



91天 学算法

遇见更好的自己



01

基础篇

数组，队列，栈
链表
树与递归
哈希表
双指针

02

进阶篇

堆
前缀树
并查集
跳表
剪枝技巧
RK 和 KMP
高频面试题

03

专题篇

二分法
滑动窗口
位运算
背包问题
搜索（BFS, DFS, 回溯）

动态规划
分治
贪心

扫码关注了解更多



目录

第一章 - 先导篇	1.1
数据结构与算法概述	1.1.1
如何衡量算法的性能	1.1.2
如何使用本仓库	1.1.3
如何高效刷题	1.1.4
第二章 - 基础篇	1.2
【91 算法-基础篇】01.数组, 栈, 队列	1.2.1
【91 算法-基础篇】02.链表	1.2.2
【91 算法-基础篇】03.树	1.2.3
【91 算法-基础篇】04.哈希表	1.2.4
【91 算法-基础篇】05.双指针	1.2.5
【91 算法-基础篇】06.图	1.2.6
【91 算法-基础篇】07.模拟, 枚举与递推	1.2.7
【91 算法-基础篇】08.排序	1.2.8
第三章 - 专题篇	1.3
【91 算法-专题篇】01.二分法	1.3.1
【91 算法-专题篇】02.滑动窗口	1.3.2
【91 算法-专题篇】03.搜索	1.3.3
【91 算法-专题篇】04.背包问题	1.3.4
【91 算法-专题篇】05.动态规划	1.3.5
【91 算法-专题篇】06.分治	1.3.6
【91 算法-专题篇】07.贪心	1.3.7
【91 算法-专题篇】08.位运算	1.3.8
第四章 - 进阶篇	1.4
【91 算法-进阶篇】01.Trie	1.4.1
【91 算法-进阶篇】02.并查集	1.4.2
【91 算法-进阶篇】03.剪枝	1.4.3
【91 算法-进阶篇】04.字符串匹配	1.4.4
【91 算法-进阶篇】05.堆	1.4.5
【91 算法-进阶篇】06.跳表	1.4.6
基础篇	1.5
 基础篇	1.5.1

989. 数组形式的整数加法

989. 数组形式的整数加法	1.5.1.1
821. 字符的最短距离	1.5.1.2
1381. 设计一个支持增量操作的栈	1.5.1.3
394. 字符串解码	1.5.1.4
232. 用栈实现队列	1.5.1.5
768. 最多能完成排序的块 II	1.5.1.6
61. 旋转链表	1.5.1.7
24. 两两交换链表中的节点	1.5.1.8
109. 有序链表转换二叉搜索树	1.5.1.9
160. 相交链表	1.5.1.10
142. 环形链表 II	1.5.1.11
146. LRU 缓存机制	1.5.1.12
104. 二叉树的最大深度	1.5.1.13
100. 相同的树	1.5.1.14
129. 求根到叶子节点数字之和	1.5.1.15
513. 找树左下角的值	1.5.1.16
297. 二叉树的序列化与反序列化	1.5.1.17
987. 二叉树的垂序遍历	1.5.1.18
两数之和	1.5.1.19
347. 前 K 个高频元素	1.5.1.20
447. 回旋镖的数量	1.5.1.21
3. 无重复字符的最长子串	1.5.1.22
30. 串联所有单词的子串	1.5.1.23
Delete Sublist to Make Sum Divisible By K	1.5.1.24
876. 链表的中间结点	1.5.1.25
26. 删除排序数组中的重复项	1.5.1.26
35. 搜索插入位置	1.5.1.27
239. 滑动窗口最大值	1.5.1.28
专题篇	1.5.2
69. x 的平方根	1.5.2.1
278. 第一个错误的版本	1.5.2.2
762. Number Stream to Intervals	1.5.2.3
796. Minimum Light Radius	1.5.2.4
822. Kth-Pair-Distance	1.5.2.5
778. 水位上升的泳池中游泳	1.5.2.6

989. 数组形式的整数加法

1456. 定长子串中元音的最大数目	1.5.2.7
837. 新 21 点	1.5.2.8
438. 找到字符串中所有字母异位词	1.5.2.9
76. 最小覆盖子串	1.5.2.10
Number of Operations to Decrement Target to Zero	
401. 二进制手表	1.5.2.12 1.5.2.11
52. N 皇后 II	1.5.2.13
695. 岛屿的最大面积	1.5.2.14
1162. 地图分析	1.5.2.15
Shortest-Cycle-Containing-Target-Node	1.5.2.16
Top-View-of-a-Tree	1.5.2.17
746. 使用最小花费爬楼梯	1.5.2.18
198. 打家劫舍	1.5.2.19
673. 最长递增子序列的个数	1.5.2.20
1143. 最长公共子序列	1.5.2.21
62. 不同路径	1.5.2.22
688. “马”在棋盘上的概率	1.5.2.23
464. 我能赢么	1.5.2.24
416. 分割等和子集	1.5.2.25
494. 目标和	1.5.2.26
322. 零钱兑换	1.5.2.27
518. 零钱兑换 II	1.5.2.28
455. 分发饼干	1.5.2.29
435. 无重叠区间	1.5.2.30
881. 救生艇	1.5.2.31
96. 不同的二叉搜索树	1.5.2.32
23. 合并 K 个排序链表	1.5.2.33
932. 漂亮数组	1.5.2.34
260. 只出现一次的数字 III	1.5.2.35
78. 子集	1.5.2.36
进阶篇	1.5.3
实现 Trie (前缀树)	1.5.3.1
677. 键值映射	1.5.3.2
面试题 17.17 多次搜索	1.5.3.3
547. 省份数量	1.5.3.4

989. 数组形式的整数加法

924. 尽量减少恶意软件的传播	1.5.3.5
1319. 连通网络的操作次数	1.5.3.6
814 二叉树剪枝	1.5.3.7
39 组合总和	1.5.3.8
40 组合总数 II	1.5.3.9
47 全排列 II	1.5.3.10
28 实现 strStr()	1.5.3.11
28 实现 strStr()	1.5.3.12
215. 数组中的第 K 个最大元素	1.5.3.13
1046. 最后一块石头的重量	1.5.3.14
23. 合并 K 个排序链表	1.5.3.15
451 根据字符出现频率排序	1.5.3.16
378. 有序矩阵中第 K 小的元素	1.5.3.17
1054. 距离相等的条形码	1.5.3.18
1206. 设计跳表	1.5.3.19
二叉树遍历系列	1.5.3.20
反转链表系列	1.5.3.21
位运算系列	1.5.3.22
动态规划系列	1.5.3.23
有效括号系列	1.5.3.24
设计系列	1.5.3.25
前缀和系列	1.5.3.26
排序系列	1.5.3.27

先导篇

数据结构与算法是什么？

作为一个程序员，我们会写各种各样的代码。但是不管是什么功能，只要我们对其拆解得足够细，你会发现其都是**数据结构 + 算法**，其重要程度可见一斑。那么数据结构与算法究竟是什么呢？

数据结构就是数据的存储形式，算法就是对数据的一系列操作。

我们先来看数据结构。本质上来说，数据结构就是一堆数据，这些数据内部是结构化的。这里有两个要点：

- 一堆数据。也就是说不是单一的值。比如一个字符，一个数字等。
我们所说的数据结构通常就是一堆数据，或者说一系列数据。
- 结构化。那么一堆数据怎么组织呢？一个个紧邻排就是数据，用一个指针指向下一项就是链表等等。

接下来，我们看一下什么是算法。

前面说**算法就是对数据的一系列操作**。而这些操作从本质上来说就是**增，删，查**，我们无时无刻不在与这些东西打交道。而算法的学习就需要我们稳稳地抓住这三点。当你明确了这三点之后，你会发现数据结构就是手到渠成的事情了，也就是说数据结构是为了算法服务的。当你的算法分析好了，数据结构自然也会到位。

比如我们想从一个城市出发到达另外一个城市。显然，我们有很多方法。比如乘坐火车，飞机，自己开车等等。就算交通工具确定了，那么路线也有很多种。因此我们有无数种方法可选，那么哪一种性价比最高？

首先我们需要对问题进行定义。比如什么是性价比，再比如究竟有哪些交通工具和路线，每种的时间和金钱花费是多少等等。接下来，我们可以对问题进行抽象，使用计算机来解决。而使用计算机帮助我们解决这个问题编写的代码，我们就可以称之为算法。显然这是一种狭义的算法定义。更为广义的算法指的是解决问题的方法，只不过对我们做题和理解帮助不大罢了。使用这种算法，我需要存储一些中间数据，那么**如何组织这些中间数据**以使得性能最优呢？这就是数据结构的话题了。

最后总结一下：一般而言，我们遇到一个算法问题，可以大致分为如下三个步骤。

- 第一步是建立算法模型。
- 第二步则是根据算法模型分析我们需要对数据进行哪些操作（增删改）。

- 第三步，我们需要分析哪些操作最频繁，对算法的影响最大。最后，我们需要根据第三步的分析结果选择合适的数据结构。

可以看出，我们的分析过程是一层一层，逐步递进的。每一步都需要前一步分析的结果。如果你碰到的问题都严格按照我的这个思维模型进行的话，久而久之，你会逐步培养起自己的算法思维。这个是最最重要的，大家一定要把算法思维的培养放到学习算法中最高的位置。

希望大家在接下来的章节，可以用这个思维模型去练习。

这是 91 天学算法的先导篇。里面不会深入讲解一些知识点，而是从大的方向上描绘数据结构与算法的蓝图。除此之外，也有一些参与活动必须知道的东西。

小册特色

本小册的主要目的是帮助算法初学者入门，以解决诸如面试中的算法问题和部分设计问题。

—很多设计问题都可以抽象为纯粹的算法问题。

相比于传统的算法入门书籍和训练营，我们的特色是：

- 内容完整，几乎覆盖了所有算法面试的考点。
- 语言通俗易懂，讲解丰富，对初学者友好。部分小专题篇幅达到了 2 万多字，同时有大量的图画和题目，力求大家真正理解。
- 很多题目都提供了多种语言（Python, Java, CPP, JS 等），方便不同语言的人进行学习。
- 侧重刷题。通过结合力扣题目，真正做到大家学习认真完成之后可以解决大部分的力扣题目，做到在面试中迎刃有余。

91 算法共分为三篇，基础篇，进阶篇 和 专题篇。

让你：

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

第一阶段基础篇(30 天)。预计五个子栏目，每个子栏目 6 天。到时候发讲义给大家，题目的话天一道。讲义的内容大概是我在下方讲义部分放出的链接那样哦。

91 天学算法主要讲什么？

- 基础数据结构

- 常见的基础算法，几乎可以覆盖到面试的 80% 的考点。如果大家把这 80% 学精了，那么其他的 20 % 也是手到擒来。后续会陆续更新更多内容
- 面试题。每天更新一道题，这些题都是经过讲师精挑细选的，质量很高。并且题目之间梯度把控合理，难度循序渐进。

什么人适合这门课？

- 准备算法面试的人
- 想要提高程序员内功的人
- . . .

那什么样的人不需要学呢？

- 得过 ACM 区域赛冠亚军
- 力扣竞赛排名前 500
- 平时的 OJ 题目困难都可以做出来（可能不是特别顺）
- . . .

规则

大家的问题，打卡题目，讲义都在这里更新哦，冲鸭 🐧 。91 天见证更好的自己！不过要注意一周不打卡会被强制清退。

需要提前准备些什么？

- 数据结构与算法的基础知识。推荐看一下大学里面的教材讲义，或者看一些入门的图书，视频等，比如《图解算法》，邓俊辉的《数据结构与算法》免费视频课程。总之，至少你要知道有哪些常见的数据结构与算法以及他们各自的特点。
- 有 Github 账号，且会使用 Github 常用操作。比如提 issue，留言等。
- 有 LeetCode 账号，且会用其提交代码。

语言不限，大家可以自己喜欢的任何语言。同时我也希望你不要纠结于语言本身。

具体形式是什么样的？

- 总共三个大的阶段
- 每个大阶段划分为几个小阶段
- 每个小阶段前会将这个小阶段的资料发到群里
- 每个小阶段的时间内，每天都会出关于这个阶段的题目，第二天进行解答

比如：

- 第一个大阶段是基础
- 基础中第一个小阶段是 数组，栈和队列 。
- 数组，栈和队列 正式开始前，会将资料发到群里，大家可以提前预习。
- 之后的每天都会围绕 数组，栈和队列 出一道题，第二天进行解答。大家可以在出题当天上 Github 上打卡。

大家遇到问题可以在群里回答，对于比较好的问题，会记录到 github issue 中，让更多的人看到。Github 仓库地址届时会在群里公布。

如何打卡？

- 每天都会有一道题目放到 issue 里
- 大家在对应 issue 下留言即可

想要坚持打卡抽奖的小伙伴注意了，必须当天打卡才算打卡哦。不是当天的话需要补签卡，补签卡需要连续打卡一周才可以获得一张的

打卡格式：

- 思路
- 代码
- 复杂度分析（时间和空间）

一个例子：

```
### 思路  
(此处撰写思路)  
  
### 代码  
  
```java (此处换成你的语言，比如js, py 等)  
(此处撰写代码)

```  
  
**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:  $O(1)$ 
```

数据结构与算法概述

本篇是数据结构与算法的先导篇，目的是帮大家认识数据结构和算法的世界，具体的数据结构和算法会在之后的讲义详细进行讲述。

数据结构算法面面观

狭义的算法指的是经典的具体算法，广义的算法指的是解决问题的方法。

比如 `math.sqrt` 就是一种广义的算法，其具体的算法可以是牛顿迭代法，二分法，也可以是暴力法等。

在这里，我们往往关注的是狭义的算法，并且是那些被反复验证的经典算法思想。

研究这些经典算法很重要，它不仅可以帮我们解决实际的问题，锻炼思维，而且还可以帮助我们交流，在这个层面上来说，其作用类似设计模式。比如我跟你说这道题用二分法就行了，你如果也恰好明白什么是二分，就可能知道我的意思。而如果你不知道二分，我只能这样解释你才可能明白“用两个指针，分别指向数组的头部和尾部，然后计算中间位置，通过比较目标元素和中间位置的关系移动两个指针。。。”。

而数据结构是计算机存储、组织数据的方式，指相互之间存在一种或多种**特定关系**的数据元素的集合。要想深刻理解其数据结构，一定要结合算法。因为任何数据结构都是为了实现某一个或者多个算法的，数据结构是为算法所服务的。就好像你要做红烧鱼需要鱼，你做可乐鸡翅需要鸡翅。当你用到特定算法的话，自然会用到特定的数据结构。

我们知道，内存的物理表现实际上就是一系列连续的内存单元，每一个内存单元的大小是固定的，这也是内存支持随机访问的本质原因。那么应该怎么存储和检索以及修改内存呢（增删查）？这就是数据结构研究的话题。

上面提到的是内存的物理结构，我们也可以基于这个物理结构拓展逻辑结构。比如，上面提到物理内存是固定大小连续存储的，那我可不可以将一段大的数据存储在不连续的空间呢？当然可以，这需要你自己处理，当你尝试去处理的时候，实际上就涉及到了逻辑结构。你所知道的，以及我们即将研究的全部都是逻辑结构。

不同的逻辑结构存储实际上有不同的特点，我们需要结合实际的场景分析。要想拥有具体问题具体分析的能力，我们首先要对基本的数据结构要特别清楚。包括他们的特点，适用场景，不适合场景等。我的建议是先过一遍数据结构，有个大致印象，然后研究具体算法。之后结合算法回头继续复习数据结构。

例子

我们公司前台可以代收快递。但是当我们取快递的时候，需要我们在一个事先写了我们报告信息的表格上签字。表格大概是这样的：

| 时间 | 名字 | 快递公司 | 单号 | 签字 |
|------------|----|------|----------|----|
| 2020-09-09 | 西法 | SF | sf298001 | 西法 |
| 2020-09-10 | 张三 | SF | sf133990 | |

由于是来一个快递，快递小哥就会在表格增加一个记录，因此时间那一列是升序排列的。

如果你某一天要来公司取你的快递，那么你肯定想要快速找到你的包裹所在的行。一个好的方式则是先根据日期检索，确定大概范围之后再根据快递公司找，由于一个快递小哥通常会一次登记很多快递，因此会出现连续的同一个快递公司的常见现象，这样你可以快速定位到你的快递公司那几行数据，最后再根据你的单号来最终确认即可。

我们来分析一下上面的问题。实际上，上面的例子查询的次数要远远大于插入。而查询的过程除了时间（只有时间是有序的）可以二分，其他条件则不可以。如果我只有一个快递，而和我同天的快递很多，那么要快速找到自己的快递就不那么容易。

如果相同名字的包裹放在一起，这样我们不是就可以像查字典一样快速定位到自己的包裹了么？

确实如此，如果名字还能按照拼写排序，我们也可利用查字典的方式快速定位。

假设我们真的按照名字拼写顺序进行组织，**并且名字是挨着写的**，即张三后面直接跟着另外一个人，中间没有空行。大概是这样的：

```
张三
张四
xxx
xxx
xxx
```

这样的话假设又来了一个张三的快递，我们就需要擦掉张四那一行，写上张三的信息。然后将擦掉的张四的信息重写到下一行。由于这样做会将下面的xxx给覆盖，因此我们需要执行同样的操作。如果张四下面人数很多这个效率可想而知。

怎么解决呢？

其实我们也可以给公司的所有人准备一个表格，每一个人的快递都写到对应的表格。这就对快递员（写入）有了新的要求。并且会更加浪费表格（空间），但是相应地查询速度会更快。

但是这样还有一个问题。如果公司有很多人都不寄快递到公司呢？为他们也准备表格就显得多余。另外我们需要给每个人都准备同样大小的表格么？这都是我们需要考虑的。实际上，我们可以取舍一下，比如给所有名字 L 开头的用三张表，给所有名字 B 开头的一张表。

我们假设公司 L 开头的人比较多，是 B 开头的大约三倍。

计算机也是类似，不同的存储有不同的特点。有好处也有坏处，我们要做的是根据实际情况选择合适的数据结构。

数据结构与算法就是研究在什么情况下**应该如何高效组织和操作数据**的一门学科。请大家务必记住这一句话。

逻辑结构

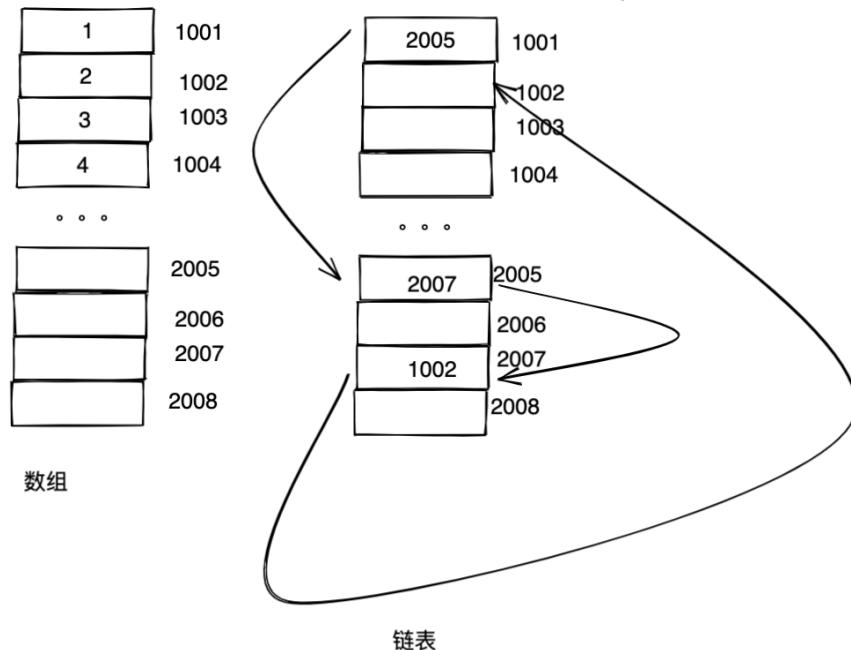
我们平常说的数据结构都指的是逻辑结构。比如数组，链表，二叉树等等，这个我们会在基础篇做详细介绍。这里我们就简单带大家了解一下就行了。

数组用来表示连续的内存空间，而链表通常用来表示不连续的内存空间。
这个连续性指的是在内存中存放的数据是否物理上连续。

连续我们比较好理解，毕竟内存本来就是连续的。那么如何理解不连续的内存空间呢？

其实，我们只需要区分数据域和指针域即可。数组的话，我们可以根据内存的物理地址来索引，比如 A[0] 就是数组第一项，A[12] 就是数组第 13 项。

由于数组中的元素大小固定，因此我们可以通过基址 + 偏移量的方式实现随机访问。链表则不行，因为其下个元素并不一定是紧邻的，我们无法通过上面的基址和偏移量算出来。因此需要多一个指针域，来表示其下一个元素的内存单元位置。不难看出，链表相比数组，会增加额外的空间负担。有什么好处呢？好处则是增加或删除变得容易，这个我们留到基础篇详细谈谈。



如上图是一段物理内存。如果我在连续内存中存值，并通过内存编号访问那就是数组。而如果我在内存中除了存值，还多存一个内存编号，用于寻找下一个数据，那就是链表。

内存还是那个内存，通过不同的逻辑抽象就是两种数据结构了。当然这个解释比较粗糙，但是如果却可以帮助你认识本质。

总结

本节我们学习了数据结构和算法的关系，以及狭义和广义的算法。

对于算法而言本讲义的全部内容都是围绕狭义的算法展开，帮助大家理解经典的算法思想，并将这些思想串联起来进行综合运用。

对于数据结构而言本讲义的全部内容都是围绕逻辑结构而已，逻辑结构只不过是我们使用物理结构的一种抽象表示而已。基于数组和链表这两种基本的逻辑结构，拓展了无数的丰富的数据结构，这些数据结构都是为了解决特定问题产生的，因此理解数据结构一定要结合算法。接下来基础篇的章节，我们会带你剖析常见的数据结构，准备好了么？

如何衡量算法的性能

本节部分内容截取自我的新书

介绍

学习算法，首先要知道的就是如何判断一个算法的好坏。好的程序有很多的评判标准，包括但不限于可读性，扩展性，性能等。这里我们来看其中一项指标——性能。坏的程序性能不一定差，但是好的程序通常性能都比较好。那么如何分析一个算法的性能好坏呢？这就是我们要讲的复杂度分析，所有的数据结构教程都会把这个放在前面来讲，不仅是因为它们是基础，更因为它们真的非常重要。学会了复杂度分析，你才能够对算法进行分析，从而帮助你写出复杂度更优的算法。如果你对一种算法的复杂度推导很熟悉，那么我相信你已经掌握了这个算法。本章主要介绍时间复杂度，空间复杂度的分析方法也是类似，并且相对于时间复杂度，大多数情况下空间复杂度的分析更容易。如果你对复杂度不熟悉，或者看完本文还是不太理解其含义和用法，那么建议你搭配《算法》（第四版）中 1.4 算法分析 一起学习。

时间复杂度和空间复杂度分别衡量程序运行时间的长短和运行时所占空间的大小。如何衡量一个程序运行时间长短，占用内存大小呢？内存倒还好，但是运行时长，由于不同计算机的性能不同，执行时间也会不同，甚至有可能有数倍的差距，那么究竟应该如何衡量 程序运行时间的长短呢？

《计算机程序设计艺术》的作者高德纳（Donald Knuth）提出了一种方法，这种方法的核心思想很简单，就是 一个程序运行时间主要和两个因素有关，分别是1. 执行每条语句的耗时 2. 执行每条语句的频率。而前者取决于硬件，后者取决于算法本身和程序的输入。那么如何统计算法 执每条语句的频率 呢？我们举个例子来说明。

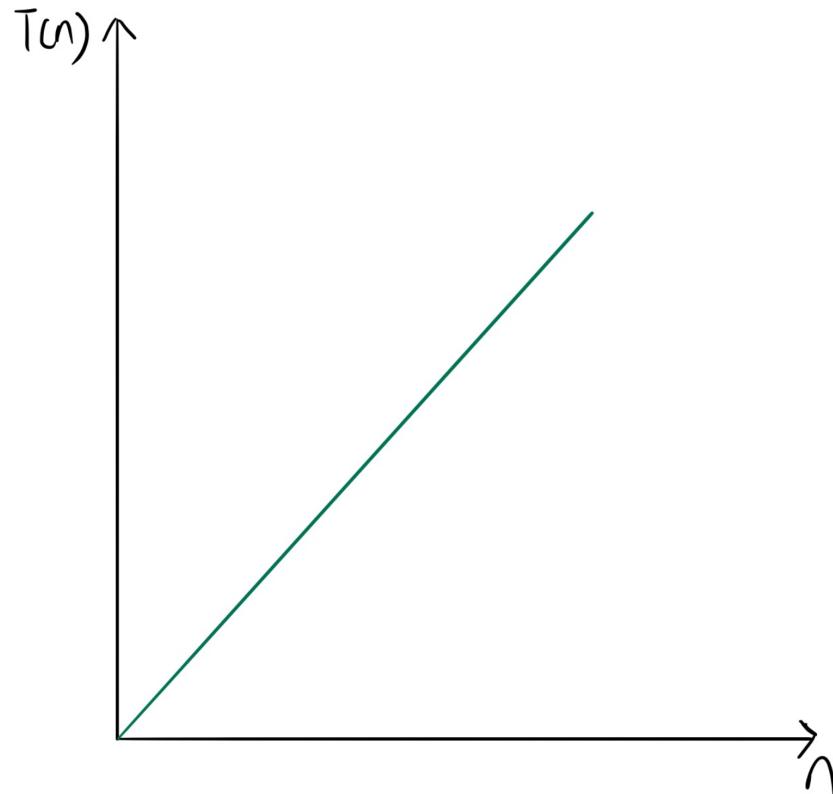
如下是一个从 1 累加到 n 的一个算法，这个算法用了一层循环，并且借助了一个变量 res 来完成。

```
def sum(n: int) -> int:
    res = 0
    for i in range(1, n + 1):
        res += i
    return res
```

(代码 1.3.1)

我们将这个方法从更加微观的角度来分析一下。上述代码会执行 N 次循环体的内容，假设每一次执行都是常数时间，不妨假设其执行时间是 x，res = 0 和 return res 的执行时间分别为 y 和 z。那么总的时间

就等于 $n x + y + z$, 如果 粗略 地将 x , y 和 z 都看成一样的, 那么可以得出总时间为 $(n + 2)x$ 。如果用图来表示的话就是这样的:



(图 数据规模和操作数的关系图)

换句话说算法的运行时间和数据的规模成正比。

在渐进意义上讲, 我们常常忽略较小项, 如上的 $2x$, 而仅保留最大项, 如上的 nx , 这样可以大大减少分析工作量, 因此这种复杂度分析方法也被成为渐进复杂度分析。实际上这在现实中也很常见, 即 程序运行时间往往取决于其中一小部分指令。

大 O 表示法

以上正是一种叫做大 O 表示法的基本思想, 它是一种描述算法性能的记法, 这种描述和编译系统、机器结构、处理器的快慢等因素无关。这种描述的参数是 N , 表示数据的规模。这里的 O 表示量级 (order), 比如说“二分查找的时间复杂度是 $O(\log N)$ ”, 就是说它需要“通过 $\log N$ 量级的操作去查找一个规模为 N 的数据结构”。这种估测对算法的理论分析和大致比较是非常有价值的, 我们可以很快地对算法进行一个大致的估算。

例如一个拥有较小常数项的 $O(N^2)$ 算法在规模 N 较小的情况下可能比一个高常数项的 $O(N)$ 算法运行地更快。但是随着 N 足够大以后, 具有较慢上升趋势的算法必然运行地更快, 因此在采用大 O 表示复杂度

的时候，可以忽略系数，这也是我们可以忽略不同性能计算机执行差异的原因，因为你可以把不同性能计算机性能差异看成是系数的差异。

除此之外，我们还应该区分算法的最好情况，最坏情况和平均情况，但是这不在本书的讨论范畴，本书的所有复杂度如不做特殊说明，均指的是最坏复杂度。

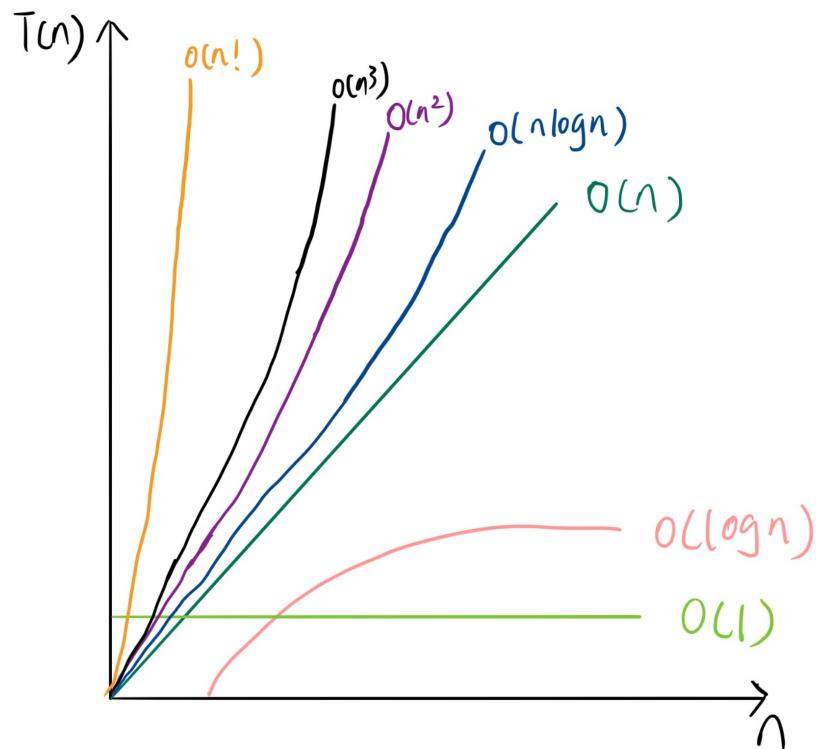
实际上大O表示就是来表示最坏复杂的，而平均复杂度和最好复杂度不是这个符号，不过不需要大家掌握。

空间复杂度也是类似，不过空间复杂度往往更好分析，因为很少空间复杂度会有诸如对数情况。

不过需要注意的是，递归的空间复杂度容易被大家忽略，也就是说每次开辟调用栈空间容易被大家忽略。我们可以将递归算法，看作是使用了栈的迭代算法。而其栈的空间就是递归中调用栈的空间。因此递归的空间复杂度需要额外加上开辟的栈空间。

值得注意的是本书全部内容的空间复杂度均指的是额外空间复杂度。比如题目给的入参，题目要求的返回值是不计入空间复杂度的。我们只计算由于采用了这个算法，而不得不多用的空间。

最后我给出了这几种常见的时间复杂度的趋势图对比，大家可以直观地感受一下趋势变化。



(图 各种复杂度的渐进趋势对比)

复杂度分析的意义

如果你不会复杂度分析，说明你很可能没有彻底明白这道题，从这个意义上将可以帮你找到算法短板。

复杂度分析可以帮助你快速排除不可能的算法。比如你想到一个算法的时间复杂度是指数，而题目的数组范围是 10^6 ，这基本可以认为是不对的算法。因为出题人不太会出一个明显看起来不靠谱的题。

总结

复杂度分析是对算法执行效率的一个粗略评价，可以大致地锁定一个算法的性能。

我们研究算法的目的其实就是提高算法执行的效率，这种效率可以是时间上的，也可以是空间上的。而实际业务中，时间上的优化更重要，因此我们也会讲解很多空间换时间的技巧。

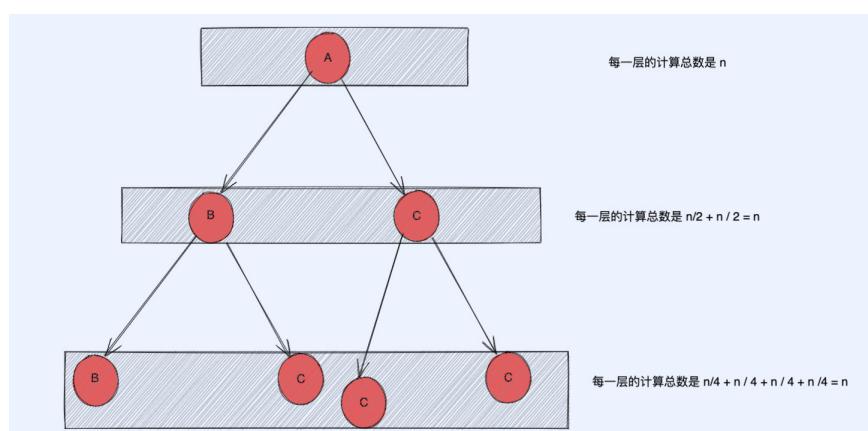
对于时间复杂度来说，我们只需要关注算法的基本操作的次数即可。给大家几个实操技巧是关注：

- 如果有循环，则关注下最内层循环的执行次数。

不是两层循环就是 n^2 ，而是看最内层执行了多少次

- 如果有递归那就是：递归树的节点数 * 递归函数内部的基础操作数。
而这句话的前提是所有递归函数内部的基本操作数是一样的，这样才能直接乘。

而如果递归函数的基本操作数不一样，可以看下递归树每一层的基础操作数是否一样，如果每一层的基础操作数一样，则可以通过**递归深度 * 每一层基础操作数**计算得出，比如归并排序的复杂度分析中就用到了这个技巧。这里我画个图方便大家理解：



不过这种时间复杂度只能应付简单的情况。如果递归函数内部操作数不那么确定，则必须通过列出转移方程，然后根据方程推导出基础操作数的数量级，这样就得出了时间复杂度。这部分内容我已经写到了我的新书中了。

989. 数组形式的整数加法

1. 如果用了一些 API。比如数组用了头部添加的 API，不要忽略了这是一个时间复杂度 $O(n)$ 的操作。

额外空间复杂度稍微简单一点。我们只需要看下算法用了什么样的集合数据结构就行了，而不需关心非集合数据结构，比如哈希表，数组等等，关心它是如何随着数据规模变化而变化就好了。比如我写动态规划，初始化了一个 $O(m n)$ 的二维矩阵，那么此时空间复杂度就已经是 $O(m n)$ 了。

而布尔值，number 值 等等不需要考虑。上面的做法其实没有考虑递归，如果你用了递归，那么多考虑一个递归栈就行了，递归栈的开销就是递归树的深度。

复杂度分析很重要，说其最重要都不为过。因此大家打卡的时候，尽量要给出复杂度的分析。

当你对某一个算法分析不出来的时候，可以请教我们的导师，一来二去慢慢就会了。相信我，只要每道题你不仅做出来，还对复杂度分析透透的，很快算法你就上道了。

数组，栈，队列

大家好，本节是数据结构的开篇内容。本节主要讲述数组，栈以及队列。

数组的知识大家可以轻松地迁移到字符串，因此本书不对字符串进行特殊讲解。

数组

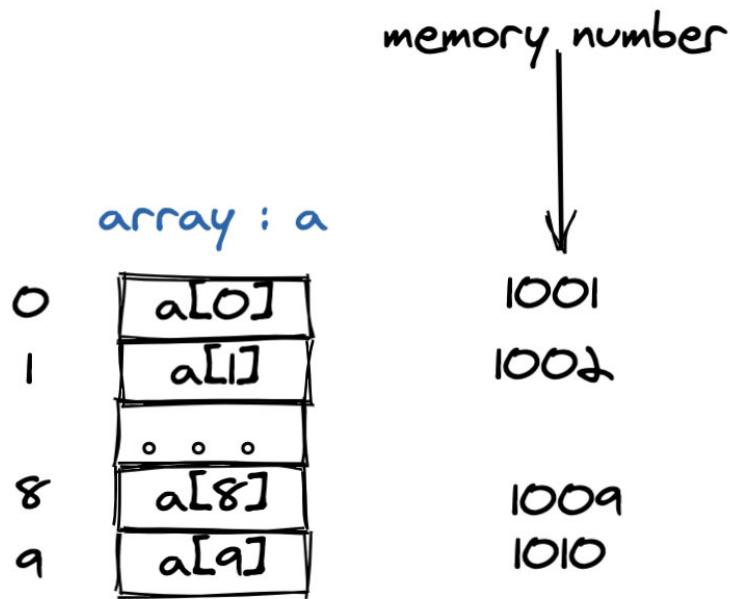
数组是一种使用最为广泛的数据结构，尤其是在大家的日常开发中，原因无非就是操作简单和支持随机访问。而字符串大家也可以将其看成是一个字符数组，这更加夯实了数组的重要性。

数组是我们要讲的第一个重要数据结构（另外一个是链表），很多的数据结构都是基于其产生的。比如后文要讲的二叉树，图等。

比如给你一个数组 `parents`，其中 `parents[i]` 表示 `i` 的父节点。比如 `[-1, 0, 0]` 就表示索引 1 和 2 的父节点是 0，而 0 没有任何的父节点，我们不妨用 `-1` 表示。这就完成了用数组来表示二叉树的过程。图也是类似的，后文我们讲图的时候，也会用数组来存图中边的关系。

虽然数据结构有很多，比如树，图，哈希表等。但真正的实现还需要落实到具体的基础数据结构，即数组和链表。之所以说他们是基础的数据结构，是因为它们直接控制物理内存的使用。

数组使用连续的内存空间，来存储一系列同一数据类型的值。如图表示的是数组的每一项都使用一个 `byte` 存储的情况。



那么为什么数组要存储相同类型的值呢？为什么有的语言（比如 JS）就可以存储不同类型的值呢？

实际上存储相同的类型有两个原因：

1. 相同的类型大小是固定且连续的（这里指的是基本类型，而不是引用类型，当然引用类型也可以存一个大小固定的指针，而将真实的内容放到别的地方，比如内存堆），这样数组就可以随机访问了。试想数组第一项是 4 字节，第二项是 8 字节，第三项是 6 字节，我如何才能随机访问？而如果数组元素的大小都一样，我们就可以用基址 + 偏移量来定位任意一个元素，其中基址指的是数组的引用地址，如上图就是 1001。偏移量指的是数组的索引 * 数组每一项所占用的内存空间大小。
2. 静态语言要求指定数组的类型。

虽然在一些语言，比如 JavaScript 中，数组可以保存不同类型的值，这是因为其内部做了处理。对于 V8 引擎来说，它将数据类型分为基本类型和引用类型，基本类型直接存储值在栈上，而引用类型存储指针在栈上，真正的内容存到堆上。因此不同的数据类型也可以保持同样的长度。

数组的一个特点就是支持随机访问，请务必记住这一点。当你需要支持随机访问的数据结构的话，自然而然应该想到数组。

本质上，数组是一段连续的地址空间，这个是和我们之后要讲的链表的本质差别。虽然二者从逻辑上来看都是线性的数据结构。

这里我总结了数组的几个特性，供大家参考：

- 一个数组表示的是一系列的元素
- 数组 (static array) 的长度是固定的，一旦创建就不能改变（但是可以有 dynamic array）
- 所有的元素需要是同一类型（个别的语言除外）
- 可以通过下标索引获取到所储存的元素（随机访问）。比如 `array[index]`
- 下标可以是 0 到 `array.length - 1` 的任意整数

当数组里的元素也是一个数组的时候，就可以形成多维数组。例子：

1. 用一个多维数组表示坐标
2. 用一个多维数组来记录照片上每一个 pixel 的数值

力扣中有很多二维数组的题目，我一般称其为 `board` 或者 `matrix`，这样通过名字一眼就能看出其是一个二维数组。

比如后面要讲的动态规划，如果题目的状态不止一个，我们就需要使用多维数组来存储。每一个状态对应数组中的一个维度。另外，后面的图部分，我们很多时候都会用二维数组来建立邻接矩阵。

数组的常见操作

了解了数组的底层之后，我们来看下数组的基本操作以及对应的时间复杂度。

1. 随机访问，时间复杂度 $O(1)$

```
arr = [1, 2, 33]
arr[0] # 1
arr[2] # 33
```

1. 遍历，时间复杂度 $O(N)$

```
for num in nums:
    print(num)
```

1. 任意位置插入元素、删除元素

```
arr = [1, 2, 33]
# 在索引2前插入一个5
arr.insert(2, 5)
print(arr) # [1, 2, 5, 33]
```

我们不难发现，插入 2 之后，新插入的元素之后的元素（最后一个元素）的索引发生了变化，从 2 变成了 3，而其前面的元素没有影响。从平均上来看，数组插入元素和删除元素的时间复杂度为 $O(N)$ 。最好的情况

989. 数组形式的整数加法

删除和插入发生在尾部，时间复杂度为 $O(1)$ 。

基本上数组都支持这些方法。虽然命名各有不同，但是都是上面四种操作的实现：

- `each()`: 遍历数组
- `pop(index)`: 删除数组中索引为 `index` 的元素
- `insert(item, index)`: 数组索引为 `index` 处插入元素

时间复杂度分析小结

- 随机访问 $\rightarrow O(1)$
- 根据索引修改 $\rightarrow O(1)$
- 遍历数组 $\rightarrow O(N)$
- 插入数值到数组 $\rightarrow O(N)$
- 插入数值到数组最后 $\rightarrow O(1)$
- 从数组删除数值 $\rightarrow O(N)$
- 从数组最后删除数值 $\rightarrow O(1)$

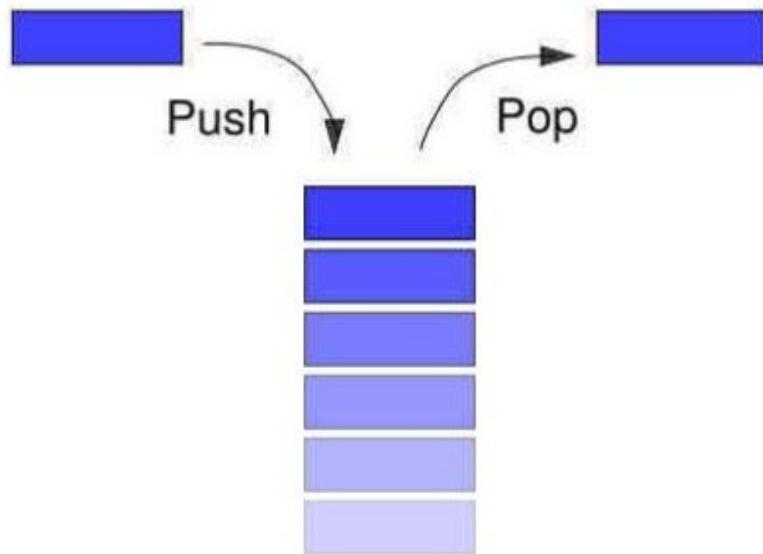
题目推荐

- [414. 第三大的数](#)
- [剑指 Offer 53 - II. 0 ~ n-1 中缺失的数字](#)
- [88. 合并两个有序数组](#)
- [380. 常数时间插入、删除和获取随机元素](#)
- [41. 缺失的第一个正数](#)

另外推荐两个思考难度小，但是边界多的题目，这种题目如果可以一次写出 bug free 的代码会很加分。

- [59. 螺旋矩阵 II](#)
- [859. 亲密字符串](#)

栈



栈是一种受限的数据结构，体现在只允许新的内容从一个方向插入或删除，这个方向我们叫栈顶，另一端一般称为栈底。除了栈顶的其他位置获取或操作内容都是不被允许的。

栈最显著的特征就是 LIFO(Last In, First Out - 后进先出)

举个例子：

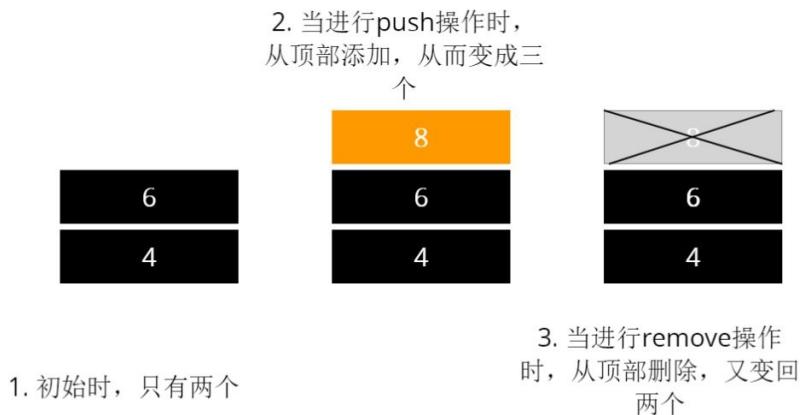
栈就像是一个放书本的抽屉，进栈的操作就好比是想抽屉里放一本书，新进去的书永远在最上层，而出栈则相当于从里往外拿书本，永远是从最上层开始拿，所以拿出来的永远是最后进去的哪一个。

栈的常用操作与时间复杂度

- 进栈 - push - 将元素放置到栈顶
- 出栈 - pop - 将栈顶元素弹出
- 取栈顶 - top - 得到栈顶元素的值
- 判断是否为空栈 - isEmpty - 判断栈内是否有元素

复杂度分析：

- 进栈 - O(1)
- 出栈 - O(1)
- 取栈顶 - O(1)
- 判断是否为空栈 - O(1)



实现

由于栈只允许在尾部操作，我们用数组进行模拟的话，可以很容易达到 $O(1)$ 的时间复杂度。

当然也可以用链表实现，即链式栈。

我们可以使用一个数组加一个变量（记录栈顶的位置）的方式很方便的实现栈。实现过程也非常简单，即将数组的 API 删除几个就好了。

比如数组支持在头部添加和删除元素以及遍历数组等 API，我们将其删除就可以直接将其看成是栈。这也充分应证了我开头的话栈是一种受限的数据结构。

应用

栈是实现深度优先遍历的基础。除此之外，栈的应用还有很多，这里列举几个常见的。

- 函数调用栈
- 浏览器前进后退
- 匹配括号
- 单调栈用来寻找下一个更大（更小）元素

除此之外，有两个在数学和计算机都应用超级广泛的就是是 波兰表示法 和 逆波兰表示法，之所以叫波兰表示法，是因为其是波兰人发明的。

波兰表示法 (Polish notation, 或波兰记法)，是一种逻辑、算术和代数表示方法，其特点是操作符置于操作数的前面，因此也称做前缀表示法。如果操作符的元数 (arity) 是固定的，则语法上不需要括号仍然能被无歧义地解析。波兰记法是波兰数学家扬·武卡谢维奇 1920 年代引入的，用于简化命题逻辑。

扬·武卡谢维奇本人提到：[1]

“我在 1924 年突然有了一个无需括号的表达方法，我在文章第一次使用了这种表示法。”

以下是不同表示法的直观差异：

- 前缀表示法 $(+ 3 4)$
- 中缀表示法 $(3 + 4)$
- 后缀表示法 $(3 4 +)$

LISP 的 S-表达式中广泛地使用了前缀记法，S-表达式中使用了括号是因为它的算术操作符有可变的元数（arity）。逆波兰表示法在许多基于堆栈的程序语言（如 PostScript）中使用，以及是一些计算器（特别是惠普）的运算原理。

题目推荐

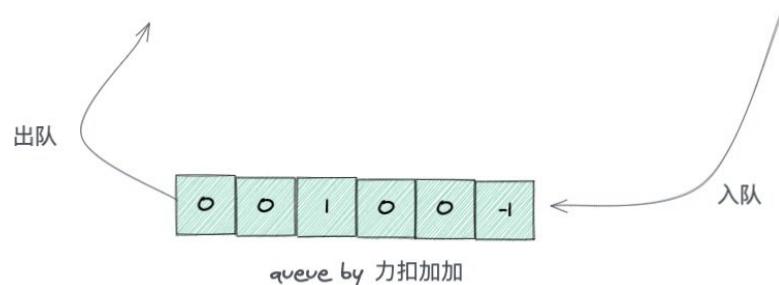
- [150. 逆波兰表达式求值](#)
- [1381. 设计一个支持增量操作的栈](#)
- [394. 字符串解码](#)
- [946. 验证栈序列](#)

另外还有[两个计算器的题目](#)也值得练习。

队列

同样地，队列也是一种受限的数据结构。

和栈相反，队列是只允许在一端进行插入，在另一端进行删除的线性表。因此队列(Queue)是一种先进先出(FIFO - First In First Out)的数据结构，通常情况下，我们称队列中插入元素的一端为尾部，删除元素的一端为头部。



队列也是一种逻辑结构，底层同样可以用数组实现，也可以用链表实现，不同实现有不同的取舍。

如果用数组实现，那么入队或者出队的时间复杂度一定有且仅有一个是 $O(N)$ 的，其中 N 为队列的长度。而使用链表实现则可以在 $O(1)$ 的时间完成任何合法的队列操作。这得益于链表对动态添加和删除的友好性。关于链表的队列的实现，我们会在后面的 队列的实现 (Linked List) 部分讲解。

队列的操作与时间复杂度

- 插入 - 在队列的尾部添加元素
- 删除 - 在队列的头部删除元素
- 查看首个元素 - 返回队列头部的元素的值

时间复杂度取决于你的底层实现是数组还是链表。我们知道直接用数组模拟队列的话，在队头删除元素是无法达到 $O(1)$ 的复杂度的，上面提到了由于存在调整数组的原因，时间复杂度为 $O(N)$ 。因此我们需要一种别的方式，这种方式就是下面要讲的 Linked List。

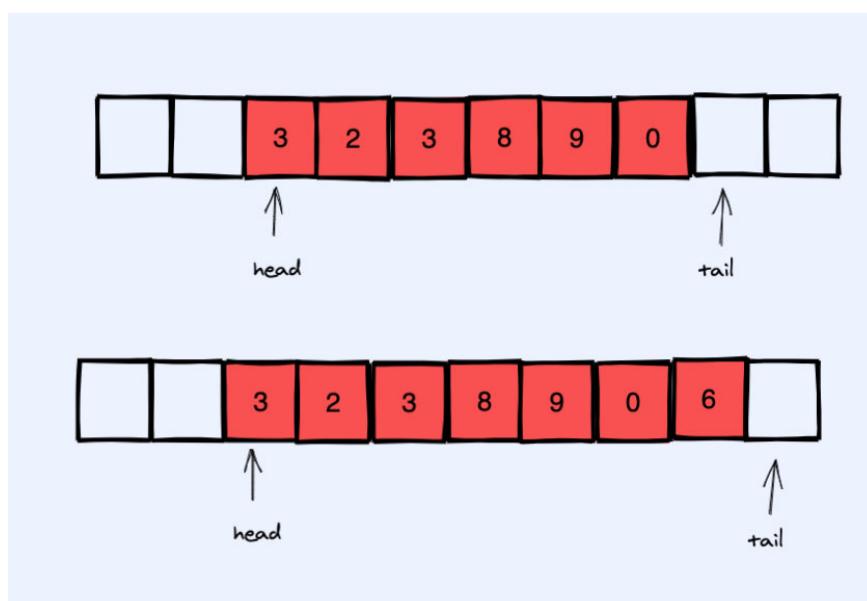
以链表为例，其时间复杂度：

- 插入 - $O(1)$
- 删除 - $O(1)$
- 查看首个元素 - $O(1)$

实际上队列也可用数组来实现，并且插入和删除时间复杂度都是 $O(1)$ ，如何实现呢？

其实我们只需要用两个指针 `head` 和 `tail` 表示队列的头和尾部，然后给队列分别一个空间。当插入的时候，我们在 `tail` 的前一个内存单元插入一个值，并更新 `tail` 即可。

如图是在一个队列中插入一个数字 6 的内部情况。



在头部删除也是类似的。

这种实现的方式，需要动态开辟内存。如果不考虑内存不够而 copy 内存的情况，那么整体的时间复杂度可以控制在 $O(1)$ 。

CPP 的 deque 就是这么实现的

应用

队列的应用同样广泛。在做题中最主要的一个应用就是广度优先遍历 (BFS)。而实际使用中同样使用广泛，比如消费队列，浏览器的 HTTP 请求队列等等。

队列的实现 (Linked List)

我们知道链表的删除操作，尤其是删除头节点的情况下，是很容易做到 $O(1)$ 。

那么我们是否可利用这一点来弥补上面说的删除无法达到 $O(1)$ ？

删除非头节点可以做到 $O(1)$ 吗？什么情况下可以？

但是在链表末尾插入需要遍历到尾部的话就不是 $O(1)$ ，而是 $O(N)$ 了。

解决这个问题其实不复杂，只要维护一个变量 tail，存放当前链表的尾节点引用即可在 $O(1)$ 的时间完成插入操作。

因此使用链表进行模拟的话。

入队就是：

```
tail.next = newNode()
tail = newNode()
```

类似地，我们维护一个 head 虚拟节点也可是在 $O(1)$ 时间出队。

出队就是：

```
nnext = head.next.next
head.next = nnext
```

具体的代码大家可以在学习完链表章节在回头补充。

另外大家在平时做题的时候可以直接使用内置的队列，比如 Python 的 deque。

严格意义上 deque 是双端队列，其允许在两端同时进行插入和删除。因此比普通队列的操作更宽松，不是严格的队列。不过和栈类似，我们删除几个 API 就可以将其看成是一个基于链表实现的队列了。

除此之外，还有一种队列是循环队列，用的不是很多。篇幅所限，不在这里展开，感兴趣的可以自己查一下。

推荐题目

- [min-stack](#)
- [evaluate-reverse-polish-notation](#)
- [decode-string](#)
- [binary-tree-inorder-traversal](#)
- [clone-graph](#)
- [number-of-islands](#)
- [largest-rectangle-in-histogram](#)
- [implement-queue-using-stacks](#)
- [01-matrix](#)

相关专题

前缀和

关于前缀和，看我的这篇文章就够了～ [【西法带你学算法】一次搞定前缀和](#)

单调栈

单调栈适合的题目是求解第一个一个大于 xxx 或者第一个小于 xxx 这种题目。所有当你有这种需求的时候，就应该想到[单调栈](#)。

下面两个题帮助你理解单调栈，并让你明白什么时候可以用单调栈进行算法优化。

- [84. 柱状图中最大的矩形](#)
- [739. 每日温度](#)

单调栈的使用实在是太多了，我之前写的一个系列题解核心也就是单调栈。参考：[一招吃遍力扣四道题，妈妈再也不用担心我被套路啦～](#)。

单调队列和单调栈的思路比较类似，感兴趣的可以自己查阅一下相关资料作为扩展。这里只推荐一道题 [239. 滑动窗口最大值](#)。

栈匹配

当你需要比较类似栈结构的匹配的时候，就应该想到使用栈。

比如判断有效括号。我们知道有效的括号是形如：((())) 这样的括号，其中第一个左括号和最后一个右括号匹配，因此一种简单的思路是把左括号看出是入栈，右括号看出是出栈即可轻松利用栈的特性求解。

再比如链表的回文判断。我们就可以一次遍历压栈，再一次遍历出栈的同时和当前元素比较即可。这也是利用了栈的特性。

- 20. 有效的括号

分桶 & 计数

[49.字母的异位词分组](#), [825. 适龄的朋友](#) 以及 [【每日一题】Largest Range](#) 等就是分桶思想的应用。力扣关于分桶思想的题目有很多，大家只要多留心就不难发现。

从算法上看，我们通常会建立一个 counts 数组来计数，其本质和 Python 的 collections.Counter 类似，你也可用数组进行模拟，代码也比较简单。比如：

```
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        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 list(str_dict.values())
```

如果代码就是 [49.字母的异位词分组](#) 的一个可行解。代码中的 s_key 就是一个桶，其中桶的大小为 26，表示一个单词中的 26 个字母出现的频率分布，这点有点像计数排序。

适合用分桶思想的题目一定是不在乎顺序的，这一点也不难理解，比较分桶之后原来的顺序信息就丢失了。

总结

数组和链表是最最基础的数据结构，大家一定要掌握，其他数据结构都是基于两者产生的。

栈和队列是两种受限的数据结构，我们人为地给数组和链表增加一个限制就产生了它们。那我们为什么要自己给自己设限制呢？目的就是为了简化一些常见问题，这就好像是人类模仿鸟制造了飞机，模仿鸽子做了地震仪

一样。栈和队列能帮我们简化问题。比如队列的特性就很适合做 BFS，栈的特性就很适合做括号匹配等等。你可以这么理解。我们一开始做 BFS 的时候，没有队列。慢慢大家写地多了，发现是不是可以**抽象一个数据结构单独来处理这种通用的需求？**队列就产生了，其他数据结构也是一样。

参考

- [基础数据结构 by lucifer](#)

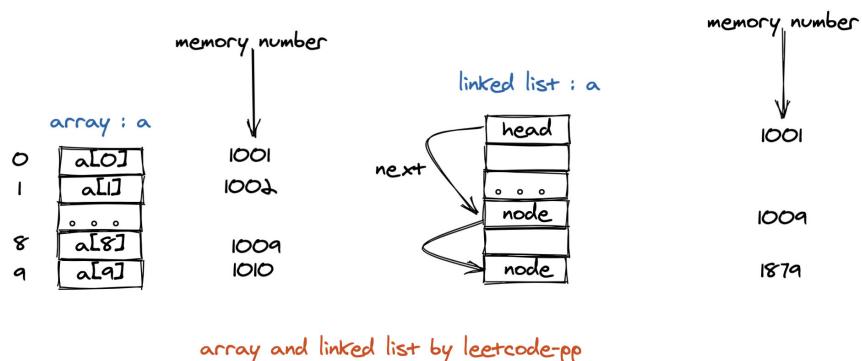
链表

关于链表的使用技巧，我在[几乎刷完了力扣所有的链表题，我发现了这些东西。。。](#)中进行了细致的讲解，建议大家看完本节内容后，再去看下这篇文章。

简介

本节给大家介绍的是和数组同一个级别的重量级数据结构 - 链表，很多的数据结构都是基于其产生的，比如前文讲的队列。

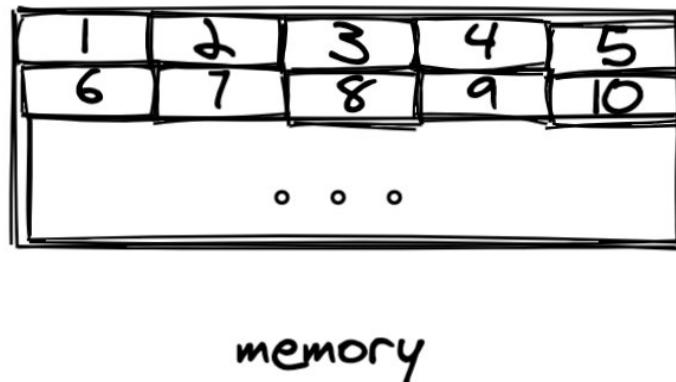
各种数据结构，不管是队列，栈等线性数据结构还是树，图的等非线性数据结构，从根本上底层都是数组和链表。两者在物理上存储是非常不一样的，如图：



array and linked list by leetcode-pp

(图 1. 数组和链表的物理存储图)

物理内存是一个个大小相同的内存单元构成的，如图：



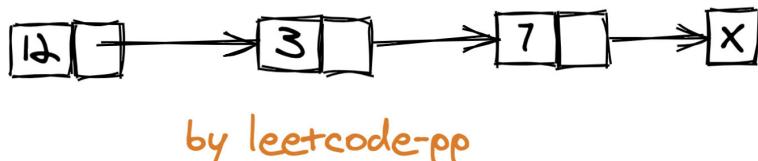
(图 2. 物理内存)

不难看出，数组和链表只是使用物理内存的两种方式。

数组是连续的内存空间，通常每一个单位的大小也是固定的，因此可以按下标随机访问。而链表则不一定连续，因此其查找只能依靠别的方式，一般我们是通过一个叫 `next` 指针来遍历查找。

链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。

链表适合在数据需要有一定顺序，但是又需要进行频繁增删除的场景。



(图 3. 一个典型的链表逻辑表示图)

后面所有的图都是基于逻辑结构，而不是物理结构

链表只有一个后驱节点 `next`，如果是双向链表还会有一个前驱节点 `pre`。

由于只有一个前驱和后继，因此链表是一种线性的数组结构。而如果增加一个前驱或者后继，那么链表就变成了二叉树。

这仅仅是逻辑上的二叉树，而不是物理上的。因为树可以使用数组来实现，也可以使用链表来实现。这是因为树本身是不需要支持随机访问的

基本概念

虚拟节点

定义：数据结构中，在链表的第一个结点之前附设一个结点，它没有直接前驱，称之为虚拟结点。虚拟结点的数据域可以不存储任何信息，虚拟结点的指针域存储指向第一个结点的指针。

作用：对链表进行增删时统一算法逻辑，减少边界处理（避免了判断是否是空表或者是增删的节点是否为第一个节点）

尾节点

定义：数据结构中，尾结点是指链表中最后一个节点，即存储最后一个元素的节点。

作用：由于移动到链表末尾需要线性的时间，因此在链表末尾插入元素会很耗时，增加尾节点便于在链表末尾以 $O(1)$ 的时间插入元素。

静态链表

定义：用数组描述的链表，它的内存空间是连续的，称为静态链表。相对地，动态链表因为是动态申请内存的，所以每个节点的物理地址可以不连续，要通过指针来顺序访问。

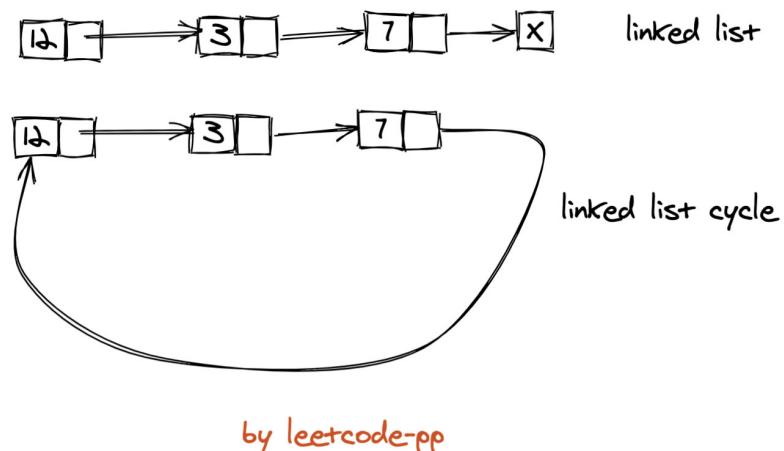
作用：既可以像数组一样在 $O(1)$ 的时间对访问任意元素，又可以像链表一样在 $O(1)$ 的时间对节点进行增删

静态链表和动态链表这个知识点对刷题帮助不大，作为了解即可。

链表分类

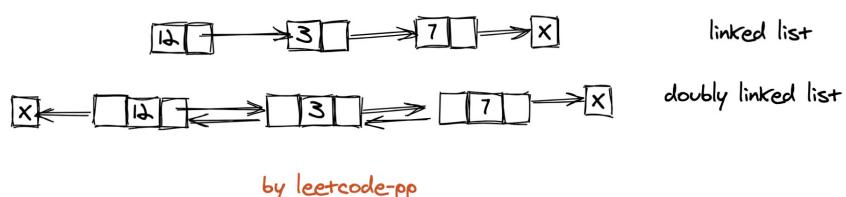
以下分类是两种分类标准，也就是一个链表可以既属于循环链表，也属于单链表，这是毋庸置疑的。

按照是否循环分为：循环链表和非循环链表



当我们需要在遍历到尾部之后重新开始遍历的时候，可以考虑使用循环链表。需要注意的是，如果链表长度始终不变，那么使用循环链表很容易造成死循环，因此循环链表经常会伴随着节点的删除操作，比如[约瑟夫环问题](#)。

按照指针个数分为：单链表和双链表



- 单链表。每个节点包括两部分：一个是存储数据的数据域，另一个是存储下一个节点指针的指针域。
- 双向链表。每个节点包括三部分：一个是存储数据的数据域，一个存储下一个节点指针的指针域，一个存储上一个节点指针的指针域。

Java 中的 LinkedHashMap 以及 Python 中的 OrderedDict 底层都是双向链表。其好处在于删除和插入的时候，可以更快地找到前驱指针。如果用单链表的话，那么时间复杂度最坏的情况是 $O(n)$ 。双向链表的本质就是空间换时间，因此如果题目对时间有要求，可以考虑使用双向链表，比如力扣的双向链表的本质就是空间换时间，因此如果题目对时间有要求，可以考虑使用双向链表，比如力扣的 [146. LRU 缓存机制](#)。

链表的基本操作

插入

插入只需要考虑要插入位置前驱节点和后继节点（双向链表的情况下需要更新后继节点）即可，其他节点不受影响，因此在给定指针的情况下插入的操作时间复杂度为 $O(1)$ 。这里给定指针中的指针指的是插入位置的前驱节点。

伪代码：

```
temp = 待插入位置的前驱节点.next
待插入位置的前驱节点.next = 待插入指针
待插入指针.next = temp
```

如果没有给定指针，我们需要先遍历找到节点，因此最坏情况下时间复杂度为 $O(N)$ 。

提示 1：考虑头尾指针的情况。

提示 2：新手推荐先画图，再写代码。等熟练之后，自然就不需要画图了。

删除

只需要将需要删除的节点的前驱指针的 next 指针修正为其下下个节点即可，注意考虑边界条件。

伪代码：

```
待删除位置的前驱节点.next = 待删除位置的前驱节点.next.next
```

提示 1: 考虑头尾指针的情况。

提示 2: 新手推荐先画图, 再写代码。等熟练之后, 自然就不需要画图了。

遍历

遍历比较简单, 直接上伪代码。

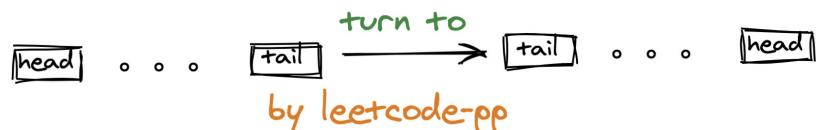
伪代码:

```
当前指针 = 头指针
while 当前指针不为空 {
    print(当前节点)
    当前指针 = 当前指针.next
}
```

常见题型

题型一：反转链表

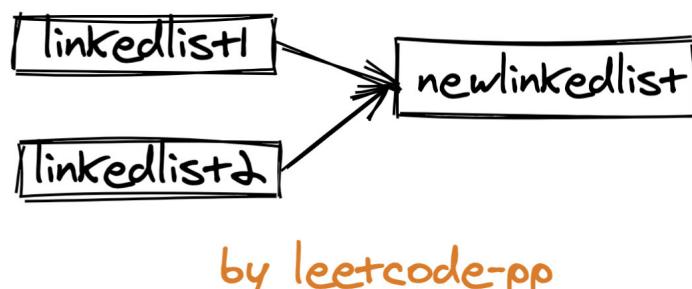
1. 将某个链表进行反转
2. 在 $O(n)$ 时间, $O(1)$ 空间复杂度下逆序读取链表的某个值
3. 将某个链表按 K 个一组进行反转



(图 1)

题型二：合并链表

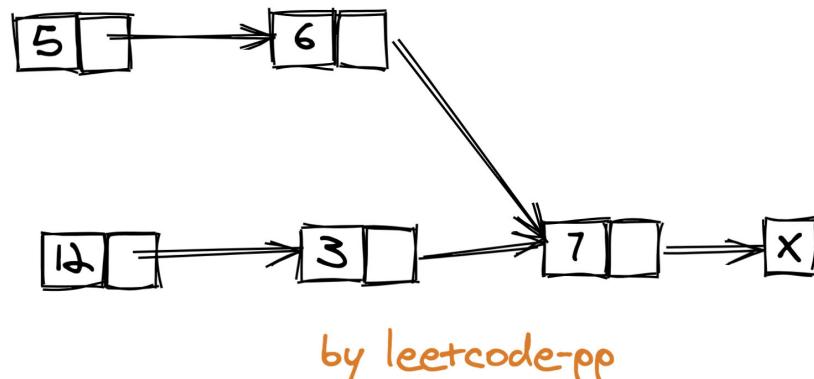
1. 将两条有序或无序的链表合并成一条有序链表
2. 将 k 条有序链表合并成一条有序链表



(图 2)

题型三：相交或环形链表

1. 判断某条链表是否存在环
2. 获取某条链表环的大小
3. 获取某两条链表的相交节点



(图 3)

题型四：设计题

要求设计一种数据结构，可以在指定的时间或空间复杂度下完成 XX 操作，这种题目的套路就是牢记所有基本数据结构的基本操作以及其复杂度。分析算法的瓶颈，并辅以恰当的数据结构进行优化。

常见套路

针对上面的四种题型，我们分别介绍如何用套路进行应对。

套路一：反转链表

伪代码：

```
当前指针 = 头指针
前一个节点 = null;
while 当前指针不为空 {
   下一个节点 = 当前指针.next;
   当前指针.next = 前一个节点
   前一个节点 = 当前指针
   当前指针 = 下一个节点
}
return 前一个节点;
```

JS 代码参考：

```
let cur = head;
let pre = null;
while (cur) {
    const next = cur.next;
    cur.next = pre;
    pre = cur;
    cur = next;
}
return pre;
```

复杂度分析

令 n 为链表总的节点数。

- 时间复杂度: $O(n)$
- 空间复杂度: $O(1)$

套路二：合并链表

伪代码:

```
ans = new Node(-1) // ans 为需要返回的头节点
cur = ans
// l1和l2分别为需要合并的两个链表的头节点
while l1 和 l2 都不为空
    cur.next = min(l1.val, l2.val)
    更新较小的指针，往后移动一位
if l1 == null
    cur.next = l2
if l2 == null
    cur.next = l1
return ans.next
```

JS 代码参考:

```

let ans = (now = new ListNode(0));
while (l1 !== null && l2 !== null) {
    if (l1.val < l2.val) {
        now.next = l1;
        l1 = l1.next;
    } else {
        now.next = l2;
        l2 = l2.next;
    }
    now = now.next;
}

if (l1 === null) {
    now.next = l2;
} else {
    now.next = l1;
}
return ans.next;

```

复杂度分析

令 n 为链表总的节点数。

- 时间复杂度: $O(n)$
- 空间复杂度: $O(1)$

套路三：相交或环形链表

链表相交求交点

解法一：哈希法

- 有 A, B 这两条链表, 先遍历其中一个, 比如 A 链表, 并将 A 中的所有节点存入哈希表。
- 遍历 B 链表, 检查节点是否在哈希表中, 第一个存在的就是相交节点

伪代码:

989. 数组形式的整数加法

```
data = new Set() // 存放A链表的所有节点的地址

while A不为空{
    哈希表中添加A链表当前节点
    A指针向后移动
}

while B不为空{
    if 如果哈希表中含有B链表当前节点
        return B
    B指针向后移动
}

return null // 两条链表没有相交点
```

JS 代码参考:

```
let data = new Set();
while (A !== null) {
    data.add(A);
    A = A.next;
}
while (B !== null) {
    if (data.has(B)) return B;
    B = B.next;
}
return null;
```

复杂度分析

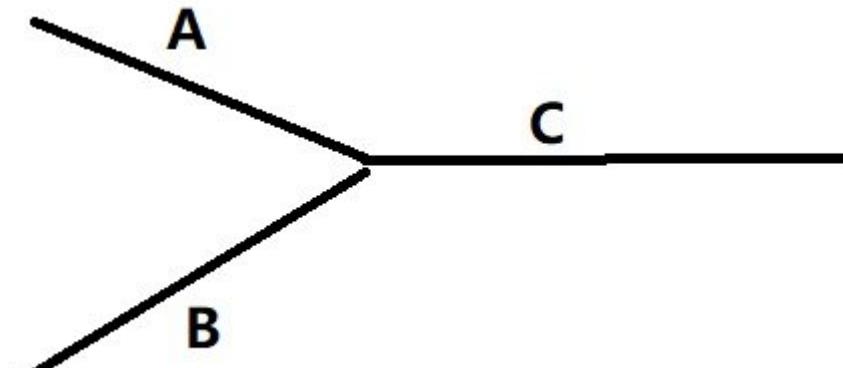
令 n 为链表总的节点数。

- 时间复杂度: $O(N)$
- 空间复杂度: $O(n)$

解法二：双指针

例如使用 a, b 两个指针分别指向 A, B 这两条链表的头，两个指针相同的速度向后移动。

- 当 a 到达链表的尾部时,重定位到链表 B 的头结点
- 当 b 到达链表的尾部时,重定位到链表 A 的头结点。
- a, b 指针相遇的点为相交的起始节点, 否则没有相交点



(图 5)

为什么 a, b 指针相遇的点一定是相交的起始节点？我们证明一下：

如果我们将两条链表按相交的起始节点截断。 a 距离交点的长度为 s_1 , b 距离交点的长度为 s_2 , 交点到链表尾部的距离是 s 。

那么 A 链表长度为: $s_1 + s_3$, B 链表为: $s_2 + s_3$ 。当 a 指针将 A 链表遍历完后, 重定位到链表 B 的头结点, 然后继续遍历至相交点。

此时 a 指针遍历的距离为 $s_1 + s_3 + s_2$, 同理 b 指针遍历的距离为 $s_2 + s_3 + s_1$ 。

显然两者走的距离是相等的, 因此该点就是相遇点。

伪代码:

```
a = headA
b = headB
while a,b指针不相等时 {
    a, b指针都向后移动
    if a, b指针都为空
        return null //没有相交点
    if a指针为空时
        a指针重定位到链表 B的头结点
    if b指针为空时
        b指针重定位到链表 A的头结点
}
return a
```

JS 代码参考:

989. 数组形式的整数加法

```
let a = headA,
    b = headB;
while (a != b) {
    a = a ? a.next : null;
    b = b ? b.next : null;
    if (a == null && b == null) return null;
    if (a == null) a = headB;
    if (b == null) b = headA;
}
return a;
```

Python 代码参考：

```
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode):
        a, b = headA, headB
        while a != b:
            a = a.next if a else headB
            b = b.next if b else headA
        return a
```

复杂度分析

令 n 为链表总的节点数。

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$

环形链表求环的起点

解法一：哈希法

- 遍历整个链表,同时将每个节点都插入哈希表,
- 如果当前节点在哈希表中不存在,继续遍历,
- 如果存在,那么当前节点就是环的入口节点

伪代码:

```
data = new Set() // 声明哈希表
while head不为空{
    if 当前节点在哈希表中存在{
        return head // 当前节点就是环的入口节点
    } else {
        将当前节点插入哈希表
    }
    head指针后移
}
return null // 环不存在
```

JS 代码参考:

```
let data = new Set();
while (head) {
    if (data.has(head)) {
        return head;
    } else {
        data.add(head);
    }
    head = head.next;
}
return null;
```

复杂度分析

令 n 为链表总的节点数。

- 时间复杂度: $O(n)$
- 空间复杂度: $O(n)$

解法二：快慢指针法

具体算法：

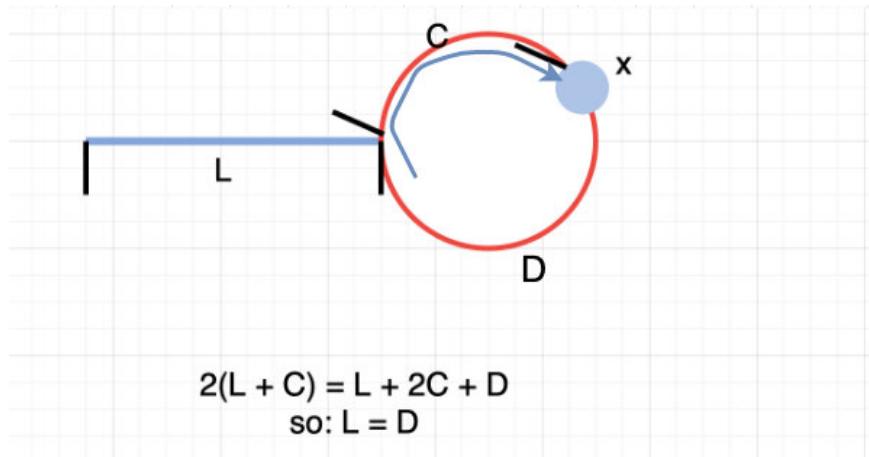
1. 定义一个 fast 指针,每次前进两步,一个 slow 指针,每次前进一步
2. 当两个指针相遇时
 - i. 将 fast 指针重定位到链表头部,同时 fast 指针每次只前进一步
 - ii. slow 指针继续前进,每次前进一步
3. 当两个指针再次相遇时,当前节点就是环的入口

下面我们对此方法的正确性进行简单证明:

- x 表示第一次相遇点
- L 是起点到环的入口点的距离
- C 是环的入口点到第一次相遇点的距离

989. 数组形式的整数加法

- D 是环的周长减去 C



$L + C$ 是慢指针走的距离，而快指针走的距离是慢指针的两倍，也就是 $2(L + C)$ ，而快指针走的距离也可以用 $L + 2C + D$ 表示，二者结合可以得出 L 和 D 相等。

因此我们可以在两者第一次相遇后将快指针放回开头，这样二者再次相遇的点一定是环的入口点，这是因为 L 和 D 是相等的。

That's all!

参考： [【每日一题】 - 2020-01-14 - 142. 环形链表 II](#)

伪代码：

```
fast = head
slow = head // 快慢指针都指向头部
do {
    快指针向后两步
    慢指针向后一步
} while 快慢指针不相等时
if 指针都为空时{
    return null // 没有环
}
while 快慢指针不相等时{
    快指针向后一步
    慢指针向后一步
}
return fast
```

JS 代码参考：

```
if (head == null || head.next == null) return null;
let fast = (slow = head);
do {
    if (fast != null && fast.next != null) {
        fast = fast.next.next;
    } else {
        fast = null;
    }
    slow = slow.next;
} while (fast != slow);
if (fast == null) return null;
fast = head;
while (fast != slow) {
    fast = fast.next;
    slow = slow.next;
}
return fast;
```

复杂度分析

令 n 为链表总的节点数。

- 时间复杂度: $O(n)$
- 空间复杂度: $O(1)$

套路四：求链表倒数第 k 个节点

我们假设 k 是一个不大于链表长度的正整数。

算法：

使用两个指针。第一个指针先走 k 步，接下来第二个指针再走。这样当第一个指针最到链表末尾（指向空）的时候，第二个指针刚好位于倒数第 k 个。

套路五：设计题

这个直接结合一个例子来给大家讲解一下。

题目描述：

设计一个算法支持以下操作：

获取数据 `get` 和 写入数据 `put`。

获取数据 `get(key)` – 如果关键字（key）存在于缓存中，则获取关键字的值

写入数据 `put(key, value)` – 如果关键字已经存在，则变更其数据值；如果

在 $O(1)$ 时间复杂度内完成这两种操作

思路：

1. 确定需要使用的数据结构

- i. 根据题目要求，存储的数据需要保证顺序关系（逻辑层面） \implies 使用数组、链表等保证循序关系
- ii. 同时需要对数据进行频繁的增删，时间复杂度 $O(1)$ \implies 使用链表等
- iii. 对数据进行读取时，时间复杂度 $O(1)$ \implies 使用哈希表

最终采取双向链表 + 哈希表

- i. 双向链表按最后一次访问的时间的顺序进行排列，链表头部为最近访问的节点
- ii. 哈希表，以关键字为键，以链表节点的地址为值

2. `put` 操作

通过哈希表，查看传入的关键字对应的链表节点，是否存在

- i. 如果存在，
 - i. 将该链表节点的值更新
 - ii. 将该链表节点调整至链表头部
- ii. 如果不存在
 - i. 如果链表容量未满，
 - i. 新生成节点，
 - ii. 将该节点位置调整至链表头部
 - ii. 如果链表容量已满
 - i. 删除尾部节点
 - ii. 新生成节点
 - iii. 将该节点位置调整至链表头部
 - iii. 将新生成的节点，按关键字为键，节点地址为值插入哈希表

3. `get` 操作

通过哈希表，查看传入的关键字对应的链表节点，是否存在

- i. 节点存在
 - i. 将该节点位置调整至链表头部

989. 数组形式的整数加法

- ii. 返回该节点的值
- ii. 节点不存在, 返回 null

伪代码:

```
var LRUcache = function(capacity) {
    保存一个该数据结构的最大容量
    生成一个双向链表,同时保存该链表的头结点与尾节点
    生成一个哈希表
};

function get (key) {
    if 哈希表中存在该关键字 {
        根据哈希表获取该链表节点
        将该节点放置于链表头部
        return 链表节点的值
    } else {
        return -1
    }
};

function put (key, value) {
    if 哈希表中存在该关键字 {
        根据哈希表获取该链表节点
        将该链表节点的值更新
        将该节点放置于链表头部
    } else {
        if 容量已满 {
            删除链表尾部的节点
            新生成一个节点
            将该节点放置于链表头部
        } else {
            新生成一个节点
            将该节点放置于链表头部
        }
    }
};
```

JS 代码参考:

989. 数组形式的整数加法

```
function ListNode(key, val) {
    this.key = key;
    this.val = val;
    this.pre = this.next = null;
}

var LRUcache = function (capacity) {
    this.capacity = capacity;
    this.size = 0;
    this.data = {};
    this.head = new ListNode();
    this.tail = new ListNode();
    this.head.next = this.tail;
    this.tail.pre = this.head;
};

function get(key) {
    if (this.data[key] !== undefined) {
        let node = this.data[key];
        this.removeNode(node);
        this.appendHead(node);
        return node.val;
    } else {
        return -1;
    }
}

function put(key, value) {
    let node;
    if (this.data[key] !== undefined) {
        node = this.data[key];
        this.removeNode(node);
        node.val = value;
    } else {
        node = new ListNode(key, value);
        this.data[key] = node;
        if (this.size < this.capacity) {
            this.size++;
        } else {
            key = this.removeTail();
            delete this.data[key];
        }
    }
    this.appendHead(node);
}

function removeNode(node) {
    let preNode = node.pre,
```

```
nextNode = node.next;
preNode.next = nextNode;
nextNode.pre = preNode;
}

function appendHead(node) {
    let firstNode = this.head.next;
    this.head.next = node;
    node.pre = this.head;
    node.next = firstNode;
    firstNode.pre = node;
}

function removeTail() {
    let key = this.tail.pre.key;
    this.removeNode(this.tail.pre);
    return key;
}
```

题目推荐

如果你将本文的内容全部看完了，则可以尝试以下题目。当解题过程遇到困难，不妨回头看一下讲义。

- [21. 合并两个有序链表](#)
- [82. 删除排序链表中的重复元素 II](#)
- [83. 删除排序链表中的重复元素](#)
- [86. 分隔链表](#)
- [92. 反转链表 II](#)
- [138. 复制带随机指针的链表](#)
- [141. 环形链表](#)
- [142. 环形链表 II](#)
- [143. 重排链表](#)
- [148. 排序链表](#)
- [206. 反转链表](#)
- [234. 回文链表](#)

总结

链表是比较简单也非常基础的数据结构，所以大家一定要熟练掌握。

链表的常规题目基本都是考察指针操作，只要你做到心中有链，切记出现环，就离成功不远了。再利用一些诸如哨兵节点的技巧简化代码，相信你写出 bug free 代码也不是难事。

989. 数组形式的整数加法

在碰到设计题这种对数据结构的设计能力要求较高时，一般会需要使用到 2-3 种数据结构，这时要根据具体的使用场景去分析，如何将各种数据结构的优势结合到一起，慢慢大家写的多了，碰到设计题就有了固定的思维模式，ac 也就水到渠成。

树

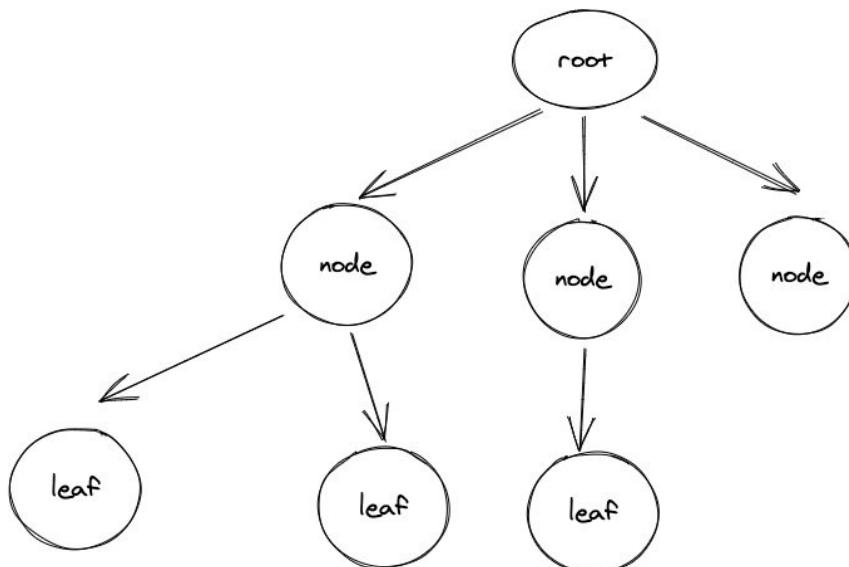
介绍

计算机的数据结构是对现实世界物体间关系的一种抽象。比如家族的族谱，公司架构中的人员组织关系，电脑中的文件夹结构，html 渲染的 dom 结构等等，这些有层次关系的结构在计算机领域都叫做树。

我们平时做题时候的树其实是一种逻辑结构。

基本概念

树是一种非线性数据结构。树结构的基本单位是节点。节点之间的链接，称为分支（branch）。节点与分支形成树状，结构的开端，称为根（root），或根结点。根节点之外的节点，称为子节点（child）。没有链接到其他子节点的节点，称为叶节点（leaf）。如下图是一个典型的树结构：



每个节点可以用以下数据结构来表示：

```

Node {
  value: any; // 当前节点的值
  children: Array<Node>; // 指向其儿子
}
  
```

其他重要概念：

- 树的高度：节点到叶子节点的最大值就是其高度。

- 树的深度：高度和深度是相反的，高度是从下往上数，深度是从上往下。因此根节点的深度和叶子节点的高度是 0；
- 树的层：根开始定义，根为第一层，根的孩子为第二层。
- 二叉树，三叉树，。。。N 叉树，由其子节点最多可以有几个决定，最多有 N 个就是 N 叉树。

二叉树

二叉树是树结构的一种，两个叉就是说每个节点最多只有两个子节点，我们习惯称之为左节点和右节点。

注意这个只是名字而已，并不是实际位置上的左右

二叉树也是我们做算法题最常见的一种树，因此我们花大篇幅介绍它，大家也要花大量时间重点掌握。

二叉树可以用以下数据结构表示：

```
Node {
    value: any; // 当前节点的值
    left: Node | null; // 左儿子
    right: Node | null; // 右儿子
}
```

二叉树分类

- 完全二叉树
- 满二叉树
- 二叉搜索树
- [平衡二叉树](#)
- 红黑树
- . . .

二叉树的表示

- 链表存储
- 数组存储。非常适合完全二叉树

二叉树遍历

二叉树的大部分题都围绕二叉树遍历展开，二叉树主要有以下遍历方式：

1. 前序遍历
2. 中序遍历
3. 后序遍历

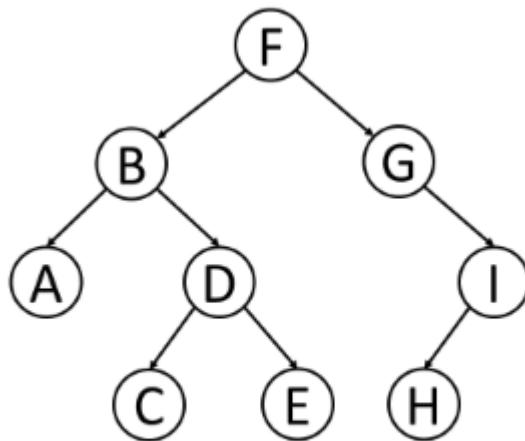
4. 层序遍历(BFS)

你如果想锯齿遍历也可以， 不过除非特意考察你这个点， 否则我们仅考虑以上的基本遍历方式

前序遍历

- 前序遍历的顺序
 1. 访问当前节点
 2. 遍历左子树
 3. 遍历右子树

如下动图很好地演示了前序遍历算法的过程。



Preorder:

前序遍历的伪代码：

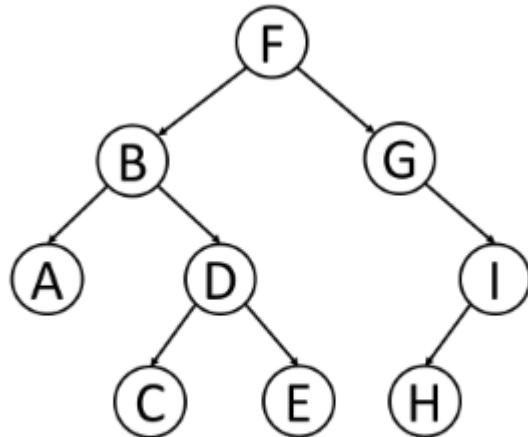
```

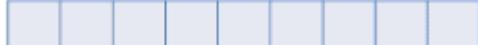
preorder(root) {
    if not root: return
    doSomething(root)
    preorder(root.left)
    preorder(root.right)
}
  
```

中序遍历

- 中序遍历的顺序
 1. 遍历左子树
 2. 访问当前节点
 3. 遍历右子树

如下动图很好地演示了中序遍历算法的过程。



Inorder: 

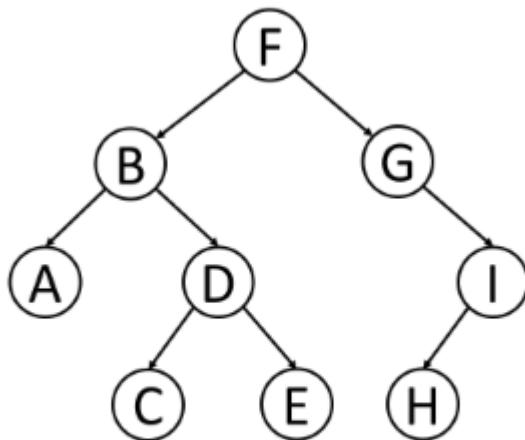
中序遍历的伪代码：

```
inorder(root) {  
    if not root: return  
    inorder(root.left)  
    doSomething(root)  
    inorder(root.right)  
}
```

后续遍历

- 后序遍历的顺序
 1. 遍历左子树
 2. 遍历右子树
 3. 访问当前节点

如下动图很好地演示了后序遍历算法的过程。



Postorder:

后序遍历的伪代码：

```

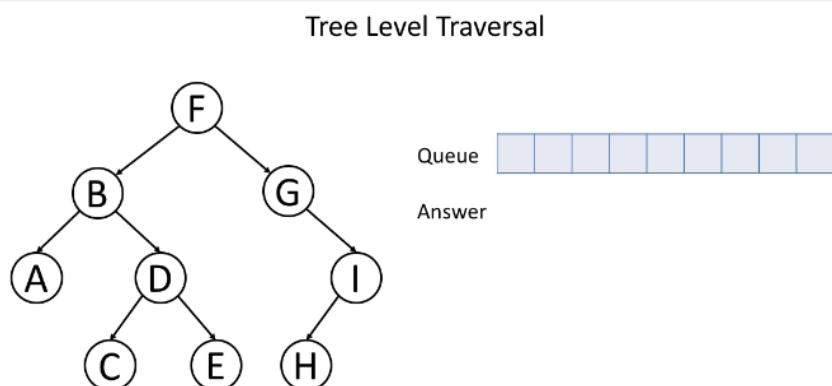
postorder(root) {
    if not root: return
    postorder(root.left)
    postorder(root.right)
    dosomething(root)
  
```

层序遍历(BFS)

层次遍历从直观上会先遍历树的第一层，再遍历树的第二层，以此类推。

具体算法上，我们可是使用 DFS 并记录当前访问层级的方式实现，不过更多的时候还是使用借助队列的先进先出的特性来实现。关于队列，我们已经在第一节的时候讲过了。

如下动图很好地演示了层次遍历算法的过程。



二叉树层次遍历伪代码：

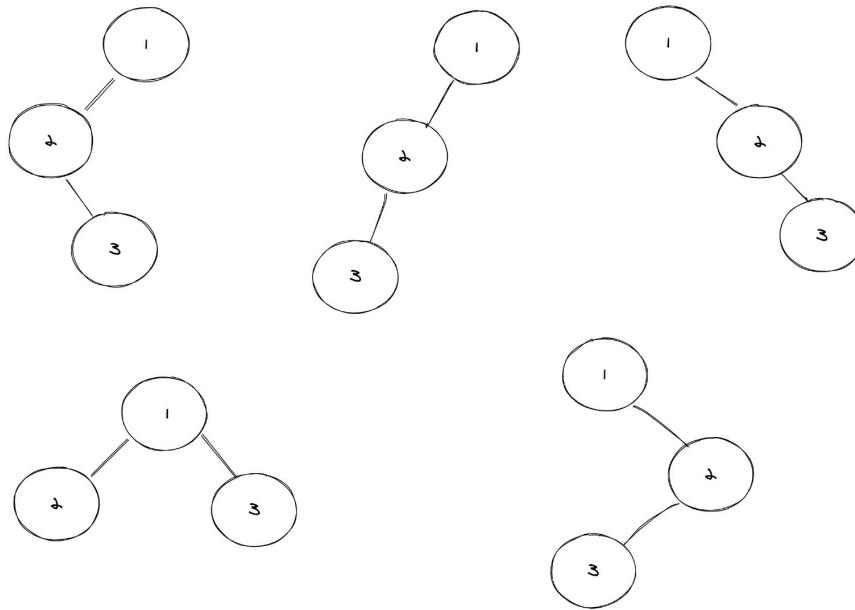
```

bfs(root) {
    queue = []
    queue.push(root)
    while queue.length {
        curLevel = queue
        queue = []
        for i = 0 to curLevel.length {
            doSomething(curLevel[i])
            if (curLevel[i].left) {
                queue.push(curLevel[i].left)
            }
            if (curLevel[i].right) {
                queue.push(curLevel[i].right)
            }
        }
    }
}

```

二叉树构建

二叉树有一个经典的题型就是构造二叉树。注意单前/中/后序遍历是无法确定一棵树，比如以下所有二叉树的前序遍历都为 123



但是中序序列和前、后，层次序列任意组合唯一确定一颗二叉树（前提是遍历是基于引用的或者二叉树的值都不相同）。

前、中，层次序列都是提供根结点的信息，中序序列用来区分左右子树。

实际上构造一棵树的本质是：

1. 确定根节点

2. 确定其左子树
3. 确定其右子树

比如拿到前序遍历结果 `preorder` 和中序遍历 `inorder`, 在 `preorder` 我们可以确定树根 `root`, 拿到 `root` 可以将中序遍历切割成左右子树。这样就可以确定并构造一棵树，整个过程我们可以用递归完成。详情见 [构建二叉树专题](#)

二叉搜索树

二叉搜索树是二叉树的一种，具有以下性质

1. 左子树的所有节点值小于根的节点值（注意不含等号）
2. 右子树的所有节点值大于根的节点值（注意不含等号）

另外二叉搜索树的中序遍历结果是一个有序列表，这个性质很有用。比如 [1008. 前序遍历构造二叉搜索树](#)。根据先序遍历构建对应的二叉搜索树，由于二叉树的中序遍历是一个有序列表，我们可以有以下思路

1. 对先序遍历结果排序，[排序结果是中序遍历结果](#)
2. 根据先序遍历和中序遍历确定一棵树

原问题就又转换为了上面我们讲的《二叉树构建》。

类似的题目很多，不再赘述。练习的话大家可以做一下这几道题。

- [94. 二叉树的中序遍历](#)
- [98. 验证二叉搜索树](#)
- [173. 二叉搜索树迭代器](#)
- [250. 统计同值子树](#)

大家如果碰到二叉搜索树的搜索类题目，一定先想下能不能利用这个性质来做。

堆

在这里讲堆是因为堆可以被看作近似的完全二叉树。堆通常以数组形式的存储，而非上述的链式存储。

表示堆的数组 `A` 中，如果 `A[1]` 为根节点，那么给定任意节点 `i`，其父子节点分别为

- 父亲节点：`Math.floor(i / 2)`
- 左子节点：`2 * i`
- 右子节点：`2 * i + 1`

如果 `A[parent(i)] ≥ A[i]`，则称该堆为最大堆，如果 `A[parent(i)] ≤ A[i]`，称该堆为最小堆。

堆这个数据结构有很多应用，比如堆排序，TopK问题，共享计算机系统的作业调度(优先队列)等。下面看下给定一个数据如何构建一个最大堆。

伪代码：

```
// 自底向上建堆
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = Math.floor(A.length / 2) downto 1
        MAX-HEAPIFY(A, i)

// 维护最大堆的性质
MAX-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    // 找到当前节点和左右儿子节点中最大的一个，并交换
    if l <= A.heap-size and A[l] > A[i]
        largest = l
    else largest = i
    if r <= A.heap-size and A[r] > A[largest]
        largest = r
    if largest != i
        exchange A[i] with A[largest]
        // 递归维护交换后的节点堆性质
        MAX-HEAPIFY(A, largest)
```

ps: 伪代码参考自算法导论

关于堆的更多内容，请参考：

- [几乎刷完了力扣所有的堆题，我发现了这些东西。。。 \(上\)](#)
- [几乎刷完了力扣所有的堆题，我发现了这些东西。。。 \(下\)](#)

递归

简介

二叉树是一种递归的数据结构，是最能体现递归美感的结构之一，看到二叉树的题第一反应就应该是用递归去写。

递归就是方法或者函数调用自身的方式成为递归调用。在这个过程中，调用称之为递，返回成为归。

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)
```

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。

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。

推荐题目

- 汉诺塔问题
- fibonacci 数列
- 二叉树的前中后序遍历

- 归并排序
- 求阶乘
- 递归求和

相关专题

- 二叉树的最大路径和
- 给出所有路径和等于给定值的路径
- 最近公共祖先
- 各种遍历。前中后，层次，拉链式等。
- [专题篇 - 搜索](#)
- [二叉树的遍历](#)
- [前缀树专题](#)

题目推荐

- [589. N 叉树的前序遍历](#) (熟悉 N 叉树)
- [662. 二叉树最大宽度](#) (请分别使用 BFS 和 DFS 解决，空间复杂度尽可能低)
- [834. 树中距离之和](#) (谷歌面试题)
- [967. 连续差相同的数字](#) (隐形树的遍历)
- [1145. 二叉树着色游戏](#) (树上进行决策)

总结

树是一种很重要的数据结构，而我们研究树又以研究二叉树为主。

二叉树去掉一个子节点就是链表，增加环就是图。它和很多数据结构和算法都有关联。因此掌握树以及树的各种算法就显得尤其重要。本章提到的内容都是经过我的筛选，去掉那些对刷题不那么重要的内容，因此这剩下的内容大家一定要熟练使用才行。

对于刷题来说，二叉树特别适合练习递归。一方面是其数据结构天生的递归性，另一方面树比链表这种递归数据结构复杂，树是非线性的，因此可以出的题相对比较多。

关于树的题目的几个技巧，请参考 [几乎刷完了力扣所有的树题，我发现了这些东西。。。](#)

参考文献

- 图片参考自 <https://wylu.me/posts/e85d694a/>
- 《算法导论》

哈希表

哈希表是一种在平均时间复杂度 $O(1)$ 内可实现任何操作的数据结构，这里的操作包括查询，插入，删除以及修改。需要注意的是这里描述的是平均时间复杂度，最坏的情况仍然有可能是线性的。

平均时间复杂度 是否能达到 $O(1)$ 和哈希算法以及冲突处理算法都有关，也就是说需要你的哈希函数设计地足够好才能达到平均时间复杂度 $O(1)$

听起来很神奇？没错！那么问题来了。

- 既然哈希表这么好，那数组和链表还有存在的价值么？
- 哈希表是如何实现平均时间复杂度 $O(1)$ 的呢？

带着这两个疑问，大家继续看下看。

介绍

哈希的基本思想其实就是建立恰当大小值域的数组进行某种映射。比如：

```
size = 10
hash_number = 0

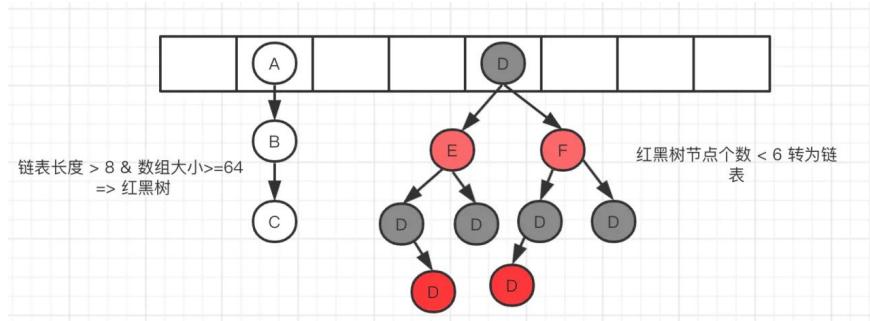
for(c in s):
    hash_number += ord(c)
    hash_number %= size
print(hash_number)
```

如上建立了一个大小为 10 的值域，值域的值分别是 0, 1, 2, 3, 4, 5, 6, 7, 8, 9。接下来，我们对字符串逐个字符取 ascii 码并相加模 10，模 10 可以确保值不会越界。这就是一个最简单的 **hash** 思想。

不要小看这个思想，后面我们要讲的进阶篇 RK 算法的核心也是如此。

虽然要想实现一个工业级别的哈希表需要考虑的东西要远远大于这些，不过哈希表的实现的核心也是如此。

散列表（Hash table，也叫哈希表），是根据关键码值(Key)而直接进行访问的数据结构。散列表可以使用数组 + 链表的方式来实现，也可以用别的方式，比如数组 + 红黑树。JDK1.8 的 HashMap 就同时使用了这两种方式。



两个精髓

哈希表查询的精髓就在于数组，哈希表查找的平均时间复杂度 $O(1)$ 就是因为这个。用数组存数据，查询时间复杂度 $O(1)$ 很容易，但是数据删除和新增操作，使用数组的平均时间复杂度会变成 $O(N)$ ，这个我们在之前的章节中讲述过。

如何解决数组在动态性下的弱势呢？答案是链表或者树，链表和树对动态数据很友好，而哈希表删除和新增的精髓就在于链表或者树。哈希表新增和删除的精髓就在于链表或树，哈希表修改和删除的平均时间复杂度 $O(1)$ 就是因为这个。

这两个精髓大家要牢牢记住，具体内容我们后面再详细讲述。

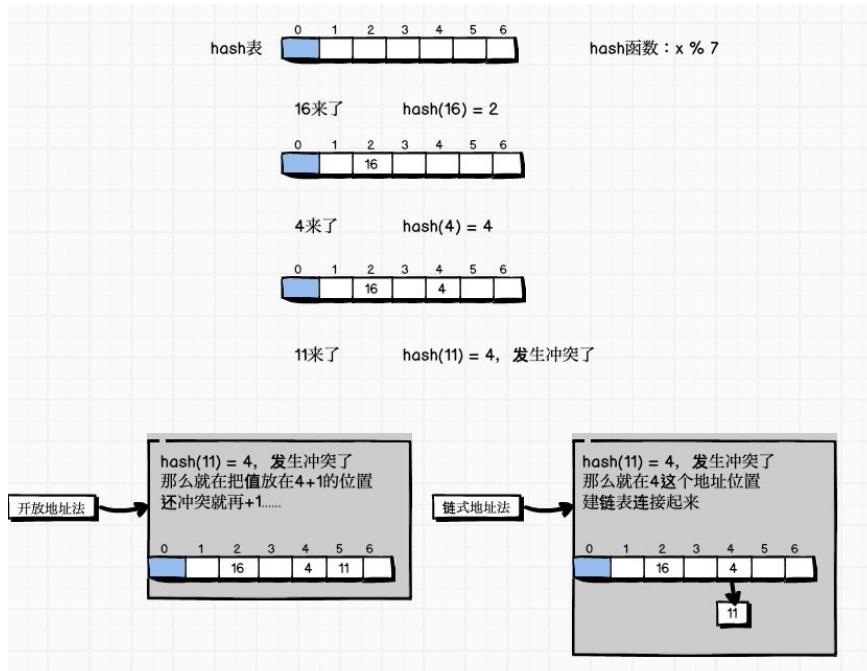
冲突

我前面说了哈希查询的精髓在于数组，而这里数组的索引就是哈希表的 key，我们知道哈希表可以存任意 string，而数组索引只能是数字，且数组的索引范围是 $[0, n)$ ，其中 n 为数组长度。这怎么办呢？我们使用哈希函数来解决这个问题。你可以把哈希函数当成一个神奇的函数，它的输入是 key，返回值是索引，并且相同的 key 计算出的索引是确定不变的，也就是说哈希函数需要是数学中的函数。

而理论上两个不同的 key 是可以算出同样的 hashcode 的。这个就叫做哈希冲突。那什么是哈希冲突呢？

哈希函数不是万能的，根据抽屉原理，除非你给一个容器足够大的抽屉（工程中不现实），否则不可避免的可能会造成两个不同的 Key 算出来的 hash 值相同，比如 hash 函数是 $x \% 3$ ，这样 key=2 和 key=5 算出的 hash 值都为 2，这是要怎么办呢？一般我们有两种方法来处理，开放地址法和拉链法，具体大家可以查阅相关资料。这里简单的画了个图给大家直观看一下大概意思：

989. 数组形式的整数加法



上面中两种方法都是遇到冲突的解决方式。其实我们也可以防患于未然，编写冲突小的哈希函数。比如 JDK1.8 的哈希函数就是先拿到通过 key 的 hashCode，是 32 位的 int 值，然后让 hashCode 的高 16 位和低 16 位进行异或操作。之所以这么做是因为这样做的哈希冲突的概率会小，

构造一个冲突小，稳定性高的 hash 函数是很重要的，我们在刷题的时候大部分时间都不会去考虑这个问题，但是实际工程中有时不可避免需要我们自己构造 hash 函数，这时就要根据实际情况进行分析测试啦。

最后偷偷说一下，用 Java 的同学有兴趣可以看看 HashSet 的源码，底层也是用的 HashMap 😊

以上是一些理论知识，大家可以通过力扣的 [705. 设计哈希集合](#) 和 [706. 设计哈希映射](#) 检验下自己的学习成果哦~

常用操作的时间复杂度

- 插入：O(1)
- 删除：O(1)
- 查找：O(1)

常用的操作在非极其特殊情况下，平均的时间复杂度都为**O(1)**

常见题目类型

- 统计 xx 出现次数/频率/（见下方多人运动）

该种题比较直观，若已知数据范围较小且比较连续，可以考虑用数组来实现。

题目推荐：

- [811. 子域名访问计数](#)
- [需要查找/增加/删除操作为 O\(1\)时间复杂度](#) (一些设计题)

见到这种要求的题可以考虑一下是否需要 hash 表来做，比如 LRU, LFU 之类的题，题目中要求了时间复杂度，就是用 hash 表+双向链表解决的。

- 题目类型为图数据结构相关 (比如并查集)

这样可能需要构建有向图/无向图，这时可以用 hash 表来表示图并进行后续操作。

- 需要存储之前的状态以减少计算开销 (比如经典的两数和)

相信大家做过 dp 的一些题目就知道，记忆化搜索，该方法就利用 hash 表来存储历史状态，这样可以大大减少重复计算。

- 状态压缩 (本质就是 bit 上的哈希结构)。参考 [状压 DP 是什么？这篇题解带你入门](#)
- 记录第一次出现的位置，以便求最远或者最近。比如题目 [697. 数组的度](#)
- 等等，大家多做类似的题目，相信可以总结出一套自己的思路。

模板（伪代码）

1. 判断目标值是否出现过 (例题如：两数之和、是否存在重复元素、合法数独等等)

```
for num in nums:
    if num(该处为目标值target) in hashtable:
        return true
    return false
```

1. 统计频率

数据比较离散

```
for num in nums:
    if num in hashtable:
        hashtable[num] += 1
    else:
        hashtable[num] = 1
# 后续操作
-----
```

数据范围较小且连续则可以用数组代替

```
// 假设数据范围是0-n且n较小
int[] hashtable = new int[n + 1];

for num in nums:
    hashtable[num] += 1;

// 后续操作
-----
```

题目推荐 - 2 人运动

题目描述

已知小猪每晚都要约好几个女生到酒店房间。每个女生 i 与小猪约好的时间由 $[s_i, e_i]$ 表示，其中 s_i 表示女生进入房间的时间， e_i 表示女生离开房间的时间。由于小猪心胸开阔，思想开明，不同女生可以同时存在于小猪的房间。请计算出小猪最多同时在做几人的「多人运动」。

例子：

Input : [[0 , 30] ,[5 , 10] , [15 , 20]]

OutPut : 最多同时有两个女生的「三人运动」

思路

这个题解法不止一种，但是我们这里因为在讲 hash 表，统计频率。下面我只写一下大致思路的伪代码，具体细节大家不妨可以尝试自己实现一下。

```
// 上面刚刚说了关于频率统计的方法，这里读完题，是不是就立刻想到了：  
// 用hash表来统计每个时刻房间内的人数并维护一个最大值就是我们所求的结  
  
res = -1  
  
for everyGirl in girls:  
    for curTime in [everyGirl.start, everyGirl.end]:  
        // 套上面板子  
        if curTime in hashtable:  
            hashtable[curTime] += 1  
        else:  
            hashtable[curTime] = 1  
  
        // 维护最大值  
        res = max(res, hashtable[curTime])  
  
-----
```

线下验证通过可以贴到这里哦， [【每日一题】 - 2020-04-27 - 多人运动](#)

这里还有各种解题方法，大家都可以学习下思路并试着自己做一做！

其他题目推荐：

- [218. 天际线](#) (使用哈希统计可能会 OOM，但是思路上可行)
- [面试题 01.04. 回文排列](#)
- [500. 键盘行](#)
- [36. 有效的数独](#)
- [37. 解数独](#) 与 36 类似，还需要点回溯的思想

回答开头的两个问题

我们来看下开头提出的两个问题：

- 既然哈希表这么好，那数组和链表还有存在的价值么？

这其实是一个伪问题。实际上哈希表是数组和链表（或者树）实现的。没有数组和链表这些基础数据结构，哈希表没办法实现。

那我是不是可以在任何用数组的地方都用哈希表呢？

对于数组来说，当然可以。无非就是把数字的索引变成对应的字符串即可，但是会造成空间和时间上的浪费。

其实很多时候，我们会用数组来实现哈希表的计数功能。比如给你一个字符串 s，s 只包括小写英文字母，要你对字符串 s 中的所有字符进行统计其出现的次数。使用哈希表当然可以，但是这种知道容量的情况，我们通常使用数组来做。在这里，容量就是固定的 26。

代码：

```
def count(s):
    counts = [0] * 26
    for i in range(len(s)):
        counts[ord(s[i]) - ord('a')] += 1
```

这种使用固定大小数组的方法非常常见。其实如果你仔细观察，这就是一个不需要处理冲突的迷你哈希表。哈希函数就是 `ord(s[i]) - ord('a')`。

而对于链表来说，哈希表是没有办法替代的。因此哈希表的一些基础底层操作被哈希表封装了，无法使用到了。

- 哈希表是如何实现平均时间复杂度 $O(1)$ 的呢？

上面讲的两个精髓可以很好地回答这个问题。

- 哈希表查询的精髓就在于数组，哈希表查找的平均时间复杂度 $O(1)$ 就是因为这个。
- 哈希表新增和删除的精髓就在于链表或树，哈希表修改和删除的平均时间复杂度 $O(1)$ 就是因为这个。

总结

哈希表是一种“比较全能”的数据结构，常用的操作在非极其特殊情况下，平均的时间复杂度都为 $O(1)$ 。

做题的时候，不会太关注哈希表的原理以及冲突，我们对此有一定的了解即可。大家应该把重点放在应用常见上。

这里我列举了几种常见的哈希表的应用场景，分别是：

- 统计 xx 出现次数/频率/
- 需要查找/增加/删除操作为 $O(1)$ 时间复杂度（一些设计题）
- 题目类型为图数据结构相关（比如并查集）
- 需要存储之前的状态以减少计算开销（比如经典的两数和）
- 状态压缩（本质就是 bit 上的哈希结构）

最后给大家几个模板，大家可以使用这几个模板和文章给的做题思路去完成文章中推荐的题目。

双指针

什么是双指针

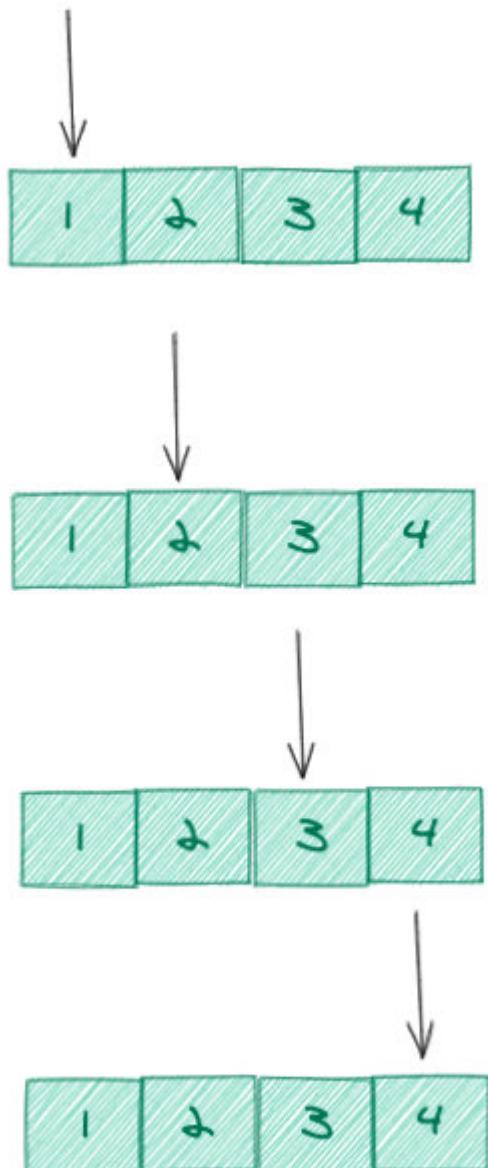
顾名思议，双指针就是两个指针，但是不同于 C, C++ 等语言中的指针，其是一种算法思想。

如果说迭代一个数组，并输出数组每一项需要一个指针来记录当前遍历项，这个过程我们叫单指针的话。

这里的“指针”指的是数组的索引

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

989. 数组形式的整数加法



(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。

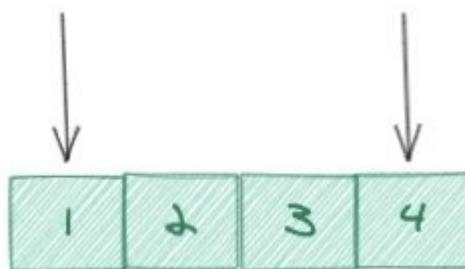
当然这里的“指针”不仅可以是数组的索引，也可以是别的。不过在大多数情况下，都是数组的索引。因此数组双指针就是有两个这样的数组索引。

```

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

while (l < r) {
    if(一定条件) return 答案
    if(一定条件) l++
    if(一定条件) r--
}
// 由于循环结束的时候 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型，分别是：

1. 快慢指针（两个指针步长不同，一个步长大，一个步长小。典型的是一个步长为 1，另外一个步长为 2）
2. 左右端点指针（两个指针分别指向头尾，并往中间移动，步长关系不确定）
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，比如二分法每次将数据规模缩小为一半，其时间复杂度就是 $O(\log N)$ 。实际上，很多二分法都需要两个指针，一个指向边界，一个指向中心，这个时候其实它就是一种特殊的双指针。

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

1. 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。而这两道题都可以使用快慢双指针解决。

- [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，我给出一个题解，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](#)
- 一次遍历（One Pass）求链表的中点

直观的思路是先进行一次遍历求出链表长度 n ，然后再次遍历链表，走 $n/2$ 次即可。而这需要两次遍历，我们可以使用快慢双指针来优化这个过程。

具体算法是 使用两个指针。快指针每次走两步，慢指针每次走一步，这样当快指针走到链表尾部的时候，慢指针刚好到达链表中间位置。

2. 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。

3. 固定间距指针

1. 一次遍历（One Pass）求链表的倒数第 k 个元素

2. 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)

更多二分推荐：

- [search-for-range](#)
- [search-insert-position](#)
- [search-a-2d-matrix](#)
- [first-bad-version](#)

- [find-minimum-in-rotated-sorted-array](#)
- [find-minimum-in-rotated-sorted-array-ii](#)
- [search-in-rotated-sorted-array](#)
- [search-in-rotated-sorted-array-ii](#)

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)

滑动窗口

其实滑动窗口就是借助双指针来完成的，其中两个指针分别是窗口的左右两个边界。

- 如果我使用固定间距的双指针，那就是窗口大小固定的双指针。
- 如果我使用快慢双指针，那就是可变窗口大小的双指针，一般这种题目都是求满足一定条件的窗口的最大或者最小值。

滑动窗口见专题篇的[滑动窗口专题](#)

