就是sell[i -1]
    sell[i] = Math.max(sell[i - 1], buy[i - 1] + prices[i]
  }

  return Math.max(buy[prices.length - 1], sell[prices.lengt
};
```

相关题目

- 121.best-time-to-buy-and-sell-stock
- 122.best-time-to-buy-and-sell-stock-ii

题目地址

https://leetcode.com/problems/coin-change/description/

题目描述

```
You are given coins of different denominations and a total

Example 1:

Input: coins = [1, 2, 5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
Example 2:

Input: coins = [2], amount = 3
Output: -1
Note:
You may assume that you have an infinite number of each kir
```

前置知识

- 贪心算法
- 动态规划

公司

- 腾讯
- 百度
- 字节
- 阿里巴巴（盒马生鲜）

岗位信息

- 阿里巴巴（盒马生鲜）：前端技术二面

思路

思路

假如我们把 coin 逆序排列，然后逐个取，取到刚好不大于 amout，依次类推。

```
eg: 对于 [1,2,5] 组成 11 块

- 排序[5,2,1]

- 取第一个5, 更新amout 为 11 - 5 = 6 (1口)
      6 > 5 继续更新 为 6 - 5 = 1 (2口)
      1 < 5 退出

- 取第二个2
      1 < 2 退出

- 取最后一个元素，也就是1

      1 === 1 更新为 1 - 1 = 0 (3口)

- amout 为 0 退出


因此结果是 3
```

熟悉贪心算法的同学应该已经注意到了，这就是贪心算法，贪心算法更 amount 尽快地变得更小。 `经验表明，贪心策略是正确的` 。 注意，我说的是经验表明， 贪心算法也有可能出错。 就拿这道题目来说， 他也是不正确的！ 比如 `coins = [1, 5, 11] amout = 15` , 因此这种做法有时候不靠谱，我们还是采用靠谱的做法.

如果我们暴力求解，对于所有的组合都计算一遍，然后比较， 那么这样的复杂度是 2 的 n 次方（这个可以通过数学公式证明，这里不想啰嗦了）， 这个是不可以接受的。那么我们是否可以动态规划解决呢？答案是可以，原因就是可以划分为子问题，子问题可以推导出原问题

对于动态规划我们可以先画一个二维表，然后观察，其是否可以用一维表代替。 关于动态规划为什么要画表，我已经在这篇文章解释了

比较容易想到的是二维数组：

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) ->
        if amount < 0:
            return - 1
        dp = [[amount + 1 for _ in range(len(coins) + 1)]
              for _ in range(amount + 1)]
        # 初始化第一行为0，其他为最大值（也就是amount + 1）

        for j in range(len(coins) + 1):
            dp[0][j] = 0

        for i in range(1, amount + 1):
            for j in range(1, len(coins) + 1):
                if i - coins[j - 1] >= 0:
                    dp[i][j] = min(
                        dp[i][j - 1], dp[i - coins[j - 1]]
                else:
                    dp[i][j] = dp[i][j - 1]

        return -1 if dp[-1][-1] == amount + 1 else dp[-1][-
```

复杂度分析

- 时间复杂度：$O(amonut * len(coins))$
- 空间复杂度：$O(amount * len(coins))$

dp[i][j] 依赖于 `dp[i][j - 1]` 和 `dp[i - coins[j - 1]][j] + 1)` 这是一个优化的信号，我们可以将其优化到一维,具体见下方。

关键点解析

- 动态规划
- 子问题

用 dp[i] 来表示组成 i 块钱，需要最少的硬币数，那么

1. 第 j 个硬币我可以选择不拿 这个时候， 硬币数 = dp[i]
2. 第 j 个硬币我可以选择拿 这个时候， 硬币数 = dp[i - coins[j]] + 1
3. 和背包问题不同， 硬币是可以拿任意个
4. 对于每一个 dp[i] 我们都选择遍历一遍 coin， 不断更新 dp[i]

代码

- 语言支持：JS，C++,Python3

JavaScript Code：

```
var coinChange = function (coins, amount) {
  if (amount === 0) {
    return 0;
  }
  const dp = Array(amount + 1).fill(Number.MAX_VALUE);
  dp[0] = 0;
  for (let i = 1; i < dp.length; i++) {
    for (let j = 0; j < coins.length; j++) {
      if (i - coins[j] >= 0) {
        dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1);
      }
    }
  }

  return dp[dp.length - 1] === Number.MAX_VALUE ? -1 : dp[d
};
```

C++ Code：

> C++中采用 INT_MAX，因此判断时需要加上 `dp[a - coin] < INT_MAX` 以防止溢出

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        auto dp = vector<int>(amount + 1, INT_MAX);
        dp[0] = 0;
        for (auto a = 1; a <= amount; ++a) {
            for (const auto & coin : coins) {
                if (a >= coin && dp[a - coin] < INT_MAX) {
                    dp[a] = min(dp[a], dp[a-coin] + 1);
                }
            }
        }
        return dp[amount] == INT_MAX ? -1 : dp[amount];
    }
};
```

Python3 Code:

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) ->
        dp = [amount + 1] * (amount + 1)
        dp[0] = 0

        for i in range(1, amount + 1):
            for j in range(len(coins)):
                if i >= coins[j]:
                    dp[i] = min(dp[i], dp[i - coins[j]] + 1

        return -1 if dp[-1] == amount + 1 else dp[-1]
```

复杂度分析

- 时间复杂度：$O(amonut * len(coins))$
- 空间复杂度：$O(amount)$

扩展

这是一道很简单描述的题目， 因此很多时候会被用到大公司的电面中。

相似问题:

518.coin-change-2

题目地址

https://leetcode.com/problems/odd-even-linked-list/description/

题目描述

```
Given a singly linked list, group all odd nodes together fo

You should try to do it in place. The program should run in

Example 1:

Input: 1->2->3->4->5->NULL
Output: 1->3->5->2->4->NULL
Example 2:

Input: 2->1->3->5->6->4->7->NULL
Output: 2->3->6->7->1->5->4->NULL
Note:

The relative order inside both the even and odd groups shou
The first node is considered odd, the second node even and
```

前置知识

- 链表

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

符合直觉的想法是，先遍历一遍找出奇数的节点。然后再遍历一遍找出偶数节点，最后串起来。

但是有两个问题，如果不修改节点的话，需要借助额外的空间，空间复杂度是 N。如果修改的话，会对第二次遍历（遍历偶数节点）造成影响。

因此可以采用一种做法： 遍历一次，每一步同时修改两个节点就好了，这样就可以规避上面两个问题。

关键点解析

- 用虚拟节点来简化操作
- 循环的结束条件设置为 `odd && odd.next && even && even.next` ，不应该是 `odd && even` ，否则需要记录一下奇数节点的最后一个节点，复杂了操作

代码

- 语言支持：JS，C++

JavaScript Code:

```javascript
/*
 * @lc app=leetcode id=328 lang=javascript
 *
 * [328] Odd Even Linked List
 *
 *
 */
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */
var oddEvenList = function(head) {
  if (!head || !head.next) return head;

  const dummyHead1 = {
    next: head
  };
  const dummyHead2 = {
    next: head.next
  };

  let odd = dummyHead1.next;
  let even = dummyHead2.next;

  while (odd && odd.next && even && even.next) {
    const oddNext = odd.next.next;
    const evenNext = even.next.next;

    odd.next = oddNext;
    even.next = evenNext;

    odd = oddNext;
    even = evenNext;
  }

  odd.next = dummyHead2.next;

  return dummyHead1.next;
};
```

C++ Code：

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if (head == nullptr) return head;
        auto odd = head, evenHead = head->next, even = head
        // 因为“每次循环之后依然保持odd在even之前”，循环条件可以只
        while (even != nullptr && even->next != nullptr) {
            odd->next = even->next;
            odd = odd->next;
            even->next = odd->next;
            even = even->next;
        }
        odd->next = evenHead;
        return head;
    }
};
```

题目地址

https://leetcode.com/problems/increasing-triplet-subsequence/description/

题目描述

```
Given an unsorted array return whether an increasing subsec

Formally the function should:

Return true if there exists i, j, k
such that arr[i] < arr[j] < arr[k] given 0 ≤ i < j < k ≤ n-
Note: Your algorithm should run in O(n) time complexity anc

Example 1:

Input: [1,2,3,4,5]
Output: true
Example 2:

Input: [5,4,3,2,1]
Output: false
```

前置知识

- 双指针

公司

- 百度
- 字节

思路

这道题是求解顺序数字是否有三个递增的排列， 注意这里没有要求连续的，因此诸如滑动窗口的思路是不可以的。 题目要求O(n)的时间复杂度和O(1)的空间复杂度，因此暴力的做法就不用考虑了。

我们的目标就是 `依次` 找到三个数字，其顺序是递增的。因此我们的做法可以是依次遍历， 然后维护三个变量，分别记录最小值，第二小值，第三小值。只要我们能够填满这三个变量就返回true，否则返回false。

关键点解析

- 维护三个变量，分别记录最小值，第二小值，第三小值。只要我们能够填满这三个变量就返回true，否则返回false

代码

```
/*
 * @lc app=leetcode id=334 lang=javascript
 *
 * [334] Increasing Triplet Subsequence
 *
 * https://leetcode.com/problems/increasing-triplet-subsequ
 *
 * algorithms
 * Medium (39.47%)
 * Total Accepted:    89.6K
 * Total Submissions: 226.6K
 * Testcase Example:  '[1,2,3,4,5]'
 *
 * Given an unsorted array return whether an increasing sub
 * exists or not in the array.
 *
 * Formally the function should:
 *
 * Return true if there exists i, j, k
 * such that arr[i] < arr[j] < arr[k] given 0 ≤ i < j < k ≤
 * false.
 *
 * Note: Your algorithm should run in O(n) time complexity
 * complexity.
 *
 *
 * Example 1:
 *
 *
 * Input: [1,2,3,4,5]
 * Output: true
 *
 *
 *
 * Example 2:
 *
 *
 * Input: [5,4,3,2,1]
 * Output: false
 *
 *
 *
 */
/**
 * @param {number[]} nums
 * @return {boolean}
```

```
 */
var increasingTriplet = function(nums) {
    if (nums.length < 3) return false;
    let n1 = Number.MAX_VALUE;
    let n2 = Number.MAX_VALUE;

    for(let i = 0; i < nums.length; i++) {
        if (nums[i] <= n1) {
            n1 = nums[i]
        } else if (nums[i] <= n2) {
            n2 = nums[i]
        } else {
            return true;
        }
    }

    return false;
};
```

题目地址(337. 打家劫舍 III)

https://leetcode-cn.com/problems/house-robber-iii/

题目描述

```
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。

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.
```

前置知识

- 二叉树
- 动态规划

公司

- 阿里

- 腾讯
- 百度
- 字节

思路

和 198.house-robber 类似，这道题也是相同的思路。 只不过数据结构从数组换成了树。

我们仍然是对每一项进行决策：**如果我抢的话，所得到的最大价值是多少。如果我不抢的话，所得到的最大价值是多少。**

- 遍历二叉树，都每一个节点我们都需要判断抢还是不抢。
    - 如果抢了的话， 那么我们不能继续抢其左右子节点
    - 如果不抢的话，那么我们可以继续抢左右子节点，当然也可以不抢。抢不抢取决于哪个价值更大。
- 抢不抢取决于哪个价值更大。

这是一个明显的递归问题，我们使用递归来解决。由于没有重复子问题，因此没有必要 cache ， 也没有必要动态规划。

关键点

- 对每一个节点都分析，是抢还是不抢

代码

语言支持：JS, C++，Java，Python

JavaScript Code：

```
function helper(root) {
  if (root === null) return [0, 0];
  // 0: rob 1: notRob
  const l = helper(root.left);
  const r = helper(root.right);

  const robed = root.val + l[1] + r[1];
  const notRobed = Math.max(l[0], l[1]) + Math.max(r[0], r

  return [robed, notRobed];
}
/**
 * @param {TreeNode} root
 * @return {number}
 */
var rob = function (root) {
  const [robed, notRobed] = helper(root);
  return Math.max(robed, notRobed);
};
```

C++ Code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {
 * };
 */
class Solution {
public:
    int rob(TreeNode* root) {
        pair<int, int> res = dfs(root);
        return max(res.first, res.second);
    }

    pair<int, int> dfs(TreeNode* root)
    {
        pair<int, int> res = {0, 0};
        if(root == NULL)
        {
            return res;
        }

        pair<int, int> left = dfs(root->left);
        pair<int, int> right = dfs(root->right);
        // 0 代表不偷，1 代表偷
        res.first = max(left.first, left.second) + max(rig
        res.second = left.first + right.first + root->val;
        return res;
    }

};
```

Java Code:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public int rob(TreeNode root) {
        int[] res = dfs(root);
        return Math.max(res[0], res[1]);
    }

    public int[] dp(TreeNode root)
    {
        int[] res = new int[2];
        if(root == null)
        {
            return res;
        }

        int[] left = dfs(root.left);
        int[] right = dfs(root.right);
        // 0 代表不偷，1 代表偷
        res[0] = Math.max(left[0], left[1]) + Math.max(rig
        res[1] = left[0] + right[0] + root.val;
        return res;
    }
}
```

Python Code:

```
class Solution:
    def rob(self, root: TreeNode) -> int:
        def dfs(node):
            if not node:
                return [0, 0]
            [l_rob, l_not_rob] = dfs(node.left)
            [r_rob, r_not_rob] = dfs(node.right)
            return [node.val + l_not_rob + r_not_rob, max(
        return max(dfs(root))


# @lc code=end
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为树的节点个数。
- 空间复杂度：$O(h)$，其中 h 为树的高度。

相关题目

- 198.house-robber

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 35K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(343. 整数拆分)

https://leetcode-cn.com/problems/integer-break/

题目描述

给定一个正整数 n，将其拆分为至少两个正整数的和，并使这些整数的乘积最大化。 返回你可以获得的最大乘积。

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。 说明: 你可以假设 n 不小于 2 且不大于 58。

前置知识

- 递归
- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

希望通过这篇题解让大家知道“题解区的水有多深”，让大家知道“什么才是好的题解”。

我看了很多人的题解直接就是两句话，然后跟上代码:

```
class Solution:
    def integerBreak(self, n: int) -> int:
        dp = [1] * (n + 1)
        for i in range(3, n + 1):
            for j in range(1, i):
                dp[i] = max(j * dp[i - j], j * (i - j), dp
        return dp[n]
```

这种题解说实话，只针对那些”自己会， 然后去题解区看看有没有新的更好的解法的人“。但是大多数看题解的人是那种 `自己没思路，不会做的人` 。那么这种题解就没什么用了。

我认为 `好的题解应该是新手友好的，并且能够将解题人思路完整展现的题解` 。比如看到这个题目，我首先想到了什么（对错没有关系），然后头脑中经过怎么样的筛选将算法筛选到具体某一个或某几个。我的最终算法是如何想到的，有没有一些先行知识。

当然我也承认自己有很多题解也是直接给的答案，这对很多人来说用处不大，甚至有可能有反作用，给他们一种”我已经会了“的假象。实际上他们根本不懂解题人本身原本的想法， 也许是写题解的人觉得”这很自然“，也可能”只是为了秀技“。

Ok，下面来讲下 `我是如何解这道题的` 。

抽象

首先看到这道题，自然而然地先对问题进行抽象，这种抽象能力是必须的。LeetCode 实际上有很多这种穿着华丽外表的题，当你把这个衣服扒开的时候，会发现都是差不多的，甚至两个是一样的，这样的例子实际上有很多。 就本题来说，就有一个剑指 Offer 的原题《剪绳子》和其本质一样，只是换了描述方式。类似的有力扣 137 和 645 等等，大家可以自己去归纳总结。

> 137 和 645 我贴个之前写的题解 https://leetcode-cn.com/problems/single-number/solution/zhi-chu-xian-yi-ci-de-shu-xi-lie-wei-yun-suan-by-3/

培养自己抽象问题的能力，不管是在算法上还是工程上。 务必记住这句话！

数学是一门非常抽象的学科，同时也很方便我们抽象问题。为了显得我的题解比较高级，引入一些你们看不懂的数学符号也是很有必要的（开玩笑，没有什么高级数学符号啦）。

> 实际上这道题可以用纯数学角度来解，但是我相信大多数人并不想看。即使你看了，大多人的感受也是“好 nb，然而并没有什么用”。

这道题抽象一下就是：

令：

$$n == \sum_{i=1}^{n} n_i$$

（图 1） 求：

$$max(\prod_{i=1}^{n} n_i)$$

（图 2）

第一直觉

经过上面的抽象，我的第一直觉这可能是一个数学题，我回想了下数学知识，然后用数学法 AC 了。 数学就是这么简单平凡且枯燥。

然而如果没有数学的加持的情况下，我继续思考怎么做。我想是否可以枚举所有的情况（如图 1），然后对其求最大值（如图 2）。

问题转化为如何枚举所有的情况。经过了几秒钟的思考，我发现这是一个很明显的递归问题。 具体思考过程如下：

- 我们将原问题抽象为 f(n)
- 那么 f(n) 等价于 max(1 * fn(n - 1), 2 * f(n - 2), ..., (n - 1) * f(1))。

用数学公式表示就是：

$$max(\prod_{i=1}^{n} i * f(n-i)))$$

（图 3）

截止目前，是一点点数学 + 一点点递归，我们继续往下看。现在问题是不是就很简单啦？直接翻译图三为代码即可，我们来看下这个时候的代码：

```
class Solution:
    def integerBreak(self, n: int) -> int:
        if n == 2: return 1
        res = 0
        for i in range(1, n):
            res = max(res, max(i * self.integerBreak(n - i
        return res
```

毫无疑问，超时了。原因很简单，就是算法中包含了太多的重复计算。如果经常看我的题解的话，这句话应该不陌生。我随便截一个我之前讲过这个知识点的图。

> 一个人爬楼梯，每次只能爬 1 个或 2 个台阶，假设有 n 个台阶，那么这个人有多少种不同的爬楼梯方法?
>
> 代码：
>
> ```
> function climbStairs(n) {
>  if (n === 1) return 1;
>  if (n === 2) return 2;
>  return climbStairs(n - 1) + climbStairs(n - 2);
> }
> ```
>
> 这道题和 fibnacci 数列一摸一样，我们继续用一个递归树来直观感受以下：
>
>
>
>
> 可以看出这里面有很多重复计算，我们可以使用一个 hashtable 去缓存中间计算结果，从而省去不必要的计算。 那么动态规划是怎么解决这个问题呢？ 答案就是“查表”。

(图 4)

> 原文链接：https://github.com/azl397985856/leetcode/blob/master/thinkings/dynamic-programming.md

大家可以尝试自己画图理解一下。

> 看到这里，有没有种殊途同归的感觉呢?

考虑优化

如上，我们可以考虑使用记忆化递归的方式来解决。只是用一个 hashtable 存储计算过的值即可。

```
class Solution:
    @lru_cache()
    def integerBreak(self, n: int) -> int:
        if n == 2: return 1
        res = 0
        for i in range(1, n):
            res = max(res, max(i * self.integerBreak(n - i
        return res
```

为了简单起见（偷懒起见），我直接用了 lru_cache 注解， 上面的代码是可以 AC 的。

动态规划

看到这里的同学应该发现了，这个套路是不是很熟悉？下一步就是将其改造成动态规划了。

如图 4，我们的思考方式是从顶向下，这符合人们思考问题的方式。将其改造成如下图的自底向上方式就是动态规划。

动态规划的查表过程如果画成图，就是这样的：

(图 5)

现在再来看下文章开头的代码：

```
class Solution:
    def integerBreak(self, n: int) -> int:
        dp = [1] * (n + 1)
        for i in range(3, n + 1):
            for j in range(1, i):
                dp[i] = max(j * dp[i - j], j * (i - j), dp
        return dp[n]
```

dp table 存储的是图 3 中 f(n)的值。一个自然的想法是令 dp[i] 等价于 f(i)。而由于上面分析了原问题等价于 f(n)，那么很自然的原问题也等价于 dp[n]。

而 dp[i]等价于 f(i)，那么上面针对 f(i) 写的递归公式对 dp[i] 也是适用的，我们拿来试试。

```
// 关键语句
res = max(res, max(i * self.integerBreak(n - i),i * (n - i
```

翻译过来就是：

```
dp[i] = max(dp[i], max(i * dp(n - i),i * (n - i)))
```

而这里的 n 是什么呢？我们说了 `dp是自底向下的思考方式` ，那么在达到 n 之前是看不到整体的 `n` 的。因此这里的 n 实际上是 1,2,3,4... n。

自然地，我们用一层循环来生成上面一系列的 n 值。接着我们还要生成一系列的 i 值，注意到 n - i 是要大于 0 的，因此 i 只需要循环到 n - 1 即可。

思考到这里，我相信上面的代码真的是 `不难得出` 了。

关键点

- 数学抽象
- 递归分析
- 记忆化递归
- 动态规划

代码

```
class Solution:
    def integerBreak(self, n: int) -> int:
        dp = [1] * (n + 1)
        for i in range(3, n + 1):
            for j in range(1, i):
                dp[i] = max(j * dp[i - j], j * (i - j), dp
        return dp[n]
```

总结

培养自己的解题思维很重要， 不要直接看别人的答案。而是要将别人的东西变成自己的， 而要做到这一点，你就要知道“他们是怎么想到的”，“想到这点是不是有什么前置知识”，“类似题目有哪些”。

最优解通常不是一下子就想到了，这需要你在不那么优的解上摔了很多次跟头之后才能记住的。因此在你没有掌握之前，不要直接去看最优解。在你掌握了之后，我不仅鼓励你去写最优解，还鼓励去一题多解，从多个解决思考问题。 到了那个时候， 萌新也会惊讶地呼喊“哇塞， 这题还可以这么解啊？”。 你也会低调地发出“害，解题就是这么简单平凡且枯燥。”的声音。

扩展

正如我开头所说，这种套路实在是太常见了。希望大家能够识别这种问题的本质，彻底掌握这种套路。另外我对这个套路也在我的新书《LeetCode 题解》中做了介绍，本书目前刚完成草稿的编写，如果你想要第一时间获取到我们的题解新书，那么请发送邮件到 `azl397985856@gmail.com` ，标题著明“书籍《LeetCode 题解》预定”字样。。

题目地址

https://leetcode.com/problems/water-and-jug-problem/description/

题目描述

```
You are given two jugs with capacities x and y litres. Ther

If z liters of water is measurable, you must have z liters

Operations allowed:

Fill any of the jugs completely with water.
Empty any of the jugs.
Pour water from one jug into another till the other jug is
Example 1: (From the famous "Die Hard" example)

Input: x = 3, y = 5, z = 4
Output: True
Example 2:

Input: x = 2, y = 6, z = 5
Output: False
```

BFS（超时）

前置知识

- BFS
- 最大公约数

公司

- 阿里
- 百度
- 字节

思路

两个水壶的水我们考虑成状态，然后我们不断进行倒的操作，改变状态。那么初始状态就是（0 0） 目标状态就是 （any, z）或者 (z, any)，其中 any 指的是任意升水。

已题目的例子，其过程示意图，其中括号表示其是由哪个状态转移过来的：

0 0 3 5(0 0) 3 0 (0 0)0 5(0 0) 3 2(0 5) 0 3(0 0) 0 2(3 2) 2 0(0 2) 2 5(2 0) 3 4(2 5) bingo

代码

```
class Solution:
    def canMeasureWater(self, x: int, y: int, z: int) -> bo
        if x + y < z:
            return False
        queue = [(0, 0)]
        seen = set((0, 0))

        while(len(queue) > 0):
            a, b = queue.pop(0)
            if a ==z or b == z or a + b == z:
                return True
            states = set()

            states.add((x, b))
            states.add((a, y))
            states.add((0, b))
            states.add((a, 0))
            states.add((min(x, b + a), 0 if b < x - a else
            states.add((0 if a + b < y else a - (y - b), m
            for state in states:
                if state in seen:
                    continue;
                queue.append(state)
                seen.add(state)
        return False
```

复杂度分析

- 时间复杂度：由于状态最多有$O((x+1)*(y+1))$ 种，因此总的时间复杂度为$O(x*y)$。
- 空间复杂度：我们使用了队列来存储状态，set 存储已经访问的元素，空间复杂度和状态数目一致，因此空间复杂度是$O(x*y)$。

上面的思路很直观，但是很遗憾这个算法在 LeetCode 的表现是 TLE(Time Limit Exceeded)。不过如果你能在真实面试中写出这样的算法，我相信大多数情况是可以过关的。

我们来看一下有没有别的解法。实际上，上面的算法就是一个标准的 BFS。如果从更深层次去看这道题，会发现这道题其实是一道纯数学问题，类似的纯数学问题在 LeetCode 中也会有一些，不过大多数这种题目，我们仍然可以采取其他方式 AC。那么让我们来看一下如何用数学的方式来解这个题。

数学法 - 最大公约数

思路

这是一道关于 `数论` 的题目，确切地说是关于 `裴蜀定理` （英语：Bézout's identity）的题目。

摘自wiki的定义：

```
对任意两个整数 a、b，设 d是它们的最大公约数。那么关于未知数  x和  y的

ax+by=m

有整数解 (x,y) 当且仅当  m是  d的整数倍。裴蜀等式有解时必然有无穷多
```

因此这道题可以完全转化为 `裴蜀定理` 。还是以题目给的例子 `x = 3, y = 5, z = 4` ，我们其实可以表示成 `3 * 3 - 1 * 5 = 4` , 即 `3 * x - 1 * y = z` 。我们用a和b分别表示3 升的水壶和5升的水壶。那么我们可以：

- 倒满a（**1**）
- 将a倒到b
- 再次倒满a（**2**）
- 再次将a倒到b（a这个时候还剩下1升）
- 倒空b（**-1**）
- 将剩下的1升倒到b
- 将a倒满（**3**）
- 将a倒到b
- b此时正好是4升

上面的过程就是 `3 * x - 1 * y = z` 的具体过程解释。

也就是说我们只需要求出x和y的最大公约数d，并判断z是否是d的整数倍即可。

代码

代码支持：Python3，JavaScript

Python Code：

```
class Solution:
    def canMeasureWater(self, x: int, y: int, z: int) -> bo
        if x + y < z:
            return False

        if (z == 0):
            return True

        if (x == 0):
            return y == z

        if (y == 0):
            return x == z

        def GCD(a, b):
            smaller = min(a, b)
            while smaller:
                if a % smaller == 0 and b % smaller == 0:
                    return smaller
                smaller -= 1

        return z % GCD(x, y) == 0
```

JavaScript：

```
/**
 * @param {number} x
 * @param {number} y
 * @param {number} z
 * @return {boolean}
 */
var canMeasureWater = function(x, y, z) {
  if (x + y < z) return false;

  if (z === 0) return true;

  if (x === 0) return y === z;

  if (y === 0) return x === z;

  function GCD(a, b) {
    let min = Math.min(a, b);
    while (min) {
      if (a % min === 0 && b % min === 0) return min;
      min--;
    }
    return 1;
  }

  return z % GCD(x, y) === 0;
};
```

实际上求最大公约数还有更好的方式，比如辗转相除法：

```
def GCD(a, b):
    if b == 0: return a
    return GCD(b, a % b)
```

复杂度分析

- 时间复杂度：$O(log(max(a,b)))$
- 空间复杂度：空间复杂度取决于递归的深度，因此空间复杂度为 $O(log(max(a,b)))$

关键点分析

- 数论
- 裴蜀定理

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经接近30K star啦。

大家也可以关注我的公众号《脑洞前端》获取更多更新鲜的LeetCode题解

题目地址

https://leetcode.com/problems/kth-smallest-element-in-a-sorted-matrix/description/

题目描述

```
Given a n x n matrix where each of the rows and columns are

Note that it is the kth smallest element in the sorted orde

Example:

matrix = [
   [ 1,  5,  9],
   [10, 11, 13],
   [12, 13, 15]
],
k = 8,

return 13.
Note:
You may assume k is always valid, 1 ≤ k ≤ n2.
```

前置知识

- 二分查找
- 堆

公司

- 阿里
- 腾讯
- 字节

思路

显然用大顶堆可以解决，时间复杂度 Klogn n 为总的数字个数, 但是这种做法没有利用题目中 sorted matrix 的特点，因此不是一种好的做法.

一个巧妙的方法是二分法，我们分别从第一个和最后一个向中间进行扫描，并且计算出中间的数值与数组中的进行比较， 可以通过计算中间值在这个数组中排多少位，然后得到比中间值小的或者大的数字有多少个，然后与 k 进行比较，如果比 k 小则说明中间值太小了，则向后移动，否则向前移动。

这个题目的二分确实很难想，我们来一步一步解释。

最普通的二分法是有序数组中查找指定值（或者说满足某个条件的值）。由于是有序的，我们可以根据索引关系来确定大小关系， 因此这种思路比较直接，但是对于这道题目索引大小和数字大小没有直接的关系，因此这种二分思想就行不通了。

(普通的基于索引判断的二分法)

- 我们能够找到矩阵中最大的元素（右下角）和最小的元素（左上角）。我们可以求出值的中间，而不是上面那种普通二分法的索引的中间。

- 找到中间值之后，我们可以拿这个值去计算有多少元素是小于等于它的。 具体方式就是比较行的最后一列，如果中间值比最后一列大，说明中间元素肯定大于这一行的所有元素。 否则我们从后往前遍历直到不大于。

- 上一步我们会计算一个count，我们拿这个count和k进行比较
- 如果count小于k，说明我们选择的中间值太小了，肯定不符合条件，我们需要调整左区间为mid + 1
- 如果count大于k，说明我们选择的中间值正好或者太大了。我们调整右区间 mid

> 由于count大于k 也可能我们选择的值是正好的， 因此这里不能调整为mid - 1， 否则可能会得不到结果

- 最后直接返回start, end, 或者 mid都可以，因此三者最终会收敛到矩阵中的一个元素，这个元素也正是我们要找的元素。

整个计算过程是这样的：

这里有一个大家普遍都比较疑惑的点，也是我当初非常疑惑，困扰我很久的点， leetcode评论区也有很多人来问，就是“能够确保最终我们找到的元素一定在矩阵中么？”

答案是可以，`相等的时候一定在matrix里面。 因为原问题一定有解，找下界使得start不断的逼近于真实的元素` .

我是看了评论区一个大神的评论才明白的，以下是@GabrielaSong的评论原文：

```
The lo we returned is guaranteed to be an element in the ma
Let us assume element m is the kth smallest number in the r
When we are about to reach convergence, if mid=m-1, its cou
so we would set lo as (m-1)+1=m, in this case the hi will t
and if mid=m+1, its count value would be k+x-1, so we would
To sum up, because the number lo found by binary search fir
 The equal sign guarantees there exists and only exists one
 So lo must be the only element satisfying this element in
```

更多解释,可以参考leetcode discuss

> 如果是普通的二分查找，我们是基于索引去找，因此不会有这个问题。

关键点解析

- 二分查找
- 有序矩阵的套路（文章末尾还有一道有序矩阵的题目）
- 堆（优先级队列）

代码

```
/*
 * @lc app=leetcode id=378 lang=javascript
 *
 * [378] Kth Smallest Element in a Sorted Matrix
 */
function notGreaterCount(matrix, target) {
  // 等价于在matrix 中搜索mid，搜索的过程中利用有序的性质记录比mid小

  // 我们选择左下角，作为开始元素
  let curRow = 0;
  // 多少列
  const COL_COUNT = matrix[0].length;
  // 最后一列的索引
  const LAST_COL = COL_COUNT - 1;
  let res = 0;

  while (curRow < matrix.length) {
    // 比较最后一列的数据和target的大小
    if (matrix[curRow][LAST_COL] < target) {
      res += COL_COUNT;
    } else {
      let i = COL_COUNT - 1;
      while (i < COL_COUNT && matrix[curRow][i] > target) {
        i--;
      }
      // 注意这里要加1
      res += i + 1;
    }
    curRow++;
  }

  return res;
}
/**
 * @param {number[][]} matrix
 * @param {number} k
 * @return {number}
 */
var kthSmallest = function(matrix, k) {
  if (matrix.length < 1) return null;
  let start = matrix[0][0];
  let end = matrix[matrix.length - 1][matrix[0].length - 1]
  while (start < end) {
    const mid = start + ((end - start) >> 1);
    const count = notGreaterCount(matrix, mid);
    if (count < k) start = mid + 1;
    else end = mid;
  }
```

```
  // 返回start,mid, end 都一样
  return start;
};
```

相关题目

- 240.search-a-2-d-matrix-ii

题目地址（380. 常数时间插入、删除和获取随机元素）

https://leetcode-cn.com/problems/insert-delete-getrandom-o1/description/

题目描述

```
设计一个支持在平均 时间复杂度 O(1) 下，执行以下操作的数据结构。

insert(val)：当元素 val 不存在时，向集合中插入该项。
remove(val)：元素 val 存在时，从集合中移除该项。
getRandom：随机返回现有集合中的一项。每个元素应该有相同的概率被返回。
示例 ：

// 初始化一个空的集合。
RandomizedSet randomSet = new RandomizedSet();

// 向集合中插入 1 。返回 true 表示 1 被成功地插入。
randomSet.insert(1);

// 返回 false ，表示集合中不存在 2 。
randomSet.remove(2);

// 向集合中插入 2 。返回 true 。集合现在包含 [1,2] 。
randomSet.insert(2);

// getRandom 应随机返回 1 或 2 。
randomSet.getRandom();

// 从集合中移除 1 ，返回 true 。集合现在包含 [2] 。
randomSet.remove(1);

// 2 已在集合中，所以返回 false 。
randomSet.insert(2);

// 由于 2 是集合中唯一的数字，getRandom 总是返回 2 。
randomSet.getRandom();
```

前置知识

- 数组
- 哈希表

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这是一个设计题。这道题的核心就是考察基本数据结构和算法的操作以及复杂度。

我们来回顾一下基础知识：

- 数组支持随机访问，其按照索引查询的时间复杂度为$O(1)$,按值查询的时间复杂度为$O(N)$， 而插入和删除的时间复杂度为$O(N)$。
- 链表不支持随机访问，其查询的时间复杂度为$O(N)$,但是对于插入和删除的复杂度为$O(1)$（不考虑找到选要处理的节点花费的时间）。
- 对于哈希表，正常情况下其查询复杂度平均为$O(N)$，插入和删除的复杂度为$O(1)$。

由于题目要求 getRandom 返回要随机并且要在$O(1)$复杂度内，那么如果单纯使用链表或者哈希表肯定是不行的。

而又由于对于插入和删除也需要复杂度为$O(1)$，因此单纯使用数组也是不行的，因此考虑多种使用数据结构来实现。

> 实际上 LeetCode 设计题，几乎没有单纯一个数据结构搞定的，基本都需要多种数据结构结合，这个时候需要你对各种数据结构以及其基本算法的复杂度有着清晰的认知。

对于 getRandom 用数组很简单。对于判断是否已经有了存在的元素，我们使用哈希表也很容易做到。因此我们可以将数组随机访问，以及哈希表$O(1)$按检索值的特性结合起来，即同时使用这两种数据结构。

对于删除和插入，我们需要一些技巧。

对于插入：

- 我们直接往 append，并将其插入哈希表即可。
- 对于删除，我们需要做到 O(1)。删除哈希表很明显可以，但是对于数组，平均时间复杂度为 O(1)。

因此如何应付删除的这种性能开销呢？ 我们知道对于数据删除，我们的时间复杂度来源于

1. `查找到要删除的元素`
2. 以及 `重新排列被删除元素后面的元素` 。

对于 1，我们可以通过哈希表来实现。 key 是插入的数字，value 是数组对应的索引。删除的时候我们根据 key 反查出索引就可以快速找到。

> 题目说明了不会存在重复元素，所以我们可以这么做。思考一下，如果没有这个限制会怎么样？

对于 2，我们可以通过和数组最后一项进行交换的方式来实现，这样就避免了数据移动。同时数组其他项的索引仍然保持不变，非常好！

> 相应地，我们插入的时候，需要维护哈希表

图解：

以依次【1，2，3，4】之后为初始状态，那么此时状态是这样的：

而当要插入一个新的5的时候， 我们只需要分别向数组末尾和哈希表中插入这条记录即可。

而删除的时候稍微有一点复杂：

关键点解析

- 数组
- 哈希表
- 数组 + 哈希表
- 基本算法时间复杂度分析

代码

```
from random import random


class RandomizedSet:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.data = dict()
        self.arr = []
        self.n = 0

    def insert(self, val: int) -> bool:
        """
        Inserts a value to the set. Returns true if the set
        """
        if val in self.data:
            return False
        self.data[val] = self.n
        self.arr.append(val)
        self.n += 1

        return True

    def remove(self, val: int) -> bool:
        """
        Removes a value from the set. Returns true if the s
        """
        if val not in self.data:
            return False
        i = self.data[val]
        # 更新data
        self.data[self.arr[-1]] = i
        self.data.pop(val)
        # 更新arr
        self.arr[i] = self.arr[-1]
        # 删除最后一项
        self.arr.pop()
        self.n -= 1

        return True

    def getRandom(self) -> int:
        """
        Get a random element from the set.
        """
```

```
        return self.arr[int(random() * self.n)]


# Your RandomizedSet object will be instantiated and called
# obj = RandomizedSet()
# param_1 = obj.insert(val)
# param_2 = obj.remove(val)
# param_3 = obj.getRandom()
```

复杂度分析

- 时间复杂度：$O(1)$
- 空间复杂度：$O(1)$

更多题解可以访问我的LeetCode题解仓库：https://github.com/azl397985856/leetcode 。 目前已经30K star啦。

大家也可以关注我的公众号《力扣加加》获取更多更新鲜的LeetCode题解

题目地址

416. 分割等和子集

题目描述

> 给定一个只包含正整数的非空数组。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。
>
> 注意:
>
> 每个数组中的元素不会超过 100 数组的大小不会超过 200
>
> 示例 1:
>
> 输入: [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: [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.

