



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
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
第三章 - 进阶篇	1.3
【91 算法-进阶篇】 01.并查集	1.3.1
【91 算法-进阶篇】 02.Trie	1.3.2
【91 算法-进阶篇】 03.KMP & RK	1.3.3
【91 算法-进阶篇】 04.跳表	1.3.4
【91 算法-进阶篇】 05.剪枝	1.3.5
【91 算法-进阶篇】 07.高频面试题	1.3.6
【91 算法-进阶篇】 08.堆	1.3.7
第四章 - 专题篇	1.4
【91 算法-专题篇】 01.二分法	1.4.1
【91 算法-专题篇】 02.滑动窗口	1.4.2
【91 算法-专题篇】 03.位运算	1.4.3
【91 算法-专题篇】 04.搜索	1.4.4
【91 算法-专题篇】 05.背包问题	1.4.5
【91 算法-专题篇】 06.动态规划	1.4.6
【91 算法-专题篇】 07.分治	1.4.7
第五章 - 题解篇	1.5
基础篇	1.5.1
【Day 1】 2020-11-01 - 66. 加一	1.5.1.1
官方题解	1.5.1.1.1
【Day 2】 2020-11-02 - 821. 字符的最短距离	1.5.1.2
官方题解	1.5.1.2.1

精选题解	1.5.1.2.2
【Day 3】 2020-11-03 - 1381. 设计一个支持增量操作的栈	
官方题解	1.5.1.3.1 1.5.1.3
【Day 4】 2020-11-04 - 394. 字符串解码	1.5.1.4
官方题解	1.5.1.4.1
【Day 5】 2020-11-05 - 232. 用栈实现队列	1.5.1.5
官方题解	1.5.1.5.1
【Day 6】 2020-11-06 - 768. 最多能完成排序的块 II	
官方题解	1.5.1.6.1 1.5.1.6
精选题解	1.5.1.6.2
【Day 7】 2020-11-07 - 24. 两两交换链表中的节点	
官方题解	1.5.1.7.1 1.5.1.7
精选题解	1.5.1.7.2
【Day 8】 2020-11-08 - 61. 旋转链表	1.5.1.8
官方题解	1.5.1.8.1
【Day 9】 2020-11-09 - 109. 有序链表转换二叉搜索树	
官方题解	1.5.1.9.1 1.5.1.9
【Day 10】 2020-11-10 - 160. 相交链表	1.5.1.10
官方题解	1.5.1.10.1
精选题解	1.5.1.10.2
【Day 11】 2020-11-11 - 142. 环形链表 II	1.5.1.11
官方题解	1.5.1.11.1
精选题解	1.5.1.11.2
【Day 12】 2020-11-12 - 146. LRU 缓存机制	1.5.1.12
官方题解	1.5.1.12.1
精选题解	1.5.1.12.2
【Day 13】 2020-11-13 - 104. 二叉树的最大深度	
官方题解	1.5.1.13.1 1.5.1.13
精选题解	1.5.1.13.2
精选题解	1.5.1.13.3
【Day 14】 2020-11-14 - 100. 相同的树	1.5.1.14
官方题解	1.5.1.14.1
精选题解	1.5.1.14.2
【Day 15】 2020-11-15 - 129. 求根到叶子节点数字之和	
官方题解	1.5.1.15.1 1.5.1.15

【Day 16】 2020-11-16 - 513. 找树左下角的值	1.5.1.16
官方题解	1.5.1.16.1
精选题解	1.5.1.16.2
【Day 17】 2020-11-17 - 297. 二叉树的序列化与反序列化	
官方题解	1.5.1.17.1 1.5.1.17
【Day 18】 2020-11-18 - 987. 二叉树的垂序遍历	
官方题解	1.5.1.18.1 1.5.1.18
精选题解	1.5.1.18.2
【Day 19】 2020-11-19 - 1. 两数之和	1.5.1.19
官方题解	1.5.1.19.1
精选题解	1.5.1.19.2
【Day 20】 2020-11-20 - 347. 前 K 个高频元素	1.5.1.20
官方题解	1.5.1.20.1
精选题解	1.5.1.20.2
【Day 21】 2020-11-21 - 447. 回旋镖的数量	1.5.1.21
官方题解	1.5.1.21.1
精选题解	1.5.1.21.2
【Day 22】 2020-11-22 - 3. 无重复字符的最长子串	
官方题解	1.5.1.22.1 1.5.1.22
精选题解	1.5.1.22.2
【Day 23】 2020-11-23 - 30. 串联所有单词的子串	
官方题解	1.5.1.23.1 1.5.1.23
【Day 24】 2020-11-24 - 30. 解数独	1.5.1.24
官方题解	1.5.1.24.1
【Day 25】 2020-11-25 - 35. 搜索插入位置	1.5.1.25
官方题解	1.5.1.25.1
【Day 26】 2020-11-26 74. 搜索二维矩阵	1.5.1.26
官方题解	1.5.1.26.1
【Day 27】 2020-11-27 26. 删除排序数组中的重复项	
官方题解	1.5.1.27.1 1.5.1.27
【Day 28】 2020-11-28 876. 链表的中间结点	1.5.1.28
官方题解	1.5.1.28.1
【Day 29】 2020-11-29 1052. 爱生气的书店老板	1.5.1.29
官方题解	1.5.1.29.1
【Day 30】 2020-11-30 239. 滑动窗口最大值	1.5.1.30

官方题解	1.5.1.30.1
进阶篇	1.5.2
【Day 34】2020-12-04 - 树的遍历系列	1.5.2.1
官方题解	1.5.2.1.1
【Day 35】2020-12-05 - 反转链表系列	1.5.2.2
官方题解	1.5.2.2.1
【Day 36】2020-12-06 - 位运算系列	1.5.2.3
官方题解	1.5.2.3.1
【Day 37】2020-12-07 - 动态规划系列	1.5.2.4
官方题解	1.5.2.4.1
【Day 38】2020-12-08 - 有效括号系列	1.5.2.5
官方题解	1.5.2.5.1
【Day 39】2020-12-09 - 设计系列	1.5.2.6
官方题解	1.5.2.6.1
【Day 40】2020-12-10 - 前缀树系列	1.5.2.7
官方题解	1.5.2.7.1
【Day 41】2020-12-11 - 208. 实现 Trie	1.5.2.8
官方题解	1.5.2.8.1
【Day 42】2020-12-12 - 677. 键值映射	1.5.2.9
官方题解	1.5.2.9.1
【Day 43】2020-12-13 - 面试题 17.17. 多次搜索	1.5.2.10
官方题解	1.5.2.10.1
【Day 44】2020-12-14 - 547. 朋友圈	1.5.2.11
官方题解	1.5.2.11.1
【Day 45】2020-12-15 - 924. 尽量减少恶意软件的传播	1.5.2.12
官方题解	1.5.2.12.1 1.5.2.12
【Day 46】2020-12-16 - 1319. 连通网络的操作次数	1.5.2.13
官方题解	1.5.2.13.1 1.5.2.13
【Day 47】2020-12-17 - 1206. 设计跳表	1.5.2.14
官方题解	1.5.2.14.1
【Day 48】2020-12-18 - 814. 二叉树剪枝	1.5.2.15
官方题解	1.5.2.15.1
【Day 49】2020-12-19 - 39. 组合总和	1.5.2.16
官方题解	1.5.2.16.1
【Day 50】2020-12-20 - 40. 组合总和 II	1.5.2.17

官方题解	1.5.2.17.1
【Day 51】 2020-12-21 - 47. 全排列 II	1.5.2.18
官方题解	1.5.2.18.1
【Day 52】 2020-12-22 - 28. 实现 strStr()-BF&RK	
官方题解	1.5.2.19.1 1.5.2.19

先导篇

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**。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。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。

一般而言，我们遇到一个问题，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。

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。**

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，大家可以用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

数据结构与算法概述

简介

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。

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的算法思想。

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，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。如果当天的快递很多，那么效率下降会很明显。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？如果这么做的话，又来了一个西法的快递，我们就需要擦掉张三那一行，写上西法的信息。然后将擦掉的张三的信息重写到下一行。

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。

那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。

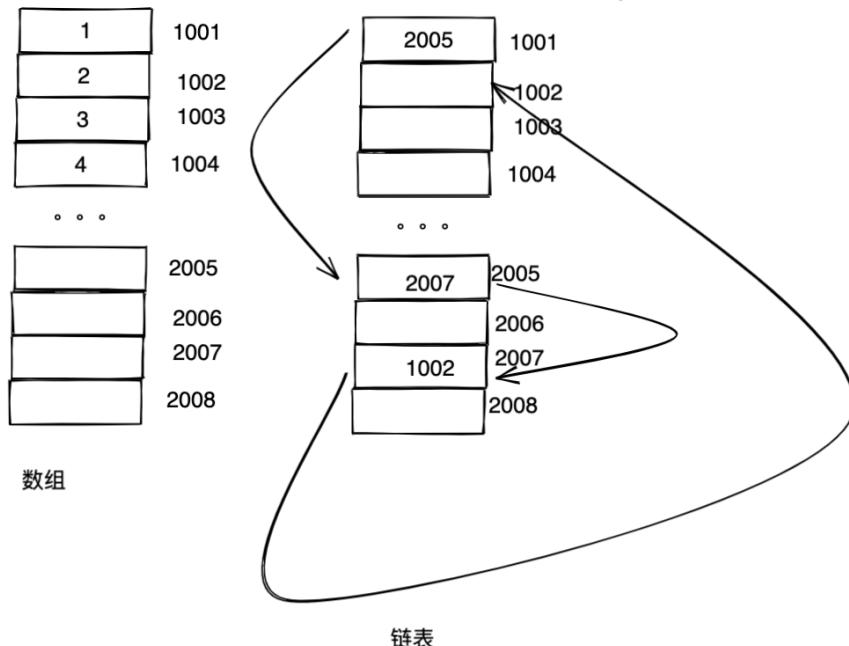
物理结构和逻辑结构

我前面说了。内存的物理表现实际上就是一系列连续的内存单元，这是物理结构。数据结构只有两个对应基础的数据结构，它们是数组和链表。其他数据结构都是基于它们产生的。

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。

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，由于数组中的元素大小固定，因此我们可以实现随机访问。链表则不行，因为其下个元素并不一定是紧邻的。

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。好处则是，增加或删除变得容易。



如上图是一段物理内存。如果我在内存中存值，并通过内存编号访问那就是数组。如果我在内存中存内存编号，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。

想想上面快递的例子。

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，帮我大家理解就经典的算法思想，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。

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。

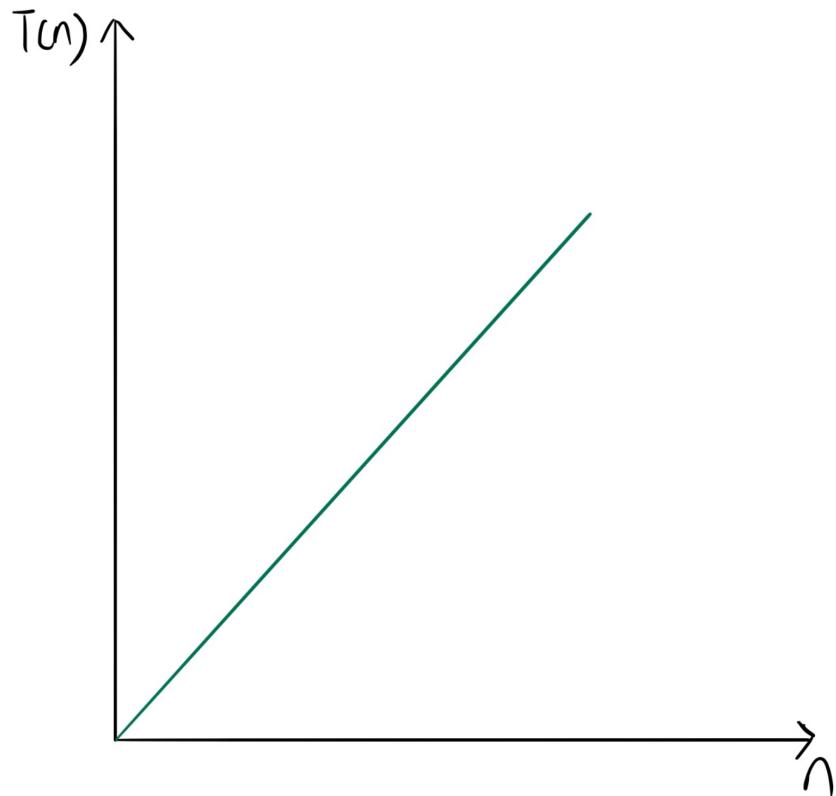
如下是一个从 1 累加到 n 的一个算法，这个算法用了一层循环，并且借助了一个变量 res 来完成。

```
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 $2 * x$ ，而仅保留最
大项，如上的 $n * x$ ，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

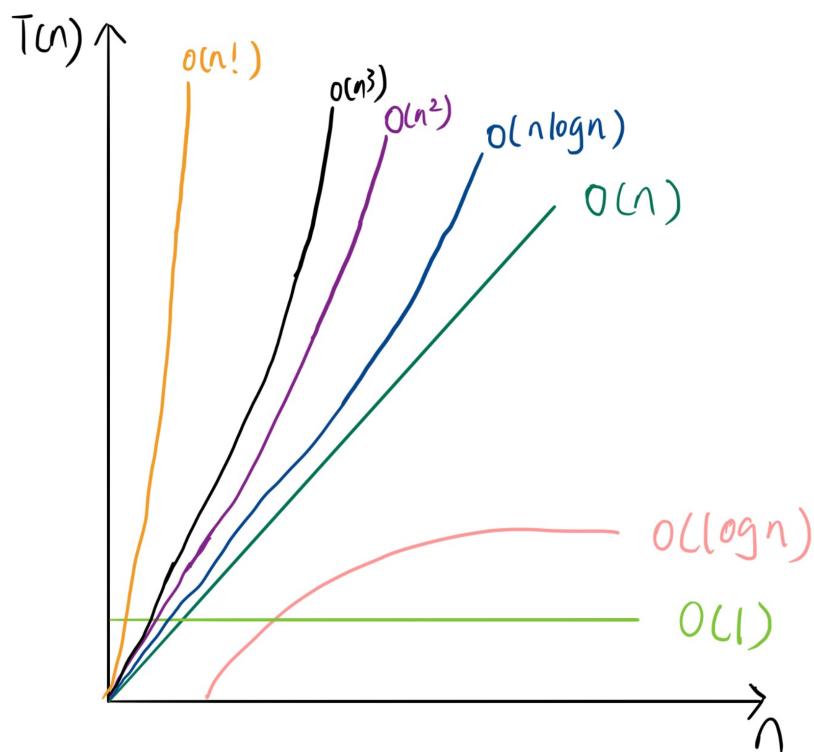
以上正是一种叫做大 O 表示法的基本思想，它是一种描述算法性能的记
法，这种描述和编译系统、机器结构、处理器的快慢等因素无关。这种
描述的参数是 N ，表示数据的规模。这里的 O 表示量级 (order)，比
如说“二分查找的时间复杂度是 $O(\log N)$ ”，就是说它需要“通过 $\log N$
量级的操作去查找一个规模为 N 的数据结构”。这种估测对算法的理论
分析和大致比较是非常有价值的，我们可以很快地对算法进行一个大致的
估算。

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。

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。

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。

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。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:  $O(1)$ 
```

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。



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。

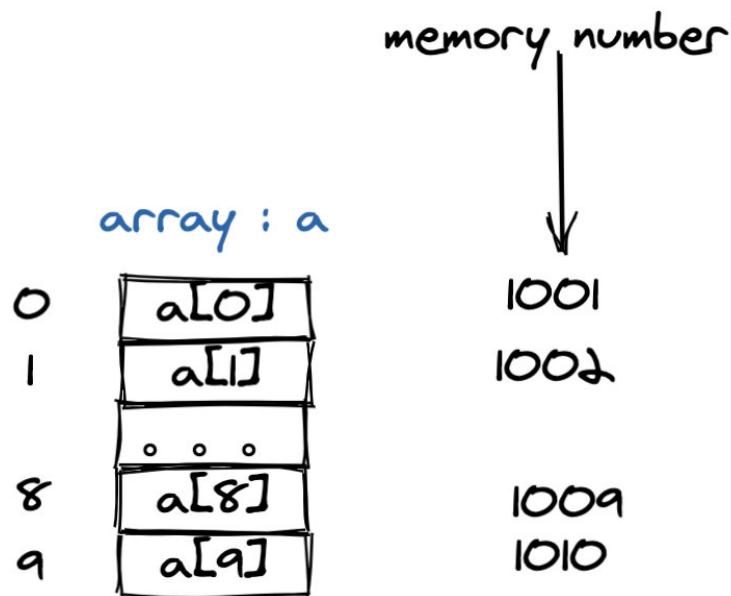
数组的知识大家可以轻松地迁移到字符串，因此本书不对字符串进行特殊讲解。

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。

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，其中基值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。

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。

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, 3]  
# 在索引2前插入一个5  
arr.insert(2, 5)  
print(arr) # [1, 2, 5, 3]
```

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

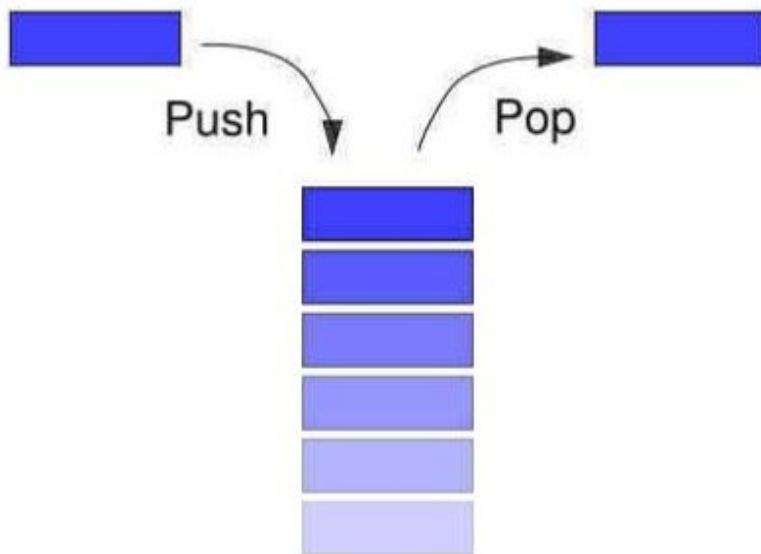
题目推荐

- [28. 实现 strStr\(\)](#)
- [75. 颜色分类](#)
- [380. 常数时间插入、删除和获取随机元素](#)

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，而从其他位置获取内容是不被允许的。

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，而退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。

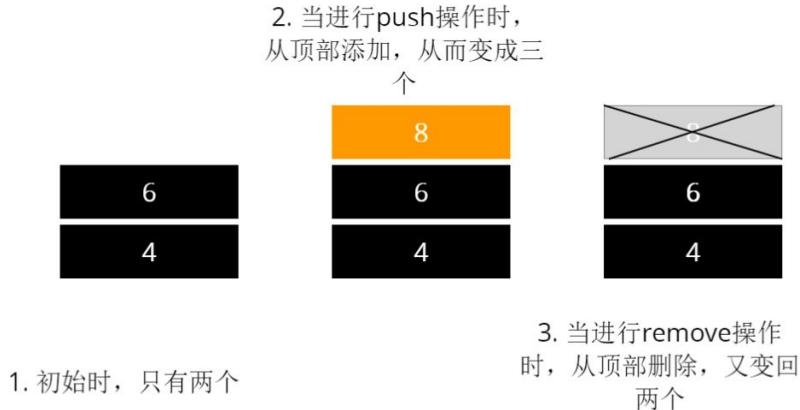
栈的常用操作与时间复杂度

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

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。

1. 进栈 - $O(1)$
2. 出栈 - $O(1)$
3. 取栈顶 - $O(1)$
4. 判断是否为空栈 - $O(1)$



应用

- 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。

力扣相关题目推荐：

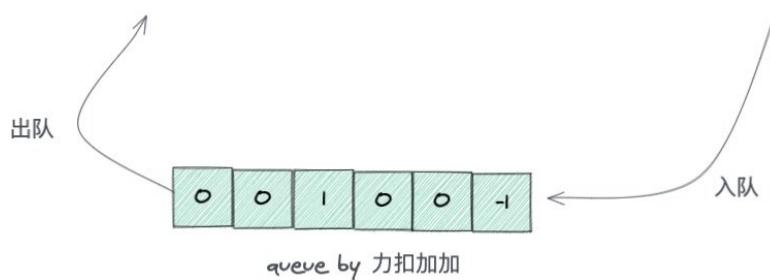
- [150. 逆波兰表达式求值](#)

题目推荐

- [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](#)

队列

同样地，队列也是一种受限的数据结构。和栈相反的是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 - $O(1)$

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就不是 $O(1)$ ，而是 $O(N)$ 了。其实只要维护一个变量 `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。

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](#)

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

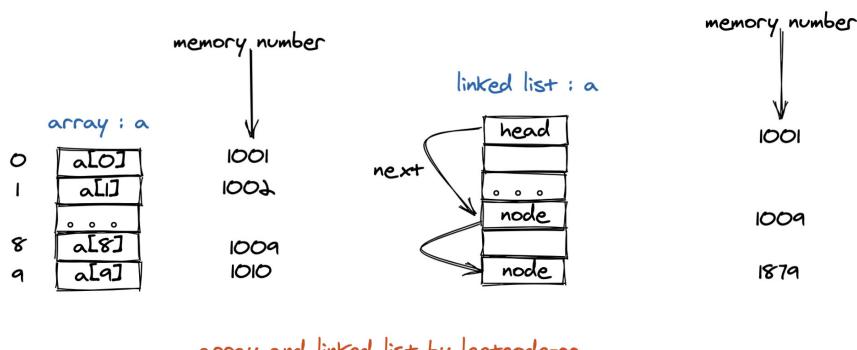
- [基础数据结构 by lucifer](#)

链表

简介

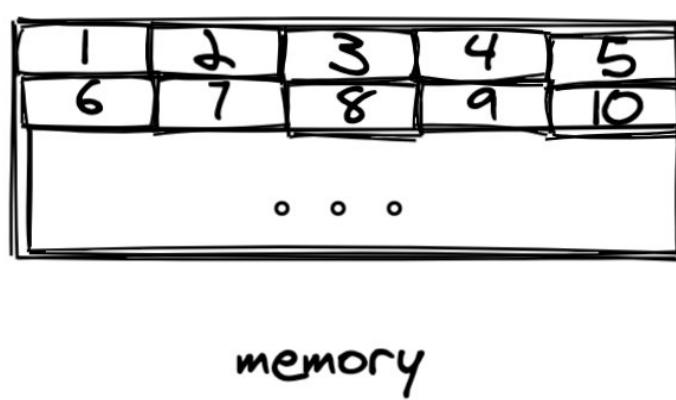
本节给大家介绍的是和数组同一个级别的重量级数据结构，其他数据结构都是基于其进行扩展的。

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：



(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：



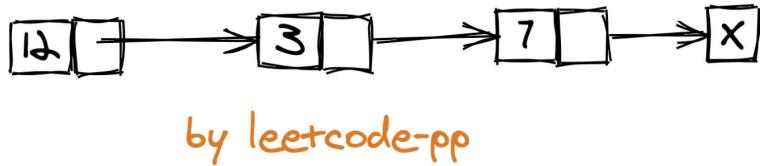
(图 2. 物理内存)

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`。

有没有想过为啥只有二叉树，而没有一叉树。实际上链表就是特殊的树，即一叉树。

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。

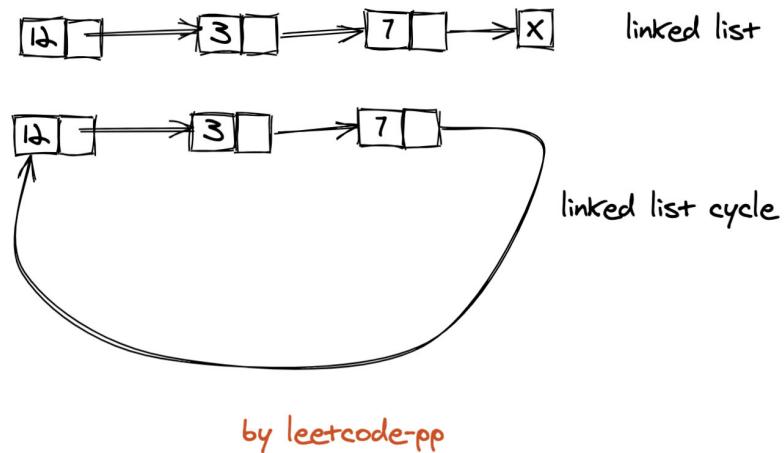
作用：既可以像数组一样在 $O(1)$ 的时间对访问任意元素，又可以像链表一样在 $O(1)$ 的时间对节点进行增删

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

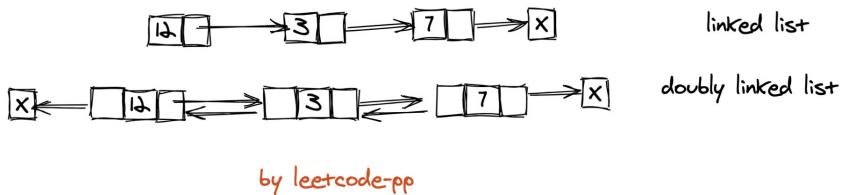
以下分类是两种分类标准，也就是一个链表可以既属于循环链表，也属于单链表，这是毋庸置疑的。

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，一个是存储下一个节点指针的指针域，一个是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 [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。

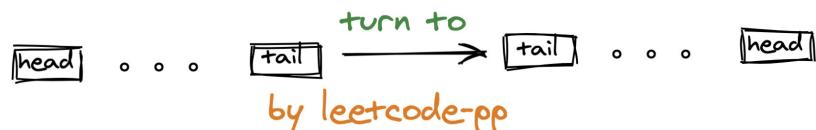
伪代码：

```
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

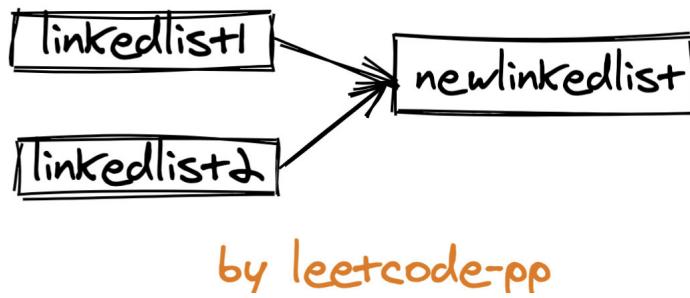
1. 将某个链表进行反转
2. 在 $O(n)$ 时间, $O(1)$ 空间复杂度下逆序读取链表的某个值
3. 将某个链表按 K 个一组进行反转



(图 1)

题型二：合并链表

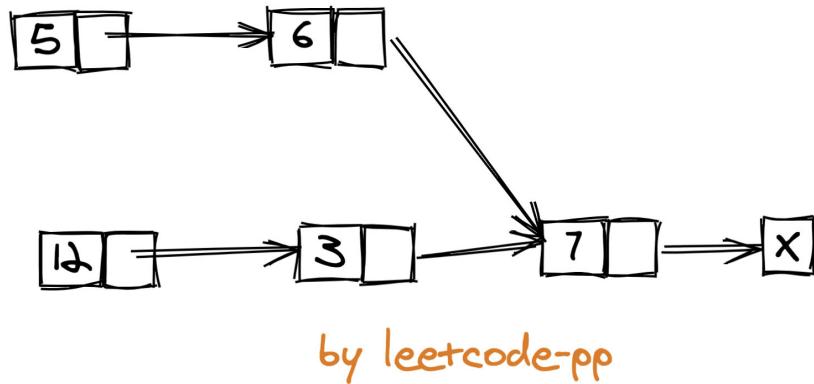
1. 将两条有序或无序的链表合并成一条有序链表
2. 将 k 条有序链表合并成一条有序链表



(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

- 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

- 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:

```
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:

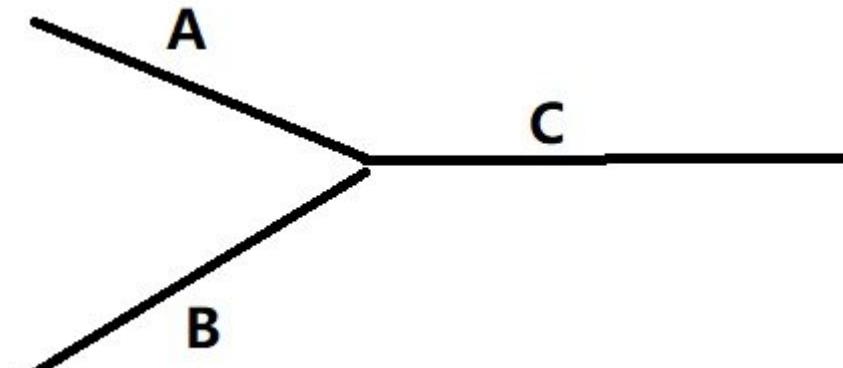
```
let data = new Set();
while (A !== null) {
    data.add(A);
    A = A.next;
}
while (B !== null) {
    if (data.has(B)) return B;
    B = B.next;
}
return null;
```

复杂度分析

- 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, 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：

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 B 的头结点,然后继续遍历直至相交点(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 {
    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
```

JS 代码参考：

```
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 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

- 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:

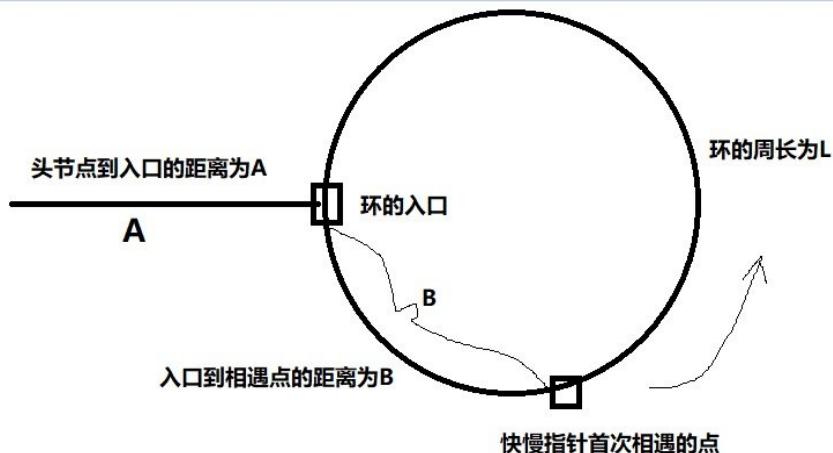
```
let data = new Set();
while (head) {
    if (data.has(head)) {
        return head;
    } else {
        data.add(head);
    }
    head = head.next;
}
return null;
```

复杂度分析

- 时间复杂度: $O(N)$
- 空间复杂度: $O(N)$

解法二：快慢指针法

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 指针指向链表头部,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



(图 6)

为什么第二次相遇的点为环的入口? 原因如下:

- 第一次相遇时
- 慢指针移动的距离为 $s_1 = A + B + n_1 * L$
- 快指针移动的距离为 $s_2 = A + B + n_2 * L$
- 快指针是慢指针速度的两倍,所以 $s_2 = 2 * s_1$
- $A + B + n_2 L = 2A + 2B + n_1 L \implies A = -B + (n_2 - n_1) * L$

- 因为圆的性质 $(n_2 - n_1) * L \implies$ 绕圆 $(n_2 - n_1)$ 圈 $\implies 0$
- $A = -B + (n_2 - n_1) * L \implies A = -B$
- 即在第一次相遇点, 向前走 A 步 \implies 向后走 B 步
- **第一次相遇后**
- 快指针从头节点走 A 步会到达环的入口
- 慢指针从第一次相遇点走 A 步, 相当于向后走 B 步, 也会到达环的入口

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

- 时间复杂度: $O(N)$
- 空间复杂度: $O(1)$

套路四：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
 - 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：

```

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

- [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。

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

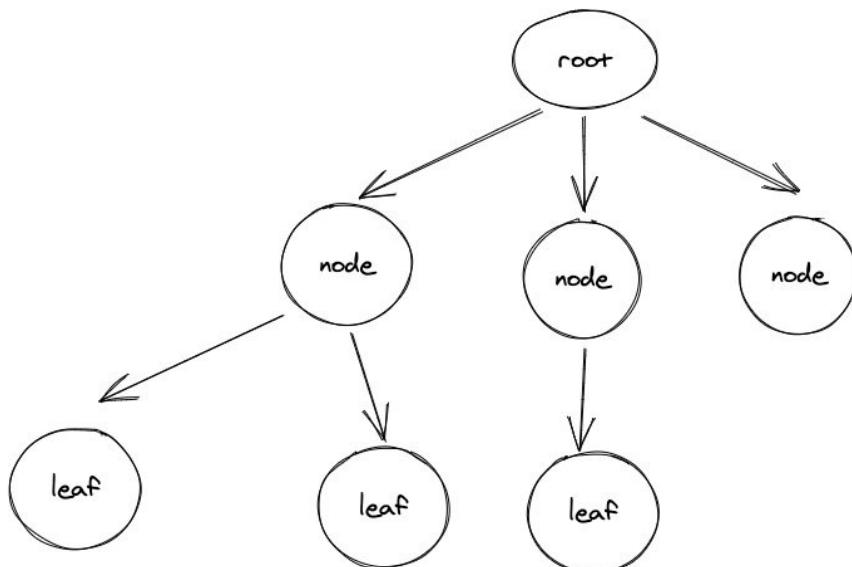
介绍

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

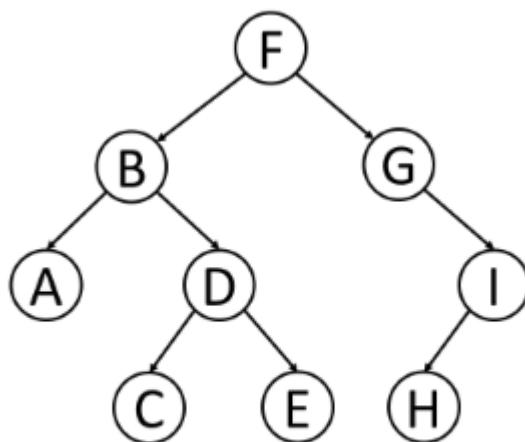
4. 层序遍历(BFS)

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:

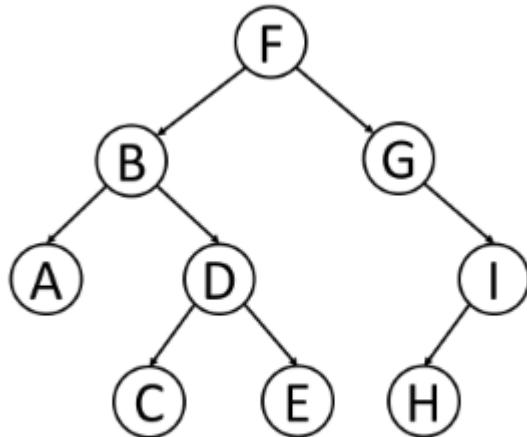
前序遍历的伪代码：

```
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:

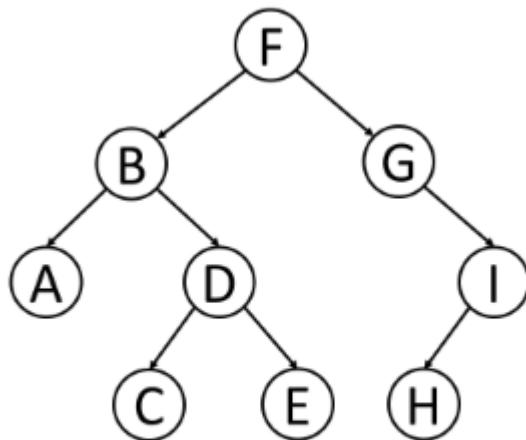
中序遍历的伪代码：

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

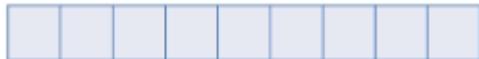
后续遍历

- 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：

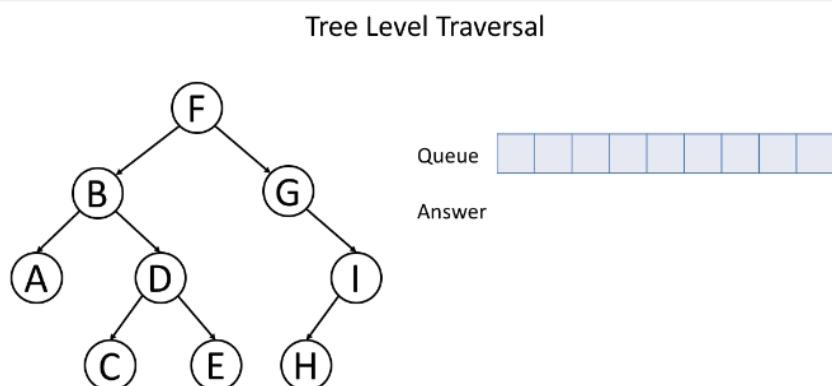
```
postorder(root) {
    if not root: return
    postorder(root.left)
    postorder(root.right)
    dosomething(root)
```

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：

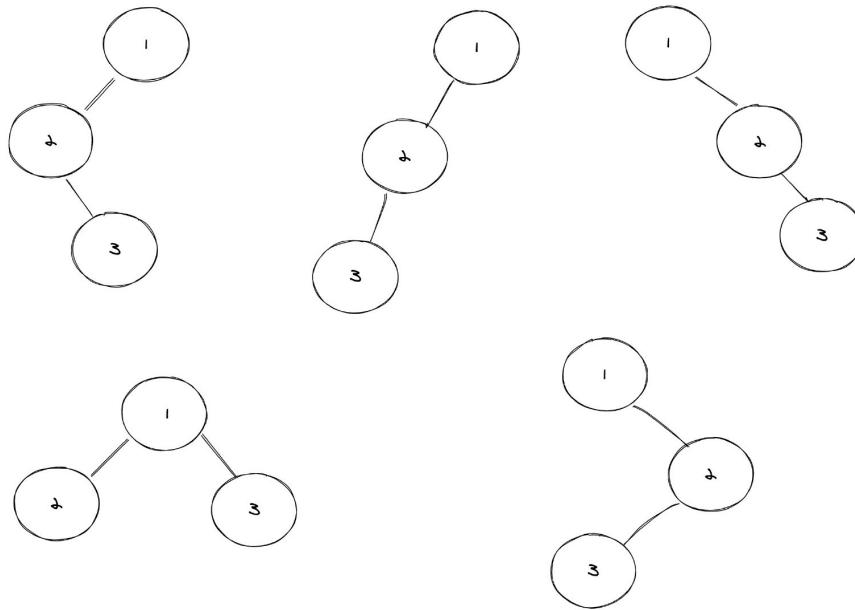
```

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》。

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： $\text{Math.floor}(i / 2)$
- 左子节点： $2 * i$
- 右子节点： $2 * i + 1$

如果 $A[\text{parent}(i)] \geq A[i]$ ，则称该堆为最大堆，如果 $A[\text{parent}(i)] \leq A[i]$ ，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

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。

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 $O(1)$

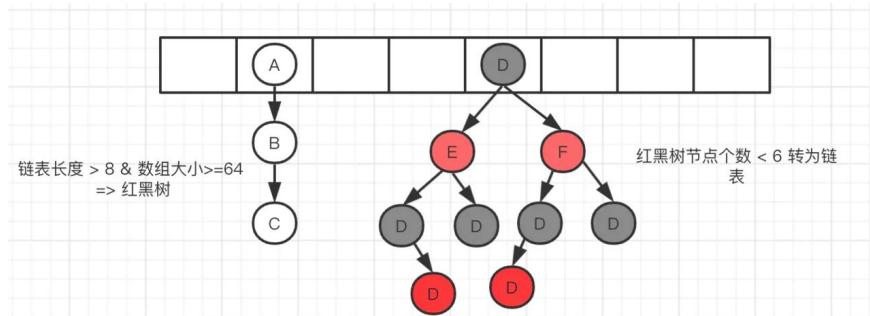
听起来很神奇？没错！那么问题来了。

- 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

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。

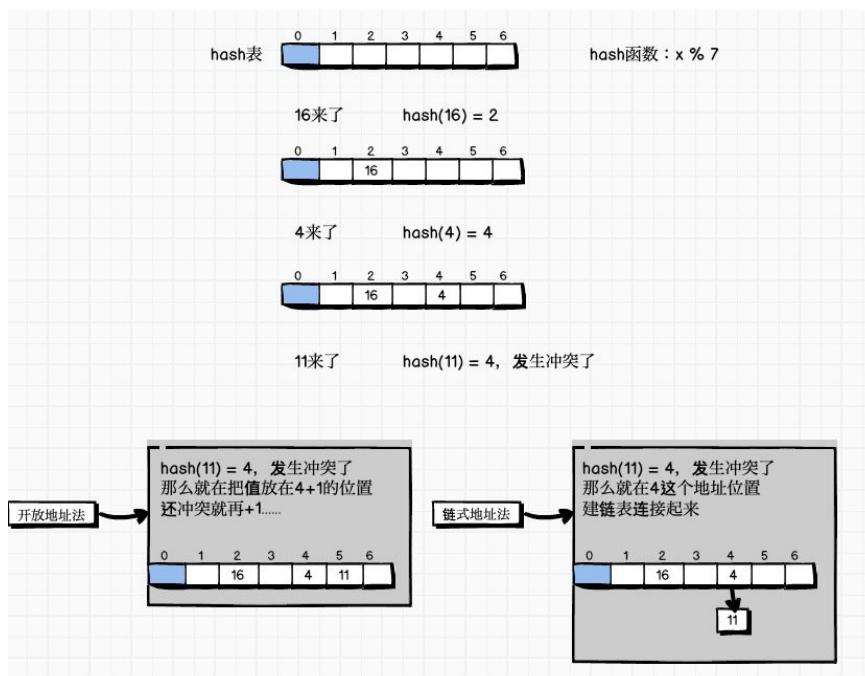
这两个精髓大家要牢牢记住，具体内容我们后面再详细讲述。

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：



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😊

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)
- 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 - 多人运动

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

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



力扣加加

努力做西湖区最好的算法题解

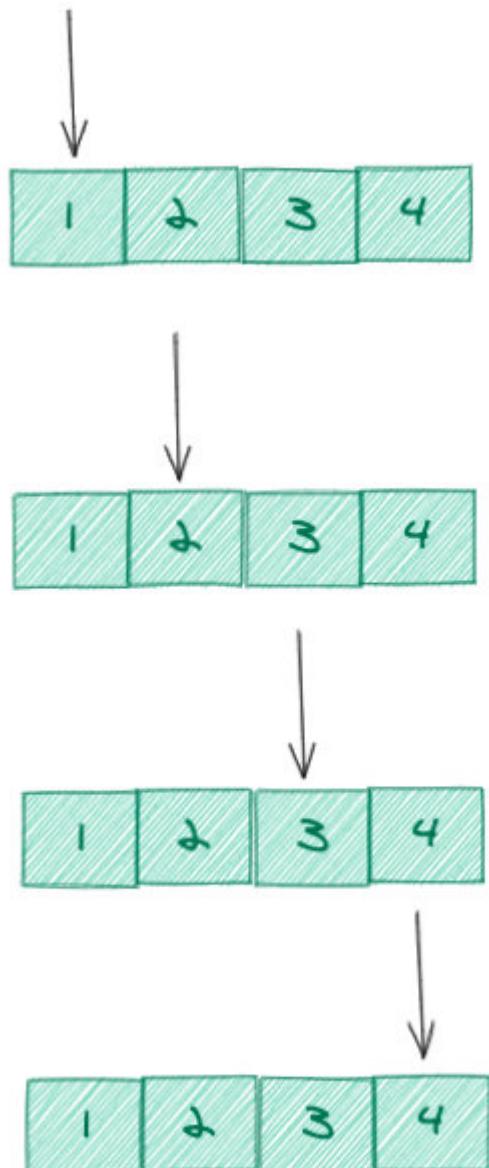
什么是双指针

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]);  
}
```



(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。

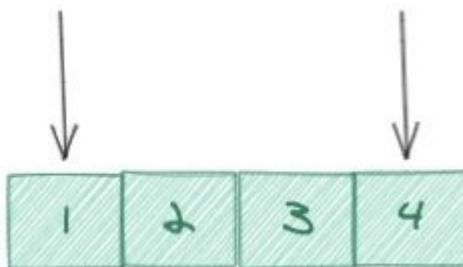
当然这里的“指针”不仅可以是数组的索引，也可以是别的。不过在大多数情况下，都是数组的索引。因此数组双指针就是有两个这样的数组索引。

```

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，一个指向左界，一个指向右界，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。

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. 快慢指针
3. 固定间距指针

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/>

图

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

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

基本概念

- 无向图 & 有向图
- 有权图 & 无权图
- 入度 & 出度
- 路径 & 环
- 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。

邻接矩阵（常见）

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 $\text{graph}[i][j] = 1$ 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 $\text{graph}[i][j] = 0$ 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。 $\text{times}[i] = (u,$

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: $\text{times} = [[2,1,1],[2,3,1],[3,4,1]], N = 4, K = 2$

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 $\text{times}[i] = (u, v, w)$ 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：

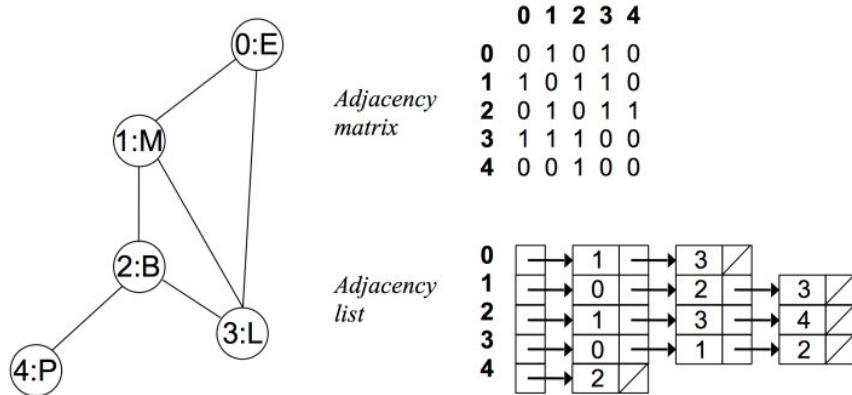
```
graph = collections.defaultdict(list)
for fr, to, w in times:
    graph[fr - 1].append((to - 1, w))
```

这就构造了一个临界矩阵，之后我们基于这个邻接矩阵遍历图即可。

邻接表

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

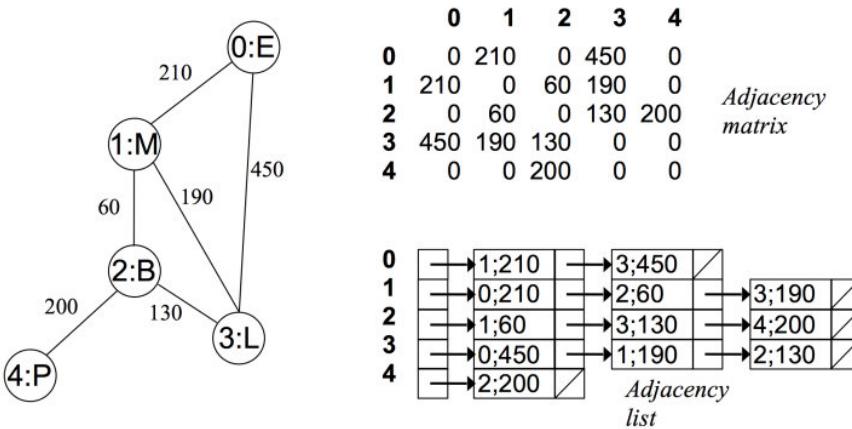
例如在无向无权图中：



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

可以看出在无向图中，邻接矩阵关于对角线对称，而邻接链表总有两条对称的边。

而在有向无权图中：



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

图的遍历

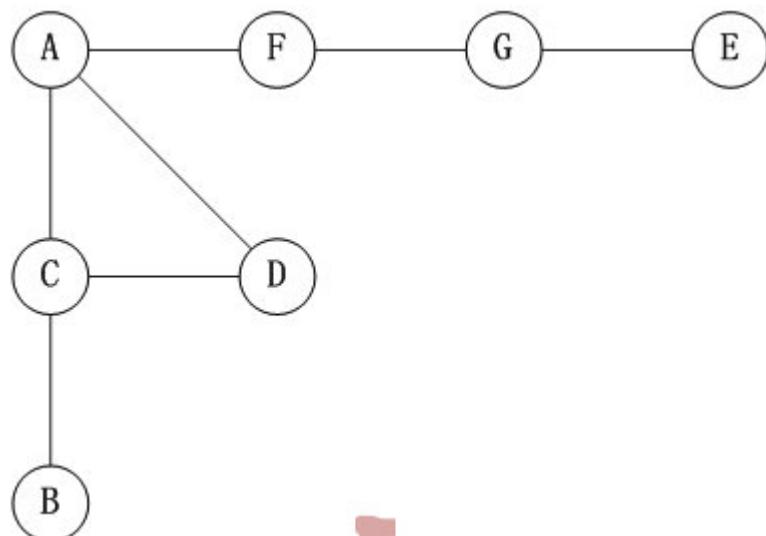
图建立好了，接下来就是要遍历。不管你是什么算法，肯定都要遍历的，一般有以下两种方法（其他奇葩的遍历方式实际意义不大，没有必要学习）。不管是哪一种遍历，如果图有环，就一定要记录节点的访问情

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)

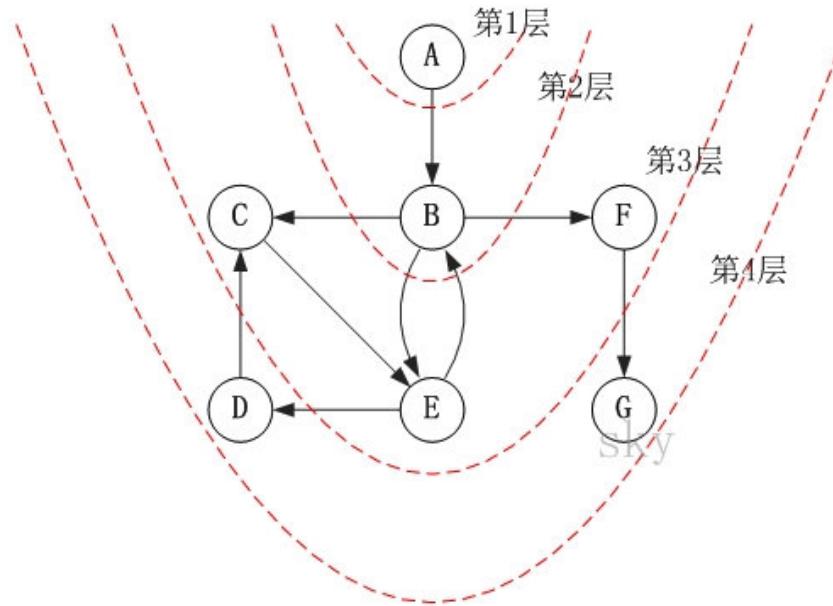
深度优先遍历图的方法是，从图中某顶点 v 出发，不断访问邻居，邻居的邻居直到访问完毕。



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。



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_marshall
- 最小生成树 (Kruskal & Prim)
- A 星寻路算法
- 二分图 (染色法)
- 拓扑排序

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。

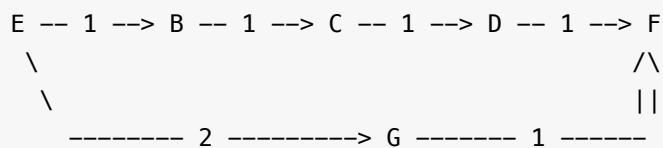
算法的基本思想是贪心，每次都遍历所有邻居，并从中找到距离最小的，本质上是一种广度优先遍历。这里我们借助堆这种数据结构，使得可以在 $\log N$ 的时间内找到 cost 最小的点。

代码模板：

```
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
```

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

```

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：

```

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, K - 1, to)
            if dist == -1: return -1
            ans = max(ans, dist)
        return ans

```

你学会了么？

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~

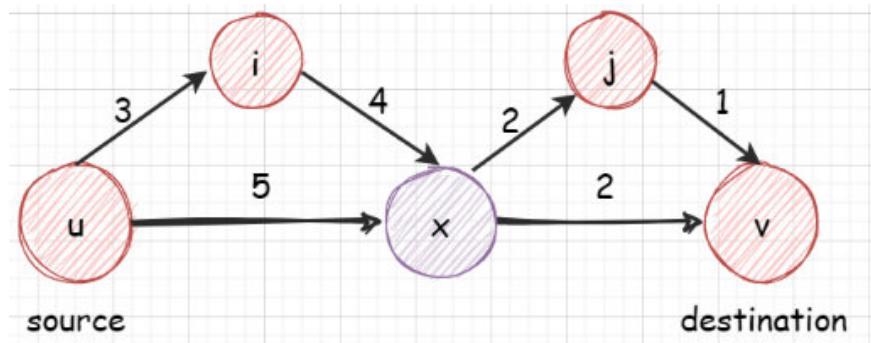
floyd_warshal 算法

floyd_warshal 也是解决两个点距离的算法，除此之外，贝尔曼-福特算法也是解决单源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。

还有一个非常重要的点是 floyd_warshal 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_warshal 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。

思考题：最长无环路径可以用动态规划来解么？

代码模板：

```
# graph 是邻接矩阵, v 是顶点个数
def floyd_warshall(graph, v):
    dist = [[float("inf")] for _ in range(v)] for _ in range(v)

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

    # check vertex k against all other vertices (i, j)
    for k in range(v):
        # looping through rows of graph array
        for i in range(v):
            # looping through columns of graph array
            for j in range(v):
                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, v
```

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：

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

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

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 a 也是 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` = $\{\}$

输出: [true,true]

示例 4：

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

输出: [false,true]

示例 5：

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

输出: [true,false,true,false]

提示：

$2 \leq n \leq 100$

$0 \leq \text{prerequisite.length} \leq (n * (n - 1) / 2)$

$0 \leq \text{prerequisite}[i][0], \text{prerequisite}[i][1] < n$

$\text{prerequisite}[i][0] \neq \text{prerequisite}[i][1]$

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_marshall 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_marshall 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：

```
class Solution:
    def floyd_marshall(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_marshall(graph, n)
        for to, fr in queries:
            ans.append(bool(dist[fr][to]))
        return ans
```

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，但但是完整没有 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，
- $g(n)$ 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 $H(n) = D * (abs(n.x - goal.x) + abs(n.y - goal.y))$ 。



(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](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：

```

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 actions
            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](#)

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
        using BFS
    """

    indegree = [0] * len(graph)
    queue = []
    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.pop(0)
        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 这两个算法暂时先不写了，先留个模板给大家。

Kruskal

```

from typing import List, Tuple

def kruskal(num_nodes: int, num_edges: 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, num_edges, edges)

```

Prim

```

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]

```

```

        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 Vertices
# Minimum Distance of explored vertex with neighboring vertices
# formed in graph
Distance_TV = [] # Heap of Distance of vertices from tree root
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[get_position(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))
```

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。

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。



Trie

简介

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。

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？

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”

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。

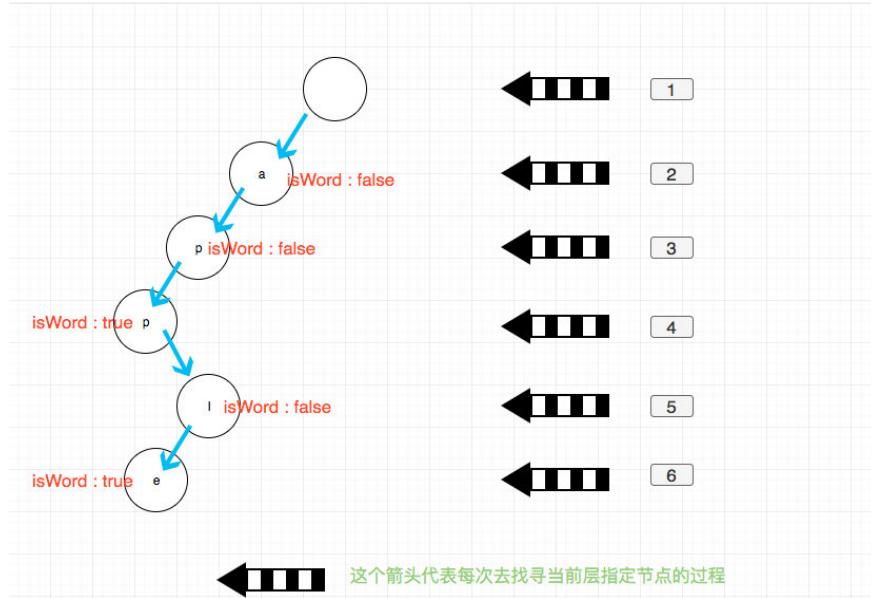
再比如给你一个长句和一堆敏感词，找出长句中所有敏感词出现的所有位置（想下，有时候我们口吐芬芳，结果发送出去却变成了****，懂了吧）

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。

基本概念

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。

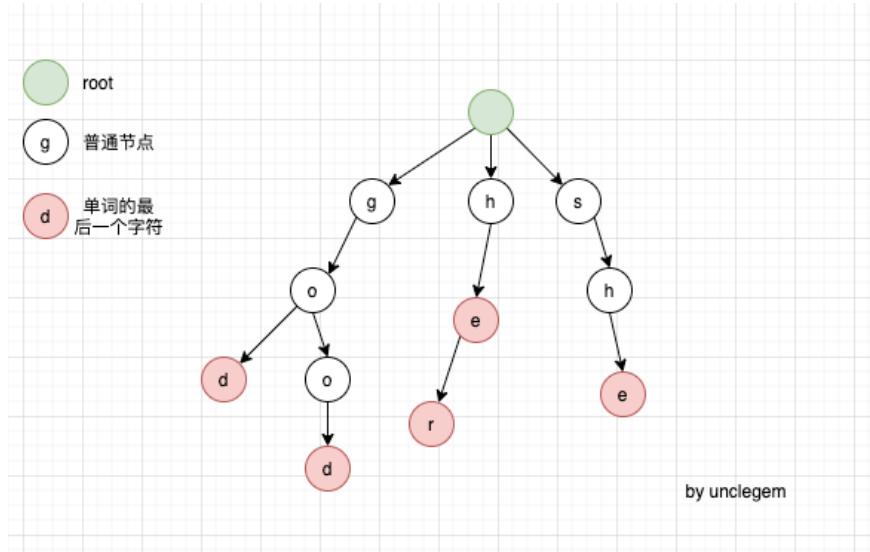
接下来，我们看下 Trie 里面的概念。

节点：

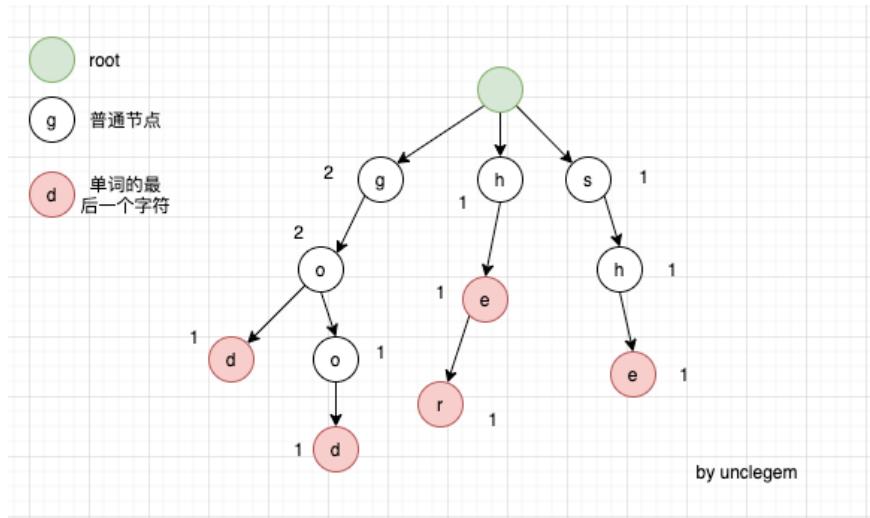
- 根结点无实际意义
- 每一个节点代表一个字符
- 每个节点中的数据结构可以自定义，如 isWord(是否是单词)，count(该前缀出现的次数)等，需实际问题实际分析需要什么。

Trie 的插入

- 假定给出几个单词如[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:

```
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;
        }
    }
}
```

```
        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:

```

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

```

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。

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。

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，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 $0 \leq s < n-m$ ，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\$。

暴力(BF)

核心思路

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。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）
- 初始化大小为模式串长的滑窗
- 固定当前窗口，将当前窗口的子串与模式串匹配，若匹配成功，则记录相关信息，如该位置下标
- 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
    end if
end for

```

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？

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 \$h1\$，对 \$P\$ 进行 hash 运算，生成 hash 值 \$h2\$，比对 \$h1\$ 和 \$h2\$，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\$。

该方法和 BF 主要步骤是一样的，如果每步都算子串的 hash 值，那么每步 hash 的时间复杂度为 \$O(m)\$，那么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)\$，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：

- 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 hash 值
- 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，若相等，则记录相关信息，如该位置下标
- 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）

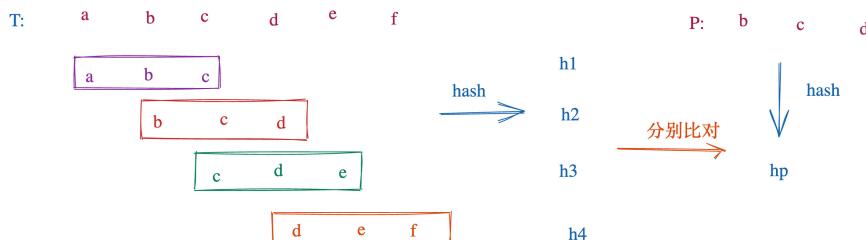
伪代码

```

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 \$

假设\$P\$和\$T\$全由\$d\$个字符组成的，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，我们更特殊的假设\$P\$和\$T\$全由[0-9]10个数字组成。\$P\$的10进制为：

$$f(P) = P[0] * 10^{(m-1)} + P[1] * 10^{(m-2)} \dots + P[m-1]$$

\$T[s..s+m]\$的10进制为：

$$f(T[s..s+m]) = T[s] * 10^{(m-1)} + T[s+1] * 10^{(m-2)} \dots + T[m-1]$$

\$T[s+1..s+m+1]\$可以根据\$T[s..s+m]\$推导

$$\begin{aligned} f(T[s+1..s+m+1]) &= T[s+1] * 10^{(m-1)} + T[s+2] * \dots \\ &= (f(T[s..s+m]) - T[s] * 10^{(m-1)}) * 10 + T[s+m+1] \end{aligned}$$

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？

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。但是这种方案并不完美，\$f(P)\%q==f(T[s])\%q\$并不能代表\$f(P)==f(T[s])\$。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。

KMP

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next数组，从而可以去匹配任意的主串。
- 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，BF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
```

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](#)

参考

- [维基百科](#)
- [从头到尾彻底理解 KMP](#)
- [Finite Automata algorithm for Pattern Searching](#)

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。

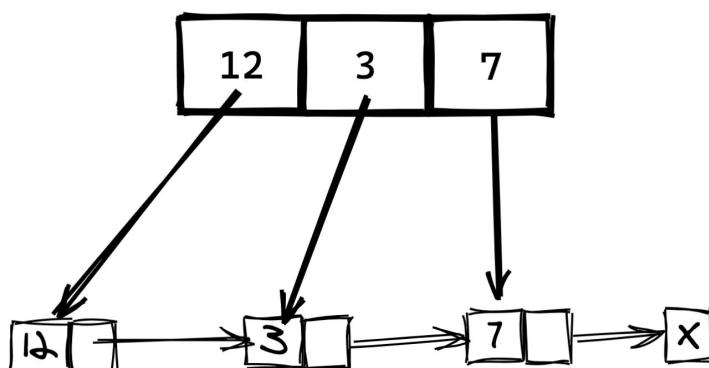
不借助额外空间的情况下，在链表中查找一个值，需要按照顺序一个个查找，时间复杂度为 $O(N)$ ，其中 N 为链表长度。



(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



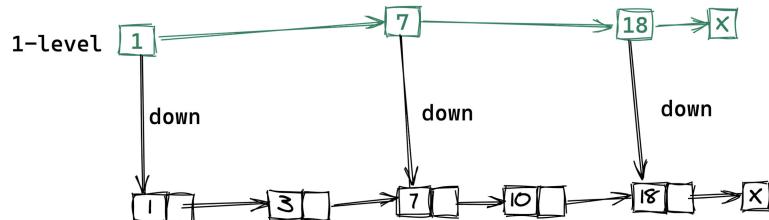
(单链表 + 哈希表)

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。

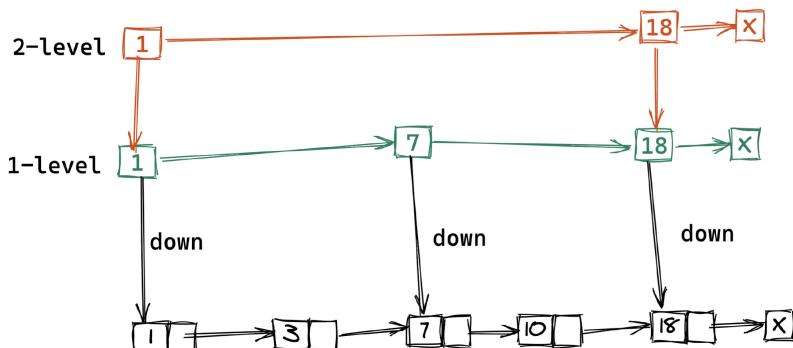


(建立一级索引)

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 $O(n)$ 。当然你如果没三个建立一个索引节点的话，空间会更省，但是复杂度不变。

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 $O(N)$ ；

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，相信大家都会用，所以这次的讲义希望将大家对剪枝中模糊不清对概念有一个比较清晰的认识。

剪枝的目的

上面也说了，我们剪枝就是为了降低我们算法的复杂度，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，日常生活中剪枝剪的就是树的枝桠，在搜索空间中剪枝剪掉的就是必得不到解的部分来减小搜索空间。

剪枝遵循的三原则

- 正确性：这个很好理解，我们把这个树杈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，那还搜个啥呢。
- 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。
- 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。
- 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。
- 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。

- 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。

练习剪枝

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超时。毫不夸张地说，剪枝剪得早，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

高频面试题

所谓高频题目指的是企业面试的过程中经常的题目。练习好这些题目，碰到原题和换皮题的可能性很大。即使没碰到碰到原题和换皮题，由于高频题基本覆盖了所有的题型，因此从查缺补漏，最大化利用自己时间的角度来看，掌握高频面试题也是非常有必要的。

题目来源

我们的《高频面试题》来源有以下几个：

1. 🔥 热题 HOT 100
2. 🎓 精选 TOP 面试题
3. 企业题库。比如 🚗 腾讯精选练习 50 题，企业题库 - 字节跳动
4. 剑指 Offer
5. 网友内幕（主要是面经）

建议：

- 先 1, 2，加起来一共 245。由于有重叠，因此实际上不到 245。
(刷 80% 以上)
- 剑指 Offer，一共 75 道。 (刷 80% 以上)
- 找到你想去的公司，针对性刷 50 道（各种难度和题型）企业题库 或者 面经的题目。

刷题顺序和时间分配（150 题）

- 递归 (10)
- BFS & DFS (20)
- 双指针 (20)
- 滑动窗口 (6)
- 哈希表 (20)
- 回溯 (5)
- 动态规划 (20)
- 排序 (3)
- 分治 (20)
- 堆 (3)
- 贪心 (5)
- 设计题 (5)
- 图 (5)
- 位运算 (5)
- 并查集 (3)

推荐几个题

- [797. 所有可能的路径 \(#图#DFS\)](#)
- [78. 子集 \(#位运算\)](#)

大纲

day01:二叉树遍历系列

- [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 \(迭代和递归\)](#)

day02: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](#)

day03:位运算系列

- [78.子集](#)
- [面试题 01.01. 判定字符是否唯一](#)

day04: 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](#)

day05:有效括号

- [20.有效的括号 要求 \$O\(1\)\$ 空间复杂度。](#)
- [32.最长有效括号 要求 \$O\(1\)\$ 空间复杂度。](#)

day06: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](#)

day07:前缀和系列

- [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](#)

堆

堆（英语：Heap）是计算机科学中的一种特别的树状数据结构。

堆的性质

- 在一个 最小堆(min heap) 中, 如果 P 是 C 的一个父级节点, 那么 P 的 key(或 value)应小于或等于 C 的对应值. 正因如此, 堆顶元素一定是最小的, 我们可以利用这个特点求最小值。或者结合 pop 操作 求第 k 小的值。
- 在一个 最大堆(max heap) 中, P 的 key(或 value)大于 C 的对应值。类似的, 我们可以利用这个特点求最大和第 k 大的值。

需要注意的是优先队列不仅有堆一种, 还有其它实现, 但这里我们把两者做等价。而堆的实现有很多, 本文提到的堆实现都是数组模拟完全二叉树。

堆的常见操作（以大顶堆为例）

- 插入

在堆中插入一个元素时, 一般把元素插入到堆的尾部, 然后对该元素进行上浮的操作（接下来会讲到）。

为什么选择在尾部插入, 然后将元素上浮, 而不是在头部插入, 然后将元素下沉呢?