可变窗口大小模板（伪代码）

固定窗口大小和上面代码类似，直接使用就行。因此这里重点看下可变窗口大小模板。

```
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
    更新答案
    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]) -> int:
        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）](#)
- [978. 最长湍流子数组](#)
- [【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）](#)
- [1658. 将 x 减到 0 的最小操作数](#)

如果你理解了滑动窗口，那就快用我的模板试试解决这些问题吧~

扩展阅读

- [LeetCode Sliding Window Series Discussion](#)

总结

广义的双指针是一个非常宽泛的话题，因为其实我只需要有两个指针就可以了。双指针只是一个方便记忆的名字，实际上如果是固定窗口大小的滑动窗口只需要一个指针就够了，用不到双指针，不过我习惯还是将其归于

989. 数组形式的整数加法

双指针。

而我们讨论的是狭义的双指针，具体来说有以下三种类型：

1. 快慢指针
2. 首尾双指针
3. 滑动窗口双指针。 (使用两个指针表示一个窗口)

每一种都给了使用场景和模板供大家参考，这节涉及的题目比较多，但是使用我们的思维方式和模板都可以轻松解决，前提是你要花时间理解和练习。

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/>

双指针还有很多其他的类型，比如之后要讲的二分法。二分法的实现大多也是使用两个指针，并通过不断二分解空间实现。

图 Graph

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。

基本概念

- 无向图 & 有向图 [Undirected Graph & Directed Graph]
- 有权图 & 无权图 [Weighted Graph & Unweighted Graph]
- 入度 & 出度 [Indegree & Outdegree]
- 路径 & 环 [路径：Path]
 - 有环图 [Cyclic Graph]
 - 无环图 [Acyclic Graph]
- 连通图 & 强连通图

在无向图中，若任意两个顶点 i 与 j 都有路径相通，则称该无向图为连通图。

在有向图中，若任意两个顶点 i 与 j 都有路径相通，则称该有向图为强连通图。

- 生成树

一个连通图的生成树是指一个连通子图，它含有图中全部 n 个顶点，但只有足以构成一棵树的 $n-1$ 条边。一颗有 n 个顶点的生成树有且仅有 $n-1$ 条边，如果生成树中再添加一条边，则必定成环。在连通网的所有生成树中，所有边的代价和最小的生成树，称为最小生成树，其中代价和指的是所有边的权重和。

图的建立

一般图的题目都不会给你一个现成的图结构。当你知道这是一个图的题目时候，解题的第一步通常就是建图。这里我简单介绍两种常见的建图方式。

图是有点和边组成的。理论上，我们只要存储图中的所有的边关系即可。

因此我们可以使用数组或者哈希表来存储图，这里我们用二维数组来存储。

邻接矩阵（常见）（Adjacency Matrixs）

使用一个 $n * n$ 的矩阵来描述图 graph，其就是一个二维的矩阵，其中 $graph[i][j]$ 描述边的关系。

一般而言，我都用 $graph[i][j] = 1$ 来表示 顶点 i 和顶点 j 之间有一条边，并且边的指向是从 i 到 j。用 $graph[i][j] = 0$ 来表示 顶点 i 和顶点 j 之间不存在一条边。对于有权图来说，我们可以存储其他数字，表示的是权重。

这种存储方式的空间复杂度为 $O(n^2)$ ，其中 n 为顶点个数。如果是稀疏图（图的边的数目远小于顶点的数目），那么会很浪费空间。并且如果图是无向图，始终至少会有 50 % 的空间浪费。下面的图也直观地反应了这一点。

邻接矩阵的优点主要有：

1. 直观，简单。
2. 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 $O(1)$

由于使用起来比较简单，因此我的所有的需要建图的题目基本都用这种方式。

比如力扣 743. 网络延迟时间。题目描述：

有 N 个网络节点，标记为 1 到 N。

给定一个列表 times，表示信号经过有向边的传递时间。 $times[i] = (u,$

现在，我们从某个节点 K 发出一个信号。需要多久才能使所有节点都收到信号？

示例：

输入: times = [[2,1,1],[2,3,1],[3,4,1]], N = 4, K = 2

输出: 2

注意：

N 的范围在 [1, 100] 之间。

K 的范围在 [1, N] 之间。

times 的长度在 [1, 6000] 之间。

所有的边 $times[i] = (u, v, w)$ 都有 $1 \leq u, v \leq N$ 且 $0 \leq w <$

这是一个典型的图的题目，对于这道题，我们如何用邻接矩阵建图呢？

一个典型的建图代码：

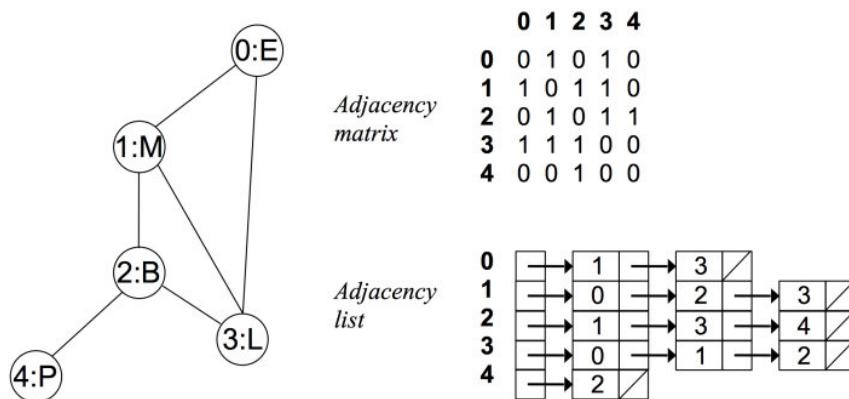
```
graph = collections.defaultdict(list)
for fr, to, w in times:
    graph[fr - 1].append((to - 1, w))
```

这就构造了一个邻接矩阵，之后我们基于这个邻接矩阵遍历图即可。

邻接表 [Adjacency List]

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。

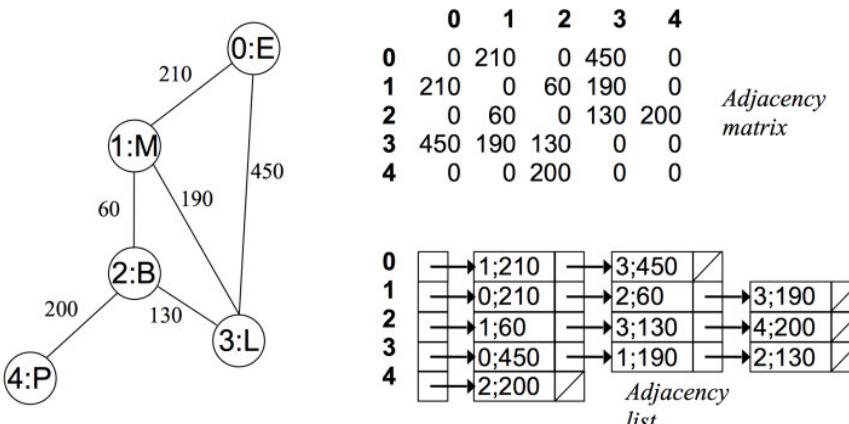
例如在无向无权图中：



(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>)

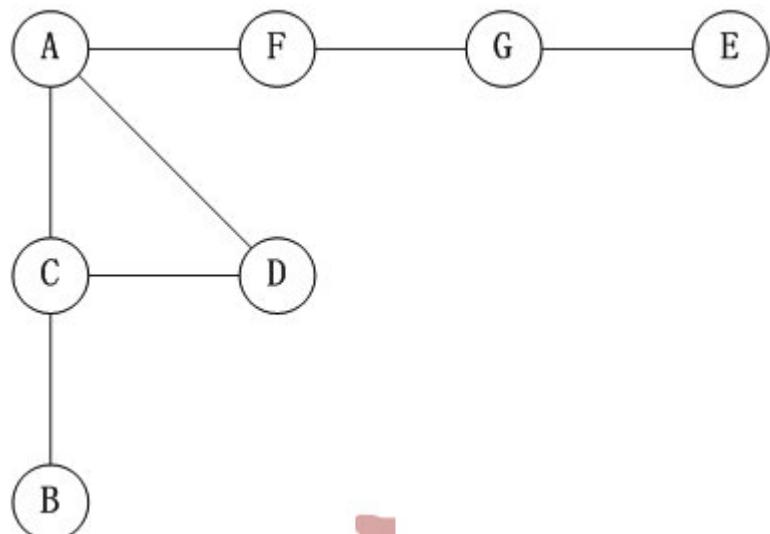
图的遍历

图建立好了，接下来就是要遍历。不管你是什么算法，肯定都要遍历的，一般有以下两种方法（其他奇葩的遍历方式实际意义不大，没有必要学习）。不管是哪一种遍历，如果图有环，就一定要记录节点的访问情况，防止死循环。当然你可能不需要真正地使用一个集合记录节点的访问情况，比如使用一个数据范围外的数据原地标记，这样的空间复杂度会是 $O(1)$ 。

这里以有向图为例，有向图也是类似，这里不再赘述。

深度优先遍历 (Depth First Search, DFS)

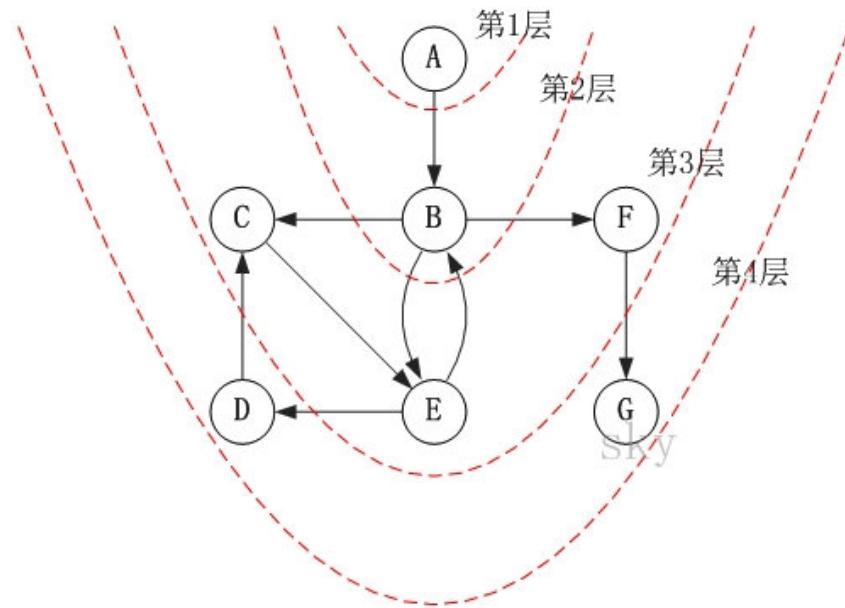
深度优先遍历图的方法是，从图中某顶点 v 出发，不断访问邻居，邻居的邻居直到访问完毕。



如上图，如果我们使用 DFS，并且从 A 节点开始的话，一个可能的访问顺序是： $A \rightarrow C \rightarrow B \rightarrow D \rightarrow F \rightarrow G \rightarrow E$ ，当然也可能是 $A \rightarrow D \rightarrow C \rightarrow B \rightarrow F \rightarrow G \rightarrow E$ 等，具体取决于你的代码，但他们都是深度优先的。

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。



如上图，如果我们使用 BFS，并且从 A 节点开始的话，一个可能的访问顺序是：A → B → C → F → E → G → D，当然也可能是 A → B → F → E → C → G → D 等，具体取决于你的代码，但他们都是广度优先的。

需要注意的是 DFS 和 BFS 只是一种算法思想，不是一种具体的算法。因此其有着很强的适应性，而不是局限于特点的数据结构的，本文讲的图可以用，前面讲的树也可以用。实际上，只要是**非线性的数据结构都可以用**。

常见算法

图的题目的算法比较适合套模板。题目类型主要有：

- Dijkstra
- Floyd-Warshall
- 最小生成树 (Kruskal & Prim)
- A 星寻路算法
- 二分图 (染色法) (Bipartite)
- 拓扑排序 (Topological Sort)

下面列举常见算法的模板，以下所有的模板都是基于邻接矩阵。

最短距离，最短路径

Dijkstra 算法

DIJKSTRA 算法主要解决的是图中任意一点的图中某一个点的最短距离，即单源最短路径。

989. 数组形式的整数加法

Dijkstra 这个名字比较难记，大家可以简单记为**DJ 算法**，有没有好记很多？

比如给你几个城市，以及城市之间的距离。让你规划一条最短的从城市 a 到城市 b 的路线。这个问题，我们就可以使用 dijkstra 来做。

算法的基本思想是贪心，每次都遍历所有邻居，并从中找到距离最小的，本质上是一种广度优先遍历。这里我们借助堆这种数据结构，使得可以在 $\log N$ 的时间内找到 cost 最小的点。

代码模板：

Python

```
import heapq

def dijkstra(graph, start, end):
    # 堆里的数据都是 (cost, i) 的二元组，其含义是“从 start 走到 i
    heap = [(0, start)]
    visited = set()
    while heap:
        (cost, u) = heapq.heappop(heap)
        if u in visited:
            continue
        visited.add(u)
        if u == end:
            return cost
        for v, c in graph[u]:
            if v in visited:
                continue
            next = cost + c
            heapq.heappush(heap, (next, v))
    return -1
```

JavaScript

989. 数组形式的整数加法

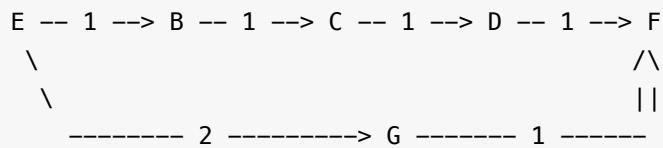
```
const dijkstra = (graph, start, end) => {
  const visited = new Set()
  const minHeap = new MinPriorityQueue();
  //注: 此处new MinPriorityQueue()用了LC的内置API, 它的enqueue方法接受一个对象, 包含
  //element 和 priority。
  //堆会按照priority排序, 可以用element记录一些内容。
  minHeap.enqueue(startPoint, 0)

  while(!minHeap.isEmpty()){
    const {element, priority} = minHeap.dequeue();
    //下面这两个变量不是必须的, 只是便于理解
    const curPoint = element;
    const curCost = priority;

    if(curPoint === end) return curCost;
    if(visited.has(curPoint)) continue;
    visited.add(curPoint);

    if(!graph[curPoint]) continue;
    for(const [nextPoint, nextCost] of graph[curPoint]){
      if(visited.has(nextPoint)) continue;
      //注意heap里面的一定是从startPoint到某个点的距离;
      //curPoint到nextPoint的距离是nextCost; 但curPoint不一定是最短距离
      const accumulatedCost = nextCost + curCost;
      minHeap.enqueue(nextPoint, accumulatedCost);
    }
  }
  return -1
}
```

比如一个图是这样的:



我们使用邻接矩阵来构造:

989. 数组形式的整数加法

```
G = {
    "B": [["C", 1]],
    "C": [["D", 1]],
    "D": [["F", 1]],
    "E": [[["B", 1], ["G", 2]]],
    "F": [],
    "G": [[["F", 1]]],
}

shortDistance = dijkstra(G, "E", "C")
print(shortDistance) # E -- 3 --> F -- 3 --> C == 6
```

学会了这个算法模板，你就可以去 AC 743. 网络延迟时间 了。

完整代码：

Python

```
class Solution:
    def dijkstra(self, graph, start, end):
        heap = [(0, start)]
        visited = set()
        while heap:
            (cost, u) = heapq.heappop(heap)
            if u in visited:
                continue
            visited.add(u)
            if u == end:
                return cost
            for v, c in graph[u]:
                if v in visited:
                    continue
                next = cost + c
                heapq.heappush(heap, (next, v))
        return -1
    def networkDelayTime(self, times: List[List[int]], N: int):
        graph = collections.defaultdict(list)
        for fr, to, w in times:
            graph[fr - 1].append((to - 1, w))
        ans = -1
        for to in range(N):
            dist = self.dijkstra(graph, 0, to)
            if dist == -1: return -1
            ans = max(ans, dist)
        return ans
```

JavaScript

989. 数组形式的整数加法

```
const networkDelayTime = (times, n, k) => {
    //咳咳这个解法并不是Dijkstra在本题的最佳解法
    const graph = {};
    for(const [from, to, weight] of times){
        if(!graph[from]) graph[from] = [];
        graph[from].push([to, weight]);
    }

    let ans = -1;
    for(let to = 1; to <= n; to++){
        let dist = dijkstra(graph, k, to)
        if(dist === -1) return -1;
        ans = Math.max(ans, dist);
    }
    return ans;
};

const dijkstra = (graph, startPoint, endPoint) => {
    const visited = new Set()
    const minHeap = new MinPriorityQueue();
    //注: 此处new MinPriorityQueue()用了LC的内置API, 它的enqueue方法接受
    //element 和 priority。
    //堆会按照priority排序, 可以用element记录一些内容。
    minHeap.enqueue(startPoint, 0)

    while(!minHeap.isEmpty()){
        const {element, priority} = minHeap.dequeue();
        //下面这两个变量不是必须的, 只是便于理解
        const curPoint = element;
        const curCost = priority;
        if(visited.has(curPoint)) continue;
        visited.add(curPoint)
        if(curPoint === endPoint) return curCost;

        if(!graph[curPoint]) continue;
        for(const [nextPoint, nextCost] of graph[curPoint]){
            if(visited.has(nextPoint)) continue;
            //注意heap里面的一定是从startPoint到某个点的距离;
            //curPoint到nextPoint的距离是nextCost; 但curPoint不一定是
            const accumulatedCost = nextCost + curCost;
            minHeap.enqueue(nextPoint, accumulatedCost);
        }
    }
    return -1
}
```

你学会了么?

DJ 算法的时间复杂度为 $v \log v + e$ ，其中 v 和 e 分别为图中的点和边的个数。

最后给大家留一个思考题：如果是计算一个点到图中所有点的距离呢？我们的算法会有什么样的调整？

提示：你可以使用一个 dist 哈希表记录开始点到每个点的最短距离来完成。想出来的话，可以用力扣 882 题去验证一下哦~

Floyd-Warshall 算法

Floyd-Warshall 也可以解决任意两个点距离，即多源最短路径。

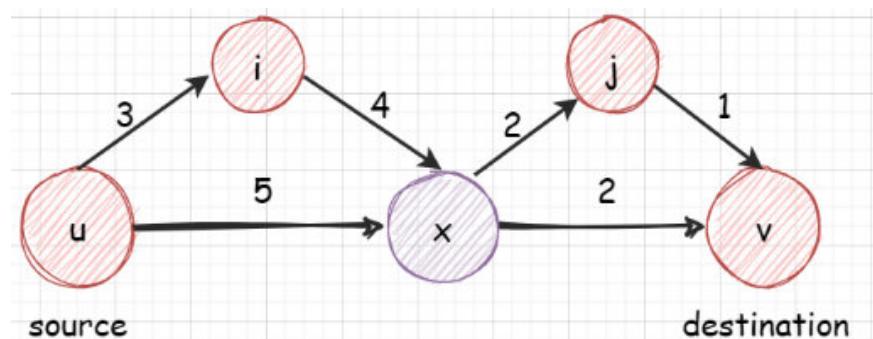
除此之外，贝尔曼-福特算法也是解决最短路径的经典动态规划算法。相比上面的 dijkstra 算法，由于其计算过程会把中间运算结果保存起来防止重复计算，因此其特别适合求图中任意两点的距离，比如力扣的 1462. 课程安排 IV。除了这个优点。贝尔曼-福特算法相比于此算法最大的区别在于本算法是多源最短路径，而贝尔曼-福特则是单源最短路径。不管是复杂度和写法，贝尔曼-福特算法都更简单，我们后面给大家介绍。

当然就不是说贝尔曼算法以及上面的 dijkstra 就不支持多源最短路径，你只需要加一个 for 循环枚举所有的起点罢了。

还有一个非常重要的点是 Floyd-Warshall 算法由于使用了动态规划的思想而不是贪心，因此其可以处理负权重的情况。动态规划的详细内容请参考之后的动态规划专题和背包问题。

Floyd-Warshall 的基本思想是动态规划。该算法的时间复杂度是 $O(N^3)$ ，空间复杂度是 $O(N^2)$ ，其中 N 为顶点个数。

算法也不难理解，简单来说就是： i 到 j 的最短路径 = i 到 k 的最短路径 + k 到 j 的最短路径的最小值。如下图：



u 到 v 的最短距离是 u 到 x 的最短距离 + x 到 v 的最短距离。上图 x 是 u 到 v 的必经之路，如果不是的话，我们需要多个中间节点的值，并取最小的。

算法的正确性不言而喻，因为从 i 到 j ，要么直接到，要么经过图中的另外另一个点 k ，中间节点 k 可能有多个，经过中间点的情况取出最小的，自然就是 i 到 j 的最短距离。

989. 数组形式的整数加法

思考题：最长无环路径可以用动态规划来解么？

代码模板：

Python

```
# graph 是邻接矩阵, n 是顶点个数
# graph 形如: graph[u][v] = w

def floyd_marshall(graph, n):
    dist = [[float("inf") for _ in range(n)] for _ in range(n)]

    for i in range(n):
        for j in range(n):
            dist[i][j] = graph[i][j]

    # check vertex k against all other vertices (i, j)
    for k in range(n):
        # looping through rows of graph array
        for i in range(n):
            # looping through columns of graph array
            for j in range(n):
                if (
                    dist[i][k] != float("inf")
                    and dist[k][j] != float("inf")
                    and dist[i][k] + dist[k][j] < dist[i][j]
                ):
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist
```

JavaScript

989. 数组形式的整数加法

```
const floydWarshall = (graph, v)=>{
    const dist = new Array(v).fill(0).map(() => new Array(v).

        for(let i = 0; i < v; i++){
            for(let j = 0; j < v; j++){
                //两个点相同, 距离为0
                if(i === j) dist[i][j] = 0;
                //i 和 j 的距离已知
                else if(graph[i][j]) dist[i][j] = graph[i][j];
                //i 和 j 的距离未知, 默认是最大值
                else dist[i][j] = Number.MAX_SAFE_INTEGER;
            }
        }

        //检查是否有一个点 k 使得 i 和 j 之间距离更短, 如果有, 则更新最短距离
        for(let k = 0; k < v; k++){
            for(let i = 0; i < v; i++){
                for(let j = 0; j < v; j++){
                    dist[i][j] = Math.min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }
    }
    return 看需要
}
```

我们回过头来看下如何套模板解决 力扣的 1462. 课程安排 IV，题目描述：

989. 数组形式的整数加法

你总共需要上 n 门课，课程编号依次为 0 到 $n-1$ 。

有的课会有直接的先修课程，比如如果想上课程 0 ，你必须先上课程 1 ，那么课程 1 就是课程 0 的先修课程。

给你课程总数 n 和一个直接先修课程数对列表 `prerequisite` 和一个查询对列表 `queries`。对于每个查询对 `queries[i]`，请判断 `queries[i][0]` 是否是 `queries[i][1]` 的先修课程。

请返回一个布尔值列表，列表中每个元素依次分别对应 `queries` 每个查询对的结果。

注意：如果课程 a 是课程 b 的先修课程且课程 b 是课程 c 的先修课程，那么课程 a 也是课程 c 的先修课程。

示例 1：

输入: $n = 2$, `prerequisites` = $\[[[1,0]]\}$, `queries` = $\[[[0,1],[1,0]]\}$
输出: [false,true]

解释: 课程 0 不是课程 1 的先修课程，但课程 1 是课程 0 的先修课程。

示例 2：

输入: $n = 2$, `prerequisites` = $\[]$, `queries` = $\[[[1,0],[0,1]]\}$
输出: [false,false]

解释: 没有先修课程对，所以每门课程之间是独立的。

示例 3：

输入: $n = 3$, `prerequisites` = $\[[[1,2],[1,0],[2,0]]\}$, `queries` = $\[[[0,1],[1,2]]\}$
输出: [true,true]

示例 4：

输入: $n = 3$, `prerequisites` = $\[[[1,0],[2,0]]\}$, `queries` = $\[[[0,1],[1,2]]\}$
输出: [false,true]

示例 5：

输入: $n = 5$, `prerequisites` = $\[[[0,1],[1,2],[2,3],[3,4]]\}$, `queries` = $\[[[0,1],[1,2],[2,3],[3,4]]\}$
输出: [true,false,true,false]

提示：

```
2 <= n <= 100
0 <= prerequisite.length <= (n * (n - 1) / 2)
0 <= prerequisite[i][0], prerequisite[i][1] < n
prerequisite[i][0] != prerequisite[i][1]
```

989. 数组形式的整数加法

先修课程图中没有环。

先修课程图中没有重复的边。

```
1 <= queries.length <= 10^4
queries[i][0] != queries[i][1]
```

这道题也可以使用 Floyd-Warshall 来做。你可以这么想，如果从 i 到 j 的距离大于 0，那不就是先修课么。而这道题数据范围 $queries$ 大概是 10^4 ，用上面的 dijkstra 算法肯定超时，因此 Floyd-Warshall 算法是明智的选择。

我这里直接套模板，稍微改下就过了。完整代码：Python

```
class Solution:
    def Floyd-Warshall(self, dist, v):
        for k in range(v):
            for i in range(v):
                for j in range(v):
                    dist[i][j] = dist[i][j] or (dist[i][k]

        return dist

    def checkIfPrerequisite(self, n: int, prerequisites: List[List[int]]):
        graph = [[False] * n for _ in range(n)]
        ans = []

        for to, fr in prerequisites:
            graph[fr][to] = True
        dist = self.Floyd-Warshall(graph, n)
        for to, fr in queries:
            ans.append(bool(dist[fr][to]))
        return ans
```

JavaScript

```
//咳咳这个写法不是本题最优
var checkIfPrerequisite = function(numCourses, prerequisites) {
    const graph = {}
    for(const [course, pre] of prerequisites){
        if(!graph[pre]) graph[pre] = {}
        graph[pre][course] = true
    }

    const ans = []

    const dist = Floyd-Warshall(graph, numCourses)
    for(const [course, pre] of queries){
        ans.push(dist[pre][course])
    }

    return ans
};

var Floyd-Warshall = function(graph, n){
    dist = Array.from({length: n + 1}).map(() => Array.from({length: n + 1}).fill(Infinity))
    for(let k = 0; k < n; k++){
        for(let i = 0; i < n; i++){
            for(let j = 0; j < n; j++){
                if(graph[i] && graph[i][j]) dist[i][j] = true
                if(graph[i] && graph[k]){
                    dist[i][j] = (dist[i][j]) || (dist[i][k] && graph[k][j])
                }else if(graph[i]){
                    dist[i][j] = dist[i][j]
                }
            }
        }
    }
    return dist
}
}
```

如果这道题你可以解决了，我再推荐一道题给你 [1617. 统计子树中城市之间最大距离](#)，国际版有一个题解代码挺清晰，挺好理解的，只不过没有使用状态压缩性能不是很好罢了，地址：

<https://leetcode.com/problems/count-subtrees-with-max-distance-between-cities/discuss/1136596/Python-Floyd-Warshall-and-check-all-subtrees>

贝尔曼-福特算法

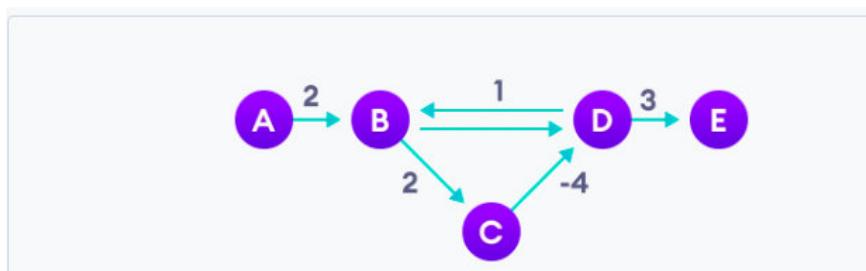
和上面的算法类似。这种解法主要解决单源最短路径，即图中某一点到其他点的最短距离。

其基本思想也是动态规划。

核心算法为：

- 初始化起点距离为 0
- 对图中的所有边进行若干次处理，直到稳定。处理的依据是：对于每一个有向边 (u,v) ，如果 $dist[u] + w < dist[v]$ ，那么意味着我们找到了一条到达 v 更近的路，更新之。
- 上面的若干次的上限是顶点 V 的个数，因此不妨直接进行 n 次处理。
- 最后检查一下是否存在负边引起的环。（注意）

举个例子。对于如下的一个图，存在一个 $B \rightarrow C \rightarrow D \rightarrow B$ ，这样 B 到 C 和 D 的距离理论上可以无限小。我们需要检测到这一种情况，并退出。



此算法时间复杂度： $O(V^*E)$ ， 空间复杂度： $O(V)$ 。

代码示例：

Python

```

# return -1 for not exsit
# else return dis map where dis[v] means for point s the length of shortest path
def bell_mann(edges, s):
    dis = defaultdict(lambda: math.inf)
    dis[s] = 0
    for _ in range(n):
        for u, v, w in edges:
            if dis[u] + w < dis[v]:
                dis[v] = dis[u] + w

    for u, v, w in edges:
        if dis[u] + w < dis[v]:
            return -1

    return dis
  
```

JavaScript

```

const BellmanFord = (edges, startPoint) => {
  const n = edges.length;
  const dist = new Array(n).fill(Number.MAX_SAFE_INTEGER);
  dist[startPoint] = 0;

  for(let i = 0; i < n; i++) {
    for(const [u, v, w] of edges) {
      if(dist[u] + w < dist[v]) {
        dist[v] = dist[u] + w;
      }
    }
  }

  for(const [u, v, w] of edges) {
    if(dist[u] + w < dist[v]) return -1;
  }

  return dist
}

```

推荐阅读:

- [bellman-ford-algorithm](#)

题目推荐:

- [Best Currency Path](#)

拓扑排序

在计算机科学领域，有向图的拓扑排序是对其顶点的一种线性排序，使得对于从顶点 u 到顶点 v 的每个有向边 uv ， u 在排序中都在之前。当且仅当图中没有定向环时（即有向无环图），才有可能进行拓扑排序。

典型的题目就是给你一堆课程，课程之间有先修关系，让你给出一种可行的学习路径方式，要求先修的课程要先学。任何有向无环图至少有一个拓扑排序。已知有算法可以在线性时间内，构建任何有向无环图的拓扑排序。

Kahn 算法

简单来说，假设 L 是存放结果的列表，先找到那些入度为零的节点，把这些节点放到 L 中，因为这些节点没有任何的父节点。然后把与这些节点相连的边从图中去掉，再寻找图中的入度为零的节点。对于新找到的这些入度为零的节点来说，他们的父节点已经都在 L 中了，所以也可以放入 L 。

重复上述操作，直到找不到入度为零的节点。如果此时 L 中的元素个数和节点总数相同，说明排序完成；如果 L 中的元素个数和节点总数不同，说明原图中存在环，无法进行拓扑排序。

```

def topologicalSort(graph):
    """
    Kahn's Algorithm is used to find Topological ordering of
    nodes using BFS
    """

    indegree = [0] * len(graph)
    queue = collections.deque()
    topo = []
    cnt = 0

    for key, values in graph.items():
        for i in values:
            indegree[i] += 1

    for i in range(len(indegree)):
        if indegree[i] == 0:
            queue.append(i)

    while queue:
        vertex = queue.popleft()
        cnt += 1
        topo.append(vertex)
        for x in graph[vertex]:
            indegree[x] -= 1
            if indegree[x] == 0:
                queue.append(x)

    if cnt != len(graph):
        print("Cycle exists")
    else:
        print(topo)

# Adjacency List of Graph
graph = {0: [1, 2], 1: [3], 2: [3], 3: [4, 5], 4: [], 5: []}
topologicalSort(graph)

```

最小生成树

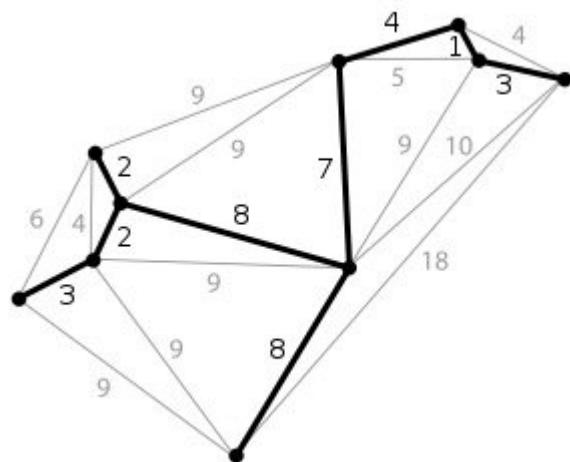
Kruskal 和 Prim 是两个经典的求最小生成树的算法，本节我们就来了解一下它们。

什么是最小生成树，这两个算法又是如何计算最小生成树的呢？

首先我们来看下什么是生成树。生成树是一个图的一部分，生成树包含图的**所有顶点**，且不包含环，这也是为什么叫做生成树，而不是生成图的原因。你可以将生成树看成是根节点不确定的多叉树。

最小生成树是在生成树的基础上加了**最小关键字**，是最小权重生成树的简称。其指的是对于带权图来说，生成树的权重是其所有边的权重和，那么最小生成树就是权重和最小的生成树，由此可看出，不管是生成树还是最小生成树都可能不唯一。

这在实际生活中有很强的价值。比如我要修建一个地铁，并覆盖 n 个站，如果建造才能使得花费最小？由于每个站之间的路线不同，因此造价也不一样，因此这就是一个最小生成树的实际使用场景，类似的例子还有很多。



(图来自维基百科)

Kruskal

Kruskal 算法也被形象地称为**加边法**，每前进一次都选择权重最小的边，加入到结果集。为了防止环的产生（增加环是无意义的，只要权重是正数，一定会使结果更差），我们需要检查下当前选择的边是否和已经选择的边联通了。如果联通了，是没有必要选取的，因为这会使得环产生。因此算法上，我们可使用并查集辅助完成。下面算法中的 `find_parent` 部分，实际上就是并查集的核心代码，只是我们没有将其封装并使用罢了。

Kruskal 具体算法：

1. 对边进行排序
2. 将 n 个顶点初始化为 n 个联通域
3. 按照权值从小到大选择边加入到结果集，如果当前选择的边是否和已经选择的边联通了，则放弃选择，否则进行选择，加入到结果集。
4. 重复 3 直到我们找到了一个联通域大小为 n 的子图

代码模板：

989. 数组形式的整数加法

```
from typing import List, Tuple

def kruskal(num_nodes: int, edges: List[Tuple[int, int, int]]):
    """
    >>> kruskal(4, 3, [(0, 1, 3), (1, 2, 5), (2, 3, 1)])
    [(2, 3, 1), (0, 1, 3), (1, 2, 5)]

    >>> kruskal(4, 5, [(0, 1, 3), (1, 2, 5), (2, 3, 1), (0, 2, 1),
    [(2, 3, 1), (0, 2, 1), (0, 1, 3)

    >>> kruskal(4, 6, [(0, 1, 3), (1, 2, 5), (2, 3, 1), (0, 1, 1),
    ... (2, 1, 1)])
    [(2, 3, 1), (0, 2, 1), (2, 1, 1)]
    """
    edges = sorted(edges, key=lambda edge: edge[2])

    parent = list(range(num_nodes))

    def find_parent(i):
        if i != parent[i]:
            parent[i] = find_parent(parent[i])
        return parent[i]

    minimum_spanning_tree_cost = 0
    minimum_spanning_tree = []

    for edge in edges:
        parent_a = find_parent(edge[0])
        parent_b = find_parent(edge[1])
        if parent_a != parent_b:
            minimum_spanning_tree_cost += edge[2]
            minimum_spanning_tree.append(edge)
            parent[parent_a] = parent_b

    return minimum_spanning_tree

if __name__ == "__main__": # pragma: no cover
    num_nodes, num_edges = list(map(int, input().strip().split()))
    edges = []

    for _ in range(num_edges):
        node1, node2, cost = [int(x) for x in input().strip().split()]
        edges.append((node1, node2, cost))

    kruskal(num_nodes, edges)
```

Prim

Prim 算法也被形象地称为**加点法**，每前进一次都选择权重最小的点，加入到结果集。形象地看就像一个不断生长的真实世界的树。

Prim 具体算法：

1. 初始化最小生成树点集 MV 为图中任意一个顶点，最小生成树边集 ME 为空。我们的目标是将 MV 填充到和 V 一样，而边集则根据 MV 的产生自动计算。
2. 在集合 E 中（集合 E 为原始图的边集）选取最小的边 其中 u 为 MV 中已有的元素，而 v 为 MV 中不存在的元素（像不像上面说的**不断生长的真实世界的树**），将 v 加入到 MV ，将 加到 ME 。
3. 重复 2 直到我们找到了一个联通域大小为 n 的子图

算法模板：

为了体现完整性，代码中关于堆的部分采用了手动实现的方式。

989. 数组形式的整数加法

```
import sys
from collections import defaultdict

def PrimsAlgorithm(l): # noqa: E741

    nodePosition = []

    def get_position(vertex):
        return nodePosition[vertex]

    def set_position(vertex, pos):
        nodePosition[vertex] = pos

    def top_to_bottom(heap, start, size, positions):
        if start > size // 2 - 1:
            return
        else:
            if 2 * start + 2 >= size:
                m = 2 * start + 1
            else:
                if heap[2 * start + 1] < heap[2 * start + 2]:
                    m = 2 * start + 1
                else:
                    m = 2 * start + 2
            if heap[m] < heap[start]:
                temp, temp1 = heap[m], positions[m]
                heap[m], positions[m] = heap[start], positions[start]
                heap[start], positions[start] = temp, temp1

                temp = get_position(positions[m])
                set_position(positions[m], get_position(positions[m]))
                set_position(positions[start], temp)

            top_to_bottom(heap, m, size, positions)

    # Update function if value of any node in min-heap decreases
    def bottom_to_top(val, index, heap, position):
        temp = position[index]

        while index != 0:
            if index % 2 == 0:
                parent = int((index - 2) / 2)
            else:
                parent = int((index - 1) / 2)

            if val < heap[parent]:
                heap[index] = heap[parent]
```

989. 数组形式的整数加法

```
position[index] = position[parent]
set_position(position[parent], index)
else:
    heap[index] = val
    position[index] = temp
    set_position(temp, index)
    break
index = parent
else:
    heap[0] = val
    position[0] = temp
    set_position(temp, 0)

def heapify(heap, positions):
    start = len(heap) // 2 - 1
    for i in range(start, -1, -1):
        top_to_bottom(heap, i, len(heap), positions)

def deleteMinimum(heap, positions):
    temp = positions[0]
    heap[0] = sys.maxsize
    top_to_bottom(heap, 0, len(heap), positions)
    return temp

visited = [0 for i in range(len(l))]
Nbr_TV = [-1 for i in range(len(l))] # Neighboring Tree
# Minimum Distance of explored vertex with neighboring
# formed in graph
Distance_TV = [] # Heap of Distance of vertices from tree
Positions = []

for x in range(len(l)):
    p = sys.maxsize
    Distance_TV.append(p)
    Positions.append(x)
    nodePosition.append(x)

TreeEdges = []
visited[0] = 1
Distance_TV[0] = sys.maxsize
for x in l[0]:
    Nbr_TV[x[0]] = 0
    Distance_TV[x[0]] = x[1]
heapify(Distance_TV, Positions)

for i in range(1, len(l)):
    vertex = deleteMinimum(Distance_TV, Positions)
    if visited[vertex] == 0:
```

```

        TreeEdges.append((Nbr_TV[vertex], vertex))
        visited[vertex] = 1
        for v in l[vertex]:
            if visited[v[0]] == 0 and v[1] < Distance_TV[v[0]]:
                Distance_TV[get_position(v[0])] = v[1]
                bottom_to_top(v[1], get_position(v[0]),
                Nbr_TV[v[0]]) = vertex
    return TreeEdges

if __name__ == "__main__": # pragma: no cover
    # < ----- Prims Algorithm ----- >
    n = int(input("Enter number of vertices: ").strip())
    e = int(input("Enter number of edges: ").strip())
    adjlist = defaultdict(list)
    for x in range(e):
        l = [int(x) for x in input().strip().split()] # no
        adjlist[l[0]].append([l[1], l[2]])
        adjlist[l[1]].append([l[0], l[2]])
    print(PrimsAlgorithm(adjlist))

```

两种算法比较

为了后面描述方便，我们令 V 为图中的顶点数， E 为图中的边数。那么 KruKal 的算法复杂度是 $O(E \log E)$ ，Prim 的算法时间复杂度为 $E + V \log V$ 。

KruKal 是基于图的联通性贪心算法。而 Prim 则是基于堆的贪心算法。

A 星寻路算法

A 星寻路解决的问题是在一个二维的表格中找出任意两点的最短距离或者最短路径。常用于游戏中的 NPC 的移动计算，是一种常用启发式算法。一般这种题目都会有障碍物。除了障碍物，力扣的题目还会增加一些限制，使得题目难度增加。

这种题目一般都是力扣的困难难度。理解起来不难，但是要完整地没有 bug 地写出来却不容易。

在该算法中，我们从起点开始，检查其相邻的四个方格并尝试扩展，直至找到目标。A 星寻路算法的寻路方式不止一种，感兴趣的可以自行了解一下。

公式表示为： $f(n) = g(n) + h(n)$ 。

其中：

- $f(n)$ 是从初始状态经由状态 n 到目标状态的估计代价，

989. 数组形式的整数加法

- $g(n)$ 是在状态空间中从初始状态到状态 n 的实际代价,
- $h(n)$ 是从状态 n 到目标状态的最佳路径的估计代价。

如果 $g(n)$ 为 0, 即只计算任意顶点 n 到目标的评估函数 $h(n)$, 而不计算起点到顶点 n 的距离, 则算法转化为使用贪心策略的最良优先搜索, 速度最快, 但可能得不出最优解; 如果 $h(n)$ 不大于顶点 n 到目标顶点的实际距离, 则一定可以求出最优解, 而且 $h(n)$ 越小, 需要计算的节点越多, 算法效率越低, 常见的评估函数有——欧几里得距离、曼哈顿距离、切比雪夫距离; 如果 $h(n)$ 为 0, 即只需求出起点到任意顶点 n 的最短路径 $g(n)$, 而不计算任何评估函数 $h(n)$, 则转化为单源最短路径问题, 即 Dijkstra 算法, 此时需要计算最多的顶点;

这里有一个重要的概念是估价算法, 一般我们使用 曼哈顿距离来进行估价, 即 $H(n) = D * (\text{abs}(n.x - \text{goal}.x) + \text{abs}(n.y - \text{goal}.y))$ 。



(图来自维基百科

https://zh.wikipedia.org/wiki/A*_%E6%90%9C%E5%B0%8B%E6%BC%94%E7%AE%97%E6%B3%95)

一个完整的代码模板:

989. 数组形式的整数加法

```
grid = [
    [0, 1, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0], # 0 are free path whereas 1's are
    [0, 1, 0, 0, 0, 0],
    [0, 1, 0, 0, 1, 0],
    [0, 0, 0, 0, 1, 0],
]

.....
heuristic = [[9, 8, 7, 6, 5, 4],
             [8, 7, 6, 5, 4, 3],
             [7, 6, 5, 4, 3, 2],
             [6, 5, 4, 3, 2, 1],
             [5, 4, 3, 2, 1, 0]]"""

init = [0, 0]
goal = [len(grid) - 1, len(grid[0]) - 1] # all coordinates
cost = 1

# the cost map which pushes the path closer to the goal
heuristic = [[0 for row in range(len(grid[0]))] for col in
for i in range(len(grid)):
    for j in range(len(grid[0])):
        heuristic[i][j] = abs(i - goal[0]) + abs(j - goal[1])
        if grid[i][j] == 1:
            heuristic[i][j] = 99 # added extra penalty in

# the actions we can take
delta = [[-1, 0], [0, -1], [1, 0], [0, 1]] # go up # go right

# function to search the path
def search(grid, init, goal, cost, heuristic):

    closed = [
        [0 for col in range(len(grid[0]))] for row in range(len(grid))
    ] # the reference grid
    closed[init[0]][init[1]] = 1
    action = [
        [0 for col in range(len(grid[0]))] for row in range(len(grid))
    ] # the action grid

    x = init[0]
    y = init[1]
    g = 0
    f = g + heuristic[init[0]][init[1]]
    cell = [[f, g, x, y]]
```

```

found = False # flag that is set when search is complete
resign = False # flag set if we can't find expand

while not found and not resign:
    if len(cell) == 0:
        return "FAIL"
    else: # to choose the least costliest action so as to
        cell.sort()
        cell.reverse()
    next = cell.pop()
    x = next[2]
    y = next[3]
    g = next[1]

    if x == goal[0] and y == goal[1]:
        found = True
    else:
        for i in range(len(delta)): # to try out all possible moves
            x2 = x + delta[i][0]
            y2 = y + delta[i][1]
            if x2 >= 0 and x2 < len(grid) and y2 >= 0 and y2 < len(grid[0]):
                if closed[x2][y2] == 0 and grid[x2][y2] == 0:
                    g2 = g + cost
                    f2 = g2 + heuristic[x2][y2]
                    cell.append([f2, g2, x2, y2])
                    closed[x2][y2] = 1
                    action[x2][y2] = i

        invpath = []
        x = goal[0]
        y = goal[1]
        invpath.append([x, y]) # we get the reverse path from here
        while x != init[0] or y != init[1]:
            x2 = x - delta[action[x][y]][0]
            y2 = y - delta[action[x][y]][1]
            x = x2
            y = y2
            invpath.append([x, y])

        path = []
        for i in range(len(invpath)):
            path.append(invpath[len(invpath) - 1 - i])
        print("ACTION MAP")
        for i in range(len(action)):
            print(action[i])

return path

```

```
a = search(grid, init, goal, cost, heuristic)
for i in range(len(a)):
    print(a[i])
```

典型题目[1263. 推箱子](#)

二分图

二分图我在这两道题中讲过了，大家看一下之后把这两道题做一下就行了。其实这两道题和一道题没啥区别。

- [0886. 可能的二分法](#)
- [0785. 判断二分图](#)

推荐顺序为：先看 886 再看 785。

参考

- [维基百科 - 最小生成树](#)

总结

理解图的常见概念，我们就算入门了。接下来，我们就可以做题了，一般的图题目第一步都是建图，第二步都是基于第一步的图进行遍历以寻找可行解。

图的题目相对而言比较难，尤其是代码书写层面。但是就面试题目而言，图的题目类型却不多，而且很多题目都是套模板就可以解决。因此建议大家多练习模板，并自己多手敲，确保可以自己敲出来。

模拟，枚举与递推

本篇讲义隶属于基础篇。基础不意味着简单，有时候一道困难的题目也背后的算法可能是基础算法。因此我们需要做的是熟悉各种基础数据结构和算法，做到融会贯通，举一反三，这样面对复杂问题才能迎刃有余。

这里我们讲三种基础思想和方法，这几种方法将贯穿整个算法系列的始终，它们分别是模拟，枚举与递推。

模拟并不是一种算法，而是一种思维方式。枚举则是一个最基础的技能，但是枚举却不一定和你想象的一样简单。递推则是一种很重要的编程思想，是递归，动态规划等高级主题的基础。

模拟

简单来说，我们这里的模拟指的就是题目让你做什么，你就做什么。即将题目描述翻译为可执行的代码。这在我们做工程的时候，将产品经理的需求转化为代码是类似的。

模拟并不是一类具体的算法，而是一种思想。

模拟指的是将题目描述转化为可执行的代码，其中我们会用到编程语言的基础内容，最常见的就是循环。

简单的题目，通常直接模拟就够了，比如 [874. 模拟行走机器人](#)。

而如果是中等和困难的题目，除了使用模拟，我们还需要使用一些别的技巧。比如有的题目就是模拟 + 堆，这是因为模拟的过程我们需要动态获取极值。再比如 [799. 香槟塔](#)，其实就是模拟 + 动态规划。可以看出，中等的题目通常模拟只是辅助，还需要结合其他知识。

那么是否所有的题目都是模拟？当然不是。比如动态规划的题目，这就不能说是模拟。因为题目并没有告诉你明确的操作步骤，而是需要你根据题目信息，自己挖掘转移方程进行求解。

枚举

枚举简单来说就是指尝试所有可能，是一种最基本的算法。

最常见的枚举就是暴力搜索。即在解空间中暴力枚举所有解空间中的值，并逐个判断其是否是答案。

枚举可以很简单。比如枚举一个数组的所有项。

```

for(int i = 0; i < A.length(); i++) {
    // 打印 A[i]
}

```

实际上枚举也可以很复杂。比如：

- 维度的上升（枚举 3 维数组）
- 方向的选择（从前往后还是从后往前）等。

而且同样是枚举，不同的枚举策略也可能有不同的结果和效率。关于这点，我们将在后面的剪枝专题给大家做更多介绍。

枚举三要素

枚举有三个东西需要考虑。

1. 状态。都有哪些状态需要我们枚举？
2. 不重不漏。如何枚举才不会重复，且不会漏过正确解？
3. 效率。采取怎么样的枚举策略可以最大化提供算法效果？

接下来，我们通过一个具体的例子，带大家领会这三点。

经典实例 - 枚举子集

比如需要枚举一个数字集合 S 的所有子集，你会如何做？

1. 状态。

我们可以用一个和 S 相同大小的数组 picked 记录每个数被选取的信息，用 0 表示没有选取，用 1 表示选取。

比如 S 大小为 3，picked 数组 [1,1,0]，表示 S 中的第一项和第二项被选择（索引从 1 开始）。如果 S 的大小为 n，就需要用一个长度为 n 的数组来存储，那么就有 2^n 种状态。

由于数组的值不是 0 就是 1，满足二值性，因此更多时候我们会使用一个数字 y 来表示状态，而不是上面的 picked 数组。其中 y 的二进制位对应上面提到的 picked 数组中的一项。比如如上的 picked 数组 [1,1,0] 可以用 ob110，也就是二进制的 6 来表示。

1. 不重不漏。

我们可以用一个数 x 来模拟集合 S，用 y 来模拟 picked 数组。这样问题就转化为两个数（x 和 y）的位运算。

由于我们使用 1 表示被选取，0 表示选取。因此如果 x 对应位为 0，其实 y 也只能是 0，而如果 x 对应位是 1，y 却可能是 0 或者 1。也就是说 y 一定小于等于 x，因此可以枚举所有小于等于 x 的数的二进制，并逐个

判断其是否真的是 x 的子集。

具体来说，我们可以令 n 为 x 的二进制位数。不难写出如下代码：

```
// 外层枚举所有小于等于 x 的数
ans = [];
for (i = 1; i < 1 << n; i++) {
    if ((x | i) === x) ans.push(i);
}
// ans 就是所有非空子集
```

这种算法的复杂度大约是 $O(4^n)$ ，也就是说和 x 成正比。这种算法 n 最多取到 12 左右。

这样做不重不漏么？

答案是可以的。因为 $(x | i) === x$ 就是 i 是 x 的子集的充要条件，当然你也可用 $\&$ ，即 $(x | i) \& i == i$ 来表示 i 是 x 的子集。

如果二进制你不好理解，其实你可以转化为十进制理解。比如我给你一个数 132，让你找 132 的子集，这里的子集我简单定义为当前位的数字是否小于等于原数字当前位的数字。这样我们就可以先从 1 枚举到 132，因为这些数潜在都可能是 132 的子集。如果我枚举了一个数字 030，由于 0 小于等于 1，3 小于等于 3，0 小于等于 2，因此 030 是 132 的子集。而如果我枚举了一个数字 040，由于 4 大于 3，因此 040 不是 132 的子集。

1. 效率。

上面的枚举方法虽然也可保证不重不漏，但是却不是最优的，这里介绍一种更好的枚举方法。

具体做法就是将 x_i 和 x 进行 $\&$ （与）运算。与运算可以快速跳到下一个子集。

```
ans = [];
// 外层枚举所有小于等于 x 的数
for (i = x; i != 0; i = (i - 1) & x) {
    ans.push(i);
}
// ans 就是所有非空子集
```

这样做不重不漏么？算法的关键在于 $i = (i - 1) \& x$ 。这个操作首先将 $i - 1$ ，从而把 i 最右边的 1 变成了 0，然后把这位之后的所有 0 变成了 1。经过这样的处理再与 x 求与，就保证了得到的结果是 x 的子集，并且一定是所有子集中小于 i 的最大的一个。直观来看就是倒序枚举除了所有非空子集。

对于有 n 个 1 的二进制数字，需要 2^n 的时间复杂度。而有 n 个 1 的二进制数字有 $C(n,i)$ 个，所以这段代码的时间复杂度为 $\sum_{i=0}^n C(n,i) \times 2^i$ ，大约是 $O(3^n)$ 。和上面一样，这种算法的时间复杂度也和 x 成正比。但是这种算法 n 最多取到 15 左右。

这种方法对题目有一定要求，即：

1. 数据范围要合适，否则数字无法表示了。
2. 只能有两种状态，这样才可以用二进制位 0 和 1 进行模拟。

枚举技巧

- 当你需要枚举 n 元组的时候，可以长度枚举 $n - 1$ ，然后使用一些技巧找第 n 个数。比如三数和为 target 的元组，我们可以先枚举两个数，然后使用二分或者哈希表加速找第三个数。
- 枚举的剪枝也是一个技巧。比如前面提到的有时候从后往前遍历可以剪枝。有时候利用序列的单调性进行剪枝。

大家可以通过以下题目进行联系，感受一下这两个技巧。

- [611. 有效三角形的个数](#)
- n 数和问题。

递推

高中的时候我们应该知道函数递推关系式，类似 $f(n) = f(n-1) + n$ 。

当我们能够得出函数的递推关系的时候，就可以根据递归关系一步步从 base case 逐步推导到原问题的解。

而上面的递推关系是最核心的，它可能是题目直接给出的递归关系，比如题目让你求 fibonacci 数列第 n 项。也有可能是需要自己挖掘，比如绝大多数的动态规划问题。

这里我们主要都假设已经递推关系的情况下，如果从 base case 递推到原问题。而关于递归关系的寻找，则放到后面的动态规划篇进行详细介绍。

仍然是上面的 $f(n) = f(n-1) + n$ 为例。

如果题目让你求 $f(100)$ 你会如何求？

首先需要明确的是，想这种递归关系的求解，一定要有一个 base case，否则会陷入无限循环。

那么 base case 是什么呢？我们可以先手动随意选择一种情况为 base case，这并不影响问题的求解。

比如我分别以：

- $f(99)$

989. 数组形式的整数加法

- $f(0)$
- $f(200)$

为 base case。

那么计算的结果会有所不同呢？不会的。不同的只是我们的代码。

那么 $f(99)$ 是多少呢？这不确定，这需要从题目中挖掘。

比如 $f(99)$ 是 4950，那么我们就可以写出如下代码：

```
function f(n) {
    if (n == 99) return 4950;
    return f(n - 1) + n;
}
```

比如 $f(0)$ 是 0，那么我们就可以写出如下代码：

```
function f(n) {
    if (n == 0) return 0;
    return f(n - 1) + n;
}
```

比如 $f(200)$ 是 4950，那么我们就可以写出如下代码：

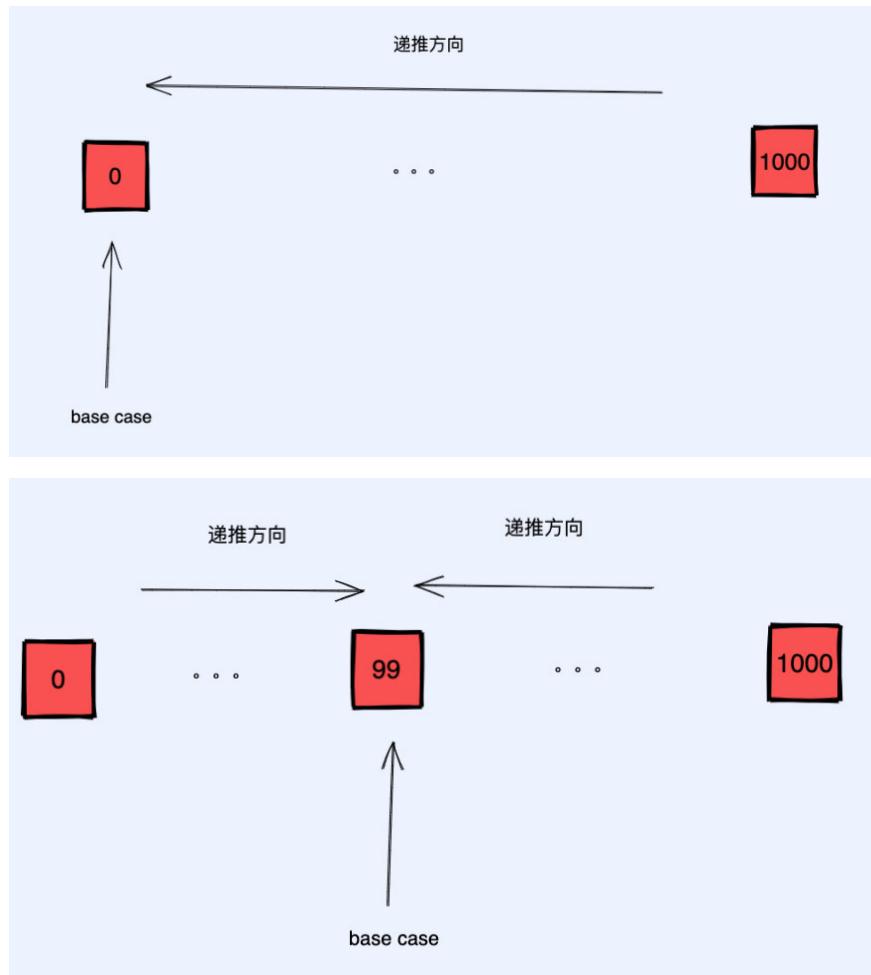
```
function f(n) {
    if (n == 99) return 20100;
    return f(n - 1) + n;
}
```

只不过上面的代码在特定的时候可能有 bug。比如选择了 $f(99)$ 为 base case，那么计算 $f(98)$ 就会陷入无限循环。

如何解决呢？答案是夹逼。

- 如果 n 大于 base case，则减少 n 到 base case。
- 如果 n 小于 base case，则增大 n 到 base case。

989. 数组形式的整数加法



以这道题来说，递推关系为 $f(n) = f(n-1) + n$ ，那么移项得 $f(n-1) = f(n) - n$ ，将 n 用 $n + 1$ 替换， $n - 1$ 用 n 替换，得 $f(n) = f(n + 1) - (n + 1)$

代码：

```
function f(n) {
    if (n == 99) return 4950;
    if (n > 99) return f(n - 1) + n;
    return f(n + 1) - (n + 1);
}
```

简单来说，我们的递归方向是不断趋向 **base case** 的。

因此我们可以得出结论：如果 **base case** 选择的题目的取值范围的中间位置，那么代码会比较难写，需要考虑解的方向使用不同的递归公式。而如果 **base case** 选择在题目取值范围的端点，就可以很好的解决这个问题。

因此如果题目取值范围是 $[0, 100000]$ ，那么选择 0 或者 100000 作为 **base case** 都是很方便的。类似地，如果题目取值范围是 $[-10, 100]$ ，那么选择 -10 或者 100 都是很方便的。

递推应用的非常广泛。比如前面树专题中讲解的求树的高度。计算树的深度就利用了递推关系 $f(x) = f(y) + 1$, 其中 x 为 y 的子节点, 而 base case 就是 x 为根节点, 此时 $f(x) = 0$ 。利用这个 base case 和递推关系就可以计算树中任意节点的深度。

由于计算树深度的 base case 是根节点, 因此我们需要前序遍历自顶向下的计算。而如果计算某个节点的子节点个数。不难得出有如下递推关系
$$f(x) = \sum_{i=0}^{n-1} f(a_i)$$
 其中 x 为树中的某一个节点, a_i 为树中节点的子节点。而 base case 则是没有任何子节点(也就是叶子节点), 此时 $f(x) = 1$ 。因此我们可以利用后序遍历自底向上完成子节点个数的统计。

总结

模拟是一种常见的思想, 简单题目很多都是直接模拟。对于中等和困难, 如果需要模拟, 则通常是模拟 + 一些其他知识点。不管是什么, 对于模拟来说, 我们要做的仅仅是将题目描述转化为代码, 这考察了我们代码能力。

枚举就是列出所有的可能, 有时候枚举可以很容易, 而有时候也可以很困难。大家在做枚举的时候, 要把握好三个要点, 并明确枚举的方向。有的时候仅仅是枚举策略的不同, 就会导致一个超时, 而另一个通过。

递推则是建立原问题和子问题之间的递推关系, 根据递推关系进行推导, 逐步从 base case 推导到原问题进行求解。递推在递归, 动态规划等高级算法主题中有着很重要的作用。

排序算法

很多时候，我们都需要用到排序算法。它在我们的世界中至关重要。比如，各种 app 都有的按价格、距离排序，再比如各种排行榜。他们的数据量可能非常巨大，因此排序算法的性能至关重要。虽然，实际工作和生活中我们不大可能去手写或者发明一个排序算法，不过理解其思想，对我们还是有帮助的。

排序算法已有有几十上百种算法了。我们不可能在这里一一讲解，不过我会选择若干有代表性的，并着重讲解其思想。

第一个关键点：交换。

排序的普遍性和重要性

经典的排序

冒泡排序

选择排序

插入排序

快速排序

归并排序

总结

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 狹义地二分查找，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出 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。

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

基本概念

首先我们要知道几个基本概念。这些概念对学习二分有着很重要的作用，之后遇到这些概念就不再讲述了，默认大家已经掌握。

解空间

解空间指的是题目所有可能的解构成的集合。比如一个题目所有解的可能是 1,2,3,4,5，但具体在某一种情况只能是其中某一个数（即可能是 1, 2, 3, 4, 5 中的一个数）。那么这里的解空间就是 1,2,3,4,5 构成的集合，在某一个具体的情况下可能是其中任意一个值，**我们的目标就是在某个具体的情况下判断其具体是哪个**。如果线性枚举所有的可能，就枚举这部分来说时间复杂度就是 $O(n)$ 。

举了例子：

如果让你在一个数组 `nums` 中查找 `target`，如果存在则返回对应索引，如果不存在则返回 -1。那么对于这道题来说其解空间是什么？

很明显解空间是区间 $[0, n-1]$ ，其中 n 为 `nums` 的长度。

需要注意的是上面题目的解空间只可能是区间 $[0, n-1]$ 之间的整数。而诸如 1.2 这样的小数是不可能存在的。这其实也是大多数二分的情况。但也有少部分题目解空间包括小数的。如果解空间包括小数，就可能会涉及到精度问题，这一点大家需要注意。

比如让你求一个数 x 的平方根，答案误差在 10^{-6} 次方都认为正确。这里容易知道其解空间大小可定义为 $[0, x]$ （当然可以定义得更精确，之后我们再讨论这个问题），其中解空间应该包括所有区间的实数，不仅仅是整数而已。这个时候解题思路和代码都没有太大变化，唯二需要变化的是：

1. 更新答案的步长。比如之前的更新是 `l = mid + 1`，现在可能就不行了，因此这样可能会错过正确解，比如正确解恰好就在区间 $[mid, mid+1]$ 内的某一个小数。
2. 判断条件时候需要考虑误差。由于精度的问题，判断的结束条件可能就要变成 **与答案的误差在某一个范围内**。

对于**搜索类题目**，解空间一定是有限的，不然问题不可解。对于搜索类问题，第一步就是需要明确解空间，这样你才能够在解空间内进行搜索。这个技巧不仅适用于二分法，只要是搜索问题都可以使用，比如 DFS，BFS 以及回溯等。只不过对于二分法来说，**明确解空间显得更为重要**。如果现在还不理解这句话也没关系，看完本文或许你就理解了。

定义解空间的时候的一个原则是：可以大但不可以小。因为如果解空间偏大（只要不是无限大）无非就是多做几次运算，而如果解空间过小则可能**错失正确解**，导致结果错误。比如前面我提到的求 x 的平方根，我们当然可以将解空间定义的更小，比如定义为 $[1, x/2]$ ，这样可以减少运算的次数。但如果设置地太小，则可能会错过正确解。这是新手容易犯错的点之一。

有的同学可能会说我看不出来怎么办呀。我觉得如果你实在拿不准也完全没关系，比如求 x 的平方根，就可以甚至为 $[1, x]$ ，就让它多做几次运算嘛。我建议你给上下界设置一个宽泛的范围。等你对二分逐步了解之后可以卡地更死一点。

序列有序

我这里说的是序列，并不是数组，链表等。也就是说二分法通常要求的序列有序，不一定是数组，链表，也有可能是其他数据结构。另外有的**序列有序**题目直接讲出来了，会比较容易。而有些则隐藏在题目信息之中。乍一看，题目并没有**有序**关键字，而有序其实就隐藏在字里行间。比如题目给了数组 `nums`，并且没有限定 `nums` 有序，但限定了 `nums` 为非负。这样如果给 `nums` 做前缀和或者前缀或（位运算或），就可以得到一个有序的序列啦。

更多技巧在四个应用部分展开哦。

虽然二分法不意味着需要序列有序，但大多数二分题目都有**有序**这个显著特征。只不过：

- 有的是题目直接限定了有序。这种题目通常难度不高，也容易让人想到用二分。
- 有的是需要你自己构造有序序列。这种类型的题目通常难度不低，需要大家有一定的观察能力。

比如[Triple Inversion](#)。题目描述如下：

```

Given a list of integers nums, return the number of pairs :

Constraints: n ≤ 100,000 where n is the length of nums
Example 1
Input:
nums = [7, 1, 2]
Output:
2
Explanation:
We have the pairs (7, 1) and (7, 2)

```

这道题并没有限定数组 `nums` 是有序的，但是我们可以构造一个有序序列 `d`，进而在 `d` 上做二分。代码：

```

class Solution:
    def solve(self, A):
        d = []
        ans = 0

        for a in A:
            i = bisect.bisect_right(d, a * 3)
            ans += len(d) - i
            bisect.insort(d, a)
        return ans

```

如果暂时不理解代码也没关系，大家先留个印象，知道有这么一种类型题即可，大家可以看完本章的所有内容（上下两篇）之后再回头做这道题。

极值

类似我在[堆专题](#)提到的极值。只不过这里的极值是静态的，而不是动态的。这里的极值通常指的是求第 k 大（或者第 k 小）的数。

堆的一种很重要的用法是求第 k 大的数，而二分法也可以求第 k 大的数，只不过二者的思路完全不同。使用堆求第 k 大的思路我已经在前面提到的堆专题里详细解释了。那么二分呢？这里我们通过一个例子来感受一下：这道题是 [Kth Pair Distance](#)，题目描述如下：

Given a list of integers nums and an integer k, return the

Constraints: $n \leq 100,000$ where n is the length of nums

Example 1

Input:

nums = [1, 5, 3, 2]

k = 3

Output:

2

Explanation:

Here are all the pair distances:

$\text{abs}(1 - 5) = 4$

$\text{abs}(1 - 3) = 2$

$\text{abs}(1 - 2) = 1$

$\text{abs}(5 - 3) = 2$

$\text{abs}(5 - 2) = 3$

$\text{abs}(3 - 2) = 1$

Sorted in ascending order we have [1, 1, 2, 2, 3, 4].

简单来说，题目就是给的一个数组 nums，让你求 nums 第 k 大的任意两个数的差的绝对值。当然，我们可以使用堆来做，只不过使用堆的时间复杂度会很高，导致无法通过所有的测试用例。这道题我们可以使用二分法来降维打击。

对于这道题来说，解空间就是从 0 到数组 nums 中最大最小值的差，用区间表示就是 $[0, \max(\text{nums}), \min(\text{nums})]$ 。明确了解空间之后，我们就需要对解空间进行二分。对于这道题来说，可以选当前解空间的中间值 mid，然后计算小于等于这个中间值的任意两个数的差的绝对值有几个，我们不妨令这个数字为 x。

- 如果 x 大于 k，那么任何解空间中大于等于 mid 的数都不可能是答案。
- 如果 x 小于 k，那么任何解空间中小于等于 mid 的数都不可能是答案。
- 如果 x 等于 k，那么 mid 就是答案。

基于此，我们可使用二分来解决。这种题型，我总结为计数二分。我会在后面的四大应用部分重点讲解。

代码：

```

class Solution:
    def solve(self, A, k):
        A.sort()
        def count_not_greater(diff):
            i = ans = 0
            for j in range(1, len(A)):
                while A[j] - A[i] > diff:
                    i += 1
                ans += j - i
            return ans
        l, r = 0, A[-1] - A[0]

        while l <= r:
            mid = (l + r) // 2
            if count_not_greater(mid) > k:
                r = mid - 1
            else:
                l = mid + 1
        return l

```

如果暂时不理解代码也没关系，大家先留个印象，知道有这么一种类型题即可，大家可以看完本章的所有内容（上下两篇）之后再回头做这道题。

一个中心

二分法的一个中心大家一定牢牢记住。其他（比如序列有序，左右双指针）都是二分法的手和脚，都是表象，并不是本质，而**折半才是二分法的灵魂**。

前面已经给大家明确了解空间的概念。而这里的折半其实就是解空间的折半。

比如刚开始解空间是 $[1, n]$ (n 为一个大于 n 的整数)。通过某种方式，我们确定 $[1, m]$ 区间都**不可能是答案**。那么解空间就变成了 $(m, n]$ ，持续此过程知道解空间变成平凡（直接可解）。

注意区间 $(m, n]$ 左侧是开放的，表示 m 不可能取到。

显然折半的难点是根据什么条件舍弃哪一步部分。这里有两个关键字：

1. 什么条件
2. 舍弃哪部分

几乎所有的二分的难点都在这两个点上。如果明确了这两点，几乎所有的二分问题都可以迎刃而解。幸运的是，关于这两个问题的答案通常都是有限的，题目考察的往往就是那几种。这其实也就是所谓的做题套路。关于这

些套路，我会在之后的四个应用部分给大家做详细介绍。

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： $\$O(\log N)\$$
- 最坏时间复杂度： $\$O(\log N)\$$
- 最优时间复杂度： $\$O(1)\$$
- 空间复杂度
 - 迭代： $\$O(1)\$$
 - 递归： $\$O(\log N)\$$ （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。我们将范围缩小到了 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。我们将范围缩小到了 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 $left \leq right$ 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 $\text{nums}[mid]$ 与 目标值比对。
 - 如果 $\text{nums}[mid]$ 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 $\text{nums}[mid]$ 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 $\text{nums}[mid]$ 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

989. 数组形式的整数加法

```
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

```
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

989. 数组形式的整数加法

```
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++

```
int binarySearch(vector<int>& nums, int target){
    if(nums.size() == 0)
        return -1;

    int left = 0, right = nums.size() - 1;
    while(left <= right){
        int mid = left + ((right - left) >> 1);
        if(nums[mid] == target){ return mid; }
        // 搜索区间变为 [mid+1, right]
        else if(nums[mid] < target)
            left = mid + 1;
        // 搜索区间变为 [left, mid - 1]
        else
            right = mid - 1;
    }
    return -1;
}
```

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 $\text{nums}[\text{mid}] > \text{target}$ 和 $\text{nums}[\text{mid}] == \text{target}$ 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

989. 数组形式的整数加法

```
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

```
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 -1;
    return left;
}
```

C++

989. 数组形式的整数加法

```
int binarySearchLeft(vector<int>& nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0, right = nums.size() - 1;
    while (left <= right) {
        int mid = left + ((right - left) >> 1);
        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.size() || nums[left] != target)
        return -1;
    return left;
}
```

例题解析

给你一个严格递增的数组 `nums`，让你找到第一个满足 `nums[i] == i` 的索引，如果没有这样的索引，返回 -1。 (你的算法需要有 $\log N$ 的复杂度)。

首先我们做一个小小的变换，将原数组 `nums` 转换为 `A`，其中 $A[i] = \text{nums}[i] - i$ 。这样新的数组 `A` 就是一个不严格递增的数组。这样原问题转换为在一个不严格递增的数组 `A` 中找第一个等于 0 的索引。接下来，我们就可以使用最左满足模板，找到最左满足 `nums[i] == i` 的索引。

代码：

```
class Solution:
    def solve(self, nums):
        l, r = 0, len(nums) - 1
        while l <= r:
            mid = (l + r) // 2
            if nums[mid] >= mid:
                r = mid - 1
            else:
                l = mid + 1
        return l if l < len(nums) and nums[l] == l else -1
```

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 $left \leq right$ 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 $left \leq right$ 。

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 $\text{nums}[mid]$ 与 目标值比对。
 - 如果 $\text{nums}[mid]$ 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 $\text{nums}[mid]$ 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 $\text{nums}[mid]$ 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 $\text{nums}[right]$ 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 $\text{nums}[mid] < target$ 和 $\text{nums}[mid] == target$ 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

989. 数组形式的整数加法

```
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

```
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

989. 数组形式的整数加法

```
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++

```
int binarySearchRight(vector<int>& nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0, right = nums.size() - 1;
    while (left <= right) {
        int mid = left + ((right - left) >> 1);
        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;
}
```

寻找最左插入位置

上面我们讲了 寻找最左满足条件的值。如果找不到，就返回 -1。那如果我想让你找不到不是返回 -1，而是应该插入的位置，使得插入之后列表仍然有序呢？

比如一个数组 `nums: [1,3,4]`, `target` 是 2。我们应该将其插入（注意不是真的插入）的位置是索引 1 的位置，即 `[1,2,3,4]`。因此 寻找最左插入位置 应该返回 1，而 寻找最左满足条件 应该返回-1。

另外如果有多个满足条件的值，我们返回最左侧的。比如一个数组 `nums: [1,2,2,2,3,4]`, `target` 是 2，我们应该插入的位置是 1。

思维框架

如果你将寻找最左插入位置看成是寻找最右满足小于 x 的位置 + 1，那就
可以和前面的知识产生联系，使得代码更加统一。

比如一个数组 `nums: [1,2,2,2,3,4]`, `target` 是 2，寻找最右满足小于 2 的
位置是 0，+ 1 就是 1，这其实就是最左插入位置。

具体算法：

- 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`。
- 当 `A[mid] >= x`，说明找到一个备胎，我们令 `r = mid - 1` 将 `mid` 从搜
索区间排除，继续看看有没有更好的备胎。
- 当 `A[mid] < x`，说明 `mid` 根本就不是答案，直接更新 `l = mid + 1`，从
而将 `mid` 从搜索区间排除。
- 最后搜索区间的 `l` 就是最好的备胎，备胎转正。

代码模板

Python

```

def bisect_left(nums, x):
    # 内置 api
    bisect.bisect_left(nums, x)
    # 手写
    l, r = 0, len(A) - 1
    while l <= r:
        mid = (l + r) // 2
        if A[mid] >= x: r = mid - 1
        else: l = mid + 1
    return l

```

其他语言暂时空缺，欢迎 [PR](#)

寻找最右插入位置

思维框架

如果你将寻找最右插入位置看成是寻找最左满足大于 x 的值，那就可以和前面的知识产生联系，使得代码更加统一。

比如一个数组 nums : [1,2,2,2,3,4], target 是 2，寻找最左满足大于 2 的位置是 4，这其实就是最右插入位置。

具体算法：

- 首先定义搜索区间为 $[\text{left}, \text{right}]$ ，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 $[\text{left}, \text{right}]$ ，因此当 $\text{left} \leq \text{right}$ 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 $\text{left} \leq \text{right}$ 。
- 当 $A[\text{mid}] > x$ ，说明找到一个备胎，我们令 $\text{r} = \text{mid} - 1$ 将 mid 从搜索区间排除，继续看看有没有更好的备胎。
- 当 $A[\text{mid}] \leq x$ ，说明 mid 根本就不是答案，直接更新 $\text{l} = \text{mid} + 1$ ，从而将 mid 从搜索区间排除。
- 最后搜索区间的 l 就是最好的备胎，备胎转正。

代码模板

Python

```

def bisect_right(nums, x):
    # 内置 api
    bisect.bisect_right(nums, x)
    # 手写
    l, r = 0, len(A) - 1
    while l <= r:
        mid = (l + r) // 2
        if A[mid] <= x: l = mid + 1
        else: r = mid - 1
    return l

```

其他语言暂时空缺，欢迎 PR

小结

对于二分题目首先要明确解空间，然后根据一定条件（通常是和中间值比较），舍弃其中一半的解。大家可以先从查找满足条件的值的二分入手，进而学习最左和最右二分。同时大家只需要掌握最左和最右二分即可，因此后者功能大于前者。

对于最左和最右二分，简单用两句话总结一下：

1. 最左二分不断收缩右边界，最终返回左边界
2. 最右二分不断收缩左边界，最终返回右边界

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 $\log n$, 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, 即 `[mid,end]` 部分有序。假如 `mid` 大于 `start`, 则 `mid` 一定在左边有序部分, 即 `[start,mid]` 部分有序。这是这类题目的突破口。

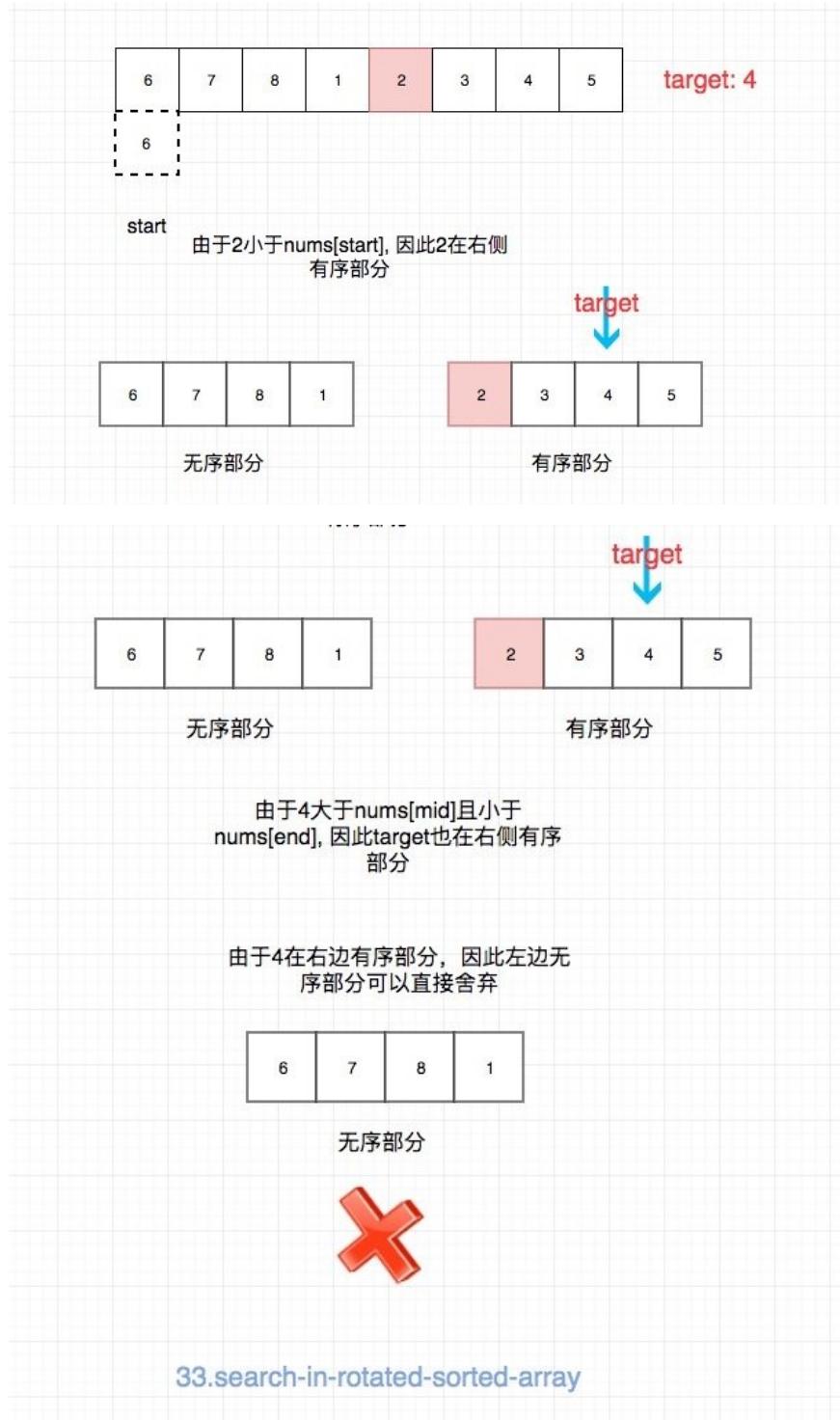
注意我没有考虑等号, 之后我会讲。

- 然后我们继续判断 `target` 在哪一部分, 就可以舍弃另一部分了。

也就是说只需要比较 `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]` 有序。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(通过 `start = mid + 1` 实现), 反之亦然。

我们以 $([6,7,8,1,2,3,4,5], 4)$ 为例讲解一下:

989. 数组形式的整数加法



接下来，我们考虑重复元素的问题。如果存在重复数字，就可能会发生 $\text{nums}[\text{mid}] == \text{nums}[\text{start}]$ 了，比如 30333。这个时候可以选择舍弃 start，也就是 start 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 start 和 mid，因此去除一个 start，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]: # trick
                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: $O(\log N)$
- 空间复杂度: $O(1)$

扩展

如果题目不是让你返回 true 和 false, 而是返回最左/最右等于 target 的索引呢? 这不就又和前面的知识建立联系了么? 比如我让你在一个旋转数组中找最左等于 target 的索引, 其实就是 [面试题 10.03. 搜索旋转数组](#)。

思路和前面的最左满足类似, 仍然是通过压缩区间, 更新备胎, 最后返回备胎的方式来实现。具体看代码吧。

Python Code:

```

class Solution:
    def search(self, nums: List[int], target: int) -> int:
        l, r = 0, len(nums) - 1
        while l <= r:
            mid = l + (r - l) // 2
            # # 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
            elif nums[l] > nums[mid]:
                # target is in the second half
                if nums[l] <= target or target <= nums[mid]:
                    r = mid - 1
                else:
                    l = mid + 1
            elif nums[l] == nums[mid]:
                if nums[l] != target:
                    l += 1
                else:
                    # l 是一个备胎
                    r = l - 1
        return l if l < len(nums) and nums[l] == target else -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](#)

989. 数组形式的整数加法

编写一个高效的算法来判断 $m \times n$ 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: int) -> bool:
        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 $\text{nums}[mid]$ 大于 $\text{nums}[r]$, 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 $l \geq r$ 或者 $\text{nums}[l] < \text{nums}[r]$ 的时候退出循环

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

代码 (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: $O(\log N)$
- 空间复杂度: $O(1)$

另一种二分法

思路

我们当然也可以和 $\text{nums}[l]$ 比较，而不是上面的 $\text{nums}[r]$ ，我们发现：

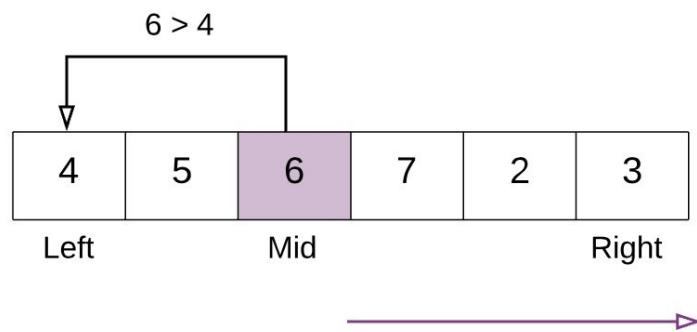
- 旋转点左侧元素都大于数组第一个元素
- 旋转点右侧元素都小于数组第一个元素

这样就建立了 $\text{nums}[mid]$ 和 $\text{nums}[0]$ 的联系。

具体算法：

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

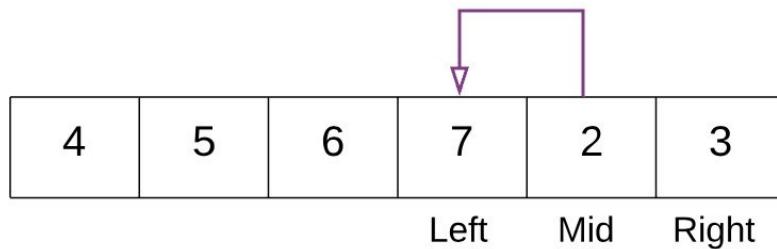
989. 数组形式的整数加法



- 如果中间元素 \leq 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. $\text{nums}[\text{mid}] > \text{nums}[\text{mid} + 1]$ ，因此 $\text{mid} + 1$ 是最小值。
3. $\text{nums}[\text{mid} - 1] > \text{nums}[\text{mid}]$ ，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 element
        # 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 element
            # This point would be the point of change. From here
            # onwards all elements will be greater than the previous
            if nums[mid] > nums[mid + 1]:
                return nums[mid + 1]
            # if the mid element is lesser than its previous element
            if nums[mid - 1] > nums[mid]:
                return nums[mid]

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

```

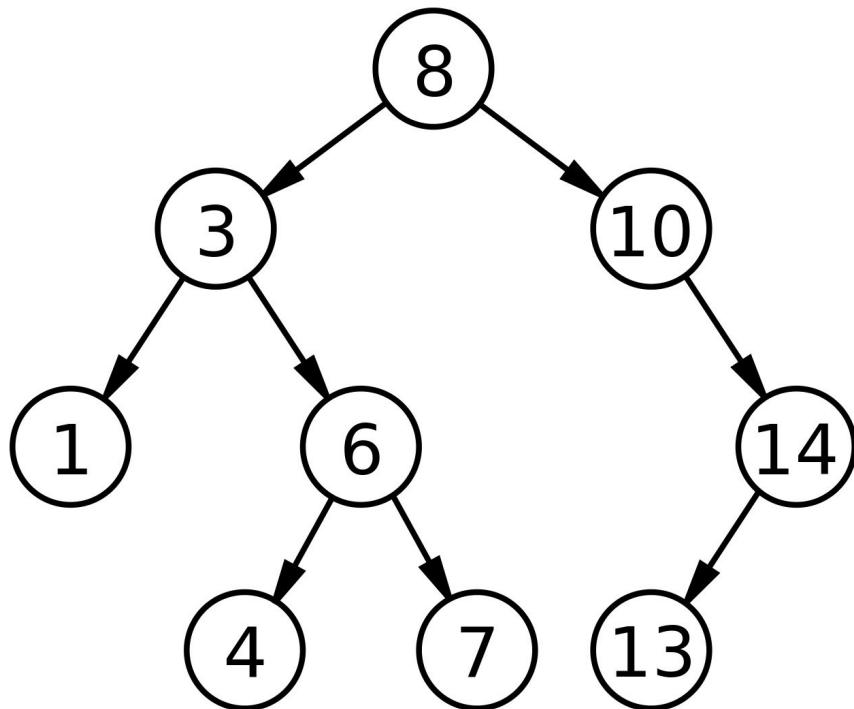
复杂度分析

- 时间复杂度: $O(\log N)$
- 空间复杂度: $O(1)$

二叉树

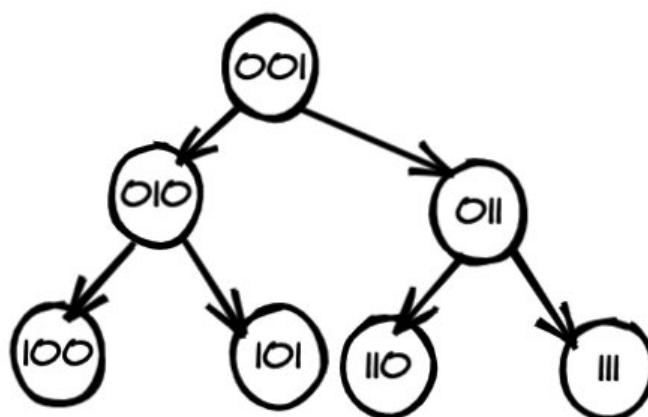
对于一个给定的二叉树，其任意节点最多只有两个子节点。从这个定义，我们似乎可以嗅出一点二分法的味道，但是这并不是二分。但是，二叉树中却和二分有很多联系，我们来看一下。

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，在一个二叉搜索树中进行搜索的过程就是二分法。

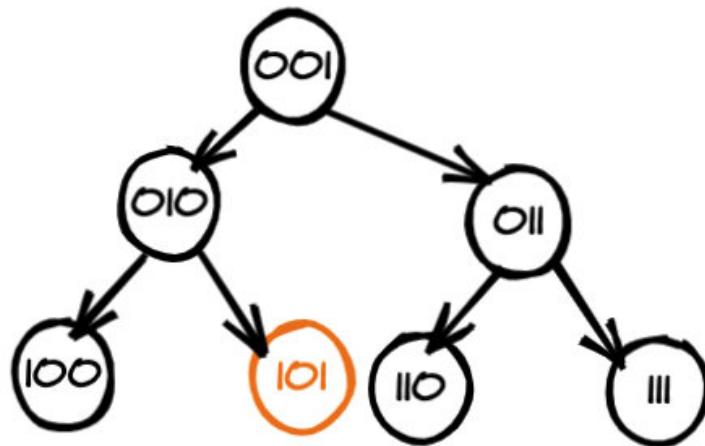


如上图，我们需要在这样一个二叉搜索树中搜索 7。那么我们的搜索路径则会是 $8 \rightarrow 3 \rightarrow 6 \rightarrow 7$ ，这也是一种二分法。只不过相比于普通的有序序列查找给定值二分，其时间复杂度的下界更差，原因在于二叉搜索树并不一定是二叉平衡树。

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 $01, 10, 11, \dots$ 。



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

四大应用

基础知识铺垫了差不多了。接下来，我们开始干货技巧。

接下来要讲的：

- 能力检测和计数二分本质差不多，都是普通二分的泛化。
- 前缀和二分和插入排序二分，本质都是在构建有序序列。

那让我们开始吧。

能力检测二分

能力检测二分一般是：定义函数 `possible`，参数是 `mid`，返回值是布尔值。外层根据返回值调整“解空间”。

示例代码（以最左二分为例）：

```
def ability_test_bs(nums):
    def possible(mid):
        pass
    l, r = 0, len(A) - 1
    while l <= r:
        mid = (l + r) // 2
        # 只有这里和最左二分不一样
        if possible(mid): l = mid + 1
        else: r = mid - 1
    return l
```

和最左最右二分这两种最最基本的类型相比，能力检测二分只是将 **while** 内部的 **if** 语句调整为了一个函数罢了。因此能力检测二分也分最左和最右两种基本类型。

基本上大家都可以用这个模式来套。明确了解题的框架，我们最后来看下能力检测二分可以解决哪些问题。这里通过三道题目带大家感受一下，类似的题目还有很多，大家课后自行体会。

875. 爱吃香蕉的珂珂（中等）

题目地址

<https://leetcode-cn.com/problems/koko-eating-bananas/>

题目描述

989. 数组形式的整数加法

珂珂喜欢吃香蕉。这里有 N 堆香蕉，第 i 堆中有 $piles[i]$ 根香蕉。警卫已经睡着了，珂珂可以决定她吃香蕉的速度 K （单位：根/小时）。每个小时，她将选择一堆香蕉并吃完。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

题目是让我们求 H 小时内吃掉所有香蕉的最小速度。

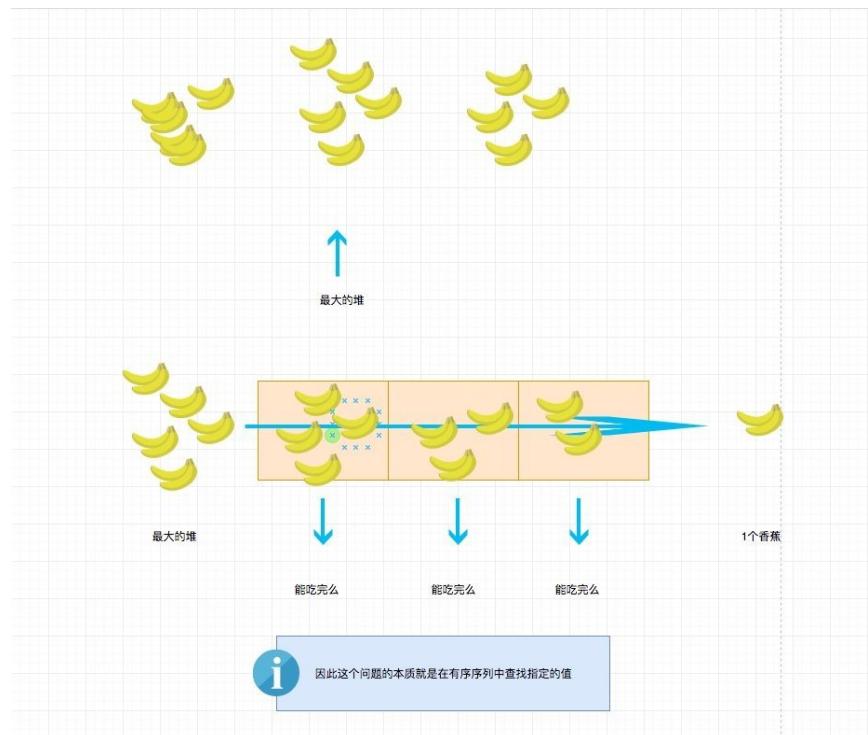
符合直觉的做法是枚举所有可能的速度，找出所有的可以吃完香蕉的速度，接下来选择最小的速度即可。由于需要返回最小的速度，因此选择从小到大枚举会比较好，因为可以提前退出。这种解法的时间复杂度比较高，为 $O(N * M)$ ，其中 N 为 $piles$ 长度， M 为 $Piles$ 中最大的数（也就是解空间的最大值）。

观察到需要检测的解空间是个**有序序列**，应该想到可能能够使用二分来解决，而不是线性枚举。可以使用二分解决的关键和前面我们简化的二分问题并无二致，关键点在于**如果速度 k 吃不完所有香蕉，那么所有小于等于 k 的解都可以被排除**。

二分解决的关键在于：

- 明确解空间。对于这道题来说，解空间就是 $[1, \max(\text{piles})]$ 。
- 如何收缩解空间。关键点在于**如果速度 k 吃不完所有香蕉，那么所有小于等于 k 的解都可以被排除**。

综上，我们可以使用最左二分，即不断收缩右边界。



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

- 二分查找模板

代码

代码支持：Python, JavaScript

Python Code:

989. 数组形式的整数加法

```
class Solution:
    def solve(self, piles, k):
        def possible(mid):
            t = 0
            for pile in piles:
                t += (pile + mid - 1) // mid
            return t <= k

        l, r = 1, max(piles)

        while l <= r:
            mid = (l + r) // 2
            if possible(mid):
                r = mid - 1
            else:
                l = mid + 1
        return l
```

JavaScript Code:

989. 数组形式的整数加法

```
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 都是一样的
    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： $O(\max(N, N * \log M))$ ，其中 N 为 $piles$ 长度， M 为 $Piles$ 中最大的数。
- 空间复杂度： $O(1)$

最小灯半径（困难）

题目描述

```
You are given a list of integers nums representing coordinates on a 2D plane.
```

Constraints

```
 $n \leq 100,000$  where n is the length of nums
```

```
Example 1
```

```
Input
```

```
nums = [3, 4, 5, 6]
```

```
Output
```

```
0.5
```

```
Explanation
```

```
If we place the lamps on 3.5, 4.5 and 5.5 then with r = 0.5 all houses are covered.
```

前置知识

- 排序
- 二分法

二分法

思路

本题和力扣 [475. 供暖器](#) 类似。

这道题的意思是给你一个数组 `nums`, 让你在 $[\min(\text{nums}), \max(\text{nums})]$ 范围内放置 3 个灯, 每个灯覆盖半径都是 r , 让你求最小的 r 。

之所以不选择小于 $\min(\text{nums})$ 的位置和大于 $\max(\text{nums})$ 的位置是因为没有必要。比如选取了小于 $\min(\text{nums})$ 的位置 pos , 那么选取 pos 一定不如选择 $\min(\text{nums})$ 位置结果更优。

这道题的核心点还是一样的思维模型, 即:

- 确定解空间。这里的解空间其实就是 r 。不难看出 r 的下界是 0, 上界是 $\max(\text{nums}) - \min(\text{nums})$ 。

没必要十分精准, 只要错过正确解即可, 这个我们在前面讲过, 这里再次强调一下。

- 对于上下界之间的所有可能 x 进行枚举 (不妨从小到大枚举), 检查半径为 x 是否可以覆盖所有, 返回第一个可以覆盖所有的 x 即可。

注意到我们是在一个有序序列进行枚举, 因此使用二分就应该想到。可使用二分的核心点在于: 如果 x 不行, 那么小于 x 的所有半径都必然不行。

接下来的问题就是给定一个半径 x , 判断其是否可覆盖所有的房子。

判断其是否可覆盖就是所谓的能力检测, 我定义的函数 `possible` 就是能力检测。

989. 数组形式的整数加法

首先对 `nums` 进行排序，这在后面会用到。然后从左开始模拟放置灯。先在 `nums[0] + r` 处放置一个灯，其可以覆盖 $[0, 2r]$ 。由于 `nums` 已经排好序了，那么这个等可以覆盖到的房间其实就是 `nums` 中坐标小于等于 $2r$ 所有房间，使用二分查找即可。对于 `nums` 右侧的所有房间我们需要继续放置灯，采用同样的方式即可。

能力检测核心代码：

```
def possible(diameter):
    start = nums[0]
    end = start + diameter
    for i in range(LIGHTS):
        idx = bisect_right(nums, end)
        if idx >= N:
            return True
        start = nums[idx]
        end = start + diameter
    return False
```

由于我们想要找到满足条件的最小值，因此可直接套用**最左二分模板**。

代码

代码支持：Python3

Python3 Code:

989. 数组形式的整数加法

```
class Solution:
    def solve(self, nums):
        nums.sort()
        N = len(nums)
        if N <= 3:
            return 0
        LIGHTS = 3
        # 这里使用的是直径，因此最终返回需要除以 2
        def possible(diameter):
            start = nums[0]
            end = start + diameter
            for i in range(LIGHTS):
                idx = bisect_right(nums, end)
                if idx >= N:
                    return True
                start = nums[idx]
                end = start + diameter
            return False

        l, r = 0, nums[-1] - nums[0]
        while l <= r:
            mid = (l + r) // 2
            if possible(mid):
                r = mid - 1
            else:
                l = mid + 1
        return l / 2
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：由于进行了排序，因此时间复杂度大约是 $O(n \log n)$
- 空间复杂度：取决于排序的空间消耗

778. 水位上升的泳池中游泳（困难）

题目地址

<https://leetcode-cn.com/problems/swim-in-rising-water>

题目描述

989. 数组形式的整数加法

在一个 $N \times N$ 的坐标方格 grid 中，每一个方格的值 $\text{grid}[i][j]$ 表示在

现在开始下雨了。当时间为 t 时，此时雨水导致水池中任意位置的水位为

你从坐标方格的左上平台 $(0, 0)$ 出发。最少耗时多久你才能到达坐标方格的右下角 (N, N) ？

示例 1：

输入： $[[0,2],[1,3]]$

输出： 3

解释：

时间为 0 时，你位于坐标方格的位置为 $(0, 0)$ 。

此时你不能游向任意方向，因为四个相邻方向平台的高度都大于当前时间为 0 时的水位 0。

等时间到达 3 时，你才可以游向平台 $(1, 1)$ 。因为此时的水位是 3，坐标方格中所有高度都小于或等于 3。

示例 2：

输入： $[[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20]]$

输出： 16

解释：

0 1 2 3 4

24 23 22 21 5

12 13 14 15 16

11 17 18 19 20

10 9 8 7 6

最终的路线用加粗进行了标记。

我们必须等到时间为 16，此时才能保证平台 $(0, 0)$ 和 $(4, 4)$ 是连通的。

提示：

$2 \leq N \leq 50$ 。

$\text{grid}[i][j]$ 位于区间 $[0, \dots, N \times N - 1]$ 内。

前置知识

- [DFS](#)
- [二分](#)

思路

首先明确一下解空间。不难得出，解空间是 $[0, \max(\text{grid})]$ ，其中 $\max(\text{grid})$ 表示 grid 中的最大值。

因此一个简单的思路是一个个试。

- 试试 a 可以不可以
- 试试 $a+1$ 可以不可以

• . . .

试试 x 是否可行就是能力检测。

实际上，如果 x 不可以，那么小于 x 的所有值都是不可以的，这正是本题的突破口。基于此，我们同样可使用讲义中的最左二分模板解决。

伪代码：

```
def test(x):
    pass
while l <= r:
    mid = (l + r) // 2
    if test(mid, 0, 0):
        r = mid - 1
    else:
        l = mid + 1
return l
```

这个模板会在很多二分中使用。比如典型的计数型二分，典型的就是计算小于等于 x 的有多少，然后根据答案更新解空间。

明确了这点，剩下要做的就是完成能力检测部分（`test` 函数）了。其实这个就是一个普通的二维网格 `dfs`，我们从 $(0,0)$ 开始在一个二维网格中搜索，直到无法继续或达到 $(N-1,N-1)$ ，如果可以达到 $(N-1,N-1)$ ，我们返回 `true`，否则返回 `False` 即可。对二维网格的 `DFS` 不熟悉的同学可以看下我之前写的小岛专题

[代码](#)

```

class Solution:
    def swimInWater(self, grid: List[List[int]]) -> int:
        l, r = 0, max([max(vec) for vec in grid])
        seen = set()

        def test(mid, x, y):
            if x > len(grid) - 1 or x < 0 or y > len(grid[0]) - 1:
                return False
            if grid[x][y] > mid:
                return False
            if (x, y) == (len(grid) - 1, len(grid[0]) - 1):
                return True
            if (x, y) in seen:
                return False
            seen.add((x, y))
            ans = test(mid, x + 1, y) or test(mid, x - 1, y) or test(mid, x, y + 1) or test(mid, x, y - 1)
            return ans
        while l <= r:
            mid = (l + r) // 2
            if test(mid, 0, 0):
                r = mid - 1
            else:
                l = mid + 1
            seen = set()
        return l

```

复杂度分析

- 时间复杂度: $O(N \log M)$, 其中 M 为 grid 中的最大值, N 为 grid 的总大小。
- 空间复杂度: $O(N)$, 其中 N 为 grid 的总大小。

计数二分

计数二分和上面的思路已经代码都基本一致。直接看代码会清楚一点：

```

def count_bs(nums, k):
    def count_not_greater(mid):
        pass
    l, r = 0, len(A) - 1
    while l <= r:
        mid = (l + r) // 2
        # 只有这里和最左二分不一样
        if count_not_greater(mid) > k: r = mid - 1
        else: l = mid + 1
    return l

```

可以看出只是将 `possible` 变成了 `count_not_greater`，返回值变成了数字而已。

实际上，我们可以将上面的代码稍微改造一下，使得两者更像：

```

def count_bs(nums, k):
    def possible(mid, k):
        # ...
        return cnt > k
    l, r = 0, len(A) - 1
    while l <= r:
        mid = (l + r) // 2
        if possible(mid, k): r = mid - 1
        else: l = mid + 1
    return l

```

是不是基本一致了？

由于和上面基本一致，因此这里直接推荐一个题目，大家用我的思路练习一下，看看我的技巧灵不灵。

- 第 k 小的距离对

前缀和二分

前面说了：如果数组全是正的，那么其前缀和就是一个严格递增的数组，基于这个特性，我们可以在其之上做二分。类似的有单调栈/队列。这种题目类型很多，为了节省篇幅就不举例说明了。提出前缀和二分的核心的点在于让大家保持对有序序列的敏感度。

插入排序二分

除了上面的前缀和之外，我们还可以自行维护有序序列。一般有两种方式：

989. 数组形式的整数加法

- 直接对序列排序。

代码表示：

```
nums.sort()  
bisect.bisect_left(nums, x) # 最左二分  
bisect.bisect_right(nums, x) # 最右二分
```

- 遍历过程维护一个新的有序序列，有序序列的内容为已经遍历过的值的集合。

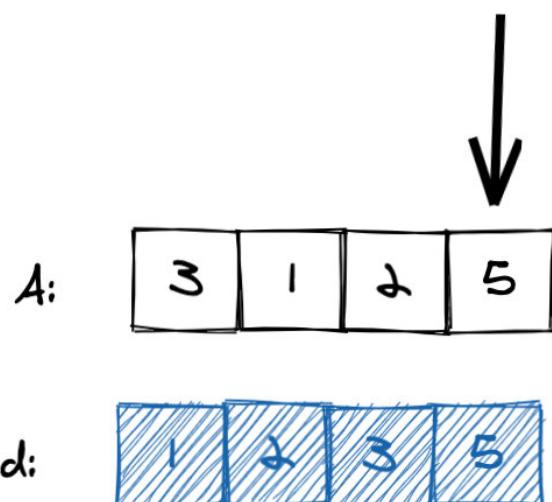
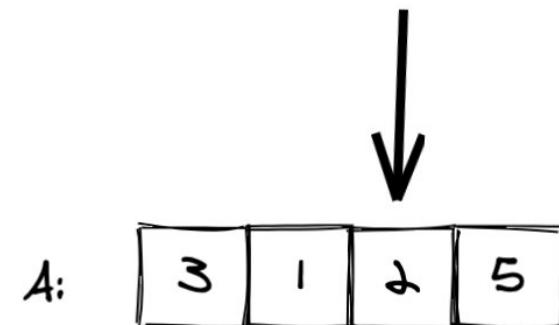
比如无序数组 [3,2,10,5]，遍历到索引为 2 的项（也就是值为 10 的项）时，我们构建的有序序列为 [2,3,10]。

注意我描述的是有序序列，并不是指数组、链表等具体的数据结构。而实际上，这个有序序列很多情况下是平衡二叉树。后面题目会体现这一点。

代码表示：

```
d = SortedList()  
for a in A:  
    d.add(a) # 将 a 添加到 d，并维持 d 中数据有序
```

上面代码的 d 就是有序序列。



理论知识到此为止，接下来通过一个例子来说明。

327. 区间和的个数(困难)

题目地址

<https://leetcode-cn.com/problems/count-of-range-sum>

题目描述

989. 数组形式的整数加法

给定一个整数数组 `nums`。区间和 $S(i, j)$ 表示在 `nums` 中，位置从 i 到 j 的所有整数的和，包括 i 、 j 两个端点。
请你以下标 i ($0 \leq i \leq \text{nums.length}$) 为起点，元素个数逐次递增，
当元素和落在范围 $[\text{lower}, \text{upper}]$ (包含 `lower` 和 `upper`) 之内时，计
求数组中，值位于范围 $[\text{lower}, \text{upper}]$ (包含 `lower` 和 `upper`) 之内的

注意：

最直观的算法复杂度是 $O(n^2)$ ，请在此基础上优化你的算法。

示例：

输入: `nums = [-2,5,-1]`, `lower = -2`, `upper = 2`,

输出: 3

解释:

下标 $i = 0$ 时，子数组 $[-2]$ 、 $[-2,5]$ 、 $[-2,5,-1]$ ，对应元素和分别为 -2 、 3 、 0 。
下标 $i = 1$ 时，子数组 $[5]$ 、 $[5,-1]$ ，元素和 5 、 4 ；没有满足题意的有效区间。
下标 $i = 2$ 时，子数组 $[-1]$ ，元素和 -1 ；记录有效区间和 $S(2,2)$ 。
故，共有 3 个有效区间。

提示:

$0 \leq \text{nums.length} \leq 10^4$

思路

题目很好理解。

由前缀和的性质知道：区间 i 到 j (包含) 的和 $\text{sum}(i,j) = \text{pre}[j] - \text{pre}[i-1]$ ，
其中 $\text{pre}[i]$ 为数组前 i 项的和 $0 \leq i < n$ 。

但是题目中的数字可能是负数，前缀和不一定是单调的啊？这如何是好呢？答案是手动维护前缀和的有序性。

比如 $[-2,5,-1]$ 的前缀和为 $[-2,3,2]$ ，但是我们可以将求手动维护为 $[-2,2,3]$ ，这样就有序了。但是这丧失了索引信息，因此这个技巧仅适用于无需考虑索引，也就是不需要求具体的子序列，只需要知道有这么一个子序列就行了，具体是哪个，我们不关心。

比如当前的前缀和是 cur ，那么前缀和小于等于 $\text{cur} - \text{lower}$ 有多少个，就说明以当前结尾的区间和大于等于 lower 的有多少个。类似地，前缀和小于等于 $\text{cur} - \text{upper}$ 有多少个，就说明以当前结尾的区间和大于等于 upper

989. 数组形式的整数加法

的有多少个。

基于这个想法，我们可使用二分在 $\$logn\$$ 的时间快速求出这两个数字，
使用平衡二叉树代替数组可使得插入的时间复杂度降低到 $\$O(logn)\$$ 。
Python 可使用 SortedList 来实现，Java 可用 TreeMap 代替。

代码

```
from sortedcontainers import SortedList
class Solution:
    def countRangeSum(self, A: List[int], lower: int, upper: int):
        ans, pre, cur = 0, SortedList([0]), 0
        for a in A:
            cur += a
            ans += pre.bisect_right(cur - upper) - pre.bisect_left(cur - lower)
            pre.add(cur)
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度: $\$O(nlogn)\$$
- 空间复杂度: $\$O(nlogn)\$$

493. 翻转对 (困难)

题目地址

<https://leetcode-cn.com/problems/reverse-pairs/>

题目描述

989. 数组形式的整数加法

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

- 二分

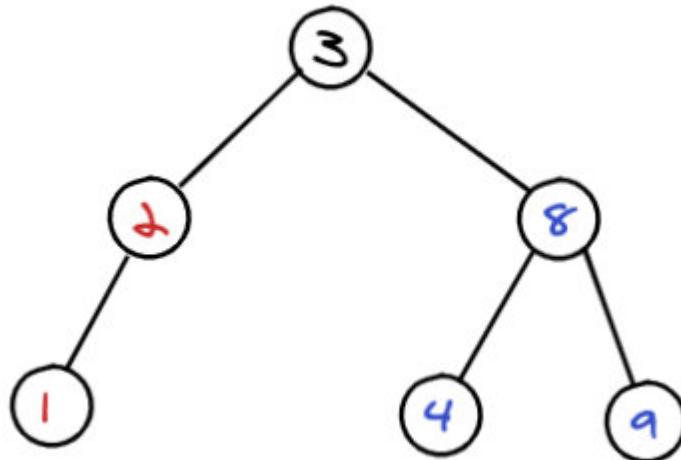
公司

- 暂无

思路

我们可以一边遍历一边维护一个有序序列 d ，其中 d 为已经遍历过的值的集合。对于每一个位置 $0 \leq i < n$ ，我们统计 d 中大于 $2 * A[i]$ 的个数，这个个数就是题目要求的翻转对。这里的关键在于 d 中的值是比当前索引小的全部值。

我们当然可以线性遍历 d ，求出个数。一个更好的方法是在遍历的同时维持 d 是有序的，这样我们就可以用二分了。和上面题目一样，使用平衡二叉树代替数组可使得插入的时间复杂度降低到 $O(\log n)$ 。



关键点

- 插入排序二分

代码

- 语言支持: Python3

Python3 Code:

```
from sortedcontainers import SortedList
class Solution:
    def reversePairs(self, A: List[int]) -> int:
        d = SortedList()
        ans = 0
        for a in A:
            ans += len(d) - d.bisect_right(2*a)
            d.add(a)
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度: $O(n \log n)$
- 空间复杂度: $O(n)$

小结

四个应用讲了两种构造有序序列的方式，分别是前缀和，插入排序，插入排序的部分其实也可以看下我之前写的[最长上升子序列系列](#)，那里面的贪心解法就是自己构造有序序列再二分的。另外理论上单调栈/队列也是有

序的，也可是用来做二分，但是相关题目太少了，因此大家只要保持对有序序列的敏感度即可。

能力检测二分很常见，不过其仅仅是将普通二分的 if 部分改造成了函数而已。而对于计数二分，其实就是能力检测二分的特例，只不过其太常见了，就将其单独提取出来了。

另外，有时候有序序列也会给你稍微变化一种形式。比如二叉搜索树，大家都知道可以在 $\$logn\$$ 的时间完成查找，这个查找过程本质也是二分。

二叉查找树有有序序列么？有的！二叉查找树的中序遍历恰好就是一个有序序列。因此如果一个数比当前节点值小，一定在左子树（也就是有序序列的左侧），如果一个数比当前节点值大，一定在右子树（也就是有序序列的右侧）。

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，必要的时候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。

另外本文主要讲了两种二分类型：最左和最右，模板已经给大家了，大家只需要根据题目调整解空间和判断条件即可。关于四种应用更多的还是让大家理解二分的核心折半。表面上来看，二分就是对有序序列的查找。其实不然，只不过有序序列很容易做二分罢了。因此战术上大家保持对有序序列的敏感度，战略上要明确二分的本质是折半，核心在于什么时候将哪一半折半。

一个问题能否用二分解决的关键在于检测一个值的时候是否可以排除解空间中的一半元素。比如我前面反复提到的如果 x 不行，那么解空间中所有小于等于 x 的值都不行。

对于简单题目，通常就是给你一个有序序列，让你在上面找满足条件的位置。顶多变化一点，比如数组局部有序，一维变成二维等。对于这部分可以看下我写的[91 算法 - 二分查找讲义](#)

中等题目可能需要让你自己构造有序序列。

困难题则可能是二分和其他专题的结合，比如上面的 778. 水位上升的泳池中游泳（困难），就是二分和搜索（我用的是 DFS）的结合。

以上就是本文的全部内容了，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。我是 lucifer，维护西湖区最好的算法题解，Github 超 40K star。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。

另外我整理的 1000 多页的电子书已限时免费下载，大家可以去我的公众号《力扣加加》后台回复电子书获取。

滑动窗口

简介

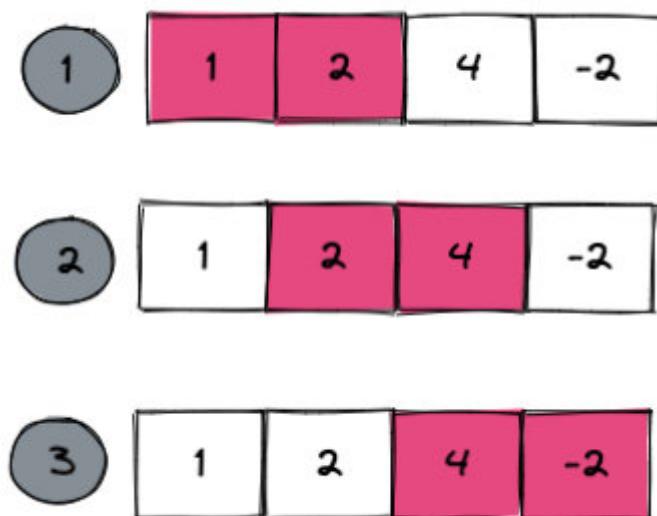
笔者最早接触滑动窗口是滑动窗口协议，滑动窗口协议（Sliding Window Protocol），属于TCP协议的一种应用，用于网络数据传输时的流量控制，以避免拥塞的发生。发送方和接收方分别有一个窗口大小 w_1 和 w_2 。窗口大小可能会根据网络流量的变化而有所不同，但是在更简单的实现中它们是固定的。窗口大小必须大于零才能进行任何操作。

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。

算法中的滑动窗口（以下简称滑窗），也就是 Sliding Window，是利用双指针（两个指针用于界定窗口的左右边界）在某种数据结构上（大部分基本上都是 Array）虚构出了一个窗口，并且该窗口还会进行移动，窗口移动的时候利用已有的窗口的计算信息重新出新的窗口的计算信息，其本质是一种空间换时间的算法。该专题希望大家可以对该思想有一个系统的认识以及训练。

滑动窗口的核心

为了方便大家理解，不妨思考一下如何求数组连续 k 个数的和的最大值。比如数组是 [1,2,4,-2], $k = 2$ 。那么连续 k 个数的子序列有 [1,2], [2,4], [4,-2]，其中最大的是 [2,4]，其和为 6。



989. 数组形式的整数加法

首先我们使用暴力枚举的方式进行求解。即暴力枚举所有的长度为 k 的子序列并比较大小返回最大的。

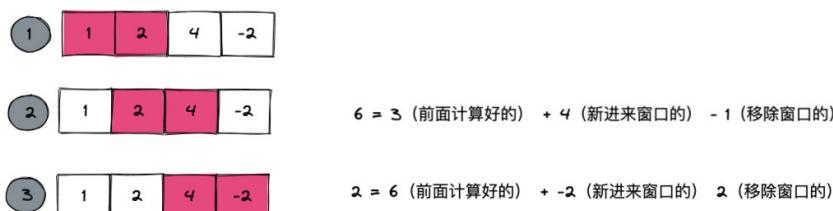
暴力枚举的方法也比较容易想到，就是直接使用双层循环固定左右两个数组端点。最后只需要将两个端点间的数相加即可。而由于窗口长度固定为 k，

因此固定一个端点，另外一个端点也随之固定。比如我们固定了左端点 l，那么右端点就是 l + k，其中 $0 \leq l < n - k + 1$ 。

```
res = 1
# 枚举左端点
for l in range(n - k + 1):
    res = max(res, sum(nums[l:l+k]))
return res
```

不难发现时间复杂度已经达到了 $O(N^2)$ 。这是因为我们每移动一次都抛弃了前次的信息并重新计算，效率大打折扣。

不难发现，每向后移动一次大小为 k 的窗口，实际上变化的只有窗口两端的元素，具体地，新窗口元素和 = 旧窗口元素和 - 左边移除的元素 + 右边进来的元素。



这样就可以写出如下 $O(N)$ 时间复杂度的代码了：

```
window_sum = sum([elem for elem in n[:k]])
res = window_sum

for i in range(k, len(n) - k + 1):
    # window_sum 很好的利用了前面的计算好的 window_sum, 而不是傻{
    window_sum = window_sum - n[i] + n[i + k]
    res = max(res, window_sum)

return res
```

这次就利用上了前面计算过的部分信息啦。这就是滑动窗口的核心，也就是滑动窗口主要是为了解决没有利用前面状态计算好的信息而重新计算带来的计算复杂度增加的问题。很多算法都是基于这种优化思想产生的，比如前缀和。

如果数组不是一次性给出的，而是基于流的，那么使用滑动窗口是必须的。而流在工程中经常出现，比如我要计算一段时间内股票 k 线图，因此滑动窗口算法实际上有着非常广泛的意义。

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：

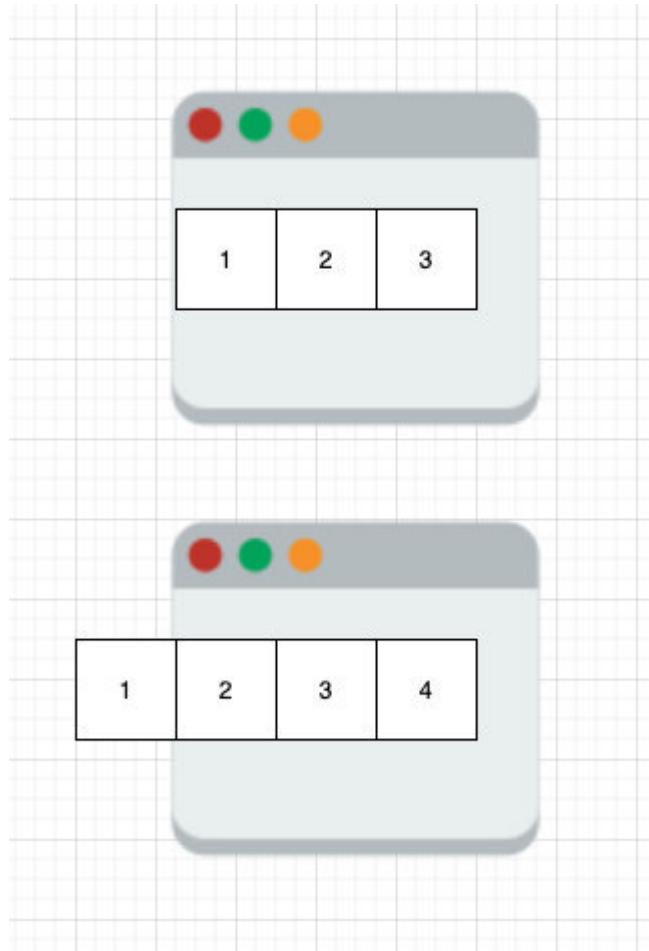
- 固定窗口大小
- 窗口大小不固定，求解最大的满足条件的窗口
- 窗口大小不固定，求解最小的满足条件的窗口（上面的 209 题就属于这种）

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。

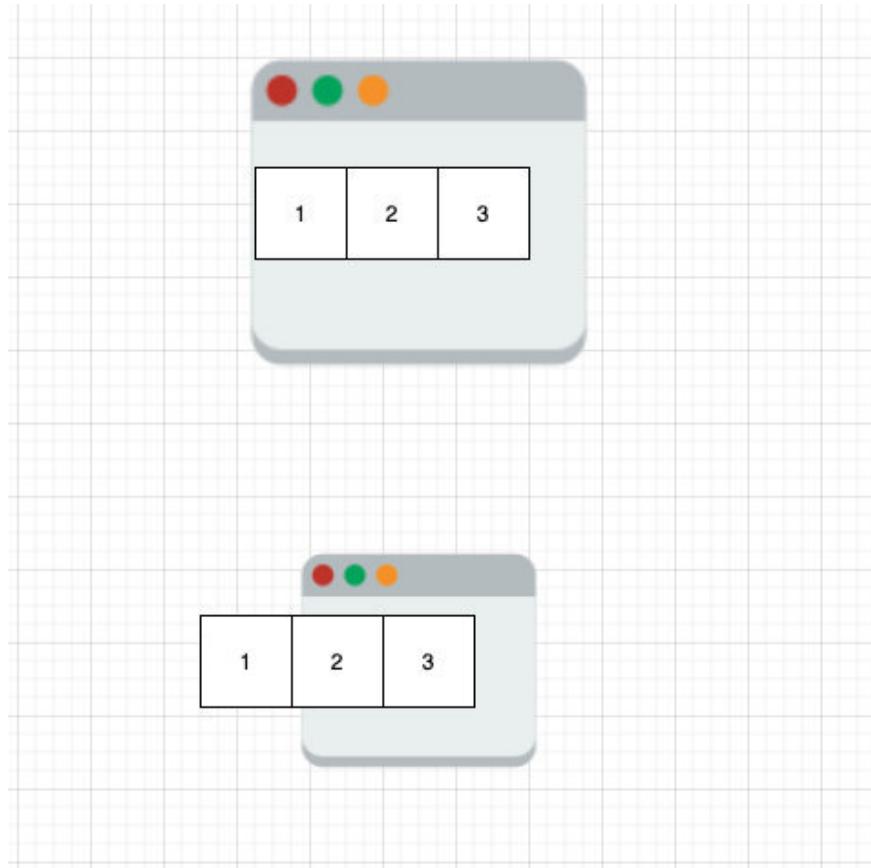


可变窗口大小

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。



解题步骤

上面的常见套路都提到了是否满足条件。而这个条件其实也就是题目的条件，不同的题目条件不同，没有一个普适的标准，大家做题的时候如果不会可以回头再看下讲义，反思一下题目的条件。

另外我们主要介绍了窗口求和。其实除此之外也可以求其他的统计信息，比如求异或。比如我出一个题目求数组连续 k 个数的异或的最大值。你怎么求？当然是一样的了。这里只需要知道异或的自反性即可。

比如加的反运算是减，而异或的自反性指的是异或的反运算还是本身。

然而不管是可变窗口还是不可变窗口，我们都可以将算法流程大致抽象出以下三个步骤：

1. 向窗口添加元素：需要判断当前情况我们是否需要移动右侧边界来进行添加。
2. 从窗口删除元素：需要判断当前情况我们是否需要移动左侧边界来进行删除。
3. 更新信息：只要窗口的边界情况发生了改变，我们就需要动态的更新窗口中我们所需的信息。

上面三步可不是线性执行顺序，添加和删除在一些情况可能是连续进行的，也就是第一步或第二部可能有连续执行的情况，只要有第一步或第二步执行过，那么我们就要执行一次第三步。

因此滑动窗口的难点其实需要我们明确在什么情况下移动左/右边界。这里不做过多展开解释，我会选出比较经典的题来给大家练习讲解。

下面给出伪代码：

```
初始化窗口window

while 右边界 < 合法条件:
    # 右边界扩张
    window右边界+1
    更新状态信息
    # 左边界收缩
    while left < right and 符合收缩条件:
        window左边界+1
        更新状态信息
```

下面给一个真实可运行代码。这段是 209 题的代码，使用 Python 编写，大家意会即可。

```
class Solution:
    def minSubArrayLen(self, s: int, nums: List[int]) -> int:
        l = total = 0
        ans = len(nums) + 1
        # 右指针 r, 左指针 l
        for r in range(len(nums)):
            # 统计窗口信息 (求和)
            total += nums[r]
            # 判断是否需要收缩
            while total >= s:
                ans = min(ans, r - l + 1)
                total -= nums[l]
                # 收缩
                l += 1
        return 0 if ans == len(nums) + 1 else ans
```

推荐题目

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](#))
- [978. 最长湍流子数组](#)
- [【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](#))
- [1658. 将 x 减到 0 的最小操作数](#)

总结

滑动窗口核心就是解决没有利用前置状态计算好的信息而带来的计算复杂度增加的问题，因为每次移动窗口变化的其实只是窗口两端的部分，中间的内容是不会变的。

从窗口大小是否可变，我们可将滑动窗口分为可变滑动窗口和固定滑动窗口，相对来说前者更为复杂。

关于滑动窗口我们需要判断何时更新指针。关于如何更新指针，很难用几句话归纳，因此通过做题来总结是一种好的方式。希望大家在做题的过程多多回顾讲义内容，结合题目攻克滑动窗口难关。

需要注意的是，一旦涉及到非定长窗口大小的问题，一般都较难界定何种情况来移动对应指针，所以在 lc 上直观表现就是个 hard 题，但实际上，如果一旦做了出来或者看题解，也很容易发现题目并不难，只是我们没有想清楚而已。

扩展阅读

- [LeetCode Sliding Window Series Discussion](#)

搜索

大话搜索

搜索一般指在有限的状态空间中进行枚举，通过穷尽所有的可能来找到符合条件的解或者解的个数。根据搜索方式的不同，搜索算法可以分为 DFS，BFS，A*算法等。这里只介绍 DFS 和 BFS，以及发生在 DFS 上一种技巧-回溯。

搜索问题覆盖面非常广泛，并且在算法题中也占据了很高的比例。我甚至还在公开演讲中提到了 [前端算法面试中搜索类占据了很大的比重，尤其是国内公司。](#)

搜索专题中的子专题有很多，而大家所熟知的 BFS，DFS 只是其中特别基础的内容。除此之外，还有状态记录与维护，剪枝，联通分量，拓扑排序等等。这些内容，我会在这里一一给大家介绍。

另外即使仅仅考虑 DFS 和 BFS 两种基本算法，里面能玩的花样也非常多。比如 BFS 的双向搜索，比如 DFS 的前中后序，迭代加深等等。

关于搜索，其实在二叉树部分已经做了介绍了。而这里的搜索，其实就是进一步的泛化。数据结构不再局限于前面提到的数组，链表或者树。而扩展到了诸如二维数组，多叉树，图等。不过核心仍然是一样的，只不过数据结构发生了变化而已。

搜索的核心是什么？

实际上搜索题目本质就是将题目中的状态映射为图中的点，将状态间的联系映射为图中的边。根据题目信息构建状态空间，然后对状态空间进行遍历，遍历过程需要记录和维护状态，并通过剪枝和数据结构等提高搜索效率。

状态空间的数据结构不同会导致算法不同。比如对数组进行搜索，和对树，图进行搜索就不太一样。

再次强调一下，我这里讲的数组，树和图是状态空间的逻辑结构，而不是题目给的数据结构。比如题目给了一个数组，让你求数组的搜索子集。虽然题目给的线性的数据结构数组，然而实际上我们是对树这种非线性数据结构进行搜索。这是因为这道题对应的状态空间是非线性的。

对于搜索问题，我们核心关注的信息有哪些？又该如何计算呢？这也是搜索篇核心关注的。而市面上很多资料讲述的不是很详细。搜索的核心需要关注的指标有很多，比如树的深度，图的 DFS 序，图中两点间的距离等

等。这些指标都是完成高级算法必不可少的，而这些指标可以通过一些经典算法来实现。这也是为什么我一直强调一定要先学习好基础的数据结构与算法的原因。

不过要讲这些讲述完整并非容易，以至于如果完整写完可能需要花很多的时间，因此一直没有动手去写。

另外由于其他数据结构都可以看做是图的特例。因此研究透图的基本思想，就很容易将其扩展到其他数据结构上，比如树。因此我打算围绕图进行讲解，并逐步具象化到其他特殊的数据结构，比如树。

状态空间

结论先行：状态空间其实就是一个图结构，图中的节点表示状态，图中的边表示状态之前的联系，这种联系就是题目给出的各种关系。

搜索题目的状态空间通常是非线性的。比如上面提到的例子：求一个数组的子集。这里的状态空间实际上就是数组的各种组合。

对于这道题来说，其状态空间的一种可行的划分方式为：

- 长度为 1 的子集
- 长度为 2 的子集
- . . .
- 长度为 n 的子集（其中 n 为数组长度）

而如何确定上面所有的子集呢。

一种可行的方案是可以采取类似分治的方式逐一确定。

比如我们可以：

- 先确定某一种子集的第一个数是什么
- 再确定第二个数是什么
- . . .