前置知识

- DFS
- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

抽象能力不管是在工程还是算法中都占据着绝对重要的位置。比如上题我们可以抽象为：

给定一个非空数组，和是 sum，能否找到这样的一个子序列，使其和为 2/sum

我们做过二数和，三数和， 四数和，看到这种类似的题会不会舒适一点，思路更开阔一点。

老司机们看到转化后的题，会立马想到背包问题，这里会提供**深度优先搜索**和**背包**两种解法。

深度优先遍历

我们再来看下题目描述，sum 有两种情况，

1. 如果 sum % 2 === 1, 则肯定无解，因为 sum/2 为小数，而数组全由整数构成，子数组和不可能为小数。
2. 如果 sum % 2 === 0, 需要找到和为 2/sum 的子序列 针对 2，我们要在 nums 里找到满足条件的子序列 subNums。 这个过程可以类比为在一个大篮子里面有 N 个球，每个球代表不同的数字，我们用一小篮子去抓取球，使得拿到的球数字和为 2/sum。那么很自然的一个想法就是，对大篮子里面的每一个球，我们考虑取它或者不取它，如果我们足够耐心，最后肯定能穷举所有的情况，判断是否有解。上述思维表述为伪代码如下：

```
令 target = sum / 2, nums 为输入数组, cur 为当前当前要选择的数字
nums 为输入数组，target为当前求和目标，cur为当前判断的数
function dfs(nums, target, cur)
  如果target < 0 或者 cur > nums.length
    return false
  否则
    如果 target = 0, 说明找到答案了，返回true
    否则
      取当前数或者不取，进入递归 dfs(nums, target - nums[cur],
```

因为对每个数都考虑取不取，所以这里时间复杂度是 O(2 ^ n), 其中 n 是 nums 数组长度，

javascript 实现

```
var canPartition = function (nums) {
  let sum = nums.reduce((acc, num) => acc + num, 0);
  if (sum % 2) {
    return false;
  }
  sum = sum / 2;
  return dfs(nums, sum, 0);
};

function dfs(nums, target, cur) {
  if (target < 0 || cur > nums.length) {
    return false;
  }
  return (
    target === 0 ||
    dfs(nums, target - nums[cur], cur + 1) ||
    dfs(nums, target, cur + 1)
  );
}
```

不出所料，这里是超时了，我们看看有没优化空间

1. 如果 nums 中最大值 > 2/sum, 那么肯定无解
2. 在搜索过程中，我们对每个数都是取或者不取，并且数组中所有项都为正数。我们设取的数和为 `pickedSum` ，不难得 pickedSum <= 2/sum， 同时要求丢弃的数为 `discardSum` ，不难得 pickedSum <= 2 / sum。

我们同时引入这两个约束条件加强剪枝：

优化后的代码如下

```
var canPartition = function (nums) {
  let sum = nums.reduce((acc, num) => acc + num, 0);
  if (sum % 2) {
    return false;
  }
  sum = sum / 2;
  nums = nums.sort((a, b) => b - a);
  if (sum < nums[0]) {
    return false;
  }
  return dfs(nums, sum, sum, 0);
};

function dfs(nums, pickRemain, discardRemain, cur) {
  if (pickRemain === 0 || discardRemain === 0) {
    return true;
  }

  if (pickRemain < 0 || discardRemain < 0 || cur > nums.len
    return false;
  }

  return (
    dfs(nums, pickRemain - nums[cur], discardRemain, cur +
    dfs(nums, pickRemain, discardRemain - nums[cur], cur +
  );
}
```

leetcode 是 AC 了，但是时间复杂度 O(2 ^ n), 算法时间复杂度很差，我们看看有没更好的。

DP 解法

在用 DFS 是时候，我们是不关心取数的规律的，只要保证接下来要取的数在之前没有被取过即可。那如果我们有规律去安排取数策略的时候会怎么样呢，比如第一次取数安排在第一位，第二位取数安排在第二位，在判断第 i 位是取数的时候，我们是已经知道前 i-1 个数每次是否取的所有子序列组合，记集合 S 为这个子序列的和。再看第 i 位取数的情况, 有两种情况取或者不取

1. 取的情况，如果 target-nums[i]在集合 S 内，则返回 true，说明前 i 个数能找到和为 target 的序列
2. 不取的情况，如果 target 在集合 S 内，则返回 true，否则返回 false

也就是说，前 i 个数能否构成和为 target 的子序列取决为前 i-1 数的情况。

记 F[i, target] 为 nums 数组内前 i 个数能否构成和为 target 的子序列的可能，则状态转移方程为

`F[i, target] = F[i - 1, target] || F[i - 1, target - nums[i]]`

状态转移方程出来了，代码就很好写了，DFS + DP 都可以解，有不清晰的可以参考下 递归和动态规划， 这里只提供 DP 解法

伪代码表示

```
n = nums.length
target 为 nums 各数之和
如果target不能被2整除,
  返回false

令dp为n * target 的二维矩阵, 并初始为false
遍历0:n, dp[i][0] = true 表示前i个数组成和为0的可能

遍历 0 到 n
  遍历 0 到 target
    if 当前值j大于nums[i]
      dp[i + 1][j] = dp[i][j-nums[i]] || dp[i][j]
    else
      dp[i+1][j] = dp[i][j]
```

算法时间复杂度 O(n*m), 空间复杂度 O(n*m), m 为 sum(nums) / 2

javascript 实现

```javascript
var canPartition = function (nums) {
  let sum = nums.reduce((acc, num) => acc + num, 0);
  if (sum % 2) {
    return false;
  } else {
    sum = sum / 2;
  }

  const dp = Array.from(nums).map(() =>
    Array.from({ length: sum + 1 }).fill(false)
  );

  for (let i = 0; i < nums.length; i++) {
    dp[i][0] = true;
  }

  for (let i = 0; i < dp.length - 1; i++) {
    for (let j = 0; j < dp[0].length; j++) {
      dp[i + 1][j] =
        j - nums[i] >= 0 ? dp[i][j] || dp[i][j - nums[i]] :
    }
  }

  return dp[nums.length - 1][sum];
};
```

再看看有没有优化空间，看状态转移方程 `F[i, target] = F[i - 1, target] || F[i - 1, target - nums[i]]` 第 n 行的状态只依赖于第 n-1 行的状态，也就是说我们可以把二维空间压缩成一维

伪代码

```
遍历 0 到 n
  遍历 j 从 target 到 0
    if 当前值j大于nums[i]
      dp[j] = dp[j-nums[i]] || dp[j]
    else
      dp[j] = dp[j]
```

时间复杂度 O(n*m), 空间复杂度 O(n) javascript 实现

```
var canPartition = function (nums) {
  let sum = nums.reduce((acc, num) => acc + num, 0);
  if (sum % 2) {
    return false;
  }
  sum = sum / 2;
  const dp = Array.from({ length: sum + 1 }).fill(false);
  dp[0] = true;

  for (let i = 0; i < nums.length; i++) {
    for (let j = sum; j > 0; j--) {
      dp[j] = dp[j] || (j - nums[i] >= 0 && dp[j - nums[i]]
    }
  }

  return dp[sum];
};
```

其实这道题和 leetcode 518 是换皮题，它们都可以归属于背包问题

背包问题

背包问题描述

有 N 件物品和一个容量为 V 的背包。放入第 i 件物品耗费的费用是 Ci，得到的 价值是 Wi。求解将哪些物品装入背包可使价值总和最大。

背包问题的特性是，每种物品，我们都可以选择放或者不放。令 F[i, v]表示前 i 件物品放入到容量为 v 的背包的状态。

针对上述背包，F[i, v]表示能得到最大价值，那么状态转移方程为

```
F[i, v] = max{F[i-1, v], F[i-1, v-Ci] + Wi}
```

针对 416. 分割等和子集这题，F[i, v]的状态含义就表示前 i 个数能组成和为 v 的可能，状态转移方程为

```
F[i, v] = F[i-1, v] || F[i-1, v-Ci]
```

再回过头来看下leetcode 518, 原题如下

> 给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

带入背包思想，F[i,v] 表示用前 i 种硬币能兑换金额数为 v 的组合数，状态转移方程为 `F[i, v] = F[i-1, v] + F[i-1, v-Ci]`

javascript 实现

```
/**
 * @param {number} amount
 * @param {number[]} coins
 * @return {number}
 */
var change = function (amount, coins) {
  const dp = Array.from({ length: amount + 1 }).fill(0);
  dp[0] = 1;
  for (let i = 0; i < coins.length; i++) {
    for (let j = 1; j <= amount; j++) {
      dp[j] = dp[j] + (j - coins[i] >= 0 ? dp[j - coins[i]]
    }
  }
  return dp[amount];
};
```

参考

背包九讲

题目地址

https://leetcode.com/problems/add-two-numbers-ii/description/

题目描述

```
You are given two non-empty linked lists representing two r

You may assume the two numbers do not contain any leading z

Follow up:
What if you cannot modify the input lists? In other words,

Example:

Input: (7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)
Output: 7 -> 8 -> 0 -> 7
```

前置知识

- 链表
- 栈

公司

- 腾讯
- 百度
- 字节

思路

由于需要从低位开始加，然后进位。 因此可以采用栈来简化操作。 依次将两个链表的值分别入栈 stack1 和 stack2，然后相加入栈 stack，进位操作用一个变量 carried 记录即可。

最后根据 stack 生成最终的链表即可。

> 也可以先将两个链表逆置，然后相加，最后将结果再次逆置。

关键点解析

- 栈的基本操作

- carried 变量记录进位
- 循环的终止条件设置成 `stack.length > 0` 可以简化操作
- 注意特殊情况， 比如 1 + 99 = 100

代码

- 语言支持：JS，C++, Python3

JavaScript Code:

```javascript
/*
 * @lc app=leetcode id=445 lang=javascript
 *
 * [445] Add Two Numbers II
 */
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} l1
 * @param {ListNode} l2
 * @return {ListNode}
 */
var addTwoNumbers = function(l1, l2) {
  const stack1 = [];
  const stack2 = [];
  const stack = [];

  let cur1 = l1;
  let cur2 = l2;
  let curried = 0;

  while (cur1) {
    stack1.push(cur1.val);
    cur1 = cur1.next;
  }

  while (cur2) {
    stack2.push(cur2.val);
    cur2 = cur2.next;
  }

  let a = null;
  let b = null;

  while (stack1.length > 0 || stack2.length > 0) {
    a = Number(stack1.pop()) || 0;
    b = Number(stack2.pop()) || 0;

    stack.push((a + b + curried) % 10);

    if (a + b + curried >= 10) {
      curried = 1;
    } else {
```

```
      curried = 0;
    }
  }

  if (curried === 1) {
    stack.push(1);
  }

  const dummy = {};

  let current = dummy;

  while (stack.length > 0) {
    current.next = {
      val: stack.pop(),
      next: null
    };

    current = current.next;
  }

  return dummy.next;
};
```

C++ Code：

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        auto carry = 0;
        auto ret = (ListNode*)nullptr;
        auto s1 = vector<int>();
        toStack(l1, s1);
        auto s2 = vector<int>();
        toStack(l2, s2);
        while (!s1.empty() || !s2.empty() || carry != 0) {
            auto v1 = 0;
            auto v2 = 0;
            if (!s1.empty()) {
                v1 = s1.back();
                s1.pop_back();
            }
            if (!s2.empty()) {
                v2 = s2.back();
                s2.pop_back();
            }
            auto v = v1 + v2 + carry;
            carry = v / 10;
            auto tmp = new ListNode(v % 10);
            tmp->next = ret;
            ret = tmp;
        }
        return ret;
    }
private:
    // 此处若返回而非传入vector，跑完所有测试用例多花8ms
    void toStack(const ListNode* l, vector<int>& ret) {
        while (l != nullptr) {
            ret.push_back(l->val);
            l = l->next;
        }
    }
};

// 逆置，相加，再逆置。跑完所有测试用例比第一种解法少花4ms
class Solution {
```

```
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        auto rl1 = reverseList(l1);
        auto rl2 = reverseList(l2);
        auto ret = add(rl1, rl2);
        return reverseList(ret);
    }
private:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = NULL;
        ListNode* cur = head;
        ListNode* next = NULL;
        while (cur != NULL) {
            next = cur->next;
            cur->next = prev;
            prev = cur;
            cur = next;
        }
        return prev;
    }

    ListNode* add(ListNode* l1, ListNode* l2) {
        ListNode* ret = nullptr;
        ListNode* cur = nullptr;
        int carry = 0;
        while (l1 != nullptr || l2 != nullptr || carry != 0
            carry += (l1 == nullptr ? 0 : l1->val) + (l2 ==
            auto temp = new ListNode(carry % 10);
            carry /= 10;
            if (ret == nullptr) {
                ret = temp;
                cur = ret;
            }
            else {
                cur->next = temp;
                cur = cur->next;
            }
            l1 = l1 == nullptr ? nullptr : l1->next;
            l2 = l2 == nullptr ? nullptr : l2->next;
        }
        return ret;
    }
};
```

Python Code:

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) ->
        def listToStack(l: ListNode) -> list:
            stack, c = [], l
            while c:
                stack.append(c.val)
                c = c.next
            return stack

        # transfer l1 and l2 into stacks
        stack1, stack2 = listToStack(l1), listToStack(l2)

        # add stack1 and stack2
        diff = abs(len(stack1) - len(stack2))
        stack1 = ([0]*diff + stack1 if len(stack1) < len(st
        stack2 = ([0]*diff + stack2 if len(stack2) < len(st
        stack3 = [x + y for x, y in zip(stack1, stack2)]

        # calculate carry for each item in stack3 and add c
        carry = 0
        for i, val in enumerate(stack3[::-1]):
            index = len(stack3) - i - 1
            carry, stack3[index] = divmod(val + carry, 10)
            if carry and index == 0:
                stack3 = [1] + stack3
            elif carry:
                stack3[index - 1] += 1

        # transfer stack3 to a linkedList
        result = ListNode(0)
        c = result
        for item in stack3:
            c.next = ListNode(item)
            c = c.next

        return result.next
```

题目地址

https://leetcode.com/problems/4sum-ii/description/

题目描述

```
Given four lists A, B, C, D of integer values, compute how

To make problem a bit easier, all A, B, C, D have same leng

Example:

Input:
A = [ 1, 2]
B = [-2,-1]
C = [-1, 2]
D = [ 0, 2]

Output:
2

Explanation:
The two tuples are:
1. (0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) +
2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) +
```

前置知识

- hashTable

公司

- 阿里
- 字节

思路

如果按照常规思路去完成查找需要四层遍历，时间复杂是O(n^4), 显然是行不通的。 因此我们有必要想一种更加高效的算法。

我一个思路就是我们将四个数组分成两组，两两结合。 然后我们分别计算 `两两结合能够算出的和有哪些，以及其对应的个数` 。

如图：

这个时候我们得到了两个 `hashTable` ， 我们只需要进行简单的数学运算就可以得到结果。

关键点解析

- 空间换时间
- 两两分组，求出两两结合能够得出的可能数，然后合并即可。

代码

语言支持： `JavaScript` ， `Python3`

`JavaScript` ：

```
/*
 * @lc app=leetcode id=454 lang=javascript
 *
 * [454] 4Sum II
 *
 * https://leetcode.com/problems/4sum-ii/description/
/**
 * @param {number[]} A
 * @param {number[]} B
 * @param {number[]} C
 * @param {number[]} D
 * @return {number}
 */
var fourSumCount = function(A, B, C, D) {
  const sumMapper = {};
  let res = 0;
  for (let i = 0; i < A.length; i++) {
    for (let j = 0; j < B.length; j++) {
        sumMapper[A[i] + B[j]] = (sumMapper[A[i] + B[j]] |
    }
  }

  for (let i = 0; i < C.length; i++) {
    for (let j = 0; j < D.length; j++) {
        res += sumMapper[- (C[i] + D[j])] || 0;
    }
  }

  return res;
};
```

`Python3` :

```
class Solution:
    def fourSumCount(self, A: List[int], B: List[int], C: L
        mapper = {}
        res = 0
        for i in A:
            for j in B:
                mapper[i + j] = mapper.get(i + j, 0) + 1

        for i in C:
            for j in D:
                res += mapper.get(-1 * (i + j), 0)
        return res
```

Problem

https://leetcode.com/problems/ones-and-zeroes/

Problem Description

```
In the computer world, use restricted resource you have to

For now, suppose you are a dominator of m 0s and n 1s respe

Now your task is to find the maximum number of strings that

Note:

The given numbers of 0s and 1s will both not exceed 100
The size of given string array won't exceed 600.

Example 1:

Input: Array = {"10", "0001", "111001", "1", "0"}, m = 5, n
Output: 4

Explanation: This are totally 4 strings can be formed by th

Example 2:

Input: Array = {"10", "0", "1"}, m = 1, n = 1
Output: 2

Explanation: You could form "10", but then you'd have nothi
```

Solution

When see the requirement of returning maximum number, length etc, and not require to list all possible value. Usually it can be solved by DP.

This problem we can see is a `0-1 backpack` issue, either take current string or not take current string.

And during interview, DP problem is hard to come up with immediately, recommend starting from Brute force, then optimize the solution step by step, until interviewer is happy, :-)

Below give 4 solutions based on complexities analysis.

Solution #1 - Brute Force (Recursively)

Brute force solution is to calculate all possible subsets. Then count `0s` and `1s` for each subset, use global variable `max` to keep track of maximum number. if `count(0) <= m && count(1) <= n;` , then current string can be taken into counts.

for `strs` length `len` , total subsets are `2^len` , Time Complexity is too high in this solution.

Complexity Analysis

- *Time Complexity:* `O(2^len * s) - len is Strs length, s is the average string length`
- *Space Complexity:* `O(1)`

Solution #2 - Memorization + Recursive

In Solution #1, brute force, we used recursive to calculate all subsets, in reality, many cases are duplicate, so that we can use memorization to record those subsets which realdy been calculated, avoid dup calculation. Use a memo array, if already calculated, then return memo value, otherwise, set the max value into memo.

`memo[i][j][k] - maximum number of strings can be formed by j 0s and k 1s in range [0, i] strings`

`helper(strs, i, j, k, memo)` recursively：

1. if `memo[i][j][k] != 0` , meaning already checked for j 0s and k 1s case, return value.
2. if not checked, then check condition `count0 <= j && count1 <= k` ， a. if true， take current strings `strs[i]` , so `0s -> j-count0` , and `1s -> k-count1` , check next string `helper(strs, i+1, j-count0, k-count1, memo)`
3. not take current string `strs[i]` , check next string `helper(strs, i+1, j, k, memo)`
4. save max value into `memo[i][j][k]`
5. recursively

Complexity Analysis

- *Time Complexity:* `O(l * m * n) - l is length of strs, m is number of 0s, n is number of 1s`
- *Space Complexity:* `O(l * m * n) - memo 3D Array`

Solution #3 - 3D DP

In Solution #2, used memorization + recursive, this Solution use 3D DP to represent iterative way

`dp[i][j][k] - the maximum number of strings can be formed using j 0s and k 1s in range [0, i] strings`

DP Formula： `dp[i][j][k] = max(dp[i][j][k], dp[i - 1][j - count0][k - count1])` - `count0 - number of 0s in strs[i]` and `count1 - number of 1s in strs[i]`

compare `j` and `count0` , `k` and `count1` , based on taking current string or not, DP formula as below：

- `j >= count0 && k >= count1` ,

  `dp[i][j][k] = max(dp[i - 1][j][k], dp[i - 1][j - count0][k - count1] + 1)`

- not meet condition, not take current string

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

`dp[l][m][n]` is result.

Complexity Analysis

- *Time Complexity:* `O(l * m * n) - l is strs length, m is number of 0s, n is number of 1s`
- *Space Complexity:* `O(l * m * n) - dp 3D array`

Solution #4 - 2D DP

In Solution #3, we kept track all state value, but we only need previous state, so we can reduce 3 dimention to 2 dimention array, here we use `dp[2][m][n]` , rotate track previous and current values. Further observation, we notice that first row (track previous state), we don't need the whole row values, we only care about 2 position value: `dp[i - 1][j][k]` and `dp[i - 1][j - count0][k - count1]` . so it can be reduced to 2D array. `dp[m][n]` .

2D DP definition:

`dp[m+1][n+1] - maximum counts, m is number of 0, n is number of 1`

DP formula：

`dp[i][j] = max(dp[i][j], dp[i - count0][j - count1] + 1)`

For example：

- *Time Complexity:* `O(l * m * n) - l is strs length, m is number of 0, n number of 1`
- *Space Complexity:* `O(m * n) - dp 2D array`

Key Points Analysis

Code (`Java/Python3`)

Solution #1 - Recursive (TLE)

Java code

```
class OnesAndZerosBFRecursive {
 public int findMaxForm2(String[] strs, int m, int n) {
    return helper(strs, 0, m, n);
 }
 private int helper(String[] strs, int idx, int j, int k) {
    if (idx == strs.length) return 0;
    // count current idx string number of zeros and ones
    int[] counts = countZeroOnes(strs[idx]);
    // if j >= count0 && k >= count1, take current index st
    int takeCurrStr = j - counts[0] >= 0 && k - counts[1] >
        ? 1 + helper(strs, idx + 1, j - counts[0], k - cour
        : -1;
    // don't take current index string strs[idx], continue
    int notTakeCurrStr = helper(strs, idx + 1, j, k);
    return Math.max(takeCurrStr, notTakeCurrStr);
 }
 private int[] countZeroOnes(String s) {
    int[] res = new int[2];
    for (char ch : s.toCharArray()) {
      res[ch - '0']++;
    }
    return res;
 }
}
```

Python3 code

```
class Solution:
    def findMaxForm(self, strs: List[str], m: int, n: int)
        return self.helper(strs, m, n, 0)

    def helper(self, strs, m, n, idx):
        if idx == len(strs):
            return 0
        take_curr_str = -1
        count0, count1 = strs[idx].count('0'), strs[idx].co
        if m >= count0 and n >= count1:
            take_curr_str = max(take_curr_str, self.helper(
        not_take_curr_str = self.helper(strs, m, n, idx + 1
        return max(take_curr_str, not_take_curr_str)
```

Solution #2 - Memorization + Recursive

Java code

```
class OnesAndZerosMemoRecur {
  public int findMaxForm4(String[] strs, int m, int n) {
      return helper(strs, 0, m, n, new int[strs.length][m -
  }
  private int helper(String[] strs, int idx, int j, int k,
      if (idx == strs.length) return 0;
      // check if already calculated, return value
      if (memo[idx][j][k] != 0) {
        return memo[idx][j][k];
      }
      int[] counts = countZeroOnes(strs[idx]);
      // if satisfy condition, take current string, strs[ic
      int takeCurrStr = j - counts[0] >= 0 && k - counts[1]
          ? 1 + helper(strs, idx + 1, j - counts[0], k - co
          : -1;
      // not take current string
      int notTakeCurrStr = helper(strs, idx + 1, j, k, memo
      // always keep track the max value into memory
      memo[idx][j][k] = Math.max(takeCurrStr, notTakeCurrSt
      return memo[idx][j][k];
  }
  private int[] countZeroOnes(String s) {
       int[] res = new int[2];
       for (char ch : s.toCharArray()) {
         res[ch - '0']++;
       }
       return res;
  }
}
```

Python3 code - (TLE)

```
class Solution:
    def findMaxForm(self, strs: List[str], m: int, n: int)
        memo = {k:[[0]*(n+1) for _ in range(m+1)] for k in
        return self.helper(strs, 0, m, n, memo)

    def helper(self, strs, idx, m, n, memo):
        if idx == len(strs):
            return 0
        if memo[idx][m][n] != 0:
            return memo[idx][m][n]
        take_curr_str = -1
        count0, count1 = strs[idx].count('0'), strs[idx].co
        if m >= count0 and n >= count1:
            take_curr_str = max(take_curr_str, self.helper(
        not_take_curr_str = self.helper(strs, idx + 1, m, n
        memo[idx][m][n] = max(take_curr_str, not_take_curr_
        return memo[idx][m][n]
```

Solution #3 - 3D DP

Java code

```
class OnesAndZeros3DDP {
  public int findMaxForm(String[] strs, int m, int n) {
      int l = strs.length;
      int [][][] d = new int[l + 1][m + 1][n + 1];
      for (int i = 0; i <= l; i ++){
        int [] nums = new int[]{0,0};
        if (i > 0){
          nums = countZeroOnes(strs[i - 1]);
        }
        for (int j = 0; j <= m; j ++){
          for (int k = 0; k <= n; k ++){
            if (i == 0) {
              d[i][j][k] = 0;
            } else if (j >= nums[0] && k >= nums[1]){
              d[i][j][k] = Math.max(d[i - 1][j][k], d[i - 1
            } else {
              d[i][j][k] = d[i - 1][j][k];
            }
          }
        }
      }
      return d[l][m][n];
  }
}
```

Solution #4 - 2D DP

Java code

```
class OnesAndZeros2DDP {
  public int findMaxForm(String[] strs, int m, int n) {
      int[][] dp = new int[m + 1][n + 1];
      for (String s : strs) {
        int[] counts = countZeroOnes(s);
        for (int i = m; i >= counts[0]; i--) {
          for (int j = n; j >= counts[1]; j--) {
            dp[i][j] = Math.max(1 + dp[i - counts[0]][j - c
          }
        }
      }
      return dp[m][n];
  }
  private int[] countZeroOnes(String s) {
       int[] res = new int[2];
       for (char ch : s.toCharArray()) {
         res[ch - '0']++;
       }
       return res;
  }
}
```

Python3 code

```
class Solution:
    def findMaxForm(self, strs: List[str], m: int, n: int)
        l = len(strs)
        dp = [[0]*(n+1) for _ in range(m+1)]
        for i in range(1, l + 1):
              count0, count1 = strs[i - 1].count('0'), strs
              for i in reversed(range(count0, m + 1)):
                for j in reversed(range(count1, n + 1)):
                    dp[i][j] = max(dp[i][j], 1 + dp[i - cou
        return dp[m][n]
```

Similar problems

- Leetcode 600. Non-negative Integers without Consecutive Ones
- Leetcode 322. Coin Change

题目地址

https://leetcode.com/problems/target-sum/description/

题目描述

```
You are given a list of non-negative integers, a1, a2, ...,

Find out how many ways to assign symbols to make sum of int

Example 1:
Input: nums is [1, 1, 1, 1, 1], S is 3.
Output: 5
Explanation:

-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

There are 5 ways to assign symbols to make the sum of nums
Note:
The length of the given array is positive and will not exce
The sum of elements in the given array will not exceed 1000
Your output answer is guaranteed to be fitted in a 32-bit i
```

前置知识

- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

题目是给定一个数组，让你在数字前面添加 `+` 或者 `-` ，使其和等于 target.

暴力法的时间复杂度是指数级别的，因此我们不予考虑。我们需要换种思路.

我们将最终的结果分成两组，一组是我们添加了 `+` 的，一组是我们添加了 `-` 的。

如上图，问题转化为如何求其中一组，我们不妨求前面标 `+` 的一组

> 如果求出一组，另一组实际就已知了，即总集和这一组的差集。

通过进一步分析，我们得到了这样的关系：

因此问题转化为，求解 `sumCount(nums, target)` ,即 nums 数组中能够组成 target 的总数一共有多少种，这是一道我们之前做过的题目，使用动态规划可以解决。

关键点解析

- 对元素进行分组，分组的依据是符号， 是 `+` 或者 `-`
- 通过数学公式推导可以简化我们的求解过程，这需要一点 `数学知识和数学意识`

代码

```javascript
/*
 * @lc app=leetcode id=494 lang=javascript
 *
 * [494] Target Sum
 *
 */
// 这个是我们熟悉的问题了
// 我们这里需要求解的是nums里面有多少种可以组成target的方式
var sumCount = function(nums, target) {
  // 这里通过观察，我们没必要使用二维数组去存储这些计算结果
  // 使用一维数组可以有效节省空间
  const dp = Array(target + 1).fill(0);
  dp[0] = 1;
  for (let i = 0; i < nums.length; i++) {
    for (let j = target; j >= nums[i]; j--) {
      dp[j] += dp[j - nums[i]];
    }
  }
  return dp[target];
};
const add = nums => nums.reduce((a, b) => (a += b), 0);
/**
 * @param {number[]} nums
 * @param {number} S
 * @return {number}
 */
var findTargetSumWays = function(nums, S) {
  const sum = add(nums);
  if (sum < S) return 0;
  if ((S + sum) % 2 === 1) return 0;
  return sumCount(nums, (S + sum) >> 1);
};
```

扩展

如果这道题目并没有限定所有的元素以及 target 都是正数怎么办?

题目地址

https://leetcode.com/problems/longest-palindromic-subsequence/description/

题目描述

```
Given a string s, find the longest palindromic subsequence

Example 1:
Input:

"bbbab"
Output:
4
One possible longest palindromic subsequence is "bbbb".
Example 2:
Input:

"cbbd"
Output:
2
One possible longest palindromic subsequence is "bb".
```

前置知识

- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这是一道最长回文的题目，要我们求出给定字符串的最大回文子序列。

解决这类问题的核心思想就是两个字“延伸”，具体来说

- 如果一个字符串是回文串，那么在它左右分别加上一个相同的字符，那么它一定还是一个回文串，因此 `回文长度增加2`
- 如果一个字符串不是回文串，或者在回文串左右分别加不同的字符，得到的一定不是回文串,因此 `回文长度不变，我们取[i][j-1]和[i+1][j]的较大值`

事实上，上面的分析已经建立了大问题和小问题之间的关联， 基于此，我们可以建立动态规划模型。

我们可以用 dp[i][j] 表示 s 中从 i 到 j（包括 i 和 j）的回文序列长度， 状态转移方程只是将上面的描述转化为代码即可：

```
if (s[i] === s[j]) {
  dp[i][j] = dp[i + 1][j - 1] + 2;
} else {
  dp[i][j] = Math.max(dp[i][j - 1], dp[i + 1][j]);
}
```

base case 就是一个字符（轴对称点是本身）

关键点

- ”延伸“（extend）

代码

```
/*
 * @lc app=leetcode id=516 lang=javascript
 *
 * [516] Longest Palindromic Subsequence
 */
/**
 * @param {string} s
 * @return {number}
 */
var longestPalindromeSubseq = function(s) {
  // bbbab 返回4
  // tag : dp
  const dp = [];

  for (let i = s.length - 1; i >= 0; i--) {
    dp[i] = Array(s.length).fill(0);
    for (let j = i; j < s.length; j++) {
      if (i - j === 0) dp[i][j] = 1;
      else if (s[i] === s[j]) {
        dp[i][j] = dp[i + 1][j - 1] + 2;
      } else {
        dp[i][j] = Math.max(dp[i][j - 1], dp[i + 1][j]);
      }
    }
  }

  return dp[0][s.length - 1];
};
```

相关题目

- 5.longest-palindromic-substring

题目地址（518. 零钱兑换 II）

https://leetcode-cn.com/problems/coin-change-2/description/

题目描述

给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

示例 1:

输入: amount = 5, coins = [1, 2, 5] 输出: 4 解释: 有四种方式可以凑成总金额: 5=5 5=2+2+1 5=2+1+1+1 5=1+1+1+1+1 示例 2:

输入: amount = 3, coins = [2] 输出: 0 解释: 只用面额 2 的硬币不能凑成总金额 3。 示例 3:

输入: amount = 10, coins = [10] 输出: 1

注意:

你可以假设：

0 <= amount (总金额) <= 5000 1 <= coin (硬币面额) <= 5000 硬币种类不超过 500 种 结果符合 32 位符号整数

前置知识

- 动态规划
- 背包问题

公司

- 阿里
- 百度
- 字节

思路

这个题目和 coin-change 的思路比较类似。

我们还是按照 coin-change 的思路来， 如果将问题画出来大概是这样：

进一步我们可以对问题进行空间复杂度上的优化（这种写法比较难以理解，但是相对更省空间）

> 这里用动图会更好理解， 有时间的话我会做一个动图出来， 现在大家脑补一下吧

关键点解析

- 动态规划
- 子问题

用 dp[i] 来表示组成 i 块钱，需要最少的硬币数，那么

1. 第 j 个硬币我可以选择不拿 这个时候， 组成数 = dp[i]
2. 第 j 个硬币我可以选择拿 这个时候， 组成数 = dp[i - coins[j]] + dp[i]

3. 和 01 背包问题不同， 硬币是可以拿任意个，属于完全背包问题

4. 对于每一个 dp[i] 我们都选择遍历一遍 coin， 不断更新 dp[i]

eg:

```
if (amount === 0) return 1;

const dp = [Array(amount + 1).fill(1)];

for (let i = 1; i < amount + 1; i++) {
  dp[i] = Array(coins.length + 1).fill(0);
  for (let j = 1; j < coins.length + 1; j++) {
    // 从1开始可以简化运算
    if (i - coins[j - 1] >= 0) {
      // 注意这里是coins[j -1]而不是coins[j]
      dp[i][j] = dp[i][j - 1] + dp[i - coins[j - 1]][j]; //
    } else {
      dp[i][j] = dp[i][j - 1];
    }
  }
}

return dp[dp.length - 1][coins.length];
```

- 当我们选择一维数组去解的时候，内外循环将会对结果造成影响

eg:

```
// 这种答案是不对的。
// 原因在于比如amount = 5, coins = [1,2,5]
// 这种算法会将[1,2,2] [2,1,2] [2, 2, 1] 算成不同的

if (amount === 0) return 1;

const dp = [1].concat(Array(amount).fill(0));

for (let i = 1; i < amount + 1; i++) {
  for (let j = 0; j < coins.length; j++) {
    if (i - coins[j] >= 0) {
      dp[i] = dp[i] + dp[i - coins[j]];
    }
  }
}

return dp[dp.length - 1];

// 正确的写法应该是内外循环调换一下, 具体可以看下方代码区
```

代码

代码支持：Python3，JavaScript：

JavaSCript Code:

```javascript
/*
 * @lc app=leetcode id=518 lang=javascript
 *
 * [518] Coin Change 2
 *
 */
/**
 * @param {number} amount
 * @param {number[]} coins
 * @return {number}
 */
var change = function (amount, coins) {
  if (amount === 0) return 1;

  const dp = [1].concat(Array(amount).fill(0));

  for (let j = 0; j < coins.length; j++) {
    for (let i = 1; i < amount + 1; i++) {
      if (i - coins[j] >= 0) {
        dp[i] = dp[i] + dp[i - coins[j]];
      }
    }
  }

  return dp[dp.length - 1];
};
```

Python Code:

```python
class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        dp = [0] * (amount + 1)
        dp[0] = 1

        for j in range(len(coins)):
            for i in range(1, amount + 1):
                if i >= coins[j]:
                    dp[i] += dp[i - coins[j]]

        return dp[-1]
```

扩展

这是一道很简单描述的题目， 因此很多时候会被用到大公司的电面中。

相似问题:

322.coin-change

附录

Python 二维解法（不推荐，但是可以帮助理解）：

```
class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        dp = [[0 for _ in range(len(coins) + 1)] for _ in r
        for j in range(len(coins) + 1):
            dp[0][j] = 1

        for i in range(amount + 1):
            for j in range(1, len(coins) + 1):
                if i >= coins[j - 1]:
                    dp[i][j] = dp[i - coins[j - 1]][j] + dp
                else:
                    dp[i][j] = dp[i][j - 1]
        return dp[-1][-1]
```

Problem

https://leetcode.com/problems/friend-circles/

Problem Description

```
There are N students in a class. Some of them are friends,
For example, if A is a direct friend of B, and B is a direc
And we defined a friend circle is a group of students who a

Given a N*N matrix M representing the friend relationship b
If M[i][j] = 1, then the ith and jth students are direct fr
And you have to output the total number of friend circles a

Example 1:

Input:
[[1,1,0],
 [1,1,0],
 [0,0,1]]
Output: 2
Explanation:The 0th and 1st students are direct friends, so
The 2nd student himself is in a friend circle. So return 2.
Example 2:

Input:
[[1,1,0],
 [1,1,1],
 [0,1,1]]
Output: 1
Explanation:The 0th and 1st students are direct friends, th
so the 0th and 2nd students are indirect friends. All of th
Note:

N is in range [1,200].
M[i][i] = 1 for all students.
If M[i][j] = 1, then M[j][i] = 1.
```

Solution

We can view a given matrix as Adjacency Matrix of a graph. In this case, this problem become to find number of connected components in a undirected graph.

For example, how to transfer Adjacency Matrix into a graph problem. As below pic:

Connected components in a graph problem usually can be solved using *DFS*, *BFS*, *Union-Find*.

Below we will explain details on each approach.

Approach #1. DFS

1. Do DFS starting from every node, use `visited` array to mark visited node.
2. For each node DFS, visit all its directly connected nodes.
3. For each node DFS, DFS search will search all connected nodes, thus we count one DFS as one connected component.

as below pic show *DFS* traverse process:

Complexity Analysis

- *Time Complexity:* `O(n*n) - n is the number of students, traverse n*n matrix`
- *Space Complexity:* `O(n) - visited array of size n`

Approach #2. BFS (Level traverse)

1. Start from one node, visit all its directly connected nodes, or visit all nodes in the same level.
2. Use `visited` array to mark already visited nodes.
3. Increment count when start with a new node.

as below pic show *BFS* (Level traverse) traverse process:

Complexity Analysis

- *Time Complexity:* `O(n*n) - n is the number of students, traverse n*n matrix`
- *Space Complexity:* `O(n) - queue and visited array of size n`

Approach #3. Union-Find

Determine number of connected components, Union Find is good algorithm to use.

Use `parent` array, for every node, all its directly connected nodes will be `union`, so that two connected nodes have the same parent. After traversal all nodes, we just need to calculate the number of different values in `parent` array.

For each union, total count minus 1, after traversal and union all connected nodes, simply return counts.

Here use **weighted-union-find** to reduce `union` and `find` methods operation time.

To know more details and implementations, see further reading lists.

as below Union-Find approach process:

> **Note:** here using weighted-union-find to avoid Union and Find take `O(n)` in the worst case.

Complexity Analysis

- *Time Complexity:* `O(n*n*log(n) - traverse n*n matrix, weighted-union-find, union and find takes log(n) time complexity.`
- *Space Complexity:* `O(n) - parent and rank array of size n`

Key Points

1. Transform Adjacency matrix into Graph
2. Notice that it actually is to find number of connected components problem.
3. Connected components problem approaches (DFS, BFS, Union-Find).

Code (`Java`)

Java code - **DFS**

```
class FindCirclesDFS {
  public int findCircleNumDFS(int[][] M) {
    if (M == null || M.length == 0 || M[0].length == 0) ret
    int n = M.length;
    int numCircles = 0;
    boolean[] visited = new boolean[n];
    for (int i = 0; i < n; i++) {
      if (!visited[i]) {
        dfs(M, i, visited, n);
        numCircles++;
      }
    }
    return numCircles;
  }

  private void dfs(int[][] M, int i, boolean[] visited, int
    for (int j = 0; j < n; j++) {
      if (M[i][j] == 1 && !visited[j]) {
        visited[j] = true;
        dfs(M, j, visited, n);
      }
    }
  }
}
```

Java code - **BFS**

```java
class FindCircleBFS {
  public int findCircleNumBFS(int[][] M) {
    if (M == null || M.length == 0) return 0;
    int numCircle = 0;
    int n = M.length;
    boolean[] visited = new boolean[n];
    Queue<Integer> queue = new LinkedList<>();
    for (int i = 0; i < n; i++) {
      // already visited, skip
      if (visited[i]) continue;
      queue.add(i);
      while (!queue.isEmpty()) {
        int curr = queue.poll();
        visited[curr] = true;
        for (int j = 0; j < n; j++) {
          if (M[curr][j] == 1 && !visited[j]) {
            queue.add(j);
          }
        }
      }
      numCircle++;
    }
    return numCircle;
  }
}
```

Java code - **Union-Find**

```
class FindCircleUF {
 public int findCircleNumUF(int[][] M) {
    if (M == null || M.length == 0 || M[0].length == 0) ret
    int n = M.length;
    UnionFind uf = new UnionFind(n);
    for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
        // union friends
        if (M[i][j] == 1) {
          uf.union(i, j);
        }
      }
    }
    return uf.count;
  }
}

class UnionFind {
  int count;
  int[] parent;
  int[] rank;

  public UnionFind(int n) {
    count = n;
    parent = new int[n];
    rank = new int[n];
    for (int i = 0; i < n; i++) {
      parent[i] = i;
    }
  }

  public int find(int a) {
    return parent[a] == a ? a : find(parent[a]);
  }

  public void union(int a, int b) {
    int rootA = find(a);
    int rootB = find(b);
    if (rootA == rootB) return;
    if (rank[rootA] <= rank[rootB]) {
      parent[rootA] = rootB;
      rank[rootB] += rank[rootA];
    } else {
      parent[rootB] = rootA;
      rank[rootA] += rank[rootB];
    }
    count--;
  }
```

```
  public int count() {
    return count;
  }
}
```

References (Further reading)

1. Adjacency Matrix Wiki
2. Union-Find Wiki
3. Algorighms 4 union-find

Similar Problems

1. 323. Number of Connected Components in an Undirected Graph
2. 1101. The Earliest Moment When Everyone Become Friends

题目地址

https://leetcode.com/problems/subarray-sum-equals-k/description/

题目描述

```
Given an array of integers and an integer k, you need to f

Example 1:
Input:nums = [1,1,1], k = 2
Output: 2
Note:
The length of the array is in range [1, 20,000].
The range of numbers in the array is [-1000, 1000] and the
```

前置知识

- 哈希表
- 前缀和

公司

- 阿里
- 腾讯
- 字节

思路

符合直觉的做法是暴力求解所有的子数组，然后分别计算和，如果等于k,count 就+1.这种做法的时间复杂度为 O(n^2)，代码如下：

```
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        cnt, n =  0, len(nums)
        for i in range(n):
            sum = 0
            for j in range(i, n):
                sum += nums[j]
                if (sum == k): cnt += 1
        return cnt
```

实际上刚开始看到这题目的时候，我想“是否可以用滑动窗口解决？”。但是很快我就放弃了，因为看了下数组中项的取值范围有负数，这样我们扩张或者收缩窗口就比较复杂。第二个想法是前缀和，保存一个数组的前缀和，然后利用差分法得出任意区间段的和，这种想法是可行的，代码如下：

```
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
      cnt, n =  0, len(nums)
      pre = [0] * (n + 1)
      for i in range(1, n + 1):
          pre[i] = pre[i - 1] + nums[i - 1]
      for i in range(1, n + 1):
          for j in range(i, n + 1):
              if (pre[j] - pre[i - 1] == k): cnt += 1
      return cnt
```

这里有一种更加巧妙的方法，可以不使用前缀和数组，而是使用 hashmap 来简化时间复杂度，这种算法的时间复杂度可以达到 O(n).

具体算法：

- 维护一个 hashmap，hashmap 的 key 为累加值 acc，value 为累加值 acc 出现的次数。
- 迭代数组，然后不断更新 acc 和 hashmap，如果 acc 等于 k，那么很明显应该+1. 如果 hashmap[acc - k] 存在，我们就把它加到结果中去即可。

语言比较难以解释，我画了一个图来演示 nums = [1,2,3,3,0,3,4,2], k = 6 的情况。

如图，当访问到 nums[3]的时候，hashmap 如图所示，这个时候 count 为 2. 其中之一是[1,2,3],这个好理解。还有一个是[3,3].