- 删除

堆只能删除根节点, 不能删除其他的节点。由于堆顶被删除, 因此它的左右孩子需要继承它, 究竟选谁取决于谁更大, 这样逐渐下沉即可。为了操作简单, 我们可以在下沉之前与尾部节点进行一个交换。

为什么是尾部节点? 大家可以思考一下。

堆的恢复

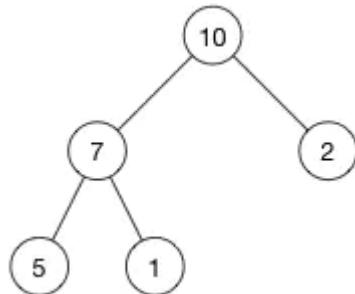
当我们向堆中插入一个元素或者从堆中删除一个元素时, 很容易破坏堆的性质。因此必须提供一种方式, 在堆被破坏的时候恢复堆的性质。那么如何恢复呢? 这里就需要借助接下来要讲的两个操作上浮和 下沉。

1. 上浮 shift_up

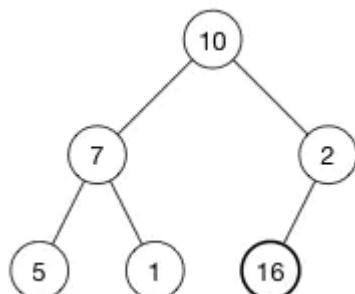
一般用于往一个堆的末尾中添加一个新元素，此时需要将上浮的节点与父节点比较，如果上浮的节点小于父节点，就将上浮节点与父节点交换位置，不断重复操作，直至上浮节点移到正确的位置。因此这里会有一个类似 `while` 循环的东西。

由于调整至设计当前元素和它的祖先元素，因此时间复杂度为取决于树的高度，如果是完全二叉树，那么树的高度就是对数，也就是 $O(\log N)$ 。

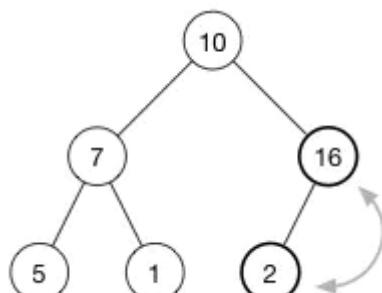
例如：新建一个大顶堆



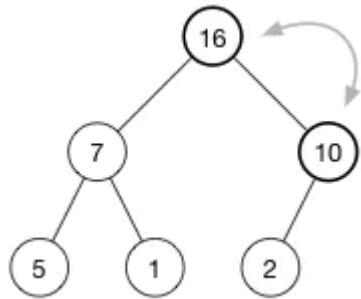
插入新元素 16



然后将新元素与父节点进行比较，如果当前节点大于父节点则进行上浮操作



重复上一步，直到根节点或者小于父节点时停止



```

var Heap = function(k) {
    this.data = new Array(K + 1)
    this.data[0] = null
};

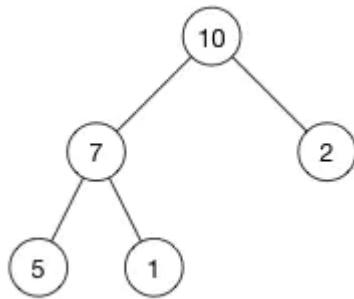
Heap.prototype.up = function(index) {
    // 如果循环到了根节点则结束递归
    if(index <= 1) return
    // 获取当前节点的父节点
    let parentIndex = parseInt(index / 2)
    // 如果当前节点大于父节点，则进行上浮操作
    if(this.data[index] > this.data[parentIndex]){
        this.swap(index, parentIndex)
        // 递归执行上浮操作
        this.up(parentIndex)
    }
};

Heap.prototype.swap = function(from, to) {
    let temp = this.data[from]
    this.data[from] = this.data[to]
    this.data[to] = temp
}
  
```

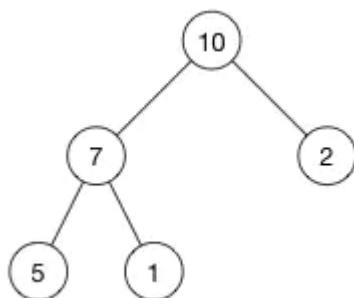
1. 下沉 shift_down

一般用于从堆中删除元素的时候，此时需要将下沉的节点与左右子节点比较，如果当下沉的节点小于左右子节点，就将下沉节点与子节点交换位置，不断重复操作，直至下沉节点移到正确的位置。因此这里会有一个类似 while 循环的东西。由于调整至设计当前元素和它的子节点中的一个，因此时间复杂度为取决于树的高度，如果是完全二叉树，那么树的高度就是对数，也就是 $O(\log N)$ 。

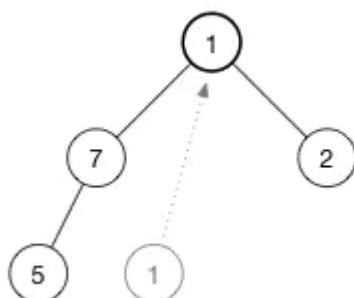
例如：新建一个大顶堆



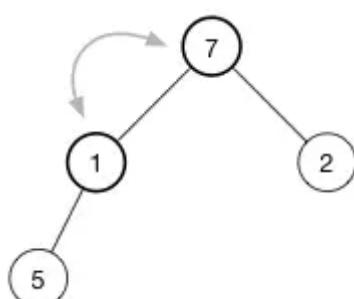
删除根节点，注意只能删除根节点



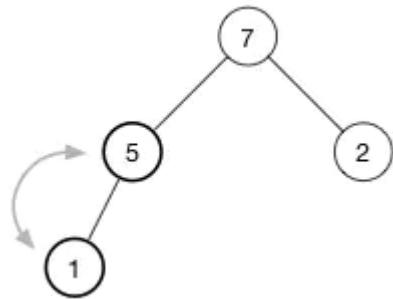
让尾节点成为根节点，



对节点的左右子节点进行比较，如果小于左右子节点，则进行下沉操作



重复上一步，直到最底层或者大于左右子节点时停止



```

var Heap = function(k) {
    this.data = new Array(K + 1)
    this.data[0] = null
};

Heap.prototype.down = function(index) {
    // 执行到了最底层，则结束递归
    if(index >= this.data.length - 1) return
    // 获取左右子节点
    let leftSon = index * 2,
        rightSon = index * 2 + 1,
        target = index
    if(leftSon < this.data.length && this.data[target] < this.data[leftSon])
        target = leftSon
    if(rightSon < this.data.length && this.data[target] < this.data[rightSon])
        target = rightSon
    // 如果父节点小于某个子节点，则进行下沉操作
    if(target != index) {
        this.swap(target, index)
        // 递归执行下沉
        this.down(target)
    }
};

Heap.prototype.swap = function(from, to) {
    let temp = this.data[from]
    this.data[from] = this.data[to]
    this.data[to] = temp
}
  
```

你可以形象地将上浮和下沉的过程经过的节点综合看成一个路径。

堆的常见题型

利用对的性质我们可以做很多有趣的事情，这里我举几个例子。

1. 取第 top K 的数

维护一个大小为 K 的堆，直接取堆顶即可。比如求第 K 大，则维护一个大小为 K 的小顶堆，依次将数据放入堆中，当堆的大小满了的时候，只需要将堆顶元素与下一个数比较：如果大于堆顶元素，则将当前的堆顶元素抛弃，并将该元素插入堆中。遍历完全部数据，堆中剩下的是按照小顶堆排列的最大的 K 个数，由于堆顶是最小的，因此 Top K 的元素就是堆顶元素。如果是求前 K 个最小的数，只需要改为大顶堆即可。

1. 求某一个数组的中位数

假如数组是有序（例如升序）的，那中位数会将数组分成两半。如果我们将其左边 $n/2$ 个较小的元素构建成一个大顶堆，再将右边 $n/2$ 个较大元素构建成一个小顶堆。那么：

- 数组数量是偶数时，中位数为：（大顶堆堆顶 + 小顶堆的堆顶）/ 2
- 数组数量为奇数，中位数为：数量较多的堆的堆顶元素
- 堆排序

3.1. 建堆

3.2. 删除顶部元素

3.3. 调整堆

3.4. 循环 2 和 3

代码

建堆

Python:

```
import heapq
nums=[1,8,2,23,7,-4,18,23,42,37,2]
print(heapq.nlargest(3,nums))# [42, 37, 23]
print(heapq.nsmallest(3,nums)) # [-4, 1, 2]
```

堆化

```
import heapq
nums=[1,8,2,23,7,-4,18,23,42,37,2]
heapq.heapify(heap)# 将列表原地转换成堆 [-4, 2, 1, 23, 7, 2, 18, 37, 42, 8, 1, 23]
```

push 和 pop

```
# heap: [-4, 2, 1, 23, 7, 2, 18, 23, 42, 37, 8]
heapq.heappop(heap) #-4
heapq.heappop(heap) # 1
heapq.heappop(heap) # 2
```



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。

前提

- 数组是有序的（如果无序，我们也可以考虑排序，不过要注意排序的复杂度）

术语

二分查找中使用的术语：

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

常见题型

查找一个数

算法描述：

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

复杂度分析

- 平均时间复杂度： $O(\log N)$
- 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。我们将范围缩小写到了 3 的右侧。
- 此时中间元素为 4, 正好是我们要找的, 返回其索引 2 即可。

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

这里给大家介绍一个我经常使用的思维框架和代码模板。

思维框架

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

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

- 由于定义的搜索区间为 **[left, right]**, 因此当 $left \leq right$ 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

```
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

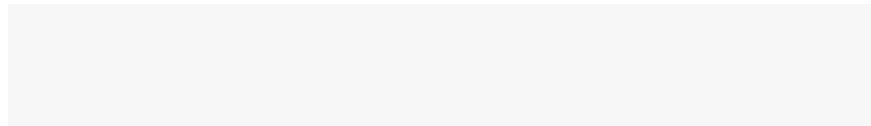
```
def binarySearch(nums, target):  
    # 左右都闭合的区间 [l, r]  
    l, r = 0, len(nums) - 1  
    while l <= r:  
        mid = (left + right) >> 1  
        if nums[mid] == target: return mid  
        # 搜索区间变为 [mid+1, right]  
        if nums[mid] < target: l = mid + 1  
        # 搜索区间变为 [left, mid -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

JavaScript

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

C++

暂时空缺，欢迎 PR



寻找最左边的满足条件的值

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

思维框架

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

代码模板

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

Java

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

Python

```
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++

暂时空缺，欢迎 PR

寻找最右边的满足条件的值

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

有没有感受到框架和模板的力量？

思维框架

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

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

- 由于我们定义的搜索区间为 $[left, right]$ ， 因此当 $left \leq right$ 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

```

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

```

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

```

C++

暂时空缺，欢迎 PR

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。

不管是寻找最左插入位置还是后面的寻找最右插入位置，我们的更新指针代码都是一样的。即：

```

l = mid + 1
# or
r = mid

```

当然也有别的方式（比如 `mid` 不是向下取整，而是向上取整），但是这样可以最小化记忆成本。

思维框架

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

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

- 由于我们定义的搜索区间为 `[left, right]`，因此当 `left <= right` 的时候，搜索区间都不为空。但由于上面提到了更新条件有一个 `r = mid`，因此如果结束条件是 `left <= right` 则会死循环。因此结束条件是 `left < right`。

有的人有疑问，这样设置结束条件会不会漏过正确的解，其实不会。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，搜索区间不为空。如果我们的答案恰好是 4，会被错过么？不会，因为我们直接返回了 4。

- 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]` 大于等于目标值，`r` 也可能是目标解，`r - 1` 可能会错过解，因此我们使用 `r = mid`。
 - 如果 `nums[mid]` 小于目标值，`mid` 以及 `mid` 左侧都不可能是解，因此我们使用 `l = mid + 1`。
- 最后直接返回 `l` 或者 `r` 即可。（并且不需要像 最左满足条件的值 那样判断了）

代码模板

Python

```
def bisect_left(nums, x):
    # 内置 api
    bisect.bisect_left(nums, x)
    # 手写
    l, r = 0, len(nums) - 1
    while l < r:
        mid = (l + r) // 2
        if nums[mid] < x:
            l = mid + 1
        else:
            r = mid
    # 由于 l 和 r 相等，因此返回谁都无所谓。
    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

和 寻找最左插入位置 类似。不同的地方在于：如果有多个满足条件的值，我们返回最右侧的。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 4。

- 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。但由于上面提到了更新条件有一个 `r = mid`，因此如果结束条件是 `left <= right` 则会死循环。因此结束条件是 `left < right`。

有的人有疑问，这样设置结束条件会不会漏过正确的解，其实不会。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，搜索区间不为空。如果我们的答案恰好是 4，会被错过么？不会，因为我们直接返回了 4。

- 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]` 小于等于目标值，`mid` 以及 `mid` 左侧都不可能是解，因此我们使用 `l = mid + 1`。
 - 如果 `nums[mid]` 大于目标值，`r` 也可能是目标解，`r - 1` 可能会错过解，因此我们使用 `r = mid`。
- 最后直接返回 `l` 或者 `r` 即可。（并且不需要像 最左满足条件的值 那样判断了）

代码模板

Python

```
def bisect_right(nums, x):
    # 内置 api
    bisect.bisect_right(nums, x)
    # 手写
    l, r = 0, len(nums) - 1
    while l < r:
        mid = (l + r) // 2
        if nums[mid] > x:
            r = mid
        else:
            l = mid + 1
    # 由于 l 和 r 相等，因此返回谁都无所谓。
    return l
```

其他语言暂时空缺，欢迎 [PR](#)

局部有序（先降后升或先升后降）

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

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

81. 搜索旋转排序数组 II

题目描述

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

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

编写一个函数来判断给定的目标值是否存在于数组中。若存在返回 `true`, 否则:

示例 1:

输入: `nums = [2,5,6,0,0,1,2], target = 0`

输出: `true`

示例 2:

输入: `nums = [2,5,6,0,0,1,2], target = 3`

输出: `false`

进阶:

这是 搜索旋转排序数组 的延伸题目, 本题中的 `nums` 可能包含重复元素。

这会影响到程序的时间复杂度吗? 会有怎样的影响, 为什么?

思路

这是一个我在网上看到的前端头条技术终面的一个算法题。我们先不考虑重复元素。

题目要求时间复杂度为 $\log n$, 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

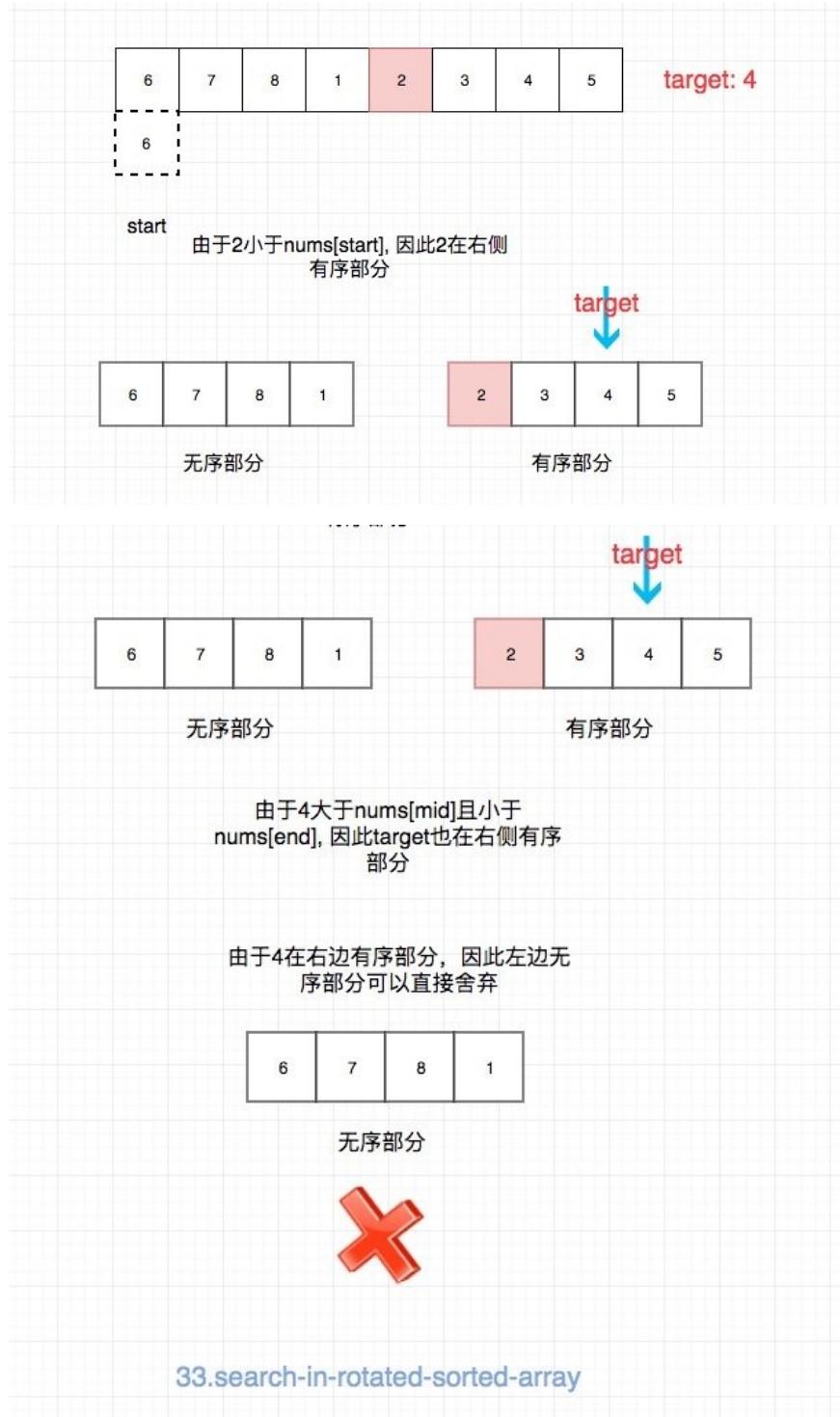
假如 `mid` 小于 `start`, 则 `mid` 一定在右边有序部分。假如 `mid` 大于 `start`, 则 `mid` 一定在左边有序部分。

注意我没有考虑等号, 之后我会讲。

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

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

我们以 $([6,7,8,1,2,3,4,5], 4)$ 为例讲解一下:



33.search-in-rotated-sorted-array

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

代码 (Python)

```
class Solution:
    def search(self, nums, target):
        l, r = 0, len(nums)-1
        while l <= r:
            mid = l + (r-l)//2
            if nums[mid] == target:
                return True
            while l < mid and nums[l] == nums[mid]: # 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: $O(1)$

二维数组

二维数组的二分查找和一维没有本质区别， 我们通过两个题来进行说明。

74. 搜索二维矩阵

题目地址

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

题目描述

编写一个高效的算法来判断 $m \times n$ 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):
        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

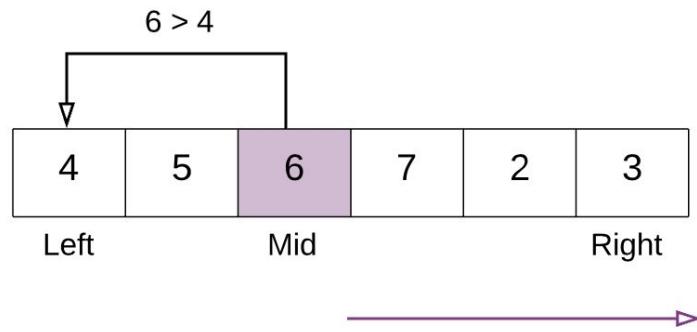
我们当然也可以和 $\text{nums}[l]$ 比较, 而不是上面的 $\text{nums}[r]$, 我们发现:

- 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:

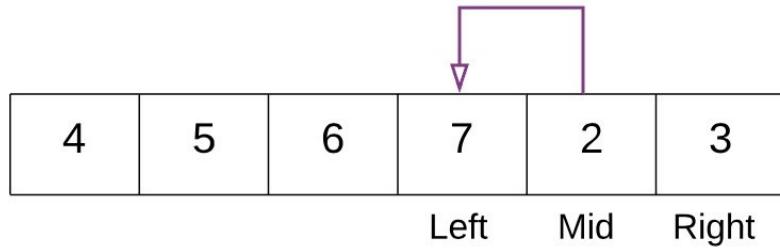
- 找到数组的中间元素 mid 。
- 如果中间元素 $>$ 数组第一个元素, 我们需要在 mid 右边搜索。



- 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

```

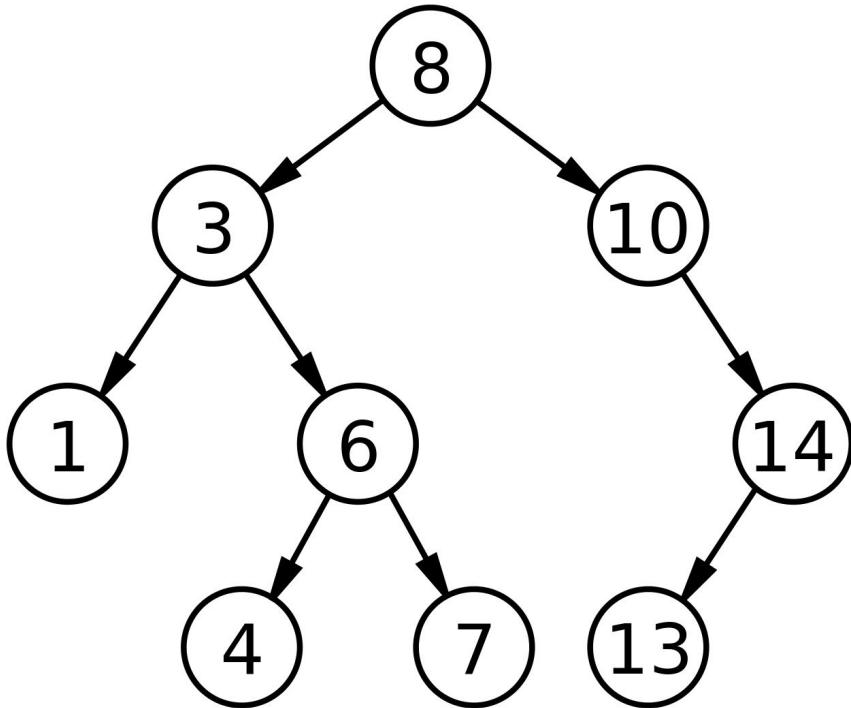
复杂度分析

- 时间复杂度: $O(\log N)$
- 空间复杂度: $O(1)$

二叉树

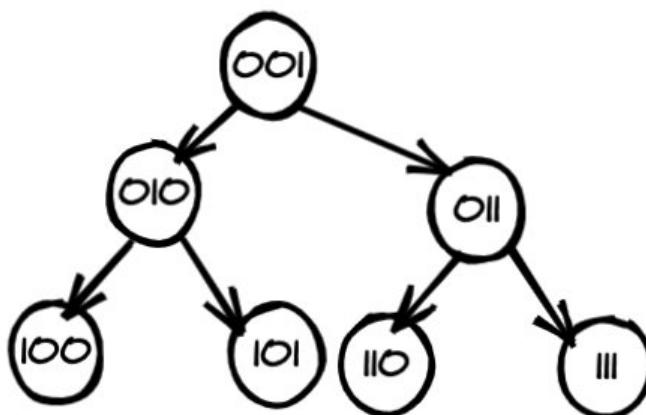
对于一个给定的二叉树，其任意节点最多只有两个子节点。从这个定义，我们似乎可以嗅出一点二分法的味道，但是这并不是二分。但是，二叉树中却和二分有很多联系，我们来看一下。

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。

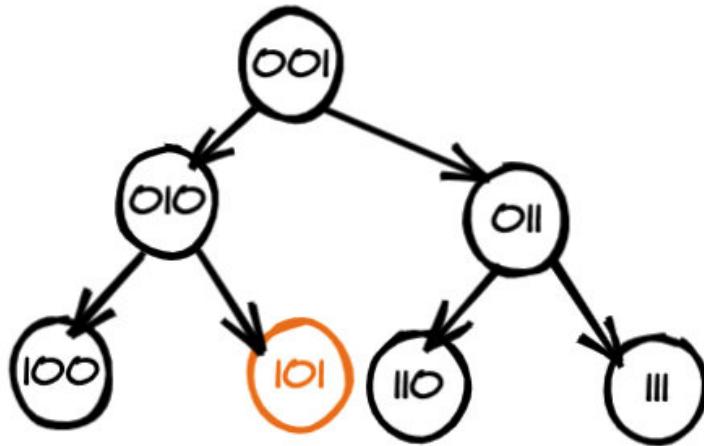


如上图，我们需要在这样一个二叉搜索树中搜索 7。那么我们的搜索路径则会是 $8 \rightarrow 3 \rightarrow 6 \rightarrow 7$ ，这也是一种二分法。只不过相比于普通的有序序列查找给定值二分，其时间复杂度的下界更差，原因在于二叉搜索树并不一定是二叉平衡树。

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

总结

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

二分查找的基本题型有：

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

不管是哪一种类型，我们的思维框架都是类似的，都是：

- 先定义搜索区间（非常重要）

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

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

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

特别需要注意的是有无重复元素对二分算法影响很大，我们需要小心对
待。

滑动窗口

简介

滑动窗口（以下简称滑窗），也就是 Sliding Window，该算法思想如其名，是利用双指针在某种数据结构上（大部分基本上都是 Array）虚构出了一个窗口，并且该窗口还会按规则向后移动，最终找到全局解。相信大家在平常会经常用到，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。

目的

该思想的目的是很明确的，可以简单想一下，求数组连续 k 个数的最大值，若不采用滑窗而采用暴力求解也即如下写法（简单写了核心部分）：

```
res = -1

for i in range(0, len(n) - k + 1):
    cur_sum = 0
    for j in range(i, i + k):
        cur_sum += n[j]
    res = max(res, cur_sum)

return res
```

- 不难发现时间复杂度已经达到了 $O(N^2)$ ，因为我们每移动一次都抛弃了前次的信息，而是重新计算，效率大打折扣。
- 相信大家经过了两个月的训练，不难发现，我们每移动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，也就是新窗口元素和 = 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 = 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。也就是滑动窗口主要是为了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（个人理解哈）。

滑窗思想的基本步骤（牢记，并按照自己的习惯总结出一套模版或框架）

其实上面的是属于窗口大小固定的解决方案，为的是引出使用该方法的思路历程，当然还有窗口大小不固定的情况，我们可以将算法流程大致抽象出以下三个步骤：

- 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。
- 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。
- 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。

下面给出个参考伪代码：

```
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
```

注意

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 leetcode 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。

总结

TODO

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

```

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

灵活使用位运算来解决或者降低时空复杂度

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],  
[]  
]
```

这一道题之前出现过，有兴趣的可以看看点击查看使用其他方式来解这道题[官方题解](#)

我们来灵活的使用位运算的性质，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;
};
```

时空间复杂度均为 $O(n)$

二进制的思维角度

- 从老鼠试毒问题来看二分法

更多

- 力扣专题 - 位运算

总结

TODO

搜索

介绍

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，双向搜索，A*算法等。这里只介绍DFS/BFS，以及发生在DFS上一种技巧-回溯。

分类

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。

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

```
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)
        }
    }
}
```

BFS

BFS 也是图论中算法的一种，不同于 DFS 的是 BFS 采用横向搜索的方式，在数据结构上通常采用队列结构。

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。
 - 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
}

```

回溯

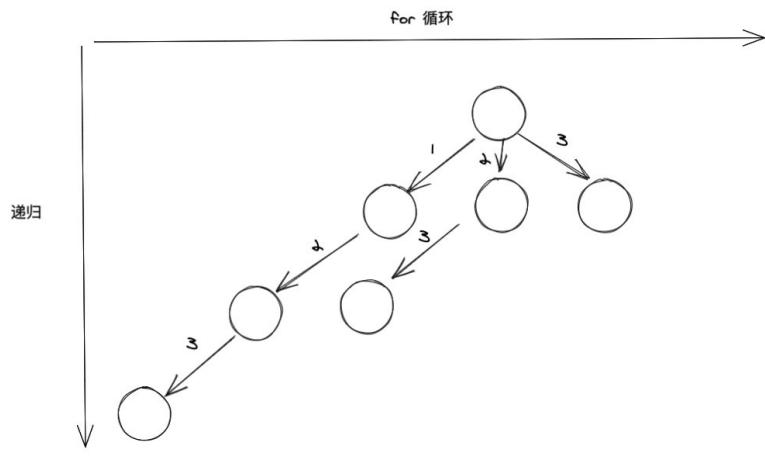
回溯是 DFS 中的一种技巧。回溯法采用 [试错](#) 的思想，它尝试分步的去解决一个问题。在分步解决问题的过程中，当它通过尝试发现现有的分步答案不能得到有效的正确的解答的时候，它将取消上一步甚至是上几步的计算，再通过其它的可能的分步解答再次尝试寻找问题的答案。

通俗上讲，回溯是一种走不通就回头的算法。

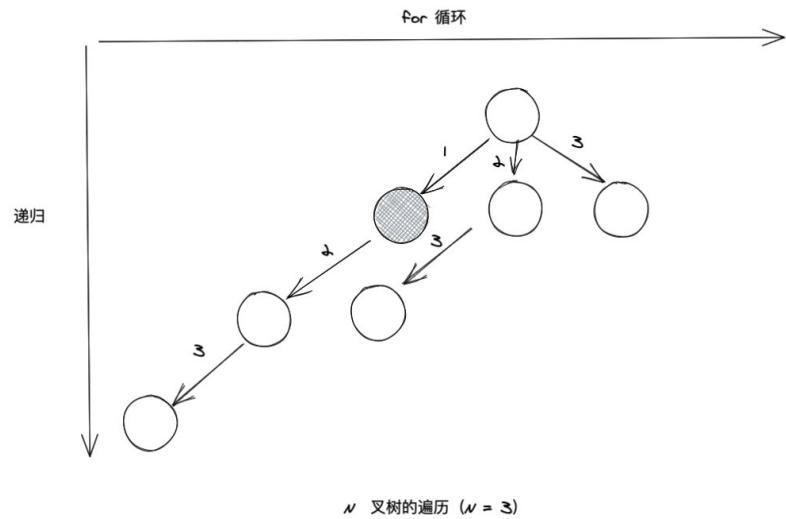
回溯的本质是穷举所有可能，尽管有时候可以通过剪枝去除一些根本不可能是答案的分支，但是从本质上讲，仍然是一种暴力枚举算法。

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数，递归的深度，都构成的树的高度。

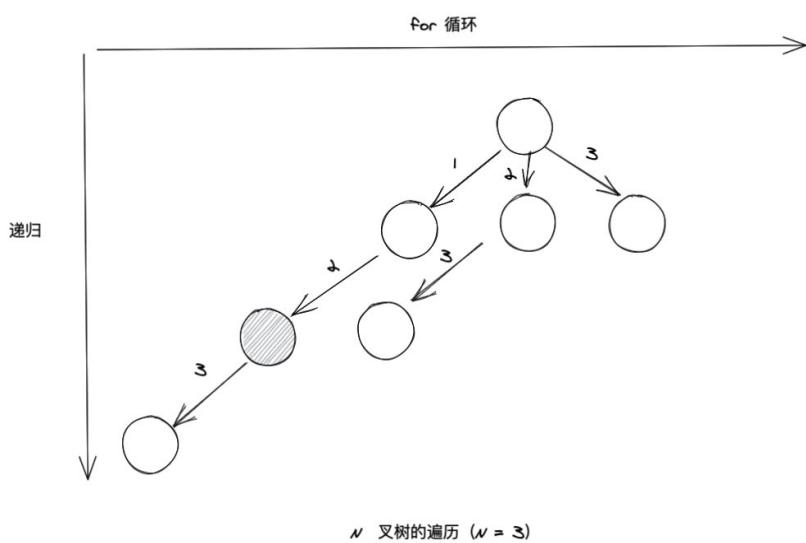
以求数组 [1,2,3] 的子集为例：



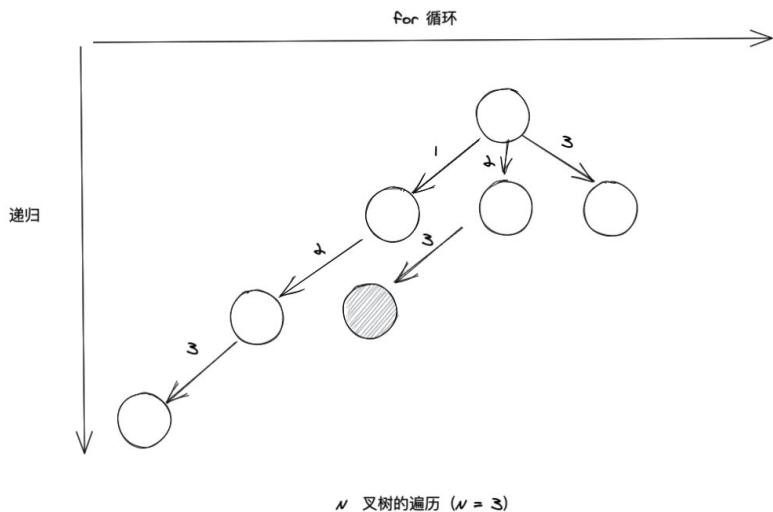
以上图来说， 我们会在每一个节点进行加入到结果集这一次操作。



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
}

```

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。

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](#)

相关专题

- [小岛问题](#)
- [深度优先遍历](#)
- [回溯算法](#)

总结

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。

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。

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

相信大家现在对动态规划都有了一定的认识和使用，可能部分同学也接触过背包问题，下面给大家归纳几种常用的背包问题及其对应模版，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也可以的。

- 01 背包问题：问题描述为有 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。

简单分析下：因为每个物品要么选要么不选，你要是搜索，那是 2^N 复杂度， N 稍微大点就很恐怖了。 dp 可以将复杂度降到 $O(NW)$ ，大大简化问题，那么我们来看看状态转移方程如何定义： $dp[i][j]$ 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）： $dp[i - 1][j - w[i]] + v[i]$ ， $w[i]$ 是第 i 个物品的重量， $v[i]$ 是第 i 个物品的价值。
- 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：

$$dp[i][j] = \max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i]), j \geq [i]$$

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]

```

这种解法其实在部分题中是不能完全AC的，因为空间复杂度是NM，不难发现当前

```

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]

```

这就是01背包问题。

- 完全背包问题：问题描述为有 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。

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）： $dp[i][j] - w[i] + v[i]$, $w[i]$ 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：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]

```

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](#)。

这是我的最后一个专题，谢谢大家！

递归和动态规划

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

递归

定义：递归是指在函数的定义中使用函数自身的方法。

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

纯粹的函数式编程中没有循环，只有递归。

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，当子问题缩小到寻常的时候，我们可以知道它的解。然后我们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，我们通常借助递归去分解问题而已。

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），也就是顺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(n):
    return n + f(n - 1)
```

更多的情况应该是：

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

练习递归

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

如果你已经对递归比较熟悉了，那么我们继续往下看。

递归中的重复计算

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

递归的时间复杂度分析

敬请期待我的新书。

小结

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

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

当你已经适应了递归的时候，那就让我们继续学习动态规划吧！

动态规划

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

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

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

爆栈

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

代码：

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

我们用递归树来直观地看一下。



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

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

重复计算

我们再举一个重复计算的例子，问题描述：

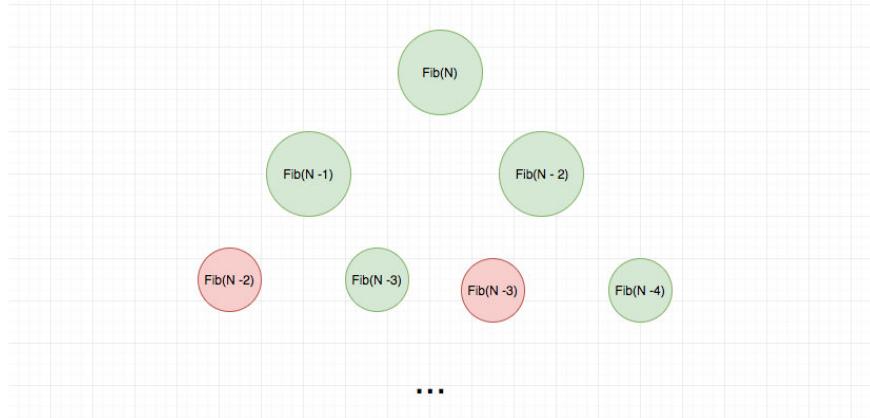
一个人爬楼梯，每次只能爬 1 个或 2 个台阶，假设有 n 个台阶，那么这个人有多少种不同的爬楼梯方法？

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。

递归代码：

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

我们继续用一个递归树来直观感受以下：



红色表示重复的计算

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

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

如果上面的爬楼梯问题，使用动态规划，代码是这样的：

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

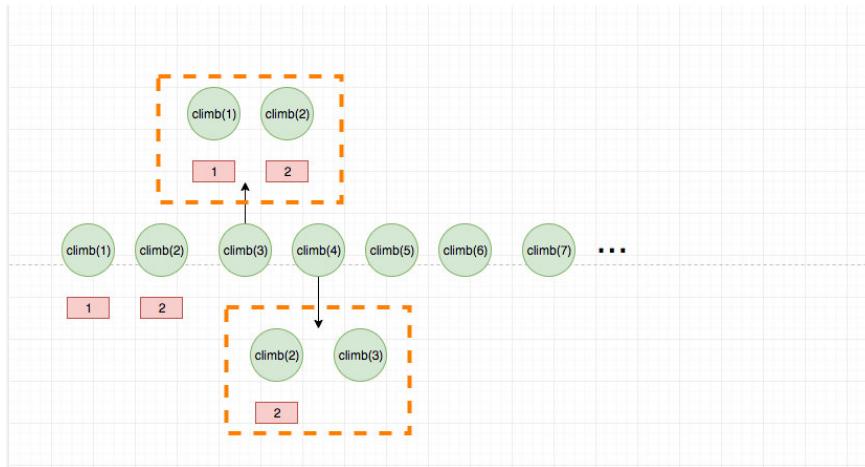
    for (let i = 2; i < n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[dp.length - 1];
}
```

不会也没关系，我们将递归的代码稍微改造一下。其实就是将函数的名字改一下：

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

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

动态规划的查表过程如果画成图，就是这样的：



虚线代表的是查表过程

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

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

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

```

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

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

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

    return temp;
}

```

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

再次强调一下：

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

动态规划的三个要素

1. 状态转移方程
2. 临界条件
3. 枚举状态

可以看出，用递归解决也是一样的思路

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

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

我用动态规划的形式表示一下：

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

可以看出两者是多么的相似。

实际上临界条件相对简单，大家只有多刷几道题，里面有感觉。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。

不过状态的定义都有特点的套路。比如一个字符串的状态，通常是 $dp[i]$ 表示字符串 s 以 i 结尾的。比如两个字符串的状态，通常是 $dp[i][j]$ 表示字符串 s_1 以 i 结尾， s_2 以 j 结尾的。

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

搞定了状态的定义，那么我们来看下状态转移方程。

状态转移方程

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

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

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

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

如何枚举状态

前面说了如何枚举状态，才能不重不漏是枚举状态的关键所在。

- 如果是一维状态，那么我们使用一层循环可以搞定。
- 如果是二维状态，那么我们使用两层循环可以搞定。
- . . .