如何确定第一个数，第二个数。. . . 呢？

暴力枚举所有可能就可以了。

这就是搜索问题的核心，其他都是辅助，所以这句话请务必记住。

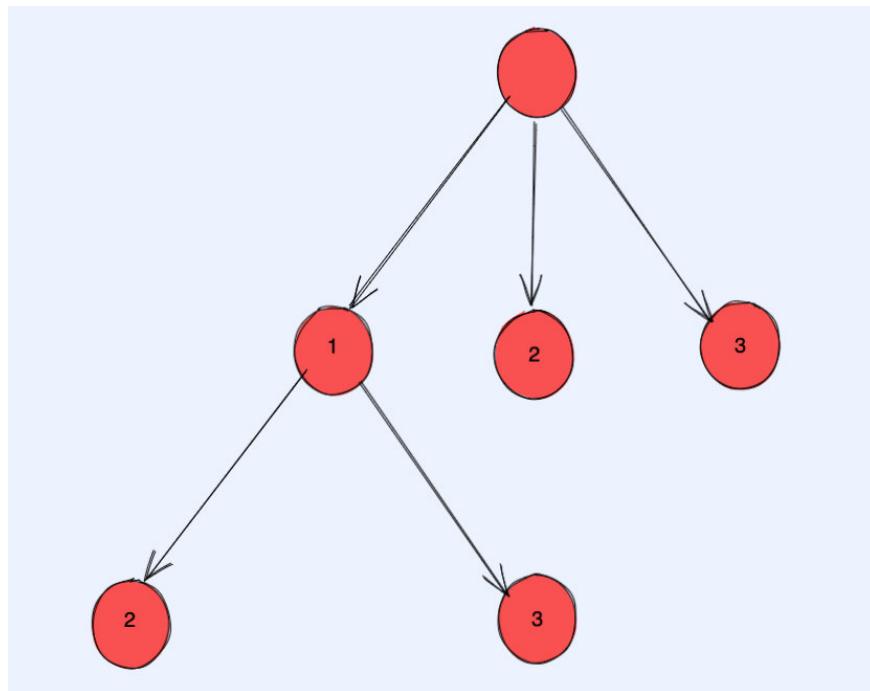
所谓的暴力枚举所有可能在这里就是尝试数组中所有可能的数字。

- 比如第一个数是什么？很明显可能是数组中任意一项。ok，我们就枚举 n 种情况。
- 第二个数呢？很明显可以是除了上面已经被选择的数之外的任意一个数。ok，我们就枚举 $n - 1$ 种情况。

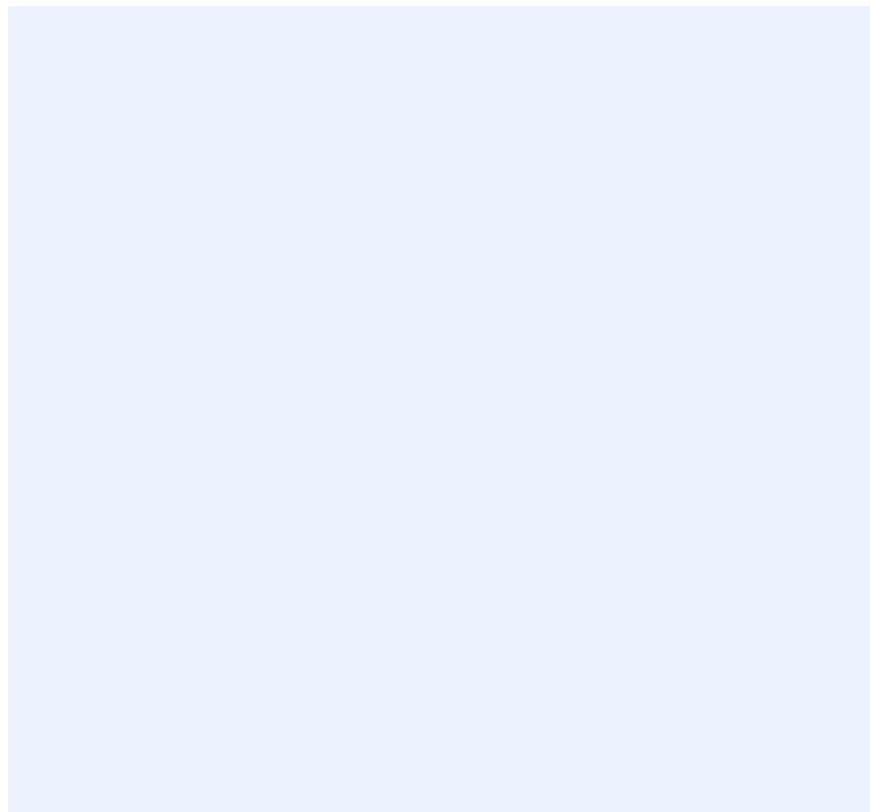
据此，你可以画出如下的决策树。

989. 数组形式的整数加法

(下图描述的是对一个长度为 3 的数组进行决策的部分过程，树节点中的数字表示索引。即确定第一个数有三个选择，确定第二个数会根据上次的选择变为剩下的两个选择)



决策过程动图演示：



一些搜索算法就是基于这个朴素的思想，本质就是模拟这个决策树。这里面其实也有很多有趣的细节，后面我们会对其进行更加详细的讲解。而现在大家只需要对解空间是什么以及如何对解空间进行遍历有一点概念就行了。后面我会继续对这个概念进行加深。

这里大家只要记住状态空间就是图，构建状态空间就是构建图。如何构建呢？当然是根据题目描述了。

DFS 和 BFS

DFS 和 BFS 是搜索的核心，贯穿搜索篇的始终，因此有必要先对其进行讲解。

DFS

DFS 的概念来自于图论，但是搜索中 DFS 和图论中 DFS 还是有一些区别，搜索中 DFS 一般指的是通过递归函数实现暴力枚举。

如果不使用递归，也可以使用栈来实现。不过本质上是类似的。

首先将题目的状态空间映射到一张图，状态就是图中的节点，状态之间的联系就是图中的边，那么 DFS 就是在这种图上进行深度优先的遍历。而 BFS 也是类似，只不过遍历的策略变为了广度优先，一层层铺开而已。所以**BFS** 和 **DFS** 只是遍历这个状态图的两种方式罢了，如何构建状态图才是关键。

本质上，对上面的图进行遍历的话会生成一颗搜索树。为了避免重复访问，我们需要记录已经访问过的节点。这些是所有的搜索算法共有的，后面不再赘述。

如果你是在树上进行遍历，是不会有环的，也自然不需要为了避免环的产生记录已经访问的节点，这是因为树本质上是一个简单无环图。

算法流程

1. 首先将根节点放入 **stack** 中。
2. 从 **stack** 中取出第一个节点，并检验它是否为目标。如果找到目标，则结束搜寻并回传结果。否则将它某一个尚未检验过的直接子节点加入 **stack** 中。
3. 重复步骤 2。
4. 如果不存在未检测过的直接子节点。将上一级节点加入 **stack** 中。重复步骤 2。
5. 重复步骤 4。
6. 若 **stack** 为空，表示整张图都检查过了——亦即图中没有欲搜寻的目标。结束搜寻并回传“找不到目标”。

这里的 **stack** 可以理解为自实现的栈，也可以理解为调用栈

算法模板

下面我们借助递归来完成 DFS。

```
const visited = []
function dfs(i) {
    if (满足特定条件) {
        // 返回结果 or 退出搜索空间
    }

    visited[i] = true // 将当前状态标为已搜索
    for (根据i能到达的下个状态j) {
        if (!visited[j]) { // 如果状态j没有被搜索过
            dfs(j)
        }
    }
}
```

常用技巧

前序遍历与后序遍历

DFS 常见的形式有前序和后序。二者的使用场景也是截然不同的。

上面讲述了搜索本质就是在状态空间进行遍历，空间中的状态可以抽象为图中的点。那么如果搜索过程中，当前点的结果需要依赖其他节点（大多数情况都会有依赖），那么遍历顺序就变得重要。

比如当前节点需要依赖其子节点的计算信息，那么使用后序遍历自底向上递推就显得必要了。而如果当前节点需要依赖其父节点的信息，那么使用先序遍历进行自顶向下的递归就不难想到。

比如下文要讲的计算树的深度。由于树的深度的递归公式为： $f(x) = f(y) + 1$ 。其中 $f(x)$ 表示节点 x 的深度，并且 x 是 y 的子节点。很明显这个递推公式的 base case 就是根节点深度为一，通过这个 base case 我们可以递推求出树中任意节点的深度。显然，使用先序遍历自顶向下的方式统计是简单而又直接的。

再比如下文要讲的计算树的子节点个数。由于树的子节点递归公式为： $f(x) = \sum_{i=0}^n f(a_i)$ 其中 x 为树中的某一个节点， a_i 为树中节点的子节点。而 base case 则是没有任何子节点(也就是叶子节点)，此时 $f(x) = 1$ 。因此我们可以利用后序遍历自底向上来完成子节点个数的统计。

关于从递推关系分析使用何种遍历方法，我在《91 天学算法》中的基础篇中的《模拟，枚举与递推》子专题中对此进行了详细的描述。91 学员可以直接进行查看。关于树的各种遍历方法，我在[树专题](#)中进行了详细的介绍。

迭代加深

迭代加深本质上是一种可行性的剪枝。关于剪枝，我会在后面的《回溯与剪枝》部分做更多的介绍。

所谓迭代加深指的是在递归树比较深的时候，通过设定递归深度阈值，超过阈值就退出的方式主动减少递归深度的优化手段。这种算法成立的前提是题目中告诉我们答案不超过 xxx ，这样我们可以将 xxx 作为递归深度阈值，这样不仅不会错过正确解，还能在极端情况下有效减少不必要的运算。

具体地，我们可以使用自顶向下的方式记录递归树的层次，和上面介绍如何计算树深度的方法是一样的。接下来在主逻辑前增加当前层次是否超过阈值的判断即可。

主代码：

```
MAX_LEVEL = 20
def dfs(root, level):
    if level > MAX_LEVEL: return
    # 主逻辑
    dfs(root, 0)
```

这种技巧在实际使用中并不常见，不过在某些时候能发挥意想不到的作用。

双向搜索

有时候问题规模很大，直接搜索会超时。此时可以考虑从起点搜索到问题规模的一半。然后将此过程中产生的状态存起来。接下来目标转化为在存储的中间状态中寻找满足条件的状态。进而达到降低时间复杂度的效果。

上面的说法可能不太容易理解。接下来通过一个例子帮助大家理解。

题目地址

<https://leetcode-cn.com/problems/closest-subsequence-sum/>

题目描述

989. 数组形式的整数加法

给你一个整数数组 `nums` 和一个目标值 `goal`。

你需要从 `nums` 中选出一个子序列，使子序列元素总和最接近 `goal`。也就是说返回 `abs(sum - goal)` 可能的 最小值。

注意，数组的子序列是通过移除原始数组中的某些元素（可能全部或无）而形成的。

示例 1：

输入: `nums = [5,-7,3,5]`, `goal = 6`

输出: `0`

解释：选择整个数组作为选出的子序列，元素和为 `6`。

子序列和与目标值相等，所以绝对差为 `0`。

示例 2：

输入: `nums = [7,-9,15,-2]`, `goal = -5`

输出: `1`

解释：选出子序列 `[7,-9,-2]`，元素和为 `-4`。

绝对差为 `abs(-4 - (-5)) = abs(1) = 1`，是可能的最小值。

示例 3：

输入: `nums = [1,2,3]`, `goal = -7`

输出: `7`

提示：

```
1 <= nums.length <= 40  
-10^7 <= nums[i] <= 10^7  
-10^9 <= goal <= 10^9
```

思路

从数据范围可以看出，这道题大概率是一个 $\$O(2^m)$ 时间复杂度的解法，其中 m 是 `nums.length` 的一半。

为什么？首先如果题目数组长度限制为小于等于 20，那么大概率是一个 $\$O(2^n)$ 的解法。

如果这个也不知道，建议看一下这篇文章

<https://lucifer.ren/blog/2020/12/21/shuati-silu3/> 另外我的刷题插件 leetcode-cheatsheet 也给出了时间复杂度速查表供大家参考。

将 40 砍半恰好就可以 AC 了。实际上，40 这个数字就是一个强有力的信号。

989. 数组形式的整数加法

回到题目中。我们可以用一个二进制位表示原数组 nums 的一个子集，这样用一个长度为 2^n 的数组就可以描述 nums 的所有子集了，这就是状态压缩。一般题目数据范围是 $<= 20$ 都应该想到。

这里 40 折半就是 20 了。

如果不熟悉状态压缩，可以看下我的这篇文章 [状压 DP 是什么？这篇题解带你入门](#)

接下来，我们使用动态规划求出所有的子集和。

令 $\text{dp}[i]$ 表示选择 i 的二进制表示的数的所有子集和。比如 i 的二进制位 1001，那么 $\text{dp}[i]$ 表示选择数组第 1 项和数组第 4 项的所有子集和，其实就是 $\text{nums}[0] + \text{nums}[3]$ 。

具体地，我们可以枚举 $1 << i$ 所有子集 j ，并依次考虑再选择数组第 i 项。

那么转移方程为： $\text{dp}[(1 << i) + j] = \text{dp}[j] + A[i]$ 。 $\text{dp}[(1 << i) + j]$ 表示选择数组第 i 项和选择情况如 j 二进制表示的子集和。

动态规划求子集和代码如下：

```
def combine_sum(A):
    n = len(A)
    dp = [0] * (1 << n)
    for i in range(n):
        for j in range(1 << i):
            dp[(1 << i) + j] = dp[j] + A[i]
    return dp
```

接下来，我们将 nums 平分为两部分，分别计算子集和：

```
n = len(nums)
c1 = combine_sum(nums[: n // 2])
c2 = combine_sum(nums[n // 2 :])
```

其中 $c1$ 就是前半部分数组的所有子集和， $c2$ 就是后半部分的所有子集和。

接下来问题转化为：在两个数组 $c1$ 和 $c2$ 中找两个数，其和最接近 $goal$ ，其中 $c1$ 和 $c2$ 分别为数组 nums 前半部分的子集和与数组 nums 后半部分的子集和。而这是一个非常经典的双指针问题，逻辑类似两数和。

只不过两数和是一个数组挑两个数，这里是两个数组分别挑一个数罢了。

这里其实只需要一个指针指向一个数组的头，另外一个指向另外一个数组的尾即可。

989. 数组形式的整数加法

代码不难写出：

```
def combine_closest(c1, c2, goal):
    # 先排序以便使用双指针
    c1.sort()
    c2.sort()
    ans = float("inf")
    i, j = 0, len(c2) - 1
    while i < len(c1) and j >= 0:
        _sum = c1[i] + c2[j]
        ans = min(ans, abs(_sum - goal))
        if _sum > goal:
            j -= 1
        elif _sum < goal:
            i += 1
        else:
            return 0
    return ans
```

上面这个代码不懂的多看看两数和。

代码

代码支持： Python3

Python3 Code:

```

class Solution:
    def minAbsDifference(self, nums: List[int], goal: int)
        def combine_sum(A):
            n = len(A)
            dp = [0] * (1 << n)
            for i in range(n):
                for j in range(1 << i):
                    dp[(1 << i) + j] = dp[j] + A[i]
            return dp

    def combine_closest(c1, c2):
        c1.sort()
        c2.sort()
        ans = float("inf")
        i, j = 0, len(c2) - 1
        while i < len(c1) and j >= 0:
            _sum = c1[i] + c2[j]
            ans = min(ans, abs(_sum - goal))
            if _sum > goal:
                j -= 1
            elif _sum < goal:
                i += 1
            else:
                return 0
        return ans

    n = len(nums)
    return combine_closest(combine_sum(nums[: n // 2]),

```

复杂度分析

令 n 为数组长度, m 为 $\frac{n}{2}$ 。

- 时间复杂度: $O(m^*2^m)$
- 空间复杂度: $O(2^m)$

相关题目推荐:

- [16. 最接近的三数之和](#)
- [1049. 最后一块石头的重量 II](#)
- [1774. 最接近目标价格的甜点成本](#)

这道题和双向搜索有什么关系呢?

回一下开头我的话: 有时候问题规模很大, 直接搜索会超时。此时可以考虑从起点搜索到问题规模的一半。然后将此过程中产生的状态存起来。接下来目标转化为在存储的中间状态中寻找满足条件的状态。进而达到降低时间复杂度的效果。

对应这道题，我们如果直接暴力搜索。那就是枚举所有子集和，然后找到和 goal 最接近的，思路简单直接。可是这样会超时，那么就搜索到一半，然后将状态存起来（对应这道题就是存到了 dp 数组）。接下来问题转化为两个 dp 数组的运算。该算法，本质上是将位于指数位的常数项挪动到了系数位。这是一种常见的双向搜索，我姑且称为 DFS 的双向搜索。目的是为了和后面的 BFS 双向搜索进行区分。

BFS

BFS 也是图论中算法的一种。不同于 DFS，BFS 采用横向搜索的方式，从初始状态一层层展开直到目标状态，在数据结构上通常采用队列结构。

具体地，我们不断从队头取出状态，然后将此状态对应的决策产生的所有新的状态推入队尾，重复以上过程直至队列为空即可。

注意这里有两个关键点：

1. 将此状态对应的决策。实际上这句话指的就是状态空间中的图的边，不管是 DFS 和 BFS 边都是确定的。也就是说不管是 DFS 还是 BFS 这个决策都是一样的。不同的是什么？不同的是进行决策的方向不同。
2. 所有新的状态推入队尾。上面说 BFS 和 DFS 是进行决策的方向不同。这就可以通过这个动作体现出来。由于直接将所有状态空间中的当前点的邻边放到队尾。由队列的先进先出的特性，当前点的邻边访问完成之前是不会继续向外扩展的。这一点大家可以和 DFS 进行对比。

最简单的 BFS 每次扩展新的状态就增加一步，通过这样一步步逼近答案。其实也就等价于在一个权值为 1 的图上进行 BFS。由于队列的单调性和二值性，当第一次取出目标状态时就是最少的步数。基于这个特性，BFS 适合求解一些最少操作的题目。

关于单调性和二值性，我会在后面的 BFS 和 DFS 的对比那块进行讲解。

前面 DFS 部分提到了不管是什么搜索都需要记录和维护状态，其中一个就是节点访问状态以防止环的产生。而 BFS 中我们常常用来求点的最短距离。值得注意的是，有时候我们会使用一个哈希表 dist 来记录从源点到图中其他点的距离。这个 dist 也可以充当防止环产生的功能，这是因为第一次到达一个点后再次到达此点的距离一定比第一次到达大，利用这点就可知道是否是第一次访问了。

算法流程

1. 首先将根节点放入队列中。
2. 从队列中取出第一个节点，并检验它是否为目标。
 - 如果找到目标，则结束搜索并回传结果。

989. 数组形式的整数加法

- 否则将它所有尚未检验过的直接子节点加入队列中。
3. 若队列为空，表示整张图都检查过了——亦即图中没有欲搜索的目标。结束搜索并回传“找不到目标”。
 4. 重复步骤 2。

算法模板

```
const visited = []
function bfs() {
    let q = new Queue()
    q.push(初始状态)
    while(q.length) {
        let i = q.pop()
        if (visited[i]) continue
        for (i的可抵达状态j) {
            if (j 合法) {
                q.push(j)
            }
        }
    }
    // 找到所有合法解
}
```

常用技巧

双向搜索

题目地址([126. 单词接龙 II](#))

<https://leetcode-cn.com/problems/word-ladder-ii/>

题目描述

按字典 wordList 完成从单词 beginWord 到单词 endWord 转化，一个表

每对相邻的单词之间仅有单个字母不同。

转换过程中的每个单词 s_i ($1 \leq i \leq k$) 必须是字典 wordList 中的单词
 $s_k == endWord$

给你两个单词 beginWord 和 endWord，以及一个字典 wordList。请你

示例 1：

输入: beginWord = "hit", endWord = "cog", wordList = ["hot",

输出: [["hit","hot","dot","dog","cog"], ["hit","hot","lot","l

解释: 存在 2 种最短的转换序列:

"hit" -> "hot" -> "dot" -> "dog" -> "cog"

"hit" -> "hot" -> "lot" -> "log" -> "cog"

示例 2：

输入: beginWord = "hit", endWord = "cog", wordList = ["hot",

输出: []

解释: endWord "cog" 不在字典 wordList 中，所以不存在符合要求的转换

提示：

```
1 <= beginWord.length <= 7
endWord.length == beginWord.length
1 <= wordList.length <= 5000
wordList[i].length == beginWord.length
beginWord、endWord 和 wordList[i] 由小写英文字母组成
beginWord != endWord
wordList 中的所有单词 互不相同
```

思路

这道题就是我们日常玩的成语接龙游戏。即让你从 beginWord 开始，接龙的 endWord。让你找到最短的接龙方式，如果有多个，则全部返回。

不同于成语接龙的字首接字尾。这种接龙需要的是下一个单词和上一个单词仅有一个单词不同。

我们可以对问题进行抽象：即构建一个大小为 n 的图，图中的每一个点表示一个单词，我们的目标是找到一条从节点 `beginWord` 到节点 `endWord` 的一条最短路径。

这是一个不折不扣的图上 BFS 的题目。套用上面的解题模板可以轻松解决。唯一需要注意的是如何构建图。更进一步说就是如何构建边。

由题目信息的转换规则：每对相邻的单词之间仅有单个字母不同。不难知道，如果两个单词的仅有单个字母不同，就说明两者之间有一条边。

明白了这一点，我们就可以构建邻接矩阵了。

核心代码：

```
neighbors = collections.defaultdict(list)
for word in wordList:
    for i in range(len(word)):
        neighbors[word[:i] + "*" + word[i + 1 :]].append(word)
```

构建好了图。BFS 剩下要做的就是明确起点和终点就好了。对于这道题来说，起点是 `beginWord`，终点是 `endWord`。

那我们就可以将 `beginWord` 入队。不断在图上做 BFS，直到第一次遇到 `endWord` 就好了。

套用上面的 BFS 模板，不难写出如下代码：

这里我用了 `cost` 而不是 `visited`，目的是为了让大家见识多种写法。
下面的优化解法会使用 `visited` 来记录。

```

class Solution:
    def findLadders(self, beginWord: str, endWord: str, wordList: List[str]) -> List[List[str]]:
        cost = collections.defaultdict(lambda: float("inf"))
        cost[beginWord] = 0
        neighbors = collections.defaultdict(list)
        ans = []

        for word in wordList:
            for i in range(len(word)):
                neighbors[word[:i] + "*" + word[i + 1 :]].append(word)

        q = collections.deque([[beginWord]])

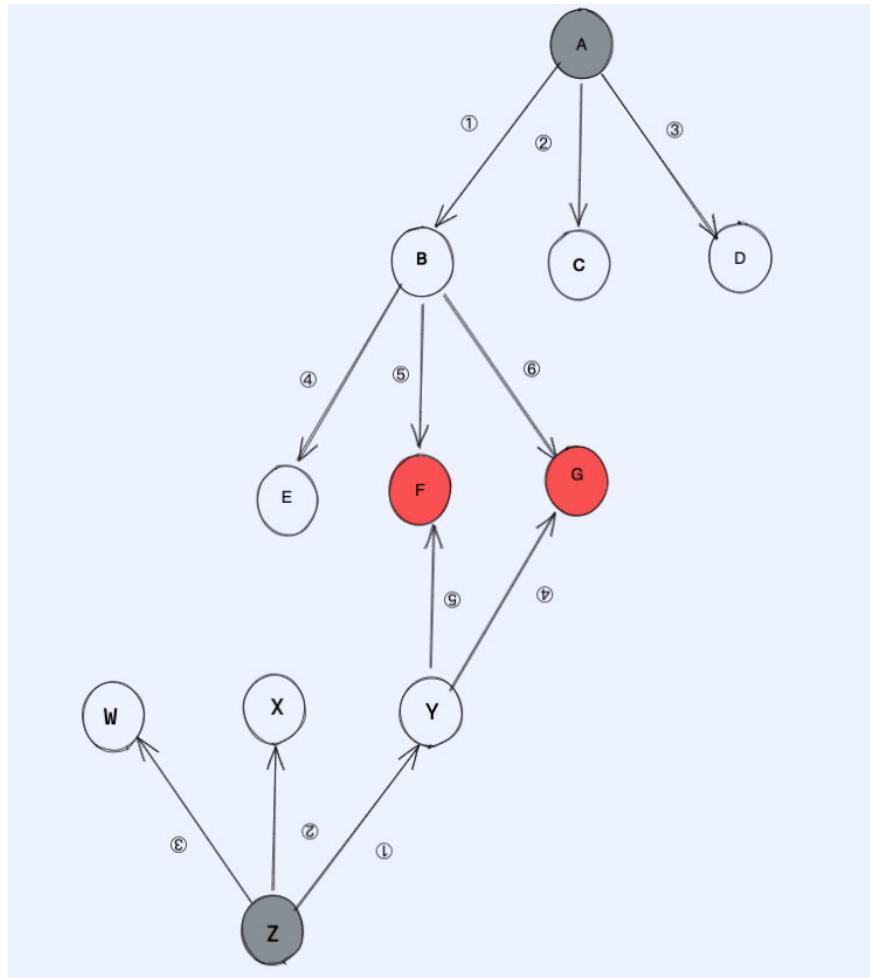
        while q:
            path = q.popleft()
            cur = path[-1]
            if cur == endWord:
                ans.append(path.copy())
            else:
                for i in range(len(cur)):
                    for neighbor in neighbors[cur[:i] + "*"]:
                        if cost[cur] + 1 <= cost[neighbor]:
                            q.append(path + [neighbor])
                            cost[neighbor] = cost[cur] + 1
        return ans

```

当终点可以逆向搜索的时候，我们也可以尝试双向 BFS。更本质一点就是：**如果你构建的状态空间的边是双向的，那么就可以使用双向 BFS。**

和 DFS 的双向搜索思想是类似的。我们只需要使用两个队列分别存储中起点和终点进行扩展的节点（我称其为起点集与终点集）即可。当起点和终点在某一时刻交汇了，说明找到了一个从起点到终点的路径，其路径长度就是两个队列扩展的路径长度和。

以上就是双向搜索的大体思路。用图来表示就是这样的：



如上图，我们从起点和重点（A 和 Z）分别开始搜索，如果起点的扩展状态和终点的扩展状态重叠（本质上就是队列中的元素重叠了），那么我们就知道了一个从节点到终点的最短路径。

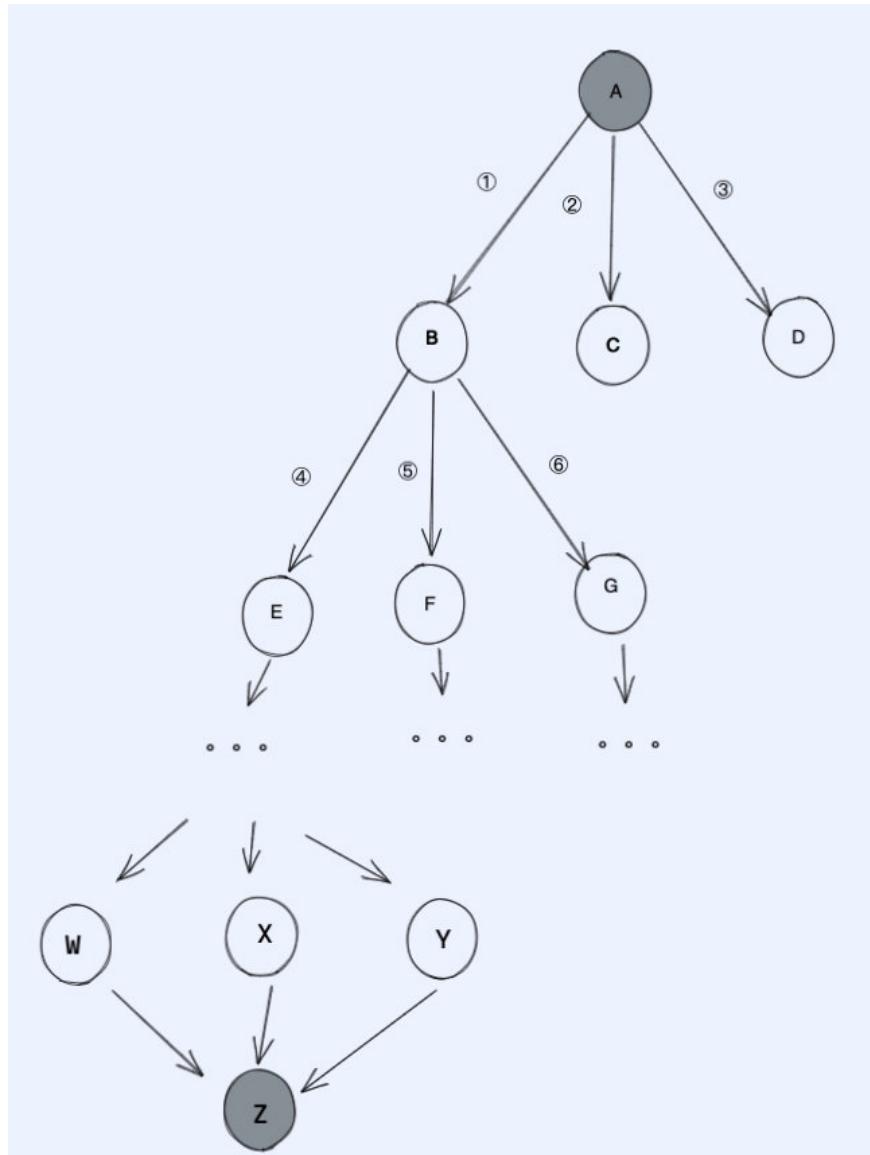
动图演示：

看到这里有必要暂停一下插几句话。

为什么双向搜索就快了？什么情况都会更快么？那为什么不都用双向搜索？有哪些使用条件？

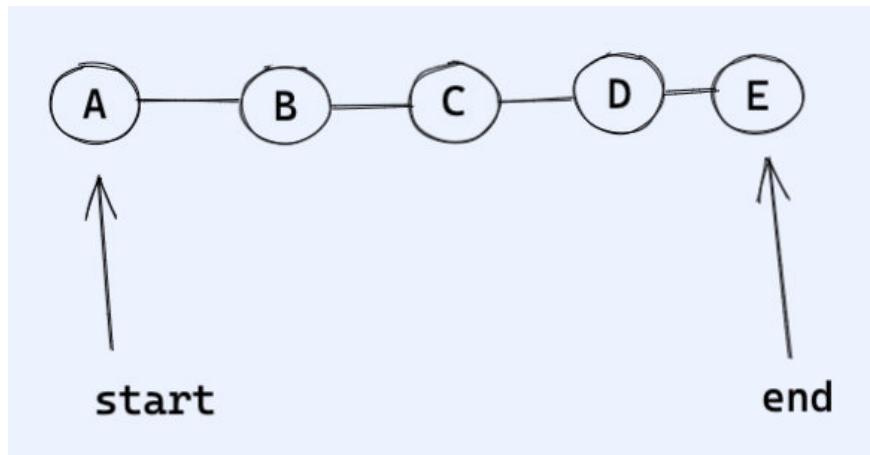
我们一个个回答。

- 为什么双向搜索更快了？通过上面的图我们发现通常刚开始的时候边比较少，队列中的数据也比较少。而随着搜索的进行，**搜索树越来越大，队列中的节点随之增多**。和上面双向搜索类似，这种增长速度很多情况下是指数级别的，而双向搜索可以**将指数的常系数移动到多项式系数**。如果不使用双向搜索那么搜索树大概是这样的：



可以看出搜索树大了很多，以至于很多点我都画不下，只好用”。。。来表示。

- 什么情况下更快？相比于单向搜索，双向搜索通常更快。当然也有例外，举个极端的例子，假如从起点到终点只要一条路径，那么无论使用单向搜索还是双向搜索结果都是一样。



如图使用单向搜索还是双向搜索都是一样的。

- 为什么不都用双向搜索？实际上做题中，我建议大家尽量使用单向搜索，因为写起来更节点，并且大多数都可以通过所有的测试用例。除非你预估到可能会超时或者提交后发现超时的时候再尝试使用双向搜索。
- 有哪些使用条件？正如前面所说：“终点可以逆向搜索的时候，可以尝试双向 BFS。更本质一点就是：如果你构建的状态空间的边是双向的，那么就可以使用双向 BFS。”

让我们继续回到这道题。为了能够判断两者是否交汇，我们可以使用两个 `hashSet` 分别存储起点集合终点集。当一个节点既出现起点集又出现在终点集，那就说明出现了交汇。

为了节省代码量以及空间消耗，我没有使用上面的队列，而是直接使用了哈希表来代替队列。这种做法可行的关键仍然是上面提到的队列的二值性和单调性。

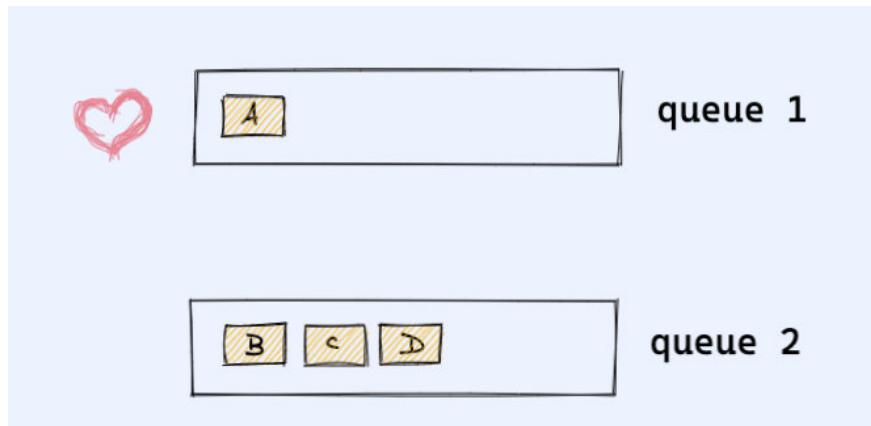
由于新一轮的出队列前，队列中的权值都是相同的。因此从左到右遍历或者从右到左遍历，甚至是任意顺序遍历都是无所谓的。（很多题都无所谓）因此使用哈希表而不是队列也是可以的。这点需要引起大家的注意。希望大家对 BFS 的本质有更深的理解。

那我们是不是不需要队列，就用哈希表，哈希集合啥的存就行了？非也！我会在双端队列部分为大家揭晓。

这道题的具体算法：

- 定义两个队列：`q1` 和 `q2`，分别从起点和终点进行搜索。
- 构建邻接矩阵
- 每次都尝试从 `q1` 和 `q2` 中的较小的进行扩展。这样可以达到剪枝的效果。

989. 数组形式的整数加法



- 如果 q1 和 q2 交汇了，则将两者的路径拼接起来即可。

代码

- 语言支持: Python3

Python3 Code:

```

class Solution:
    def findLadders(self, beginWord: str, endWord: str, wordList: List[str]) -> List[List[str]]:
        # 剪枝 1
        if endWord not in wordList: return []
        ans = []
        visited = set()
        q1, q2 = {beginWord: [[beginWord]]}, {endWord: [[endWord]]}
        steps = 0
        # 预处理, 空间换时间
        neighbors = collections.defaultdict(list)
        for word in wordList:
            for i in range(len(word)):
                neighbors[word[:i] + "*" + word[i + 1 :]].append(word)
        while q1:
            # 剪枝 2
            if len(q1) > len(q2):
                q1, q2 = q2, q1
            nxt = collections.defaultdict(list)
            for _ in range(len(q1)):
                word, paths = q1.popitem()
                visited.add(word)
                for i in range(len(word)):
                    for neighbor in neighbors[word[:i] + "*", word[i + 1 :]]:
                        if neighbor in q2:
                            # 从 beginWord 扩展过来的
                            if paths[0][0] == beginWord:
                                ans += [path1 + path2[::-1]]
                            # 从 endWord 扩展过来的
                            else:
                                ans += [path2 + path1[::-1]]
                            if neighbor in wordList and neighbor not in visited:
                                nxt[neighbor] += [path + [neighbor]]
            steps += 1
            # 剪枝 3
            if ans and steps + 2 > len(ans[0]):
                break
            q1 = nxt
        return ans

```

总结

我想通过这道题给大家传递的知识点很多。分别是：

- 队列不一定非得是常规的队列，也可以是哈希表等。不过某些情况必须是双端队列，这个等会讲双端队列给大家讲。
- 双向 BFS 是只适合双向图。也就是说从终点也往前推。
- 双向 BFS 从较少状态的一端进行扩展可以起到剪枝的效果

- visitd 和 dist/cost 都可以起到记录点访问情况以防止环的产生的作用。不过 dist 的作用更多，相应空间占用也更大。

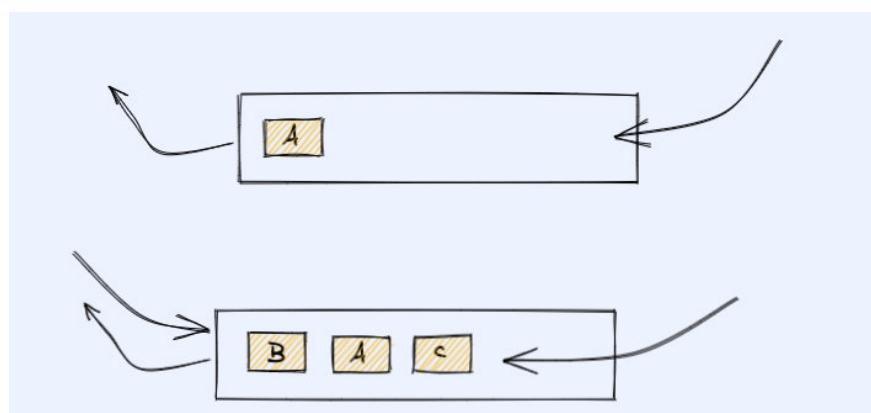
双端队列

上面提到了 BFS 本质上可以看做是在一个边权值为 1 的图上进行遍历。实际上，我们可以进行一个简单的扩展。如果图中边权值不全是 1，而是 0 和 1 呢？这样其实我们用到双端队列。

双端队列可以在头部和尾部同时进行插入和删除，而普通的队列仅允许在头部删除，在尾部插入。

使用双端队列，当每次取出一个状态的时候。如果我们可以无代价的进行转移，那么就可以将其直接放在队头，否则放在队尾。由前面讲的队列的单调性和二值性不难得出算法的正确性。而如果状态转移是有代价的，那么就将其放到队尾即可。这也是很多语言提供的内置数据结构是双端队列，而不是队列的原因之一。

如下图：



上面的队列是普通的队列。而下面的双端队列，可以看出我们在队头插队了一个 B。

动图演示：

思考：如果图对应的权值不出 0 和 1，而是任意正整数呢？

前面我们提到了是不是不需要队列，就用哈希表，哈希集合啥的存就行了？这里为大家揭秘。不可以的。因为哈希表无法处理这里的权值为 0 的情况。

DFS 和 BFS 的对比

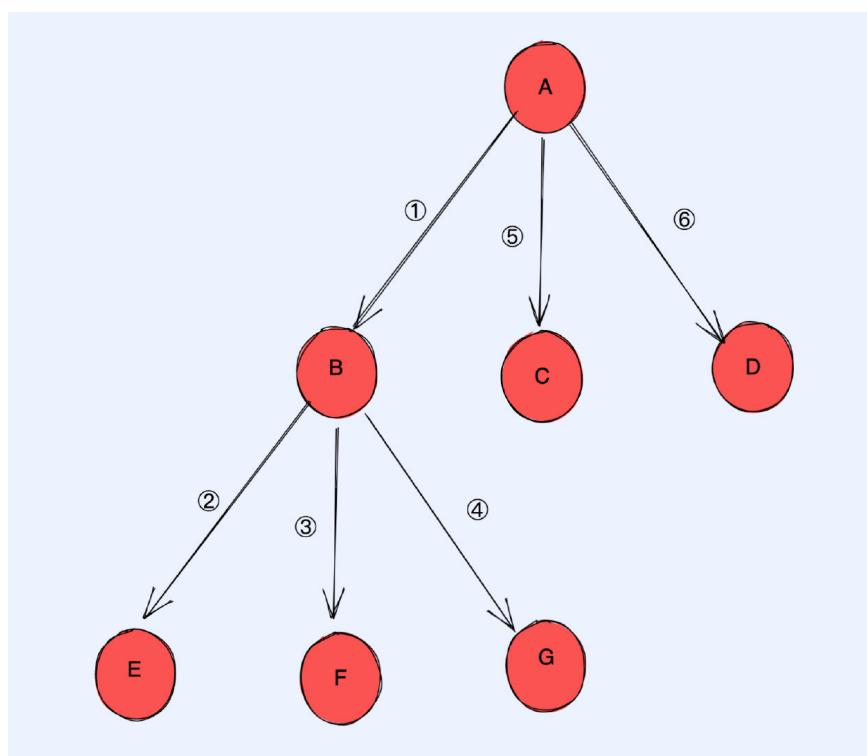
BFS 和 DFS 分别处理什么样的问题？两者究竟有什么样的区别？这些都值得我们认真研究。

简单来说，不管是 DFS 还是 BFS 都是对题目对应的状态空间进行搜索。

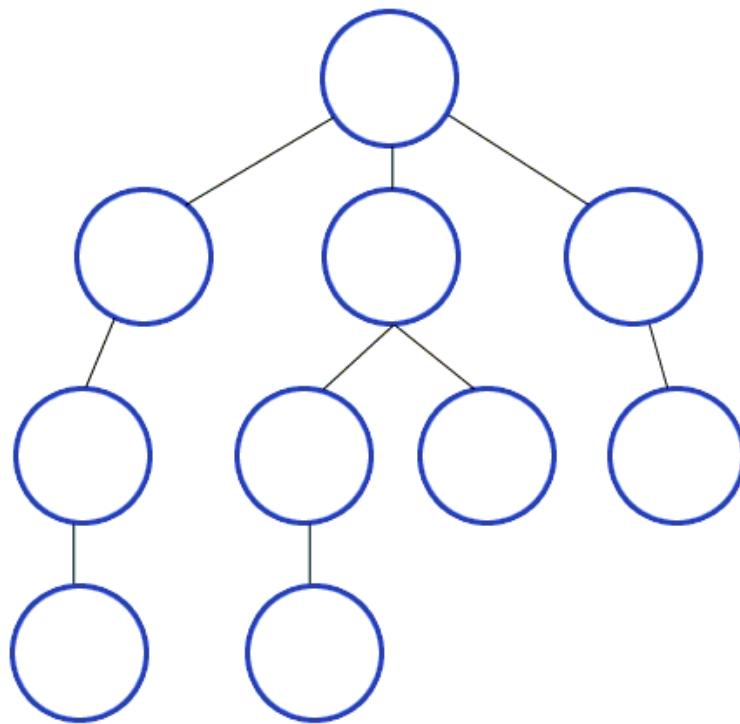
具体来说，二者区别在于：

- DFS 在分叉点会任选一条深入进入，遇到终点则返回，再次返回到分叉口后尝试下一个选择。基于此，我们可以在路径上记录一些数据。由此也可以衍生出很多有趣的东西。

如下图，我们遍历到 A，有三个选择。此时我们可以任意选择一条，比如选择了 B，程序会继续往下进行选择分支 2, 3。。。

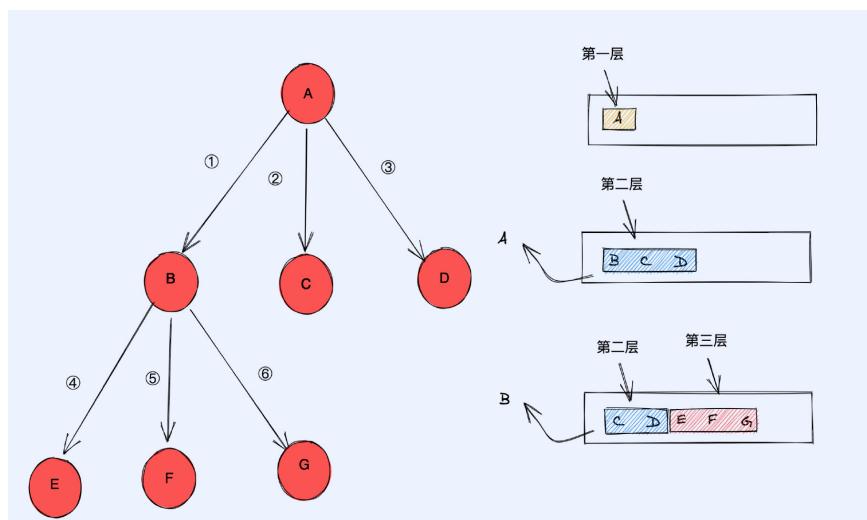


如下动图演示了一个典型的 DFS 流程。后面的章节，我们会给大家带来更复杂的图上 DFS。



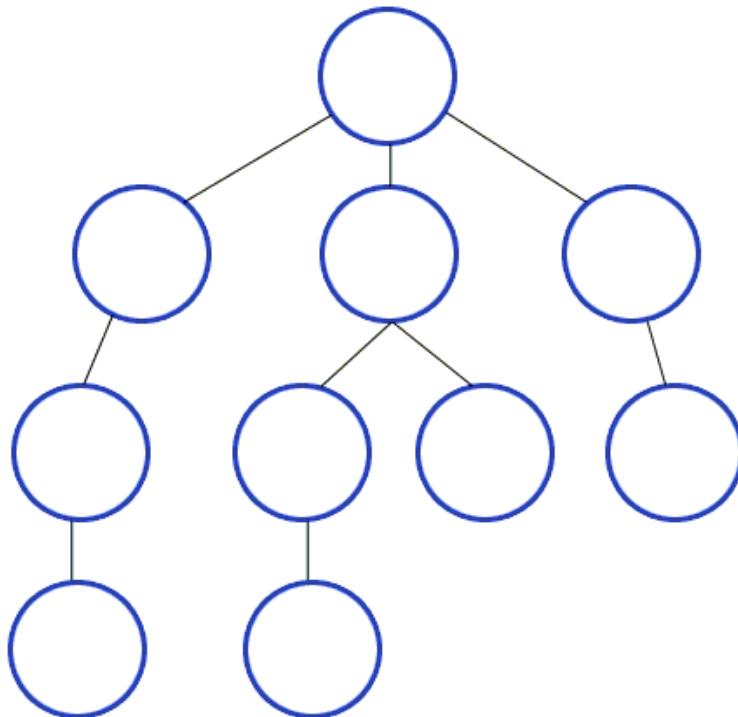
- BFS 在分叉点会选择搜索的路径各尝试一次。使用队列来存储待处理的元素时，队列中最多只会有两层的元素，这是队列的二值性。另外队列还满足单调性，即相同层的元素在一起。基于这个特点有很多有趣的优化。

如下图，广度优先遍历会将搜索的选择全部选择一遍会才会进入到下一层。和上面一样，我给大家标注了程序执行的一种可能的顺序。



可以发现，和我上面说的一样。右侧的队列始终最多有两层的节点，并且相同层的总在一起，换句话说队列的元素在层上满足单调性。

如下动图演示了一个典型的 BFS 流程。后面的章节，我们会给大家带来更复杂的图上 BFS。



回溯

回溯是 DFS 中的一种技巧。回溯法采用 [试错](#) 的思想，它尝试分步的去解决一个问题。在分步解决问题的过程中，当它通过尝试发现现有的分步答案不能得到有效的正确的解答的时候，它将取消上一步甚至是上几步的计算，再通过其它的可能的分步解答再次尝试寻找问题的答案。

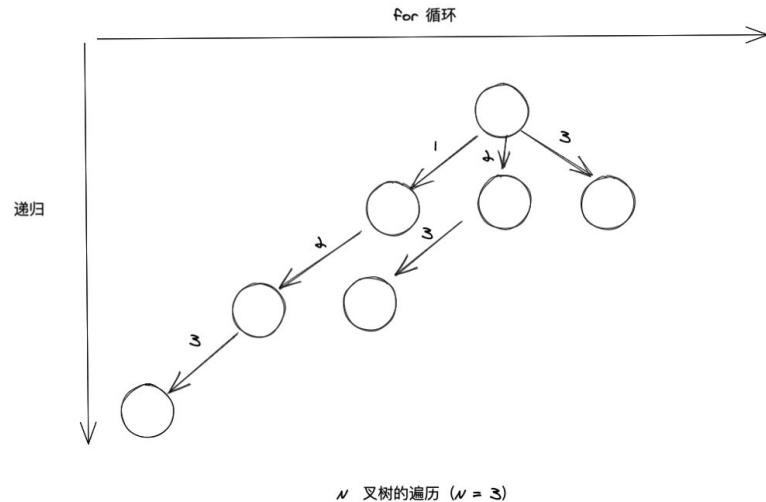
通俗上讲，回溯是一种走不通就回头的算法。

回溯的本质是穷举所有可能，尽管有时候可以通过剪枝去除一些根本不可能是答案的分支，但是从本质上讲，仍然是一种暴力枚举算法。

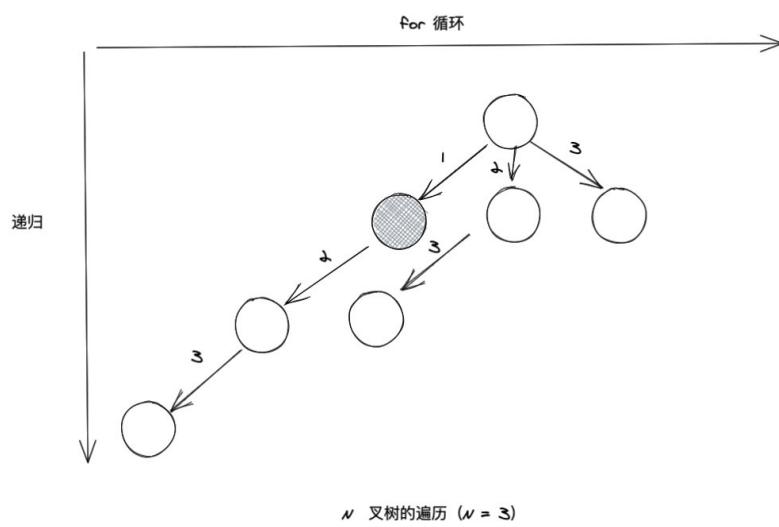
回溯法可以抽象为树形结构，并且是一颗高度有限的树（N 叉树）。回溯法解决的都是在集合中查找子集，集合的大小就是树的叉树，递归的深度，都构成的树的高度。

以求数组 [1,2,3] 的子集为例：

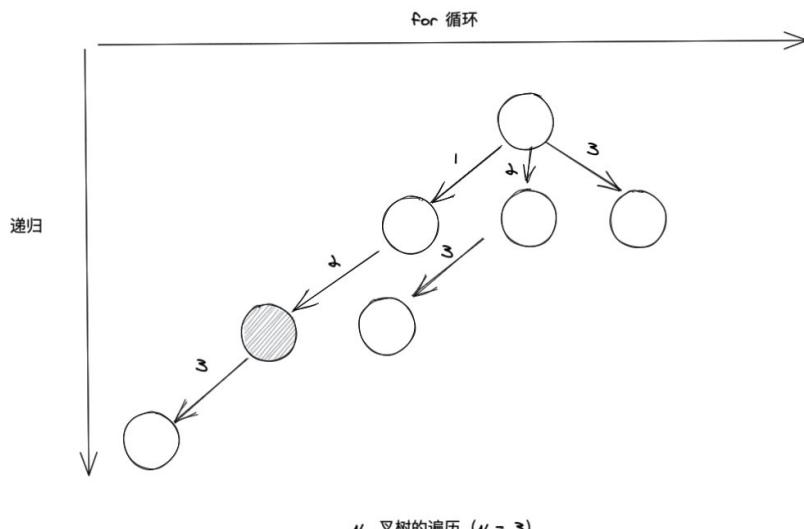
989. 数组形式的整数加法



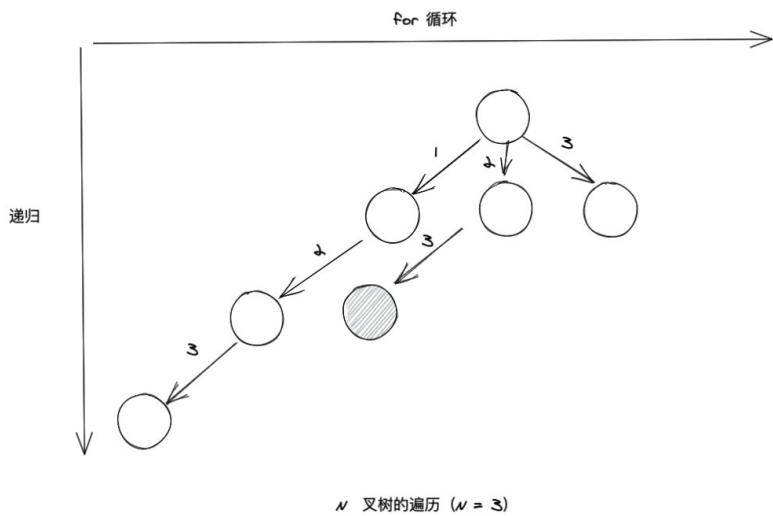
以上图来说， 我们会在每一个节点进行加入到结果集这一次操作。



对于上面的灰色节点， 加入结果集就是 [1]。



这个加入结果集就是 [1,2]。



这个加入结果集就是 [2,3]，以此类推。一共有六个子集，分别是 [1], [1,2], [1,2,3], [2], [2,3] 和 [3]。

而对于全排列问题则会在叶子节点加入到结果集，不过这都是细节问题。
掌握了思想之后，大家再去学习细节就会事半功倍。

下面我们来看下具体代码怎么写。

算法流程

1. 构造空间树。
2. 进行遍历。
3. 如遇到边界条件，即不再向下搜索，转而搜索另一条链。
4. 达到目标条件，输出结果。

算法模板

伪代码：

```

const visited = []
function dfs(i) {
    if (满足特定条件) {
        // 返回结果 or 退出搜索空间
    }

    visited[i] = true // 将当前状态标为已搜索
    dosomething(i) // 对i做一些操作
    for (根据i能到达的下个状态j) {
        if (!visited[j]) { // 如果状态j没有被搜索过
            dfs(j)
        }
    }
    undo(i) // 恢复i
}

```

经典题目

- [39. 组合总和](#)
- [40. 组合总和 II](#)
- [46. 全排列](#)
- [47. 全排列 II](#)
- [52. N 皇后 II](#)
- [78. 子集](#)
- [90. 子集 II](#)
- [113. 路径总和 II](#)
- [131. 分割回文串](#)
- [1255. 得分最高的单词集合](#)

剪枝

回溯题目的另外一个考点是剪枝，通过恰当地剪枝，可以有效减少时间，比如我通过剪枝操作将**石子游戏 V**的时间从 900 多 ms 优化到了 500 多 ms。

剪枝在每道题的技巧都是不一样的，不过一个简单的原则就是**避免根本不可能是答案的递归**。

笛卡尔积

一些回溯的题目，我们仍然也可以采用笛卡尔积的方式，将结果保存在返回值而不是路径中，这样就避免了回溯状态，并且由于结果在返回值中，因此可以使用记忆化递归，进而优化为动态规划形式。

参考题目：

- [140. 单词拆分 II](#)
- [816. 模糊坐标](#)

这类问题不同于子集和全排列，其组合是有规律的，我们可以使用笛卡尔积公式，将两个或更多子集联合起来。

常用的指标与统计方法

搜索本质是对状态空间进行遍历求解。而状态空间可以抽象为图，而对于图有很多有趣的指标。那么对于搜索来说，常用的指标有哪些呢？这里给大家介绍。

这些指标是基本的算法。当你实际应用（或者做题）的时候，往往是将几种基础算法进行组合或者稍微改动，因此掌握这些基础算法显得尤为重要。

1. 树的深度与子树大小

计算树的深度和子树大小其实我在前面《前序遍历与后序遍历》部分已经讲过了，只不过当时只是讲了递推公式而已。

这里我直接给出代码。

```
visited = set()
d = collections.defaultdict(int)
def dfs(x):
    visited.add(x) # 将 x 标记为已访问
    # 枚举 x 的所有邻居，并尝试进入
    for neibor in neibors[x]:
        if neibor in visited: continue
        d[neibor] = d[x] + 1 # 递推公式
        dfs(neibor)
}
```

经过上面的处理，`d` 存储的就是所有的节点对应的深度映射。

很多困难的题都是基于这个简单的算法完成的。比如让你求解一个点到所有其他点的距离和如何求？再比如求树的重心如何求？

关于树的重心是什么，大家可以搜索一下

2. 图的 DFS 序

树的 `dfs` 序指的是`dfs`过程中访问节点的顺序，包括进入和离开，也就是说一个节点会在 `dfs` 序中出现两次。

为了方便描述，dfs 序用 D 表示。

基于此，有一个有意思的地方是：假设一个 x 节点在 dfs 序中出现位置为 l 和 r，那么 dfs 序 D 的子序列 [l, r] 就是以 x 为根的子序列。

这有什么用？其实基于这样，我们就将逻辑上的非线性结构（树）映射到了线性结构（数组）上，这样我们就可以在数组上对子树进行一些统计。

3. 图的拓扑序

图的拓扑序其实是 BFS 的一个功能。

给一个无环图，如果有一个序列满足以下条件，那么 A 就是这个他的拓扑序。

- A 包含图中所有节点
- 对于图中的每一个边 (x, y) ， x 在 A 中的位置都在 y 之前（当然也可以反过来）

求解 A 的过程就是拓扑排序。

本质上，图的拓扑序就是使用队列来存储图中入度为 0 的点，并将其剔除。由于这些点被剔除了，因此可能会使得图中其他的点入度减少。如果其他的点入度被减少到 0，那么就继续入队重复上面的过程。

对应前文讲的队列的二值性和单调性，这里的队列中只会有度为 0 的节点。因此这个队列其实是单值性，是一种更特殊的 BFS。

代码：

```
# items 为图的所有点
# indegree 为图的入度信息
# neighbors 则是每个点的邻居信息
def tp_sort(self, items, indegree, neighbors):
    q = collections.deque([])
    ans = []
    for item in items:
        if indegree[item] == 0:
            q.append(item)
    while q:
        cur = q.popleft()
        ans.append(cur)

        for neighbor in neighbors[cur]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                q.append(neighbor)

    return ans
```

正如前面的算法描述，这里的代码结构是非常简单清晰的。

拓扑排序经常被处理一些有依赖关系的问题。比如排课系统，而课程之前有一定的前置课程，你必须修完前置课程才能学习本可能，问你如何安排才能学完所有的课程等等。

题目推荐：

- [1203. 项目管理](#)
- [1494. 并行课程 II](#)

4. 图的联通分量

从图中任意一点出发，我们可以访问到的所有点可以构成一个联通分量。因此对图中每一个点进行一次这样的遍历就可以得到若干个联通分量（也可能只有一个，比如说树）

形象地来说，你可以把图看成是几个分离的水域，图中的点水域的一部分。那么你要如何求水域的个数（不是点的个数）呢？

- 我们可以往某一个点倒墨水，让墨水进行扩散。如果此时所有水域都被染色了，那么水域个数（联通分量个数）就是1。
- 否则继续往下一个点倒墨水，重复上面过程直到所有点都被倒了墨水或者所有的水域都被染色。

同时为了防止一个点被多次（因为这是没有意义的），我们需要记录一下每个点被倒墨水的情况。

代码：

```
visited = set()
count = 0
def dfs(x):
    visited.add(x)
    for neibor in neibors:
        if neibor in visited: continue
        dfs(neibor)
for x in range(n):
    if x not in visited:
        dfs(x)
        count += 1
```

如上代码中的 count 就记录了图中联通分量的个数。力扣中很多题都用到了这个技巧，比如[小岛专题](#)。

并查集也特别适合处理联通分量个数的计算，大家不妨结合起来理解。

总结

以上就是《搜索篇（上）》的所有内容了。总结一下搜索篇的解题思路：

- 根据题目信息构建状态空间（图）。
- 对图进行遍历（BFS 或者 DFS）
- 记录和维护状态。（比如 `visited` 维护访问情况，队列和栈维护状态的决策方向等等）

BFS 是面，每一层的节点同时进行搜索。而 DFS 是线，纵向一个一个解决。一般来说找最短路径的时候使用 BFS，其他时候还是 DFS 写起来比较方便。

BFS 一般需要借助于队列这种数据结构，而 BFS 则可以借助递归或者栈来进行。如果需要求最短距离，推荐大家使用 BFS，这样可大大剪枝，当然最差情况还是和 DFS 复杂度一致。对于不是求最短距离的情况大家随意，使用 BFS 和 DFS 均可。。

战略上，我们可以将二叉树遍历，多叉树遍历都看成图的特殊情况，因此建议大家从图的遍历入手来学习。战术上，我推荐大家从数组，链表遍历开始，进而到二叉树，多叉树遍历，最后学习图的遍历。

我们花了大量的篇幅对 BFS 和 DFS 进行了详细的讲解，包括两个的对比。

核心点需要大家注意：

- DFS 通常都是有递推关系的，而递归关系就是图的边。根据递归关系大家可以选择使用前序遍历或者后序遍历。
- BFS 由于其单调性，因此适合求解最短距离问题。
- . . .

双向搜索的本质是将复杂度的常数项从一个影响较大的位置（比如指数位）移到了影响较小的位置（比如系数位）。

回溯的本质就是暴力枚举所有可能。要注意的是，由于回溯通常结果集都记录在回溯树的路径上，因此如果不进行撤销操作，则可能在回溯后状态不正确导致结果有差异，因此需要在递归到底部往上冒泡的时候进行撤销状态。**回溯我建议大家画决策图图进行学习。**

上面提到的搜索都是单向搜索。还有一种比较少见的搜索方式是双向搜索。即分别从起点和中点进行搜索，这样时间复杂度大概会降低一半左右。有时候暴力搜索刚好超时的时候，可以考虑使用双向搜索。比如如下题目：

```
Given a set of n integers where n <= 40. Each of them is at
```

Example:

```
Input : set[] = {45, 34, 4, 12, 5, 2} and S = 42
```

```
Output : 41
```

```
Maximum possible subset sum is 41 which can be  
obtained by summing 34, 5 and 2.
```

```
Input : Set[] = {3, 34, 4, 12, 5, 2} and S = 10
```

```
Output : 10
```

```
Maximum possible subset sum is 10 which can be  
obtained by summing 2, 3 and 5.
```

使用暴力回溯的时间复杂度为 $O(2^n)$ 。我们也可以：

- 将数组一份为二，并对每一部分分别进行暴力回溯，时间复杂度为 $O(2^{\{n/2\}})$
- 不妨设两个暴力回溯的解集为 S_1 和 S_2 。迭代 S_1 ，对于 S_1 的每一项 x 在 S_2 中寻找 y ，使得 $x + y \leq S$ 。
- 为了使上一步更有效率，可以对 S_2 进行一次排序，进而使用二分找到 y 。

这种算法的时间复杂度是 $O(2^{\{n/2\}} * n)$

这种算法面试的考察频率相对比较低，大家根据自己的实际情况选择性掌握即可。

搜索篇知识点比较密集，希望大家多多总结复习。

扩展阅读

- [小岛问题](#)
- [深度优先遍历](#)
- [回溯算法](#)

动态规划

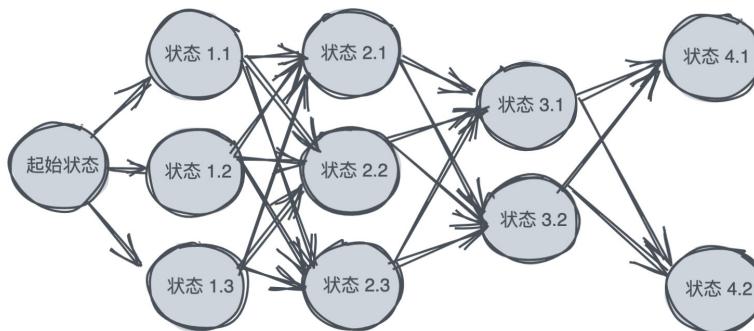
动态规划是一个从其他行业借鉴过来的词语。

它的大概意思先将一件事情分成若干阶段，然后通过阶段之间的转移达到目标。由于转移的方向通常是多个，因此这个时候就需要决策选择具体哪一个转移方向。

动态规划所要解决的事情通常是完成一个具体的目标，而这个目标往往是最优解。并且：

1. 阶段之间可以进行转移，这叫做动态。
2. 达到一个可行解(目标阶段)需要不断地转移，那如何转移才能达到最优解？这叫规划。

每个阶段抽象为状态（用圆圈来表示），状态之间可能会发生转化（用箭头表示）。可以画出类似如下的图：



那我们应该做出如何的决策序列才能使得结果最优？换句话说就是每一个状态应该如何选择到下一个具体状态，并最终到达目标状态。这就是动态规划研究的问题。

每次决策实际上不会考虑之后的决策，而只会考虑之前的状态。形象点来说，其实是走一步看一步这种短视思维。为什么这种短视可以来求解最优解呢？那是因为：

1. 我们将所有可能的转移全部模拟了一遍，最后挑了一个最优解。
2. 无后向性（这个我们后面再说，先卖个关子）

而如果你没有模拟所有可能，而直接走了一条最优解，那就是贪心算法了。

没错，动态规划刚开始就是来求最优解的。只不过有的时候顺便可以求总方案数等其他东西，这其实是动态规划的副产物。

好了，我们把动态规划拆成两部分分别进行解释，或许你大概知道了动态规划是一个什么样的东西。但是这对你做题并没有帮助。那算法上的动态规划究竟是个啥呢？

在算法上，动态规划和查表的递归（也称记忆化递归）有很多相似的地方。我建议大家先从记忆化递归开始学习。本文也先从记忆化递归开始，逐步讲解到动态规划。

记忆化递归

那么什么是递归？什么是查表（记忆化）？让我们慢慢来看。

什么是递归？

递归是指在函数中调用函数自身的方法。

有意义的递归通常会把问题分解成规模缩小的同类子问题，当子问题缩写到寻常的时候，我们可以直接知道它的解。然后通过建立递归函数之间的联系（转移）即可解决原问题。

是不是和分治有点像？分治指的是将问题一分为多，然后将多个解合并为一。而这里并不是这个意思。

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），也就是说递归会逐步缩小规模到寻常。

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(x):
    return x + f(x - 1)
```

上面的代码除非外界干预，否则会永远执行下去，不会停止。

因此更多的情况应该是：

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

使用递归通常可以使代码短小，有时候也更可读。算法中使用递归可以很简单地完成一些用循环不太容易实现的功能，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。

递归在函数式编程中地位很高。纯粹的函数式编程中没有循环，只有递归。

实际上，除了在编码上通过函数调用自身实现递归。我们也可以定义递归的数据结构。比如大家所熟知的树，链表等都是递归的数据结构。

```
Node {
    value: any; // 当前节点的值
    children: Array<Node>; // 指向其儿子
}
```

如上代码就是一个多叉树的定义形式，可以看出 `children` 就是 `Node` 的集合类，这就是一种递归的数据结构。

不仅仅是普通的递归函数

本文中所提到的记忆化递归中的递归函数实际上指的是特殊的递归函数，即在普通的递归函数上满足以下几个条件：

1. 递归函数不依赖外部变量
2. 递归函数不改变外部变量

满足这两个条件有什么用呢？这是因为我们需要函数给定参数，其返回值也是确定的。这样我们才能记忆化。关于记忆化，我们后面再讲。

如果大家了解函数式编程，实际上这里的递归其实严格来说是函数式编程中的函数。如果不了解也没关系，这里的递归函数其实就是数学中的函数。

我们来回顾一下数学中的函数：

在一个变化过程中，假设有两个变量 `x`、`y`，如果对于任意一个 `x` 都有唯一确定

而本文所讲的所有递归都是指的这种数学中的函数。

比如上面的递归函数：

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

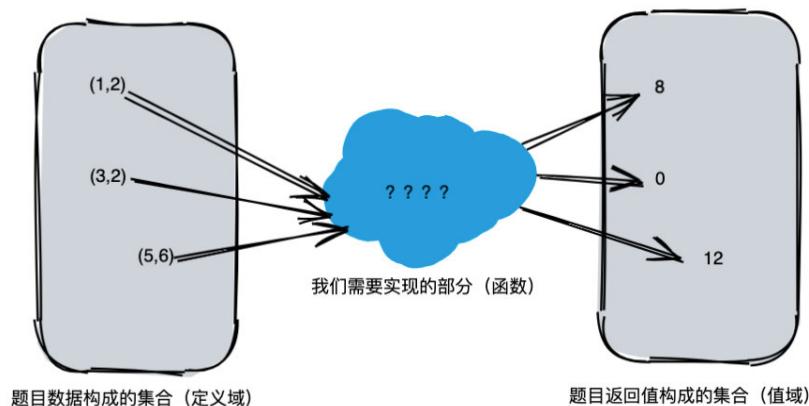
- x 就是自变量， x 的所有可能的返回值构成的集合就是定义域。
- $f(x)$ 就是函数。
- $f(x)$ 的所有可能的返回值构成的集合就是值域。

自变量也可以有多个，对应递归函数的参数可以有多个，比如 $f(x_1, x_2, x_3)$ 。

通过函数来描述问题，并通过函数的调用关系来描述问题间的关系就是记忆化递归的核心内容。

每一个动态规划问题，实际上都可以抽象为一个数学上的函数。这个函数的自变量集合就是题目的所有取值，值域就是题目要求的答案的所有可能。我们的目标其实就是填充这个函数的内容，使得给定自变量 x ，能够唯一映射到一个值 y 。（当然自变量可能有多个，对应递归函数参数可能有多个）

解决动态规划问题可以看成是填充函数这个黑盒，使得定义域中的数并正确地映射到值域。



递归并不是算法，它是和迭代对应的一种编程方法。只不过，我们通常借助递归去分解问题而已。比如我们定义一个递归函数 $f(n)$ ，用 $f(n)$ 来描述问题。就和使用普通动态规划 $f[n]$ 描述问题是一样的，这里的 f 是 dp 数组。

什么是记忆化？

为了大家能够更好地对本节内容进行理解，我们通过一个例子来切入：

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？

思路：

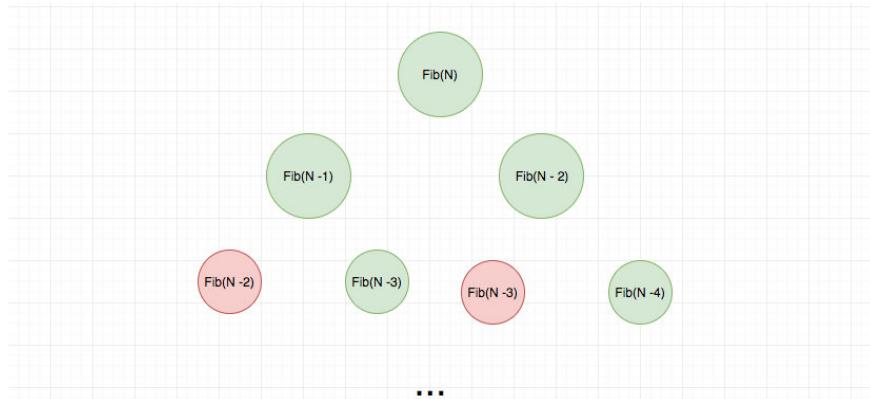
由于第 n 级台阶一定是从 $n - 1$ 级台阶或者 $n - 2$ 级台阶来的，因此到第 n 级台阶的数目就是 到第 $n - 1$ 级台阶的数目加上到第 $n - 2$ 级台阶的数目。

递归代码：

989. 数组形式的整数加法

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

我们用一个递归树来直观感受以下（每一个圆圈表示一个子问题）：



红色表示重复的计算。即 $Fib(N-2)$ 和 $Fib(N-3)$ 都被计算了两次，实际上计算一次就够了。比如第一次计算出了 $Fib(N-2)$ 的值，那么下次再次需要计算 $Fib(N-2)$ 的时候，可以直接将上次计算的结果返回。之所以可以这么做的原因正是前文提到的我们的递归函数是数学中的函数，也就是说参数一定，那么返回值也一定不会变，因此下次如果碰到相同的参数，我们就可以将上次计算过的值直接返回，而不必重新计算。这样节省的时间就等于重叠子问题的个数。

以这道题来说，本来需要计算 2^n 次，而如果使用了记忆化，只需要计算 n 次，就是这么神奇。

代码上，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。

我们使用记忆化来改造上面的代码：

```
memo = {}  
def climbStairs(n):  
    if n == 1: return 1  
    if n == 2: return 2  
    if n in memo: return memo[n]  
    ans = func(n - 1) + func(n-2)  
    memo[n] = ans  
    return ans  
climbStairs(10)
```

这里我使用了一个名为 `memo` 的哈希表来存储递归函数的返回值，其中 `key` 为参数，`value` 为递归函数的返回值。



哈希表

`key` 的形式为 (x, y) ，表示的是一个元祖。通常动态规划的参数有多少个，我们就可以使用元祖的方式来记忆化。或者也可采取多维数组的形式。对于上图来说，就可使用二维数组来表示。

大家可以通过删除和添加代码中的 `memo` 来感受一下记忆化的作用。

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

递归中如果存在重复计算（我们称重叠子问题，下文会讲到），那就是使用记忆化递归（或动态规划）解题的强有力信号之一。可以看出动态规划的核心就是使用记忆化的手段消除重复子问题的计算，如果这种重复子问题的规模是指数或者更高规模，那么记忆化递归（或动态规划）带来的收益会非常大。

为了消除这种重复计算，我们可使用查表的方式。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。下文要讲的动态规划中 DP 数组其实和这里“记录表”的作用是一样的。

如果你刚开始接触递归，建议大家先去练习一下递归再往后看。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。

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

讲了这么多递归和记忆化，终于到了我们的主角登场了。

动态规划的基本概念

我们先来学习动态规划最重要的两个概念：最优子结构和无后效性。

其中：

- 无后效性决定了是否可使用动态规划来解决。
- 最优子结构决定了具体如何解决。

最优子结构

动态规划常常适用于有重叠子问题和最优子结构性质的问题。前面讲了重叠子问题，那么最优子结构是什么？这是我从维基百科找的定义：

如果问题的最优解所包含的子问题的解也是最优的，我们就称该问题具有最优子：

举个例子：如果考试中的分数定义为 f，那么这个问题就可以被分解为语文，数学，英语等子问题。显然子问题最优的时候，总分这个大的问题的解也是最优的。

再比如 01 背包问题：定义 $f(\text{weights}, \text{values}, \text{capacity})$ 。如果我们想要求 $f([1,2,3], [2,2,4], 10)$ 的最优解。从左到右依次考虑是否将每一件物品加入背包。我们可以将其划分为如下子问题：

- 不将第三件物品装进背包（即仅考虑前两件），也就是 $f([1,2], [2,2], 10)$
- 和 将第三件物品装进背包，也就是 $f([1,2,3], [2,2,4], 10)$ （即仅考虑前两件的情况下装满容量为 $10 - 3 = 7$ 的背包）等价于 $4 + f([1,2], [2,2], 7)$ ，其中 4 为第三件物品的价值，7 为装下第三件物品后剩余可用空间，由于我们仅考虑前三件，因此前两件必须装满 $10 - 3 = 7$ 才行。

显然这两个问题还是复杂，我们需要进一步拆解。不过，这里不是讲如何拆解的。

原问题 $f([1,2,3], [2,2,4], 10)$ 等于以上两个子问题的最大值。只有两个子问题是**最优的时候整体才是最优的**，这是因为子问题之间不会相互影响。

无后效性

即子问题的解一旦确定，就不再改变，不受在这之后、包含它的更大的问题的求解决策影响。

继续以上面两个例子来说。

- 数学考得高不能影响英语（现实其实可能影响，比如时间一定，投入英语多，其他科目就少了）。
- 背包问题中 $f([1,2,3], [2,2,4], 10)$ 选择是否拿第三件物品，不应该影响是否拿前面的物品。比如题目规定了拿了第三件物品之后，第二件物品的价值就会变低或变高）。这种情况就不满足无后向性。

动态规划三要素

状态定义

动态规划的中心点是什么？如果让我说的话，那就是**定义状态**。

动态规划解题的第一步就是定义状态。定义好了状态，就可以画出递归树，聚焦最优子结构写转移方程就好了，因此我才说状态定义是动态规划的核心，动态规划问题的状态确实不容易看出。

但是一旦你能把状态定义好了，那就可以顺藤摸瓜画出递归树，画出递归树之后就聚焦最优子结构就行了。但是能够画出递归树的前提是：对问题进行划分，专业点来说就是**定义状态**。那怎么才能定义出状态呢？

好在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 s_1 以 i 结尾， s_2 以 j 结尾的。

989. 数组形式的整数加法

也就是说状态的定义通常有不同的套路，大家可以在做题的过程中进行学习和总结。但是这种套路非常多，那怎么搞定呢？

说实话，只能多练习，在练习的过程中总结套路。具体的套路参考后面的**动态规划的题型**部分内容。之后大家就可以针对不同的题型，去思考大概的状态定义方向。

两个例子

关于状态定义，真的非常重要，以至于我将其列为动态规划的核心。因此我觉得有必要举几个例子来进行说明。我直接从力扣的**动态规划专题**中抽取前两道给大家讲讲。

The screenshot shows the LeetCode platform's 'Dynamic Programming' category. At the top, there is a navigation bar with a graduation cap icon, the title '动态规划', and tabs for '题库' (Problem库) and '讨论交流' (Discussion). Below the navigation bar, there are filters for '完成度' (Completion), '状态' (Status), '难度' (Difficulty), and '出现频率' (Frequency). A progress bar indicates a completion rate of 162 / 272. The main area displays a table with two rows of data:

| 状态 | 题目 | 通过率 | 难度 | 出现频率 |
|----|-------------|-------|----|------|
| ✓ | 5. 最长回文子串 | 33.9% | 中等 | 🔒 |
| ✓ | 10. 正则表达式匹配 | 31.1% | 困难 | 🔒 |

第一道题：《5. 最长回文子串》难度中等

989. 数组形式的整数加法

给你一个字符串 s , 找到 s 中最长的回文子串。

示例 1:

输入: $s = "babad"$

输出: "bab"

解释: "aba" 同样是符合题意的答案。

示例 2:

输入: $s = "cbbd"$

输出: "bb"

示例 3:

输入: $s = "a"$

输出: "a"

示例 4:

输入: $s = "ac"$

输出: "a"

提示:

$1 \leq s.length \leq 1000$

s 仅由数字和英文字母 (大写和/或小写) 组成

这道题入参是一个字符串, 那我们要将其转化为规模更小的子问题, 那无疑就是字符串变得更短的问题, 临界条件也应该是空字符串或者一个字符这样。

因此:

- 一种定义状态的方式就是 $f(s1)$, 含义是字符串 $s1$ 的最长回文子串, 其中 $s1$ 就是题目中的字符串 s 的子串, 那么答案就是 $f(s)$ 。
- 由于规模更小指的是字符串变得更短, 而描述字符串我们也可以用两个变量来描述, 这样实际上还省去了开辟字符串的开销。两个变量可以是起点索引 + 子串长度, 也可以是终点索引 + 子串长度, 也可以是起点坐标 + 终点坐标。随你喜欢, 这里我就用起点坐标 + 终点坐标。那么状态定义就是 $f(start, end)$, 含义是子串 $s[start:end+1]$ 的最长回文子串, 那么答案就是 $f(0, len(s) - 1)$

$s[start:end+1]$ 指的是包含 $s[start]$, 而不包含 $s[end+1]$ 的连续子串。

989. 数组形式的整数加法

这无疑是一种定义状态的方式，但是一旦我们这样去定义就会发现：状态转移方程会变得难以确定（实际上很多动态规划都有这个问题，比如最长上升子序列问题）。那究竟如何定义状态呢？我会在稍后的状态转移方程继续完成这道题。我们先来看下一道题。

第二道题：《10. 正则表达式匹配》难度困难

给你一个字符串 s 和一个字符规律 p ，请你来实现一个支持 ‘.’ 和 ‘*’ 的正则表达式匹配。

‘.’ 匹配任意单个字符

‘*’ 匹配零个或多个前面的那个元素

所谓匹配，是要涵盖 整个 字符串 s 的，而不是部分字符串。

示例 1：

输入: $s = "aa"$ $p = "a"$

输出: false

解释: "a" 无法匹配 "aa" 整个字符串。

示例 2：

输入: $s = "aa"$ $p = "a*"$

输出: true

解释: 因为 '*' 代表可以匹配零个或多个前面的那个元素，在这里前面的元

示例 3：

输入: $s = "ab"$ $p = ".*"$

输出: true

解释: ".*" 表示可匹配零个或多个 ('*') 任意字符 ('.')。

示例 4：

输入: $s = "aab"$ $p = "c*a*b"$

输出: true

解释: 因为 '*' 表示零个或多个，这里 'c' 为 0 个，'a' 被重复一次。因

示例 5：

输入: $s = "mississippi"$ $p = "mis*is*p*."$

输出: false

提示：

$0 \leq s.length \leq 20$

$0 \leq p.length \leq 30$

s 可能为空，且只包含从 a-z 的小写字母。

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

保证每次出现字符 '*' 时，前面都匹配到有效的字符

这道题入参有两个，一个是 s ，一个是 p 。沿用上面的思路，我们有两种定义状态的方式。

- 一种定义状态的方式就是 $f(s1, p1)$ ，含义是 $p1$ 是否可匹配字符串 $s1$ ，其中 $s1$ 就是题目中的字符串 s 的子串， $p1$ 就是题目中的字符串 p 的子串，那么答案就是 $f(s, p)$ 。
- 另一种是 $f(s_start, s_end, p_start, p_end)$ ，含义是子串 $p1[p_start:p_end+1]$ 是否可以匹配字符串 $s[s_start:s_end+1]$ ，那么答案就是 $f(0, \text{len}(s) - 1, 0, \text{len}(p) - 1)$

而这道题实际上我们也可采用更简单的状态定义方式，不过基本思路都是差不多的。我仍旧卖个关子，后面讲转移方程再揭晓。

搞定了状态定义，你会发现时间空间复杂度都变得很明显了。这也是为啥我反复强调状态定义是动态规划的核心。

时间空间复杂度怎么个明显法了呢？

首先空间复杂度，我刚才说了动态规划其实就是查表的暴力法，因此动态规划的空间复杂度打底就是表的大小。再直白一点就是上面的哈希表 memo 的大小。而 memo 的大小基本就是状态的个数。状态个数是多少呢？这不就取决于你状态怎么定义了么？比如上面的 $f(s1, p1)$ 。状态的多少是多少呢？很明显就是每个参数的取值范围大小的笛卡尔积。 $s1$ 的所有可能取值有 $\text{len}(s)$ 种， $p1$ 的所有可能有 $\text{len}(p)$ 种，那么总的状态大小就是 $\text{len}(s) * \text{len}(p)$ 。那空间复杂度是 $\mathcal{O}(m * n)$ ，其中 m 和 n 分别为 s 和 p 的大小。

我说空间复杂度打底是状态个数，这里暂时先不考虑状态压缩的情况。

其次是时间复杂度。时间复杂度就比较难说了。但是由于我们无论如何都要枚举所有状态，因此时间复杂度打底就是状态总数。以上面的状态定义方式，时间复杂度打底就是 $\mathcal{O}(m * n)$ 。

如果你枚举每一个状态都需要和 s 的每一个字符计算一下，那时间复杂度就是 $\mathcal{O}(m^2 * n)$ 。

以上面的爬楼梯的例子来说，我们定义状态 $f(n)$ 表示到达第 n 级台阶的方法数，那么状态总数就是 n ，空间复杂度和时间复杂度打底就是 n 了。
(仍然不考虑滚动数组优化)

再举个例子：[62. 不同路径](#)

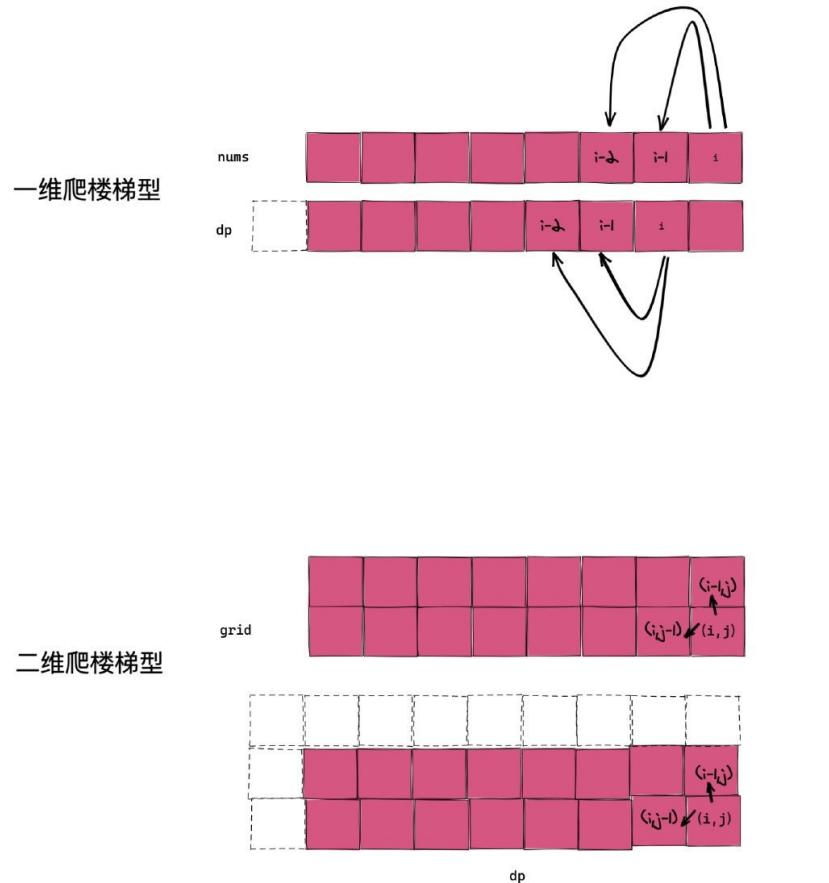
一个机器人位于一个 $m \times n$ 网格的左上角（起始点在下图中标记为“Start”）

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

问总共有多少条不同的路径？

这道题是和上面的爬楼梯很像，只不过从一维变成了二维，我把它叫做二维爬楼梯，类似的换皮题还很多，大家慢慢体会。

这道题我定义状态为 $f(i, j)$ 表示机器人到达点 (i, j) 的总的路径数。那么状态总数就是 i 和 j 的取值的笛卡尔积，也就是 $m * n$ 。



总的来说，动态规划的空间和时间复杂度打底就是状态的个数，而状态的个数通常是参数的笛卡尔积，这是由动态规划的无后向性决定的。

临界条件是比较最容易的

当你定义好了状态，剩下就三件事了：

1. 临界条件
2. 状态转移方程
3. 枚举状态

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】

可以看出记忆化递归和动态规划是多么的相似。

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 $f(1), f(2), \dots$ 就是各个独立的状态。

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

动态规划中当前阶段的状态往往是上一阶段状态和上一阶段决策的结果。
这里有两个关键字，分别是：

- 上一阶段状态
- 上一阶段决策

也就是说，如果给定了第 k 阶段的状态 $s[k]$ 以及决策 $\text{choice}(s[k])$ ，则第 $k+1$ 阶段的状态 $s[k+1]$ 也就完全确定，用公式表示就是： $s[k] + \text{choice}(s[k]) \rightarrow s[k+1]$ ，这就是状态转移方程。需要注意的是 choice 可能有多个，因此每个阶段的状态 $s[k+1]$ 也会有多个。

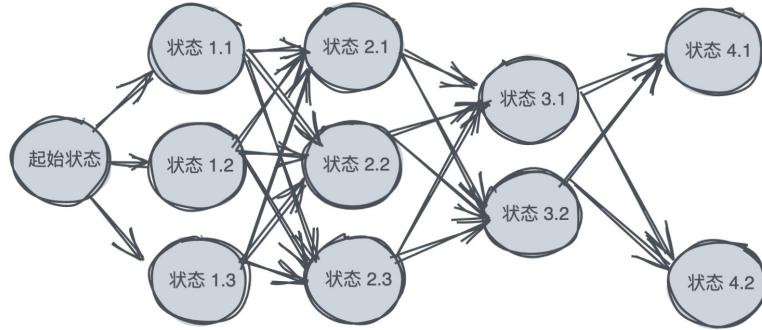
继续以上面的爬楼梯问题来说，爬楼梯问题由于上第 n 级台阶一定是从 $n - 1$ 或者 $n - 2$ 来的，因此上第 n 级台阶的数目就是 上 $n - 1$ 级台阶的数目加上 $n - 2$ 级台阶的数目。

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。

比如我们定义了状态方程，据此我们定义初始状态和目标状态。然后聚焦最优子结构，思考每一个状态究竟如何进行扩展使得离目标状态越来越近。

如下图所示：



理论差不多先这样，接下来来几个实战消化一下。

ok，接下来是解密环节。上面两道题我们都没有讲转移方程，我们在这里补上。

第一道题：《5. 最长回文子串》难度中等。上面我们的两种状态定义都不好，而我可以在上面的基础上稍微变动一点就可以使得转移方程变得非常好写。这个技巧在很多动态题目都有体现，比如最长上升子序列等，**需要大家掌握**。

以上面提到的 $f(start, end)$ 来说，含义是子串 $s[start:end+1]$ 的最长回文子串。表示方式我们不变，只是将含义变成子串 $s[start:end+1]$ 的最长回文子串，且必须包含 **start** 和 **end**。经过这样的定义，实际上我们也没有必要定义 $f(start, end)$ 的返回值是长度了，而仅仅是布尔值就行了。如果返回 `true`，则最长回文子串就是 $end - start + 1$ ，否则就是 0。

这样转移方程就可以写为：

$$f(i, j) = f(i+1, j-1) \text{ and } s[i] == s[j]$$

第二道题：《10. 正则表达式匹配》难度困难。

以我们分析的 $f(s_start, s_end, p_start, p_end)$ 来说，含义是子串 $p1[p_start:p_end+1]$ 是否可以匹配字符串 $s[s_start:s_end+1]$ 。

实际上，我们可以定义更简单的方式，那就是 $f(s_end, p_end)$ ，含义是子串 $p1[:p_end+1]$ 是否可以匹配字符串 $s[:s_end+1]$ 。也就是说固定起点为索引 0，这同样也是一个很常见的技巧，请务必掌握。

这样转移方程就可以写为：

1. if $p[j]$ 是小写字母，是否匹配取决于 $s[i]$ 是否等于 $p[j]$:

$$f(i, j) = \begin{cases} f(i-1, j-1) & s[i] == p[j] \\ \text{false} & s[i] != p[j] \end{cases}$$

989. 数组形式的整数加法

1. if $p[j] == \text{''}'$, 一定可匹配:

$$f(i, j) = f(i-1, j-1)$$

1. if $p[j] == \text{''}*'$, 表示 p 可以匹配 s 第 $j-1$ 个字符匹配任意次:

$$f(i, j) = \begin{cases} f(i-1, j) & \text{匹配1次以上} \\ f(i, j-2) & \text{匹配0次} \end{cases}$$

相信你能分析到这里，写出代码就不是难事了。具体代码可参考我的[力扣题解仓库](#)，咱就不在这里讲了。

注意到了么？所有的状态转移方程我都使用了上述的数学公式来描述。没错，所有的转移方程都可以这样描述。我建议大家做每一道动态规划题目都写出这样的公式，起初你可能觉得很烦麻烦。不过相信我，你坚持下去，会发现自己慢慢变强大。就好像我强烈建议你每一道题都分析好复杂度一样。动态规划不仅要搞懂转移方程，还要自己像我那样完整地用数学公式写出来。

是不是觉得状态转移方程写起来麻烦？这里我给大家介绍一个小技巧，那就是使用 latex, latex 语法可以方便地写出这样的公式。另外西法还贴心地写了一键生成动态规划转移方程公式的功能，帮助大家以最快速度生成公诉处。插件地址：<https://leetcode-pp.github.io/leetcode-cheat/?tab=solution-template>

如何使用？

编程语言: python3

常用公式 (点击可复制): $\sum \times \div \% \approx \sqrt{x}$ 点这个

时间复杂度: O(1) O(\sqrt{n}) O(logn) O(n) O(nlogn) O(n^2) O(2^n) O($n!$)

空间复杂度: O(1) O(\sqrt{n}) O(logn) O(n) O(nlogn) O(n^2) O(2^n) O($n!$)

lucifer 专属: 是否是 lucifer

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。

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。

如何枚举状态

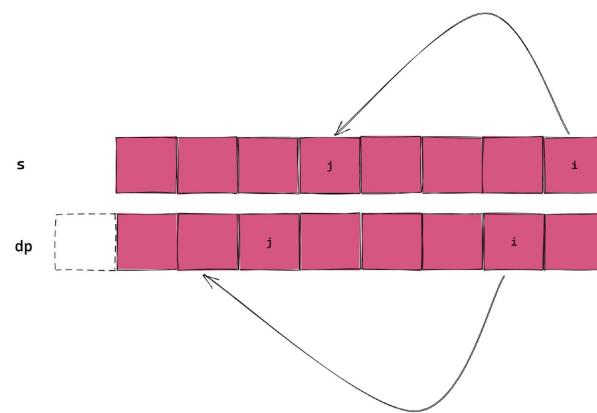
前面说了如何枚举状态，才能不重不漏是枚举状态的关键所在。

989. 数组形式的整数加法

- 如果是一维状态，那么我们使用一层循环可以搞定。

```
for i in range(1, n + 1):
    pass
```

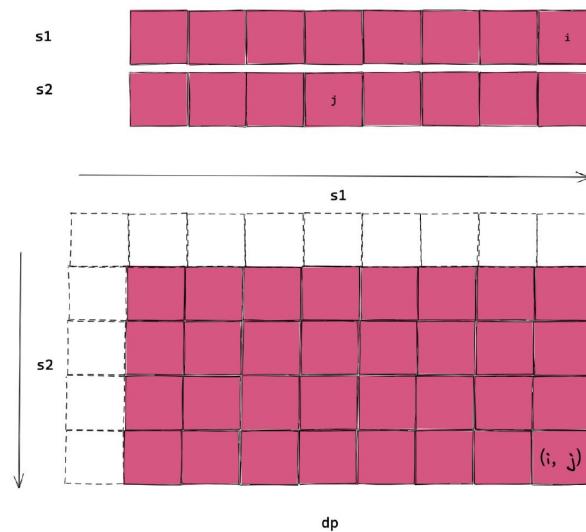
单字符串型



- 如果是两维状态，那么我们使用两层循环可以搞定。

```
for i in range(1, m + 1):
    for j in range(1, n + 1):
        pass
```

双字符串型



• . . .

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 s_1 以 i 结尾， s_2 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 VS 记忆化递归

上面我们用记忆化递归的问题巧妙地解决了爬楼梯问题。那么动态规划是怎么解决这个问题呢？

答案也是“查表”，我们平常写的 dp table 就是表，其实这个 dp table 和上面的 memo 没啥差别。

而一般我们写的 dp table，数组的索引通常对应记忆化递归的函数参数，值对应递归函数的返回值。

看起来两者似乎没有任何思想上的差异，区别的仅仅是写法？？没错。不过这种写法上的差异还会带来一些别的相关差异，这点我们之后再讲。

如果上面的爬楼梯问题，使用动态规划，代码是怎么样的呢？我们来看下：

```
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];
}
```

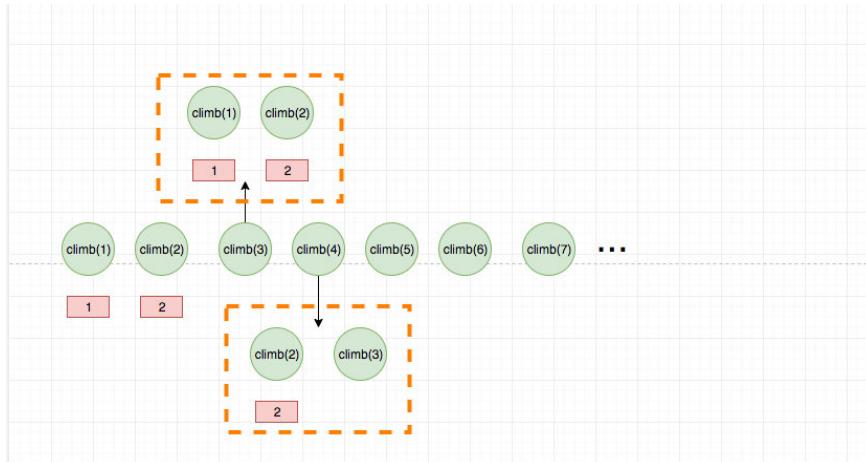
大家现在不会也没关系，我们将前文的递归的代码稍微改造一下。其实就是将函数的名字改一下：

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

经过这样的变化。我们将 $dp[n]$ 和 $dp(n)$ 对比看，这样是不是有点理解了呢？其实他们的区别只不过是递归用调用栈枚举状态，而动态规划使用迭代枚举状态。

如果需要多个维度枚举，那么记忆化递归内部也可以使用迭代进行枚举，比如最长上升子序列问题。

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

滚动数组优化

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。

这道题目是动态规划中最简单的问题了，因为仅涉及到单个因素的变化，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。

回答上面的问题：记忆化递归和动态规划除了一个用递归一个用迭代，其他没差别。那两者有啥区别呢？我觉得最大的区别就是记忆化递归无法使用滚动数组优化（不信你用上面的爬楼梯试一下），记忆化调用栈的开销比较大（复杂度不变，你可以认为空间复杂度常数项更大），不过几乎不至于 TLE 或者 MLE。因此我的建议就是没空间优化需求直接就记忆化，否则用迭代 dp。

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。

动态规划的基本类型

- 背包 DP（这个我们专门开了一个专题讲）
- 区间 DP

区间类动态规划是线性动态规划的扩展，它在分阶段地划分问题时，与阶段中元素出现的顺序和由前一阶段的哪些元素合并而来有很大的关系。令状态 $f(i,j)$ 表示将下标位置 i 到 j 的所有元素合并能获得的价值的最大值，那么 $f(i,j) = \max\{f(i,k) + f(k+1,j) + cost\}$ ， $cost$ 为将这两组元素合并起来的代价。

区间 DP 的特点：

合并：即将两个或多个部分进行整合，当然也可以反过来；

特征：能将问题分解为能两两合并的形式；

求解：对整个问题设最优值，枚举合并点，将问题分解为左右两个部分，最后合并两个部分的最优值得到原问题的最优值。

推荐两道题：

- [877. 石子游戏](#)

- [312. 戳气球](#)

- 状压 DP

关于状压 DP 可以参考下我之前写过的一篇文章：[状压 DP 是什么？这篇题解带你入门](#)

- 数位 DP

数位 DP 通常是这：给定一个闭区间，让你求这个区间中满足某种条件的数的总数。

推荐一道题 [Increasing-Digits](#)

- 计数 DP 和 概率 DP

这两个我就不多说。因为没啥规律。

之所以列举计数 DP 是因为两个原因：

1. 让大家知道确实有这个题型。
2. 计数是动态规划的副产物。

概率 DP 比较特殊，概率 DP 的状态转移公式一般是说一个状态有多大的概率从某一个状态转移过来，更像是期望的计算，因此也叫期望 DP。

推荐一道题：[837. 新 21 点](#)

更多题目类型以及推荐题目见刷题插件的学习路线。插件获取方式：公众号力扣加加回复插件。

什么时候用记忆化递归？

- 从数组两端同时进行遍历的时候使用记忆化递归方便，其实也就是区间 DP (range dp)。比如石子游戏，再比如这道题
<https://binarysearch.com/problems/Make-a-Palindrome-by-Inserting-Characters>

如果区间 dp 你的遍历方式大概需要这样：

```
class Solution:  
    def solve(self, s):  
        n = len(s)  
        dp = [[0] * n for _ in range(n)]  
        # 右边界倒序遍历  
        for i in range(n - 1, -1, -1):  
            # 左边界正序遍历  
            for j in range(i + 1, n):  
                # do something  
        return dp[0][m-1] # 一般都是使用这个区间作为答案
```

989. 数组形式的整数加法

如果使用记忆化递归则不需考虑遍历方式的问题。

代码：

```
class Solution:
    def solve(self, s):
        @lru_cache(None)
        def helper(l, r):
            if l >= r:
                return 0

            if s[l] == s[r]:
                return helper(l + 1, r - 1)

            return 1 + min(helper(l + 1, r), helper(l, r - 1))

        return helper(0, len(s) - 1)
```

- 选择 比较离散的时候，使用记忆化递归更好。比如马走棋盘。

那什么时候不用记忆化递归呢？答案是其他情况都不用。因为普通的 dp table 有一个重要的功能，这个功能记忆化递归是无法代替的，那就是滚动数组优化。如果你需要对空间进行优化，那一定要用 dp table。

热身开始

理论知识已经差不多了，我们拿一道题来试试手。

我们以一个非常经典的背包问题来练一下手。

题目：[322. 零钱兑换](#)

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以
你可以认为每种硬币的数量是无限的。

示例 1：

输入：coins = [1, 2, 5], amount = 11
输出：3
解释：11 = 5 + 5 + 1

这道题的参数有两个，一个是 coins，一个是 amount。

我们可以定义状态为 $f(i, j)$ 表示用 coins 的前 i 项找 j 元需要的最少硬币数。那么答案就是 $f(\text{len}(\text{coins}) - 1, \text{amount})$ 。

由组合原理， coins 的所有选择状态是 2^n 。状态总数就是 i 和 j 的取值的笛卡尔积，也就是 $2^{\text{len}(\text{coins})} * (\text{amount} + 1)$ 。

减 1 是因为存在 0 元的情况。

明确了这些，我们需要考虑的就是状态如何转移，也就是如何从寻常转移 to $f(\text{len}(\text{coins}) - 1, \text{amount})$ 。

如何确定状态转移方程？我们需要：

- 聚焦最优子结构
- 做选择，在选择中取最优解（如果是计数 dp 则进行计数）

对于这道题来说，我们的选择有两种：

- 选择 $\text{coins}[i]$
- 不选择 $\text{coins}[i]$

这无疑是完备的。只不过仅仅是对 coins 中的每一项进行选择与不选择，这样的状态数就已经是 2^n 了，其中 n 为 coins 长度。

如果仅仅是这样枚举肯定会超时，因为状态数已经是指数级别了。

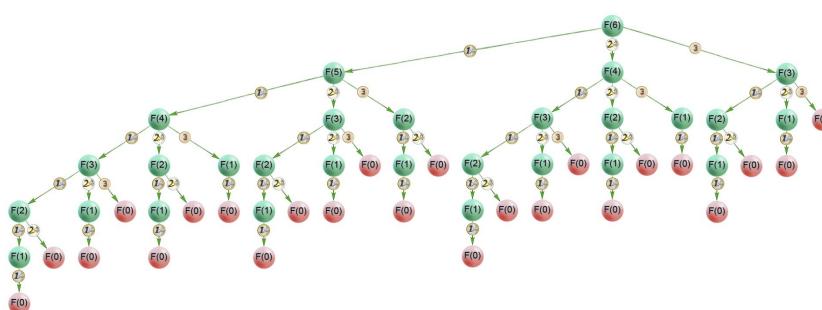
而这道题的核心在于 $\text{coins}[i]$ 选择与否其实没有那么重要，重要的其实是选择的 coins 一共有多少钱。

因此我们可以定义 $f(i, j)$ 表示选择了 coins 的前 i 项（怎么选的不关心），且组成 j 元需要的最少硬币数。

举个例子来说，比如 $\text{coins} = [1, 2, 3]$ 。那么选择 [1,2] 和 选择 [3] 虽然是不一样的状态，但是我们压根不关心。因为这两者没有区别，我们还是谁对结果贡献大就 pick 谁。

以 $\text{coins} = [1, 2, 3]$, $\text{amount} = 6$ 来说，我们可以画出如下的递归树。

Recursive tree for finding coin change of amount 6 with coin denominations of {1,2,3}.



(图片来自<https://leetcode.com/problems/coin-change/solution/>)

因此转移方程就是 $\min(\text{dp}[i][j], \text{dp}[i-1][j - \text{coins}[j]] + 1)$ ，含义就是： $\min(\text{不选择 coins}[j], \text{选择 coins}[j])$ 所需最少的硬币数。

989. 数组形式的整数加法

用公式表示就是：

$$dp[i] = \begin{cases} \min(dp[i][j], dp[i - 1][j - coins[j]] + 1) & j \geq coins[j] \\ amount + 1 & j < coins[j] \end{cases}$$

amount 表示无解。因为硬币的面额都是正整数，不可能存在一种需要 amount + 1 枚硬币的方案。

代码

记忆化递归：

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) ->
        @lru_cache(None)
    def dfs(amount):
        if amount < 0: return float('inf')
        if amount == 0: return 0
        ans = float('inf')
        for coin in coins:
            ans = min(ans, 1 + dfs(amount - coin))
        return ans
    ans = dfs(amount)
    return -1 if ans == float('inf') else ans
```

二维 dp:

```

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][-1]
    
```

$dp[i][j]$ 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。

一维 dp (滚动数组优化) :

```

class Solution:
    def coinChange(self, coins: List[int], amount: int) ->
        dp = [amount + 1] * (amount + 1)
        dp[0] = 0

        for j in range(len(coins)):
            for i in range(1, amount + 1):
                if i >= coins[j]:
                    dp[i] = min(dp[i], dp[i - coins[j]] + 1)

        return -1 if dp[-1] == amount + 1 else dp[-1]
    
```

推荐练习题目

最后推荐几道题目给大家，建议大家分别使用记忆化递归和动态规划来解决。如果使用动态规划，则尽可能使用滚动数组优化空间。

- [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。

动态规划的核心在于定义状态，定义好了状态其他都是水到渠成。

动态规划的难点在于枚举所有状态（不重不漏） 和 寻找状态转移方程。

参考

- [oi-wiki - dp](#) 这个资料推荐大家学习，非常全面。只不过更适合有一定基础的人，大家可以配合本讲义食用哦。

另外，大家可以去 LeetCode 探索中的 [递归 I](#) 中进行互动式学习。

背包专题

简介

背包问题是一类非常经典的动态规划问题，日常使用场景非常灵活。

百度百科定义：背包问题(Knapsack problem)是一种组合优化的 NP 完全问题。问题可以描述为：给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。问题的名称来源于如何选择最合适的物品放置于给定背包中。相似问题经常出现在商业、组合数学、计算复杂性理论、密码学和应用数学等领域中。也可以将背包问题描述为决定性问题，即在总重量不超过 W 的前提下，总价值是否能达到 V ？它是在 1978 年由 Merkle 和 Hellman 提出的。

常见题型及对应模版

背包问题在动态规划中的比例非常大，以至于我们单独将其从动态规划中抽取出来进行讲解。

下面给大家归纳几种常用的背包问题（01 背包，完全背包和多重背包三种类型）及其对应模版。说实话，如果实在理解不了，直接背住模版，把题目对应数据处理一下直接套题目，练习多了再回过头来复习可能就会恍然大悟。

有一点需要大家注意：几乎没有一道题是直接告诉你是背包的，这需要你自己的抽象能力。将问题抽象为背包，然后使用背包的套路去解决。我们也会在接下来的几天出几个题目，大家可以尝试将其抽象为背包问题。

01 背包问题

01 背包是最简单的类型，并且完全背包和多重背包都可以转化为 01 背包问题，因此搞清楚 01 背包是非常重要的。

问题描述

有 n 个物品，每个物品对应的重量为 w ，价值为 v ，问在不超过背包重量 M 的情况下，能够装入物品的最大价值，每个物品只能使用一次。

$w[i]$ 是第 i 个物品的重量， $v[i]$ 是第 i 个物品的价值。

分析

简单的思路是找到所有物品的组合，然后判断组合的体积和是否大于 M，如果不小于 M，则选择性更新最大价值 max_v（是否更新取决于当前的组合总价值是否大于 max_v），最后返回 max_v 即可。

n 个物品的组合的数量的“数量级”是 2^n ，n 稍微大点就很恐怖了，我们不得不考虑进行优化。

01 背包使用 dp 可以将复杂度降到 $O(n * W)$ ，具体怎么做呢？

定义状态 $dp[i][j]$ 表示仅考虑前 i 个物品将其装入承重为 j 的背包可以获得的最大价值，最终返回 $dp[n][m]$ 即可。

接下来考虑状态转移，具体会有如下两种情况：

- 当前第 i 件物品我要了（前提背包要装得下）。 $dp[i][j] = dp[i - 1][j - w[i]] + v[i]$
- 当前第 i 件物品我不要。 $dp[i][j] = dp[i - 1][j]$

由于我们的目标是价值最大，那么我们当然要选以上两种情况的最大值。

因此可以得到状态转移方程如下：

$$dp[i][j] = \max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i]), j >= [i]$$

通过上述分析可以很容易写出如下代码：

```
N, M, W, V
dp[0..N][0..M] = 0

for i in 1..N:
    for j in W[i]..M:
        dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - W[i]] +
                      V[i])

return dp[N][M]
```

不难发现当前 i 对应的状态计算只和 i - 1 有关，我们可以用动态规划专题讲到的滚动数组进行优化空间使其降至 M。

模板如下：

```
N, M, W, V
dp[0..M] = 0

for i in 1..N:
    for j in M..W[i]: # 这里必须逆向枚举，如果正向的话 i 状态会覆盖
        dp[j] = max(dp[j], dp[j - W[i]] + V[i])

return dp[M]
```

关于为何此处必须逆向枚举这个问题。简单来说，如果你不使用滚动数组，那么怎么枚举都无所谓，但是如果使用了在这里就必须逆序枚举。原因的话，我在文章末尾给大家解释。

这就是 01 背包问题。

完全背包问题

问题描述

有 n 个物品，每个物品对应的重量为 w ，价值为 v ，问在不超过背包重量 M 的情况下，能够装入物品的最大价值，与 01 背包的区别是每个物品可以使用无限次。

分析

完全背包问题状态转移方程和 01 背包问题很类似：

$dp[i][j]$ 表示将前 i 个物品装入承重为 j 的背包可以获得的最大价值。

那么 $dp[i][j]$ 求解时对应以下两种情况

- 当前第 i 件物品我要了（前提背包要装得下）： $dp[i][j] = dp[i-1][j] + v[i]$,
 $w[i]$ 是第 i 个物品的重量， $v[i]$ 是第 i 个物品的价值。（这里注意，这里是和 01 背包的区别所在，因为当前物品可以无限次被选，因此不应该用 $i-1$ 的状态计算而是继续在 i 状态）
- 当前第 i 件物品我不要： $dp[i-1][j]$

因此可以得到状态转移方程如下：

$$dp[i][j] = \max(dp[i-1][j], dp[i][j - w[i]] + v[i]), j \geq w[i]$$

一样，还是可以用滚动数组进行空间上的优化，大致模版如下：

```

N, M, W, V
dp[0..M] = 0

for i in 1..N:
    for j in W[i]..M: # 这里必须正向枚举，因为当前计算需要dp[i]
        dp[j] = max(dp[j], dp[j - W[i]] + V[i])

return dp[M]

```

多重背包问题

问题描述

该问题的描述和上面的区别仅仅在于，每个物品的个数有限制

分析

分析过程和上面也很类似，直接写出状态转移方程：

$$dp[i][j] = \max((dp[i - 1][j - h * w[i]] + h * v[i])) \text{ for every } h$$

其中 h 为装入第 i 件物品的个数， $h \leq \min(H[i], j / W[i])$ ， H 为物品及其个数的对应关系。

因为装入第 i 物品是从 $0-h$ 计算的，因此 $dp[i]$ 需要 $dp[i - 1]$ 的状态辅助完成，因此可以采用 01 背包优化的方式来优化空间使用，下面是模板代码：

```
N, M, W, V, H
dp[0..M] = 0

for i in 1...N:
    for j in M...W[i]: # 这里必须逆向枚举，因为当前计算需要dp[i]
        for h in 0...min(H[i], j / W[i]):
            dp[j] = max(dp[j], dp[j - h * W[i]] + h * V[i])

return dp[M]
```

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。

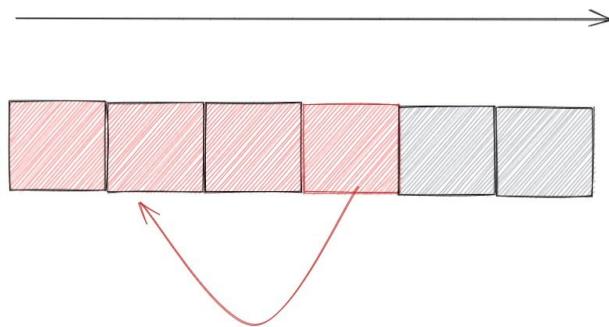
其实要回答这个问题，我要先将 01 背包和完全背包退化二维的情况。

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 - 1]])
```

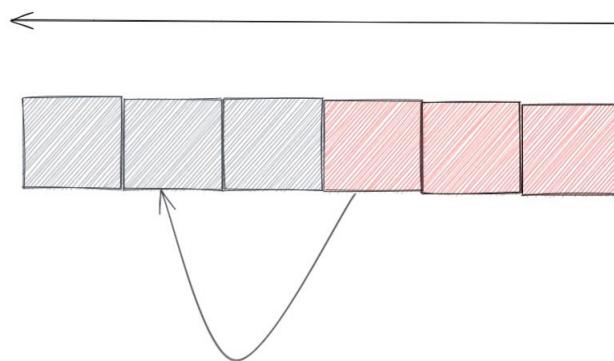
注意等号左边是 i ，右边是 $i - 1$ ，这很好理解，因为 i 只能取一次嘛。

那么如果我们不倒序遍历会怎么样呢？



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 $dp[i - 1][j - cost[i - 1]]$ 。

如果是倒序就可以了，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子问题。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 $dp[1], dp[2], \dots$ 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。

总结

万变不离其宗，还有背包的很多变形版本，以及不一定求最大价值，dp 的定义以及初始化是很灵活的，后面题目会涉及部分知识。

建议大家把模版翻译成自己擅长的语言，关于背包问题的详细介绍还请查阅背包问题经典的参考资料：[背包九讲第二版](#)。

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:

```

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: $O(amount * len(coins))$
- 空间复杂度: $O(amount)$

(322. ***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。

989. 数组形式的整数加法

Python Code:

```
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。

分治

分治即分而治之，我们可以将分治拆分一下进行解读。一个是分，另外一个是治。

- 分。将一个规模为 N 的问题分解为若干个规模较小的子问题
- 治。根据子问题的解求原问题。

从上面的描述，我们可以理出几个关键点。

- 一定是先分再治。
- 治一定是利用分的结果进行的，也就是说治依赖分。

因此一般我们可以：

将一个规模为 N 的问题分解为 K 个规模较小的子问题，并根据子问题的解求原问题。

那么就可以尝试使用分治法。

说起来很简单，但事实并非如此。分治是一种难度很大的思想，并且 LC 上题目并不多见。因此我的建议是大家透过几道经典的分支问题，好好研究，最后结合我们给出的每日一题题目进行联系，这样应付大多数面试是不成问题的。

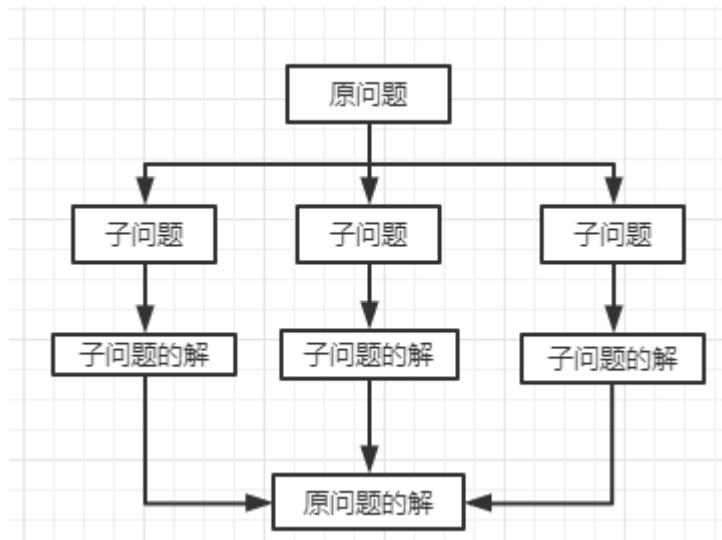
适用场景

一般题目具有以下 3 个特征，就可以考虑使用分治算法

1. 如果问题可以被分解为若干个规模较小的相同问题。（动态规划也是？那有什么区别？）
2. 这些被分解的问题的结果可以进行合并。（分治的核心标志）
3. 这些被分解的问题是相互独立的，不包含重叠的子问题（动态规划也是？那有什么区别？）

可以看出，分治和动态规划有很深的联系。另外分治和其他思想，比如二分也颇有原因，因为本质上二分就是一种只有分没有治的分治。

解题步骤



1. 将原问题分解至，达到求解边界的结构相同互相独立的子问题。（这是重点，也是难点。实际操作并不容易想到）
2. 对所有的子问题进行求解。
3. 将所有子问题的解进行合并，从而得到原问题。

相比于 2 和 3，1 既是重点也是难点。很多题目其实很难看出来可以使用分治来解决的。也无法通过几句话就讲清楚的，我认为分治是比动态规划更难的思想。因为分治动态规划只需要关注局部即可，而分治则需要关注全体，因为你需要对全体进行分和治的操作。

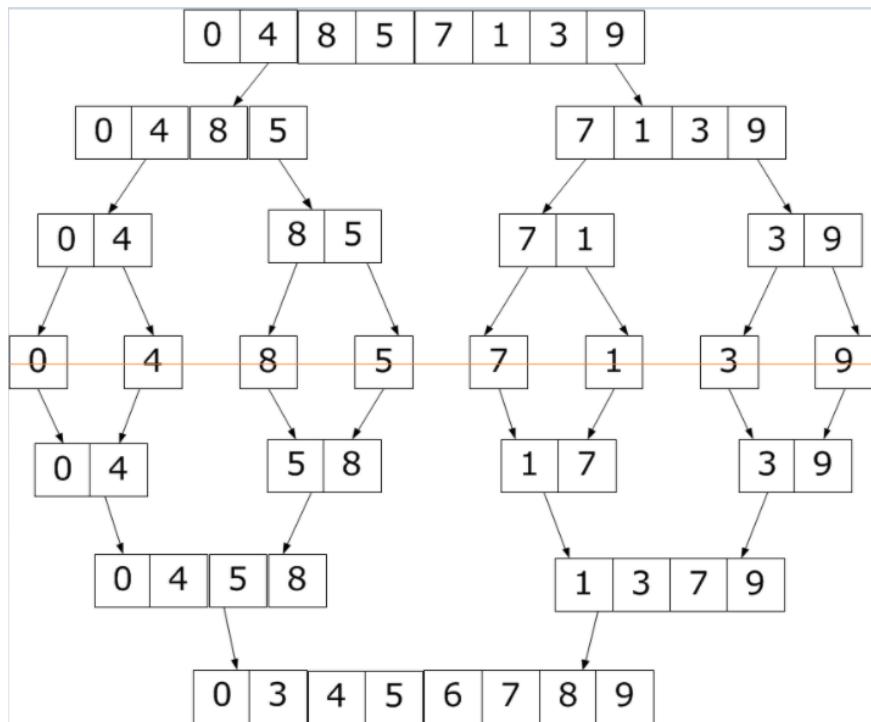
上面是分治做题的指导思想，对你做具体题目实际上并没有什么帮助。接下来我们来一点实操技巧。

实操技巧

这里给出几个解决分治问题的几个分治技巧给大家。

1. 思考子问题的求解边界(问题缩小至什么规模时可以求解)，使用函数来定义问题。（和动态规划类似）
2. 思考如果将子问题的解进行合并。（假设子问题已经计算好了，我们如何合并？）
3. 思考编码思路（一般利用递归）

我们以经典的归并排序算法为例，讲解一下如下使用这几个技巧。



如果，我们需要对一个数组进行排序。这个问题可以进行分解成独立子问题么？

1. 显然如果规模缩小到只有一个元素时，问题就变得很容易解了。（实操技巧 1）

接下来，我们思考：如果两个子数组（子问题）已经有序，我们如何将其合并？

1. 然后我们发现将子问题的解两两之间进行合并其实就是合并两个有序数组（使用[双指针](#)轻松解决）（实操技巧 2）
 2. 于是，我们可以定义问题函数为 $f(l, r)$ 为对数组 $[l, r]$ 区间进行排序。
- 接下来逐步分解到对单个数（分）进行求解（单个数天然有序），之后使用合并两个有序数组的技巧（治）来完成，这样逐步治即可得到答案。

代码 (JS)