这个[3,3]正是我们通过 hashmap[acc - k]即 hashmap[9 - 6]得到的。

关键点解析

- 前缀和
- 可以利用 hashmap 记录和的累加值来避免重复计算

代码

- 语言支持：JS, Python

Javascript Code:

```javascript
/*
 * @lc app=leetcode id=560 lang=javascript
 *
 * [560] Subarray Sum Equals K
 */
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number}
 */
var subarraySum = function (nums, k) {
  const hashmap = {};
  let acc = 0;
  let count = 0;

  for (let i = 0; i < nums.length; i++) {
    acc += nums[i];

    if (acc === k) count++;

    if (hashmap[acc - k] !== void 0) {
      count += hashmap[acc - k];
    }

    if (hashmap[acc] === void 0) {
      hashmap[acc] = 1;
    } else {
      hashmap[acc] += 1;
    }
  }

  return count;
};
```

Python Code:

```python
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        d = {}
        acc = count = 0
        for num in nums:
            acc += num
            if acc == k:
                count += 1
            if acc - k in d:
                count += d[acc-k]
            if acc in d:
                d[acc] += 1
            else:
                d[acc] = 1
        return count
```

扩展

这是一道类似的题目，但是会稍微复杂一点, 题目地址: 437.path-sum-iii

题目地址

https://leetcode.com/problems/find-duplicate-file-in-system/description/

题目描述

```
Given a list of directory info including directory path, ar

A group of duplicate files consists of at least two files t

A single directory info string in the input list has the fo

"root/d1/d2/.../dm f1.txt(f1_content) f2.txt(f2_content) ..

It means there are n files (f1.txt, f2.txt ... fn.txt with

The output is a list of group of duplicate file paths. For

"directory_path/file_name.txt"

Example 1:

Input:
["root/a 1.txt(abcd) 2.txt(efgh)", "root/c 3.txt(abcd)", "r
Output:
[["root/a/2.txt","root/c/d/4.txt","root/4.txt"],["root/a/1.


Note:

No order is required for the final output.
You may assume the directory name, file name and file conte
The number of files given is in the range of [1,20000].
You may assume no files or directories share the same name
You may assume each given directory info represents a uniqu


Follow-up beyond contest:

1. Imagine you are given a real file system, how will you s

2. If the file content is very large (GB level), how will y

3. If you can only read the file by 1kb each time, how will

4. What is the time complexity of your modified solution? W

5. How to make sure the duplicated files you find are not t
```

前置知识

- 哈希表

思路

思路就是hashtable去存储，key为文件内容，value为fullfilename， 遍历一遍去填充hashtable， 最后将hashtable中的值打印出来即可。

> 当且仅当有重复内容，我们才打印，因此我们需要过滤一下， 类似 filter(q => q.length >= 2)
>
> ## 关键点解析

- hashtable

代码

```
/*
 * @lc app=leetcode id=609 lang=javascript
 *
 * [609] Find Duplicate File in System
 *
 * https://leetcode.com/problems/find-duplicate-file-in-sys
 *
 * algorithms
 * Medium (54.21%)
 * Total Accepted:    24.1K
 * Total Submissions: 44.2K
 * Testcase Example:  '["root/a 1.txt(abcd) 2.txt(efgh)","r
 *
 * Given a list of directory info including directory path,
 * with contents in this directory, you need to find out al
 * duplicate files in the file system in terms of their pat
 *
 * A group of duplicate files consists of at least two file
 * the same content.
 *
 * A single directory info string in the input list has the
 *
 * "root/d1/d2/.../dm f1.txt(f1_content) f2.txt(f2_content)
 * fn.txt(fn_content)"
 *
 * It means there are n files (f1.txt, f2.txt ... fn.txt wi
 * f1_content, f2_content ... fn_content, respectively) in
 * root/d1/d2/.../dm. Note that n >= 1 and m >= 0. If m = 0
 * directory is just the root directory.
 *
 * The output is a list of group of duplicate file paths. E
 * contains all the file paths of the files that have the s
 * path is a string that has the following format:
 *
 * "directory_path/file_name.txt"
 *
 * Example 1:
 *
 *
 * Input:
 * ["root/a 1.txt(abcd) 2.txt(efgh)", "root/c 3.txt(abcd)",
 * 4.txt(efgh)", "root 4.txt(efgh)"]
 * Output:
 *
 * [["root/a/2.txt","root/c/d/4.txt","root/4.txt"],["root/a
 *
```

```
 *
 *
 *
 * Note:
 *
 *
 * No order is required for the final output.
 * You may assume the directory name, file name and file co
 * letters and digits, and the length of file content is in
 * [1,50].
 * The number of files given is in the range of [1,20000].
 * You may assume no files or directories share the same na
 * directory.
 * You may assume each given directory info represents a un
 * Directory path and file info are separated by a single b
 *
 *
 *
 * Follow-up beyond contest:
 *
 *
 * Imagine you are given a real file system, how will you s
 * BFS?
 * If the file content is very large (GB level), how will y
 * solution?
 * If you can only read the file by 1kb each time, how will
 * solution?
 * What is the time complexity of your modified solution? W
 * time-consuming part and memory consuming part of it? How
 * How to make sure the duplicated files you find are not t
 *
 *
 */
/**
 * @param {string[]} paths
 * @return {string[][]}
 */
var findDuplicate = function(paths) {
  const hashmap = {};

  for (let path of paths) {
    const [folder, ...files] = path.split(" ");
    for (let file of files) {
      const lpi = file.indexOf("(");
      const rpi = file.lastIndexOf(")");
      const filename = file.slice(0, lpi);
      const content = file.slice(lpi, rpi);
      const fullname = `{% math %}{folder}/{% endmath %}{fi
```

```
      if (!hashmap[content]) hashmap[content] = [];
      hashmap[content].push(fullname);
    }
  }

  return Object.values(hashmap).filter(q => q.length >= 2);
};
```

扩展

leetcode官方给的扩展我觉得就很有意思，虽然很 `老套` ，这里还是列一下好了，大家可以作为思考题来思考一下。

1. Imagine you are given a real file system, how will you search files? DFS or BFS?
2. If the file content is very large (GB level), how will you modify your solution?
3. If you can only read the file by 1kb each time, how will you modify your solution?
4. What is the time complexity of your modified solution? What is the most time-consuming part and memory consuming part of it? How to optimize?
5. How to make sure the duplicated files you find are not false positive?

题目地址(611. 有效三角形的个数)

https://leetcode-cn.com/problems/valid-triangle-number/

题目描述

```
给定一个包含非负整数的数组，你的任务是统计其中可以组成三角形三条边的三

示例 1:

输入: [2,2,3,4]
输出: 3
解释:
有效的组合是:
2,3,4 (使用第一个 2)
2,3,4 (使用第二个 2)
2,2,3
注意:

数组长度不超过1000。
数组里整数的范围为 [0, 1000]。
```

前置知识

- 排序
- 双指针
- 二分法
- 三角形边的关系

暴力法（超时）

公司

- 腾讯
- 百度
- 字节

思路

首先要有一个数学前提： `如果三条线段中任意两条的和都大于第三边，那么这三条线段可以组成一个三角形` 。即给定三个线段 a，b，c，如果满足 a + b > c and a + c > b and b + c > a，则线段 a，b，c 可以构成三角形，否则不可以。

力扣中有一些题目是需要一些数学前提的，不过这些数学前提都比较简单，一般不会超过高中数学知识，并且也不会特别复杂。一般都是小学初中知识即可。

> 如果你在面试中碰到不知道的数学前提，可以寻求面试官提示试试。

关键点解析

- 三角形边的关系
- 三层循环确定三个线段

代码

代码支持: Python

```
class Solution:
    def is_triangle(self, a, b, c):
        if a == 0 or b == 0 or c == 0: return False
        if a + b > c and a + c > b and b + c > a: return Tr
        return False
    def triangleNumber(self, nums: List[int]) -> int:
        n = len(nums)
        ans = 0
        for i in range(n - 2):
            for j in range(i + 1, n - 1):
                for k in range(j + 1, n):
                    if self.is_triangle(nums[i], nums[j], n

        return ans
```

复杂度分析

- 时间复杂度：$O(N^3)$，其中 N 为 数组长度。
- 空间复杂度：$O(1)$

优化的暴力法

思路

暴力法的时间复杂度为 $O(N^3)$， 其中 N 最大为 1000。一般来说， $O(N^3$ 的算法在数据量 <= 500 是可以 AC 的。1000 的数量级则需要考虑 $O(N^2)$ 或者更好的解法。

OK，到这里了。我给大家一个干货。 应该是其他博主不太会提的。原因可能是他们不知道， 也可能是他们觉得太小儿科不需要说。

1. 由于前面我根据数据规模推测到到了解法的复杂度区间是 N^2，$N^2 * logN$，不可能是 N （WHY？）。
2. 降低时间复杂度的方法主要有： `空间换时间` 和 `排序换时间` （我们一般都是使用基于比较的排序方法）。而 `排序换时间` 仅仅在总体复杂度大于 $O(NlogN)$ 才适用（原因不用多说了吧？）。

这里由于总体的时间复杂度是 $O(N^3)$，因此我自然想到了 `排序换时间` 。当我们对 nums 进行一次排序之后，我发现：

- is_triangle 函数有一些判断是无效的

```
    def is_triangle(self, a, b, c):
        if a == 0 or b == 0 or c == 0: return False
        # a + c > b 和  b + c > a 是无效的判断，因为恒成立
        if a + b > c and a + c > b and b + c > a: return Tr
        return False
```

- 因此我们的目标变为找到 `a + b > c` 即可，因此第三层循环是可以提前退出的。

```
for i in range(n - 2):
    for j in range(i + 1, n - 1):
        k = j + 1
        while k < n and num[i] + nums[j] > nums[k]:
            k += 1
        ans += k - j - 1
```

- 这也仅仅是减枝而已，复杂度没有变化。通过进一步观察，发现 k 没有必要每次都从 j + 1 开始。而是从上次找到的 k 值开始就行。原因很简单， 当 nums[i] + nums[j] > nums[k] 时，我们想要找到下一个满足 nums[i] + nums[j] > nums[k] 的 新的 k 值，由于进行了排序，因此这个 k 肯定比之前的大（单调递增性），因此上一个 k 值之前的数都是无效的，可以跳过。

```
for i in range(n - 2):
    k = i + 2
    for j in range(i + 1, n - 1):
        while k < n and nums[i] + nums[j] > nums[k]:
            k += 1
        ans += k - j - 1
```

由于 K 不会后退，因此最内层循环总共最多执行 N 次，因此总的时间复杂度为 $O(N^2)$。

> 这个复杂度分析有点像单调栈，大家可以结合起来理解。

关键点分析

- 排序

代码

```
class Solution:
    def triangleNumber(self, nums: List[int]) -> int:
        n = len(nums)
        ans = 0
        nums.sort()
        for i in range(n - 2):
            if nums[i] == 0: continue
            k = i + 2
            for j in range(i + 1, n - 1):
                while k < n and nums[i] + nums[j] > nums[k]
                    k += 1
                ans += k - j - 1
        return ans
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：取决于排序算法

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址（718. 最长重复子数组）

https://leetcode-cn.com/problems/maximum-length-of-repeated-subarray/

题目描述

```
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

示例 1:

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]。
说明:

1 <= len(A), len(B) <= 1000
0 <= A[i], B[i] < 100
```

前置知识

- 哈希表
- 数组
- 二分查找
- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这就是最经典的最长公共子序列问题。一般这种求解**两个数组或者字符串求最大或者最小**的题目都可以考虑动态规划，并且通常都定义 dp[i][j] 为 `以 A[i], B[j] 结尾的 xxx` 。这道题就是： `以 A[i], B[j] 结尾的两`

个数组中公共的、长度最长的子数组的长度 。 算法很简单：

- 双层循环找出所有的 i, j 组合，时间复杂度 $O(m*n)$，其中 m 和 n 分别为 A 和 B 的 长度。
  - 如果 A[i] == B[j]，dp[i][j] = dp[i - 1][j - 1] + 1
  - 否则，dp[i][j] = 0
- 循环过程记录最大值即可。

关键点解析

- dp 建模套路

代码

代码支持：Python

Python Code:

```
class Solution:
    def findLength(self, A, B):
        m, n = len(A), len(B)
        ans = 0
        dp = [[0 for _ in range(n + 1)] for _ in range(m +
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if A[i - 1] == B[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                    ans = max(ans, dp[i][j])
        return ans
```

复杂度分析

- 时间复杂度：$O(m*n)$，其中 m 和 n 分别为 A 和 B 的 长度。
- 空间复杂度：$O(m*n)$，其中 m 和 n 分别为 A 和 B 的 长度。

更多

- 你的衣服我扒了 - 《最长公共子序列》

扩展

二分查找也是可以的，不过并不容易想到，大家可以试试。

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(754. 到达终点数字)

https://leetcode-cn.com/problems/reach-a-number/

题目描述

```
在一根无限长的数轴上，你站在0的位置。终点在target的位置。

每次你可以选择向左或向右移动。第 n 次移动（从 1 开始），可以走 n 步。

返回到达终点需要的最小移动次数。

示例 1:

输入: target = 3
输出: 2
解释:
第一次移动，从 0 到 1 。
第二次移动，从 1 到 3 。
示例 2:

输入: target = 2
输出: 3
解释:
第一次移动，从 0 到 1 。
第二次移动，从 1 到 -1 。
第三次移动，从 -1 到 2 。
注意:

target是在[-10^9, 10^9]范围中的非零整数。
```

前置知识

- 数学

公司

思路

不难看出， 问题的本质就是一个有限序列， 1，2，3，4... 。我们的目标是给这个序列的元素添加正负号，是的其和为 target。这和 494.target-sum 的思路是一样的。

拿题目的 target = 3 来说， 就是 1 + 2 = 3。

拿题目的 target = 2 来说， 就是 1 - 2 + 3 = 2。

为什么是有限序列?

因为我们始终可以在 target 次以内走到 target。严格来说， 最少在根号 target 左右就可以走到 target。

和 494.target-sum 不同的是， 这道题数组是无限的，看起来似乎更难，实际上更简单， 因为数组是有规律的，每次都递增 1。

> 由于 target 正负是对称的， 因此 target 最少走多少布，-target 也是多少步。因此我们只考虑一种情况即可， 不妨只考虑正数的情况。

其实，只要找出第一个满足 1 + 2 + 3 + ... + steps > target 的 steps 即可。

- 如果 steps 是偶数， 那么我们总可以找出若干数字使其变为符号，满足 1 + 2 + 3 + ... + steps == target
- 如果 steps 是奇数， 1 + 2 + 3 + ... + steps + steps + 1 或者 1 + 2 + 3 + ... + steps + steps + 1 + steps + 2 中有且仅有一个是偶数。我们仍然可以套用上面的方法，找出若干数字使其变为符号，满足 1 + 2 + 3 + ... + steps + steps + 1 == target 或者 1 + 2 + 3 + ... + steps + steps + 1 + steps + 2 == target

关键点解析

- 对元素进行分组，分组的依据是符号， 是 `+` 或者 `-`
- 通过数学公式推导可以简化我们的求解过程，这需要一点 `数学知识和数学意识`

代码(Python)

Python Code:

```
class Solution(object):
    def reachNumber(self, target):
        target = abs(target)
        steps = 0
        while target > 0:
            steps += 1
            target -= steps
        if target & 1 == 0: return steps
        steps += 1
        if (target - steps) & 1 == 0: return steps
        return steps + 1
```

相关题目

- 494.target-sum

题目地址（785. 判断二分图）

https://leetcode-cn.com/problems/is-graph-bipartite/

题目描述

```
给定一个无向图 graph，当这个图为二分图时返回 true。

如果我们能将一个图的节点集合分割成两个独立的子集 A 和 B，并使图中的每

graph 将会以邻接表方式给出，graph[i]表示图中与节点 i 相连的所有节点

示例 1:
输入: [[1,3], [0,2], [1,3], [0,2]]
输出: true
解释:
无向图如下:
0----1
| |
| |
3----2
我们可以将节点分成两组: {0, 2} 和 {1, 3}。

示例 2:
输入: [[1,2,3], [0,2], [0,1,3], [0,2]]
输出: false
解释:
无向图如下:
0----1
| \ |
| \ |
3----2
我们不能将节点分割成两个独立的子集。
注意:

graph 的长度范围为 [1, 100]。
graph[i] 中的元素的范围为 [0, graph.length - 1]。
graph[i] 不会包含 i 或者有重复的值。
图是无向的: 如果 j 在 graph[i]里边, 那么 i 也会在 graph[j]里边。
```

前置知识

- 图的遍历
- DFS

公司

- 暂无

思路

和 886 思路一样。 我甚至**直接拿过来 dfs 函数一行代码没改就 AC 了**。

唯一需要调整的地方是 graph 。 我将其转换了一下，具体可以看代码，非常简单易懂。

具体算法：

- 设置一个长度为 N 的数组 colors，colors[i] 表示 节点 i 的颜色，0 表示无颜色， 1 表示一种颜色， - 1 表示另一种颜色。
- 初始化 colors 全部为 0
- 构图（这里有邻接矩阵） 使得 grid[i][j] 表示 i 和 j 是否有连接（这里用 0 表示无， 1 表示有）
- 遍历图。
  - 如果当前节点未染色，则染色，不妨染为颜色 1
  - 递归遍历其邻居
    - 如果邻居没有染色， 则染为另一种颜色。即 color * - 1，其中 color 为当前节点的颜色
    - 否则，判断当前节点和邻居的颜色是否一致，不一致则返回 False，否则返回 True

强烈建议两道题一起练习一下。

关键点

- 图的建立和遍历
- colors 数组

代码

```
class Solution:
    def dfs(self, grid, colors, i, color, N):
        colors[i] = color
        for j in range(N):
            if grid[i][j] == 1:
                if colors[j] == color:
                    return False
                if colors[j] == 0 and not self.dfs(grid, co
                    return False
        return True

    def isBipartite(self, graph: List[List[int]]) -> bool:
        N = len(graph)
        grid = [[0] * N for _ in range(N)]
        colors = [0] * N
        for i in range(N):
            for j in graph[i]:
                grid[i][j] = 1
        for i in range(N):
            if colors[i] == 0 and not self.dfs(grid, colors
                return False
        return True
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N)$

相关问题

- 886. 可能的二分法

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址（820. 单词的压缩编码）

https://leetcode-cn.com/problems/short-encoding-of-words/

题目描述

```
给定一个单词列表，我们将这个列表编码成一个索引字符串 S 与一个索引列表

例如，如果这个列表是 ["time", "me", "bell"]，我们就可以将其表示为

对于每一个索引，我们可以通过从字符串 S 中索引的位置开始读取字符串，直至

那么成功对给定单词列表进行编码的最小字符串长度是多少呢？



示例：

输入: words = ["time", "me", "bell"]
输出: 10
说明: S = "time#bell#" ，  indexes = [0, 2, 5] 。


提示：

1 <= words.length <= 2000
1 <= words[i].length <= 7
每个单词都是小写字母 。
```

前置知识

- 前缀树

公司

- 阿里
- 字节

思路

读完题目之后就发现如果将列表中每一个单词分别倒序就是一个后缀树问题。比如 `["time", "me", "bell"]` 倒序之后就是 ["emit", "em", "lleb"]，我们要求的结果无非就是 "emit" 的长度 + "llem"的长度 + "##"的长度（em 和 emit 有公共前缀，计算一个就好了）。

因此符合直觉的想法是使用前缀树 + 倒序插入的形式来模拟后缀树。

下面的代码看起来复杂，但是很多题目我都是用这个模板，稍微调整下细节就能 AC。我这里总结了一套前缀树专题

- 208.implement-trie-prefix-tree
- 211.add-and-search-word-data-structure-design
- 212.word-search-ii
- 472.concatenated-words

前缀树的 api 主要有以下几个：

- `insert(word)` : 插入一个单词
- `search(word)` ： 查找一个单词是否存在
- `startWith(word)` ： 查找是否存在以 word 为前缀的单词

其中 startWith 是前缀树最核心的用法，其名称前缀树就从这里而来。大家可以先拿 208 题开始，熟悉一下前缀树，然后再尝试别的题目。

一个前缀树大概是这个样子：

如图每一个节点存储一个字符，然后外加一个控制信息表示是否是单词结尾，实际使用过程可能会有细微差别，不过变化不大。

这道题需要考虑 edge case， 比如这个列表是 ["time", "time", "me", "bell"] 这种包含重复元素的情况，这里我使用 hashset 来去重。

关键点

- 前缀树
- 去重

代码

```
class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.Trie = {}

    def insert(self, word):
        """
        Inserts a word into the trie.
        :type word: str
        :rtype: void
        """
        curr = self.Trie
        for w in word:
            if w not in curr:
                curr[w] = {}
            curr = curr[w]
        curr['#'] = 1

    def search(self, word):
        """
        Returns if the word is in the trie.
        :type word: str
        :rtype: bool
        """
        curr = self.Trie
        for w in word:
            curr = curr[w]
        # len(curr) == 1 means we meet '#'
        # when we search 'em'(which reversed from 'me')
        # the result is len(curr) > 1
        # cause the curr look like { '#': 1, i: {...}}
        return len(curr) == 1
class Solution:
    def minimumLengthEncoding(self, words: List[str]) -> in
        trie = Trie()
        cnt = 0
        words = set(words)
        for word in words:
            trie.insert(word[::-1])
        for word in words:
            if trie.search(word[::-1]):
                cnt += len(word) + 1
        return cnt
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为单词长度列表中的总字符数，比如 ["time", "me"]，就是 4 + 2 = 6。
- 空间复杂度：$O(N)$，其中 N 为单词长度列表中的总字符数，比如 ["time", "me"]，就是 4 + 2 = 6。

大家也可以关注我的公众号《力扣加加》获取更多更新鲜的 LeetCode 题解

相关题目

- 0208.implement-trie-prefix-tree
- 0211.add-and-search-word-data-structure-design
- 0212.word-search-ii
- 0472.concatenated-words
- 0820.short-encoding-of-words
- 1032.stream-of-characters

题目地址（875. 爱吃香蕉的珂珂）

https://leetcode-cn.com/problems/koko-eating-bananas/description/

题目描述

```
珂珂喜欢吃香蕉。这里有 N 堆香蕉，第 i 堆中有 piles[i] 根香蕉。警卫已

珂珂可以决定她吃香蕉的速度 K （单位：根/小时）。每个小时，她将会选择一

珂珂喜欢慢慢吃，但仍然想在警卫回来前吃掉所有的香蕉。

返回她可以在 H 小时内吃掉所有香蕉的最小速度 K（K 为整数）。



示例 1：

输入: piles = [3,6,7,11], H = 8
输出: 4
示例 2：

输入: piles = [30,11,23,4,20], H = 5
输出: 30
示例 3：

输入: piles = [30,11,23,4,20], H = 6
输出: 23


提示：

1 <= piles.length <= 10^4
piles.length <= H <= 10^9
1 <= piles[i] <= 10^9
```

前置知识

- 二分查找

公司

- 字节

思路

符合直觉的做法是，选择最大的堆的香蕉数，然后试一下能不能行，如果不行则直接返回上次计算的结果，如果行，我们减少1个香蕉，试试行不行，依次类推。计算出刚好不行的即可。这种解法的时间复杂度比较高，为 $O(N*M)$，其中 N 为 piles 长度， M 为 Piles 中最大的数。。

这道题如果能看出来是二分法解决，那么其实很简单。为什么它是二分问题呢？我这里画了个图，我相信你看了就明白了。

> 香蕉堆的香蕉个数上限是 10^9， 珂珂这也太能吃了吧?

关键点解析

- 二分查找

代码

代码支持：Python，JavaScript

Python Code:

```
class Solution:
    def canEatAllBananas(self, piles, H, K):
        t = 0
        for pile in piles:
            t += math.ceil(pile / K)
        return t <= H
    def minEatingSpeed(self, piles: List[int], H: int) -> i
        l, r = 1, max(piles)
        # [l, r) ,  左闭右开的好处是如果能找到，那么返回 l 和 r 都
        while l < r:
            mid = (l + r) >> 1
            if self.canEatAllBananas(piles, H, mid):
                r = mid
            else:
                l = mid + 1
        return l
```

JavaScript Code:

```
function canEatAllBananas(piles, H, mid) {
     let h = 0;
     for(let pile of piles) {
        h += Math.ceil(pile / mid);
     }

     return h <= H;
 }
/**
 * @param {number[]} piles
 * @param {number} H
 * @return {number}
 */
var minEatingSpeed = function(piles, H) {
    let lo = 1,
    hi = Math.max(...piles);
    // [l, r) ,  左闭右开的好处是如果能找到，那么返回 l 和 r 都是-
    while(lo <= hi) {
        let mid = lo + ((hi - lo) >> 1);
        if (canEatAllBananas(piles, H, mid)) {
            hi = mid - 1;
        } else {
            lo = mid + 1;
        }
    }

    return lo; //  不能选择hi
};
```

复杂度分析

- 时间复杂度：$O(max(N, N*logM))$，其中 N 为 piles 长度， M 为 Piles 中最大的数。
- 空间复杂度：$O(1)$

模板

分享几个常用的的二分法模板。

查找一个数

```
public int binarySearch(int[] nums, int target) {
    // 左右都闭合的区间 [l, r]
    int left = 0;
    int right = nums.length - 1;

    while(left <= right) {
        int mid = left + (right - left) / 2;
        if(nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        else if (nums[mid] > target)
            // 搜索区间变为 [left, mid - 1]
            right = mid - 1;
    }
    return -1;
}
```

寻找最左边的满足条件的值

```
public int binarySearchLeft(int[] nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0;
    int right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        } else if (nums[mid] > target) {
            // 搜索区间变为 [left, mid-1]
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 收缩右边界
            right = mid - 1;
        }
    }
    // 检查是否越界
    if (left >= nums.length || nums[left] != target)
        return -1;
    return left;
}
```

寻找最右边的满足条件的值

```
public int binarySearchRight(int[] nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0
    int right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        } else if (nums[mid] > target) {
            // 搜索区间变为 [left, mid-1]
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 收缩左边界
            left = mid + 1;
        }
    }
    // 检查是否越界
    if (right < 0 || nums[right] != target)
        return -1;
    return right;
}
```

> 如果题目重点不是二分，也就是说二分只是众多步骤中的一步，大家也可以直接调用语言的 API，比如 Python 的 bisect 模块。

更多题解可以访问我的LeetCode题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经30K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode.com/problems/stone-game/description/

题目描述

```
Alex and Lee play a game with piles of stones.  There are a

The objective of the game is to end with the most stones.

Alex and Lee take turns, with Alex starting first.  Each tu

Assuming Alex and Lee play optimally, return True if and or



Example 1:

Input: [5,3,4,5]
Output: true
Explanation:
Alex starts first, and can only take the first 5 or the las
Say he takes the first 5, so that the row becomes [3, 4, 5]
If Lee takes 3, then the board is [4, 5], and Alex takes 5
If Lee takes the last 5, then the board is [3, 4], and Alex
This demonstrated that taking the first 5 was a winning mov


Note:

2 <= piles.length <= 500
piles.length is even.
1 <= piles[i] <= 500
sum(piles) is odd.
```

前置知识

- 动态规划

公司

- 阿里
- 字节

思路

由于 piles 是偶数的，并且 piles 的总和是奇数的。

因此 Alex `可以做到` 要不拿的全部是奇数，要么全部是偶数。

举个例子： 比如 Alex 第一次先拿第一个

这里有两种情况：

1. Lee 如果拿了第二块（偶数），那么 Alex 继续拿第三块，以此类推。。。
2. Lee 如果拿了最后一块（偶数），那么 Alex 继续拿倒数第二块，以此类推。。。

因此 Alex `可以` 做到只拿奇数或者偶数，只是他可以控制的，因此他要做的就是数一下，奇数加起来多还是偶数加起来多就好了。 奇数多就全部选奇数，偶数就全部选偶数。 Lee 是没有这种自由权的。

关键点解析

- 可以用 DP（动态规划）
- 可以从数学的角度去分析

> .....(😅)

代码

```
/**
 * @param {number[]} piles
 * @return {boolean}
 */
var stoneGame = function(piles) {
  return true;
};
```

扩展

腾讯面试题：一共 100 只弓箭 你和你的对手共用。你们每次只能射出一支箭或者两支箭，射击交替进行，设计一个算法，保证自己获胜。

答案： 先手，剩下的是 3 的倍数就行（100-1=99），然后按照 3 的倍数射箭必赢。 比如你先拿了 1，剩下 99 个。 对手拿了 1，你就拿 2。这样持续 33 次就赢了。如果对手拿了 2 个，你就拿 1 个，这样持续 33 次你也是赢的。

> 这是一种典型的博弈问题， 你和对手交替进行，对手的行动影响你接下来的策略。 这算是一种最简单的博弈问题了

题目地址（886. 可能的二分法）

https://leetcode-cn.com/problems/is-graph-bipartite/

题目描述

```
给定一组 N 人（编号为 1, 2, ..., N）， 我们想把每个人分进任意大小的

每个人都可能不喜欢其他人，那么他们不应该属于同一组。

形式上，如果 dislikes[i] = [a, b]，表示不允许将编号为 a 和 b 的人

当可以用这种方法将每个人分进两组时，返回 true；否则返回 false。



示例 1：

输入：N = 4, dislikes = [[1,2],[1,3],[2,4]]
输出：true
解释：group1 [1,4], group2 [2,3]
示例 2：

输入：N = 3, dislikes = [[1,2],[1,3],[2,3]]
输出：false
示例 3：

输入：N = 5, dislikes = [[1,2],[2,3],[3,4],[4,5],[1,5]]
输出：false


提示：

1 <= N <= 2000
0 <= dislikes.length <= 10000
dislikes[i].length == 2
1 <= dislikes[i][j] <= N
dislikes[i][0] < dislikes[i][1]
对于dislikes[i] == dislikes[j] 不存在 i != j
```

前置知识

- 图的遍历
- DFS

公司

- 暂无

思路

这是一个图的问题。解决这种问题一般是要遍历图才行的，这也是图的套路。 那么遍历的话，你要有一个合适的数据结构。 比较常见的图存储方式是邻接矩阵和邻接表。

而我们这里为了操作方便（代码量），直接使用邻接矩阵。由于是互相不喜欢，不存在一个喜欢另一个，另一个不喜欢一个的情况，因此这是无向图。而无向图邻接矩阵实际上是会浪费空间，具体看我下方画的图。

而题目给我们的二维矩阵并不是现成的邻接矩阵形式，因此我们需要自己生成。

我们用 1 表示互相不喜欢（dislike each other）。

```
graph = [[0] * N for i in range(N)]
for a, b in dislikes:
    graph[a - 1][b - 1] = 1
    graph[b - 1][a - 1] = 1
```

同时可以用 hashmap 或者数组存储 N 个人的分组情况， 业界关于这种算法一般叫染色法，因此我们命名为 colors，其实对应的本题叫 groups 更合适。

我们用：

- 0 表示没有分组
- 1 表示分组 1
- -1 表示分组 2

之所以用 0，1，-1，而不是 0，1，2 是因为我们会在不能分配某一组的时候尝试分另外一组，这个时候有其中一组转变为另外一组就可以直接乘以-1，而 0，1，2 这种就稍微麻烦一点而已。

具体算法：

- 遍历每一个人，尝试给他们进行分组，比如默认分配组 1.

- 然后遍历这个人讨厌的人，尝试给他们分另外一组，如果不可以分配另外一组，则返回 False

那问题的关键在于如何判断“不可以分配另外一组”呢?

实际上，我们已经用 colors 记录了分组信息，对于每一个人如果分组确定了，我们就更新 colors，那么对于一个人如果分配了一个组，并且他讨厌的人也被分组之后，**分配的组和它只能是一组**，那么“就是不可以分配另外一组”。

代码表示就是：

```
# 其中j 表示当前是第几个人，N表示总人数。 dfs的功能就是根据colors和
if colors[j] == 0 and not self.dfs(graph, colors, j, -1 * 
```

关键点

- 二分图
- 染色法
- 图的建立和遍历
- colors 数组

代码

```
class Solution:
    def dfs(self, graph, colors, i, color, N):
        colors[i] = color
        for j in range(N):
            # dislike eachother
            if graph[i][j] == 1:
                if colors[j] == color:
                    return False
                if colors[j] == 0 and not self.dfs(graph, c
                    return False
        return True

    def possibleBipartition(self, N: int, dislikes: List[Li
        graph = [[0] * N for i in range(N)]
        colors = [0] * N
        for a, b in dislikes:
            graph[a - 1][b - 1] = 1
            graph[b - 1][a - 1] = 1
        for i in range(N):
            if colors[i] == 0 and not self.dfs(graph, colo
                return False
        return True
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N)$

相关问题

- 785. 判断二分图

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode.com/problems/rle-iterator/description/

题目描述

```
Write an iterator that iterates through a run-length encode

The iterator is initialized by RLEIterator(int[] A), where

The iterator supports one function: next(int n), which exha

For example, we start with A = [3,8,0,9,2,5], which is a ru



Example 1:

Input: ["RLEIterator","next","next","next","next"], [[[3,8,
Output: [null,8,8,5,-1]
Explanation:
RLEIterator is initialized with RLEIterator([3,8,0,9,2,5]).
This maps to the sequence [8,8,8,5,5].
RLEIterator.next is then called 4 times:

.next(2) exhausts 2 terms of the sequence, returning 8.  Th

.next(1) exhausts 1 term of the sequence, returning 8.  The

.next(1) exhausts 1 term of the sequence, returning 5.  The

.next(2) exhausts 2 terms, returning -1.  This is because t
but the second term did not exist.  Since the last term exh

Note:

0 <= A.length <= 1000
A.length is an even integer.
0 <= A[i] <= 10^9
There are at most 1000 calls to RLEIterator.next(int n) per
Each call to RLEIterator.next(int n) will have 1 <= n <= 10
```

前置知识

- 哈夫曼编码和游程编码

公司

- 暂无

思路

这是一个游程编码的典型题目。

该算法分为两个部分，一个是初始化，一个是调用 `next(n)` .

我们需要做的就是初始化的时候，记住这个A。 然后每次调用 `next(n)` 的时候只需要

判断n是否大于Ai

- 如果大于A[i], 那就说明不够，我们移除数组前两项，更新n，重复1
- 如果小于A[i], 则说明够了，更新A[i]

这样做，我们每次都要更新A，还有一种做法就是不更新A，而是 `伪更新` ，即用一个变量记录，当前访问到的数组位置。

> 很多时候我们需要原始的，那么就必须这种放了，我的解法就是这种方法。

关键点解析

代码

```
/**
 * @param {number[]} A
 */
var RLEIterator = function(A) {
    this.A = A;
    this.current = 0;
};


/**
 * @param {number} n
 * @return {number}
 */
RLEIterator.prototype.next = function(n) {
    const A = this.A;
    while(this.current < A.length && A[this.current] < n){
        n = n - A[this.current];
        this.current += 2;
    }

    if(this.current >= A.length){
        return -1;
    }

    A[this.current] = A[this.current] - n; // 更新Count
    return A[this.current + 1]; // 返回element
};

/**
 * Your RLEIterator object will be instantiated and called
 * var obj = new RLEIterator(A)
 * var param_1 = obj.next(n)
 */
```

扩展阅读

哈夫曼编码和游程编码

题目地址

https://leetcode.com/problems/sort-an-array/

题目描述

```
Given an array of integers nums, sort the array in ascendi



Example 1:

Input: [5,2,3,1]
Output: [1,2,3,5]
Example 2:

Input: [5,1,1,2,0,0]
Output: [0,0,1,1,2,5]


Note:

1 <= A.length <= 10000
-50000 <= A[i] <= 50000
```

前置知识

- 数组
- 排序

公司

- 阿里
- 百度
- 字节

思路

这是一个很少见的直接考察 `排序` 的题目。 其他题目一般都是暗含 `排序` ，这道题则简单粗暴，直接让你排序。 并且这道题目的难度是 `Medium` ， 笔者感觉有点不可思议。

我们先来看题目的限制条件，这其实在选择算法的过程中是重要的。 看到这道题的时候，大脑就闪现出了各种排序算法， 这也算是一个复习 `排序算法` 的机会吧。

题目的限制条件是有两个，第一是元素个数不超过10k，这个不算大。 另外一个是数组中的每一项范围都是 `-50k` 到 `50k` （包含左右区间）。 看到这里，基本我就排除了时间复杂度为O(n^2)的算法。

> 我没有试时间复杂度 O(n^2) 的解法，大家可以试一下，看是不是会TLE。

剩下的就是基于比较的 `nlogn` 算法，以及基于特定条件的O(n)算法。

由于平时很少用到 `计数排序` 等O(n)的排序算法，一方面是空间复杂度不是常量，另一方面是其要求数据范围不是很大才行，不然会浪费很多空间。 但是这道题我感觉可以试一下。 在这里，我用了两种方法，一种是 `计数排序` ，一种是 `快速排序` 来解决。 大家也可以尝试用别的解法来解决。

解法一 - 计数排序

时间复杂度O(n)空间复杂度O(m) m 为数组中值的取值范围，在这道题就是 `50000 * 2 + 1` 。

我们只需要准备一个数组取值范围的数字，然后遍历一遍，将每一个元素放到这个数组对应位置就好了， 放的规则是 `索引为数字的值，value为出现的次数` 。

这样一次遍历，我们统计出了所有的数字出现的位置和次数。 我们再来一次遍历，将其输出到即可。

解法二 - 快速排序

快速排序和归并排序都是分支思想来进行排序的算法， 并且二者都非常流行。 快速排序的核心点在于选择轴元素。

每次我们将数组分成两部分，一部分是比pivot（轴元素）大的，另一部分是不比pivot大的。 我们不断重复这个过程， 直到问题的规模缩小的寻常（即只有一个元素的情况）。

快排的核心点在于如何选择轴元素，一般而言，选择轴元素有三种策略：

- 数组最左边的元素
- 数组最右边的元素
- 数组中间的元素（我采用的是这种，大家可以尝试下别的）
- 数组随机一项元素

(图片来自： https://www.geeksforgeeks.org/quick-sort/)

> 图片中的轴元素是最后面的元素，而提供的解法是中间元素，这点需要注意，但是这并不影响理解。

关键点解析

- 排序算法
- 注意题目的限制条件从而选择合适的算法

代码

计数排序：

代码支持： JavaScript

```
/**
 * @param {number[]} nums
 * @return {number[]}
 */
var sortArray = function(nums) {
    const counts = Array(50000 * 2 + 1).fill(0);
    const res = [];
    for(const num of nums) counts[50000 + num] += 1;
    for(let i in counts)  {
        while(counts[i]--) {
            res.push(i - 50000)
        }
    }
    return res;
};
```

快速排序：

代码支持： JavaScript

```
function swap(nums, a, b) {
    const temp = nums[a];
    nums[a] = nums[b];
    nums[b] = temp;
}

function helper(nums, start, end) {
    if (start >= end) return;
    const pivotIndex = start + ((end - start) >>> 1)
    const pivot = nums[pivotIndex]
    let i = start;
    let j = end;
    while (i <= j) {
        while (nums[i] < pivot) i++;
        while (nums[j] > pivot) j--;
        if (i <= j) {
            swap(nums, i, j);
            i++;
            j--;
        }
    }
    helper(nums, start, j);
    helper(nums, i, end);
}

/**
 * @param {number[]} nums
 * @return {number[]}
 */
var sortArray = function(nums) {
    helper(nums, 0, nums.length - 1);
    return nums;
};
```

扩展

- 你是否可以用其他方式排序算法解决

参考

- QuickSort - geeksforgeeks

题目地址 (935. 骑士拨号器)

https://leetcode-cn.com/problems/knight-dialer/

题目描述

国际象棋中的骑士可以按下图所示进行移动：

1	2	3
4	5	6
7	8	9
	0	

```
这一次，我们将 “骑士” 放在电话拨号盘的任意数字键（如上图所示）上，接下

每当它落在一个键上（包括骑士的初始位置），都会拨出键所对应的数字，总共

你能用这种方式拨出多少个不同的号码?

因为答案可能很大，所以输出答案模 10^9 + 7。



示例 1:

输入: 1
输出: 10
示例 2:

输入: 2
输出: 20
示例 3:

输入: 3
输出: 46


提示:

1 <= N <= 5000
```

前置知识

- DFS
- 记忆化搜索

深度优先遍历（DFS）

公司

- 暂无

思路

这道题要求解一个数字。并且每一个格子能够跳的状态是确定的。 因此我们的思路就是“状态机”（动态规划），暴力遍历（BFS or DFS），这里我们使用 DFS。（注意这几种思路并无本质不同）

对于每一个号码键盘，我们可以转移的状态是确定的，我们做一个”预处理“，将这些状态转移记录到一个数组 jump，其中 jump[i] 表示 i 位置可以跳的点（用一个数组来表示）。问题转化为：

- 从 0 开始所有的路径
- 从 1 开始所有的路径
- 从 2 开始所有的路径
- ...
- 从 9 开始所有的路径

不管从几开始思路都是一样的。 我们使用一个函数 f(i, n) 表示 `骑士在 i 的位置，还剩下 N 步可以走` 的时候可以拨出的总的号码数。那么问题就是求解 `f(0, N) + f(1, N) + f(2, N) + ... + f(9, N)` 。对于 f(i, n)，我们初始化 cnt 为 0，由于 i 能跳的格子是 jump[i]，我们将其 `cnt += f(j, n - 1)` ，其中 j 属于 jump[i]，最终返回 cnt 即可。

不难看出，这种算法有大量重复计算，我们使用记忆化递归形式来减少重复计算。 这种算法勉强通过。

代码

```
class Solution:
    def knightDialer(self, N: int) -> int:
        cnt = 0
        jump = [[4, 6], [6, 8], [7, 9], [4, 8], [
            0, 3, 9], [], [0, 1, 7], [2, 6], [1, 3], [2, 4]
        visited = dict()

        def helper(i, n):
            if (i, n) in visited: return visited[(i, n)]
            if n == 1:
                return 1
            cnt = 0
            for j in jump[i]:
                cnt += helper(j, n - 1)
            visited[(i, n)] = cnt
            return cnt
        for i in range(10):
            cnt += helper(i, N)
        return cnt % (10**9 + 7)
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

朴素遍历

思路

我们使用迭代的形式来优化上述过程。我们初始化十个变量分别表示键盘不同位置能够拨出的号码数，并且初始化为 1。接下来我们只要循环 N - 1 次，不断更新状态即可。不过这种算法和上述算法并无本质不同。

代码

```
class Solution:
    def knightDialer(self, N: int) -> int:
        a0 = a1 = a2 = a3 = a4 = a5 = a6 = a7 = a8 = a9 = 1
        for _ in range(N - 1):
            a0, a1, a2, a3, a4, a5, a6, a7, a8, a9 = a4 + a
                a9, a4 + a8, a0 + a3 + a9, 0, a0 + a1 + a7,
        return (a0 + a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

欢迎关注我的公众号《脑洞前端》获取更多更新鲜的 LeetCode 题解

题目地址(1014. 最佳观光组合)

https://leetcode-cn.com/problems/best-sightseeing-pair/description/

题目描述

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：

2 <= A.length <= 50000 1 <= A[i] <= 1000

前置知识

- 动态规划

公司

- 阿里
- 字节

思路

最简单的思路就是两两组合，找出最大的，妥妥超时，我们来看下代码：

```
class Solution:
    def maxScoreSightseeingPair(self, A: List[int]) -> int:
        n = len(A)
        res = 0
        for i in range(n - 1):
            for j in range(i + 1, n):
                res = max(res, A[i] + A[j] + i - j)
        return res
```

我们思考如何优化。 其实我们可以遍历一遍数组，对于数组的每一项 `A[j] - j` 我们都去前面找 `最大` 的 A[i] + i （这样才能保证结果最大）。

我们考虑使用动态规划来解决, 我们使用 dp[i] 来表示 数组 A 前 i 项的 `A[i] + i` 的最大值。

```
class Solution:
    def maxScoreSightseeingPair(self, A: List[int]) -> int:
        n = len(A)
        dp = [float('-inf')] * (n + 1)
        res = 0
        for i in range(n):
            dp[i + 1] = max(dp[i], A[i] + i)
            res = max(res, dp[i] + A[i] - i)
        return res
```

如上其实我们发现，dp[i + 1] 只和 dp[i] 有关，这是一个空间优化的信号。我们其实可以使用一个变量来记录，而不必要使用一个数组，代码见下方。

关键点解析

- 空间换时间
- dp 空间优化

代码

```
class Solution:
    def maxScoreSightseeingPair(self, A: List[int]) -> int:
        n = len(A)
        pre = A[0] + 0
        res = 0
        for i in range(1, n):
            res = max(res, pre + A[i] - i)
            pre = max(pre, A[i] + i)
        return res
```

小技巧

Python 的代码如果不使用 max，而是使用 if else 效率目测会更高，大家可以试一下。

```
class Solution:
    def maxScoreSightseeingPair(self, A: List[int]) -> int:
        n = len(A)
        pre = A[0] + 0
        res = 0
        for i in range(1, n):
            # res = max(res, pre + A[i] - i)
            # pre = max(pre, A[i] + i)
            res = res if res > pre + A[i] - i else pre + A
            pre = pre if pre > A[i] + i else A[i] + i
        return res
```

题目地址

https://leetcode-cn.com/problems/smallest-integer-divisible-by-k/description/

题目描述

```
给定正整数 K，你需要找出可以被 K 整除的、仅包含数字 1 的最小正整数 N

返回 N 的长度。如果不存在这样的 N，就返回 -1。



示例 1：

输入：1
输出：1
解释：最小的答案是 N = 1，其长度为 1。
示例 2：

输入：2
输出：-1
解释：不存在可被 2 整除的正整数 N 。
示例 3：

输入：3
输出：3
解释：最小的答案是 N = 111，其长度为 3。


提示：

1 <= K <= 10^5
```

前置知识

- 循环节

公司

- 暂无

思路

这道题是说给定一个 K 值，能否找到一个形如 1，11，111，1111 。。。这样的数字 n 使得 n % K == 0。

首先容易想到的是如果 K 是 2，4，5， 6，8 结尾的话，一定是不行的。直观的解法是从 1，11，111，1111 。。。 这样一直除下去，直到碰到可以整除的，我们返回即可。 但是如果这个数字根本就无法整除怎么办? 没错，我们会无限循环下去。我们应该在什么时刻跳出循环，返回 - 1（表示不能整除）呢?