这样可以保证不重不漏。

但是实际操作的过程有很多细节比如：

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

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

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

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

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

二维的也是一样的，大家可以试试。

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

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

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

而：

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

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

- 关于里外循环的问题，其实和上面原理类似。

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

小结

关于如何确定临界条件通常是比较简单的，多做几个题就可以快速掌握。

关于如何确定状态转移方程，这个其实比较困难。不过所幸的是，这些套路性比较强，比如一个字符串的状态，通常是 $dp[i]$ 表示字符串 s 以 i 结尾的。比如两个字符串的状态，通常是 $dp[i][j]$ 表示字符串 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。

动态规划为什么要画表格

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

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

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

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

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

- 应该选择它
- 还是不选择它

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

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

相关问题

- [0091.decode-ways](#)
- [0139.word-break](#)
- [0198.house-robber](#)

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

总结

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

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

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

另外，大家可以去 LeetCode 探索中的 [递归 I](#) 中进行互动式学习。

定义

分治算法的基本思想是将一个规模为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。这些子问题相互独立且与原问题性质相同，求出子问题的解，就可得到原问题的解。

分治法顾名思义由分和治来组成。

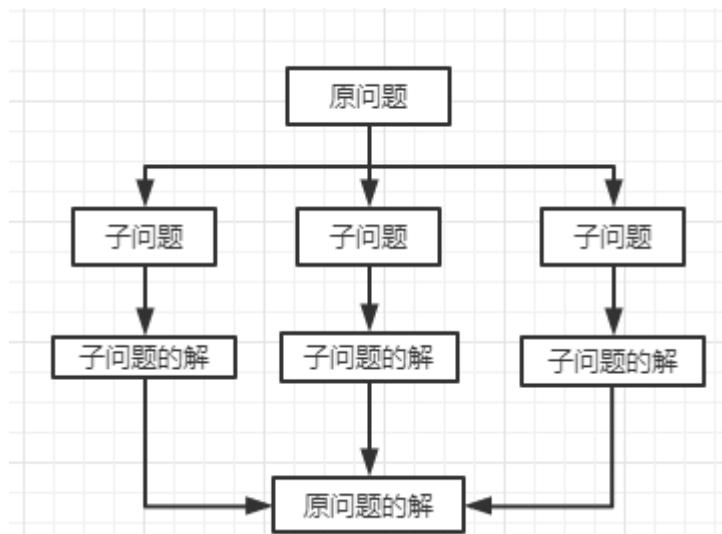
- 分。将一个规模为N的问题分解为K个规模较小的子问题
- 治。根据子问题的解求原问题

常用场景

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
- 2) 这些被分解的问题的结果可以进行合并
- 3) 这些被分解的问题是相互独立的，不包含重叠的子问题（类似动态规划？）

解题步骤



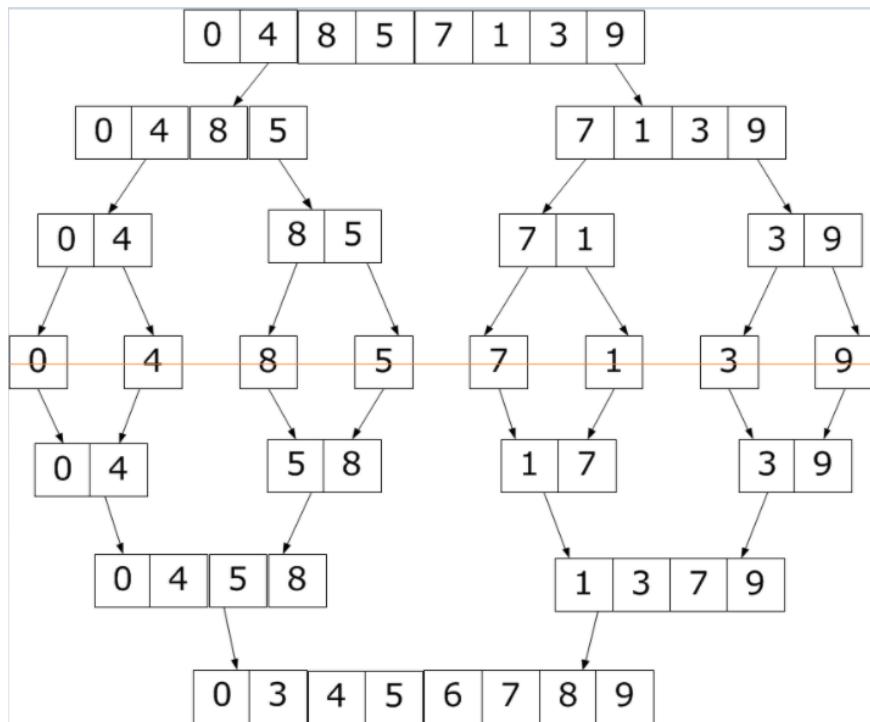
1. 将原问题分解至，达到求解边界的结构相同互相独立的子问题
2. 对所有的子问题进行求解
3. 将所有子问题的解进行合并，从而得到原问题

思路

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)与求解思路
2. 思考如果将子问题的解进行合并

3. 思考编码思路（一般利用递归）

例如经典的归并排序算法



1. 如果对成千上万个数据进行排序，肯定头大，但是如果规模缩小到只有一个元素时，问题就变得很容易解了
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）
2. 编码思路就很清晰了，
3. 将问题不断进行二分，直到所有的子问题的规模为1个元素
4. 对子问题进行求解
5. 利用合并两个有序数组的思路，将子问题两两进行合并
6. 得到最终解

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))
}
```

经典题型

面试题 08.06. 汉诺塔问题

96. 不同的二叉搜索树

例如96题

我们拿到题目可以发现该题存在以下特点

1. 如果数据规模减小时（例如只有2个节点时），求解会变得简单
2. 问题可以进行拆分，并且子问题相互独立
3. 知道了左右子树的排列情况即可求解，子问题的结果可以进行合并

按照上面提到的思考模板

1. 当给定的n小于2时我们可以直接得出答案，所以子问题的最小边界为2
2. 当我们知道左子树的节点数与右子树的节点数时，

即可得到所有的排列情况 ==> (左子树的排列情况总数) * (右子树的排列情况总数)

对子问题进行合并

- 很可能会出现大量的重复计算，所以可以把之前计算出来的结果存起来

代码 (JS) :

```
var numTrees = function(n) {
    let memory = new Array(n + 1)
    memory[0] = memory[1] = 1
    return getNum(n, memory)
};

function getNum(n, memory){
    if(memory[n]){
        return memory[n]
    }
    let ans = 0
    for(let i = 1;i <= n; i++){
        ans += getNum(i - 1, memory) * getNum(n - i, memory)
    }
    return memory[n] = ans
}
```

这道题也可以不用递归，使用动态规划的思路迭代求解，效率会更高



基础篇 - 01. 数组，栈，队列

【Day 1】 2020-11-01 - 66. 加一

- [官方题解](#)

【Day 2】 2020-11-02 - 821. 字符的最短距离

- [官方题解](#)
- [精选题解](#)

【Day 3】 2020-11-03 - 1381. 设计一个支持增量操作的栈

- [官方题解](#)

【Day 4】 2020-11-04 - 394. 字符串解码

- [官方题解](#)

【Day 5】 2020-11-05 - 232. 用栈实现队列

- [官方题解](#)

【Day 6】 2020-11-06 - 768. 最多能完成排序的块 II

- [官方题解](#)
- [精选题解](#)

基础篇 - 02. 链表

【Day 7】 2020-11-07 - 24. 两两交换链表中的节点

- [官方题解](#)
- [精选题解](#)

【Day 8】 2020-11-08 - 61. 旋转链表

- [官方题解](#)

【Day 9】 2020-11-09 - 109. 有序链表转换二叉搜索树

- [官方题解](#)

【Day 10】 2020-11-10 - 160. 相交链表

- [官方题解](#)

- 精选题解

【Day 11】 2020-11-11 - 142. 环形链表 II

- 官方题解
- 精选题解

【Day 12】 2020-11-12 - 146. LRU 缓存机制

- 官方题解
- 精选题解

基础篇 - 03.树

【Day 13】 2020-11-13 - 104. 二叉树的最大深度

- 官方题解
- 精选题解
- 精选题解

【Day 14】 2020-11-14 - 100. 相同的树

- 官方题解
- 精选题解

【Day 15】 2020-11-15 - 129. 求根到叶子节点数字之和

- 官方题解

【Day 16】 2020-11-16 - 513. 找树左下角的值

- 官方题解
- 精选题解

【Day 17】 2020-11-17 - 297. 二叉树的序列化与反序列化

- 官方题解

【Day 18】 2020-11-18 - 987. 二叉树的垂序遍历

- 官方题解
- 精选题解

基础篇 - 04. 哈希表

【Day 19】 2020-11-19 - 1. 两数之和

- [官方题解](#)
- [精选题解](#)

【Day 20】 2020-11-20 - 347. 前 K 个高频元素

- [官方题解](#)
- [精选题解](#)

【Day 21】 2020-11-21 - 447. 回旋镖的数量

- [官方题解](#)
- [精选题解](#)

【Day 22】 2020-11-22 - 3.无重复字符的最长子串

- [官方题解](#)
- [精选题解](#)

【Day 23】 2020-11-23 - 30. 串联所有单词的子串

- [官方题解](#)

【Day 24】 2020-11-24 - 30. 解数独

- [官方题解](#)

基础篇 - 05. 双指针

【Day 25】 2020-11-25 - 35. 搜索插入位置

- [官方题解](#)

【Day 26】 2020-11-26 74. 搜索二维矩阵

- [官方题解](#)

【Day 27】 2020-11-27 26. 删除排序数组中的重复项

- [官方题解](#)

【Day 28】 2020-11-28 876. 链表的中间结点

- [官方题解](#)

【Day 29】 2020-11-29 1052. 爱生气的书店老板

- [官方题解](#)

【Day 30】 2020-11-30 239. 滑动窗口最大值

- [官方题解](#)

题目地址(66. 加一)

<https://leetcode-cn.com/problems/plus-one>

题目描述

给定一个由整数组成的非空数组所表示的非负整数，在该数的基础上加一。

最高位数字存放在数组的首位，数组中每个元素只存储单个数字。

你可以假设除了整数 0 之外，这个整数不会以零开头。

示例 1：

输入： [1,2,3]

输出： [1,2,4]

解释： 输入数组表示数字 123。

示例 2：

输入： [4,3,2,1]

输出： [4,3,2,2]

解释： 输入数组表示数字 4321。

lucifer 提示：不要加直接数组转化为数字做加法再转回来。

前置知识

- 数组的遍历(正向遍历和反向遍历)

思路

这道题其实我们可以把它想象成小学生练习加法，只不过现在是固定的“加一”那么我们只需要考虑如何通过遍历来实现这个加法的过程就好了。

加法我们知道要从低位到高位进行运算，那么只需要对数组进行一次反向遍历即可。

伪代码：

```
for(int i = n - 1; i > - 1; i --) {  
    内部逻辑  
}
```

内部逻辑的话，其实有三种情况：

1. 个位上的数字小于9

$$\begin{array}{r} 17 \\ + \quad 1 \\ = \quad 18 \end{array}$$

2. 个位数上等于9，其他位数可以是0–9的任何数，但是首位不等于9

$$\begin{array}{r} 199 \\ + \quad 1 \\ = \quad 200 \end{array}$$

$$\begin{array}{r} 109 \\ + \quad 1 \\ = \quad 110 \end{array}$$

3. 所有位数都为9

$$\begin{array}{r} 99 \\ + \quad 1 \\ = \quad 100 \end{array}$$

$$\begin{array}{r} 999 \\ + \quad 1 \\ = \quad 1000 \end{array}$$

第一种情况是最简单的，我们只需将数组的最后一位进行+1 操作就好了

第二种情况稍微多了一个步骤：我们需要把个位的 carry 向前进一位并在计算是否有更多的进位

第三种其实和第二种是一样的操作，只是由于我们知道数组的长度是固定的，所以当我们遇到情况三的时候需要扩大数组的长度。我们只需要在结果数组前多加上一位就好了。

```
// 首先我们要从数组的最后一位开始我们的计算得出我们新的sum
sum = arr[arr.length - 1] + 1

// 接下来我们需要判断这个新的sum是否超过9
sum > 9 ?

// 假如大于 9, 那么我们会更新这一位为 0 并且将carry值更改为1
carry = 1
arr[i] = 0

// 假如不大于 9, 更新最后一位为sum并直接返回数组
arr[arr.length - 1] = sum
return arr

// 接着我们要继续向数组的倒数第二位重复进行我们上一步的操作
...

// 当我们完成以后, 如果数组第一位时的sum大于0, 那么我们就要给数组的首
result = new array with size of arr.length + 1
result[0] = 1
result[1] ..... result[result.length - 1] = 0 //
```

代码

代码支持: Python3, JS

Python3 Code:

```
class Solution:
    def plusOne(self, digits: List[int]) -> List[int]:
        carry = 1
        for i in range(len(digits) - 1, -1, -1):
            digits[i], carry = (carry + digits[i]) % 10, (
        return [carry] + digits if carry else digits
```

JS Code:

```
var plusOne = function (digits) {
    var carry = 1; // 我们将初始的 +1 也当做是一个在个位的 carry
    for (var i = digits.length - 1; i > -1; i--) {
        if (carry) {
            var sum = carry + digits[i];
            digits[i] = sum % 10;
            carry = sum > 9 ? 1 : 0; // 每次计算都会更新下一步需要用到
        }
    }
    if (carry === 1) {
        digits.unshift(1); // 如果carry最后停留在1, 说明有需要额外的
    }
    return digits;
};
```

复杂度分析

- 时间复杂度: $O(N)$
- 空间复杂度: $O(1)$

相关题目

- [面试题 02.05. 链表求和](#)

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(821. 字符的最短距离)

<https://leetcode-cn.com/problems/shortest-distance-to-a-character>

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。

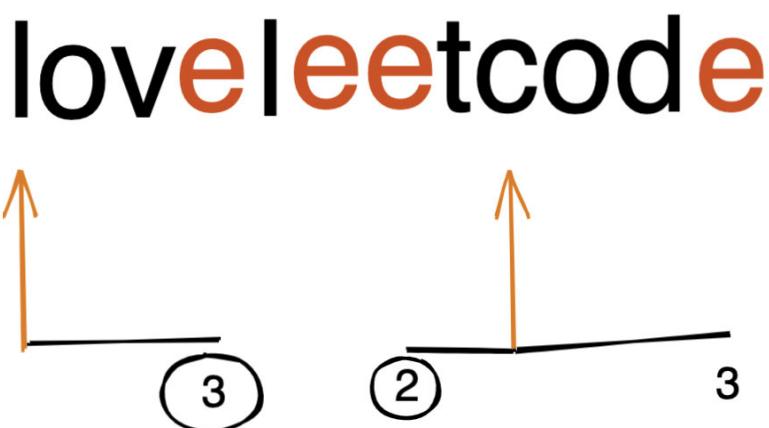
前置知识

- 数组的遍历(正向遍历和反向遍历)

思路

这道题就是让我们求的是向左或者向右距离目标字符最近的距离。

我画了个图方便大家理解：



比如我们要找第一个字符 l 的最近的字符 e , 直观的想法就是向左向右分别搜索, 遇到字符 e 就停止, 比较两侧的距离, 并取较小的即可。如上图, l 就是 3, c 就是 2。

这种直观的思路用代码来表示的话是这样的:

Python Code:

```
class Solution:
    def shortestToChar(self, S: str, C: str) -> List[int]:
        ans = []

        for i in range(len(S)):
            # 从 i 向左向右扩展
            l = r = i
            # 向左找到第一个 C
            while l > -1:
                if S[l] == C: break
                l -= 1
            # 向右找到第一个 C
            while r < len(S):
                if S[r] == C: break
                r += 1
            # 如果至死没有找到, 则赋值一个无限大的数字, 由于题目的数
            if l == -1: l = -10000
            if r == len(S): r = 20000
            # 选较近的即可
            ans.append(min(r - i, i - l))
        return ans
```

复杂度分析

- 时间复杂度: $O(N^2)$
- 空间复杂度: $O(1)$

由于题目的数据范围是 10^4 , 因此通过所有的测试用例是没有问题的。

但是实际上, 我们可以在线性的时间内解决。这里的关键点和上面的解法类似, 也是两端遍历。不过不再是盲目的查找, 因为这样做会有很多不必要的计算。

我们可以使用空间换时间的方式来解, 这里我使用类似单调栈的解法来解, 大家也可以使用其他手段。关于单调栈的技巧, 不在这里展开, 感兴趣的可以期待我后面的专题。

```

class Solution:
    def shortestToChar(self, S: str, C: str) -> List[int]:
        ans = [10000] * len(S)
        stack = []
        for i in range(len(S)):
            while stack and S[i] == C:
                ans[stack.pop()] = i - stack[-1]
            if S[i] != C:stack.append(i)
            else: ans[i] = 0
        for i in range(len(S) - 1, -1, -1):
            while stack and S[i] == C:
                ans[stack.pop()] = min(ans[stack[-1]], stack[-1] - i)
            if S[i] != C:stack.append(i)
            else: ans[i] = 0

        return ans

```

复杂度分析

- 时间复杂度: $O(N)$
- 空间复杂度: $O(N)$

实际上，我们根本不需要栈来存储。原因很简单，那就是每次我们碰到目标字符 C 的时候，我们就把栈全部清空了，因此我们用一个变量标识即可，具体参考后面的代码区。

如果碰到目标字符 C 的时候，不把栈清空，那么这个栈的空间多半是不能省的，反之可以省。

代码

代码支持：Python3, Java, CPP

Python3 Code：

```
class Solution:
    def shortestToChar(self, S: str, C: str) -> List[int]:
        pre = -10000
        ans = []

        for i in range(len(S)):
            if S[i] == C: pre = i
            ans.append(i - pre)
        pre = 20000
        for i in range(len(S) - 1, -1, -1):
            if S[i] == C: pre = i
            ans[i] = min(ans[i], pre - i)
        return ans
```

Java Code:

```
class Solution {
    public int[] shortestToChar(String S, char C) {
        int N = S.length();
        int[] ans = new int[N];
        int prev = -10000;

        for (int i = 0; i < N; ++i) {
            if (S.charAt(i) == C) prev = i;
            ans[i] = i - prev;
        }

        prev = 20000;
        for (int i = N-1; i >= 0; --i) {
            if (S.charAt(i) == C) prev = i;
            ans[i] = Math.min(ans[i], prev - i);
        }

        return ans;
    }
}
```

CPP Code:

```
class Solution {
public:
    vector<int> shortestToChar(string S, char C) {
        vector<int> ans(S.size(), 0);
        int prev = -10000;
        for(int i = 0; i < S.size(); i++){
            if(S[i] == C) prev = i;
            ans[i] = i - prev;
        }
        prev = 20000;
        for(int i = S.size() - 1; i >= 0; i--){
            if(S[i] == C) prev = i;
            ans[i] = min(ans[i], prev - i);
        }
        return ans;
    }
};
```

复杂度分析

- 时间复杂度: $O(N)$
- 空间复杂度: $O(1)$

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。

821.字符的最短距离

- 821.字符的最短距离
 - 题目描述
 - 解法 1：暴力
 - 思路
 - 复杂度分析
 - 代码
 - 解法 2：水波法
 - 思路
 - 复杂度分析
 - 代码
 - 解法 3：“前缀距离”
 - 思路
 - 复杂度分析
 - 代码
 - 解法 4：滑动窗口
 - 思路
 - 复杂度分析
 - 代码

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。

来源：力扣（LeetCode）
链接：<https://leetcode-cn.com/problems/shortest-distance-to-target-character/>
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。

解法 1：暴力

思路

最直接的想法，对于字符串 S 中的每个字符，都定义两个指针 l, r ，从当前下标开始，分别向左、右两个方向去寻找目标字符 C ，找到的第一个 C 与当前下标的距离就是题目要找的最短距离。

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

JavaScript Code

```

/**
 * @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, 取最短距离
        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;
};

```

解法 2：水波法

思路

- 先把字符 `C` 在字符串 `S` 中出现的所有下标记录在一个数组中 `cIndices`。

- 对于字符串 S 中的每个字符，在 $cIndices$ 中找到距离当前位置最近的下标，记录到结果数组中。

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，记录字符 C 出现下标的辅助数组消耗的空间。

代码

JavaScript Code

```
/*
 * @param {string} S
 * @param {character} C
 * @return {number[]}
 */
var shortestToChar = function (S, C) {
    // 统计所有 C 字符在 S 字符串中出现的下标
    var cIndices = [];
    for (let i = 0; i < S.length; i++) {
        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++) {
        if (S[i] === C) {
            res[i] = 0;
            continue;
        }

        for (const cIndex of cIndices) {
            const dist = Math.abs(cIndex - i);

            // 小剪枝一下
            if (dist >= res[i]) break;

            res[i] = dist;
        }
    }
    return res;
};
```

解法 3： “前缀距离”

思路

名字是乱取的，但思路是从前缀和想到的。

- 先从左往右遍历字符串 S，在数组 res 中记录对于每个元素来说，左侧最后一个出现的 C 字符的下标 left；
- 然后从右往左遍历，找到右侧出现的最后一个 C 字符的下标 right，在这一步进行判断，
 - 如果当前元素的左侧没有出现过 C 字符，
 - 或者 $i - left > right - i$ ，
- 则更新数组 res 中的记录为 right。
- 最后遍历一遍 res 数组计算距离绝对值即可。

优化

有一个小优化，在 res 数组中直接记录距离，而不是记录下标，这样能省掉最后一次遍历 res 数组；但算法复杂度是不变的。

复杂度分析

- 时间复杂度： $\$O(N)$ ，N 是 S 的长度，进行了 3 次遍历，时间消耗为 $\$3N$ ，常数忽略。
- 空间复杂度： $\$O(1)$ 。

代码

JavaScript Code

```
/**  
 * @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 字符的最右下标  
    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]  
    }  
  
    // 第二次遍历：从右往左  
    // 找出现在右侧的 C 字符的最左下标  
    // 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])  
    }  
  
    // 计算距离  
    for (let i = 0; i < res.length; i++) {  
        res[i] = Math.abs(res[i] - i);  
    }  
    return res;  
};
```

优化

```
/**  
 * @param {string} S  
 * @param {character} C  
 * @return {number[]} length  
 */  
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;  
        // 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];  
    }  
  
    for (let i = S.length - 1; i >= 0; i--) {  
        // 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，就用  
        // C 的索引减去 i，得到的就是最近距离  
        if (res[i] === Infinity || res[i + 1] + 1 < res[i]) res[i] = res[i + 1] + 1;  
    }  
  
    return res;  
};
```

解法 4：滑动窗口

思路



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

JavaScript Code

```
/*
 * @param {string} S
 * @param {character} C
 * @return {number[]}
 */
var shortestToChar = function (S, C) {
    let l = S[0] === C ? 0 : Infinity,
        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));

        // 遍历完了当前窗口后，将窗口右移
        if (i === r) {
            l = r;
            r = S.indexOf(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/plus-one>

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","push","push","push","inc","pop","inc","pop"]  
[[[3],[1],[2],[],[2],[3],[4],[5,100],[2,100],[],[],[],[]]]
```

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

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： $O(1)$

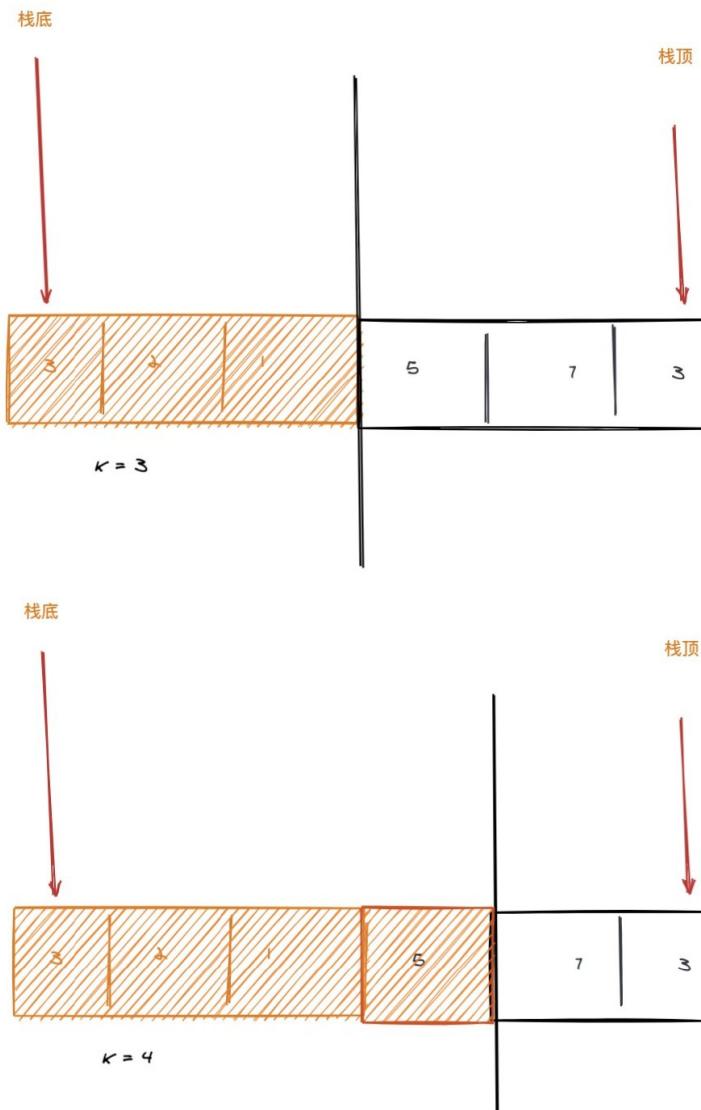
前缀和

前缀和在讲义里面提到过，大家也可是看下我的文章 [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：

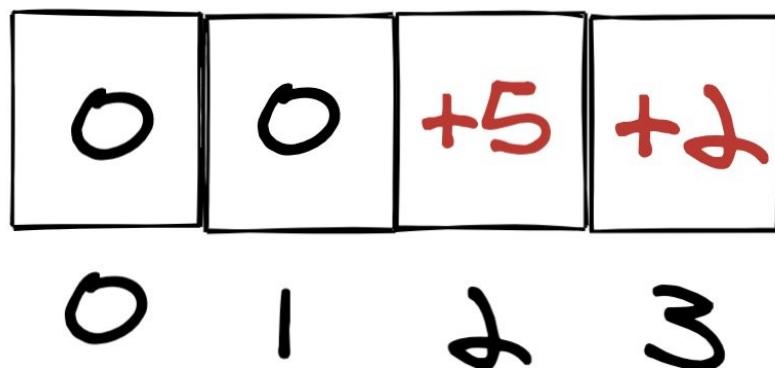
- 初始化一个大小为 `maxSize` 的数组 `incrementals`，并全部填充 0
- `push` 操作不变，和上面一样
- `increment` 的时候，我们将用到 `incremental` 信息。那么这个信息是什么，从哪来呢？我这里画了一个图



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

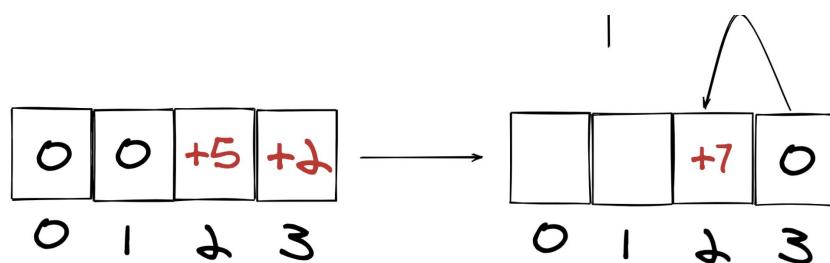
比如：

- 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

```

class CustomStack:

    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间复杂度为 $O(\maxSize / N)$ ，其中 N 为操作数。

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 $O(\maxSize)$ 的数组 `incremental`。而由于栈只能在栈顶进行操作，因此这实际上可以稍微优化一点，即维护一个大小为当前栈长度的 `incrementals`，而不是 $O(\maxSize)$ 。

每次栈 `push` 的时候，`incrementals` 也 `push` 一个 0。每次栈 `pop` 的时候，`incrementals` 也 `pop`，这样就可以了。

