

对 未 来 最 大 的 慷 慨 , 就 是 把 一 切 都 献 给 现 在



算法 101

JavaScript 描述

目录

- 写在前面
- 学习指南
- 开篇——复杂度
- 字符串
 - 翻转整数、有效的字母异位词和翻转整数
 - 报数、反转字符串和字符串中的第一个唯一字符
 - 验证回文字符串、实现 strStr()、最长公共前缀和最长回文子串
- 数学
 - 罗马数字转整数、Fizz Buzz和计数质数
 - 3的幂、Excel表列序号、快乐数和阶乘后的零
 - Pow(x, n)、两数相除、分数到小数和x的平方根
- 数组
 - 旋转数组、只出现一次的数字、两数之和、旋转图像
 - 从排序数组中删除重复项、加一、买股票的最佳时机和移动零
 - 两个数组的交集、一周中的第几天、有效的数独、除资深以外数组的乘积和存在重复元素
 - 字谜分组、三数之和、无重复字符的最长子串、矩阵置零和递增的三元子序列
- 链表
 - 回文链表、环形链表、删除链表中的节点
 - 反转链表、删除链表的倒数第N个节点、合并两个有序链表和两数相加
 - 排序链表、相交链表和奇偶链表
- 二叉树
 - 最小栈、Shuffle an Array和将有序数组转换为二叉搜索树
 - 对称二叉树、二叉树的最大深度和验证二叉搜索树
 - 二叉树的层次遍历、二叉树的序列化与反序列化和常数时间内插入删除、获得随机数
 - 中序遍历二叉树、从前序与中序遍历序列构造二叉树和二叉搜索树中第 K 小的元素
 - 填充每个节点的下一个右侧节点指针、岛屿数量和二叉树的锯齿形层次遍历
- 动态规划
 - 最大子序和、爬楼梯和买卖股票的最佳时机
 - 打家劫舍、零钱兑换和跳跃游戏
 - 不同路径、Longest Increasing Subsequence和单词拆分
- 回溯算法
 - 括号生成、子集和电话号码的字母组合
 - 实现数组的全排列和单词搜索
- 排序与搜索
 - 合并两个有序数组、第一个错误的版本和搜索旋转排序数组
 - 在排序数组中查找元素的第一个和最后一个位置、数组中的第K个最大元素和颜色分类
 - 前 K 个高频元素、寻找峰值和合并区间

- 搜索二维矩阵 II和计算右侧小于当前元素的个数
- 栈和队列
 - 汉明距离、位 1 的个数、缺失数字
 - 有效的括号、帕斯卡三角形和颠倒二进制位
 - 两整数之和、数据流的中位数和逆波兰表达式
 - Task Scheduler、有序矩阵中第K小的元素和多数元素
- 结束篇