我们拿题目给出的不能整除的 2 来说。

- 1 // 2 等于 1
- 11 // 2 等于 1
- 111 // 2 等于 1
- ...

我们再来一个不能整除的例子 6:

- 1 // 6 等于 1
- 11 // 6 等于 5
- 111 // 6 等于 3
- 1111 // 6 等于 1
- 11111 // 6 等于 5
- ...

通过观察我们发现不断整除的过程，会陷入无限循环，对于 2 来说，其循环节就是 1。对于 6 来说，其循环节来说就是 153。而且由于我们的分母是 6，也就是说余数的可能性一共只有六种情况 0,1,2,3,4,5。

上面是感性的认识， 接下来我们从数学上予以证明。上面的算法用公式来表示就是 `mod = (10 \* mod + 1) % K` 。假如出现了相同的数，我们可以肯定之后会无限循环。比如 153 之后出现了 1，我们可以肯定之后一定是 35。。。 因为我们的 mod 只是和前一个 mod 有关，上面的公式是一个 `纯函数` 。

关键点解析

- 数学（无限循环与循环节）

代码

```python
#
# @lc app=leetcode.cn id=1015 lang=python3
#
# [1015] 可被 K 整除的最小整数
#

# @lc code=start


class Solution:
    def smallestRepunitDivByK(self, K: int) -> int:
        if K % 10 in [2, 4, 5, 6, 8]:
            return - 1
        seen = set()
        mod = 0
        for i in range(1, K + 1):
            mod = (mod * 10 + 1) % K
            if mod in seen:
                return -1
            if mod == 0:
                return ix
            seen.add(mod)

        # @lc code=end
```

相关题目

- 166. 分数到小数

题目地址

https://leetcode-cn.com/problems/next-greater-node-in-linked-list/submissions/

题目描述

```
给出一个以头节点 head 作为第一个节点的链表。链表中的节点分别编号为：n

每个节点都可能有下一个更大值（next larger value）：对于 node_i，如

返回整数答案数组 answer，其中 answer[i] = next_larger(node_{i+

注意：在下面的示例中，诸如 [2,1,5] 这样的输入（不是输出）是链表的序列



示例 1：

输入：[2,1,5]
输出：[5,5,0]
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]


提示：

对于链表中的每个节点，1 <= node.val <= 10^9
给定列表的长度在 [0, 10000] 范围内
```

前置知识

- 链表
- 栈

公司

- 腾讯
- 字节

思路

看完题目就应该想到单调栈才行，LeetCode 上关于单调栈的题目还不少，难度都不小。但是一旦你掌握了这个算法，那么这些题目对你来说都不是问题了。

如果你不用单调栈，那么可以暴力$O(N^2)$的时间复杂度解决，只需要双层循环即可。但是这种做法应该是过不了关的。使用单调栈可以将时间复杂度降低到线性，当然需要额外的$O(N)$的空间复杂度。

顾名思义，单调栈即满足单调性的栈结构。与单调队列相比，其只在一端进行进出。为了描述方便，以下举例及代码以维护一个整数的单调递减栈为例。将一个元素插入单调栈时，为了维护栈的单调性，需要在保证将该元素插入到栈顶后整个栈满足单调性的前提下弹出最少的元素。

例如，栈中自顶向下的元素为 1，2，4，5 ，插入元素 3 时为了保证单调性需要依次弹出元素：

- 最开始栈是这样的： [5,4,2,1]
- 为了维护递减特性，1,2 需要被移除。此时栈是这样的： [5,4]
- 我们将 3 push 到栈顶即可
- 此时栈是这样的： [5,4,3]

用代码描述如下：

Python Code:

```
def monoStack(list):
    st = []
    for v in list:
        while len(st) > 0 and v > st[-1]:
            st.pop()
        st.append(v)
    return st
monoStack([5, 4, 2, 1, 3]) # output: [5, 4, 3]
```

关键点

- 单调栈（单调递减栈）
- 单调栈的代码模板

代码

Python Code:

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
     def nextLargerNodes(self, head):
        res, st = [], []
        while head:
            while len(st) > 0 and head.val > st[-1][1]:
                res[st.pop()[0]] = head.val
            st.append((len(res), head.val))
            res.append(0)
            head = head.next
        return res
```

扩展

甚至可以做到 O(1)的空间复杂度，请参考C# O(n) time O(1) space-time-O(1)-space>)

相关题目

- 每日一题 - 739.Daily Temperatures

题目地址

https://leetcode-cn.com/problems/number-of-enclaves/

题目描述

```
给出一个二维数组 A，每个单元格为 0（代表海）或 1（代表陆地）。

移动是指在陆地上从一个地方走到另一个地方（朝四个方向之一）或离开网格的

返回网格中无法在任意次数的移动中离开网格边界的陆地单元格的数量。



示例 1：

输入：[[0,0,0,0],[1,0,1,0],[0,1,1,0],[0,0,0,0]]
输出：3
解释：
有三个 1 被 0 包围。一个 1 没有被包围，因为它在边界上。
示例 2：

输入：[[0,1,1,0],[0,0,1,0],[0,0,1,0],[0,0,0,0]]
输出：0
解释：
所有 1 都在边界上或可以到达边界。


提示：

1 <= A.length <= 500
1 <= A[i].length <= 500
0 <= A[i][j] <= 1
所有行的大小都相同
```

前置知识

- DFS
- hashset

解法一 (暴力法)

思路

这是一个典型的可以使用 DFS 进行解决的一类题目， LeetCode 相关的题目有很多。

对于这种题目不管是思路还是代码都有很大的相似性，我们来看下。

暴力法的思路很简单，我们遍历整个矩阵：

- 如果遍历到 0，我们不予理会
- 如果遍历到 1. 我们将其加到 temp
- 我们不断拓展边界（上下左右）
- 如果 dfs 过程中碰到了边界，说明我们可以逃脱，我们将累加的 temp 清空
- 如果 dfs 过程之后没有碰到边界，说明我们无法逃脱。我们将 temp 加到 cnt
- 最终返回 cnt 即可

关键点解析

- visited 记录访问过的节点，防止重复计算

代码

Python Code:

```
class Solution:
    temp = 0
    meetEdge = False

    def numEnclaves(self, A: List[List[int]]) -> int:
        cnt = 0
        m = len(A)
        n = len(A[0])
        visited = set()

        def dfs(i, j):
            if i < 0 or i >= m or j < 0 or j >= n or (i, j)
                return
            visited.add((i, j))
            if A[i][j] == 1:
                self.temp += 1
            else:
                return
            if i == 0 or i == m - 1 or j == 0 or j == n - 1
                self.meetEdge = True
            dfs(i + 1, j)
            dfs(i - 1, j)
            dfs(i, j + 1)
            dfs(i, j - 1)
        for i in range(m):
            for j in range(n):
                dfs(i, j)
                if not self.meetEdge:
                    cnt += self.temp
                self.meetEdge = False
                self.temp = 0
        return cnt
```

解法二 (消除法)

公司

- 暂无

思路

上面的解法时间复杂度和空间复杂度都很差，我们考虑进行优化， 这里我们使用消除法。

- 从矩阵边界开始 dfs

- 如果碰到 1 就将其变成 0
- 如果碰到 0 则什么都不做
- 最后我们遍历整个矩阵，数一下 1 的个数即可。

关键点解析

- dfs 消除法

代码

Python Code:

```python
#
# @lc app=leetcode.cn id=1020 lang=python3
#
# [1020] 飞地的数量
#

# @lc code=start


class Solution:

    def numEnclaves(self, A: List[List[int]]) -> int:
        cnt = 0
        m = len(A)
        n = len(A[0])

        def dfs(i, j):
            if i < 0 or i >= m or j < 0 or j >= n or A[i][j
                return
            A[i][j] = 0

            dfs(i + 1, j)
            dfs(i - 1, j)
            dfs(i, j + 1)
            dfs(i, j - 1)
        for i in range(m):
            dfs(i, 0)
            dfs(i, n - 1)
        for j in range(1, n - 1):
            dfs(0, j)
            dfs(m - 1, j)
        for i in range(m):
            for j in range(n):
                if A[i][j] == 1:
                    cnt += 1
        return cnt

        # @lc code=end
```

参考

- 200.number-of-islands

题目地址

https://leetcode-cn.com/problems/camelcase-matching/

题目描述

```
如果我们可以将小写字母插入模式串 pattern 得到待查询项 query，那么待

给定待查询列表 queries，和模式串 pattern，返回由布尔值组成的答案列表




示例 1：

输入：queries = ["FooBar","FooBarTest","FootBall","FrameBuff
输出：[true,false,true,true,false]
示例：
"FooBar" 可以这样生成："F" + "oo" + "B" + "ar"。
"FootBall" 可以这样生成："F" + "oot" + "B" + "all".
"FrameBuffer" 可以这样生成："F" + "rame" + "B" + "uffer".
示例 2：

输入：queries = ["FooBar","FooBarTest","FootBall","FrameBuff
输出：[true,false,true,false,false]
解释：
"FooBar" 可以这样生成："Fo" + "o" + "Ba" + "r".
"FootBall" 可以这样生成："Fo" + "ot" + "Ba" + "ll".
示例 3：

输出：queries = ["FooBar","FooBarTest","FootBall","FrameBuff
输入：[false,true,false,false,false]
解释：
"FooBarTest" 可以这样生成："Fo" + "o" + "Ba" + "r" + "T" + "e



提示：

1 <= queries.length <= 100
1 <= queries[i].length <= 100
1 <= pattern.length <= 100
所有字符串都仅由大写和小写英文字母组成。
```

前置知识

- 双指针

公司

- 暂无

思路

这道题是一道典型的双指针题目。不过这里的双指针并不是指向同一个数组或者字符串，而是指向多个，这道题是指向两个，分别是 query 和 pattern，这种题目非常常见，能够识别和掌握这种题目的解题模板非常重要。对 queries 的每一项我们的逻辑是一样的，这里就以其中一项为例进行讲解。

以 query 为 FooBar，pattern 为 FB 为例。

首先我们来简化一下问题，假如我们没有 `可以在任何位置插入每个字符，也可以插入 0 个字符。` 这个规则。我们的问题会比较简单，这个时候我们的算法是什么样的呢？一起来看下：

1. 首先我们建立两个指针 i 和 j 分别指向 query 和 pattern 的首字母。
2. 当 i 和 j 指向的字母相同的时候，我们同时向后移动两个指针一个单位。
3. 当 i 和 j 指向的字母不同的时候，我们直接返回 False

假如我们要找到的不是子串，而是子序列怎么办？我们不妨假设判断 pattern 是否是 query 的子序列。 其实 LeetCode 实际上也有这样的题目，我们来看下：

1. 首先我们建立两个指针 i 和 j 分别指向 query 和 pattern 的首字母。
2. 当 i 和 j 指向的字母相同的时候，我们同时向后移动两个指针一个单位。
3. 当 i 和 j 指向的字母不同的时候，我们移动 i 指针。
4. 当 i 超出 query 范围的时候，我们只需要判断 pattern 是否达到了终点即可。当然我们也可以提前退出。

我们直接参考下 LeetCode 392. 判断子序列。

代码：

> 给定字符串 s 和 t ，判断 s 是否为 t 的子序列

Python Code:

```python
class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        i = 0
        j = 0
        while j < len(t):
            if i < len(s) and s[i] == t[j]:
                i += 1
                j += 1
            else:
                j += 1
            if i >= len (s):
                return True
        return i == len(s)
```

然后我们加上 `可以在任何位置插入每个字符，也可以插入 0 个字符。` 这个规则。来看下有什么不同：

1. 首先我们建立两个指针 i 和 j 分别指向 query 和 pattern 的首字母。
2. 当 i 和 j 指向的字母相同的时候，我们同时向后移动两个指针一个单位。
3. 当 i 和 j 指向的字母不同的时候，我们继续判断 i 指向的元素是否是小写。
4. 如果是小写我们只把 i 向后移动一个单位。
5. 如果不是小写我们直接返回 False

关键点解析

- 双指针
- 字符串匹配
- 子序列
- 子串

代码

Python Code:

```
class Solution:
    def camelMatch(self, queries: List[str], pattern: str)
        res = []
        for query in queries:
            i = 0
            j = 0
            while i < len(query):
                if j < len(pattern) and query[i] == patter
                    i += 1
                    j += 1
                elif query[i].islower():
                    i += 1
                else:
                    break
            if i == len(query) and j == len(pattern):
                res.append(True)
            else:
                res.append(False)
        return res
```

扩展

这是一个符合直觉的解法，但是却不是一个很优秀的解法，那么你有想到什么优秀的解法么？

参考

- 392. 判断子序列

题目地址

https://leetcode.com/problems/maximum-sum-of-two-non-overlapping-subarrays/

题目描述

```
Given an array A of non-negative integers, return the maxir

Formally, return the largest V for which V = (A[i] + A[i+1]

0 <= i < i + L - 1 < j < j + M - 1 < A.length, or
0 <= j < j + M - 1 < i < i + L - 1 < A.length.


Example 1:

Input: A = [0,6,5,2,2,5,1,9,4], L = 1, M = 2
Output: 20
Explanation: One choice of subarrays is [9] with length 1,
Example 2:

Input: A = [3,8,1,3,2,1,8,9,0], L = 3, M = 2
Output: 29
Explanation: One choice of subarrays is [3,8,1] with length
Example 3:

Input: A = [2,1,5,6,0,9,5,0,3,8], L = 4, M = 3
Output: 31
Explanation: One choice of subarrays is [5,6,0,9] with leng


Note:

L >= 1
M >= 1
L + M <= A.length <= 1000
0 <= A[i] <= 1000
```

前置知识

- 数组

公司

- 字节

思路(动态规划)

题目中要求在前N(数组长度)个数中找出长度分别为L和M的非重叠子数组之和的最大值, 因此, 我们可以定义数组A中前i个数可构成的非重叠子数组L和M的最大值为SUMM[i], 并找到SUMM[i]和SUMM[i-1]的关系, 那么最终解就是SUMM[N]. 以下为图解:

关键点解析

1. 注意图中描述的都是A[i-1], 而不是A[i], 因为base case为空数组, 而不是A[0];
2. 求解图中ASUM数组的时候, 注意定义的是ASUM[i] = sum(A[0:i]), 因此当i等于0时, A[0:0]为空数组, 即: ASUM[0]为0, 而ASUM[1]才等于A[0];

3. 求解图中MAXL数组时, 注意i < L时, 没有意义, 因为长度不够, 所以从i = L时才开始求解;
4. 求解图中MAXM数组时, 也一样, 要从i = M时才开始求解;
5. 求解图中SUMM数组时, 因为我们需要一个L子数组和一个M子数组, 因此长度要大于等于L+M才有意义, 所以要从i = L + M时开始求解.

代码

- 语言支持: Python

Python Code:

```
class Solution:
    def maxSumTwoNoOverlap(self, a: List[int], l: int, m: 
        """

        define asum[i] as the sum of subarray, a[0:i]
        define maxl[i] as the maximum sum of l-length suba
        define maxm[i] as the maximum sum of m-length suba
        define msum[i] as the maximum sum of non-overlap l-

        case 1: a[i] is both not in l-length subarray and 
        case 2: a[i] is in l-length subarray, then msum[i]
        case 3: a[i] is in m-length subarray, then msum[i]

        so, msum[i] = max(msum[i - 1], asum[i] - asum[i-l]
        """

        alen, tlen = len(a), l + m
        asum = [0] * (alen + 1)
        maxl = [0] * (alen + 1)
        maxm = [0] * (alen + 1)
        msum = [0] * (alen + 1)

        for i in range(tlen):
            if i == 1:
                asum[i] = a[i - 1]
            elif i > 1:
                asum[i] = asum[i - 1] + a[i - 1]
            if i >= l:
                maxl[i] = max(maxl[i - 1], asum[i] - asum[
            if i >= m:
                maxm[i] = max(maxm[i - 1], asum[i] - asum[

        for i in range(tlen, alen + 1):
            asum[i] = asum[i - 1] + a[i - 1]
            suml = asum[i] - asum[i - l]
            summ = asum[i] - asum[i - m]
            maxl[i] = max(maxl[i - 1], suml)
            maxm[i] = max(maxm[i - 1], summ)
            msum[i] = max(msum[i - 1], suml + maxm[i - l],

        return msum[-1]
```

扩展

1. 代码中, 求解了4个动态规划数组来求解最终值, 有没有可能只用两个数组来求解该题, 可以的话, 需要保留的又是哪两个数组?

2. 代码中, 求解的4动态规划数组的顺序能否改变, 哪些能改, 哪些不能改?

如果采用前缀和数组的话，可以只使用O(n)的空间来存储前缀和，O(1)的动态规划状态空间来完成。C++代码如下:

```
class Solution {
public:
    int maxSumTwoNoOverlap(vector<int>& A, int L, int M) {
        auto tmp = vector<int>{A[0]};
        for (auto i = 1; i < A.size(); ++i) {
            tmp.push_back(A[i] + tmp[i - 1]);
        }
        auto res = tmp[L + M - 1], lMax = tmp[L - 1], mMax
        for (auto i = L + M; i < tmp.size(); ++i) {
            lMax = max(lMax, tmp[i - M] - tmp[i - M - L]);
            mMax = max(mMax, tmp[i - L] - tmp[i - L - M]);
            res = max(res, max(lMax + tmp[i] - tmp[i - M],
        }
        return res;
    }
};
```

题目地址

https://leetcode-cn.com/problems/path-in-zigzag-labelled-binary-tree/description/

题目描述

在一棵无限的二叉树上，每个节点都有两个子节点，树中的节点 逐行 依次按 “之” 字形进行标记。

如下图所示，在奇数行（即，第一行、第三行、第五行……）中，按从左到右的顺序进行标记；

而偶数行（即，第二行、第四行、第六行……）中，按从右到左的顺序进行标记。

给你树上某一个节点的标号 label，请你返回从根节点到该标号为 label 节点的路径，该路径是由途经的节点标号所组成的。

示例 1：

输入：label = 14 输出：[1,3,4,14] 示例 2：

输入：label = 26 输出：[1,2,6,10,26]

提示：

1 <= label <= 10^6

前置知识

- 二叉树

公司

- 暂无

思路

假如这道题不是之字形，那么就会非常简单。 我们可以根据子节点的 label 轻松地求出父节点的 label，公示是 label // 2（其中 label 为子节点的 label）。

如果是这样的话，这道题应该是 easy 难度，代码也不难写出。我们继续考虑之字形。我们不妨先观察一下，找下规律。

以上图最后一行为例，对于 15 节点，之字变换之前对应的应该是 8 节点。14 节点对应的是 9 节点。。。

全部列举出来是这样的：

我们发现之字变换前后的 label 相加是一个定值。

因此我们只需要求解出每一层的这个定值，然后减去当前值就好了。（注意我们不需要区分偶数行和奇数行） 问题的关键转化为求解这个定值，这个定值其实很好求，因为每一层的最大值和最小值我们很容易求，而最大值和最小值的和正是我们要求的这个数字。

最大值和最小值这么求呢？由满二叉树的性质，我们知道每一层的最小值就是 `2 ** (level - 1)` ，而最大值是 `2 ** level - 1` 。 因此我们只要知道 level 即可，level 非常容易求出，具体可以看下面代码。

关键点

- 满二叉树的性质：
- 最小值是 `2 ** (level - 1)` ，最大值是 `2 ** level - 1` ，其中 level 是树的深度。
- 假如父节点的索引为 i，那么左子节点就是 2*i， 右边子节点就是 2*i + 1。
- 假如子节点的索引是 i，那么父节点的索引就是 i // 2。
- 先思考一般情况（不是之字形）， 然后通过观察找出规律

代码

```
class Solution:
    def pathInZigZagTree(self, label: int) -> List[int]:
        level = 0
        res = []
        # for each level, ranged from 2 ** (level - 1) to 2
        while 2 ** level - 1 < label:
            level += 1

        while level > 0:
            res.insert(0, label)
            label = 2 ** (level - 1) + 2 ** level - 1 - lab
            label //= 2
            level -= 1
        return res
```

题目地址（1131. 绝对值表达式的最大值）

https://leetcode-cn.com/problems/maximum-of-absolute-value-expression/description/

题目描述

给你两个长度相等的整数数组，返回下面表达式的最大值：

|arr1[i] - arr1[j]| + |arr2[i] - arr2[j]| + |i - j|

其中下标 i，j 满足 0 <= i, j < arr1.length。

示例 1：

输入：arr1 = [1,2,3,4], arr2 = [-1,4,5,6] 输出：13 示例 2：

输入：arr1 = [1,-2,-5,0,10], arr2 = [0,-2,-1,-7,-4] 输出：20

提示：

2 <= arr1.length == arr2.length <= 40000 -10^6 <= arr1[i], arr2[i] <= 10^6

前置知识

- 数组

解法一（数学分析）

公司

- 阿里
- 腾讯
- 字节

思路

如图我们要求的是这样一个表达式的最大值。arr1 和 arr2 为两个不同的数组，且二者长度相同。i 和 j 是两个合法的索引。

> 红色竖线表示的是绝对值的符号

|arr1[i] - arr1[j]| + |arr2[i] - arr2[j]| + |i - j|

我们对其进行分类讨论，有如下八种情况：

> |arr1[i] -arr1[j]| 两种情况 |arr2[i] -arr2[j]| 两种情况 |i - j| 两种情况 因此一共是 2 * 2 * 2 = 8 种

1 arr1[i] - arr1[j] + arr2[i] - arr2[j] + i - j

2 arr1[i] - arr1[j] - (arr2[i] - arr2[j]) + i - j

3 -(arr1[i] - arr1[j]) + arr2[i] - arr2[j] + i - j

4 -(arr1[i] - arr1[j]) - (arr2[i] - arr2[j]) + i - j

5 arr1[i] - arr1[j] + arr2[i] - arr2[j] - (i - j)

6 arr1[i] - arr1[j] - (arr2[i] - arr2[j]) - (i - j)

7 -(arr1[i] - arr1[j]) + arr2[i] - arr2[j] - (i - j)

8 -(arr1[i] - arr1[j]) - (arr2[i] - arr2[j]) - (i - j)

由于 i 和 j 之前没有大小关系，也就说二者可以相互替代。因此：

- 1 等价于 8
- 2 等价于 7
- 3 等价于 6
- 4 等价于 5

也就是说我们只需要计算 1，2，3，4 的最大值就可以了。（当然你可以选择其他组合，只要完备就行）

为了方便，我们将 i 和 j 都提取到一起：

$$arr1[i] + arr2[i] + i \ - (arr1[j] + arr2[j] + j)$$

$$arr1[i] - arr2[i] + i \ - (arr1[j] - arr2[j] + j)$$

$$arr2[i] - arr1[i] + i \ - (arr2[j] - arr1[j] + j)$$

$$-arr2[i] - arr1[i] + i \ - (-arr1[j] - arr2[j] + j)$$

容易看出等式的最大值就是前面的最大值，和后面最小值的差值。如图：

$$\boxed{arr1[i] + arr2[i] + i} \ - \boxed{(arr1[j] + arr2[j] + j)}$$

$$\boxed{arr1[i] - arr2[i] + i} \ - \boxed{(arr1[j] - arr2[j] + j)}$$

$$\boxed{arr2[i] - arr1[i] + i} \ - \boxed{(arr2[j] - arr1[j] + j)}$$

$$\boxed{-arr2[i] - arr1[i] + i} \ - \boxed{(-arr1[j] - arr2[j] + j)}$$

再仔细观察，会发现前面部分和后面部分是一样的，原因还是上面所说的 i 和 j 可以互换。因此我们要做的就是：

- 遍历一遍数组，然后计算四个表达式， arr1[i] + arr2[i] + i，arr1[i] - arr2[i] + i，arr2[i] - arr1[i] + i 和 -1 * arr2[i] - arr1[i] + i 的 最大值和最小值。
- 然后分别取出四个表达式最大值和最小值的差值（就是这个表达式的最大值）
- 四个表达式最大值再取出最大值

关键点

- 数学分析

代码

```
class Solution:
    def maxAbsValExpr(self, arr1: List[int], arr2: List[int
        A = []
        B = []
        C = []
        D = []
        for i in range(len(arr1)):
            a, b, c, d = arr1[i] + arr2[i] + i, arr1[i] - a
                i, arr2[i] - arr1[i] + i, -1 * arr2[i] - ar
            A.append(a)
            B.append(b)
            C.append(c)
            D.append(d)
        return max(max(A) - min(A), max(B) - min(B), max(C)
```

解法二（曼哈顿距离）

思路

曼哈顿与欧几里得距离：红、蓝与黄线分别表示所有曼哈顿距离都拥有一样长度（12），而绿线表示欧几里得距离有6×√2 ≈ 8.48的长度。

（图来自：https://zh.wikipedia.org/wiki/%E6%9B%BC%E5%93%88%E9%A0%93%E8%B7%9D%E9%9B%A2）

一维曼哈顿距离可以理解为一条线上两点之间的距离: |x1 - x2|，其值为 max(x1 - x2, x2 - x1)

在平面上，坐标（x1, y1）的点 P1 与坐标（x2, y2）的点 P2 的曼哈顿距离为：|x1-x2| + |y1 - y2|,其值为 max(x1 - x2 + y1 - y2, x2 - x1 + y1 - y2, x1 - x2 + y2 - y1, x2 -x1 + y2 - y1)

然后这道题目是更复杂的三维曼哈顿距离，其中(i, arr[i], arr[j])可以看作三位空间中的一个点，问题转化为曼哈顿距离最远的两个点的距离。 延续上面的思路，|x1-x2| + |y1 - y2| + |z1 - z2|,其值为 :

max(

x1 - x2 + y1 - y2 + z1 - z2,

x1 - x2 + y1 - y2 + z2 - z1,

x2 - x1 + y1 - y2 + z1 - z2,

x2 - x1 + y1 - y2 + z2 - z1,

x1 - x2 + y2 - y1 + z1 - z2,

x1 - x2 + y2 - y1 + z2- z1,

x2 -x1 + y2 - y1 + z1 - z2,

x2 -x1 + y2 - y1 + z2 - z1

)

我们可以将 1 和 2 放在一起方便计算：

max(

x1 + y1 + z1 - (x2 + y2 + z2),

x1 + y1 - z1 - (x2 + y2 - z2)

...

)

我们甚至可以扩展到 n 维，具体代码见下方。

关键点

- 曼哈顿距离
- 曼哈顿距离代码模板

> 解题模板可以帮助你快速并且更少错误的解题，更多解题模板请期待我的新书(未完成)

代码

```
class Solution:
    def maxAbsValExpr(self, arr1: List[int], arr2: List[int
        # 曼哈顿距离模板代码
        sign = [1, -1]
        n = len(arr1)
        dists = []
        # 三维模板
        for a in sign:
            for b in sign:
                for c in sign:
                    maxDist = float('-inf')
                    minDist = float('inf')
                    # 分别计算所有点的曼哈顿距离
                    for i in range(n):
                        dist = arr1[i] * a + arr2[i] * b +
                        maxDist = max(maxDist, dist)
                        minDist = min(minDist, dist)
                    # 将所有的点的曼哈顿距离放到dists中
                    dists.append(maxDist - minDist)
        return max(dists)
```

总结

可以看出其实两种解法都是一样的，只是思考角度不一样。

相关题目

- 1030. 距离顺序排列矩阵单元格

1030. 距离顺序排列矩阵单元格

难度 简单 18 收藏 分享 切换为英文 关注

题目描述 评论 (72) 题解(40) 提交记录 Python3 智能模式

给出 R 行 C 列的矩阵，其中的单元格的整数坐标为 (r, c)，满足 0 <= r < R 且 0 <= c < C。

另外，我们在该矩阵中给出了一个坐标为 (r0, c0) 的单元格。

返回矩阵中的所有单元格的坐标，并按到 (r0, c0) 的距离从最小到最大的顺序排，其中，两单元格 (r1, c1) 和 (r2, c2) 之间的距离是曼哈顿距离，|r1 - r2| + |c1 - c2|。（你可以按任何满足此条件的顺序返回答案。）

示例 1：

输入：R = 1, C = 2, r0 = 0, c0 = 0
输出：[[0,0],[0,1]]
解释：从 (r0, c0) 到其他单元格的距离为：[0,1]

示例 2：

输入：R = 2, C = 2, r0 = 0, c0 = 1
输出：[[0,1],[0,0],[1,1],[1,0]]
解释：从 (r0, c0) 到其他单元格的距离为：[0,1,1,2]
[[0,1],[1,1],[0,0],[1,0]] 也会被视作正确答案。

示例 3：

输入：R = 2, C = 3, r0 = 1, c0 = 2
输出：[[1,2],[0,2],[1,1],[0,1],[1,0],[0,0]]
解释：从 (r0, c0) 到其他单元格的距离为：[0,1,1,2,2,3]
其他满足题目要求的答案也会被视为正确，例如 [[1,2],
[1,1],[0,2],[1,0],[0,1],[0,0]]

```python
#
# @lc app=leetcode.cn id=1030 lang=python3
#
# [1030] 距离顺序排列矩阵单元格
#

# @lc code=start


class Solution:
    def allCellsDistOrder(self, R: int, C: int, r0: int, c0: int) -> List[List[int]]:
        points = []

        for r in range(R):
            for c in range(C):
                points.append([r,c])

        return sorted(points, key=lambda x: abs((x[0]-r0)) + abs((x[1]-c0)))


# @lc code=end

```

测试用例 代码执行结果

已完成 执行用时: 32 ms

- 1162. 地图分析

题目地址

https://leetcode.com/problems/maximum-subarray-sum-with-one-deletion/

题目描述

```
给你一个整数数组，返回它的某个 非空 子数组（连续元素）在执行一次可选的

换句话说，你可以从原数组中选出一个子数组，并可以决定要不要从中删除一个

注意，删除一个元素后，子数组 不能为空。

请看示例：

示例 1：

输入：arr = [1,-2,0,3]
输出：4
解释：我们可以选出 [1, -2, 0, 3]，然后删掉 -2，这样得到 [1, 0, 3]
示例 2：

输入：arr = [1,-2,-2,3]
输出：3
解释：我们直接选出 [3]，这就是最大和。
示例 3：

输入：arr = [-1,-1,-1,-1]
输出：-1
解释：最后得到的子数组不能为空，所以我们不能选择 [-1] 并从中删去 -1
     我们应该直接选择 [-1]，或者选择 [-1, -1] 再从中删去一个 -1。


提示：

1 <= arr.length <= 10^5
-10^4 <= arr[i] <= 10^4
```

前置知识

- 数组
- 动态规划

公司

- 字节

思路

暴力法

符合知觉的做法是求出所有的情况，然后取出最大的。 我们只需要两层循环接口，外循环用于确定我们丢弃的元素，内循环用于计算subArraySum。

```
class Solution:
  def maximumSum(self, arr: List[int]) -> int:
      res = arr[0]
      def maxSubSum(arr, skip):
          res = maxSub = float("-inf")

          for i in range(len(arr)):
              if i == skip:
                  continue
              maxSub = max(arr[i], maxSub + arr[i])
              res = max(res, maxSub)
          return res
      # 这里循环到了len(arr)项，表示的是一个都不删除的情况
      for i in range(len(arr) + 1):
          res = max(res, maxSubSum(arr, i))
      return res
```

空间换时间

上面的做法在LC上会TLE， 因此我们需要换一种思路，既然超时了，我们是否可以从空间换时间的角度思考呢？我们可以分别从头尾遍历，建立两个subArraySub的数组l和r。 其实这个不难想到，很多题目都用到了这个技巧。

具体做法：

- 一层遍历， 建立l数组，l[i]表示从左边开始的以arr[i]结尾的subArraySum的最大值
- 一层遍历， 建立r数组，r[i]表示从右边开始的以arr[i]结尾的subArraySum的最大值
- 一层遍历， 计算 l[i - 1] + r[i + 1] 的最大值
  > l[i - 1] + r[i + 1]的含义就是删除arr[i]的子数组最大值

- 上面的这个步骤得到了删除一个的子数组最大值， 不删除的只需要在上面循环顺便计算一下即可

```
class Solution:
    def maximumSum(self, arr: List[int]) -> int:
        n = len(arr)
        l = [arr[0]] * n
        r = [arr[n - 1]] * n
        if n == 1:
            return arr[0]
        res = arr[0]
        for i in range(1, n):
            l[i] = max(l[i - 1] + arr[i], arr[i])
            res = max(res, l[i])
        for i in range(n - 2, -1, -1):
            r[i] = max(r[i + 1] + arr[i], arr[i])
            res = max(res, r[i])
        for i in range(1, n - 1):
            res = max(res, l[i - 1] + r[i + 1])

        return res
```

动态规划

上面的算法虽然时间上有所改善，但是正如标题所说，空间复杂度是O(n),有没有办法改进呢？答案是使用动态规划。

具体过程：

- 定义max0，表示以arr[i]结尾且一个都不漏的最大子数组和
- 定义max1，表示以arr[i]或者arr[i - 1]结尾，可以漏一个的最大子数组和
- 遍历数组，更新max1和max0（注意先更新max1，因为max1用到了上一个max0）
- 其中 `max1 = max(max1 + arr[i], max0)` , 即删除arr[i - 1]或者删除arr[i]
- 其中 `max0 = max(max0 + arr[i], arr[i])` ， 一个都不删除

```python
#
# @lc app=leetcode.cn id=1186 lang=python3
#
# [1186] 删除一次得到子数组最大和
#

# @lc code=start


class Solution:
    def maximumSum(self, arr: List[int]) -> int:
        # DP
        max0 = arr[0]
        max1 = arr[0]
        res = arr[0]
        n = len(arr)
        if n == 1:
            return max0

        for i in range(1, n):
            # 先更新max1，再更新max0，因为max1用到了上一个max0
            max1 = max(max1 + arr[i], max0)
            max0 = max(max0 + arr[i], arr[i])
            res = max(res, max0, max1)
        return res


# @lc code=end
```

关键点解析

- 空间换时间
- 头尾双数组
- 动态规划

相关题目

- 42.trapping-rain-water

题目地址

https://leetcode-cn.com/problems/longest-arithmetic-subsequence-of-given-difference/

题目描述

```
给你一个整数数组 arr 和一个整数 difference，请你找出 arr 中所有相令



示例 1：

输入：arr = [1,2,3,4], difference = 1
输出：4
解释：最长的等差子序列是 [1,2,3,4]。
示例 2：

输入：arr = [1,3,5,7], difference = 1
输出：1
解释：最长的等差子序列是任意单个元素。
示例 3：

输入：arr = [1,5,7,8,5,3,4,2,1], difference = -2
输出：4
解释：最长的等差子序列是 [7,5,3,1]。


提示：

1 <= arr.length <= 10^5
-10^4 <= arr[i], difference <= 10^4
```

前置知识

- 数组
- 动态规划

公司

- 腾讯

思路

最直观的思路是双层循环，我们暴力的枚举出以每一个元素为开始元素，以最后元素结尾的的所有情况。很明显这是所有的情况，这就是暴力法的精髓， 很明显这种解法会TLE（超时），不过我们先来看一下代码，顺着这个思维继续思考。

暴力法

```
def longestSubsequence(self, arr: List[int], difference:
    n = len(arr)
    res = 1
    for i in range(n):
        count = 1
        for j in range(i + 1, n):
            if arr[i] + difference * count == arr[j]:
                count += 1

            if count > res:
                res = count

    return res
```

动态规划

上面的时间复杂度是O(n^2)， 有没有办法降低到O(n)呢？很容易想到的是空间换时间的解决方案。

我的想法是将 `以每一个元素结尾的最长等差子序列的长度` 统统存起来，即 `dp[num] = maxLen` 这样我们遍历到一个新的元素的时候，就去之前的存储中去找 `dp[num - difference]` ，如果找到了，就更新当前的 `dp[num] = dp[num - difference] + 1` ，否则就是不进行操作（还是默认值1）。

这种空间换时间的做法的时间和空间复杂度都是O(n)。

关键点解析

- 将 `以每一个元素结尾的最长等差子序列的长度` 统统存起来

代码

```python
#
# @lc app=leetcode.cn id=1218 lang=python3
#
# [1218] 最长定差子序列
#

# @lc code=start


class Solution:

    # 动态规划
    def longestSubsequence(self, arr: List[int], difference
        n = len(arr)
        res = 1
        dp = {}
        for num in arr:
            dp[num] = 1
            if num - difference in dp:
                dp[num] = dp[num - difference] + 1

        return max(dp.values())

# @lc code=end
```

相关题目

题目地址（1227. 飞机座位分配概率）

https://leetcode-cn.com/problems/airplane-seat-assignment-probability/description/

题目描述

```
有 n 位乘客即将登机，飞机正好有 n 个座位。第一位乘客的票丢了，他随便选

剩下的乘客将会：

如果他们自己的座位还空着，就坐到自己的座位上，

当他们自己的座位被占用时，随机选择其他座位
第 n 位乘客坐在自己的座位上的概率是多少？



示例 1：

输入：n = 1
输出：1.00000
解释：第一个人只会坐在自己的位置上。
示例 2：

输入： n = 2
输出： 0.50000
解释：在第一个人选好座位坐下后，第二个人坐在自己的座位上的概率是 0.5。


提示：

1 <= n <= 10^5
```

前置知识

- 记忆化搜索
- 动态规划

暴力递归

这是一道 LeetCode 为数不多的概率题，我们来看下。

公司

- 字节

思路

我们定义原问题为 f(n)。对于第一个人来说，他有 n 中选择，就是分别选择 n 个座位中的一个。由于选择每个位置的概率是相同的，那么选择每个位置的概率应该都是 1 / n。

我们分三种情况来讨论：

- 如果第一个人选择了第一个人的位置（也就是选择了自己的位置），那么剩下的人按照票上的座位做就好了，这种情况第 n 个人一定能做到自己的位置
- 如果第一个人选择了第 n 个人的位置，那么第 n 个人肯定坐不到自己的位置。
- 如果第一个人选择了第 i (1 < i < n)个人的位置，那么第 i 个人就相当于变成了“票丢的人”，此时问题转化为 f(n - i + 1)。

此时的问题转化关系如图：

（红色表示票丢的人）

整个过程分析：

代码

代码支持 Python3:

Python3 Code:

```
class Solution:
    def nthPersonGetsNthSeat(self, n: int) -> float:
        if n == 1:
            return 1
        if n == 2:
            return 0.5
        res = 1 / n
        for i in range(2, n):
            res += self.nthPersonGetsNthSeat(n - i + 1) * 
        return res
```

上述代码会栈溢出。

暴力递归 + hashtable

思路

我们考虑使用记忆化递归来减少重复计算，虽然这种做法可以减少运行时间，但是对减少递归深度没有帮助。还是会栈溢出。

代码

代码支持 Python3:

Python3 Code:

```
class Solution:
    seen = {}

    def nthPersonGetsNthSeat(self, n: int) -> float:
        if n == 1:
            return 1
        if n == 2:
            return 0.5
        if n in self.seen:
            return self.seen[n]
        res = 1 / n
        for i in range(2, n):
            res += self.nthPersonGetsNthSeat(n - i + 1) * 
        self.seen[n] = res
        return res
```

动态规划

思路

上面做法会栈溢出。其实我们根本不需要运行就应该能判断出栈溢出，题目已经给了数据规模是 1 <= n <= 10 ** 5。 这个量级不管什么语言，除非使用尾递归，不然一般都会栈溢出，具体栈深度大家可以查阅相关资料。

既然是栈溢出，那么我们考虑使用迭代来完成。 很容易想到使用动态规划来完成。其实递归都写出来，写一个朴素版的动态规划也难不到哪去，毕竟动态规划就是记录子问题，并建立子问题之间映射而已，这和递归并无本质区别。

代码

代码支持 Python3:

Python3 Code:

```
class Solution:
    def nthPersonGetsNthSeat(self, n: int) -> float:
        if n == 1:
            return 1
        if n == 2:
            return 0.5

        dp = [1, .5] * n

        for i in range(2, n):
            dp[i] = 1 / n
            for j in range(2, i):
                dp[i] += dp[i - j + 1] * 1 / n
        return dp[-1]
```

这种思路的代码超时了，并且仅仅执行了 35/100 testcase 就超时了。

数学分析

思路

我们还需要进一步优化时间复杂度，我们需要思考是否可以在线形的时间内完成。

我们继续前面的思路进行分析, 不难得出，我们不妨称其为等式 1：

```
f(n)
= 1/n + 0 + 1/n * (f(n-1) + f(n-2) + ... + f(2))
= 1/n * (f(n-1) + f(n-2) + ... + f(2) + 1)
= 1/n * (f(n-1) + f(n-2) + ... + f(2) + f(1))
```

似乎更复杂了？没关系，我们继续往下看，我们看下 f(n - 1)，我们不妨称其为等式 2。

```
f(n-1) = 1/(n-1) * (f(n-2) + f(n-3) + ... + f(1))
```

我们将等式 1 和等式 2 两边分别同时乘以 n 和 n - 1

```
n * f(n) = f(n-1) + f(n-2) + f(n-3) + ... + f(1)
(n-1) * f(n-1) = f(n-2) + f(n-3) + ... + f(1)
```

我们将两者相减：

```
n * f(n) - (n-1)*f(n-1) = f(n-1)
```

我们继续将 (n-1)*f(n-1) 移到等式右边，得到：

```
n * f(n) = n * f(n-1)
```

也就是说:

```
f(n) = f(n - 1)
```

当然前提是 n 大于 2。

既然如此，我们就可以减少一层循环， 我们用这个思路来优化一下上面的 dp 解法。这种解法终于可以 AC 了。

代码

代码支持 Python3:

Python3 Code:

```python
class Solution:
    def nthPersonGetsNthSeat(self, n: int) -> float:
        if n == 1:
            return 1
        if n == 2:
            return 0.5

        dp = [1, .5] * n

        for i in range(2, n):
            dp[i] = 1/n+(n-2)/n * dp[n-1]
        return dp[-1]
```

优化数学分析

思路

上面我们通过数学分析，得出了当 n 大于 2 时：

```
f(n) = f(n - 1)
```

那么是不是意味着我们随便求出一个 n 就好了？ 比如我们求出 n = 2 的时候的值，是不是就知道 n 为任意数的值了。 我们不难想出 n = 2 时候，概率是 0.5，因此只要 n 大于 1 就是 0.5 概率，否则就是 1 概率。

代码

代码支持 Python3:

Python3 Code:

```python
class Solution:
    def nthPersonGetsNthSeat(self, n: int) -> float:
        return 1 if n == 1 else .5
```

关键点

- 概率分析
- 数学推导
- 动态规划
- 递归 + mapper
- 栈限制大小
- 尾递归

题目地址（1261. 在受污染的二叉树中查找元素）

https://leetcode-cn.com/problems/find-elements-in-a-contaminated-binary-tree/submissions/

题目描述

```
给出一个满足下述规则的二叉树：

root.val == 0
如果 treeNode.val == x 且 treeNode.left != null，那么 treeNod
如果 treeNode.val == x 且 treeNode.right != null，那么 treeNo
现在这个二叉树受到「污染」，所有的 treeNode.val 都变成了 -1。

请你先还原二叉树，然后实现 FindElements 类：

FindElements(TreeNode* root) 用受污染的二叉树初始化对象，你需要先
bool find(int target) 判断目标值 target 是否存在于还原后的二叉树


示例 1：

![](https://tva1.sinaimg.cn/large/007S8ZIlly1ghlua6htirj308

输入：
["FindElements","find","find"]
[[[-1,null,-1]],[1],[2]]
输出：
[null,false,true]
解释：
FindElements findElements = new FindElements([-1,null,-1]);
findElements.find(1); // return False
findElements.find(2); // return True
示例 2：

![](https://tva1.sinaimg.cn/large/007S8ZIlly1ghlua84ataj30b

输入：
["FindElements","find","find","find"]
[[[-1,-1,-1,-1,-1]],[1],[3],[5]]
输出：
[null,true,true,false]
解释：
FindElements findElements = new FindElements([-1,-1,-1,-1,-
findElements.find(1); // return True
findElements.find(3); // return True
findElements.find(5); // return False
示例 3：

![](https://tva1.sinaimg.cn/large/007S8ZIlly1ghlua8rj84j308

输入：
["FindElements","find","find","find","find"]
[[[-1,null,-1,-1,null,-1]],[2],[3],[4],[5]]
输出：
```

```
[null,true,false,false,true]
解释:
FindElements findElements = new FindElements([-1,null,-1,-1
findElements.find(2); // return True
findElements.find(3); // return False
findElements.find(4); // return False
findElements.find(5); // return True


提示:

TreeNode.val == -1
二叉树的高度不超过 20
节点的总数在 [1, 10^4] 之间
调用 find() 的总次数在 [1, 10^4] 之间
0 <= target <= 10^6
```

前置知识

- 二进制

暴力法

公司

- 暂无

思路

最简单想法就是递归建立树，然后 find 的时候递归查找即可，代码也很简单。

代码

Pythpn Code:

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class FindElements:
    node = None
    def __init__(self, root: TreeNode):
        def recover(node):
            if not node:
                return node;
            if node.left:
                node.left.val =  2 * node.val + 1
            if node.right:
                 node.right.val = 2 * node.val + 2
            recover(node.left)
            recover(node.right)
            return node
        root.val = 0
        self.node = recover(root)


    def find(self, target: int) -> bool:
        def findInTree(node, target):
            if not node:
                return False
            if node.val == target:
                return True
            return findInTree(node.left, target) or findIn
        return findInTree(self.node, target)




# Your FindElements object will be instantiated and called
# obj = FindElements(root)
# param_1 = obj.find(target)
```

上述代码会超时，我们来考虑优化。

空间换时间

思路

上述代码会超时，我们考虑使用空间换时间。 建立树的时候，我们将所有值存到一个集合中去。当需要 find 的时候，我们直接查找 set 即可，时间复杂度 O(1)。

代码

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class FindElements:
    def __init__(self, root: TreeNode):
        # set 不能放在init外侧。 因为测试用例之间不会销毁FindElem
        self.seen = set()
        def recover(node):
            if not node:
                return node;
            if node.left:
                node.left.val =  2 * node.val + 1
                self.seen.add(node.left.val)
            if node.right:
                 node.right.val = 2 * node.val + 2
                 self.seen.add(node.right.val)
            recover(node.left)
            recover(node.right)
            return node
        root.val = 0
        self.seen.add(0)
        self.node = recover(root)


    def find(self, target: int) -> bool:
        return target in self.seen




# Your FindElements object will be instantiated and called
# obj = FindElements(root)
# param_1 = obj.find(target)
```

这种解法可以 AC，但是在数据量非常大的时候，可能 MLE，我们继续考虑优化。

二进制法

思路

这是一种非常巧妙的做法。

如果我们把树中的数全部加 1 会怎么样?

（图参考 https://leetcode.com/problems/find-elements-in-a-contaminated-binary-tree/discuss/431229/Python-Special-Way-for-find()-without-HashSet-O(1)-Space-O(logn)-Time）

仔细观察发现，每一行的左右子树分别有不同的前缀：

Ok，那么算法就来了。为了便于理解，我们来举个具体的例子，比如 target 是 9，我们首先将其加 1，二进制表示就是 1010。不考虑第一位，就是 010，我们只要：

- 0 向左 👈
- 1 向右 👉
-
  - 0 向左 👈

就可以找到 9 了。

> 0 表示向左， 1 表示向右

代码

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class FindElements:
    node = None
    def __init__(self, root: TreeNode):
        def recover(node):
            if not node:
                return node;
            if node.left:
                node.left.val =  2 * node.val + 1
            if node.right:
                 node.right.val = 2 * node.val + 2
            recover(node.left)
            recover(node.right)
            return node
        root.val = 0
        self.node = recover(root)


    def find(self, target: int) -> bool:
        node = self.node
        for bit in bin(target+1)[3:]:
            node = node and (node.left, node.right)[int(bit
        return bool(node)




# Your FindElements object will be instantiated and called
# obj = FindElements(root)
# param_1 = obj.find(target)
```

关键点解析

- 空间换时间
- 二进制思维
- 将 target + 1

题目地址（1262. 可被三整除的最大和）

https://leetcode-cn.com/problems/greatest-sum-divisible-by-three/description/

题目描述

```
给你一个整数数组 nums，请你找出并返回能被三整除的元素最大和。



示例 1：

输入：nums = [3,6,5,1,8]
输出：18
解释：选出数字 3, 6, 1 和 8，它们的和是 18（可被 3 整除的最大和）。
示例 2：

输入：nums = [4]
输出：0
解释：4 不能被 3 整除，所以无法选出数字，返回 0。
示例 3：

输入：nums = [1,2,3,4,4]
输出：12
解释：选出数字 1, 3, 4 以及 4，它们的和是 12（可被 3 整除的最大和）



提示：

1 <= nums.length <= 4 * 10^4
1 <= nums[i] <= 10^4
```

前置知识

- 数组
- 回溯法
- 排序

暴力法

公司

- 字节
- 网易有道

思路

一种方式是找出所有的能够被 3 整除的子集，然后挑选出和最大的。由于我们选出了所有的子集，那么时间复杂度就是 $O(2^N)$ ， 毫无疑问会超时。这里我们使用回溯法找子集，如果不清楚回溯法，可以参考我之前的题解，很多题目都用到了，比如78.subsets。

更多回溯题目，可以访问上方链接查看（可以使用一套模板搞定）：

- 39.combination-sum
- 40.combination-sum-ii
- 46.permutations
- 47.permutations-ii
- 90.subsets-ii
- 113.path-sum-ii
- 131.palindrome-partitioning

代码

```
class Solution:
    def maxSumDivThree(self, nums: List[int]) -> int:
        self.res = 0
        def backtrack(temp, start):
            total = sum(temp)
            if total % 3 == 0:
                self.res = max(self.res, total)
            for i in range(start, len(nums)):
                temp.append(nums[i])
                backtrack(temp, i + 1)
                temp.pop(-1)


        backtrack([], 0)

        return self.res
```

减法 + 排序

减法的核心思想是，我们求出总和。如果总和不满足题意，我们尝试减去最小的数，使之满足题意。

思路

这种算法的思想，具体来说就是：

- 我们将所有的数字加起来，我们不妨设为 total
- total 除以 3，得到一个余数 mod， mod 可能值有 0，1，2.
- 同时我们建立两个数组，一个是余数为 1 的数组 one，一个是余数为 2 的数组 two
- 如果 mod 为 0，我们直接返回即可。
- 如果 mod 为 1，我们可以减去 one 数组中最小的一个（如果有的话），或者减去两个 two 数组中最小的（如果有的话），究竟减去谁取决谁更小。
- 如果 mod 为 2，我们可以减去 two 数组中最小的一个（如果有的话），或者减去两个 one 数组中最小的（如果有的话），究竟减去谁取决谁更小。

由于我们需要取 one 和 two 中最小的一个或者两个，因此对数组 one 和 two 进行排序是可行的，如果基于排序的话，时间复杂度大致为 $O(NlogN)$，这种算法可以通过。

以题目中的例 1 为例：

nums
3
6
5
1
8
one
1
total = 3 + 6 + 5 + 1 + 8 = 23
mod = 23 % 3 = 2
two
5
8
one
1
two
5
8
res = 23 - 5 = 18
1262. 可被三整除的最大和 @lucifer

以题目中的例 2 为例：

nums
1
2
3
4
4
one
1
4
4
total = 1 + 2 + 3 + 4 + 4 = 14
mod = 14 % 3 = 2
two
2
one
1
4
4
two
2
res = 14 - min(2, 1 + 4) = 12
1262. 可被三整除的最大和 @lucifer

代码

```python
class Solution:
    def maxSumDivThree(self, nums: List[int]) -> int:
        one = []
        two = []
        total = 0

        for num in nums:
            total += num
            if num % 3 == 1:
                one.append(num)
            if num % 3 == 2:
                two.append(num)
        one.sort()
        two.sort()
        if total % 3 == 0:
            return total
        elif total % 3 == 1 and one:
            if len(two) >= 2 and one[0] > two[0] + two[1]:
                return total - two[0] - two[1]
            return total - one[0]
        elif total % 3 == 2 and two:
            if len(one) >= 2 and two[0] > one[0] + one[1]:
                return total - one[0] - one[1]
            return total - two[0]
        return 0
```

减法 + 非排序

思路

上面的解法使用到了排序。 我们其实观察发现，我们只是用到了 one 和 two 的最小的两个数。因此我们完全可以在线形的时间和常数的空间完成这个算法。我们只需要分别记录 one 和 two 的最小值和次小值即可，在这里，我使用了两个长度为 2 的数组来表示，第一项是最小值，第二项是次小值。

代码

```python
class Solution:
    def maxSumDivThree(self, nums: List[int]) -> int:
        one = [float('inf')] * 2
        two = [float('inf')] * 2
        total = 0

        for num in nums:
            total += num
            if num % 3 == 1:
                if num < one[0]:
                    t = one[0]
                    one[0] = num
                    one[1] = t
                elif num < one[1]:
                    one[1] = num
            if num % 3 == 2:
                if num < two[0]:
                    t = two[0]
                    two[0] = num
                    two[1] = t
                elif num < two[1]:
                    two[1] = num
        if total % 3 == 0:
            return total
        elif total % 3 == 1 and one:
            if len(two) >= 2 and one[0] > two[0] + two[1]:
                return total - two[0] - two[1]
            return total - one[0]
        elif total % 3 == 2 and two:
            if len(one) >= 2 and two[0] > one[0] + one[1]:
                return total - one[0] - one[1]
            return total - two[0]
        return 0
```

有限状态机

思路

我在数据结构与算法在前端领域的应用 - 第二篇 中讲到了有限状态机。

状态机表示若干个状态以及在这些状态之间的转移和动作等行为的数学模型。通俗的描述状态机就是定义了一套状态变更的流程：状态机包含一个状态集合，定义当状态机处于某一个状态的时候它所能接收的事件以及可执行的行为，执行完成后，状态机所处的状态。

状态机使用非常广泛，比如正则表达式的引擎，编译器的词法和语法分析，网络协议，企业应用等很多领域都会用到。

拿本题中来说，我们从左到右扫描数组的过程，将会不断改变状态机的状态。

我们使用 state 数组来表示本题的状态：

- state[0] 表示 mod 为 0 的 最大和
- state[1] 表示 mod 为 1 的 最大和
- state[2] 表示 mod 为 1 的 最大和

我们的状态转移方程就会很容易。说到状态转移方程，你可能会想到动态规划。没错！这种思路可以直接翻译成动态规划，算法完全一样。如果你看过我上面提到的文章，那么状态转移方程对你来说就会很容易。如果你不清楚，那么请往下看：

- 我们从左往右不断读取数字，我们不妨设这个数字为 num。
- 如果 num % 3 为 0。 那么我们的 state[0], state[1], state[2] 可以直接加上 num（题目限定了 num 为非负）， 因为任何数字加上 3 的倍数之后，mod 3 的值是不变的。
- 如果 num % 3 为 1。 我们知道 state[2] + num 会变成一个能被三整除的数，但是这个数字不一定比当前的 state[0]大。 代码表示就是 `max(state[2] + num, state[0])` 。同理 state[1] 和 state[2] 的转移逻辑类似。

- 同理 num % 3 为 2 也是类似的逻辑。
- 最后我们返回 state[0]即可。

代码

```
class Solution:
    def maxSumDivThree(self, nums: List[int]) -> int:
        state = [0, float('-inf'), float('-inf')]

        for num in nums:
            if num % 3 == 0:
                state = [state[0] + num, state[1] + num, st
            if num % 3 == 1:
                a = max(state[2] + num, state[0])
                b = max(state[0] + num, state[1])
                c = max(state[1] + num, state[2])
                state = [a, b, c]
            if num % 3 == 2:
                a = max(state[1] + num, state[0])
                b = max(state[2] + num, state[1])
                c = max(state[0] + num, state[2])
                state = [a, b, c]
        return state[0]
```

当然这个代码还可以简化：

```
class Solution:
    def maxSumDivThree(self, nums: List[int]) -> int:
        state = [0, float('-inf'), float('-inf')]

        for num in nums:
            temp = [0] * 3
            for i in range(3):
                temp[(i + num) % 3] = max(state[(i + num) %
            state = temp

        return state[0]
```

关键点解析

- 贪婪法
- 状态机
- 数学分析

扩展

实际上，我们可以采取加法（贪婪策略），感兴趣的可以试一下。

另外如果题目改成了 `请你找出并返回能被x整除的元素最大和` ，你只需要将我的解法中的 3 改成 x 即可。

题目地址（1297. 子串的最大出现次数）

https://leetcode-cn.com/problems/maximum-number-of-occurrences-of-a-substring

题目描述

```
给你一个字符串 s ，请你返回满足以下条件且出现次数最大的 任意 子串的出

子串中不同字母的数目必须小于等于 maxLetters 。
子串的长度必须大于等于 minSize 且小于等于 maxSize 。


示例 1：

输入：s = "aababcaab", maxLetters = 2, minSize = 3, maxSize
输出：2
解释：子串 "aab" 在原字符串中出现了 2 次。
它满足所有的要求：2 个不同的字母，长度为 3 （在 minSize 和 maxSize
示例 2：

输入：s = "aaaa", maxLetters = 1, minSize = 3, maxSize = 3
输出：2
解释：子串 "aaa" 在原字符串中出现了 2 次，且它们有重叠部分。
示例 3：

输入：s = "aabcabcab", maxLetters = 2, minSize = 2, maxSize
输出：3
示例 4：

输入：s = "abcde", maxLetters = 2, minSize = 3, maxSize = 3
输出：0


提示：

1 <= s.length <= 10^5
1 <= maxLetters <= 26
1 <= minSize <= maxSize <= min(26, s.length)
s 只包含小写英文字母。
```

前置知识

- 字符串
- 滑动窗口

暴力法

题目给的数据量不是很大，为 1 <= maxLetters <= 26，我们试一下暴力法。

公司

- 字节

思路

暴力法如下：

- 先找出所有满足长度大于等于 minSize 且小于等于 maxSize 的所有子串。（平方的复杂度）
- 对于 maxLetter 满足题意的子串，我们统计其出现次数。时间复杂度为 O(k),其中 k 为子串长度
- 返回最大的出现次数

代码

Pythpn Code:

```
class Solution:
    def maxFreq(self, s: str, maxLetters: int, minSize: int
        n = len(s)
        letters = set()
        cnts = dict()
        res = 0
        for i in range(n - minSize + 1):
            length = minSize
            while i + length <= n and length <= maxSize:
                t = s[i:i + length]
                for c in t:
                    if len(letters) > maxLetters:
                        break
                    letters.add(c)
                if len(letters) <= maxLetters:
                    cnts[t] = cnts.get(t, 0) + 1
                    res = max(res, cnts[t])
                letters.clear()
                length += 1
        return res
```

上述代码会超时。我们来利用剪枝来优化。

剪枝

思路

还是暴力法的思路，不过我们在此基础上进行一些优化。首先我们需要仔细阅读题目，如果你足够细心或者足够有经验，可能会发现其实题目中 maxSize 没有任何用处，属于干扰信息。

也就是说我们没有必要统计 `长度大于等于 minSize 且小于等于 maxSize 的所有子串` ，而是统计长度为 minSize 的所有字串即可。原因是，如果一个大于 minSize 长度的字串若是满足条件，那么该子串其中必定有至少一个长度为 minSize 的字串满足条件。因此一个大于 minSize 长度的字串出现了 n 次，那么该子串其中必定有一个长度为 minSize 的子串出现了 n 次。

代码

代码支持 Python3，Java：

Python Code：

```
def maxFreq(self, s: str, maxLetters: int, minSize: int, r
        counter, res = {}, 0
        for i in range(0, len(s) - minSize + 1):
            sub = s[i : i + minSize]
            if len(set(sub)) <= maxLetters:
                counter[sub] = counter.get(sub, 0) + 1
                res = max(res, counter[sub])
        return res;

# @lc code=end
```

Java Code：

```
public int maxFreq(String s, int maxLetters, int minSize,
    Map<String, Integer> counter = new HashMap<>();
    int res = 0;
    for (int i = 0; i < s.length() - minSize + 1; i++) {
        String substr = s.substring(i, i + minSize);
        if (checkNum(substr, maxLetters)) {
            int newVal = counter.getOrDefault(substr, 0) +
            counter.put(substr, newVal);
            res = Math.max(res, newVal);
        }
    }
    return res;
}
public boolean checkNum(String substr, int maxLetters) {
    Set<Character> set = new HashSet<>();
    for (int i = 0; i < substr.length(); i++)
        set.add(substr.charAt(i));
    return set.size() <= maxLetters;
}
```

关键点解析

- 滑动窗口
- 识别题目干扰信息
- 看题目限制条件，对于本题有用的信息是 `1 <= maxLetters <= 26`

扩展

我们也可以使用滑动窗口来解决，感兴趣的可以试试看。

题目地址（1310. 子数组异或查询）

https://leetcode-cn.com/problems/xor-queries-of-a-subarray

题目描述

```
有一个正整数数组 arr，现给你一个对应的查询数组 queries，其中 queri

对于每个查询 i，请你计算从 Li 到 Ri 的 XOR 值（即 arr[Li] xor ar

并返回一个包含给定查询 queries 所有结果的数组。



示例 1：

输入：arr = [1,3,4,8], queries = [[0,1],[1,2],[0,3],[3,3]]
输出：[2,7,14,8]
解释：
数组中元素的二进制表示形式是：
1 = 0001
3 = 0011
4 = 0100
8 = 1000
查询的 XOR 值为：
[0,1] = 1 xor 3 = 2
[1,2] = 3 xor 4 = 7
[0,3] = 1 xor 3 xor 4 xor 8 = 14
[3,3] = 8
示例 2：

输入：arr = [4,8,2,10], queries = [[2,3],[1,3],[0,0],[0,3]]
输出：[8,0,4,4]


提示：

1 <= arr.length <= 3 * 10^4
1 <= arr[i] <= 10^9
1 <= queries.length <= 3 * 10^4
queries[i].length == 2
0 <= queries[i][0] <= queries[i][1] < arr.length
```

前置知识

- 前缀和

暴力法

思路

最直观的思路是双层循环即可，果不其然超时了。

代码

```
class Solution:
    def xorQueries(self, arr: List[int], queries: List[List
         res = []
        for (L, R) in queries:
            i = L
            xor = 0
            while i <= R:
                xor ^= arr[i]
                i += 1
            res.append(xor)
        return res
```

前缀表达式

公司

- 暂无

思路

比较常见的是前缀和，这个概念其实很容易理解，即一个数组中，第 n 位存储的是数组前 n 个数字的和。

对 [1,2,3,4,5,6] 来说，其前缀和可以是 pre=[1,3,6,10,15,21]。我们可以使用公式 pre[i]=pre[$i-1$]+nums[i]得到每一位前缀和的值，从而通过前缀和进行相应的计算和解题。其实前缀和的概念很简单，但困难的是如何在题目中使用前缀和以及如何使用前缀和的关系来进行解题。

这道题是前缀对前缀异或，我们利用了异或的性质 `x ^ y ^ x = y` 。

代码

代码支持 Python3，Java，C++：

Python Code：

```
#
# @lc app=leetcode.cn id=1218 lang=python3
#
# [1218] 最长定差子序列
#

# @lc code=start


class Solution:
    def xorQueries(self, arr: List[int], queries: List[List
        pre = [0]
        res = []
        for i in range(len(arr)):
            pre.append(pre[i] ^ arr[i])
        for (L, R) in queries:
            res.append(pre[L] ^ pre[R + 1])
        return res

# @lc code=end
```

Java Code：

```
  public int[] xorQueries(int[] arr, int[][] queries) {

        int[] preXor = new int[arr.length];
        preXor[0] = 0;

        for (int i = 1; i < arr.length; i++)
            preXor[i] = preXor[i - 1] ^ arr[i - 1];

        int[] res = new int[queries.length];

        for (int i = 0; i < queries.length; i++) {

            int left = queries[i][0], right = queries[i][1
            res[i] = arr[right] ^ preXor[right] ^ preXor[le
        }

        return res;
    }
```

C++ Code:

```
class Solution {
public:
    vector<int> xorQueries(vector<int>& arr, vector<vector<
        vector<int>res;
        for(int i=1; i<arr.size(); ++i){
            arr[i]^=arr[i-1];
        }
        for(vector<int>temp :queries){
            if(temp[0]==0){
                res.push_back(arr[temp[1]]);
            }
            else{
                res.push_back(arr[temp[0]-1]^arr[temp[1]]);
            }
        }
        return res;
    }
};
```

关键点解析

- 异或的性质 x ^ y ^ x = y
- 前缀表达式

相关题目

- 303. 区域和检索 - 数组不可变

303. 区域和检索 - 数组不可变

难度 简单 124 收藏 分享 切换为英文 关注

描述 评论 题解 历史 Python3 智能模式

给定一个整数数组 *nums*，求出数组从索引 *i* 到 *j*（*i* ≤ *j*）范围内元素的总和，包含 *i*，*j* 两点。

示例：

```
给定 nums = [-2, 0, 3, -5, 2, -1]，求和函数为
sumRange()

sumRange(0, 2) -> 1
sumRange(2, 5) -> -1
sumRange(0, 5) -> -3
```

说明：

1. 你可以假设数组不可变。
2. 会多次调用 *sumRange* 方法。

在真实的面试中遇到过这道题？

是 否

贡献者

```python
#
# @lc app=leetcode.cn id=303 lang=python3
#
# [303] 区域和检索 - 数组不可变
#

# @lc code=start


class NumArray:

    def __init__(self, nums: List[int]):
        self.pre = [0]
        for i in range(len(nums)):
            self.pre.append(self.pre[i] + nums[i])

    def sumRange(self, i: int, j: int) -> int:
        return self.pre[j + 1] - self.pre[i]

        # Your NumArray object will be instantiated and called as such:
        # obj = NumArray(nums)
        # param_1 = obj.sumRange(i,j)
        # @lc code=end
```

- 1186.删除一次得到子数组最大和

题目地址（1334. 阈值距离内邻居最少的城市）

https://leetcode-cn.com/problems/find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance/

题目描述

```
有 n 个城市，按从 0 到 n-1 编号。给你一个边数组 edges，其中 edges[

返回能通过某些路径到达其他城市数目最少、且路径距离 最大 为 distanceT

注意，连接城市 i 和 j 的路径的距离等于沿该路径的所有边的权重之和。



示例 1：
```

```
输入：n = 4, edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], dist
输出：3
解释：城市分布图如上。
每个城市阈值距离 distanceThreshold = 4 内的邻居城市分别是：
城市 0 -> [城市 1, 城市 2]
城市 1 -> [城市 0, 城市 2, 城市 3]
城市 2 -> [城市 0, 城市 1, 城市 3]
城市 3 -> [城市 1, 城市 2]
城市 0 和 3 在阈值距离 4 以内都有 2 个邻居城市，但是我们必须返回城市
示例 2：
```

```
输入：n = 5, edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1
输出：0
解释：城市分布图如上。
每个城市阈值距离 distanceThreshold = 2 内的邻居城市分别是：
城市 0 -> [城市 1]
城市 1 -> [城市 0, 城市 4]
城市 2 -> [城市 3, 城市 4]
城市 3 -> [城市 2, 城市 4]
城市 4 -> [城市 1, 城市 2, 城市 3]
城市 0 在阈值距离 4 以内只有 1 个邻居城市。


提示：

2 <= n <= 100
1 <= edges.length <= n * (n - 1) / 2
edges[i].length == 3
0 <= fromi < toi < n
1 <= weighti, distanceThreshold <= 10^4
所有 (fromi, toi) 都是不同的。
```

前置知识

- 动态规划
- Floyd-Warshall

公司

- 暂无

思路

这道题的本质就是：

1. 在一个无向图中寻找每两个城镇的最小距离，我们使用 Floyd-Warshall 算法（英语：Floyd-Warshall algorithm），中文亦称弗洛伊德算法，是解决任意两点间的最短路径的一种算法。
2. 筛选最小距离不大于 distanceThreshold 的城镇。
3. 统计每个城镇，其满足条件的城镇有多少个
4. 我们找出最少的即可

Floyd-Warshall 算法的时间复杂度和空间复杂度都是$O(N^3)$, 而空间复杂度可以优化到$O(N^2)$。Floyd-Warshall 的基本思想是对于每两个点之间的最小距离，要么经过中间节点 k，要么不经过，我们取两者的最小值，这是一种动态规划思想，详细的解法可以参考Floyd-Warshall 算法 (wikipedia)

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def findTheCity(self, n: int, edges: List[List[int]], d
        # 构建dist矩阵
        dist = [[float('inf')] * n for _ in  range(n)]
        for i, j, w in edges:
            dist[i][j] = w
            dist[j][i] = w
        for i in range(n):
            dist[i][i] = 0
        for k in range(n):
            for i in range(n):
                for j in range(n):
                    dist[i][j] = min(dist[i][j], dist[i][k]

        # 过滤
        res = 0
        minCnt = float('inf')
        for i in range(n):
            cnt = 0
            for d in dist[i]:
                if d <= distanceThreshold:
                    cnt += 1
            if cnt <= minCnt:
                minCnt = cnt
                res = i
        return res
```

关键点解析

- Floyd-Warshall 算法
- 你可以将本文给的 Floyd-Warshall 算法当成一种解题模板使用

题目地址（1371. 每个元音包含偶数次的最长子字符串）

https://leetcode-cn.com/problems/find-the-longest-substring-containing-vowels-in-even-counts/

题目描述

```
给你一个字符串 s ，请你返回满足以下条件的最长子字符串的长度：每个元音字



示例 1：

输入：s = "eleetminicoworoep"
输出：13
解释：最长子字符串是 "leetminicowor" ，它包含 e，i，o 各 2 个，以
示例 2：

输入：s = "leetcodeisgreat"
输出：5
解释：最长子字符串是 "leetc" ，其中包含 2 个 e 。
示例 3：

输入：s = "bcbcbc"
输出：6
解释：这个示例中，字符串 "bcbcbc" 本身就是最长的，因为所有的元音 a，


提示：

1 <= s.length <= 5 x 10^5
s 只包含小写英文字母。
```

前置知识

- 前缀和
- 状态压缩

暴力法 + 剪枝

公司

- 暂无

思路

首先拿到这道题的时候，我想到第一反应是滑动窗口行不行。 但是很快这个想法就被我否定了，因为滑动窗口（这里是可变滑动窗口）我们需要扩张和收缩窗口大小，而这里不那么容易。因为题目要求的是奇偶性，而不是类似“元音出现最多的子串”等。

突然一下子没了思路。那就试试暴力法吧。暴力法的思路比较朴素和直观。 那就是 `双层循环找到所有子串，然后对于每一个子串，统计元音个数，如果子串的元音个数都是偶数，则更新答案，最后返回最大的满足条件的子串长度即可` 。

这里我用了一个小的 trick。枚举所有子串的时候，我是从最长的子串开始枚举的，这样我找到一个满足条件的直接返回就行了（early return），不必维护最大值。 `这样不仅减少了代码量，还提高了效率。`

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def findTheLongestSubstring(self, s: str) -> int:
        for i in range(len(s), 0, -1):
            for j in range(len(s) - i + 1):
                sub = s[j:j + i]
                has_odd_vowel = False
                for vowel in ['a', 'e', 'i', 'o', 'u']:
                    if sub.count(vowel) % 2 != 0:
                        has_odd_vowel = True
                        break
                if not has_odd_vowel: return  i
        return 0
```

复杂度分析

- 时间复杂度：双层循环找出所有子串的复杂度是$O(n^2)$，统计元音个数复杂度也是$O(n)$，因此这种算法的时间复杂度为$O(n^3)$。
- 空间复杂度：$O(1)$

前缀和 + 剪枝

思路

上面思路中 `对于每一个子串，统计元音个数` ，我们仔细观察的话，会发现有很多重复的统计。那么优化这部分的内容就可以获得更好的效率。

对于这种连续的数字问题，这里我们考虑使用前缀和来优化。

经过这种空间换时间的策略之后，我们的时间复杂度会降低到$O(n^2)$，但是相应空间复杂度会上升到$O(n)$，这种取舍在很多情况下是值得的。

代码

代码支持：Python3，Java

Python3 Code:

```
class Solution:
    i_mapper = {
        "a": 0,
        "e": 1,
        "i": 2,
        "o": 3,
        "u": 4
    }
    def check(self, s, pre, l, r):
        for i in range(5):
            if s[l] in self.i_mapper and i == self.i_mappe
            else: cnt = 0
            if (pre[r][i] - pre[l][i] + cnt) % 2 != 0: retu
        return True
    def findTheLongestSubstring(self, s: str) -> int:
        n = len(s)

        pre = [[0] * 5 for _ in range(n)]

        # pre
        for i in range(n):
            for j in range(5):
                if s[i] in self.i_mapper and self.i_mapper
                    pre[i][j] = pre[i - 1][j] + 1
                else:
                    pre[i][j] = pre[i - 1][j]
        for i in range(n - 1, -1, -1):
            for j in range(n - i):
                if self.check(s, pre, j, i + j):
                    return i + 1
        return 0
```

Java Code：

```
class Solution {
    public int findTheLongestSubstring(String s) {

        int len = s.length();

        if (len == 0)
            return 0;

        int[][] preSum = new int[len][5];
        int start = getIndex(s.charAt(0));
        if (start != -1)
            preSum[0][start]++;

        // preSum
        for (int i = 1; i < len; i++) {

            int idx = getIndex(s.charAt(i));

            for (int j = 0; j < 5; j++) {

                if (idx == j)
                    preSum[i][j] = preSum[i - 1][j] + 1;
                else
                    preSum[i][j] = preSum[i - 1][j];
            }
        }

        for (int i = len - 1; i >= 0; i--) {

            for (int j = 0; j < len - i; j++) {
                if (checkValid(preSum, s, j, i + j))
                    return i + 1;
            }
        }
        return 0;
    }


    public boolean checkValid(int[][] preSum, String s, int

        int idx = getIndex(s.charAt(left));

        for (int i = 0; i < 5; i++)
            if (((preSum[right][i] - preSum[left][i] + (idx
                return false;

        return true;
    }
```

```
    public int getIndex(char ch) {

        if (ch == 'a')
            return 0;
        else if (ch == 'e')
            return 1;
        else if (ch == 'i')
            return 2;
        else if (ch == 'o')
            return 3;
        else if (ch == 'u')
            return 4;
        else
            return -1;
    }
}
```

复杂度分析

- 时间复杂度：$O(n^2)$。
- 空间复杂度：$O(n)$

前缀和 + 状态压缩

思路

前面的前缀和思路，我们通过空间（prefix）换取时间的方式降低了时间复杂度。但是时间复杂度仍然是平方，我们是否可以继续优化呢？

实际上由于我们只关心奇偶性，并不关心每一个元音字母具体出现的次数。因此我们可以使用 `是奇数，是偶数` 两个状态来表示，由于只有两个状态，我们考虑使用位运算。

我们使用 5 位的二进制来表示以 i 结尾的字符串中包含各个元音的奇偶性，其中 0 表示偶数，1 表示奇数，并且最低位表示 a，然后依次是 e，i，o，u。比如 `10110` 则表示的是包含偶数个 a 和 o，奇数个 e，i，u，我们用变量 `cur` 来表示。

为什么用 0 表示偶数？1 表示奇数？

回答这个问题，你需要继续往下看。

其实这个解法还用到了一个性质，这个性质是小学数学知识：

- 如果两个数字奇偶性相同，那么其相减一定是偶数。
- 如果两个数字奇偶性不同，那么其相减一定是奇数。

看到这里，我们再来看上面抛出的问题 `为什么用 0 表示偶数？1 表示奇数？` 。因为这里我们打算用异或运算，而异或的性质是：

如果对两个二进制做异或，会对其每一位进行位运算，如果相同则位 0，否则位 1。这和上面的性质非常相似。上面说 `奇偶性相同则位偶数，否则为奇数` 。因此很自然地 `用 0 表示偶数？1 表示奇数` 会更加方便。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def findTheLongestSubstring(self, s: str) -> int:
        mapper = {
            "a": 1,
            "e": 2,
            "i": 4,
            "o": 8,
            "u": 16
        }
        seen = {0: -1}
        res = cur = 0

        for i in range(len(s)):
            if s[i] in mapper:
                cur ^= mapper.get(s[i])
            # 全部奇偶性都相同，相减一定都是偶数
            if cur in seen:
                res = max(res, i - seen.get(cur))
            else:
                seen[cur] = i
        return res
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(n)$

关键点解析

- 前缀和
- 状态压缩

相关题目

- 掌握前缀表达式真的可以为所欲为！

困难难度题目合集

困难难度题目从类型上说多是：

- 图
- 设计题
- 游戏场景题目
- 中等题目的 follow up

从解法上来说，多是：

- 图算法
- 动态规划
- 二分法
- DFS & BFS
- 状态压缩
- 剪枝

从逻辑上说， 要么就是非常难想到，要么就是非常难写代码。 这里我总结了几个技巧：

1. 看题目的数据范围， 看能否暴力模拟
2. 暴力枚举所有可能的算法往上套
3. 总结和记忆解题模板，减少解题压力

4. 0004.median-of-two-sorted-array

5. 0023.merge-k-sorted-lists
6. 0025.reverse-nodes-in-k-group
7. 0030.substring-with-concatenation-of-all-words
8. 0032.longest-valid-parentheses
9. 0042.trapping-rain-water🖊️
10. 0052.N-Queens-II
11. 0084.largest-rectangle-in-histogram
12. 0085.maximal-rectangle
13. 0124.binary-tree-maximum-path-sum
14. 0128.longest-consecutive-sequence
15. 0145.binary-tree-postorder-traversal
16. 0212.word-search-ii
17. 0239.sliding-window-maximum
18. 0295.find-median-from-data-stream
19. 0301.remove-invalid-parentheses
20. 0312.burst-balloons 🆕
21. 0335.self-crossPing
22. 0460.lfu-cache

23. 0472.concatenated-words 🆕
24. 0488.zuma-game.md 🆕
25. 0493.reverse-pairs 🆕
26. 0887.super-egg-drop
27. 0895.maximum-frequency-stack 🆕
28. 1032.stream-of-characters 🆕
29. 1168.optimize-water-distribution-in-a-village 🆕
30. 1449.form-largest-integer-with-digits-that-add-up-to-target 🆕

题目地址(4. 寻找两个正序数组的中位数)

https://leetcode.com/problems/median-of-two-sorted-arrays/

题目描述

```
给定两个大小为 m 和 n 的正序（从小到大）数组 nums1 和 nums2。

请你找出这两个正序数组的中位数，并且要求算法的时间复杂度为 O(log(m +

你可以假设 nums1 和 nums2 不会同时为空。



示例 1:

nums1 = [1, 3]
nums2 = [2]

则中位数是 2.0
示例 2:

nums1 = [1, 2]
nums2 = [3, 4]

则中位数是 (2 + 3)/2 = 2.5
```

前置知识

- 中位数
- 分治法
- 二分查找

公司

- 阿里
- 百度
- 腾讯

思路

首先了解一下 Median 的概念，一个数组中 median 就是把数组分成左右等分的中位数。

如下图：

这道题，很容易想到暴力解法，时间复杂度和空间复杂度都是 `O(m+n)` ，不符合题中给出 `O(log(m+n))` 时间复杂度的要求。 我们可以从简单的解法入手，试了一下，暴力解法也是可以被 Leetcode Accept 的. 分析中会给出两种解法，暴力求解和二分解法。

解法一 - 暴力 （Brute Force）

暴力解主要是要 merge 两个排序的数组 `（A，B）` 成一个排序的数组。

用两个 `pointer（i，j）` ， `i` 从数组 `A` 起始位置开始，即 `i=0` 开始， `j` 从数组 `B` 起始位置， 即 `j=0` 开始. 一一比较 `A[i]` 和 `B[j]` ，

1. 如果 `A[i] <= B[j]` , 则把 `A[i]` 放入新的数组中，i 往后移一位，即 `i+1` .
2. 如果 `A[i] > B[j]` , 则把 `B[j]` 放入新的数组中，j 往后移一位，即 `j+1` .
3. 重复步骤#1 和 #2，直到 `i` 移到 `A` 最后，或者 `j` 移到 `B` 最后。
4. 如果 `j` 移动到 `B` 数组最后，那么直接把剩下的所有 `A` 依次放入新的数组中.
5. 如果 `i` 移动到 `A` 数组最后，那么直接把剩下的所有 `B` 依次放入新的数组中.

Merge 的过程如下图。

时间复杂度： `O(m+n) - m is length of A, n is length of B`

空间复杂度： `O(m+n)`

解法二 - 二分查找 （Binary Search）

由于题中给出的数组都是排好序的，在排好序的数组中查找很容易想到可以用二分查找（Binary Search), 这里对数组长度小的做二分， 保证数组 A 和 数组 B 做 partition 之后

`len(Aleft)+len(Bleft)=(m+n+1)/2 - m是数组A的长度， n是数组B的长度`

对数组 A 的做 partition 的位置是区间 `[0,m]`

如图:

找median(中位数), 对数组A, B partition, 分成左右两部分. 这里我们用长度小的数组做二分.(m < n),

lo = 0, hi = m, i = lo + (hi - lo) / 2;

A以 i 做partition,

B以 j 做partition.