这里的 `incrementals` 并不是一个栈，而是一个普通数组，因此可以随机访问。

代码

```
class CustomStack:

    def __init__(self, size: int):
        self.st = []
        self.cnt = 0
        self.size = size
        self.incrementals = []

    def push(self, x: int) -> None:
        if self.cnt < self.size:
            self.st.append(x)
            self.incrementals.append(0)
            self.cnt += 1

    def pop(self) -> int:
        if self.cnt == 0: return -1
        self.cnt -= 1
        if self.cnt >= 1:
            self.incrementals[-2] += self.incrementals[-1]
        return self.st.pop() + self.incrementals.pop()

    def increment(self, k: int, val: int) -> None:
        if self.incrementals:
            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 `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/>

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：

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：

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"

前置知识

- 栈
- 括号匹配

使用栈

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。

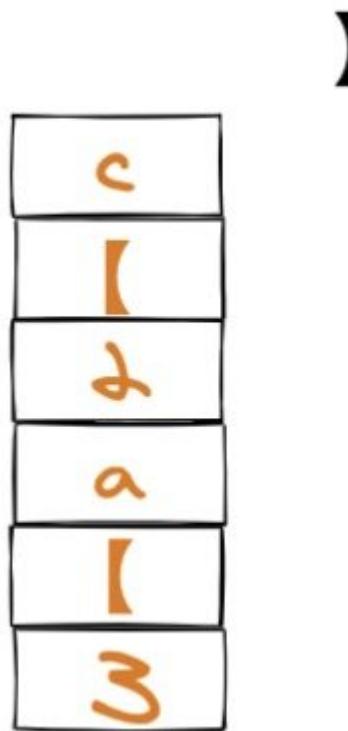
那么现在看一下迭代的解法。

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：

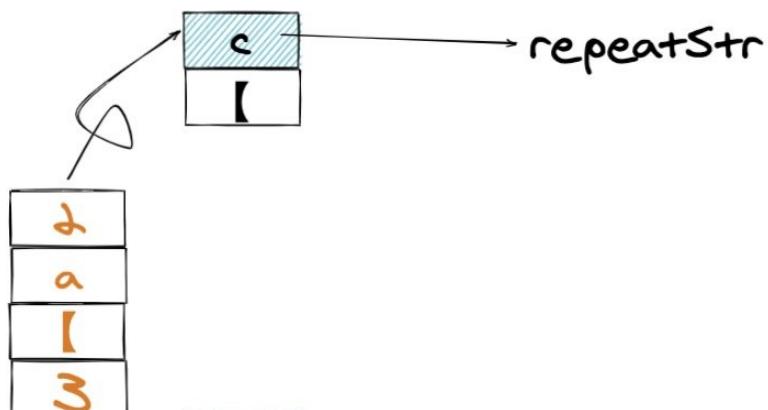
- 如果是字母 --> 添加到 stack 当中
- 如果是数字 --> 先不着急添加到 stack 中 --> 因为有可能有多位
- 如果是 [--> 说明重复字符串开始 --> 将数字入栈 --> 并且将数字清零
- 如果是] --> 说明重复字符串结束 --> 将重复字符串重复前一步储存的数字遍

拿题目给的例子 `s = "3[a2[c]]"` 来说：

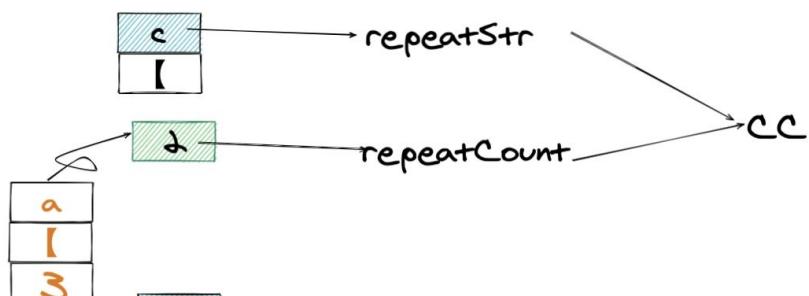
在遇到 `]` 之前，我们不断执行压栈操作：



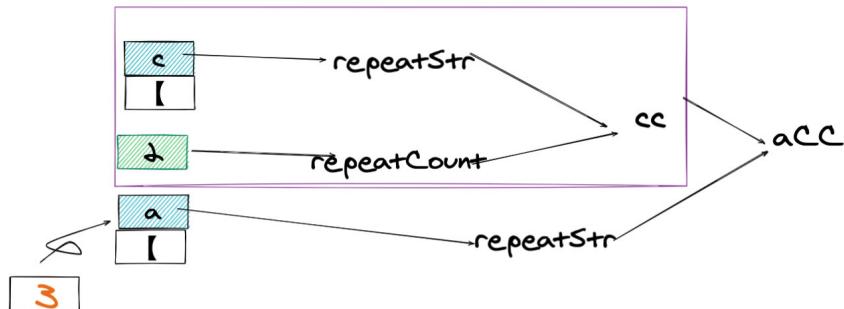
当遇到 **】** 的时候，说明我们应该出栈了，不断出栈知道对应的 **【**，这中间的就是 repeatStr。



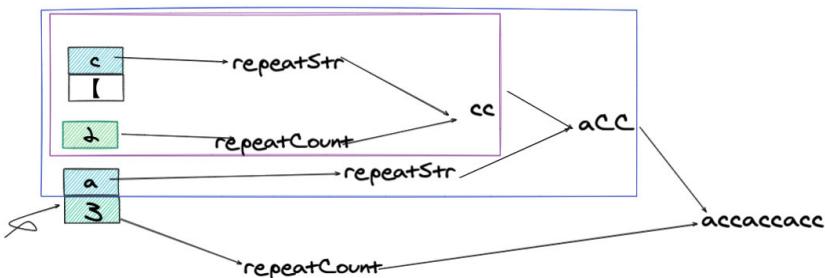
但是要重复几次呢？我们需要继续出栈，直到非数字为止，这个数字我们记录为 repeatCount。



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：

```
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
                # 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。

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，以及递归我们将在第三节讲述。

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，碰到右括号就跳出递归返回即可。

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，因此如果使用递归需要在递归返回后更新指针。

代码

```
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
                    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 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/>

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`)

前置知识

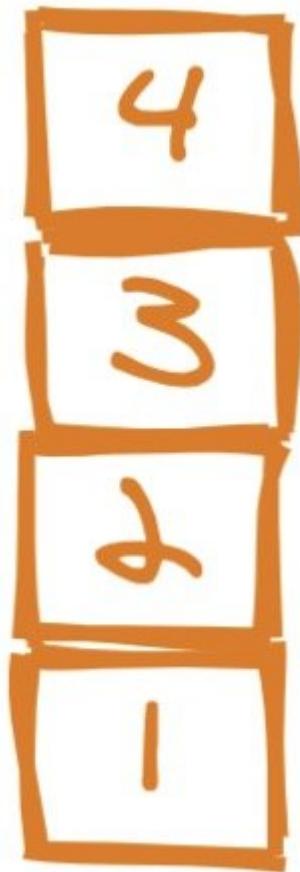
- 栈
- 队列

思路

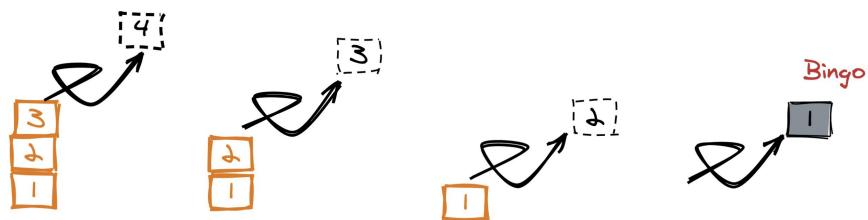
题目要求用栈的原生操作来实现队列，也就是说需要用到 `pop` 和 `push` 但是我们知道 `pop` 和 `push` 都是在栈顶的操作，而队列的 `enqueue` 和 `dequeue` 则是在队列的两端的操作，这么一看一个 `stack` 好像不太能完成。

我们来分析一下过程。

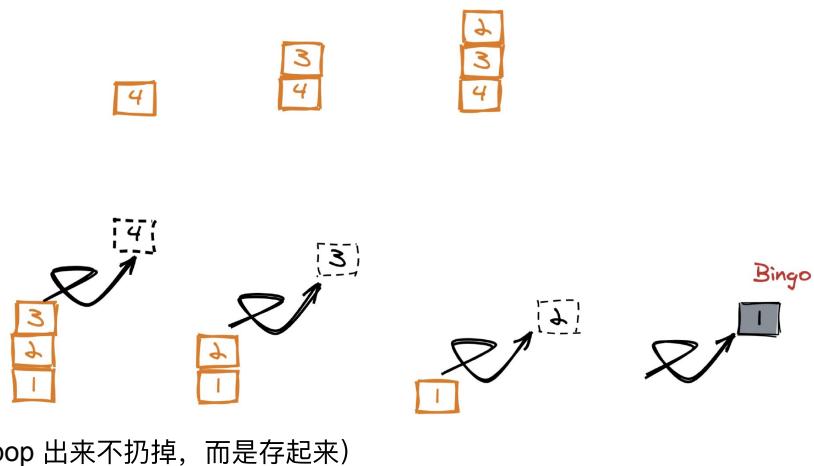
假如向栈中分别 `push` 四个数字 `1, 2, 3, 4`，那么此时栈的情况应该是：



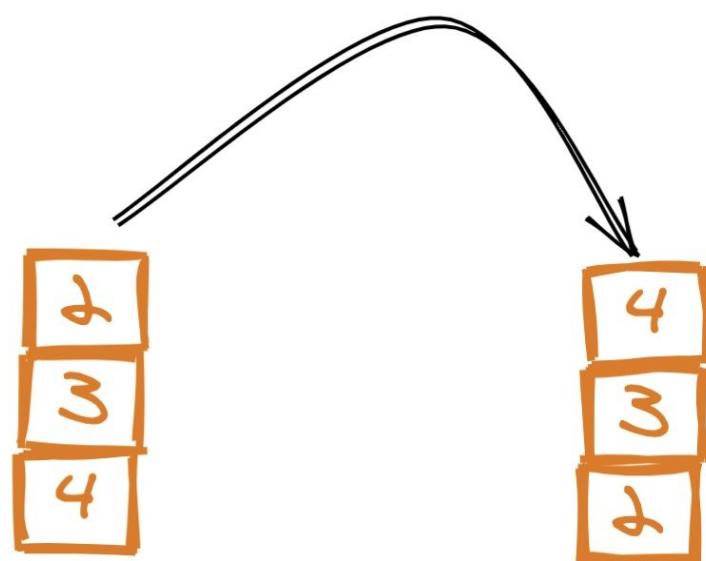
如果此时按照题目要求 pop 或者 peek 的话，应该是返回 1 才对，而 1 在栈底我们无法直接操作。如果想要返回 1，我们首先要将 2, 3, 4 分别出栈才行。



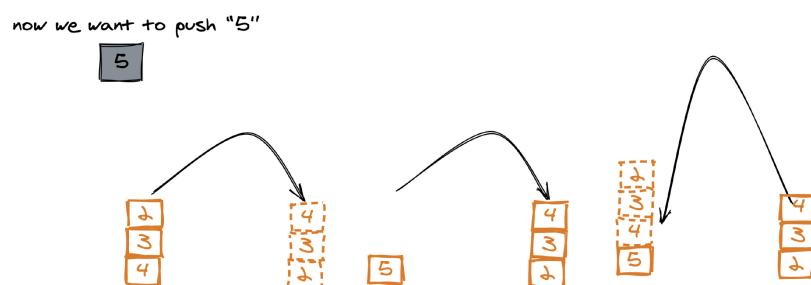
然而，如果我们这么做，1 虽然是正常返回了，但是 2, 3, 4 不就永远消失了么？一种简答方法就是，将 2, 3, 4 存起来。而题目又说了，只能使用栈这种数据结构，那么我们考虑使用一个额外的栈来存放弹出的 2, 3, 4。



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:

```

/*
 * @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 +.
 * @return {number}
 */
MyQueue.prototype.pop = function () {
    return this.stack.pop();
};

/** 
 * Get the front element.
 * @return {number}
 */
MyQueue.prototype.peek = function () {
    return this.stack[this.stack.length - 1];
};

/** 

```

```
* Returns whether the queue is empty.  
* @return {boolean}  
*/  
MyQueue.prototype.empty = function () {  
    return this.stack.length === 0;  
};  
  
/**  
 * Your MyQueue object will be instantiated and called as follows:  
 * var obj = new MyQueue()  
 * obj.push(x)  
 * var param_2 = obj.pop()  
 * var param_3 = obj.peek()  
 * var param_4 = obj.empty()  
*/
```

Python Code:

```

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 that element.
        """
        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

```

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



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/>

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。

前置知识

- 栈
- 队列

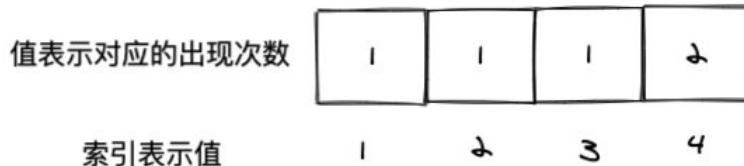
计数

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，即不管你怎样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。

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

```
class Solution(object):
    def maxChunksToSorted(self, arr):
        count_a = collections.defaultdict(int)
        count_b = collections.defaultdict(int)
        ans = 0

        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 $O(N)$ ，因此总的时间复杂度为 $O(N^2)$ ，其中 N 为数组长度。
- 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 $O(N)$ ，其中 N 为数组长度。

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 的 diff 即可。但是这也仅仅是空间上的一个常数优化而已。

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，这样算法的瓶颈就是排序了。而去除的关键点就是我们上面提到的记录 diff，具体参考下方代码。

关键点

- 计数
- count 的边界条件

代码

语言支持：Python

```
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效。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

```

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:

```

class Solution {
public:
    int maxChunksToSorted(vector<int>& arr) {
        stack<int> stack;
        for(int i = 0;i<arr.size();i++){
            // 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.empty()&&stack.top()>arr[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]){
                    stack.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:

```

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:

```

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。

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。

<https://github.com/suukii/91-days-algorithm/blob/master/basic/array-stack-queue/06.max-chunks-to-make-sorted-ii.md>

768. 最多能完成排序的块 II

<https://leetcode-cn.com/problems/max-chunks-to-make-sorted-ii/>

- 768. 最多能完成排序的块 II
 - 题目描述
 - 方法 1: 滑动窗口
 - 思路
 - 复杂度分析
 - 代码
 - 方法 2: 单调递增栈
 - 思路
 - 图解
 - 复杂度分析
 - 代码

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。

来源：力扣（LeetCode）

链接：<https://leetcode-cn.com/problems/max-chunks-to-make-sorted/>

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。

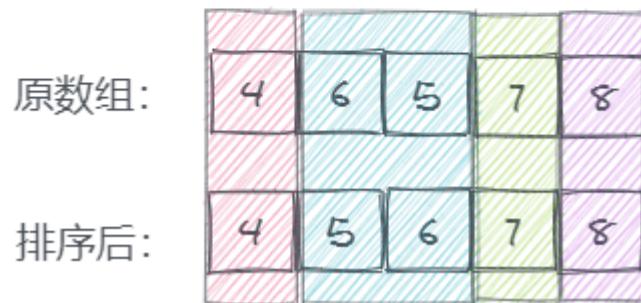
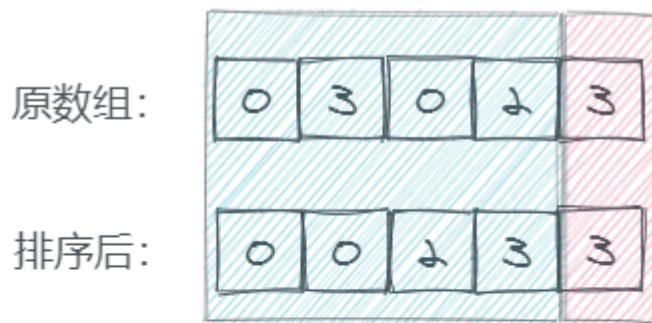
方法 1：滑动窗口

思路

题目有一个提示：

Each k for which some permutation of arr[:k] is equal to sorted(arr[:k])

也就是原数组进行分块后，每一个分块和排序后的数组中对应的分块数字
是一样的，只是排序不同。



既然每个分块中数字是一样的，那它们的和也是一样的了。我们可以用一个滑动窗口同时扫描原数组和排序数组，当窗口中数字的和一样时，就将数组进行分块，就像上图中的色块一样。

复杂度分析

- 时间复杂度: $O(N \log N)$, N 为数组长度，数组排序时间认为是 $N \log N$ ，滑动窗口遍历数组时间为 N 。
- 空间复杂度: $O(N)$, N 为数组长度。

代码

JavaScript Code

```

/**
 * @param {number[]} arr
 * @return {number}
 */
var maxChunksToSorted = function (arr) {
    const sorted = [...arr];
    sorted.sort((a, b) => a - b);

    let count = 0,
        sum1 = 0,
        sum2 = 0;

    for (let i = 0; i < arr.length; i++) {
        sum1 += arr[i];
        sum2 += sorted[i];

        if (sum1 === sum2) {
            count++;
            sum1 = sum2 = 0; // 这行不要也可以啦
        }
    }

    return count;
};

```

方法 2: 单调递增栈

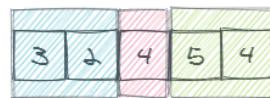
思路

根据题意，将原数组进行分块后，对各分块分别进行排序后的结果等于原数组排序后的结果。

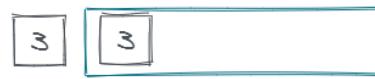
可以得到的一个结论是，每个分块中的数字相对于前一个分块都是递增的（因为有重复数字，所以也可能是相同），下一个分块中的所有数字都会大于等于上一个分块中的所有数字。

- 因为题目要求能分的最多的块数，所以我们在分块的时候要尽量把块分小，这样就能分得比较多。
- 在遍历数组的过程中，如果一个数字比之前所有分块的最大值都要大，我们就把它作为一个新的分块。
- 如果数字小于之前某些分块的最大值，那这些分块都要被合成一个分块(保持栈的单调递增)。

图解



每个分块中的最大值:



这时栈是空的，前面还没有分块，
所以 3 自己成一个分块



前一个分块的最大值是 3, $2 < 3$,
所以把 2 分到前一个分块里，
因为栈中只存分块的最大值，2 不入栈



前一个分块的最大值是 3, $4 > 3$,
可以产生一个新的分块，4 入栈



同理 5 入栈



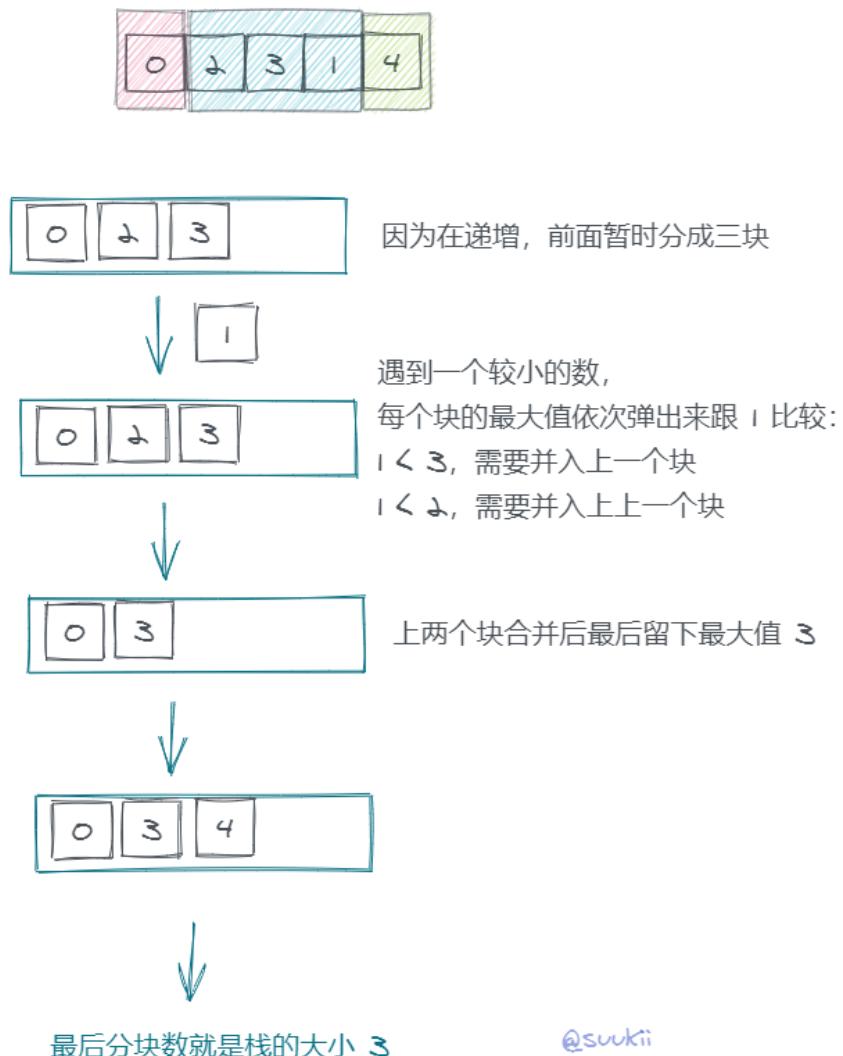
$4 < 5$ 合入上一个分块
并且 $4 == 4$, 所以不用合并上上个分块



最后分块数就是栈的大小 3

@suukii

再看一个例子:



复杂度分析

- 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, 单调栈消耗的空间。

代码

JavaScript Code

```
class Stack {
    constructor() {
        this.list = [];
    }
    push(val) {
        this.list.push(val);
    }
    pop() {
        return this.list.pop();
    }
    empty() {
        return this.list.length === 0;
    }
    peek() {
        return this.list[this.list.length - 1];
    }
    size() {
        return this.list.length;
    }
}

/**
 * @param {number[]} arr
 * @return {number}
 */
var maxChunksToSorted = function (arr) {
    const stack = new Stack();

    for (let i = 0; i < arr.length; i++) {
        if (stack.empty() || stack.peek() <= arr[i]) {
            stack.push(arr[i]);
        } else {
            const temp = stack.pop();

            while (stack.peek() > arr[i]) {
                stack.pop();
            }

            stack.push(temp);
        }
    }
    return stack.size();
};
```

题目地址(24. 两两交换链表中的节点)

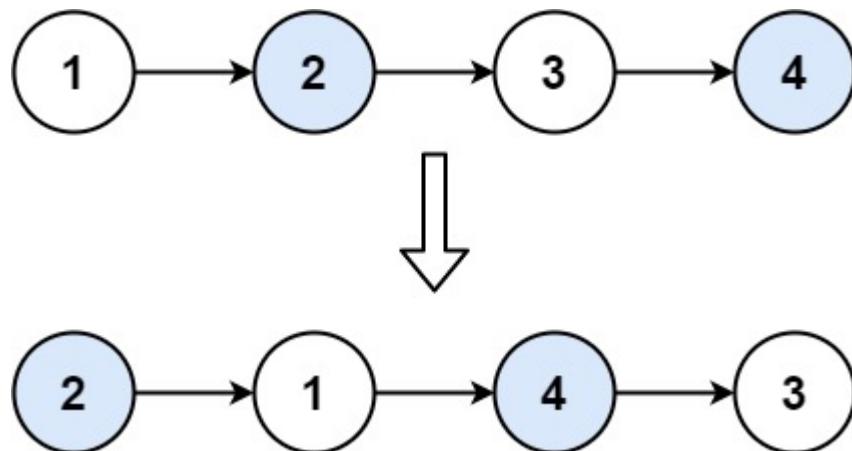
<https://leetcode-cn.com/problems/swap-nodes-in-pairs/>

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

$0 \leq \text{Node.val} \leq 100$

逆转单链表的思路

这道题其实考察的内容就是链表节点的指针的指向。

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

修改指针的顺序为：

1. A节点的next指向nextB:

```
preA -> A -> nextB  
B -> nextB
```

2. B节点的next指向A

```
preA -> A -> nextB  
B -> A
```

3. 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
```

迭代的解法

前置知识

- 链表的基本知识

思路

例如当前链表为： A -> B -> C -> D

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

```
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，继续上诉步骤逆转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
```

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 $O(1)$

代码

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
};
```

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
# 递归出口
while head and head.next:
    next = head.next
    n_next = next.next

    next.next = head
    pre.next = next
    head.next = n_next
    # 更新入参
    pre = head
    head = n_next
return ans.next
```

递归的解法

前置知识

- 链表的基本知识
- 递归

思路

1. 将两个节点进行逆转
2. 将逆转后的尾节点.next指向下次递归的返回值
3. 返回逆转后的链表头节点 (ps:逆转前的第二个节点)

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 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
```

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)\$

24. 两两交换链表中的节点

<https://leetcode-cn.com/problems/swap-nodes-in-pairs/>

- 24. 两两交换链表中的节点
 - 题目描述
 - 方法 1: 循环
 - 图解
 - 复杂度分析
 - 代码
 - 方法 2: 递归
 - 思路
 - 复杂度分析
 - 代码

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

$0 \leq \text{Node.val} \leq 100$

来源：力扣（LeetCode）

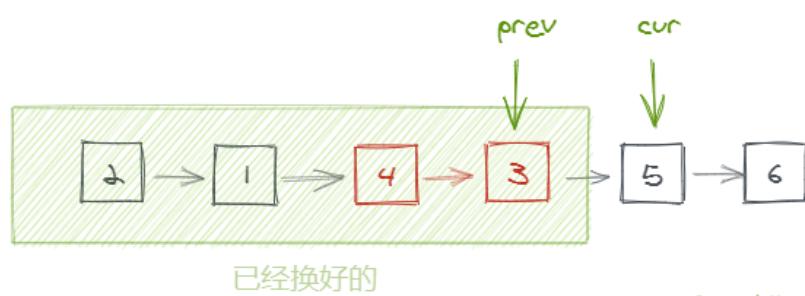
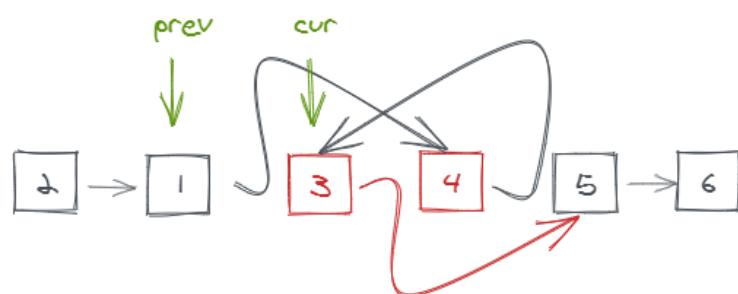
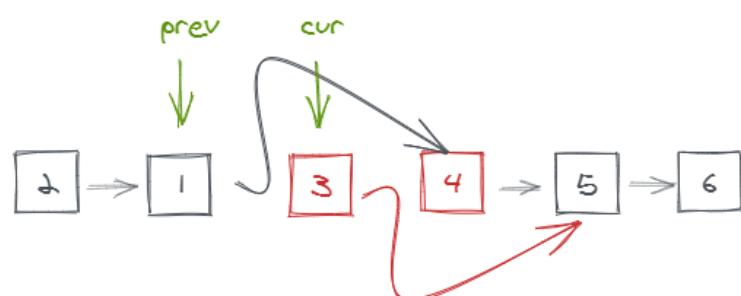
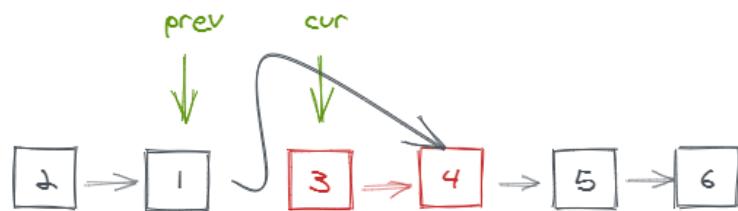
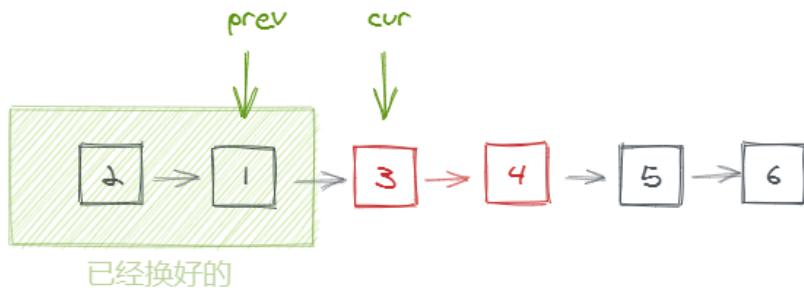
链接：<https://leetcode-cn.com/problems/swap-nodes-in-pairs>

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。

方法 1：循环

图解

- 要注意指针改变的顺序，不然可能会丢失节点。
- 用一个 dummy 节点来简化操作，不用额外考虑链表头部的情况。



@suukii

复杂度分析

- 时间复杂度: $O(N)$, N 为链表长度。
- 空间复杂度: $O(1)$ 。

代码

JavaScript Code

```
/*
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */
var swapPairs = function (head) {
    const dummy = new ListNode(null, head);
    let prev = dummy;
    let cur = prev.next;

    while (cur && cur.next) {
        // 按照上图, 指针更换顺序是这样子的
        // prev.next = cur.next
        // cur.next = prev.next.next
        // prev.next.next = cur

        // 也可以先用一个指针把下一个节点存起来
        const next = cur.next;
        cur.next = next.next;
        next.next = cur;
        prev.next = next;

        prev = cur;
        cur = cur.next;
    }
    return dummy.next;
};
```

方法 2: 递归

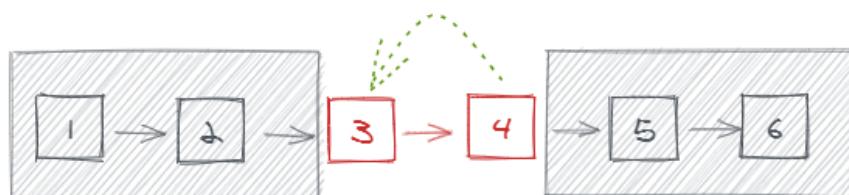
思路

“将相邻的链表节点两两交换”，我们可以把链表两两分成若干组，在组内互换节点后再组合起来。

先解决小问题，再把小问题的解组合起来，解决大问题。

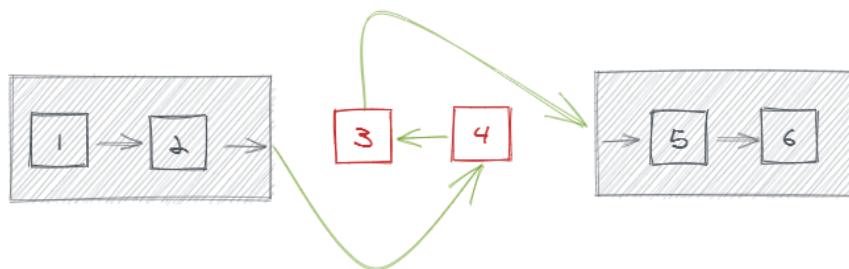
小问题

只要关注当前递归要解决的问题就行，前后都可以当成是黑匣子。在当前递归中，我们只需要将两个节点互换。



小问题之间的关系

- 下一个递归应该返回互换后的第一个节点
- 当前递归应该返回互换后的第一个节点给上一个递归



递归出口

- 链表尾部 `head === null`
- 链表最后只剩下一个元素 `head.next === null`

复杂度分析

- 时间复杂度：\$O(N)\$, N 为链表长度。
- 空间复杂度：\$O(N)\$, N 为链表长度，递归栈的空间。

代码

JavaScript Code

```
/**  
 * Definition for singly-linked list.  
 * function ListNode(val, next) {  
 *     this.val = (val===undefined ? 0 : val)  
 *     this.next = (next===undefined ? null : next)  
 * }  
 */  
/**  
 * @param {ListNode} head  
 * @return {ListNode}  
 */  
  
var swapPairs = function (head) {  
    // 递归出口  
    if (!head || !head.next) return head;  
  
    // 先保存下一个节点，避免丢失  
    const next = head.next;  
  
    // 下一个递归会返回互换后的第一个节点  
    // head 是当前组互换后的第二个节点，head.next 指向下一组就好  
    head.next = swapPairs(next.next);  
  
    // 将当前组的两个节点互换  
    next.next = head;  
  
    // 返回互换后的第一个节点  
    return next;  
};
```

题目地址(61. 旋转链表)

<https://leetcode-cn.com/problems/rotate-list/>

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： $2 \rightarrow 0 \rightarrow 1 \rightarrow \text{NULL}$

快慢指针法

前置知识-求单链表的倒数第N个节点

思路

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 N 步
4. 当快指针到达终点时，慢指针正好是倒数第 N 个节点

伪代码

```
快指针 = head  
慢指针 = head  
while(快指针.next){  
    if(N-- <= 0){  
        慢指针 = 慢指针.next  
    }  
    快指针 = 快指针.next  
}
```

js Code:

```
let slow = fast = head  
while(fast.next){  
    if(k-- <= 0){  
        slow = slow.next  
    }  
    fast = fast.next  
}
```

思路

1. 获取单链表的倒数第 $N + 1$ 与倒数第 N 个节点
2. 将倒数第 $N + 1$ 个节点的next指向null
3. 将链表尾节点的next指向head
4. 返回倒数第 N 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

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
```

js Code:

```
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:

```
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;
    }
}
```

py3 Code:

```
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
```

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(109. 有序链表转换二叉搜索树)

<https://leetcode-cn.com/problems/convert-sorted-list-to-binary-search-tree/>

题目描述

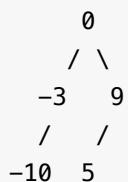
给定一个单链表，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。

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



前置知识

- 递归
- 二叉搜索树

对于树中任意一个点，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

思路

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

双指针法

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

```

var sortedListToBST = function(head) {
    if(!head) return null;
    return run(head, null);
};

function run(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 = run(head, slow);
    root.right = run(slow.next, tail);
    return root;
}

```

java Code:

```

class Solution {
    public TreeNode sortedListToBST(ListNode head) {
        if(head == null) return null;
        return run(head,null);
    }
    private TreeNode run(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 = run(head, slow);
        root.right = run(slow.next, tail);
        return root;
    }
}

```

复杂度分析

- 时间复杂度：节点最多只遍历 $N \log N$ 遍，时间复杂度为 $O(N \log N)$

- 空间复杂度：空间复杂度为 $O(1)$

缓存法

因为链表访问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

```
var sortedListToBST = function(head) {
    let res = []
    while(head){
        res.push(head.val)
        head = head.next
    }
    return run(res)
};

function run(res){
    if(res.length == 0) return null
    let mid = parseInt(res.length / 2)
    let root = new TreeNode(res[mid])
    root.left = mid > 0 ? run(res.slice(0, mid)) : null
    root.right = mid >= res.length - 1 ? null : run(res.slice(mid + 1))
    return root
}
```

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：若使用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/>

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

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:

```
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 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;

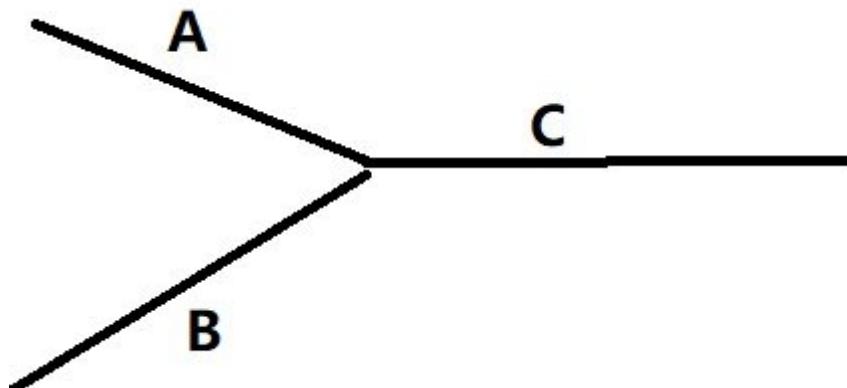
```

复杂度分析

- 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, 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:

- 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: $B + C$
- 当 a 指针将链表 1 遍历完后, 重定位到链表 B 的头结点, 然后继续遍历直至相交点(a 指针遍历的距离为 $A + C + B$)
- 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
```

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:

```
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

- 时间复杂度: $O(N)$
- 空间复杂度: $O(1)$

思路：方法1：字典

- 用字典存下A链表，然后遍历链表B寻找有没有在字典中的即可，
- 这里占用了额外的字典空间去存放A链表

方法2：双指针，

- 定义两个指针A, B，分别对应链表A和链表B，将A和B指针依次向后移动
- 如果skipA == skipB，则A和B在skipA次后相遇返回，
- 如果skipA != skipB，当B走到尽头回到headA，当A走到尽头回到headB，然后最终会相遇，因为 $\text{len}(A + C + B) == \text{len}(B + C + A)$

代码：方法1：