写在前面

作者介绍

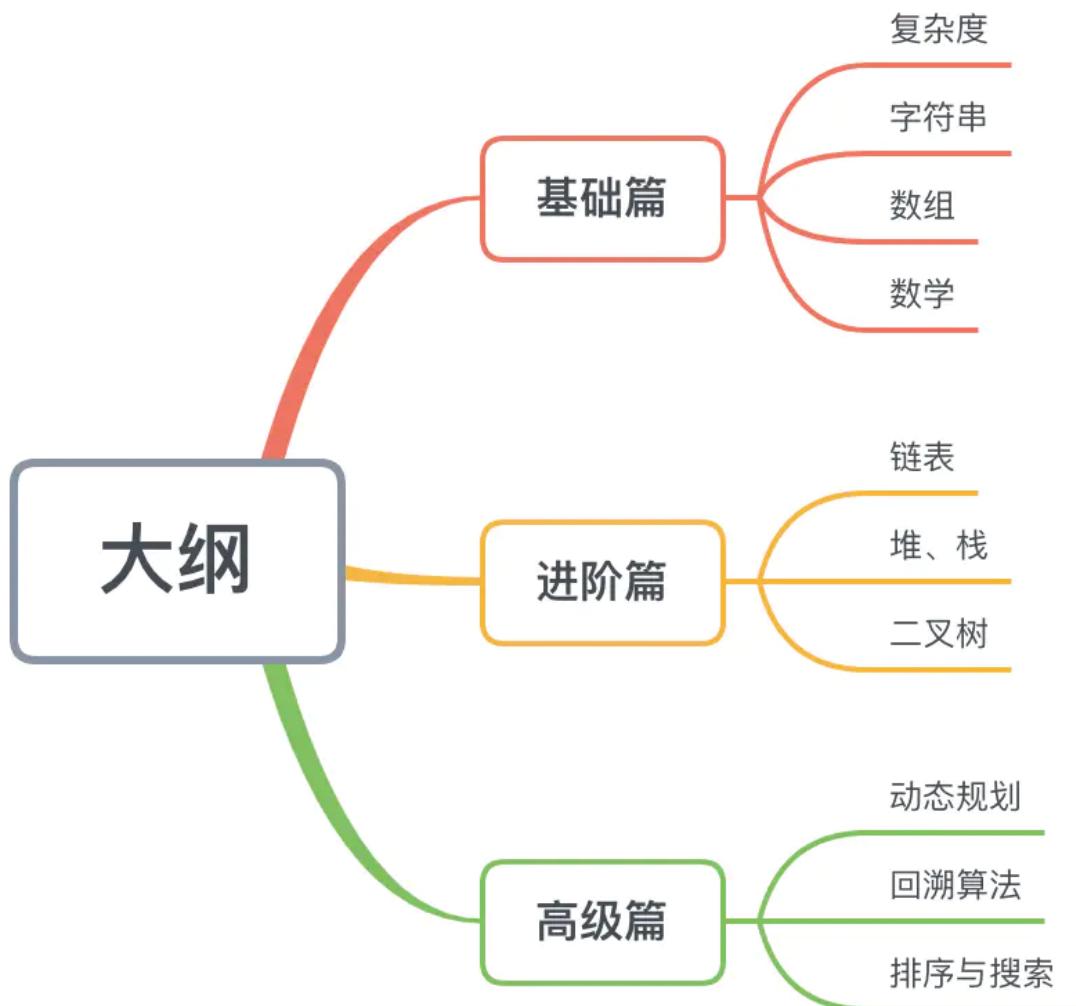
政采云前端团队（ZooTeam），一个年轻富有激情和创造力的前端团队，隶属于政采云产品研发中心，Base 在风景如画的杭州。团队现有 50 余个前端小伙伴，平均年龄 27 岁，近 3 成是全栈工程师，妥妥的青年风暴团。成员构成既有来自于阿里、网易的“老”兵，也有浙大、中科大、杭电等校的应届新人。团队在日常的业务对接之外，还在物料体系、工程平台、搭建平台、性能体验、云端应用、数据分析及可视化等方向进行技术探索和实战，推动并落地了一系列的内部技术产品，持续探索前端技术体系的新边界。

小册介绍

数据结构与算法是计算机专业必修课，但是对于前端工程师来说，沉浸在业务代码之中很少会和算法直接打交道，甚于说根本不需要用到什么算法。那么我们为什么要学习算法，意义何在？不会算法活不是一样能干。把一件事情做到极致是非常必要的职业心态，这离不开数据结构和算法。另一方面，再说面试，这和在学生时代为什么要学数理化是一个道理，考试要考，你就要学。面试造火箭，工作拧螺丝，面试官通过问几道算法题了解你的编程和逻辑思维能力并不奇怪。

万丈高楼平地起，基础知识掌握多少，一定程度上决定了我们的技术能走多远。想要作出一点事情，基础一定要扎实，要苦练“内功”。对于一名工程师来说，所谓的“内功”无非就是大学里所学的那些基础课程，计算机网络、数据结构与算法等。