满足 i+j=(m+n+1)/2

```
if (Ai <= Bj+1 && Bj <= Ai+1) {
  // odd (奇数)
  if ((m+n) % 2 == 1) {
     median = max(Ai, Bj)
   } else {
      // Even (偶数)
      median = (max(Ai,Bj) + min(Ai+1,Bj+1))/2
   }
}
```

如果Ai>Bj+1, 说明左边的比右边的要大, 要往左半部分shift, hi = i - 1,

对A 区间 [lo, i-1] 二分, 对A, B重新做partition,

I = (lo+hi)/2, j = (m+n+1)/2 - i;

如果Bj>Ai+1, 说明右边的比左边的要大, 要往右半部分shift, lo = i + 1,

对A 区间 [i+1, hi] 二分, 对A, B重新做partition,

i = (lo+hi)/2, j = (m+n+1)/2 - i;

满足 (lo <= hi)条件, 直到找到median.

这里我们对数组小的做二分, 所以时间复杂度是:

O(log(min(m, n))

下图给出几种不同情况的例子（注意但左边或者右边没有元素的时候，左边用 `INF_MIN` ，右边用 `INF_MAX` 表示左右的元素：

下图给出具体做的 partition 解题的例子步骤，

时间复杂度： `O(log(min(m, n)) - m is length of A, n is length of B`

空间复杂度： `O(1)` - 这里没有用额外的空间

关键点分析

1. 暴力求解，在线性时间内 merge 两个排好序的数组成一个数组。
2. 二分查找，关键点在于

3. 要 partition 两个排好序的数组成左右两等份，partition 需要满足 `len(Aleft)+len(Bleft)=(m+n+1)/2 - m是数组A的长度， n是数组B的长度`

4. 并且 partition 后 A 左边最大(`maxLeftA`), A 右边最小（ `minRightA`), B 左边最大（ `maxLeftB`), B 右边最小（ `minRightB`) 满足 `(maxLeftA <= minRightB && maxLeftB <= minRightA)`

有了这两个条件，那么 median 就在这四个数中，根据奇数或者是偶数，

```
奇数:
median = max(maxLeftA, maxLeftB)
偶数:
median = (max(maxLeftA, maxLeftB) + min(minRightA, minRight
```

代码

代码支持： Java，JS：

Java Code：

解法一 - 暴力解法（Brute force）

`java [] class MedianTwoSortedArrayBruteForce { public double findMedianSortedArrays(int[] nums1, int[] nums2) { int[] newArr = mergeTwoSortedArray(nums1, nums2); int n = newArr.length; if (n % 2 == 0) { // even return (double) (newArr[n / 2] + newArr[n / 2 - 1]) / 2; } else { // odd return (double) newArr[n / 2]; } } private int[] mergeTwoSortedArray(int[] nums1, int[] nums2) { int m = nums1.length; int n = nums2.length; int[] res = new int[m + n]; int i = 0; int j = 0; int idx = 0; while (i < m && j < n) { if (nums1[i] <= nums2[j]) { res[idx++] = nums1[i++]; } else { res[idx++] = nums2[j++]; } } while (i < m) { res[idx++] = nums1[i++]; } while (j < n) { res[idx++] = nums2[j++]; } return res; } }

```javascript []
/**
 * @param {number[]} nums1
 * @param {number[]} nums2
 * @return {number}
 */
var findMedianSortedArrays = function (nums1, nums2) {
  // 归并排序
  const merged = [];
  let i = 0;
  let j = 0;
  while (i < nums1.length && j < nums2.length) {
    if (nums1[i] < nums2[j]) {
      merged.push(nums1[i++]);
    } else {
      merged.push(nums2[j++]);
    }
  }
  while (i < nums1.length) {
    merged.push(nums1[i++]);
  }
  while (j < nums2.length) {
    merged.push(nums2[j++]);
  }

  const { length } = merged;
  return length % 2 === 1
    ? merged[Math.floor(length / 2)]
    : (merged[length / 2] + merged[length / 2 - 1]) / 2;
};
```

复杂度分析

- 时间复杂度：$O(max(m, n))$
- 空间复杂度：$O(m + n)$

解法二 - 二分查找（Binary Search)

`js [] /**

- 二分解法
- @param {number[]} nums1
- @param {number[]} nums2
- @return {number} */ var findMedianSortedArrays = function (nums1, nums2) { // make sure to do binary search for shorten array if (nums1.length > nums2.length) { [nums1, nums2] = [nums2,

nums1]; } const m = nums1.length; const n = nums2.length; let low = 0; let high = m; while (low <= high) { const i = low + Math.floor((high - low) / 2); const j = Math.floor((m + n + 1) / 2) - i;

const maxLeftA = i === 0 ? -Infinity : nums1[i - 1]; const minRightA = i === m ? Infinity : nums1[i]; const maxLeftB = j === 0 ? -Infinity : nums2[j - 1]; const minRightB = j === n ? Infinity : nums2[j];

if (maxLeftA <= minRightB && minRightA >= maxLeftB) { return (m + n) % 2 === 1

```
? Math.max(maxLeftA, maxLeftB)
: (Math.max(maxLeftA, maxLeftB) + Math.min(minRightA,
```

} else if (maxLeftA > minRightB) { high = i - 1; } else { low = low + 1; } } }; `

`java [] class MedianSortedTwoArrayBinarySearch { public static double findMedianSortedArraysBinarySearch(int[] nums1, int[] nums2) { // do binary search for shorter length array, make sure time complexity log(min(m,n)). if (nums1.length > nums2.length) { return findMedianSortedArraysBinarySearch(nums2, nums1); } int m = nums1.length; int n = nums2.length; int lo = 0; int hi = m; while (lo <= hi) { // partition A position i int i = lo + (hi - lo) / 2; // partition B position j int j = (m + n + 1) / 2 - i;

```
    int maxLeftA = i == 0 ? Integer.MIN_VALUE : nums1[i - 1
    int minRightA = i == m ? Integer.MAX_VALUE : nums1[i];

    int maxLeftB = j == 0 ? Integer.MIN_VALUE : nums2[j - 1
    int minRightB = j == n ? Integer.MAX_VALUE : nums2[j];

    if (maxLeftA <= minRightB && maxLeftB <= minRightA) {
      // total length is even
      if ((m + n) % 2 == 0) {
        return (double) (Math.max(maxLeftA, maxLeftB) + Mat
      } else {
        // total length is odd
        return (double) Math.max(maxLeftA, maxLeftB);
      }
    } else if (maxLeftA > minRightB) {
      // binary search left half
      hi = i - 1;
    } else {
      // binary search right half
      lo = i + 1;
    }
  }
  return 0.0;
}
```

}`

复杂度分析

- 时间复杂度：$O(log(min(m,n)))$
- 空间复杂度：$O(log(min(m,n)))$

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

大家也可以关注我的公众号《脑洞前端》获取更多更新鲜的 LeetCode 题解

题目地址（23. 合并 K 个排序链表）

https://leetcode-cn.com/problems/merge-k-sorted-lists/description/

题目描述

合并 k 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。

示例:

输入: [1->4->5, 1->3->4, 2->6] 输出: 1->1->2->3->4->4->5->6

前置知识

- 链表
- 归并排序

公司

- 阿里
- 百度
- 腾讯
- 字节

思路

这道题目是合并 k 个已排序的链表，号称 leetcode 目前 `最难` 的链表题。和之前我们解决的88.merge-sorted-array很像。 他们有两点区别：

1. 这道题的数据结构是链表，那道是数组。这个其实不复杂，毕竟都是线性的数据结构。

2. 这道题需要合并 k 个元素，那道则只需要合并两个。这个是两题的关键差别，也是这道题难度为 `hard` 的原因。

因此我们可以看出，这道题目是 `88.merge-sorted-array` 的进阶版本。其实思路也有点像，我们来具体分析下第二条。 如果你熟悉合并排序的话，你会发现它就是 `合并排序的一部分` 。

具体我们可以来看一个动画

（动画来自 https://zhuanlan.zhihu.com/p/61796021 ）

关键点解析

- 分治
- 归并排序(merge sort)

代码

代码支持 JavaScript， Python3

JavaScript Code：

```
/*
 * @lc app=leetcode id=23 lang=javascript
 *
 * [23] Merge k Sorted Lists
 *
 * https://leetcode.com/problems/merge-k-sorted-lists/desc
 *
 */
function mergeTwoLists(l1, l2) {
  const dummyHead = {};
  let current = dummyHead;
  // l1: 1 -> 3 -> 5
  // l2: 2 -> 4 -> 6
  while (l1 !== null && l2 !== null) {
    if (l1.val < l2.val) {
      current.next = l1; // 把小的添加到结果链表
      current = current.next; // 移动结果链表的指针
      l1 = l1.next; // 移动小的那个链表的指针
    } else {
      current.next = l2;
      current = current.next;
      l2 = l2.next;
    }
  }

  if (l1 === null) {
    current.next = l2;
  } else {
    current.next = l1;
  }
  return dummyHead.next;
}
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode[]} lists
 * @return {ListNode}
 */
var mergeKLists = function(lists) {
  // 图参考： https://zhuanlan.zhihu.com/p/61796021
  if (lists.length === 0) return null;
  if (lists.length === 1) return lists[0];
  if (lists.length === 2) {
```

```
    return mergeTwoLists(lists[0], lists[1]);
  }

  const mid = lists.length >> 1;
  const l1 = [];
  for (let i = 0; i < mid; i++) {
    l1[i] = lists[i];
  }

  const l2 = [];
  for (let i = mid, j = 0; i < lists.length; i++, j++) {
    l2[j] = lists[i];
  }

  return mergeTwoLists(mergeKLists(l1), mergeKLists(l2));
};
```

Python3 Code：

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def mergeKLists(self, lists: List[ListNode]) -> ListNo
        n = len(lists)

        # basic cases
        if lenth == 0: return None
        if lenth == 1: return lists[0]
        if lenth == 2: return self.mergeTwoLists(lists[0],

        # divide and conqure if not basic cases
        mid = n // 2
        return self.mergeTwoLists(self.mergeKLists(lists[:r


    def mergeTwoLists(self, l1: ListNode, l2: ListNode) ->
        res = ListNode(0)
        c1, c2, c3 = l1, l2, res
        while c1 or c2:
            if c1 and c2:
                if c1.val < c2.val:
                    c3.next = ListNode(c1.val)
                    c1 = c1.next
                else:
                    c3.next = ListNode(c2.val)
                    c2 = c2.next
                c3 = c3.next
            elif c1:
                c3.next = c1
                break
            else:
                c3.next = c2
                break

        return res.next
```

相关题目

- 88.merge-sorted-array

题目地址

https://leetcode.com/problems/reverse-nodes-in-k-group/

题目描述

```
Given a linked list, reverse the nodes of a linked list k a

k is a positive integer and is less than or equal to the le

Example:

Given this linked list: 1->2->3->4->5

For k = 2, you should return: 2->1->4->3->5

For k = 3, you should return: 3->2->1->4->5

Note:

Only constant extra memory is allowed.
You may not alter the values in the list's nodes, only node
```

前置知识

- 链表

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

题意是以 `k` 个 nodes 为一组进行翻转，返回翻转后的 `linked list` .

从左往右扫描一遍 `linked list` ，扫描过程中，以 k 为单位把数组分成若干段，对每一段进行翻转。给定首尾 nodes，如何对链表进行翻转。

链表的翻转过程，初始化一个为 `null` 的 `previous node（prev）`，然后遍历链表的同时，当前 `node （curr）` 的下一个（next）指向前一个 `node（prev）`， 在改变当前 node 的指向之前，用一个临时变量记录当前 node 的下一个 `node（curr.next）`. 即

```
ListNode temp = curr.next;
curr.next = prev;
prev = curr;
curr = temp;
```

举例如图：翻转整个链表 `1->2->3->4->null` -> `4->3->2->1->null`

这里是对每一组（ `k个nodes` ）进行翻转，

1. 先分组，用一个 `count` 变量记录当前节点的个数
2. 用一个 `start` 变量记录当前分组的起始节点位置的前一个节点
3. 用一个 `end` 变量记录要翻转的最后一个节点位置
4. 翻转一组（ `k个nodes` ）即 `(start, end) - start and end exclusively` 。
5. 翻转后， `start` 指向翻转后链表, 区间 `（start，end）` 中的最后一个节点, 返回 `start` 节点。
6. 如果不需要翻转， `end` 就往后移动一个（ `end=end.next` ），每一次移动，都要 `count+1` .

如图所示 步骤 4 和 5： 翻转区间链表区间 `（start， end）`

举例如图， `head=[1,2,3,4,5,6,7,8], k = 3`

> **NOTE**: 一般情况下对链表的操作，都有可能会引入一个新的 `dummy node` ，因为 `head` 有可能会改变。这里 `head` 从`1->3` ，`dummy (List(0))` 保持不变。

复杂度分析

- *时间复杂度:* `O(n) - n is number of Linked List`
- *空间复杂度:* `O(1)`

关键点分析

1. 创建一个 dummy node
2. 对链表以 k 为单位进行分组，记录每一组的起始和最后节点位置
3. 对每一组进行翻转，更换起始和最后的位置
4. 返回 `dummy.next` .

代码 (`Java/Python3/javascript`)

Java Code

```java
class ReverseKGroupsLinkedList {
  public ListNode reverseKGroup(ListNode head, int k) {
      if (head == null || k == 1) {
        return head;
      }
      ListNode dummy = new ListNode(0);
      dummy.next = head;

      ListNode start = dummy;
      ListNode end = head;
      int count = 0;
      while (end != null) {
        count++;
        // group
        if (count % k == 0) {
          // reverse linked list (start, end]
          start = reverse(start, end.next);
          end = start.next;
        } else {
          end = end.next;
        }
      }
      return dummy.next;
    }

    /**
     * reverse linked list from range (start, end), return
     * for example:
     * 0->1->2->3->4->5->6->7->8
     * |           |
     * start       end
     *
     * After call start = reverse(start, end)
     *
     * 0->3->2->1->4->5->6->7->8
     *          |  |
     *       start end
     *       first
     *
     */
    private ListNode reverse(ListNode start, ListNode end)
      ListNode curr = start.next;
      ListNode prev = start;
      ListNode first = curr;
      while (curr != end){
        ListNode temp = curr.next;
        curr.next = prev;
        prev = curr;
```

```
            curr = temp;
        }
        start.next = prev;
        first.next = curr;
        return first;
    }
}
```

Python3 Cose

```
class Solution:
    def reverseKGroup(self, head: ListNode, k: int) -> List
        if head is None or k < 2:
            return head
        dummy = ListNode(0)
        dummy.next = head
        start = dummy
        end = head
        count = 0
        while end:
            count += 1
            if count % k == 0:
                start = self.reverse(start, end.next)
                # end 调到下一个
                end = start.next
            else:
                end = end.next
        return dummy.next
    # (start, end)  左右都开放

    def reverse(self, start, end):
        prev, curr = start, start.next
        first = curr
        # 反转
        while curr != end:
            next = curr.next
            curr.next = prev
            prev = curr
            curr = next
        # 将反转后的链表添加到原链表中
        start.next = prev
        first.next = end
        # 返回反转前的头,  也就是反转后的尾部
        return first
```

javascript code

```
/**
 * @param {ListNode} head
 * @param {number} k
 * @return {ListNode}
 */
var reverseKGroup = function (head, k) {
  // 标兵
  let dummy = new ListNode();
  dummy.next = head;
  let [start, end] = [dummy, dummy.next];
  let count = 0;
  while (end) {
    count++;
    if (count % k === 0) {
      start = reverseList(start, end.next);
      end = start.next;
    } else {
      end = end.next;
    }
  }
  return dummy.next;

  // 翻转stat -> end的链表
  function reverseList(start, end) {
    let [pre, cur] = [start, start.next];
    const first = cur;
    while (cur !== end) {
      let next = cur.next;
      cur.next = pre;
      pre = cur;
      cur = next;
    }
    start.next = pre;
    first.next = cur;
    return first;
  }
};
```

参考（References)

- Leetcode Discussion (yellowstone)

扩展 1

- 要求从后往前以 `k` 个为一组进行翻转。**(字节跳动（ByteDance）面试题)**

  例子， `1->2->3->4->5->6->7->8, k = 3` ,

  从后往前以 `k=3` 为一组，

  - `6->7->8` 为一组翻转为 `8->7->6` ，
  - `3->4->5` 为一组翻转为 `5->4->3` .
  - `1->2` 只有 2 个 nodes 少于 `k=3` 个，不翻转。

  最后返回： `1->2->5->4->3->8->7->6`

这里的思路跟从前往后以 `k` 个为一组进行翻转类似，可以进行预处理：

1. 翻转链表
2. 对翻转后的链表进行从前往后以 k 为一组翻转。
3. 翻转步骤 2 中得到的链表。

例子： `1->2->3->4->5->6->7->8, k = 3`

1. 翻转链表得到： `8->7->6->5->4->3->2->1`
2. 以 k 为一组翻转： `6->7->8->3->4->5->2->1`
3. 翻转步骤#2 链表： `1->2->5->4->3->8->7->6`

扩展 2

如果这道题你按照 92.reverse-linked-list-ii 提到的 `p1, p2, p3, p4` （四点法） 的思路来思考的话会很清晰。

代码如下（Python）：

```
class Solution:
    def reverseKGroup(self, head: ListNode, k: int) -> List
        if head is None or k < 2:
            return head
        dummy = ListNode(0)
        dummy.next = head
        pre = dummy
        cur = head
        count = 0
        while cur:
            count += 1
            if count % k == 0:
                pre = self.reverse(pre, cur.next)
                # end 调到下一个位置
                cur = pre.next
            else:
                cur = cur.next
        return dummy.next
    # (p1, p4)  左右都开放

    def reverse(self, p1, p4):
        prev, curr = p1, p1.next
        p2 = curr
        # 反转
        while curr != p4:
            next = curr.next
            curr.next = prev
            prev = curr
            curr = next
        # 将反转后的链表添加到原链表中
        # prev 相当于 p3
        p1.next = prev
        p2.next = p4
        # 返回反转前的头,  也就是反转后的尾部
        return p2

# @lc code=end
```

相关题目

- 92.reverse-linked-list-ii
- 206.reverse-linked-list

题目地址（30. 串联所有单词的子串）

https://leetcode-cn.com/problems/substring-with-concatenation-of-all-words/description/

题目描述

```
给定一个字符串 s 和一些长度相同的单词 words。找出 s 中恰好可以由 wo

注意子串要与 words 中的单词完全匹配，中间不能有其他字符，但不需要考虑




示例 1：

输入：
  s = "barfoothefoobarman",
  words = ["foo","bar"]
输出：[0,9]
解释：
从索引 0 和 9 开始的子串分别是 "barfoo" 和 "foobar" 。
输出的顺序不重要，[9,0] 也是有效答案。
示例 2：

输入：
  s = "wordgoodgoodgoodbestword",
  words = ["word","good","best","word"]
输出：[]
```

前置知识

- 字符串
- 数组
- 哈希表

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

本题是要我们找出 words 中 `所有单词按照任意顺序串联` 形成的单词中恰好出现在 s 中的索引，因此顺序是不重要的。换句话说，我们只要统计每一个单词的出现情况即可。以题目中 s = "barfoothefoobarman", words = ["foo","bar"] 为例。 我们只需要统计 foo 出现了一次，bar 出现了一次即可。我们只需要在 s 中找到同样包含一次 foo 和一次 bar 的子串即可。由于 words 中的字符串都是等长的，因此编码上也会比较简单。

1. 我们的目标状态是 Counter(words)，即对 words 进行一次计数。
2. 我们只需从头开始遍历一次数组，每次截取 word 长度的字符，一共截取 words 长度次即可。
3. 如果我们截取的 Counter 和 Counter(words)一致，则加入到 res
4. 否则我们继续一个指针，继续执行步骤二
5. 重复执行这个逻辑直到达到数组尾部

关键点解析

- Counter

代码

- 语言支持：Python3

```
from collections import Counter


class Solution:
    def findSubstring(self, s: str, words: List[str]) -> Li
        if not s or not words:
            return []
        res = []
        n = len(words)
        word_len = len(words[0])
        window_len = word_len * n
        target = Counter(words)
        i = 0
        while i < len(s) - window_len + 1:
            sliced = []
            start = i
            for _ in range(n):
                sliced.append(s[start:start + word_len])
                start += word_len
            if Counter(sliced) == target:
                res.append(i)
            i += 1
        return res
```

题目地址

https://leetcode.com/problems/longest-valid-parentheses/

题目描述

```
Given a string containing just the characters '(' and ')',

Example 1:

Input: "(()"
Output: 2
Explanation: The longest valid parentheses substring is "()
Example 2:

Input: ")()())"
Output: 4
Explanation: The longest valid parentheses substring is "()
```

前置知识

- 动态规划

暴力（超时）

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

符合直觉的做法是：分别计算以 i 开头的 最长有效括号（i 从 0 到 n - 1）， 从中取出最大的即可。

代码

代码支持： Python

```python
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        n = len(s)
        ans = 0

        def validCnt(start):
            # cnt 为 ) 的数量减去 ( 的数量
            cnt = 0
            ans = 0
            for i in range(start, n):
                if s[i] == '(':
                    cnt += 1
                if s[i] == ')':
                    cnt -= 1
                if cnt < 0:
                    return i - start
                if cnt == 0:
                    ans = max(ans, i - start + 1)
            return ans
        for i in range(n):
            ans = max(ans, validCnt(i))

        return ans
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(1)$

栈

思路

主要思路和常规的括号解法一样，遇到'('入栈，遇到')'出栈，并计算两个括号之间的长度。 因为这个题存在非法括号对的情况且求是合法括号对的最大长度 所以有两个注意点是:

1. **栈中存的是符号的下标**
2. **当栈为空时且当前扫描到的符号是')'时，需要将这个符号入栈作为分割符**
3. 栈中初始化一个 -1，作为**分割符**

代码

- 语言支持: Python, javascript

javascript code:

```
// 用栈来解
var longestValidParentheses = function (s) {
  let stack = new Array();
  let longest = 0;
  stack.push(-1);
  for (let i = 0; i < s.length; i++) {
    if (s[i] === "(") {
      stack.push(i);
    } else {
      stack.pop();
      if (stack.length === 0) {
        stack.push(i);
      } else {
        longest = Math.max(longest, i - stack[stack.length
      }
    }
  }
  return longest;
};
```

Python Code:

```
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        if not s:
            return 0
        res = 0
        stack = [-1]
        for i in range(len(s)):
            if s[i] == "(":
                stack.append(i)
            else:
                stack.pop()
                if not stack:
                    stack.append(i)
                else:
                    res = max(res, i - stack[-1])
        return res
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

O(1) 空间

思路

我们可以采用解法一中的计数方法。

- 从左到右遍历一次，并分别记录左右括号的数量 left 和 right。
- 如果 right > left ，说明截止上次可以匹配的点到当前点这一段无法匹配，重置 left 和 right 为 0
- 如果 right == left, 此时可以匹配，此时有效括号长度为 left + right，我们获得一个局部最优解。如果其比全局最优解大，我们更新全局最优解

值得注意的是，对形如 `(((()` 这样的，更新全局最优解的逻辑永远无法执行。一种方式是再从右往左遍历一次即可，具体看代码。

> 类似的思想有哨兵元素，虚拟节点。只不过本题无法采用这种方法。

代码

代码支持：Java，Python

```
public class Solution {
    public int longestValidParentheses(String s) {
        int left = 0, right = 0, maxlength = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(') {
                left++;
            } else {
                right++;
            }
            if (left == right) {
                maxlength = Math.max(maxlength, left + rigl
            }
            if (right > left) {
                left = right = 0;
            }
        }
        left = right = 0;
        for (int i = s.length() - 1; i >= 0; i--) {
            if (s.charAt(i) == '(') {
                left++;
            } else {
                right++;
            }
            if (left == right) {
                maxlength = Math.max(maxlength, left + rigl
            }
            if (left > right) {
                left = right = 0;
            }
        }
        return maxlength;
    }
}
```

```python
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        ans = l = r = 0
        for c in s:
            if c == '(':
                l += 1
            else:
                r += 1
            if l == r:
                ans = max(ans, l + r)
            if r > l:
                l = r = 0
        l = r = 0
        for c in s[::-1]:
            if c == '(':
                l += 1
            else:
                r += 1
            if l == r:
                ans = max(ans, l + r)
            if r < l:
                l = r = 0

        return ans
```

动态规划

思路

所有的动态规划问题, 首先需要解决的就是如何寻找合适的子问题. 该题需要我们找到最长的有效括号对, 我们首先想到的就是定义**dp[i]为前 i 个字符串的最长有效括号对长度**, 但是随后我们会发现, 这样的定义, 我们无法找到 dp[i]和 dp[i-1]的任何关系. 所以, 我们需要重新找一个新的定义: 定义**dp[i]为以第 i 个字符结尾的最长有效括号对长度**. 然后, 我们通过下面这个例子找一下 dp[i]和 dp[i-1]之间的关系.

```python
s = '(())())'
```

从上面的例子我们可以观察出一下几点结论(**描述中 i 为图中的 dp 数组的下标, 对应 s 的下标应为 i-1, 第 i 个字符的 i 从 1 开始**).

1. base case: 空字符串的最长有效括号对长度肯定为 0, 即: dp[0] = 0;
2. s 的第**1**个字符结尾的最长有效括号对长度为 0, s 的第**2**个字符结尾的最长有效括号对长度也为 0, 这个时候我们可以得出结论: 最长有效括号对不可能以'('结尾, 即: dp[1] = d[2] = 0;

3. 当 i 等于 3 时, 我们可以看出 dp[2]=0, dp[3]=2, 因为第 2 个字符(**s[1]**)和第 3 个字符(**s[2]**)是配对的; 当 i 等于 4 时, 我们可以看出 dp[3]=2, dp[4]=4, 因为我们配对的是第 1 个字符(**s[0]**)和第 4 个字符(**s[3]**); 因此, 我们可以得出结论: 如果第**i**个字符和第**i-1-dp[i-1]**个字符是配对的, 则 dp[i] = dp[i-1] + 2, 其中: i-1-dp[i-1] >= 1, 因为第 0 个字符没有任何意义;
4. 根据第 3 条规则来计算的话, 我们发现 dp[5]=0, dp[6]=2, 但是显然, dp[6]应该为 6 才对, 但是我们发现可以将"(())"和"()"进行拼接, 即: dp[i] += dp[i-dp[i]], 即: dp[6] = 2 + dp[6-2] = 2 + dp[4] = 6

根据以上规则, 我们求解 dp 数组的结果为: [0, 0, 0, 2, 4, 0, 6, 0], 其中最长有效括号对的长度为 6. 以下为图解:

代码

Python Code:

```python
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        mlen = 0
        slen = len(s)
        dp = [0] * (slen + 1)
        for i in range(1, len(s) + 1):
            # 有效的括号对不可能会以'('结尾的
            if s[i - 1] == '(':
                continue

            left_paren = i - 2 - dp[i - 1]
            if left_paren >= 0 and s[left_paren] == '(':
                dp[i] = dp[i - 1] + 2

                # 拼接有效括号对
                if dp[i - dp[i]]:
                    dp[i] += dp[i - dp[i]]

                # 更新最大有效扩对长度
                if dp[i] > mlen:
                    mlen = dp[i]

        return mlen
```

关键点解析

1. 第 3 点特征, 需要检查的字符是 s[i-1]和 s[i-2-dp[i-1]], 根据定义可知: i-1 >= dp[i-1], 但是 i-2 不一定大于 dp[i-1], 因此, 需要检查越界;
2. 第 4 点特征最容易遗漏, 还有就是不需要检查越界, 因为根据定义可知: i >= dp[i], 所以 dp[i-dp[i]]的边界情况是 dp[0];

相关题目

- 20.valid-parentheses

扩展

1. 如果判断的不仅仅只有'('和')', 还有'[', ']', '{'和'}', 该怎么办?
2. 如果输出的不是长度, 而是任意一个最长有效括号对的字符串, 该怎么办?

题目地址

https://leetcode.com/problems/trapping-rain-water/description/

题目描述

```
Given n non-negative integers representing an elevation map

The above elevation map is represented by array [0,1,0,2,1,
```

```
Example:

Input: [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
```

前置知识

- 空间换时间
- 双指针
- 单调栈

公司

- 阿里
- 腾讯
- 百度
- 字节

双数组

思路

这是一道雨水收集的问题， 难度为 `hard` . 如图所示，让我们求下过雨之后最多可以积攒多少的水。

如果采用暴力求解的话，思路应该是 height 数组依次求和，然后相加。

伪代码:

```
for (let i = 0; i < height.length; i++) {
  area += (h[i] - height[i]) * 1; // h为下雨之后的水位
}
```

问题转化为求 h，那么 h[i]又等于 `左右两侧柱子的最大值中的较小值` ，即 `h[i] = Math.min(左边柱子最大值, 右边柱子最大值)`

如上图那么 h 为 [0, 1, 1, 2, 2, 2 ,2, 3, 2, 2, 2, 1]

问题的关键在于求解 `左边柱子最大值` 和 `右边柱子最大值` , 我们其实可以用两个数组来表示 `leftMax` , `rightMax` ， 以 leftMax 为例，leftMax[i]代表 i 的左侧柱子的最大值，因此我们维护两个数组即可。

关键点解析

- 建模 `h[i] = Math.min(左边柱子最大值, 右边柱子最大值)` (h 为下雨之后的水位)

代码

代码支持 JavaScript，Python3，C++:

JavaScript Code:

```
/*
 * @lc app=leetcode id=42 lang=javascript
 *
 * [42] Trapping Rain Water
 *
 */
/**
 * @param {number[]} height
 * @return {number}
 */
var trap = function (height) {
  let max = 0;
  let volume = 0;
  const leftMax = [];
  const rightMax = [];

  for (let i = 0; i < height.length; i++) {
    leftMax[i] = max = Math.max(height[i], max);
  }

  max = 0;

  for (let i = height.length - 1; i >= 0; i--) {
    rightMax[i] = max = Math.max(height[i], max);
  }

  for (let i = 0; i < height.length; i++) {
    volume = volume + Math.min(leftMax[i], rightMax[i]) - h
  }

  return volume;
};
```

Python Code:

```python
class Solution:
    def trap(self, heights: List[int]) -> int:
        n = len(heights)
        l, r = [0] * (n + 1), [0] * (n + 1)
        ans = 0
        for i in range(1, len(heights) + 1):
            l[i] = max(l[i - 1], heights[i - 1])
        for i in range(len(heights) - 1, 0, -1):
            r[i] = max(r[i + 1], heights[i])
        for i in range(len(heights)):
            ans += max(0, min(l[i + 1], r[i]) - heights[i])
        return ans
```

C++ Code:

```cpp
int trap(vector<int>& heights)
{
    if(heights == null)
        return 0;
    int ans = 0;
    int size = heights.size();
    vector<int> left_max(size), right_max(size);
    left_max[0] = heights[0];
    for (int i = 1; i < size; i++) {
        left_max[i] = max(heights[i], left_max[i - 1]);
    }
    right_max[size - 1] = heights[size - 1];
    for (int i = size - 2; i >= 0; i--) {
        right_max[i] = max(heights[i], right_max[i + 1]);
    }
    for (int i = 1; i < size - 1; i++) {
        ans += min(left_max[i], right_max[i]) - heights[i];
    }
    return ans;
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

双指针

思路

上面代码比较好理解，但是需要额外的 N 的空间。从上面解法可以看出，我们实际上只关心左右两侧较小的那一个，并不需要两者都计算出来。具体来说：

- 如果 l[i + 1] < r[i] 那么 最终积水的高度由 i 的左侧最大值决定。
- 如果 l[i + 1] >= r[i] 那么 最终积水的高度由 i 的右侧最大值决定。

因此我们不必维护完整的两个数组，而是可以只进行一次遍历，同时维护左侧最大值和右侧最大值，使用常数变量完成即可。这是一个典型的双指针问题，

具体算法：

1. 维护两个指针 left 和 right，分别指向头尾。
2. 初始化左侧和右侧最高的高度都为 0。
3. 比较 height[left] 和 height[right]
    - 3.1 如果 height[left] < height[right]
        - 3.1.1 如果 height[left] >= left_max， 则当前格子积水面积为 (left_max - height[left])
        - 3.1.2 否则无法积水，即积水面积为 0
    - 3.2 左指针右移一位
    - 3.3 如果 height[left] >= height[right]
        - 3.3.1 如果 height[right] >= right_max， 则当前格子积水面积为(right_max - height[right])
        - 3.3.2 否则无法积水，即积水面积为 0
    - 3.4 右指针左移一位

代码

代码支持 Python3，C++:

```python
class Solution:
    def trap(self, heights: List[int]) -> int:
        n = len(heights)
        l_max = r_max = 0
        l, r = 0, n - 1
        ans = 0
        while l < r:
            if heights[l] < heights[r]:
                if heights[l] < l_max:
                    ans += l_max - heights[l]
                else:
                    l_max = heights[l]
                l += 1
            else:
                if heights[r] < r_max:
                    ans += r_max - heights[r]
                else:
                    r_max = heights[r]
                r -= 1
        return ans
```

```
class Solution {
public:
    int trap(vector<int>& heights)
{
    int left = 0, right = heights.size() - 1;
    int ans = 0;
    int left_max = 0, right_max = 0;
    while (left < right) {
        if (heights[left] < heights[right]) {
            heights[left] >= left_max ? (left_max = heights
            ++left;
        }
        else {
            heights[right] >= right_max ? (right_max = heig
            --right;
        }
    }
    return ans;
}

};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

相关题目

- 84.largest-rectangle-in-histogram

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址

https://leetcode-cn.com/problems/n-queens-ii

题目描述

```
n 皇后问题研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此

给定一个整数 n，返回 n 皇后不同的解决方案的数量。

示例：

输入：4
输出：2
解释：4 皇后问题存在如下两个不同的解法。

[
 [".Q..",  // 解法 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // 解法 2
  "Q...",
  "...Q",
  ".Q.."]
]
```

前置知识

- 回溯
- 深度优先遍历

公司

- 阿里
- 百度
- 字节

思路

使用深度优先搜索配合位运算，二进制为 1 代表不可放置，0 相反

利用如下位运算公式：

- x & -x ： 得到最低位的 1 代表除最后一位 1 保留，其他位全部为 0
- x & (x-1)：清零最低位的 1 代表将最后一位 1 变成 0
- x & ((1 << n) - 1)：将 x 最高位至第 n 位(含)清零

关键点

- 位运算
- DFS（深度优先搜索）

代码

- 语言支持：JS

```
/**
 * @param {number} n
 * @return {number}
 * @param row 当前层
 * @param cols 列
 * @param pie 左斜线
 * @param na 右斜线
 */
const totalNQueens = function (n) {
    let res = 0;
    const dfs = (n, row, cols, pie, na) => {
        if (row >= n) {
            res++;
            return;
        }
        // 将所有能放置 Q 的位置由 0 变成 1，以便进行后续的位遍历
        // 也就是得到当前所有的空位
        let bits = (~(cols | pie | na)) & ((1 << n) - 1);
        while (bits) {
            // 取最低位的1
            let p = bits & -bits;
            // 把P位置上放入皇后
            bits = bits & (bits - 1);
            // row + 1 搜索下一行可能的位置
            // cols | p 目前所有放置皇后的列
            // (pie | p) << 1 和 (na | p) >> 1) 与已放置过皇后
            dfs(n, row + 1, cols | p, (pie | p) << 1, (na
        }
    }
    dfs(n, 0, 0, 0, 0);
    return res;
};
```

复杂度分析

- 时间复杂度：O(N!)
- 空间复杂度：O(N)

题目地址（84. 柱状图中最大的矩形）

https://leetcode-cn.com/problems/largest-rectangle-in-histogram/

题目描述

` 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

前置知识

- 单调栈

暴力枚举 - 左右端点法（TLE）

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

我们暴力尝试 `所有可能的矩形` 。由于矩阵是二维图形， 我我们可以使用 `左右两个端点来唯一确认一个矩阵` 。因此我们使用双层循环枚举所有的可能性即可。 而矩形的面积等于 `（右端点坐标 - 左端点坐标 + 1）* 最小的高度` ，最小的高度我们可以在遍历的时候顺便求出。

代码

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> 
        n, ans = len(heights), 0
        if n != 0:
            ans = heights[0]
        for i in range(n):
            height = heights[i]
            for j in range(i, n):
                height = min(height, heights[j])
                ans = max(ans, (j - i + 1) * height)
        return ans
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(1)$

暴力枚举 - 中心扩展法（TLE）

思路

我们仍然暴力尝试 `所有可能的矩形` 。只不过我们这一次从中心向两边进行扩展。对于每一个 i，我们计算出其左边第一个高度小于它的索引 p，同样地，计算出右边第一个高度小于它的索引 q。那么以 i 为最低点能够构成的面积就是 `（q - p - 1）* heights[i]` 。 这种算法毫无疑问也是

正确的。 我们证明一下，假设 f(i) 表示求以 i 为最低点的情况下，所能形成的最大矩阵面积。那么原问题转化为 `max(f(0), f(1), f(2), ..., f(n - 1))` 。

具体算法如下：

- 我们使用 l 和 r 数组。l[i] 表示 左边第一个高度小于它的索引，r[i] 表示 右边第一个高度小于它的索引。
- 我们从前往后求出 l，再从后往前计算出 r。
- 再次遍历求出所有的可能面积，并取出最大的。

代码

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> 
        n = len(heights)
        l, r, ans = [-1] * n, [n] * n, 0
        for i in range(1, n):
            j = i - 1
            while j >= 0 and heights[j] >= heights[i]:
                j -= 1
            l[i] = j
        for i in range(n - 2, -1, -1):
            j = i + 1
            while j < n and heights[j] >= heights[i]:
                j += 1
            r[i] = j
        for i in range(n):
            ans = max(ans, heights[i] * (r[i] - l[i] - 1))
        return ans
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N)$

优化中心扩展法（Accepted）

思路

实际上我们内层循环没必要一步一步移动，我们可以直接将 `j -= 1` 改成 `j = l[j]` ，`j += 1` 改成 `j = r[j]` 。

代码

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> 
        n = len(heights)
        l, r, ans = [-1] * n, [n] * n, 0

        for i in range(1, n):
            j = i - 1
            while j >= 0 and heights[j] >= heights[i]:
                j = l[j]
            l[i] = j
        for i in range(n - 2, -1, -1):
            j = i + 1
            while j < n and heights[j] >= heights[i]:
                j = r[j]
            r[i] = j
        for i in range(n):
            ans = max(ans, heights[i] * (r[i] - l[i] - 1))
        return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

单调栈（Accepted）

思路

实际上，读完第二种方法的时候，你应该注意到了。我们的核心是求左边第一个比 i 小的和右边第一个比 i 小的。 如果你熟悉单调栈的话，那么应该会想到这是非常适合使用单调栈来处理的场景。

从左到右遍历柱子，对于每一个柱子，我们想找到第一个高度小于它的柱子，那么我们就可以使用一个单调递增栈来实现。 如果柱子大于栈顶的柱子，那么说明不是我们要找的柱子，我们把它塞进去继续遍历，如果比栈顶小，那么我们就找到了第一个小于的柱子。 **对于栈顶元素，其右边第一个小于它的就是当前遍历到的柱子，左边第一个小于它的就是栈中下一个要被弹出的元素**，因此以当前栈顶为最小柱子的面积为**当前栈顶的柱子高度 * (当前遍历到的柱子索引 - 1 - 栈中下一个要被弹出的元素索引 - 1 + 1)**

这种方法只需要遍历一次，并用一个栈。由于每一个元素最多进栈出栈一次，因此时间和空间复杂度都是$O(N)$。

为了统一算法逻辑，减少边界处理，我在 heights 首尾添加了两个哨兵元素，**这样我们可以保证所有的柱子都会出栈**。

代码

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> 
        n, heights, st, ans = len(heights), [0] + heights 
        for i in range(n + 2):
            while st and heights[st[-1]] > heights[i]:
                ans = max(ans, heights[st.pop(-1)] * (i - 
            st.append(i)
        return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

2020-05-30 更新：

有的观众反应看不懂为啥需要两个哨兵。 其实末尾的哨兵就是为了将栈清空，防止遍历完成栈中还有没参与运算的数据。

而前面的哨兵有什么用呢？ 我这里把上面代码进行了拆解：

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> 
        n, heights, st, ans = len(heights),[0] + heights +
        for i in range(n + 2):
            while st and heights[st[-1]] > heights[i]:
                a = heights[st[-1]]
                # 如果没有前面的哨兵，这里可能会越界。
                st.pop()
                ans = max(ans, a * (i - 1 - st[-1]))
            st.append(i)
        return ans
```

相关题目

- 42.trapping-rain-water

欢迎关注我的公众号《脑洞前端》获取更多更新鲜的 LeetCode 题解

微信搜一搜
脑洞前端
牛客@牛客7325506号

题目地址（85. 最大矩形）

https://leetcode-cn.com/problems/maximal-rectangle/

题目描述

给定一个仅包含 0 和 1 的二维二进制矩阵，找出只包含 1 的最大矩形，并返回其面积。

示例：

输入：

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

输出：6

前置知识

- 单调栈

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

我在 【84. 柱状图中最大的矩形】多种方法（Python3） 使用了多种方法来解决。 然而在这道题，我们仍然可以使用完全一样的思路去完成。 不熟悉的可以看下我的题解。本题解是基于那道题的题解来进行的。

拿题目给的例子来说：

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

我们逐行扫描得到 `84. 柱状图中最大的矩形` 中的 heights 数组：

85. 最大矩形

这样我们就可以使用 `84. 柱状图中最大的矩形` 中的解法来进行了，这里我们使用单调栈来解。

代码

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> 
        n, heights, st, ans = len(heights), [0] + heights 
        for i in range(n + 2):
            while st and heights[st[-1]] > heights[i]:
                ans = max(ans, heights[st.pop(-1)] * (i - 
            st.append(i)

        return ans
    def maximalRectangle(self, matrix: List[List[str]]) -> 
        m = len(matrix)
        if m == 0: return 0
        n = len(matrix[0])
        heights = [0] * n
        ans = 0
        for i in range(m):
            for j in range(n):
                if matrix[i][j] == "0":
                    heights[j] = 0
                else:
                    heights[j] += 1
            ans = max(ans, self.largestRectangleArea(heigh
        return ans
```

复杂度分析

- 时间复杂度：$O(M * N)$
- 空间复杂度：$O(N)$

欢迎关注我的公众号《脑洞前端》获取更多更新鲜的 LeetCode 题解

题目地址

https://leetcode.com/problems/binary-tree-maximum-path-sum/description/

题目描述

```
Given a non-empty binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of node

Example 1:

Input: [1,2,3]

       1
      / \
     2   3

Output: 6
Example 2:

Input: [-10,9,20,null,null,15,7]

   -10
   / \
  9  20
    /  \
   15   7

Output: 42
```

前置知识

- 递归

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目的path让我误解了，然后浪费了很多时间来解这道题 我觉得leetcode给的demo太少了，不足以让我理解path的概念 因此我这里自己画了一个图，来补充一下，帮助大家理解path的概念，不要像我一样理解错啦。

首先是官网给的两个例子：

接着是我自己画的一个例子：

如图红色的部分是最大路径上的节点。大家可以结合上面的demo来继续理解一下path， 除非你理解了path，否则不要往下看。

树的题目，基本都是考察递归思想的。因此我们需要思考如何去定义我们的递归函数，在这里我定义了一个递归函数，它的功能是， `返回以当前节点为根节点的MaxPath`

但是有两个条件:

1. 根节点必须选择
2. 左右子树只能选择一个

为什么要有这两个条件?

我的想法是原问题可以转化为：以每一个节点为根节点，分别求出 MaxPath，最后计算最大值，因此第一个条件需要满足.

对于第二个条件，由于递归函数子节点的返回值会被父节点使用，因此我们如果两个孩子都选择了就不符合 MaxPath 的定义了。实际上这道题，当遍历到某一个节点的时候，我们需要子节点的信息，然后同时结合自身的 val 来决定要不要选取左右子树以及选取的话要选哪一个， 因此这个过程本质上就是 `后序遍历`

基本算法就是不断调用递归函数，然后在调用过程中不断计算和更新 MaxPath，最后在主函数中将 MaxPath 返回即可。

关键点解析

- 递归
- 理解题目中的path定义

代码

代码支持：JavaScript，Java，Python

- JavaScript

```javascript
/*
 * @lc app=leetcode id=124 lang=javascript
 *
 * [124] Binary Tree Maximum Path Sum
 */
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
function helper(node, payload) {
  if (node === null) return 0;

  const l = helper(node.left, payload);
  const r = helper(node.right, payload);

  payload.max = Math.max(
    node.val + Math.max(0, l) + Math.max(0, r),
    payload.max
  );

  return node.val + Math.max(l, r, 0);
}
/**
 * @param {TreeNode} root
 * @return {number}
 */
var maxPathSum = function(root) {
  if (root === null) return 0;
  const payload = {
    max: root.val
  };
  helper(root, payload);
  return payload.max;
};
```

- Java

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
  int ans;
  public int maxPathSum(TreeNode root) {
    ans = Integer.MIN_VALUE;
    helper(root);   // recursion
    return ans;
  }

  public int helper(TreeNode root) {
    if (root == null) return 0;
    int leftMax = Math.max(0, helper(root.left));     // f
    int rightMax = Math.max(0, helper(root.right));   // f
    ans = Math.max(ans, leftMax+rightMax+root.val);   // f
    return max(leftMax, rightMax) + root.val;         // a
  }
}
```

- Python

```
class Solution:
    ans = float('-inf')
    def maxPathSum(self, root: TreeNode) -> int:
        def helper(node):
            if not node: return 0
            l = helper(node.left)
            r = helper(node.right)
            self.ans = max(self.ans, max(l,0) + max(r, 0) -
            return max(l, r, 0) + node.val
        helper(root)
        return self.ans
```

相关题目

- 113.path-sum-ii

题目地址

https://leetcode.com/problems/longest-consecutive-sequence/description/

题目描述

```
Given an unsorted array of integers, find the length of the

Your algorithm should run in O(n) complexity.

Example:

Input: [100, 4, 200, 1, 3, 2]
Output: 4
Explanation: The longest consecutive elements sequence is
Accepted
200,786
Submissions
485,346
```

前置知识

- hashmap

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这是一道最最长连续数字序列长度的题目， 官网给出的难度是 `hard` .