```
class Solution(object):
    def getIntersectionNode(self, headA, headB):
        if not headA or not headB:
            return None
        dic = {}
        A = headA
        B = headB
        while A:
            dic[A] = 1
            A = A.next

        while B:
            if B in dic:
                return B
            else:
                B = B.next

        return None
```

方法2：

```
class Solution(object):
    def getIntersectionNode(self, headA, headB):
        if not headA or not headB:
            return None
        A = headA
        B = headB
        while A != B:
            if A:
                A = A.next
            else:
                A = headB
            if B:
                B = B.next
            else:
                B = headA
        return A
```

复杂度分析： 如果A链表的长度是m, B链表的长度是n 方法一：时间：
 $O(m + n)$, 空间： $O(m)$ (字典存储A链表消耗的空间) 方法二：时间：
 $O(m + n)$, 空间： $O(1)$

题目地址(142. 环形链表 II)

<https://leetcode-cn.com/problems/linked-list-cycle-ii/>

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,
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

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;

```

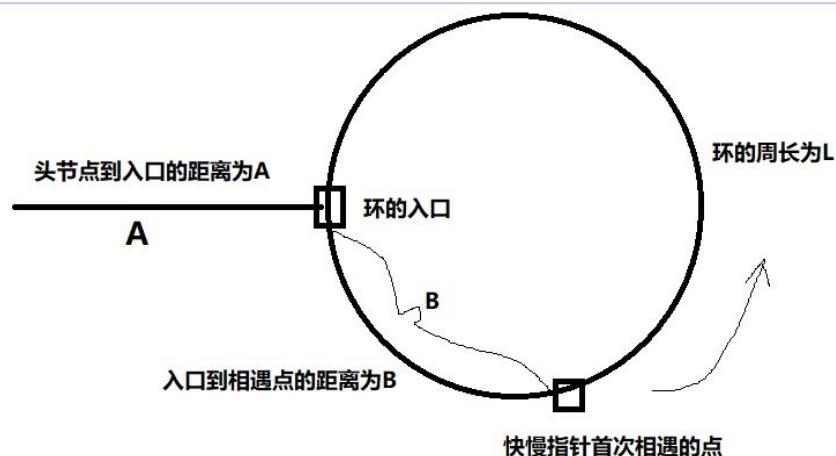
复杂度分析

- 时间复杂度: $O(N)$
- 空间复杂度: $O(N)$

快慢指针法

思路

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 指针指向链表头部,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



(图 6)

为什么第二次相遇的点为环的入口? 原因如下:

- 第一次相遇时
- 慢指针移动的距离为 $s_1 = A + B + n_1 * L$
- 快指针移动的距离为 $s_2 = A + B + n_2 * L$

- 快指针是慢指针速度的两倍,所以 $s2 = 2 * s1$
- $A + B + n2 L = 2A + 2B + n1 L \implies A = -B + (n2 - n1) * L$
- 因为圆的性质 $(n2 - n1) * L \implies$ 绕圆 $(n2 - n1)$ 圈 $\implies 0$
- $A = -B + (n2 - n1) * L \implies A = -B$
- 即在第一次相遇点, 向前走 A 步 \implies 向后走 B 步
- **第一次相遇后**
- 快指针从头节点走 A 步会到达环的入口
- 慢指针从第一次相遇点走 A 步,相当于向后走 B 步,也会到达环的入口

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

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;
```

复杂度分析

- 时间复杂度: $O(N)$
- 空间复杂度: $O(1)$

思路: 利用两个指针slow、fast, 快(fast)指针走两步, 慢(slow)指针走一步。

设起点到环的入口距离为a, 环的入口到相遇点距离为b, 环的另一半距离为c。相遇时慢指针走的距离为 $a + (b + c)m + b$, 快指针走的距离为两倍 $2(a + (b + c)m + b) = a + (b + c)*n + b$, 即head指针到入口点的距离等于slow剩余到入口处的距离 + k cycle(绕k圈), 意味着两个指针从head处和fast/slow相遇处同时出发, 一定会在cycle的入口处相遇。

结论: 当fast, slow相遇后, 让fast指向head, slow继续指向相遇处, 每次两个指针都各走一步, 等fast和slow再次相遇, 即为环的入口。

复杂度分析:

时间复杂度: $O(n * k) = O(n)$, n为结点数, k为可能的圈数

空间复杂度: $O(1)$

执行结果: 通过

- 执行用时: 4 ms, 在所有 C++ 提交中击败了99.05%的用户
- 内存消耗: 8 MB, 在所有 C++ 提交中击败了17.46%的用户

代码(C++):

```


/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        if (!head || !head->next) return nullptr;

        /* use two pointers:
           slow pointer -> one node each time
           fast pointer -> two nodes each time

           when meet means a cycle found
        */
        ListNode* fast = head;
        ListNode* slow = head;

        while (fast && fast->next) {
            fast = fast->next->next;
            slow = slow->next;
            // has a cycle
            if (fast == slow) {
                fast = head;
                // start from head, when meet again, the me
                while (fast != slow) {
                    fast = fast->next;
                    slow = slow->next;
                }
            }

            return fast;
        }
        return nullptr;
    }
};


```

题目地址(146. LRU缓存机制)

<https://leetcode-cn.com/problems/lru-cache/>

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], [-1]]
```

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
```

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
 - ii. 同时需要对数据进行频繁的增删, 时间复杂度 $O(1)$ ==> 使用链表等
 - iii. 对数据进行读取时, 时间复杂度 $O(1)$ ==> 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:

```
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:

```

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

146. LRU 缓存机制

<https://leetcode-cn.com/problems/lru-cache/>

- 146. LRU 缓存机制
 - 题目描述
 - 思路
 - 复杂度分析
 - 伪代码
 - 代码

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。

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

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：

```
LRUCache cache = new LRUCache( 2 /* 缓存容量 */ );  
  
cache.put(1, 1);  
cache.put(2, 2);  
cache.get(1);      // 返回 1  
cache.put(3, 3);      // 该操作会使得关键字 2 作废  
cache.get(2);      // 返回 -1 (未找到)  
cache.put(4, 4);      // 该操作会使得关键字 1 作废  
cache.get(1);      // 返回 -1 (未找到)  
cache.get(3);      // 返回 3  
cache.get(4);      // 返回 4
```

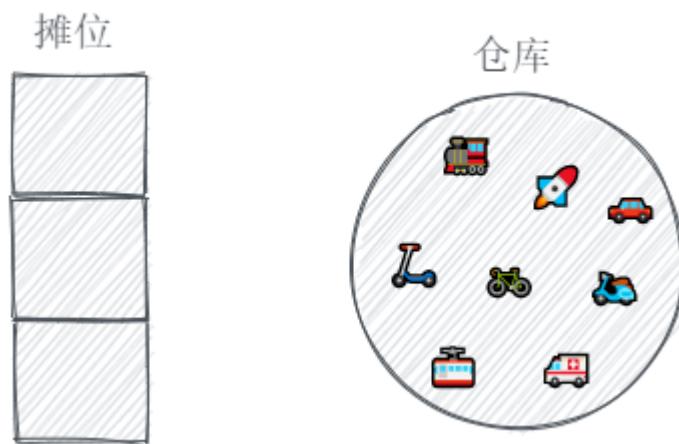
来源：力扣（LeetCode）

链接：<https://leetcode-cn.com/problems/lru-cache/>

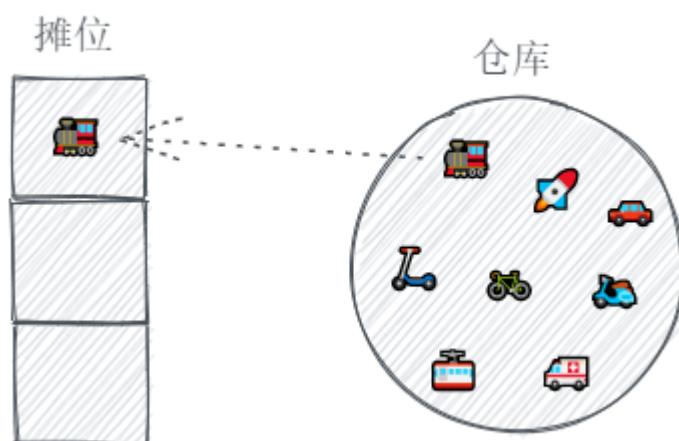
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

思路

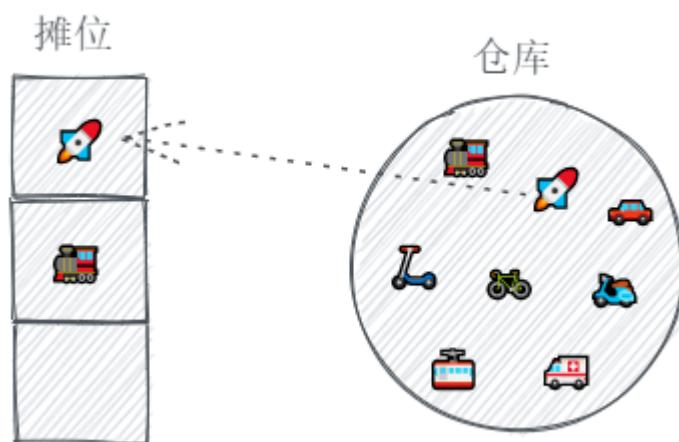
假设我们有一个玩具摊位，可以向顾客展示小玩具，但是摊位大小有限，我们不能把所有的玩具都摆在摊位上，所以我们就把大部分的玩具都放在了仓库里。



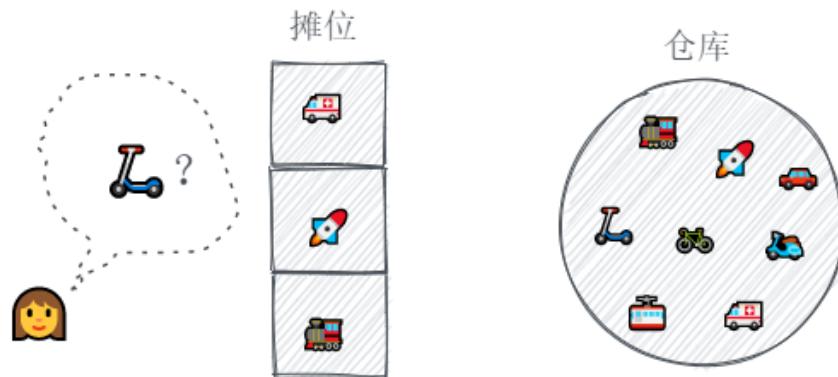
如果有顾客来问，我们就去仓库把那个玩具拿出来，摆在摊位上。



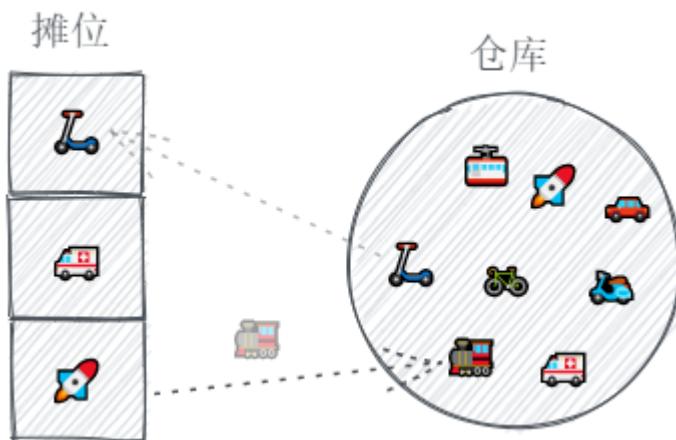
因为最上面的那个位置最显眼，所以我们想总是把最新拿出来的玩具放在那。



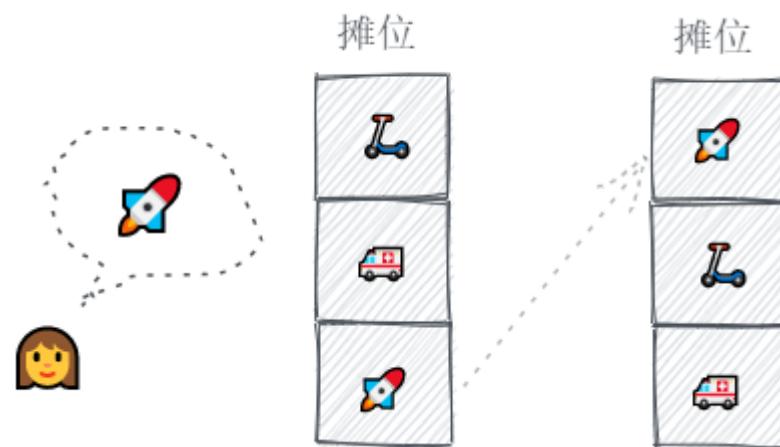
但是摊位大小有限，很快就摆满了，如果这时又来了顾客想看新玩具。



我们只能把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)，腾出一个位置来放新玩具。



如果顾客想看的玩具就在摊位上，我们就可以直接展示这个玩具，同时把它放到最上面的位置(有人问说明它受欢迎嘛)，其他的玩具就要挪挪位置了。



回到计算机问题上面来，我们要用什么来表示我们的玩具摊位呢，如果用数组，玩具在摊位上的位置挪来挪去的，时间复杂度得是 $O(N)$ ，所以只能用链表了。

用链表的话，随意移除一个节点的时间复杂度是 $O(1)$ ，移除节点后，我们还得把它前后两个节点连起来，所以用双向链表会比较方便。

但是链表获取节点的时间复杂度是 $O(N)$ ，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，但是计算机没有那么聪明，所以我们还需要一个数据结构(哈希表)来帮计算机记录什么玩具在什么位置上。

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 $O(N)$ 。

伪代码

```
// 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
    在哈希表中增加映射
    把节点加到链表头部


// 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 Code

```

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 with edge cases.
        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;
        }
    }
}

```

```
        }

    }

    put(key, value) {
        if (key in this.hashmap) {
            // If key exists, update the corresponding node.
            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
     * var obj = new LRUCache(capacity)
     * var 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>

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

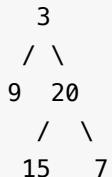
给定一个二叉树，找出其最大深度。

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]，



返回它的最大深度 3。

前置知识

- 递归

思路

树的题目很适合用来递归来做。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。因此这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。

104.二叉树的最大深度

<https://leetcode-cn.com/problems/maximum-depth-of-binary-tree/>

- 104.二叉树的最大深度
 - 题目描述
 - 方法 1：递归
 - 思路
 - 复杂度分析
 - 代码
 - 方法 2：循环+队列
 - 思路
 - 复杂度分析
 - 伪代码
 - 代码

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 。

来源：力扣（LeetCode）

链接：<https://leetcode-cn.com/problems/maximum-depth-of-binary-tree/>

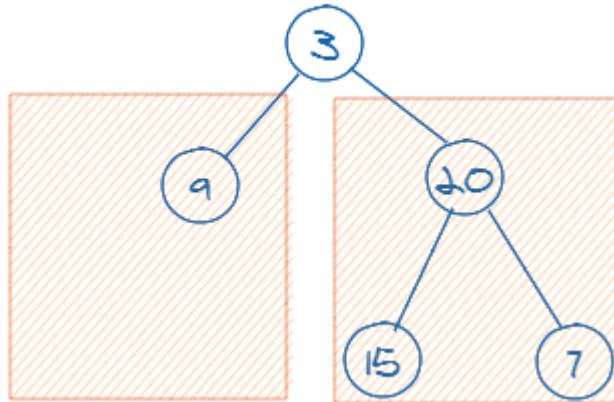
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

方法 1：递归

思路

递归地分别计算出左子树和右子树的高度，取最大值加一，就是当前二叉树的最大深度。

$$\max(\text{left}, \text{right}) + 1 = 3$$



假设我们已经计算出了左子树的最大深度是 1，右子树的最大深度是 2，那整棵二叉树的最大深度就是左右子树最大深度的最大值加上根节点，也就是 3。

接下来我们只需要递归地去计算左右子树的高度：

1. base condition: 首先在递归中很重要的事情就是找到递归的出口，在这道题目中，明显能想到的出口是：
 - i. 当遍历到叶子节点的时候： `if not root.left and not root.right`，这时候我们需要 `return 1`。
 - ii. 还有一种情况是，当前遍历的节点为 `null`，这时候我们只需要 `return 0` 就好了。
2. 如果当前节点有左子树，需要递归计算左子树的高度 `leftHeight = maxDepth(root.left)`。
3. 如果当前节点有右子树，需要递归计算右子树的高度 `rightHeight = maxDepth(root.right)`。
4. 取左右子树最大高度的最大值，加上当前节点返回 `max(leftHeight, rightHeight) + 1` 即可。

复杂度分析

- 时间复杂度：\$O(N)\$，N 为二叉树的节点数，因为每个节点都只遍历一次。
- 空间复杂度：\$O(N)\$，递归栈的空间。

代码

JavaScript Code

```

/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
var maxDepth = function (root) {
    if (!root) return 0;
    if (!root.left && !root.right) return 1;
    return Math.max(maxDepth(root.left), maxDepth(root.right));
};

```

Python Code

```

# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def maxDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        if not root: return 0
        return max(self.maxDepth(root.left), self.maxDepth(

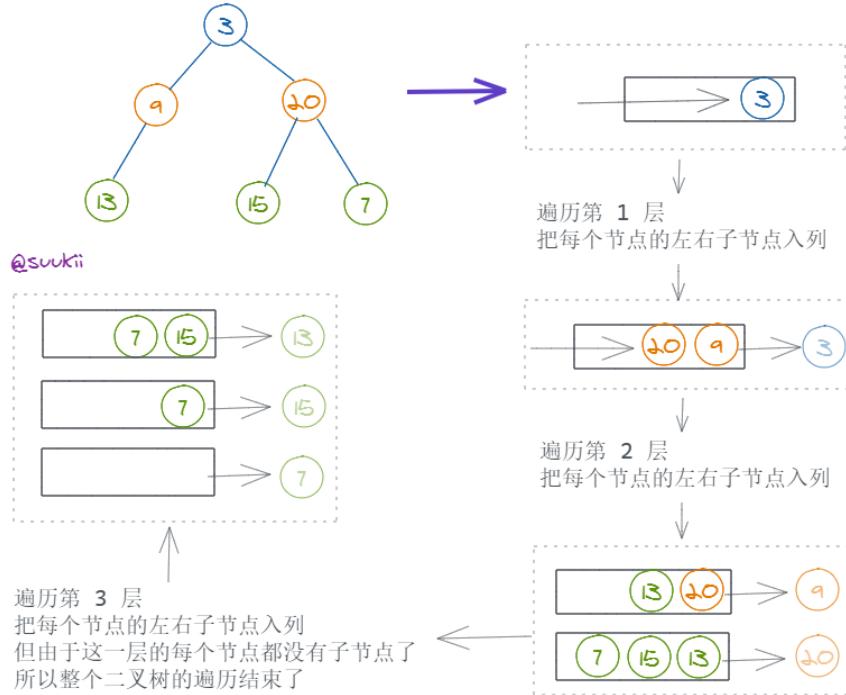
```

方法 2：循环+队列

思路

我们还可以用层级遍历的方法来找到二叉树的最大高度，一层层地遍历二叉树，每遍历一层，`height` 加一。

1. 记录当前这一层有多少个节点 `nodeCount`，当 `nodeCount` 等于 0 的时候，当前层遍历结束，`height++`，继续遍历下一层；
2. 当下一层的节点数为 0 的时候，停止遍历，返回 `height`；



复杂度分析

- 时间复杂度: $O(N)$, 因为每个节点都分别入列出列一次, 入列和出列操作的时间复杂度都是 $O(1)$, 所以总的时间复杂度是 $O(2N)$, 也就是 $O(N)$ 。
- 空间复杂度: $O(h)$ 。

伪代码

```

新建一个队列来存放遍历的节点
把 root 推入队列
初始化 height 为 0

重复以下步骤:
    记录当前层级的节点数 nodeCount (即队列中的节点数)

    if nodeCount === 0: return height // 二叉树遍历结束
    height++

    // 遍历当前层级的每个节点
    while nodeCount > 0:
        出列一个节点, 将它的左右子节点入列
        nodeCount--
    // 当前层级遍历结束, 此时队列中的是下一层的节点

```

代码

JavaScript Code

```
/*
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
var maxDepth = function (root) {
    if (!root) return 0;

    let height = 0;
    const queue = [root];

    while (queue.length) {
        let len = queue.length;
        height++;

        while (len > 0) {
            const node = queue.shift();
            node.left && queue.push(node.left);
            node.right && queue.push(node.right);
            len--;
        }
    }

    return height;
};
```

Python Code

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def maxDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """

        queue = []
        if root is not None: queue.append(root)
        height = 0

        while True:
            nodeCount = len(queue)
            if nodeCount == 0: return height
            height += 1

            while nodeCount > 0:
                node = queue.pop(0)
                if node.left is not None: queue.append(node.left)
                if node.right is not None: queue.append(node.right)
            nodeCount -= 1
```

思路

DFS

1. post-order traverse

1. 给定一个 node 他的高度是 左子树和 右子树高度的最大值加 1。
2. base condition: leaf node 的高度为 1。
3. 符合 post-order 的思路

伪代码

```
left_child_height = maxDepth(node.left)
right_child_height = maxDepth(node.right)
current_height = max(left_child_height, right_child_height)
```

```
class Solution(object):
    def maxDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """

        # post order traversal
        return self.dfs(root)
    def dfs(self, root):
        # base case
        if not root:
            return 0
        return max(self.dfs(root.left), self.dfs(root.right))
```

2. pre-order traverse

记录递归的最大深度则为，每递归一次，高度加一。重点是用一个全局变量记录当前递归最大深度。

```

class Solution(object):
    def maxDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """

    def dfs(node, depth):
        if not node:
            return
        self.ans = max(self.ans, depth)
        dfs(node.left, depth+1)
        dfs(node.right, depth+1)
    self.ans = 0
    dfs(root, 1)
    return self.ans

```

BFS

层次遍历是更加自然的想法，每遍历一层，高度加一。BFS 的重点是如何记录当前所在高度。如果只是单纯的在 queue 中 pop or push 的话是无法记录当前所在的高度。@suukii 引入了 node_count 来记录当前高度的所有 node 的个数，每 pop 一个元素，则 node_count 减少 1.直到为 0 时，我们知道遍历了当前高度的所有 node。

```

class Solution(object):
    def maxDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """

    queue = []
    if root: queue.append(root)
    height = 0
    while True:
        num_node_in_current_level = len(queue)
        if num_node_in_current_level == 0: return height
        height += 1
        while num_node_in_current_level > 0:
            current_node = queue.pop(0)
            if current_node.left: queue.append(current_node.left)
            if current_node.right: queue.append(current_node.right)
            num_node_in_current_level -= 1

```

复杂度

DFS

- time: $O(n)$ --- 每个 node 都需要访问
- space: $O(h)$ --- stack 数量刚好是树的深度 h , $\log_2 n \leq h \leq n$

BFS

- time: $O(n)$ --- 每个 node 都需要访问
- space: $O(\log_2 n)$ --- 需要用一个 queue 来装当前层的所有 node, 其最大可能值是 $\log_2 n$

题目地址 (100. 相同的树)

<https://leetcode-cn.com/problems/same-tree/>

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
- 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

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)
    );
};
```

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

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

JS Code:

```

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 false;
        }
    }

    return true;
};

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);
}

```

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

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;
}

```

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。

100.相同的树

<https://leetcode-cn.com/problems/same-tree/>

- 100.相同的树
 - 题目描述
 - 方法 1：递归
 - 思路
 - 伪代码
 - 复杂度分析
 - 代码
 - 方法 2：比较前序和中序遍历结果
 - 思路
 - 复杂度分析
 - 代码
 - 方法 3：BFS
 - 思路
 - 复杂度分析
 - 代码

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

来源：力扣（LeetCode）

链接：<https://leetcode-cn.com/problems/same-tree>

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。

方法 1：递归

思路

用 lucifer 的《产品经理法》来解决递归问题。

1. 定义函数功能，先不用管其具体实现。

我们需要一个函数，给定两个二叉树的根节点，返回这两个二叉树是否相同的判断结果。假设我们已经有这个函数 `F`，那问题就转化为 `F(root1, root2)` 了。

1. 确定大问题和小问题的关系。

要解决 `F(root1, root2)`，明显需要先解决的问题是：

- `F(root1.left, root2.left)`
- `F(root1.right, root2.right)`

而它们之间的关系也是显而易见的，两个二叉树要相等的话，当然其根节点和左右子节点都要相等，所以：

```
F(root1, root2) = root1 === root2 && F(root1.left,
root2.left) && F(root1.right, root2.right)
```

2. 补充递归终止条件

- `p, q` 不相同的话返回 `false`
- `p, q` 都是 `null` 的时候返回 `true`

伪代码

如果两个节点都存在：

1) 俩节点值相等：

 返回 `F(左子节点, 左子节点) and F(右子节点, 右子节点)`

1) 俩节点值不等：

 返回 `false`

如果两个节点都不存在：

 返回 `true`

如果两个节点一个存在一个不存在：

 返回 `false`

复杂度分析

- 时间复杂度： $O(N)$ ， N 为节点数，每个节点都要比较一次。
- 空间复杂度： $O(h)$ ， h 为树的高度。

代码

JavaScript 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} p
 * @param {TreeNode} q
 * @return {boolean}
 */
var isSameTree = function (p, q) {
    if (!p && !q) return true;
    if (!p || !q || p.val !== q.val) return false;
    return isSameTree(p.left, q.left) && isSameTree(p.right,
};

```

Python Code

```

# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution(object):
    def isSameTree(self, p, q):
        """
        :type p: TreeNode
        :type q: TreeNode
        :rtype: bool
        """

        if p is None and q is None: return True
        if p is None or q is None: return False
        if p.val != q.val: return False
        return self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)

```

方法 2：比较前序和中序遍历结果

思路

前序+中序遍历结果可以确定一棵树，所以比较这两个遍历结果就好。要注意的是，遍历的时候给空节点也留个位置。

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: $O(N)$, N 为二叉树的节点数, 遍历结果辅助数组的空间, 遍历时递归栈的最大空间是 $O(h)$, h 为二叉树的高度。

代码

JavaScript Code

```

/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} p
 * @param {TreeNode} q
 * @return {boolean}
 */
var isSameTree = function (p, q) {
    const preOrderP = preorderTraversal(p).join("##");
    const preOrderQ = preorderTraversal(q).join("##");

    const inOrderP = inorderTraversal(p).join("##");
    const inOrderQ = inorderTraversal(q).join("##");

    return preOrderP === preOrderQ && inOrderP === inOrderQ;
};

/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var preorderTraversal = function (root, res = []) {
    if (!root) {
        // 标记空节点
        res.push("#");
        return res;
    }

    res.push(root.val);
    preorderTraversal(root.left, res);
    preorderTraversal(root.right, res);

    return res;
};

/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var inorderTraversal = function (root, res = []) {
    if (!root) {
        // 标记空节点
        res.push("#");
    }
}

```

```
        return res;
    }

    inorderTraversal(root.left, res);
    res.push(root.val);
    inorderTraversal(root.right, res);

    return res;
};
```

方法 3：BFS

思路

比较两棵树的结构，可以对两棵树同时进行层级遍历，在遍历中比较节点，如果有不同的节点就提前退出。

需要注意的是遍历过程中空节点也要入列。

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： $O(\log N)$ ， N 为二叉树的节点数。

代码

JavaScript Code

```
/*
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} p
 * @param {TreeNode} q
 * @return {boolean}
 */
var isSameTree = function (p, q) {
    const queueP = [p];
    const queueQ = [q];

    while (queueP.length && queueQ.length) {
        let lenP = queueP.length;
        let lenQ = queueQ.length;

        // 如果两棵树同一层的节点数都不同，肯定不是同一棵树
        if (lenP !== lenQ) return false;

        while (lenP-- && lenQ--) {
            const nodeP = queueP.shift();
            const nodeQ = queueQ.shift();

            // 两个节点都是 null，直接继续比较下一个节点
            if (!nodeP && !nodeQ) continue;
            // 遇到不同的节点，说明不是同一棵树，提前返回
            if (!nodeP || !nodeQ || nodeP.val !== nodeQ.val) return false;

            // 将下一层的节点入列，空节点也要入列
            queueP.push(nodeP.left, nodeP.right);
            queueQ.push(nodeQ.left, nodeQ.right);
        }
    }
    return true;
};
```

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/>

题目描述

给定一个二叉树，它的每个结点都存放一个 0–9 的数字，每条从根到叶子节点的路径代表一个数字。

例如，从根到叶子节点路径 1->2->3 代表数字 123。

计算从根到叶子节点生成的所有数字之和。

说明：叶子节点是指没有子节点的节点。

示例 1：

输入： [1,2,3]



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

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 helper(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:

```
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

- 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。

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

- 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。

题目地址 (513. 找树左下角的值)

<https://leetcode-cn.com/problems/find-bottom-left-tree-value/>

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：

```
2
/ \
1   3
```

输出：

1

示例 2：

输入：

```
1
/
2   3
/   / \
4   5   6
/
7
```

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:

```
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:

```
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);
    }

}
```

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。

513.找树左下角的值

<https://leetcode-cn.com/problems/find-bottom-left-tree-value/>

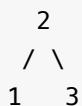
- 513.找树左下角的值
 - 题目描述
 - 方法 1: BFS
 - 思路
 - 伪代码
 - 代码
 - 复杂度分析
 - 方法 2: DFS
 - 思路
 - 伪代码
 - 复杂度分析
 - 代码

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

注意：您可以假设树（即给定的根节点）不为 NULL。

来源：力扣（LeetCode）

链接：<https://leetcode-cn.com/problems/find-bottom-left-tree-node/>

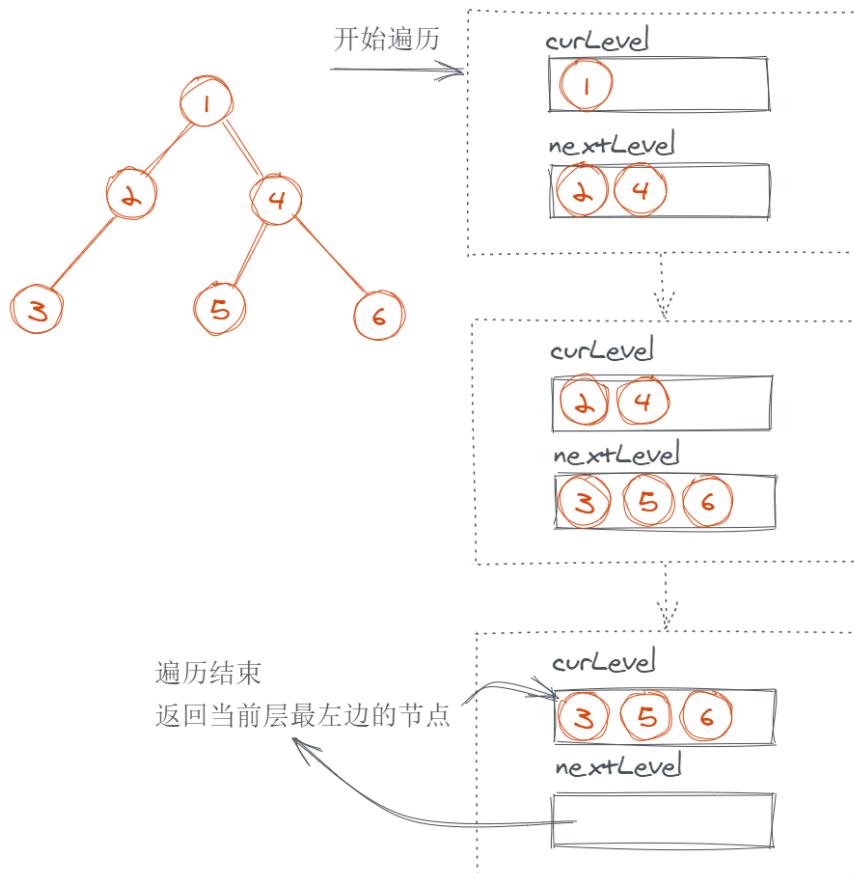
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

方法 1：BFS

思路

按照题目意思来做，找到最后一层，返回最左边的节点。

所以只要，对二叉树进行层次遍历，遍历到最后一层的时候，返回第一个节点就行了。



伪代码

层次遍历模板 1:

```

新建 curLevel 数组保存当前层的节点;
新建 nextLevel 数组保存下一层要遍历的节点;

将 root 加入到 curLevel 中, 开始遍历;

重复以下操作:
    遍历 curLevel 中的节点:
        如果该节点存在左子节点, 把左子节点加入到 nextLevel 中;
        如果该节点存在右子节点, 把右子节点加入到 nextLevel 中;

    判断 nextLevel 中是否有节点:
        如果没有, 说明已经遍历到树的最深层次, 此时 curLevel 中放的是
            让 nextLevel 作为下一个循环中要遍历的对象, 即 curLevel = nex
            将 nextLevel 清空;

```

层次遍历模板 2:

```
const queue = [root];

while (queue.length) {
    // 先记录这一层的节点数
    let len = queue.length;

    while (len--) {
        const node = queue.shift();
        node.left && queue.push(node.left);
        node.right && queue.push(node.right);
    }
}
```

代码

JavaScript Code

```
/*
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
var findBottomLeftValue = function (root) {
    let curLevel = [root],
        nextLevel = [];

    while (true) {
        for (let node of curLevel) {
            node.left && nextLevel.push(node.left);
            node.right && nextLevel.push(node.right);
        }

        if (!nextLevel.length) return curLevel[0].val;

        curLevel = nextLevel;
        nextLevel = [];
    }
};
```

Python Code

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def findBottomLeftValue(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """

        curLevel, nextLevel = [root], []
        while True:
            for node in curLevel:
                if node.left: nextLevel.append(node.left)
                if node.right: nextLevel.append(node.right)
            if not nextLevel: return curLevel[0].val
            curLevel, nextLevel = nextLevel, []
```

JavaScript Code

```

/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
var findBottomLeftValue = function (root) {
    if (!root) return 0;

    const queue = [root];
    let bottomLeft = null;
    while (queue.length) {
        let len = queue.length;

        // 每遍历一层就更新一次最左节点
        // 最后一层更新就是最后一层的最左节点
        bottomLeft = queue[0];

        while (len--) {
            const node = queue.shift();
            node.left && queue.push(node.left);
            node.right && queue.push(node.right);
        }
    }
    return bottomLeft.val;
};