本小册子由浅入深大致可以分为 3 个部分：



- 基础篇

这部分的内容先从考评算法的复杂度开始介绍，再从比较基础的字符串、数组入手，最后是一些数学相关的题目。让大家先从简单的内容上手，练好基本功，不要一上来就被算法吓到。

- 进阶篇

剖析稍复杂的数据结构与算法，再加上经典题目的实战练习，帮助你更加深入理解算法的精髓、提升算法思维，开始修炼更高深的“内功”。

- 高级篇

本部分来介绍一些比较高级的算法，可以理解为高深的“心法”，虽然难学，但是学会了之后会发现很管用，走遍天下都不怕。

我们不能解题不能一知半解，每道题每一种解法，都写上了解题思路和详细的解题步骤。让读者在解题的过程当中，更好地训练思维，知道答案是怎么来的，能够举一反三。

本小册子选取大厂面试高频算法题共计 101 道，大多数题目至少两种 JavaScript 解法，并附有详细的思路和解题过程，来帮你夯实、强化算法知识。

你会收获到什么？

- 更好的逻辑思维能力
- 对数据结构更深的理解
- 能够写出更加牛逼的代码
- 一份体面的工作

适宜人群

- 想看看机会的同学
- 一直想学习算法，出于某些原因没认真学的同学

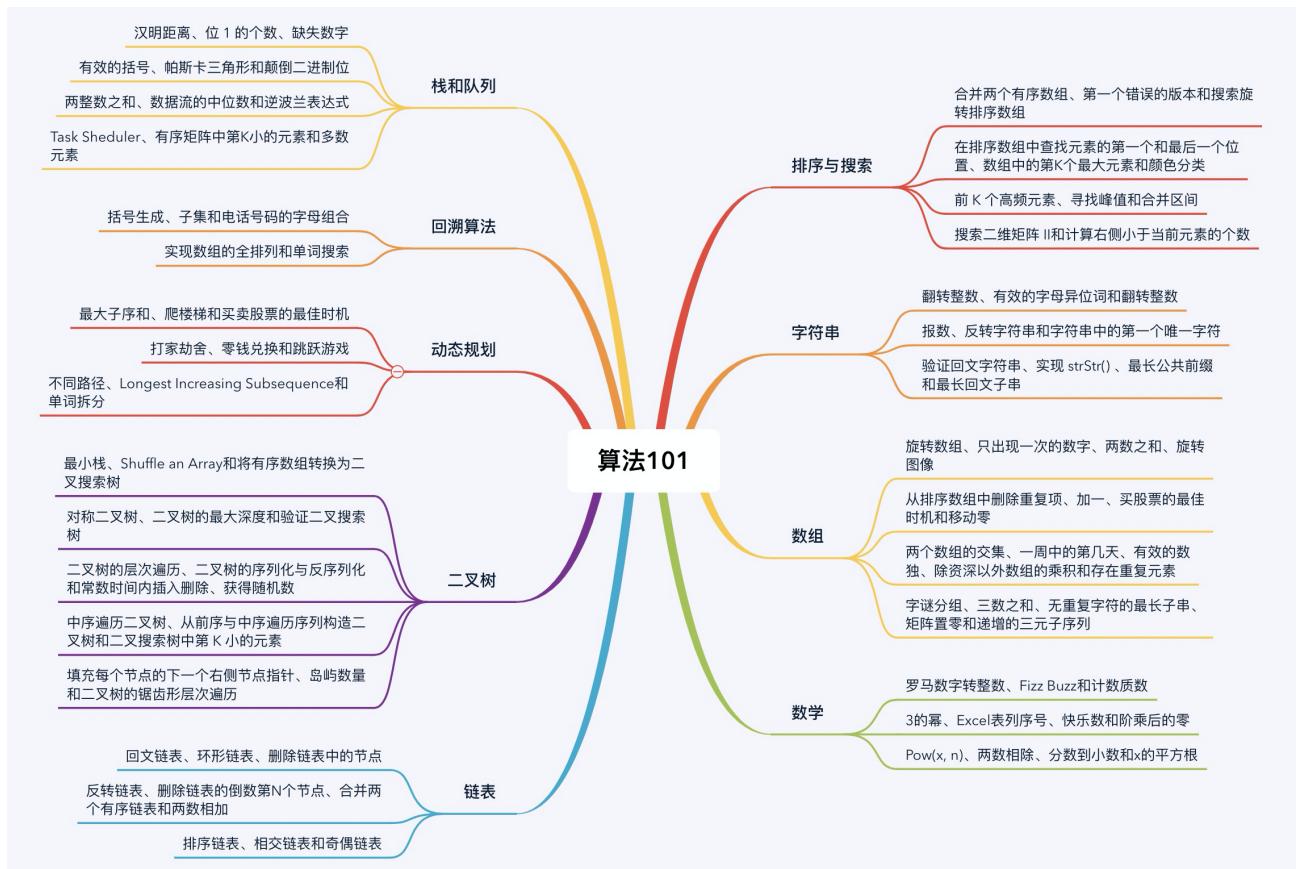
你需要准备什么

- JavaScript 语言基础
- 一台电脑一杯咖啡
- 一颗热爱学习的心

学习指南

本小册子的内容可能比较多，为了能更好地帮助到大家，在阅读小册子的时候能够有更好的体验，在这里梳理整个小册子的目录结构。

小册子共分为 9 大章节：



也希望各位读者可以通过脑图可以对小册子的内容有更加明确的认识，提升阅读的收益。

如果觉得自己水平比较好的，完全可以跳着看，挑选能帮助你提高的内容来阅读。

高效地学习

大家都经历过高考，其实刷算法题就和刷高考题一样。举个例子，如果你的立体几何已经掌握得非常不错了，而导数是弱项，再去做 100 道立体几何还不如 10 道导数的题目进步的快。经常做自己熟悉的题目，进步并不是很大，投入产出比不高。算法也是如此，需要经常去做你不会的题目，同时一个题目挖掘更多更高效的解法。如果你发现学习的时候脑壳很痛，那么恭喜你，你真的在进步。