符合直觉的做法是先排序，然后用一个变量记录最大值，遍历去更新最大值即可，

代码：

```
if (nums.length === 0) return 0;
let count = 1;
let maxCount = 1;
// 这里其实可以不需要排序，这么做只不过是为了方便理解
nums = [...new Set(nums)].sort((a, b) => a - b);
for (let i = 0; i < nums.length - 1; i++) {
  if (nums[i + 1] - nums[i] === 1) {
    count++;
  } else {
    if (count > maxCount) {
      maxCount = count;
    }
    count = 1;
  }
}
return Math.max(count, maxCount);
```

但是需要排序时间复杂度会上升，题目要求时间复杂度为 O(n), 那么我们其实可以不用排序去解决的。

思路就是将之前"排序之后，通过比较前后元素是否相差 1 来判断是否连续"的思路改为 不排序而是 `直接遍历，然后在内部循环里面查找是否存在当前值的邻居元素` ，但是马上有一个 问题，内部我们 `查找是否存在当前值的邻居元素` 的过程如果使用数组，时间复杂度是 O(n), 那么总体的复杂度就是 O(n^2)，完全不可以接受。怎么办呢?

我们换个思路，用空间来换时间。比如用类似于 hashmap 这样的数据结构优化查询部分，将时间复杂度降低到 O(1), 代码见后面 `代码部分`

关键点解析

- 空间换时间

代码

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var longestConsecutive = function(nums) {
  set = new Set(nums);
  let max = 0;
  let temp = 0;
  set.forEach(x => {
    // 说明x是连续序列的开头元素
    if (!set.has(x - 1)) {
      temp = x + 1;
      while (set.has(y)) {
        temp = temp + 1;
      }
      max = Math.max(max, y - x); // y - x 就是从x开始到最后有
    }
  });
  return max;
};
```

题目地址

https://leetcode.com/problems/binary-tree-postorder-traversal/description/

题目描述

```
Given a binary tree, return the postorder traversal of its

For example:
Given binary tree {1,#,2,3},

   1
    \
     2
    /
   3


return [3,2,1].

Note: Recursive solution is trivial, could you do it iterat
```

前置知识

- 栈
- 递归

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

相比于前序遍历，后续遍历思维上难度要大些，前序遍历是通过一个stack，首先压入父亲结点，然后弹出父亲结点，并输出它的value，之后压人其右儿子，左儿子即可。

然而后序遍历结点的访问顺序是：左儿子 -> 右儿子 -> 自己。那么一个结点需要两种情况下才能够输出： 第一，它已经是叶子结点； 第二，它不是叶子结点，但是它的儿子已经输出过。

那么基于此我们只需要记录一下当前输出的结点即可。对于一个新的结点，如果它不是叶子结点，儿子也没有访问，那么就需要将它的右儿子，左儿子压入。 如果它满足输出条件，则输出它，并记录下当前输出结点。输出在stack为空时结束。

关键点解析

- 二叉树的基本操作（遍历）
  > 不同的遍历算法差异还是蛮大的
- 如果非递归的话利用栈来简化操作
- 如果数据规模不大的话，建议使用递归
- 递归的问题需要注意两点，一个是终止条件，一个如何缩小规模
- 终止条件，自然是当前这个元素是null（链表也是一样）
- 由于二叉树本身就是一个递归结构， 每次处理一个子树其实就是缩小了规模， 难点在于如何合并结果，这里的合并结果其实就是 `left.concat(right).concat(mid)` , mid是一个具体的节点，left和right `递归求出即可`

代码

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var postorderTraversal = function(root) {
  // 1. Recursive solution

  // if (!root) return [];

  // return postorderTraversal(root.left).concat(postorder

  // 2. iterative solutuon

  if (!root) return [];
  const ret = [];
  const stack = [root];
  let p = root; // 标识元素，用来判断节点是否应该出栈

  while (stack.length > 0) {
    const top = stack[stack.length - 1];
    if (
      top.left === p ||
      top.right === p || // 子节点已经遍历过了
      (top.left === null && top.right === null) // 叶子元素
    ) {
      p = stack.pop();
      ret.push(p.val);
    } else {
      if (top.right) {
        stack.push(top.right);
      }
      if (top.left) {
        stack.push(top.left);
      }
    }
  }

  return ret;
};
```

相关专题

- 二叉树的遍历

题目地址（212. 单词搜索 II）

https://leetcode-cn.com/problems/word-search-ii/description/

题目描述

```
给定一个二维网格 board 和一个字典中的单词列表 words，找出所有同时在二

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

示例：

输入：
words = ["oath","pea","eat","rain"] and board =
[
  ['o','a','a','n'],
  ['e','t','a','e'],
  ['i','h','k','r'],
  ['i','f','l','v']
]

输出：["eat","oath"]
说明：
你可以假设所有输入都由小写字母 a-z 组成。

提示：

你需要优化回溯算法以通过更大数据量的测试。你能否早点停止回溯？
如果当前单词不存在于所有单词的前缀中，则可以立即停止回溯。什么样的数据
```

前置知识

- 前缀树
- DFS

公司

- 百度
- 字节

思路

我们需要对矩阵中每一项都进行深度优先遍历（DFS）。 递归的终点是

1. 超出边界
2. 递归路径上组成的单词不在 words 的前缀。

比如题目示例：words = ["oath","pea","eat","rain"]，那么对于 oa，oat 满足条件，因为他们都是 oath 的前缀，但是 oaa 就不满足条件。

为了防止环的出现，我们需要记录访问过的节点。而返回结果是需要去重的。出于简单考虑，我们使用集合（set），最后返回的时候重新转化为 list。

刚才我提到了一个关键词“前缀”，我们考虑使用前缀树来优化。使得复杂度降低为$O(h)$, 其中 h 是前缀树深度，也就是最长的字符串长度。

关键点

- 前缀树（也叫字典树），英文名 Trie（读作 tree 或者 try）
- DFS
- hashmap 结合 dfs 记录访问过的元素的时候，注意结束之后需要将 hashmap 的值重置。（下方代码的 `del seen[(i, j)]` ）

代码

- 语言支持：Python3

Python3 Code：

关于 Trie 的代码:

```
class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.Trie = {}

    def insert(self, word):
        """
        Inserts a word into the trie.
        :type word: str
        :rtype: void
        """
        curr = self.Trie
        for w in word:
            if w not in curr:
                curr[w] = {}
            curr = curr[w]
        curr['#'] = 1

    def startsWith(self, prefix):
        """
        Returns if there is any word in the trie that start
        :type prefix: str
        :rtype: bool
        """

        curr = self.Trie
        for w in prefix:
            if w not in curr:
                return False
            curr = curr[w]
        return True
```

主逻辑代码：

```
class Solution:
    def findWords(self, board: List[List[str]], words: List
        m = len(board)
        if m == 0:
            return []
        n = len(board[0])
        trie = Trie()
        seen = None
        res = set()
        for word in words:
            trie.insert(word)

        def dfs(s, i, j):
            if (i, j) in seen or i < 0 or i >= m or j < 0 o
                return
            s += board[i][j]
            seen[(i, j)] = True

            if s in words:
                res.add(s)
            dfs(s, i + 1, j)
            dfs(s, i - 1, j)
            dfs(s, i, j + 1)
            dfs(s, i, j - 1)

            del seen[(i, j)]

        for i in range(m):
            for j in range(n):
                seen = dict()
                dfs("", i, j)
        return list(res)
```

相关题目

- 0208.implement-trie-prefix-tree
- 0211.add-and-search-word-data-structure-design
- 0472.concatenated-words
- 0820.short-encoding-of-words
- 1032.stream-of-characters

题目地址

https://leetcode.com/problems/sliding-window-maximum/description/

题目描述

```
Given an array nums, there is a sliding window of size k wh

Example:

Input: nums = [1,3,-1,-3,5,3,6,7], and k = 3
Output: [3,3,5,5,6,7]
Explanation:

Window position                Max
---------------               -----
[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
Note:
You may assume k is always valid, 1 ≤ k ≤ input array's siz

Follow up:
Could you solve it in linear time?
```

前置知识

- 队列
- 滑动窗口

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

符合直觉的想法是直接遍历 nums, 然后然后用一个变量 slideWindow 去承载 k 个元素， 然后对 slideWindow 求最大值，这是可以的，时间复杂度是 O(n * k).代码如下：

JavaScript:

```
var maxSlidingWindow = function(nums, k) {
  // bad 时间复杂度O(n * k)
  if (nums.length === 0 || k === 0) return [];
  let slideWindow = [];
  const ret = [];
  for (let i = 0; i < nums.length - k + 1; i++) {
    for (let j = 0; j < k; j++) {
      slideWindow.push(nums[i + j]);
    }
    ret.push(Math.max(...slideWindow));
    slideWindow = [];
  }
  return ret;
};
```

Python3:

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) ->
        if k == 0: return []
        res = []
        for r in range(k - 1, len(nums)):
            res.append(max(nums[r - k + 1:r + 1]))
        return res
```

但是如果真的是这样，这道题也不会是 hard 吧？这道题有一个 follow up，要求你用线性的时间去完成。 我们可以用双端队列来完成，思路是用一个双端队列来保存 `接下来的滑动窗口可能成为最大值的数` 。具体做法：

- 入队列
- 移除失效元素，失效元素有两种
- 一种是已经超出窗口范围了，比如我遍历到第4个元素，k = 3，那么i = 0的元素就不应该出现在双端队列中了 具体就是 `索引大于 i - k + 1的元素都应该被清除`
- 小于当前元素都没有利用价值了，具体就是 `从后往前遍历（双端队列是一个递减队列）双端队列，如果小于当前元素就出队列`

如果你仔细观察的话，发现双端队列其实是一个递减的一个队列。因此队首的元素一定是最大的。用图来表示就是：

关键点解析

- 双端队列简化时间复杂度
- 滑动窗口

代码

JavaScript:

```
var maxSlidingWindow = function(nums, k) {
  // 双端队列优化时间复杂度，时间复杂度O(n)
  const deque = []; // 存放在接下来的滑动窗口可能成为最大值的数
  const ret = [];
  for (let i = 0; i < nums.length; i++) {
    // 清空失效元素
    while (deque[0] < i - k + 1) {
      deque.shift();
    }

    while (nums[deque[deque.length - 1]] < nums[i]) {
      deque.pop();
    }

    deque.push(i);

    if (i >= k - 1) {
      ret.push(nums[deque[0]]);
    }
  }
  return ret;
};
```

Python3:

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) ->
        deque, res, n = [], [], len(nums)
        for i in range(n):
            while deque and deque[0] < i - k + 1:
                deque.pop(0)
            while deque and nums[i] > nums[deque[-1]]:
                deque.pop(-1)
            deque.append(i)
            if i >= k - 1: res.append(nums[deque[0]])
        return res
```

扩展

为什么用双端队列

因为删除无效元素的时候，会清除队首的元素（索引太小了 ）或者队尾(元素太小了)的元素。 因此需要同时对队首和队尾进行操作，使用双端队列是一种合乎情理的做法。

题目地址

https://leetcode.com/problems/find-median-from-data-stream/description/

题目描述

```
Median is the middle value in an ordered integer list. If t

For example,
[2,3,4], the median is 3

[2,3], the median is (2 + 3) / 2 = 2.5

Design a data structure that supports the following two ope

void addNum(int num) - Add a integer number from the data s
double findMedian() - Return the median of all elements so


Example:

addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2


Follow up:

If all integer numbers from the stream are between 0 and 10
If 99% of all integer numbers from the stream are between 0
```

前置知识

- 堆
- 队列

公司

- 阿里
- 百度
- 字节

思路

这道题目是求动态数据的中位数，在 leetcode 难度为 `hard` . 如果这道题是求静态数据的中位数，我们用数组去存储， 空间复杂度 O(1), 时间复杂度 O(1)

> 空间复杂度指的是除了存储数据之外额外开辟的用于计算等任务的内存空间

代码也比较简单

```
function findMedian(a) {
  return a.length % 2 === 0
    ? (a[a.length >> 1] + a[a.length >> (1 + 1)]) / 2
    : a[a.length >> 1];
}
```

但是题目要求是动态数据， 那么是否可以每次添加数据的时候，都去排一次序呢？ 假如我们每次插入都用 `快速排序` 进行排序的话，那么时间复杂度是 O(nlogn) + O(1)

> O(nlogn) 是排序的时间复杂度 O(1)是查询中位数的时间复杂度

如果你用这种思路进行的话， 恐怕 leetcode 会超时。

那么如何优化呢？ 答案是使用堆， Java， C++等语言都有 `优先级队列` 中这种数据结构， 优先级队列本质上就是一个堆。 关于堆和优先级队列的关系，我会放在《数据结构和算法》部分讲解。这里不赘述

如果借助堆这种数据结构， 就可以轻易实现了。

具体的做法是，建立两个堆，这两个堆需要满足:

1. 大顶堆元素都比小顶堆小（由于堆的特点其实只要比较堆顶即可）
2. 大顶堆元素不小于小顶堆，且最多比小顶堆多一个元素

满足上面两个条件的话，如果想要找到中位数，就比较简单了

- 如果两个堆数量相等（本质是总数为偶数）, 就两个堆顶元素的平均数
- 如果两个堆数量不相等（本质是总数为奇数）， 就取大顶堆的堆顶元素

比如对于[1,2,3] 求中位数：

再比如对于[1,2,3, 4] 求中位数：

关键点解析

- 用两个堆（一个大顶堆，一个小顶堆）来简化时间复杂度
- 用优先级队列简化操作

> JavaScript 不像 Java， C++等语言都有 `优先级队列` 中这种数据结构， 因此大家可以使用社区的实现 个人认为没有非要纠结于优先级队列怎么实现， 至少这道题不是考这个的 优先级队列的实现个人认为已经超过了这道题想考察的范畴
>
> ## 代码

如果不使用现成的 `优先级队列` 这种数据结构，代码可能是这样的：

```
/**
 * initialize your data structure here.
 */
var MedianFinder = function() {
  this.maxHeap = [];
  this.minHeap = [];
};

function minHeapify() {
  this.minHeap.unshift(null);
  const a = this.minHeap;

  // 为了方便大家理解，这里选用了粗暴的实现
  // 时间复杂度为O(n)
  // 其实可以降到O(logn)，  具体细节我不想在这里讲解和实现
  for (let i = a.length - 1; i >> 1 > 0; i--) {
    // 自下往上堆化
    if (a[i] < a[i >> 1]) { // 如果子元素更小，则交换位置
      const temp = a[i];
      this.minHeap[i] = a[i >> 1];
      this.minHeap[i >> 1] = temp;
    }
  }
  this.minHeap.shift(null);
}

function maxHeapify() {
  this.maxHeap.unshift(null);
  const a = this.maxHeap;

  // 为了方便大家理解，这里选用了粗暴的实现
  // 时间复杂度为O(n)
  // 其实可以降到O(logn)，  具体细节我不想在这里讲解和实现
  for (let i = a.length - 1; i >> 1 > 0; i--) {
    // 自下往上堆化
    if (a[i] > a[i >> 1]) { // 如果子元素更大，则交换位置
      const temp = a[i];
      this.maxHeap[i] = a[i >> 1];
      this.maxHeap[i >> 1] = temp;
    }
  }
  this.maxHeap.shift(null);
}

/**
 * @param {number} num
 * @return {void}
 */
```

```
MedianFinder.prototype.addNum = function(num) {
  // 为了大家容易理解，这部分代码写的比较冗余

  // 插入
  if (num >= (this.minHeap[0] || Number.MIN_VALUE)) {
    this.minHeap.push(num);
  } else {
    this.maxHeap.push(num);
  }
  // 调整两个堆的节点数量平衡
  // 使得大顶堆的数量最多大于小顶堆一个， 且一定不小于小顶堆数量
  if (this.maxHeap.length > this.minHeap.length + 1) {
    // 大顶堆的堆顶元素移动到小顶堆
    this.minHeap.push(this.maxHeap.shift());
  }

  if (this.minHeap.length > this.maxHeap.length) {
    // 小顶堆的堆顶元素移动到大顶堆
    this.maxHeap.push(this.minHeap.shift());
  }

  // 调整堆顶元素
  if (this.maxHeap[0] > this.minHeap[0]) {
    const temp = this.maxHeap[0];
    this.maxHeap[0] = this.minHeap[0];
    this.minHeap[0] = temp;
  }

  // 堆化
  maxHeapify.call(this);
  minHeapify.call(this);
};

/**
 * @return {number}
 */
MedianFinder.prototype.findMedian = function() {
  if ((this.maxHeap.length + this.minHeap.length) % 2 === 0
    return (this.minHeap[0] + this.maxHeap[0]) / 2;
  } else {
    return this.maxHeap[0];
  }
};

/**
 * Your MedianFinder object will be instantiated and called
 * var obj = new MedianFinder()
 * obj.addNum(num)
```

```
 * var param_2 = obj.findMedian()
 */
```

其中 `minHeapify` 和 `maxHeapify` 的过程都有一个hack操作，就是：

```
this.heap.unshift(null);
// ....
this.heap.shift(null);
```

其实就是为了存储的数据从1开始，这样方便计算。 即对于下标为i的元素， `i >> 1` 一定是父节点的下标。

> 这是因为我用满二叉树来存储的堆

这个实现比较繁琐，下面介绍一种优雅的方式，假设JS和Java和C++等语言一样有 `PriorityQueue` 这种数据结构，那么我们实现就比较简单了。

代码：

> 关于PriorityQueue的实现，感兴趣的可以看下
> https://github.com/janogonzalez/priorityqueuejs

```
/*
 * @lc app=leetcode id=295 lang=javascript
 *
 * [295] Find Median from Data Stream
 *
 * https://leetcode.com/problems/find-median-from-data-stre
 *
 * algorithms
 * Hard (35.08%)
 * Total Accepted:    101.2K
 * Total Submissions: 282.4K
 * Testcase Example:  '["MedianFinder","addNum","addNum","f
 *
 * Median is the middle value in an ordered integer list. I
 * list is even, there is no middle value. So the median is
 * middle value.
 * For example,
 *
 * [2,3,4], the median is 3
 *
 * [2,3], the median is (2 + 3) / 2 = 2.5
 *
 * Design a data structure that supports the following two
 *
 *
 * void addNum(int num) - Add a integer number from the dat
 * structure.
 * double findMedian() - Return the median of all elements
 *
 *
 *
 *
 * Example:
 *
 *
 * addNum(1)
 * addNum(2)
 * findMedian() -> 1.5
 * addNum(3)
 * findMedian() -> 2
 *
 *
 *
 *
 * Follow up:
 *
 *
 * If all integer numbers from the stream are between 0 and
```

```
 * optimize it?
 * If 99% of all integer numbers from the stream are betwee
 * would you optimize it?
 *
 *
 */
/**
 * initialize your data structure here.
 */
var MedianFinder = function() {
  this.maxHeap = new PriorityQueue((a, b) => a - b);
  this.minHeap = new PriorityQueue((a, b) => b - a);
};

/**
 * @param {number} num
 * @return {void}
 */
MedianFinder.prototype.addNum = function(num) {
    // 我们的目标就是建立两个堆，一个大顶堆，一个小顶堆
    // 结合中位数的特点
    // 这两个堆需要满足:
    // 1. 大顶堆元素都比小顶堆小（由于堆的特点其实只要比较堆顶即可）
    // 2. 大顶堆元素不小于小顶堆，且最多比小顶堆多一个元素

    // 满足上面两个条件的话，如果想要找到中位数，就比较简单了
    // 如果两个堆数量相等（本质是总数为偶数），就两个堆顶元素的平均数
    // 如果两个堆数量不相等（本质是总数为奇数）， 就取大顶堆的堆顶元素

    // 问题如果保证满足上述两个特点

    // 1. 保证第一点
    this.maxHeap.enq(num);
    // 由于小顶堆的所有数都来自大顶堆的堆顶元素（最大值）
    // 因此可以保证第一点
    this.minHeap.enq(this.maxHeap.deq());

    // 2. 保证第二点
    if (this.maxHeap.size() < this.minHeap.size()){
        this.maxHeap.enq(this.minHeap.deq());
    }
};

/**
 * @return {number}
 */
MedianFinder.prototype.findMedian = function() {
    if (this.maxHeap.size() == this.minHeap.size()) return
```

```
    else return this.maxHeap.peek();
};

/**
 * Your MedianFinder object will be instantiated and called
 * var obj = new MedianFinder()
 * obj.addNum(num)
 * var param_2 = obj.findMedian()
 */
```

题目地址

https://leetcode.com/problems/remove-invalid-parentheses/description/

题目描述

```
Remove the minimum number of invalid parentheses in order t

Note: The input string may contain letters other than the p

Example 1:

Input: "()())()"
Output: ["()()()", "(())()"]
Example 2:

Input: "(a)())()"
Output: ["(a)()()", "(a())()"]
Example 3:

Input: ")("
Output: [""]
```

前置知识

- BFS
- 队列

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

我们的思路是先写一个函数用来判断给定字符串是否是有效的。 然后再写一个函数，这个函数 依次删除第i个字符，判断是否有效，有效则添加进最终的返回数组。

这样的话实现的功能就是， 删除 `一个` 小括号使之有效的所有可能。因此只需要递归调用 `依次删除第i个字符` 的功能就可以了。

而且由于题目要求是要删除最少的小括号，因此我们的思路是使用广度优先遍历，而不是深度有限的遍历。

> 没有动图，请脑补

关键点解析

- 广度优先遍历
- 使用队列简化操作
- 使用一个visited的mapper， 来避免遍历同样的字符串

代码

```
/*
 * @lc app=leetcode id=301 lang=javascript
 *
 * [301] Remove Invalid Parentheses
 *
 * https://leetcode.com/problems/remove-invalid-parentheses
 *
 * algorithms
 * Hard (38.52%)
 * Total Accepted:    114.3K
 * Total Submissions: 295.4K
 * Testcase Example:  '"()())()"'
 *
 * Remove the minimum number of invalid parentheses in orde
 * string valid. Return all possible results.
 *
 * Note: The input string may contain letters other than th
 * ).
 *
 * Example 1:
 *
 *
 * Input: "()())()"
 * Output: ["()()()", "(())()"]
 *
 *
 * Example 2:
 *
 *
 * Input: "(a)())()"
 * Output: ["(a)()()", "(a())()"]
 *
 *
 * Example 3:
 *
 *
 * Input: ")("
 * Output: [""]
 *
 */
var isValid = function(s) {
  let openParenthes = 0;
  for(let i = 0; i < s.length; i++) {
    if (s[i] === '(') {
      openParenthes++;
    } else if (s[i] === ')') {
      if (openParenthes === 0) return false;
      openParenthes--;
```

```
    }
  }
  return openParenthes === 0;
};
/**
 * @param {string} s
 * @return {string[]}
 */
var removeInvalidParentheses = function(s) {
  if (!s || s.length === 0) return [""];
  const ret = [];
  const queue = [s];
  const visited = {};
  let current = null;
  let removedParentheses = 0; // 只记录最小改动

  while ((current = queue.shift())) {
    let hit = isValid(current);
    if (hit) {
      if (!removedParentheses) {
       removedParentheses =  s.length - current.length
      }
      if (s.length - current.length > removedParentheses) r
      ret.unshift(current);
      continue;
    }
    for (let i = 0; i < current.length; i++) {
      if (current[i] !== ')' && current[i] !== '(') continu
      const subString = current.slice(0, i).concat(current.
      if (visited[subString]) continue;
      visited[subString] = true;
      queue.push(subString);
    }
  }

  return ret.length === 0 ? [""] : ret;
};
```

扩展

相似问题:

validParentheses

题目地址（312. 戳气球）

https://leetcode-cn.com/problems/burst-balloons/

题目描述

```
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

现在要求你戳破所有的气球。每当你戳破一个气球 i 时，你可以获得 nums[l

求所能获得硬币的最大数量。

说明:

你可以假设 nums[-1] = nums[n] = 1，但注意它们不是真实存在的所以并不
0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100
示例:

输入: [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
```

前置知识

- 回溯法
- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

回溯法

分析一下这道题，就是要戳破所有的气球，获得硬币的最大数量，然后左右两边的气球相邻了。我的第一反应就是暴力，回溯法。

但是肯定会超时，为什么呢？因为题目给的气球数量有点多，最多 500 个；500 的阶乘，会超时爆栈；但是我们依然写一下代码，找下突破口，小伙伴们千万不要看不起暴力，暴力是优化的突破口；

如果小伙伴对回溯法不太熟悉，我建议你记住下面的模版，也可以看我之前写的文章，回溯法基本可以使用以下的模版写。回溯法省心省力，0 智商负担。

代码

```
var maxCoins = function (nums) {
  let res = Number.MIN_VALUE;
  backtrack(nums, 0);
  return res;
  // 回溯法，状态树很大
  function backtrack(nums, score) {
    if (nums.length == 0) {
      res = Math.max(res, score);
      return;
    }
    for (let i = 0, n = nums.length; i < n; i++) {
      let point =
        (i - 1 < 0 ? 1 : nums[i - 1]) *
        nums[i] *
        (i + 1 >= n ? 1 : nums[i + 1]);
      let tempNums = [].concat(nums);
      // 做选择 在 nums 中删除元素 nums[i]
      nums.splice(i, 1);
      // 递归回溯
      backtrack(nums, score + point);
      // 撤销选择
      nums = [...tempNums];
    }
  }
};
```

动态规划

回溯法的缺点也很明显，复杂度很高，对应本题戳气球；小伙伴们可以脑补一下执行过程的状态树，这里我偷个懒就不画了；通过仔细观察这个状态树，我们会发现这个状态树的【选择】上，会有一些重复的选择分支；很明显存在了重复子问题；自然我就想到了能不能用动态规划来解决；

判读能不能用动态规划解决，还有一个问题，就是必须存在最优子结构；什么意思呢？其实就是根据局部最优，推导出答案；假设我们戳破第 k 个气球是最优策略的最后一步，和上一步有没有联系呢？根据题目意思，戳

破第 k 个，前一个和后一个就变成相邻的了，看似是会有联系，其实是没有的。因为戳破第 k 个和 k-1 个是没有联系的，脑补一下回溯法的状态树就更加明确了；

既然用动态规划，那就老套路了，把动态规划的三个问题想清楚定义好；然后找出题目的【状态】和【选择】，然后根据【状态】枚举，枚举的过程中根据【选择】计算递推就能得到答案了。

那本题的【选择】是什么呢？就是戳哪一个气球。那【状态】呢？就是题目给的气球数量。

1. 定义状态
2. 这里有个细节，就是题目说明有两个虚拟气球，nums[-1] = nums[n] = 1；如果当前戳破的气球是最后一个或者第一个，前面/后面没有气球了，不能乘以 0，而是乘以 1。
3. 定义状态的最关键两个点，往子问题（问题规模变小）想，最后一步最优策略是什么；我们假设最后戳破的气球是 k，戳破 k 获得最大数量的银币就是 nums[i] *nums[k]* nums[j] 再加上前面戳破的最大数量和后面的最大数量，即：nums[i] *nums[k]* nums[j] + 前面最大数量 + 后面最大数量，就是答案。

> 注意 i 不一定是 k - 1，同理 j 也不一定是 k + 1，因此可能 i - 1 和 i + 1 已经被戳破了。

- 而如果我们不考虑两个虚拟气球而直接定义状态，戳到最后两个气球的时候又该怎么定义状态来避免和前面的产生联系呢？这两个虚拟气球就恰到好处了，这也是本题的一个难点之一。
- 那我们可以这样来定义状态，dp[i][j] = x 表示戳破气球 i 和气球 j 之间（开区间，不包括 i 和 j）的所有气球，可以获得的最大硬币数为 x。为什么开区间？因为不能和已经计算过的产生联系，我们这样定义之后，利用两个虚拟气球，戳到最后两个气球的时候就完美的避开了所有状态的联系。
- 状态转移方程
- 而对于 dp[i][j]，i 和 j 之间会有很多气球，到底该戳哪个先呢？我们直接设为 k，枚举选择最优的 k 就可以了。
    - 1。
- 所以，最终的状态转移方程为：dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] + nums[k] *nums[i]* nums[j])。由于是开区间，因此 k 为 i + 1, i + 2... j - 1。
- 初始值和边界
- 由于我们利用了两个虚拟气球，边界就是气球数 n + 2

- 初始值，当 i == j 时，很明显两个之间没有气球，所有为 0；
- 如何枚举状态
- 因为我们最终要求的答案是 dp[0][n + 1]，就是戳破虚拟气球之间的所有气球获得的最大值；
- 当 i == j 时，i 和 j 之间是没有气球的，所以枚举的状态很明显是 dp table 的左上部分，也就是 j 大于 i，如下图所示，只给出一部分方便思考。

	1	2	3	4	5	6
0	0					dp[0][n+1]
1		0		dp[k][i]	dp[i][j]	
2			0		dp[j][k]	
3				0		

（图有错误。图中 dp[k][i] 应该是 dp[i][k]，dp[j][k] 应该是 dp[k][j]）

> 从上图可以看出，我们需要从下到上，从左到右进行遍历。

代码

代码支持： JS, Python

JS Code:

```
var maxCoins = function (nums) {
  let n = nums.length;
  // 添加两侧的虚拟气球
  let points = [1, ...nums, 1];
  let dp = Array.from(Array(n + 2), () => Array(n + 2).fil
  // 最后一行开始遍历,从下往上
  for (let i = n; i >= 0; i--) {
    // 从左往右
    for (let j = i + 1; j < n + 2; j++) {
      for (let k = i + 1; k < j; k++) {
        dp[i][j] = Math.max(
          dp[i][j],
          points[j] * points[k] * points[i] + dp[i][k] + d
        );
      }
    }
  }
  return dp[0][n + 1];
};
```

Python Code:

```
class Solution:
    def maxCoins(self, nums: List[int]) -> int:
        n = len(nums)
        points = [1] + nums + [1]
        dp = [[0] * (n + 2) for _ in range(n + 2)]

        for i in range(n, -1, -1):
            for j in range(i + 1, n + 2):
                for k in range(i + 1, j):
                    dp[i][j] = max(dp[i][j], dp[i][k] + dp
        return dp[0][-1]
```

复杂度分析

- 时间复杂度：$O(N^3)$
- 空间复杂度：$O(N^2)$

总结

简单的 dp 题目会直接告诉你怎么定义状态，告诉你怎么选择计算，你只需要根据套路判断一下能不能用 dp 解题即可，而判断能不能，往往暴力就是突破口。而困难点的 dp，我觉的都是细节问题了，要注意的细节太多了。感觉力扣加加，路西法大佬，把我领进了动态规划的大门，共勉。

更多题解可以访问我的LeetCode题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经30K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址（335. 路径交叉）

https://leetcode-cn.com/problems/self-crossing/

题目描述

```
给定一个含有 n 个正数的数组 x。从点 (0,0) 开始，先向北移动 x[0] 米，

编写一个 O(1) 空间复杂度的一趟扫描算法，判断你所经过的路径是否相交。



示例 1:

┌────┐
│    │
└────┼──>
     │

输入: [2,1,1,2]
输出: true
示例 2:

┌───────┐
│       │
│
│
└───────────>

输入: [1,2,3,4]
输出: false
示例 3:

┌────┐
│    │
└────┼>

输入: [1,1,1,1]
输出: true
```

前置知识

- 滑动窗口

公司

- 暂无

思路

符合直觉的做法是$O(N)$时间和空间复杂度的算法。这种算法非常简单，但是题目要求我们使用空间复杂度为$O(1)$的做法。

关于空间复杂度为$O(N)$的算法可以参考我之前的874.walking-robot-simulation。 思路基本是类似，只不过 obstacles（障碍物）不是固定的，而是我们不断遍历的时候动态生成的，我们每遇到一个点，就将其标记为 obstacle。随着算法的进行，我们的 obstacles 逐渐增大，最终和 N 一个量级。

我们考虑进行优化。我们仔细观察发现，如果想让其不相交，从大的范围来看只有两种情况：

1. 我们画的圈不断增大。
2. 我们画的圈不断减少。

（有没有感觉像迷宫？）

这样我们会发现，其实我们画最新一笔的时候，并不是之前画的所有的都需要考虑，我们只需要最近的几个就可以了，实际上是最近的五个，不过不知道也没关系，我们稍后会讲解。

红色部分指的是我们需要考虑的，而剩余没有被红色标注的部分则无需考虑。不是因为我们无法与之相交，而是我们 一旦与之相交，则必然我们也一定会与红色标记部分相交 。

然而我们画的方向也是不用考虑的。比如我当前画的方向是从左到右，那和我画的方向是从上到下有区别么？在这里是没区别的，不信我帮你将上图顺时针旋转 90 度看一下：

顺时针旋转90度

方向对于我们考虑是否相交没有差别。

当我们仔细思考的时候，会发现其实相交的情况只有以下几种：

这个时候代码就呼之欲出了。

- 我们只需要遍历数组 x，假设当前是第 i 个元素。
- 如果 x[i] >= x[i - 2] and x[i - 1] <= x[i - 3]，则相交（第一种情况）
- 如果 x[i - 1] <= x[i - 3] and x[i - 2] <= x[i]，则相交（第二种情况）
- 如果 i > 3 and x[i - 1] == x[i - 3] and x[i] + x[i - 4] == x[i - 2]，则相交（第三种情况）
- 如果 i > 4 and x[i] + x[i - 4] >= x[i - 2] and x[i - 1] >= x[i - 3] - x[i - 5] \ and x[i - 1] <= x[i - 3] and x[i - 2] >= x[i - 4] and x[i - 3] >= x[i - 5] ，则相交（第四种情况）
- 否则不相交

关键点解析

- 一定要画图辅助
- 对于这种$O(1)$空间复杂度有固定的套路。常见的有：
- 直接修改原数组
- 滑动窗口（当前状态并不是和之前所有状态有关，而是仅和某几个有关）。

我们采用的是滑动窗口。但是难点就在于我们怎么知道当前状态和哪几个有关。对于这道题来说，画图或许可以帮助你打开思路。另外面试的时候说出$O(N)$的思路也不失为一个帮助你冷静分析问题的手段。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def isSelfCrossing(self, x: List[int]) -> bool:
        n = len(x)
        if n < 4:
            return False
        for i in range(3, n):
            if x[i] >= x[i - 2] and x[i - 1] <= x[i - 3]:
                return True
            if x[i - 1] <= x[i - 3] and x[i - 2] <= x[i]:
                return True
            if i > 3 and x[i - 1] == x[i - 3] and x[i] + x
                return True
            if i > 4 and x[i] + x[i - 4] >= x[i - 2] and x
                    and x[i - 1] <= x[i - 3] and x[i - 2] >
                return True
        return False
```

题目地址

https://leetcode.com/problems/lfu-cache/

题目描述

```
Design and implement a data structure for Least Frequently

get(key) - Get the value (will always be positive) of the k
put(key, value) - Set or insert the value if the key is not

Follow up:
Could you do both operations in O(1) time complexity?