```

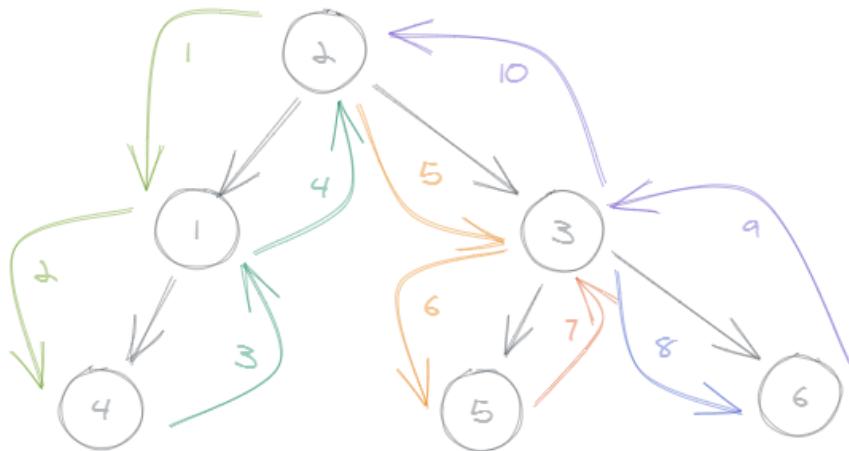
复杂度分析

- 时间复杂度: $O(N)$, 其中 N 为节点数。
- 空间复杂度: $O(q)$, q 为队列长度。最坏情况是满二叉树, 此时 q 为 $2^{\lfloor \log_2 N \rfloor}$, 即为 $O(N)$, N 为二叉树节点数。

方法 2: DFS

思路

想一下递归的路线, 如果我们先递归遍历左子树再遍历右子树, 那么每遍历到新一层的时候, 都是先访问最左边的节点。



递归路线

- 所以我们只需要在遍历到新一层的时候，记录第一个节点。
- 用 `depth` 来记录当前层次，用 `maxDepth` 来记录已经遍历过的最深层次，当 `depth > maxDepth` 的时候，说明遍历到了新层次：
 - 记录第一个节点的值
 - 更新 `maxDepth`

伪代码

创建 `ans` 来记录遍历中遇到的最左边的子节点
 创建 `maxDepth` 来记录已经遍历到的最深层次

定义一个 `dfs` 函数来遍历二叉树，函数接收 `(node, depth)` 两个参数，
 如果 `node` 为 `null`，终止遍历

```
// 因为我们是先遍历左子节点，再遍历右子节点
// 所以第一次 depth > maxDepth 的时候，遍历到的就是 depth 这个节点
// 接着我们更新 maxDepth，之后遍历同一 depth 的节点时，ans 都会是这个节点
如果当前层级 depth > maxDepth:
  更新 ans 为 node.val
  更新 maxDepth 为 depth

// 分别递归遍历左右子树
dfs(node.left, depth + 1)
dfs(node.right, depth + 1)
```

调用 `dfs` 函数，传入 `(root, 0)`
 返回 `ans`

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。

代码

JavaScript Code

```
/**  
 * Definition for a binary tree node.  
 * function TreeNode(val) {  
 *     this.val = val;  
 *     this.left = this.right = null;  
 * }  
 */  
/**  
 * @param {TreeNode} root  
 * @return {number}  
 */  
var findBottomLeftValue = function (root) {  
    let maxDepth = 0;  
    let ans = 0;  
    helper(root, 1);  
    return ans;  
  
    // *****  
  
    function helper(root, depth) {  
        if (!root) return 0;  
  
        if (depth > maxDepth) {  
            maxDepth = depth;  
            ans = root.val;  
        }  
  
        helper(root.left, depth + 1);  
        helper(root.right, depth + 1);  
    }  
};
```

Python Code

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    _ans = 0
    _maxDepth = 0

    def helper(self, root, depth):
        if root is None: return 0

        if depth > self._maxDepth:
            self._maxDepth = depth
            self._ans = root.val

        self.helper(root.left, depth + 1)
        self.helper(root.right, depth + 1)

    def findBottomLeftValue(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        self.helper(root, 1)
        return self._ans
```

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/>

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：

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

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:

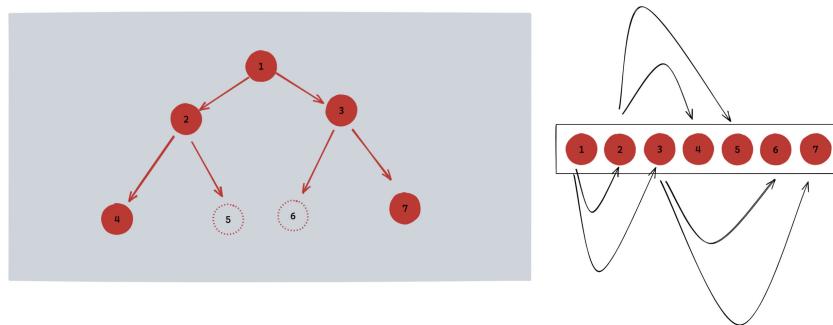
    def serialize(self, root):
        q = collections.deque([root])
        ans = ''
        while q:
            cur = q.popleft()
            if cur:
                ans += str(cur.val) + ','
                q.append(cur.left)
                q.append(cur.right)
            else:
                # 除了这里不一样，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 $(i + 1) / 2$ 。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：

```

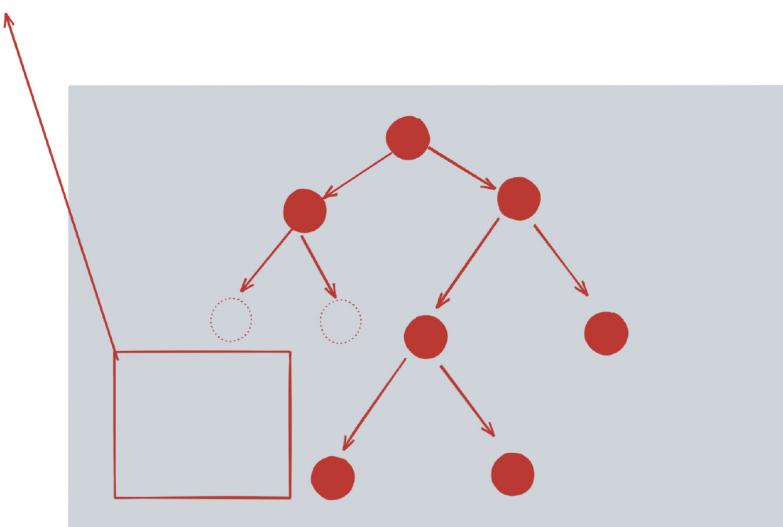
def deserialize(self, data):
    if data == 'null': return None
    nodes = data.split(',')
    root = TreeNode(nodes[0])
    # 从一号开始编号，编号信息一起入队
    q = collections.deque([(root, 1)])
    while q:
        cur, i = q.popleft()
        #  $2 * i$  是左节点，而  $2 * i$  编号对应的其实是索引为  $2 * i - 1$ 
        if 2 * 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

```

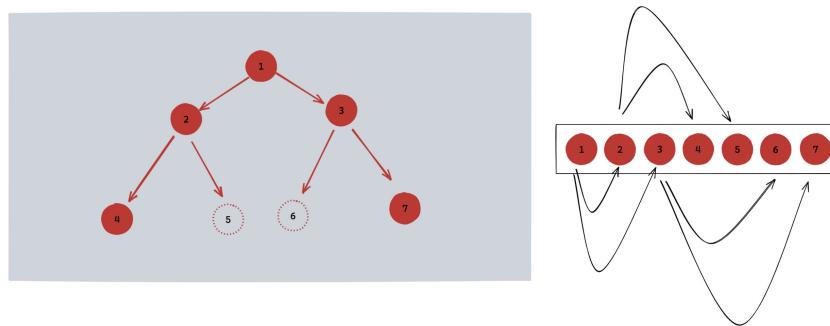
但是上面的代码是不对的，因为我们序列化的时候其实不是完全二叉树，这也是上面我埋下的伏笔。因此遇到类似这样的 case 就会挂：

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（如果存在的話），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:

```

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:

```

# 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>

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该结点值。

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。

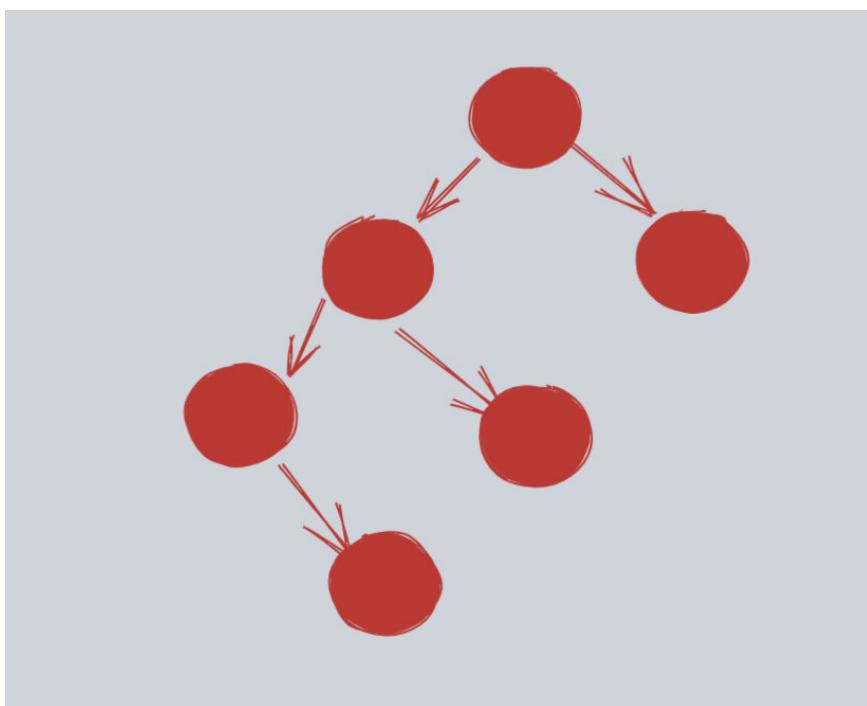
前置知识

- 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:

```
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):

```

/**
 * 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);
}

```

```

    }
};
```

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。

987. 二叉树的垂序遍历

<https://leetcode-cn.com/problems/vertical-order-traversal-of-a-binary-tree/>

- 987. 二叉树的垂序遍历
 - 题目描述
 - 方法 1: DFS 记录坐标+排序
 - 思路
 - 复杂度分析
 - 代码
 - 方法 2: BFS 记录坐标+排序
 - 思路
 - 复杂度分析
 - 代码

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，报告该结点的值。

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。

来源：力扣（LeetCode）

链接：<https://leetcode-cn.com/problems/vertical-order-traversal-of-binary-tree/>

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。

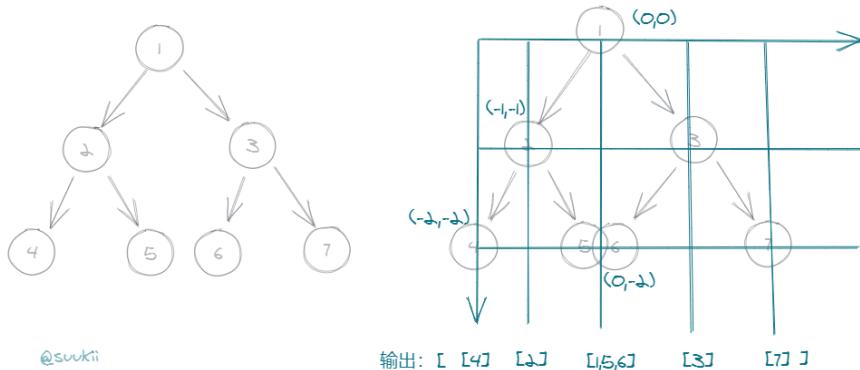
方法 1：DFS 记录坐标+排序

思路

其实理解了题意之后，就是一个简单的 `DFS+排序` 而已。

我们把二叉树放到坐标网格上看看。以根节点为原点，往左的节点 `x--`，往右的节点 `x++`，往下的节点 `y--`。

- 把节点坐标和值按 `x` 坐标分组，然后给这些分组按升序排序。
- 在分组内，给节点按 `y` 坐标降序排序，如果 `y` 坐标相同，再按节点值升序排。



复杂度分析

- 时间复杂度： $O(N \log N)$ ， N 为二叉树的节点数。遍历二叉树的时间是 $O(N)$ ，排序的时间就认为是 $O(N \log N)$ 吧。
- 空间复杂度： $O(N)$ ， N 为二叉树的节点数。用来存储节点坐标信息的 `pos` 空间是 N ，递归栈的空间是 $O(h)$ ， h 为二叉树高度。

代码

JavaScript Code

```

/**
 * 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);
}

```

```
    }  
};
```

方法 2：BFS 记录坐标+排序

思路

跟方法 1 的区别只在于遍历方式。

复杂度分析

- 时间复杂度： $O(N \log N)$ ，N 为二叉树的节点数。遍历二叉树的时间是 $O(N)$ ，排序的时间就认为是 $O(N \log N)$ 吧。
- 空间复杂度： $O(N)$ ，N 为二叉树的节点数。用来存储节点坐标信息的 `pos` 空间是 N，队列的空间是 $O(q)$ ，最坏情况下 q 与 N 同阶。

代码

JavaScript Code

```


/**
 * 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 = bfs(root);

    // 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 bfs(root) {
    // 队列中数据格式是: [x, y, val]
    const queue = [[0, 0, root]];
    const pos = {};
    while (queue.length) {
        let size = queue.length;

        while (size--) {
            const [x, y, node] = queue.shift();
            x in pos || (pos[x] = []);
            // 保存坐标数据到 pos，格式是: [y, val]
        }
    }
}


```

```
    pos[x].push([y, node.val]);
    node.left && queue.push([x - 1, y - 1, node.left]);
    node.right && queue.push([x + 1, y - 1, node.right]);
}
}
return pos;
};
};
```

题目地址 - 两数之和

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

题目描述

给定一个整数数组 `nums` 和一个目标值 `target`, 请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。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。

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

代码

```
function twoSum(nums: number[], target: number): number[] {
    for (let i = 0; i < nums.length; i++) {
        for (let j = i + 1; j < nums.length; j++) {
            if (nums[i] + nums[j] === target) {
                return [i, j]
            }
        }
    }
    return [0, 0]
};
```

复杂度分析

- 空间复杂度: $O(1)$
- 时间复杂度: $O(n * n)$, n 为数组长度

思路

代码

上面是用于搜索整个数组的方式来判断 $\text{target} - \text{num}$ 是否也存在 nums ，我们也可以用哈希表记录所有已经遍历过的数字，判断 $\text{target} - \text{num}$ 是否出现时，直接查表即可。

哈希表是非常常用的时间换空间的方式

```
function twoSum(nums: number[], target: number): number[] {
    const m = new Map()
    for (let i = 0; i < nums.length; i++) {
        if (m.has(target - nums[i])) {
            return [m.get(target - nums[i]), i]
        }
    }
    return [0, 0]
};
```

复杂度分析

- 空间复杂度: $O(n)$
- 时间复杂度: $O(n)$

题目地址 - 两数之和

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

解法一

时空复杂度

时间复杂度: $O(n)$

空间复杂度: $O(n)$

抢个沙发

```
var twoSum = function (nums, target) {
    let map = new Map()
    for (let i = 0; i < nums.length; i++) {
        const getIndex = map.get(target - nums[i])
        if (getIndex != undefined) {
            return [getIndex, i]
        } else {
            map.set(nums[i], i)
        }
    }
};
```

[three sum](#)

解法一

时空复杂度

时间复杂度: $O(n^2)$ 排序sort $n \log n$ + 双层遍历 $n^2 \Rightarrow n^2$

空间复杂度: $O(1)$

```
var threeSum = function (nums) {
    let res = []
    if (nums.length < 3) return []
    nums.sort((a, b) => a - b)
    for (let i = 0; i < nums.length; i++) {
        if (nums[i] > 0) continue
        let left = i + 1
        let right = nums.length - 1
        let target = 0 - nums[i]
        while (left < right) {
            const leftCarry = nums[left]
            const rightCarry = nums[right]
            const sum = leftCarry + rightCarry
            if (sum === target) res.push([nums[i], leftCarry, rightCarry])
            if (sum <= target) while (left < right && nums[left] === nums[left + 1]) left++
            if (sum >= target) while (left < right && nums[right] === nums[right - 1]) right--
        }
    }
    return res
};
```

解法二

时空复杂度

时间复杂度: $O(n^2)$

空间复杂度: $O(1)$

看了眼题解 可以抽离出来 后面可以做递归nSum

```
var twoSumTarget = function (arr, start, target) {
    let left = start
    let right = arr.length - 1
    let res = []
    while (left < right) {
        let leftCarry = arr[left]
        let rightCarry = arr[right]
        let sum = leftCarry + rightCarry
        if (sum === target) res.push([nums[i], leftCarry, i])
        else if (sum < target) left++
        else right--
    }
    return res
}

var threeSum = function (nums) {
    nums.sort((a, b) => a - b)
    let res = []
    for (let i = 0; i < nums.length; i++) {
        if (nums[i] > 0) continue
        let data = twoSumTarget(nums, i + 1, 0 - nums[i])
        data.forEach(v => res.push([nums[i], ...v]))
        while (i < nums.length - 1 && nums[i] == nums[i + 1])
            i++
    }
    return res
};
```

four sum

解法一

时空复杂度

时间复杂度: $O(n^3)$

空间复杂度: $O(1)$

改一下threesum 就是套娃

```

var threeSum = function (nums, start, highTarget) {
    let res = []
    if (nums.length < 3) return []
    for (let i = start; i < nums.length; i++) {
        let left = i + 1
        let right = nums.length - 1
        let target = highTarget - nums[i]
        while (left < right) {
            const leftCarry = nums[left]
            const rightCarry = nums[right]
            const sum = leftCarry + rightCarry
            if (sum === target) res.push([nums[i], leftCarry, rightCarry])
            if (sum <= target) while (left < right && nums[left] === nums[left + 1]) left++
            if (sum >= target) while (left < right && nums[right] === nums[right - 1]) right--
        }
        while (i < nums.length - 1 && nums[i] === nums[i + 1]) i++
    }
    return res
};

var fourSum = function (nums, target) {
    let res = []
    if (nums.length < 4) return []
    nums.sort((a, b) => a - b)
    for (let i = 0; i < nums.length; i++) {
        let data = threeSum(nums, i + 1, target - nums[i])
        data.forEach(v => res.push([nums[i], ...v]))
        while (i < nums.length - 1 && nums[i] === nums[i + 1]) i++
    }
    return res
};

```

扩展 nSum

将for循环中找值的操作作为n=2时的递归终止操作，将找到的数组往上递归拼接

```
var fourSum = function (nums, target) {
    let res = []
    nums.sort((a, b) => a - b)
    return nSum(nums, target, 0, 4)
};

var nSum = function (nums, target, start, n) {
    if (n < 2 || nums.length < n) return []
    let res = []
    if (n == 2) {
        let left = start
        let right = nums.length - 1
        while (left < right) {
            const leftCarry = nums[left]
            const rightCarry = nums[right]
            const sum = leftCarry + rightCarry
            if (sum === target) res.push([leftCarry, rightCarry])
            if (sum <= target) while (left < right && nums[left] === nums[left + 1]) left++
            if (sum >= target) while (left < right && nums[right] === nums[right - 1]) right--
        }
    } else {
        for (let i = start; i < nums.length; i++) {
            let data = nSum(nums, target - nums[i], i + 1, n - 1)
            data.forEach(v => res.push([nums[i], ...v]))
            while (i < nums.length - 1 && nums[i] === nums[i + 1]) i++
        }
    }
    return res
}
```

题目地址 - 347. 前 K 个高频元素

<https://leetcode-cn.com/problems/top-k-frequent-elements/>

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。

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

以下是代码以及详细说明

思路 - 排序

1. 哈希表记录数值频次
2. 数值去重后根据频次排序取前K大的

```
var topKFrequent = function (nums, k) {
  const counts = {};
  for (let num of nums) {
    counts[num] = (counts[num] || 0) + 1;
  }
  return [...new Set(nums)].sort((a, b) => counts[b] - counts[a]).slice(0, k);
};
```

- 时间复杂度: $O(N * \log N)$, N 为数组长度
- 空间复杂度: $O(N)$, N 为数组长度

思路 - 建堆

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 ，并维持小顶堆性质。

```

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, 所有通过拆分出 K个数, 使得前K个数都大于后面 $n - k$ 个数即可。

代码

```

/**
 * @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;
    }
}

```

```
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 $O(N * N)$
- 空间复杂度: $O(N)$

347. 前 K 个高频元素

<https://leetcode-cn.com/problems/top-k-frequent-elements/>

- 347. 前 K 个高频元素
 - 题目描述
 - 方法 1：哈希表
 - 思路
 - 复杂度分析
 - 代码
 - 方法 2：大顶堆
 - 思路
 - 复杂度分析
 - 代码
 - 方法 3：小顶堆
 - 思路
 - 复杂度分析
 - 代码
 - 方法 4：快速选择
 - 思路
 - 复杂度分析
 - 代码

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。

方法 1：哈希表

思路

- 遍历一遍数组，用哈希表统计每个数字出现的次数。
- 按次数排序，然后输出前 k 个数字。

复杂度分析

- 时间复杂度： $O(m \log m)$ ， m 是数组中不同数字的数量，最大是 N ，数组的长度，这是排序的时间。
- 空间复杂度： $O(m)$ ， m 是数组中不同数字的数量，哈希表的空间。

代码

JavaScript Code

```

/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number[]}
 */
var topKFrequent = function (nums, k) {
    // 统计出现次数
    const map = {};
    for (const n of nums) {
        n in map || (map[n] = 0);
        map[n]++;
    }
    // 对次数进行排序然后输出前 k 个
    return Object.entries(map)
        .sort((a, b) => b[1] - a[1])
        .slice(0, k)
        .map(a => a[0]);
};

```

方法 2：大顶堆

思路

看到 前 k 个 这种描述就能想到 堆 了，还是先把数字出现的次数统计在哈希表中，然后入堆按次数排序，再吐出来 k 个。

复杂度分析

- 时间复杂度： $O(k \log m)$ ， m 是数组中不同数字的数量，堆中有 m 个元素，移除堆顶的时间是 $\log m$ ，重复操作了 k 次。
- 空间复杂度： $O(m)$ ， m 是数组中不同数字的数量，哈希表和堆的空间。

代码

JavaScript Code

```

    /**
     * @param {number[]} nums
     * @param {number} k
     * @return {number[]}
     */
    var topKFrequent = function (nums, k) {
        // 统计出现次数
        const map = {};
        for (const n of nums) {
            n in map || (map[n] = 0);
            map[n]++;
        }
        // 入堆，格式是：[数字, 次数]，按次数排序
        const maxHeap = new MaxHeap(Object.entries(map), function (
            inserted,
            compared,
        ) {
            return inserted[1] < compared[1];
        });
        // 输出前 k 个
        const res = [];
        while (k-- > 0) {
            res.push(maxHeap.pop()[0]);
        }
        return res;
    };

    // ****

    class Heap {
        constructor(list = [], comparator) {
            this.list = list;

            if (typeof comparator != 'function') {
                this.comparator = function comparator(inserted,
                    compared) {
                    return inserted < compared;
                };
            } else {
                this.comparator = comparator;
            }

            this.init();
        }

        init() {
            const size = this.size();
            for (let i = Math.floor(size / 2) - 1; i >= 0; i--) {
                this.heapify(this.list, size, i);
            }
        }

        heapify(list, size, index) {
            let smallest = index;
            const left = 2 * index + 1;
            const right = 2 * index + 2;
            const len = list.length;

            if (left < size && this.comparator(list[left], list[smallest]) < 0) {
                smallest = left;
            }
            if (right < size && this.comparator(list[right], list[smallest]) < 0) {
                smallest = right;
            }
            if (smallest != index) {
                [list[index], list[smallest]] = [list[smallest], list[index]];
                this.heapify(list, size, smallest);
            }
        }

        size() {
            return this.list.length;
        }

        push(item) {
            this.list.push(item);
            this.heapify(this.list, this.size(), this.size() - 1);
        }

        pop() {
            if (this.size() === 0) {
                return null;
            }
            const item = this.list[0];
            this.list[0] = this.list[this.size() - 1];
            this.list.pop();
            this.heapify(this.list, this.size(), 0);
            return item;
        }

        peek() {
            return this.list[0];
        }
    }
}

```

```

        }
    }

    insert(n) {
        this.list.push(n);
        const size = this.size();
        for (let i = Math.floor(size / 2) - 1; i >= 0; i--) {
            this.heapify(this.list, size, i);
        }
    }

    peek() {
        return this.list[0];
    }

    pop() {
        const last = this.list.pop();
        if (this.size() === 0) return last;
        const returnItem = this.list[0];
        this.list[0] = last;
        this.heapify(this.list, this.size(), 0);
        return returnItem;
    }

    size() {
        return this.list.length;
    }
}

class MaxHeap extends Heap {
    constructor(list, comparator) {
        super(list, comparator);
    }

    heapify(arr, size, i) {
        let largest = i;
        const left = Math.floor(i * 2 + 1);
        const right = Math.floor(i * 2 + 2);

        if (left < size && this.comparator(arr[largest], arr[left]) < 0)
            largest = left;
        if (right < size && this.comparator(arr[largest], arr[right]) < 0)
            largest = right;

        if (largest !== i) {
            [arr[largest], arr[i]] = [arr[i], arr[largest]];
            this.heapify(arr, size, largest);
        }
    }
}

```

```
    }  
}
```

方法 3：小顶堆

思路

统计每个数字出现的次数后，维护一个大小为 k 的小顶堆。

复杂度分析

- 时间复杂度： $O(k \log k)$ 。
- 空间复杂度： $O(m)$ ， m 是数组中不同数字的数量，哈希表的空间，小顶堆的空间是 $O(k)$ ， $m \geq k$ 。

代码

JavaScript Code

```

/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number[]}
 */
var topKFrequent = function (nums, k) {
    // 统计出现次数
    const map = {};
    for (const n of nums) {
        n in map || (map[n] = 0);
        map[n]++;
    }

    const minHeap = new MinHeap([], function comparator(inserted, compared) {
        return inserted[1] > compared[1];
    });

    // 维护一个大小为 k 的小顶堆，堆元素格式是：[数字, 次数]，按次数
    Object.entries(map).forEach(([num, times]) => {
        const [, minTimes] = minHeap.peek() || [, 0];
        // 小顶堆大小还没达到 k，继续插入新元素
        if (minHeap.size() < k) {
            minHeap.insert([num, times]);
        }

        // 小顶堆大小为 k，如果新元素次数大于堆顶，弹出堆顶，插入新元素
        // 否则就不用管这个元素了
        else if (minHeap.size() === k && times > minTimes)
            minHeap.pop();
        minHeap.insert([num, times]);
    });
};

// 反序输出小顶堆中的所有元素
const res = Array(k);
while (k-- > 0) {
    res[k] = minHeap.pop()[0];
}
return res;
};

// ****

class MinHeap extends Heap {
    constructor(list, comparator) {
        if (typeof comparator != 'function') {
            comparator = function comparator(inserted, compared) {
                return inserted > compared;
            };
        }
        this.list = list;
        this.comparator = comparator;
        this.size = list.length;
        this.buildMinHeap();
    }

    buildMinHeap() {
        for (let i = Math.floor(this.size / 2) - 1; i >= 0; i--) {
            this.heapify(i);
        }
    }

    heapify(index) {
        let leftIndex = index * 2 + 1;
        let rightIndex = index * 2 + 2;
        let smallestIndex = index;

        if (leftIndex < this.size && this.comparator(this.list[leftIndex], this.list[smallestIndex]) < 0) {
            smallestIndex = leftIndex;
        }

        if (rightIndex < this.size && this.comparator(this.list[rightIndex], this.list[smallestIndex]) < 0) {
            smallestIndex = rightIndex;
        }

        if (smallestIndex !== index) {
            [this.list[index], this.list[smallestIndex]] = [this.list[smallestIndex], this.list[index]];
            this.heapify(smallestIndex);
        }
    }

    insert(item) {
        this.list.push(item);
        this.size++;
        this.heapify(Math.floor((this.size - 1) / 2));
    }

    pop() {
        if (this.size === 0) {
            return null;
        }

        const item = this.list[0];
        const lastItem = this.list.pop();
        this.size--;
        if (this.size === 0) {
            return item;
        }

        this.list[0] = lastItem;
        this.heapify(0);
        return item;
    }

    peek() {
        return this.list[0];
    }

    size() {
        return this.size;
    }
}

```

```

        };
    }
    super(list, comparator);
}

heapify(arr, size, i) {
    let smallest = i;
    const left = Math.floor(i * 2 + 1);
    const right = Math.floor(i * 2 + 2);
    if (left < size && this.comparator(arr[smallest], arr[left]) < 0)
        smallest = left;
    if (right < size && this.comparator(arr[smallest], arr[right]) < 0)
        smallest = right;

    if (smallest !== i) {
        [arr[smallest], arr[i]] = [arr[i], arr[smallest]];
        this.heapify(arr, size, smallest);
    }
}
}
}

```

方法 4：快速选择

思路

https://github.com/suukii/Articles/blob/master/articles/dsa/quick_select.md

复杂度分析

- 时间复杂度： $O(N)$ ，平均是 $O(N)$ ，虽然理论上最差情况是 $O(N^2)$ ，但实际应用的话效率还是不错的。
- 空间复杂度： $O(N)$ ，哈希表的空间是 $O(m)$ ， m 是数组中不同数字的数量。快速选择中递归栈最差应该是 $O(N)$ 。

代码

JavaScript Code

```

/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number[]}
 */
var topKFrequent = function (nums, k) {
    // 统计出现次数
    const map = {};
    for (const n of nums) {
        n in map || (map[n] = 0);
        map[n]++;
    }

    const list = Object.entries(map);
    quickSelect(list, k, 0, list.length - 1, function (item) {
        return item[1] >= pivot[1];
    });

    return list.slice(0, k).map(el => el[0]);
};

/**
 * 把 arr[r] 当成是 pivot
 * 把大于等于 pivot 的数字放到左边
 * 把小于 pivot 的数字放到右边
 * @param {number[]} arr
 * @param {number} l
 * @param {number} r
 */
function partition(arr, l, r, comparator) {
    if (typeof comparator != 'function') {
        comparator = function (num, pivot) {
            return num >= pivot;
        };
    }

    let i = l;
    for (let j = l; j < r; j++) {
        if (comparator(arr[j], arr[r])) {
            [arr[i], arr[j]] = [arr[j], arr[i]];
            i++;
        }
    }
    // 将 pivot 换到分界点
    [arr[i], arr[r]] = [arr[r], arr[i]];
    // 返回 pivot 的下标
    return i;
}

```

```
/*
 * 寻找第 k 大元素
 * 如果 pivot 的下标刚好是 k - 1, 那我们就找到了
 * 如果下标大于 k - 1, 那就在 [left, pivotIndex - 1] 这段找第 k
 * 如果下标小于 k - 1, 那就对 [pivotIndex + 1, right] 这段找第
 * @param {number[]} list
 * @param {number} left
 * @param {number} right
 * @param {number} k
 * @param {function} comparator
 */
function quickSelect(list, k, left = 0, right = list.length) {
    if (left >= right) return list[left];
    const pivotIndex = partition(list, left, right, comparator);

    if (pivotIndex - left === k - 1) return list[pivotIndex];
    else if (pivotIndex - left > k - 1)
        return quickSelect(list, k, left, pivotIndex - 1, comparator);
    else
        return quickSelect(
            list,
            k - pivotIndex + left - 1,
            pivotIndex + 1,
            right,
            comparator,
        );
}
```

题目地址

<https://leetcode-cn.com/problems/number-of-boomerangs/>

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) ，其中 i 和 j 之间的距离和 i 和 k 之间的距离相等（需要考虑元组的顺序）。

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 $[-10000, 10000]$ 中。

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 $[[1, 0], [2, 0], [0, 0]]$

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：

$$x = (x_1, x_2)$$

$$y = (y_1, y_2)$$

$$\text{dis}(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

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;
}

```

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 $n^*(n-1)$, 这是说距离为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! 上代码:

```

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;
}

private int getDistance(int[] x, int[] y) {

    int x1 = y[0] - x[0];
    int y1 = y[1] - x[1];

    return x1 * x1 + y1 * y1;
}

```

这样时间复杂度就被优化为 $O(N^2 * \max(\text{different_distance_count}))$

447.回旋镖的数量

<https://leetcode-cn.com/problems/number-of-boomerangs>

- 447.回旋镖的数量
 - 题目描述
 - 方法 1：哈希表
 - 思路
 - 复杂度分析
 - 代码
 - 输入输出
 - 方法 2：暴力法
 - 思路
 - 复杂度
 - 代码

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) ，其中

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]]`

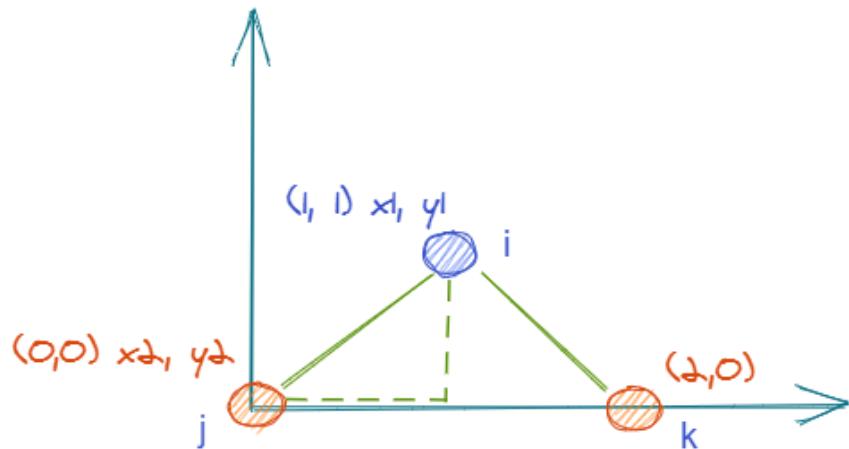
来源：力扣（LeetCode）

链接：<https://leetcode-cn.com/problems/number-of-boomerangs>

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。

方法 1：哈希表

思路



输出: [[1, 1], [0, 0], [2, 0]], [[1, 1], [2, 0], [0, 0]]

根据题意，回旋镖是指，坐标上有三个点 i, j, k ， i 到 j 的距离等于 i 到 k 的距离。上图中，蓝色的点到两个橙色的点距离是一样的，所以 (i, j, k) 构成了一个回旋镖，然后其实 (i, k, j) 也构成了一个回旋镖。

其实可以想到，对于每个点，我们只需要将其他点按照距离分组记录，比如上图中的 $[1, 1]$ ，跟它距离 $\sqrt{2}$ 的点有 $[0, 0]$ 和 $[2, 0]$ 。然后再计算排列组合数，也就是对 $\sqrt{2}$ 这个距离的分组中的点进行两两排列组合，就可以得出答案了。

如果用哈希表来存，它的结构大概是这样的：