一起变得更好

如果在阅读过程发现有什么错误，欢迎留言，我们会第一时间回复，也希望你的留言可以帮助到其他同学。

小册子的解题方法写得还是相对详细的，但是这个只是我们觉得。不要我们觉得，要大家觉得。毕竟每个人的基础不一样，大家在阅读过程当中如果有什么不明白的地方，想不明白的问题，欢迎在后台留言。

我们一直非常关注读者的阅读体验，如果觉得小册子的排版有哪里不够好，期待你提出宝贵的建议。

最后

数据结构和算法的学习是一个循序渐进的过程，如果可以仔细地阅读这本小册子，相信一定可以帮助到你。同时自己的思考和坚持很重要。好吧，说了那么多，还不赶快学习去。

开篇——复杂度

时间复杂度

时间复杂度是描述算法运行的时间。我们把算法需要运算的次数用输入大小为 n 的函数来表示，记作 $T(n)$ 。时间复杂度通常用 $\mathcal{O}(n)$ 来表示，公式为 $T(n) = \mathcal{O}(f(n))$ ，其中 $f(n)$ 表示每行代码的执行次数之和，注意是执行次数。

常见的时间复杂度

名称	运行时间 $T(n)$	时间举例	算法举例
常数	$\mathcal{O}(1)$	3	-
线性	$\mathcal{O}(n)$	n	操作数组
平方	$\mathcal{O}(n^2)$	n^2	冒泡排序
对数	$\mathcal{O}(\log(n))$	$\log(n)$	二分搜索

- $\mathcal{O}(1)$ 复杂度

算法执行所需要的时间不随着某个变量 n 的大小而变化，即此算法时间复杂度为一个常量，可表示为 $\mathcal{O}(1)$

直接上代码

```
1 const a = 1;
2 console.log(a);
```

```
1 const a = 1;
2 console.log(a, 1);
3 console.log(a, 2);
4 console.log(a, 2);
```