Example:

LFUCache cache = new LFUCache( 2 /* capacity */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // returns 1
cache.put(3, 3);    // evicts key 2
cache.get(2);       // returns -1 (not found)
cache.get(3);       // returns 3.
cache.put(4, 4);    // evicts key 1.
cache.get(1);       // returns -1 (not found)
cache.get(3);       // returns 3
cache.get(4);       // returns 4
```

前置知识

- 链表
- HashMap

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

本题已被收录到我的新书中，敬请期待～

LFU（Least frequently used) 但内存容量满的情况下，有新的数据进来，需要更多空间的时候，就需要删除被访问频率最少的元素。

举个例子，比如说 cache 容量是 3，按顺序依次放入 `1，2，1，2，1，3` ， cache 已存满 3 个元素 (1，2，3)， 这时如果想放入一个新的元素 4 的时候，就需要腾出一个元素空间。 用 LFU，这里就淘汰 3， 因为 3 的次数只出现依次， 1 和 2 出现的次数都比 3 多。

题中 `get` 和 `put` 都是 `O(1)` 的时间复杂度，那么删除和增加都是 `O(1)` ，可以想到用双链表，和 `HashMap` ，用一个 `HashMap, nodeMap,` 保存当前 `key` ，和 `node{key, value, frequent}` 的映射。 这样 `get(key)` 的操作就是 `O(1)` . 如果要删除一个元素，那么就需要另一个 `HashMap, freqMap,` 保存元素出现次数 `(frequent)` 和双链表 `(DoublyLinkedlist)` 映射， 这里双链表存的是 frequent 相同的元素。每次 `get` 或 `put` 的时候， `frequent+1` ，然后把 `node` 插入到双链表的 `head node, head.next=node` 每次删除 `freqent` 最小的双链表的 `tail node, tail.prev` 。

用给的例子举例说明：

```
1. put(1, 1),
   - 首先查找 nodeMap 中有没有 key=1 对应的 value,
       没有就新建 node(key, value, freq) -> node1(1, 1, 1),
   - 查找 freqMap 中有没有 freq=1 对应的 value,
       没有就新建 doublylinkedlist(head, tail), 把 node1 插入
   如下图,
```

```
2. put(2, 2),
   - 首先查找 nodeMap 中有没有 key=2 对应的 value,
       没有就新建 node(key, value, freq) -> node2(2, 2, 1),
   - 查找 freqMap 中有没有 freq=1 对应的 value,
       没有就新建 doublylinkedlist(head, tail), 把 node2 插入
   如下图,
```

```
3. get(1),
   - 首先查找 nodeMap 中有没有 key=1 对应的 value, nodeMap:{[1
       找到 node1, 把 node1 freq+1 -> node1(1,1,2)
   - 更新 freqMap, 删除 freq=1, node1
   - 更新 freqMap, 插入 freq=2, node1
   如下图,
```

```
4. put(3, 3),
   - 判断 cache 的 capacity，已满，需要淘汰使用次数最少的元素，找
     如果 tailnode.prev != null，删除。然后从 nodeMap 中删除
   - 首先查找 nodeMap 中有没有 key=3 对应的 value，
     没有就新建 node(key, value, freq) -> node3(3, 3, 1),
   - 查找 freqMap 中有没有 freq=1 对应的 value，
     没有就新建 doublylinkedlist(head, tail)，把 node3 插入
   如下图，
```

```
5. get(2)
   - 查找 nodeMap，如果没有对应的 key 的 value，返回 -1。

6. get(3)
   - 首先查找 nodeMap 中有没有 key=3 对应的 value，nodeMap:{[1
       找到 node3，把 node3 freq+1 -> node3(3,3,2)
   - 更新 freqMap，删除 freq=1，node3
   - 更新 freqMap，插入 freq=2，node3
   如下图，
```

```
7. put(4, 4),
   - 判断 cache 的 capacity，已满，需要淘汰使用次数最少的元素，找
       如果 tailnode.prev != null，删除。然后从 nodeMap 中删除
   - 首先查找 nodeMap 中有没有 key=4 对应的 value，
       没有就新建 node(key, value, freq) -> node4(4, 4, 1),
   - 查找 freqMap 中有没有 freq=1 对应的 value，
       没有就新建 doublylinkedlist(head, tail)，把 node4 插入
   如下图，
```

```
8. get(1)
   - 查找 nodeMap，如果没有对应的 key 的 value，返回 -1。

9. get(3)
   - 首先查找 nodeMap 中有没有 key=3 对应的 value，nodeMap:{[4
     找到 node3，把 node3 freq+1 -> node3(3,3,3)
   - 更新 freqMap，删除 freq=2，node3
   - 更新 freqMap，插入 freq=3，node3
   如下图，
```

```
10. get(4)
    - 首先查找 nodeMap 中有没有 key=4 对应的 value, nodeMap:{[4
        找到 node4, 把 node4 freq+1 -> node4(4,4,2)
    - 更新 freqMap, 删除 freq=1, node4
    - 更新 freqMap, 插入 freq=2, node4
    如下图,
```

关键点分析

用两个 `Map` 分别保存 `nodeMap {key, node}` 和 `freqMap{frequent, DoublyLinkedList}` 。 实现 `get` 和 `put` 操作都是 `O(1)` 的时间复杂度。

可以用 Java 自带的一些数据结构，比如 HashLinkedHashSet，这样就不需要自己自建 Node，DoublelyLinkedList。 可以很大程度的缩减代码量。

代码（Java code）

```
public class LC460LFUCache {
  class Node {
    int key, val, freq;
    Node prev, next;

    Node(int key, int val) {
      this.key = key;
      this.val = val;
      freq = 1;
    }
  }

  class DoubleLinkedList {
    private Node head;
    private Node tail;
    private int size;

    DoubleLinkedList() {
      head = new Node(0, 0);
      tail = new Node(0, 0);
      head.next = tail;
      tail.prev = head;
    }

    void add(Node node) {
      head.next.prev = node;
      node.next = head.next;
      node.prev = head;
      head.next = node;
      size++;
    }

    void remove(Node node) {
      node.prev.next = node.next;
      node.next.prev = node.prev;
      size--;
    }

    // always remove last node if last node exists
    Node removeLast() {
      if (size > 0) {
        Node node = tail.prev;
        remove(node);
        return node;
      } else return null;
    }
  }
```

```
    // cache capacity
    private int capacity;
    // min frequent
    private int minFreq;
    Map<Integer, Node> nodeMap;
    Map<Integer, DoubleLinkedList> freqMap;
    public LC460LFUCache(int capacity) {
      this.minFreq = 0;
      this.capacity = capacity;
      nodeMap = new HashMap<>();
      freqMap = new HashMap<>();
    }

    public int get(int key) {
      Node node = nodeMap.get(key);
      if (node == null) return -1;
      update(node);
      return node.val;
    }

    public void put(int key, int value) {
      if (capacity == 0) return;
      Node node;
      if (nodeMap.containsKey(key)) {
        node = nodeMap.get(key);
        node.val = value;
        update(node);
      } else {
        node = new Node(key, value);
        nodeMap.put(key, node);
        if (nodeMap.size() == capacity) {
          DoubleLinkedList lastList = freqMap.get(minFreq);
          nodeMap.remove(lastList.removeLast().key);
        }
        minFreq = 1;
        DoubleLinkedList newList = freqMap.getOrDefault(node.
        newList.add(node);
        freqMap.put(node.freq, newList);
      }
    }

    private void update(Node node) {
      DoubleLinkedList oldList = freqMap.get(node.freq);
      oldList.remove(node);
      if (node.freq == minFreq && oldList.size == 0) minFreq-
      node.freq++;
      DoubleLinkedList newList = freqMap.getOrDefault(node.fr
      newList.add(node);
```

```
    freqMap.put(node.freq, newList);
  }
}
```

参考（References）

1. LFU(Least frequently used) Cache
2. Leetcode discussion mylzsd-Solution-Using-Two-HashMap-and-One-DoubleLinkedList)
3. Leetcode discussion aaaeeeo-Solution-Using-Two-HashMap-and-One-DoubleLinkedList)

题目地址（472. 连接词）

https://leetcode-cn.com/problems/concatenated-words/

题目描述

```
给定一个不含重复单词的列表，编写一个程序，返回给定单词列表中所有的连接

连接词的定义为：一个字符串完全是由至少两个给定数组中的单词组成的。

示例：

输入: ["cat","cats","catsdogcats","dog","dogcatsdog","hippo

输出: ["catsdogcats","dogcatsdog","ratcatdogcat"]

解释: "catsdogcats"由"cats", "dog" 和 "cats"组成;
     "dogcatsdog"由"dog", "cats"和"dog"组成;
     "ratcatdogcat"由"rat", "cat", "dog"和"cat"组成。
说明:

给定数组的元素总数不超过 10000。
给定数组中元素的长度总和不超过 600000。
所有输入字符串只包含小写字母。
不需要考虑答案输出的顺序。
```

前置知识

- 前缀树

公司

- 阿里
- 字节

思路

本题我的思路是直接使用前缀树来解决。**标准的前缀树模板**我在之前的题解中提到了，感兴趣的可以到下方的相关题目中查看。

这道题这里我们不需要 search，我们的做法是：

- 先进行一次遍历，将 words 全部插入（insert）到前缀树中。

- 然后再进行一次遍历，查找每一个单词有几个单词表中的单词组成
- 如果大于 2，则将其加入到 res 中
- 最后返回 res 即可

我们构造的前缀树大概是这样的：

问题的关键在于第二步中的**查找每一个单词有几个单词表中的单词组成**。其实如果你了解前缀树的话应该不难写出来。 比如查找 catsdogcats：

- 我们先从 c 到 a 到 t，发现 t 是单词结尾，我们数量 + 1
- 然后将剩下的部分“sdogcats”重新执行上述过程。
- s 发现找不到，我们返回 0
- 因此最终结果是 1

很明显这个逻辑是错误的，正确的划分应该是：

- 我们先从 c 到 a 到 t，再到 s，此时数量 + 1
- 然后将剩下的“dogcats”重复上述过程
- dog 找到了，数量 + 1
- 最后将 cats 加入。也找到了，数量再加 1

由于我们并不知道 cat 这里断开，结果更大？还是 cats 这里断开结果更大？因此我们的做法是将其全部递归求出，然后取出最大值即可。如果我们直接这样递归的话，可能会超时，卡在最后一个测试用例上。一个简单的方式是记忆化递归，从而避免重复计算，经测试这种方法能够通过。

代码

代码支持：Python3

Python3 Code:

```
class Trie:

    def __init__(self):
        self.Trie = {}
        self.visited = {}

    def insert(self, word):
        curr = self.Trie
        for w in word:
            if w not in curr:
                curr[w] = {}
            curr = curr[w]
        curr['#'] = 1

    def cntWords(self, word):
        if not word:
            return 0
        if word in self.visited:
            return self.visited[word]
        curr = self.Trie
        res = float('-inf')

        for i, w in enumerate(word):
            if w not in curr:
                return res
            curr = curr[w]
            if '#' in curr:
                res = max(res, 1 + self.cntWords(word[i + 1
        self.visited[word] = res
        return res


class Solution:
    def findAllConcatenatedWordsInADict(self, words: List[s
        self.trie = Trie()
        res = []

        for word in words:
            self.trie.insert(word)
        for word in words:
            if self.trie.cntWords(word) >= 2:
                res.append(word)
        return res
```

关键点分析

- 前缀树

相关题目

- 0208.implement-trie-prefix-tree
- 0211.add-and-search-word-data-structure-design
- 0212.word-search-ii
- 0820.short-encoding-of-words
- 1032.stream-of-characters

题目地址（488. 祖玛游戏）

https://leetcode-cn.com/problems/zuma-game/

题目描述

```
回忆一下祖玛游戏。现在桌上有一串球，颜色有红色(R)，黄色(Y)，蓝色(B)，

每一次，你可以从手里的球选一个，然后把这个球插入到一串球中的某个位置上

找到插入并可以移除掉桌上所有球所需的最少的球数。如果不能移除桌上所有的

示例:
输入: "WRRBBW", "RB"
输出: -1
解释: WRRBBW -> WRR[R]BBW -> WBBW -> WBB[B]W -> WW  (翻译者标

输入: "WWRRBBWW", "WRBRW"
输出: 2
解释: WWRRBBWW -> WWRR[R]BBWW -> WWBBWW -> WWBB[B]WW -> WWW

输入:"G", "GGGGG"
输出: 2
解释: G -> G[G] -> GG[G] -> empty

输入: "RBYYBBRRB", "YRBGB"
输出: 3
解释: RBYYBBRRB -> RBYY[Y]BBRRB -> RBBBRRB -> RRRB -> B ->
标注:

你可以假设桌上一开始的球中，不会有三个及三个以上颜色相同且连着的球。
桌上的球不会超过20个，输入的数据中代表这些球的字符串的名字是 "board"
你手中的球不会超过5个，输入的数据中代表这些球的字符串的名字是 "hand"。
输入的两个字符串均为非空字符串，且只包含字符 'R','Y','B','G','W'。
```

前置知识

- 回溯
- 哈希表
- 双指针

公司

- 百度

思路

面试题困难难度的题目常见的题型有：

- DP
- 设计题
- 图
- 游戏

本题就是游戏类题目。 如果你是一个前端， 说不定还会考察你如何实现一个 zuma 游戏。这种游戏类的题目，可以简单可以困难， 比如力扣经典的石子游戏，宝石游戏等。这类题目没有固定的解法。我做这种题目的思路就是先暴力模拟，再尝试优化算法瓶颈。

注意下数据范围球的数目 <= 5，因此暴力法就变得可行。基本思路是暴力枚举手上的球可以消除的地方， 我们可以使用回溯法来完成暴力枚举的过程，在回溯过程记录最小值即可。由于回溯树的深度不会超过 5，因此这种解法应该可以 AC。

上面提到的 `可以消除的地方` ，指的是**连续相同颜色 + 手上相同颜色的球大于等于 3**，这也是题目说明的消除条件。

因此我们只需要两个指针记录连续相同颜色球的位置，如果可以消除，消除即可。

如图，我们记录了连续红球的位置， 如果手上有红球， 则可以尝试将其清除，这一次决策就是回溯树（决策树）的一个分支。之后我们会撤回到这个决策分支， 尝试其他可行的决策分支。

以 board = RRBBRR ， hand 为 RRBB 为例，其决策树为：

其中虚线表示无需手动干预，系统自动消除。叶子节点末尾的黄色表示全部消除需要的手球个数。路径上的文字后面的数字表示此次消除需要的手球个数

> 如果你对回溯不熟悉，可以参考下我之前写的几篇题解：比如 46.permutations。

可以看出， 如果选择先消除中间的蓝色，则只需要一步即可完成。

关于计算连续球位置的核心代码(Python3):

```
i = 0
while i < len(board):
    j = i + 1
    while j < len(board) and board[i] == board[j]: j += 1
    # 其他逻辑

    # 更新左指针
    i = j
```

具体算法：

1. 用哈希表存储手上的球的种类和个数，这么做是为了后面**快速判断连续的球是否可以被消除**。由于题目限制手上求不会超过 5，因此哈希表的最大容量就是 5，可以认为这是一个常数的空间。
2. 回溯。

   2.1 确认可以消除的位置，算法参考上面的代码。

   2.2 判断手上是否有足够相同颜色的球可以消除。

   2.3 回溯的过程记录全局最小值。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def findMinStep(self, board: str, hand: str) -> int:
        def backtrack(board):
            if not board: return 0
            i = 0
            ans = 6
            while i < len(board):
                j = i + 1
                while j < len(board) and board[i] == board
                balls = 3 - (j - i)
                if counter[board[i]] >= balls:
                    balls = max(0, balls)
                    counter[board[i]] -= balls
                    ans = min(ans, balls + backtrack(board
                    counter[board[i]] += balls
                i = j
            return ans

        counter = collections.Counter(hand)
        ans = backtrack(board)
        return -1 if ans > 5 else ans
```

复杂度分析

- 时间复杂度：$O(2^(min(C,5)))$，其中 C 为连续相同颜色球的次数，比如 WWRRRR， C 就是 2， WRBDD， C 就是 4。min(C, 5) 是因为题目限定了手上球的个数不大于 5。
- 空间复杂度：$O(min(C,5)*Board)$，其中 C 为连续相同颜色球的次数，Board 为 Board 的长度。

关键点解析

- 回溯模板
- 双指针写法

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 36K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（493. 翻转对）

https://leetcode-cn.com/problems/reverse-pairs/description/

题目描述

```
给定一个数组 nums ，如果 i < j 且 nums[i] > 2*nums[j] 我们就将

你需要返回给定数组中的重要翻转对的数量。

示例 1:

输入: [1,3,2,3,1]
输出: 2
示例 2:

输入: [2,4,3,5,1]
输出: 3
注意:

给定数组的长度不会超过50000。
输入数组中的所有数字都在32位整数的表示范围内。
```

前置知识

- 归并排序
- 逆序数
- 分治

公司

- 阿里
- 百度
- 字节

暴力法

思路

读完这道题你应该就能联想到逆序数才行。求解逆序数最简单的做法是使用双层循环暴力求解。我们仿照求解决逆序数的解法来解这道题（其实唯一的区别就是系数从 1 变成了 2）。

代码

代码支持：Python3

Python3 Code:

```
class Solution(object):
    def reversePairs(self, nums):
        n = len(nums)
        cnt = 0
        for i in range(n):
            for j in range(i + 1, n):
                if nums[i] > 2 * nums[j]:
                    cnt += 1
        return cnt
```

分治法

思路

如果你能够想到逆序数，那么你很可能直到使用类似归并排序的方法可以求解逆序数。实际上逆序数只是归并排序的副产物而已。

我们在正常的归并排序的代码中去计算逆序数即可。由于每次分治的过程，左右两段数组分别是有序的，因此我们可以减少一些运算。 从时间复杂度的角度上讲，我们从$O(N^2)$优化到了 $O(NlogN)$。

具体来说，对两段有序的数组，有序数组内部是不需要计算逆序数的。我们计算逆序数的逻辑只是计算两个数组之间的逆序数，我们假设两个数组是 A 和 B，并且 A 数组最大的元素不大于 B 数组最小的元素。而要做到这样，我们只需要常规的归并排序即可。

接下来问题转化为求解两个有序数组之间的逆序数，并且两个有序数组之间满足关系 `A数组最大的元素不大于B数组最小的元素` 。

关于计算逆序数的核心代码(Python3):

```
l = r = 0
while l < len(left) and r < len(right):
    if left[l] <= 2 * right[r]:
        l += 1
    else:
        self.cnt += len(left) - l
        r += 1
```

代码

代码支持：Python3

Python3 Code:

```
class Solution(object):
    def reversePairs(self, nums):
        self.cnt = 0

        def mergeSort(lst):
            L = len(lst)
            if L <= 1:
                return lst
            return mergeTwo(mergeSort(lst[:L//2]), mergeSo

        def mergeTwo(left, right):
            l = r = 0
            while l < len(left) and r < len(right):
                if left[l] <= 2 * right[r]:
                    l += 1
                else:
                    self.cnt += len(left) - l
                    r += 1
            return sorted(left+right)

        mergeSort(nums)
        return self.cnt
```

对于具体的排序过程我们偷懒直接使用了语言内置的方法 sorted，这在很多时候是有用的，即使你是参加面试，这种方式通常也是允许的。省略非核心的考点，可以使得问题更加聚焦，这是一种解决问题的思路，在工作中也很有用。

关键点解析

- 归并排序

- 逆序数
- 分治
- 识别考点，其他非重点可以使用语言内置方法

扩展

这道题还有很多别的解法，感性的可以参考下这个题解 General principles behind problems similar to "Reverse Pairs"

题目地址

https://leetcode.com/problems/super-egg-drop/description/

题目描述

```
You are given K eggs, and you have access to a building wit

Each egg is identical in function, and if an egg breaks, yo

You know that there exists a floor F with 0 <= F <= N such

Each move, you may take an egg (if you have an unbroken one

Your goal is to know with certainty what the value of F is.

What is the minimum number of moves that you need to know w



Example 1:

Input: K = 1, N = 2
Output: 2
Explanation:
Drop the egg from floor 1.  If it breaks, we know with cert
Otherwise, drop the egg from floor 2.  If it breaks, we kno
If it didn't break, then we know with certainty F = 2.
Hence, we needed 2 moves in the worst case to know what F i
Example 2:

Input: K = 2, N = 6
Output: 3
Example 3:

Input: K = 3, N = 14
Output: 4


Note:

1 <= K <= 100
1 <= N <= 10000
```

前置知识

- 动态规划

思路

> 本题已经重制，重制版更清晰 《丢鸡蛋问题》重制版来袭～

这是一道典型的动态规划题目，但是又和一般的动态规划不一样。

拿题目给的例子为例，两个鸡蛋，六层楼，我们最少扔几次?

一个符合直觉的做法是，建立 dp[i][j], 代表 i 个鸡蛋，j 层楼最少扔几次，然后我们取 dp[K][n]即可。

代码大概这样的：

```
const dp = Array(K + 1);
dp[0] = Array(N + 1).fill(0);
for (let i = 1; i < K + 1; i++) {
  dp[i] = [0];
  for (let j = 1; j < N + 1; j++) {
    // 只有一个鸡蛋
    if (i === 1) {
      dp[i][j] = j;
      continue;
    }
    // 只有一层楼
    if (j === 1) {
      dp[i][j] = 1;
      continue;
    }

    // 每一层我们都模拟一遍
    const all = [];
    for (let k = 1; k < j + 1; k++) {
      const brokenCount = dp[i - 1][k - 1]; // 如果碎了
      const notBrokenCount = dp[i][j - k]; // 如果没碎
      all.push(Math.max(brokenCount, notBrokenCount)); // 
    }
    dp[i][j] = Math.min(...all) + 1; // 最坏的集合中我们取最好
  }
}

return dp[K][N];
```

果不其然，当我提交的时候，超时了。 这个的时复杂度是很高的，可以看到，我们内层暴力的求解所有可能，然后 取最好的，这个过程非常耗时，大概是 O(N^2 * K).

然后我看了一位 leetcode网友的回答, 他的想法是 `dp[M][K]means that, given K eggs and M moves, what is the maximum number of floor that we can check.`

我们按照他的思路重新建模：

可以看到右下角的部分根本就不需要计算，从而节省很多时间

关键点解析

- dp 建模思路要发生变化，即 `dp[M][K]means that, given K eggs and M moves, what is the maximum number of floor that we can check.`

代码

```
/**
 * @param {number} K
 * @param {number} N
 * @return {number}
 */
var superEggDrop = function (K, N) {
  // 不选择dp[K][M]的原因是dp[M][K]可以简化操作
  const dp = Array(N + 1)
    .fill(0)
    .map((_) => Array(K + 1).fill(0));

  let m = 0;
  while (dp[m][K] < N) {
    m++;
    for (let k = 1; k <= K; ++k) dp[m][k] = dp[m - 1][k - 1
  }
  return m;
};
```

题目地址（895. 最大频率栈）

https://leetcode-cn.com/problems/maximum-frequency-stack/

题目描述

```
实现 FreqStack，模拟类似栈的数据结构的操作的一个类。

FreqStack 有两个函数：

push(int x)，将整数 x 推入栈中。
pop()，它移除并返回栈中出现最频繁的元素。
如果最频繁的元素不只一个，则移除并返回最接近栈顶的元素。


示例：

输入：
["FreqStack","push","push","push","push","push","push","pop
[[],[5],[7],[5],[7],[4],[5],[],[],[],[]]
输出：[null,null,null,null,null,null,null,5,7,5,4]
解释：
执行六次 .push 操作后，栈自底向上为 [5,7,5,7,4,5]。然后：

pop() -> 返回 5，因为 5 是出现频率最高的。
栈变成 [5,7,5,7,4]。

pop() -> 返回 7，因为 5 和 7 都是频率最高的，但 7 最接近栈顶。
栈变成 [5,7,5,4]。

pop() -> 返回 5 。
栈变成 [5,7,4]。

pop() -> 返回 4 。
栈变成 [5,7]。


提示：

对 FreqStack.push(int x) 的调用中 0 <= x <= 10^9。
如果栈的元素数目为零，则保证不会调用  FreqStack.pop()。
单个测试样例中，对 FreqStack.push 的总调用次数不会超过 10000。
单个测试样例中，对 FreqStack.pop 的总调用次数不会超过 10000。
所有测试样例中，对 FreqStack.push 和 FreqStack.pop 的总调用次数不
```

前置知识

- 栈
- 哈希表

公司

- 暂无

思路

我们以题目给的例子来讲解。

- 使用fraq 来存储对应的数字出现次数。key 是数字，value频率

- 由于题目限制“如果最频繁的元素不只一个，则移除并返回最接近栈顶的元素。”，我们考虑使用栈来维护一个频率表 fraq_stack。key是频率，value是数字组成的栈。

- 同时用max_fraq 记录当前的最大频率值。
- 第一次pop的时候，我们最大的频率是3。由fraq_stack 知道我们需要pop掉5。

- 之后pop依次是这样的（红色数字表示顺序）

关键点解析

- 栈的基本性质
- hashtable的基本性质
- push和pop的时候同时更新fraq，max_fraq 和 fraq_stack。

代码

```python
class FreqStack:

    def __init__(self):
        self.fraq = collections.defaultdict(lambda: 0)
        self.fraq_stack = collections.defaultdict(list)
        self.max_fraq = 0

    def push(self, x: int) -> None:
        self.fraq[x] += 1
        if self.fraq[x] > self.max_fraq:
            self.max_fraq = self.fraq[x]
        self.fraq_stack[self.fraq[x]].append(x)

    def pop(self) -> int:
        ans = self.fraq_stack[self.max_fraq].pop()
        self.fraq[ans] -= 1
        if not self.fraq_stack[self.max_fraq]:
            self.max_fraq -= 1
        return ans

# Your FreqStack object will be instantiated and called as
# obj = FreqStack()
# obj.push(x)
# param_2 = obj.pop()
```

复杂度分析

- 时间复杂度：push 和 pop 平均时间复杂度是 $O(1)$
- 空间复杂度：$O(N)$，其中N为数字的总数。

大家也可以关注我的公众号《脑洞前端》获取更多更新鲜的LeetCode题解

题目地址(1032. 字符流)

https://leetcode-cn.com/problems/stream-of-characters/

题目描述

```
按下述要求实现 StreamChecker 类：

StreamChecker(words)：构造函数，用给定的字词初始化数据结构。
query(letter)：如果存在某些 k >= 1，可以用查询的最后 k个字符（按从


示例：

StreamChecker streamChecker = new StreamChecker(["cd","f","
streamChecker.query('a');          // 返回 false
streamChecker.query('b');          // 返回 false
streamChecker.query('c');          // 返回 false
streamChecker.query('d');          // 返回 true，因为 'cd' 在
streamChecker.query('e');          // 返回 false
streamChecker.query('f');          // 返回 true，因为 'f' 在
streamChecker.query('g');          // 返回 false
streamChecker.query('h');          // 返回 false
streamChecker.query('i');          // 返回 false
streamChecker.query('j');          // 返回 false
streamChecker.query('k');          // 返回 false
streamChecker.query('l');          // 返回 true，因为 'kl' 在


提示：

1 <= words.length <= 2000
1 <= words[i].length <= 2000
字词只包含小写英文字母。
待查项只包含小写英文字母。
待查项最多 40000 个。
```

前置知识

- 前缀树

公司

- 字节

思路

题目要求 `按从旧到新顺序` 查询，因此可以将从旧到新的 query 存起来形成一个单词 stream。

比如：

```
streamChecker.query("a"); // stream:  a
streamChecker.query("b"); // stream: ba
streamChecker.query("c"); // stream: cba
```

这里有两个小的点需要注意：

1. 如果用数组来存储， 由于每次都往数组头部插入一个元素，因此每次 query 操作的时间复杂度为 $O(N)$，其中 N 为截止当前执行 query 的次数，我们可以使用双端队列进行优化。
2. 由于不必 query 形成的查询全部命中。比如 stream 为 cba 的时候，找到单词 c， bc， abc 都是可以的。如果是找到 c，cb，cba 比较好吧，现在是反的。其实我们可以反序插入是，类似的技巧在211.add-and-search-word-data-structure-design 也有用到。

之后我们用拼接的单词在 words 中查询即可， 最简单的方式当然是每次 query 都去扫描一次，这种方式毫无疑问会超时。

我们可以采用构建 Trie 的形式，即已空间环时间， 其代码和常规的 Trie 类似，只需要将 search(word) 函数做一个简单修改即可，我们不需要检查整个 word 是否存在， 而已 word 的前缀存在即可。

> 提示：可以通过对 words 去重，来用空间换区时间。

具体算法：

- init 中 构建 Trie 和 双端队列 stream
- query 时，往 stream 的左边 append 即可。
- 调用 Trie 的 search（和常规的 search 稍有不同， 我上面已经讲了）

核心代码（Python）：

```python
class StreamChecker:

    def __init__(self, words: List[str]):
        self.trie = Trie()
        self.stream = deque([])

        for word in set(words):
            self.trie.insert(word[::-1])

    def query(self, letter: str) -> bool:
        self.stream.appendleft(letter)
        return self.trie.search(self.stream)
```

关键点解析

- 前缀树模板
- 倒序插入

代码

- 语言支持: Python

Python Code:

```python
class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.Trie = {}

    def insert(self, word):
        """
        Inserts a word into the trie.
        :type word: str
        :rtype: void
        """
        curr = self.Trie
        for w in word:
            if w not in curr:
                curr[w] = {}
            curr = curr[w]
        curr['#'] = 1

    def search(self, word):
        """
        Returns if the word is in the trie.
        :type word: str
        :rtype: bool
        """
        curr = self.Trie
        for w in word:
            if w not in curr:
                return False
            if "#" in curr[w]:
                return True
            curr = curr[w]
        return False


class StreamChecker:

    def __init__(self, words: List[str]):
        self.trie = Trie()
        self.stream = deque([])

        for word in set(words):
            self.trie.insert(word[::-1])

    def query(self, letter: str) -> bool:
```

```
        self.stream.appendleft(letter)
        return self.trie.search(self.stream)
```

相关题目

- 0208.implement-trie-prefix-tree
- 0211.add-and-search-word-data-structure-design
- 0212.word-search-ii
- 0472.concatenated-words
- 0820.short-encoding-of-words

题目地址

https://leetcode.com/problems/optimize-water-distribution-in-a-village/

题目描述

```
There are n houses in a village. We want to supply water fo

For each house i, we can either build a well inside it dire

Find the minimum total cost to supply water to all houses.

Example 1:

Input: n = 3, wells = [1,2,2], pipes = [[1,2,1],[2,3,1]]
Output: 3
Explanation:
The image shows the costs of connecting houses using pipes.
The best strategy is to build a well in the first house wit

Constraints:

1 <= n <= 10000
wells.length == n
0 <= wells[i] <= 10^5
1 <= pipes.length <= 10000
1 <= pipes[i][0], pipes[i][1] <= n
0 <= pipes[i][2] <= 10^5
pipes[i][0] != pipes[i][1]
```

前置知识

- 图
- 最小生成树

公司

- 暂无

思路

题意，在每个城市打井需要一定的花费，也可以用其他城市的井水，城市之间建立连接管道需要一定的花费，怎么样安排可以花费最少的前灌溉所有城市。

这是一道连通所有点的最短路径/最小生成树问题，把城市看成图中的点，管道连接城市看成是连接两个点之间的边。这里打井的花费是直接在点上，而且并不是所有 点之间都有边连接，为了方便，我们可以假想一个点 `（root）0` ，这里自身点的花费可以与 `0` 连接，花费可以是 `0-i` 之间的花费。这样我们就可以构建一个连通图包含所有的点和边。 那在一个连通图中求最短路径/最小生成树的问题.

参考延伸阅读中，维基百科针对这类题给出的几种解法。

解题步骤：

1. 创建 `POJO EdgeCost(node1, node2, cost) - 节点1 和 节点2 连接边的花费` 。
2. 假想一个 `root` 点 `0` ，构建图
3. 连通所有节点和 `0` ， `[0,i] - i 是节点 [1,n]` ， `0-1` 是节点 `0` 和 `1` 的边，边的值是节点 `i` 上打井的花费 `wells[i]` ；
4. 把打井花费和城市连接点转换成图的节点和边。
5. 对图的边的值排序（从小到大）
6. 遍历图的边，判断两个节点有没有连通 （ `Union-Find` ），
    - 已连通就跳过，继续访问下一条边
    - 没有连通，记录花费，连通节点
7. 若所有节点已连通，求得的最小路径即为最小花费，返回

8. 对于每次 `union` ，节点数 `n-1` ，如果 `n==0` 说明所有节点都已连通，可以提前退出，不需要继续访问剩余的边。

> 这里用加权Union-Find 判断两个节点是否连通，和连通未连通的节点。

举例： `n = 5, wells=[1,2,2,3,2], pipes=[[1,2,1],[2,3,1],[4,5,7]]`

如图：

从图中可以看到，最后所有的节点都是连通的。

复杂度分析

- *时间复杂度:* `O(ElogE) - E 是图的边的个数`
- *空间复杂度:* `O(E)`

> 一个图最多有 `n(n-1)/2 - n 是图中节点个数` 条边 （完全连通图）

关键点分析

1. 构建图，得出所有边
2. 对所有边排序
3. 遍历所有的边（从小到大）
4. 对于每条边，检查是否已经连通，若没有连通，加上边上的值，连通两个节点。若已连通，跳过。

代码 (`Java/Python3`)

Java code

```
class OptimizeWaterDistribution {
  public int minCostToSupplyWater(int n, int[] wells, int
    List<EdgeCost> costs = new ArrayList<>();
    for (int i = 1; i <= n; i++) {
      costs.add(new EdgeCost(0, i, wells[i - 1]));
    }
    for (int[] p : pipes) {
      costs.add(new EdgeCost(p[0], p[1], p[2]));
    }
    Collections.sort(costs);
    int minCosts = 0;
    UnionFind uf = new UnionFind(n);
    for (EdgeCost edge : costs) {
      int rootX = uf.find(edge.node1);
      int rootY = uf.find(edge.node2);
      if (rootX == rootY) continue;
      minCosts += edge.cost;
      uf.union(edge.node1, edge.node2);
      // for each union, we connnect one node
      n--;
      // if all nodes already connected, terminate early
      if (n == 0) {
        return minCosts;
      }
    }
    return minCosts;
  }

  class EdgeCost implements Comparable<EdgeCost> {
    int node1;
    int node2;
    int cost;
    public EdgeCost(int node1, int node2, int cost) {
      this.node1 = node1;
      this.node2 = node2;
      this.cost = cost;
    }

    @Override
    public int compareTo(EdgeCost o) {
      return this.cost - o.cost;
    }
  }

  class UnionFind {
    int[] parent;
    int[] rank;
    public UnionFind(int n) {
```

```
        parent = new int[n + 1];
        for (int i = 0; i <= n; i++) {
          parent[i] = i;
        }
        rank = new int[n + 1];
      }
      public int find(int x) {
        return x == parent[x] ? x : find(parent[x]);
      }
      public void union(int x, int y) {
        int px = find(x);
        int py = find(y);
        if (px == py) return;
        if (rank[px] >= rank[py]) {
          parent[py] = px;
          rank[px] += rank[py];
        } else {
          parent[px] = py;
          rank[py] += rank[px];
        }
      }
    }
  }
```

Pythong3 code

```
class Solution:
    def minCostToSupplyWater(self, n: int, wells: List[int]
        union_find = {i: i for i in range(n + 1)}

        def find(x):
            return x if x == union_find[x] else find(union_

        def union(x, y):
            px = find(x)
            py = find(y)
            union_find[px] = py

        graph_wells = [[cost, 0, i] for i, cost in enumerat
        graph_pipes = [[cost, i, j] for i, j, cost in pipes
        min_costs = 0
        for cost, x, y in sorted(graph_wells + graph_pipes)
            if find(x) == find(y):
                continue
            union(x, y)
            min_costs += cost
            n -= 1
            if n == 0:
                return min_costs
```

延伸阅读

1. 最短路径问题
2. Dijkstra算法
3. Floyd-Warshall算法
4. Bellman-Ford算法
5. Kruskal算法
6. Prim's 算法
7. 最小生成树

题目地址（1449. 数位成本和为目标值的最大数字）

https://leetcode-cn.com/problems/form-largest-integer-with-digits-that-add-up-to-target/

题目描述

```
给你一个整数数组 cost 和一个整数 target 。请你返回满足如下规则可以得

给当前结果添加一个数位（i + 1）的成本为 cost[i] （cost 数组下标从 0
总成本必须恰好等于 target 。
添加的数位中没有数字 0 。
由于答案可能会很大，请你以字符串形式返回。

如果按照上述要求无法得到任何整数，请你返回 "0" 。



示例 1：

输入：cost = [4,3,2,5,6,7,2,5,5], target = 9
输出："7772"
解释：添加数位 '7' 的成本为 2 ，添加数位 '2' 的成本为 3 。所以 "777
 数字      成本
  1  ->   4
  2  ->   3
  3  ->   2
  4  ->   5
  5  ->   6
  6  ->   7
  7  ->   2
  8  ->   5
  9  ->   5
示例 2：

输入：cost = [7,6,5,5,5,6,8,7,8], target = 12
输出："85"
解释：添加数位 '8' 的成本是 7 ，添加数位 '5' 的成本是 5 。"85" 的成
示例 3：

输入：cost = [2,4,6,2,4,6,4,4,4], target = 5
输出："0"
解释：总成本是 target 的条件下，无法生成任何整数。
示例 4：

输入：cost = [6,10,15,40,40,40,40,40,40], target = 47
输出："32211"


提示：

cost.length == 9
1 <= cost[i] <= 5000
1 <= target <= 5000
```

前置知识

- 数组
- 动态规划
- 背包问题

公司

- 暂无

思路

由于数组可以重复选择，因此这是一个完全背包问题。

01 背包

对于 01 背包问题，我们的套路是：

```
for i in 0 to N:
    for j in 1 to V + 1:
        dp[j] = max(dp[j], dp[j - cost[i])
```

而一般我们为了处理边界问题，我们一般会这么写代码：

```
for i in 1 to N + 1:
    # 这里是倒序的，原因在于这里是01背包。
    for j in V to 0:
        dp[j] = max(dp[j], dp[j - cost[i - 1])
```

其中 dp[j] 表示只能选择前 i 个物品，背包容量为 j 的情况下，能够获得的最大价值。

> dp[j] 不是没 i 么？ 其实我这里 i 指的是 dp[j]当前所处的循环中的 i 值

完全背包问题

回到问题，我们这是完全背包问题:

```
for i in 1 to N + 1:
    # 这里不是倒序，原因是我们这里是完全背包问题
    for j in 1 to V + 1:
        dp[j] = max(dp[j], dp[j - cost[i - 1])
```

为什么 01 背包需要倒序，而完全背包则不可以

实际上，这是一个骚操作，我来详细给你讲一下。

其实要回答这个问题，我要先将 01 背包和完全背包退化二维的情况。

对于 01 背包：

```
for i in 1 to N + 1:
    for j in V to 0:
        dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - cost[i -
```

注意等号左边是 i，右边是 i - 1，这很好理解，因为 i 只能取一次嘛。

那么如果我们不降序遍历会怎么样呢？

如图橙色部分表示已经遍历的部分，而让我们去用[j - cost[i - 1]] 往前面回溯的时候，实际上回溯的是 dp[i]j - cost[i - 1]]，而不是 dp[i - 1]j - cost[i - 1]]。

如果是降序就可以了，如图：

这个明白的话，我们继续思考为什么完全背包就要不降序了呢？

我们还是像上面一样写出二维的代码：

```
for i in 1 to N + 1:
   for j in 1 to V + 1:
       dp[i][j] = max(dp[i - 1][j], dp[i][j - cost[i - 1])
```

由于 i 可以取无数次，那么正序遍历正好可以满足，如上图。

恰好装满 VS 可以不装满

题目有两种可能，一种是要求背包恰好装满，一种是可以不装满（只要不超过容量就行）。而本题是要求 `恰好装满` 的。而这两种情况仅仅影响我们 `dp数组初始化` 。

- 恰好装满。只需要初始化 dp[0] 为 0， 其他初始化为负数即可。
- 可以不装满。 只需要全部初始化为 0，即可，

原因很简单，我多次强调过 dp 数组本质上是记录了一个个自问题。 dp[0]是一个子问题，dp[1]是一个子问题。。。

有了上面的知识就不难理解了。 初始化的时候，我们还没有进行任何选择，那么也就是说 dp[0] = 0，因为我们可以通过什么都不选达到最大值 0。而 dp[1],dp[2]...则在当前什么都不选的情况下无法达成，也就是无解，因为为了区分，我们可以用负数来表示，当然你可以用任何可以区分的东西表示，比如 None。

回到本题

而这道题和普通的完全背包不一样，这个是选择一个组成的最大数。由小学数学知识 `一个数字的全排列中，按照数字降序排列是最大的` ，我这里用了一个骚操作，那就是 cost 从后往前遍历，因为后面表示的数字大。

代码

```
class Solution:
    def largestNumber(self, cost: List[int], target: int) -
        dp = [0] + [float('-inf')] * target
        for i in range(9, 0, -1):
            for j in range(1, target+1):
                if j >= cost[i - 1]:
                    dp[j] = max(dp[j], (dp[j-cost[i - 1]] >
        return str(dp[target]) if dp[target] > 0 else '0'
```

复杂度分析

- 时间复杂度：$O(target))$
- 空间复杂度：$O(target)$

扩展

最后贴几个我写过的背包问题，让大家看看历史是多么的相似。

除了dp[0]全部初始化一个不可能的解

Python3 Code:

```python
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        dp = [amount + 1] * (amount + 1)
        dp[0] = 0

        for i in range(1, amount + 1):
            for j in range(len(coins)):
                if i >= coins[j]:
                    dp[i] = min(dp[i], dp[i - coins[j]] + 1)

        return -1 if dp[-1] == amount + 1 else dp[-1]
```

复杂度分析

- 时间复杂度：$O(amonut * len(coins))$
- 空间复杂度：$O(amount)$

（322. 硬币找零(完全背包问题)）

> 这里内外循环和本题正好是反的，我只是为了"秀技"(好玩)，实际上在这里对答案并不影响。

Python Code:

```python
class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        dp = [0] * (amount + 1)
        dp[0] = 1

        for j in range(len(coins)):
            for i in range(1, amount + 1):
                if i >= coins[j]:
                    dp[i] += dp[i - coins[j]]

        return dp[-1]
```

（518. 零钱兑换 II）

> 这里内外循环和本题正好是反的，但是这里必须这么做，否则结果是不对的，具体可以点进去链接看我那个题解

所以这两层循环的位置起的实际作用是什么？ 代表的含义有什么不同？

本质上:

```
for i in 1 to N + 1:
    for j in V to 0:
        ...
```

这种情况选择物品 1 和物品 3（随便举的例子），是一种方式。选择物品 3 个物品 1（注意是有顺序的）是同一种方式。 **原因在于你是固定物品，去扫描容量。**

而:

```
for j in V to 0:
    for i in 1 to N + 1:
        ...
```

这种情况选择物品 1 和物品 3（随便举的例子），是一种方式。选择物品 3 个物品 1（注意是有顺序的）也是一种方式。**原因在于你是固定容量，去扫描物品。**

因此总的来说，如果你认为[1,3]和[3,1]是一种，那么就用方法 1 的遍历，否则用方法 2。

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

大家也可以关注我的公众号《脑洞前端》获取更多更新鲜的 LeetCode 题解

后记

以上就是本电子书的全部内容了。 后期的话文章会第一时间在公众号和我的博客更新，并定期整理到这个电子书中来。

因此你可以关注我的公众号或者博客， 也可以关注我的同步电子书的网站 西法带你学算法 - 在线版 获得内容的更新。

如果想加入读者交流群， 在公众号回复 leetcode 即可，西法在群里等着你。

公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

博客地址：https://lucifer.ren/blog/