```

var sortArray = function (nums) {
    var len = nums.length;
    if (len < 2) {
        return nums;
    }
    var middle = parseInt(len / 2),
        left = nums.slice(0, middle),
        right = nums.slice(middle);
    return merge(sortArray(left), sortArray(right));
};

function merge(left, right) {
    let ans = [];
    let leftPoint = (rightPoint = 0);
    let leftLen = left.length,
        rightLen = right.length;
    while (leftPoint < leftLen && rightPoint < rightLen) {
        if (left[leftPoint] < right[rightPoint]) {
            ans.push(left[leftPoint]);
            leftPoint++;
        } else {
            ans.push(right[rightPoint]);
            rightPoint++;
        }
    }
    return ans.concat(
        leftPoint < leftLen ? left.slice(leftPoint) : right.slice(rightPoint)
    );
}

```

时间复杂度为 $O(n \log n)$, 空间复杂度是 $O(n)$ (可通过原地交换优化到 $O(1)$)

分治和二分的异同

分治和二分其实还是有点像的。只不过二分只对问题进行分，分完直接舍弃。而分治不仅需要对问题进行分解，而且需要对问题的多个问题进行治。

分治和动态规划都涉及到问题的问题，并且都需要保证子问题不重不漏。那么两者有啥不同呢？动态规划是通过递推和选择进行转移，从特殊推广到一般。而分治也可能涉及到选择，比如快速选择算法就是一个分治算法。那它俩不就更一样了吗？我觉得真的挺像的，只不过有一点两者有很大的不同。那就是动态规划解决的问题往往伴随重叠子问题（因为不重叠没必要动态规划），而分治则不是。

总结

如果一个问题可以被分解为若干个**不重叠**的独立子问题，并且子问题可以推导出原问题。我们就可以对问题进行定义，并使用分治来解决。具体地，我们先对问题进行分解（可以借助递归完成），接下来对问题的解进行合并（通常结合其他基础算法知识）即可。

分治是一种很难掌握且考察相对比较少的思想。另外它和二分，动态规划等其他专题渊源颇深，大家可以结合起来进行理解。

通过我们给出的经典例题，大家可以感受一下分治和二分以及动态规划的异同，这可以很好地帮助你学习分治算法。

练习题

最后推荐几道题帮助大家理解和运用分治思维。

- 手写堆
- 手写线段树
- 归并排序
- [面试题 08.06. 汉诺塔问题](#)
- [96. 不同的二叉搜索树](#)

贪心

简介

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

贪心算法不能保证每次都能找到最优解，有时候只能找到接近最优解的方案。所以求解时，要确定问题具有贪心选择性质：每一次选择的局部最优可以导致问题的整体最优。比如硬币找零问题（后面证明部分会讲）使用贪心解法则可能会得到错误的答案。

另外即使题目可以通过贪心解决，那么如何选择贪心策略也是关键，因此可能存在一种错误的贪心策略。

总的来说，贪心算法是仅考虑局部最优的算法，难点在于如何识别出是贪心问题以及贪心的策略选择。更严谨地，我们则需要证明贪心的正确性，这通常比想到并用贪心做出来更难。

贪心算法的运用非常广泛，比如哈夫曼树，Dijkstra 算法等。

使用场景

1. 贪心选择的局部最优解能得到整体的最优解。而正向思考比较困难，我们一般从反向进行思考。如果能举出反例局部最优解不能得到全局最优解那么一定不能使用贪心。
2. 贪心策略无后效性，即当前贪心选择不会影响以后的状态，只与当前状态有关。这一点其实和动态规划是一样的。我们可以将贪心算法看成是不需要回溯的动态规划。因为动态规划很多时候需要回溯以及计算好的局部最优解，通过它们得到当前的最优解。

证明

贪心适合求解的问题是极值的问题，并且贪心策略通常也是显而易见的那种。虽然贪心策略显而易见，但是却不一定正确的。

比如给你一堆硬币，面值分别为 [1,3,5]，你需要找零 9 元，如果才能是的找的硬币数目最少？贪心的策略是优先取最大的，这样可以更快地缩小问题的规模。因此我们先选 5，为了凑够剩下的 4，我们需要选择 4 枚面值为 1 的硬币。不过这显然不是最优的，我们其实可以直接选 3 枚面值为 3 的硬币。

上面是一个错误使用贪心策略的例子。而证明贪心不可能只需要像上面一样举个反例就行了。但是想证明贪心算法是对的就很难了。也就是说贪心算法的难点在于如何知道贪心的策略是正确的。那如何证明贪心算法的正确性呢？常见的两种方法是：反证法和数学归纳法。

- 反证法：如果交换方案中任意两个元素，答案不会变得更好，那么可以推定目前的解已经是最优解了。
- 数学归纳法：先算得出边界情况的最优解，比如 $F(1)$ ，然后再证明 $F(n)$ 都可以由 $F(n-1)$ 推导出来

当然也可能是 $F(n-2)$ 等，只要是可以推导出且规模更小就行了。

常见题型

- 先排序，再按照价值从高到低或者从低到高选取。
- 事后诸葛亮。具体见我写的[堆专题](#)。

解题步骤

1. 将问题分解为子问题
2. 求出当前子问题的局部最优解
3. 通过这个局部最优解推导出全局最优解

经典题目

881. 救生艇

题目地址

<https://leetcode-cn.com/problems/boats-to-save-people/>

题目描述

第 i 个人的体重为 $\text{people}[i]$, 每艘船可以承载的最大重量为 limit 。

每艘船最多可同时载两人, 但条件是这些人的重量之和最多为 limit 。

返回载到每一个人所需的最小船数。(保证每个人都能被船载)。

示例 1:

输入: $\text{people} = [1, 2]$, $\text{limit} = 3$

输出: 1

解释: 1 艘船载 (1, 2)

示例 2:

输入: $\text{people} = [3, 2, 2, 1]$, $\text{limit} = 3$

输出: 3

解释: 3 艘船分别载 (1, 2), (2) 和 (3)

示例 3:

输入: $\text{people} = [3, 5, 3, 4]$, $\text{limit} = 5$

输出: 4

解释: 4 艘船分别载 (3), (3), (4), (5)

提示:

$1 \leq \text{people.length} \leq 50000$

$1 \leq \text{people}[i] \leq \text{limit} \leq 30000$

思路

1. 题目要求船数最少, 即每个船都尽可能多装一些重量 (分解为子问题)

2. 每个船在容量固定情况下, 尽可能多装一些重量 (类似背包问题)。

由于题目限制船最多装两个人, 所以每次装人的时候先将最大重量的装上, 然后再从轻到重遍历剩下的人, 看剩余容量能否再装下一人 (制定贪心策略)。

这种贪心策略可行的原因我们可以使用上面提到的反证法证明。由于每个人都必须被装进去, 所以每个人都必须思考如何载运。现在我们不妨仅思考如何安排最重的那个人, 由于船最多载两个人, 因此他可以选择和另外一个人同时乘船, 为了使船更少, 显然我们需要尽量让他和其他人同一艘船, 那选谁呢?

如果最轻的人 a 和最重的人 b 重量和超过了 limit , 那么其他人不用看了, 肯定都不行。那如果最轻的人 a 可以和最重的人 b 同时载运, 此时会存在另外一种方案 (其选择非最轻的人 c 和最重的人 b 配对) 比其更优

么？答案是不能，因此假设存在这样的更优方案，那么我们必然可以通过交换 **a** 和 **c** 得到不比其差的答案。因此最重的人和最轻的人配对是最合适的，换句话说没有比这种方法更优的解。

显然，最优解可能有多重。这种方案选择的只是其中一种罢了。

1. 将已经被装上船的人踢出列表，继续按 2 的策略装直到所有人都上船
(局部最优解推导出全局最优解)

代码

JS Code:

```
var numRescueBoats = function (people, limit) {
    // 用到了排序。这个是我们总结的贪心两种题型中的一种
    people.sort((a, b) => a - b);
    let ans = 0,
        start = 0,
        end = people.length - 1;
    while (start <= end) {
        if (people[end] + people[start] <= limit) {
            start++;
            end--;
        } else {
            end--;
        }
        ans++;
    }
    return ans;
};
```

令 n 为数组长度。

复杂度分析

- 时间复杂度: $O(n \log n)$
- 空间复杂度: $O(1)$

765. 情侣牵手

题目地址

<https://leetcode-cn.com/problems/couples-holding-hands/>

题目描述

N 对情侣坐在连续排列的 $2N$ 个座位上，想要牵到对方的手。计算最少交换座人和座位用 0 到 $2N-1$ 的整数表示，情侣们按顺序编号，第一对是 $(0, 1)$ 。这些情侣的初始座位 $\text{row}[i]$ 是由最初坐在第 i 个座位上的人决定的。

示例 1：

输入： $\text{row} = [0, 2, 1, 3]$

输出： 1

解释： 我们只需要交换 $\text{row}[1]$ 和 $\text{row}[2]$ 的位置即可。

示例 2：

输入： $\text{row} = [3, 2, 0, 1]$

输出： 0

解释： 无需交换座位，所有的情侣都已经可以手牵手了。

说明：

$\text{len}(\text{row})$ 是偶数且数值在 $[4, 60]$ 范围内。

可以保证 row 是序列 $0 \dots \text{len}(\text{row})-1$ 的一个全排列。

思路

我们需要将所有的情侣按 $(0, 1), (2, 3)$ 安排座位，那么：

- 如果某一个人最终编号是奇数，那么他对应的情侣应该在他的左边。
- 如果某一个人最终编号是偶数，那么他对应的情侣应该在他的右边。

因此我们可以进行一次遍历，采用两两一对的遍历方式（步长为 2）。如果遍历的时候，相邻的不是情侣，则进行一次交换。那么和谁交换呢？显然交换依据应该是上面提到的关键点（通过其编号判断其情侣在 ta 的左边还是右边）。这提示我们先将每一个人的最终编号和当前编号进行一次哈希映射。

具体算法：

1. 我们先做一个哈希表进行映射，key 是数组值，value 是其下标。
2. 依次遍历所有人，如果他的情侣坐在他旁边就继续下一轮循环，如果不在他旁边就通过哈希表找到情侣当前在哪儿，然后把伴侣跟身边的基佬换个位置这样就又凑成一对（制定贪心策略）

交换位置后别忘了更新哈希表

1. 循环执行第 2 步直到所有人都凑到一起了（局部最优解推导出全局最优解）

这种算法的时间和空间复杂度都是 $O(n)$ ，其中 n 为数组长度。

接下来我们证明一下贪心策略的正确性。

我发现宫水三叶写的这个证明蛮不错的，因此就不写了。地址：

<https://leetcode-cn.com/problems/couples-holding-hands/solution/liang-chong-100-de-jie-fa-bing-cha-ji-ta-26a6/>

为了防止内容被删除，我摘要主要内容如下：

我们这样的做法本质是什么？

其实相当于，当我处理到第 k 个位置的时候，前面的 $k - 1$ 个位置的情侣已经牵手成功了。我接下来怎么处理，能够使得总花销最低。

分两种情况讨论：

- a. 现在处理第 k 个位置，使其牵手成功：

那么我要使得第 k 个位置的情侣也牵手成功，那么必然是保留第 k 个位置的情侣中其中一位，再进行修改，这样的成本是最小的（因为只需要交换一次）。

而且由于前面的情侣已经牵手成功了，因此交换的情侣必然在 k 位置的后面。

然后我们再考虑交换左边或者右边对最终结果的影响。

分两种情况来讨论：

1. 与第 k 个位置的匹配的两个情侣不在同一个位置上：这时候无论交换左边还是右边，后面需要调整的「情侣对数量」都是一样。假设处理第 k 个位置前需要调整的数量为 n 的话，处理完第 k 个位置（交换左边或是右边），需要调整的「情侣对数量」都为 $n - 1$ 。



1. 与第 k 个位置的匹配的两个情侣在同一个位置上：这时候无论交换左边还是右边，后面需要调整的「情侣对数量」都是一样。假设处理第 k 个位置前需要调整的数量为 n 的话，处理完第 k 个位置（交换左边或是右边），需要调整的「情侣对数量」都为 $n - 2$ 。



因此对于第 k 个位置而言，交换左边还是右边，并不会影响后续需要调整的「情侣对数量」。

b. 现在先不处理第 k 个位置，等到后面的情侣处理的时候「顺便」处理第 k 位置：

由于我们最终都是要所有位置的情侣牵手，而且每一个数值对应的情侣数值是唯一确定的。

因此我们这个等“后面”的位置处理，其实就是等与第 k 个位置互为情侣的位置处理（对应上图的就是我们是在等【0 x】和【8 y】或者【0 8】这些位置被处理）。

由于被处理都是同一批的联通位置，因此和「a. 现在处理第 k 个位置」的分析结果是一样的。

不失一般性的，我们可以将这个分析推广到第一个位置，其实就已经是符合「当我处理到第 k 个位置的时候，前面的 $k - 1$ 个位置的情侣已经牵手成功了」的定义了。

综上所述，我们只需要确保从前往后处理，并且每次处理都保留第 k 个位置的其中一位，无论保留的左边还是右边都能得到最优解。

代码

代码支持：Java, JS

Java Code：

989. 数组形式的整数加法

```
class Solution {
    public int minSwapsCouples(int[] row) {
        int n = row.length;
        int ans = 0;
        int[] map = new int[n];
        for (int i = 0; i < n; i++) map[row[i]] = i;
        for (int i = 0; i < n - 1; i += 2) {
            int a = row[i], b = a ^ 1;
            if (row[i + 1] != b) {
                int src = i + 1, tar = map[b];
                map[row[tar]] = src;
                map[row[src]] = tar;
                swap(row, src, tar);
                ans++;
            }
        }
        return ans;
    }
    void swap(int[] nums, int a, int b) {
        int c = nums[a];
        nums[a] = nums[b];
        nums[b] = c;
    }
}
```

JS Code:

```

var minSwapsCouples = function (row) {
    if (row.length <= 2) return 0;
    var len = row.length;
    var map = new Array(len);
    for (var index in row) map[row[index]] = index;
    var next = (count = 0);
    for (var i = 0; i <= len - 2; i += 2) {
        next = row[i] + (row[i] % 2 === 0 ? 1 : -1);
        if (row[i + 1] != next) {
            var temp = row[i + 1];
            row[i + 1] = row[map[next]];
            row[map[next]] = temp;
            temp = map[row[map[next]]];
            map[row[map[next]]] = map[row[i + 1]];
            map[row[i + 1]] = temp;
            count++;
        }
    }
    return count;
};

```

令 n 为数组长度。

复杂度分析

- 时间复杂度: $O(n)$
- 空间复杂度: $O(n)$

本题也有很多其他优秀的方法，大家可以参考下力扣的相关题解。

总结

实际做题的过程，更关注的是如何使用贪心策略，而不是如何证明。常见的策略有：

- 排序
- 事后诸葛亮

既然重点是做出来，那么大家不妨直接根据直觉入手，写出来代码。如果代码通过，则说明可能是没问题的。如果不通过，则说明可能是不可行的（相当于 OJ 平台给你了一个反例）。

而如果你执意要证明，那么可以考虑我们提到的两种方法：数学归纳法和反证法。两种方法都不容易，并且没有什么通常的思考技巧。因此建议大家从几个经典例题入手思考如何证明。最后附上经典题目推荐：

989. 数组形式的整数加法

- <https://github.com/azl397985856/leetcode/blob/master/thinkings/greedy.md>
- <https://leetcode-cn.com/problems/minimum-initial-energy-to-finish-tasks/>

位运算

计算机中编码表示方式

原码

原码就是一个数字的二进制表示。

反码

对于单个数值（二进制的 0 和 1）而言，对其进行取反操作就是将 0 变为 1，1 变为 0。对一个数字每一位都进行一次取反，就可以得到它的反码。

补码

英文名 2's complement。是一种用二进制表示有号数的方法，也是一种将数字的正负号变号的方式，常在计算机科学中使用。补码以有符号比特的二进制数定义。正数和 0 的补码就是该数字本身。负数的补码则是将其对应正数按位取反（反码）再加 1。

补码系统的最大优点是可以在加法或减法处理中，不需因为数字的正负而使用不同的计算方式。只要一种加法电路就可以处理各种有号数加法，而且减法可以用一个数加上另一个数的补码来表示，因此只要有加法电路及补码电路即可完成各种有号数加法及减法，在电路设计上相当方便。简单来说，就是可以统一加减法。

另外，补码系统的 0 就只有一个表示方式，这和反码系统不同（在反码系统中，0 有二种表示方式），因此在判断数字是否为 0 时，只要比较一次即可。

常见的位运算

| 符号 | 描述 | 运算规则 | |
|----|----|---|------------------|
| & | 与 | 两个位都为 1 时，结果才为 1 | |
| \ | | 或 | 两个位都为 0 时，结果才为 0 |
| ^ | 异或 | 两个位相同为 0，相异为 1 | |
| ~ | 取反 | 0 变 1，1 变 0 | |
| << | 左移 | 各二进位全部左移若干位，高位丢弃，低位补 0 | |
| >> | 右移 | 各二进位全部右移若干位，对无符号数，高位补 0，有符号数，各编译器处理方法不一样，有的补符号位（算术右移），有的补 0（逻辑右移） | |

在算法题中通常考察，并且大家比较容易忽略的就是异或和位移运算（左移和右移）。

两个数字异或，我们需要对其每一位的数字异或得到结果。如果两位的数字相同则结果为 0，不同则为 1。因此异或有以下性质：

- 任何数和本身异或则为 0
- 任何数和 0 异或是 本身
- 异或运算满足交换律，即： $a \wedge b \wedge c = a \wedge c \wedge b$

一个简单的异或的应用是通过异或运算，在不借助第三个变量的情况下可以实现两数对调：

```

a = a ^ b
b = a ^ b
a = a ^ b

```

常见套路

- 如果你想将某几个二进制位变成 1，其他二进制位不变，可以用或运算。比如 $a | 0xff$ ，就是将 a 的低八位变为 1，其他位不变。

- 如果你想将某几个二进制位不变，其他二进制变成 0，可以用与运算。比如 $a \& 0xff$ ，就是将 a 的低八位保持不变，其他位置为 0。
- 当你需要用的异或的自反性的话，就考虑使用异或。自反性指的是异或的“反运算”还是异或。
- 非运算用的比较少，做题过程我基本没有用过。
- 右移运算等价于乘以 2，但是要比乘法快得多。因此建议大家使用位移，而不是乘以 2（或者 2 的幂）

左移也是同理

- 当题目的数据范围有一项是 30 以内，可以考虑是否可以使用状态压缩。这样就可以使用位运算来优化性能。

位运算常见题型

直接考察位运算性质

- 231. 2 的幂
- 268. 缺失数字

求某一个二进制位的值

比如我们要求 a 的第 n 位（从低到高）是多少。那么可以通过 $(a >> n) \& 1$ 的方式来获取。实际上 $(a >> n) \& 1$ 就是一个左边全部是 0，最低位和 a 的第 n 位保持一致的数。

当然也可以用别的方式。比如 $(a \& (1 << n)) >> n$ 。

当 $n == 31$ 时，程序可能有问题。那么会有什么问题呢？

状态压缩

将状态进行压缩，可使用位来模拟。实际上使用状态压缩和不使用压缩的「思路一模一样，只是 API 不一样」罢了。这部分主要考察大家灵活使用位运算来解决或者降低时空复杂度。

举个例子来进行说明。题目描述：

989. 数组形式的整数加法

给定一组不含重复元素的整数数组 `nums`, 返回该数组所有可能的子集（幂集）

说明：解集不能包含重复的子集。

示例：

输入: `nums = [1,2,3]`

输出:

```
[  
[3],  
[1],  
[2],  
[1,2,3],  
[1,3],  
[2,3],  
[1,2],  
[]  
]
```

这道题我们可以直接使用暴力回溯来解决。

JS Code:

```
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;  
};
```

复杂度分析

- 时间复杂度：由排列组合原理可知，一共有 2^N 种组合，因此时间复杂度为 $O(2^N)$ ，其中 N 为数字的个数。

- 空间复杂度：由于调用栈深度最多为 $\$N\$$ ，且临时数组长度不会超过 $\$N\$$ ，因此空间复杂度为 $\$O(N)\$$ ，其中 $\$N\$$ 为数字的个数。

实际上，我们还可以对其进行优化，活地使用位运算的性质，来解决这道题。

例如我们现在有 3 个元素，那我们分别给这 3 个元素编号为 A B C

实际上这三个元素能取出的所有子集就是这 3 个元素的使用与不使用这两种状态的笛卡尔积。我们使用 0 与 1 分别表示这 3 个元素的使用与不使用的状态。那么这 3 个元素能构成的所有情况其实就是：

```
000, 001, 010 ... 111
```

那么我们就依次遍历这些数，将为 1 的元素取出，即为子集：

代码(JS):

```
var subsets = function (nums) {
    let res = [],
        sum = 1 << nums.length,
        temp;
    for (let now = 0; now < sum; now++) {
        temp = [];
        for (let i = 0; now >> i > 0; i++) {
            if (((now >> i) & 1) == 1) {
                temp.push(nums[i]);
            }
        }
        res.push(temp);
    }
    return res;
};
```

时间和空间复杂度均为 $\$O(n)\$$ ，其中 n 为 nums 中数字总和。

更多参考：

- [状压 DP 是什么？这篇题解带你入门](#)

二进制的思维角度

有时间从二进制角度思考问题或许可以实现降维打击的效果。不过这种思维方式不是很容易掌握，需要大家不断练习来掌握。

题目推荐：

- [从老鼠试毒问题来看二分法](#)

更多

- [力扣专题 - 位运算](#)

总结

位运算的题目，首先要知道的就是各种位运算有哪些，对应的功能以及性质。很多题目的考点基本都是围绕性质展开。另外一种题目的考点是状态压缩，大大减少时间和空间复杂度。使用位运算的状态压缩一点都不神秘，只是 api 不一样罢了。如果你不会，只能说明你堆位运算 api 不熟悉，多用几次其实就好了。

Trie

字典树也叫前缀树、Trie。它本身就是一个树型结构，也就是一颗多叉树，学过树的朋友应该非常容易理解，它的核心操作是插入，查找。删除很少使用，因此这个讲义不包含删除操作。

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。

简介

我们想一下用百度搜索时候，打个“一语”，搜索栏中会给出“一语道破”，“一语成谶(四声的 chen)”等推荐文本，这种叫模糊匹配，也就是给出一个模糊的 query，希望给出一个相关推荐列表，很明显， hashmap 并不容易做到模糊匹配，而 Trie 可以实现基于前缀的模糊搜索。

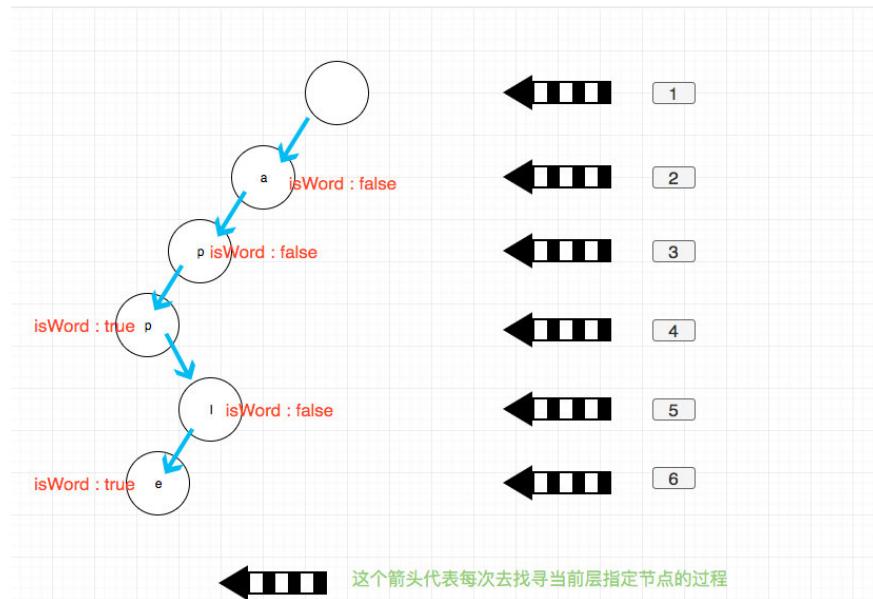
注意这里的模糊搜索也仅仅是基于前缀的。比如还是上面的例子，搜索“道破”就不会匹配到“一语道破”，而只能匹配“道破 xx”

基本概念

假想一个场景：给你若干单词 words 和一系列关键字 keywords，让你判断 keywords 是否在 words 中存在，或者判断 keywords 中的单词是否有 words 中的单词的前缀。比 pre 就是 pres 的前缀之一。

朴素的想法是遍历 keywords，对于 keywords 中的每一项都遍历 words 列表判断二者是否相等，或者是否是其前缀。这种算法的时间复杂度是 $O(m * n)$ ，其中 m 为 words 的平均长度，n 为 keywords 的平均长度。那么是否有可能对其进行优化呢？答案就是本文要讲的前缀树。

我们可以将 words 存储到一个树上，这棵树叫做前缀树。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。

为了搞明白前缀树是如何优化暴力算法的。我们需要了解一下前缀树的基本概念和操作。

节点：

- 根结点无实际意义
- 每一个节点**数据域**存储一个字符
- 每个节点中的**控制域**可以自定义，如 isWord(是否是单词)，count(该前缀出现的次数)等，需实际问题实际分析需要什么。

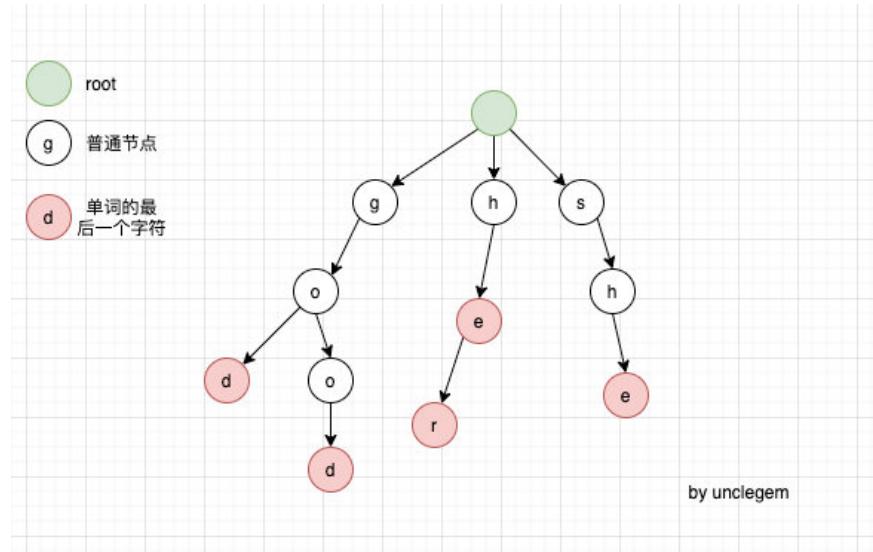
一个可能的前缀树节点：

```
class TrieNode {
    private TrieNode[] children;
    public boolean isWord;

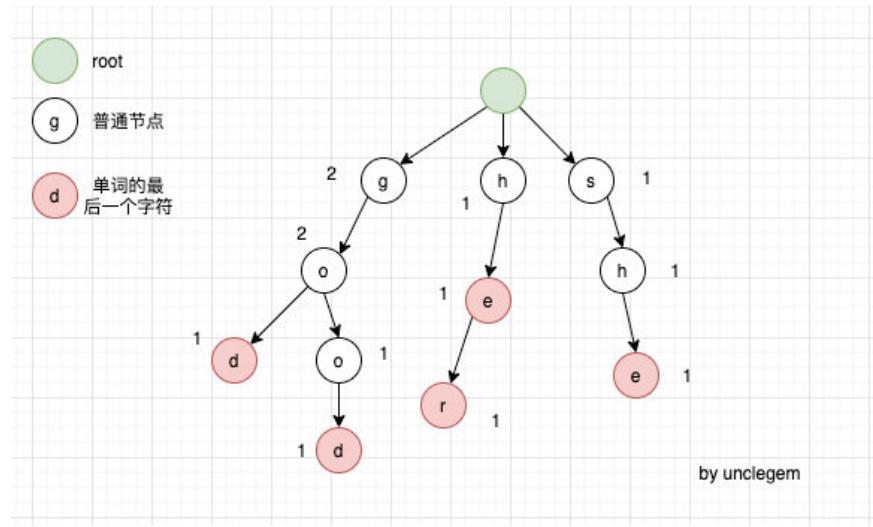
    // Initialize your data structure here.
    public TrieNode() {
        children = new TrieNode[26];
        isWord = false;
    }
}
```

Trie 的插入

构建 Trie 的核心就是插入。而插入指的就是将单词 (words) 全部依次插入到前缀树中。假定给出几个单词 words [she,he,her,good,god]构造出一个 Trie 如下图：



也就是说从根结点出发到某一粉色节点所经过的字符组成的单词，在单词列表中出现过，当然我们也可以给树的每个节点加个 count 属性，代表根结点到该节点所构成的字符串前缀出现的次数



可以看出树的构造非常简单：插入新单词的时候就从根结点出发一个字符一个字符插入，有对应的字符节点就更新对应的属性，没有就创建一个！

Trie 的查询

查询更简单了，给定一个 Trie 和一个单词，和插入的过程类似，一个字符一个字符找

- 若中途有个字符没有对应节点 → Trie 不含该单词
- 若字符串遍历完了，都有对应节点，但最后一个字符对应的节点并不是粉色的，也就不是一个单词 → Trie 不含该单词

Trie 模版

了解了 Trie 的使用场景以及基本的 API，那么最后就是用代码来实现了。这里我提供了 Python 和 Java 两种语言的代码。

Java Code:

989. 数组形式的整数加法

```
class Trie {

    TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String word) {
        TrieNode node = root;

        for (int i = 0; i < word.length(); i++) {
            if (node.children[word.charAt(i) - 'a'] == null)
                node.children[word.charAt(i) - 'a'] = new TrieNode();

            node = node.children[word.charAt(i) - 'a'];
            node.preCount++;
        }

        node.count++;
    }

    public boolean search(String word) {
        TrieNode node = root;

        for (int i = 0; i < word.length(); i++) {
            if (node.children[word.charAt(i) - 'a'] == null)
                return false;

            node = node.children[word.charAt(i) - 'a'];
        }

        return node.count > 0;
    }

    public boolean startsWith(String prefix) {
        TrieNode node = root;

        for (int i = 0; i < prefix.length(); i++) {
            if (node.children[prefix.charAt(i) - 'a'] == null)
                return false;
        }
    }
}
```

989. 数组形式的整数加法

```
        node = node.children[prefix.charAt(i) - 'a'];
    }

    return node.preCount > 0;
}

private class TrieNode {

    int count; //表示以该处节点构成的串的个数
    int preCount; //表示以该处节点构成的前缀的字串的个数
    TrieNode[] children;

    TrieNode() {

        children = new TrieNode[26];
        count = 0;
        preCount = 0;
    }
}
}
```

Python Code:

989. 数组形式的整数加法

```
class TrieNode:
    def __init__(self):
        self.count = 0
        self.preCount = 0
        self.children = {}

class Trie:

    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for ch in word:
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch]
            node.preCount += 1
        node.count += 1

    def search(self, word):
        node = self.root
        for ch in word:
            if ch not in node.children:
                return False
            node = node.children[ch]
        return node.count > 0

    def startsWith(self, prefix):
        node = self.root
        for ch in prefix:
            if ch not in node.children:
                return False
            node = node.children[ch]
        return node.preCount > 0
```

JavaScript Code

```

var Trie = function() {
    this.children = {};
};

Trie.prototype.insert = function(word) {
    let node = this.children;
    for(let char of word){
        if(!node[char]) node[char] = {};
        node = node[char];
    }
    node.isEnd = true;
};

Trie.prototype.search = function(word) {
    let node = this.children;
    for(let char of word){
        if(!node[char]) return false;
        node = node[char];
    }
    return node.isEnd == true;
};

Trie.prototype.startsWith = function(prefix) {
    let node = this.children;
    for(let char of prefix){
        if(!node[char]) return false;
        node = node[char];
    }
    return true;
};

```

复杂度分析

- 插入和查询的时间复杂度自然是 $O(\text{len}(\text{key}))$, key 是待插入(查找)的字串。
- 建树的最坏空间复杂度是 $O(m^{\{n\}})$, m 是字符集中字符个数, n 是字符串长度。

回答开头的问题

前面我们抛出了一个问题：给你若干单词 words 和一系列关键字 keywords , 让你判断 keywords 是否在 words 中存在, 或者判断 keywords 中的单词是否有 words 中的单词的前缀。比 pre 就是 pres 的前缀之一。

如果使用 Trie 来解，会怎么样呢？首先我们需要建立 Trie，这部分的时间复杂度是 $O(t)$ ，其中 t 为 words 的总字符。预处理完毕之后就是查询了。对于查询，由于树的高度是 $O(m)$ ，其中 m 为 words 的平均长度，因此查询基本操作的次数不会大于 m 。当然查询的基本操作次数也不会大于 k ，其中 k 为被查询单词 keyword 的长度，因此对于查询来说，时间复杂度为 $O(\min(m, k))$ 。时间上优化的代价是空间上的消耗，对于空间来说则是预处理的消耗，空间复杂度为 $O(t)$ 。

前缀树的特点

简单来说，前缀树就是一个树。前缀树一般是将一系列的单词记录到树上，如果这些单词没有公共前缀，则和直接用数组存没有任何区别。而如果有公共前缀，则公共前缀仅会被存储一次。可以想象，如果一系列单词的公共前缀很多，则会有效减少空间消耗。

而前缀树的意义实际上是空间换时间，这和哈希表，动态规划等的初衷是一样的。

其原理也很简单，正如我前面所言，其公共前缀仅会被存储一次，因此如果我想在一堆单词中找某个单词或者某个前缀是否出现，我无需进行完整遍历，而是遍历前缀树即可。本质上，使用前缀树和不使用前缀树减少的时间就是公共前缀的数目。也就是说，一堆单词没有公共前缀，使用前缀树没有任何意义。

知道了前缀树的特点，接下来我们自己实现一个前缀树。关于实现可以参考 [0208.implement-trie-prefix-tree](#)

应用场景及分析

正如上面所说，前缀树的核心思想是用空间换时间，利用字符串的公共前缀来降低查询的时间开销。

比如给你一个字符串 query，问你这个字符串是否在字符串集合中出现过，这样我们就可以将字符串集合建树，建好之后来匹配 query 是否出现，那有的朋友肯定会问，之前讲过的 hashmap 岂不是更好？

因此，这里我的理解是：上述精确查找只是模糊查找一个特例，模糊查找 hashmap 显然做不到，并且如果在精确查找问题中，hashmap 出现过多冲突，效率还不一定比 Trie 高，有兴趣的朋友可以做一下测试，看看哪个快。

再比如给你一个长句和一堆敏感词，找出长句中所有敏感词出现的所有位置（想下，有时候我们口吐芬芳，结果发送出去却变成了****，懂了吧）

小提示：实际上 AC 自动机就利用了 trie 的性质来实现敏感词的匹配，性能非常好。以至于很多编辑器都是用的 AC 自动机的算法。

还有些其他场景，这里不过多讨论，有兴趣的可以 google 一下。

题目推荐

以下是本专题的六道题目的题解，内容会持续更新，感谢你的关注～

- [0208. 实现 Trie \(前缀树\)](#)
- [0211. 添加与搜索单词 - 数据结构设计](#)
- [0212. 单词搜索 II](#)
- [0472. 连接词](#)
- [648. 单词替换](#)
- [0820. 单词的压缩编码](#)
- [1032. 字符流](#)

总结

前缀树的核心思想是用空间换时间，利用字符串的公共前缀来降低查询的时间开销。因此如果题目中公共前缀比较多，就可以考虑使用前缀树来优化。

前缀树的基本操作就是插入和查询，其中查询可以完整查询，也可以前缀查询，其中基于前缀查询才是前缀树的灵魂，也是其名字的来源。

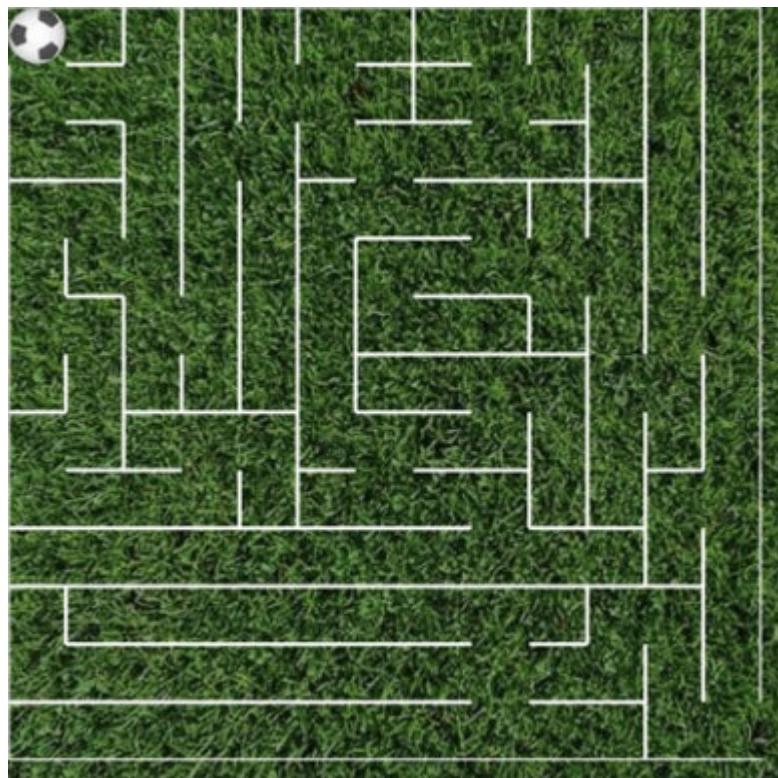
最后给大家提供了两种语言的前缀树模板，大家如果需要用，直接将其封装成标准 API 调用即可。

基于前缀树的题目变化通常不大，使用模板就可以解决。如何知道该使用前缀树优化是一个难点，不过大家只要牢牢记一点即可，那就是**算法的复杂度瓶颈在字符串查找，并且字符串有很多公共前缀，就可以用前缀树优化。**

并查集

背景

相信大家都玩过下面的迷宫游戏。你的目标是从地图的某一个角落移动到地图的出口。规则很简单，仅仅你不能穿过墙。



实际上，这道题并不能够使用并查集来解决。不过如果我将规则变成，“是否存在一条从入口到出口的路径”，那么这就是一个简单的联通问题，这样就可以借助本节要讲的并查集来完成。

另外如果地图不变，而不断改变入口和出口的位置，并依次让你判断起点和终点是否联通，并查集的效果高的超出你的想象。

另外并查集还可以在人工智能中用作图像人脸识别。比如将同一个人的不同角度，不同表情的面部数据进行联通。这样就可以很容易地回答两张图片是否是同一个人，无论拍摄角度和面部表情如何。

概述

并查集使用的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。

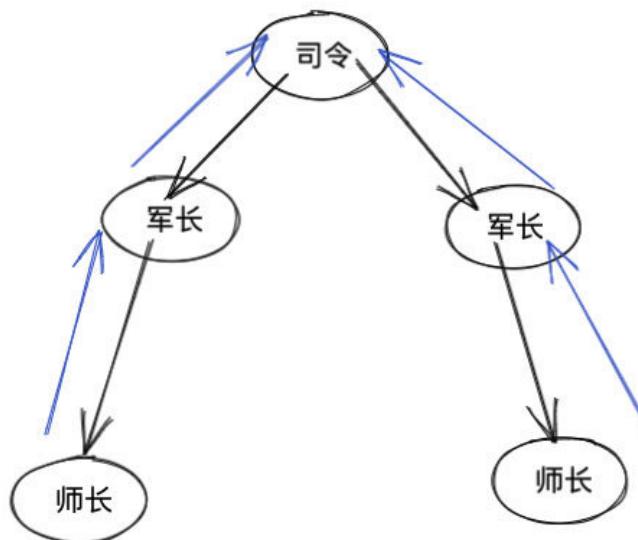
比如让你求两个人是否间接认识，两个地点之间是否有至少一条路径。上面的例子其实都可以抽象为联通性问题。即如果两个点联通，那么这两个点就有至少一条路径能够将其连接起来。值得注意的是，并查集只能回答“联通与否”，而不能回答诸如“具体的联通路径是什么”。如果要回答“具体的联通路径是什么”这个问题，则需要借助其他算法，比如广度优先遍历。

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。。。

判断两个节点是否联通

我们如何判断某两个师长是否归同一个司令管呢（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，那么两个人就归同一个司令管。（假设这两人级别比司令低）

如果我让你判断两个士兵是否归同一个师长管，也可以向上搜索到师长，如果搜索到的两个师长是同一个，那就说明这两个士兵归同一师长管。

（假设这两人级别比师长低）

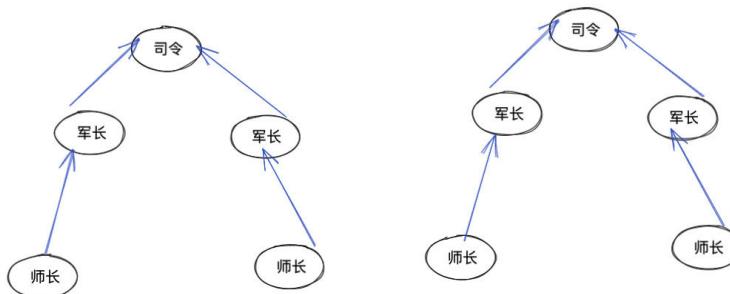
代码上我们可以用 `parent[x] = y` 表示 `x` 的 `parent` 是 `y`，通过不断沿着搜索 `parent` 搜索找到 `root`，然后比较 `root` 是否相同即可得出结论。这里的 `root` 实际上就是上文提到的集合代表。

之所以使用 `parent` 存储每个节点的父节点，而不是使用 `children` 存储每个节点的子节点是因为“我们需要找到某个元素的代表（也就是根）”

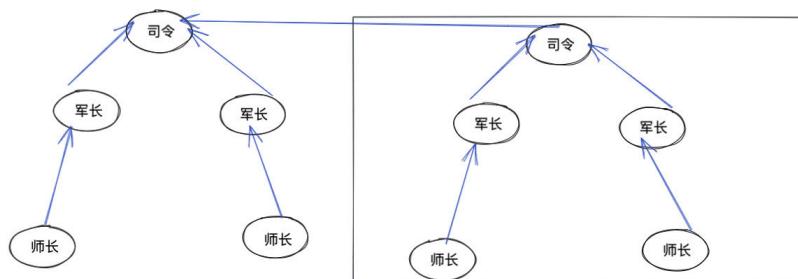
这个不断往上找的操作，我们一般称为 `find`，使用 `ta` 我们可以很容易地求出两个节点是否连通。

合并两个联通区域

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`，`connected` 和 `union`，的形象化解释，下面我们来看下代码实现。

核心 API

并查集（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。

首先我们初始化每一个点都是一个连通域，类似下图：



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进行合并。初始时，每个人的代表都是自己本身。

这里的代表就是上面的“司令”。

并查集元素一般用树来表示，树中的每个节点代表一个成员，每棵树表示一个集合，多棵树构成一个并查集森林。每个集合中，树根即其代表元。

```
interface Node {
    parent: Node;
}
```

比如我们的 `parent` 长这个样子：

```
{
  "0": "1",
  "1": "3",
  "2": "3",
  "4": "3",
  "3": "3"
}
```

find

假如我让你在上面的 `parent` 中找 0 的代表如何找？

首先，树的根 在 `parent` 中满足“`parent[x] == x`”。因此我们可以先找到 0 的父亲 `parent[0]` 也就是 1，接下来我们看 1 的父亲 `parent[1]` 发现是 3，因此它不是根，我们继续找 3 的父亲，发现是 3 本身。也就是说 3 就是我们要找的代表，我们返回 3 即可。

上面的过程具有明显的递归性，我们可以根据自己的喜好使用递归或者迭代来实现。

递归：

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

迭代：

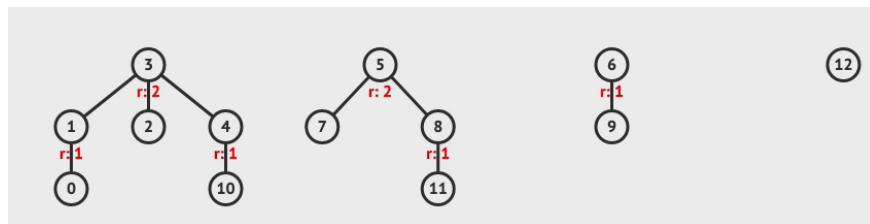
也可使用递归来实现。

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

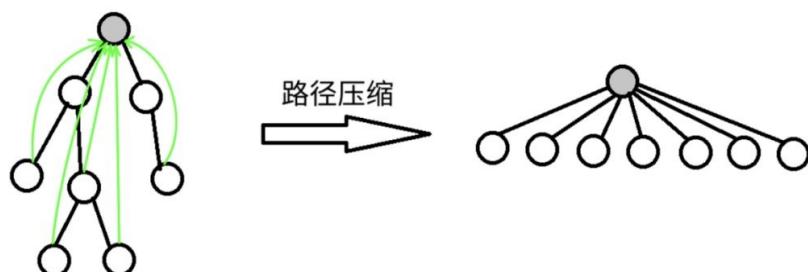
这里我在递归实现的 `find` 过程进行了路径的压缩，每次往上查找之后都会将树的高度降低到 2。

这有什么用呢？我们知道每次 `find` 都会从当前节点往上不断搜索，直到到达根节点，因此 `find` 的时间复杂度大致相等于节点的深度，树的高度如果不加控制则可能为节点数，因此 `find` 的时间复杂度可能会退化到 $O(n)$ 。而如果进行路径压缩，那么树的平均高度不会超过 $\log n$ ，如果使用了路径压缩和下面要讲的按秩合并那么时间复杂度可以趋近 $O(1)$ ，具体证明略。不过给大家画了一个图来辅助大家理解。

注意是趋近 $O(1)$ ，准确来说是阿克曼函数的某个反函数。



极限情况下，每一个路径都会被压缩，这种情况下继续查找的时间复杂度就是 $O(1)$ 。



connected

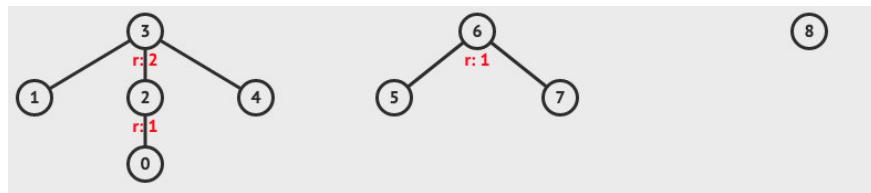
直接利用上面实现好的 `find` 方法即可。如果两个节点的祖先相同，那么其就联通。

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

union

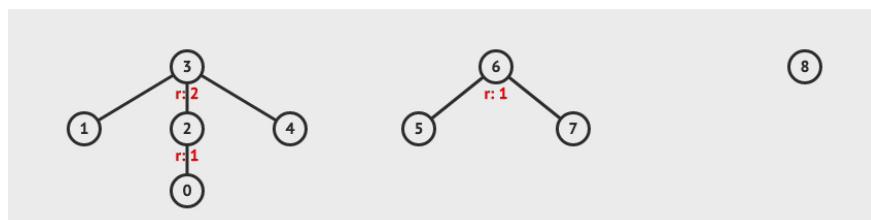
将其中一个节点挂到另外一个节点的祖先上，这样两者祖先就一样了。也就是说，两个节点联通了。

对于如下的一个图：



如果我们将 0 和 7 进行一次合并。即 `union(0, 7)`，则会发生如下过程。

- 找到 0 的根节点 3
- 找到 7 的根节点 6
- 将 6 指向 3。（为了使得合并之后的树尽可能平衡，一般选择将小树挂载到大树上面，下面的代码模板会体现这一点。3 的秩比 6 的秩大，这样更利于树的平衡，避免出现极端的情况）



上面讲的小树挂大树就是所谓的**按秩合并**。

代码：

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

这里我并没有判断秩的大小关系，目的是方便大家理清主脉络。完整代码见下面代码区。

不带权并查集

平时做题过程，遇到的更多的是不带权的并查集。相比于带权并查集，其实现过程也更加简单。

代码模板

```

class UF:
    def __init__(self, M):
        self.parent = {}
        self.size = {}
        self.cnt = 0
        # 初始化 parent, size 和 cnt
        # size 是一个哈希表, 记录每一个联通域的大小, 其中 key 是联通域的根节点
        # cnt 是整数, 表示一共有多少个联通域
        for i in range(M):
            self.parent[i] = i
            self.cnt += 1
            self.size[i] = 1

    def find(self, x):
        if x != self.parent[x]:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[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_q] += self.size[leader_p]
        else:
            self.parent[leader_q] = leader_p
            self.size[leader_p] += self.size[leader_q]
        self.cnt -= 1

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

```

带权并查集

上面讲到的其实都是有向无权图, 因此仅仅使用 `parent` 表示节点关系就可以了。而如果使用的是有向带权图呢? 实际上除了维护 `parent` 这样的节点指向关系, 我们还需要维护节点的权重, 一个简单的想法是使用另外一个哈希表 `weight` 存储节点的权重关系。比如 `weight[a] = 1` 表示 `a` 到其父节点的权重是 `1`。

如果是带权的并查集, 其查询过程的路径压缩以及合并过程会略有不同, 因为我们不仅关心节点指向的变更, 也关心权重如何更新。比如:

| | |
|---|---|
| a | b |
| ^ | ^ |
| | |
| | |
| x | y |

如上表示的是 x 的父节点是 a , y 的父节点是 b , 现在我需要将 x 和 y 进行合并。

| | |
|---|------|
| a | b |
| ^ | ^ |
| | |
| | |
| x | -> y |

假设 x 到 a 的权重是 $w(xa)$, y 到 b 的权重为 $w(yb)$, x 到 y 的权重是 $w(xy)$ 。合并之后会变成如图的样子:

| | |
|---|------|
| a | -> b |
| ^ | ^ |
| | |
| | |
| x | y |

那么 a 到 b 的权重应该被更新为什么呢? 我们知道 $w(xa) + w(ab) = w(xy) + w(yb)$, 也就是说 a 到 b 的权重 $w(ab) = w(xy) + w(yb) - w(xa)$ 。

当然上面关系式是加法, 减法, 取模还是乘法, 除法等完全由题目决定, 我这里只是举了一个例子。不管怎么样, 这种运算一定需要满足可传导性。

代码模板

这里以加法型带权并查集为例, 讲述一下代码应该如何书写。

```

class UF:
    def __init__(self, M):
        # 初始化 parent, weight
        self.parent = {}
        self.weight = {}
        for i in range(M):
            self.parent[i] = i
            self.weight[i] = 0

    def find(self, x):
        if self.parent[x] != x:
            ancestor, w = self.find(self.parent[x])
            self.parent[x] = ancestor
            self.weight[x] += w
        return self.parent[x], self.weight[x]

    def union(self, p, q, dist):
        if self.connected(p, q): return
        leader_p, w_p = self.find(p)
        leader_q, w_q = self.find(q)
        self.parent[leader_p] = leader_q
        self.weight[leader_p] = dist + w_q - w_p

    def connected(self, p, q):
        return self.find(p)[0] == self.find(q)[0]

```

典型题目：

- 399. 除法求值

复杂度分析

令 n 为图中点的个数。

首先分析空间复杂度。空间上，由于我们需要存储 parent （带权并查集还有 weight ），因此空间复杂度取决于于图中的点的个数，空间复杂度不难得出为 $O(n)$ 。

并查集的时间消耗主要是 union 和 find 操作，路径压缩和按秩合并优化后的时间复杂度接近于 $O(1)$ 。更加严谨的表达是 $O(\log(m \times \text{Alpha}(n)))$ ， n 为合并的次数， m 为查找的次数，这里 Alpha 是 Ackerman 函数的某个反函数。但如果只有路径压缩或者只有按秩合并，两者时间复杂度为 $O(\log x)$ 和 $O(\log y)$ ， x 和 y 分别为合并与查找的次数。

应用

- 检测图是否有环

思路：只需要将边进行合并，并在合并之前判断是否已经联通即可，如果合并之前已经联通说明存在环。

代码：

```
uf = UF()
for a, b in edges:
    if uf.connected(a, b): return False
    uf.union(a, b)
return True
```

题目推荐：

- [684. 冗余连接](#)
- [Forest Detection](#)
- [最小生成树经典算法 Kruskal](#)

练习

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](#)
- [1697. 检查边长度限制的路径是否存在](#)

上面的题目前面四道都是无权图的连通性问题，第五道题是带权图的连通性问题。两种类型大家都要会，上面的题目关键字都是连通性，代码都是套模板。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。

总结

如果题目有连通、等价的关系，那么你就可以考虑并查集，另外使用并查集的时候要注意路径压缩，否则随着树的高度增加复杂度会逐渐增大。

对于带权并查集实现起来比较复杂，主要是路径压缩和合并这块不一样，不过我们只要注意节点关系，画出如下的图：

```
a -> b
^   ^
|   |
|   |
x   y
```

就不难看出应该如何更新拉。

本文提供的题目模板是西法我用的比较多的，用了它不仅出错概率大大降低，而且速度也快了很多，整个人都更自信了呢 ^_^

参考

1. 算法导论
2. [维基百科](#)
3. [并查集详解 ——图文解说,简单易懂\(转\)](#)
4. [并查集专题](#)

剪枝

简介

关于剪枝这个概念，有的同学对机器学习有一定了解的肯定能脱口而出：“剪枝是为了解决决策树过拟合，为了降低模型复杂度的一种手段。”

而我们这次要介绍的剪枝又何尝不是为了降低我们所写程序的时空复杂度呢？

我们在日常编程中或多或少都用到了剪枝，只不过大家没有系统去了解过这块的概念而已，所以希望通过这个专题将大家对剪枝中模糊不清对概念有一个比较清晰的认识。

剪枝的概念

剪枝最常出现在搜索相关的问题上，我们常用的搜索算法，其实描绘出的搜索空间就是一个树形结构，我们成为搜索树。

算法中的剪枝指的是在搜索过程中，提前退出根本不可能是答案的决策分支，进而减少时空复杂度。由于我们提前退出了，因此搜索树的规模自然就会变小，形象地看像是一棵树被我们减掉了。日常生活中剪枝剪的就是树的枝桠，在搜索树中剪枝剪掉的就是必得不到解的子树来减小搜索空间。

算法中的剪枝和现实生活这工人剪枝是非常类似的，只不过目的不同。我们是为了提高算法性能，而现实中则是为了植物更好生长，亦或是为了美观。

那如何剪枝？常见的剪枝策略又是什么？

剪枝遵循的三原则

- 正确性：这个很好理解，我们把这个树权剪掉的前提是剪掉的这块一定不存在我们所要搜寻的解，不然我们把正确结果都剪没了。这对应上文的减掉的是必得不到解的子树。
- 准确性：我们在保证正确性的前提下，尽可能多的剪掉不包含所搜寻解的枝叶，也就是咱们剪，就要努力剪到最好。这里的关键是尽可能多。
- 高效性：这个就是一个衡量我们剪枝是否必要的一个标准了，比如我们设计出了一个非常优秀的剪枝策略，可以把搜索规模控制在非常小范围，很棒！但是我们去实现这个剪枝策略的时候，又耗费了大量的

时间和空间，是不是有点得不偿失呢？也就是我们需要在算法的整体效率和剪枝策略之间 trade-off。

常用的剪枝策略

- 可行性剪枝：如果我们当前的状态已经不合法了，我们也没有必要继续搜索了，直接把这块搜索空间剪掉，也就是 return。比如题目让我们求某个数组的子集，要求子集个数不大于 5。那么当我们暴力回溯的时候，如果已经选择的数字大于 5 了，那么就没必须继续选择了，可以提前退出。实际中的可行性剪枝则更加复杂，需要根据题目进行分析。**这是搜索回溯问题的难点**

eg:

```
def dfs(pos, path):
    # 从题目中挖掘非法状态，在这里判断，提前退出以达到剪枝的目的
    if 不合法: return
    # go deeper
```

- 记忆化：常做 dp 题的同学应该也知道，我们把已经计算出来的问题答案保存下来，下次遇到该问题就可以直接取答案而不用重复计算。这为什么叫剪枝呢？试想，如果你不使用记忆化，那么搜索树的规模则很大（通常是指数），而使用了记忆化则可最好可优化到多项式，这时搜索树的规模是不是变小了？这就是剪枝。

eg:

```
memo = {}
def dp(i):
    if i <= 0: return 0
    ans = max(dp(i-1), dp(i-2))
    memo[i] = ans
    return ans
```

- 搜索顺序剪枝：在我们已知一些有用的先验信息的前提下，定义我们的搜索顺序。举个最简单例子，有时候我们正序遍历数组遇到答案返回，这种解法会 TLE，但是，我们倒着遍历却过了，这就是对搜索顺序进行剪枝。再比如回溯法解决背包问题，当我们先在背包中放入一个较大的物品，那么剩余的搜索空间就更小，选择也更少。这比先放小的，最后放大的（发现放不下）在很多情况下都快。**力扣中的很多回溯题都使用了这个思路，值得大家重点关注。**

背包问题应该使用 dp 来做，这里只是举个例子。

eg:

```
nums.sort(reverse=True)

def knapsack(pos):
    pass
knapsack(0)
```

- 最优性剪枝：也叫上下边界剪枝，Alpha-Beta 剪枝，常用于对抗类游戏。当算法评估出某策略的后续走法比之前策略的还差时，就会剪掉该策略的后续发展。
- 等等。

用好剪枝，会让我们的算法事半功倍，所以大家一定要掌握剪枝这一强有力的思想。

总结

练习剪枝最有利的方式就是通过回溯法。由于回溯本质就是暴力穷举，因此如果不剪枝很可能会超时。毫不夸张地说，剪枝剪地好，回溯 AC 少不了。

比如 N 皇后问题，如果暴力穷举就是 N^N ，如果使用剪枝则可大大减少时间，可以减少到 $N!$$ 。推荐一篇 N 皇后的文章给大家 https://old-panda.com/2020/12/12/eight-queens-puzzle/?hmsr=toutiao.io&utm_medium=toutiao.io&utm_source=toutiao.io

字符串匹配专题

字符串匹配算法（String matching algorithms）是字符串问题下的一个搜索问题，因此它本质是一个搜索问题，不过和我们前面讲的 BFS, DFS，回溯不太一样，这些侧重的是对搜索树进行搜索，即状态空间是非线性的。而本章的状态空间是线性的。

什么是字符串匹配

字符串匹配用来在一长字符串或文章中，找出其是否包含某一个或多个字符串以及其位置的算法。该算法的应用非常广泛，比如：生物基因匹配、信息检索（比如编辑器的 $ctrl + f$ 搜索功能）等。

用数学语言描述如下：

假设 \$T\$ 是一个长度为 \$n\$ 的文本串，\$P\$ 是长度为 \$m\$ 的模式串。如果有 \$0 \leq s \leq n-m\$，使得 \$T[s, s+1, \dots, s+m]\$ 等于 \$P\$，则称 \$P\$ 在 \$T\$ 中出现且位移为 \$s\$。

比如 \$T\$ 为 "lucifer", "lu"（开始位置为 0），"cifer"（开始位置为 2）在 \$T\$ 中出现过。而 "xifa", "hello" 等没有在 \$T\$ 中出现过。

字符串匹配常见算法

暴力(Brute Force)

在日常编码生活中，我们肯定会遇到类似的问题，大部分数据量其实不大，串长也较短，因此自然会想到窗口大小固定的滑动窗口找出所有子串并依次和模式串按字母顺序依次比对看是否匹配并记录。

这里的窗口大小就是模式串的长度

Brute Force 有时候也简称为 BF。其核心思路非常朴素。该算法可以抽象为以下几步：

1. 非法情况处理，如模式串长度大于待匹配串等（防御性编程）
2. 初始化大小为模式串长的滑窗
3. 固定当前窗口，将当前窗口的子串与模式串匹配
 - 2.1. 若匹配成功，则记录相关信息，如该位置下标
 - 2.2. 否则窗口向后移动一格

伪代码

```

n = length[T]
m = length[P]

for s = 0 to n - m
    do if T[s..s+m] == P
        save info

```

时间复杂度 $O(n*m)$, 空间复杂度 $O(1)$

这种暴力算法对于数据规模小，串短的问题已经足够了，但是很多场景下数据规模很大，那暴力算法就显得捉襟见肘了，毕竟时间复杂度摆在那里，响应时间过长。

我们稍加分析其实不难发现，我们在每次窗口后移一位进行匹配的时候，实际上是把上一个窗口的所有状态信息全部都丢掉不要了，这会造成信息的浪费，那么都有哪些常见且优秀的解决方案呢？

核心其实还是前面讲滑动窗口提到的变化仅仅是窗口边缘，窗口中间不变。这就是 RK 算法的本质。

Rabin-Karp 算法(RK)

核心思路

RK 算法主要是对 T 中每个长度为 m 的子字符串 $T[s..s+m]$ 进行 hash 运算，生成 hash 值 h_1 ，对 P 进行 hash 运算，生成 hash 值 h_2 ，比对 h_1 和 h_2 ，如果两个 hash 值(不考虑冲突)相等，则判断 P 在 T 中出现，且位移为 s 。

该算法需要计算 P 一次哈希，并计算 $T[n-m+1]$ 次哈希。而计算哈希的复杂度和被计算哈希的字符串长度线性相关，每步 hash 的时间复杂度为 $O(m)$ 。在这里被计算哈希的字符串长度为 m ，因此如果仅仅使用哈希，而不对算法进行任何其他优化的话总的时间复杂度和前面提到的暴力法一样，最后的整体复杂度和 BF 一样都为 $O(m*n)$ 。

RK 算法妙在滑动窗口的时候，设计了一个适合的哈希函数，有效保留了上一个状态的部分信息，这样第一次计算子串 hash 值时间复杂度为 $O(m)$ ，而后续就可以达到 $O(1)$ ，这就是有效利用了前面计算的窗口信息，而不是全盘计算，这不就是滑动窗口的精髓么？因此 RK 算法最终的时间复杂度就降为 $O(m+n)$ 。

该方法可以抽象为以下几步：

1. 非法情况处理，如模式串长度大于待匹配串等（防御性编程）
2. 计算出模式串 hash 值
3. 初始化大小为模式串长的滑窗并计算出 hash 值，判断当前 hash 值是否和模式串 hash 值相等。
 - 2.1. 若相等，则记录相关信息，如该位

置下标 2.2. 否则窗口向后移动一格，并再次计算 hash 值（此处利用上个状态可直接一步计算）

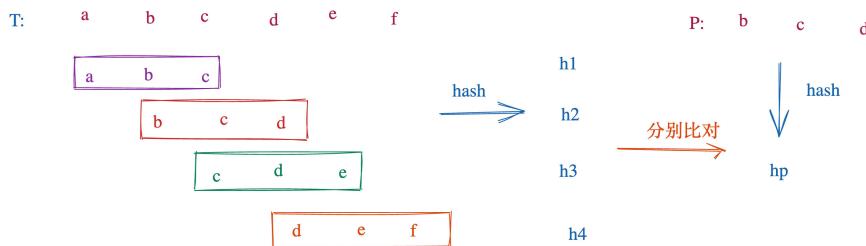
伪代码

```

n = length[T]
m = length[P]
hp = hash(P)

for s = 0 to n - m
    hs = hash(T[s..s+m])
    if hp == hs and double check is right
        save info

```



我们这里选取的哈希函数为： $f(P)=P$ 表示的10进制值\$。

接下来的题目中，哈希函数可能会有所不同，比如 26 进制值来计算仅含 26 个英文字母的字符串的哈希。

假设\$P\$和\$T\$全由\$d\$个字符组成的（也就是说字符集大小为 \$d\$），则我们可以选择\$d\$进制表示\$P\$和\$T\$，再将\$d\$进制转为\$10\$进制便于计算。

为了简化说明，我们更特殊地假设\$P\$和\$T\$全由[0-9]10 个数字组成。

\$P\$的 10 进制为：

$$f(P) = P[0] * 10^{(m-1)} + P[1] * 10^{(m-2)} \dots + P[1]$$

接下来，我们代入上面的哈希函数计算 \$T\$ 的所有子串的哈希值。

\$T[s..s+m]\$的 10 进制为：

$$f(T[s..s+m]) = T[s] * 10^{(m-1)} + T[s+1] * 10^{(m-2)} \dots + T[m]$$

相应地，下一个字符串\$T[s+1..s+m+1]\$的哈希值可以根据前面计算好的\$T[s..s+m]\$的哈希值推导出来。即加上窗口中新增加的字符的哈希值贡献，再减去窗口中移除的字符的哈希值贡献

该哈希算法是将字符串的每一个位的字符转换成对应的数字，再根据一定的权重（这里是 10）相乘得到一个数值。

具体过程如下：

```
f(T[s+1..s + m + 1]) = T[s+1] * 10 ^ (m - 1) + T[s + 2] * :
= (f(T[s..s + m]) - T[s] * 10 ^ (m - 1)) * 10 + T[s + m + :
```

这样就把以上 $hp == hs$ 的哈希比较转化为正常的 10 进制比较。

到目前为止，以上假设我们回避的一个问题是如果 $f(P)$ 或者 $f(T)$ 计算的 10 进制过大，导致运算溢出怎么办？比如我们用 32 位整形存储哈希值，那么如果计算的哈希值超过了 32 位整形所能表示的上限怎么办？

即使在 Python 等支持大数的语言，我们不会遇到溢出问题。这种算法同样有问题，因为这种算法性能会随着计算的哈希值变大而降低。

如何解决溢出和性能问题呢？

这里我们通过选择一个比较大的素数 q ，计算后的 10 进制数对 q 取模后再进行比较。由于我们使用对 q 取模，因此理论上必然会有冲突的情况，并且冲突的平均个数 n/q 。

取模的作用是防止溢出以及提供性能。但是有冲突的可能，
 $f(P) \% q == f(T[s]) \% q$ 并不能代表 $f(P) == f(T[s])$ 。因此任何的
 $f(P) \% q == f(T[s]) \% q$ 都需要额外进行再次验证，这里我们通过检测
 $P == T[s..s+m]$ 来完成。

从这里有可以看出，如果我们的代码冲突很多，那么效率和暴力法相差无几。极端情况，每次都哈希冲突，那么算法效率和暴力解一样，如果算法计算哈希值的开销，那么效率会比暴力法更差。

因此设计一个好的哈希算法，减少冲突是关键。如果你是参加面试或者比赛则可以尝试不同的模数来调参，直到通过所有的测试用例。不过这种做法非常不建议在工程中使用，即使在面试中，也希望你能先和面试官进行沟通之后再做决定。

题目推荐：

- [5803. 最长公共子路径](#)

总的来说，RK 算法就是：哈希函数 + 滑动窗口。使用哈希函数使得我们后续可以在 $O(1)$ 的时间计算字符串的哈希值。

KMP

建议先观看文末扩展部分的视频后再来读讲义。

KMP 本质上是个预处理 + dp。

- 预处理指的是经过这样的处理一个模式串会生成一个 **唯一的 next 数组**，从而可以去匹配任意的主串。

唯一指的是给定模式串，生成的 next 数组就是确定的。和主串是没有任何关系的。

- dp 指的是建立 next 数组的部分使用到了动态规划的算法。

而匹配的过程，赵丽算法中主串会有很多回溯，使用 KMP 可以**避免主串回溯，而只回溯模式串**。形象地看就是模式串不停地在对齐主串。

核心思路

首先我们定义模式串的前缀函数 $f(i)$ 为 模式串 P 中 $P[1...i]$ 相同前缀后缀的最大长度。对 $P[1...m]$ 中的每个 i , ($i > 0 \&& i \leq m$), 用一个数组 $next$ 记录。KMP 算法由 Knuth, Morris 和 Pratt 三个大佬联合发明，KMP 算法名字由三个大佬名字首位字符组成。

首先我们定义模式串的函数\$f(i)\$为模式串\$P\$中\$P[:i]\$相同前缀后缀的最大长度。对\$P\$中的每个\$i\$的信息，用一个数组\$next\$统一记录。KMP 算法在每次失配后，会根据上一次的比对信息跳转到相应的\$s\$处，借助的就是上述的\$next\$数组。推导过程可以参考[从头到尾彻底理解 KMP](#)，个人觉得这篇讲的非常透彻，这里就不班门弄斧了。该方法可以抽象为以下几步：

- 非法情况处理，如模式串长度大于待匹配串等（防御性编程）
- 计算出模式串的 next 数组。
- 开始从待匹配串开始进行匹配
- 若匹配成功，则记录相关信息；若失配，则按 next 数组回退到上一个待匹配状态继续进行匹配

以下是计算\$next\$数组的伪代码

```
get_next(P):
    m = P.length
    使得 next 为长度为m的数组
    next[1] = 0
    k = 0
    for i = 2 to m
        while(k > 0 并且 P[k+1] != P[i])
            k = next[k]
        if P[k+1] == P[i]
            k = k + 1
        next[i] = k
    return next
```

以下是 KMP 的伪代码

```

KMP(T, P)
    n = T.length
    m = P.length
    next = getNext(P)
    q = 0
    for let i = 1 to n:
        while(q > 0 并且 P[q + 1] != T[i])
            q = next[q]
        if P[q + 1] == T[i]
            q = q + 1
        if (q == m)
            找到匹配位移 s = i

```

总结

字符串匹配本质就是求一个模式串是否在主串中出现过，以及出现的具体位置。我们本章讲解了字符串匹配的三种常见方法：暴力法，RK 算法和 KMP 算法。

其中

- 暴力法简单直接
- RK 算法就是前面滑动窗口专题的应用，核心在于哈希函数的选择
- KMP 则是前面动态规划的应用，核心在于 next 数组的生成。

虽然本章是讲字符串的匹配，但是你可以将其进行简单的扩展，以实现知识的迁移。

比如给你两个数组 A 和 B，其中 A 为 [1,2,3,4,5]，B 为 [2,8,9,10]。让你在 A 中找到 B 的最长相似数组。其中相似数组指的是：如果数组 A[i], A[i+1], ... A[j] 与 B[p], B[p+1], ... B[q]，满足 A[i] == B[p] + x, A[i+1] == B[p+1] + x, ..., A[j] == B[q] + x，其中 x 为任意数字。

这个题目可以将 A 和 B 做一个简单的变换，变换之后就可以看成是字符串匹配，即让你在 A 中找 B。

那么如何变化呢？答案是相邻项做差。

```

for i in range(1, len(A)):
    A[i] -= A[i-1]
    B[i] -= B[i-1]

```

经过这样的处理 A 和 B 就变为了：

```

A = [1,1,1,1,1]
B = [2,1,1,1]

```

989. 数组形式的整数加法

可以看出除了首项外，A 和 B 相似数组的值是相同的。

我们先不看 A 和 B 的首项，这样的话我们只要在 A 中找到 [1,1,1] 就行了，最后在考虑首项即可。

此时我们就可以使用本章的方法来解决，比如 RK 算法（选择一个合适的哈希函数即可）。

参考

- [维基百科](#)
- [从头到尾彻底理解 KMP](#)
- [Finite Automata algorithm for Pattern Searching](#)

扩展

基于有限自动机的字符串匹配

这个算法仅限了解即可，这里不做展开，感兴趣可以参考 [Finite Automata algorithm for Pattern Searching](#)

推荐学习视频(需翻墙)

- [油管 KMP 讲解](#)

堆

堆（英语：Heap）是计算机科学中的一种特别的树状数据结构。

一个核心

堆的问题核心点就一个，那就是**动态求极值**。动态和极值二者缺一不可。

求极值比较好理解，无非就是求最大值或者最小值，而动态却不然。比如要你求一个数组的第 k 小的数，这是动态么？这其实完全看你怎么理解。而在我们这里，这种情况就是动态的。

如何理解上面的例子是动态呢？

你可以这么想。由于堆只能求极值。比如能求最小值，但不能直接求第 k 小的值。

那我们是不是先求最小的值，然后将其出队。然后继续求最小的值，这个时候求的就是第 2 小了。如果要求第 k 小，那就如此反复 k 次即可。

这个过程，你会发现数据是在**动态变化的**，对应的就是堆的大小在变化

两种实现

上面简单提到了堆的几种实现。这里介绍两种常见的实现，一种是基于链表的实现- 跳表，另一种是基于数组的实现 - 二叉堆。

使用跳表的实现，如果你的算法没有经过精雕细琢，性能会比较不稳定，且在数据量大的情况下内存占用会明显增加。因此我们仅详细讲述二叉堆的实现，而对于跳表的实现，仅讲述它的基本原理，对于代码实现等更详细的内容由于比较偏就不在这里讲了。

跳表

跳表也是一种数据结构，因此 ta 其实也是服务于某种算法的。

跳表虽然在面试中出现的频率不大，但是在工业中，跳表会经常被用到。力扣中关于跳表的题目只有一个。但是跳表的设计思路值得我们去学习和思考。其中有很多算法和数据结构技巧值得我们学习。比如空间换时间的思想，比如效率的取舍问题等。

上面提到了应付插队问题是设计**堆**应该考虑的首要问题。堆的跳表实现是如何解决这个问题的呢？

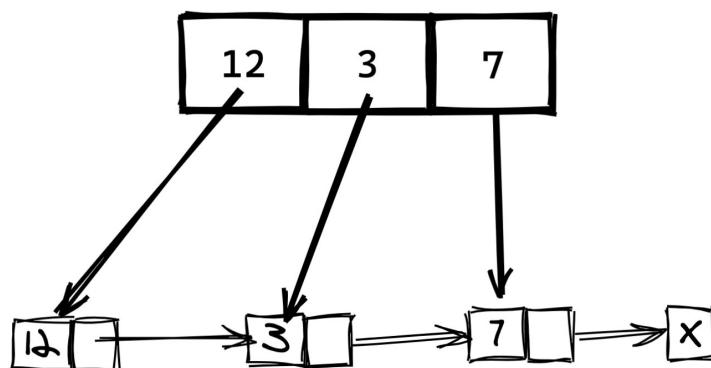
我们知道，不借助额外空间的情况下，在链表中查找一个值，需要按照顺序一个个查找，时间复杂度为 $O(N)$ ，其中 N 为链表长度。



(单链表)

当链表长度很大的时候，这种时间是很难接受的。一种常见的优化方式是建立哈希表，将所有节点都放到哈希表中，以空间换时间的方式减少时间复杂度，这种做法时间复杂度为 $O(1)$ ，但是空间复杂度为 $O(N)$ 。

hashtable



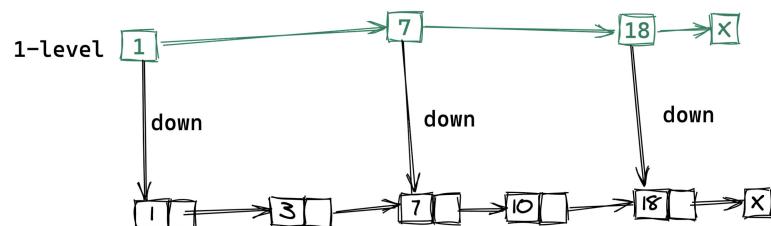
(单链表 + 哈希表)

为了防止链表中出现重复节点带来的问题，我们需要序列化节点，再建立哈希表，这种空间占用会更高，虽然只是系数级别的增加，但是这种开销也是不小的。更重要的是，哈希表不能解决查找极值的问题，其仅适合根据 key 来获取内容。

为了解决上面的问题，跳表应运而生。

如下图所示，我们从链表中每两个元素抽出来，加一级索引，一级索引指向了原始链表，即：通过一级索引 7 的 down 指针可以找到原始链表的 7。那怎么查找 10 呢？

注意这个算法要求链表是有序的。



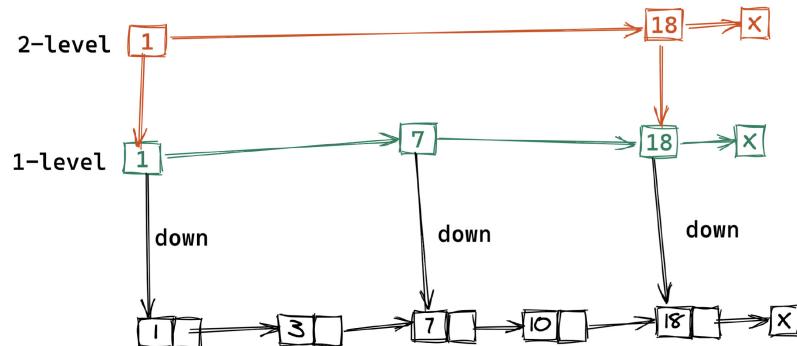
(建立一级索引)

我们可以：

- 通过现在一级跳表中搜索到 7，发现下一个 18 大于 10，也就是说我们要找的 10 在这两者之间。

- 通过 down 指针回到原始链表，通过原始链表的 next 指针我们找到了 10。

这个例子看不出性能提升。但是如果元素继续增大，继续增加索引的层数，建立二级，三级。。。索引，使得链表能够实现二分查找，从而获得更好的效率。但是相应地，我们需要付出额外空间的代价。



(增加索引层数)

理解了上面的点，你可以形象地将跳表想象为玩游戏的存档。

一个游戏有 10 关。如果我想要玩第 5 关的某一个地方，那么我可以直接从第五关开始，这样要比从第一关开始快。我们甚至可以在每一关同时设置很多的存档。这样我如果想玩第 5 关的某一个地方，也可以不用从第 5 关的开头开始，而是直接选择离你想玩的地方更近的存档，这就相当于跳表的二级索引。

跳表的时间复杂度和空间复杂度不是很好分析。由于时间复杂度 = 索引的高度 * 平均每层索引遍历元素的个数，而高度大概为 $\log n$ ，并且每层遍历的元素是常数，因此时间复杂度为 $\log n$ ，和二分查找的空间复杂度是一样的。

空间复杂度就等同于索引节点的个数，以每两个节点建立一个索引为例，大概是 $n/2 + n/4 + n/8 + \dots + 8 + 4 + 2$ ，因此空间复杂度是 $O(n)$ 。当然你如果每三个建立一个索引节点的话，空间会更省，但是复杂度不变。

理解了上面的内容，使用跳表实现堆就不难了。

- 入堆操作，只需要根据索引插到链表中，并更新索引（可选）。
- 出堆操作，只需要删除头部（或者尾部），并更新索引（可选）。

大家如果想检测自己的实现是否有问题，可以去力扣的[1206. 设计跳表](#) 检测。

接下来，我们看下一种更加常见的实现 - 二叉堆。

二叉堆

二叉堆的实现，我们仅讲解最核心的两个操作：heappop（出堆）和 heappush（入堆）。对于其他操作不再讲解，不过我相信你会了这两个核心操作，其他的应该不是难事。

实现之后的使用效果大概是这样的：

```

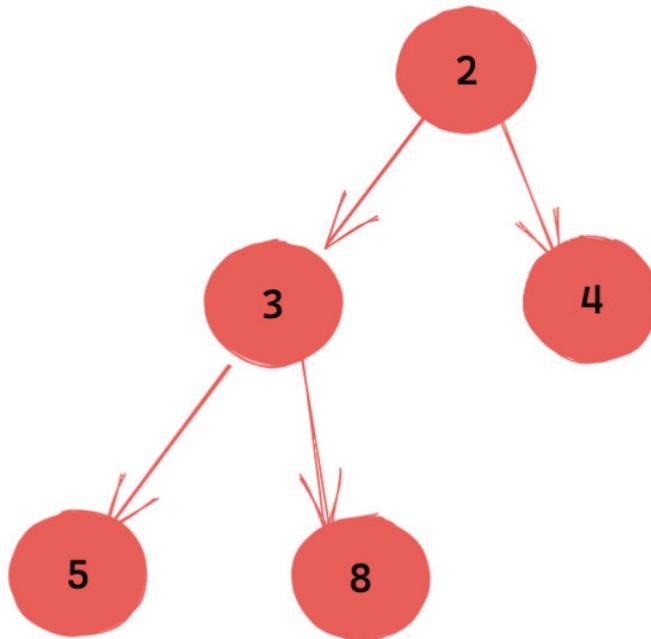
h = min_heap()
h.build_heap([5, 6, 2, 3])

h.heappush(1)
h.heappop() # 1
h.heappop() # 2
h.heappush(1)
h.heappop() # 1
h.heappop() # 3

```

基本原理

本质上来说，二叉堆就是一颗特殊的完全二叉树。它的特殊性只体现在一点，那就是父节点的权值不大于儿子的权值（小顶堆）。



(一个小顶堆)

上面这句话需要大家记住，一切的一切都源于上面这句话。

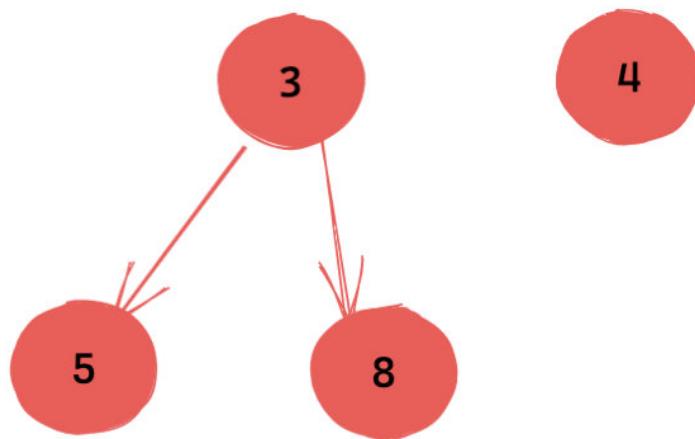
由于父节点的权值不大于儿子的权值（小顶堆），那么很自然能推导出树的根节点就是最小值。这就起到了堆的取极值的作用了。

那动态性呢？二叉堆是怎么做到的呢？

出堆

假如，我将树的根节点出堆，那么根节点不就空缺了么？我应该将第二小的顶替上去。怎么顶替上去呢？一切的一切还是那句话父节点的权值不大于儿子的权值（小顶堆）。

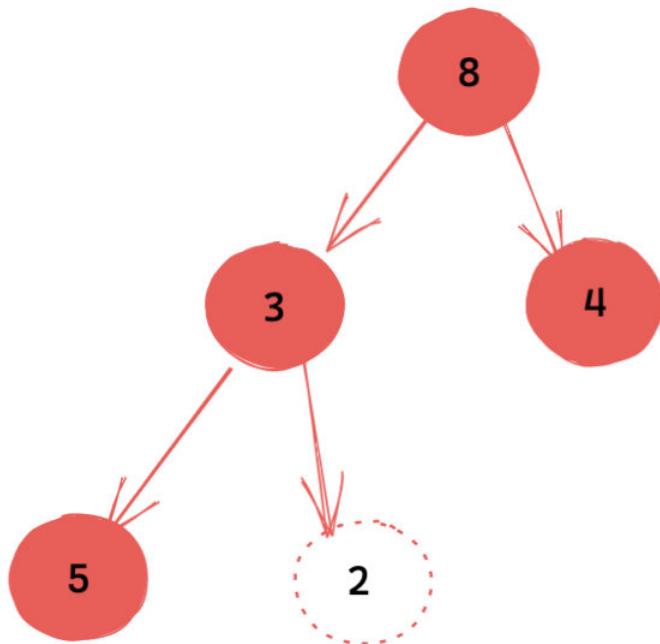
如果仅仅是删除，那么一个堆就会变成两个堆了，问题变复杂了。



(上图出堆之后会生成两个新的堆)

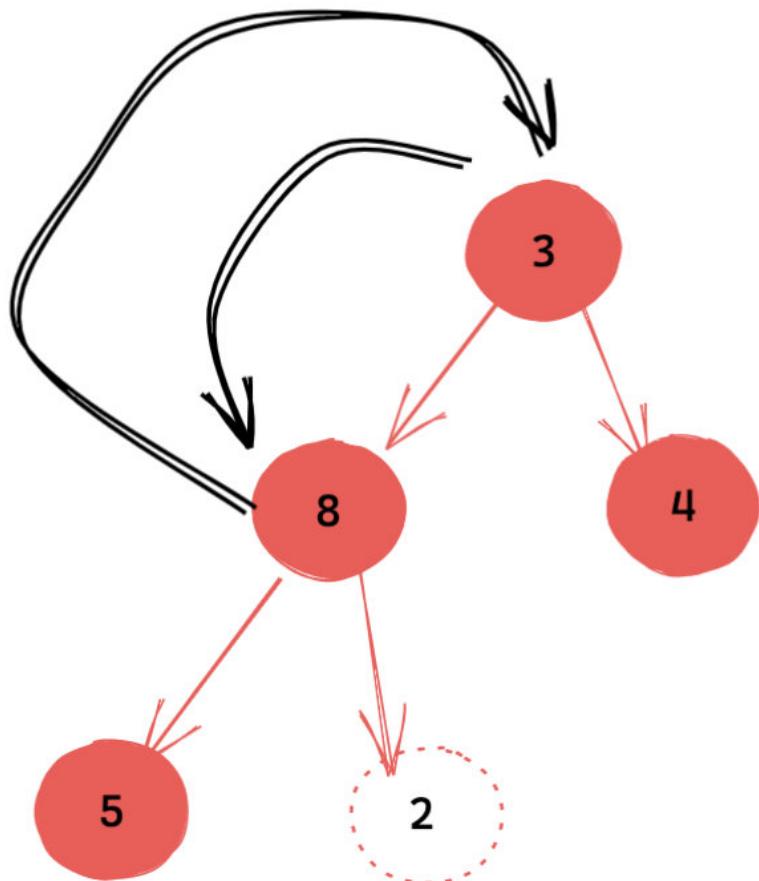
一个常见的操作是，把根结点和最后一个结点交换。但是新的根结点可能不满足父节点的权值不大于儿子的权值（小顶堆）。

如下图，我们将根节点的 2 和尾部的数字进行交换后，这个时候是不满足堆性质的。

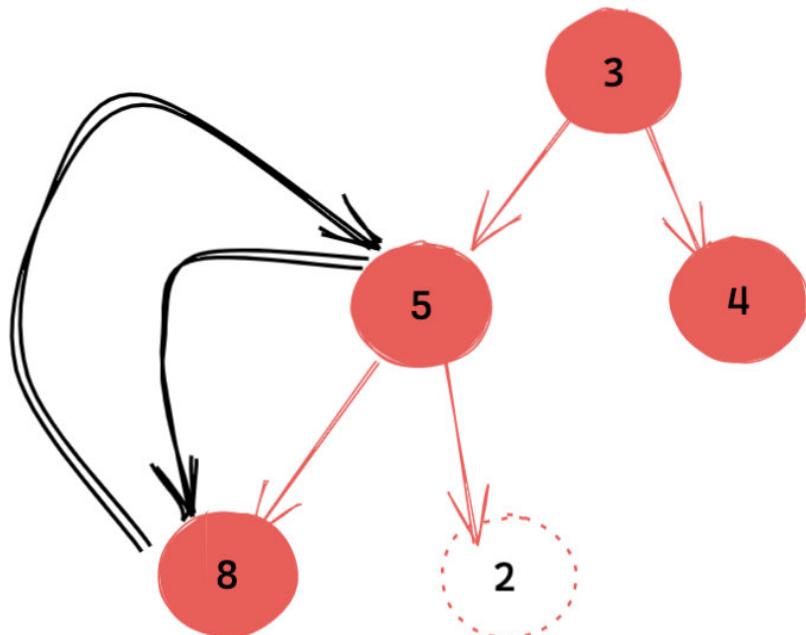


这个时候，其实只需要将新的根节点下沉到正确位置即可。这里的**正确位置**，指的还是那句话父节点的权值不大于儿子的权值（小顶堆）。如果不满足这一点，我们就继续下沉，直到满足。

我们知道根节点往下下沉的过程，其实有两个方向可供选择，是下沉到左子节点？还是下沉到右子节点？以小顶堆来说，答案应该是下沉到较小的子节点处，否则会错失正确答案。以上面的堆为例，如果下沉到右子节点4，那么就无法得到正确的堆顶3。因此我们需要下沉到左子节点。



下沉到如图位置，还是不满足 父节点的权值不大于儿子的权值（小顶堆），于是我们继续执行同样的操作。



有的同学可能有疑问。弹出根节点前堆满足堆的性质，但是弹出之后经过你上面讲的下沉操作，一定还满足么？

答案是肯定的。这个也不难理解。由于最后的叶子节点被提到了根节点，它其实最终在哪是不确定的，但是经过上面的操作，我们可以看出：

- 其下沉路径上的节点一定都满足堆的性质。
- 不在下沉路径上的节点都保持了堆之前的相对关系，因此也满足堆的性质。

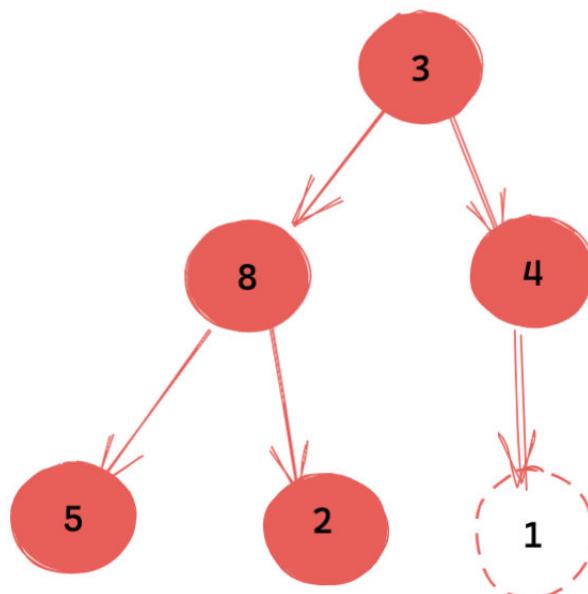
因此弹出根节点后，经过上面的下沉操作一定仍然满足堆的性质。

时间复杂度方面可以证明，下沉和树的高度成正相关，因此时间复杂度为 $O(h)$ ，其中 h 为树高。而由于二叉堆是一颗完全二叉树，因此树高大约是 $\log N$ ，其中 N 为树中的节点个数。

入堆

入堆和出堆类似。我们可以直接往树的最后插入一个节点。和上面类似，这样的操作同样可能会破坏堆的性质。

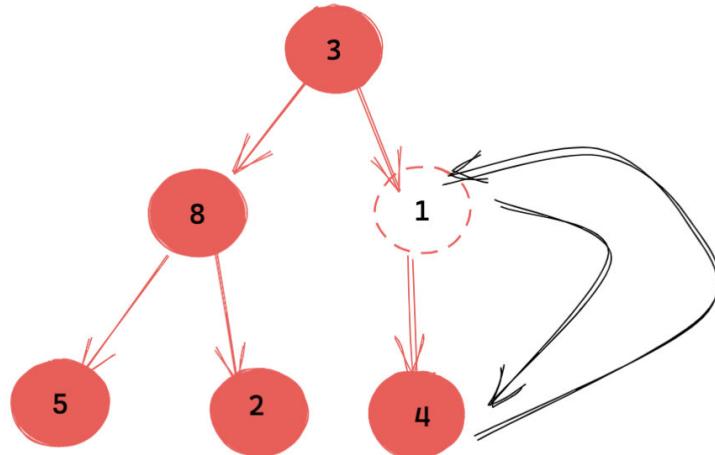
之所以这么做的其中一个原因是时间复杂度更低，因为我们是用数组进行模拟的，而在数组尾部添加元素的时间复杂度为 $O(1)$ 。



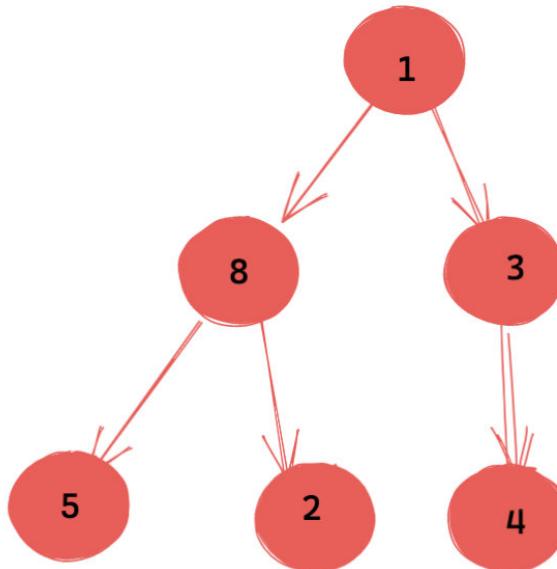
这次我们发现，不满足堆的节点目前是刚刚被插入节点的尾部节点，因此不能进行下沉操作了。这一次我们需要执行上浮操作。

叶子节点是只能上浮的（根节点只能下沉，其他节点既可以下沉，又可以上浮）

和上面基本类似，如果不满足堆的性质，我们将其和父节点交换（上浮），继续这个过程，直到满足堆的性质。



(第一次上浮，仍然不满足堆特性，继续上浮)



(满足了堆特性，上浮过程完毕)

经过这样的操作，其还是一个满足堆性质的堆。证明过程和上面类似，不再赘述。

需要注意的是，由于上浮只需要拿当前节点和父节点进行比对就可以了，由于省去了判断左右子节点哪个更小的过程，因此更加简单。

以数组 [1,2,3,4,5,6,7,8] 为例。我们如果对其进行：

1. 先堆化
2. 再逐个 pop，直到堆空。

那么整个过程可以用如下的动图来表示（建议大家多看几遍）。

动图上部分是二叉树的数组表示（可以理解为物理结构），下部分是二叉树的逻辑结构。

6 5 3 1 8 7 2 4

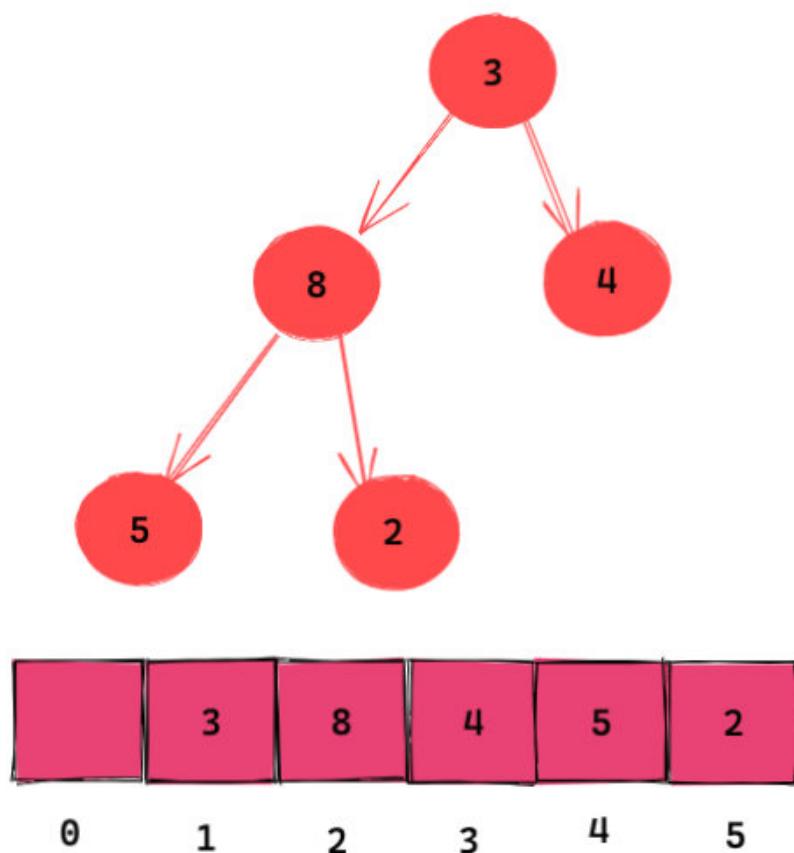
实现

对于完全二叉树来说使用数组实现非常方便。因为：

- 如果节点在数组中的下标为 i , 那么其左子节点下标为 $2 \times i$, 右节点为 $2 \times i + 1$ 。
- 如果节点在数组中的下标为 i , 那么父节点下标为 $i/2$ （地板除）。

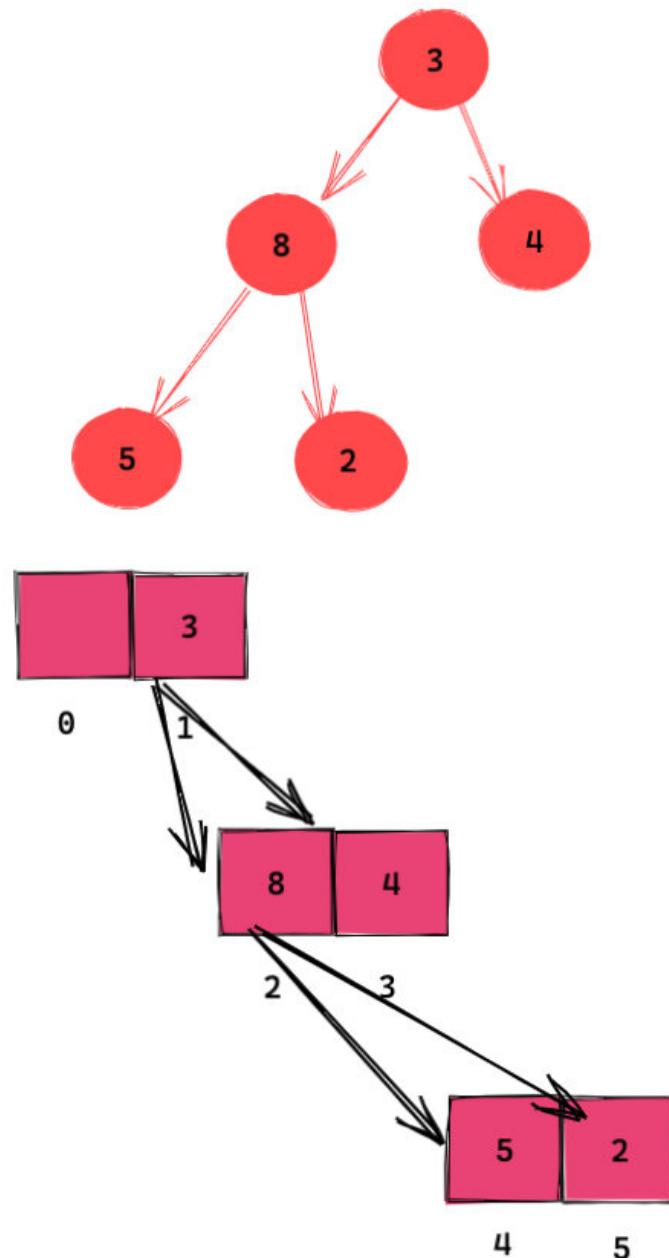
当然这要求你的数组从 1 开始存储数据。如果不是，上面的公式其实微调一下也可以达到同样的效果。不过这是一种业界习惯，我们还是和业界保持一致比较好。从 1 开始存储的另外一个好处是，我们可以将索引为 0 的位置空出来存储诸如堆大小的信息，这是一些大学教材里的做法，大家作为了解即可。

如图所示是一个完全二叉树和树的数组表示法。



(注意数组索引的对应关系)

形象点来看，我们可以画出如下的对应关系图：



这样一来，是不是和上面的树差不多一致了？有没有容易理解一点呢？

上面已经讲了上浮和下沉的过程。刚才也讲了父子节点坐标的关系。那么代码就呼之欲出了。我们来下最核心的上浮和下沉的代码实现吧。

伪代码：

989. 数组形式的整数加法

```
// x 是要上浮的元素，从树的底部开始上浮
private void shift_up(int x) {
    while (x > 1 && h[x] > h[x / 2]) {
        // swap 就是交换数组两个位置的值
        swap(h[x], h[x / 2]);
        x /= 2;
    }
}

// x 是要下沉的元素，从树的顶部开始下沉
private void shift_down(int x) {
    while (x * 2 <= n) {
        // minChild 是获取更小的子节点的索引并返回
        mc = minChild(x);
        if (h[mc] <= h[x]) break;
        swap(h[x], h[mc]);
        x = mc;
    }
}
```

这里 Java 语言为例，讲述一下代码的编写。其他语言的二叉堆实现可以去我的刷题插件 **leetcode-cheatsheet** 中获取。插件的获取方式在公众号 **力扣加加** 里，回复插件即可。

989. 数组形式的整数加法

```
import java.util.Arrays;
import java.util.Comparator;

/**
 * 用完全二叉树来构建 堆
 * 前置条件 起点为 1
 * 那么 子节点为 i <<1 和 i<<1 + 1
 * 核心方法为
 * shiftDown 交换下沉
 * shiftUp 交换上浮
 * <p>
 * build 构建堆
 */

public class Heap {

    public int size = 0;
    private int queue[];

    public Heap(int initialCapacity) {
        if (initialCapacity < 1)
            throw new IllegalArgumentException();
        this.queue = new int[initialCapacity];
    }

    public Heap(int[] arr) {
        size = arr.length;
        queue = new int[arr.length + 1];
        int i = 1;
        for (int val : arr) {
            queue[i++] = val;
        }
        this.buildHeap();
    }

    private void shiftDown(int i) {

        int temp = queue[i];

        while ((i << 1) <= size) {
            int child = i << 1;
            // child!=size 判断当前元素是否包含右节点
            if (child != size && queue[child + 1] < queue[child])
                child++;
            if (temp > queue[child]) {
                queue[i] = queue[child];
                i = child;
            }
        }
    }
}
```

989. 数组形式的整数加法

```
        } else {
            break;
        }
    }
    queue[i] = temp;
}

private void shiftUp(int i) {
    int temp = queue[i];
    while ((i >> 1) > 0) {
        if (temp < queue[i >> 1]) {
            queue[i] = queue[i >> 1];
            i >>= 1;
        } else {
            break;
        }
    }
    queue[i] = temp;
}

// 堆是否为空
public boolean isEmpty(){
    return size <= 0;
}

public int peek(){
    if(this.isEmpty()){
        throw new NullPointerException();
    }
    int res = queue[1];
    return res;
}

public int pop(){
    if(this.isEmpty()){
        throw new NullPointerException();
    }
    int res = queue[1];
    if(size < queue.length / 2){
        queue = Arrays.copyOf(queue, queue.length / 2);
    }
    queue[1] = queue[size--];
    shiftDown(1);
    return res;
}

public void push(int val) {
```

```

    if (size == queue.length - 1) {
        queue = Arrays.copyOf(queue, size << 1+1);
    }
    queue[++size] = val;
    shiftUp(size);
}

private void buildHeap() {
    for (int i = size >> 1; i > 0; i--) {
        shiftDown(i);
    }
}

public void display () {
    System.out.print("[");
    for(int i = 1; i <= size; i++) {
        System.out.print(queue[i] + ",");
    }
    System.out.print("\b]" + "\n");
}
}

```

小结

堆的实现有很多。比如基于链表的跳表，基于数组的二叉堆和基于红黑树的实现等。这里我们详细地讲述了二叉堆的实现，不仅是其实现简单，而且其在很多情况下表现都不错，推荐大家重点掌握二叉堆实现。

对于二叉堆的实现，核心点就一点，那就是始终维护堆的性质不变，具体是什么性质呢？那就是 **父节点的权值不大于儿子的权值（小顶堆）**。为了达到这个目的，我们需要在入堆和出堆的时候，使用上浮和下沉操作，并恰当地完成元素交换。具体来说就是上浮过程和比它大的父节点进行交换，下沉过程和两个子节点中较小的进行交换，当然前提是它有子节点且子节点比它小。

关于堆化我们并没有做详细分析。不过如果你理解了本文的入堆操作，这其实很容易。因此堆化本身就是一个不断入堆的过程，只不过将时间上的离散的操作变成了一次性操作而已。

三个技巧

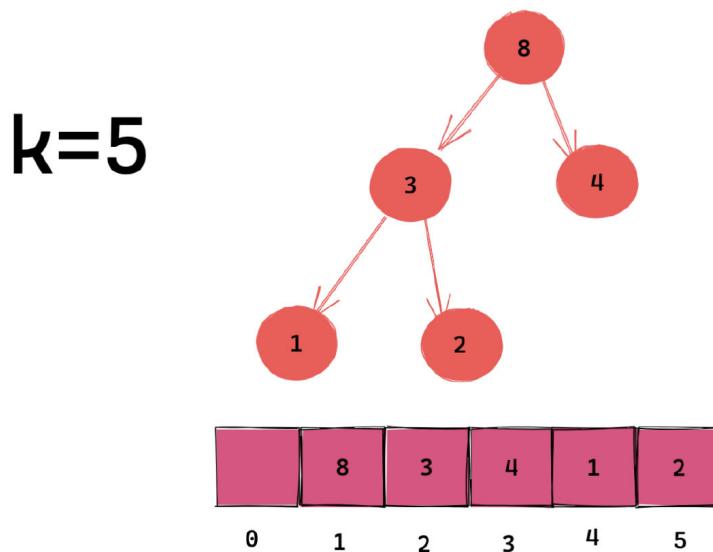
技巧一 - 固定堆

这个技巧指的是固定堆的大小 k 不变，代码上可通过每 **pop** 出去一个就 **push** 进来一个来实现。而由于初始堆可能是 0，我们刚开始需要一个一个 **push** 进堆以达到堆的大小为 k ，因此严格来说应该是维持堆的大小不大于 k 。

固定堆一个典型的应用就是求第 k 小的数。其实求第 k 小的数最简单的思路是建立小顶堆，将所有的数先全部入堆，然后逐个出堆，一共出堆 k 次。最后一次出堆的就是第 k 小的数。

然而，我们也可不先全部入堆，而是建立大顶堆（注意不是上面的小顶堆），并维持堆的大小为 k 个。如果新的数入堆之后堆的大小大于 k ，则需要将堆顶的数和新的数进行比较，并将较大的移除。这样可以保证堆中的数是全体数字中最小的 k 个，而这最小的 k 个中最大的（即堆顶）不就是第 k 小的么？这也就是选择建立大顶堆，而不是小顶堆的原因。

堆是最小的 k 个数，堆顶又是最大的，因此堆顶就是第 k 小的



简单一句话总结就是固定一个大小为 k 的大顶堆可以快速求第 k 小的数，反之固定一个大小为 k 的小顶堆可以快速求第 k 大的数。比如力扣 2020-02-24 的周赛第三题[5663. 找出第 K 大的异或坐标值](#)就可以用固定小顶堆技巧来实现（这道题让你求第 k 大的数）。

这么说可能你的感受并不强烈，接下来我给大家举两个例子来帮助大家加深印象。

295. 数据流的中位数

题目描述

中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。

例如，

[2,3,4] 的中位数是 3

[2,3] 的中位数是 $(2 + 3) / 2 = 2.5$

设计一个支持以下两种操作的数据结构：

`void addNum(int num)` – 从数据流中添加一个整数到数据结构中。

`double findMedian()` – 返回目前所有元素的中位数。

示例：

```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

进阶：

如果数据流中所有整数都在 0 到 100 范围内，你将如何优化你的算法？

如果数据流中 99% 的整数都在 0 到 100 范围内，你将如何优化你的算法？

思路

这道题实际上可看出是求第 k 小的数的特例了。

- 如果列表长度是奇数，那么 k 就是 $(n + 1) / 2$ ，中位数就是第 k 个数。比如 n 是 5， k 就是 $(5 + 1) / 2 = 3$ 。
- 如果列表长度是偶数，那么 k 就是 $(n + 1) / 2$ 和 $(n + 1) / 2 + 1$ ，中位数则是这两个数的平均值。比如 n 是 6， k 就是 $(6 + 1) / 2 = 3$ 和 $(6 + 1) / 2 + 1 = 4$ 。

因此我们可以维护两个固定堆，固定堆的大小为 $\lfloor (n + 1) / 2 \rfloor$ 和 $\lceil n - (n + 1) / 2 \rceil$ ，也就是两个堆的大小最多相差 1，更具体的就是 $0 \leq (n + 1) / 2 - (n - (n + 1) / 2) \leq 1$ 。

基于上面提到的知识，我们可以：

- 建立一个大顶堆，并存放最小的 $\lfloor (n + 1) / 2 \rfloor$ 个数，这样堆顶的数就是第 $\lfloor (n + 1) / 2 \rfloor$ 小的数，也就是奇数情况的中位数。
- 建立一个小顶堆，并存放最大的 $n - \lfloor (n + 1) / 2 \rfloor$ 个数，这样堆顶的数就是第 $n - \lfloor (n + 1) / 2 \rfloor$ 大的数，结合上面的大顶堆，可求出偶数情况的中位数。

有了这样一个知识，剩下的只是如何维护两个堆的大小了。

- 如果大顶堆的个数比小顶堆少，那么就将小顶堆中最小的转移到大顶堆。而由于小顶堆维护的是最大的 k 个数，大顶堆维护的是最小的 k 个数，因此小顶堆堆顶一定大于等于大顶堆堆顶，并且这两个堆顶是此时的中位数。
- 如果大顶堆的个数比小顶堆的个数多 2，那么就将大顶堆中最大的转移到小顶堆，理由同上。

至此，可能你已经明白了为什么分别建立两个堆，并且需要一个大顶堆一个小顶堆。这其中的原因正如上面所描述的那样。

固定堆的应用常见还不止于此，我们继续看一道题。

代码

```
class MedianFinder:
    def __init__(self):
        self.min_heap = []
        self.max_heap = []
    def addNum(self, num: int) -> None:
        if not self.max_heap or num < -self.max_heap[0]:
            heapq.heappush(self.max_heap, -num)
        else:
            heapq.heappush(self.min_heap, num)
        if len(self.max_heap) > len(self.min_heap) + 1:
            heappush(self.min_heap, -heappop(self.max_heap))
        elif len(self.min_heap) > len(self.max_heap):
            heappush(self.max_heap, -heappop(self.min_heap))
    def findMedian(self) -> float:
        if len(self.min_heap) == len(self.max_heap): return
        return -self.max_heap[0]
```

技巧二 - 多路归并

这个技巧其实在前面讲超级丑数的时候已经提到了，只是没有给这种类型的题目一个名字。

其实这个技巧，叫做多指针优化可能会更合适，只不过这个名字实在太过于朴素且容易和双指针什么的混淆，因此我给它起了个别致的名字 - 多路归并。

- 多路体现在：有多条候选路线。代码上，我们可使用多指针来表示。
- 归并体现在：结果可能是多个候选路线中最长的或者最短，也可能是第 k 个等。因此我们需要对多条路线的结果进行比较，并根据题目描述舍弃或者选取某一个或多个路线。

这样描述比较抽象，接下来通过几个例子来加深一下大家的理解。

这里我给大家精心准备了四道难度为 **hard** 的题目。掌握了这个套路就可以去快乐地 AC 这四道题啦。

1439. 有序矩阵中的第 k 个最小数组和

题目描述

给你一个 $m * n$ 的矩阵 mat , 以及一个整数 k , 矩阵中的每一行都以非递减

你可以从每一行中选出 1 个元素形成一个数组。返回所有可能数组中的第 k 个

示例 1:

输入: $\text{mat} = [[1,3,11],[2,4,6]]$, $k = 5$

输出: 7

解释: 从每一行中选出一个元素, 前 k 个和最小的数组分别是:

$[1,2]$, $[1,4]$, $[3,2]$, $[3,4]$, $[1,6]$ 。其中第 5 个的和是 7 。

示例 2:

输入: $\text{mat} = [[1,3,11],[2,4,6]]$, $k = 9$

输出: 17

示例 3:

输入: $\text{mat} = [[1,10,10],[1,4,5],[2,3,6]]$, $k = 7$

输出: 9

解释: 从每一行中选出一个元素, 前 k 个和最小的数组分别是:

$[1,1,2]$, $[1,1,3]$, $[1,4,2]$, $[1,4,3]$, $[1,1,6]$, $[1,5,2]$, $[1,5,3]$ 。

示例 4:

输入: $\text{mat} = [[1,1,10],[2,2,9]]$, $k = 7$

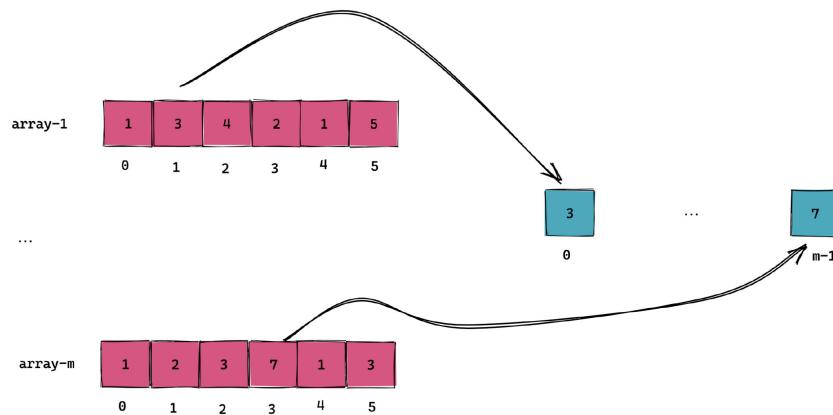
输出: 12

提示:

```
m == mat.length
n == mat.length[i]
1 <= m, n <= 40
1 <= k <= min(200, n ^ m)
1 <= mat[i][j] <= 5000
mat[i] 是一个非递减数组
```

思路

其实这道题就是给你 m 个长度均相同的一维数组，让我们从这 m 个数组中分别选出一个数，即一共选取 m 个数，求这 m 个数的和是所有选取可能性中和第 k 小的。



一个朴素的想法是使用多指针来解。对于这道题来说就是使用 m 个指针，分别指向 m 个一维数组，指针的位置表示当前选取的是该一维数组中第几个。

以题目中的 `mat = [[1,3,11],[2,4,6]]`, $k = 5$ 为例。

- 先初始化两个指针 $p1, p2$ ，分别指向两个一维数组的开头，代码表示就是全部初始化为 0。
- 此时两个指针指向的数字和为 $1 + 2 = 3$ ，这就是第 1 小的和。
- 接下来，我们移动其中一个指针。此时我们可以移动 $p1$ ，也可以移动 $p2$ 。
- 那么第 2 小的一定是移动 $p1$ 和 移动 $p2$ 这两种情况的较小值。而这里移动 $p1$ 和 $p2$ 实际上都会得到 5，也就是说第 2 和第 3 小的和都是 5。

到这里已经分叉了，出现了两种情况(注意看粗体的位置，粗体表示的是指针的位置)：

- [1,3,11],[2,4,6]** 和为 5
- [1,3,11],[2,4,6]** 和为 5

接下来，这两种情况应该齐头并进，共同进行下去。

对于情况 1 来说，接下来移动又有两种情况。

- [1,3,11],[2,4,6]** 和为 13
- [1,3,11],[2,4,6]** 和为 7

对于情况 2 来说，接下来移动也有两种情况。

- [1,3,11],[2,4,6]** 和为 7
- [1,3,11],[2,4,6]** 和为 7

我们通过比较这四种情况，得出结论：第 4, 5, 6 小的数都是 7。但第 7 小的数并不一定是 13。原因和上面类似，可能第 7 小的就隐藏在前面的 7 分裂之后的新情况中，实际上确实如此。因此我们需要继续执行上述逻辑。

进一步，我们可以将上面的思路拓展到一般情况。

上面提到了题目需要求的其实是第 k 小的和，而最小的我们是容易知道的，即所有的一维数组首项和。我们又发现，根据最小的，我们可以推导出第 2 小，推导的方式就是移动其中一个指针，这就一共分裂出了 n 种情况了，其中 n 为一维数组长度，第 2 小的就在这分裂中的 n 种情况中，而筛选的方式是这 n 种情况和最小的，后面的情况也是类似。不难看出每次分裂之后极值也发生了变化，因此这是一个明显的求动态求极值的信号，使用堆是一个不错的选择。

那代码该如何书写呢？

上面说了，我们先要初始化 m 个指针，并赋值为 0。对应伪代码：

```
# 初始化堆
h = []
# sum(vec[0] for vec in mat) 是 m 个一维数组的首项和
# [0] * m 就是初始化了一个长度为 m 且全部填充为 0 的数组。
# 我们将上面的两个信息组装成元祖 cur 方便使用
cur = (sum(vec[0] for vec in mat), [0] * m)
# 将其入堆
heapq.heappush(h, cur)
```

接下来，我们每次都移动一个指针，从而形成分叉出一条新的分支。每次从堆中弹出一个最小的，弹出 k 次就是第 k 小的了。伪代码：

```
for i to K:
    # acc 当前的和, pointers 是指针情况。
    acc, pointers = heapq.heappop(h)
    # 每次都粗暴地移动指针数组中的一个指针。每移动一个指针就分叉一次,
    for i, pointer in enumerate(pointers):
        # 如果 pointer == len(mat[0]) - 1 说明到头了, 不能移动
        if pointer != len(mat[0]) - 1:
            # 下面两句话的含义是修改 pointers[i] 的指针为 pointer + 1
            new_pointers = pointers.copy()
            new_pointers[i] += 1
            # 将更新后的 acc 和指针数组重新入堆
            heapq.heappush(h, (acc + mat[i][pointer + 1] -
```

这是多路归并问题的核心代码，请务必记住。

代码看起来很多，其实去掉注释一共才七行而已。

上面的伪代码有一个问题。比如有两个一维数组，指针都初始化为 0。第一次移动第一个一维数组的指针，第二次移动第二个数组的指针，此时指针数组为 [1, 1]，即全部指针均指向下标为 1 的元素。而如果第一次移动第二个一维数组的指针，第二次移动第一个数组的指针，此时指针数组仍然为 [1, 1]。这实际上是一种情况，如果不加控制会被计算两次导致出错。

一个可能的解决方案是使用 `hashset` 记录所有的指针情况，这样就避免了同样的指针被计算多次的问题。为了做到这一点，我们需要对指针数组的使用做一些微调，即使用元组代替数组。原因在于数组是无法直接哈希化的。具体内容请参考代码区。

多路归并的题目，思路和代码都比较类似。为了后面的题目能够更高地理解，请务必搞定这道题，后面我们将不会这么详细地进行分析。

代码

```
class Solution:
    def kthSmallest(self, mat, k: int) -> int:
        h = []
        cur = (sum(vec[0] for vec in mat), tuple([0] * len(mat)))
        heapq.heappush(h, cur)
        seen = set(cur)

        for _ in range(k):
            acc, pointers = heapq.heappop(h)
            for i, pointer in enumerate(pointers):
                if pointer != len(mat[0]) - 1:
                    t = list(pointers)
                    t[i] = pointer + 1
                    tt = tuple(t)
                    if tt not in seen:
                        seen.add(tt)
                        heapq.heappush(h, (acc + mat[i][pointer], t))

        return acc
```

技巧三 - 事后小诸葛



这个技巧指的是：当从左到右遍历的时候，我们是不知道右边是什么的，需要等到你到了右边之后才知道。

如果想知道右边是什么，一种简单的方式是遍历两次，第一次遍历将数据记录下来，当第二次遍历的时候，用上次遍历记录的数据。这是我们使用最多的方式。不过有时候，我们也可以在遍历到指定元素后，往前回溯，这样就可以边遍历边存储，使用一次遍历即可。具体来说就是将从左到右的数据全部收集起来，等到需要用的时候，从里面挑一个用。如果我们要取最大值或者最小值且极值会发生变动，就可使用堆加速。直观上就是使用了时光机回到之前，达到了事后诸葛亮的目的。

这样说你肯定不明白啥意思。没关系，我们通过几个例子来讲一下。当你看完这些例子之后，再回头看这句话。

1488. 避免洪水泛滥

题目描述

989. 数组形式的整数加法

你的国家有无数个湖泊，所有湖泊一开始都是空的。当第 n 个湖泊下雨的时候，

给你一个整数数组 `rains`，其中：

`rains[i] > 0` 表示第 i 天时，第 `rains[i]` 个湖泊会下雨。

`rains[i] == 0` 表示第 i 天没有湖泊会下雨，你可以选择一个湖泊并抽干。请返回一个数组 `ans`，满足：

`ans.length == rains.length`

如果 `rains[i] > 0`，那么 `ans[i] == -1`。

如果 `rains[i] == 0`，`ans[i]` 是你第 i 天选择抽干的湖泊。

如果有多种可行解，请返回它们中的任意一个。如果没办法阻止洪水，请返回 `-1`。

请注意，如果你选择抽干一个装满水的湖泊，它会变成一个空的湖泊。但如果你选择抽干一个空的湖泊，它会变成一个装满水的湖泊。

示例 1：

输入： `rains = [1,2,3,4]`

输出： `[-1,-1,-1,-1]`

解释： 第一天后，装满水的湖泊包括 `[1]`

第二天后，装满水的湖泊包括 `[1,2]`

第三天后，装满水的湖泊包括 `[1,2,3]`

第四天后，装满水的湖泊包括 `[1,2,3,4]`

没有哪一天你可以抽干任何湖泊的水，也没有湖泊会发生洪水。

示例 2：

输入： `rains = [1,2,0,0,2,1]`

输出： `[-1,-1,2,1,-1,-1]`

解释： 第一天后，装满水的湖泊包括 `[1]`

第二天后，装满水的湖泊包括 `[1,2]`

第三天后，我们抽干湖泊 `2`。所以剩下装满水的湖泊包括 `[1]`

第四天后，我们抽干湖泊 `1`。所以暂时没有装满水的湖泊了。

第五天后，装满水的湖泊包括 `[2]`。

第六天后，装满水的湖泊包括 `[1,2]`。

可以看出，这个方案下不会有洪水发生。同时，`[-1,-1,1,2,-1,-1]` 也是另一个可行解。

示例 3：

输入： `rains = [1,2,0,1,2]`

输出： `[]`

解释： 第二天后，装满水的湖泊包括 `[1,2]`。我们可以在第三天抽干一个湖泊的水。

但第三天后，湖泊 `1` 和 `2` 都会再次下雨，所以不管我们第三天抽干哪个湖泊的水，都会有洪水发生。

示例 4：

输入： `rains = [69,0,0,0,69]`

输出： `[-1,69,1,1,-1]`

解释： 任何形如 `[-1,69,x,y,-1]`, `[-1,x,69,y,-1]` 或者 `[-1,x,y,69]` 的数组都是可行解。

示例 5：

输入: rains = [10, 20, 20]

输出: []

解释: 由于湖泊 20 会连续下 2 天的雨, 所以没有办法阻止洪水。

提示:

$1 \leq \text{rains.length} \leq 10^5$

$0 \leq \text{rains}[i] \leq 10^9$

思路

如果上面的题用事后诸葛亮描述比较牵强的话, 那后面这两个题可以说很适合了。

题目说明了我们可以在不下雨的时候抽干一个湖泊, 如果有多个下满雨的湖泊, 我们该抽干哪个湖呢? 显然应该是抽干最近即将被洪水淹没的湖。但是现实中无论如何我们都不可能知道未来哪天哪个湖泊会下雨的, 即使有天气预报也不行, 因此它也不 100% 可靠。

但是代码可以啊。我们可以先遍历一遍 rain 数组就知道第几天哪个湖泊下雨了。有了这个信息, 我们就可以事后诸葛亮了。

“今天天气很好, 我开了天眼, 明天湖泊 2 会被洪水淹没, 我们今天就先抽干它, 否则就洪水泛滥了。”。



和上面的题目一样, 我们也可以不先遍历 rain 数组, 再模拟每天的变化, 而是直接模拟, 即使当前是晴天我们也不抽干任何湖泊。接着在模拟的过程记录晴天的情况, 等到洪水发生的时候, 我们再考虑前面哪一个晴天应该抽干哪个湖泊。因此这个事后诸葛亮体现在我们是等到洪水泛滥了才去想应该在之前的某天采取什么手段。

算法：

- 遍历 rain，模拟每天的变化
- 如果 rain 当前是 0 表示当前是晴天，我们不抽干任何湖泊。但是我们将当天记录到 sunny 数组。
- 如果 rain 大于 0，说明有一个湖泊下雨了，我们去看下下雨的这个湖泊是否发生了洪水泛滥。其实就是看下下雨前是否已经有水了。这提示我们用一个数据结构 lakes 记录每个湖泊的情况，我们可以用 0 表示没有水，1 表示有水。这样当湖泊 i 下雨的时候且 lakes[i] = 1 就会发生洪水泛滥。
- 如果当前湖泊发生了洪水泛滥，那么就去 sunny 数组找一个晴天去抽干它，这样它就不会洪水泛滥，接下来只需要保持 lakes[i] = 1 即可。

这道题没有使用到堆，我是故意的。之所以这么做，是让大家明白事后诸葛亮这个技巧并不是堆特有的，实际上这就是一种普通的算法思想，就好像从后往前遍历一样。只不过，很多时候，我们事后诸葛亮的场景，需要动态取最大最小值，这个时候就应该考虑使用堆了，这其实又回到文章开头的一个中心了，所以大家一定要灵活使用这些技巧，不可生搬硬套。

下一道题是一个不折不扣的事后诸葛亮 + 堆优化的题目。

代码

```
class Solution:
    def avoidFlood(self, rains: List[int]) -> List[int]:
        ans = [1] * len(rains)
        lakes = collections.defaultdict(int)
        sunny = []

        for i, rain in enumerate(rains):
            if rain > 0:
                ans[i] = -1
                if lakes[rain - 1] == 1:
                    if 0 == len(sunny):
                        return []
                    ans[sunny.pop()] = rain
                    lakes[rain - 1] = 1
                else:
                    sunny.append(i)
        return ans
```

(代码 1.3.11)

2021-04-06 fixed: 上面的代码有问题。错误的原因在于上述算法如果当前湖泊发生了洪水泛滥，那么就去 sunny 数组找一个晴天去抽干它，这样它就不会洪水泛滥部分的实现不对。sunny 数组找一个晴天去抽干它的根

本前提是 **出现晴天的时候湖泊里面要有水才能抽**，如果晴天的时候，湖泊里面没有水也不行。这提示我们的 `lakes` 不存储 0 和 1，而是存储发生洪水是第几天。这样问题就变为在 `sunny` 中找一个日期大于 `lakes[rain-1]` 的项，并将其移除 `sunny` 数组。由于 `sunny` 数组是有序的，因此我们可以使用二分来进行查找。

由于我们需要删除 `sunny` 数组的项，因此时间复杂度不会因为使用了二分而降低。

正确的代码应该为：

```
class Solution:
    def avoidFlood(self, rains: List[int]) -> List[int]:
        ans = [1] * len(rains)
        lakes = {}
        sunny = []

        for i, rain in enumerate(rains):
            if rain > 0:
                ans[i] = -1
                if rain - 1 in lakes:
                    j = bisect.bisect_left(sunny, lakes[rain - 1])
                    if j == len(sunny):
                        return []
                    ans[sunny.pop(j)] = rain
                    lakes[rain - 1] = i
            else:
                sunny.append(i)
        return ans
```

1642. 可以到达的最远建筑

题目描述

给你一个整数数组 `heights`，表示建筑物的高度。另有一些砖块 `bricks` 和

你从建筑物 0 开始旅程，不断向后面的建筑物移动，期间可能会用到砖块或梯子。

当从建筑物 `i` 移动到建筑物 `i+1`（下标从 0 开始）时：

如果当前建筑物的高度 大于或等于 下一建筑物的高度，则不需要梯子或砖块。

如果当前建筑的高度 小于 下一个建筑的高度，您可以使用 一架梯子 或 $(h[i] - h[i+1])$

如果以最佳方式使用给定的梯子和砖块，返回你可以到达的最远建筑物的下标（即 `heights` 中的最大下标）。

989. 数组形式的整数加法



示例 1:

输入: heights = [4,2,7,6,9,14,12], bricks = 5, ladders = 1
输出: 4

解释: 从建筑物 0 出发, 你可以按此方案完成旅程:

- 不使用砖块或梯子到达建筑物 1 , 因为 $4 \geq 2$
 - 使用 5 个砖块到达建筑物 2 。你必须使用砖块或梯子, 因为 $2 < 7$
 - 不使用砖块或梯子到达建筑物 3 , 因为 $7 \geq 6$
 - 使用唯一的梯子到达建筑物 4 。你必须使用砖块或梯子, 因为 $6 < 9$
- 无法越过建筑物 4 , 因为没有更多砖块或梯子。

示例 2:

输入: heights = [4,12,2,7,3,18,20,3,19], bricks = 10, ladders = 0
输出: 7

示例 3:

输入: heights = [14,3,19,3], bricks = 17, ladders = 0
输出: 3

提示:

```
1 <= heights.length <= 105
1 <= heights[i] <= 106
0 <= bricks <= 109
0 <= ladders <= heights.length
```

思路

我们可以将梯子看出是无限的砖块, 只不过只能使用一次, 我们当然希望能将好梯用在刀刃上。和上面一样, 如果是现实生活, 我们是无法知道啥时候用梯子好, 啥时候用砖头好的。

没关系, 我们继续使用事后诸葛亮法, 一次遍历就可完成。和前面的思路类似, 那就是我无脑用梯子, 等梯子不够用了, 我们就要开始事后诸葛亮了, 要是前面用砖头就好了。那什么时候用砖头就好了呢? 很明显就是当初用梯子的时候高度差, 比现在的高度差小。

直白点就是当初我用梯子爬了个 5 米的墙, 现在这里有个十米的墙, 我没梯子了, 只能用 10 个砖头了。要是之前用 5 个砖头, 现在不就可以用一个梯子, 从而省下 5 个砖头了吗?