$\mathcal{O}(1)$ 表示常数级别的复杂度，不管你是 $\mathcal{O}(\text{几})$ ，统一给你计作 $\mathcal{O}(1)$

- $\mathcal{O}(n)$ 复杂度

```
1  for (let i = 0; i < n; i++) {  
2      // do something  
3 }
```

上面这段代码，写了一个 `for` 循环，从 0 到 n ，不管 n 是多少，都要循环 n 次，而且只循环 n 次，所以得到复杂度为 $\mathcal{O}(n)$

- $\mathcal{O}(n^2)$ 复杂度

```
1  for (let i = 0; i < n; i++) {  
2      for (let j = 0; j < n; j++) {  
3          // do something  
4      }  
5 }
```

上面的程序嵌套了两个循环，外层是 0 到 n ，内层基于每一个不同的 i ，也要从 0 到 n 执行，得到复杂度为 $\mathcal{O}(n^2)$ 。可以看出，随着 n 增大，复杂度会成平方级别增加。

- $\mathcal{O}(\log(n))$ 对数复杂度

```
1  for (let i = 1; i <= n; i *= 2) {  
2      // do something  
3 }
```

讲到这里顺便来复习一下高中数学知识，函数 $y = \log_a x$ 叫做对数函数， a 就是对数函数的底数。

对数复杂度是比较常见的一种复杂度，也是比较难分析的一种复杂度。观察上面的代码， i 从 1 开始，每循环一次就乘以 2，直到 i 大于 n 时结束循环。

$$2^1 --> 2^2 --> 2^3 \dots --> 2^x$$

观察上面列出 i 的取值发现，是一个等比数列，要知道循环了多少次，求出 x 的值即可。由 $2^x = n$ 得到， $x = \log_2 n$ ，所以这段代码的时间复杂度为 $\log_2 n$ 。

如果把上面的 `i *= 2` 改为 `i *= 3`，那么这段代码的时间复杂度就是 $\log_3 n$ 。

根据[换底公式](#)：

$$\log_c a * \log_a b = \log_c b$$

因此 $\log_3 n = \log_3 2 * \log_2 n$ ，而 $\log_3 2$ 是一个常量，得到 $\mathcal{O}(\log_3(n)) = \mathcal{O}(\log_2(n))$ 。所以，在对数时间复杂度的表示中，我们忽略对数的“底”，我不管你底数是多少，统一计作 $\mathcal{O}(\log(n))$ 。

递归的时间复杂度

在面试的时候，可能会写到一些递归的程序，那么递归的时间复杂度如何考虑？

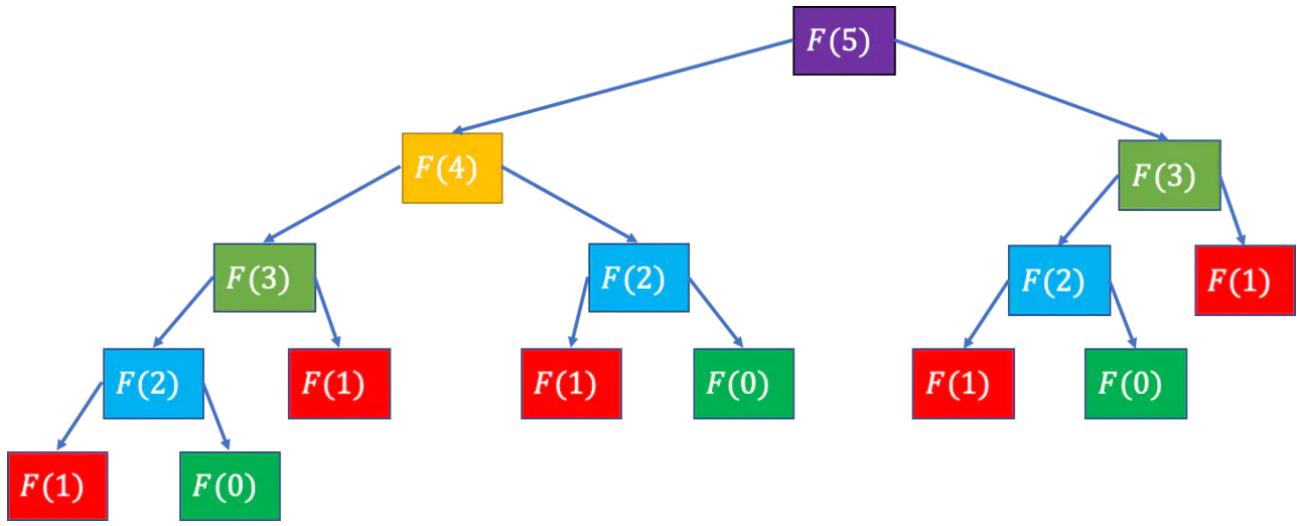
递归算法中，每个递归函数的时间复杂度为 $O(s)$ ，递归的调用次数为 n ，则该递归算法的时间复杂度为 $O(n) = n * O(s)$