```
{
    point1: {
        dist1: [point2, point3],
        dist2: [point4]
    },
    point2: {},
}
```

每个点都维护一个哈希表，哈希表的键是这个点到其他点的距离，值就是在这个距离的点有哪些。

不过题目只要求输出回旋镖的数量，所以我们只需要记录在某个距离的点有几个，并不需要记录具体的坐标，比如 `dist1: 2` 就可以了。

计算两点距离公式：

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

不过其实我们只关心距离是否一样，并不关心实际距离是多少，所以实际上不需要开根号。

两两组合数：

一个集合中有 N 个数，每个数都可以跟其余 $N-1$ 个数进行组合，一共有 $N(N-1)$ 种组合数。

复杂度分析

- 时间复杂度： $O(N^2)$ ， N 是数组长度。在内层循环中，寻找该点到其他点的距离时间是 $O(N)$ ，计算组合数的时间是 $O(m)$ ， m 是每个点到其他点的不同距离总数的最大值， m 最大值是 $N-1$ 。所以总的时间是 $O(N(N+m))$ ，差不多是 $O(N^2)$ 。
- 空间复杂度： $O(N)$ ，最坏的情况是每个点到其他点的距离都不一样，那每个哈希表的大小就是 $N-1$ 。

代码

JavaScript Code

```
/**
 * @param {number[][]} points
 * @return {number}
 */
var number0fBoomerangs = function (points) {
    let count = 0;

    points.forEach((a, i) => {
        const map = {};

        points.forEach((b, j) => {
            if (a !== b) {
                const dist = calcDist0f2Points(a, b);
                map[dist] = (map[dist] || 0) + 1;
            }
        });
        for (const dist in map) {
            const num = map[dist];
            if (num > 1) count += num * (num - 1);
        }
    });

    return count;

    // *****
    function calcDist0f2Points([x1, y1], [x2, y2]) {
        return (x1 - x2) ** 2 + (y1 - y2) ** 2;
    }
};
```

输入输出

Node.js

```
const __main__ = function () {
    const readline = require('readline');
    const rl = readline.createInterface({
        input: process.stdin,
        output: process.stdout,
    });

    console.log('\n输入: \n');
    rl.prompt();
    const inputs = [];
    rl.on('line', line => inputs.push(line));

    const outputs = [];
    rl.on('close', () => {
        inputs.forEach(line => {
            const output = numberOfBoomerangs(JSON.parse(line));
            outputs.push(output);
        });
        console.log('\n输出: \n');
        outputs.forEach(el => console.log(` ${el}\n`));
    });
};
```

方法 2：暴力法

思路

三层循环吧，我就不写了。

复杂度

- 时间复杂度： $O(N^3)$ 。
- 空间复杂度： $O(1)$ 。

代码

略

题目地址

<https://leetcode-cn.com/problems/longest-substring-without-repeating-characters/>

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
```

思路

哈希表

遍历字符串，对每个位置*i*，查看从*i*开始的最长无重复子串长度。这里查看字符有无重复时，可以通过一个哈希表记录字符出现情况

代码 - javascript

```

/**
 * @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
};

```

- 时间复杂度: $O(n * n)$, 为字符串长度
- 空间复杂度: $O(n)$, n 为字符串长度

哈希表 + 滑动窗口

主要到每次 i 指针变化时, j 指针对应重置, 但是注意到之前的 i 到之前的 j 这段区间是无重复的, j 指针重置后, 这段区间的重复信息又得重新计算, 这里可以用双指针利用之前的计算结果。

具体算法如下

1. i, j 指针分别指向字符串开头
2. j 指针自增, 如果出现重复字符, j 指针不动, i 指针向前走

代码 - javascript

```
/**  
 * @param {string} s  
 * @return {number}  
 */  
var lengthOfLongestSubstring = function (s) {  
    let res = 0  
    s = '#' + s  
    const set = new Set(['#'])  
    for (let i = 1, j = 1; i < s.length; i++) {  
        set.delete(s[i - 1])  
        while (j < s.length && !set.has(s[j])) {  
            set.add(s[j])  
            res = Math.max(res, j - i + 1)  
            j++  
        }  
    }  
    return res  
};
```

- 时间复杂度: $O(n)$
- 空间复杂度: $O(n)$

3. 无重复字符的最长子串

<https://leetcode-cn.com/problems/longest-substring-without-repeating-characters/>

- 3. 无重复字符的最长子串
 - 题目描述
 - 方法 1：滑动窗口+哈希表
 - 思路
 - 复杂度分析
 - 代码

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

来源：力扣（LeetCode）

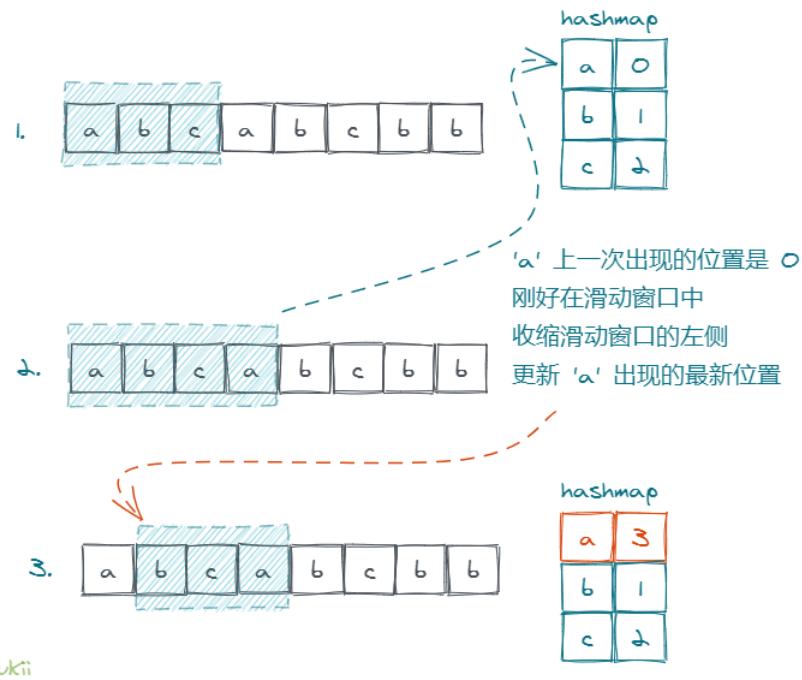
链接：<https://leetcode-cn.com/problems/longest-substring-without-repeating-characters/>
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

方法 1：滑动窗口+哈希表

思路

- 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。



复杂度分析

- 时间复杂度: $O(N)$, N 为 s 长度。
- 空间复杂度: $O(d)$, d 是字符集的大小，但哈希表最大的大小也只是 $O(N)$ 。

代码

JavaScript Code

```
/**  
 * @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;  
};
```

题目地址

<https://leetcode-cn.com/problems/substring-with-concatenation-of-all-words>

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 $words$ 中所有单词串联形成的子串的起始位置。

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 $words$ 中单词串联的顺序。

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: []

思路

还是从题意暴力入手，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 $(words.length)! / (words$ 中单词重复次数相乘)，时间复杂度为 $O(m!)$ ， m 为 $words$ 长度。

阶乘的算法可以直接贴死囚了，看第二种思路：遍历 s 串的时间复杂度为 $O(n - m)$ ， n 为 s 字符串长度， m 为 $words[0].length * words.length$ 。

关键在于如何判断子串Y是否可以由words数组构成，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hash表比对出现次数即可。

代码

```
/*
 * @param {string} s
 * @param {string[]} words
 * @return {number[]}
 */
var findSubstring = function (s, words) {
    const map = {};
    const res = [];
    const len = words[0] ?.length || 0;
    // 哈希表，记录单词出现次数
    for (let word of words) {
        map[word] = (map[word] || 0) + 1;
    }
    // 遍历所有子串
    for (let i = 0; i < s.length; i++) {
        const cache = { ...map };
        for (let j = 0; j < words.length; j++) {
            const curSub = s.substr(i + j * len, len);
            if (!cache[curSub]) {
                break;
            } else {
                // 对每一个子串做减法
                cache[curSub]--;
                if (j === words.length - 1) {
                    res.push(i);
                }
            }
        }
    }
    return res;
};
```

- 时间复杂度: $O(n * m)$, n为字符串S长度, m为words数组字符长度
- 空间复杂度: $O(m)$, m 为words数组长度

题目地址

<https://leetcode-cn.com/problems/sudoku-solver/>

题目内容

编写一个程序，通过填充空格来解决数独问题。

一个数独的解法需遵循如下规则：

数字 1–9 在每一行只能出现一次。

数字 1–9 在每一列只能出现一次。

数字 1–9 在每一个以粗实线分隔的 3×3 宫内只能出现一次。

空白格用 ‘.’ 表示。

答案被标成红色。

提示：

给定的数独序列只包含数字 1–9 和字符 ‘.’。

你可以假设给定的数独只有唯一解。

给定数独永远是 9×9 形式的。

来源：力扣（LeetCode） 链接：<https://leetcode-cn.com/problems/sudoku-solver> 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。

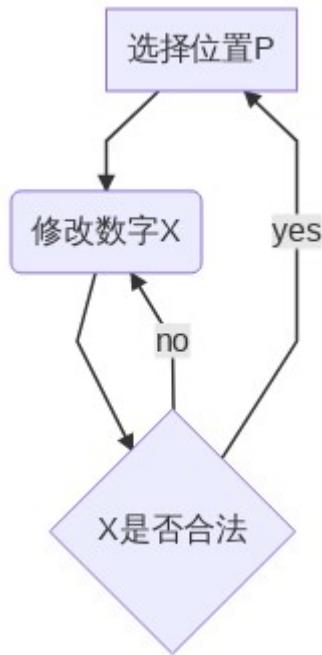
思路

先想象一下，如果让你玩一个数独游戏，你会怎么做？

分解一下，我们一般会通过以下流程去解数独

1. 尝试在一个空位写下一个数 x , $1 \leq x \leq 9$
2. 看这个数是否符合数独规则(横纵线, 9个小矩形是否有重复)
 - i. 如果写下的数字合法，回到第1步
 - ii. 如果数字非法，擦掉 x ，换一个数，回到2

流程图



那么我们通过代码把上述方式表述出来。这里有两个问题

1. 如何有效判断位置P的X是否合法
2. 如果非法，如何修改X

针对 1, 每个位置X涉及到横纵以及九宫格3个信息，判断是否合法也就是说P上的X在3个地方是否出现过，那最简单的方法就是用3个哈希表 T1, T2, T3 分别表示横纵九宫格3个区域，看X是否在T1, T2, T3 中是否出现。这里可以用哈希表，也可以用数组，bit做状态压缩，这些取决于你

针对二，可以用回溯的方法，回溯之后会有展开，这里先简单提下，回溯是搜索的一种应用，而搜索，说白了就是穷举搜索空间内的所有状态，我们对每个位置P穷举1-9数字，如果合法就进入下一个位置继续，非法回到上一个位置继续下一个数字，回溯模板可以参考

```
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
}
```

有以上分析后，代码就很好写了，以下是js代码

```

/**
 * 1. 根据给定数组生成3个hashtable
 * 2. 通过回溯枚举所有可能序列
 * @param {character[][]} board
 * @return {void} Do not return anything, modify board in-place
 */
var solveSudoku = function (board) {
    const cols = [];
    const rows = [];
    const boxes = [];
    for (let i = 0; i < 9; i++) {
        const term = Array.from({ length: 10 }).fill(0);
        cols.push(term.slice());
        rows.push(term.slice());
        boxes.push(term.slice());
    }

    // 生成3个hashMap
    for (let i = 0; i < 9; i++) {
        for (let j = 0; j < 9; j++) {
            const boxIndex = Math.floor(i / 3) * 3 + Math.floor(j / 3);
            const value = board[i][j];
            if (value !== '.') {
                cols[i][value] = rows[j][value] = boxes[boxIndex][value] = true;
            }
        }
    }

    // 从第一个出现 . 的位置进行深搜
    for (let i = 0; i < 9; i++) {
        for (let j = 0; j < 9; j++) {
            if (board[i][j] === '.') {
                backtrack(i, j);
                return board;
            }
        }
    }
}

// 考虑坐标(x, y)
function backtrack(x, y) {
    // 递归边界出口
    if (x >= 9 || y >= 9) {
        return true;
    }

    // 九宫格编号
    const boxIndex = Math.floor(x / 3) * 3 + Math.floor(y / 3);
    for (let i = 1; i <= 9; i++) {
        if (!cols[i][value] && !rows[j][value] && !boxes[boxIndex][value]) {
            board[i][j] = value;
            if (backtrack(i, j)) {
                return true;
            }
            board[i][j] = '.';
        }
    }
}

```

```
if (cols[x][i] === 0 && rows[y][i] === 0 && boxes[boxIndex][i] === 0) {
    cols[x][i] = rows[y][i] = boxes[boxIndex][i] = 1;
    board[x][y] = `${i}`;
    const [nx, ny] = genNextPos(x, y);
    if (backtrack(nx, ny)) {
        return true;
    } else {
        // 回溯
        board[x][y] = '.';
        cols[x][i] = rows[y][i] = boxes[boxIndex][i] = 0;
    }
}
}

// 获取下一个待处理的坐标
function genNextPos(x, y) {
    const next = x * 9 + y + 1;
    const nx = Math.floor(next / 9);
    const ny = next % 9;
    return nx >= 9 || board[nx][ny] === '.' ? [nx, ny] : genNextPos(x, y + 1);
};
```

复杂度分析：

- 时间复杂度: $O(N!)$, 这里N是数独大小
- 空间复杂度: $O(N!)$, 这里N是数独大小

题目地址

<https://leetcode-cn.com/problems/search-insert-position>

题目内容

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

思路 - 暴力

根据题意暴力寻找第一个大于等于 target 的值

代码

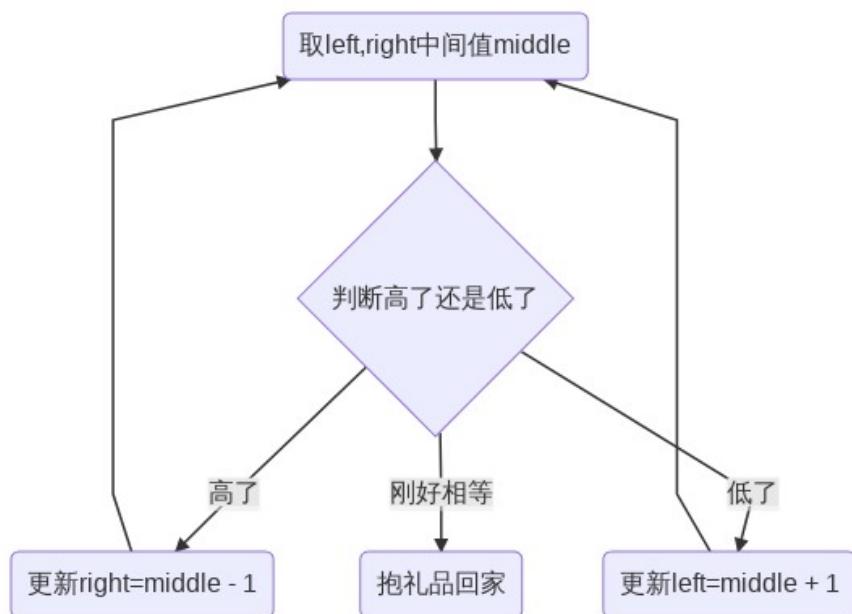
```
var searchInsert = function (nums, target) {
    for (let i = 0; i < nums.length; i++) {
        if (nums[i] >= target) {
            return i;
        }
    }
    return nums.length;
};
```

时间复杂度: $O(n)$ 空间复杂度: $O(1)$

思路 - 二分

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

```

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
  
```

```
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;
};
```

时间复杂度: $O(\log n)$ 空间复杂度: $O(1)$

题目地址

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

题目描述

编写一个高效的算法来判断 $m \times n$ 矩阵中，是否存在一个目标值。该矩阵具有如下特性：

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 = 5
输出: true

示例 2:

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

示例 3:

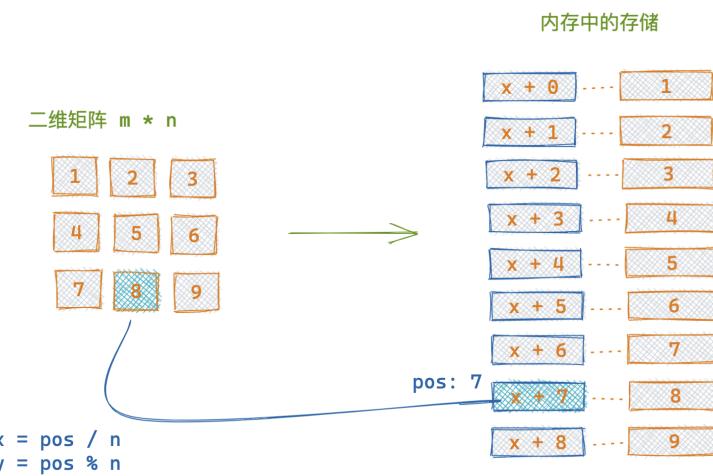
输入: matrix = [], target = 0
输出: false

提示:

```
m == matrix.length
n == matrix[i].length
0 <= m, n <= 100
-104 <= matrix[i][j], target <= 104
```

思路

前置知识：二维矩阵在内存中都是线性存储的，二维矩阵 $m \times n$ 某个点坐标 $[x, y]$ 在一维矩阵中的对应坐标为 $pos = x \cdot n + y$ ，相应的，如果知道一维矩阵坐标 pos ，我们可以得到其在二维矩阵中的坐标 $x = pos // n, y = pos \% n$



由题意知道，输入矩阵从左上角到右下角大小递增，也就是说如果二维矩阵拍平到一维矩阵，问题就转换为

给定一个有序数据，寻找值等于target的下标

看到熟悉的题，还等什么，二分法安排上

```

var searchMatrix = function (matrix, target) {
    if (!matrix.length) {
        return false;
    }
    const m = matrix.length;
    const n = matrix[0].length;
    let left = 0,
        right = m * n - 1;
    while (left <= right) {
        const mid = (left + right) >> 1;
        if (matrix[Math.floor(mid / n)][mid % n] === target) {
            return true;
        } else if (matrix[Math.floor(mid / n)][mid % n] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return false;
};

```

时间复杂度: $O(\log(n * m)) = O(\log n + \log m)$, n, m 分别为二维矩阵纵列大小
空间复杂度: $O(1)$

26.删除排序数组中的重复项

<https://leetcode-cn.com/problems/remove-duplicates-from-sorted-array/>

- 26.删除排序数组中的重复项
 - 题目描述
 - 双指针
 - 思路
 - 复杂度分析
 - 代码

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：

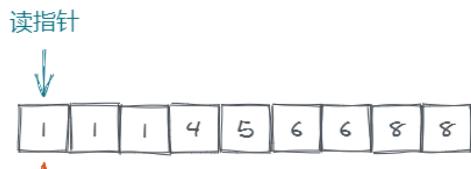
```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 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]);
}
```

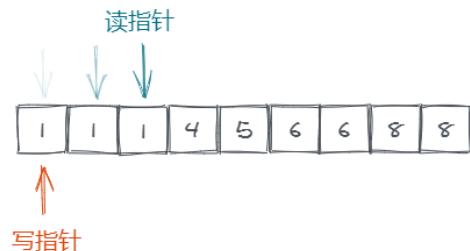
双指针

思路

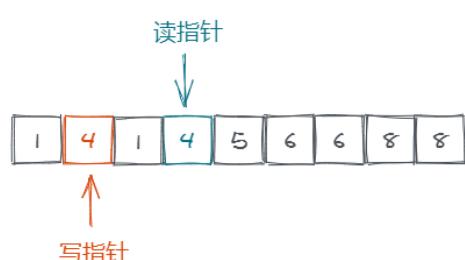
- 用一个读指针，一个写指针遍历数组。
- 遇到重复的元素 读指针 就继续前移。
- 遇到不同的元素 写指针 就前移一步，写入那个元素。



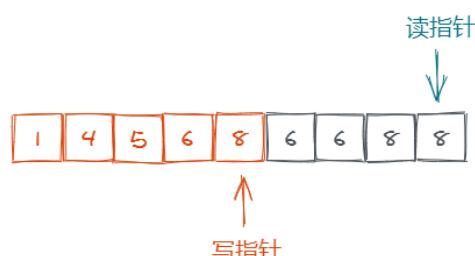
读、写指针一样时，
写指针原地不动
读指针继续往前走



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

复杂度分析

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

代码

JavaScript 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;
};
```

Python 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
```

题目地址

<https://leetcode-cn.com/problems/middle-of-the-linked-list/>

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。

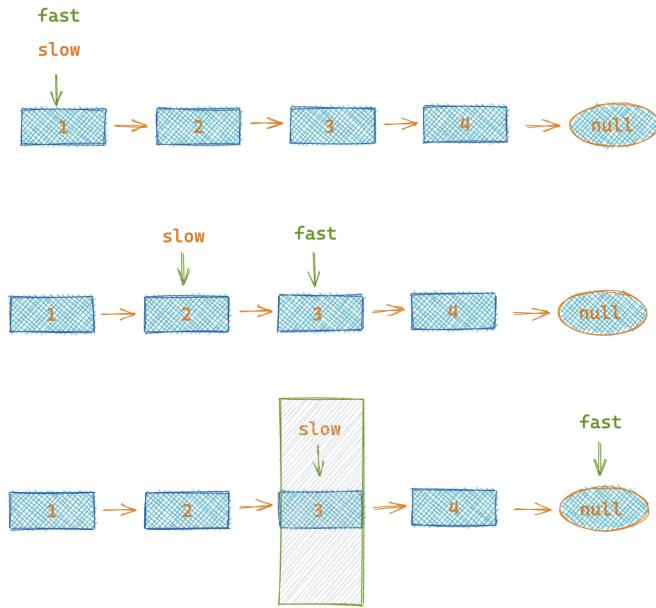
来源：力扣（LeetCode）

链接：<https://leetcode-cn.com/problems/middle-of-the-linked-list/>

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。

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。



代码

```
/*
 * @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;
};
```

时间复杂度: $O(n)$, n为链表长度 空间复杂度: $O(1)$

题目地址

<https://leetcode-cn.com/problems/grumpy-bookstore-owner/>

题目描述

今天，书店老板有一家店打算试营业 `customers.length` 分钟。每分钟都有一些顾客 (`customers[i]`) 会进入书店，所有这些顾客都会在那一分钟结束后离开。

在某些时候，书店老板会生气。如果书店老板在第 i 分钟生气，那么 $\text{grumpy}[i] = 1$ ，否则 $\text{grumpy}[i] = 0$ 。当书店老板生气时，那一分钟的顾客就会不满意，不生气则他们是满意的。

书店老板知道一个秘密技巧，能抑制自己的情绪，可以让自己连续 X 分钟不生气，但却只能使用一次。

请你返回这一天营业下来，最多有多少客户能够感到满意的数量。

示例：

输入: `customers = [1,0,1,2,1,1,7,5]`, `grumpy = [0,1,0,1,0,1,0,1]`
输出: 16
解释：
书店老板在最后 3 分钟保持冷静。
感到满意的最大客户数量 = $1 + 1 + 1 + 1 + 7 + 5 = 16$.

提示：

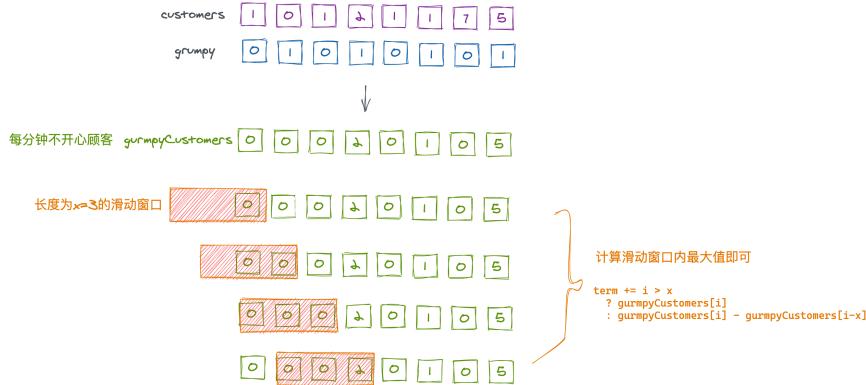
$1 \leq X \leq \text{customers.length} == \text{grumpy.length} \leq 20000$
 $0 \leq \text{customers}[i] \leq 1000$
 $0 \leq \text{grumpy}[i] \leq 1$

思路

如果老板没有大招，那么在给定顾客 `customers` 以及 生气时间 `grumpy`，满意顾客的数量是确定的

满意顾客数量 = 来访顾客总数 - 在生气时候来顾客总数

但是，现在老板有大招，可以使得区间长度为 X 的时间内不生气，现在要使得最多的客户能满意，根据贪心，我们只需要找到一个长度为 X 的区间 $[a, b]$ ，且 $[a, b]$ 内不满意客户数量即可。



代码

```

var maxSatisfied = function (customers, grumpy, X) {
    grumpy = customers.map((c, i) => c * grumpy[i])
    let maxGrumpSize = -Infinity
    let termGrumpSize = 0
    for (let i = 0; i < grumpy.length; i++) {
        termGrumpSize += i < X ? grumpy[i] : grumpy[i] - grumpy[i-X]
        maxGrumpSize = Math.max(maxGrumpSize, termGrumpSize)
    }

    return sum(customers) - sum(grumpy) + maxGrumpSize

    function sum(arr) {
        return arr.reduce((acc, cur) => acc + cur, 0)
    }
};

```

时间复杂度: $O(n)$, n 为`customers`长度

题目地址(239. 滑动窗口最大值)

<https://leetcode-cn.com/problems/sliding-window-maximum/>

题目描述

给定一个数组 `nums`, 有一个大小为 `k` 的滑动窗口从数组的最左侧移动到数组的最右侧，找出所有窗口中的最大值。

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

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

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  
}
```

整体代码如下

```
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;
}
```

时间复杂度: $O(n * k)$, n为nums长度

堆/优先队列

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(来自leetcode某题解)

```
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;
}
```

时间复杂度: $O(N * \log k)$

单调队列

类似于单调栈，这里我们对滑动窗口维护一个单调队列，队列保证数据从队头到队尾递减

```

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];
    }
}

```

js deque 实现可以参考[deque](#)

时间复杂度: \$O(n)\$

另外可以参考 [leetcode仓库的题解](#)

进阶篇 - 01. 高频面试题

【Day 34】 2020-12-04 - 树的遍历系列

- [官方题解](#)

【Day 35】 2020-12-05 - 反转链表系列

- [官方题解](#)

【Day 36】 2020-12-06 - 位运算系列

- [官方题解](#)

【Day 37】 2020-12-07 - 动态规划系列

- [官方题解](#)

【Day 38】 2020-12-08 - 有效括号系列

- [官方题解](#)

【Day 39】 2020-12-09 - 设计系列

- [官方题解](#)

【Day 40】 2020-12-10 - 前缀树系列

- [官方题解](#)

进阶篇 - 02. Trie

【Day 41】 2020-12-11 - 208. 实现 Trie

- [官方题解](#)

【Day 42】 2020-12-12 - 677. 键值映射

- [官方题解](#)

【Day 43】 2020-12-13 - 面试题 17.17. 多次搜索

- [官方题解](#)

进阶篇 - 03. 并查集

【Day44】 2020-12-14 - 547. 朋友圈

- [官方题解](#)

【Day 45】 2020-12-15 - 924. 尽量减少恶意软件的传播

- [官方题解](#)

【Day 46】 2020-12-16 - 1319. 连通网络的操作次数

- [官方题解](#)

进阶篇 - 04. 跳表

【Day 47】 2020-12-17 - 1206. 设计跳表

- [官方题解](#)

进阶篇 - 05. 剪枝

【Day 48】 2020-12-18 - 814. 二叉树剪枝

- [官方题解](#)

【Day 49】 2020-12-19 - 39. 组合总和

- [官方题解](#)

【Day 50】 2020-12-20 - 40. 组合总和 II

- [官方题解](#)

【Day 51】 2020-12-21 - 47. 全排列 II

- [官方题解](#)

进阶篇 - 06. 字符串匹配

【Day 52】 2020-12-22 - 28. 实现 strStr()-BF&RK

- [官方题解](#).md>

208. 实现 Trie (前缀树)

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.startsWith("app"); // 返回 false  
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

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

- Java

```
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;
        }
    }
}
```

```
        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, 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

```

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

```

```
for ch in prefix:  
    if ch not in node.children:  
        return False  
    node = node.children[ch]  
return node.preCount > 0
```

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。

677. 键值映射

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代成新的键值对。

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

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。

⇒ 暴力法

方法二：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，键值映射事儿 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。

⇒ Trie 解法

代码：

- 方法一 (Python) :

```
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
```

- 方法一 (Java) :

```
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;  
    }  
}
```

- 方法二 (Java) :

```

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

方法一：

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。

方法二：

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：

```
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;
```

```
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;  
    }  
}
```

最后，由于后面的写的代码行数越来越多，欢迎大家批评指正！！！

面试题 17.17 多次搜索

题目地址

<https://leetcode-cn.com/problems/multi-search-lcci>

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

于 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

```

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;

```

```
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

```

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();
        }
    }
}

```

```
    }

    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）

```
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

建树的时空复杂度请参照讲义

时间复杂度: $O(N \cdot K)$, 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!

547. 朋友圈

题目地址

<https://leetcode-cn.com/problems/friend-circles/>

题目描述

班上有 N 名学生。其中有些人是朋友，有些则不是。他们的友谊具有传递性。如果已知 A 是 B 的朋友，B 是 C 的朋友，那么我们可以认为 A 也是 C 的朋友。所谓的朋友圈，是指所有朋友的集合。

给定一个 $N \times N$ 的矩阵 M，表示班级中学生之间的朋友关系。如果 $M[i][j] = 1$ ，表示已知第 i 个和 j 个学生互为朋友关系，否则为不知道。你必须输出所有学生中的已知的朋友圈总数。

示例 1:

输入:

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

输出: 2

解释: 已知学生 0 和学生 1 互为朋友，他们在同一个朋友圈。

第2个学生自己在一个朋友圈。所以返回 2 。

示例 2:

输入:

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

输出: 1

解释: 已知学生 0 和学生 1 互为朋友，学生 1 和学生 2 互为朋友，所以学

提示:

```
1 <= N <= 200  
M[i][i] == 1  
M[i][j] == M[j][i]
```

题解

每个学生看作图中的一个节点，求朋友圈数目其实就是求图的强连通分量。这里提供三种题解

1. dfs
2. 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，直到遍历结束，连通图数目加一
3. 选取另外一个未遍历的节点，重复上述过程
4. 直到所有节点都被遍历
5. bfs，算法同上述 dfs，只是把图遍历方式从 dfs 改为 bfs
6. 并查集求强连通分量
7. 初始时，强连通分量为 count = M.length
8. MAKE-SET，将每个节点的 parent 指向其本身
9. FIND，并查集常规搜索，添加路径压缩
10. 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--

以下是上述算法的实现

DFS

```
/*
 * @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);
            }
        }
    }
};


```

BFS

```
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;
};
```

并查集

```
var findCircleNum = function (M) {
    let count = M.length;
    let parnets = Array.from(M).map((item, index) => index);
    function find(x) {
        if (parnets[x] === x) {
            return x;
        }
        return (parnets[x] = find(parnets[x]));
    }

    function union(x, y) {
        if (find(x) === find(y)) {
            return;
        }
        parnets[parnets[x]] = parnets[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;
};
```

题目地址

<https://leetcode-cn.com/problems/minimize-malware-spread>

题目内容

在节点网络中，只有当 $\text{graph}[i][j] = 1$ 时，每个节点 i 能够直接连接到另一个节点 j 。

一些节点 initial 最初被恶意软件感染。只要两个节点直接连接，且其中至少一个节点受到恶意软件的感染，那么两个节点都将被恶意软件感染。这种恶意软件的传播将继续，直到没有更多的节点可以被这种方式感染。

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终节点数。

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 $M(\text{initial})$ ，则返回该节点。如果有多个节点满足条件，就返回索引最小的节点。

请注意，如果某个节点已从受感染节点的列表 initial 中删除，它以后可能仍然因恶意软件传播而受到感染。

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

题解

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。

并查集

```

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];
        }
    }
});
}
);

```

```
    return res;  
};
```

DFS

```

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;
    }
}

```

```
for (let i = 0; i < N; i++) {
    if (graph[start][i] === 1 && colors[i] === 0) {
        dfs(i, color);
    }
}
};


```

1319. 连通网络的操作次数

题目地址

<https://leetcode-cn.com/problems/number-of-operations-to-make-network-connected/>

题目内容

1. 连通网络的操作次数 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 $n-1$ 。线缆用 `connections` 表示，其中 `connections[i] = [a, b]` 连接了计算机 a 和 b 。

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意一台计算机。

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机之间的线缆，并用它连接一对未直连的计算机。请你计算并返回使所有计算机都连通所需的最少操作次数。如果不可能，则返回 -1。

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

```
/*
 * @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
        }
    }
};
```

1206. 设计跳表

题目地址

<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. 可以想象跳表是一个网状结构，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

```

// 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;
    }
}

```

```

// 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;
};

```

814 二叉树剪枝

题目地址

<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

```
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 ? null : root;  
}
```

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。
- 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。

39 组合总和

题目地址

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

题目描述

给定一个无重复元素的数组 `candidates` 和一个目标数 `target`，找出 `candidates` 中的所有可能组合，使它们相加等于 `target`。

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, int target) {
    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嘛。
- 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

代码支持： Java

```

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);
    }
}

```

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

我发现 Leetcode 上国区很少讨论时空复杂度的，比较奇怪。。。

- 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 $O(N^{\lceil \frac{\text{target}}{\min} \rceil})$, 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： $O(\text{target}^2)$, 递归的空间复杂度上面分析过了。

40 组合总数 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

```
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[] candidates, int target, int cur) {
    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]);
        // 变化在下面这行呢
        helper(res, list, candidates, cur - candidates[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

```

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);
        }
    }
}

```

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： $O(2^N)$ ，其中 N 是数组长度，每个元素有选和不选两种可能。（`res.add` 这步操作应该是非 $O(1)$ 的，有兴趣小伙伴可以自行了解一下）

- 空间复杂度：递归栈空间复杂度同上，最终 res 数组复杂度：
 $O(target^2)$

47 全排列 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），那么我们就需要一个辅助数组 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）
 - $\text{nums}[i] == \text{nums}[i - 1] \&& \text{visit}[i - 1]$ ： 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 $\text{nums}[0]=1$ 还没用过，符合条件，故两个 1, 1, 2 只会存下来 1 个。
 - $\text{nums}[i] == \text{nums}[i - 1] \&& \text{!visit}[i - 1]$ ： 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

```
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);  
        }  
    }  
}
```

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 $O(N! op(res))$ ，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。