这提示我们将用前面用梯子跨越的建筑物高度差存起来, 等到后面梯子用完了, 我们将前面被用的梯子“兑换”成砖头继续用。以上面的例子来说, 我们就可以先兑换 10 个砖头, 然后将 5 个砖头用掉, 也就是相当于增加

了 5 个砖头。

如果前面多次使用了梯子，我们优先“兑换”哪次呢？显然是优先兑换高度差大的，这样兑换的砖头才最多。这提示每次都从之前存储的高度差中选最大的，并在“兑换”之后将其移除。这种动态求极值的场景用什么数据结构合适？当然是堆啦。

代码

```
class Solution:
    def furthestBuilding(self, heights: List[int], bricks: int, ladders: int) -> int:
        h = []
        for i in range(1, len(heights)):
            diff = heights[i] - heights[i - 1]
            if diff <= 0:
                continue
            if bricks < diff and ladders > 0:
                ladders -= 1
                if h and -h[0] > diff:
                    bricks -= heapq.heappop(h)
                else:
                    continue
            bricks -= diff
            if bricks < 0:
                return i - 1
            heapq.heappush(h, -diff)
        return len(heights) - 1
```

(代码 1.3.12)

四大应用

接下来是本文的最后一个部分《四大应用》，目的是通过这几个例子来帮助大家巩固前面的知识。

1. topK

求解 topK 是堆的一个很重要的功能。这个其实已经在前面的固定堆部分给大家介绍过了。

这里直接引用前面的话：

“其实求第 k 小的数最简单的思路是建立小顶堆，将所有的数先全部入堆，然后逐个出堆，一共出堆 k 次。最后一次出堆的就是第 k 小的数。然而，我们也可不先全部入堆，而是建立大顶堆（注意不是上面的小顶堆），并维持堆的大小为 k 个。如果新的数入堆之后堆的大小大于 k，则

需要将堆顶的数和新的数进行比较，并将较大的移除。这样可以保证堆中的数是全体数字中最小的 k 个，而这最小的 k 个中最大的（即堆顶）不就是第 k 小的么？这也就是选择建立大顶堆，而不是小顶堆的原因。”

其实除了第 k 小的数，我们也可以将中间的数全部收集起来，这就可以求出最小的 k 个数。和上面第 k 小的数唯一不同的点在于需要收集 popp 出来的所有的数。

需要注意的是，有时候权重并不是原本数组值本身的大小，也可以是距离，出现频率等。

相关题目：

- [面试题 17.14. 最小 K 个数](#)
- [347. 前 K 个高频元素](#)
- [973. 最接近原点的 K 个点](#)

力扣中有关第 k 的题目很多都是堆。除了堆之外，第 k 的题目其实还会有一些找规律的题目，对于这种题目则可以通过分治+递归的方式来解决，具体就不再这里展开了，感兴趣的可以和我留言讨论。

2. 带权最短距离

关于这点，其实我在前面部分也提到过了，只不过当时只是一带而过。原话是“不过 BFS 真的就没人用优先队列实现么？当然不是！比如带权图的最短路径问题，如果用队列做 BFS 那就需要优先队列才可以，因为路径之间是有权重的差异的，这不就是优先队列的设计初衷么。使用优先队列的 BFS 实现典型的就是 dijkstra 算法。”

DIJKSTRA 算法主要解决的是图中任意两点的最短距离。

算法的基本思想是贪心，每次都遍历所有邻居，并从中找到距离最小的，本质上是一种广度优先遍历。这里我们借助堆这种数据结构，使得可以在 $\log N$ 的时间内找到 cost 最小的点，其中 N 为堆的大小。

代码模板：

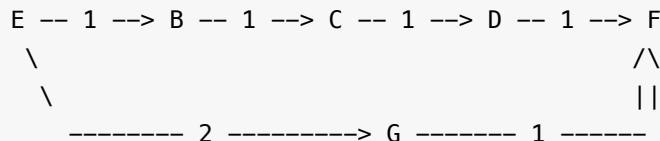
989. 数组形式的整数加法

```
def dijkstra(graph, start, end):
    # 堆里的数据都是 (cost, i) 的二元组, 其含义是“从 start 走到 i
    heap = [(0, start)]
    visited = set()
    while heap:
        (cost, u) = heapq.heappop(heap)
        if u in visited:
            continue
        visited.add(u)
        if u == end:
            return cost
        for v, c in graph[u]:
            if v in visited:
                continue
            next = cost + c
            heapq.heappush(heap, (next, v))
    return -1
```

(代码 1.4.1)

可以看出代码模板和 BFS 基本是类似的。如果你自己将堆的 key 设定为 steps 也可模拟实现 BFS, 这个在前面已经讲过了, 这里不再赘述。

比如一个图是这样的:



我们使用邻接矩阵来构造:

```
G = {
    "B": [["C", 1]],
    "C": [["D", 1]],
    "D": [["F", 1]],
    "E": [[["B", 1], ["G", 2]]],
    "F": [],
    "G": [[["F", 1]]],
}

shortDistance = dijkstra(G, "E", "C")
print(shortDistance) # E -- 3 --> F -- 3 --> C == 6
```

会了这个算法模板, 你就可以去 AC 743. 网络延迟时间 了。

完整代码：

```

class Solution:
    def dijkstra(self, graph, start, end):
        heap = [(0, start)]
        visited = set()
        while heap:
            (cost, u) = heapq.heappop(heap)
            if u in visited:
                continue
            visited.add(u)
            if u == end:
                return cost
            for v, c in graph[u]:
                if v in visited:
                    continue
                next = cost + c
                heapq.heappush(heap, (next, v))
        return -1
    def networkDelayTime(self, times: List[List[int]], N: int):
        graph = collections.defaultdict(list)
        for fr, to, w in times:
            graph[fr - 1].append((to - 1, w))
        ans = -1
        for to in range(N):
            # 调用封装好的 dijkstra 方法
            dist = self.dijkstra(graph, K - 1, to)
            if dist == -1: return -1
            ans = max(ans, dist)
        return ans

```

(代码 1.4.2)

你学会了么？

上面的算法并不是最优解，我只是为了体现将 **dijkstra** 封装为 **api** 调用的思想。一个更好的做法是一次遍历记录所有的距离信息，而不是每次都重复计算。时间复杂度会大大降低。这在计算一个点到图中所有点的距离时有很大的意义。为了实现这个目的，我们的算法会有什么样的调整？

提示：你可以使用一个 **dist** 哈希表记录开始点到每个点的最短距离来完成。想出来的话，可以用力扣 882 题去验证一下哦~

其实只需要做一个小的调整就可以了，由于调整很小，直接看代码会比较好。

代码：

```

class Solution:
    def dijkstra(self, graph, start, end):
        heap = [(0, start)] # cost from start node, end node
        dist = {}
        while heap:
            (cost, u) = heapq.heappop(heap)
            if u in dist:
                continue
            dist[u] = cost
            for v, c in graph[u]:
                if v in dist:
                    continue
                next = cost + c
                heapq.heappush(heap, (next, v))
        return dist
    def networkDelayTime(self, times: List[List[int]], N: int):
        graph = collections.defaultdict(list)
        for fr, to, w in times:
            graph[fr - 1].append((to - 1, w))
        ans = -1
        dist = self.dijkstra(graph, K - 1, to)
        return -1 if len(dist) != N else max(dist.values())

```

(代码 1.4.3)

可以看出我们只是将 `visitd` 替换成了 `dist`, 其他不变。另外 `dist` 其实只是带了 key 的 `visited`, 它这里也起到了 `visitd` 的作用。

如果你需要计算一个节点到其他所有节点的最短路径, 可以使用一个 `dist` (一个 hashmap) 来记录出发点到所有点的最短路径信息, 而不是使用 `visited` (一个 hashset) 。

类似的题目也不少, 我再举一个给大家 [787. K 站中转内最便宜的航班](#)。

题目描述:

989. 数组形式的整数加法

有 n 个城市通过 m 个航班连接。每个航班都从城市 u 开始，以价格 w 抵达
现在给定所有的城市和航班，以及出发城市 src 和目的地 dst ，你的任务是找

示例 1:

输入:

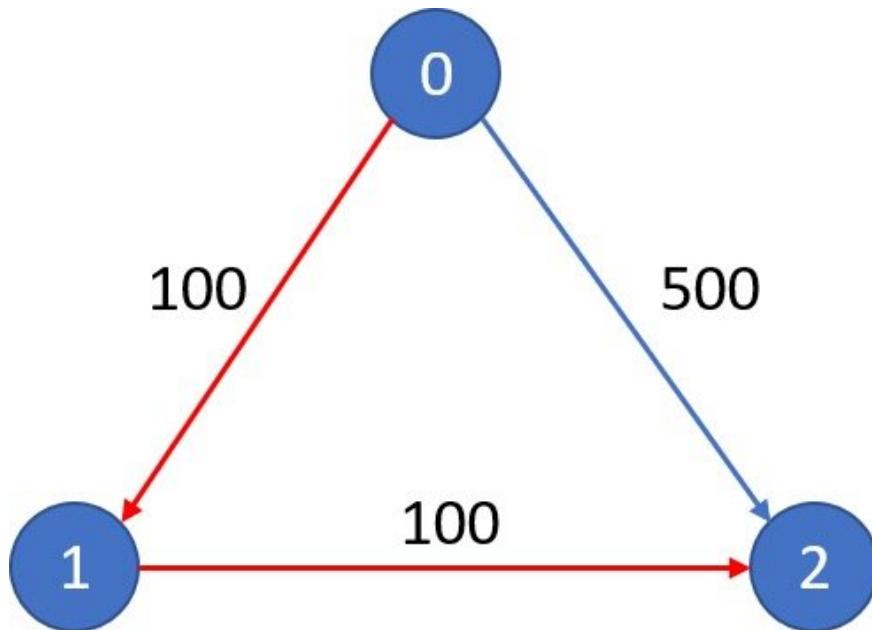
$n = 3$, edges = $[[0,1,100], [1,2,100], [0,2,500]]$

$src = 0$, $dst = 2$, $k = 1$

输出: 200

解释:

城市航班图如下



从城市 0 到城市 2 在 1 站中转以内的最便宜价格是 200，如图中红色所示。
示例 2:

输入:

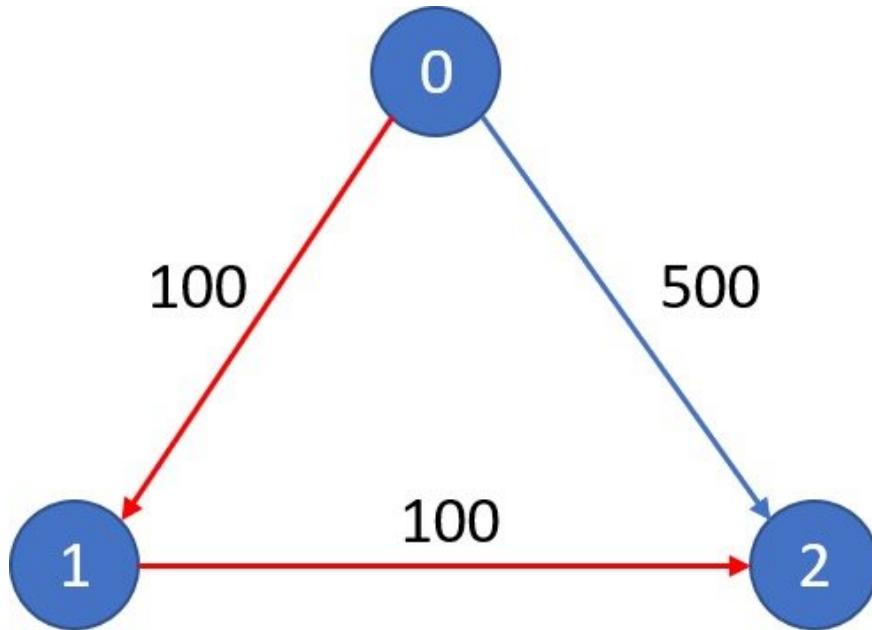
$n = 3$, edges = $[[0,1,100], [1,2,100], [0,2,500]]$

$src = 0$, $dst = 2$, $k = 0$

输出: 500

解释:

城市航班图如下



从城市 0 到城市 2 在 0 站中转以内的最便宜价格是 500，如图中蓝色所示。

提示：

n 范围是 [1, 100]，城市标签从 0 到 n - 1

航班数量范围是 [0, n * (n - 1) / 2]

每个航班的格式 (src, dst, price)

每个航班的价格范围是 [1, 10000]

k 范围是 [0, n - 1]

航班没有重复，且不存在自环

这道题和上面的没有本质不同，我仍然将其封装成 API 来使用，具体看代码就行。

这道题唯一特别的点在于如果中转次数大于 k，也认为无法到达。这个其实很容易，我们只需要在堆中用元组来多携带一个 **steps** 即可，这个 **steps** 就是不带权 BFS 中的距离。如果 pop 出来 **steps** 大于 K，则认为非法，我们跳过继续处理即可。

```

class Solution:
    # 改造一下，增加参数 K，堆多携带一个 steps 即可
    def dijkstra(self, graph, start, end, K):
        heap = [(0, start, 0)]
        visited = set()
        while heap:
            (cost, u, steps) = heapq.heappop(heap)
            if u in visited:
                continue
            visited.add((u, steps))
            if steps > K: continue
            if u == end:
                return cost
            for v, c in graph[u]:
                if (v, steps) in visited:
                    continue
                next = cost + c
                heapq.heappush(heap, (next, v, steps + 1))
        return -1
    def findCheapestPrice(self, n: int, flights: List[List[int]]):
        graph = collections.defaultdict(list)
        for fr, to, price in flights:
            graph[fr].append((to, price))
        # 调用封装好的 dijkstra 方法
        return self.dijkstra(graph, src, dst, K + 1)

```

(代码 1.4.4)

3. 因子分解

和上面两个应用一下，这个我在前面《313. 超级丑数》部分也提到了。

回顾一下丑数的定义：丑数就是质因数只包含 **2, 3, 5** 的正整数。因此丑数本质就是一个数经过**因子分解**之后只剩下 2, 3, 5 的整数，而不携带别的因子了。

关于丑数的题目有很多，大多数也可以从堆的角度考虑来解。只不过有时候因子个数有限，不使用堆也容易解决。比如：[264. 丑数 II](#) 就可以使用三个指针来记录即可，这个技巧在前面也讲过了，不再赘述。

一些题目并不是丑数，但是却明确提到了类似**因子**的信息，并让你求第 k 大的 xx，这个时候优先考虑使用堆来解决。如果题目中夹杂一些其他信息，**比如有序**，则也可考虑二分法。具体使用哪种方法，要具体问题具体分析，不过在此之前大家要对这两种方法都足够熟悉才行。

4. 堆排序

前面的三种应用或多或少在前面都提到过。而堆排序却未曾在前面提到。

直接考察堆排序的题目几乎没有。但是面试却有可能会考察，另外学习堆排序对你理解分治等重要算法思维都有重要意义。个人感觉，堆排序，构造二叉树，构造线段树等算法都有很大的相似性，掌握一种，其他都可以触类旁通。

实际上，经过前面的堆的学习，我们可以封装一个堆排序，方法非常简单。

这里我放一个使用堆的 api 实现堆排序的简单的示例代码：

```

h = [9, 5, 2, 7]
heapq.heapify(h)
ans = []

while h:
    ans.append(heapq.heappop(h))
print(ans) # 2,5,7,9

```

明白了示例，那封装成通用堆排序就不难了。

```

def heap_sort(h):
    heapq.heapify(h)
    ans = []
    while h:
        ans.append(heapq.heappop(h))
    return ans

```

这个方法足够简单，如果你明白了前面堆的原理，让你手撸一个堆排序也不难。可是这种方法有个弊端，它不是原位算法，也就是说你必须使用额外的空间承接结果，空间复杂度为 $O(N)$ 。但是其实调用完堆排序的方法后，原有的数组内存可以被释放了，因此理论上来说空间也没浪费，只不过我们计算空间复杂度的时候取的是使用内存最多的时刻，因此使用原地算法毫无疑问更优秀。如果你实在觉得不爽这个实现，也可以采用原地的修改的方式。这倒也不难，只不过稍微改造一下前面的堆的实现即可，由于篇幅的限制，这里不多讲了。

总结

堆和队列有千丝万缕的联系。很多题目我都是先思考使用堆来完成。然后发现每次入堆都是 + 1，而不会跳着更新，比如下一个 + 2, +3 等等，因此使用队列来完成性能更好。比如 [649. Dota2 参议院](#) 和 [1654. 到家的最少跳跃次数](#) 等。

堆的中心就一个，那就是动态求极值。

而求极值无非就是最大值或者最小值，这不难看出。如果求最大值，我们可以使用大顶堆，如果求最小值，可以用最小堆。而实际上，如果没有动态两个字，很多情况下没有必要使用堆。比如可以直接一次遍历找出最大的即可。而动态这个点不容易看出来，这正是题目的难点。这需要你先对问题进行分析，分析出这道题**其实**就是**动态求极值**，那么使用堆来优化就应该被想到。

堆的实现有很多。比如基于链表的跳表，基于数组的二叉堆和基于红黑树的实现等。这里我们介绍了**两种主要实现**并详细地讲述了二叉堆的实现，不仅是其实现简单，而且其在很多情况下表现都不错，推荐大家重点掌握二叉堆实现。

对于二叉堆的实现，**核心点**就一点，那就是始终维护堆的性质不变，具体是什么性质呢？那就是**父节点的权值不大于儿子的权值（小顶堆）**。为了达到这个目的，我们需要在入堆和出堆的时候，使用上浮和下沉操作，并恰当地完成元素交换。具体来说就是上浮过程和比它大的父节点进行交换，下沉过程和两个子节点中较小的进行交换，当然前提是它有子节点且子节点比它小。

关于堆化我们并没有做详细分析。不过如果你理解了本文的入堆操作，这其实很容易。因此堆化本身就是一个不断入堆的过程，只不过**将时间上的离散的操作变成了一次性操作而已**。

另外我给大家介绍了三个堆的做题技巧，分别是：

- 固定堆，不仅可以解决第 k 问题，还可有效利用已经计算的结果，避免重复计算。
- 多路归并，本质就是一个暴力解法，和暴力递归没有本质区别。如果你将其转化为递归，也是一种不能记忆化的递归。因此更像是**回溯算法**。
- 事后小诸葛。有些信息，我们在当前没有办法获取，就可用一种数据结构存起来，方便之后“东窗事发”的时候查。这种数据解决可以是很多，常见的有哈希表和堆。你也可以将这个技巧看成是**事后后悔**，有的人比较能接受这种叫法，不过不管叫法如何，指的都是这个含义。

最后给大家介绍了四种应用，这四种应用除了堆排序，其他在前面或多或少都讲过，它们分别是：

- topK
- 带权最短路径
- 因子分解
- 堆排序

这四种应用实际上还是围绕了堆的一个中心**动态取极值**，这四种应用只不过是灵活使用了这个特点罢了。因此大家在做题的时候只要死记**动态求极值**即可。如果你能够分析出这道题和动态取极值有关，那么请务必考虑堆。接下来我们就要在脑子中过一下复杂度，对照一下题目数据范围就大概可以估算出是否可行啦。

参考

- 几乎刷完了力扣所有的堆题，我发现了这些东西。。。 （第一弹）
- 几乎刷完了力扣所有的堆题，我发现了这些东西。。。 （第二弹）

跳表

虽然在面试中出现的频率不大，但是在工业中，跳表会经常被用到。力扣中关于跳表的题目只有一个。但是跳表的设计思路值得我们去学习和思考。其中有很多算法和数据结构技巧值得我们学习。比如空间换时间的思想，比如效率的取舍问题等。

解决的问题

只有知道跳表试图解决的问题，后面学习才会有针对性。实际上，跳表解决的问题非常简单，一句话就可以说清楚，那就是为了减少链表长度增加，查找链表节点时带来的额外比较次数。

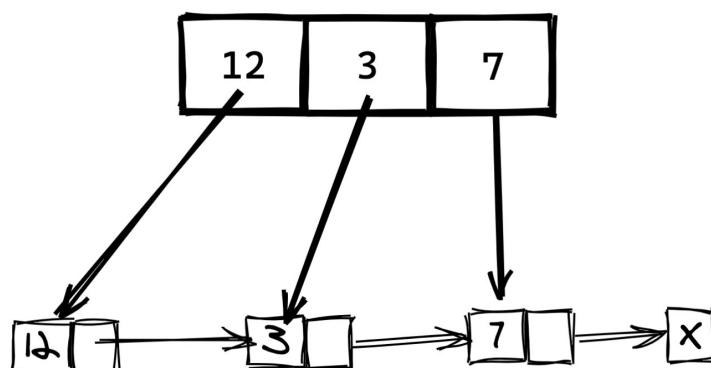
不借助额外空间的情况下，在链表中查找一个值，需要按照顺序一个个查找，时间复杂度为 $O(N)$ ，其中 N 为链表长度。



(单链表)

当链表长度很大的时候，这种时间是很难接受的。一种常见的的优化方式是建立哈希表，将所有节点都放到哈希表中，以空间换时间的方式减少时间复杂度，这种做法时间复杂度为 $O(1)$ ，但是空间复杂度为 $O(N)$ 。

hashtable



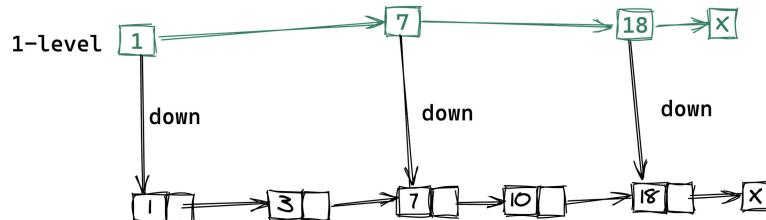
(单链表 + 哈希表)

为了防止链表中出现重复节点带来的问题，我们需要序列化节点，再建立哈希表，这种空间占用会更高，虽然只是系数级别的增加，但是这种开销也是不小的。

为了解决上面的问题，跳表应运而生。

如下图所示，我们从链表中每两个元素抽出来，加一级索引，一级索引指向了原始链表，即：通过一级索引 7 的 down 指针可以找到原始链表的 7。那怎么查找 10 呢？

注意这个算法要求链表是有序的。

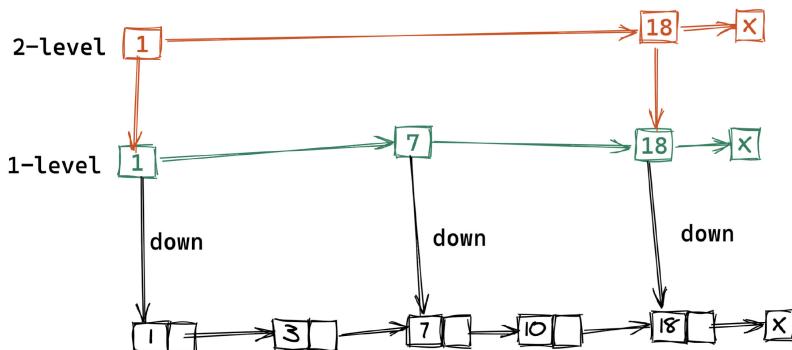


(建立一级索引)

我们可以：

- 通过现在一级跳表中搜索到 7，发现下一个 18 大于 10，也就是说我们要找的 10 在这两者之间。
- 通过 down 指针回到原始链表，通过原始链表的 next 指针我们找到了 10。

这个例子看不出性能提升。但是如果元素继续增大，继续增加索引的层数，建立二级，三级。。。索引，使得链表能够实现二分查找，从而获得更好的效率。但是相应地，我们需要付出额外空间的代价。



(增加索引层数)

理解了上面的点，你可以形象地将跳表想象为玩游戏的存档。

一个游戏有 10 关。如果我想要玩第 5 关的某一个地方，那么我可以直接从第五关开始，这样要比从第一关开始快。我们甚至可以在每一关同时设置很多的存档。这样我如果想玩第 5 关的某一个地方，也可以不用从第 5 关的开头开始，而是直接选择离你想玩的地方更近的存档，这就相当于跳表的二级索引。

跳表的时间复杂度和空间复杂度不是很好分析。由于时间复杂度 = 索引的高度 * 平均每层索引遍历元素的个数，而高度大概为 $\log n$ ，并且每层遍历的元素是常数，因此时间复杂度为 $\log n$ ，和二分查找的空间复杂度

是一样的。

空间复杂度就等同于索引节点的个数，以每两个节点建立一个索引为例，大概是 $n/2 + n/4 + n/8 + \dots + 8 + 4 + 2$ ，因此空间复杂度是 $\$O(n)\$$ 。当然你如果每三个建立一个索引节点的话，空间会更省，但是复杂度不变。

代码实现

关于代码，我先卖一个关子。这是因为跳表我们只准备了一道题，那就是实现跳表。因此我打算在每日一题的时候再给大家分析以及具体的代码。大家现在可以自己想想如何实现，然后对照官方题解看一下自己的实现和官方题解有什么不同，谁更好。

总结

- 跳表是可以实现二分查找的有序链表；
- 跳表由多层构成，最底层是包含所有的元素原始链表，往上是索引链表；
- 实际的设计中，需要做好取舍，设定合理数量的索引。
- 跳表查询、插入、删除的时间复杂度为 $\$O(\log N)\$$ ，空间复杂度为 $\$O(N)\$$ ；

989. 数组形式的整数加法

- 989. 数组形式的整数加法
- 821. 字符的最短距离
- 1381. 设计一个支持增量操作的栈
- 394. 字符串解码
- 232. 用栈实现队列
- 768. 最多能完成排序的块 II
- 61. 旋转链表
- 24. 两两交换链表中的节点
- 109. 有序链表转换二叉搜索树
- 160. 相交链表
- 142. 环形链表 II
- 146. LRU 缓存机制
- 104. 二叉树的最大深度
- 100. 相同的树
- 129. 求根到叶子节点数字之和
- 513. 找树左下角的值
- 297. 二叉树的序列化与反序列化
- 987. 二叉树的垂序遍历
- 两数之和
- 347. 前 K 个高频元素
- 447. 回旋镖的数量
- 3. 无重复字符的最长子串
- 30. 串联所有单词的子串
- Delete Sublist to Make Sum Divisible By K
- 876. 链表的中间结点
- 26. 删除排序数组中的重复项
- 35. 搜索插入位置
- 239. 滑动窗口最大值

题目地址(989. 数组形式的整数加法)

<https://leetcode-cn.com/problems/add-to-array-form-of-integer/>

入选理由

1. 简单题目，适合大家上手。
2. 之前力扣官方的每日一题，质量比较高

题目描述

989. 数组形式的整数加法

对于非负整数 X 而言， X 的数组形式是每位数字按从左到右的顺序形成的数组。

给定非负整数 X 的数组形式 A ，返回整数 $X+K$ 的数组形式。

示例 1:

输入: $A = [1,2,0,0]$, $K = 34$

输出: $[1,2,3,4]$

解释: $1200 + 34 = 1234$

示例 2:

输入: $A = [2,7,4]$, $K = 181$

输出: $[4,5,5]$

解释: $274 + 181 = 455$

示例 3:

输入: $A = [2,1,5]$, $K = 806$

输出: $[1,0,2,1]$

解释: $215 + 806 = 1021$

示例 4:

输入: $A = [9,9,9,9,9,9,9,9,9,9]$, $K = 1$

输出: $[1,0,0,0,0,0,0,0,0,0]$

解释: $999999999 + 1 = 10000000000$

提示:

$1 \leq A.length \leq 10000$

$0 \leq A[i] \leq 9$

$0 \leq K \leq 10000$

如果 $A.length > 1$, 那么 $A[0] \neq 0$

难度

- 简单

标签

- 数组

前置知识

- 数组的遍历

思路

如果你没做出来这道题，不妨先试试 66. 加一，那道题是这道题的简化版，即 $K = 1$ 的特殊形式。

这道题的思路是从低位到高位计算，注意进位和边界处理。细节都在代码里。

为了简化判断，我将 carry（进位）和 K 进行了统一处理，即 $\text{carry} = \text{carry} + K$

关键点

- 处理进位

代码

语言支持：Python3

```
```py {5-6,11-12} class Solution: def addToArrayForm(self, A: List[int], K: int) -> List[int]: carry = 0 for i in range(len(A) - 1, -1, -1): A[i], carry =
```

```
(carry + A[i] + K % 10) % 10, (carry + A[i] + K % 10) // 10 K // 10 B = []
```

```
 # 如果全部加完还有进位，需要特殊处理。比如 A = [2], K = 998
 carry = carry + K
 while carry:
 B = [(carry) % 10] + B
 carry //= 10
 return B + A
```

...

## 复杂度分析

令  $N$  为数组长度。

- 时间复杂度： $O(N + \max(0, K-N)^2)$
- 空间复杂度： $O(\max(1, K - N))$

## 题目地址(821. 字符的最短距离)

<https://leetcode-cn.com/problems/shortest-distance-to-a-character>

### 入选理由

1. 仍然是一道简单题，不过比昨天的题目难度增加一点
2. 虽然这是一个字符串的题目，但其实字符串和数组没有本质差别，这在讲义中也提到了。

### 题目描述

给定一个字符串 S 和一个字符 C。返回一个代表字符串 S 中每个字符到字符 C 的最短距离。

示例 1:

输入: S = "loveleetcode", C = 'e'  
输出: [3, 2, 1, 0, 1, 0, 0, 1, 2, 2, 1, 0]  
说明:

- 字符串 S 的长度范围为 [1, 10000]。
- C 是一个单字符，且保证是字符串 S 里的字符。
- S 和 C 中的所有字母均为小写字母。

### 标签

- 字符串

### 难度

- 简单

### 前置知识

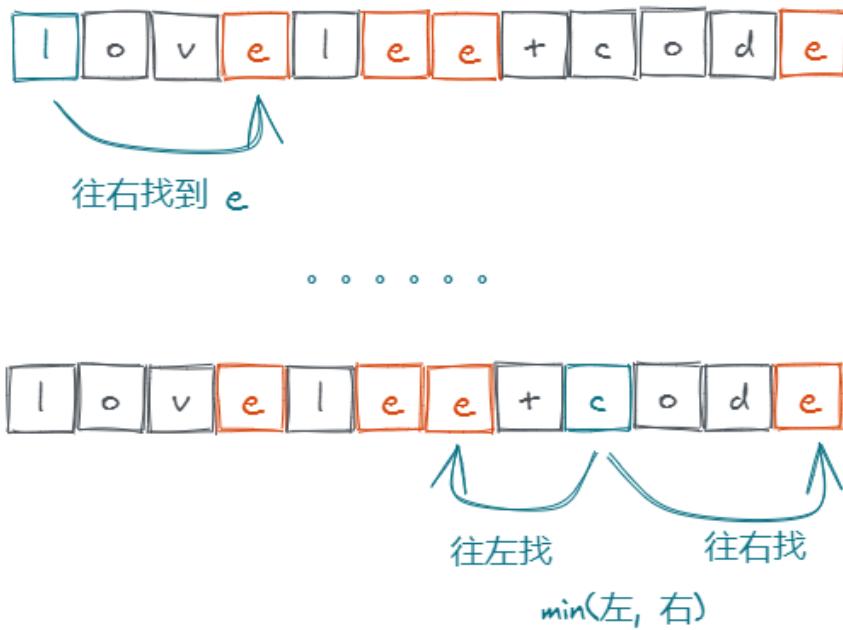
- 数组的遍历(正向遍历和反向遍历)

### 解法 1：中心扩展法

### 思路

这是最符合直觉的思路，对每个字符分别进行如下处理：

- 从当前下标出发，分别向左、右两个方向去寻找目标字符  $c$ 。
- 只在一个方向找到的话，直接计算字符距离。
- 两个方向都找到的话，取两个距离的最小值。



## 复杂度分析

- 时间复杂度： $O(N^2)$ ， $N$  为  $S$  的长度，两层循环。
- 空间复杂度： $O(1)$ 。

## 代码 (JS/C++)

JavaScript Code

## 989. 数组形式的整数加法

```
/**
 * @param {string} S
 * @param {character} C
 * @return {number[]}
 */
var shortestToChar = function (S, C) {
 // 结果数组 res
 var res = Array(S.length).fill(0);

 for (let i = 0; i < S.length; i++) {
 // 如果当前是目标字符，就什么都不用做
 if (S[i] === C) continue;

 // 定义两个指针 l, r 分别向左、右两个方向寻找目标字符 C, 取最短
 let l = i,
 r = i,
 shortest = Infinity;

 while (l >= 0) {
 if (S[l] === C) {
 shortest = Math.min(shortest, i - l);
 break;
 }
 l--;
 }

 while (r < S.length) {
 if (S[r] === C) {
 shortest = Math.min(shortest, r - i);
 break;
 }
 r++;
 }

 res[i] = shortest;
 }
 return res;
};
```

C++ Code

```

class Solution {
public:
 vector<int> shortestToChar(string S, char C) {
 vector<int> res(S.length());

 for (int i = 0; i < S.length(); i++) {
 if (S[i] == C) continue;

 int left = i;
 int right = i;
 int dist = 0;

 while (left >= 0 || right <= S.length() - 1) {
 if (S[left] == C) {
 dist = i - left;
 break;
 }
 if (S[right] == C) {
 dist = right - i;
 break;
 }
 if (left > 0) left--;
 if (right < S.length() - 1) right++;
 }

 res[i] = dist;
 }

 return res;
 }
};

```

## 解法 2：空间换时间

### 思路

空间换时间是编程中很常见的一种 trade-off (反过来，时间换空间也是)。

因为目标字符 `C` 在 `S` 中的位置是不变的，所以我们可以提前将 `C` 的所有下标记录在一个数组 `cIndices` 中。

然后遍历字符串 `S` 中的每个字符，到 `cIndices` 中找到距离当前位置最近的下标，计算距离。

### 复杂度分析

- 时间复杂度:  $O(N \cdot K)$ ,  $N$  是  $S$  的长度,  $K$  是字符  $C$  在字符串中出现的次数,  $K \leq N$ 。
- 空间复杂度:  $O(K)$ ,  $K$  为字符  $C$  出现的次数, 这是记录字符  $C$  出现下标的辅助数组消耗的空间。

## 代码 (JS/C++)

JavaScript Code

```
/*
 * @param {string} S
 * @param {character} C
 * @return {number[]}
 */
var shortestToChar = function (S, C) {
 // 记录 C 字符在 S 字符串中出现的所有下标
 var cIndices = [];
 for (let i = 0; i < S.length; i++) {
 if (S[i] === C) cIndices.push(i);
 }

 // 结果数组 res
 var res = Array(S.length).fill(Infinity);

 for (let i = 0; i < S.length; i++) {
 // 目标字符, 距离是 0
 if (S[i] === C) {
 res[i] = 0;
 continue;
 }

 // 非目标字符, 到下标数组中找最近的下标
 for (const cIndex of cIndices) {
 const dist = Math.abs(cIndex - i);

 // 小小剪枝一下
 // 注: 因为 cIndices 中的下标是递增的, 后面的 dist 也会越来越大
 if (dist >= res[i]) break;

 res[i] = dist;
 }
 }
 return res;
};
```

C++ Code

```

class Solution {
public:
 vector<int> shortestToChar(string S, char C) {
 int n = S.length();
 vector<int> c_indices;
 // Initialize a vector of size n with default value
 vector<int> res(n, n);

 for (int i = 0; i < n; i++) {
 if (S[i] == C) c_indices.push_back(i);
 }

 for (int i = 0; i < n; i++) {
 if (S[i] == C) {
 res[i] = 0;
 continue;
 }

 for (int j = 0; j < c_indices.size(); j++) {
 int dist = abs(c_indices[j] - i);
 if (dist > res[i]) break;
 res[i] = dist;
 }
 }

 return res;
 }
};

```

## 解法 3：贪心

### 思路

其实对于每个字符来说，它只关心离它最近的那个 `C` 字符，其他的它都不管。所以这里还可以用贪心的思路：

1. 先 从左往右 遍历字符串 `S`，用一个数组 `left` 记录每个字符 左侧  
出现的最后一个 `C` 字符的下标；
2. 再 从右往左 遍历字符串 `S`，用一个数组 `right` 记录每个字符 右  
侧 出现的最后一个 `C` 字符的下标；
3. 然后同时遍历这两个数组，计算距离最小值。

### 优化 1

## 989. 数组形式的整数加法

再多想一步，其实第二个数组并不需要。因为对于左右两侧的 `C` 字符，我们也只关心其中距离更近的那个，所以第二次遍历的时候可以看情况覆盖掉第一个数组的值：

1. 字符左侧没有出现过 `C` 字符
2.  $i - \text{left} > \text{right} - i$  (`i` 为当前字符下标, `left` 为字符左侧最近的 `C` 下标, `right` 为字符右侧最近的 `C` 下标)

如果出现以上两种情况，就可以进行覆盖，最后再遍历一次数组计算距离。

### 优化 2

如果我们是直接记录 `C` 与当前字符的距离，而不是记录 `C` 的下标，还可以省掉最后一次遍历计算距离的过程。

## 复杂度分析

- 时间复杂度：\$O(N)\$，`N` 是 `S` 的长度。
- 空间复杂度：\$O(1)\$。

## 代码 (JS/C++/Python)

JavaScript Code

## 989. 数组形式的整数加法

```
/**
 * @param {string} S
 * @param {character} C
 * @return {number[]}
 */
var shortestToChar = function (S, C) {
 var res = Array(S.length);

 // 第一次遍历：从左往右
 // 找到出现在左侧的 C 字符的最后下标
 for (let i = 0; i < S.length; i++) {
 if (S[i] === C) res[i] = i;
 // 如果左侧没有出现 C 字符的话，用 Infinity 进行标记
 else res[i] = res[i - 1] === void 0 ? Infinity : res[i - 1];
 }

 // 第二次遍历：从右往左
 // 找到出现在右侧的 C 字符的最后下标
 // 如果左侧没有出现过 C 字符，或者右侧出现的 C 字符距离更近，就更新
 for (let i = S.length - 1; i >= 0; i--) {
 if (res[i] === Infinity || res[i + 1] - i < i - res[i])
 res[i] = res[i + 1];
 }

 // 计算距离
 for (let i = 0; i < res.length; i++) {
 res[i] = Math.abs(res[i] - i);
 }
 return res;
};
```

直接计算距离：

JavaScript Code

## 989. 数组形式的整数加法

```
/*
 * @param {string} S
 * @param {character} C
 * @return {number[]}
 */
var shortestToChar = function (S, C) {
 var res = Array(S.length);

 for (let i = 0; i < S.length; i++) {
 if (S[i] === C) res[i] = 0;
 // 记录距离: res[i - 1] + 1
 else res[i] = res[i - 1] === void 0 ? Infinity : res[i - 1] + 1;
 }

 for (let i = S.length - 1; i >= 0; i--) {
 // 更新距离: res[i + 1] + 1
 if (res[i] === Infinity || res[i + 1] + 1 < res[i]) res[i] = res[i + 1] + 1;
 }

 return res;
};
```

## C++ Code

```
class Solution {
public:
 vector<int> shortestToChar(string S, char C) {
 int n = S.length();
 vector<int> dist(n, n);

 for (int i = 0; i < n; i++) {
 if (S[i] == C) dist[i] = 0;
 else if (i > 0) dist[i] = dist[i - 1] + 1;
 }

 for (int i = n - 1; i >= 0; i--) {
 if (dist[i] == n
 || (i < n - 1 && dist[i + 1] + 1 < dist[i]))
 dist[i] = dist[i + 1] + 1;
 }

 return dist;
 }
};
```

## Python Code

```
class Solution(object):
 def shortestToChar(self, s, c):
 """
 :type s: str
 :type c: str
 :rtype: List[int]
 """

 n = len(s)
 res = [0 if s[i] == c else None for i in range(n)]

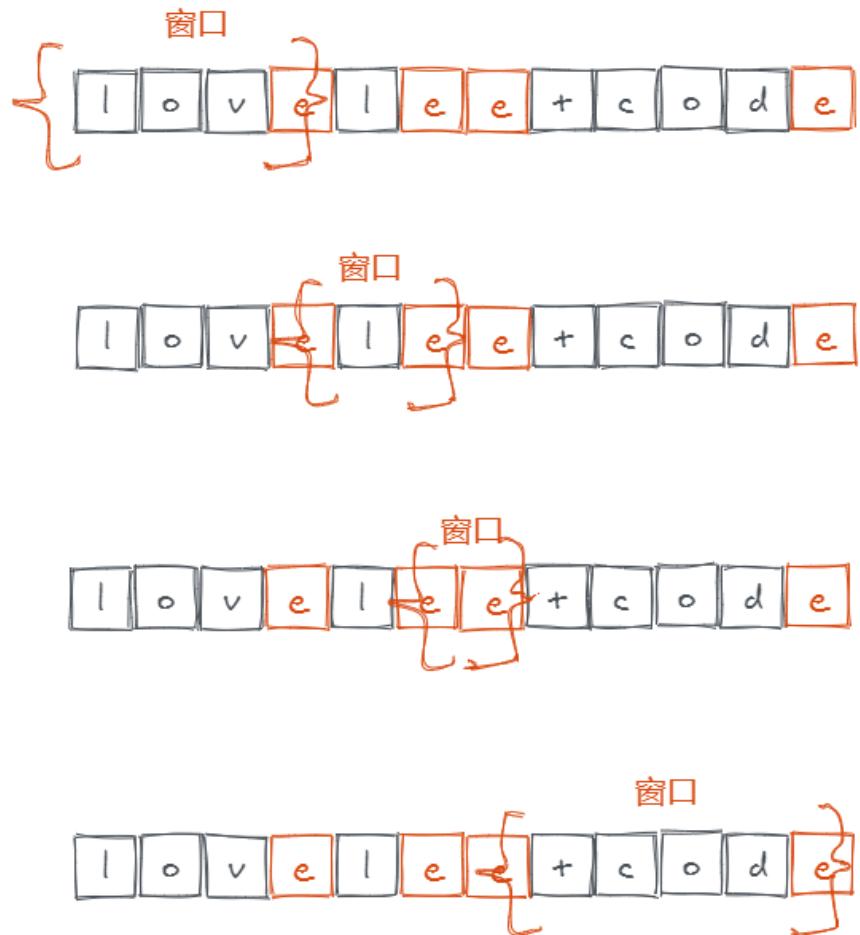
 for i in range(1, n):
 if res[i] != 0 and res[i - 1] is not None:
 res[i] = res[i - 1] + 1

 for i in range(n - 2, -1, -1):
 if res[i] is None or res[i + 1] + 1 < res[i]:
 res[i] = res[i + 1] + 1
 return res
```

## 解法 4：窗口

### 思路

把 `c` 看成分界线，将 `s` 划分成一个个窗口。然后对每个窗口进行遍历，分别计算每个字符到窗口边界的距离最小值。



## 复杂度分析

- 时间复杂度:  $O(N)$ ,  $N$  是  $S$  的长度。
- 空间复杂度:  $O(1)$ 。

## 代码 (JS/C++/Python)

JavaScript Code

## 989. 数组形式的整数加法

```
/*
 * @param {string} S
 * @param {character} C
 * @return {number[]}
 */
var shortestToChar = function (S, C) {
 // 窗口左边界, 如果没有就初始化为 Infinity, 初始化为 S.length 也
 let l = S[0] === C ? 0 : Infinity,
 // 窗口右边界
 r = S.indexOf(C, 1);

 const res = Array(S.length);

 for (let i = 0; i < S.length; i++) {
 // 计算字符到当前窗口左右边界的最小距离
 res[i] = Math.min(Math.abs(i - l), Math.abs(r - i));

 // 遍历完了当前窗口的字符后, 将整个窗口右移
 if (i === r) {
 l = r;
 r = S.indexOf(C, l + 1);
 }
 }

 return res;
};
```

C++ Code

## 989. 数组形式的整数加法

```
class Solution {
public:
 vector<int> shortestToChar(string S, char C) {
 int n = S.length();

 int l = S[0] == C ? 0 : n;
 int r = S.find(C, 1);

 vector<int> dist(n);

 for (int i = 0; i < n; i++) {
 dist[i] = min(abs(i - l), abs(r - i));
 if (i == r) {
 l = r;
 r = S.find(C, r + 1);
 }
 }

 return dist;
 }
};
```

Python Code

```
class Solution(object):
 def shortestToChar(self, s, c):
 """
 :type s: str
 :type c: str
 :rtype: List[int]
 """

 n = len(s)
 res = [0 for _ in range(n)]

 l = 0 if s[0] == c else n
 r = s.find(c, 1)

 for i in range(n):
 res[i] = min(abs(i - l), abs(r - i))
 if i == r:
 l = r
 r = s.find(c, l + 1)

 return res
```

## 题目地址(1381. 设计一个支持增量操作的栈)

<https://leetcode-cn.com/problems/design-a-stack-with-increment-operation/>

### 入选理由

1. 前两天是数组，今后三天都是栈。 栈的难度不低，值得大家注意。  
队列的应用我们放到后面 bfs 和 堆部分出题
2. 难度中等。可以和前两天的题目形成难度梯度。

### 题目描述

请你设计一个支持下述操作的栈。

实现自定义栈类 `CustomStack`：

`CustomStack(int maxSize)`: 用 `maxSize` 初始化对象, `maxSize` 是栈中  
`void push(int x)`: 如果栈还未增长到 `maxSize`, 就将 `x` 添加到栈顶。  
`int pop()`: 弹出栈顶元素, 并返回栈顶的值, 或栈为空时返回 `-1`。  
`void inc(int k, int val)`: 栈底的 `k` 个元素的值都增加 `val`。如果栈

示例：

输入：

```
["CustomStack","push","push","pop","push","push","push","inc","pop"]
[[3],[1],[2],[1],[2],[3],[4],[5,100],[2,100],[1],[1],[1]]
```

输出：

```
[null,null,null,2,null,null,null,null,103,202,201,-1]
```

解释：

```
CustomStack customStack = new CustomStack(3); // 栈是空的 []
customStack.push(1); // 栈变为 [1]
customStack.push(2); // 栈变为 [1, 2]
customStack.pop(); // 返回 2 --> 返回栈顶值 2, 栈变为 [1]
customStack.push(2); // 栈变为 [1, 2]
customStack.push(3); // 栈变为 [1, 2, 3]
customStack.push(4); // 栈仍然是 [1, 2, 3], 不能添加其他元素使栈
customStack.increment(5, 100); // 栈变为 [101, 102, 103]
customStack.increment(2, 100); // 栈变为 [201, 202, 103]
customStack.pop(); // 返回 103 --> 返回栈顶值 103, 栈变为 [201
customStack.pop(); // 返回 202 --> 返回栈顶值 202, 栈变为 [201
customStack.pop(); // 返回 201 --> 返回栈顶值 201, 栈变为 []
customStack.pop(); // 返回 -1 --> 栈为空, 返回 -1
```

提示：

```
1 <= maxSize <= 1000
1 <= x <= 1000
1 <= k <= 1000
0 <= val <= 100
```

每种方法 `increment`, `push` 以及 `pop` 分别最多调用 1000 次

## 难度

- 简单

## 标签

- 栈

## 前置知识

- 栈
- 前缀和

## increment 时间复杂度为 $O(k)$ 的方法

### 思路

首先我们来看一种非常符合直觉的方法，然而这种方法并不好，`increment` 操作需要的时间复杂度为  $O(k)$ 。

`push` 和 `pop` 就是普通的栈操作。唯一要注意的是边界条件，这个已经在题目中指明了，具体来说就是：

- `push` 的时候要判断是否满了
- `pop` 的时候要判断是否空了

而做到上面两点，只需要一个 `cnt` 变量记录栈的当前长度，一个 `size` 变量记录最大容量，并在 `pop` 和 `push` 的时候更新 `cnt` 即可。

### 代码

```

class CustomStack:

 def __init__(self, size: int):
 self.st = []
 self.cnt = 0
 self.size = size

 def push(self, x: int) -> None:
 if self.cnt < self.size:
 self.st.append(x)
 self.cnt += 1

 def pop(self) -> int:
 if self.cnt == 0: return -1
 self.cnt -= 1
 return self.st.pop()

 def increment(self, k: int, val: int) -> None:
 for i in range(0, min(self.cnt, k)):
 self.st[i] += val

```

### 复杂度分析

- 时间复杂度：push 和 pop 操作的时间复杂度为  $O(1)$ （讲义有提到），而 increment 操作的时间复杂度为  $O(\min(k, \text{cnt}))$
- 空间复杂度： $O(1)$

## 前缀和

前缀和在讲义里面提到过，大家也可看下我的文章 [一次搞定前缀和](#)

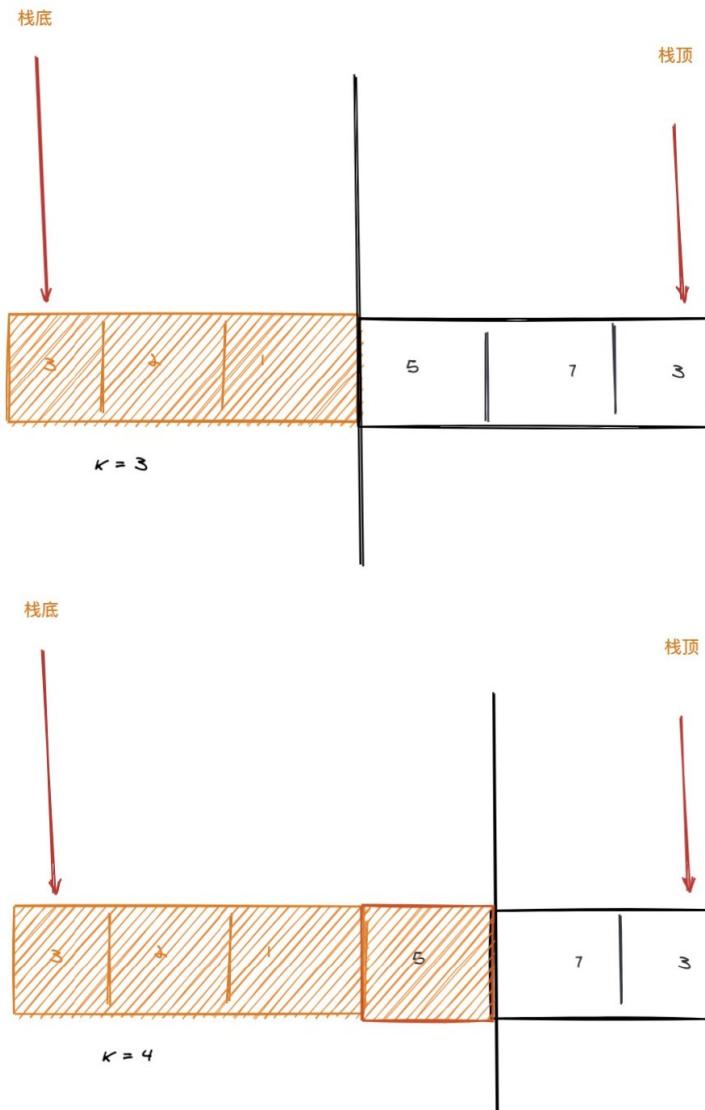
## 思路

和上面的思路类似，不过我们采用空间换时间的方式。采用一个额外的数组 `incrementals` 来记录每次 `incremental` 操作。

具体算法如下：

- 初始化一个大小为 `maxSize` 的数组 `incrementals`，并全部填充 0
- `push` 操作不变，和上面一样
- `increment` 的时候，我们将用到 `incremental` 信息。那么这个信息是什么，从哪来呢？我这里画了一个图

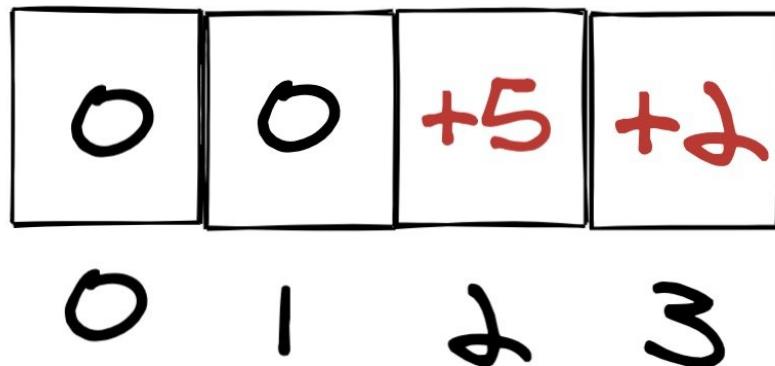
### 989. 数组形式的整数加法



如图黄色部分是我们需要执行增加操作，我这里画了一个挡板分割，实际上这个挡板不存在。那么如何记录黄色部分的信息呢？我举个例子来说

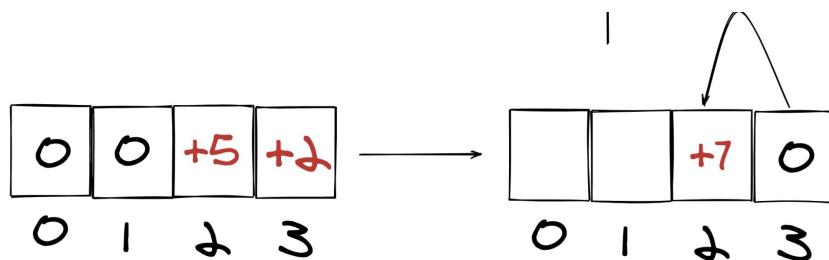
比如：

- 调用了 `increment(3, 2)`, 就把 `increment[3]` 增加 2。
- 继续调用 `increment(2, 5)`, 就把 `increment[2]` 增加 5。



而当我们 pop 的时候：

- 只需要将栈顶元素加上 `increment[cnt - 1]` 即可，其中 `cnt` 为栈当前的大小。
- 另外，我们需要将 `increment[cnt - 1]` 更新到 `increment[cnt - 2]`，并将 `increment[cnt - 1]` 重置为 0。



## 代码

```
```py {18, 27} class CustomStack:
```

989. 数组形式的整数加法

```
def __init__(self, size: int):
    self.st = []
    self.cnt = 0
    self.size = size
    self.incrementals = [0] * size

def push(self, x: int) -> None:
    if self.cnt < self.size:
        self.st.append(x)
        self.cnt += 1

def pop(self) -> int:
    if self.cnt == 0: return -1
    if self.cnt >= 2:
        self.incrementals[self.cnt - 2] += self.incrementals[-1]
    ans = self.st.pop() + self.incrementals[self.cnt - 1]
    self.incrementals[self.cnt - 1] = 0
    self.cnt -= 1
    return ans

def increment(self, k: int, val: int) -> None:
    if self.cnt:
        self.incrementals[min(self.cnt, k) - 1] += val
```

****复杂度分析****

- 时间复杂度：全部都是 $O(1)$
- 空间复杂度：我们维护了一个大小为 \maxSize 的数组，因此平均到每次的空

优化的前缀和

思路

上面的思路无论如何，我们都需要维护一个大小为 \maxSize 的数组 `increments`

每次栈 `push` 的时候，`increments` 也 `push` 一个 `0`。每次栈 `pop` 的时

> 这里的 `increments` 并不是一个栈，而是一个普通数组，因此可以随机访问。

代码

```
```py {11-12,20-21, 26}
class CustomStack:

 def __init__(self, size: int):
 self.st = []
 self.cnt = 0
 self.size = size
 self.increments = []

 def push(self, x: int) -> None:
 if self.cnt < self.size:
 self.st.append(x)
 self.increments.append(0)
 self.cnt += 1

 def pop(self) -> int:
 if self.cnt == 0: return -1
 self.cnt -= 1
 if self.cnt >= 1:
 self.increments[-2] += self.increments[-1]
 return self.st.pop() + self.increments.pop()

 def increment(self, k: int, val: int) -> None:
 if self.increments:
 self.increments[min(self.cnt, k) - 1] += val
```

**复杂度分析**

- 时间复杂度：全部都是  $O(1)$
- 空间复杂度：我们维护了一个大小为  $cnt$  的数组，因此平均到每次的空间复杂度为  $O(cnt / N)$ ，其中  $N$  为操作数， $cnt$  为操作过程中的栈的最大长度（小于等于  $maxSize$ ）。

可以看出优化的解法在  $maxSize$  非常大的时候是很有意义的。

## 相关题目

- [155. 最小栈](#)

更多题解可以访问我的 LeetCode 题解仓库：

<https://github.com/azl397985856/leetcode>。目前已经 37K 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



欢迎长按关注



## 题目地址(394. 字符串解码)

<https://leetcode-cn.com/problems/decode-string/>

### 入选理由

1. 前面说了栈的难度比较大，因此接下来几天都是栈，今天这个就是。
2. 今天的难度相比昨天难度增加。关键的是这道题的解法很有用，很多力扣的题都用这种思路，甚至是 hard 题目，基本思路也是一样的。实际上，这题就是一个括号匹配而已，匹配的括号对作为一层。大家可以尝试使用递归和迭代两种方式解决，来直观感受一下。

### 题目描述

给定一个经过编码的字符串，返回它解码后的字符串。

编码规则为：k[encoded\_string]，表示其中方括号内部的 encoded\_string

你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号是合法的。

此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 k，例如，如果输入为 "3[a]2[bc]"，则输出为 "aaabcbc"。

示例 1：

输入: s = "3[a]2[bc]"

输出: "aaabcbc"

示例 2：

输入: s = "3[a2[c]]"

输出: "accaccacc"

示例 3：

输入: s = "2[abc]3[cd]ef"

输出: "abcabcccdcdcdef"

示例 4：

输入: s = "abc3[cd]xyz"

输出: "abccdcdcdxyz"

### 难度

- 中等

## 标签

- 栈
- DFS

## 前置知识

- 栈
- 括号匹配

## 方法一：栈

### 思路

题目要求将一个经过编码的字符解码并返回解码后的字符串。题目给定的条件是只有四种可能出现的字符

1. 字母
2. 数字
3. [
4. ]

并且输入的方括号总是满足要求的（成对出现），数字只表示重复次数。

那么根据以上条件，可以看出其括号符合栈先进后出的特性以及递归的特质，稍后我们使用递归来解。

那么现在看一下迭代的解法。

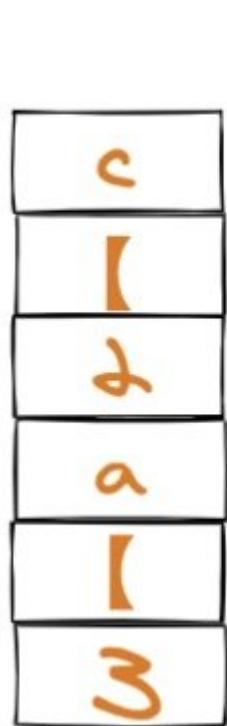
我们可以利用 stack 来实现这个操作，遍历这个字符串 s，判断每一个字符的类型：

- 如果是字母 --> 添加到 stack 当中
- 如果是数字 --> 先不着急添加到 stack 中 --> 因为有可能有多位
- 如果是 [ --> 说明重复字符串开始 --> 将数字入栈 --> 并且将数字清零
- 如果是 ] --> 说明重复字符串结束 --> 将重复字符串重复前一步储存的数字遍

拿题目给的例子 `s = "3[a2[c]]"` 来说：

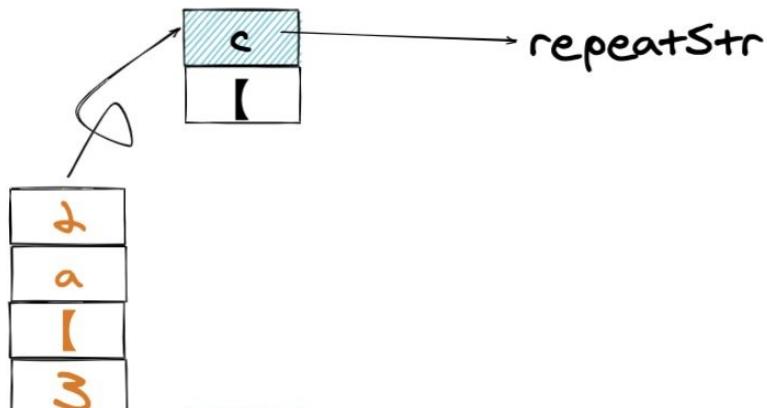
3[a2[c]]

在遇到 `】` 之前，我们不断执行压栈操作：

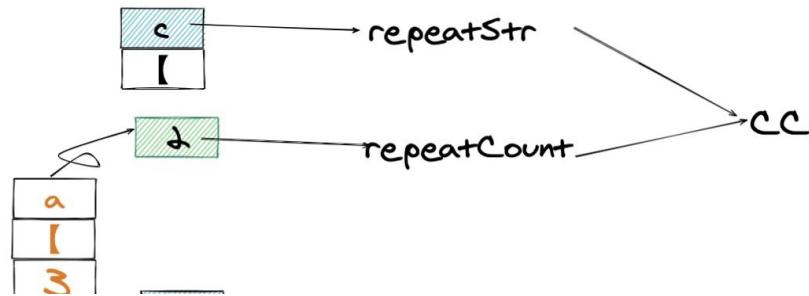


】

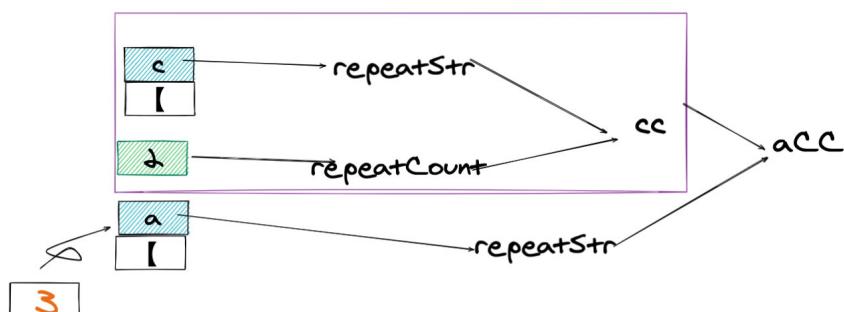
当遇到 `】` 的时候，说明我们应该出栈了，不断出栈知道对应的 `【`，这中间的就是 `repeatStr`。



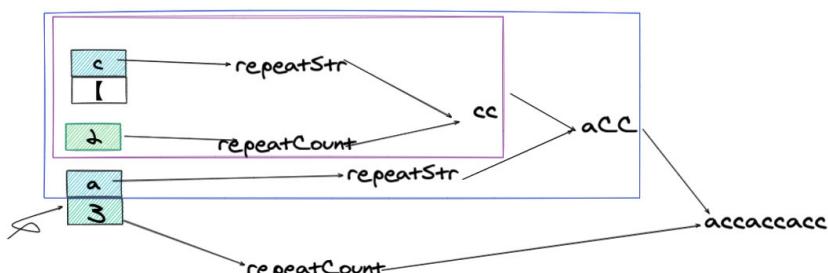
但是要重复几次呢？我们需要继续出栈，直到非数字为止，这个数字我们记录为 `repeatCount`。



而最终的字符串就是 `repeatCount` 个 `repeatStr` 拼接的形式。并将其看成一个字母压入栈中。



继续，后面的逻辑是一样的：



(最终图)

## 代码

代码支持：Python

Python：

```
```py {9, 13-14} class Solution: def decodeString(self, s: str) -> str: stack = [] for c in s: if c == ']': repeatStr = "" repeatCount = " " while stack and stack[-1] != '[': repeatStr = stack.pop() + repeatStr
```

989. 数组形式的整数加法

```
# pop 掉 "["
stack.pop()
while stack and stack[-1].isnumeric():
    repeatCount = stack.pop() + repeatCount
    stack.append(repeatStr * int(repeatCount))
else:
    stack.append(c)
return "" .join(stack)
```

****复杂度分析****

- 时间复杂度: $O(N)$, 其中 N 为解码后的 s 的长度。
- 空间复杂度: $O(N)$, 其中 N 为解码后的 s 的长度。

方法二：递归

思路

递归的解法也是类似。由于递归的解法并不比迭代书写简单，以及递归我们将在：

主逻辑仍然和迭代一样。只不过每次碰到左括号就进入递归，碰到右括号就跳出。

唯一需要注意的是，我这里使用了 `start` 指针跟踪当前遍历到的位置，因此我们

总结一下：

- 遇到数字，我们需要将其累加到 `count` 中。（因为数字可能有多位）
- 遇到普通字符，将其直接添加到栈
- 遇到 “[”，开启一轮新的递归。由于需要继续上次递归结束的地方继续处理，
- 遇到 “]”，结束递归，返回索引和栈上的字符。

这是一个典型的 DFS + 栈的题目，值得大家掌握。

代码

```
```py {11, 16}
class Solution:

 def decodeString(self, s: str) -> str:
 def dfs(start):
 repeat_str = repeat_count = ''
 while start < len(s):
 if s[start].isnumeric():
 repeat_count += s[start]
 elif s[start] == '[':
 # 更新指针
 start, t_str = dfs(start + 1)
 # repeat_count 仅作用于 t_str, 而不作用于当前的 repeat_str
 repeat_str = repeat_str + t_str * int(repeat_count)
 repeat_count = ''
 elif s[start] == ']':
 return start, repeat_str
 else:
 repeat_str += s[start]
 start += 1
 return start, repeat_str
```

```
return repeat_str
return dfs(0)
```

### 复杂度分析

- 时间复杂度:  $O(N)$ , 其中 N 为解码后的 s 的长度。
- 空间复杂度:  $O(N)$ , 其中 N 为解码后的 s 的长度。

更多题解可以访问我的 LeetCode 题解仓库:

<https://github.com/azl397985856/leetcode>。目前已经 37K 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



欢迎长按关注



## 题目地址(232. 用栈实现队列)

<https://leetcode-cn.com/problems/implement-queue-using-stacks/>

### 入选理由

1. 这题贼经典，考察次数很多。

### 题目描述

使用栈实现队列的下列操作：

`push(x)` -- 将一个元素放入队列的尾部。

`pop()` -- 从队列首部移除元素。

`peek()` -- 返回队列首部的元素。

`empty()` -- 返回队列是否为空。

示例：

```
MyQueue queue = new MyQueue();
```

```
queue.push(1);
```

```
queue.push(2);
```

```
queue.peek(); // 返回 1
```

```
queue.pop(); // 返回 1
```

```
queue.empty(); // 返回 false
```

说明：

你只能使用标准的栈操作 -- 也就是只有 `push to top`, `peek/pop from bottom`。你所使用的语言也许不支持栈。你可以使用 `list` 或者 `deque` (双端队列) 来实现。假设所有操作都是有效的、（例如，一个空的队列不会调用 `pop` 或者 `peek`）

### 难度

- 简单

### 标签

- 栈

### 前置知识

- 栈

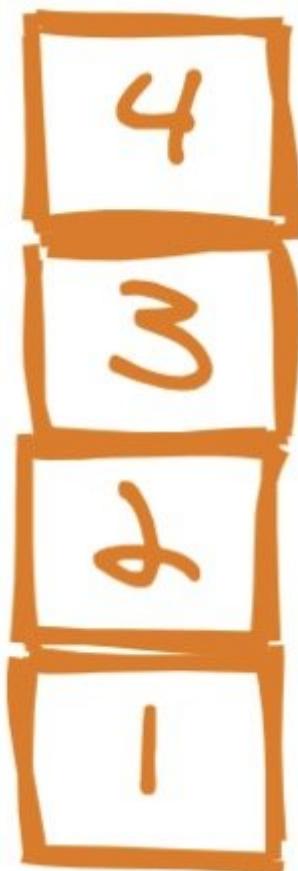
- 队列

## 思路

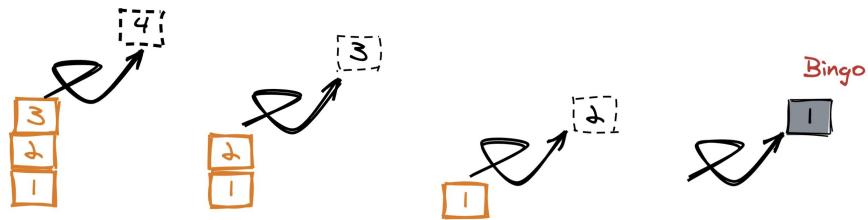
题目要求用栈的原生操作来实现队列，也就是说需要用到 pop 和 push 但是我们知道 pop 和 push 都是在栈顶的操作，而队列的 enqueue 和 dequeue 则是在队列的两端的操作，这么一看一个 stack 好像不太能完成。

我们来分析一下过程。

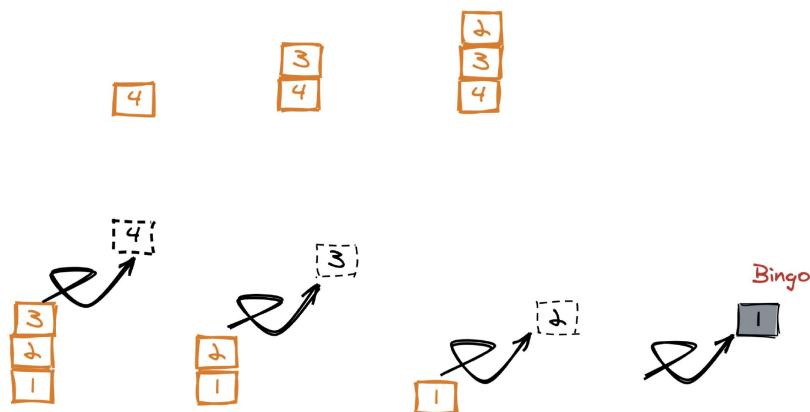
假如向栈中分别 push 四个数字 1, 2, 3, 4，那么此时栈的情况应该是：



如果此时按照题目要求 pop 或者 peek 的话，应该是返回 1 才对，而 1 在栈底我们无法直接操作。如果想要返回 1，我们首先要将 2, 3, 4 分别出栈才行。

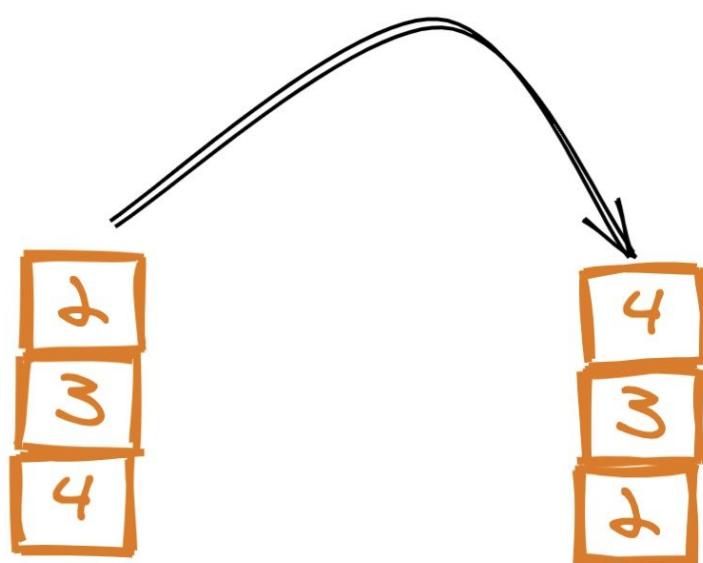


然而, 如果我们这么做, 1 虽然是正常返回了, 但是 2, 3, 4 不就永远消失了么? 一种简答方法就是, 将 2, 3, 4 存起来。而题目又说了, 只能使用栈这种数据结构, 那么我们考虑使用一个额外的栈来存放弹出的 2, 3, 4。

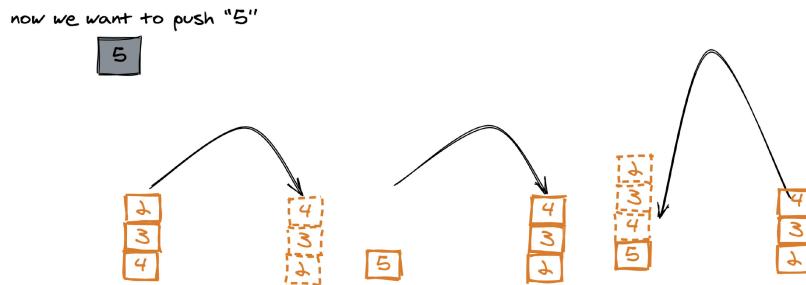


(pop 出来不扔掉, 而是存起来)

整个过程类似这样:



比如, 这个时候, 我们想 push 一个 5, 那么大概就是这样的:



然而这一过程，我们也可以发生在 push 阶段。

总之，就是我们需要在 push 或者 pop 的时候，将数组在两个栈之间倒腾一次。

## 关键点

- 在 push 的时候利用辅助栈(双栈)

## 代码

- 语言支持：JS, Python, Java

Javascript Code:

```
```js {12,22-29} /*
  • @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 that
  element.
  • @return {number} */
MyQueue.prototype.pop = function () { return
  this.stack.pop(); }

```

```

## 989. 数组形式的整数加法

- 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 such:
  - 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 {8, 14-18} 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 returns
    """
    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 such:

obj = MyQueue()

obj.push(x)

param_2 = obj.pop()

param_3 = obj.peek()

param_4 = obj.empty()

Java Code

```
```java {2-3,12-14,20-22,28-30}
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 returns that element. */
 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 such:
 * 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： $O(N)$ ，其中  $N$  为 栈中元素个数，因为每次我们都要倒腾一次。
- 空间复杂度： $O(N)$ ，其中  $N$  为 栈中元素个数，多使用了一个辅助栈，这个辅助栈的大小和原栈的大小一样。

## 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 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

989. 数组形式的整数加法



欢迎长按关注



*Originally posted by @azl397985856 in <https://github.com/leetcode-pp/91alg-1/issues/21#issuecomment-639573715>*

## 题目地址(768. 最多能完成排序的块 II)

<https://leetcode-cn.com/problems/max-chunks-to-make-sorted-ii/>

### 入选理由

1. 第一个 hard 题
2. 这是一个哈希表的题目，也可使用栈来优化。

### 题目描述

这个问题和“最多能完成排序的块”相似，但给定数组中的元素可以重复，输入数  
arr是一个可能包含重复元素的整数数组，我们将这个数组分割成几个“块”，并  
我们最多能将数组分成多少块？

示例 1：

输入： arr = [5,4,3,2,1]  
输出： 1  
解释：  
将数组分成2块或者更多块，都无法得到所需的结果。  
例如，分成 [5, 4], [3, 2, 1] 的结果是 [4, 5, 1, 2, 3]，这不是有序的。

示例 2：

输入： arr = [2,1,3,4,4]  
输出： 4  
解释：  
我们可以把它分成两块，例如 [2, 1], [3, 4, 4]。  
然而，分成 [2, 1], [3], [4], [4] 可以得到最多的块数。  
注意：

arr的长度在[1, 2000]之间。  
arr[i]的大小在[0, 10\*\*8]之间。

### 难度

- 困难

### 标签

- 栈
- 哈希表

## 前置知识

- 栈
- 队列

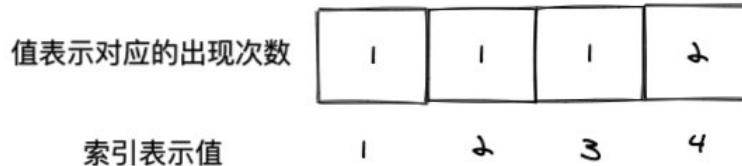
## 计数

### 思路

这里可以使用类似计数排序的技巧来完成。以题目给的 [2,1,3,4,4] 来说：



可以先计数，比如用一个数组来计数，其中数组的索引表示值，数组的值表示其对应的出现次数。比如上面，除了 4 出现了两次，其他均出现一次，因此 count 就是 [0,1,1,1,2]。



其中 counts[4] 就是 2，表示的就是 4 这个值出现了两次。

实际上 count 最开始的 0 是没有必要的，不过这样方便理解罢了。

如果我们使用数组来计数，那么空间复杂度就是  $\$upper - lower\$$ ，其中 upper 是 arr 的最大值，lower 是 arr 的最小值。

计数完毕之后，我们要做的是比较当前的 arr 和最终的 arr（已经有序的 arr）的计数数组的关系即可。

这里有一个关键点：如果两个数组的计数信息是一致的，那么两个数组排序后的结果也是一致的。如果你理解计数排序，应该明白我的意思。不明白也没有关系，我稍微解释一下你就懂了。

如果我把一个数组打乱，然后排序，得到的数组一定是确定的，即不管你  
怎么打乱排好序都是一个确定的有序序列。这个论点的正确性是毋庸置疑的。  
而实际上，一个数组无论怎么打乱，其计数结果也是确定的，这也是  
毋庸置疑的。反之，如果是两个不同的数组，打乱排序后的结果一定  
是不同的，计数也是同理。



(这两个数组排序后的结果以及计数信息是一致的)

因此我们的算法有了：

- 先排序 arr, 不妨记排序后的 arr 为 sorted\_arr
- 从左到右遍历 arr, 比如遍历到了索引为 i 的元素, 其中  $0 \leq i < \text{len}(\text{arr})$
- 如果  $\text{arr}[:i+1]$  的计数信息和  $\text{sorted\_arr}[:i+1]$  的计数信息一致, 那么说明可以分桶, 否则不可以。

$\text{arr}[:i+1]$  指的是 arr 的切片, 从索引 0 到 索引 i 的一个切片。

## 关键点

- 计数

## 代码

语言支持：Python

```
```py {7-10} class Solution(object):
    def maxChunksToSorted(self, arr):
        count_a = collections.defaultdict(int)
        count_b = collections.defaultdict(int)
        ans = 0
```

989. 数组形式的整数加法

```
for a, b in zip(arr, sorted(arr)):
    count_a[a] += 1
    count_b[b] += 1
    if count_a == count_b: ans += 1

return ans
```

****复杂度分析****

- 时间复杂度：内部 `count_a` 和 `count_b` 的比较时间复杂度也是 $O(N)$
- 空间复杂度：使用了两个 `counter`，其大小都是 N ，因此空间复杂度为 $O(N)$

优化的计数

思路

实际上，我们不需要两个 `counter`，而是使用一个 `counter` 来记录 `arr` 和 `sorted(arr)` 中的计数。

我们还可以在时间上进一步优化，去除内部 `count_a` 和 `count_b` 的比较，

关键点

- 计数
- `count` 的边界条件

代码

语言支持：Python

```
```py {9-15}
class Solution(object):
 class Solution(object):
 def maxChunksToSorted(self, arr):
 count = collections.defaultdict(int)
 non_zero_cnt = 0
 ans = 0

 for a, b in zip(arr, sorted(arr)):
 if count[a] == -1: non_zero_cnt -= 1
 if count[a] == 0: non_zero_cnt += 1
 count[a] += 1
 if count[b] == 1: non_zero_cnt -= 1
 if count[b] == 0: non_zero_cnt += 1
 count[b] -= 1
 if non_zero_cnt == 0: ans += 1

 return ans
```

**复杂度分析**

- 时间复杂度：瓶颈在于排序，因此时间复杂度为  $O(N \log N)$ ，其中  $N$  为数组长度。

- 空间复杂度：使用了一个 counter，其大小是 N，因此空间复杂度为  $O(N)$ ，其中 N 为数组长度。

## 单调栈

### 思路

通过题目给的三个例子，应该可以发现一些端倪。

- 如果 arr 是非递减的，那么答案为 1。
- 如果 arr 是非递增的，那么答案是 arr 的长度。

并且由于只有分的块内部可以排序，块与块之间的相对位置是不能变的。因此直观上我们的核心其实找到从左到右开始不减少（增加或者不变）的地方并分块。

比如对于 [5,4,3,2,1] 来说：

- 5 的下一个数是 4，比 5 小，因此如果分块，那么永远不能变成 [1,2,3,4,5]。
- 同理，4 的下一个数是 3，比 4 小，因此如果分块，那么永远不能变成 [1,2,3,4,5]。
- ...

最后就是不能只能是整体是一个大块，我们返回 1 即可。

我们继续分析一个稍微复杂一点的，即题目给的 [2,1,3,4,4]。

- 2 的下一个数是 1，比 2 小，不能分块。
- 1 的下一个数是 3，比 1 大，可以分块。
- 3 的下一个数是 4，比 3 大，可以分块。
- 4 的下一个数是 4，一样大，可以分块。

因此答案就是 4，分别是：

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

然而上面的算法步骤是不正确的，原因在于只考虑局部，没有考虑整体，比如 [4,2,2,1,1] 这样的测试用例，实际上只应该返回 1，原因是后面碰到了 1，使得前面不应该分块。

因为把数组分成数个块，分别排序每个块后，组合所有的块就跟整个数组排序的结果一样，这就意味着后面块中的最小值一定大于前面块的最大值，这样才能保证分块有。因此直观上，我们又会觉得是不是“只要后面有较小值，那么前面大于它的都应该在一个块里面”，实际上的确如此。

有没有注意到我们一直在找下一个比当前小的元素？这就是一个信号，使用单调递增栈即可以空间换时间的方式解决。对单调栈不熟悉的小伙伴可以看下我的[单调栈专题](#)

不过这还不够，我们要把思路逆转！



这是《逆转裁判》中经典的台词，主角在深处绝境的时候，会突然冒出这句话，从而逆转思维，寻求突破口。

这里的话，我们将思路逆转，不是分割区块，而是融合区块。

比如 [2,1,3,4,4]，遍历到 1 的时候会发现 1 比 2 小，因此 2, 1 需要在一块，我们可以将 2 和 1 融合，并重新压回栈。那么融合成 1 还是 2 呢？答案是 2，因为 2 是瓶颈，这提示我们可以用一个递增栈来完成。

因此本质上栈存储的每一个元素就代表一个块，而栈里面的每一个元素的值就是块的最大值。

以 [2,1,3,4,4] 来说，stack 的变化过程大概是：

- [2]
- 1 被融合了，保持 [2] 不变
- [2,3]
- [2,3,4]
- [2,3,4,4]

简单来说，就是将一个减序列压缩合并成最该序列的最大的值。因此最终返回 stack 的长度就可以了。

具体算法参考代码区，注释很详细。

## 代码

语言支持：Python, CPP, Java, JS

## 989. 数组形式的整数加法

```
class Solution:
 def maxChunksToSorted(self, A: [int]) -> int:
 stack = []
 for a in A:
 # 遇到一个比栈顶小的元素，而前面的块不应该有比 a 小的
 # 而栈中每一个元素都是一个块，并且栈的存的是块的最大值,
 if stack and stack[-1] > a:
 # 我们需要将融合后的区块的最大值重新放回栈
 # 而 stack 是递增的，因此 stack[-1] 是最大的
 cur = stack[-1]
 # 维持栈的单调递增
 while stack and stack[-1] > a: stack.pop()
 stack.append(cur)
 else:
 stack.append(a)
 # 栈存的是块信息，因此栈的大小就是块的数量
 return len(stack)
```

CPP:

```
```cpp {8,13-15} class Solution { public: int maxChunksToSorted(vector& arr) { stack stack; for(int i = 0; i < arr.size(); i){ // 我们需要将融合后的区块的最大值重新放回栈 // 而 stack 是递增的，因此 stack[-1] 是最大的 int cur = stack.top(); // 维持栈的单调递增 while(!stack.empty() && stack.top() > arr[i]) { sstackta.pop(); } stack.push(cur); }else{ stack.push(arr[i]); } } // 栈存的是块信息，因此栈的大小就是块的数量 return stack.size(); } };
```

JAVA:

```

```java {7,12-14}
class Solution {
 public int maxChunksToSorted(int[] arr) {
 LinkedList<Integer> stack = new LinkedList<Integer>;
 for (int num : arr) {
 // 遇到一个比栈顶小的元素，而前面的块不应该有比 a 小的
 // 而栈中每一个元素都是一个块，并且栈的存的是块的最大值
 if (!stack.isEmpty() && num < stack.getLast())
 // 我们需要将融合后的区块的最大值重新放回栈
 // 而 stack 是递增的，因此 stack[-1] 是最大的
 int cur = stack.removeLast();
 // 维持栈的单调递增
 while (!stack.isEmpty() && num < stack.getLast())
 stack.removeLast();
 }
 stack.addLast(cur);
 } else {
 stack.addLast(num);
 }
 }
 // 栈存的是块信息，因此栈的大小就是块的数量
 return stack.size();
}
```

```

JS:

```

js {5,7} var maxChunksToSorted = function (arr) { const
stack = []; for (let i = 0; i < arr.length; i++) { a =
arr[i]; if (stack.length > 0 && stack[stack.length - 1] > a)
{ const cur = stack[stack.length - 1]; while (stack &&
stack[stack.length - 1] > a) stack.pop(); stack.push(cur); }
else { stack.push(a); } } return stack.length; };

```

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: $O(N)$, 其中 N 为数组长度。

总结

实际上本题的单调栈思路和 [【力扣加加】从排序到线性扫描\(57. 插入区间\)](#) 以及 [394. 字符串解码](#) 都有部分相似，大家可以结合起来理解。

989. 数组形式的整数加法

融合与 [【力扣加加】从排序到线性扫描\(57. 插入区间\)](#) 相似，重新压栈和 [394. 字符串解码](#) 相似。

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 37K 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。

题目地址(61. 旋转链表)

<https://leetcode-cn.com/problems/rotate-list/>

入选理由

1. 难度低，适合链表开篇
2. 考察频率高，不瞒您说，我在面试中就被问到过

标签

- 链表

难度

- 简单

前置知识

-求单链表的倒数第 N 个节点

题目描述

给定一个链表，旋转链表，将链表每个节点向右移动 k 个位置，其中 k 是非负整数。

示例 1：

输入： $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow \text{NULL}$, $k = 2$

输出： $4 \rightarrow 5 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow \text{NULL}$

解释：

向右旋转 1 步： $5 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow \text{NULL}$

向右旋转 2 步： $4 \rightarrow 5 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow \text{NULL}$

示例 2：

输入： $0 \rightarrow 1 \rightarrow 2 \rightarrow \text{NULL}$, $k = 4$

输出： $2 \rightarrow 0 \rightarrow 1 \rightarrow \text{NULL}$

解释：

向右旋转 1 步： $2 \rightarrow 0 \rightarrow 1 \rightarrow \text{NULL}$

向右旋转 2 步： $1 \rightarrow 2 \rightarrow 0 \rightarrow \text{NULL}$

向右旋转 3 步： $0 \rightarrow 1 \rightarrow 2 \rightarrow \text{NULL}$

向右旋转 4 步： $2 \rightarrow 0 \rightarrow 1 \rightarrow \text{NULL}$

思路

首先我们看下如何返回链表倒数第 k 个节点。

1. 采用快慢指针
2. 快指针与慢指针都以每步一个节点的速度向后遍历
3. 快指针比慢指针先走 k 步
4. 当快指针到达终点时，慢指针正好是倒数第 k 个节点

伪代码：

```
快指针 = head;
慢指针 = head;
while (快指针.next) {
    if (k-- <= 0) {
        慢指针 = 慢指针.next;
    }
    快指针 = 快指针.next;
}
```

我们将上面的代码改为真实代码。

JS Code:

```
let slow = (fast = head);
while (fast.next) {
    if (k-- <= 0) {
        slow = slow.next;
    }
    fast = fast.next;
}
```

有了上面的知识，我们来看下具体如何解决这道题。

算法描述：

1. 获取单链表的倒数第 1 (尾节点) 与倒数第 2 个节点
2. 将倒数第 2 个节点的 next 指向 null
3. 将尾节点的 next 指向 head (拼起来)
4. 返回倒数第 1 个节点

经过这样的处理，我们旋转了一位，而题目是要旋转 k 位，实际上我们只需要将上面的算法微调即可。将 1 改成 k ，2 改成 $k + 1$ 。

算法描述：

1. 获取单链表的倒数第 k 与倒数第 $k + 1$ 个节点
2. 将倒数第 $k + 1$ 个节点的 next 指向 null

989. 数组形式的整数加法

3. 将尾节点 next 指向 head (拼起来)
4. 返回倒数第 k 个节点

例如链表 A -> B -> C -> D -> E 右移 2 位，依照上述步骤为：

1. 获取节点 C 与 D
2. A -> B -> C -> null, D -> E
3. D -> E -> A -> B -> C -> null
4. 返回节点 D

注意：假如链表节点长度为 len，则右移 K 位与右移动 $k \% len$ 的效果是一样的就像是长度为 1000 米的环形跑道，你跑 1100 米与跑 100 米到达的是同一个地点

据此不难写出如下伪代码：

```
获取链表的长度  
k = k % 链表的长度  
获取倒数第k + 1, 倒数第K个节点与链表尾节点  
倒数第k + 1个节点.next = null  
链表尾节点.next = head  
return 倒数第k个节点
```

代码

代码支持：JS, Java, Python

989. 数组形式的整数加法

```
var rotateRight = function (head, k) {
    if (!head || !head.next) return head;
    let count = 0,
        now = head;
    while (now) {
        now = now.next;
        count++;
    }
    k = k % count;
    let slow = (fast = head);
    while (fast.next) {
        if (k-- <= 0) {
            slow = slow.next;
        }
        fast = fast.next;
    }
    fast.next = head;
    let res = slow.next;
    slow.next = null;
    return res;
};
```

Java Code:

989. 数组形式的整数加法

```
class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if(head == null || head.next == null) return head;
        int count = 0;
        ListNode now = head;
        while(now != null){
            now = now.next;
            count++;
        }
        k = k % count;
        ListNode slow = head, fast = head;
        while(fast.next != null){
            if(k-- <= 0){
                slow = slow.next;
            }
            fast = fast.next;
        }
        fast.next = head;
        ListNode res = slow.next;
        slow.next = null;
        return res;
    }
}
```

Python Code:

989. 数组形式的整数加法

```
class Solution:
    def rotateRight(self, head: ListNode, k: int) -> ListNode:
        # 双指针
        if head:
            p1 = head
            p2 = head
            count = 1
            i = 0
            while i < k:
                if p2.next:
                    count += 1
                    p2 = p2.next
                else:
                    k = k % count
                    i = -1
                    p2 = head
                i += 1

            while p2.next:
                p1 = p1.next
                p2 = p2.next

            if p1.next:
                tmp = p1.next
            else:
                return head
            p1.next = None
            p2.next = head
            return tmp
```

C++ Code

```
ListNode* rotateRight(ListNode* head, int k) {
    if (head == nullptr
        || head->next == nullptr
        || k == 0)
        return head;

    int len = 1;
    ListNode* cur = head;
    while (cur->next != nullptr) {
        cur = cur->next;
        len++;
    }

    k %= len;

    ListNode* fast = head;
    ListNode* slow = head;

    while (fast->next != nullptr) {
        if (k-- <= 0) {
            slow = slow->next;
        }
        fast = fast->next;
    }

    fast->next = head;
    ListNode* new_head = slow->next;
    slow->next = nullptr;
    return new_head;
}
```

复杂度分析

- 时间复杂度：节点最多只遍历两遍，时间复杂度为 $O(n)$
- 空间复杂度：未使用额外的空间，空间复杂度 $O(1)$

题目地址(24. 两两交换链表中的节点)

<https://leetcode-cn.com/problems/swap-nodes-in-pairs/>

入选理由

1. 链表常规操作就是改变指针，这次其实就是两两反转再拼接，因此比昨天的题多了操作，那么你还会么？
2. 练习虚拟节点的使用，这个技巧在头结点可能会发生变化的时候都可以用。

标签

- 链表

前置知识

- 链表的基本知识

难度

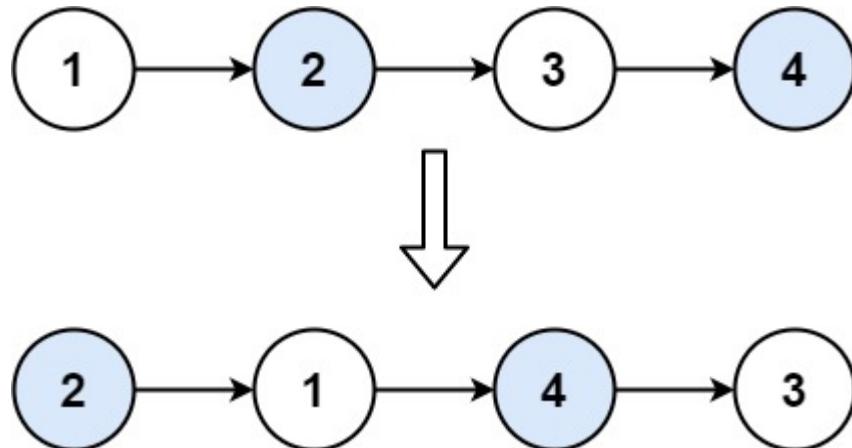
- 中等

题目描述

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

示例 1：



输入: head = [1,2,3,4]

输出: [2,1,4,3]

示例 2:

输入: head = []

输出: []

示例 3:

输入: head = [1]

输出: [1]

提示:

链表中节点的数目在范围 [0, 100] 内

$0 \leq \text{Node.val} \leq 100$

迭代

思路

这道题其实考察的内容就是链表节点的指针的指向。

由于所有的两两交换逻辑都是一样的，因此我们只要关注某一个两两交换如何实现就可以了。

因为要修改的是二个一组的链表节点，所以需要操作 4 个节点。例如：将链表 A -> B 进行逆转，我们需要得到 A,B 以及 A 的前置节点 preA, 以及 B 的后置节点 nextB

原始链表为 preA -> A -> B -> nextB，我们需要改为 preA -> B -> A -> nextB，接下来用同样的逻辑交换 nextB 以及 nextB 的下一个元素。

那么修改指针的顺序为：

1. A 节点的 next 指向 nextB:

```
preA -> A -> nextB  
B -> nextB
```

1. B 节点的 next 指向 A

```
preA -> A -> nextB  
B -> A
```

1. preA 节点的 next 指向 B

```
preA -> B -> A -> nextB
```

伪代码：

```
A.next = next.B;  
B.next = A;  
preA.next = B;
```

我们可以创建一个空节点 preHead，让其 next 指针指向 A(充当 preA 的角色)，这样是我们专注于算法逻辑，避免判断边界条件。这涉及到链表指针修改的时候头节点可能发生变化的时候非常常用。

例如当前链表为： A -> B -> C -> D， 使用虚拟节点后为 preHead -> A -> B -> C -> D

按照上诉步骤修改指针的 3 步后，链表为

```
preHead -> B -> A -> C -> D
```

这时让 preHead 指向 A，继续上述步骤逆转 C -> D, 循环此步骤直到整个链表被逆转

伪代码：

```

if 为空表或者只有一个节点{
    return head
}
let 前置指针 = new 链表节点
前置指针.next = head
第一个节点 = head
返回的结果 = 第一个节点
while(第一个节点存在 && 第一个节点.next不为空){
    第二个节点 = 第一个节点.next
    后置指针 = 第二个节点.next

    // 对链表进行逆转
    第一个节点.next = 后置指针
    第二个节点.next = 第一个节点
    前置指针.next = 第二个节点

    // 修改指针位置, 进行下一轮逆转
    前置指针 = 第一个节点
    第一个节点 = 后置指针
}
return 返回的结果

```

代码

代码支持: JS, Java, Python3, C++

JS Code:

```

var swapPairs = function (head) {
    if (!head || !head.next) return head;
    let res = head.next;
    let now = head;
    let preNode = new ListNode();
    preNode.next = head;
    while (now && now.next) {
        let nextNode = now.next;
        let nnNode = nextNode.next;
        now.next = nnNode;
        nextNode.next = now;
        preNode.next = nextNode;
        preNode = now;
        now = nnNode;
    }
    return res;
};

```

989. 数组形式的整数加法

Java Code:

```
class Solution {
    public ListNode swapPairs(ListNode head) {
        if(head == null || head.next == null) return head;
        ListNode preNode = new ListNode(-1, head), res;
        preNode.next = head;
        res = head.next;
        ListNode firstNode = head, secondNode, nextNode;
        while(firstNode != null && firstNode.next != null){
            secondNode = firstNode.next;
            nextNode = secondNode.next;

            firstNode.next = nextNode;
            secondNode.next = firstNode;
            preNode.next = secondNode;

            preNode = firstNode;
            firstNode = nextNode;
        }
        return res;
    }
}
```

Python3 Code:

```
if not head or not head.next: return head
ans = ListNode()
ans.next = head.next
pre = ans
while head and head.next:
    next = head.next
    n_next = next.next

    next.next = head
    pre.next = next
    head.next = n_next
    # 更新指针
    pre = head
    head = n_next
return ans.next
```

C++ Code:

```

ListNode* swapPairs(ListNode* head) {
    if (head == nullptr || head->next == nullptr) return head;

    ListNode* dummy = new ListNode(-1, head);
    ListNode* prev = dummy;
    ListNode* cur = prev->next;

    while (cur != nullptr && cur->next != nullptr) {
        ListNode* next = cur->next;
        cur->next = next->next;
        next->next = cur;
        prev->next = next;

        prev = cur;
        cur = cur->next;
    }
    return dummy->next;
}

```

复杂度分析

- 时间复杂度：所有节点只遍历一遍，时间复杂度为\$O(N)\$
- 空间复杂度：未使用额外的空间，空间复杂度\$O(1)\$

递归

思路

- 关注最小子结构，即将两个节点进行逆转。
- 将逆转后的尾节点.next 指向下一次递归的返回值
- 返回逆转后的链表头节点 (ps:逆转前的第二个节点)

如果看不懂，建议看下我写的[链表专题](#)，里面详细讲述了这个技巧。

伪代码：

```

function run(head)
    if 为空表或者只有一个节点{
        return head
    }
    后一个节点 = head.next
    head.next = run(后一个节点.next)
    后一个节点.next = head
    return 后一个节点
}

```

代码

代码支持: JS, Java, Python, C++

JS Code:

```
var swapPairs = function (head) {
    if (!head || !head.next) return head;
    let nextNode = head.next;
    head.next = swapPairs(nextNode.next);
    nextNode.next = head;
    return nextNode;
};
```

Java Code:

```
class Solution {
    public ListNode swapPairs(ListNode head) {
        if(head == null || head.next == null) return head;
        ListNode nextNode = head.next;
        head.next = swapPairs(nextNode.next);
        nextNode.next = head;
        return nextNode;
    }
}
```

Python3 Code:

```
class Solution:
    def swapPairs(self, head: ListNode) -> ListNode:
        if not head or not head.next: return head

        next = head.next
        head.next = self.swapPairs(next.next)
        next.next = head

        return next
```

C++ Code:

989. 数组形式的整数加法

```
ListNode* swapPairs(ListNode* head) {
    if (head == nullptr || head->next == nullptr) return head;

    ListNode* first = head;
    ListNode* second = first->next;

    ListNode* head_of_next_group = swapPairs(second->next);

    first->next = head_of_next_group;
    second->next = first;

    return second;
}
```

复杂度分析

- 时间复杂度：所有节点只遍历一遍，时间复杂度为 $O(N)$
- 空间复杂度：未使用额外的空间(递归造成的函数栈除外)，空间复杂度 $O(1)$

题目地址(109. 有序链表转换二叉搜索树)

<https://leetcode-cn.com/problems/convert-sorted-list-to-binary-search-tree/>

入选理由

1. 和二叉搜索树联动，大家可以提前预习一下。
2. 链表的题目，我们核心思路就是链表的基本操作，别想别的。比如链表上的快排，如果你不会快排，那么肯定做不出来，但不表示你不会链表，因此大家学习的时候一定要分清这些。

难度

- 中等

标签

- 链表
- 二叉搜索树

题目描述

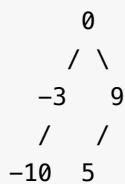
给定一个单链表，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。

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



前置知识

- 递归
- 二叉搜索树的任意一个节点，当前节点的值必然大于所有左子树节点的值。同理，当前节点的值必然小于所有右子树节点的值

双指针法

思路

使用快慢双指针可定位中间元素，具体可参考双指针的讲义。这里我简单描述一下算法流程：

1. 获取当前链表的中点
2. 以链表中点为根
3. 中点左边的值都小于它，可以构造左子树
4. 同理构造右子树
5. 循环第一步

具体算法：

1. 定义一个快指针每步前进两个节点，一个慢指针每步前进一个节点
2. 当快指针到达尾部的时候，正好慢指针所到的点为中点

代码

代码支持：JS,Java,Python,C++

JS Code

```
var sortedListToBST = function (head) {
    if (!head) return null;
    return dfs(head, null);
};

function dfs(head, tail) {
    if (head == tail) return null;
    let fast = head;
    let slow = head;
    while (fast != tail && fast.next != tail) {
        fast = fast.next.next;
        slow = slow.next;
    }
    let root = new TreeNode(slow.val);
    root.left = dfs(head, slow);
    root.right = dfs(slow.next, tail);
    return root;
}
```

989. 数组形式的整数加法

Java Code:

```
class Solution {
    public TreeNode sortedListToBST(ListNode head) {
        if(head == null) return null;
        return dfs(head,null);
    }
    private TreeNode dfs(ListNode head, ListNode tail){
        if(head == tail) return null;
        ListNode fast = head, slow = head;
        while(fast != tail && fast.next != tail){
            fast = fast.next.next;
            slow = slow.next;
        }
        TreeNode root = new TreeNode(slow.val);
        root.left = dfs(head, slow);
        root.right = dfs(slow.next, tail);
        return root;
    }
}
```

Python 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
```

C++ Code:

```

class Solution {
public:
    TreeNode* sortedListToBST(ListNode* head) {
        if (head == nullptr) return nullptr;
        return sortedListToBST(head, nullptr);
    }
    TreeNode* sortedListToBST(ListNode* head, ListNode* tail) {
        if (head == tail) return nullptr;

        ListNode* slow = head;
        ListNode* fast = head;

        while (fast != tail && fast->next != tail) {
            slow = slow->next;
            fast = fast->next->next;
        }

        TreeNode* root = new TreeNode(slow->val);
        root->left = sortedListToBST(head, slow);
        root->right = sortedListToBST(slow->next, tail);
        return root;
    }
};

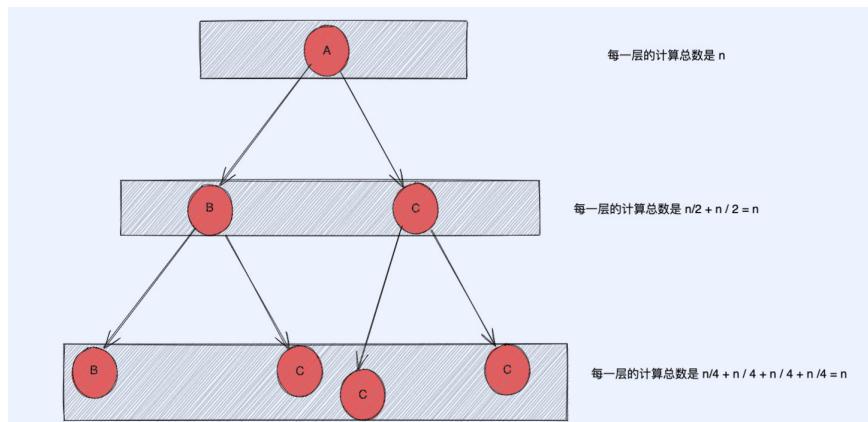
```

复杂度分析

令 n 为链表长度。

- 时间复杂度：递归树的深度为 $\log n$ ，每一层的基本操作数为 n ，因此总的时间复杂度为 $O(n \log n)$
- 空间复杂度：空间复杂度为 $O(\log n)$

有的同学不太会分析递归的时间复杂度和空间复杂度，我们在这里给大家再次介绍一下。



首先我们尝试画出如下的递归树。由于递归树的深度为 $\log n$ 因此空间复杂度就是 $\log n * \text{递归函数内部的空间复杂度}$, 由于递归函数内空间复杂度为 $O(1)$, 因此总的空间复杂度为 $O(\log n)$ 。

时间复杂度稍微困难一点点。之前西法在先导篇给大家说过：如果有递归那就是：递归树的节点数 * 递归函数内部的基础操作数。而这句话的前提是所有递归函数内部的基本操作数是一样的，这样才能直接乘。而这里递归函数的基本操作数不一样。

不过我们发现递归树内部每一层的基本操作数都是固定的，为啥固定已经在图上给大家算出来了。因此总的空间复杂度其实可以通过递归深度 * 每一层基础操作数计算得出，也就是 $n \log n$ 。类似的技巧可以用于归并排序的复杂度分析中。

另外大家也直接可以通过公式推导得出。对于这道题来说，设基本操作数 $T(n)$ ，那么就有 $T(n) = T(n/2) * 2 + n/2$ ，推导出来 $T(n)$ 大概是 $n \log n$ 。这应该高中的知识。具体推导过程如下：

$$T(n) = T(n/2) + n/2 = \frac{n}{2} + 2\left(\frac{n}{2}\right)^2 + 2^2\left(\frac{n}{2}\right)^3 + \dots = \log n$$

类似地，如果递推公式为 $T(n) = T(n/2) * 2 + 1$ ，那么 $T(n)$ 大概就是 $\log n$ 。

缓存法

思路

因为访问链表中点的时间复杂度为 $O(n)$ ，所以可以使用数组将链表的值存储，以空间换时间。

代码

代码支持：JS, C++

JS Code:

989. 数组形式的整数加法

```
var sortedListToBST = function (head) {
    let res = [];
    while (head) {
        res.push(head.val);
        head = head.next;
    }
    return dfs(res, 0, res.length - 1);
};

function dfs(res, l, r) {
    if (l > r) return null;
    let mid = parseInt((l + r) / 2 + r);
    let root = new TreeNode(res[mid]);
    root.left = dfs(res, l, mid - 1);
    root.right = dfs(res, mid + 1, r);
    return root;
}
```

C++ Code:

```
class Solution {
public:
    TreeNode* sortedListToBST(ListNode* head) {
        vector<int> nodes;
        while (head != nullptr) {
            nodes.push_back(head->val);
            head = head->next;
        }
        return sortedListToBST(nodes, 0, nodes.size());
    }

    TreeNode* sortedListToBST(vector<int>& nodes, int start, int end) {
        if (start >= end) return nullptr;

        int mid = (end - start) / 2 + start;
        TreeNode* root = new TreeNode(nodes[mid]);
        root->left = sortedListToBST(nodes, start, mid);
        root->right = sortedListToBST(nodes, mid + 1, end);
        return root;
    }
};
```

复杂度分析

令 n 为链表长度。

- 时间复杂度：递归树每个节点的时间复杂度为 $O(1)$ ，每次处理一个节点，因此总的节点数就是 n ，也就是说总的时间复杂度为

989. 数组形式的整数加法

$\$O(n)\$$ 。

- 空间复杂度：使用了数组对链表的值进行缓存，空间复杂度为 $\$O(n)\$$

题目地址(160. 相交链表)

<https://leetcode-cn.com/problems/intersection-of-two-linked-lists/>

入选理由

1. 讲义里面的题目，不应该不会
2. 考察频率相当的高

标签

- 双指针
- 链表

难度

- 简单

题目描述

编写一个程序，找到两个单链表相交的起始节点。

前置知识

- 链表
- 双指针

解法 1: 哈希法

思路

- 有 A, B 两条链表，先遍历其中一个，比如 A 链表，并将 A 中的所有节点存入哈希表。
- 接着遍历 B 链表，检查每个节点是否在哈希表中，存在于哈希表中的那个节点就是 A 链表和 B 链表的相交节点。

伪代码

989. 数组形式的整数加法

```
data = new Set() // 存放A链表的所有节点的地址

while A不为空{
    哈希表中添加A链表当前节点
    A指针向后移动
}

while B不为空{
    if 如果哈希表中含有B链表当前节点
        return B
    B指针向后移动
}

return null // 两条链表没有相交点
```

代码(JS/C++)

JS Code:

```
let data = new Set();
while (A !== null) {
    data.add(A);
    A = A.next;
}
while (B !== null) {
    if (data.has(B)) return B;
    B = B.next;
}
return null;
```

C++ Code:

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    if (headA == NULL || headB == NULL) return NULL;

    map<ListNode*, bool> seen;
    while (headA) {
        seen.insert(pair<ListNode*, bool>(headA, true));
        headA = headA->next;
    }
    while (headB) {
        if (seen.find(headB) != seen.end()) return headB;
        headB = headB->next;
    }
    return NULL;
}
```

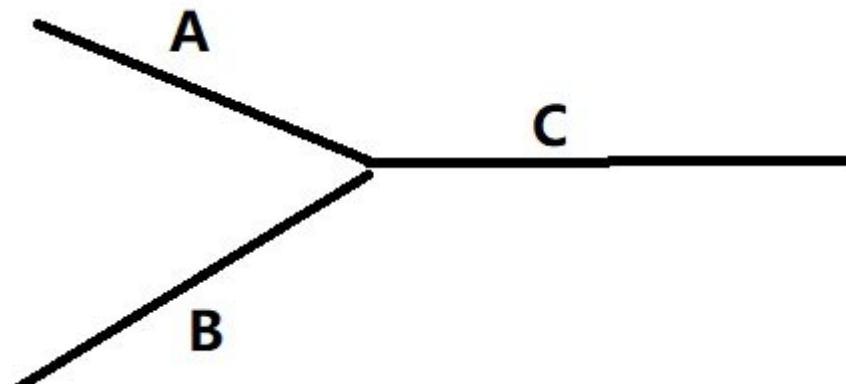
复杂度分析

- 时间复杂度: $O(N)$
- 空间复杂度: $O(N)$

解法 2：双指针

思路

- 使用两个指针如指针 a, b 分别指向 A, B 这两条链表的头节点, 两个指针以相同的速度向后移动。
- 当 a 到达链表 A 的尾部时, 将它重定位到链表 B 的头节点;
- 当 b 到达链表 B 的尾部时, 将它重定位到链表 A 的头节点;
- 若在此过程中 a, b 指针相遇, 则相遇节点为两链表相交的起始节点, 否则说明两个链表不存在相交点。



(图 5)

为什么 a, b 指针相遇的点一定是相交的起始节点? 我们证明一下:

989. 数组形式的整数加法

1. 将两条链表按相交的起始节点继续截断，链表 1 为： A + C，链表 2 为： B + C；
2. 当 a 指针将链表 1 遍历完后，重定位到链表 2 的头节点，然后继续遍历直至相交点，此时 a 指针遍历的距离为 A + C + B；
3. 同理 b 指针遍历的距离为 B + C + A；

伪代码

```
a = headA
b = headB
while a,b指针不相等时 {
    if a指针为空时
        a指针重定位到链表 B的头结点
    else
        a指针向后移动一位
    if b指针为空时
        b指针重定位到链表 A的头结点
    else
        b指针向后移动一位
}
return a
```

代码(JS/Python/C++)

JS Code:

```
var getIntersectionNode = function (headA, headB) {
    let a = headA,
        b = headB;
    while (a != b) {
        a = a === null ? headB : a.next;
        b = b === null ? headA : b.next;
    }
    return a;
};
```

Python Code:

989. 数组形式的整数加法

```
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode):
        a, b = headA, headB
        while a != b:
            a = a.next if a else headB
            b = b.next if b else headA
        return a
```

C++ Code:

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    if (headA == NULL || headB == NULL) return NULL;

    ListNode* pA = headA;
    ListNode* pB = headB;
    while (pA != pB) {
        pA = pA == NULL ? headB : pA->next;
        pB = pB == NULL ? headA : pB->next;
    }

    return pA;
}
```

复杂度分析

- 时间复杂度: $O(N)$
- 空间复杂度: $O(1)$

题目地址(142. 环形链表 II)

<https://leetcode-cn.com/problems/linked-list-cycle-ii/>

标签

- 双指针
- 链表

难度

- 中等

入选理由

1. 和昨天题目有点类似，结合起来练习效果比较好
2. 同样也是讲义中的题，考察频率同样很高

题目描述

给定一个链表，返回链表开始入环的第一个节点。如果链表无环，则返回 `null`。

为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置。

说明：不允许修改给定的链表。

进阶：

你是否可以使用 $O(1)$ 空间解决此题？

哈希法

思路

1. 遍历整个链表,同时将每个节点都插入哈希表。由于题目没有限定每个节点的值均不同，因此我们必须将节点的引用作为哈希表的键。
2. 如果当前节点在哈希表中不存在,继续遍历。
3. 如果存在,那么当前节点就是环的入口节点。

989. 数组形式的整数加法

这种做法的正确性不言而喻。这是因为如果没有环，不可能遍历到这个节点之前就已经存在于哈希表。并且第一次遍历两次的节点一定是环的位置。

伪代码：

```
data = new Set() // 声明哈希表
while head不为空{
    if 当前节点在哈希表中存在{
        return head // 当前节点就是环的入口节点
    } else {
        将当前节点插入哈希表
    }
    head指针后移
}
return null // 环不存在
```

代码

JS Code:

```
let data = new Set();
while (head) {
    if (data.has(head)) {
        return head;
    } else {
        data.add(head);
    }
    head = head.next;
}
return null;
```

C++ Code:

```
ListNode *detectCycle(ListNode *head) {
    set<ListNode*> seen;
    ListNode *cur = head;
    while (cur != NULL) {
        if (seen.find(cur) != seen.end()) return cur;
        seen.insert(cur);
        cur = cur->next;
    }
    return NULL;
}
```

复杂度分析

令 n 为链表中总的节点数。

- 时间复杂度: $O(N)$
- 空间复杂度: $O(N)$

快慢指针法

思路

上面的空间复杂度是 $O(n)$, 题目的进阶是使用 $O(1)$ 的空间来解决。该如何做呢?

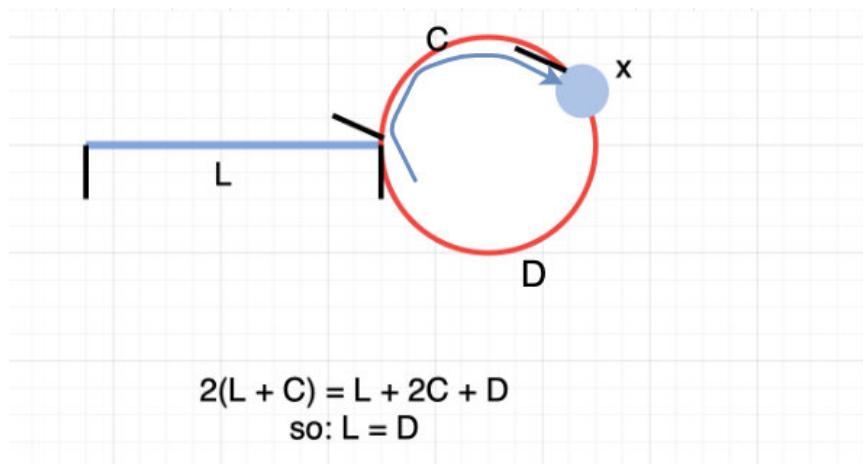
我们可以使用双指针的技巧来完成。关于双指针, 后面的专题也会详细地进行介绍。

具体算法:

1. 定义一个 `fast` 指针, 每次前进两步, 一个 `slow` 指针, 每次前进一步
2. 当两个指针相遇时
 - i. 将 `fast` 指针重定位到链表头部, 同时 `fast` 指针每次只前进一步
 - ii. `slow` 指针继续前进, 每次前进一步
3. 当两个指针再次相遇时, 当前节点就是环的入口

下面我们对此方法的正确性进行简单证明:

- x 表示第一次相遇点
- L 是起点到环的入口点的距离
- C 是环的入口点到第一次相遇点的距离
- D 是环的周长减去 C



$L + C$ 是慢指针走的路程, 而快指针走的距离是慢指针的两倍, 也就是 $2(L + C)$, 而快指针走的距离也可以用 $L + C + n(C + D)$ 表示, 其中 n 是大于等于 1 的整数, 二者结合可以得出 $L = (n-1) * (C + D) + D$ 。

989. 数组形式的整数加法

因此我们可以在两者第一次相遇后将快指针放回开头，这样二者再次相遇的点一点是环的入口点，此时慢指针又走的距离为 $D + (n-1) * (C + D)$ ，也就是说慢指针走了 D 加上绕环的 $n - 1$ 圈的距离。

That's all!

伪代码：

```
fast = head
slow = head // 快慢指针都指向头部
do {
    快指针向后两步
    慢指针向后一步
} while 快慢指针不相等时
if 指针都为空时{
    return null // 没有环
}
while 快慢指针不相等时{
    快指针向后一步
    慢指针向后一步
}
return fast
```

代码

代码支持：JS, Python3, CPP

JS Code：

```
if (head == null || head.next == null) return null;
let fast = (slow = head);
do {
    if (fast != null && fast.next != null) {
        fast = fast.next.next;
    } else {
        fast = null;
    }
    slow = slow.next;
} while (fast != slow);
if (fast == null) return null;
fast = head;
while (fast != slow) {
    fast = fast.next;
    slow = slow.next;
}
return fast;
```

989. 数组形式的整数加法

Python3 Code:

```
class Solution:
    def detectCycle(self, head: ListNode) -> ListNode:
        slow = fast = head
        x = None

        while fast and fast.next:
            fast = fast.next.next
            slow = slow.next
            if fast == slow:
                x = fast
                break
        if not x:
            return None
        slow = head
        while slow != x:
            slow = slow.next
            x = x.next
        return slow
```

C++ Code:

```
ListNode *detectCycle(ListNode *head) {
    ListNode *slow = head;
    ListNode *fast = head;

    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) {
            fast = head;
            while (slow != fast) {
                slow = slow->next;
                fast = fast->next;
            }
            return slow;
        }
    }
    return NULL;
}
```

复杂度分析

令 n 为链表总的节点数。

989. 数组形式的整数加法

- 时间复杂度: $O(N)$
- 空间复杂度: $O(1)$

题目地址(146. LRU 缓存机制)

<https://leetcode-cn.com/problems/lru-cache/>

标签

- 哈希表
- 链表

难度

- 困难

入选理由

1. 书写难度较大，当压轴题
2. 设计题是最考察数据结构知识的，希望通过这道题让大家感受链表

题目描述

运用你所掌握的数据结构，设计和实现一个 LRU（最近最少使用）缓存机制。
实现 LRUCache 类：

`LRUCache(int capacity)` 以正整数作为容量 `capacity` 初始化 LRU 缓存。
`int get(int key)` 如果关键字 `key` 存在于缓存中，则返回关键字的值，否则返回 -1。
`void put(int key, int value)` 如果关键字已经存在，则变更其数据值；如果不存在，则插入该组键值对。

进阶：你是否可以在 $O(1)$ 时间复杂度内完成这两种操作？

示例：

输入

```
["LRUCache", "put", "put", "get", "put", "get", "put", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [5, 5]]
```

输出

```
[null, null, null, 1, null, -1, null, -1, 3, 4]
```

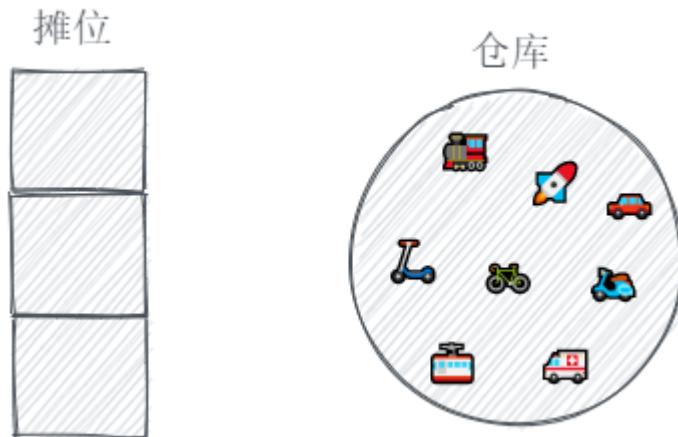
解释

```
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // 缓存是 {1=1}
lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}
lRUCache.get(1); // 返回 1
lRUCache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}
lRUCache.get(2); // 返回 -1 (未找到)
lRUCache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}
lRUCache.get(1); // 返回 -1 (未找到)
lRUCache.get(3); // 返回 3
lRUCache.get(4); // 返回 4
```

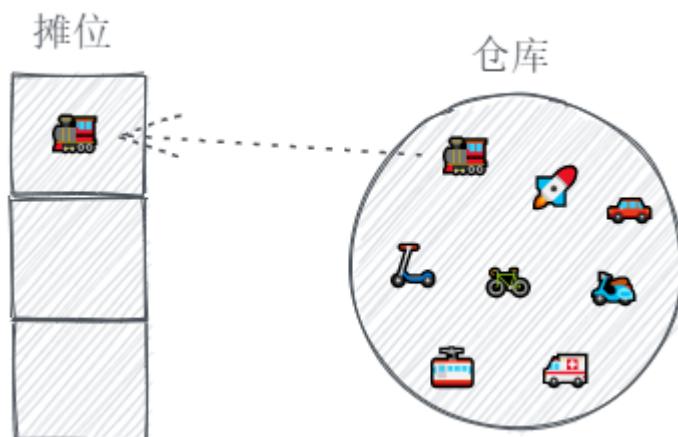
方法 1：哈希表+双向链表

思路

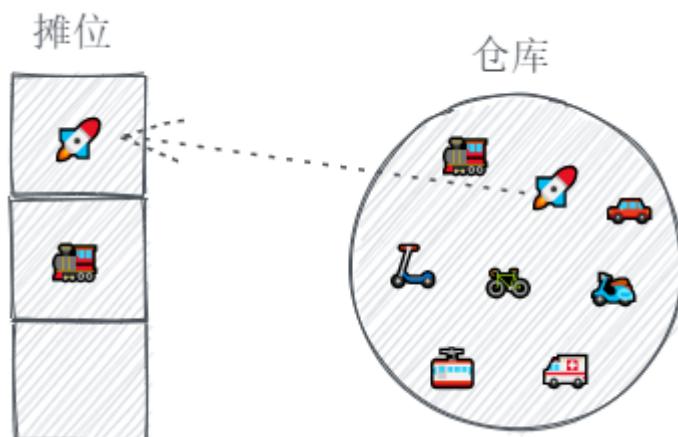
先来看个非计算机的例子理解下题意，假设我们有一个玩具摊位，可以向顾客展示小玩具。玩具很多，但摊位大小有限，不能一次性展示所有玩具，于是我们就把大部分的玩具都放在了仓库里。



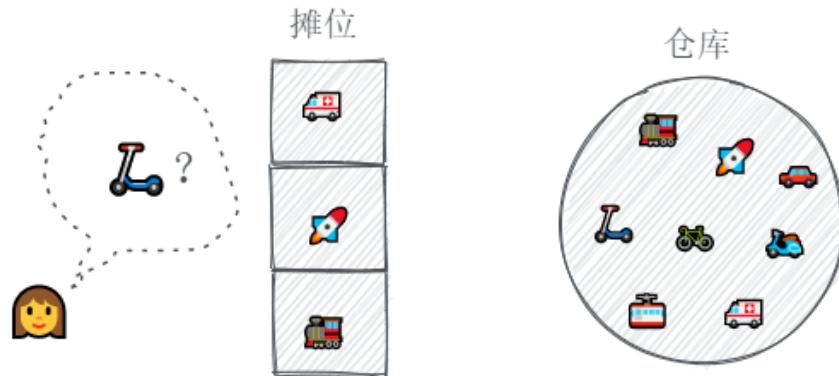
如果有顾客来询问某个玩具，我们就去仓库把那个玩具拿出来，摆在摊位上。



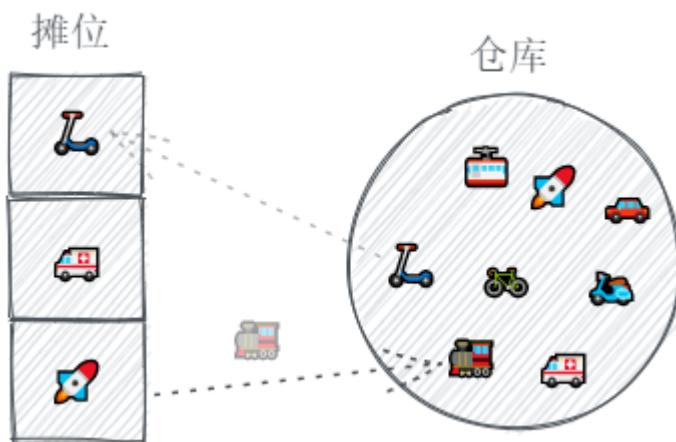
因为摊位最上面的位置最显眼，所以我们总是把最新拿出来的玩具放在那。



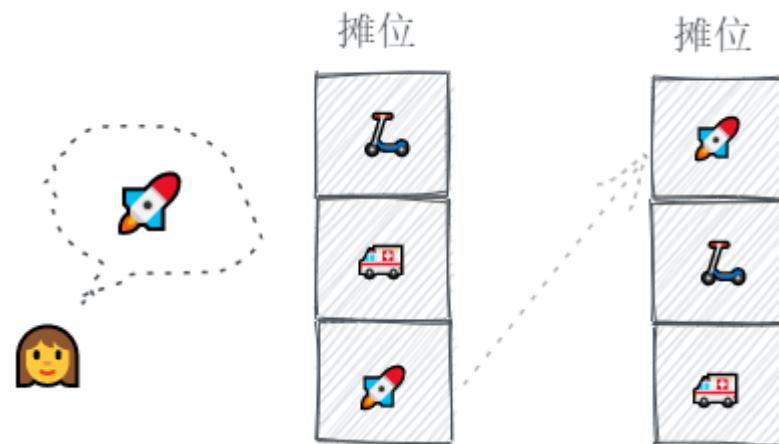
不过由于摊位大小有限，很快就摆满了，这时如果又来了顾客想看新玩具。



我们只能把摊位最下面的玩具拿回仓库(因为最下面的位置相对没那么受欢迎), 然后其他玩具往下移, 腾出最上面的位置来放新玩具。



如果顾客想看的玩具就摆在摊位上, 我们就可以把这个玩具直接移到摊位最上面的位置, 其他的玩具就要往下挪挪位置了。还记得我们的规则吧, 最近有人询问的玩具要摆在最上面显眼的位置。



回到计算机问题上面来, 玩具摊位代表的就是缓存空间, 我们需要考虑的问题是使用哪种数据结构来表示玩具摊位。

选择 1: 数组

如果选择数组，因为玩具在摊位上的位置会挪来挪去，时间复杂度是 $O(N)$ ，不符合题意。

选择 2：链表

- 如果选择链表，我们知道在已知位置上新增节点，或者移除一个已知节点的时间复杂度是 $O(1)$ 。不过，链表查找节点的时间复杂度是 $O(N)$ ，同样不符合题意，但这还有办法补救。
- 在玩具摊位的例子中，我们手动移动玩具的时候，只需要看一眼就知道要找的玩具在哪个位置上，但计算机没那么聪明，因此还需要给它一个脑子(哈希表)来记录什么玩具在什么位置上，也就是要用一个哈希表来记录每个 key 对应的链表节点引用。这样查找链表节点的时间复杂度就降到了 $O(1)$ ，不过代价是空间复杂度增加到了 $O(N)$ 。
- 另外，由于移除链表节点后还需要把该节点前后的两个节点连起来，因此我们需要的是双向链表而不是单向链表。

复杂度分析

- 时间复杂度： $O(1)$ 。
- 空间复杂度：链表 $O(N)$ ，哈希表 $O(N)$ ，结果还是 $O(N)$ ， N 为容量大小。

伪代码

989. 数组形式的整数加法

```
// put

if key 存在:
    更新节点值
    把节点移到链表头部

else:
    if 缓存满了:
        移除最后一个节点
        删除它在哈希表中的映射

    新建一个节点
    把节点加到链表头部
    在哈希表中增加映射


// get

if key 存在:
    返回节点值
    把节点移到链表头部
else:
    返回 -1
```

代码(JavaScript/C++)

JavaScript Code

989. 数组形式的整数加法

```
class DoubleLinkedListNode {
    constructor(key, value) {
        this.key = key;
        this.value = value;
        this.prev = null;
        this.next = null;
    }
}

class LRUCache {
    constructor(capacity) {
        this.capacity = capacity;
        // Mappings of key->node.
        this.hashmap = {};
        // Use two dummy nodes so that we don't have to deal w:
        this.dummyHead = new DoubleLinkedListNode(null, null);
        this.dummyTail = new DoubleLinkedListNode(null, null);
        this.dummyHead.next = this.dummyTail;
        this.dummyTail.prev = this.dummyHead;
    }

    _isFull() {
        return Object.keys(this.hashmap).length === this.capacity;
    }

    _removeNode(node) {
        node.prev.next = node.next;
        node.next.prev = node.prev;
        node.prev = null;
        node.next = null;
        return node;
    }

    _addToHead(node) {
        const head = this.dummyHead.next;
        node.next = head;
        head.prev = node;
        node.prev = this.dummyHead;
        this.dummyHead.next = node;
    }

    get(key) {
        if (key in this.hashmap) {
            const node = this.hashmap[key];
            this._addToHead(this._removeNode(node));
            return node.value;
        } else {
            return -1;
        }
    }
}
```

989. 数组形式的整数加法

```
        }

    put(key, value) {
        if (key in this.hashmap) {
            // If key exists, update the corresponding node and remove it from the list.
            const node = this.hashmap[key];
            node.value = value;
            this._addToHead(this._removeNode(node));
        } else {
            // If it's a new key.
            if (this._isFull()) {
                // If the cache is full, remove the tail node.
                const node = this.dummyTail.prev;
                delete this.hashmap[node.key];
                this._removeNode(node);
            }
            // Create a new node and add it to the head.
            const node = new DoubleLinkedListNode(key, value);
            this.hashmap[key] = node;
            this._addToHead(node);
        }
    }

    /**
     * Your LRUCache object will be instantiated and called as follows:
     * var obj = new LRUCache(capacity)
     * var param_1 = obj.get(key)
     * obj.put(key,value)
     */
}
```

C++ Code

989. 数组形式的整数加法

```
class DLinkedListNode {
public:
    int key;
    int value;
    DLinkedListNode *prev;
    DLinkedListNode *next;
    DLinkedListNode() : key(0), value(0), prev(NULL), next(NULL) {}
    DLinkedListNode(int k, int val) : key(k), value(val), prev(NULL), next(NULL) {}
};

class LRUCache {
public:
    LRUCache(int capacity) : capacity_(capacity) {
        // 创建两个 dummy 节点来简化操作，这样就不用特殊对待头尾节点了
        dummy_head_ = new DLinkedListNode();
        dummy_tail_ = new DLinkedListNode();
        dummy_head_->next = dummy_tail_;
        dummy_tail_->prev = dummy_head_;
    }

    int get(int key) {
        if (!key_exists_(key)) {
            return -1;
        }
        // 1. 通过哈希表找到 key 对应的节点
        // 2. 将节点移到链表头部
        // 3. 返回节点值
        DLinkedListNode *node = key_node_map_[key];
        move_to_head_(node);
        return node->value;
    }

    void put(int key, int value) {
        if (key_exists_(key)) {
            // key 存在的情况
            DLinkedListNode *node = key_node_map_[key];
            node->value = value;
            move_to_head_(node);
        } else {
            // key 不存在的情况:
            // 1. 如果缓存空间满了，先删除尾节点，再新建节点
            // 2. 否则直接新建节点
            if (is_full_()) {
                DLinkedListNode *tail = dummy_tail_->prev;
                remove_node_(tail);
                key_node_map_.erase(tail->key);
            }
            DLinkedListNode *new_node = new DLinkedListNode(key, value);
            key_node_map_[key] = new_node;
            move_to_head_(new_node);
        }
    }
};
```

989. 数组形式的整数加法

```
DLinkedListNode *new_node = new DLinkedListNode(key);
add_to_head_(new_node);
key_node_map_[key] = new_node;
}
}

private:
unordered_map<int, DLinkedListNode*> key_node_map_;
DLinkedListNode *dummy_head_;
DLinkedListNode *dummy_tail_;
int capacity_;

void move_to_head_(DLinkedListNode *node) {
    remove_node_(node);
    add_to_head_(node);
};

void add_to_head_(DLinkedListNode *node) {
    DLinkedListNode *prev_head = dummy_head_->next;

    dummy_head_->next = node;
    node->prev = dummy_head_;

    node->next = prev_head;
    prev_head->prev = node;
};

void remove_node_(DLinkedListNode *node) {
    node->prev->next = node->next;
    node->next->prev = node->prev;
    node->prev = node->next = NULL;
};

bool key_exists_(int key) {
    return key_node_map_.count(key) > 0;
};

bool is_full_() {
    return key_node_map_.size() == capacity_;
};

/** 
 * Your LRUCache object will be instantiated and called as
 * LRUCache* obj = new LRUCache(capacity);
 * int param_1 = obj->get(key);
 * obj->put(key,value);
 */

}
```

题目地址(104. 二叉树的最大深度)

<https://leetcode-cn.com/problems/maximum-depth-of-binary-tree>

标签

- DFS
- 树

难度

- 简单

入选理由

- 这是一个难度为 easy 的题目，适合作为第一题。
- 此题适合练习递归。
- 这是一个非常常见的考点，只不过有的时候是作为题目的一部分出现，而不是单独考察而已。

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

思路

树的题目很适合用递归来做。基本上和树的搜索有关的，都可以用递归来做，为什么？

因为树是一种递归的数据结构。而穷举搜索一棵树必然需要遍历其所有节点，而搜索的逻辑对所有的子树都是一样的。因此就很适合用递归来解决了。

这里给大家介绍一种写递归的小方法 **产品经理法**。

1. 定义函数功能，不用管其具体实现。

从高层次的角度来定义函数功能。你可以把自己想象成**产品经理**。只需要知道要做什么事情就行了，而怎么实现我不管，那是码农的事情。

具体来说，我需要的功能是**给定一个二叉树的节点，返回以这个节点为根节点的子树的最大深度**。假设这个函数为 f。那么问题转化为 f(root)。

1. 确定大问题和小问题的关系。

要解决 f(root) 这个问题。可以先解决 f(root.right) 和 f(root.left)，当然我们仍然不关心 f 怎么实现。

f(root) 与 f(root.right) 和 f(root.left) 有什么关系呢？不难看出 `1 + max(f(root.right), f(root.left))`。

到这里我们还不知道 f 怎么实现的，但是我们已经完成了产品经理的需求。

实际上我们知道了，我们怎么知道的？

1. 补充递归终止条件。

如果递归到叶子节点的时候，返回 0 即可。

代码（Python）

```
# Definition for a binary tree node.
class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        if not root: return 0
        return 1 + max(self.maxDepth(root.left), self.maxDepth(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：\$O(h)\$，其中 \$h\$ 为树的深度，最坏的情况 \$h\$ 等于 \$N\$，其中 N 为节点数，此时树退化到链表。

题目地址 (100. 相同的树)

<https://leetcode-cn.com/problems/same-tree/>

标签

- DFS
- BFS
- 树

难度

- 简单

入选理由

1. 树的题目一个中心是什么？
2. 难度仍然是 easy，今天继续给大家平和一下
3. 由于树的结构的递归性，树的题目用来练习问题分解很有用，本题也是一样

题目描述

给定两个二叉树，编写一个函数来检验它们是否相同。

如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

示例 1：

输入：
 1 1
 / \ / \
 2 3 2 3

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

输出： true

示例 2：

输入：
 1 1
 / \
 2 2

[1,2], [1,null,2]

输出： false

示例 3：

输入：
 1 1
 / \ / \
 2 1 1 2

[1,2,1], [1,1,2]

输出： false

前置知识

- 递归
- 层序遍历
- 前中序确定一棵树

递归

思路

最简单的想法是递归，这里先介绍下递归三要素

- 递归出口，问题最简单的情况

989. 数组形式的整数加法

- 递归调用总是去尝试解决更小的问题，这样问题才会被收敛到最简单的情况
- 递归调用的父问题和子问题没有交集

尝试用递归去解决相同的树

1. 分解为子问题，相同的树分解为左子是否相同，右子是否相同
2. 递归出口：当树高度为 1 时，判断递归出口

代码

语言支持：JS, Python3

JS Code：

```
var isSameTree = function (p, q) {
    if (!p || !q) {
        return !p && !q;
    }
    return (
        p.val === q.val &&
        isSameTree(p.left, q.left) &&
        isSameTree(p.right, q.right)
    );
};
```

Python3 Code：

```
class Solution:
    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:
        if not p and not q:
            return True
        if not p or not q:
            return False
        return p.val == q.val and self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)
```

复杂度分析

- 时间复杂度： $O(N)$ ，其中 N 为树的节点数。
- 空间复杂度： $O(h)$ ，其中 h 为树的高度。

层序遍历

思路

989. 数组形式的整数加法

判断两棵树是否相同，只需要判断树的整个结构相同，判断树的结构是否相同，只需要判断树的每层内容是否相同。

代码

语言支持：JS, Python3

JS Code:

989. 数组形式的整数加法

```
var isSameTree = function (p, q) {
    let curLevelA = [p];
    let curLevelB = [q];

    while (curLevelA.length && curLevelB.length) {
        let nextLevelA = [];
        let nextLevelB = [];
        const isOK = isSameCurLevel(curLevelA, curLevelB, nextLevelA, nextLevelB);
        if (!isOK) {
            curLevelA = nextLevelA;
            curLevelB = nextLevelB;
        } else {
            return true;
        }
    }

    return false;
};

function isSameCurLevel(curLevelA, curLevelB, nextLevelA, nextLevelB) {
    if (curLevelA.length !== curLevelB.length) {
        return false;
    }
    for (let i = 0; i < curLevelA.length; i++) {
        if (!isSameNode(curLevelA[i], curLevelB[i])) {
            return false;
        }
        curLevelA[i] && nextLevelA.push(curLevelA[i].left, curLevelA[i].right);
        curLevelB[i] && nextLevelB.push(curLevelB[i].left, curLevelB[i].right);
    }
    return true;
}

function isSameNode(nodeA, nodeB) {
    if (!nodeA || !nodeB) {
        return nodeA === nodeB;
    }
    return nodeA.val === nodeB.val;
    // return nodeA === nodeB || (nodeA && nodeB && nodeA.val === nodeB.val);
}
```

Python3 Code:

```

class Solution:
    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:
        if not q and not p:
            return True
        if not q or not q:
            return False
        q1 = collections.deque([p])
        q2 = collections.deque([q])
        while q1 and q2:
            node1 = q1.popleft()
            node2 = q2.popleft()
            if node1.val != node2.val:
                return False
            left1, right1 = node1.left, node1.right
            left2, right2 = node2.left, node2.right
            if (not left1) ^ (not left2):
                return False
            if (not right1) ^ (not right2):
                return False
            if left1:
                q1.append(left1)
            if right1:
                q1.append(right1)
            if left2:
                q2.append(left2)
            if right2:
                q2.append(right2)
        return not q1 and not q2

```

复杂度分析

- 时间复杂度: $O(N)$, 其中 N 为树的节点数。
- 空间复杂度: $O(Q)$, 其中 Q 为队列的长度最大值, 在这里不会超过相邻两层的节点数的最大值。

前中序确定一棵树

思路

前序和中序的遍历结果确定一棵树, 那么当两棵树前序遍历和中序遍历结果都相同, 那是否说明两棵树也相同。

代码

语言支持: JS, Java

989. 数组形式的整数加法

JS Code:

```
var isSameTree = function (p, q) {
    const preorderP = preorder(p, []);
    const preorderQ = preorder(q, []);
    const inorderP = inorder(p, []);
    const inorderQ = inorder(q, []);
    return (
        preorderP.join("") === preorderQ.join("") &&
        inorderP.join("") === inorderQ.join("")
    );
};

function preorder(root, arr) {
    if (root === null) {
        arr.push(" ");
        return arr;
    }
    arr.push(root.val);
    preorder(root.left, arr);
    preorder(root.right, arr);
    return arr;
}

function inorder(root, arr) {
    if (root === null) {
        arr.push(" ");
        return arr;
    }
    inorder(root.left, arr);
    arr.push(root.val);
    inorder(root.right, arr);
    return arr;
}
```

Java Code:

989. 数组形式的整数加法

```
/*
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */

class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        // preorder
        List<Integer> pretraversalP = new ArrayList<>();
        List<Integer> pretraversalQ = new ArrayList<>();
        preorder(p, pretraversalP);
        preorder(q, pretraversalQ);

        // inorder
        List<Integer> intraversalP = new ArrayList<>();
        List<Integer> intraversalQ = new ArrayList<>();
        inorder(p, intraversalP);
        inorder(q, intraversalQ);

        return (pretraversalP+"").equals((pretraversalQ+""));
    }

    private void preorder(TreeNode root, List<Integer> traversal) {
        if (root == null) {
            traversal.add(null);
            return;
        }
        traversal.add(root.val);
        preorder(root.left, traversal);
        preorder(root.right, traversal);
    }

    private void inorder(TreeNode root, List<Integer> traversal) {
        if (root == null) {
            traversal.add(null);
        }
    }
}
```

989. 数组形式的整数加法

```
        return;
    }
    preorder(root.left, traversal);
    traversal.add(root.val);
    preorder(root.right, traversal);
}
}
```

复杂度分析

- 时间复杂度: $O(N)$, 其中 N 为树的节点数。
- 空间复杂度: 使用了中序遍历的结果数组, 因此空间复杂度为 $O(N)$, 其中 N 为树的节点数。

题目地址 (129. 求根到叶子节点数字之和)

<https://leetcode-cn.com/problems/sum-root-to-leaf-numbers/>

入选理由

1. 路径是一种经典的树的题目，掌握路径是树必须的技能之一

标签

- 树
- BFS
- DFS

难度

- 中等

题目描述

989. 数组形式的整数加法

给定一个二叉树，它的每个结点都存放一个 0-9 的数字，每条从根到叶子节点的路径代表一个整数。

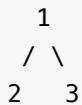
例如，从根到叶子节点路径 1->2->3 代表数字 123。

计算从根到叶子节点生成的所有数字之和。

说明：叶子节点是指没有子节点的节点。

示例 1：

输入： [1,2,3]



输出： 25

解释：

从根到叶子节点路径 1->2 代表数字 12.

从根到叶子节点路径 1->3 代表数字 13.

因此，数字总和 = 12 + 13 = 25.

示例 2：

输入： [4,9,0,5,1]



输出： 1026

解释：

从根到叶子节点路径 4->9->5 代表数字 495.

从根到叶子节点路径 4->9->1 代表数字 491.

从根到叶子节点路径 4->0 代表数字 40.

因此，数字总和 = 495 + 491 + 40 = 1026.

前置知识

- DFS
- BFS
- 前序遍历

DFS

思路

989. 数组形式的整数加法

求从根到叶子的路径之和，那我们只需要把每条根到叶子的路径找出来，并求和即可，这里用 DFS 去解，DFS 也是最容易想到的。

代码

代码支持: JS, Java, C++, Python

C++ Code:

```
class Solution {
public:
    int sum = 0;
    int sumNumbers(TreeNode* root) {
        dfs(root, 0);
        return sum;
    }

    void dfs(TreeNode* root, int num) {
        if (!root) return;
        if (!root->left && !root->right) {
            sum += num * 10 + root->val;
            return;
        }
        dfs(root->left, num * 10 + root->val);
        dfs(root->right, num * 10 + root->val);
    }
};
```

Java Code:

989. 数组形式的整数加法

```
class Solution {
    public int ans;

    public int sumNumbers(TreeNode root) {
        dfs(root, 0);
        return ans;
    }

    public void dfs(TreeNode root, int last){
        if(root == null) return;
        if(root.left == null && root.right == null) {
            ans += last * 10 + root.val;
            return;
        }
        dfs(root.left, last * 10 + root.val);
        dfs(root.right, last * 10 + root.val);
    }
}
```

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 sumNumbers(self, root: TreeNode) -> int:
        def dfs(root, cur):
            if not root: return 0
            if not root.left and not root.right: return cur
            return dfs(root.left, cur * 10 + root.val) + dfs(root.right, cur * 10 + root.val)
        return dfs(root, 0)
```

JS Code:

```

function sumNumbers1(root) {
    let sum = 0;
    function dfs(root, cur) {
        if (!root) {
            return;
        }
        let curSum = cur * 10 + root.val;
        if (!root.left && !root.right) {
            sum += curSum;
            return;
        }
        dfs(root.left, curSum);
        dfs(root.right, curSum);
    }
    dfs(root, 0);
    return sum;
}

```

复杂度分析

令 n 为节点总数， h 为树的高度。

- 时间复杂度: $O(n)$
- 空间复杂度: $O(h)$

BFS

思路

如果说 DFS 是孤军独入，取敌将首级，那么 BFS 就是堂堂正正，车马摆开，层层推进。BFS 可能没那么优雅，但是掌握模板之后简直就是神器。

要求根到的叶子的路径的和，那我们把中间每一层对应的值都求出来，当前层的节点是叶子节点，把对应值相加即可。

代码

```

function sumNumbers(root) {
    let sum = 0;
    let curLevel = [];
    if (root) {
        curLevel.push(root);
    }
    while (curLevel.length) {
        let nextLevel = [];
        for (let i = 0; i < curLevel.length; i++) {
            let cur = curLevel[i];
            if (cur.left) {
                cur.left.val = cur.val * 10 + cur.left.val;
                nextLevel.push(cur.left);
            }
            if (cur.right) {
                cur.right.val = cur.val * 10 + cur.right.val;
                nextLevel.push(cur.right);
            }
            if (!cur.left && !cur.right) {
                sum += cur.val;
            }
            curLevel = nextLevel;
        }
    }
    return sum;
}

```

复杂度分析

令 n 为节点总数， q 为队列长度。

- 时间复杂度: $O(n)$
- 空间复杂度: $O(q)$ 。最坏的情况是满二叉树，此时和 n 同阶。

为什么空间复杂度和 n 同阶呢？这是因为叶子节点的数目在极端情况下是完全二叉树，此时的叶子节点的数目差不多和 $n/2$ 相等。具体推导过程如下。

令 h 为树高度。

$$k = h - 1$$

$$n = \sum_{i=0}^k 2^i \quad (1)$$

$$\frac{n}{2} = \sum_{i=-1}^{k-1} 2^i \quad (2)$$

$$\frac{n}{2} = 2^k - \frac{1}{2} \quad (1) - (2)$$

$$T(n) = 2^k = n/2 + \frac{1}{2} = O(n)$$

题目地址（513. 找树左下角的值）

<https://leetcode-cn.com/problems/find-bottom-left-tree-value/>

入选理由

1. 和昨天一样，这是一道典型的树的搜索题，大家用两种搜索方式感受一下

标签

- 树
- BFS
- DFS

难度

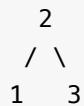
- 中等

题目描述

给定一个二叉树，在树的最后一行找到最左边的值。

示例 1：

输入：

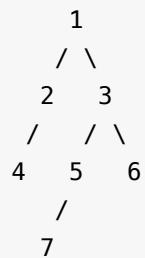


输出：

1

示例 2：

输入：



输出：

7

BFS

思路

其实问题本身就告诉你怎么做了

在树的最后一行找到最左边的值。

问题再分解一下

- 找到树的最后一行
- 找到那一行的第一个节点

不用层序遍历简直对不起这个问题，这里贴一下层序遍历的流程

```

令curLevel为第一层节点也就是root节点
定义nextLevel为下层节点
遍历node in curLevel,
    nextLevel.push(node.left)
    nextLevel.push(node.right)
令curLevel = nextLevel, 重复以上流程直到curLevel为空

```

代码

代码支持: JS, Python, Java, CPP

JS Code:

```

var findBottomLeftValue = function (root) {
    let curLevel = [root];
    let res = root.val;
    while (curLevel.length) {
        let nextLevel = [];
        for (let i = 0; i < curLevel.length; i++) {
            curLevel[i].left && nextLevel.push(curLevel[i].left);
            curLevel[i].right && nextLevel.push(curLevel[i].right);
        }
        res = curLevel[0].val;
        curLevel = nextLevel;
    }
    return res;
};

```

Python Code:

```

class Solution(object):
    def findBottomLeftValue(self, root):
        queue = collections.deque()
        queue.append(root)
        while queue:
            length = len(queue)
            res = queue[0].val
            for _ in range(length):
                cur = queue.popleft()
                if cur.left:
                    queue.append(cur.left)
                if cur.right:
                    queue.append(cur.right)
        return res

```

Java:

989. 数组形式的整数加法

```
class Solution {
    Map<Integer, Integer> map = new HashMap<>();
    int maxLevel = 0;
    public int findBottomLeftValue(TreeNode root) {
        if (root == null) return 0;
        LinkedList<TreeNode> deque = new LinkedList<>();
        deque.add(root);
        int res = 0;
        while(!deque.isEmpty()) {
            int size = deque.size();
            for (int i = 0; i < size; i++) {
                TreeNode node = deque.pollFirst();
                if (i == 0) {
                    res = node.val;
                }
                if (node.left != null) deque.addLast(node.left);
                if (node.right != null) deque.addLast(node.right);
            }
        }
        return res;
    }
}
```

CPP:

```
class Solution {
public:
    int findBottomLeftValue_bfs(TreeNode* root) {
        queue<TreeNode*> q;
        TreeNode* ans = NULL;
        q.push(root);
        while (!q.empty()) {
            ans = q.front();
            int size = q.size();
            while (size--) {
                TreeNode* cur = q.front();
                q.pop();
                if (cur->left)
                    q.push(cur->left);
                if (cur->right)
                    q.push(cur->right);
            }
        }
        return ans->val;
    }
}
```

复杂度分析

- 时间复杂度: $O(N)$, 其中 N 为树的节点数。
- 空间复杂度: $O(Q)$, 其中 Q 为队列长度, 最坏的情况是满二叉树, 此时和 N 同阶, 其中 N 为树的节点总数

DFS

思路

树的最后一行找到最左边的值, 转化一下就是找第一个出现的深度最大的节点, 这里用先序遍历去做, 其实中序遍历也可以, 只需要保证左节点在右节点前被处理即可。具体算法为, 先序遍历 root, 维护一个最大深度的变量, 记录每个节点的深度, 如果当前节点深度比最大深度要大, 则更新最大深度和结果项。

代码

代码支持: JS, Python, Java, CPP

JS Code:

```
function findBottomLeftValue(root) {
    let maxDepth = 0;
    let res = root.val;

    dfs(root.left, 0);
    dfs(root.right, 0);

    return res;

    function dfs(cur, depth) {
        if (!cur) {
            return;
        }
        const curDepth = depth + 1;
        if (curDepth > maxDepth) {
            maxDepth = curDepth;
            res = cur.val;
        }
        dfs(cur.left, curDepth);
        dfs(cur.right, curDepth);
    }
}
```

Python Code:

989. 数组形式的整数加法

```
class Solution(object):

    def __init__(self):
        self.res = 0
        self.max_level = 0

    def findBottomLeftValue(self, root):
        self.res = root.val
        def dfs(root, level):
            if not root:
                return
            if level > self.max_level:
                self.res = root.val
                self.max_level = level
            dfs(root.left, level + 1)
            dfs(root.right, level + 1)
        dfs(root, 0)

    return self.res
```

Java Code:

```
class Solution {
    int max = 0;
    Map<Integer, Integer> map = new HashMap<>();
    public int findBottomLeftValue(TreeNode root) {
        if (root == null) return 0;
        dfs(root, 0);
        return map.get(max);
    }

    void dfs (TreeNode node,int level){
        if (node == null){
            return;
        }
        int curLevel = level+1;
        dfs(node.left,curLevel);
        if (curLevel > max && !map.containsKey(curLevel)){
            map.put(curLevel,node.val);
            max = curLevel;
        }
        dfs(node.right,curLevel);
    }
}
```

989. 数组形式的整数加法

CPP:

```
class Solution {
public:
    int res;
    int max_depth = 0;
    void findBottomLeftValue_core(TreeNode* root, int depth) {
        if (root->left || root->right) {
            if (root->left)
                findBottomLeftValue_core(root->left, depth + 1);
            if (root->right)
                findBottomLeftValue_core(root->right, depth + 1);
        } else {
            if (depth > max_depth) {
                res = root->val;
                max_depth = depth;
            }
        }
    }
    int findBottomLeftValue(TreeNode* root) {
        findBottomLeftValue_core(root, 1);
        return res;
    }
};
```

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。

题目地址 (297. 二叉树的序列化与反序列化)

<https://leetcode-cn.com/problems/serialize-and-deserialize-binary-tree/>

入选理由

- 难度不小的一个题，但是我的树专题其实已经分析了这道题，同样可以使用两种方式来解决，大家来试试吧

标签

- 树
- BFS
- DFS

难度

- 困难

题目描述

序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换，

请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序

示例：

你可以将以下二叉树：

```
    1
   / \
  2   3
  / \
 4   5
```

序列化为 "[1,2,3,null,null,4,5]"

提示：这与 LeetCode 目前使用的方式一致，详情请参阅 LeetCode 序列化

说明：不要使用类的成员 / 全局 / 静态变量来存储状态，你的序列化和反序

思路(BFS)

如果我将一个二叉树的完全二叉树形式序列化，然后通过 BFS 反序列化，这不就是力扣官方序列化树的方式么？比如：



序列化为 "[1,2,3,null,null,4,5]"。这不就是我刚刚画的完全二叉树么？就是将一个普通的二叉树硬生生当成完全二叉树用了。

其实这并不是序列化成了完全二叉树，下面会纠正。

将一颗普通树序列化为完全二叉树很简单，只要将空节点当成普通节点入队处理即可。代码：

```

class Codec:

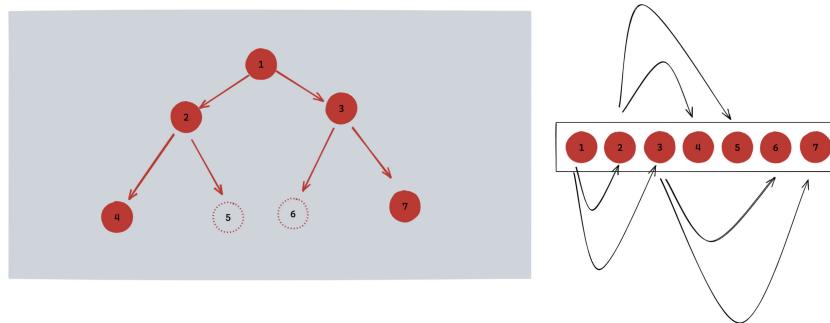
    def serialize(self, root):
        q = collections.deque([root])
        ans = ''
        while q:
            cur = q.popleft()
            if cur:
                ans += str(cur.val) + ','
                q.append(cur.left)
                q.append(cur.right)
            else:
                # 除了这里不一样，其他和普通的不记录层的 BFS 没区别。
                ans += 'null,'

        # 末尾会多一个逗号，我们去掉它。
        return ans[:-1]
  
```

细心的同学可能会发现，我上面的代码其实并不是将树序列化成了完全二叉树，这个我们稍后就会讲到。另外后面多余的空节点也一并序列化了。这其实是可以优化的，优化的方式也很简单，那就是去除末尾的 null 即可。

你只要彻底理解我刚才讲的 我们可以给完全二叉树编号，这样父子之间就可以通过编号轻松求出。比如我给所有节点从左到右从上到下依次从 1 开始编号。那么已知一个节点的编号是 i ，那么其左子节点就是 $2 * i$ ，右子节点就是 $2 * i + 1$ ，父节点就是 $i / 2$ 。这句话，那么反序列化对你就不是难事。

如果我用一个箭头表示节点的父子关系，箭头指向节点的两个子节点，那么大概是这样的：



我们刚才提到了：

- 1号节点的两个子节点的2号和3号。
- 2号节点的两个子节点的4号和5号。
- . . .
- i号节点的两个子节点的`2 * i`号和`2 * i + 1`号。

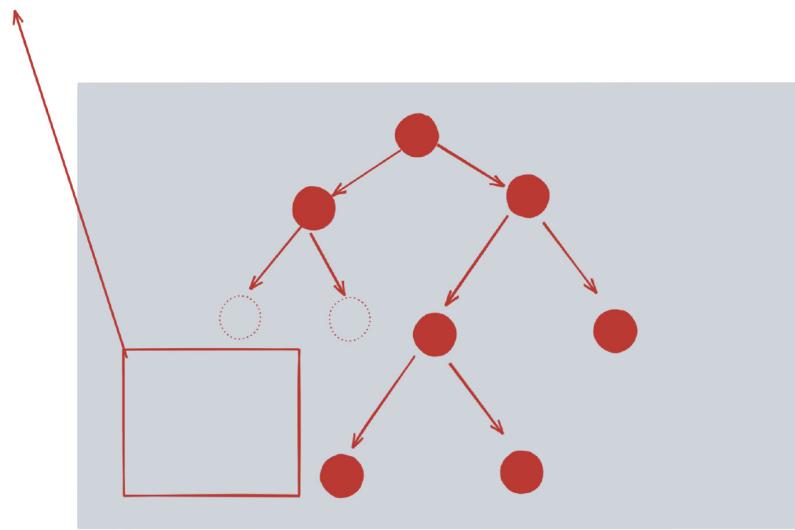
此时你可能会写出类似这样的代码：

```
def deserialize(self, data):
    if data == 'null': return None
    nodes = data.split(',')
    root = TreeNode(nodes[0])
    # 从一号开始编号，编号信息一起入队
    q = collections.deque([(root, 1)])
    while q:
        cur, i = q.popleft()
        # 2 * i 是左节点，而 2 * i 编号对应的其实是索引为 2 :
        if 2 * i - 1 < len(nodes): lv = nodes[2 * i - 1]
        if 2 * i < len(nodes): rv = nodes[2 * i]
        if lv != 'null':
            l = TreeNode(lv)
            # 将左节点和它的编号 2 * i 入队
            q.append((l, 2 * i))
            cur.left = l
        if rv != 'null':
            r = TreeNode(rv)
            # 将右节点和它的编号 2 * i + 1 入队
            q.append((r, 2 * i + 1))
            cur.right = r

    return root
```

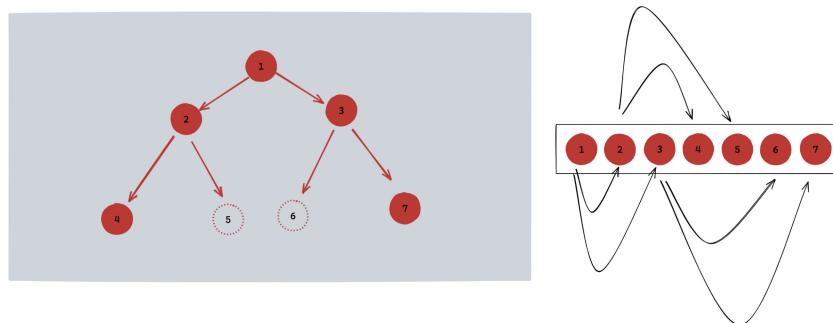
但是上面的代码是不对的，因为我们序列化的时候其实不是完全二叉树，这也是上面我埋下的伏笔。因此遇到类似这样的 case 就会挂：

这一块没有序列化



这也是我前面说“上面代码的序列化并不是一颗完全二叉树”的原因。

其实这个很好解决，核心还是上面我画的那种图：



其实我们可以：

- 用三个指针分别指向数组第一项，第二项和第三项（如果存在的），这里用 $p1$, $p2$, $p3$ 来标记，分别表示当前处理的节点，当前处理的节点的左子节点和当前处理的节点的右子节点。
- $p1$ 每次移动一位， $p2$ 和 $p3$ 每次移动两位。
- $p1.left = p2$; $p1.right = p3$ 。
- 持续上面的步骤直到 $p1$ 移动到最后。

因此代码就不难写出了。反序列化代码如下：

```
def deserialize(self, data):
    if data == 'null': return None
    nodes = data.split(',')
    root = TreeNode(nodes[0])
    q = collections.deque([root])
    i = 0
    while q and i < len(nodes) - 2:
        cur = q.popleft()
        lv = nodes[i + 1]
        rv = nodes[i + 2]
        i += 2
        if lv != 'null':
            l = TreeNode(lv)
            q.append(l)
            cur.left = l
        if rv != 'null':
            r = TreeNode(rv)
            q.append(r)
            cur.right = r

    return root
```

这个题目虽然并不是完全二叉树的题目，但是却和完全二叉树很像，有借鉴完全二叉树的地方。

代码

代码支持：JS, Python

JS Code:

989. 数组形式的整数加法

```
const serialize = (root) => {
    const queue = [root];
    let res = [];
    while (queue.length) {
        const node = queue.shift();
        if (node) {
            res.push(node.val);
            queue.push(node.left);
            queue.push(node.right);
        } else {
            res.push("#");
        }
    }
    return res.join(",");
};

const deserialize = (data) => {
    if (data == "#") return null;

    const list = data.split(",");
    const root = new TreeNode(list[0]);
    const queue = [root];
    let cursor = 1;

    while (cursor < list.length) {
        const node = queue.shift();

        const leftVal = list[cursor];
        const rightVal = list[cursor + 1];

        if (leftVal != "#") {
            const leftNode = new TreeNode(leftVal);
            node.left = leftNode;
            queue.push(leftNode);
        }
        if (rightVal != "#") {
            const rightNode = new TreeNode(rightVal);
            node.right = rightNode;
            queue.push(rightNode);
        }
        cursor += 2;
    }
    return root;
};
```

Python Code:

989. 数组形式的整数加法

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

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 += '#,'

        print(ans[:-1])
        return ans[:-1]

    def deserialize(self, data: str):
        if data == '#': return None
        nodes = data.split(',')
        if not nodes: return None
        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
            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: $O(N)$, 其中 N 为树的节点数。
- 空间复杂度: $O(Q)$, 其中 Q 为队列长度, 最坏的情况是满二叉树, 此时和 N 同阶, 其中 N 为树的节点总数

DFS 其实思路也是类似的, 比如我使用前序遍历, 那么代码就是这样的:

Python Code:

```
class Codec:
    def serialize(self, root):
        def preorder(root):
            if not root:
                return "null,"
            return str(root.val) + "," + preorder(root.left)
            return preorder(root)[: :-1]

        def deserialize(self, data: str):
            nodes = data.split(",")

            def preorder(i):
                if i >= len(nodes) or nodes[i] == "null":
                    return i, None
                root = TreeNode(nodes[i])
                j, root.left = preorder(i + 1)
                k, root.right = preorder(j + 1)
                return k, root

            return preorder(0)[1]
```

复杂度分析

- 时间复杂度: $O(N)$, 其中 N 为树的节点数。
- 空间复杂度: $O(h)$, 其中 h 为树的高度, 最坏的情况是链表, 此时和 N 同阶, 其中 N 为树的节点总数

题目地址（987. 二叉树的垂序遍历）

<https://leetcode-cn.com/problems/vertical-order-traversal-of-a-binary-tree>

标签

- 哈希表
- 树
- 排序

难度

- 中等

入选理由

1. 很有意思的一个题目，一个很特殊的遍历方式。但不管怎么奇葩的遍历，我们都可以用基础的遍历方式来解决

题目描述

给定二叉树，按垂序遍历返回其结点值。

对位于 (X, Y) 的每个结点而言，其左右子结点分别位于 $(X-1, Y-1)$ 和 $(X+1, Y-1)$ 。

把一条垂线从 $X = -\infty$ 移动到 $X = +\infty$ ，每当该垂线与结点相交时报告结点值。

如果两个结点位置相同，则首先报告的结点值较小。

按 X 坐标顺序返回非空报告的列表。每个报告都有一个结点值列表。

示例 1：

输入：[3,9,20,null,null,15,7]

输出：[[9],[3,15],[20],[7]]

解释：

在不丧失其普遍性的情况下，我们可以假设根结点位于 $(0, 0)$ ：

然后，值为 9 的结点出现在 $(-1, -1)$ ；

值为 3 和 15 的两个结点分别出现在 $(0, 0)$ 和 $(0, -2)$ ；

值为 20 的结点出现在 $(1, -1)$ ；

值为 7 的结点出现在 $(2, -2)$ 。

示例 2：

输入：[1,2,3,4,5,6,7]

输出：[[4],[2],[1,5,6],[3],[7]]

解释：

根据给定的方案，值为 5 和 6 的两个结点出现在同一位置。

然而，在报告 “[1,5,6]” 中，结点值 5 排在前面，因为 5 小于 6。

提示：

树的结点数介于 1 和 1000 之间。

每个结点值介于 0 和 1000 之间。

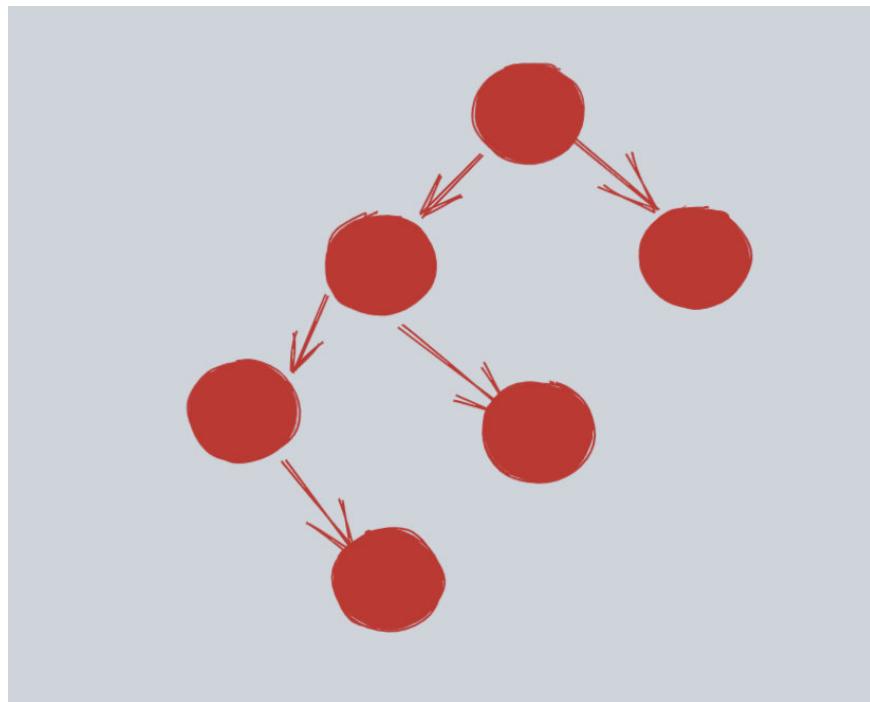
前置知识

- DFS
- 排序

思路

经过前面几天的学习，希望大家对 DFS 和 BFS 已经有了一定的了解了。

我们先来简化一下问题。假如题目没有 从上到下的顺序报告结点的值（Y 坐标递减），甚至也没有 如果两个结点位置相同，则首先报告的结点值较小。的限制。是不是就比较简单了？



如上图，我们只需要进行一次搜索，不妨使用 DFS（没有特殊理由，我一般都是 DFS），将节点存储到一个哈希表中，其中 key 为节点的 x 值，value 为横坐标为 x 的节点值列表（不妨用数组表示）。形如：

```
{
  1: [1, 3, 4]
  -1: [5]
}
```

数据是瞎编的，不和题目例子有关联

经过上面的处理，这个时候只需要对哈希表中的数据进行一次排序输出即可。

ok，如果这个你懂了，我们尝试加上面的两个限制加上去。

1. 从上到下的顺序报告结点的值（Y 坐标递减）
2. 如果两个结点位置相同，则首先报告的结点值较小。

关于第一个限制。其实我们可以再哈希表中再额外增加一层来解决。形如：

```
{
  1: {
    -2, [1, 3, 4]
    -3, [5]

  },
  -1: {
    -3: [6]
  }
}
```

这样我们除了对 x 排序，再对里层的 y 排序即可。

再来看第二个限制。其实看到上面的哈希表结构就比较清晰了，我们再对值排序即可。

总的来说，我们需要进行三次排序，分别是对 x 坐标，y 坐标 和 值。

那么时间复杂度是多少呢？我们来分析一下：

- 哈希表最外层的 key 总个数是最大是树的宽度。
- 哈希表第二层的 key 总个数是树的高度。
- 哈希表值的总长度是树的节点数。

也就是说哈希表的总容量和树的总的节点数是同阶的。因此空间复杂度为 $O(N)$ ，排序的复杂度大致为 $N \log N$ ，其中 N 为树的节点总数。

代码

代码支持：Python, JS, CPP

Python Code:

989. 数组形式的整数加法

```
class Solution(object):
    def verticalTraversal(self, root):
        seen = collections.defaultdict(
            lambda: collections.defaultdict(list))

        def dfs(root, x=0, y=0):
            if not root:
                return
            seen[x][y].append(root.val)
            dfs(root.left, x-1, y+1)
            dfs(root.right, x+1, y+1)

        dfs(root)
        ans = []
        # x 排序、
        for x in sorted(seen):
            level = []
            # y 排序
            for y in sorted(seen[x]):
                # 值排序
                level += sorted(v for v in seen[x][y])
            ans.append(level)

        return ans
```

JS Code(by @suukii):

989. 数组形式的整数加法

```
/*
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[][]}
 */
var verticalTraversal = function (root) {
    if (!root) return [];

    // 坐标集合以 x 坐标分组
    const pos = {};
    // dfs 遍历节点并记录每个节点的坐标
    dfs(root, 0, 0);

    // 得到所有节点坐标后，先按 x 坐标升序排序
    let sorted = Object.keys(pos)
        .sort((a, b) => +a - +b)
        .map((key) => pos[key]);

    // 再给 x 坐标相同的每组节点坐标分别排序
    sorted = sorted.map((g) => {
        g.sort((a, b) => {
            // y 坐标相同的，按节点值升序排
            if (a[0] === b[0]) return a[1] - b[1];
            // 否则，按 y 坐标降序排
            else return b[0] - a[0];
        });
        // 把 y 坐标去掉，返回节点值
        return g.map((el) => el[1]);
    });

    return sorted;
}

// *****
function dfs(root, x, y) {
    if (!root) return;

    x in pos || (pos[x] = []);
    // 保存坐标数据，格式是：[y, val]
    pos[x].push([y, root.val]);

    dfs(root.left, x - 1, y - 1);
    dfs(root.right, x + 1, y - 1);
}
```

989. 数组形式的整数加法

```
    }  
};
```

CPP(by @Francis-xsc):

```
class Solution {  
public:  
    struct node  
    {  
        int val;  
        int x;  
        int y;  
        node(int v,int X,int Y):val(v),x(X),y(Y){};  
    };  
    static bool cmp(node a,node b)  
    {  
        if(a.x>b.x)  
            return a.x<b.x;  
        if(a.y>b.y)  
            return a.y<b.y;  
        return a.val<b.val;  
    }  
    vector<node> a;  
    int minx=1000,maxx=-1000;  
    vector<vector<int>> verticalTraversal(TreeNode* root) ->  
    {  
        dfs(root,0,0);  
        sort(a.begin(),a.end(),cmp);  
        vector<vector<int>>ans(maxx-minx+1);  
        for(auto xx:a)  
        {  
            ans[xx.x-minx].push_back(xx.val);  
        }  
        return ans;  
    }  
    void dfs(TreeNode* root,int x,int y)  
    {  
        if(root==nullptr)  
            return;  
        if(x<minx)  
            minx=x;  
        if(x>maxx)  
            maxx=x;  
        a.push_back(node(root->val,x,y));  
        dfs(root->left,x-1,y+1);  
        dfs(root->right,x+1,y+1);  
    }  
};
```

复杂度分析

- 时间复杂度: $O(N \log N)$, 其中 N 为树的节点总数。
- 空间复杂度: $O(N)$, 其中 N 为树的节点总数。

题目地址(两数之和)

<https://leetcode-cn.com/problems/two-sum>

入选理由

1. 两数之和的经典程度不用我多说了，大家都应该知道。
2. 这道题不仅是入门题目，而且和后面我们要讲的双指针有联系。

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值的两个数。
你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍。

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]`

来源：力扣（LeetCode）
链接：<https://leetcode-cn.com/problems/two-sum>
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

标签

- 哈希表
- 双指针

难度

- 简单

前置知识

- 哈希表

思路 - 暴力

思路很简单，遍历数据，对每一个出现的 `num` 判断其另一半 `target - num` 是否也出现在数组中即可

989. 数组形式的整数加法

代码

代码支持: Java

```
```java {7} public class Solution {

 public int[] twoSum(int[] nums, int target) {

 for(int i = 0; i < nums.length; i++)
 for(int j = i + 1; j < nums.length; j++)
 if(nums[i] + nums[j] == target)
 return new int[]{i, j};

 return new int[]{-1, -1};
 }

}
```

#### #### 复杂度分析

- 空间复杂度:  $O(1)$
- 时间复杂度:  $O(n^2)$ ,  $n$ 为数组长度

#### ### 思路

上面是用于搜索整个数组的方式来判断 `target - num` 是否也存在 `nums`

#### #### 代码

\*\*哈希表是非常常用的时间换空间的方式\*\*

代码支持: Java

```
``` java {9-10,12}
class Solution {

    public int[] twoSum(int[] nums, int target) {

        Map<Integer, Integer> map = new HashMap<>();

        for (int i = 0; i < nums.length; i++) {

            if (map.containsKey(nums[i]))
                return new int[]{map.get(nums[i]), i};

            map.put(target - nums[i], i);
        }

        return new int[]{};
    }
}
```

复杂度分析

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n^2)$

题目地址(347. 前 K 个高频元素)

<https://leetcode-cn.com/problems/top-k-frequent-elements/>

入选理由

- 统计频率是哈希表的一个应用。当然如果数据范围小，可以考虑使用数组，理论性能更好（复杂度不变）
- 推荐大家和今天的力扣每日一题结合练习。<https://leetcode-cn.com/problems/degree-of-an-array/> 那道题考察了一个哈希表记录最近和最远位置的点，这个考点也很常见。

题目描述

给定一个非空的整数数组，返回其中出现频率前 k 高的元素。

示例 1：

输入: `nums = [1,1,1,2,2,3]`, $k = 2$

输出: `[1,2]`

示例 2：

输入: `nums = [1]`, $k = 1$

输出: `[1]`

提示：

- 你可以假设给定的 k 总是合理的，且 $1 \leq k \leq$ 数组中不相同的元素的个数。
- 你的算法的时间复杂度必须优于 $O(n \log n)$ ， n 是数组的大小。
- 题目数据保证答案唯一，换句话说，数组中前 k 个高频元素的集合是唯一的。
- 你可以按任意顺序返回答案。

来源：力扣（LeetCode）

链接：<https://leetcode-cn.com/problems/top-k-frequent-elements/>
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

标签

- 堆

- 排序
- 哈希表

难度

- 中等

前置知识

- 哈希表
- 堆排序
- 快速选择

思路

直接根据题意，可以把问题化解两个小问题

- 计算每个数的频次
- 在生成频次里取前 K 大的

对频次计算的话，我们可以采用哈希表，key 为列表的数，value 为出现的频次

生成频次里取 K 大的，也就是我们熟悉 TOP K 问题，通过以下方式求取

1. 排序，取前 K 大的
2. 建堆
3. 快速选择

思路 - 哈希表+桶排序

1. 哈希表记录数值频次
2. 从最后一个桶开始遍历直到取出K个数

```
``` java {9-10,19-25} class Solution {
```

## 989. 数组形式的整数加法

```
public int[] topKFrequent(int[] nums, int k) {

 List<Integer> res = new LinkedList<>();
 List<Integer>[] bucket = new List[nums.length + 1];
 Map<Integer, Integer> counter = new HashMap<>();

 for (int num: nums)
 counter.put(num, counter.getOrDefault(num, 0) + 1);

 for (Map.Entry<Integer, Integer> entry: counter.entrySet())
 int val = entry.getValue();
 if (bucket[val] == null)
 bucket[val] = new LinkedList<>();
 bucket[val].add(entry.getKey());
 }

 int kNum = 0;
 for (int i = bucket.length - 1; i >= 0; i--)
 if (bucket[i] != null)
 for (int elem: bucket[i]) {

 res.add(elem);
 kNum++;
 }

 int[] ret = new int[k];
 for (int i = 0; i < ret.length; i++)
 ret[i] = res.get(i);

 return ret;
}

}
```

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

### 思路 - 哈希表+堆排序

1. 建立一个 size 为 K 的小顶堆
2. 对每个频次 C , 与堆顶 T 比较, 如果 C > T, C 替换 T, 并维持小顶堆

```
```c++ {10-19}
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int,int> counts;
        // 计算频次
        for(int i : nums) counts[i]++;
        // 最小堆
        priority_queue<pair<int,int>, vector<pair<int,int>> q;
        // 堆中元素为 [频次, 数值] 元组, 并根据频次维护小顶堆特性
        for(auto it : counts) {
            if (q.size() != k) {
                q.push(make_pair(it.second, it.first));
            } else {
                if (it.second > q.top().first) {
                    q.pop();
                    q.push(make_pair(it.second, it.first));
                }
            }
        }
        vector<int> res;
        while(q.size()) {
            res.push_back(q.top().second);
            q.pop();
        }
        return vector<int>(res.rbegin(), res.rend());
    }
};
```

- 时间复杂度: $O(N * \log K)$, N 为数组长度
- 空间复杂度: $O(N)$, N 为数组长度, 主要为哈希表开销

思路 - 快速选择

快速排序变种, 快速排序的核心是选出一个拆分点, 将数组分为 `left`, `right` 两个 part, 对两个 part 内的元素分治处理, 时间是 $O(n * \log n)$, 但是注意, 我们只是需要找出前 K 个数, 并不需要其有序, 所以通过拆分

989. 数组形式的整数加法

出 K 个数，使得前 K 个数都大于后面 $n - k$ 个数即可。

和快速排序的唯一不同是，快速选择每次不会递归访问 $pivot$ 两侧，而是仅访问一侧。

和快速排序一样，最坏的情况时间复杂度是平方。这和 $pivot$ 的选择有关，因此实际应用中更多的是检测到数组相对无序才会使用该算法。

代码

989. 数组形式的整数加法

```
/*
 * @param {number[]} nums
 * @param {number} k
 * @return {number[]}
 */
var topKFrequent = function (nums, k) {
    const counts = {};
    for (let num of nums) {
        counts[num] = (counts[num] || 0) + 1;
    }
    let pairs = Object.keys(counts).map((key) => [counts[key]

        select(0, pairs.length - 1, k);
        return pairs.slice(0, k).map((item) => item[1]);

    // 快速选择
    function select(left, right, offset) {
        if (left >= right) {
            return;
        }
        const pivotIndex = partition(left, right);
        console.log({ pairs, pivotIndex });
        if (pivotIndex === offset) {
            return;
        }

        if (pivotIndex <= offset) {
            select(pivotIndex + 1, right, offset);
        } else {
            select(left, pivotIndex - 1);
        }
    }

    // 拆分数组为两个part
    function partition(left, right) {
        const [pivot] = pairs[right];
        let cur = left;
        let leftPartIndex = left;
        while (cur < right) {
            if (pairs[cur][0] > pivot) {
                swap(leftPartIndex++, cur);
            }
            cur++;
        }
        swap(right, leftPartIndex);
        return leftPartIndex;
    }
}
```

989. 数组形式的整数加法

```
function swap(x, y) {
    const term = pairs[x];
    pairs[x] = pairs[y];
    pairs[y] = term;
}
};
```

- 时间复杂度: $O(N)$, 最坏能到 $O(N^2)$
- 空间复杂度: $O(N)$

题目地址 (447. 回旋镖的数量)

<https://leetcode-cn.com/problems/number-of-boomerangs/>

入选理由

1. 哈希表的一个重要作用就是空间换时间，当你想出暴力算法，可以考虑是否可用哈希表来优化。哈希表优化时间复杂度算是最最简单的一种优化手段了。相比单调栈，二分等简单很多。

题目描述

给定平面上 n 对不同的点，“回旋镖”是由点表示的元组 (i, j, k) ，其找到所有回旋镖的数量。你可以假设 n 最大为 500，所有点的坐标在闭区间示例：

输入：

$[[0, 0], [1, 0], [2, 0]]$

输出：

2

解释：

两个回旋镖为 $[[1, 0], [0, 0], [2, 0]]$ 和 $[[1, 0], [2, 0], [0, 0]]$

标签

- Math
- 哈希表

难度

- 中等

前置知识

- 哈希表
- 两点间距离计算方法

思路

多读两遍题，大概就明白了题意：就是找出所有符合三个点 x,y,z ，并且 $\text{dis}(x,y)=\text{dis}(x,z)$ 这种点的个数。首先要明确两点间距离怎么计算：

对于点 $x=(x_1,x_2)$, $y=(y_1,y_2)$, 那么 $\text{dis}(x,y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$

由于求的是算数平方根，所以我们算距离的时候也没必要开根号了。我们可以很容易地想到暴力解法，也就是来个三重循环。

```
public int numberOfBoomerangs(int[][] points) {

    if (points == null || points.length <= 2)
        return 0;

    int res = 0;

    for (int i = 0; i < points.length; i++) {
        for (int j = 0; j < points.length; j++) {
            if (i == j)
                continue;

            for (int k = 0; k < points.length; k++) {
                if (k == i || k == j)
                    continue;

                if (getDistance(points[i], points[j]) == getDistance(points[i], points[k]))
                    res++;
            }
        }
    }

    return res;
}

private int getDistance(int[] x, int[] y) {
    int x1 = y[0] - x[0];
    int y1 = y[1] - x[1];

    return x1 * x1 + y1 * y1;
}
```

989. 数组形式的整数加法

这就相当于把题目翻译了一遍，但是提交就会发现 TLE 了，也不难发现时间复杂度是 $O(N^3)$ ，

也就是我们需要优化代码了。。。。

首先题目说 n 个点不同且答案考虑元组顺序，那么我们最外层循环是跑不掉了，因为需要固定每一个点。

里面两层循环可不可以优化一下呢？

其实不难想，当我们固定其中一个点 A 的时候，并且想算距离为 3 的点的个数，那么我们就找出所有和点 A 距离为 3 的点，然后来一个简单的排列组合嘛！

比如找到了 n 个距离为 3 的点，那么我们选择第二个点有 n 种方案，选择第三个点有 $(n - 1)$ 个方案。那么固定点 A 且距离为 3 的所有可能就是 $n*(n-1)$ 种，这只考虑了距离为 3，还有许多其他距离呢，这不就又回到了我们统计元素频率的问题上了嘛，当然哈希表用起来！不明白的看下讲义。

代码

代码支持： Java

Java Code:

```
```java {13-14,17-18} public int numberOfBoomerangs(int[][] points) {

 if (points == null || points.length <= 2)
 return 0;

 int res = 0;
 Map<Integer, Integer> equalCount = new HashMap<>();

 for (int i = 0; i < points.length; ++i) {
 for (int j = 0; j < points.length; ++j) {
 int dinstance = getDistance(points[i], points[j]);
 equalCount.put(dinstance, equalCount.getOrDefault(dinstance, 0) + 1);
 }

 for (int count : equalCount.values())
 res += count * (count - 1);
 equalCount.clear();
 }

 return res;
}
```

## 989. 数组形式的整数加法

```
}

private int getDistance(int[] x, int[] y) {

 int x1 = y[0] - x[0];
 int y1 = y[1] - x[1];

 return x1 * x1 + y1 * y1;

}'''
```

### 复杂度分析

- 时间复杂度:  $O(n^2)$
- 空间复杂度:  $O(n)$

## 题目地址(3. 无重复字符的最长子串)

<https://leetcode-cn.com/problems/longest-substring-without-repeating-characters/>

## 入选理由

1. 这是最最经典的滑动窗口题目。滑动窗口有时候是需要计数的，那用什么计数呢？哈哈 下一篇咱就讲双指针，滑动窗口就是双指针中的一个部分。大家可以通过这道题预习一下。

## 题目描述

给定一个字符串，请你找出其中不含有重复字符的 最长子串 的长度。

示例 1:

输入: "abcabcbb"

输出: 3

解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

示例 2:

输入: "bbbbb"

输出: 1

解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。

示例 3:

输入: "pwwkew"

输出: 3

解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。

请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子

## 标签

- 双指针
- 滑动窗口
- 哈希表

## 难度

- 中等

## 前置知识

- 哈希表
- 双指针

## 方法1:暴力

我们可以枚举所有的子串。并注意判断是否满足无重复字符的条件。

## 代码

代码支持: JavaScript

Javascript Code:

```
/*
 * @param {string} s
 * @return {number}
 */
var lengthOfLongestSubstring = function (s) {
 let res = 0;
 for (let i = 0; i < s.length; i++) {
 let map = {};
 for (let j = i; j < s.length; j++) {
 if (map[s[j]] !== undefined) {
 break;
 }
 map[s[j]] = true;
 res = Math.max(res, j - i + 1);
 }
 }
 return res;
};
```

## 复杂度分析

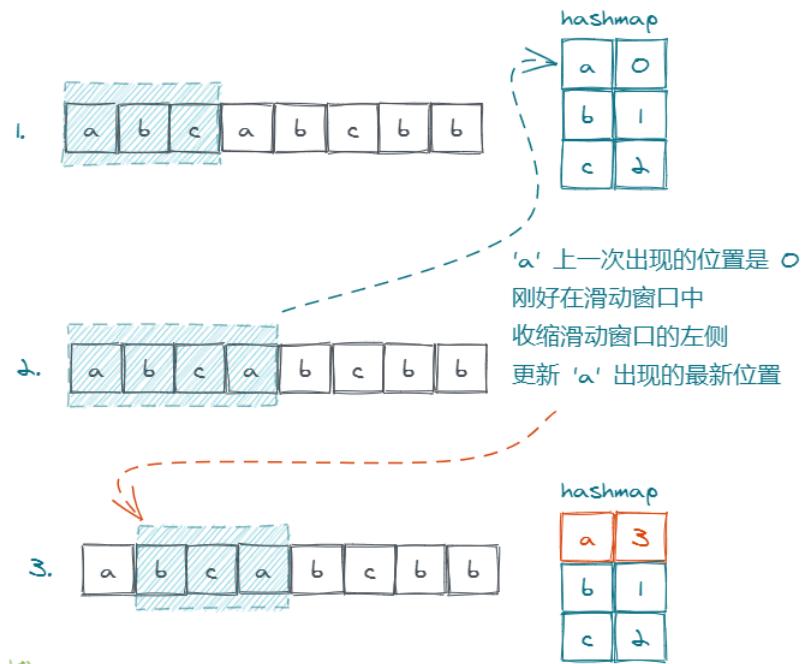
- 时间复杂度:  $O(n^2)$ ,  $n$ 为字符串长度
- 空间复杂度:  $O(s)$ ,  $s$ 为字符集元素个数

## 方法2:哈希表 + 滑动窗口

### 思路A

- 维护一个滑动窗口，当窗口中的字符不重复时，继续向右扩大窗口。

- 当遇到重复字符 `d` 时，将窗口左侧收缩到 `d` 字符上次出现的位置 + 1。
- 为了快速找到字符上次出现的位置，我们可以用一个哈希表来记录每个字符最新出现的位置。
- 在滑动窗口遍历数组的过程中用一个变量记录窗口的最大长度。



## 代码(JavaScript/C++)

JavaScript Code

## 989. 数组形式的整数加法

```
/*
 * @param {string} s
 * @return {number}
 */
var lengthOfLongestSubstring = function (s) {
 const map = {};
 let l = 0,
 r = 0,
 max = 0;

 while (r < s.length) {
 const pos = map[s[r]];
 // 如果 s[r] 曾在 [l, r] 滑动窗口中出现
 // 就收缩滑动窗口左侧，把 l 指针移动到 s[r] 上次出现的位置
 if (pos >= l && pos <= r) l = pos + 1;

 // 更新 s[r] 出现的位置
 map[s[r]] = r;
 // 计算滑动窗口大小
 max = Math.max(max, r - l + 1);
 // 滑动窗口继续右移扩张
 r++;
 }
 return max;
};
```

C++ Code

```

class Solution {
public:
 int lengthOfLongestSubstring(string s) {
 unordered_map<char,int> seen;
 int max_len = 0, l = 0, r = 0;
 while (r < s.size()) {
 if (seen.count(s[r]) > 0) {
 int last_pos = seen[s[r]];
 if (last_pos >= l && last_pos <= r) {
 l = last_pos + 1;
 }
 }
 max_len = max(max_len, r - l + 1);
 seen[s[r]] = r;
 r++;
 }
 return max_len;
 }
};

```

## 思路B

利用**HashSet**来判断是否有重复字符，并且用一个变量记录起始位置，若set中出现了重复字符，则拿当前字串长度和maxLen取最大值，接着将起始位置右移至重复字符的后一个位置，并把之前的字符从set中去除。最后maxLen存的值即为最长的不重复的子串，注意最后返回条件还需要判断一次，因为可能题中所给的字串已经是不重复的了。

## 代码

代码支持： Java

Java Code:

```
``` java {11,14-18} class Solution {
```

989. 数组形式的整数加法

```
public int lengthOfLongestSubstring(String s) {  
  
    int left = 0;  
    int maxLen = 0;  
    Set<Character> set = new HashSet<>();  
  
    for (int i = 0; i < s.length(); i++) {  
  
        if (!set.add(s.charAt(i))) {  
  
            maxLen = Math.max(maxLen, set.size());  
            while (s.charAt(left) != s.charAt(i)) {  
  
                set.remove(s.charAt(left));  
                left++;  
            }  
  
            left += 1;  
        }  
    }  
  
    return Math.max(maxLen, set.size());  
}  
  
}'''
```

复杂度分析

- 时间复杂度: $O(n)$, n 为字符串长度
- 空间复杂度: $O(s)$, s 为字符集元素个数

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>

入选理由

1. 今天和明天都是困难难度，作为压轴。
2. 这道题哈希是如何优化我们的算法的呢？

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：[]

标签

- 字符串
- 双指针
- 哈希表

难度

- 困难

前置知识

- 哈希表
- 双指针
-

思路

还是从题意暴力入手。大体上会有两个想法：

1. 从 words 入手，words 所有单词排列生成字符串 X，通过字符串匹配查看 X 在 s 中的出现位置
2. 从 s 串入手，遍历 s 串中所有长度为 (words[0].length * words.length) 的子串 Y，查看 Y 是否可以由 words 数组构造生成

先看第一种思路：构造 X 的时间开销是 $(words.length)! / (words \text{ 中单词重}\overline{\text{复次数相乘}})$ ，时间复杂度为 $O(m!)$ ，m 为 words 长度。阶乘的时间复杂度基本不可能通过。

下面看第二种思路：仅考虑遍历过程，遍历 s 串的时间复杂度为 $O(n - m + 1)$ ，其中 n 为 s 字符串长度，m 为 $words[0].length * words.length$ ，也就是 words 的字符总数。问题关键在于如何判断 s 的子串 Y 是否可以由 words 数组的构成，由于 words 中单词长度固定，我们可以将 Y 拆分成对应 words[0] 长度的一个个子串 parts，只需要判断 words 和 parts 中的单词是否一一匹配即可，这里用两个哈希表比对出现次数即可。一旦一个对应不上，意味着此种分割方法不正确，继续尝试下一种即可。

代码

```
``` Java {22,28,30-31,35-36} class Solution {
```

## 989. 数组形式的整数加法

```
public List<Integer> findSubstring(String s, String[] words) {
 List<Integer> res = new ArrayList<>();

 Map<String, Integer> map = new HashMap<>();

 if (words == null || words.length == 0)
 return res;

 for (String word : words)
 map.put(word, map.getOrDefault(word, 0) + 1);

 int sLen = s.length(), wordLen = words[0].length(), count = words.length;

 int match = 0;

 for (int i = 0; i < sLen - wordLen * count + 1; i++) {

 //得到当前窗口字符串
 String cur = s.substring(i, i + wordLen * count);
 Map<String, Integer> temp = new HashMap<>();
 int j = 0;

 for (; j < cur.length(); j += wordLen) {

 String word = cur.substring(j, j + wordLen);
 // 剪枝
 if (!map.containsKey(word))
 break;

 temp.put(word, temp.getOrDefault(word, 0) + 1);
 // 剪枝
 if (temp.get(word) > map.get(word))
 break;
 }

 if (j == cur.length())
 res.add(i);
 }

 return res;
}
```

} ``

**复杂度分析**

## 989. 数组形式的整数加法

- 时间复杂度:  $O(n * m * k)$ , n 为字符串 S 长度, m 为 words 数组元素个数, k 为单个 word 字串长度
- 空间复杂度:  $O(m)$ , m 为 words 数组元素个数

## 题目地址(Delete Sublist to Make Sum Divisible By K)

<https://binarysearch.com/problems/Delete-Sublist-to-Make-Sum-Divisible-By-K>

### 入选理由

1. 同余定理+前缀和的巧妙结合

### 题目描述

```
You are given a list of positive integers nums and a posit:

Constraints

 $1 \leq n \leq 100,000$ where n is the length of nums
Example 1
Input
 $\text{nums} = [1, 8, 6, 4, 5]$
 $k = 7$
Output
2
Explanation
We can remove the sublist $[6, 4]$ to get $[1, 8, 5]$ which sur
```

### 标签

- 前缀和
- 数组
- Math
- 哈希表

### 难度

- 中等

### 前置知识

- 哈希表
- 同余定理及简单推导

- 前缀和

## 思路

题目的意思是让我们移除一段最短连续子数组，使得剩下的数字和为 k 的整数倍。

暴力的思路仍然是枚举所有的连续子数组，然后计算连续子数组的和 sum\_range。如果数组的总和 total - sum\_range 是 k 的整数倍，那么我们就得到了一个备胎，遍历完所有的子数组，取备胎中最短的即可。当然如果没有任何备胎，需要返回 -1。

当然，上述方法即使加上剪枝时间复杂度也相对较高，看到被 x 整除，求余数等问题都可以尝试考虑是否可以使用数学中的同余定理，看到连续子数组就可以考虑用前缀和进行优化。

本题可以使用前缀和 + 同余定理进行优化：

- 由前缀和我们知道子数组  $A[i:j]$  的和就是  $\text{pres}[j] - \text{pres}[i-1]$ ，其中 pres 为 A 的前缀和。
- 由同余定理我们知道两个模 k 余数相同的数字相减，得到的值一定可以被 k 整除。

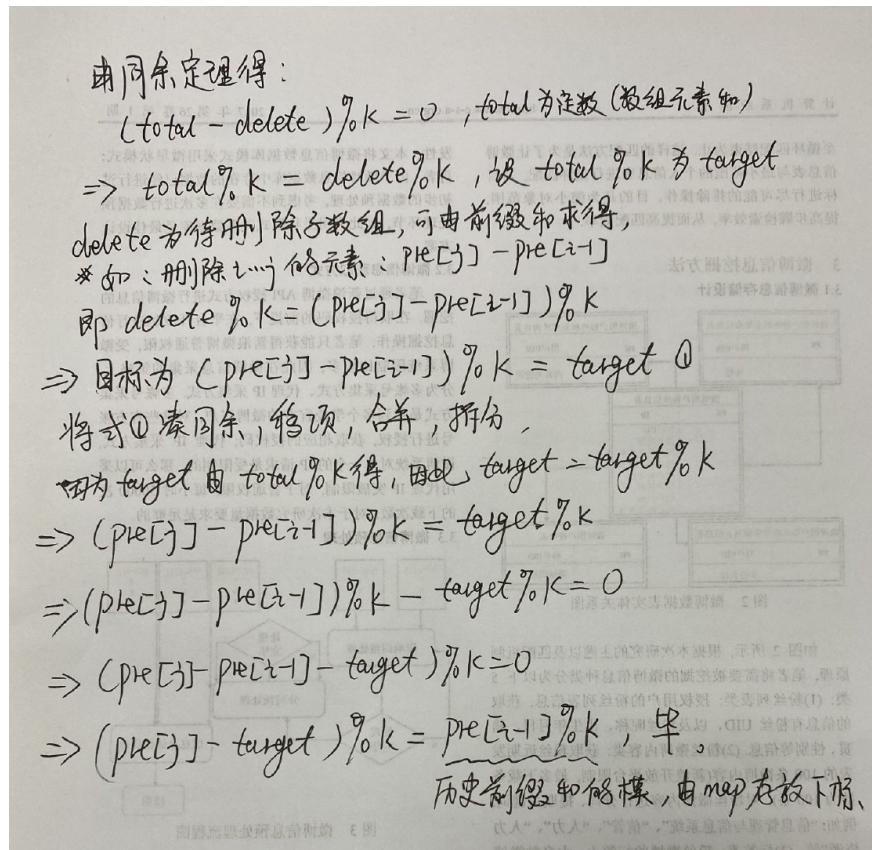
于是，我们可以将前缀和模 k 的余数 x 放到哈希表中，这个哈希表就充当了前缀和的角色，来记录最新的余数 x 对应的下标，记录最新的目的是为了找出符合要求的最短的连续子数组。

算法上，我们可以先计算出总体的数组和 total 模 k 的余数，记为 target，那么我们的目标就是找到一段模 k 等于 target 的子数组。注意，我们需要提前在哈希表中放一个：

```
{
 0: -1
}
```

以应对从数组索引 0 处开始取子数组的情况。

## 推导过程（理解的可跳过此部分）



## 代码

代码支持： Java

Java Code:

注：-1 % 4 在 Java 中为-1，而我们期望为 3，为了解决正负数求余统一，采用 **Math.floorMod**

```
```java {15,23-26} import java.util.*;
```

```
class Solution {
```

989. 数组形式的整数加法

```
public int solve(int[] nums, int k) {  
  
    int tar = 0;  
  
    for (int n : nums)  
        tar += n;  
  
    tar = Math.floorMod(tar, k);  
  
    Map<Integer, Integer> map = new HashMap<>();  
    map.put(0, -1);  
  
    int prefix = 0, res = nums.length;  
  
    for (int i = 0; i < nums.length; i++) {  
  
        prefix += nums[i];  
        int mod = Math.floorMod(prefix, k);  
        map.put(mod, i);  
  
        if (map.containsKey(Math.floorMod(prefix - tar, k)))  
            res = Math.min(res, i - map.get(Math.floorMod(prefix - tar, k)));  
    }  
  
    return res == nums.length ? -1 : res;  
}  
  
}  
}'''
```

复杂度分析

令 n 为数组长度。

- 时间复杂度: $\$O(n)\$$
- 空间复杂度: $\$O(\min(n, k))\$$

相关题目 (换皮题)

- [974. 和可被 K 整除的子数组](#)
- [523. 连续的子数组和](#)

题目地址(876. 链表的中间结点)

<https://leetcode-cn.com/problems/middle-of-the-linked-list/>

入选理由

1. 简单难度的双指针。接下来几天的题目类型分别是：二分，快慢指针，滑动窗口

标签

- 双指针
- 链表

难度

- 简单

题目描述

给定一个头结点为 `head` 的非空单链表，返回链表的中间结点。

如果有两个中间结点，则返回第二个中间结点。

示例 1：

输入：[1,2,3,4,5]

输出：此列表中的结点 3（序列化形式：[3,4,5]）

返回的结点值为 3。 （测评系统对该结点序列化表述是 [3,4,5]）。

注意，我们返回了一个 `ListNode` 类型的对象 `ans`，这样：

`ans.val = 3, ans.next.val = 4, ans.next.next.val = 5`，以及

示例 2：

输入：[1,2,3,4,5,6]

输出：此列表中的结点 4（序列化形式：[4,5,6]）

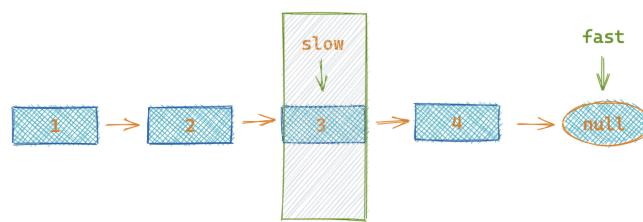
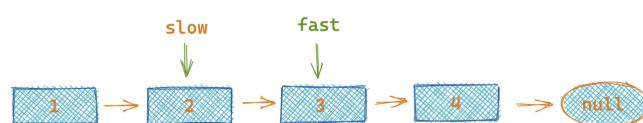
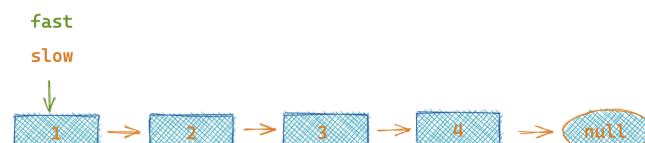
由于该列表有两个中间结点，值分别为 3 和 4，我们返回第二个结点。

提示：

给定链表的结点数介于 1 和 100 之间。

思路

用两个指针记为快指针和慢指针，快指针每次走 2 步，慢指针每次走 1 步，当快指针走到末尾的时候，慢指针刚好到达链表中点。



989. 数组形式的整数加法

证明方法见讲义。

代码

代码支持: JS, Python3

JS Code:

```
/**  
 * @param {ListNode} head  
 * @return {ListNode}  
 */  
var middleNode = function (head) {  
    let slow = (fast = head);  
    while (slow && fast && fast.next) {  
        fast = fast.next.next;  
        slow = slow.next;  
    }  
    return slow;  
};
```

Python3 Code:

```
class Solution:  
    def middleNode(self, head: ListNode) -> ListNode:  
        slow = fast = head  
        while fast and fast.next:  
            fast = fast.next.next  
            slow = slow.next  
        return slow
```

复杂度分析

令 n 为链表长度

- 时间复杂度: $O(n)$
- 空间复杂度: $O(1)$

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/>

入选理由

1. 双指针中的一种类型：读写双指针

标签

- 数组
- 双指针

难度

- 简单

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。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,

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调用者可见。

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);

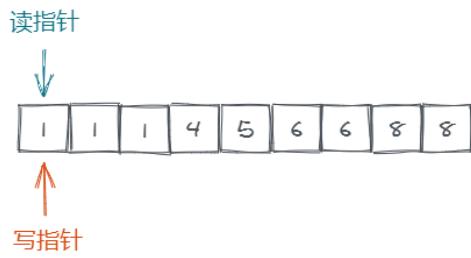
// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

双指针

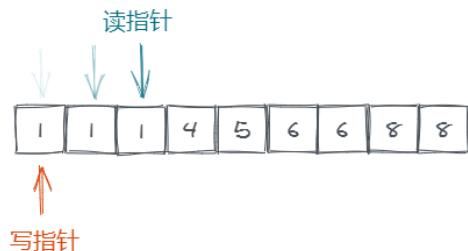
思路

- 用一个读指针，一个写指针遍历数组。
- 遇到重复的元素 读指针 就继续前移。
- 遇到不同的元素 写指针 就前移一步，写入那个元素。

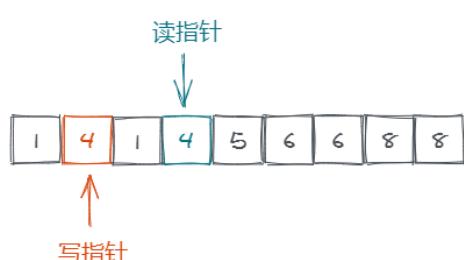
989. 数组形式的整数加法



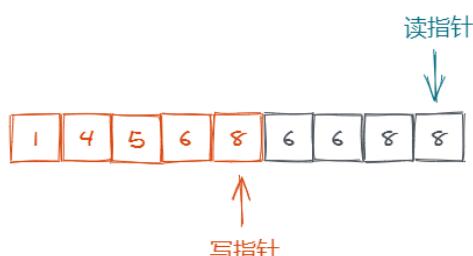
读、写指针一样时，
写指针原地不动
读指针继续往前走



当读、写指针不同时
写指针前进一步
将读指针内容写入写指针的位置



读指针继续往前走
重复上面的步骤



最后返回写指针位置+1
就是新数组的长度了

@suukii

代码

代码支持: JS, Python3

JS Code:

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var removeDuplicates = function (nums) {
    let p1 = 0,
        p2 = 0;

    while (p2 < nums.length) {
        if (nums[p1] != nums[p2]) {
            p1++;
            nums[p1] = nums[p2];
        }
        p2++;
    }
    return p1 + 1;
};
```

Python3 Code:

```
class Solution(object):
    def removeDuplicates(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """

        if not nums: return 0

        l, r = 0, 0
        while r < len(nums):
            if nums[l] != nums[r]:
                l += 1
                nums[l] = nums[r]
            r += 1
        return l + 1
```

复杂度分析

令 n 为数组长度

时间复杂度: $O(n)$ 空间复杂度: $O(1)$

题目地址 (35. 搜索插入位置)

<https://leetcode-cn.com/problems/search-insert-position>

入选理由

1. 接下来两天是二分，双指针中的一种类型。专题篇会单独对二分进行详细整理，包教包会的那种 ^_^
2. 难度是简单，大家一定要打卡哦

标签

- 双指针
- 二分

难度

- 简单

题目描述

给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。
如果目标值不存在于数组中，返回它将会被按顺序插入的位置。

你可以假设数组中无重复元素。

示例 1:

输入: [1,3,5,6], 5

输出: 2

示例 2:

输入: [1,3,5,6], 2

输出: 1

示例 3:

输入: [1,3,5,6], 7

输出: 4

示例 4:

输入: [1,3,5,6], 0

输出: 0

暴力法

思路

一次遍历枚举数组，寻找第一个大于等于 target 的值即可。

代码

代码支持: JS

```
var searchInsert = function (nums, target) {
    for (let i = 0; i < nums.length; i++) {
        if (nums[i] >= target) {
            return i;
        }
    }
    return nums.length;
};
```

复杂度分析

令 n 为数组长度

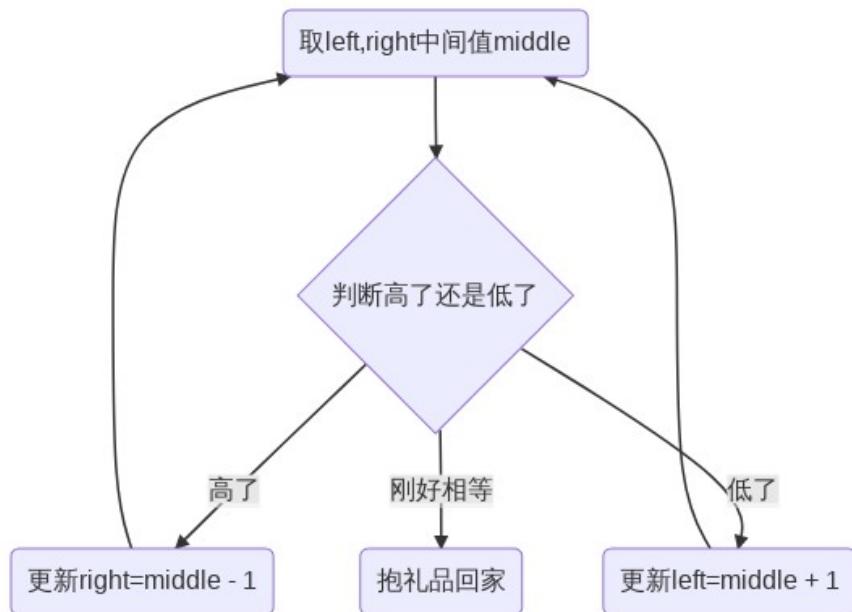
- 时间复杂度: $O(n)$
- 空间复杂度: $O(1)$

二分

思路

举个例子，不知道看官们有没有看过这样一档电视节目，给一个商品，找观众去猜价格，如果猜中了，商品就作为礼品给观众，观众猜一个价格后，主持人会告诉你高了还是低了。如果你对这个商品的价值一无所知，你会怎么做？

我们会给一个中间值，通过主持人给的信息更新中间值，更新流程如下



这就是典型的二分法，可以直接套讲义模板

代码

代码支持：Java, JS

Java Code:

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

while (l <= r) {
    int mid = (l + r) >> 1
    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 = mid + 1
    if(一定条件) r = mid - 1
}
// 看具体题意, 此时 l === r + 1
return l
```

JS Code:

989. 数组形式的整数加法

```
var searchInsert = function (nums, target) {
    let left = 0,
        right = nums.length - 1;
    while (left <= right) {
        const middle = (left + right) >> 1;
        const middleValue = nums[middle];
        if (middleValue === target) {
            return middle;
        } else if (middleValue < target) {
            left = middle + 1;
        } else {
            right = middle - 1;
        }
    }
    return left;
};
```

复杂度分析

令 n 为数组长度

- 时间复杂度: $O(\log n)$
- 空间复杂度: $O(1)$

题目地址(239. 滑动窗口最大值)

<https://leetcode-cn.com/problems/sliding-window-maximum/>

入选理由

1. 双指针最后一种类型，滑动窗口。由于专题篇会继续，因此这里就只
整一道

标签

- 双指针
- 滑动窗口

难度

- 困难

题目描述

989. 数组形式的整数加法

给定一个数组 `nums`, 有一个大小为 `k` 的滑动窗口从数组的最左侧移动到数组的最右侧，返回滑动窗口中的最大值。

进阶：

你能在线性时间复杂度内解决此题吗？

示例：

输入: `nums = [1,3,-1,-3,5,3,6,7]`, 和 `k = 3`

输出: `[3,3,5,5,6,7]`

解释:

滑动窗口的位置	最大值
<code>[1 3 -1] -3 5 3 6 7</code>	<code>3</code>
<code>1 [3 -1 -3] 5 3 6 7</code>	<code>3</code>
<code>1 3 [-1 -3 5] 3 6 7</code>	<code>5</code>
<code>1 3 -1 [-3 5 3] 6 7</code>	<code>5</code>
<code>1 3 -1 -3 [5 3 6] 7</code>	<code>6</code>
<code>1 3 -1 -3 5 [3 6 7]</code>	<code>7</code>

提示:

```
1 <= nums.length <= 10^5
-10^4 <= nums[i] <= 10^4
1 <= k <= nums.length
```

前置知识

- 队列
- 滑动窗口

公司

- 阿里
- 腾讯
- 百度
- 字节

暴力

思路

题目很好理解，简单来说就是寻找滑动窗口内的最大值，所以算法框架就有了

- 维护一个滑动窗口，每次获取滑动窗口最大值

算法描述为

```
function solution(nums, k) {  
    const res = [];  
    for (let i = 0; i <= nums.length - k; i++) {  
        let cur = maxInSlidingWindow(nums, i, i + k);  
    }  
    return res;  
}
```

接下来就是去实现 `maxInSlidingWindow`，

暴力线性比较滑动窗口内的每个值

```
function maxInSlidingWindow(nums, start, end) {  
    let max = -Infinity;  
    for (let i = start; i < end; i++) {  
        max = Math.max(nums[i], max);  
    }  
    return max;  
}
```

代码

代码支持：JS,Python3

JS Code:

```

var maxSlidingWindow = function (nums, k) {
    const res = [];
    for (let i = 0; i <= nums.length - k; i++) {
        let cur = maxInSlidingWindow(nums, i, i + k);
        res.push(cur);
    }
    return res;
};

function maxInSlidingWindow(nums, start, end) {
    let max = -Infinity;
    for (let i = start; i < end; i++) {
        max = Math.max(nums[i], max);
    }
    return max;
}

```

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

```

复杂度分析

令 n 为数组长度

- 时间复杂度: $O(n * k)$
- 空间复杂度: $O(1)$

堆/优先队列

思路

求极值，特别是待求队列内容变动的场景下，用堆/优先队列是一种常见的方案，这里可以对滑动窗口建立一个大小为 k 的大顶堆，窗口滑动时，从堆中去除一个滑动窗口最前的一个数，添加滑动窗口后一个数，取得窗口最大值，每次堆操作时间复杂度 $O(\log K)$

代码

代码支持: Java

Java Code:

```
public ArrayList<Integer> maxInWindows2(int[] num, int size) {
    if (num == null || num.length == 0 || size <= 0 || size > num.length)
        return new ArrayList<>();
    }
    ArrayList<Integer> result = new ArrayList<>();
    PriorityQueue<Integer> q = new PriorityQueue(size);
    for (int i = 0; i < num.length; i++) {
        if (q.size() == size)
            q.remove(num[i - size]);
        }
        q.add(num[i]);
        if (i >= size - 1) {
            result.add(q.peek());
        }
    }
    int[] arr = new int[result.size()];
    for (int i = 0; i < result.size(); i++) {
        arr[i] = result.get(i);
    }
    return result;
}
```

复杂度分析

令 n 为数组长度

- 时间复杂度: $O(n \log k)$
- 空间复杂度: $O(k)$

单调队列

思路

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。

其实，我们没必要存储窗口内的所有元素。如果新进入的元素比前面的大，那么前面的元素就不再有利用价值，可以直接移除。这提示我们使用一个[单调递增栈](#)来完成。

但由于窗口每次向右移动的时候，位于窗口最左侧的元素是需要被擦除的，而栈只能在一端进行操作。

989. 数组形式的整数加法

而如果你使用数组实现，就是可以在另一端操作了，但是时间复杂度仍然是 $O(k)$ ，和上面的暴力算法时间复杂度一样。

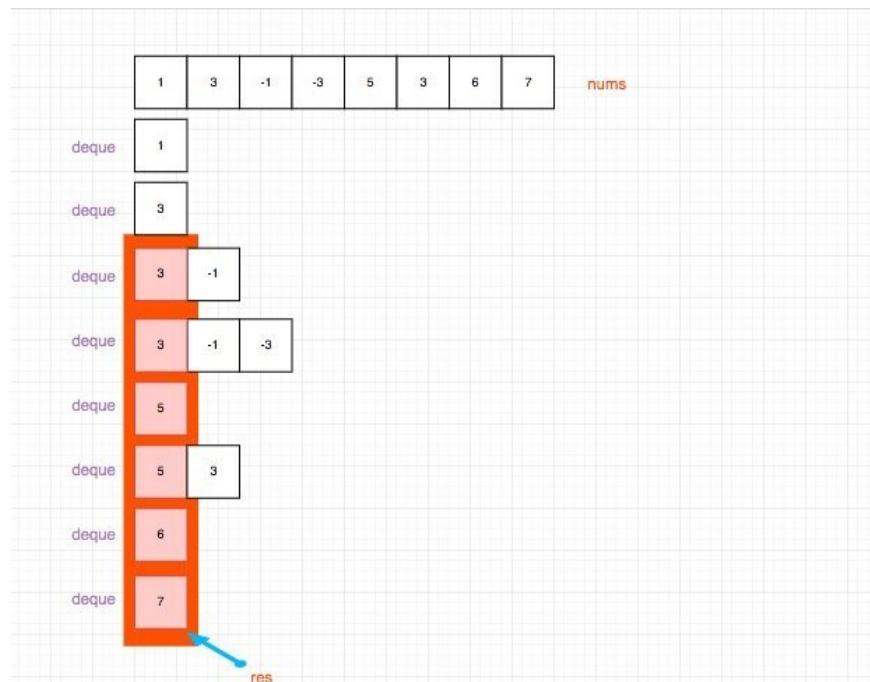
因此，我们考虑使用链表来实现，维护两个指针分别指向头部和尾部即可，这样做的时间复杂度是 $O(1)$ ，这就是双端队列。

因此思路就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

经过上面的分析，不难知道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小的。用图来表示就是：



代码

代码支持：JS, Python3

JS Code:

989. 数组形式的整数加法

```
var maxSlidingWindow = function (nums, k) {
    const res = [];
    const dequeue = new Dequeue([]);
    // 前 k - 1 个数入队
    for (let i = 0; i < k - 1; i++) {
        dequeue.push(nums[i]);
    }

    // 滑动窗口
    for (let i = k - 1; i < nums.length; i++) {
        dequeue.push(nums[i]);
        res.push(dequeue.max());
        dequeue.shift(nums[i - k + 1]);
    }
    return res;
};

class Dequeue {
    constructor(nums) {
        this.list = nums;
    }

    push(val) {
        const nums = this.list;
        // 保证数据从队头到队尾递减
        while (nums[nums.length - 1] < val) {
            nums.pop();
        }
        nums.push(val);
    }

    // 队头出队
    shift(val) {
        let nums = this.list;
        if (nums[0] === val) {
            // 这里的js实现shift()理论上复杂度应该是O(k)，就不去真实实现
            nums.shift();
        }
    }

    max() {
        return this.list[0];
    }
}
```

Python3:

989. 数组形式的整数加法

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) ->
        q = collections.deque() # 本质就是单调队列
        ans = []
        for i in range(len(nums)):
            while q and nums[q[-1]] <= nums[i]: q.pop() # 移除失
            while q and i - q[0] >= k: q.popleft() # 移除失去
            q.append(i)
            if i >= k - 1: ans.append(nums[q[0]])
        return ans
```

复杂度分析

令 n 为数组长度

- 时间复杂度: $O(n)$
- 空间复杂度: $O(k)$

参考

JS 的 `dequeue` 没有使用链表模拟，而是使用了数组。具体的链表实现可以参考[deque](#)

另外可以参考 [leetcode 仓库的题解](#)

- 69. [x 的平方根](#)
- 278. [第一个错误的版本](#)
- 762. [Number Stream to Intervals](#)
- 796. [Minimum Light Radius](#)
- 822. [Kth-Pair-Distance](#)
- 778. [水位上升的泳池中游泳](#)
- 1456. [定长子串中元音的最大数目](#)
- 837. [新 21 点](#)
- 438. [找到字符串中所有字母异位词](#)
- 76. [最小覆盖子串](#)
- Number of Operations to Decrement Target to Zero
- 401. [二进制手表](#)
- 52. [N 皇后 II](#)
- 695. [岛屿的最大面积](#)
- 1162. [地图分析](#)
- Shortest-Cycle-Containing-Target-Node
- Top-View-of-a-Tree
- 746. [使用最小花费爬楼梯](#)
- 198. [打家劫舍](#)
- 673. [最长递增子序列的个数](#)
- 1143. [最长公共子序列](#)
- 62. [不同路径](#)
- 688. [“马”在棋盘上的概率](#)
- 464. [我能赢么](#)
- 416. [分割等和子集](#)
- 494. [目标和](#)
- 322. [零钱兑换](#)
- 518. [零钱兑换 II](#)
- 455. [分发饼干](#)
- 435. [无重叠区间](#)
- 881. [救生艇](#)
- 96. [不同的二叉搜索树](#)
- 23. [合并 K 个排序链表](#)
- 932. [漂亮数组](#)
- 260. [只出现一次的数字 III](#)
- 78. [子集](#)

题目地址(69. x 的平方根)

<https://leetcode-cn.com/problems/sqrtx>

入选理由

1. 最基本的二分类型，讲义来写有哦。那么这是哪一种呢？题解里告诉我吧

标签

- 二分

难度

- 简单

题目描述

实现 `int sqrt(int x)` 函数。

计算并返回 x 的平方根，其中 x 是非负整数。

由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

示例 1:

输入: 4

输出: 2

示例 2:

输入: 8

输出: 2

说明: 8 的平方根是 2.82842...,

由于返回类型是整数，小数部分将被舍去

前置知识

- 二分法

思路

989. 数组形式的整数加法

此题就是我讲义提到的**寻找最右边的满足条件的值** 的变种。

变种的点在于本题不是给定一个有序数组和 target。不过我们只需要一点点抽象即可轻松转化为我们已知的问题，进而使用模板解决。

简单抽象一下，nums 数组就是 $[0,1,2,3,4,\dots,x]$ target 就是 x 的平方根。以题目的 8 为例，我们先不考虑结果只保留整数的部分最后再将小数部分去掉即可。·这么来看， $2,\dots,2.82841,2.82842,\dots$ 都是符合的。由于需要返回不带小数的，那不就是返回 **最左边的满足条件的值** 么？但是这种算法比较复杂，原因在于计算误差，比如题目限定了 10^{-5} 以内的误差都可以，那么是可以的。

仍然以 8 为例， 我们想要在 $1,2,3,4,5\dots$ （注意我这里不考虑小数了）找满足条件的 ans，使得 ans^2 刚好小于等于 8，也就是找所有满足 $ans^2 \leq 8$ 的最大值，也就是 **找最右边的满足条件的值**，这里的条件就是 ≤ 8 。

我们可以找所有满足 $ans^2 \geq 8$ 的最小值，也就是 **找最左边的满足条件的值**，这里的条件就是 ≥ 8 么？很明显不可以。这样算的话，答案就是 3 了。（如果题目让我们向上取整就可以用啦 ^_ ^）

代码

Python：

```
class Solution:
    def mySqrt(self, x: int) -> int:
        ans, l, r = 0, 0, x
        while l <= r:
            mid = (l + r) // 2
            if mid ** 2 > x:
                r = mid - 1
            if mid ** 2 <= x:
                ans = mid
                l = mid + 1
        return int(ans)
```

Java：

989. 数组形式的整数加法

```
class Solution {
    public int mySqrt(int x) {
        if(x==1)
            return 1;
        int left=0;
        int right=46340;
        while(left<=right){
            int mid=left+(right-left)/2;
            if(mid*mid>x){
                right=mid-1;
            }else if(mid*mid<x){
                left=mid+1;
            }else{
                return mid;
            }
        }
        return right;
    }
}
```

C++

```
class Solution {
public:
    int mySqrt(int x) {
        int l = 1, r = x, mid;
        while(l <= r){
            mid = l + (r-l)/2;
            if(x/mid > mid) l = mid + 1;
            else if (x/mid < mid) r = mid - 1;
            else return mid;
        }
        return r;
    };
};
```

复杂度分析

- 时间复杂度: $O(\log_x)$
- 空间复杂度: $O(1)$

题目地址 (278. 第一个错误的版本)

<https://leetcode-cn.com/problems/first-bad-version>

入选理由

1. 仍然是一个简单二分。出这个题目的是为了让大家抽离问题本质，然后使用合适算法。简单来说就是换个皮大家也要会

标签

- 二分

难度

- 简单

题目描述

你是产品经理，目前正在带领一个团队开发新的产品。不幸的是，你的产品的最新版本有bug。

假设你有 n 个版本 [1, 2, ..., n]，你想找出导致之后所有版本出错的第一个错误的版本。

你可以通过调用 bool isBadVersion(version) 接口来判断版本号 version 是否在你的产品中有bug。

示例：

给定 n = 5，并且 version = 4 是第一个错误的版本。

调用 isBadVersion(3) -> false

调用 isBadVersion(5) -> true

调用 isBadVersion(4) -> true

所以，4 是第一个错误的版本。

前置知识

- 二分法

思路

989. 数组形式的整数加法

典型的二分寻找最左边的满足条件的值，具体看我的讲义。一句话概括就是：寻找最左边和寻找指定值的差别就是碰到等于号的处理情况。如果是寻找最左边那么碰到等于继续收缩右边界（寻找最右边就是收缩左边界），查找指定值则是直接返回。

我们直接套模板即可。

代码

Python3:

```
class Solution:
    def firstBadVersion(self, n):
        l, r = 1, n
        while l <= r:
            mid = (l + r) // 2
            if isBadVersion(mid):
                # 收缩
                r = mid - 1
            else:
                l = mid + 1
        return l
```

CPP:

```
// The API isBadVersion is defined for you.
// bool isBadVersion(int version);

class Solution {
public:
    int firstBadVersion(int n) {
        int l=1,r=n;
        while(l<r){
            int mid=l+(r-l)/2;
            if(isBadVersion(mid)){
                r=mid;
            }
            else
                l=mid+1;
        }

        return l;
    }
};
```

Java:

989. 数组形式的整数加法

```
public class Solution extends VersionControl {
    public int firstBadVersion(int n) {
        int l = 1;
        int r = n;
        while (l <= r) {
            int mid = l + ((r - l) >> 1);
            if (isBadVersion(mid)) {
                r = mid - 1;
            } else {
                l = mid + 1;
            }
        }

        return l;
    }
}
```

JS:

```
var solution = function (isBadVersion) {
    /**
     * @param {integer} n Total versions
     * @return {integer} The first bad version
     */
    return function (n) {
        let left = 1;
        let right = n;

        while (left <= right) {
            let mid = Math.floor((right + left) / 2);

            if (isBadVersion(mid)) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        return left;
    };
};
```

复杂度分析

- 时间复杂度: $O(\log N)$
- 空间复杂度: $O(1)$

题目地址 (**762.Number Stream to Intervals**)

<https://binarysearch.com/problems/Triple-Inversion>

入选理由

1. 通常二分都是基于有序序列二分，题目直接给了有序序列自然就简单，如果题目没给就需要我们自己构造，这道题就是。2. 和专题篇的某一个专题联动（会是谁呢？）3. 力扣中有换皮题，其实这道题就是典型的xxx（猜猜是啥）。

标签

- 二分

难度

- 困难

题目描述

```
Given a list of integers nums, return the number of pairs :  
  
Constraints  
  
n ≤ 100,000 where n is the length of nums  
Example 1  
Input  
nums = [7, 1, 2]  
Output  
2  
Explanation  
We have the pairs (7, 1) and (7, 2)
```

前置知识

- 二分法

暴力法（超时）

思路

本题和力扣 [493. 翻转对](#) 和 [剑指 Offer 51. 数组中的逆序对](#) 一样，都是求逆序对的换皮题。

暴力的解法可以枚举所有可能的 j ，然后往前找 i 使得满足 $\text{nums}[i] > \text{nums}[j] * 3$ ，我们要做的就是将满足这种条件的 i 数出来有几个即可。这种算法时间复杂度为 $O(n^2)$ 。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def solve(self, A):
        ans = 0
        for i in range(len(A)):
            for j in range(i+1, len(A)):
                if A[i] > A[j] * 3: ans += 1
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度： $O(n^2)$
- 空间复杂度： $O(1)$

二分法

思路

这道题我们也可以反向思考。即思考：对于 nums 中的每一项 num ，我们找前面出现过的大于 $\text{num} * 3$ 的数。

我们可以自己构造有序序列 d ，然后在 d 上做二分。如何构建 d 呢？很简单，就是将 nums 中已经遍历过的数字全部放到 d 中即可。

代码表示就是：

```
d = []
for a in A:
    bisect.insort(d, a)
```

989. 数组形式的整数加法

`bisect.insort` 指的是使用二分找到插入点，并将数插入到数组中，使得插入后数组仍然有序。虽然使用了二分，使得找到插入点的时间复杂度为 $O(\log n)$ ，但是由于数组的特性，插入导致的数组项后移的时间复杂度为 $O(n)$ ，因此总的时间复杂度为 $O(n^2)$ 。

Python3 Code:

```
class Solution:
    def solve(self, A):
        d = []
        ans = 0

        for a in A:
            i = bisect.bisect_right(d, a * 3)
            ans += len(d) - i
            bisect.insort(d, a)
```

由于上面的算法瓶颈在于数组的插入后移带来的时间。因此我们可以使用平衡二叉树来减少这部分时间，使用平衡二叉树可以使得插入时间稳定在 $O(\log n)$ ，Python 可使用 `SortedList` 来实现，Java 可用 `TreeMap` 代替。

代码

代码支持：Python3

Python3 Code:

```
from sortedcontainers import SortedList
class Solution:
    def solve(self, A):
        d = SortedList()
        ans = 0

        for a in A:
            i = d.bisect_right(a * 3)
            ans += len(d) - i
            d.add(a)
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度： $O(n \log n)$
- 空间复杂度： $O(n)$

分治法

思路

我们接下来介绍更广泛使用的，效率更高的解法 分治。我们进行一次归并排序，并在归并过程中计算逆序数，换句话说 逆序对是归并排序的副产物。

如果不熟悉归并排序，可以先查下相关资料。如果你直接想看归并排序代码，那么将的代码统计 cnt 部分删除就好了。

归并排序实际上会把数组分成两个有序部分，我们不妨称其为左和右，归并排序的过程中会将左右两部分合并成一个有序的部分，对于每一个左右部分，我们分别计算其逆序数，然后全部加起来就是我们要求的逆序数。那么分别如何求解左右部分的逆序数呢？

首先我们知道归并排序的核心在于合并，而合并两个有序数组是一个[简单题目](#)。我这里给贴一下大概算法：

```
def mergeTwo(nums1, nums2):
    res = []
    i = j = 0
    while i < len(nums1) and j < len(nums2):
        if nums1[i] < nums[j]:
            res.append(nums1[i])
            i += 1
        else:
            res.append(nums[j])
            j += 1
    while i < len(nums1):
        res.append(nums1[i])
        i += 1
    while j < len(nums2):
        res.append(nums2[j])
        j += 1
    return res
```

而我们要做的就是在上面的合并过程中统计逆序数。

为了方便描述，我将题目中的： $i < j$ such that $\text{nums}[i] > \text{nums}[j]$ * 3，改成 $i < j$ such that $\text{nums}[i] > \text{nums}[j]$ 。也就是将 3 倍变成一倍。如果你理解了这个过程，只需要比较的时候乘以 3 就行，其他逻辑不变。

比如对于左: [1, 2, 3, 4]右: [2, 5]。由于我的算法是按照 [start, mid], [mid, end] 区间分割的, 因此这里的 mid 为 3 (具体可参考下方代码区)。其中 i, j 指针如粗体部分 (左数组的 3 和右数组的 2)。那么逆序数就是 $mid - i + 1$ 也就是 $3 - 2 + 1 = 2$ 即 (3, 2) 和 (4, 2)。其原因在于如果 3 大于 2, 那么 3 后面不用看了, 肯定都大于 2。之后会变成: [1, 2, 3, 4] 右: [2, 5] (左数组的 3 和右数组的 5), 继续按照上面的方法计算直到无法进行即可。

```

class Solution:
    def solve(self, nums: List[int]) -> int:
        self.cnt = 0
        def merge(nums, start, mid, end):
            i, j, temp = start, mid + 1, []
            while i <= mid and j <= end:
                if nums[i] <= nums[j]:
                    temp.append(nums[i])
                    i += 1
                else:
                    self.cnt += mid - i + 1
                    temp.append(nums[j])
                    j += 1
            while i <= mid:
                temp.append(nums[i])
                i += 1
            while j <= end:
                temp.append(nums[j])
                j += 1

            for i in range(len(temp)):
                nums[start + i] = temp[i]

        def mergeSort(nums, start, end):
            if start >= end: return
            mid = (start + end) // 2
            mergeSort(nums, start, mid)
            mergeSort(nums, mid + 1, end)
            merge(nums, start, mid, end)
            mergeSort(nums, 0, len(nums) - 1)
        return self.cnt

```

注意上述算法在 mergeSort 中我们每次都开辟一个新的 temp, 这样空间复杂度大概相当于 $N \log N$, 实际上我们完全每必要每次 mergeSort 都开辟一个新的, 而是大家也都用一个。具体见下方代码区。

代码

989. 数组形式的整数加法

代码支持: Python3

Python3 Code:

```
class Solution:
    def solve(self, nums) -> int:
        self.cnt = 0
    def merge(nums, start, mid, end, temp):
        i, j = start, mid + 1
        while i <= mid and j <= end:
            if nums[i] <= nums[j]:
                temp.append(nums[i])
                i += 1
            else:
                temp.append(nums[j])
                j += 1
        # 防住
        # 这里代码开始
        ti, tj = start, mid + 1
        while ti <= mid and tj <= end:
            if nums[ti] <= 3 * nums[tj]:
                ti += 1
            else:
                self.cnt += mid - ti + 1
                tj += 1
        # 这里代码结束
        while i <= mid:
            temp.append(nums[i])
            i += 1
        while j <= end:
            temp.append(nums[j])
            j += 1
        for i in range(len(temp)):
            nums[start + i] = temp[i]
        temp.clear()

    def mergeSort(nums, start, end, temp):
        if start >= end: return
        mid = (start + end) >> 1
        mergeSort(nums, start, mid, temp)
        mergeSort(nums, mid + 1, end, temp)
        merge(nums, start, mid, end, temp)
        mergeSort(nums, 0, len(nums) - 1, [])
        return self.cnt
```

复杂度分析

989. 数组形式的整数加法

令 n 为数组长度。

- 时间复杂度: $O(n \log n)$
- 空间复杂度: $O(n)$

力扣的小伙伴可以[关注我](#)，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。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 40K 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。

题目地址 (796. Minimum Light Radius)

<https://binarysearch.com/problems/Minimum-Light-Radius>

入选理由

1. 能力检测 + 最左二分的典型题目

标签

- 二分

难度

- 困难

题目描述

```
You are given a list of integers nums representing coordinates on a 1D plane.  
Now consider k lamps are placed at integer coordinates on the same 1D plane.  
Each lamp has a light radius of r. If the distance between a lamp at coordinate  
x and a point at coordinate y is less than or equal to r, then the point is  
considered to be illuminated by the lamp.  
  
Constraints  
  
 $n \leq 100,000$  where n is the length of nums  
Example 1  
Input  
nums = [3, 4, 5, 6]  
Output  
0.5  
Explanation  
If we place the lamps on 3.5, 4.5 and 5.5 then with  $r = 0.5$ , all points  
are illuminated.
```

前置知识

- 排序
- 二分法

二分法

思路

本题和力扣 475. 供暖器 类似。

这道题含义是给你一个数组 `nums`, 让你在 $[min(nums), max(nums)]$ 范围内放置 3 个灯, 每个灯覆盖范围都是 r , 让你求最小的 r 。

之所以不选择小于 $min(nums)$ 的位置和大于 $max(nums)$ 的位置是因为没有必要。比如选取了小于 $min(nums)$ 的位置 pos , 那么选取 pos 一定不比选择 $min(nums)$ 位置好。

这道题的核心点还是一样的思维模型, 即:

- 确定 r 的上下界, 这里 r 的下界是 0 上界是 $max(nums) - min(nums)$ 。
- 对于上下界之间的所有可能 x 进行枚举 (不妨从小到大枚举), 检查半径为 x 是否可以覆盖所有, 返回第一个可以覆盖所有的 x 即可。

注意到我们是在一个有序序列进行枚举, 因此使用二分就应该想到。可使用二分的核心点在于: 如果 x 不行, 那么小于 x 的所有半径都必然不行。

接下来的问题就是给定一个半径 x , 判断其是否可覆盖所有的房子。

这其实就是我们讲义中提到的能力检测二分, 我定义的函数 `possible` 就是能力检测。

首先对 `nums` 进行排序, 这在后面会用到。然后从左开始模拟放置灯。先在 $nums[0] + r$ 处放置一个灯, 其可以覆盖 $[0, 2r]$ 。由于 `nums` 已经排好序了, 那么这个灯可以覆盖到的房间其实就是 `nums` 中坐标小于等于 $2r$ 所有房间, 使用二分查找即可。对于 `nums` 右侧的所有房间我们需要继续放置灯, 采用同样的方式即可。

能力检测核心代码:

```
def possible(diameter):
    start = nums[0]
    end = start + diameter
    for i in range(LIGHTS):
        idx = bisect_right(nums, end)
        if idx >= N:
            return True
        start = nums[idx]
        end = start + diameter
    return False
```

由于我们想要找到满足条件的最小值, 因此可直接套用最左二分模板。

代码

代码支持: Python3

Python3 Code:

```

class Solution:
    def solve(self, nums):
        nums.sort()
        N = len(nums)
        if N <= 3:
            return 0
        LIGHTS = 3
        # 这里使用的是直径，因此最终返回需要除以 2
        def possible(diameter):
            start = nums[0]
            end = start + diameter
            for i in range(LIGHTS):
                idx = bisect_right(nums, end)
                if idx >= N:
                    return True
                start = nums[idx]
                end = start + diameter
            return False

        l, r = 0, nums[-1] - nums[0]
        while l <= r:
            mid = (l + r) // 2
            if possible(mid):
                r = mid - 1
            else:
                l = mid + 1
        return l / 2

```

复杂度分析

令 n 为数组长度。

- 时间复杂度：`possible` 复杂度为 $\log n$ ，主函数复杂度为 $O(n \log n + \log n * \log(\text{MAX-MIN}))$ 。其中 MAX 为最大值，MIN 为数组最小值。除此之外，还进行了排序，因此时间复杂度大约是 $O(n \log n + \log n * \log(\text{MAX-MIN}))$ 。
- 空间复杂度：取决于排序的空间消耗

力扣的小伙伴可以[关注我](#)，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时会一一查看回答。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 40K 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。

题目地址 (822. Kth-Pair-Distance)

<https://binarysearch.com/problems/Kth-Pair-Distance>

入选理由

1. 能力检测二分
2. 和其他专题有联动（哪个专题呢？）

标签

- 二分

难度

- 困难

题目描述

```
Given a list of integers nums and an integer k, return the

Constraints

n ≤ 100,000 where n is the length of nums
Example 1
Input
nums = [1, 5, 3, 2]
k = 3
Output
2
Explanation
Here are all the pair distances:

abs(1 - 5) = 4
abs(1 - 3) = 2
abs(1 - 2) = 1
abs(5 - 3) = 2
abs(5 - 2) = 3
abs(3 - 2) = 1
Sorted in ascending order we have [1, 1, 2, 2, 3, 4].
```

前置知识

- 排序
- 二分法

堆（超时）

思路

堆很适合动态求极值。我在堆的专题中也说了，使用固定堆可求第 k 大的或者第 k 小的数。这道题是求第 k 小的绝对值差。于是可将所有决定值差动态加入到大顶堆中，并保持堆的大小为 k 不变。这样堆顶的就是第 k 小的绝对值差啦。

其实也可用小顶堆保存所有的绝对值差，然后弹出 k 次，最后一次弹出的就是第 k 小的绝对值差啦。

可惜的是，不管使用哪种方法都无法通过。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def solve(self, A, k):
        A.sort()
        h = [(A[i] - A[i-1], i-1, i) for i in range(1, len(A))]
        heapq.heapify(h)

        while True:
            top, i, j = heapq.heappop(h)
            if not k: return top
            k -= 1
            if j + 1 < len(A): heapq.heappush(h, (A[j+1] -
```

二分法

思路

这道题是典型的计数二分。

计数二分基本就是求第 k 大（或者第 k 小）的数。其核心思想是找到一个数 x ，使得小于等于 x 的数恰好有 k 个。

不能看出，有可能答案不止一个

对应到这道题来说就是找到一个绝对值差 $diff$ ，使得绝对值差小于等于 $diff$ 的恰好有 k 个。

这种类型是否可用二分解决的关键在于：

如果小于等于 $diff$ 的数恰好有 p 个：

1. p 小于 k ，那么可舍弃一半解空间
2. p 大于 k ，同样可舍弃一半解空间

无论如何，我们都可以舍弃一半的解空间。简单来说，就是让未知世界无机可乘。无论如何我都可以舍弃一半。

回到这道题，如果小于等于 $diff$ 的绝对值差有大于 k 个，那么 $diff$ 有点大了，也就是说可以舍弃大于等于 $diff$ 的所有值。反之也是类似，具体大家看代码吧。

最后只剩下两个问题：

- 确定解空间上下界
- 如果计算小于等于 $diff$ 的有即可

第一个问题：下界是 0，上界是 $\max(nums) - \min(min)$ 。

第二个问题：可以使用双指针一次遍历解决。大家可以回忆趁此机会回忆一下双指针。具体地，首先对数组排序，然后使用右指针 j 和左指针 i 。

如果 $nums[j] - nums[i]$ 大于 $diff$ ，我们收缩 i 直到 $nums[j] - nums[i] \leq diff$ 。这个时候，我们就可计算出以索引 j 结尾的绝对值差小于等于 $diff$ 的个数，个数就是 $j - i$ 。我们可以使用滑动窗口技巧分别计算所有的 j 的个数，并将其累加起来就是答案。

代码

代码支持：Python3

Python3 Code:

```

class Solution:
    def solve(self, A, k):
        A.sort()
        def count_not_greater(diff):
            i = ans = 0
            for j in range(1, len(A)):
                while A[j] - A[i] > diff:
                    i += 1
                ans += j - i
            return ans
        l, r = 0, A[-1] - A[0]
        k += 1 # zero based -> one based
        while l <= r:
            mid = (l + r) // 2
            if count_not_greater(mid) >= k:
                r = mid - 1
            else:
                l = mid + 1
        return l

```

复杂度分析

令 n 为数组长度。

- 时间复杂度：由于进行了排序，因此时间复杂度大约是 $O(n \log n)$
- 空间复杂度：取决于排序的空间消耗

思考

- 解空间的值并不都是原数组的差值 (diff)，那么二分能够保证答案的 diff （代码中最后返回的 l ）一定存在于原数组中么？提示：我们使用的是最左二分。

力扣的小伙伴可以[关注我](#)，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时会一一查看回答。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 40K 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。

题目地址 (778. 水位上升的泳池中游泳)

<https://leetcode-cn.com/problems/swim-in-rising-water>

入选理由

1. 难度是 hard，适合做压轴。
2. 这是一个新的二分题型 - **DFS + 二分**，类似的题目有很多，比如第 1439 题。这种题套路都很像，难度其实也不算大。

标签

- 二分

难度

- 困难

题目描述

在一个 $N \times N$ 的坐标方格 grid 中，每一个方格的值 $grid[i][j]$ 表示在位置 (i,j) 的平台高度。

现在开始下雨了。当时间为 t 时，此时雨水导致水池中任意位置的水位为 t 。你可以从一个平台游向四周相邻的任意一个平台，但是前提是此时水位必须同时淹没这两个平台。假定你可以瞬间移动无限距离，也就是默认在方格内部游动是不耗时的。当然，在你游泳的时候你必须待在坐标方格里面。

你从坐标方格的左上平台 $(0, 0)$ 出发。最少耗时多久你才能到达坐标方格的右下平台 $(N-1, N-1)$ ？

示例 1:

输入: $[[0,2],[1,3]]$ 输出: 3 解释: 时间为 0 时，你位于坐标方格的位置为 $(0, 0)$ 。此时你不能游向任意方向，因为四个相邻方向平台的高度都大于当前时间为 0 时的水位。

等时间到达 3 时，你才可以游向平台 $(1, 1)$ 。因为此时的水位是 3，坐标方格中的平台没有比水位 3 更高的，所以你可以游向坐标方格中的任意位置
示例 2:

989. 数组形式的整数加法

```
输入: [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20],  
[10,9,8,7,6]] 输出: 16 解释: 0 1 2 3 4 24 23 22 21 5 12 13 14 15 16 11 17  
18 19 20 10 9 8 7 6
```

最终的路线用加粗进行了标记。我们必须等到时间为 16，此时才能保证平台 (0, 0) 和 (4, 4) 是连通的

提示:

$2 \leq N \leq 50$. $\text{grid}[i][j]$ 位于区间 $[0, \dots, N^2 - 1]$ 内。

前置知识

- [DFS](#)
- [二分](#)

思路

二分查找在 CP 中的一个常见应用是二分答案。在这一类题目中，我们往往需要求出满足条件的最大值或最小值。如果这一取值和条件的成立与否之间满足有序性，我们就可以通过对整个定义域进行二分查找，来找到我们需要的最值。

很明显，这道题的答案是一个连续的空间，从 0 到 $\text{max}(\text{grid})$ ，其中 $\text{max}(\text{grid})$ 表示 grid 中的最大值。

因此一个简单的思路是一个个试。实际上，如果 x 不可以，那么小于 x 的所有值都是不可以的，这正是本题的突破口。基于此，我们可使用讲义中的**最左二分模板**解决。

伪代码:

```
def test(x):  
    pass  
while l <= r:  
    mid = (l + r) // 2  
    if test(mid, 0, 0):  
        r = mid - 1  
    else:  
        l = mid + 1  
return l
```

这个模板会在很多二分中使用。比如典型的计数型二分，典型的就是计算小于等于 x 的有多少，然后根据答案更新搜索区间。

明确了这点，剩下要做的就是完成 test 了。其实这个就是一个普通的二维网格 dfs，我们从 (0,0) 开始在一个二维网格中搜索，直到无法继续或达到 (N-1,N-1)，如果可以达到 (N-1,N-1)，我们返回 true，否则返回 False 即可。对二维网格的 DFS 不熟悉的同学可以看下我之前写的小岛专题

代码

```
class Solution:
    def swimInWater(self, grid: List[List[int]]) -> int:
        l, r = 0, max([max(vec) for vec in grid])
        seen = set()

        def test(mid, x, y):
            if x > len(grid) - 1 or x < 0 or y > len(grid[0]) - 1:
                return False
            if grid[x][y] > mid:
                return False
            if (x, y) == (len(grid) - 1, len(grid[0]) - 1):
                return True
            if (x, y) in seen:
                return False
            seen.add((x, y))
            ans = test(mid, x + 1, y) or test(mid, x - 1, y) or test(mid, x, y + 1) or test(mid, x, y - 1)
            return ans
        while l <= r:
            mid = (l + r) // 2
            if test(mid, 0, 0):
                r = mid - 1
            else:
                l = mid + 1
            seen = set()
        return l
```

复杂度分析

- 时间复杂度：\$O(N \log M)\$，其中 \$M\$ 为 grid 中的最大值，\$N\$ 为 grid 的总大小。
- 空间复杂度：\$O(N)\$，其中 \$N\$ 为 grid 的总大小。

题目地址(1456. 定长子串中元音的最大数目)

<https://leetcode-cn.com/problems/maximum-number-of-vowels-in-a-substring-of-given-length>

入选理由

1. 滑动窗口第一道题，难度不算大，而且是非常常规的题目。

标签

- 滑动窗口

难度

- 中等

题目描述

989. 数组形式的整数加法

给你字符串 s 和整数 k 。

请返回字符串 s 中长度为 k 的单个子字符串中可能包含的最大元音字母数。

英文中的 元音字母 为 (a, e, i, o, u)。

示例 1:

输入: $s = "abciiidef"$, $k = 3$

输出: 3

解释: 子字符串 "iii" 包含 3 个元音字母。

示例 2:

输入: $s = "aeiou"$, $k = 2$

输出: 2

解释: 任意长度为 2 的子字符串都包含 2 个元音字母。

示例 3:

输入: $s = "leetcode"$, $k = 3$

输出: 2

解释: "lee"、"eet" 和 "ode" 都包含 2 个元音字母。

示例 4:

输入: $s = "rhythms"$, $k = 4$

输出: 0

解释: 字符串 s 中不含任何元音字母。

示例 5:

输入: $s = "tryhard"$, $k = 4$

输出: 1

提示:

$1 \leq s.length \leq 10^5$

s 由小写英文字母组成

$1 \leq k \leq s.length$

前置知识

- 滑动窗口
- 哈希表

思路

拿这个题作为本专题第一道滑动窗口练手题再合适不过了，题目直观清晰。

- 元音字母有五个，为了避免我们总要写 if 啊 switch 啊这种，我们可以用个哈希表来存着方便后续查找是否存在。
- 题目要求我们找出所有 k 长度子串中可能包含的最大元音字母数，那我们遍历一遍所有长度为 k 的子串不就知道啦 → 暴力，上代码

```
public int maxVowels(String s, int k) {

    if (s == null || s.length() < k)
        return 0;

    int res = 0;
    Set<Character> set = new HashSet<>() {{
        add('a'); add('e'); add('i'); add('o'); add('u');
    }};

    for (int i = 0; i < s.length() - k + 1; i++) {

        String sub = s.substring(i, i + k);
        int count = 0;

        for (int j = 0; j < sub.length(); j++)
            if (set.contains(sub.charAt(j)))
                count++;

        res = Math.max(res, count);
    }

    return res;
}
```

很直观，但是提交会发现 TLE 了，我们也不难发现复杂度为 $O((N - K + 1) * K)$ ，有什么优化方法呢？其实也容易想到：

- 利用前缀和，只不过我们前缀和数组元素 i 存的是子串 $0..i$ 的元音字母个数，这样再遍历一遍前缀和数组就可以求出结果 → 前缀和方案（有兴趣可以自行实现）
- 其实我们完全没有必要去构建这个前缀和数组，我们维护一个窗口大小为 k 的滑窗即可，每移动一次可以归纳为：
 - 窗口左端弹出一个字符（删除步）
 - 若删除了元音则计数器-1（更新步）
 - 窗口右端加进来一个字符（添加步）

989. 数组形式的整数加法

- 若添加的字符是元音则计数器+1 (更新步)
- 这样就得到了 → 滑动窗口解决方案

前面的部分也提到了前缀和，你能试着总结一下和前缀和相关的考点有哪些么？

代码

代码支持：Python, Java, JS, CPP

Python Code:

```
class Solution:  
    def maxVowels(self, s: str, k: int) -> int:  
        res = 0  
        temp = 0  
        vowels = set(['a','e','i','o','u'])  
        for i in range(k):  
            if s[i] in vowels:  
                res += 1  
        if res==k: return k  
        temp = res  
        for i in range(k, len(s)):  
            if s[i] in vowels:  
                temp += 1  
            if s[i-k] in vowels:  
                temp -= 1  
            res = max(temp, res)  
            if res ==k: return k  
        return res
```

Java Code:

989. 数组形式的整数加法

```
public int maxVowels(String s, int k) {

    if (s == null || s.length() < k)
        return 0;

    int res = 0;
    Set<Character> set = new HashSet<>(){{
        add('a');add('e');add('i');add('o');add('u');
    }};

    // init
    for (int i = 0; i < k; i++)
        if (set.contains(s.charAt(i)))
            res++;

    int cur = res;
    for (int i = 1; i < s.length() - k + 1; i++) {

        if (set.contains(s.charAt(i - 1)))
            cur--;
        if (set.contains(s.charAt(i + k - 1)))
            cur++;

        res = Math.max(res, cur);
    }

    return res;
}
```

JS Code:

989. 数组形式的整数加法

```
var maxVowels = function (s, k) {
    const dict = new Set(["a", "e", "i", "o", "u"]);
    let ret = 0;
    for (let i = 0; i < k; i++) {
        if (dict.has(s[i])) ret++;
    }

    let temp = ret;
    for (let i = k, j = 0; i < s.length; i++, j++) {
        if (dict.has(s[i])) temp++;
        if (dict.has(s[j])) temp--;

        ret = Math.max(temp, ret);
    }

    return ret;
};
```

CPP Code:

```
class Solution {
public:
    int maxVowels(string s, int k) {
        unordered_set<char> vowels = {'a', 'e', 'i', 'o', 'u'};
        int cnt = 0, ans = 0;
        int n = s.size();
        for (int i = 0; i < n; i++) {
            if (i >= k) cnt -= vowels.count(s[i - k]);
            cnt += vowels.count(s[i]);
            ans = max(ans, cnt);
        }
        return ans;
    }
};
```

复杂度分析

- 时间复杂度: $O(n)$, n 为字符串长度
- 空间复杂度: $O(1)$

题目地址(837. 新 21 点)

<https://leetcode-cn.com/problems/new-21-game>

入选理由

1. dp + 滑动窗口的结合。让大家理解滑动窗口是如何和其他技巧联系的。之前我也出了二分 + dfs 这种联动性题目，这种题目对于你学习知识很有帮助。

标签

- 二分
- 滑动窗口

难度

- 中等

题目描述

爱丽丝参与一个大致基于纸牌游戏“21点”规则的游戏，描述如下：

爱丽丝以 0 分开始，并在她的得分少于 K 分时抽取数字。抽取时，她从 $[1$

当爱丽丝获得不少于 K 分时，她就停止抽取数字。爱丽丝的分数不超过 N 的

示例 1：

输入: $N = 10, K = 1, W = 10$

输出: 1.00000

说明：爱丽丝得到一张卡，然后停止。

示例 2：

输入: $N = 6, K = 1, W = 10$

输出: 0.60000

说明：爱丽丝得到一张卡，然后停止。

在 $W = 10$ 的 6 种可能下，她的得分不超过 $N = 6$ 分。

示例 3：

输入: $N = 21, K = 17, W = 10$

输出: 0.73278

提示：

$0 \leq K \leq N \leq 10000$

$1 \leq W \leq 10000$

如果答案与正确答案的误差不超过 10^{-5} ，则该答案将被视为正确答案通过。

此问题的判断限制时间已经减少。

前置知识

- 动态规划
- 滑动窗口

思路

我们倒着往前思考，由于每次选择的数字范围都是 $[1, W]$ ，并且如果当前分数大于 K 就会停止，因此最后一次抽取的时候分数一定是小于 K 的，并且抽取的数字 + 当前的分数要大于等于 K 。

而我们要求的就是分数大于等于 K 的这些情况中不大于 N 的概率，即满足 $K \leq x \leq N$ 中的概率，其中 x 为分数。也就是求满足 $K \leq x \leq N$ 的总个数再除以 W ，因为一次抽取有 W 种可能，分别是 $[1, W]$ ，且概率均相等，都为 $1/W$ 。

如果用 $dp[i]$ 表示当前分数为 i 的情况下，爱丽丝的分数不超过 N 的概率。那么：

```
dp[i] = sum(dp[i + j] for j in range(1, W + 1)) / W
```

其实也就是说 $dp[i] = (dp[i+1] + dp[i+2] + \dots + dp[i+W]) / W$ ，这就是动态转移方程。

由于我们的转移方程的 $dp[i]$ 依赖 $dp[i+x]$ ，其中 x 属于 $[1, W]$ ，也就是说我们需要从后往前遍历，这样才能保证结果的正确性。

另外，我们需要初始化 $[K, \min(K+W-1, N)]$ 范围内的 dp 为 1，这是我们的边界条件，有了它们的存在，才能推动算法动态运算下去，他们的作用就像是给一个在高山上的雪球一个推力，帮助雪球滚落。

基于此，我们可以写出下面的代码。

Python3 代码：

```
class Solution:
    def new21Game(self, N: int, K: int, W: int) -> float:
        dp = [0] * (K + W)
        for i in range(K, K + W):
            if i <= N:
                dp[i] = 1

        for i in range(K - 1, -1, -1):
            dp[i] = sum(dp[i + j] for j in range(1, W + 1))
        return dp[0]
```

复杂度分析

- 时间复杂度：\$O(KW)\$
- 空间复杂度：\$O(K+W)\$

时间复杂度是 \$O(KW)\$，代入题目的限制条件 $0 \leq K \leq N \leq 10000$, $1 \leq W \leq 10000$ ，得到总的计算次数大约是 10^8 ，大于 10^7 ，因此会超时。

不明白为什么是 10^7 ？力扣加加公众号看《来和大家聊聊我是如何刷题的（第三弹）》

989. 数组形式的整数加法

通过观察发现，每次 sum 我们变化的其实仅仅是左右两侧的数，中间的是不会变的。这不就是我们讲义中的滑动窗口使用场景么？

也就是说，我们只需要预先计算窗口的总和，之后更新窗口的时候都减去窗口右侧过期的数，再加上窗口左侧刚进去的数就行了。这种算法的时间复杂度是 $O(W + K)$

代码

代码支持：Python3, JS, Java, CPP

```
class Solution:
    def new21Game(self, N: int, K: int, W: int) -> float:
        # 滑动窗口优化（固定窗口大小为 W 的滑动窗口）
        dp = [0] * (K + W)
        win_sum = 0
        for i in range(K, K + W):
            if i <= N:
                dp[i] = 1
            win_sum += dp[i]

        for i in range(K - 1, -1, -1):
            dp[i] = win_sum / W
            win_sum += dp[i] - dp[i + W]
        return dp[0]
```

JS Code:

```
var new21Game = function (n, k, maxPts) {
    const dp = new Array(k + maxPts + 2).fill(0);

    let windowSum = 0;
    for (let i = k; i < k + maxPts; i++) {
        if (i <= n) dp[i] = 1;
        windowSum += dp[i];
    }

    for (let i = k - 1; i >= 0; i--) {
        dp[i] = windowSum / maxPts;
        windowSum -= dp[i + maxPts];
        windowSum += dp[i];
    }

    return dp[0];
};
```

989. 数组形式的整数加法

Java Code:

```
class Solution {
    public double new21Game(int N, int K, int W) {
        if (K == 0 || N >= K + W) return 1;
        double dp[] = new double[N + 1], wsum = 1, res = 0;
        dp[0] = 1;
        for (int i = 1; i <= N; ++i) {
            dp[i] = wsum / W;
            if (i < K) wsum += dp[i];
            else res += dp[i];
            if (i - W >= 0) wsum -= dp[i - W];
        }
        return res;
    }
}
```

CPP Code:

```
class Solution {
public:
    double new21Game(int N, int K, int W) {
        if (K == 0) return 1.0;
        vector<double> dp(K + W);
        for (int i = K; i <= N && i <= K + W - 1; i++) dp[i] = 1.0 * min(N - K + 1, W) / W;
        for (int i = K - 1; i >= 0; i--) dp[i] = dp[i + 1];
        return dp[0];
    }
};
```

复杂度分析

- 时间复杂度: $O(K+W)$
- 空间复杂度: $O(K+W)$

题目地址（438. 找到字符串中所有字母异位词）

<https://leetcode-cn.com/problems/find-all-anagrams-in-a-string/>

入选理由

1. 依然是经典的滑动窗口，相信你做完这道题应该对滑动窗口的套路有了一定的理解了。掌握知识的最笨但却好用的方法就是重复重复再重复。

标签

- 滑动窗口

难度

- 中等

题目描述

给定一个字符串 s 和一个非空字符串 p , 找到 s 中所有是 p 的字母异位词的子串。
字符串只包含小写英文字母, 并且字符串 s 和 p 的长度都不超过 20100。

说明:

字母异位词指字母相同, 但排列不同的字符串。

不考虑答案输出的顺序。

示例 1:

输入:

$s: "cbaebabacd"$ $p: "abc"$

输出:

[0, 6]

解释:

起始索引等于 0 的子串是 "cba", 它是 "abc" 的字母异位词。

起始索引等于 6 的子串是 "bac", 它是 "abc" 的字母异位词。

示例 2:

输入:

$s: "abab"$ $p: "ab"$

输出:

[0, 1, 2]

解释:

起始索引等于 0 的子串是 "ab", 它是 "ab" 的字母异位词。

起始索引等于 1 的子串是 "ba", 它是 "ab" 的字母异位词。

起始索引等于 2 的子串是 "ab", 它是 "ab" 的字母异位词。

前置知识

- Sliding Window
- 哈希表

思路

咳咳, 暴力题解俺就不写了哈, 因为和昨天基本一致

来分析一下, 首先题中说找到 s 中所有是 p 的字母异位词的字串, 就这句话, 就包含了如下两个重要信息:

- 找到符合要求的子串长度都是 p

- 何为字母异位词？也就是我们不关心 p 这个串的顺序，只关心字母是否出现以及出现的次数，这种问题解决方案一般有两种，一种是利用排序强制顺序，另一种就是用哈希表的方法。

这么一抽象，是不是和昨天那个题很相似呢？那么问题的关键就是：

- 如何构建滑窗
- 如何更新状态，也即如何存储 p 串及更新窗口信息

针对问题 1 很容易，因为是长度固定为 p 的滑动窗口，而针对如何存储 p 串这个问题，我们可以考虑用桶来装，这个桶既可以用 26 个元素的数组（作用其实也是哈希表）也可以用哈希表

那么我们解决方案就很明朗了：

- 初始化个滑窗
- 不断移动该固定窗口，并用一个 rest 变量来记录剩余待匹配字符的个数
- 只要当前窗口符合要求，即把窗口左指针下标添加到结果集合中去。

代码

代码支持：Java, Python3

Java Code:

989. 数组形式的整数加法

```
public List<Integer> findAnagrams(String s, String p) {  
  
    List<Integer> res = new LinkedList<>();  
    if (s == null || p == null || s.length() < p.length())  
        return res;  
  
    int[] ch = new int[26];  
    //统计p串字符个数  
    for (char c : p.toCharArray())  
        ch[c - 'a']++;  
    //把窗口扩成p串的长度  
    int start = 0, end = 0, rest = p.length();  
    for (; end < p.length(); end++) {  
        char temp = s.charAt(end);  
        ch[temp - 'a']--;  
        if (ch[temp - 'a'] >= 0)  
            rest--;  
    }  
  
    if (rest == 0)  
        res.add(0);  
    //开始一步一步向右移动窗口。  
    while (end < s.length()) {  
        //左边的拿出来一个并更新状态  
        char temp = s.charAt(start);  
        if (ch[temp - 'a'] >= 0)  
            rest++;  
        ch[temp - 'a']++;  
        start++;  
        //右边的拿进来一个并更新状态  
        temp = s.charAt(end);  
        ch[temp - 'a']--;  
        if (ch[temp - 'a'] >= 0)  
            rest--;  
        end++;  
        // 状态合法就存到结果集合  
        if (rest == 0)  
            res.add(start);  
    }  
  
    return res;  
}
```

Python 解法具体做法稍有一点不同，没有使用 rest 变量，而是直接取的哈希表的长度。其中 哈希表的 key 是字符，value 是窗口内字符出现次数。这样当 value 为 0 时，我们移除 key，这样当哈希表容量为 0，说明我们找到了一个异位词。

989. 数组形式的整数加法

Python3 Code:

```
class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        target = collections.Counter(p)
        ans = []
        for i in range(len(s)):
            if i >= len(p):
                target[s[i - len(p)]] += 1
                if target[s[i - len(p)]] == 0:
                    del target[s[i - len(p)]]
            target[s[i]] -= 1
            if target[s[i]] == 0:
                del target[s[i]]
            if len(target) == 0:
                ans.append(i - len(p) + 1)
        return ans
```

你也可以将窗口封装成一个类进行操作。虽然代码会更长，但是如果你将窗口类看成黑盒，那么逻辑会很简单。

这里我提供一个 Python3 版本的**封装类解法**。

```

class FrequencyDict:
    def __init__(self, s):
        self.d = collections.Counter()
        for char in s:
            self.increment(char)

    def _del_if_zero(self, char):
        if self.d[char] == 0:
            del self.d[char]

    def is_empty(self):
        return not self.d

    def decrement(self, char):
        self.d[char] -= 1
        self._del_if_zero(char)

    def increment(self, char):
        self.d[char] += 1
        self._del_if_zero(char)

class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        ans = []

        freq = FrequencyDict(p)

        for char in s[:len(p)]:
            freq.decrement(char)

        if freq.is_empty():
            ans.append(0)

        for i in range(len(p), len(s)):
            start, end = s[i - len(p)], s[i]
            freq.increment(start)
            freq.decrement(end)
            if freq.is_empty():
                ans.append(i - len(p) + 1)

        return ans

```

复杂度分析

令 s 的长度为 n 。

- 时间复杂度: $O(n)$

989. 数组形式的整数加法

- 空间复杂度：虽然我们使用了数组（或者哈希表）存储计数信息，但是大小不会超过 26，因此空间复杂度为 $O(1)$ 。

入选理由

- 难度变大了哦，来看看滑动窗口的 hard 题到底 hard 在哪。

标签

- 滑动窗口

难度

- 困难

题目地址(76. 最小覆盖子串)

<https://leetcode-cn.com/problems/minimum-window-substring>

题目描述

给你一个字符串 S、一个字符串 T 。请你设计一种算法，可以在 $O(n)$ 的时间复杂度内找到 S 中恰好包含 T 所有字符的最小子串。

示例：

输入：S = "ADOBECODEBANC"，T = "ABC"

输出："BANC"

提示：

如果 S 中不存这样的子串，则返回空字符串 ""。

如果 S 中存在这样的子串，我们保证它是唯一的答案。

前置知识

- Sliding Window
- 哈希表

思路

读完该题，是否发现和前一天的题目有些类似呢，前一天的那个说法叫异位词，今天这个直接说包含 T 的所有字符，意思其实是一样的，那不一样的在哪呢？

- 这次的窗口长度并不固定为 T 的长度，实际窗口大小是 $\geq T.length$ 的
- 这次输出的是最小子串，也就是长度最小的子串，因此我们要维护一个 min，代表当前符合要求的子串长度，遇到更短的，则进行更新。

针对上面的，我们开始分析讲义中所讲的滑窗流程的核心三步，在分析之前，再贴一遍简单的伪代码：

```

while 右边界 < 合法条件:
    # 右边界扩张
    window右边界+1
    更新状态信息
    # 左边界收缩
    while 符合收缩条件:
        window左边界+1
        更新状态信息
    
```

按上边的模版一步步分析：

- 右边界<合法条件：条件自然是 right 不能超过字符串长度
 - 右端 add：将当前字符加入窗口
 - 更新 update：当前加入窗口的字符是否 match 了 T 的字符集，match 了则更新状态（这里需要注意的是，如果当前 match 的字符已经够了，则只更新哈希表中的状态而不更新 match 计数器）
 - 循环左端 delete，目的是尽量缩短窗口大小达到题目最小的要求：符合收缩条件的前提自然是当前已经 match 了 T 的所有字符，然后不断缩小窗口，移除左端字符。
 - 更新 update：当前移除的字符若在 T 字符集中则要更新状态，方式同上。
- 记得记录一下最短子串对应窗口的左右指针方便后续返回结果。

按照上述大致分析流程得到如下代码：

代码

989. 数组形式的整数加法

```
public String minWindow(String s, String t) {  
  
    Map<Character, Integer> map = new HashMap<>();  
  
    int num = t.length();  
  
    for (int i = 0; i < num; i++)  
        map.put(t.charAt(i), map.getOrDefault(t.charAt(i),  
  
            int len = Integer.MAX_VALUE, match = 0, resLeft = 0, re  
  
            int left = 0, right = 0;  
  
    while (right < s.length()) {  
  
        char ch = s.charAt(right);  
  
        if (map.containsKey(ch)) {  
  
            int val = map.get(ch) + 1;  
            if (val <= 0)  
                match++;  
            map.put(ch, val);  
        }  
  
        while (match == num) {  
  
            if (right - left + 1 < len) {  
  
                len = right - left + 1;  
                resLeft = left;  
                resRight = right;  
            }  
  
            char c = s.charAt(left);  
            if (map.containsKey(c)) {  
  
                int val = map.get(c) - 1;  
                if (val < 0)  
                    match--;  
                map.put(c, val);  
            }  
  
            left++;  
        }  
  
        right++;  
    }  
}
```

989. 数组形式的整数加法

```
    return len == Integer.MAX_VALUE ? "" : s.substring(res)
```

复杂度分析

- 时间复杂度: $O(N + K)$, N 为 S 串长度, K 为 T 串长度
- 空间复杂度: $O(S)$, 其中 S 为 T 字符集元素个数

题目地址 (Number of Operations to Decrement Target to Zero)

<https://binarysearch.com/problems/Number-of-Operations-to-Decrement-Target-to-Zero>

入选理由

1. 这道题太常见了，以至于力扣就有一个类似的。<https://leetcode-cn.com/problems/maximum-points-you-can-obtain-from-cards/>

标签

- 滑动窗口

难度

- 中等

题目描述

You are given a list of positive integers nums and an integer target . Consider an operation where we remove a number v from either the front or the back of nums and decrement target by v .

Return the minimum number of operations required to decrement target to zero. If it's not possible, return -1.

Constraints

$n \leq 100,000$ where n is the length of nums Example 1 Input $\text{nums} = [3, 1, 1, 2, 5, 1, 1]$ target = 7 Output 3 Explanation We can remove 1, 1 and 5 from the back to decrement target to zero.

Example 2 Input $\text{nums} = [2, 4]$ target = 7 Output -1 Explanation There's no way to decrement target = 7 to zero.

前置知识

- 二分法
- 滑动窗口

二分法

思路

这道题的意思是给你一个数组，你只可以移除数组两端的数。求最小移除次数，使得移除的数字和为 target。

我们可以反向思考，删除和为 target 的若干数字等价于保留若干和为 $\text{sum}(A) - \text{target}$ 的数。这样问题就转化为 **求连续子数组和为 $\text{sum}(A) - \text{target}$ 的最长子数组**。这种问题可以使用滑动窗口来解决。

注意抽象后的问题有“连续”关键字，就应该想到可能会用到滑动窗口优化。具体能不能用再根据题目信息做二次判断。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def solve(self, A, target):
        if not A and not target: return 0
        target = sum(A) - target
        ans = len(A) + 1
        i = t = 0

        for j in range(len(A)):
            t += A[j]
            while i <= j and t > target:
                t -= A[i]
                i += 1
            if t == target: ans = min(ans, len(A) - (j - i))
        return -1 if ans == len(A) + 1 else ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度: $O(n)$
- 空间复杂度: 1

力扣的小伙伴可以[关注我](#)，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时会一一查看回答。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 40K 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。

989. 数组形式的整数加法

题目地址(401. 二进制手表)

<https://leetcode-cn.com/problems/binary-watch/>

入选理由

1. 搜索篇的题目为：回溯 -> BFS/DFS（BFS 和 DFS 不做区分，因为基本上能用 DFS 的，也能用 BFS，互通的），先从简单的回溯开始。大家注意画图哦

标签

- 回溯

难度

- 简单

题目描述

二进制手表顶部有 4 个 LED 代表 小时 (0-11) ，底部的 6 个 LED 代表每个 LED 代表一个 0 或 1，最低位在右侧。



例如，上面的二进制手表读取“3:25”。

给定一个非负整数 n 代表当前 LED 亮着的数量，返回所有可能的时间。

示例：

输入： $n = 1$

返回： ["1:00", "2:00", "4:00", "8:00", "0:01", "0:02", "0:0"]

提示：

输出的顺序没有要求。

小时不会以零开头，比如“01:00”是不允许的，应为“1:00”。

分钟必须由两位数组成，可能会以零开头，比如“10:2”是无效的，应为“10:02”。超过表示范围（小时 0–11，分钟 0–59）的数据将被舍弃，也就是说不会出现“12:00”或“12:30”。

来源：力扣（LeetCode）

链接：<https://leetcode-cn.com/problems/binary-watch>

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

前置知识

- 笛卡尔积
- 回溯

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

一看题目就是一个笛卡尔积问题。

即给你一个数字 num ，我可以将其分成两部分。其中一部分（不妨设为 a ）给小时，另一部分给分（就是 $num - a$ ）。最终的结果就是 a 能表示的所有小时的集合和 $num - a$ 所能表示的分的集合的笛卡尔积。

用代码表示就是：

```
# 枚举小时
for a in possible_number(i):
    # 小时确定了，分就是 num - i
    for b in possible_number(num - i, True):
        ans.add(str(a) + ":" + str(b).rjust(2, '0'))
```

枚举所有可能的 $(a, num - a)$ 组合即可。

核心代码：

```
for i in range(min(4, num + 1)):
    for a in possible_number(i):
        for b in possible_number(num - i, True):
            ans.add(str(a) + ":" + str(b).rjust(2, '0'))
```

代码

```
class Solution:
    def readBinaryWatch(self, num: int) -> List[str]:
        def possible_number(count, minute=False):
            if count == 0: return [0]
            if minute:
                return filter(lambda a: a < 60, map(sum, combinations(range(1, 12), count)))
            return filter(lambda a: a < 12, map(sum, combinations(range(1, 12), count)))
        ans = set()
        for i in range(min(4, num + 1)):
            for a in possible_number(i):
                for b in possible_number(num - i, True):
                    ans.add(str(a) + ":" + str(b).rjust(2, '0'))
        return list(ans)
```

进一步思考，实际上，我们要找的就是 a 和 b 相加等于 num ，并且 a 和 b 就是二进制表示中 1 的个数。因此可以将逻辑简化为：

```
class Solution:
    def readBinaryWatch(self, num: int) -> List[str]:
        return [str(a) + ":" + str(b).rjust(2, '0') for a :
```

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 37K 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。

题目地址(52. N 皇后 II)

<https://leetcode-cn.com/problems/n-queens-ii/>

入选理由

1. 回溯就两道题（这是第二道），这道题难度比较大。但是只要分析好问题，画好图就不是难事。

标签

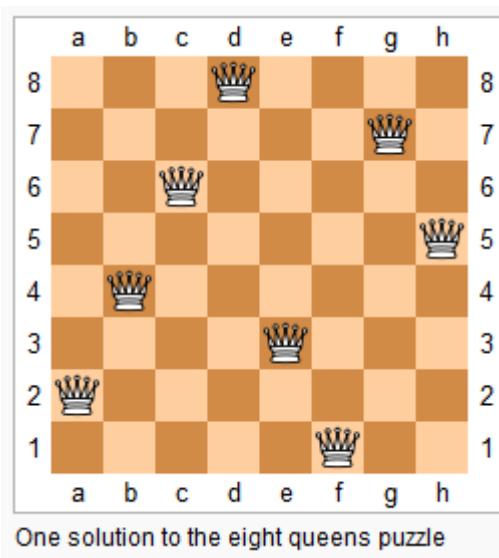
- 回溯

难度

- 困难

题目描述

n 皇后问题研究的是如何将 n 个皇后放置在 $n \times n$ 的棋盘上，并且使皇后彼此之间不能相互攻击。



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) 与已放置过皇后的位置
      dfs(n, row + 1, cols | p, (pie | p) << 1, (na | p) >> 1);
    }
  };
  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: $O(N)$

989. 数组形式的整数加法

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 37K 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(695. 岛屿的最大面积)

<https://leetcode-cn.com/problems/max-area-of-island/>

入选理由

1. BFS 和 DFS 大家都可以试试，一般都是通的

标签

- BFS
- DFS

难度

- 中等

题目描述

给定一个包含了一些 0 和 1 的非空二维数组 grid。
一个 岛屿 是由一些相邻的 1 (代表土地) 构成的组合，
这里的「相邻」要求两个 1 必须在水平或者竖直方向上相邻。
你可以假设 grid 的四个边缘都被 0 (代表水) 包围着。
找到给定的二维数组中最大的岛屿面积。(如果没有岛屿，则返回面积为 0)

示例 1：

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

对于上面这个给定矩阵应返回 6。注意答案不应该是 11，因为岛屿只能包含水

示例 2：

```
[[0,0,0,0,0,0,0,0]]
```

对于上面这个给定的矩阵，返回 0。

注意：给定的矩阵grid 的长度和宽度都不超过 50

公司

- 字节跳动

思路

和 [200. 岛屿数](#) 思路一样，只不过 200 是求小岛个数，这个是求小岛最大面积，这也就是多一个变量记录一下的事情。

start

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

[200] Number of Islands

这道题目仍然可以采用原位修改的方式避免记录 `visited` 的开销。我们的做法是将 `grid[i][j] = 0`, 需要注意的是, 我们无需重新将 `grid[i][j] = 1`, 因为题目没有这个要求。另外如果你这么做的话, 也会产生 bug, 比如:

```
111  
111
```

上面加粗的 1, 如果在遍历了上下左右邻居之后, 将 0, 重新变成 1。那么就会被重复计算。如下, 粗体上方的 1 就会被计算多次

```
111  
101
```

代码

- 语言支持: Python

```
class Solution:
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        m = len(grid)
        if m == 0: return 0
        n = len(grid[0])
        ans = 0
        def dfs(i, j):
            if i < 0 or i >= m or j < 0 or j >= n: return 0
            if grid[i][j] == 0: return 0
            grid[i][j] = 0
            top = dfs(i + 1, j)
            bottom = dfs(i - 1, j)
            left = dfs(i, j - 1)
            right = dfs(i, j + 1)
            return 1 + sum([top, bottom, left, right])
        for i in range(m):
            for j in range(n):
                ans = max(ans, dfs(i, j))
        return ans
```

复杂度分析

- 时间复杂度: $O(m \cdot n)$
- 空间复杂度: $O(m \cdot n)$

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 37K 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(1162. 地图分析)

<https://leetcode-cn.com/problems/as-far-from-land-as-possible/>

入选理由

- 继续来一道常规的搜索，最后两天我们再来点不一样的

标签

- BFS

难度

- 中等

题目描述

你现在手里有一份大小为 $N \times N$ 的 网格 grid，上面的每个 单元格 都用 0 表示海洋，1 表示陆地，请你找出一个海洋单元格，这个海洋单元格到离它最近的陆地单元格的距离是最大的。

我们这里说的距离是「曼哈顿距离」（Manhattan Distance）：
 (x_0, y_0) 和 (x_1, y_1) 这两个单元格之间的距离是 $|x_0 - x_1| + |y_0 - y_1|$ 。

如果网格上只有陆地或者海洋，请返回 -1。

示例 1：

1	0	1
0	0	0
1	0	1

989. 数组形式的整数加法

输入: [[1,0,1],[0,0,0],[1,0,1]]
输出: 2
解释:
海洋单元格 (1, 1) 和所有陆地单元格之间的距离都达到最大，最大距离为 2
示例 2:

1	0	0
0	0	0
0	0	0

输入: [[1,0,0],[0,0,0],[0,0,0]]
输出: 4
解释:
海洋单元格 (2, 2) 和所有陆地单元格之间的距离都达到最大，最大距离为 4

提示:

`1 <= grid.length == grid[0].length <= 100`
`grid[i][j]` 不是 0 就是 1

来源: 力扣 (LeetCode)

链接: <https://leetcode-cn.com/problems/as-far-from-land-as-possible/>
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

公司

- 字节跳动

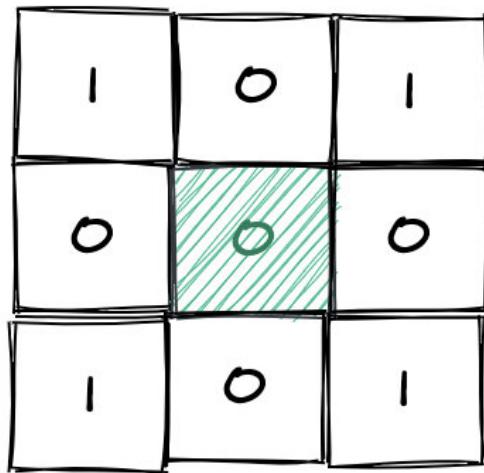
思路

这里我们继续使用[上面两道题的套路](#)，即不用 `visited`，而是原地修改。由于这道题求解的是最远的距离，而距离我们可以使用 BFS 来做。算法：

- 对于每一个海洋，我们都向四周扩展，寻找最近的陆地，每次扩展 `steps` 加 1。
- 如果找到了陆地，我们返回 `steps`。
- 我们的目标就是所有 `steps` 中的最大值。

989. 数组形式的整数加法

实际上上面算法有很多重复计算，如图中间绿色的区域，向外扩展的时候，如果其周边四个海洋的距离已经计算出来了，那么没必要扩展到陆地。实际上只需要扩展到周边的四个海洋格子就好了，其距离陆地的最近距离就是 $1 + \text{周边四个格子中到达陆地的最小距离}$ 。



一种优化的方式是将所有陆地加入队列，而不是海洋。陆地不断扩展到海洋，每扩展一次就 `steps` 加 1，直到无法扩展位置。最终返回 `steps` 即可。

图解：

989. 数组形式的整数加法

1	0	1
0	0	0
1	0	1

1	0	1
0	0	0
1	0	1

steps 0

1	-1	1
-1	0	-1
1	-1	1

steps 1

1	-1	1
-1	-1	-1
1	-1	1

steps 2

代码

- 语言支持: Python

```
class Solution:  
    def maxDistance(self, grid: List[List[int]]) -> int:  
        n = len(grid)  
        steps = -1  
        queue = collections.deque([(i, j) for i in range(n)  
                                    if len(queue) == 0 or len(queue) == n ** 2: return  
        while len(queue) > 0:  
            for _ in range(len(queue)):  
                x, y = queue.popleft(0)  
                for xi, yj in [(x + 1, y), (x - 1, y), (x,  
                                                if xi >= 0 and xi < n and yj >= 0 and yj < n:  
                                                    queue.append((xi, yj))  
                                                    grid[xi][yj] = -1  
            steps += 1  
  
        return steps
```

复杂度分析

- 时间复杂度: $O(N^2)$
- 空间复杂度: $O(N^2)$

入选理由

1. 正向你会了，那么反向搜索你会么？有时候反向思考，有意向不到的收获。

标签

- BFS

难度

- 中等

题目地址 (**Shortest-Cycle-Containing-Target-Node**)

<https://binarysearch.com/problems/Shortest-Cycle-Containing-Target-Node>

题目描述

You are given a two-dimensional list of integers graph representing a directed graph as an adjacency list. You are also given an integer target.

Return the length of a shortest cycle that contains target. If a solution does not exist, return -1.

Constraints

$n, m \leq 250$ where n and m are the number of rows and columns in graph

前置知识

- BFS

思路

题目让你返回最短的环。

回想一下，我们遍历图的时候是如何防止环的产生的。我们通常使用一个集合 `visited`，记录已经访问过的节点。每次遇到一个新的节点，我们都检查其是否在 `visited` 中存在。如果存在，我们就不再进行处理，这样就避

免了环的影响。

因此检测环的存在也是一样的思路。我们也可使用一个集合记录访问过的节点。同时为了记录最短的环，我们不妨进行 BFS，这样当我们遇到一个环的时候，就一定最短的，直接返回 BFS 遍历的层即可，这提示我们使用带层信息的 BFS。

而题目要求的是返回最短的包含 target 的环。与其从各个点作为搜索起点，并在检测到环的时候再判断环中是否有 target，我们不妨从 target 开始搜索，这样检测到环就不用判断环中是否有 target 了。

下面代码使用的就是一个标准带层的 BFS 模板。

关键点

- 反向搜索，即从 target 开始搜索。而不是从图中所有的节点开始搜。

代码

代码支持 Python3:

```
class Solution:
    def solve(self, graph, target):
        q = collections.deque([target])
        visited = set()
        steps = 0
        while q:
            for i in range(len(q)):
                cur = q.popleft()
                visited.add(cur)
                for neighbor in graph[cur]:
                    if neighbor not in visited:
                        q.append(neighbor)
                    elif neighbor == target:
                        return steps + 1
            steps += 1
        return -1
```

令 v 节点数， e 为边数。

复杂度分析

- 时间复杂度: $O(v + e)$
- 空间复杂度: $O(v)$

入选理由

1. 和哈希以及树专题都有联动，帮助大家复习

标签

- 哈希表
- 树
- BFS

难度

- 中等

题目地址 (Top-View-of-a-Tree)

<https://binarysearch.com/problems/Top-View-of-a-Tree>

题目描述

Given a binary tree root, return the top view of the tree, sorted left-to-right.

Constraints

$n \leq 100,000$ where n is the number of nodes in root

前置知识

- BFS
- 哈希表

思路

首先我们分析一下题目。

1. 题目中给的示例并没有输出树中所有的节点，可以看到节点 4 和 节点 5 并没有被输出。其原因在于这两个节点被挡住了。具体来说是节点 4 被节点 1 挡住了，节点 5 被节点 3 挡住了。
2. 同时，题目要求我们返回的顺序是从左到右的。如何做到这一点？

对于第一个问题，我们可以记录一下每一个节点的横坐标和纵坐标。如果两个节点横坐标相同，那么纵坐标大的覆盖纵坐标小的。

对于第二个问题，我们需要记录横坐标，最终按照横坐标从小到大的顺序输出即可。

因此解决问题的核心在于记录横纵坐标。假设一个节点的坐标为 (x, y) 。这样进行遍历的时候左节点就是 $(x - 1, y + 1)$ ，右节点就是 $(x + 1, y + 1)$ 。我们只需要初始化 root 为 $(0,0)$ 染回遍历，将所有节点的横纵坐标以及 value 放到一个哈希表中，最终将哈希值中的数据排序输出即可。

当然你给左右子节点编号是 $(x-dx, y+dy), (x+dx, y+dy)$ 也是可以的，不过这并没有必要。

实际上，我们也可不用记录纵坐标，仅记录横坐标即可。这样我们可以采用层次遍历 + 直接覆盖的方式进行，而无需考虑纵坐标的关系。

关键点

- 对节点进行横纵坐标的编号，以及节点和左右子节点的编号关系。
- 层次遍历简化纵坐标的判断

代码

代码支持 Python3：

```
# class Tree:
#     def __init__(self, val, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def solve(self, root):
        q = collections.deque([(root, 0)])
        d = {}
        while q:
            cur, pos = q.popleft()
            if pos not in d:
                d[pos] = cur.val
            if cur.left:
                q.append((cur.left, pos - 1))
            if cur.right:
                q.append((cur.right, pos + 1))
        return list(map(lambda x:x[1], sorted(d.items(), key
```

令 n 节点数

复杂度分析

- 时间复杂度: $O(n \log n)$ 。我们使用了排序
- 空间复杂度: $O(n)$

746. 使用最小花费爬楼梯

题目地址 ([746. 使用最小花费爬楼梯](https://leetcode-cn.com/problems/min-cost-climbing-stairs/))

<https://leetcode-cn.com/problems/min-cost-climbing-stairs/>

标签

- 动态规划

难度

- 简单

入选理由

1. 我们讲几道爬楼梯以及爬楼梯的换皮题。让大家感受一下套路是什么

题目描述

数组的每个下标作为一个阶梯，第 i 个阶梯对应着一个非负数的体力花费值 $cost[i]$ 。

每当你爬上一个阶梯你都要花费对应的体力值，一旦支付了相应的体力值，你就不能再返回到这个阶梯。

请你找出达到楼层顶部的最低花费。在开始时，你可以选择从下标为 0 或 1 的阶梯开始。

示例 1：

输入: `cost = [10, 15, 20]`

输出: `15`

解释: 最低花费是从 `cost[1]` 开始，然后走两步即可到阶梯顶，一共花费 `15`。

示例 2：

输入: `cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]`

输出: `6`

解释: 最低花费方式是从 `cost[0]` 开始，逐个经过那些 `1`，跳过 `cost[3]`。

提示:

`cost` 的长度范围是 `[2, 1000]`。

`cost[i]` 将会是一个整型数据，范围为 `[0, 999]`。

前置知识

- 动态规划

分析

该题其实就是讲义中爬楼梯的变形题目，核心思路是不变的，只不过所求目标变成了**登完所有台阶所需要的最小花费**

- 定义 `dp` 数组，`dp[i]` 定义为登完 i 阶台阶所需最小花费（子问题）
- 思考：登完当前第 i 阶台阶所需花费是第 i 阶台阶消耗体力 + (`dp[i-1]` or `dp[i - 2]`)，由于所求为最小，故可得状态转移方程为：

$$dp[i] = \min(dp[i - 1], dp[i - 2]) + cost[i]$$

- 注意初始化第 1 阶和第 2 阶的情况

代码：

Java

989. 数组形式的整数加法

```
class Solution {
    public int minCostClimbingStairs(int[] cost) {

        if (cost == null || cost.length == 0)
            return 0;

        int[] dp = new int[cost.length + 1];
        dp[0] = cost[0];
        dp[1] = cost[1];

        for(int i = 2; i <= cost.length; i++)
            dp[i] = Math.min(dp[i - 1], dp[i - 2]) + (i ==

        return dp[cost.length];
    }
}
```

进阶：尝试将空间复杂度优化到\$O(1)\$

复杂度分析

设：\$N\$台阶

时间复杂度：\$O(N)\$

空间复杂度：\$O(N)\$

题目地址(198. 打家劫舍)

<https://leetcode-cn.com/problems/house-robber/>

入选理由

1. 爬楼梯换皮题给大家出一个，后面再列举几个大家有时间自己做做

标签

- 动态规划

难度

- 中等

题目描述

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你的决策的因素是房子是否被警报装置所覆盖。

给定一个代表每个房屋存放金额的非负整数数组，计算你不触动警报装置的情况下，能够偷窃到的最高金额。

示例 1:

输入: [1,2,3,1]

输出: 4

解释: 偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。

偷窃到的最高金额 = 1 + 3 = 4 。

示例 2:

输入: [2,7,9,3,1]

输出: 12

解释: 偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9), 接着偷窃 5 号房屋 (金额 = 1)。

提示:

$0 \leq \text{nums.length} \leq 100$

$0 \leq \text{nums}[i] \leq 400$

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 fibonacci 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 + $dp[i - 2]$

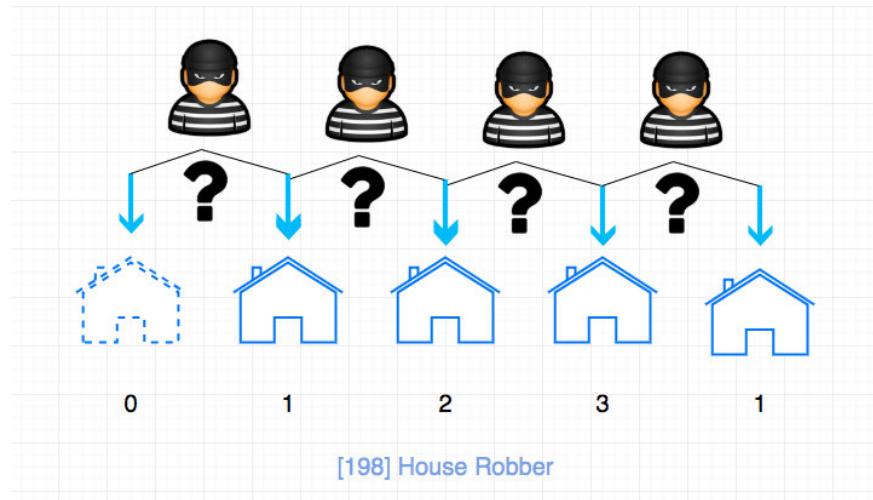
i - 1 不能抢，否则会触发警铃

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 $dp[i] = \max(dp[i - 2] + nums[i - 2], dp[i - 1]);$ (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:

989. 数组形式的整数加法

```
/**  
 * @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:

```
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](#)

其他题目推荐

- [Minimum-Sum-Subsequence](#)

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 37K 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(673. 最长递增子序列的个数)

<https://leetcode-cn.com/problems/number-of-longest-increasing-subsequence/>

入选理由

1. 这是 DP 问题的另一个经典类型 - LIS (最长上升子序列) , 而今天这个题目是这个系列最难的之一。如果这个不会, 可以先看下我的 LIS 专题 (自己搜吧) 。

标签

- 动态规划

难度

- 中等

题目描述

给定一个未排序的整数数组，找到最长递增子序列的个数。

示例 1:

输入: [1,3,5,4,7]

输出: 2

解释: 有两个最长递增子序列，分别是 [1, 3, 4, 7] 和 [1, 3, 5, 7]。

示例 2:

输入: [2,2,2,2,2]

输出: 5

解释: 最长递增子序列的长度是1，并且存在5个子序列的长度为1，因此输出5。

注意: 给定的数组长度不超过 2000 并且结果一定是32位有符号整数。

前置知识

- 动态规划

公司

- 暂无

思路

这道题其实就是最长上升子序列（LIS）的变种题。如果对 LIS 不了解的可以先看下我之前写的一篇文章[穿上衣服我就不认识你了？来聊聊最长上升子序列](#)，里面将这种题目的套路讲得很清楚了。

回到这道题。题目让我们求最长递增子序列的个数，而不是通常的最长递增子序列的长度。因此我想到使用另外一个变量记录最长递增子序列的个数信息即可。类似的套路有[股票问题](#)，这种问题的套路在于只是单独存储一个状态以无法满足条件，对于这道题来说，我们存储的单一状态就是最长递增子序列的长度。那么一个自然的想法是不存储最长递增子序列的长度，而是仅存储最长递增子序列的个数可以么？这是不可以的，因为最长递增子序列的个数 隐式地条件是你要先找到最长的递增子序列才行。

如何存储两个状态呢？一般有两种方式：

- 二维数组 $dp[i][0]$ 第一个状态 $dp[i][1]$ 第二个状态
- $dp1[i]$ 第一个状态 $dp2[i]$ 第二个状态

使用哪个都可以，空间复杂度也是一样的，使用哪种看你自己。这里我们使用第一种，并且 $dp[i][0]$ 表示以 $nums[i]$ 结尾的最长上升子序列的长度， $dp[i][1]$ 表示以 $nums[i]$ 结尾的长度为 $dp[i][0]$ 的子序列的个数。

明确了要多存储一个状态之后，我们来看下状态如何转移。

LIS 的一般过程是这样的：

```
for i in range(n):
    for j in range(i + 1, n):
        if nums[j] > nums[i]:
            # ...
```

这道题也是类似，遍历到 $nums[j]$ 的时候往前遍历所有的满足 $i < j$ 的 i 。

- 如果 $nums[j] \leq nums[i]$ ， $nums[j]$ 无法和前面任何的序列拼接成递增子序列
- 否则说明我们可以拼接。但是拼接与否取决于拼接之后会不会更长。如果更长了就拼，否则不拼。

上面是 LIS 的常规思路，下面我们加一点逻辑。

- 如果拼接后的序列更长，那么 $dp[j][1] = dp[i][1]$ （这点容易忽略）
- 如果拼接之后序列一样长，那么 $dp[j][1] += dp[i][1]$ 。
- 如果拼接之后变短了，则不应该拼接。

关键点解析

- 最长上升子序列问题
- $dp[j][1] = dp[i][1]$ 容易忘记

代码

代码支持: Python

```
class Solution:
    def findNumberOfLIS(self, nums: List[int]) -> int:
        n = len(nums)
        # dp[i][0] -> LIS
        # dp[i][1] -> NumberofLIS
        dp = [[1, 1] for i in range(n)]
        ans = [1, 1]
        longest = 1
        for i in range(n):
            for j in range(i + 1, n):
                if nums[j] > nums[i]:
                    if dp[i][0] + 1 > dp[j][0]:
                        dp[j][0] = dp[i][0] + 1
                        # 下面这行代码容易忘记, 导致出错
                        dp[j][1] = dp[i][1]
                        longest = max(longest, dp[j][0])
                    elif dp[i][0] + 1 == dp[j][0]:
                        dp[j][1] += dp[i][1]
        return sum(dp[i][1] for i in range(n)) if dp[i][0] ==
```

复杂度分析

令 N 为数组长度。

- 时间复杂度: $O(N^2)$
- 空间复杂度: $O(N)$

扩展

这道题也可以使用线段树来解决，并且性能更好，不过由于不算是常规解法，因此不再这里展开，感兴趣的同學可以尝试一下。

更多题解可以访问我的 LeetCode 题解仓库：

<https://github.com/azl397985856/leetcode>。目前已经 37K 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。

989. 数组形式的整数加法



欢迎长按关注



题目地址（1143.最长公共子序列）

<https://leetcode-cn.com/problems/longest-common-subsequence>

入选理由

1. DP 另一个重要内容：LCS（最长公共子序列）

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：

$1 \leq \text{text1.length} \leq 1000$ $1 \leq \text{text2.length} \leq 1000$ 输入的字符串只含有小写英文字母。

前置知识

- 数组
- 动态规划

标签

- 动态规划

难度

- 中等

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 $A[i] \neq B[j]$ ，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) -> int:
        m, n = len(A), len(B)
        ans = 0
        dp = [[0 for _ in range(n + 1)] for _ in range(m + 1)]
        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 - 1])
        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。

题目地址(62. 不同路径)

<https://leetcode-cn.com/problems/unique-paths/>

入选理由

- 1. 二维网格 dp

标签

- 动态规划

难度

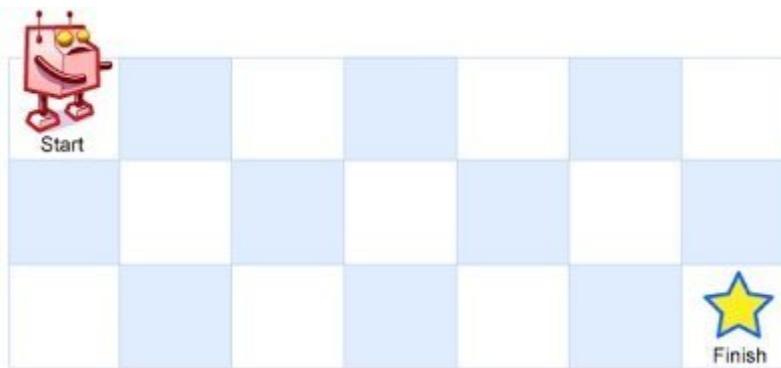
- 中等

题目描述

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

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

问总共有多少条不同的路径？



989. 数组形式的整数加法

例如，上图是一个 7×3 的网格。有多少可能的路径？

示例 1：

输入： $m = 3, n = 2$

输出： 3

解释：

从左上角开始，总共有 3 条路径可以到达右下角。

1. 向右 -> 向右 -> 向下
2. 向右 -> 向下 -> 向右
3. 向下 -> 向右 -> 向右

示例 2：

输入： $m = 7, n = 3$

输出： 28

提示：

$1 \leq m, n \leq 100$

题目数据保证答案小于等于 $2 * 10^9$

前置知识

- 排列组合
- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

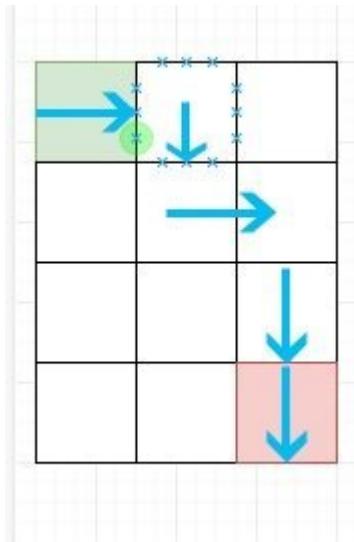
思路

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。

$$C_n^r = \frac{n!}{r!(n-r)!}$$

而这道题我们也可以用动态规划来解。其实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坏事。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。



这不就是二维平面的爬楼梯么？和爬楼梯又有什么不同呢？

代码大概是：

Python Code:

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        d = [[1] * n for _ in range(m)]

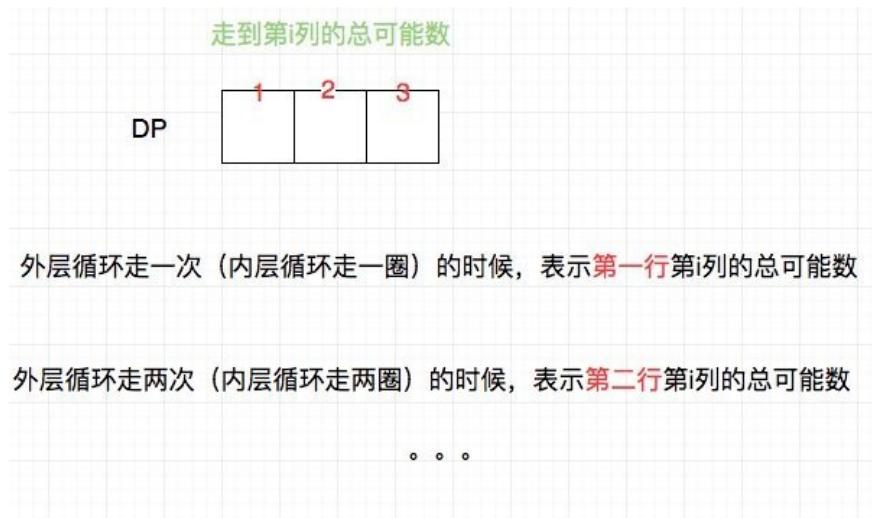
        for col in range(1, m):
            for row in range(1, n):
                d[col][row] = d[col - 1][row] + d[col][row - 1]

        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 $O(n)$.



外层循环走一次（内层循环走一圈）的时候，表示第一行第*i*列的总可能数

外层循环走两次（内层循环走两圈）的时候，表示第二行第*i*列的总可能数

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
        return self.uniquePaths(m - 1, n) + self.uniquePaths(m, n - 1)
```

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

989. 数组形式的整数加法

代码支持 JavaScript, Python3, CPP

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]
```

CPP Code:

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        vector<int> dp(n + 1, 0);
        dp[n - 1] = 1;
        for (int i = m - 1; i >= 0; --i) {
            for (int j = n - 1; j >= 0; --j) dp[j] += dp[j]
        }
        return dp[0];
    }
};
```

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](#)
- [【每日一题】 - 2020-09-14 - 小兔的棋盘](#)

题目地址 (688. “马”在棋盘上的概率)

<https://leetcode-cn.com/problems/knight-probability-in-chessboard/>

入选理由

1. 典型的不那么“连续”的 dp。比如 LIS，我们 dp formula 都是相连的，这个则是不连续的。

标签

- 动态规划

难度

- 中等

题目描述

已知一个 $N \times N$ 的国际象棋棋盘，棋盘的行号和列号都是从 0 开始。即最左上角为 $(0, 0)$ ，最右下角为 $(N-1, N-1)$ 。

现有一个“马”（也译作“骑士”）位于 (r, c) ，并打算进行 K 次移动。

如下图所示，国际象棋的“马”每一步先沿水平或垂直方向移动 2 个格子，然

现在“马”每一步都从可选的位置（包括棋盘外部的）中独立随机地选择一个进

求移动结束后，“马”仍留在棋盘上的概率。

前置知识

- 动态规划
- 数组

分析

读完题也不难发现是个动态规划问题，并且还有爬楼梯的影子，只不过他的移动方式是类似“马走日”的方式，也就是当前位置可以由其他八个位置移动过来。更细致地说：

- 我们用个二维数组来存储马跳到每个格子的概率
- 首先，开一个二维数组来存初始状态，即马的初始状态概率为 1，其他位置都为 0。
- 接下来我们按步一次一次更新二维数组每个位置的概率，我们可以通过当前位置来知道上一轮有哪些地方（在棋盘内合法的位置）的马跳一次可以到达该位置，这样我们把这些地方的概率 $\times 0.125$ （因为这些地方每一个都有八种选择路线，选择到该位置的概率自然是 $1/8$ ）加到当前的位置上即完成更新。
- 该次棋盘更新完成要把该棋盘作为上一轮的状态棋盘以便下次循环来使用

代码：

Java

989. 数组形式的整数加法

```
class Solution {

    private int[][] dir = {{-1, -2}, {1, -2}, {2, -1}, {2,
        1}, {-2, 1}, {-2, -1}, {-1, 2}, {1, 2}};

    public double knightProbability(int N, int K, int r, int c) {
        double[][] dp = new double[N][N];
        dp[r][c] = 1;

        for (int step = 0; step <= K; step++) {
            double[][] dpTemp = new double[N][N];

            for (int i = 0; i < N; i++)
                for (int j = 0; j < N; j++)
                    for (int[] direction : dir) {
                        int lastR = i - direction[0];
                        int lastC = j - direction[1];
                        if (lastR >= 0 && lastR < N && lastC >= 0 && lastC < N)
                            dpTemp[i][j] += dp[lastR][lastC];
                    }

            dp = dpTemp;
        }

        double res = 0;
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                res += dp[i][j];
        return res;
    }
}
```

复杂度分析

设：棋盘为\$NxN\$

时间复杂度：\$O(KN^2)\$

空间复杂度：\$O(N^2)\$

题目地址（464. 我能赢么）

<https://leetcode-cn.com/problems/can-i-win/>

入选理由

1. 我们要讲的 DP 最后一个类型：状压 DP。其他的 dp (比如数位 dp, 由于时间关系，暂时不讲了)

标签

- 动态规划

难度

- 中等

题目描述

在 "100 game" 这个游戏中，两名玩家轮流选择从 1 到 10 的任意整数，累加直到总和达到或超过 100。如果我们将游戏规则改为“玩家不能重复使用整数”呢？

例如，两个玩家可以轮流从公共整数池中抽取从 1 到 15 的整数（不放回），给定一个整数 maxChoosableInteger（整数池中可选择的最大数）和另一个整数 desiredTotal（累积和的目标值），你可以假设 maxChoosableInteger 不会大于 20，desiredTotal 不会大于 maxChoosableInteger * (maxChoosableInteger + 1) / 2。

示例：

输入：
maxChoosableInteger = 10
desiredTotal = 11

输出：
false

解释：
无论第一个玩家选择哪个整数，他都会失败。
第一个玩家可以选择从 1 到 10 的整数。
如果第一个玩家选择 1，那么第二个玩家只能选择从 2 到 10 的整数。
第二个玩家可以通过选择整数 10（那么累积和为 $11 \geq desiredTotal$ ），同样地，第一个玩家选择任意其他整数，第二个玩家都会赢。

前置知识

- 动态规划
- 回溯

公司

- 阿里
- linkedin

暴力解（超时）

思路

题目的函数签名如下：

```
def canIWin(self, maxChoosableInteger: int, desiredTotal: int) -> bool:
```

即给你两个整数 maxChoosableInteger 和 desiredTotal，让你返回一个布尔值。

两种特殊情况

首先考虑两种特殊情况，后面所有的解法这两种特殊情况都适用，因此不再赘述。

- 如果 desiredTotal 是小于等于 maxChoosableInteger 的，直接返回 True，这不难理解。
- 如果 $[1, \text{maxChoosableInteger}]$ 全部数字之和小于 desiredTotal，谁都无法赢，返回 False。

一般情况

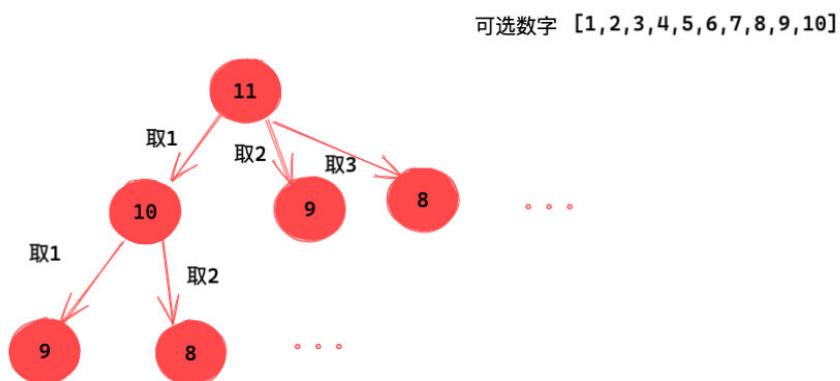
考虑完了特殊情况，我们继续思考一般情况。

首先我们来简化一下问题，如果数字可以随便选呢？这个问题就简单多了，和爬楼梯没啥区别。这里考虑暴力求解，使用 DFS + 模拟的方式来解决。

注意到每次可选的数字都不变，都是 $[1, \text{maxChoosableInteger}]$ ，因此无需通过参数传递。或者你想传递的话，把引用往下传也是可以的。

这里的 $[1, \text{maxChoosableInteger}]$ 指的是一个左右闭合的区间。

为了方便大家理解，我画了一个逻辑树：



接下来，我们写代码遍历这棵树即可。

可重复选的暴力核心代码如下：

```

class Solution:
    def canIWin(self, maxChoosableInteger: int, desiredTotal: int) -> bool:
        # acc 表示当前累计的数字和
        def dfs(acc):
            if acc >= desiredTotal:
                return False
            for n in range(1, maxChoosableInteger + 1):
                # 对方有一种情况赢不了，我就选这个数字就能赢了，返回 True
                if not dfs(acc + n):
                    return True
            return False

        # 初始化集合，用于保存当前已经选择过的数。
        return dfs(0)

```

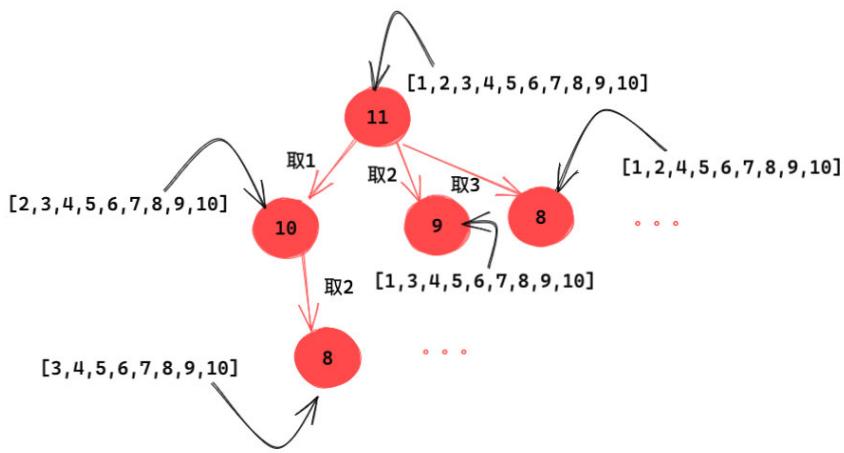
上面代码已经很清晰了，并且加了注释，我就不多解释了。我们继续来看下如果数字不允许重复选会怎么样？

一个直观的思路是使用 `set` 记录已经被取的数字。当选数字的时候，如果是在 `set` 中则不取即可。由于可选数字在动态变化。也就是说上面的逻辑树部分，每个树节点的可选数字都是不同的。

那怎么办呢？很简单，通过参数传递呗。而且：

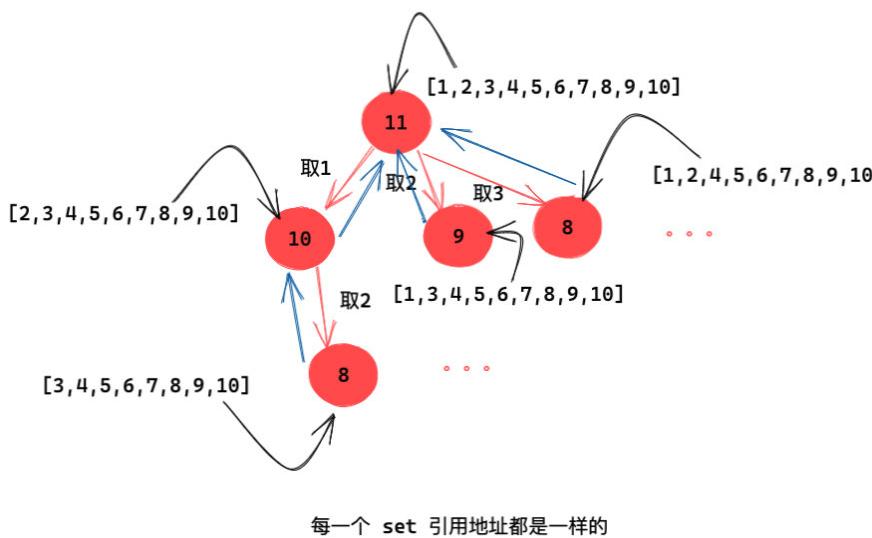
- 要么 `set` 是值传递，这样不会相互影响。
- 要么每次递归返回的是时候主动回溯状态。关于这块不熟悉的，可以看下我之前写过的[回溯专题](#)。

如果使用值传递，对应是这样的：



每一个 `set` 引用地址都是不同的

如果在每次递归返回的是时候主动回溯状态，对应是这样的：



注意图上的蓝色的新增的线，他们表示递归返回的过程。我们需要在返回的过程撤销选择。比如我选了数组 2， 递归返回的时候再把数字 2 从 set 中移除。

简单对比下两种方法。

- 使用 `set` 的值传递，每个递归树的节点都会存一个完整的 `set`，空间大概是 `节点的数目 X set 中数字个数`，因此空间复杂度大概是 $O(2^{\max\text{ChoosableInteger}} * \max\text{ChoosableInteger})$ ，这个空间根本不可想象，太大了。
- 使用本状态回溯的方式。由于每次都要从 `set` 中移除指定数字，时间复杂度是 $O(\max\text{ChoosableInteger} * \text{节点数})$ ，这样做时间复杂度又太高了。

这里我用了第二种方式 - 状态回溯。和上面代码没有太大的区别，只是加了一个 `set` 而已，唯一需要注意的是需要在回溯过程恢复状态 (`picked.remove(n)`)。

代码

代码支持：Python3

Python3 Code:

```

class Solution:
    def canIWin(self, maxChoosableInteger: int, desiredTotal: int) -> bool:
        if desiredTotal <= maxChoosableInteger:
            return True
        if sum(range(maxChoosableInteger + 1)) < desiredTotal:
            return False
        # picked 用于保存当前已经选择过的数。
        # acc 表示当前累计的数字和
        def backtrack(picked, acc):
            if acc >= desiredTotal:
                return False
            if len(picked) == maxChoosableInteger:
                # 说明全部都被选了，没得选了，返回 False，代表输
                return False
            for n in range(1, maxChoosableInteger + 1):
                if n not in picked:
                    picked.add(n)
                    # 对方有一种情况赢不了，我就选这个数字就能赢了
                    if not backtrack(picked, acc + n):
                        picked.remove(n)
                        return True
                    picked.remove(n)
            return False

        # 初始化集合，用于保存当前已经选择过的数。
        return backtrack(set(), 0)

```

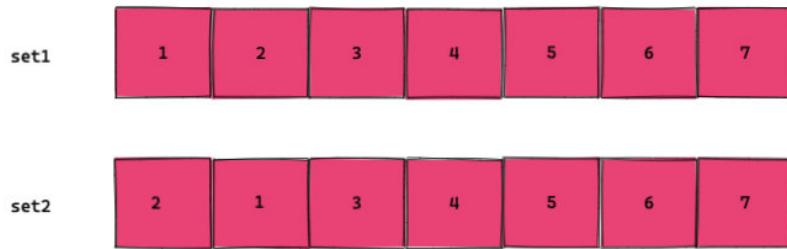
状态压缩 + 回溯

思路

有的同学可能会问，为什么不使用记忆化递归？这样可以有效减少逻辑树的节点数，从指数级下降到多项式级。这里的原因在于 `set` 是不可直接序列化的，因此不可直接存储到诸如哈希表这样的数据结构。

而如果你自己写序列化，比如最粗糙的将 `set` 转换为字符串或者元祖存。看起来可行，`set` 是 `ordered` 的，因此如果想正确序列化还需要排序。当然你可用一个 `orderedhashset`，不过效率依然不好，感兴趣的可以研究一下。

如下图，两个 `set` 应该一样，但是遍历的结果顺序可能不同，如果不排序就可能有错误。



至此，问题的关键基本上锁定为找到一个可以序列化且容量大大减少的数据结构来存是不是就可行了？

注意到 `maxChoosableInteger` 不会大于 20 这是一个强有力的提示。由于 20 是一个不大于 32 的数字，因此这道题很有可能和状态压缩有关，比如用 4 个字节存储状态。力扣相关的题目还有不少，具体大家可参考文末的相关题目。

我们可以将状态进行压缩，使用位来模拟。实际上使用状态压缩和上面思路一模一样，只是 API 不一样罢了。

假如我们使用的这个用来代替 set 的数字名称为 `picked`。

- `picked` 第一位表示数字 1 的使用情况。
- `picked` 第二位表示数字 2 的使用情况。
- `picked` 第三位表示数字 3 的使用情况。
- . . .

比如我们刚才用了集合，用到的集合 api 有：

- `in` 操作符，判断一个数字是否在集合中
- `add(n)` 函数，用于将一个数加入到集合
- `len()`，用于判断集合的大小

那我们其实就用位来模拟实现这三个 api 就罢了。详细可参考我的这篇题解 - [面试题 01.01. 判定字符是否唯一](#)

如果实现 `add` 操作？

这个不难。比如我要模拟 `picked.add(n)`，只要将 `picked` 第 `n` 位置为 1 就行。也就是说 1 表示在集合中，0 表示不在。

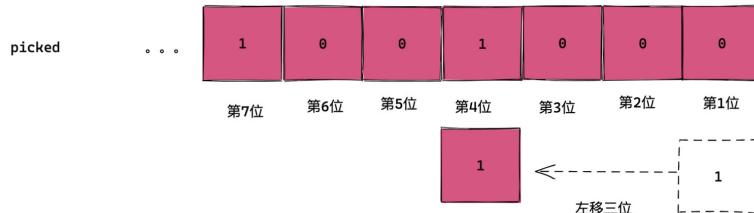


使用或运算和位移运算可以很好的完成这个需求。

位移运算

```
1 << a
```

指的是 1 的二进制表示全体左移 a 位，右移也是同理



| 操作

```
a | b
```

指的是 a 和 b 每一位都进行或运算的结构。常见的用法是 a 和 b 其中一个当成是 seen。这样就可以当二值数组和哈希表用了。比如：

```
seen = 0b00000000
a = 0b00000001
b = 0b00000010

seen |= a 后,    seen 为 0b00000001
seen |= b 后,    seen 为 0b00000011
```

这样我就可以知道 a 和 b 出现过了。当然 a, b 以及其他你需要统计的数字只能用一位。典型的是题目只需要存 26 个字母，那么一个 int(32 bit) 足够了。如果是包括大写，那就是 52，就需要至少 52 bit。

如何实现 in 操作符？

有了上面的铺垫就简单了。比如要模拟 n in picked。那只要判断 picked 的第 n 位是 0 还是 1 就行了。如果是 0 表示不在 picked 中，如果是 1 表示在 picked 中。

用或运算和位移运算可以很好的完成这个需求。

& 操作

```
a & b
```

指的是 a 和 b 每一位都进行与运算的结构。常见的用法是 a 和 b 其中一个是 mask。这样就可以得指定位是 0 还是 1 了。比如：

```

mask = 0b0000010
a & mask == 1 说明 a 在第二位（从低到高）是 1
a & mask == 0 说明 a 在第二位（从低到高）是 0

```

如何实现 len

其实只要逐个 bit 比对，如果当前 bit 是 1 则计数器 + 1，最后返回计数器的值即可。

这没有问题。而实际上，我们只关心集合大小是否等于 `maxChoosableInteger`。也就是我只关心第 `maxChoosableInteger` 位以及低于 `maxChoosableInteger` 的位是否全部是 1。

这就简单了，我们只需要将 1 左移 `maxChoosableInteger + 1` 位再减去 1 即可。一行代码搞定：

```
picked == (1 << (maxChoosableInteger + 1)) - 1
```

上面代码返回 true 表示满了，否则没满。

至此大家应该感受到了，使用位来代替 set 思路上没有任何区别。不同的仅仅是 API 而已。如果你只会使用 set 不会使用位运算进行状态压缩，只能说明你对位的 api 不熟而已。多练习几道就行了，文末我列举了几道类似的题目，大家不要错过哦~

关键点分析

- 回溯
- 动态规划
- 状态压缩

代码

代码支持：Java,CPP,Python3,JS

Java Code:

989. 数组形式的整数加法

```
public class Solution {
    public boolean canIWin(int maxChoosableInteger, int desiredTotal) {
        if (maxChoosableInteger >= desiredTotal) return true;
        if ((1 + maxChoosableInteger) * maxChoosableInteger / 2 < desiredTotal) return false;

        Boolean[] dp = new Boolean[(1 << maxChoosableInteger)];
        return dfs(maxChoosableInteger, desiredTotal, 0, dp);
    }

    private boolean dfs(int maxChoosableInteger, int desiredTotal, int state, Boolean[] dp) {
        if (dp[state] != null)
            return dp[state];
        for (int i = 1; i <= maxChoosableInteger; i++){
            int tmp = (1 << (i - 1));
            if ((tmp & state) == 0){
                if (desiredTotal - i <= 0 || !dfs(maxChoosableInteger, desiredTotal - i, state | tmp, dp))
                    dp[state] = true;
                return true;
            }
        }
        dp[state] = false;
        return false;
    }
}
```

C++ Code:

989. 数组形式的整数加法

```
class Solution {
public:
    bool canIWin(int maxChoosableInteger, int desiredTotal) {
        int sum = (1+maxChoosableInteger)*maxChoosableInteger/2;
        if(sum < desiredTotal){
            return false;
        }
        unordered_map<int,int> d;
        return dfs(maxChoosableInteger,0,desiredTotal,0,d);
    }

    bool dfs(int n,int s,int t,int S,unordered_map<int,int> d){
        if(d[S]) return d[S];
        int& ans = d[S];

        if(s >= t){
            return ans = true;
        }
        if(S == (((1 << n)-1) << 1)){
            return ans = false;
        }

        for(int m = 1;m <=n;++m){
            if(S & (1 << m)){
                continue;
            }
            int nextS = S|(1 << m);
            if(s+m >= t){
                return ans = true;
            }
            bool r1 = dfs(n,s+m,t,nextS,d);
            if(!r1){
                return ans = true;
            }
        }
        return ans = false;
    };
};
```

Python3 Code:

989. 数组形式的整数加法

```
class Solution:
    def canIWin(self, maxChoosableInteger: int, desiredTotal: int) -> bool:
        if desiredTotal <= maxChoosableInteger:
            return True
        if sum(range(maxChoosableInteger + 1)) < desiredTotal:
            return False

        @lru_cache(None)
        def dp(picked, acc):
            if acc >= desiredTotal:
                return False
            if picked == (1 << (maxChoosableInteger + 1)) - 1:
                return False
            for n in range(1, maxChoosableInteger + 1):
                if picked & 1 << n == 0:
                    if not dp(picked | 1 << n, acc + n):
                        return True
            return False

        return dp(0, 0)
```

JS Code:

```

var canIWin = function (maxChoosableInteger, desiredTotal)
    // 直接获胜
    if (maxChoosableInteger >= desiredTotal) return true;

    // 全部拿完也无法到达
    var sum = (maxChoosableInteger * (maxChoosableInteger + 1)) / 2;
    if (desiredTotal > sum) return false;

    // 记忆化
    var dp = {};

    /**
     * @param {number} total 剩余的数量
     * @param {number} state 使用二进制位表示抽过的状态
     */
    function f(total, state) {
        // 有缓存
        if (dp[state] !== undefined) return dp[state];

        for (var i = 1; i <= maxChoosableInteger; i++) {
            var curr = 1 << i;
            // 已经抽过这个数
            if (curr & state) continue;
            // 直接获胜
            if (i >= total) return (dp[state] = true);
            // 可以让对方输
            if (!f(total - i, state | curr)) return (dp[state] = false);
        }

        // 没有任何让对方输的方法
        return (dp[state] = false);
    }

    return f(desiredTotal, 0);
};

```

相关题目

- 面试题 01.01. 判定字符是否唯一 纯状态压缩，无 DP
- 698. 划分为 k 个相等的子集
- 1681. 最小不兼容性

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 37K star 啦。大

989. 数组形式的整数加法

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(416. 分割等和子集)

<https://leetcode-cn.com/problems/partition-equal-subset-sum/>

入选理由

1. 背包换皮题，锻炼大家抽象能力

标签

- 动态规划
- DFS

难度

- 中等

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。

989. 数组形式的整数加法

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，使得它们的和相等。

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.

思路

这次是背包的第一个题目，我们讲详细一点，之后就直接提重点信息。如果你还是没懂，建议回头再看讲义或者这篇题解。

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 $\text{sum} \% 2 === 0$, 需要找到和为 $2/\text{sum}$ 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 $2/\text{sum}$ 。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 $> 2/\text{sum}$, 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.length) {
        return false;
    }

    return (
        dfs(nums, pickRemain - nums[cur], discardRemain, cur + 1) ||
        dfs(nums, pickRemain, discardRemain - nums[cur], cur + 1)
    );
}
```

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 $\text{target} - \text{nums}[i]$ 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 的子序列取决于前 $i-1$ 数的情况。

记 $F[i, \text{target}]$ 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 $\text{sum}(\text{nums}) / 2$

javascript 实现

989. 数组形式的整数加法

```
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 $F[i, target] = F[i - 1, target] \mid F[i - 1, target - nums[i]]$ 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 C_i ，得到的价值是 W_i 。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

$$F[i, v] = \max\{F[i-1, v], F[i-1, v-C_i] + W_i\}$$

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

$$F[i, v] = F[i-1, v] \mid\mid F[i-1, v-C_i]$$

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。

989. 数组形式的整数加法

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 $F[i, v] = F[i-1, v] + F[i-1, v-C_i]$

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 = 0; j <= amount; j++) {  
            dp[j] = dp[j] + (j - coins[i] >= 0 ? dp[j - coins[i]] : 0);  
        }  
    }  
    return dp[amount];  
};
```

参考

[背包九讲](#)

494. 目标和

入选理由

1. 和昨天题目很像，你还会么？

题目地址（494. 目标和）

<https://leetcode-cn.com/problems/target-sum/>

题目描述

给定一个非负整数数组， a_1, a_2, \dots, a_n ，和一个目标数， $target$ 。现在你有两个符号 + 和 -。对于数组中的任意一个整数，你都可以从 + 或 - 中选择一个符号添加在前面。

返回可以使最终数组和为目标数 $target$ 的所有添加符号的方法数。

示例：

输入：nums: [1, 1, 1, 1, 1], target: 3

输出：5

解释：

-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

一共有5种方法让最终目标和为3。

前置知识

- 背包
- 数学

分析

题目说了数组元素都是非负的，然后我们假定最终选择的数组中：全正子数组和是 positive，全负子数组和是 negative，数组元素绝对值总和是 total，目标数是 target，若想符合题目要求，则必然满足如下等式：

$$\text{positive} + \text{negative} = \text{target}$$

$$\text{positive} - \text{negative} = \text{total}$$

则可以推出：

$$\text{positive} = \frac{(\text{target} + \text{total})}{2}$$

那么我们来开始抽象：

1. 数组的元素就是背包的物体
2. 元素大小就是物体重量，默认 1 为所有元素价值。
3. 包大小就是上述推出的 positive
4. 每个元素只能用一次

这就是 01 背包，目标是求出恰好装满背包的所有方案数目，可以忽略价值，不难写出如下代码。需要注意的是一些 edge case，具体参考下方代码，edge case 均在代码顶部进行了处理。

代码

代码支持：Java, Python3, JS

Java Code:

```
public int findTargetSumWays(int[] nums, int target) {

    int sum = 0;
    for (int num : nums)
        sum += num;

    if (sum < Math.abs(target))
        return 0;

    if (((sum + target) & 1) == 1)
        return 0;

    sum = (sum + target) / 2;
    int[] dp = new int[sum + 1];
    dp[0] = 1;

    for (int i = 0; i < nums.length; i++)
        for (int j = sum; j >= nums[i]; j--)
            dp[j] = dp[j] + dp[j - nums[i]];

    return dp[sum];
}
```

Python3 Code:

989. 数组形式的整数加法

```
class Solution:
    def findTargetSumWays(self, nums, target) -> bool:
        t = sum(nums) + target
        if t % 2:
            return 0
        t = t // 2

        dp = [0] * (t + 1)
        dp[0] = 1

        for i in range(len(nums)):
            for j in range(t, nums[i] - 1, -1):
                dp[j] += dp[j - nums[i]]
        return dp[-1]
```

JS Code:

```
var findTargetSumWays = function (nums) {
    const sum = nums.reduce((a, b) => a + b, 0);
    let t = sum + target;
    if (t % 2) return 0;
    t = Math.floor(t / 2);
    const dp = Array(t + 1).fill(0);
    dp[0] = 1;
    for (const n of nums) {
        for (let i = t; i >= n; i--) {
            dp[i] += dp[i - n];
        }
    }
    return dp[t];
};
```

复杂度分析

令 total 为元素总和，negative 为元素个数

- 时间复杂度: $O(\frac{negative \cdot (total + target)}{2})$
- 空间复杂度: $O(\frac{total + target}{2})$

扩展

下面是二维 dp 的做法，虽然不推荐，但是可以帮助看不懂上面滚动数组优化解法的胖友理解：

989. 数组形式的整数加法

```
class Solution:
    def solve(self, nums, target):
        if (sum(nums) + target) % 2 == 1: return 0
        t = (sum(nums) + target) // 2
        dp = [[0] * (len(nums) + 1) for _ in range(t + 1)]
        dp[0][0] = 1
        for i in range(t + 1):
            for j in range(1, len(nums) + 1):
                dp[i][j] = dp[i][j-1]
                if i - nums[j-1] >= 0: dp[i][j] += dp[i - i]
```

另外也可使用递归来写，简单直接。

```
class Solution:
    def solve(self, nums, target):
        @lru_cache(None)
        def f(i, cur_sum):
            if i == len(nums):
                if target == cur_sum:
                    return 1
                return 0

            return f(i + 1, cur_sum + nums[i]) + f(i + 1, cur_sum)

        return f(0, 0)
```

322. 零钱兑换

题目地址(322. 零钱兑换)

<https://leetcode-cn.com/problems/coin-change/>

标签

- 动态规划

难度

- 中等

题目描述

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

你可以认为每种硬币的数量是无限的。

示例 1:

输入: coins = [1, 2, 5], amount = 11

输出: 3

解释: $11 = 5 + 5 + 1$

示例 2:

输入: coins = [2], amount = 3

输出: -1

示例 3:

输入: coins = [1], amount = 0

输出: 0

示例 4:

输入: coins = [1], amount = 1

输出: 1

示例 5:

输入: coins = [1], amount = 2

输出: 2

提示:

```
1 <= coins.length <= 12
1 <= coins[i] <= 231 - 1
0 <= amount <= 104
```

思路

零钱系列是很经典的背包问题的变形，读题可以发现，每种硬币是没有数量限制的，硬币就是物品，amount 就是背包的大小，因此该题抽象出来就是个完全背包问题，只不过专题讲义用的是获得的最大价值，该题是求最小价值，所谓背包中的价值就是装硬币的个数。

需要注意的是由于专题问题定义为最大价值，因此 dp 初始化为 0。而该题需求最小价值，因此 dp 初始化为 max_value 且 $dp[0] = 0$

按照上述思路分析+专题给出的模板，可以很轻松地写出如下动态规划代码。

代码

代码支持: Java, Python3

989. 数组形式的整数加法

Java Code:

```
public int coinChange(int[] coins, int amount) {  
  
    if (coins == null || coins.length == 0 || amount <= 0)  
        return 0;  
  
    int[] dp = new int[amount + 1];  
  
    Arrays.fill(dp, amount + 1);  
    dp[0] = 0;  
  
    for (int coin : coins) {  
  
        for (int i = coin; i <= amount; i++) {  
  
            dp[i] = Math.min(dp[i], 1 + dp[i - coin]);  
        }  
    }  
  
    return dp[amount] == amount + 1 ? -1 : dp[amount];  
}
```

Python3 Code:

```
class Solution(object):  
    def coinChange(self, coins, amount):  
        dp = [amount + 1] * (amount+1)  
        dp[0] = 0  
        for i in range(1, amount+1):  
            for coin in coins:  
                if i >= coin:  
                    dp[i] = min(dp[i], dp[i-coin]+1)  
        return -1 if dp[amount] == amount + 1 else dp[amount]
```

复杂度分析

- 时间复杂度: $O(N * amount)$, 其中 N 是物品个数即硬币种类
- 空间复杂度: $O(amount)$, 其中 amount 为总金额也即背包大小

518. 零钱兑换 II

入选理由

- 昨天题目的系列题目。一起做更好理解背包的变种

题目地址(518. 零钱兑换 II)

<https://leetcode-cn.com/problems/coin-change-2/>

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

标签

- 动态规划

难度

- 中等

思路

定义状态 `dp[i][j]` 为使用前 i 个硬币组成金额 j 的组合数，则有状态转移方程为

$$dp[i][j] = dp[i-1][j] + dp[i][j - coins[i]]$$

压缩空间后的代码

```
var change = function (amount, coins) {
    const dp = Array.from({ length: amount + 1 }).fill(0);
    dp[0] = 1;
    for (let coin of coins) {
        for (let i = coin; i <= amount; i++) {
            dp[i] += dp[i - coin];
        }
    }
    return dp[amount];
};
```

题目地址(455. 分发饼干)

<https://leetcode-cn.com/problems/assign-cookies/>

入选理由

1. 贪心入门题目

标签

- 贪心

难度

- 简单

题目描述

989. 数组形式的整数加法

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

本题可用贪心求解。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。

算法：

- 将需求因子 g 和 s 分别从小到大进行排序
- 使用贪心思想，配合两个指针，每个饼干只尝试一次，成功则换下一个孩子来尝试，不成功则换下一个饼干 🍪 来尝试。

关键点

- 先排序再贪心

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，更新满足的孩子数并移
        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，需要换大的
            sj++;
        }
    }
    return res;
};
```

复杂度分析

989. 数组形式的整数加法

- 时间复杂度：由于使用了排序，因此时间复杂度为 $O(N \log N)$
- 空间复杂度： $O(1)$

435. 无重叠区间

题目地址 (435. 无重叠区间)

<https://leetcode-cn.com/problems/non-overlapping-intervals/>

入选理由

1. 贪心的进阶难度。贪心的题目都比较符合“常识和直觉”，不好归纳以及证明。因此大家可适当练习，我们就出两三道题 mu

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。

标签

- 贪心

难度

- 中等

动态规划

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 $\text{[[1,2], [2,3], [3,4]]}$ 。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。

代码

```
class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        n = len(intervals)
        if n == 0: return 0
        dp = [1] * n
        ans = 1
        intervals.sort(key=lambda a: a[0])

        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)
                    break # 由于是按照开始时间排序的，因此可以剪枝

        return n - max(dp)
```

复杂度分析

- 时间复杂度: $O(n^2)$
- 空间复杂度: $O(n)$

贪心

思路

这道题还有一种贪心的解法，其效率要比动态规划更好。

LIS 也可以用 贪心 + 二分 达到不错的效率。

代码

```
class Solution:
    def lengthOfLIS(self, A: List[int]) -> int:
        d = []
        for s, e in A:
            i = bisect.bisect_left(d, e)
            if i < len(d):
                d[i] = e
            elif not d or d[-1] <= s:
                d.append(e)
        return len(d)
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        n = len(intervals)
        if n == 0: return 0
        ans = 1
        intervals.sort(key=lambda a: a[0])
        return n - self.lengthOfLIS(intervals)
```

复杂度分析

- 时间复杂度: $O(n \log n)$
- 空间复杂度: $O(n)$

参考

LIS 问题都可以用贪心的策略来解决，关于 LIS 问题可参考：

- [穿上衣服我就不认识你了？来聊聊最长上升子序列](#)

题目地址 (881. 救生艇)

<https://leetcode-cn.com/problems/boats-to-save-people/>

入选理由

- 和前面的区间不同，这是另外一种贪心的类型。给大家看看不同的题型 难度也是中等

标签

- 贪心

难度

- 中等

题目描述

第 i 个人的体重为 $\text{people}[i]$ ，每艘船可以承载的最大重量为 limit 。

每艘船最多可同时载两人，但条件是这些人的重量之和最多为 limit 。

返回载到每一个人所需的最小船数。(保证每个人都能被船载)。

示例 1：

输入： $\text{people} = [1,2]$, $\text{limit} = 3$ 输出： 1 解释： 1 艘船载 (1, 2) 示例 2：

输入： $\text{people} = [3,2,2,1]$, $\text{limit} = 3$ 输出： 3 解释： 3 艘船分别载 (1, 2), (2) 和 (3) 示例 3：

输入： $\text{people} = [3,5,3,4]$, $\text{limit} = 5$ 输出： 4 解释： 4 艘船分别载 (3), (3), (4), (5) 提示：

$1 \leq \text{people.length} \leq 50000$ $1 \leq \text{people}[i] \leq \text{limit} \leq 30000$

暴力 DFS (超时)

思路

定义函数 `dfs(people, limit, i, remain)`，其功能是计算从 0 到 i 的 people 在 limit 的限制下，当前一个船还有 remain 位置的情况下需要的最小的船数。那么答案自然就是 $1 + \text{dfs}(\text{people}, \text{limit}, 0,$

`limit)`。

为了防止每次在 `dfs` 中遍历查找是否有可以装下的人，我们可以事先进行一次排序，这样可以达到剪枝的目的。`dfs` 中我们最多有两个选择：

- 装 `people[i]`
- 不装 `people[i]`

之所以说最多有两个选择，是因为存在装不下的情况。

因此我们要做的是 `dfs` 中枚举这两种情况，`dfs` 的深度（也就是递归树的深度）是 `people` 的长度，因此总的时间复杂度就是指数级别。

代码

代码支持：Python3

```
class Solution:
    def numRescueBoats(self, people: List[int], limit: int):
        def dfs(people, limit, i, remain):
            if (i >= len(people)):
                return 0
            # 这个时候需要增加一个新船
            elif (people[i] > remain):
                return 1 + dfs(people, limit, i, limit)
            # 无需增加新船
            return dfs(people, limit, i + 1, remain - people[i])
        people.sort()
        return 1 + dfs(people, limit, 0, limit)
```

复杂度分析

令 n 为 `people` 的长度。

- 时间复杂度： $\mathcal{O}(n \log n)$
- 空间复杂度： $\mathcal{O}(1)$

排序 + 双指针

思路

上面的思路可行，但是太过复杂。而且前面的思路有点像前面我们讲过的 01 背包问题。有没有办法对前面的方法进行优化呢？答案是有的。

由于题目要求船数最少，那显然我们希望每个船都尽可能多装一些重量，这样才能使得结果更优。

每次装人的时候先将最大重量的装上，然后再遍历剩下的人，看剩余容量能否再装下一人，实际上由于题目限定了只能坐两个人，那么我们优先选择较轻的总是没错的，因此轻的更容易被满足并且不会比重的结果差（制定贪心策略）。最后将已经被装上船的人踢出列表，继续按上述策略装，直到所有人都上船。

每次遍历剩下的人以及将人踢出列表的时间复杂度过高，我们可以采用排序 + 双指针的具体策略来完成。因此这道题大的层面上是贪心，具体战术上采用的是排序 + 双指针，这同时也是贪心问题的一个常见的做法。

具体地：

- 先对 people 进行一次排序（不妨进行一次升序）。
- 选择头尾两个人。如果可以同时载就运载这两个人。如果不可行，那么这个重的人和剩下任何人都无法配对，只能自己走了。

采用上面的策略直到全部运走即可。

代码

代码支持：JS

JS Code:

```
var numRescueBoats = function (people, limit) {
    people.sort((a, b) => a - b);
    let ans = 0,
        start = 0,
        end = people.length - 1;
    while (start <= end) {
        if (people[end] + people[start] <= limit) {
            start++;
            end--;
        } else {
            end--;
        }
        ans++;
    }
    return ans;
};
```

Python3 Code:

989. 数组形式的整数加法

```
class Solution:
    def numRescueBoats(self, people: List[int], limit: int):
        res = 0
        l = 0
        r = len(people) - 1
        people.sort()

        while l < r:
            total = people[l] + people[r]
            if total > limit:
                r -= 1
                res += 1
            else:
                r -= 1
                l += 1
                res += 1
        if (l == r):
            return res + 1
        return res
```

复杂度分析

- 时间复杂度: $O(n \log n)$
- 空间复杂度: $O(1)$

入选理由

1. 二叉搜索树是最最适合练习分治的数据结构

标签

- 分治

难度

- 中等

题目地址 (96. 不同的二叉搜索树)

<https://leetcode-cn.com/problems/unique-binary-search-trees/>

题目描述

给定一个整数 n , 求以 $1 \dots n$ 为节点组成的二叉搜索树有多少种?

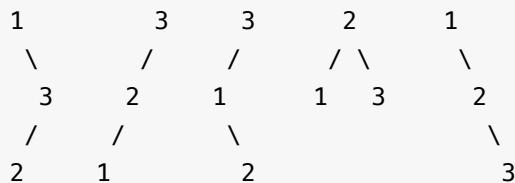
示例:

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:



前置知识

- 二叉搜索树
- 分治

公司

- 阿里

- 腾讯
- 百度
- 字节

岗位信息

- 腾讯（广州） - 安卓 - 社招 - 三面

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 $1 \sim n$ 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 - i)
        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, CPP

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 - i)
        self.visited[n] = res
        return res
```

CPP Code:

```
class Solution {
    vector<int> visited;
    int dp(int n) {
        if (visited[n]) return visited[n];
        int ans = 0;
        for (int i = 0; i < n; ++i) ans += dp(i) * dp(n - i);
        visited[n] = ans;
    }
public:
    int numTrees(int n) {
        visited.assign(n + 1, 0);
        visited[0] = 1;
        return dp(n);
    }
};
```

复杂度分析

989. 数组形式的整数加法

- 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。

入选理由

- 合并 k 个有序数组相信大家都会，那你觉得这是分治么？链表你会么？

标签

- 分治

难度

- 中等

题目地址 (23. 合并 K 个排序链表)

<https://leetcode-cn.com/problems/merge-k-sorted-lists/>

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

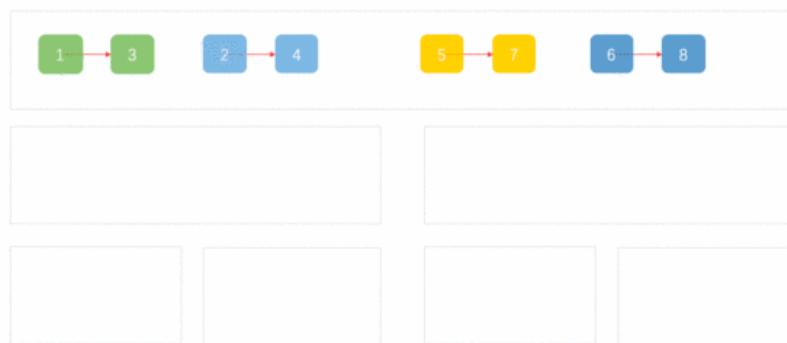
思路

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



©五分钟学算法

(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, CPP

JavaScript Code:

989. 数组形式的整数加法

```
/*
 * @lc app=leetcode id=23 lang=javascript
 *
 * [23] Merge k Sorted Lists
 *
 * https://leetcode.com/problems/merge-k-sorted-lists/description/
 *
 */
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) {
        let l1 = lists[0];
        let l2 = lists[1];
        let result = mergeTwoLists(l1, l2);
        return result;
    }
    let mid = Math.floor(lists.length / 2);
    let leftList = lists.slice(0, mid);
    let rightList = lists.slice(mid);
    let leftResult = mergeKLists(leftList);
    let rightResult = mergeKLists(rightList);
    let mergedList = mergeTwoLists(leftResult, rightResult);
    return mergedList;
}
```

989. 数组形式的整数加法

```
    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:

989. 数组形式的整数加法

```
# 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]) -> ListNode:
        n = len(lists)

        # basic cases
        if length == 0: return None
        if length == 1: return lists[0]
        if length == 2: return self.mergeTwoLists(lists[0], lists[1])

        # divide and conquer if not basic cases
        mid = n // 2
        return self.mergeTwoLists(self.mergeKLists(lists[:mid]),
                                 self.mergeKLists(lists[mid:]))

    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> 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
```

CPP Code:

```

class Solution {
private:
    ListNode* mergeTwoLists(ListNode* a, ListNode* b) {
        ListNode head(0), *tail = &head;
        while (a && b) {
            if (a->val < b->val) { tail->next = a; a = a->next; }
            else { tail->next = b; b = b->next; }
            tail = tail->next;
        }
        tail->next = a ? a : b;
        return head.next;
    }
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if (lists.empty()) return NULL;
        for (int N = lists.size(); N > 1; N = (N + 1) / 2)
            for (int i = 0; i < N / 2; ++i) {
                lists[i] = mergeTwoLists(lists[i], lists[N - 1 - i]);
            }
        return lists[0];
    }
};

```

复杂度分析

- 时间复杂度: $O(kn \log k)$
- 空间复杂度: $O(\log k)$

相关题目

- [88.merge-sorted-array](#)

扩展

这道题其实可以用堆来做，感兴趣的同學尝试一下吧。

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 37K star 啦。大

989. 数组形式的整数加法

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



入选理由

- 一个难度不小的分治。需要一点点数学知识。

标签

- 分治

难度

- 中等

题目地址(932. 漂亮数组)

<https://leetcode-cn.com/problems/beautiful-array/>

题目描述

对于某些固定的 N , 如果数组 A 是整数 $1, 2, \dots, N$ 组成的排列, 使得:

对于每个 $i < j$, 都不存在 k 满足 $i < k < j$ 使得 $A[k] * 2 = A[i]$

那么数组 A 是漂亮数组。

给定 N , 返回任意漂亮数组 A (保证存在一个)。

示例 1:

输入: 4

输出: [2,1,4,3]

示例 2:

输入: 5

输出: [3,1,2,5,4]

提示:

$1 \leq N \leq 1000$

前置知识

- 分治

公司

- 暂无

思路

由数字的奇偶特性, 可知: 奇数 + 偶数 = 奇数。

因此如果要使得: 对于每个 $i < j$, 都不存在 k 满足 $i < k < j$ 使得 $A[k] * 2 = A[i] + A[j]$ 成立, 我们可以令 $A[i]$ 和 $A[j]$ 一个为奇数, 另一个为偶数即可。

另外还有两个非常重要的性质，也是本题的突破口。那就是：

性质 1：如果数组 A 是漂亮数组，那么将 A 中的每一个数 x 进行 $kx + b$ 的映射，其仍然为漂亮数组。其中 k 为不等于 0 的整数， b 为整数。
性质 2：如果数组 A 和 B 分别是不同奇偶性的漂亮数组，那么将 A 和 B 拼接起来仍为漂亮数组。

举个例子。我们要求长度为 N 的漂亮数组。那么一定是有 $N / 2$ 个偶数 和 $N - N / 2$ 个奇数。

这里的除法为地板除。

假设长度为 $N / 2$ 和 $N - N / 2$ 的漂亮数组被计算出来了。那么我们只需要对长度为 $N / 2$ 的漂亮数组通过性质 1 变换成全部为偶数的漂亮数组，并将长度为 $N - N / 2$ 的漂亮数组也通过性质 1 变换成全部为奇数的漂亮数组。接下来利用性质 2 将其进行拼接即可得到一个漂亮数组。

刚才我们假设长度为 $N / 2$ 和 $N - N / 2$ 的漂亮数组被计算出来了，实际上我们并没有计算出来，那么其实可以用同样的方法来计算。其实就是分治，将问题规模缩小了，问题本质不变。递归的终点自然是 $N == 1$ ，此时可直接返回 [1]。

关键点

- 利用性质奇数 + 偶数 = 奇数
- 对问题进行分解

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def beautifulArray(self, N: int) -> List[int]:
        @lru_cache(None)
        def dp(n):
            if n == 1:
                return [1]
            ans = []
            # [1,n] 中奇数比偶数多1或一样
            for a in dp(n - n // 2):
                ans += [a * 2 - 1]
            for b in dp(n // 2):
                ans += [b * 2]
            return ans

        return dp(N)
```

复杂度分析

令 n 为数组长度。

- 时间复杂度: $O(n \log n)$
- 空间复杂度: $O(n + \log n)$

此题解由 [力扣刷题插件](#) 自动生成。

力扣的小伙伴可以[关注我](#)，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。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 40K 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。

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



260. 只出现一次的数字 III

题目地址 (260. 只出现一次的数字 III)

<https://leetcode-cn.com/problems/single-number-iii/>

标签

- 位运算

难度

- 中等

入选理由

- 位运算基本是两者题型。一种是直接考察位运算基础知识和基础 api 操作，另一种是实际应用，这里的应用又以状压压缩为主。因此第一道就是一个位运算的基本题目，另外我总结了好几道基础位运算题目，推荐大家私底下都看看。明天我们是一个状态压缩应用题。

题目描述

给定一个整数数组 `nums`，其中恰好有两个元素只出现一次，其余所有元素均出现两次。

示例：

输入： [1, 2, 1, 3, 2, 5]

输出： [3, 5]

注意：

结果输出的顺序并不重要，对于上面的例子，[5, 3] 也是正确答案。

你的算法应该具有线性时间复杂度。你能否仅使用常数空间复杂度来实现？

前置知识

- 位运算
- 数组
- 哈希表

分析

这个题可以很直观的用哈希表来做，遍历一遍数组存入哈希表，再遍历一遍 key，找到 value 为 1 的那两个数就是最后答案，该解决方案的时间复杂度是线性，但是空间复杂度是\$O(keys)\$，不符合题意。

其实做过该系列的前两道题的应该都知道，本题可以巧用位运算的方式来实现\$O(1)\$空间复杂度，再具体来说使用到了异或(xor)的性质(这个要多做题总结，而不是第一眼看到题就能想出来用位运算)。

首先来再复习一下 xor 的主要性质

$$\begin{array}{ll} 0 \text{ xor } 1 = 1 & 0 \text{ xor } 0 = 0 \\ 1 \text{ xor } 1 = 0 & 1 \text{ xor } 0 = 1 \end{array}$$

也就是说当比较的两个bit不同时， xor 的结果才为1

$$\begin{array}{ll} a \text{ xor } b = b \text{ xor } a & \text{满足交换律} \\ a \text{ xor } a = 0 & \text{与自身异或为0} \end{array}$$

也就是说，如果两个数相同那么必定 xor 没了，而该题说只有两个数出现一次，其他都是两次，再利用 $0 \text{ xor } 1 = 1$ 这条，可得如下解法：

- 将 nums 中所有数异或起来得到数 x，x 必定不为 0，因为相同的两个数都约掉了，相当于那两个只出现了一次的数进行 xor。
- 随便找一个 x 的 bit 为 1 的位置，为 1 就代表这两个出现一次的数在该位置的 bit 不同。
- 这样就可以根据这个为 1 的 bit 位来将原问题分解为两个子问题，子问题的定义是：给定一个数组，该数组只有一个数出现一次，其他数都出现两次。
- 这样就转换为基本的找出只出现一次数的问题了，直接将这个数组所有元素 xor 起来得到的就是答案。
- 为方便求解，本题使用的是低位最早出现 1 的位置。

代码

代码支持：Java,Python,JS,CPP

Java Code:

989. 数组形式的整数加法

```
class Solution {
    public int[] singleNumber(int[] nums) {

        int xor = 0;
        for (int i : nums)
            xor ^= i;

        int mask = 1;
        while ((mask & xor) == 0)
            mask <<= 1;

        int[] res = new int[2];
        for (int i : nums) {

            if ((i & mask) == 0)
                res[0] ^= i;
            else
                res[1] ^= i;
        }
        return res;
    }
}
```

Python Code:

```
class Solution:
    def singleNumber(self, nums: List[int]) -> List[int]:
        xor = a = b = 0
        right_bit = 1
        length = len(nums)
        for i in nums:
            xor ^= i
        while right_bit & xor == 0:
            right_bit <<= 1
        for i in nums:
            if right_bit & i:
                a ^= i
            else:
                b ^= i
        return [a, b]
```

JS Code:

989. 数组形式的整数加法

```
var singleNumber = function (nums) {
    let bitmask = 0;

    for (let n of nums) {
        bitmask ^= n;
    }

    bitmask &= -bitmask;

    const ret = [0, 0];

    for (let n of nums) {
        if ((n & bitmask) === 0) ret[0] ^= n;
        else ret[1] ^= n;
    }

    return ret;
};
```

CPP Code:

```
class Solution {
public:
    vector<int> singleNumber(vector<int>& nums) {
        int ret = 0;
        for (int n : nums)
            ret ^= n;
        int div = 1;
        while ((div & ret) == 0)
            div <= 1;
        int a = 0, b = 0;
        for (int n : nums)
            if (div & n)
                a ^= n;
            else
                b ^= n;
        return vector<int>{a, b};
    }
};
```

复杂度分析

设: \$N\$个数

时间复杂度: \$O(N)\$

空间复杂度: \$O(1)\$

989. 数组形式的整数加法

78. 子集

入选理由

1. 昨天和大家打过预防针了，今天就是一个位运算应用题。建议和我讲义里面的那道状态压缩一起食用。

标签

- 回溯
- 位运算

难度

- 中等

题目地址（78. 子集）

<https://leetcode-cn.com/problems/subsets/>

题目描述

给你一个整数数组 `nums`，数组中的元素 互不相同 。返回该数组所有可能的子集。解集 不能 包含重复的子集。你可以按 任意顺序 返回解集。

示例 1:

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

示例 2:

输入: `nums = [0]`
 输出: `[[],[0]]`

提示:

`1 <= nums.length <= 10`
`-10 <= nums[i] <= 10`
`nums` 中的所有元素 互不相同

前置知识

- 位运算
- 回溯

分析

这道题第一眼是可以用搜索/回溯来做的，每进行一次搜索就把当前结果存入结果集。这种求子集的类型题其实还有另一种做法：

每个元素有两种状态，拿或者不拿，那么如果一共有\$N\$个数，那就一共有 2^N 中可能，也就是有这么多个子集（子集包括全集和空集）。既然每一个数只有两种状态，那么我们不妨用一个 bit 来表示。这样题中的 `[1,2,3]`，我们可以看成一个三个比特的组合：

比如 `0 0 0` 就代表空集，`1 1 1` 就代表全集，`1 0 0` 就代表`[1]`（可正可反）。这样我们就可以进行位操作，`$0 - 2^n - 1$` 的数的二进制数位为 1 的位置，就把对应的元素填入集合中。

PS: `((1 << i) & sign) != 0` 的意思是用第 `i` 位是 1 比特与当前 `sign` 相与，若结果不为 0 就代表第 `i` 位比是 1

进阶：用回溯解法解决该问题

代码

代码支持 Java, Python,CPP,JS

Java Code:

```
class Solution {

    public List<List<Integer>> subsets(int[] nums) {

        List<List<Integer>> res = new LinkedList<>();

        int start = 0, end = 1 << nums.length;

        for (int sign = start; sign < end; sign++) {

            List<Integer> list = new LinkedList<>();

            for (int i = 0; i < nums.length; i++)
                if (((1 << i) & sign) != 0)
                    list.add(nums[i]);

            res.add(list);
        }

        return res;
    }
}
```

Python Code:

```
class Solution:
    def subsets(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """

        res, end = [], 1 << len(nums)
        for sign in range(end):
            subset = []
            for i in range(len(nums)):
                if ((1 << i) & sign) != 0:
                    subset.append(nums[i])
            res.append(subset)
        return res
```

CPP Code:

989. 数组形式的整数加法

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        int n = nums.size();
        vector<int> t;
        vector<vector<int>> res;
        for (int i = 0; i < (1 << n); ++i) {
            t.clear();
            for (int j = 0; j < n; ++j) {
                if (i & (1 << j)) {
                    t.push_back(nums[j]);
                }
            }
            res.push_back(t);
        }
        return res;
    }
};
```

JS Code:

```
var subsets = function (nums) {
    const ans = [];
    const n = nums.length;
    for (let mask = 0; mask < 1 << n; ++mask) {
        const t = [];
        for (let i = 0; i < n; ++i) {
            if (mask & (1 << i)) {
                t.push(nums[i]);
            }
        }
        ans.push(t);
    }
    return ans;
};
```

复杂度分析

令 N 为数组长度

- 时间复杂度: $O(N \cdot 2^N)$
- 空间复杂度: $O(N)$, 最长子集为整个数组长, 不考虑返回结果。

- 实现 Trie (前缀树)
- 677. 键值映射
- 面试题 17.17 多次搜索
- 547. 省份数量
- 924. 尽量减少恶意软件的传播
- 1319. 连通网络的操作次数
- 814 二叉树剪枝
- 39 组合总和
- 40 组合总数 II
- 47 全排列 II
- 28 实现 strStr()
- 28 实现 strStr()
- 215. 数组中的第 K 个最大元素
- 1046. 最后一块石头的重量
- 23. 合并 K 个排序链表
- 451 根据字符出现频率排序
- 378. 有序矩阵中第 K 小的元素
- 1054. 距离相等的条形码
- 1206. 设计跳表
- 二叉树遍历系列
- 反转链表系列
- 位运算系列
- 动态规划系列
- 有效括号系列
- 设计系列
- 前缀和系列
- 排序系列

208. 实现 Trie (前缀树)

入选理由

1. 学习完讲义了，那么就动手撸一个吧。后面的题都建立在手撸前缀树的前提下哦~

标签

- 前缀树

难度

- 中等

题目地址（实现 Trie (前缀树)）

leetcode-cn.com/problems/implement-trie-prefix-tree

题目描述

实现一个 Trie (前缀树)，包含 insert, search, 和 startsWith 这三个操作。

示例:

```
Trie trie = new Trie();  
  
trie.insert("apple"); trie.search("apple"); // 返回 true  
trie.search("app"); // 返回 false  
trie.startsWith("app"); // 返回 true  
trie.insert("app");  
trie.search("app"); // 返回 true 说明:
```

你可以假设所有的输入都是由小写字母 a-z 构成的。保证所有输入均为非空字符串。

前置知识

- 树
- Trie

思路

989. 数组形式的整数加法

大家是否看完了讲义呢，看完了正准备自己动手实现的话，这个题正合适，由于这个题已经说让我们实现一个 Trie，我们也就别想啥其他操作了，老老实实实现一个 Trie 就好，插入和查找及其时间复杂度分析我在讲义里写清楚啦，有啥不清楚的可以回过头看下讲义，这里我贴个我自己实现的 Java 和 Python(两年前写的，直接粘过来 😊)版本的代码吧

代码

代码支持：Java, Python

Java Code:

989. 数组形式的整数加法

```
class Trie {

    TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String word) {
        TrieNode node = root;

        for (int i = 0; i < word.length(); i++) {
            if (node.children[word.charAt(i) - 'a'] == null)
                node.children[word.charAt(i) - 'a'] = new TrieNode();

            node = node.children[word.charAt(i) - 'a'];
            node.preCount++;
        }

        node.count++;
    }

    public boolean search(String word) {
        TrieNode node = root;

        for (int i = 0; i < word.length(); i++) {
            if (node.children[word.charAt(i) - 'a'] == null)
                return false;

            node = node.children[word.charAt(i) - 'a'];
        }

        return node.count > 0;
    }

    public boolean startsWith(String prefix) {
        TrieNode node = root;

        for (int i = 0; i < prefix.length(); i++) {
            if (node.children[prefix.charAt(i) - 'a'] == null)
                return false;
        }
    }
}
```

989. 数组形式的整数加法

```
        node = node.children[prefix.charAt(i) - 'a'];
    }

    return node.preCount > 0;
}

private class TrieNode {

    int count; //表示以该处节点构成的串的个数
    int preCount; //表示以该处节点构成的前缀的字串的个数
    TrieNode[] children;

    TrieNode() {

        children = new TrieNode[26];
        count = 0;
        preCount = 0;
    }
}
}
```

Python Code:

这里我 children 用的字典，因为我不太喜欢 python 里的 ord, chr, 用起来嫌乱，大家可以用 ord,chr 来实现 children

989. 数组形式的整数加法

```
class TrieNode:
    def __init__(self):
        self.count = 0
        self.preCount = 0
        self.children = {}

class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.root = TrieNode()

    def insert(self, word):
        """
        Inserts a word into the trie.
        :type word: str
        :rtype: void
        """
        node = self.root
        for ch in word:
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch]
            node.preCount += 1
        node.count += 1

    def search(self, word):
        """
        Returns if the word is in the trie.
        :type word: str
        :rtype: bool
        """
        node = self.root
        for ch in word:
            if ch not in node.children:
                return False
            node = node.children[ch]
        return node.count > 0

    def startsWith(self, prefix):
        """
        Returns if there is any word in the trie that starts
        :type prefix: str
        :rtype: bool
        """
        node = self.root
```

989. 数组形式的整数加法

```
for ch in prefix:  
    if ch not in node.children:  
        return False  
    node = node.children[ch]  
return node.preCount > 0
```

JS Code:

989. 数组形式的整数加法

```
var Trie = function () {
    this.root = {};
};

/**
 * Inserts a word into the trie.
 * @param {string} word
 * @return {void}
 */
Trie.prototype.insert = function (word) {
    let node = this.root;
    for (const c of word) {
        if (!node[c]) node[c] = {};
        node = node[c];
    }
    node.isEnd = true;
};

/**
 * Returns if the word is in the trie.
 * @param {string} word
 * @return {boolean}
 */
Trie.prototype.search = function (word, node = this.root) {
    for (const c of word) {
        if (!node[c]) return false;
        node = node[c];
    }

    return node.isEnd === true;
};

/**
 * Returns if there is any word in the trie that starts with
 * @param {string} prefix
 * @return {boolean}
 */
Trie.prototype.startsWith = function (prefix, node = this.root) {
    for (const c of prefix) {
        if (!node[c]) return false;
        node = node[c];
    }
    return true;
};
```

CPP Code:

989. 数组形式的整数加法

```
class TrieNode {
public:
    TrieNode *child[26];
    bool isWord;
    // 初始化
    TrieNode(): isWord(false){
        for (auto &c: child) c = nullptr;
    }
};

class Trie {
private:
    TrieNode* root;

public:
    /** Initialize your data structure here. */
    Trie() {
        root = new TrieNode();
    }

    /** Inserts a word into the trie. */
    void insert(string word) {
        TrieNode *p = root;

        for (auto a: word) {
            int index = a - 'a';
            if (!p->child[index]) p->child[index] = new TrieNode();
            p = p->child[index];
        }
        p->isWord = true;
    }

    /** Returns if the word is in the trie. */
    bool search(string word) {
        TrieNode *p = root;

        for (auto a: word) {
            int index = a - 'a';
            if (!p->child[index]) return false;
            p = p->child[index];
        }
        return p->isWord;
    }

    /** Returns if there is any word in the trie that starts
     *  with the given prefix. */
    bool startsWith(string prefix) {
        TrieNode *p = root;
```

```
        for (auto a : prefix) {
            int index = a - 'a';
            if (!p->child[index]) return false;
            p = p->child[index];
        }
        return true;
    }
};
```

复杂度分析

令 n 为待操作的字符串的长度。

- 时间复杂度：创建 Trie: $O(1)$ ，其余操作为 $O(n)$ 。
- 空间复杂度：最坏情况下没有任何前缀，此时空间复杂度为所有操作的单词所占的空间，具体来说就是 $O(\text{字符集大小} * \text{单词总字符数})$ 。
不过随着前缀相同的单词增多，效率会变好。（也就是说如果字符串中的相同前缀比较多，则性能优化明显）

上面的是我常用的板子，直接拿来放到这个题上用就可以了，操作也都挺直观的，其中的 `startsWith` 操作我在讲义里倒是没写，但是大家看一下，`startsWith` 的操作逻辑是不是和 `search` 几乎相同。

自己动手实现好、优化好的 Trie 保存好当作以后的 Trie 板子岂不是美滋滋，不过我这两天的题大家还是从头自己敲吧，等以后再活用板子。

入选理由

- 通过昨天的学习，相信大家已经会手撸前缀树了，那么用前缀树可以解决什么样的题目呢？今天我们就来看看~

标签

- 前缀树

难度

- 中等

题目地址(677. 键值映射)

<https://leetcode-cn.com/problems/map-sum-pairs>

题目描述

实现一个 MapSum 类里的两个方法， insert 和 sum。

对于方法 insert，你将得到一对（字符串， 整数）的键值对。字符串表示键， 整数表示值。如果键已经存在，那么原来的键值对将被替换成新的键值对。

对于方法 sum，你将得到一个表示前缀的字符串，你需要返回所有以该前缀开头的键的值的总和。

示例 1:

输入: insert("apple", 3), 输出: Null
输入: sum("ap"), 输出: 3
输入: insert("app", 2), 输出: Null
输入: sum("ap"), 输出: 5

前置知识

- 哈希表
- Trie
- DFS

哈希表

思路

题目说的简单也明白，就是让我们实现两个方法。

方法一：题目既然都叫“键值映射”了，我们自然而然就可以想到 hashmap，接下来分析是否可行：

- 对于 insert 方法，输入是键值对且键重复覆盖值， hashmap 完美契合。
- 对于 sum 方法，要求是找到所有以给定字符串为前缀的键的值的求和，那我们遍历一遍键不就知道了。

代码

代码支持：Python, Java

```
class MapSum:

    def __init__(self):
        self.m = {}

    def insert(self, key, val):
        self.m[key] = val

    def sum(self, prefix):
        count = 0
        for key in self.m:
            if key.startswith(prefix):
                count += self.m[key]
        return count
```

```

class MapSum {

    Map<String, Integer> map;

    public MapSum() {
        map = new HashMap<>();
    }

    public void insert(String key, int val) {
        map.put(key, val);
    }

    public int sum(String prefix) {
        int count = 0;

        for (String key: map.keySet())
            if(key.startsWith(prefix))
                count += map.get(key);

        return count;
    }
}

```

复杂度分析

- 空间复杂度: $O(N)$, 其中 N 是不重复的 key 的个数
- 时间复杂度: 插入是 $O(1)$, 求和操作是 $O(N * S)$, 其中 N 是目前为止 key 的个数, S 是前缀长度。

前缀树

思路

我们继续考虑, 这个 sum 方法是找所有以 xxx 为前缀的字符串, 那么就想到了 Trie(关键词: 字符串前缀), 那么我们分析一下是否可行:

- 对于 insert 方法, 键值映射这儿 Trie 也可以胜任, 因为我们的 Node 节点我们想怎么设定就怎么设定。
- 对于 sum 方法, 这事就该交给 Trie 来办, 找到指定前缀的结尾所对应 Trie 的 Node, 直接把所有分叉全都遍历一遍不就完事了, 遇到是键的从对应节点里取我们的值就可以了。

代码:

代码支持: Python, Java

989. 数组形式的整数加法

```
class MapSum {

    TrieNode root;

    public MapSum() {
        root = new TrieNode();
    }

    public void insert(String key, int val) {
        TrieNode temp = root;
        for (int i = 0; i < key.length(); i++) {
            if (temp.children[key.charAt(i) - 'a'] == null)
                temp.children[key.charAt(i) - 'a'] = new TrieNode();
            temp = temp.children[key.charAt(i) - 'a'];
        }
        temp.count = val;
    }

    public int sum(String prefix) {
        TrieNode temp = root;
        for (int i = 0; i < prefix.length(); i++) {
            if (temp.children[prefix.charAt(i) - 'a'] == null)
                return 0;
            temp = temp.children[prefix.charAt(i) - 'a'];
        }
        return dfs(temp);
    }

    public int dfs(TrieNode node) {
        int sum = 0;
        for (TrieNode t : node.children)
            if (t != null)
                sum += dfs(t);
        return sum + node.count;
    }
}
```

```
private class TrieNode {  
  
    int count; //表示以该处节点构成的串为前缀的个数  
    TrieNode[] children;  
  
    TrieNode() {  
  
        count = 0;  
        children = new TrieNode[26];  
    }  
}
```

复杂度分析

- 空间复杂度：参考讲义 Trie 复杂度分析
- 时间复杂度：插入操作是线性复杂度，sum 操作最坏情况是 $O(m^n)$ （可以理解成从根结点遍历了所有节点，该题可以将 sum 操作的时间优化成线性，避免 dfs 这种搜索操作，大家可以试试）

这里还是给出优化后的代码供大家参考：

989. 数组形式的整数加法

```
class MapSum {

    TrieNode root;

    public MapSum() {
        root = new TrieNode();
    }

    public void insert(String key, int val) {
        TrieNode temp = root;
        int oldVal = searchValue(key);

        for (int i = 0; i < key.length(); i++) {
            if (temp.children[key.charAt(i) - 'a'] == null)
                temp.children[key.charAt(i) - 'a'] = new Ti
            temp = temp.children[key.charAt(i) - 'a'];
            // update val
            temp.count = temp.count - oldVal + val;
        }

        temp.val = val;
        temp.isWord = true;
    }

    public int searchValue(String key) {
        TrieNode temp = root;
        for (int i = 0; i < key.length(); i++) {
            if (temp.children[key.charAt(i) - 'a'] == null)
                return 0;
            temp = temp.children[key.charAt(i) - 'a'];
        }

        return temp.isWord ? temp.val : 0;
    }

    public int sum(String prefix) {
        TrieNode temp = root;
```

989. 数组形式的整数加法

```
for (int i = 0; i < prefix.length(); i++) {

    if (temp.children[prefix.charAt(i) - 'a'] == null)
        return 0;

    temp = temp.children[prefix.charAt(i) - 'a'];
}

return temp.count;
}

private class TrieNode {

    int count; //表示以该处节点构成的串为前缀的个数
    int val;
    TrieNode[] children;
    boolean isWord;

    TrieNode() {

        count = 0;
        children = new TrieNode[26];
        isWord = false;
        val = 0;
    }
}
}
```

面试题 17.17 多次搜索

题目地址(面试题 17.17 多次搜索)

<https://leetcode-cn.com/problems/multi-search-lcci>

标签

- 前缀树

难度

- 中等

题目描述

给定一个较长字符串 `big` 和一个包含较短字符串的数组 `small`, 设计一个方

示例:

输入:

```
big = "mississippi"  
small = ["is","ppi","hi","sis","i","ssippi"]
```

输出: [[1,4],[8],[],[3],[1,4,7,10],[5]]

提示:

```
0 <= len(big) <= 1000  
0 <= len(small[i]) <= 1000  
small 的总字符数不会超过 100000。  
你可以认为 small 中没有重复字符串。  
所有出现的字符均为英文小写字母。
```

前置知识

- 字符串匹配
- Trie

思路

最清晰直观的方式就是直接暴力：挨个子串检索 → 暴力解法（不建议），不做过多说明。

该题这个情景我们挺常见的，我们打游戏的时候有时候生气骂人发出去的却是被和谐掉了，这就是因为我们发送的文本中包含敏感词，于是把敏感词替换成***，而 Trie 的其中一种作用就是检测敏感词，接下来我们做个分析：

- 拿什么建树?
 - 长句：把长句所有的子串遍历一遍添加到 Trie 中，并且做个下标的记录，这样我们在遍历每个敏感词，查看这个敏感词是否存在与 Trie 中，还记得我们讲义说的 Trie 建树的空间复杂度吧，也就是树越深，复杂度很可能越高，一个长句最长的子串就是它本身，因此该种方法可能会 A 了这个题，但是并不建议使用。
 - 敏感词：把所有的敏感词都添加到 Trie 中，由于敏感词基本上长度都比较短，毕竟是个词，建成的树所消耗的空间理论上远小于用长句建树的空间的。为了方便后面找到对应敏感词所对应的下标，我们可以在 Node 中新增一个 ID 属性。

→ 用敏感词建树

- 如何 check 呢?
 - 建立好一颗由敏感词构成的 Trie。
 - 遍历长句中所有的子串，遇到符合的，直接把起始下表添加到对应敏感词的结果集中去，要注意，我们在遍历一个以某一字符为起始字符的所有子串时，在顺序遍历过程中遇到了某个子串不存在于 Trie，那么就没必要继续遍历了，因为 Trie 中并没有以该子串为 prefix 的敏感词。

→ Trie 解决方案

代码

第一种方法是 Trie 的解决方法，该题说白了也是字符串匹配问题，字符串匹配也很容易想到 KMP，因此第二种方法是 KMP 方法，我仅贴出来供大家查阅，在后续 KMP 专题结束后大家可以回过头来将该题用 KMP 方法解决一遍，最后一种方法是暴力做法。

- Trie

989. 数组形式的整数加法

```
class Solution {

    private Node root = new Node();

    public int[][] multiSearch(String big, String[] smalls)

        int n = smalls.length;
        // 初始化结果集
        List<Integer>[] res = new List[n];
        for(int i = 0 ; i < n ; i++)
            res[i] = new ArrayList<>();
        // 建树
        for(int i = 0 ; i < smalls.length; i++)
            insert(smalls[i], i);

        for(int i = 0 ; i < big.length(); i++){

            Node tmp = root;

            for(int j = i ; j < big.length(); j++){
                //不存在以该串为prefix的敏感词
                if(tmp.children[big.charAt(j) - 'a'] == null)
                    break;
                tmp = tmp.children[big.charAt(j) - 'a'];

                if(tmp.isWord)
                    res[tmp.id].add(i);
            }
        }
        // 返回二维数组
        int[][] ret = new int[n][];

        for(int i = 0 ; i < n ; i++){

            ret[i] = new int[res[i].size()];

            for(int j = 0 ; j < ret[i].length; j++)
                ret[i][j] = res[i].get(j);
        }

        return ret;
    }

    private void insert(String word, int id){

        Node tmp = root;
```

989. 数组形式的整数加法

```
for(int i = 0; i < word.length(); i++){

    if(tmp.children[word.charAt(i) - 'a'] == null)
        tmp.children[word.charAt(i) - 'a'] = new Node();

    tmp = tmp.children[word.charAt(i) - 'a'];
}

tmp.isWord = true;
tmp.id = id;
}

class Node {

    Node[] children;
    boolean isWord;
    int id;

    public Node() {

        children = new Node[26];
        isWord = false;
        id = 0;
    }
}
}
```

- KMP

989. 数组形式的整数加法

```
class Solution {

    public int[][] multiSearch(String big, String[] smalls) {
        int[][] res = new int[smalls.length][];
        List<Integer> cur = new ArrayList<>();
        for (int i = 0; i < smalls.length; i++) {
            String small = smalls[i];
            if (small.length() == 0) {
                res[i] = new int[] {};
                continue;
            }
            // kmp
            int[] next = getNext(small);
            int x = 0, y = 0;
            while (x < big.length() && y < small.length()) {
                if (big.charAt(x) == small.charAt(y)) {
                    x++;
                    y++;
                } else {
                    if (y > 0)
                        y = next[y - 1];
                    else
                        x++;
                }
                if (y == small.length()) {
                    y = next[y - 1];
                    cur.add(x - small.length());
                }
            }
            res[i] = new int[cur.size()];
            for (int j = 0; j < res[i].length; j++)
                res[i][j] = cur.get(j);
            cur.clear();
        }
    }

    private int[] getNext(String s) {
        int[] next = new int[s.length()];
        next[0] = -1;
        int j = 0, k = -1;
        for (int i = 1; i < s.length(); i++) {
            while (k > -1 && s.charAt(i) != s.charAt(k))
                k = next[k];
            if (s.charAt(i) == s.charAt(k))
                next[i] = ++k;
            else
                next[i] = -1;
        }
        return next;
    }
}
```

989. 数组形式的整数加法

```
    }

    return res;
}

public int[] getNext(String pattern) {

    int j = 0;
    int[] next = new int[pattern.length()];

    for (int i = 1; i < pattern.length(); i++) {

        if (pattern.charAt(i) == pattern.charAt(j)) {

            next[i] = j + 1;
            j++;
        } else {

            while (j > 0 && pattern.charAt(j) != pattern.charAt(i))
                j = next[j - 1];

            if (pattern.charAt(j) == pattern.charAt(i))

                next[i] = j + 1;
                j++;
            }
        }
    }

    return next;
}
}
```

- Java 暴力（用库就完事了）

989. 数组形式的整数加法

```
public int[][] multiSearch(String big, String[] smalls) {  
  
    int[][] res = new int[smalls.length][];  
  
    List<Integer> cur = new ArrayList<>();  
  
    for (int i = 0; i < smalls.length; i++) {  
  
        String small = smalls[i];  
  
        if (small.length() == 0) {  
  
            res[i] = new int[]{};  
            continue;  
        }  
  
        int startIdx = 0;  
        while (true) {  
  
            int idx = big.indexOf(small, startIdx);  
            if (idx == -1)  
                break;  
  
            cur.add(idx);  
            startIdx = idx + 1;  
        }  
  
        res[i] = new int[cur.size()];  
        for (int j = 0; j < res[i].length; j++)  
            res[i][j] = cur.get(j);  
  
        cur.clear();  
    }  
  
    return res;  
}
```

复杂度分析

- 时间复杂度: $O(N*K)$, 其中 K 是敏感词中最长单词长度, N 是长句的长度。
- 空间复杂度: $O(S)$, S 为所有匹配成功的位置的个数

本次 Trie 专题结束了, 相信大家对 Trie 有了充分认识, 希望多加练习, 以后活用好这种方便的数据结构, 谢谢大家!