我们先来看一个经典的问题，斐波那契数列(Fibonacci sequence)：

$$F(0) = 1, F(2) = 1, F(n) = F(n - 1) + F(n - 2) \quad (n \geq 2, n \in N*)$$

```
1 function fibonacci(n) {
2     if (n === 0 || n === 1) {
3         return 1;
4     }
5     return fibonacci(n - 1) + fibonacci(n - 2);
6 }
```

我们很容易写出上面这样一段递归的代码，往往会忽略了时间复杂度是多少，换句话说调用多少次。可以代一个数进去，例如 $n = 5$ ，完了之后大概就能理解递归的时间复杂度是怎么来的。



上图把 $n = 5$ 的情况都列举出来。可以看出，虽然代码非常简单，在实际运算的时候会有大量的重复计算。

在 n 层的完全二叉树中，节点的总数为 $2^n - 1$ ，所以得到 $F(n)$ 中递归数目的上限为 $2^n - 1$ 。因此我们可以毛估出 $F(n)$ 的时间复杂度为 $\mathcal{O}(2^n)$ 。

时间复杂度为 $\mathcal{O}(2^n)$ ，指数级的时间复杂度，显然不是最优的解法，让计算机傻算了很多次，所以在面试时要稍微留意，如果写出这样的代码，可能会让你的面试官不太满意。

空间复杂度

空间复杂度是对算法运行过程中临时占用空间大小的度量，一个算法所需的存储空间用 $f(n)$ 表示，可得出 $S(n) = \mathcal{O}(f(n))$ ，其中 n 为问题的规模， $S(n)$ 表示空间复杂度。通常用 $S(n)$ 来定义。

常见的空间复杂度

- $\mathcal{O}(1)$ 复杂度

算法执行所需要的临时空间不随着某个变量 n 的大小而变化，即此算法空间复杂度为一个常量，可表示为 $\mathcal{O}(1)$

```

1 const a = 1;
2 const b = 2;
3 console.log(a);
4 console.log(b);

```

以上代码，分配的空间不会随着处理数据量的变化而变化，因此得到空间复杂度为 $\mathcal{O}(1)$

- $\mathcal{O}(n)$ 复杂度

先来看这样一段代码

```
1 const arr = new Array(n);
2 for (let i = 0; i < n; i++) {
3     // do something
4 }
```