入选理由

- 并查集的一大重要的应用就是 **** (此处略去七个字) , 而这道题就是 **** (此处略去七个字) 。

标签

- 并查集

难度

- 中等

题目地址(547. 省份数量)

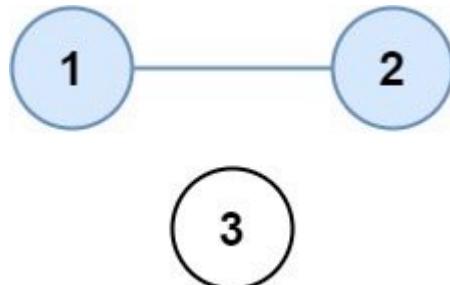
<https://leetcode-cn.com/problems/number-of-provinces/>

题目描述

有 n 个城市，其中一些彼此相连，另一些没有相连。如果城市 a 与城市 b 直属省份 是一组直接或间接相连的城市，组内不含其他没有相连的城市。

给你一个 $n \times n$ 的矩阵 `isConnected`，其中 `isConnected[i][j] = 1` 返回矩阵中 省份 的数量。

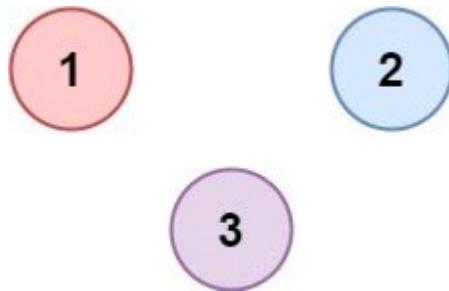
示例 1:



输入: `isConnected = [[1,1,0],[1,1,0],[0,0,1]]`

输出: 2

示例 2:



```
输入: isConnected = [[1,0,0],[0,1,0],[0,0,1]]  
输出: 3
```

提示:

```
1 <= n <= 200  
n == isConnected.length  
n == isConnected[i].length  
isConnected[i][j] 为 1 或 0  
isConnected[i][i] == 1  
isConnected[i][j] == isConnected[j][i]
```

DFS

思路

每个学生看作图中的一个节点，那么问题就转化为求图的强连通分量。这里提供三种解法，这三种方法在求解联通分量个数的时候都可以使用。

首先要讲的是 DFS，这是一种相对直接容易想到且代码较为通俗易懂的解法。

算法流程：

- 选定一个节点，开始深度优先搜索，将遍历到的节点标记为 `visited`，直到无法继续遍历，连通图数目加一
- 选取下一个未遍历的节点，重复上述过程，直到所有节点都被遍历

代码

```

/*
 * @lc app=leetcode.cn id=547 lang=javascript
 *
 * [547] 朋友圈
 */

// @lc code=start
/**
 * DFS
 * @param {number[][]} M
 * @return {number}
 */
var findCircleNum = function (M) {
    const visited = Array.from({ length: M.length }).fill(0);
    let res = 0;
    for (let i = 0; i < visited.length; i++) {
        if (!visited[i]) {
            visited[i] = 1;
            dfs(i);
            res++;
        }
    }
    return res;
}

function dfs(i) {
    for (let j = 0; j < M.length; j++) {
        if (i !== j && !visited[j] && M[i][j]) {
            visited[j] = 1;
            dfs(j);
        }
    }
}
};


```

令 n 为矩阵 M 的大小。

- 时间复杂度: $O(n^2)$
- 空间复杂度: $O(n)$

BFS

思路

和上面思路类似。不过搜索的方向不再是深度优先，而是广度优先。由于我们并不会提前终止，因此使用 bfs 算法效率并没有提升。

代码

```

var findCircleNum = function (M) {
    const visited = Array(M.length).fill(0);
    let res = 0;
    const queue = [];
    for (let i = 0; i < M.length; i++) {
        if (!visited[i]) {
            visited[i] = 1;
            res++;
            queue.push(i);
        }

        while (queue.length) {
            const cur = queue.shift();
            for (let j = 0; j < M.length; j++) {
                if (cur !== j && M[cur][j] && !visited[j]) {
                    queue.push(j);
                    visited[j] = 1;
                }
            }
        }
    }
    return res;
};

```

令 n 为矩阵 M 的大小。

- 时间复杂度: $O(n^2)$
- 空间复杂度: $O(n)$

并查集

思路

使用并查集求联通分量个数再合适不过了。我们只需要在合并的过程中记录一下联通分量的个数即可。

具体来说，我们初始化 n 个联通分量，如果两个人是朋友，那么将其合并。如果合并之前他们已经在一个朋友圈了，那么联通分量个数不会变化，否则联通分量个数会减 1。

算法：

- 初始时，强连通分量为 $count = M.length$
- MAKE-SET, 将每个节点的 $parent$ 指向其本身
- FIND, 并查集常规搜索，添加路径压缩

- UNION(x, y)
 - 如果(x, y)属于同一个子集，返回
 - 如果(x, y)属于不同子集，将两个子集合并， $count--$
- 最终返回 $count$

代码

代码支持： JS

JS Code:

```
var findCircleNum = function (M) {
  let count = M.length;
  let parents = Array.from(M).map((item, index) => index);
  function find(x) {
    if (parents[x] === x) {
      return x;
    }
    return (parents[x] = find(parents[x]));
  }

  function union(x, y) {
    if (find(x) === find(y)) {
      return;
    }
    parents[parents[x]] = parents[y];
    // 两个集合合并，集合数 -1
    count--;
  }

  for (let i = 0; i < M.length; i++) {
    for (let j = i + 1; j < M[i].length; j++) {
      if (M[i][j]) {
        // 如果两个人有边，尝试合并
        union(i, j);
      }
    }
  }

  return count;
};
```

复杂度分析

令 n 为矩阵 M 的大小。

- 时间复杂度：由于仅使用了一种优化（路径压缩），因此时间复杂度为 $O(n \log n)$ 。

989. 数组形式的整数加法

- 空间复杂度： $O(n)$ 。

入选理由

- 和昨天的题目类似，但是又不那么直接？毕竟是困难，要点面子不是？

标签

- 并查集
- DFS

难度

- 中等

题目地址（924. 尽量减少恶意软件的传播）

<https://leetcode-cn.com/problems/minimize-malware-spread>

题目描述

在节点网络中，只有当 $\text{graph}[i][j] = 1$ 时，每个节点 i 能够直接连接到一些节点 initial 最初被恶意软件感染。只要两个节点直接连接，且其中至少假设 $M(\text{initial})$ 是在恶意软件停止传播之后，整个网络中感染恶意软件的最我们可以从初始列表中删除一个节点。如果移除这一节点将最小化 $M(\text{initial})$ 请注意，如果某个节点已从受感染节点的列表 initial 中删除，它以后可能仍

示例 1:

输入: $\text{graph} = [[1,1,0],[1,1,0],[0,0,1]]$, $\text{initial} = [0,1]$
输出: 0
示例 2:

输入: $\text{graph} = [[1,0,0],[0,1,0],[0,0,1]]$, $\text{initial} = [0,2]$
输出: 0
示例 3:

输入: $\text{graph} = [[1,1,1],[1,1,1],[1,1,1]]$, $\text{initial} = [1,2]$
输出: 1

提示:

```
1 < graph.length = graph[0].length <= 300
0 <= graph[i][j] == graph[j][i] <= 1
graph[i][i] == 1
1 <= initial.length < graph.length
0 <= initial[i] < graph.length
```

思路

这道题抽象一下就是在求联通分量

1. 根据 initial 节点去求联通分量
2. 如果两个 initial 节点在同一个联通分量，这两个节点肯定不是答案，因为不管排除哪个，这个联通分量的节点都会被感染
3. 统计只含有一个初始节点的联通分量，找到联通分量中节点数最多的即可，如果有多个联通分量节点数最多，返回含有最小下标初始节点

上述过程就是找联通分量过程，并查集天然适合找联通分量。

代码

989. 数组形式的整数加法

代码支持: JS, Python

JS Code:

989. 数组形式的整数加法

```
var minMalwareSpread = function (graph, initial) {
    const father = Array.from(graph, (v, i) => i);
    function find(v) {
        if (v === father[v]) {
            return v;
        }
        father[v] = find(father[v]);
        return father[v];
    }
    function union(x, y) {
        if (find(x) !== find(y)) {
            father[x] = find(y);
        }
    }
}

for (let i = 0; i < graph.length; i++) {
    for (let j = 0; j < graph[0].length; j++) {
        if (graph[i][j]) {
            union(i, j);
        }
    }
}

initial.sort((a, b) => a - b);

let counts = graph.reduce((acc, cur, index) => {
    let root = find(index);
    if (!acc[root]) {
        acc[root] = 0;
    }
    acc[root]++;
    return acc;
}, {});

let res = initial[0];
let count = -Infinity;

initial
.map((v) => find(v))
.forEach((item, index, arr) => {
    if (arr.indexOf(item) === arr.lastIndexOf(item)) {
        if (count === -Infinity || counts[item] > count) {
            res = initial[index];
            count = counts[item];
        }
    }
});
});
```

989. 数组形式的整数加法

```
    return res;  
};
```

Python Code:

```

class UnionFind:
    def __init__(self):
        self.father = {}
        self.size = {}

    def find(self, x):
        self.father.setdefault(x, x)
        if x != self.father[x]:
            self.father[x] = self.find(self.father[x])
        return self.father[x]

    def union(self, x, y):
        fx, fy = self.find(x), self.find(y)
        if self.size.setdefault(fx, 1) < self.size.setdefault(fy, 1):
            self.father[fx] = fy
            self.size[fy] += self.size[fx]
        elif fx != fy:
            self.father[fy] = fx
            self.size[fx] += self.size[fy]

class Solution:
    def minMalwareSpread(self, graph: List[List[int]], initial):
        uf = UnionFind()

        for i in range(len(graph)):
            for j in range(i, len(graph)):
                if graph[i][j]:
                    uf.union(i, j)

        initial.sort()
        max_size, index, fi = 0, -1, []
        cnt = collections.defaultdict(int)
        for init in initial:
            fi.append(uf.find(init))
            cnt[fi[-1]] += 1
        for i in range(len(initial)):
            if cnt[fi[i]] > 1:
                continue
            if uf.size[fi[i]] > max_size:
                max_size = uf.size[fi[i]]
                index = initial[i]

        return index if index != -1 else initial[0]

```

复杂度分析

令 d 为矩阵 M 的大小。

- 时间复杂度：由于使用了路径压缩和按秩合并，因此时间复杂度为 $O(\log(m \times \text{Alpha}(n)))$ ， n 为合并的次数， m 为查找的次数，这里 Alpha 是 Ackerman 函数的某个反函数
- 空间复杂度： $O(d)$ 。

DFS

思路

正如之前所说，能用并查集通常也能用搜索（BFS 或者 DFS）。

使用 DFS 的思路比较常规，就是从每个点启动一次搜索。

代码

989. 数组形式的整数加法

```
var minMalwareSpread = function (graph, initial) {
    const N = graph.length;
    initial.sort((a, b) => a - b);
    let colors = Array.from({ length: N }).fill(0);
    let curColor = 1;
    // 给联通分量标色
    for (let i = 0; i < N; i++) {
        if (colors[i] === 0) {
            dfs(i, curColor++);
        }
    }

    let counts = Array.from({ length: curColor }).fill(0);
    for (node of initial) {
        counts[colors[node]]++;
    }

    let maybe = [];
    for (node of initial) {
        if (counts[colors[node]] === 1) {
            maybe.push(node);
        }
    }

    counts.fill(0);

    for (let i = 0; i < N; i++) {
        counts[colors[i]]++;
    }

    let res = -1;
    let maxCount = -1;

    for (let node of maybe) {
        if (counts[colors[node]] > maxCount) {
            maxCount = counts[colors[node]];
            res = node;
        }
    }

    if (res === -1) {
        res = Math.min(...initial);
    }

    return res;

    function dfs(start, color) {
        colors[start] = color;
```

989. 数组形式的整数加法

```
for (let i = 0; i < N; i++) {
    if (graph[start][i] === 1 && colors[i] === 0) {
        dfs(i, color);
    }
}
};


```

复杂度分析

令 d 为矩阵 M 的大小, e 为 initial 长度。

- 时间复杂度: 由于使用了排序, 因此排序需要时间为 $\$eloge\$$ 。而 dfs 部分, 由于每个点呗访问最多一次, 因此时间为 $\$O(d^2)\$$
- 空间复杂度: 我们使用了 colors 和 counts , 因此空间复杂度为 $\$O(d)\$$ 。

标签

- 并查集

难度

- 中等

题目地址 (**1319. 连通网络的操作次数**)

<https://leetcode-cn.com/problems/number-of-operations-to-make-network-connected/>

题目描述

989. 数组形式的整数加法

用以太网线缆将 n 台计算机连接成一个网络，计算机的编号从 0 到 $n-1$ 。线：

网络中的任何一台计算机都可以通过网络直接或者间接访问同一个网络中其他任：

给你这个计算机网络的初始布线 `connections`，你可以拔开任意两台直连计算

示例 1:

输入: `n = 4, connections = [[0,1],[0,2],[1,2]]`

输出: 1

解释: 拔下计算机 1 和 2 之间的线缆，并将它插到计算机 1 和 3 上。

示例 2:

输入: `n = 6, connections = [[0,1],[0,2],[0,3],[1,2],[1,3]]`

输出: 2

示例 3:

输入: `n = 6, connections = [[0,1],[0,2],[0,3],[1,2]]`

输出: -1

解释: 线缆数量不足。

示例 4:

输入: `n = 5, connections = [[0,1],[0,2],[3,4],[2,3]]`

输出: 0

提示:

```
1 <= n <= 10^5
1 <= connections.length <= min(n*(n-1)/2, 10^5)
connections[i].length == 2
0 <= connections[i][0], connections[i][1] < n
connections[i][0] != connections[i][1]
```

没有重复的连接。

两台计算机不会通过多条线缆连接。

思路

这题稍微难一点的地方在于问题抽象，不管怎么样，网络总会有部分节点连接形成子网，只要我们找到网络中的子网数目，使得整个网络连通的操作次数其实就是将所有子网  的次数。求子网数量其实就是求图中联通分量的数量，求联通分量可以用 DFS 或者并查集，这里提供并查集解法。

代码

代码支持：JS, Python3, Java

JS Code:

```
/*
 * @param {number} n
 * @param {number[][]} connections
 * @return {number}
 */
var makeConnected = function (n, connections) {
    // 连接 n 台电脑至少需要 n - 1 根线缆
    if (connections.length < n - 1) {
        return -1;
    }
    // 计算联通分量，最小操作次数就是将联通分量链接的次数
    let father = Array.from({ length: n }, (v, i) => i);
    let count = n;
    for (connection of connections) {
        union(...connection);
    }

    return count - 1;

    function find(v) {
        if (father[v] !== v) {
            father[v] = find(father[v]);
        }
        return father[v];
    }

    function union(x, y) {
        if (find(x) !== find(y)) {
            count--;
            father[find(x)] = find(y);
            // 联通分量数减一
        }
    }
};
```

989. 数组形式的整数加法

Python Code:

```
class Solution:
    def makeConnected(self, n: int, connections: List[List[int]]):
        root = [i for i in range(n)]

        def find(p):
            while p != root[p]:
                root[p] = root[root[p]]
                p = root[p]

        return p

        def union(p, q):
            root[find(p)] = find(q)

        have = 0
        for connec in connections:
            a, b = connec
            if find(a) != find(b):
                union(a, b)
            else:
                have += 1

        diff_root = set()
        for i in range(n):
            diff_root.add(find(i))

        return len(diff_root) - 1 if have >= len(diff_root)
```

Java Code:

989. 数组形式的整数加法

```
class Solution {
    public int makeConnected(int n, int[][] connections) {
        if (n - 1 > connections.length) {
            return -1;
        }
        UnionFind unionFind = new UnionFind(n);
        for (int[] connection : connections) {
            unionFind.union(connection[0], connection[1]);
        }
        return unionFind.cnt - 1;
    }

    class UnionFind {
        public int cnt;
        public int[] parents;

        public UnionFind(int n) {
            this.cnt = n;
            this.parents = new int[n];
            for (int i = 0; i < n; i++) {
                parents[i] = i;
            }
        }

        public void union(int x, int y) {
            int x_root = find(x);
            int y_root = find(y);
            if (x_root != y_root) {
                cnt--;
            }
            parents[x_root] = y_root;
        }

        public int find(int x) {
            if (x == parents[x]) {
                return x;
            }
            parents[x] = find(parents[x]);
            return parents[x];
        }
    }
}
```

复杂度分析

令 v 图的点的个数，也就是计算机的个数。

989. 数组形式的整数加法

- 时间复杂度：由于仅仅使用了路径压缩，因此合并与查找时间复杂度为 $O(\log x)$ 和 $O(\log y)$ ， x 和 y 分别为合并与查找的次数。
- 空间复杂度： $O(v)$ 。

标签

- 剪枝

难度

- 中等

入选理由

1. 剪枝通常都是对递归树剪，最典型的就是回溯。而这道题就是树，让你形象化认识剪枝

题目地址（814 二叉树剪枝）

<https://leetcode-cn.com/problems/binary-tree-pruning>

题目描述

给定二叉树根结点 `root`，此外树的每个结点的值要么是 0，要么是 1。

返回移除了所有不包含 1 的子树的原二叉树。

(节点 X 的子树为 X 本身，以及所有 X 的后代。)

示例1：

输入： [1,null,0,0,1]

输出： [1,null,0,null,1]

示例2：

输入： [1,0,1,0,0,0,1]

输出： [1,null,1,null,1]

示例3：

输入： [1,1,0,1,1,0,1,0]

输出： [1,1,0,1,1,null,1]

说明：

给定的二叉树最多有 100 个节点。

每个节点的值只会为 0 或 1

前置知识

- 二叉树
- 递归

思路

这个题可是算真正意义的“剪枝”了，出这个题的主要原因是想让大家理解，其实我们日常使用的各种搜索算法其实和这颗二叉树很像，这个题里让我们剪掉全 0 的子树，这就和我们剪掉重复解或者不可行解非常类似，因此这个题用来自了解搜索空间和剪枝很合适。

说了半天看这道题吧，一般树的题是跑不了递归的，我说一下我做树这种题的初使递归的考虑过程：

- 首先只考虑只有一个根结点的树桩：是 0 返回 null 不是 0 返回这个节点
- 再考虑只有一个根结点和左右两个叶子节点的树：先去看左叶子节点是否是 0，是剪掉，否则留下来，右叶子节点同理，如果左右节点都剪掉了就又回到了第一种情况。
- 泛化上述过程：首先我们去对根结点的左子树修剪，再对右子树修剪，如果左右子树都被剪没了，那就判断根结点是不是也要被剪掉。

上述分析过程很容易抽象出如下递归的代码。

代码

代码支持：Java, Python, JS

Java Code:

```
public TreeNode pruneTree(TreeNode root) {
    if (root == null)
        return null;

    root.left = pruneTree(root.left);
    root.right = pruneTree(root.right);

    return root.val == 0 && root.left == null && root.right == null;
}
```

Python Code:

989. 数组形式的整数加法

```
class Solution(object):
    def pruneTree(self, root):
        def containsOne(node):
            if not node: return False
            left = containsOne(node.left)
            right = containsOne(node.right)
            if not left: node.left = None
            if not right: node.right = None
            return node.val == 1 or left or right

        return root if containsOne(root) else None
```

JS Code:

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {TreeNode}
 */
// [0,null,0,0,0]
var pruneTree = function (root) {
    function dfs(root) {
        if (!root) return 0;
        const l = dfs(root.left);
        const r = dfs(root.right);
        if (l == 0) root.left = null;
        if (r == 0) root.right = null;
        return root.val + l + r;
    }
    ans = new TreeNode(-1);
    ans.left = root;
    dfs(ans);
    return ans.left;
};
```

复杂度分析

- 空间复杂度：没有额外空间使用，因此空间复杂度就是递归栈的最大深度\$O(H)\$，其中 H 是树高。

989. 数组形式的整数加法

- **时间复杂度：**最坏情况就是所有节点都剪掉了，因此时间复杂度是 $O(N)$ ，其中 N 是树节点的个数。

入选理由

1. 熟悉了剪枝的形象化，接下来开始真正的剪枝
2. 回溯的题目基本都需要剪枝，这是回溯的一个考点

标签

- 剪枝
- 回溯

难度

- 中等

题目地址（39 组合总和）

<https://leetcode-cn.com/problems/combination-sum/>

题目描述

989. 数组形式的整数加法

给定一个无重复元素的数组 `candidates` 和一个目标数 `target`，找出 `candidates` 中的数字可以无限制重复被选取。

说明：

所有数字（包括 `target`）都是正整数。

解集不能包含重复的组合。

示例 1：

输入: `candidates = [2,3,6,7]`, `target = 7`,

所求解集为：

```
[  
[7],  
[2,2,3]  
]
```

示例 2：

输入: `candidates = [2,3,5]`, `target = 8`,

所求解集为：

```
[  
[2,2,2,2],  
[2,3,3],  
[3,5]  
]
```

提示：

`1 <= candidates.length <= 30`

`1 <= candidates[i] <= 200`

`candidate` 中的每个元素都是独一无二的。

`1 <= target <= 500`

前置知识

- 剪枝
- 回溯

思路

读完题，首先自然考虑最容易想到的解决方案，遍历数组！但是发现这同一个元素能用无限次，这可咋遍历。

没错，遇到 `for` 循环解决不了的，我们自然的就会想到搜索（回溯递归解决）方法。一个搜索策略+合适的剪枝可以大大提高算法效率哦。

相信回溯方法大家也都不陌生了，直接上个回溯代码：

```

public List<List<Integer>> combinationSum(int[] candidates,
    List<List<Integer>> res = new ArrayList<>();
    List<Integer> list = new LinkedList<>();
    backtrack(res, list, candidates, target);
    return res;
}

public void backtrack(List<List<Integer>> res, List<Integer> list,
    int cur) {
    if (cur < 0)
        return;
    if (cur == 0) {
        res.add(new LinkedList<>(list));
        return;
    }

    for (int i = 0; i < candidates.length; i++) {
        list.add(candidates[i]);
        helpbacktracker(res, list, candidates, cur - candidates[i]);
        list.remove(list.size() - 1);
    }
}

```

开开心心提交，结果发现没过去，尴尬，问题也很直观，就是我们没有去重，比如 2, 2, 3 和 2, 3, 2 这种都会存在于结果集中，那么怎么办呢？我们直接对结果集去重嘛？其实很直观发现，对结果集去重复杂度可不低啊，那么我们可不可以把重复的解剪掉呢？

当然可以

- 我们可以发现每次递归数组都是从头遍历的，并没有对顺序进行任何限制，那么我们不妨就限制一下顺序，比如 3, 4 就只能是 3, 4 不能是 4, 3。
- 那我们递归的时候每次只能在当前的位置往后拿，不就避免了这种无序导致的重复情况了吗。
- 我们在参数中再传入一个 pos，来记录当前位置。
- 注意：因为一个元素可以重复多次，因此我们 pos 没必要每次递归 +1，只限制不取之前的元素就好。

代码

989. 数组形式的整数加法

代码支持: Java, Python, JS

```
public List<List<Integer>> combinationSum(int[] candidates, int target) {
    List<List<Integer>> res = new ArrayList<>();
    List<Integer> list = new LinkedList<>();
    backtrack(res, list, candidates, target, 0);
    return res;
}

public void backtrack(List<List<Integer>> res, List<Integer> list, int[] candidates, int cur, int pos) {
    if (cur < 0)
        return;

    if (cur == 0) {
        res.add(new LinkedList<>(list));
        return;
    }

    for (int i = pos; i < candidates.length; i++) {
        list.add(candidates[i]);
        backtrack(res, list, candidates, cur - candidates[i], i);
        list.remove(list.size() - 1);
    }
}
```

Python Code:

```
class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        def backtrack(ans,tempList, candidates, remain, start):
            if remain < 0:
                return
            elif remain == 0:
                ans.append(tempList.copy())
                return
            # 枚举所有以 i 为开始的部分解空间
            for i in range(start, len(candidates)):
                tempList.append(candidates[i])
                backtrack(ans, tempList, candidates, remain - candidates[i], i)
                tempList.pop()

        ans = []
        backtrack(ans, [], candidates, target, 0)
        return ans;
```

JS Code:

```

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),
        target,
        0
    );
    return list;
};

```

我们仅仅额外利用了一个 pos 参数，就完美的剪掉了重复解。

当然，上述解决方案可能只是把重复的解剪掉了，是否还可以继续剪，比如提前终止搜索？留给大家思考啦。

复杂度分析

- 时间复杂度：该题不是很好分析，我个人分析是最坏情况，也就是没有任何剪枝时 $O(N^{\lceil \text{target}/\min \rceil})$, 其中 N 时候选数组的长度， \min 时数组元素最小值， target/\min 也就是递归栈的最大深度。
- 空间复杂度：递归调用栈的长度不大于 target/\min ，同时用于记录路径信息的 list 长度也不大于 target/\min ，因此空间复杂度为 $O(\text{target}/\min)$

入选理由

1.上一题的进阶版，如果会了上一题，稍微拓展一下你会么？

标签

- 剪枝
- 回溯

难度

- 中等

题目地址 (40 组合总数 II)

<https://leetcode-cn.com/problems/combination-sum-ii/>

题目描述

给定一个数组 `candidates` 和一个目标数 `target`，找出 `candidates` 中 `candidates` 中的每个数字在每个组合中只能使用一次。

说明：

所有数字（包括目标数）都是正整数。

解集不能包含重复的组合。

示例 1：

输入： `candidates = [10,1,2,7,6,1,5]`, `target = 8`,

所求解集为：

```
[  
[1, 7],  
[1, 2, 5],  
[2, 6],  
[1, 1, 6]  
]
```

示例 2：

输入： `candidates = [2,5,2,1,2]`, `target = 5`,

所求解集为：

```
[  
[1,2,2],  
[5]  
]
```

前置知识

- 剪枝
- 数组
- 回溯

思路

套娃题，既然大家都做过了 39，这个题也不难理解，肯定是要用搜索了，那么看一下区别吧：

- 39 中数组无重复元素，40 数组中可能有重复元素。
- 39 一个元素可以用无数次，40 一个元素只能用一次

首先我们大致的搜索过程其实和上一个题没有啥太大差距，把上一个题基础上加个限制，就是每次搜索指针后移一位，这样保证一个元素只用了一次，看代码

989. 数组形式的整数加法

```
public List<List<Integer>> combinationSum(int[] candidates, int target) {
    List<List<Integer>> res = new ArrayList<>();
    List<Integer> list = new LinkedList<>();
    helper(res, list, candidates, target, 0);
    return res;
}

public void helper(List<List<Integer>> res, List<Integer> list, int cur, int pos) {
    if (cur < 0)
        return;

    if (cur == 0) {
        res.add(new LinkedList<>(list));
        return;
    }

    for (int i = pos; i < candidates.length; i++) {
        list.add(candidates[i]);
        helper(res, list, cur - candidates[i], i);
        list.remove(list.size() - 1);
    }
}
```

没问题，提交，发现又错了。。。。。结果一看，怎么还有重复的，我不是都限制 pos 了么：

- 我们限制的 pos 只是限制了元素出现的先后顺序，由于 39 无重复元素，因此可行。
- 在看 40，如果有重复元素，那限制元素出现顺序就不能将重复解剪干净。
- 下面所说的方法是搜索中常用的去重策略：
 - 先将整个数组排好序
 - 在搜索 (dfs) 过程中，若该元素和前一个元素相等，那么因为前一个元素打头的解都已经搜所完毕了，因此没必要在搜这个元素了，故 pass

```
if (i > start && candidates[i] == candidates[i - 1])
    continue;
```

这样我们就把重复的解给剪干净了。

代码

代码支持: Java, Python, JS

Java Code:

```
public List<List<Integer>> combinationSum2(int[] candidates) {
    Arrays.sort(candidates);
    List<List<Integer>> res = new ArrayList<>();
    List<Integer> list = new LinkedList<>();
    helper(res, list, target, candidates, 0);
    return res;
}

public void helper(List<List<Integer>> res, List<Integer> list, int target, int[] candidates, int start) {
    if (target == 0) {
        res.add(new LinkedList<>(list));
        return;
    }

    for (int i = start; i < candidates.length; i++) {
        if (target - candidates[i] >= 0) {
            if (i > start && candidates[i] == candidates[i - 1])
                continue;

            list.add(candidates[i]);
            helper(res, list, target - candidates[i], candidates, i + 1);
            list.remove(list.size() - 1);
        }
    }
}
```

Python Code:

989. 数组形式的整数加法

```
class Solution:
    def combinationSum2(self, candidates, target):
        lenCan = len(candidates)
        if lenCan == 0:
            return []
        candidates.sort()
        path = []
        res = []
        self.backtrack(candidates, target, lenCan, 0, 0, path)
        return res

    def backtrack(self, curCandidates, target, lenCan, curSum, indexBegin, indBegin):
        # 终止条件
        if curSum == target:
            res.append(path.copy())
        for index in range(indexBegin, lenCan):
            nextSum = curSum + curCandidates[index]
            # 减枝操作
            if nextSum > target:
                break
            # 通过减枝避免重复解的出现
            if index > indexBegin and curCandidates[index] == curCandidates[index - 1]:
                continue
            path.append(curCandidates[index])
            # 由于元素只能用一次, 所以indexBegin = index+1
            self.backtrack(curCandidates, target, lenCan, nextSum, index + 1, indexBegin)
            path.pop()
```

JS Code:

```

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++) {
        if (i > start && nums[i] == nums[i - 1]) continue; // 去重
        tempList.push(nums[i]);
        backtrack(list, tempList, nums, remain - nums[i], i + 1);
        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),
        target,
        0
    );
    return list;
};

```

可能我的代码剪的并不是最优，大家可以自行按照思路修改。

复杂度分析

- 时间复杂度：在最坏的情况下，数组中的每个数都不相同，数组中所有数的和不超过 target，那么每个元素有选和不选两种可能，一共就有 2^n 种选择，又因为我们每一个选择，**最多需要 $O(n)$** 的时间 push 到结果中。因此一个粗略的时间复杂度上界为 $O(N \cdot 2^N)$ ，其中 N 是数组长度。更加严格的复杂度意义不大，不再分析。
- 空间复杂度：递归调用栈的长度不大于 $\lceil \frac{\text{target}}{\min} \rceil$ ，同时用于记录路径信息的 list 长度也不大于 $\lceil \frac{\text{target}}{\min} \rceil$ ，因此空间复杂度为 $O(\lceil \frac{\text{target}}{\min} \rceil)$

入选理由

1. 昨天题目的进阶，变了条件你还会么？

标签

- 回溯
- 剪枝

难度

- 中等

题目地址 (47 全排列 II)

<https://leetcode-cn.com/problems/permutations-ii/>

题目描述

给定一个可包含重复数字的序列，返回所有不重复的全排列。

示例：

输入： [1,1,2]

输出：

```
[  
[1,1,2],  
[1,2,1],  
[2,1,1]]
```

前置知识

- 回溯
- 数组
- 剪枝

思路

其实这个题应该和 46. 全排列做对比，因为 46 和 47 和前两天的 39, 40 很相似的，46 题大家有兴趣也可以去自己做。

该题的题干很简单，就是让我输出所有不重复的全排列，为什么全排列需要强调不重复，因为可能含有重复元素。分析：

- 得到的每个全排列都是由数组中所有元素构成的且每个元素只出现一次，因此和前两天得不一样，反而不能去限制顺序（避免剪掉可行解），那么我们就要一个辅助数组 visit 来避免重复使用元素。
- 如何去重的：其实和昨天的也很像，如果单纯用 set 则复杂度过高。
- 下面简单说一下两种剪掉重复解的方式：假设数据是[1,1,2]（这里注意要给数组先排序再 dfs）
 - `nums[i] == nums[i - 1] && visit[i - 1]`: 该种情况是优先取右，举个简单例子，第一个我们从左到右是 1, 1, 2, 这种情况是不可取的，因为当到第二个 1 时候，第一个 1 已经用过了，正相反，当我们从第二个 1 开始的时候，取第二个数也就是 `nums[0]=1` 还没用过，符合条件，故两个 1, 1, 2 只会存下来 1 个。
 - `nums[i] == nums[i - 1] && !visit[i - 1]`: 这个跟第一种过滤方式刚好相反，不过多解释。

代码

代码支持：Java

Java Code:

989. 数组形式的整数加法

```
public List<List<Integer>> permuteUnique(int[] nums) {  
  
    List<List<Integer>> res = new ArrayList<>();  
  
    if (nums == null || nums.length == 0)  
        return res;  
  
    boolean[] visited = new boolean[nums.length];  
    Arrays.sort(nums);  
    dfs(nums, res, new ArrayList<Integer>(), visited);  
  
    return res;  
}  
  
public void dfs(int[] nums, List<List<Integer>> res, List<Integer>  
                tmp) {  
  
    if (tmp.size() == nums.length) {  
  
        res.add(new ArrayList(tmp));  
        return;  
    }  
  
    for (int i = 0; i < nums.length; i++) {  
  
        if (i > 0 && nums[i] == nums[i - 1] && visited[i - 1])  
            continue;  
  
        //backtracking  
        if (!visited[i]) {  
  
            visited[i] = true;  
            tmp.add(nums[i]);  
            dfs(nums, res, tmp, visited);  
            visited[i] = false;  
            tmp.remove(tmp.size() - 1);  
        }  
    }  
}
```

JS Code:

989. 数组形式的整数加法

```
function backtrack(list, nums, tempList, visited) {
    if (tempList.length === nums.length) return list.push(..)
    for (let i = 0; i < nums.length; i++) {
        if (visited[i]) continue;
        // visited[i - 1] 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;
};
```

Python Code:

```
class Solution:
    def backtrack(numbers, pre):
        nonlocal res
        if len(numbers) <= 1:
            res.append(pre + numbers)
            return
        for key, value in enumerate(numbers):
            if value not in numbers[:key]:
                backtrack(numbers[:key] + numbers[key + 1:])

        res = []
        if not len(nums): return []
        backtrack(nums, [])
        return res
```

复杂度分析

- 时间复杂度：由于由 visit 数组的控制使得每递归一次深度-1，因此递归的时间复杂度是 $N(N - 1) \dots 1$ 也就是 $O(N! \text{ op(res)})$ ，其中 N 是数组长度，op 操作即是昨天说的 res.add 算子的时间复杂度。
- 空间复杂度： $O(N * N!)$ ，考虑数组 N 个不重复元素，每一个排列占 $O(N)$ ，共有 $N!$ 个排列。

入选理由

1. 字符串匹配问题经典中的经典，本次专题不要求深度，要求掌握即可
2. 一题两做，本次要求大家用 BF 和 RK 两种方法 AC

标签

- 字符串

难度

- 简单

题目地址 (28 实现 strStr()-BF&RK)

<https://leetcode-cn.com/problems/implement-strstr/>

题目描述

实现 `strStr()` 函数。

给定一个 `haystack` 字符串和一个 `needle` 字符串，在 `haystack` 字符串中找到 `needle` 第一个发生的位置 (从 0 开始)。如果不存在，则返回 -1。

示例 1:

输入: `haystack` = "hello", `needle` = "ll"

输出: 2

示例 2:

输入: `haystack` = "aaaaa", `needle` = "bba"

输出: -1

说明:

当 `needle` 是空字符串时，我们应当返回什么值呢？这是一个在面试中很好的问题。

对于本题而言，当 `needle` 是空字符串时我们应当返回 0 。这与 C 语言的 `strstr` 函数一致。

前置知识

- 滑动窗口
- 字符串

- Hash 运算

暴力法

思路

该题基本上就是字符串匹配问题的入门，选这个题的原因也很简单，一般 KMP&RK 算法出现在面试中的频率相对较低，因此不需要过分考察深度，只需要掌握该算法基本即可。该题稍微注意一下的地方就是待匹配串可能多个符合模式串的子串，我们只需要返回第一次匹配成功的位置即可。

代码

代码支持： Python3

Python3 Code：

```
class Solution:
    def strStr(self, haystack, needle):
        """
        :type haystack: str
        :type needle: str
        :rtype: int
        """

        lenA, lenB = len(haystack), len(needle)
        if not lenB:
            return 0
        if lenB > lenA:
            return -1

        for i in range(lenA - lenB + 1):
            if haystack[i:i + lenB] == needle:
                return i
        return -1
```

复杂度分析

设：待匹配串长为\$N\$，模式串串长为\$M\$

- 时间复杂度: \$O(NM)\$
- 空间复杂度: \$O(1)\$

进阶

- 能否实现查找所有匹配成功的位置？

RK (滚动哈希)

首先我们把 needle 用 hash 算法计算一次哈希值，接下来我们的目标就是在 haystack 中找到一个连续子串使得其哈希值也是 needle 计算出来的哈希值。

由于 haystack 和 needle 最多只有 26 个字符，因此我们可以用 26 进制进制编码。

比如 hi 就编码为 $26 * (\text{ord('h')} - \text{ord('a')}) + \text{ord('i')} - \text{ord('a')}$ ，这就是我们的哈希算法。

于是这个问题可以使用滑动窗口来解决。具体来说：

- 先计算出和 needle 长度一致的哈希值，也就是 haystack[:len(needle)] 的哈希值。
- 然后我们需要移动窗口，每次移动一格。这样哈希值仅需要减去左侧移除窗口的值和右侧移入窗口的值即可。
- 滚动过程发现哈希值和 needle 的哈希值一致，则说明找到了，直接返回。
- 到最后都没有找到，则返回 -1。

这个算法也有一个比较形象的名字（滚动哈希）

思路

代码

代码支持： Java, Python3

Java Code:

989. 数组形式的整数加法

```
class Solution {

    int prime = 101; // 取模的作用是防止溢出以及提供性能。但是有冲突
    int base = 26 // 题目限定了位英文小写，因此总共 26 个字符

    public int strStr(String haystack, String needle) {

        if (haystack == null || needle == null || haystack.length() < needle.length())
            return -1;

        int n = haystack.length(), m = needle.length();

        long pHashVal = initHash(needle, m);
        long tHashVal = initHash(haystack, m);

        for (int i = 0; i <= n - m; i++) {

            if (i > 0 && i <= n - m)
                tHashVal = recalHash(haystack, i - 1, i + m - 1);

            if (pHashVal == tHashVal && isEqual(haystack, i, i + m))
                return i;
        }

        return -1;
    }

    public long initHash(String text, int end) {

        long hashVal = 0;

        for (int i = 0; i < end; i++)
            hashVal += text.charAt(i) * Math.pow(base, i);

        return hashVal%prime;
    }

    public long recalHash(String text, int oldIdx, int newIdx) {

        long newHash = hashVal - text.charAt(oldIdx);
        newHash /= base;
        newHash += text.charAt(newIdx) * Math.pow(base, patLen);

        return newHash%prime;
    }

    public boolean isEqual(String text, String pattern, int start) {
    }
}
```

989. 数组形式的整数加法

```
int end = tStart + pattern.length();
int pStart = 0;

while (tStart < end) {

    if (text.charAt(tStart) != pattern.charAt(pStart))
        return false;

    tStart++;
    pStart++;
}

return true;
}
```

这里 Java 代码稍作解释一下，我这用个 101 的素数太小了，但是因为毕竟在刷题，不用设置过大，如果工程上使用还是要谨慎选取的。

Python3 Code:

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        if not haystack and not needle:
            return 0
        if not haystack or len(haystack) < len(needle):
            return -1
        if not needle:
            return 0
        hash_val = 0
        target = 0
        prime = 101
        for i in range(len(haystack)):
            if i < len(needle):
                hash_val = hash_val * 26 + (ord(haystack[i]) - ord("a"))
                hash_val %= prime
                target = target * 26 + (ord(needle[i]) - ord("a"))
            else:
                hash_val = (
                    hash_val - (ord(haystack[i - len(needle)]) - ord("a")) * 26
                ) * 26 + (ord(haystack[i]) - ord("a"))
                hash_val %= prime
            if i >= len(needle) - 1 and hash_val == target:
                return i - len(needle) + 1
        return 0 if hash_val == target else -1
```

复杂度分析

设：待匹配串长为\$N\$，模式串串长为\$M\$

- 时间复杂度：一般情况是\$O(N+M)\$
- 空间复杂度：\$O(1)\$

入选理由

- 1.字符串匹配问题经典中的经典，本次专题不要求深度，要求掌握即可
- 2.一题两做，本次要求大家用 KMP 方法 AC

标签

- 字符串

难度

- 简单

题目地址 (28 实现 strStr()-KMP)

<https://leetcode-cn.com/problems/implement-strstr/>

题目描述

实现 `strStr()` 函数。

给定一个 `haystack` 字符串和一个 `needle` 字符串，在 `haystack` 字符串中找到 `needle` 第一个发生的位置 (从 0 开始)。如果不存在，则返回 -1。

示例 1:

输入: `haystack = "hello", needle = "ll"`

输出: 2

示例 2:

输入: `haystack = "aaaaa", needle = "bba"`

输出: -1

说明:

当 `needle` 是空字符串时，我们应当返回什么值呢？这是一个在面试中很好的问题。

对于本题而言，当 `needle` 是空字符串时我们应当返回 `0`。这与C语言的 `strstr` 函数一致。

前置知识

- 滑动窗口
- 字符串

- Hash 运算

分析

该题基本上就是字符串匹配问题的入门，选这个题的原因也很简单，一般 KMP&RK 算法出现在面试中的频率相对较低，因此不需要过分考察深度，只需要掌握该算法基本即可。该题稍微注意一下的地方就是待匹配串可能多个符合模式串的子串，我们只需要返回第一次匹配成功的位置即可。

进阶：能否实现查找所有匹配成功的位置？

代码

代码支持：Java, Python3, JS, CPP

Java Code：

989. 数组形式的整数加法

```
class Solution {
    public int strStr(String haystack, String needle) {

        if (needle.length() == 0)
            return 0;

        int i = 0, j = 0;

        int[] next = getNext(needle);

        while (i < haystack.length() && j < needle.length())

            if (haystack.charAt(i) == needle.charAt(j)) {

                i++;
                j++;
            } else {

                if (j > 0)
                    j = next[j - 1];
                else
                    i++;
            }

            if (j == needle.length())
                return i - j;
        }

        return -1;
    }

    public int[] getNext(String pattern) {

        int[] next = new int[pattern.length()];

        int j = 0;
        for (int i = 1; i < pattern.length(); i++) {

            if (pattern.charAt(i) == pattern.charAt(j))
                next[i] = ++j;
            else {

                while (j > 0 && pattern.charAt(j) != pattern.charAt(i))
                    j = next[j - 1];

                if (pattern.charAt(i) == pattern.charAt(j))
                    next[i] = ++j;
            }
        }
    }
}
```

989. 数组形式的整数加法

```
    }

    return next;
}

}
```

Python3 Code:

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        n, m = len(haystack), len(needle)
        if not needle: return 0
        if m > n: return -1

        # 维护一个pattern的next数组
        def KMPNext(needle):
            next = [None] * len(needle)
            j = 0
            for i in range(1, len(needle)):
                while needle[i] != needle[j]:
                    if j > 0:
                        j = next[j - 1]
                    else:
                        next[i] = 0
                        break
                if needle[i] == needle[j]:
                    j += 1
                    next[i] = j
            return next

        next = KMPNext(needle)
        i, j = 0, 0

        while i < n and j < m:
            if haystack[i] == needle[j]:
                i += 1
                j += 1
            else:
                if j > 0:
                    j = next[j - 1]
                else:
                    i += 1
            if j == m:
                return i - j

        return -1
```

989. 数组形式的整数加法

JS Code:

```
var strStr = function (haystack, needle) {
    if (needle.length === 0) return 0;

    const n = haystack.length,
        m = needle.length;
    const s = " " + haystack;
    const p = " " + needle;
    const next = new Array(m + 1).fill(0);

    for (let i = 2, j = 0; i <= m; i++) {
        while (j > 0 && p[i] !== p[j + 1]) j = next[j];
        if (p[i] === p[j + 1]) j++;
        next[i] = j;
    }

    for (let i = 1, j = 0; i <= n; i++) {
        while (j > 0 && s[i] !== p[j + 1]) j = next[j];
        if (s[i] === p[j + 1]) j++;
        if (j === m) return i - m;
    }
    return -1;
};
```

CPP Code:

```

class Solution {
public:
    int strStr(string haystack, string needle) {
        int n = haystack.size(), m = needle.size();
        if (m == 0) {
            return 0;
        }
        vector<int> pi(m);
        for (int i = 1, j = 0; i < m; i++) {
            while (j > 0 && needle[i] != needle[j]) {
                j = pi[j - 1];
            }
            if (needle[i] == needle[j]) {
                j++;
            }
            pi[i] = j;
        }
        for (int i = 0, j = 0; i < n; i++) {
            while (j > 0 && haystack[i] != needle[j]) {
                j = pi[j - 1];
            }
            if (haystack[i] == needle[j]) {
                j++;
            }
            if (j == m) {
                return i - m + 1;
            }
        }
        return -1;
    }
};

```

复杂度分析

设：待匹配串长为\$N\$，模式串串长为\$M\$

时间复杂度：

- BF: \$O(NM)\$
- RK: 若 hash function 选的差，冲突多，则最坏是\$(NM)\$，一般情况是\$O(N+M)\$
- KMP: \$O(N+M)\$

空间复杂度：

- BF: \$O(1)\$
- RK: \$O(1)\$
- KMP: \$O(M)\$

入选理由

- 经典第 K 个最大(小)元素问题
- 堆问题入门

标签

- 堆

难度

- 中等

题目地址(215. 数组中的第 K 个最大元素)

<https://leetcode-cn.com/problems/kth-largest-element-in-an-array/>

题目描述

在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素。

示例 1:

输入: [3,2,1,5,6,4] 和 $k = 2$

输出: 5

示例 2:

输入: [3,2,3,1,2,4,5,5,6] 和 $k = 4$

输出: 4

说明:

你可以假设 k 总是有效的，且 $1 \leq k \leq$ 数组的长度。

前置知识

- 堆
- 排序

解法一（排序）

思路

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)。

当然你也可以从大到小排序，然后直接取第 k 个。

例如：

```
[3,2,1,5,6,4], k = 2
```

第一步。数组排序：

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

第二步。找第 $(n-k)$ 小的数

```
n-k=4, nums[4]=5 (第2大的数)
```

代码

代码支持：Java, Python3, CPP

Java Code：

```
class KthLargestElementSort {

    public int findKthlargest2(int[] nums, int k) {

        Arrays.sort(nums);
        return nums[nums.length - k];
    }
}
```

Python3 Code：

```
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        size = len(nums)
        nums.sort()
        return nums[size - k]
```

CPP Code：

```

#include <iostream>
#include <vector>

using namespace std;

class Solution {
public:
    int findKthLargest(vector<int> &nums, int k) {
        int size = nums.size();
        sort(begin(nums), end(nums));
        return nums[size - k];
    }
};

```

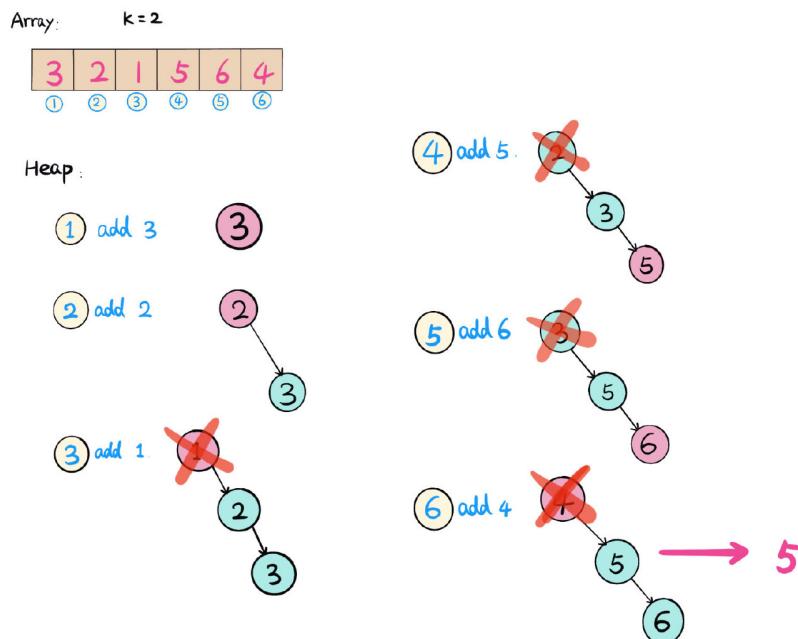
时间和空间复杂度取决于排序算法本身。

解法二 - 小顶堆 (Heap)

思路

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：



这其实就是讲义中提到的固定堆技巧。

代码

使用自带的数据结构。

代码支持: Java, CPP, Python3, JS

Java Code:

```
class Solution {

    public int findKthLargest(int[] nums, int k) {

        PriorityQueue<Integer> pq = new PriorityQueue<>();

        for (int num : nums) {

            if (pq.size() < k)
                pq.offer(num);
            else if (pq.peek() < num) {

                pq.poll();
                pq.offer(num);
            }
        }

        return pq.peek();
    }
}
```

Cpp Code:

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {

        priority_queue<int, vector<int>, less<int>> big_heap;
        for(int i = 0; i < nums.size(); ++i)
            big_heap.push(nums[i]);

        for(int i = 0; i < k-1; ++i)
            big_heap.pop();
        return big_heap.top();
    }
};
```

Python3 Code:

989. 数组形式的整数加法

```
import heapq
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        size = len(nums)

        h = []
        for index in range(k):
            # heapq 默认就是小顶堆
            heapq.heappush(h, nums[index])

        for index in range(k, size):
            if nums[index] > h[0]:
                heapq.heapreplace(h, nums[index])
        return h[0]
```

JS Code:

```
var findKthLargest = function (nums, k) {
    const pq = new MinPriorityQueue();
    i = 0;
    while (i < nums.length) {
        pq.enqueue("x", nums[i]);
        if (pq.size() > k) pq.dequeue();
        i++;
    }
    return pq.dequeue()["priority"];
};
```

手撕实现。

代码支持: Java, CPP

Java Code:

989. 数组形式的整数加法

```
class Solution {

    int count = 0;

    public int findKthLargest(int[] nums, int k) {

        int[] heap = new int[k + 1];
        for (int num: nums)
            add(heap, num, k);
        return heap[1];
    }

    public void add(int[] heap, int elem, int k) {

        if (count == k && heap[1] >= elem)
            return;

        if (count == k && heap[1] < elem) {

            heap[1] = heap[count--];
            siftDown(heap, k, 1);
        }

        heap[++count] = elem;
        siftUp(heap, count);
    }

    public void siftUp(int[] heap, int index) {

        while (index > 1 && heap[index] <= heap[index / 2])

            exch(heap, index, index / 2);
        index /= 2;
    }
}

public void siftDown(int[] heap, int k, int index) {

    while (index * 2 <= k) {

        int j = index * 2;

        if (j < k && heap[j] > heap[j + 1])
            j++;

        if (heap[index] < heap[j])
            break;
    }
}
```

989. 数组形式的整数加法

```
        exch(heap, index, j);
        index = j;
    }
}

public void exch(int[] heap, int x, int y) {

    int temp = heap[x];
    heap[x] = heap[y];
    heap[y] = temp;
}
```

CPP Code:

```

class Solution {
public:
    void maxHeapify(vector<int>& a, int i, int heapSize) {
        int l = i * 2 + 1, r = i * 2 + 2, largest = i;
        if (l < heapSize && a[l] > a[largest]) {
            largest = l;
        }
        if (r < heapSize && a[r] > a[largest]) {
            largest = r;
        }
        if (largest != i) {
            swap(a[i], a[largest]);
            maxHeapify(a, largest, heapSize);
        }
    }

    void buildMaxHeap(vector<int>& a, int heapSize) {
        for (int i = heapSize / 2; i >= 0; --i) {
            maxHeapify(a, i, heapSize);
        }
    }

    int findKthLargest(vector<int>& nums, int k) {
        int heapSize = nums.size();
        buildMaxHeap(nums, heapSize);
        for (int i = nums.size() - 1; i >= nums.size() - k
             swap(nums[0], nums[i]);
             --heapSize;
             maxHeapify(nums, 0, heapSize);
        }
        return nums[0];
    }
};

```

时间复杂度: $O(n * \log k)$, n is array length

空间复杂度: $O(k)$

跟排序相比，以空间换时间。

总结

1. 直接排序很简单，但是时间复杂度过高。
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。本质上是空间换时间。

入选理由

1. 一道难度简单的堆题目，大家来练练手
2. 这道题能很好地体现堆的动态求极值特点。

标签

- 堆

难度

- 简单

题目地址（1046.最后一块石头的重量）

<https://leetcode-cn.com/problems/last-stone-weight/>

题目描述

有一堆石头，每块石头的重量都是正整数。

每一回合，从中选出两块 最重的 石头，然后将它们一起粉碎。假设石头的重量

如果 $x == y$ ，那么两块石头都会被完全粉碎；

如果 $x != y$ ，那么重量为 x 的石头将会完全粉碎，而重量为 y 的石头新重量为 $x + y$ 。最后，最多只会剩下一块石头。返回此石头的重量。如果没有石头剩下，就返回 0。

示例：

输入：[2, 7, 4, 1, 8, 1]

输出：1

解释：

先选出 7 和 8，得到 1，所以数组转换为 [2, 4, 1, 1, 1]，

再选出 2 和 4，得到 2，所以数组转换为 [2, 1, 1, 1]，

接着是 2 和 1，得到 1，所以数组转换为 [1, 1, 1]，

最后选出 1 和 1，得到 0，最终数组转换为 [0]，这就是最后剩下那块石头的重量。

提示：

`1 <= stones.length <= 30`

`1 <= stones[i] <= 1000`

前置知识

- 堆

思路

读完题目可以发现，核心是每次取石头堆中最重的两个石头进行粉碎，如果直接实现排序后，粉碎后的石头重新插入数组，则需要再次排序。因此，不难想到插入删除复杂度都为 $\log N$ 的堆。主要步骤如下：

- 把石头构大顶堆（Java 默认小顶堆，需重写比较器）
- 每次取出前两个（前提 $size \geq 2$ ）进行判断，若重量相同，则抛弃，否则将重量差入堆。
- 当堆中元素不足 2 个则停止并返回相应结果。

代码

代码支持：Java, Python3, JS

989. 数组形式的整数加法

Java Code:

```
class Solution {

    public int lastStoneWeight(int[] stones) {
        PriorityQueue<Integer> pq = new PriorityQueue<>((x, y) -> x - y);
        for (int i : stones)
            pq.offer(i);

        while (pq.size() >= 2) {
            int x = pq.poll();
            int y = pq.poll();

            if (x > y)
                pq.offer(x - y);
        }

        return pq.size() == 1 ? pq.peek() : 0;
    }
}
```

Python3 Code:

```
class Solution:
    def lastStoneWeight(self, stones: List[int]) -> int:
        h = [-stone for stone in stones]
        heapq.heapify(h)

        while len(h) > 1:
            a, b = heapq.heappop(h), heapq.heappop(h)
            if a != b:
                heapq.heappush(h, a - b)
        return -h[0] if h else 0
```

JS Code:

989. 数组形式的整数加法

```
var lastStoneWeight = function (stones) {
    const pq = new MaxPriorityQueue();
    for (const stone of stones) {
        pq.enqueue("x", stone);
    }

    while (pq.size() > 1) {
        const a = pq.dequeue()["priority"];
        const b = pq.dequeue()["priority"];
        if (a > b) {
            pq.enqueue("x", a - b);
        }
    }
    return pq.isEmpty() ? 0 : pq.dequeue()["priority"];
};
```

复杂度分析

令石头个数为 N

- 时间复杂度: $O(N \log N)$
- 空间复杂度: $O(N)$

入选理由

- 复习链表。
- 练习分而治之的基本思想。

标签

- 链表

难度

- 分治

题目地址（23.合并 K 个排序链表）

<https://leetcode-cn.com/problems/merge-k-sorted-lists/>

题目描述

989. 数组形式的整数加法

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

示例 1：

输入: lists = [[1,4,5],[1,3,4],[2,6]]

输出: [1,1,2,3,4,4,5,6]

解释：链表数组如下：

```
[  
    1->4->5,  
    1->3->4,  
    2->6  
]
```

将它们合并到一个有序链表中得到。

1->1->2->3->4->4->5->6

示例 2：

输入: lists = []

输出: []

示例 3：

输入: lists = [[]]

输出: []

提示：

```
k == lists.length  
0 <= k <= 10^4  
0 <= lists[i].length <= 500  
-10^4 <= lists[i][j] <= 10^4  
lists[i] 按 升序 排列  
lists[i].length 的总和不超过 10^4
```

前置知识

- 堆
- 链表
- 分而治之

思路

该题其实就是合并两个有序链表的进阶问题，不难理解。下面介绍几种方法解决：

方法一：暴力

这种方法最好想，就是先拿出来两个合并，合并成一个再去拿一个其他的合并，最终合并为一条。

方法二：堆

想象一下，每次是有 K 个指针对应的节点进行比较并找出最小，接着添加一个新元素进来，是不是有点点像前一天的石头题，我们用堆进行插入和删除可以实现 $O(\log K)$ 复杂度。

- 将 K 个链表的 head 建堆（重写比较器）
- 取出 val 最小的节点并将其 next 节点入堆
- 当堆空的时候，最终链表也就连接好了

进阶-方法三：归并

看了暴力方法之后，是不是可以分而治之，两两合并，类似数组归并排序？请同学们自行实现该方法并分析出时空复杂度。

代码

方法一（Java）

989. 数组形式的整数加法

```
class Solution {

    public ListNode mergeKLists(ListNode[] lists) {

        if (lists == null || lists.length == 0)
            return null;

        ListNode res = null;

        for (int i = 0; i < lists.length; i++)
            res = mergeTwoLists(res, lists[i]);

        return res;
    }

    public ListNode mergeTwoLists(ListNode l1, ListNode l2)

        if (l1 == null || l2 == null)
            return l1 == null ? l2 : l1;

        ListNode fakehead = new ListNode(0), tmp = fakehead;

        while (l1 != null && l2 != null) {

            if (l1.val < l2.val) {

                fakehead.next = l1;
                l1 = l1.next;
            } else {

                fakehead.next = l2;
                l2 = l2.next;
            }

            fakehead = fakehead.next;
        }

        fakehead.next = l1 != null ? l1 : l2;
        return tmp.next;
    }
}
```

方法二（Java）

989. 数组形式的整数加法

```
class Solution {

    public ListNode mergeKLists(ListNode[] lists) {

        PriorityQueue<ListNode> queue = new PriorityQueue<>;
        ListNode fakeHead = new ListNode(0);
        ListNode temp = fakeHead;

        for (ListNode head: lists)
            if (head != null)
                queue.offer(head);

        while (!queue.isEmpty()) {

            ListNode curr = queue.poll();

            if (curr.next != null)
                queue.offer(curr.next);

            temp.next = curr;
            temp = temp.next;
        }
        return fakeHead.next;
    }
}
```

复杂度分析

设：每条链表\$N\$个节点，共\$K\$条

时间复杂度：

- 暴力：第一次合并 res 长\$N\$（初始化），第二次为\$2N\$，以此类推
最终复杂度为 $O((1+2+\dots+K)*N)=O(\frac{(K+1)*K}{2}*N)$ ，等价于 $O(K^2*N)$
- 堆： $O(K*N*\log K)$

空间复杂度：

- 暴力： $O(1)$
- 堆： $O(K)$

入选理由

- 复习哈希表

标签

- 链表
- 堆

难度

- 中等

题目地址（451 根据字符出现频率排序）

<https://leetcode-cn.com/problems/sort-characters-by-frequency/comments/>

题目描述

给定一个字符串，请将字符串里的字符按照出现的频率降序排列。

示例 1：

输入：

"tree"

输出：

"eert"

解释：

'e' 出现两次，'r' 和 't' 都只出现一次。

因此 'e' 必须出现在 'r' 和 't' 之前。此外，"eetr" 也是一个有效的答案。

示例 2：

输入：

"cccaaa"

输出：

"cccaaa"

解释：

'c' 和 'a' 都出现三次。此外，"aaaccc" 也是有效的答案。

注意 "cacaca" 是不正确的，因为相同的字母必须放在一起。

示例 3：

输入：

"Aabb"

输出：

"bbAa"

解释：

此外，"bbaA" 也是一个有效的答案，但 "Aabb" 是不正确的。

注意 'A' 和 'a' 被认为是两种不同的字符。

前置知识

- 排序算法
- 堆
- 哈希表

方法一 直接排序

思路

989. 数组形式的整数加法

基本记录就是统计字符个数，再把个数排个序就可以，采用哈希表的方式存储每个字符的出现次数，然后选择个排序算法。

- 用哈希表统计每个字符的出现次数
- 按每个字符出现的次数调用库函数进行排序
- 将排序后的字符进行拼接

代码

代码支持：Python3, Java, CPP

Python3 Code：

```
class Solution:
    def frequencySort(self, s: str) -> str:

        dict = {}
        for ch in s:
            dict[ch] = dict.get(ch, 0) + 1

        vals = sorted(dict.items(), key=lambda x : x[1], reverse=True)

        res = ""

        for k, v in vals:
            res += k * v

        return res
```

Java Code：

989. 数组形式的整数加法

```
class Solution {

    public String frequencySort(String s) {
        Map<Character, Integer> counter = new HashMap<>();
        for (int i = 0; i < s.length(); i++)
            counter.put(s.charAt(i), counter.getOrDefault(s.charAt(i), 0) + 1);
        List<Map.Entry<Character, Integer>> list = new ArrayList<Map.Entry<Character, Integer>>(counter.entrySet());
        Collections.sort(list, new Comparator<Map.Entry<Character, Integer>>() {
            @Override
            public int compare(Map.Entry<Character, Integer> o1, Map.Entry<Character, Integer> o2) {
                return o2.getValue() - o1.getValue();
            }
        });
        StringBuilder res = new StringBuilder();
        for (Map.Entry<Character, Integer> entry : list)
            for (int i = 0; i < entry.getValue(); i++)
                res.append(entry.getKey());
        return res.toString();
    }
}
```

CPP Code:

```

class Solution {
public:
    string frequencySort(string s) {
        unordered_map<char, int> mp;
        int length = s.length();
        for (auto &ch : s) {
            mp[ch]++;
        }
        vector<pair<char, int>> vec;
        for (auto &it : mp) {
            vec.emplace_back(it);
        }
        sort(vec.begin(), vec.end(), [](const pair<char, int> &a, const pair<char, int> &b) {
            return a.second > b.second;
        });
        string ret;
        for (auto &[ch, num] : vec) {
            for (int i = 0; i < num; i++) {
                ret.push_back(ch);
            }
        }
        return ret;
    }
};

```

复杂度分析

设\$N\$为字符个数, \$K\$为去重字符个数

- 时间复杂度: $O(N+K\log K)$
- 空间复杂度: 直接排序: $O(K)$

进阶: 是否可以实现 $O(n+k)$ 时间复杂度呢 (提示: 桶排序)

方法二 - 堆排序

思路

思路与法一一致, 不过这里不使用库里自带的排序。而是自己手写实现。

具体算法:

- 新建一个大顶堆
- 循环字符串, 将所有元素入堆
- 利用大顶堆的性质, 对堆进行取顶 (得到的是字符串中出现频率最高的字符)
- 拼接字符串, 重复步骤 3 - 4 直到堆为空

代码

代码支持: Java

Java Code:

```

class Solution {

    public String frequencySort(String s) {

        Map<Character, Integer> map = new HashMap();
        PriorityQueue<Character> pq = new PriorityQueue<>();

        for (int i = 0; i < s.length(); i++)
            map.put(s.charAt(i), map.getOrDefault(s.charAt(i), 0) + 1);

        for (char ch : map.keySet())
            pq.offer(ch);

        StringBuilder res = new StringBuilder();

        while(!pq.isEmpty()){

            char c = pq.poll();
            int count = map.get(c);

            for (int i = 0; i < count; i++)
                res.append(c);

        }

        return res.toString();
    }
}

```

复杂度分析

设\$N\$为字符个数, \$K\$为去重字符个数

- 时间复杂度: $O(N+K\log K)$
- 空间复杂度: $O(K)$

入选理由

1. 第 k 大是堆的一个很经典的应用，值得大家掌握之。

题目地址(378. 有序矩阵中第 K 小的元素)

<https://leetcode-cn.com/problems/kth-smallest-element-in-a-sorted-matrix/>

题目描述

给定一个 $n \times n$ 矩阵，其中每行和每列元素均按升序排序，找到矩阵中第 k 小的元素。
请注意，它是排序后的第 k 小元素，而不是第 k 个不同的元素。

示例：

```
matrix = [
    [1, 5, 9],
    [10, 11, 13],
    [12, 13, 15]
],
k = 8,
```

返回 13。

提示：

你可以假设 k 的值永远是有效的， $1 \leq k \leq n^2$ 。

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 $K \log n$ 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。

1	5	9	10	11	12	13	13	15

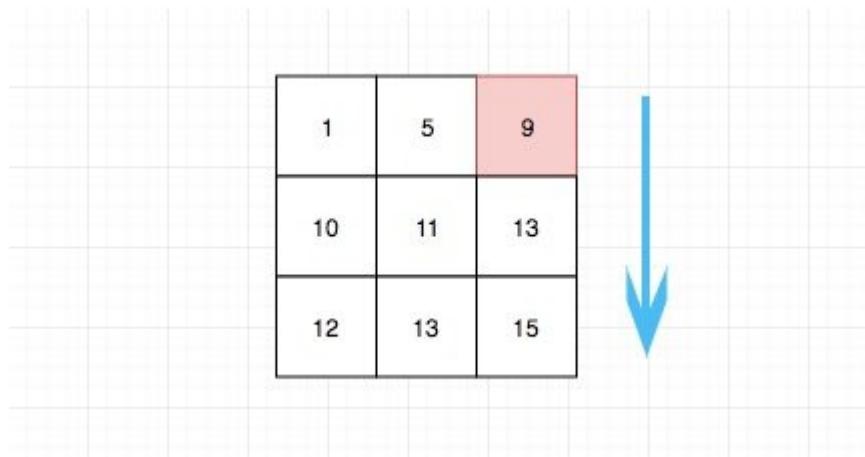
(普通的基于索引判断的二分法)

- 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。

1	5	9						
10	11	13						
12	13	15						

- 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。

989. 数组形式的整数加法



- 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：

start	end	mid	notGreaterCount
1	15	8	2
9	15	12	6
13	15	14	8
13	14	13	8

[378] Kth Smallest Element in a Sorted Matrix

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 matrix. Let us assume element m is the kth smallest number in the matrix. When we are about to reach convergence, if mid=m-1, its count value would be k-1, so we would set lo as (m-1)+1=m, in this case the hi will be m+1 and if mid=m+1, its count value would be k+x-1, so we would set lo as m. To sum up, because the number lo found by binary search fits the condition of being the kth smallest number. The equal sign guarantees there exists and only exists one such number. So lo must be the only element satisfying this element in the matrix.

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

989. 数组形式的整数加法

```
/*
 * @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/
    // 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;  
};
```

复杂度分析

- 时间复杂度：二分查找进行次数为 $O(\log(r - l))$ ，每次操作时间复杂度为 $O(n)$ ，因此总的时间复杂度为 $O(n \log(r - l))$ 。
- 空间复杂度： $O(1)$ 。

相关题目

- [240.search-a-2-d-matrix-ii](#)

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 37K 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



入选理由

- 堆和贪心也有“一腿”，我们来看看。

标签

- 堆
- 贪心

难度

- 中等

题目地址 (1054. 距离相等的条形码)

<https://leetcode-cn.com/problems/distant-barcodes/>

题目描述

在一个仓库里，有一排条形码，其中第 i 个条形码为 $\text{barcodes}[i]$ 。

请你重新排列这些条形码，使其中两个相邻的条形码 不能 相等。 你可以返回任何满足要求的答案。

示例 1:

输入: [1,1,1,2,2,2]

输出: [2,1,2,1,2,1]

示例 2:

输入: [1,1,1,1,2,2,3,3]

输出: [1,3,1,3,2,1,2,1]

提示:

$1 \leq \text{barcodes.length} \leq 10000$

$1 \leq \text{barcodes}[i] \leq 10000$

前置知识

- 堆
- 贪心

分析

该题就是让数组重新排列使得相邻的元素互不相同，思考一下不难发现，如果某个元素超过了 $(len+1)/2$ ，则必定不能实现，而题中说必有答案，因此我们只需要用贪心思想，每次安排元素的时候都取出现次数最多的前两个元素，这样多的解决了，少的自然也能解决掉。而我们每次取出后要更新一下元素频率，这样就会涉及到很多插入删除排序操作，堆再合适不过。

进阶： 你还能想到其他解决方法么？

代码：

Java

989. 数组形式的整数加法

```
class Solution {

    public int[] rearrangeBarcodes(int[] barcodes) {

        Map<Integer, Integer> counter = new HashMap<>();
        PriorityQueue<Integer> pq = new PriorityQueue<>((x, y) -> Math.abs(counter.get(x)) - Math.abs(counter.get(y)));

        for (int num : barcodes)
            counter.put(num, counter.getOrDefault(num, 0) + 1);

        for (int num : counter.keySet())
            pq.offer(num);

        int idx = 0;

        while (pq.size() > 1) {

            int first = pq.poll(), second = pq.poll();

            barcodes[idx++] = first;
            barcodes[idx++] = second;

            counter.put(first, counter.get(first) - 1);
            counter.put(second, counter.get(second) - 1);

            if (counter.get(first) > 0)
                pq.offer(first);
            if (counter.get(second) > 0)
                pq.offer(second);
        }

        if (pq.size() > 0)
            barcodes[idx++] = pq.poll();

        return barcodes;
    }
}
```

复杂度分析

设：元素个数为\$N\$，不重复元素个数为\$K\$

时间复杂度：\$O(N\log K)\$

空间复杂度：\$O(K)\$

入选理由

1. 跳表就一道题，非他莫属了^_^。

作为了解即可，很少有手写跳表的。

标签

- 跳表

难度

- 困难

题目地址 (1206. 设计跳表)

<https://leetcode-cn.com/problems/design-skiplist/>

题目内容

不使用任何库函数，设计一个跳表。

跳表是在 $O(\log(n))$ 时间内完成增加、删除、搜索操作的数据结构。跳表相比于树堆与红黑树，其功能与性能相当，并且跳表的代码长度相较下更短，其设计思想与链表相似。

例如，一个跳表包含 [30, 40, 50, 60, 70, 90]，然后增加 80、45 到跳表中，以下图的方式操作：

Artyom Kalinin [CC BY-SA 3.0], via Wikimedia Commons

跳表中有很多层，每一层是一个短的链表。在第一层的作用下，增加、删除和搜索操作的时间复杂度不超过 $O(n)$ 。跳表的每一个操作的平均时间复杂度是 $O(\log(n))$ ，空间复杂度是 $O(n)$ 。

在本题中，你的设计应该要包含这些函数：

`bool search(int target)` : 返回 `target` 是否存在于跳表中。
`void add(int num)`: 插入一个元素到跳表。
`bool erase(int num)`: 在跳表中删除一个值，如果 `num` 不存在，直接返回 `false`. 如果存在多个 `num`，删除其中任意一个即可。 了解更多：https://en.wikipedia.org/wiki/Skip_list

注意，跳表中可能存在多个相同的值，你的代码需要处理这种情况。

样例:

```
Skiplist skiplist = new Skiplist();

skiplist.add(1);
skiplist.add(2);
skiplist.add(3);
skiplist.search(0);    // 返回 false
skiplist.add(4);
skiplist.search(1);    // 返回 true
skiplist.erase(0);    // 返回 false, 0 不在跳表中
skiplist.erase(1);    // 返回 true
skiplist.search(1);    // 返回 false, 1 已被擦除
约束条件:

0 <= num, target <= 20000
最多调用 50000 次 search, add, 以及 erase操作。
```

思路

因为是设计题，具体参考[讲义](#)，这里说两个注意点

1. 可以想象调表是一个网状结构，每个节点有两个指针，往右和往下
2. 寻找节点的时候，可以想象从最左上角开始往右搜索，网络每层是有序的
3. 插入时记录每层可能需要插入的位置，从下往上逐个插入，是否插入策略由抛硬币决定
4. 删除时，从上往下删，把每层符合要求的节点从当前层链表删除

989. 数组形式的整数加法

```
// 维护一个next指针和down指针
function Node(val, next = null, down = null) {
    this.val = val;
    this.next = next;
    this.down = down;
}

var SkipList = function () {
    this.head = new Node(null);
};

/** 
 * @param {number} target
 * @return {boolean}
 */
SkipList.prototype.search = function (target) {
    let head = this.head;
    while (head) {
        // 链表有序, 从前往后走
        while (head.next && head.next.val < target) {
            head = head.next;
        }
        if (!head.next || head.next.val > target) {
            // 向下走
            head = head.down;
        } else {
            return true;
        }
    }
    return false;
};

/** 
 * @param {number} num
 * @return {void}
 */
SkipList.prototype.add = function (num) {
    const stack = [];
    let cur = this.head;
    // 用一个栈记录每一层可能会插入的位置
    while (cur) {
        while (cur.next && cur.next.val < num) {
            cur = cur.next;
        }
        stack.push(cur);
        cur = cur.down;
    }
}
```

989. 数组形式的整数加法

```
// 用一个标志位记录是否要插入，最底下一层一定需要插入(对应栈顶元素)
let isNeedInsert = true;
let downNode = null;
while (isNeedInsert && stack.length) {
    let pre = stack.pop();
    // 插入元素，维护 next/down 指针
    pre.next = new Node(num, pre.next, downNode);
    downNode = pre.next;
    // 抛硬币确定下一个元素是否需要被添加
    isNeedInsert = Math.random() < 0.5;
}

// 如果人品好，当前所有层都插入了改元素，还需要继续往上插入，则新建
if (isNeedInsert) {
    this.head = new Node(null, new Node(num, null, downNode), downNode);
}
};

/**
 * @param {number} num
 * @return {boolean}
 */
SkipList.prototype.erase = function (num) {
    let head = this.head;
    let seen = false;
    while (head) {
        // 在当前层往前走
        while (head.next && head.next.val < num) {
            head = head.next;
        }
        // 往下走
        if (!head.next || head.next.val > num) {
            head = head.down;
        } else {
            // 找到了该元素
            seen = true;
            // 从当前链表删除
            head.next = head.next.next;
            // 往下
            head = head.down;
        }
    }
    return seen;
};
```

复杂度分析

参考讲义

入选理由

1. 面试频率高
2. 锻炼大家对栈的灵活应用

题目地址 (二叉树遍历系列)

[144. 二叉树的前序遍历\(迭代和递归\)](#) [94. 二叉树的中序遍历\(迭代和递归\)](#)

[145. 二叉树的后序遍历\(迭代和递归\)](#) [102. 二叉树的层序遍历 \(迭代和递归\)](#)

145. 二叉树的后序遍历

递归

思路

1. 遍历左子树
2. 遍历右子树
3. 遍历当前节点

在遍历顺序中，根节点是在最后面的，所以叫做后序遍历。

```
preorder(root.left)
preorder(root.right)
print(root)
```

代码

JavaScript Code

```
var postorderTraversal = function (root, res = []) {
    if (!root) return [];
    root.left && postorderTraversal(root.left, res);
    root.right && postorderTraversal(root.right, res);
    res.push(root.val);
    return res;
};
```

复杂度分析

- 时间复杂度：\$O(n)\$ 将所有节点遍历一遍。

- 空间复杂度: $O(h)$, h 为二叉树的高度 (递归时函数栈帧占用的空间)。

迭代

思路

- 用栈来模拟 dfs
- 先将 root 入栈, pre 指针初始化为空 (用于记录上次 root 指针的指向, 避免环路)
- root 指针指向 root 的左节点直到没有左子树, 此过程中依次将左子树入栈
- 如果 root 有右节点则 root 指向右节点且右节点不等于 pre 指针, 然后循环第 3 步
- 否则将当前节点的值存入结果数组 res 中, pre 指针指向 root, 弹出栈顶元素 node, root 指针指向 node, 循环第 3 步

代码

JavaScript Code

```
var postorderTraversal = function (root) {
    if (!root) return [];
    let data = [],
        stack = [],
        pre = null;
    while (root || stack.length) {
        while (root) {
            stack.push(root);
            root = root.left;
        }
        root = stack.pop();
        if (root.right && root.right !== pre) {
            stack.push(root);
            root = root.right;
        } else {
            data.push(root.val);
            pre = root;
            root = null;
        }
    }
    return data;
};
```

复杂度分析

- 时间复杂度: $O(n)$ 将所有节点遍历一遍。

- 空间复杂度: $O(h)$, h 为二叉树的高度 (类似于使用递归时占用的函数栈帧)。

双色球

代码

JavaScript Code

```
var postorderTraversal = function (root) {
    if (!root) return [];
    const WHITE = 1,
        GREAY = 0;
    let data = [],
        stack = [[WHITE, root]];
    while (stack.length) {
        let [color, node] = stack.pop();
        if (color === WHITE) {
            stack.push([GREAY, node]);
            node.right && stack.push([WHITE, node.right]);
            node.left && stack.push([WHITE, node.left]);
        }
        if (color === GREAY) {
            data.push(node.val);
        }
    }
    return data;
};
```

扩展

前序与中序遍历与后序遍历的思路与后序遍历一样都是 dfs, 只是某些步骤做一些调整, 这两种遍历就不再赘述。

102.二叉树的层序遍历

迭代

思路

1. 使用一个数组 `level` 用来存当前层的所有节点
2. 遍历该数组, 同时将下一层的节点存入 `newLevel` 数组
3. 令 `level` 等于 `newLevel` 循环第一步直到 `level` 为空

代码

JavaScript Code

```

var levelOrder = function (root) {
    if (!root) return [];
    let res = [],
        level = [root];
    while (level.length) {
        let newLevel = [],
            levelVal = [];
        level.forEach((item) => {
            levelVal.push(item.val);
            if (item.left) newLevel.push(item.left);
            if (item.right) newLevel.push(item.right);
        });
        res.push(levelVal);
        level = newLevel;
    }
    return res;
};

```

复杂度分析

- 时间复杂度: $O(n)$, n 为树的节点数。
- 空间复杂度: $O(h)$, h 为每层的节点个数。

递归

思路

- 使用一个标志位标志当前是树的第 n 层。
- 遍历左子树
- 遍历右子树
- 将当前节点存入第 n 层的结果数组中

代码

JavaScript Code

```

var postorderTraversal = function (root, res = []) {
    if (!root) return [];
    root.left && postorderTraversal(root.left, res);
    root.right && postorderTraversal(root.right, res);
    res.push(root.val);
    return res;
};

```

复杂度分析

- 时间复杂度: $O(n)$, n 为二叉树的节点数。
- 空间复杂度: $O(h)$, h 是二叉树的深度 (递归产生的函数调用栈)。

入选理由

- 面试考察频率高

题目地址 (反转链表系列)

[206. 反转链表](#) [92. 反转链表 II](#) [25. K 个一组翻转链表](#)

前置知识

- 链表的翻转

这个直接上模板，有不清楚的地方可以再复习下链表的讲义

伪代码：

```
当前指针 = 头指针
前一个节点 = null;
while 当前指针不为空 {
   下一个节点 = 当前指针.next;
   当前指针.next = 前一个节点
   前一个节点 = 当前指针
   当前指针 = 下一个节点
}
return 前一个节点;
```

JS 代码参考：

```
let cur = head;
let pre = null;
while (cur) {
    const 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)

- 递归

迭代解法

思路

1. 定义一个翻转 k 个节点的辅助函数
2. 遍历链表按 k 个一组进行翻转
3. 每次翻转后注意指针的指向避免形成环

代码

JavaScript Code

```

var reverseKGroup = function (head, k) {
    if (!head || !head.next) return head;
    let reverseList = (head, k) => {
        let pre = null;
        while (head && k--) {
            let nextNode = head.next;
            head.next = pre;
            pre = head;
            head = nextNode;
        }
        return pre;
    };
    let index = 0,
        preLinkEnd = null,
        start = (res = head);
    while (head && ++index) {
        let nextNode = head.next;
        if (index % k == 0) {
            reverseList(start, k);
            if (!preLinkEnd) {
                res = head;
            } else {
                preLinkEnd.next = head;
            }
            preLinkEnd = start;
            start = nextNode;
        }
        head = nextNode;
    }
    preLinkEnd.next = start;
    return res;
};

```

复杂度分析

- 时间复杂度: $O(n)$, 最多将所有节点遍历 2 遍。
- 空间复杂度: $O(1)$

递归

思路

- 如果当前链表的长度小于 k 直接返回 (递归的边界条件)
- 将当前链表的前 k 个进行翻转
- 递归执行第一步, 并将第 k 个节点的 `next` 指针指向递归返回的结果

代码

JavaScript Code

```
var reverseKGroup = function (head, k) {
    if (!head || !head.next) return head;
    let currentNode = head,
        num = k;
    while (num--) {
        if (!currentNode) return head;
        currentNode = currentNode.next;
    }
    num = k;
    currentNode = head;
    let preNode = null;
    while (num--) {
        let nextNode = currentNode.next;
        currentNode.next = preNode;
        preNode = currentNode;
        currentNode = nextNode;
    }
    head.next = reverseKGroup(currentNode, k);
    return preNode;
};
```

复杂度分析

- 时间复杂度: $O(n)$, 最多将所有节点遍历 2 遍。
- 空间复杂度: $O(h)$, h 为链表的分组数量 (递归时占用的函数栈帧)。

扩展

剩下的两道题

989. 数组形式的整数加法

- 1. 反转链表
- 2. 反转链表 II

也是直接套链表翻转的模板，注意处理一下特殊的边界条件即可，这里就不再赘述

入选理由

1. 考察大家对位运算的灵应用，有的时候解题用位运算这种“奇淫巧计”会有很不错的效果

题目地址（位运算系列）

- [136. 只出现一次的数字 1](#)
- [137. 只出现一次的数字 2](#)
- [645. 错误的集合](#)
- [260. 只出现一次的数字 3](#)

学有余力的推荐再做一下这几道：

- [190. 颠倒二进制位](#) (简单)
- [191. 位 1 的个数](#) (简单)
- [338. 比特位计数](#) (中等)
- [1072. 按列翻转得到最大值等行数](#) (中等)

思路

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 $a \wedge b$ 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：

$$a \wedge b \wedge c = a \wedge c \wedge b$$

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} 。

989. 数组形式的整数加法

之所以这样不会有错误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。

989. 数组形式的整数加法

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。

```
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

- [190. 颠倒二进制位](#) (简单)
- [191. 位 1 的个数](#) (简单)
- [338. 比特位计数](#) (中等)
- [1072. 按列翻转得到最大值等行数](#) (中等)

989. 数组形式的整数加法

更多题解可以访问我的 LeetCode 题解仓库：
<https://github.com/azl397985856/leetcode>。目前已经 38K 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。

入选理由

1. 面试考察的重点题型

题目地址 (动态规划系列)

[70. 爬楼梯](#) [62. 不同路径](#) [121. 买卖股票的最佳时机](#) [122. 买卖股票的最佳时机 II](#) [123. 买卖股票的最佳时机 III](#) [188. 买卖股票的最佳时机 IV](#) [309. 最佳买卖股票时机含冷冻期](#) [714. 买卖股票的最佳时机含手续费](#)

70. 爬楼梯

题目描述

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定 n 是一个正整数。

示例 1：

输入： 2

输出： 2

解释： 有两种方法可以爬到楼顶。

1. 1 阶 + 1 阶
2. 2 阶

示例 2：

输入： 3

输出： 3

解释： 有三种方法可以爬到楼顶。

1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶

思路

第 n 个台阶只能由第 $n - 1$ 个台阶或者第 $n - 2$ 个台阶处到达，所以 $f(n) = f(n - 1) + f(n - 2)$

代码

JS Code:

```
var climbStairs = function (n) {
    if (n <= 2) return n;
    let res = 2,
        pre = 1;
    while (n-- > 2) {
        let temp = res;
        res += pre;
        pre = temp;
    }
    return res;
};
```

复杂度分析

- 时间复杂度: $O(n)$
- 空间复杂度: $O(1)$

62. 不同路径

题目描述

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

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

问总共有多少条不同的路径？



例如，上图是一个 7×3 的网格。有多少可能的路径？

示例 1：

输入： $m = 3, n = 2$

输出： 3

解释：

从左上角开始，总共有 3 条路径可以到达右下角。

1. 向右 -> 向右 -> 向下
2. 向右 -> 向下 -> 向右
3. 向下 -> 向右 -> 向右

示例 2：

输入： $m = 7, n = 3$

输出： 28

提示：

$1 \leq m, n \leq 100$

题目数据保证答案小于等于 $2 * 10^9$

前置知识

- 排列组合
- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

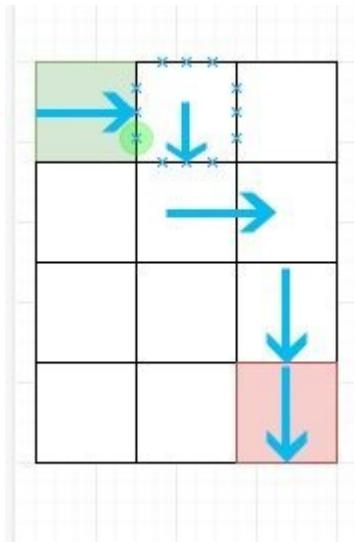
思路

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。

$$C_n^r = \frac{n!}{r!(n-r)!}$$

而这道题我们也可以用动态规划来解。其实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坏事。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。



这不就是二维平面的爬楼梯么？和爬楼梯又有什么不同呢？

代码大概是：

Python Code:

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        d = [[1] * n for _ in range(m)]

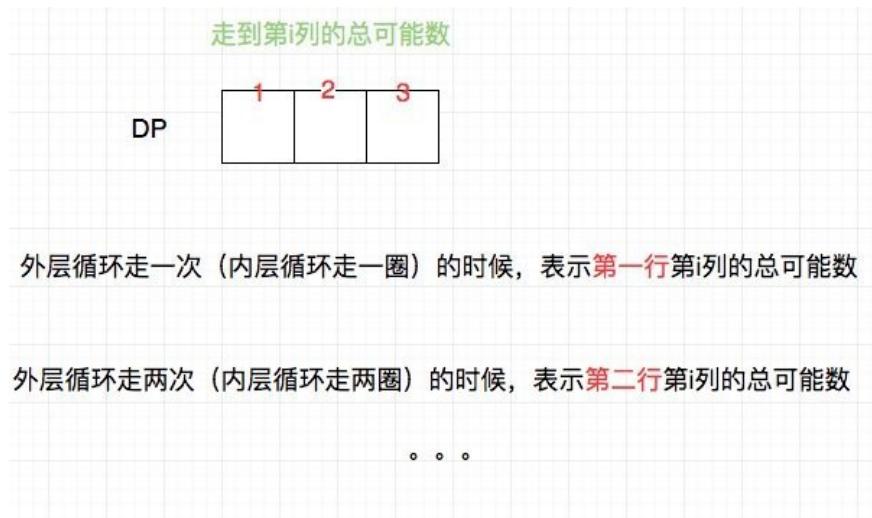
        for col in range(1, m):
            for row in range(1, n):
                d[col][row] = d[col - 1][row] + d[col][row - 1]

        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
        return self.uniquePaths(m - 1, n) + self.uniquePaths(m, n - 1)
```

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

989. 数组形式的整数加法

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

989. 数组形式的整数加法

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

股票系列

由于篇幅原因，股票系列可以在我的 [Github](#) 中找相关的文章。

入选理由

- 深入考察了对栈的灵活应用
- 这两道题我本人都在真实面试中被考察过

题目地址（有效括号系列）

[20. 有效的括号](https://leetcode-cn.com/problems/longest-valid-parentheses/) [32. 最长有效括号](https://leetcode-cn.com/problems/longest-valid-parentheses/)

32. 最长有效括号

<https://leetcode-cn.com/problems/longest-valid-parentheses/>

题目描述

给定一个只包含 '(' 和 ')' 的字符串，找出最长的包含有效括号的子串的长度。

示例 1：

输入： "()"

输出： 2

解释： 最长有效括号子串为 "()"

示例 2：

输入： ")()())"

输出： 4

解释： 最长有效括号子串为 "()()"

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

```
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的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 - 1]);
            }
        }
    }
    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

989. 数组形式的整数加法

Java Code:

```
public class Solution {
    public int longestValidParentheses(String s) {
        int left = 0, right = 0, maxlen = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(') {
                left++;
            } else {
                right++;
            }
            if (left == right) {
                maxlen = Math.max(maxlen, left + right);
            }
            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) {
                maxlen = Math.max(maxlen, left + right);
            }
            if (left > right) {
                left = right = 0;
            }
        }
        return maxlen;
    }
}
```

Python3 Code:

```

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.

```
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: $dp[1] = dp[2] = 0$;

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: $i-1-dp[i-1] \geq 1$, 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:

s索引	0	1	2	3	4	5	6
s	(())	())
dp	0	0	0	0	0	0	0
dp	0	0	0	0	0	0	0
dp	0	0	0	0	0	0	0
dp	0	0	0	2	0	0	0
dp	0	0	0	2	4	0	0
dp	0	0	0	2	4	0	0
dp	0	0	0	2	4	0	6
dp索引	0	1	2	3	4	5	6

描述中, i 为 dp 的索引, 对应的 s 索引为 $i-1$, 第 i 个字符也表示 $s[i-1]$

- $i=0$, 初始化 $dp[0]$ 为 0, 表示空字符串的最长有效括号对长度为 0
- $i=1$, $s[i-1]$ 为 '(', 根据特征 2, 有效括号对不可能以 '(' 结尾, 所以 $dp[1]$ 为 0
- $i=2$, 同 $i=1$, 可得 $dp[2]$ 也为 0
- $i=3$, $s[i-1]$ 为 ')', 根据特征 3, 需要对比 $s[i-1]$ 和 $s[i-2-dp[i-1]]$, 即: $s[2]$ 和 $s[1]$, 发现是配对的, 则 $dp[3]=dp[1]+2$, 所以 $dp[3]$ 为 2, 根据特征 4, 需要拼接字符串, $dp[3] += dp[1-dp[3]]$, 发现仍是 2
- $i=4$, $s[i-1]$ 为 ')', 根据特征 3, 需要对比 $s[i-1]$ 和 $s[i-2-dp[i-1]]$, 即: $s[2]$ 和 $s[1]$, 发现是配对的, 则 $dp[4]=dp[1]+2$, 所以 $dp[4]$ 为 2, 根据特征 4, 需要拼接字符串, $dp[4] += dp[1-dp[4]]$, 发现仍是 2
- $i=5$, 同 $i=1$, 可得 $dp[5]$ 也为 0
- $i=6$, $s[i-1]$ 为 ')', 根据特征 3, 需要对比 $s[i-1]$ 和 $s[i-2-dp[i-1]]$, 即: $s[5]$ 和 $s[4]$, 发现是配对的, 则 $dp[6]=dp[1]+2$, 所以 $dp[6]$ 为 2, 根据特征 4, 需要拼接字符串, $dp[6] += dp[1-dp[6]]$, 发现变为 6
- 最终, 可得最长有效括号对的长度为 6

代码

Python Code:

```

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

```

复杂度分析

- 时间复杂度: $O(N)$
- 空间复杂度: $O(N)$

关键点解析

- 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: $i-1 \geq dp[i-1]$, 但是 $i-2$ 不一定大于 $dp[i-1]$, 因此, 需要检查越界;
- 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: $i \geq dp[i]$, 所以 $dp[i-dp[i]]$ 的边界情况是 $dp[0]$;

相关题目

- [20.valid-parentheses](#)

扩展

- 如果判断的不仅仅只有'('和')', 还有'[', ']', '{'和'}', 该怎么办?
- 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?

989. 数组形式的整数加法

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 37K 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。



设计系列

入选理由

- 面试时考察对数据结构的实际应用与理解，避免了“刷题家”们背题大法
- 同时也考察了大家的代码功底，在实现算法的同时，也要把代码写的简洁优美，具有可维护性与健壮性

题目地址（设计系列）

[剑指 Offer 09. 用两个栈实现队列](#) [641. 设计循环双端队列](#) [146. LRU 缓存机制](#) [1206. 设计跳表](#)

剑指 Offer 09. 用两个栈实现队列

思路

维护两个栈，一个主栈用来存放数据，另外一个作为辅助栈

- 当 push 数据时，直接 push 到主栈就好了
- 当 pop 时，看辅助栈内有没有数据，有就直接 pop
- 辅助栈没数据时就将主栈的数据依次 push 进辅助栈，然后再 pop

代码

JavaScript Code

```

var CQueue = function () {
    this.data = [];
    this.data2 = [];
};

CQueue.prototype.appendTail = function (value) {
    this.data.push(value);
};

CQueue.prototype.deleteHead = function () {
    if (this.data2.length) {
        return this.data2.pop();
    }
    if (this.data.length) {
        while (this.data.length) {
            this.data2.push(this.data.pop());
        }
        return this.data2.pop();
    }
    return -1;
};

```

复杂度分析

- 时间复杂度：\$O(1)\$ push 很明显为 \$O(1)\$, pop 时步骤 3 的复杂度为 \$O(n)\$,但是辅助栈里的这 \$n\$ 个元素后面 pop 的复杂度为 \$O(1)\$，所以平均为 \$O(1)\$。
- 空间复杂度：\$O(n)\$。

641. 设计循环双端队列

思路

需要频繁进行增删，并且还要元素有序，所以直接使用双向链表（方便进行增删操作）

1. 为了逻辑上的统一，设置一个空的头节点与尾节点
2. 内部再设置一个变量存当前节点个数（每次对节点增删时同步更新），方便判满与判空
3. 剩下的操作都是链表的插入删除操作，如果有不清楚的地方可以去复习下链表的讲义

代码

989. 数组形式的整数加法

JavaScript Code

989. 数组形式的整数加法

```
let Node = function (val) {
    this.val = val;
    this.pre = this.next = null;
};

var MyCircularDeque = function (k) {
    this.maxSize = k;
    this.currentSize = 0;
    this.head = new Node();
    this.tail = new Node();
    this.head.next = this.tail;
    this.head.pre = this.tail;
    this.tail.next = this.head;
    this.tail.pre = this.head;
};

MyCircularDeque.prototype.insertFront = function (value) {
    if (this.isFull()) {
        return false;
    }
    let newNode = new Node(value);
    this.head.next.pre = newNode;
    newNode.next = this.head.next;
    newNode.pre = this.head;
    this.head.next = newNode;
    this.currentSize++;
    return true;
};

MyCircularDeque.prototype.insertLast = function (value) {
    if (this.isFull()) {
        return false;
    }
    let newNode = new Node(value);
    this.tail.pre.next = newNode;
    newNode.pre = this.tail.pre;
    newNode.next = this.tail;
    this.tail.pre = newNode;
    this.currentSize++;
    return true;
};

MyCircularDeque.prototype.deleteFront = function () {
    if (this.isEmpty()) {
        return false;
    }
    this.head = this.head.next;
    this.currentSize--;
}
```

```
        return true;
    };

MyCircularDeque.prototype.deleteLast = function () {
    if (this.isEmpty()) {
        return false;
    }
    this.tail = this.tail.pre;
    this.currentSize--;
    return true;
};

MyCircularDeque.prototype.getFront = function () {
    return this.isEmpty() ? -1 : this.head.next.val;
};

MyCircularDeque.prototype.getRear = function () {
    return this.isEmpty() ? -1 : this.tail.pre.val;
};

MyCircularDeque.prototype.isEmpty = function () {
    return !this.currentSize;
};

MyCircularDeque.prototype.isFull = function () {
    return this.currentSize === this.maxSize;
};
```

复杂度分析

- 时间复杂度: $O(1)$ 。
- 空间复杂度: $O(n)$ 。

146. LRU 缓存机制

哈希法

思路

1. 确定需要使用的数据结构

i. 根据题目要求,存储的数据需要保证顺序关系(逻辑层面) ==> 使用数组,链表等保证顺序关系

989. 数组形式的整数加法

ii. 同时需要对数据进行频繁的增删, 时间复杂度 $O(1) \Rightarrow$ 使用链表等

iii. 对数据进行读取时, 时间复杂度 $O(1) \Rightarrow$ 使用哈希表

最终采取双向链表 + 哈希表

- i. 双向链表按最后一次访问的时间的顺序进行排列, 链表头部为最近访问的节点
- ii. 哈希表, 以关键字为键, 以链表节点的地址为值

2. put 操作

通过哈希表, 查看传入的关键字对应的链表节点, 是否存在

- i. 如果存在,
 - i. 将该链表节点的值更新
 - ii. 将该链表节点调整至链表头部
- ii. 如果不存在
 - i. 如果链表容量未满,
 - i. 新生成节点,
 - ii. 将该节点位置调整至链表头部
 - ii. 如果链表容量已满
 - i. 删除尾部节点
 - ii. 新生成节点
 - iii. 将该节点位置调整至链表头部
 - iii. 将新生成的节点, 按关键字为键, 节点地址为值插入哈希表

3. get 操作

通过哈希表, 查看传入的关键字对应的链表节点, 是否存在

- i. 节点存在
 - i. 将该节点位置调整至链表头部
 - ii. 返回该节点的值
- ii. 节点不存在, 返回 null

伪代码:

989. 数组形式的整数加法

```
var LRUCache = function(capacity) {
    // 保存一个该数据结构的最大容量
    // 生成一个双向链表，同时保存该链表的头结点与尾节点
    // 生成一个哈希表
};

function get (key) {
    if 哈希表中存在该关键字 {
        根据哈希表获取该链表节点
        将该节点放置于链表头部
        return 链表节点的值
    } else {
        return -1
    }
};

function put (key, value) {
    if 哈希表中存在该关键字 {
        根据哈希表获取该链表节点
        将该链表节点的值更新
        将该节点放置于链表头部
    } else {
        if 容量已满 {
            删除链表尾部的节点
            新生成一个节点
            将该节点放置于链表头部
        } else {
            新生成一个节点
            将该节点放置于链表头部
        }
    }
};
```

JS 代码参考:

989. 数组形式的整数加法

```
function ListNode(key, val) {
    this.key = key;
    this.val = val;
    this.pre = this.next = null;
}

var LRUCache = function (capacity) {
    this.capacity = capacity;
    this.size = 0;
    this.data = {};
    this.head = new ListNode();
    this.tail = new ListNode();
    this.head.next = this.tail;
    this.tail.pre = this.head;
};

function get(key) {
    if (this.data[key] !== undefined) {
        let node = this.data[key];
        this.removeNode(node);
        this.appendHead(node);
        return node.val;
    } else {
        return -1;
    }
}

function put(key, value) {
    let node;
    if (this.data[key] !== undefined) {
        node = this.data[key];
        this.removeNode(node);
        node.val = value;
    } else {
        node = new ListNode(key, value);
        this.data[key] = node;
        if (this.size < this.capacity) {
            this.size++;
        } else {
            key = this.removeTail();
            delete this.data[key];
        }
    }
    this.appendHead(node);
}

function removeNode(node) {
    let preNode = node.pre,
```

```
nextNode = node.next;
preNode.next = nextNode;
nextNode.pre = preNode;
}

function appendHead(node) {
    let firstNode = this.head.next;
    this.head.next = node;
    node.pre = this.head;
    node.next = firstNode;
    firstNode.pre = node;
}

function removeTail() {
    let key = this.tail.pre.key;
    this.removeNode(this.tail.pre);
    return key;
}
```

复杂度分析

- 时间复杂度: $O(1)$
- 空间复杂度: $O(n)$ n 为容量的大小

设计跳表

请参考前面设计跳表的题解。

前缀和系列

入选理由

- 面试考察频率较高

题目地址（前缀和系列）

- 网易面试题

有一个班级有 n 个人，给出 n 个元素，第 i 个元素代表 第 i 位同学的考试成绩。

输入描述：第一行输入两个数 n 和 m ，两个数以空格隔开，表示 n 个同学和 m 个人。

输出描述：输出 m 行，每一行输出一个百分数 p ，代表超过班级百分之几的人。

示例1：

输入：

3 2

50 60 70

1 2

输出

33.33333%

66.66667%

[1371. 每个元音包含偶数次的最长子字符串](#)

[560. 和为 K 的子数组](#)

其他：

- 308
- 525
- 1139
- 1176
- 1182
- 1277

- 1292
- 1314
- 1504

网易面试题

最直观的思路是 m 次查询，每次都暴力地计算第 i 个人所处的位置。这种算法的时间复杂度为 $O(m * n)$ ，如果提交会超时，我们考虑优化。

思路

一种思路是先对分数排序，然后每次查询的时候可以通过二分查找进行匹配，由于用到了排序，需要花 $O(n \log n)$ 的时间， m 次查询花的时间大致为 $O(m \log n)$ ，时间复杂度可以算是 $O(\max(m, n) * \log n)$ ，这个时间复杂度通过所有的测试用例。由于本次是前缀和，代码就不贴了。

接下来我们介绍一种更加巧妙的前缀和方法。由于每个同学的分数都在 0 - 150 的离散区间，因此我们可以用一个数组 arr ，然后让 $\text{arr}[i]$ 表示分数不超过 i 的人数。前缀和的构造：

```
for(i = 1; i < arr.length; i++){
    arr[i] = arr[i] + arr[i - 1];
}
```

如果求类似 $\text{arr}[i] \sim \text{arr}[j]$ 区间的和，这个时候就可以考虑使用前缀和的方式了。

代码

Python Code:

989. 数组形式的整数加法

```
class Solution:
    def f(self, scores, m):
        n = len(scores)
        arr = [0] * 151
        ans = []
        # 离散化，统计分数为 i 的有多少人
        for i in range(n):
            arr[scores[i]] += 1
        # 构造前缀和
        for i in range(1, len(arr)):
            arr[i] = arr[i] + arr[i - 1]
        # 模拟 m 次询问
        for i in range(m):
            score = scores[m[i] - 1]
            sum = arr[score]
            ans.append(sum / n * 100)
        return ans
```

Java Code:

```
class Solution {
    public List<Integer> f(String s) {
        int n = scores.length;
        int[] arr = new int[151];
        List<Integer> ans = new ArrayList<Integer>();

        // 离散化，统计分数为 i 的有多少人
        for (int i = 0; i < n; i++) {
            arr[scores[i]]++;
        }

        // 构造前缀和
        for (int i = 1; i < arr.length; i++) {
            arr[i] = arr[i] + arr[i - 1];
        }

        // 模拟 m 次询问
        for (int i = 0; i < m.length; i++) {
            int score = scores[m[i] - 1];
            int sum = arr[score];
            ans.add(sum * 1.0 / n * 100);
        }
        return ans;
    }
}
```

复杂度分析

- 时间复杂度: $O(m + n)$
- 空间复杂度: $O(1)$, 我们只开辟了额外的 151 长度的数组。

1371. 每个元音包含偶数次的最长子字符串

这个可以参考我之前写过的题解

[1371. 每个元音包含偶数次的最长子字符串](#)

560. 和为 K 的子数组

这个同样可以参考我之前写过的题解

[560. 和为 K 的子数组](#)

排序系列

入选理由

- 排序是以前很喜欢考的算法题了，现在比重有所降低。但还是会有让你手撕排序的。因此大家也需要掌握。
- 数据结构的话，数组和链表排序都需要会。
- 排序的具体算法，我推荐使用快速排序，其他需要掌握的有归并排序。冒泡，选择，插入这种平方时间复杂度的次要掌握即可。

题目地址（排序系列）

- [912. 排序数组](#)
- [148. 排序链表](#)

这两道题，一道是数组，一道是链表。大家可以用着两道题进行排序算法的手撕练习。

912. 排序数组

思路

几大排序的思路我想有太多人讲过了，我就不班门弄斧了。大家直接看代码吧。

有的算法我给大家贴心地准备了好理解版本和性能好版本，大家可以根据自己的阶段选择性学习。

代码

代码支持： Python3

989. 数组形式的整数加法

```
# 1. 归并排序（推荐！其他排序方法都不推荐在竞赛中使用）
# 归并排序丐版
class Solution:
    def sortArray(self, nums: List[int]) -> List[int]:
        def mergeSort(l, r):
            if l >= r:
                return
            mid = (l + r) // 2
            mergeSort(l, mid)
            mergeSort(mid + 1, r)
            temp = []
            i, j = l, mid + 1
            while i <= mid and j <= r:
                if nums[i] < nums[j]:
                    temp.append(nums[i])
                    i += 1
                else:
                    temp.append(nums[j])
                    j += 1
            while i <= mid:
                temp.append(nums[i])
                i += 1
            while j <= r:
                temp.append(nums[j])
                j += 1
            nums[l : r + 1] = temp

        mergeSort(0, len(nums) - 1)
        return nums

# 归并排序优化版
class Solution:
    def sortArray(self, nums: List[int]) -> List[int]:
        temp = [0] * len(nums)

        def mergeSort(l, r):
            if l >= r:
                return
            mid = (l + r) // 2
            mergeSort(l, mid)
            mergeSort(mid + 1, r)
            i, j = l, mid + 1
            k = 0
            while i <= mid and j <= r:
                if nums[i] < nums[j]:
                    temp[k] = nums[i]
                    i += 1
                else:
                    temp[k] = nums[j]
                    j += 1
                k += 1
            while i <= mid:
                temp[k] = nums[i]
                i += 1
            while j <= r:
                temp[k] = nums[j]
                j += 1
            for i in range(l, r + 1):
                nums[i] = temp[i]
```

989. 数组形式的整数加法

```
j += 1
k += 1
while i <= mid:
    temp[k] = nums[i]
    i += 1
    k += 1
while j <= r:
    temp[k] = nums[j]
    j += 1
    k += 1
nums[l : r + 1] = temp[: r - l + 1]

mergeSort(0, len(nums) - 1)
return nums

# 2. 快速排序
# 快速排序乞丐版
class Solution:
    def sortArray(self, nums: List[int]) -> List[int]:
        temp = [0] * len(nums)

        def quickSort(nums):
            if not nums: return []
            pivot = nums[0]
            nums = nums[1:]
            l = quickSort([num for num in nums if num <= pivot])
            r = quickSort([num for num in nums if num > pivot])
            return l + [pivot] + r

        return quickSort(nums)

# 快速排序优化版
class Solution:
    def sortArray(self, nums: List[int]) -> List[int]:
        temp = [0] * len(nums)

        def partition(l, r):
            pivot = nums[l]

            while l < r:
                while l < r and nums[r] >= pivot:
                    r -= 1
                nums[l] = nums[r]
                while l < r and nums[l] <= pivot:
                    l += 1
                nums[r] = nums[l]
            nums[l] = pivot
            return l
```

```

def quickSort(l, r):
    if l >= r:
        return
    pivot = partition(l, r)
    quickSort(l, pivot - 1)
    quickSort(pivot + 1, r)

quickSort(0, len(nums) - 1)
return nums

# 3. 插入排序
class Solution:
    def sortArray(self, nums: List[int]) -> List[int]:
        n = len(nums)
        for i in range(1, n):
            t = nums[i]
            j = i - 1
            while j > -1 and nums[j] > t:
                nums[j + 1] = nums[j]
                j -= 1
            nums[j + 1] = t
        return nums

# 4. 选择排序
class Solution:
    def sortArray(self, nums: List[int]) -> List[int]:
        n = len(nums)
        for i in range(n - 1):
            k = i
            for j in range(i + 1, n):
                if nums[j] < nums[k]:
                    k = j
            nums[i], nums[k] = nums[k], nums[i]
        return nums

# 5. 冒泡排序
class Solution:
    def sortArray(self, nums: List[int]) -> List[int]:
        n = len(nums)
        for i in range(n):
            for j in range(i + 1, n):
                if nums[j] < nums[i]:
                    nums[i], nums[j] = nums[j], nums[i]
        return nums

```

148. 排序链表

思路

几大排序的思路我想有太多人讲过了，我就不班门弄斧了。大家直接看代码吧。

代码

代码支持： Python3

```

# 1. 归并排序（推荐！其他排序方法都不推荐在竞赛中使用）
class Solution:
    def sortList(self, head: ListNode) -> ListNode:
        def mergeSort(head: ListNode) -> ListNode:
            if not head or not head.next:
                return head
            dummyHead = ListNode(-1)
            dummyHead.next = head
            slow, fast = dummyHead, head
            while fast and fast.next:
                slow = slow.next
                fast = fast.next.next
            nxt = slow.next
            slow.next = None
            return merge(mergeSort(head), mergeSort(nxt))

        def merge(head1: ListNode, head2: ListNode) -> ListNode:
            dummyHead = ListNode(-1)
            temp, l1, l2 = dummyHead, head1, head2
            while l1 and l2:
                if l1.val <= l2.val:
                    temp.next = l1
                    l1 = l1.next
                else:
                    temp.next = l2
                    l2 = l2.next
                temp = temp.next
            if l1:
                temp.next = l1
            elif l2:
                temp.next = l2
            return dummyHead.next

        return mergeSort(head)

# 2. 快速排序
class Solution:
    def sortList(self, head):
        # 最坏情况也是  $n^2$ ，因此面试或者竞赛不建议使用
        def quickSort(head, end):

            if head != end:
                pivot = partition(head, end)
                quickSort(head, pivot)
                quickSort(pivot.next, end)

        def partition(head, end):
            # p1是写指针, p2是读指针

```

989. 数组形式的整数加法

```
# 最终 p1 是大的链表的头, head 是小的链表的头
pivot_val = head.val
p1, p2 = head, head.next

while p2 != end:
    if p2.val < pivot_val:
        # 相当于数组的 append 方法
        p1 = p1.next
        p1.val, p2.val = p2.val, p1.val
    p2 = p2.next
head.val, p1.val = p1.val, pivot_val
return p1

quickSort(head, None)
return head

# 3. 插入排序
class Solution:
    def sortList(self, head):
        if head == None or head.next == None:
            return head

        dummy = ListNode(-1)
        dummy.next = head
        pre = dummy
        cur = head
        while cur:
            # 准备将 last 插入到合适位置
            last = cur.next
            if last and last.val < cur.val:
                # 从 dummy 到 cur 线性遍历找第一个满足条件的位置
                while pre.next and pre.next.val < last.val:
                    pre = pre.next
                tmp = pre.next
                pre.next = last
                cur.next = last.next # 别忘了这个, 否则成环
                last.next = tmp
                pre = dummy
            else:
                cur = last

        return dummy.next

# 4. 选择排序
class Solution:
    def sortList(self, head):
        temp = head

        while temp:
            min_node = temp
            ...
```

989. 数组形式的整数加法

```
r = temp.next
while r:
    if min_node.val > r.val:
        min_node = r
    r = r.next
temp.val, min_node.val = min_node.val, temp.val
temp = temp.next
return head

# 5. 冒泡排序
class Solution:
    def sortList(self, head):
        if not head:
            return None
        swaped = True
        while swaped:
            swaped = False
            temp = head
            while temp.next:
                if temp.val > temp.next.val:
                    swaped = True
                    temp.val, temp.next.val = temp.next.val,
temp = temp.next
return head
```