上面这段代码的第一行，申请了长度为 n 的数组空间，下面的 for 循环中没有分配新的空间，可以得出这段代码的时间复杂度为 $\mathcal{O}(n)$ 。

对数阶的空间复杂度非常少见，而且空间复杂度的分析相对与时间复杂度分析简单很多，这部分不再阐述。

时间空间相互转换

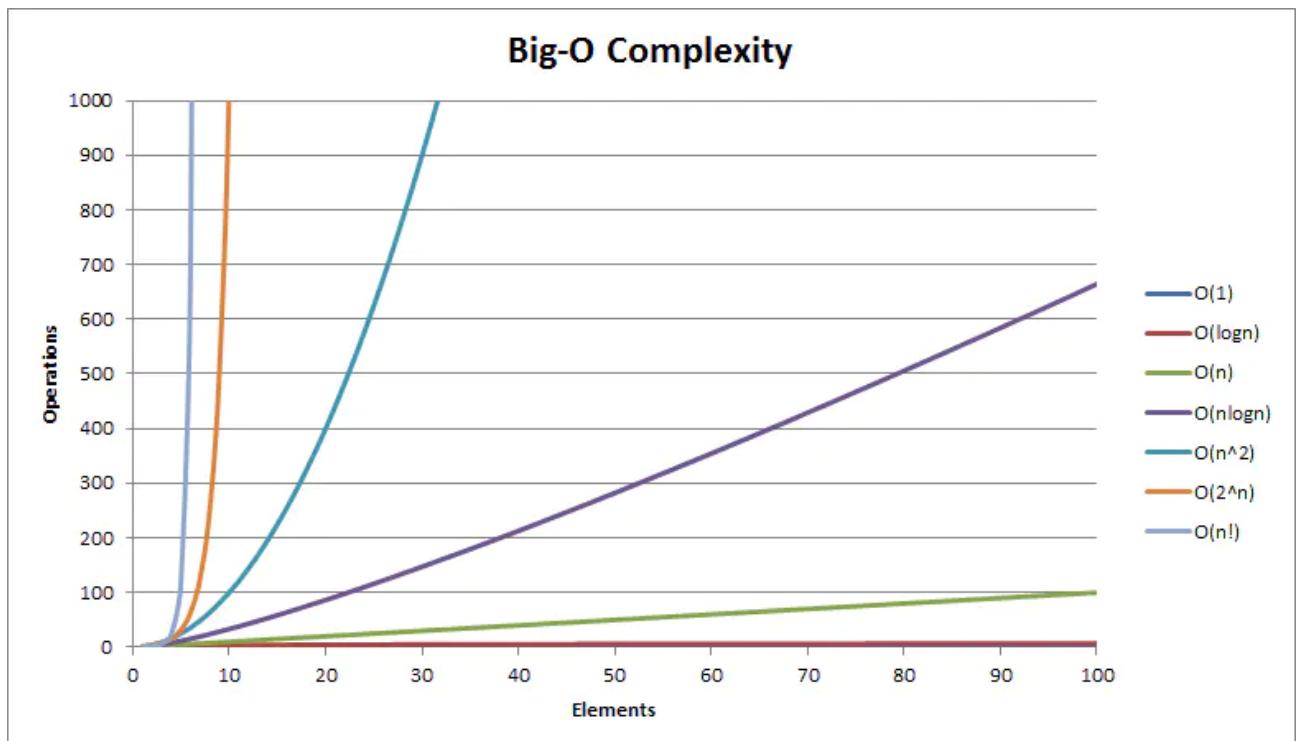
对于一个算法来说，它的时间复杂度和空间复杂度往往是相互影响的。

那我们熟悉的 Chrome 来说，流畅性方面比其他厂商好了多人，但是占用的内存空间略大。

当追求一个较好的时间复杂度时，可能需要消耗更多的储存空间。 反之，如果追求较好的空间复杂度，算法执行的时间可能就会变长。

总结

常见的复杂度不多，从低到高排列就这么几个： $\mathcal{O}(1)$ 、 $\mathcal{O}(\log(n))$ 、 $\mathcal{O}(n)$ 、 $\mathcal{O}(n^2)$ ，等学完后面的章节你会发现，复杂度基本上逃不走，都是上面这几个。



字符串

字符串

在计算机中，字符串是由零个或多个字符组成的有限序列。字符串也是 `JavaScript` 中最基本的数据类型，学习字符串也是学习编程的基础。

说到字符串，我相信你肯定很熟悉了，是不是觉得很简单。接下来看题。

本章节分为 3 个部分：

- Part 1
 - 翻转数组
 - 有效的字母异位词
 - 字符串翻转整数
- Part 2
 - 报数
 - 反转字符串
 - 字符串中的第一个唯一字符
- Part 3
 - 验证回文字符串
 - 实现 `strStr()`
 - 最长公共前缀
 - 最长回文子串

阅读完本章节，你将有以下收获：

- 熟悉 `JavaScript` 中的基本操作
- 能够熟练解决字符串一些较基础的问题

翻转整数、有效的字母异位词和翻转整数

翻转整数

给出一个 32 位的有符号整数，你需要将这个整数中每位上的数字进行反转。

示例

```
1  示例 1:  
2  
3  输入: 123  
4  输出: 321  
5  示例 2:  
6  
7  输入: -123  
8  输出: -321  
9  示例 3:  
10  
11 输入: 120  
12 输出: 21
```

注意:

假设我们的环境只能存储得下 32 位的有符号整数，则其数值范围为 $[-2^{31}, 2^{31} - 1]$ 。请根据这个假设，如果反转后整数溢出那么就返回 0。

方法一 翻转字符串方法

思路

如果将数字看成是有符号位的字符串，那么我们就能够通过使用 JS 提供的字符串方法来实现非符号部分的翻转，又因为整数的翻转并不影响符号，所以我们最后补充符号，完成算法。

详解

1. 首先设置边界极值；
2. 使用字符串的翻转函数进行主逻辑；
3. 补充符号
4. 然后拼接最终结果

代码

```
1  /**
2  * @param {number} x
3  * @return {number}
4  */
5 const reverse = (x) => {
6    // 非空判断
7    if (typeof x !== 'number') {
8      return;
9    }
10   // 极值
11   const MAX = 2147483647;
12   const MIN = -2147483648;
13
14   // 识别数字剩余部分并翻转
15   const rest =
16     x > 0
17       ? String(x)
18         .split('')
19         .reverse()
20         .join('')
21       : String(x)
22         .slice(1)
23         .split('')
24         .reverse()
25         .join('');
26
27   // 转换为正常值，区分正负数
28   const result = x > 0 ? parseInt(rest, 10) : 0 - parseInt(rest, 10);
29
30   // 边界情况
31   if (result >= MIN && result <= MAX) {
32     return result;
33   }
34   return 0;
35 };
```

复杂度分析

- 时间复杂度： $O(n)$

代码中 `reverse` 函数时间复杂度为 $O(n)$ ， n 为整数长度，因此时间复杂度为 $O(n)$ ，考虑到32位整数最大长度为 11，即 -2147483648，也可认为是常数时间复杂度 $O(1)$ 。

- 空间复杂度： $O(n)$

代码中创建临时 `String` 对象， n 为整数长度，因此空间复杂度为 $O(n)$ ，考虑到32位整数最大长度为11，即-2147483648，因此空间复杂度为 $O(1)$ 。

方法二 类似 欧几里得算法 求解

思路

我们借鉴欧几里得求最大公约数的方法来解题。符号的处理逻辑同方法一，这里我们通过模 10 取到最低位，然后又通过乘 10 将最低位迭代到最高位，完成翻转。

详解

1. 设置边界极值；
2. 取给定数值的绝对值，遍历循环生成每一位数字，借鉴欧几里得算法，从 num 的最后一位开始取值拼成新的数
3. 同步剔除掉被消费的部分
4. 如果最终结果为异常值，则直接返回 0；如果原本数据为负数，则对最终结果取反
5. 返回最终结果

代码

```
1  /**
2   * @param {number} x
3   * @return {number}
4   */
5  const reverse = (x) => {
6    // 获取相应数的绝对值
7    let int = Math.abs(x);
8    // 极值
9    const MAX = 2147483647;
10   const MIN = -2147483648;
11   let num = 0;
12
13   // 遍历循环生成每一位数字
14   while (int !== 0) {
15     // 借鉴欧几里得算法，从 num 的最后一位开始取值拼成新的数
16     num = (int % 10) + (num * 10);
17     // 剔除掉被消费的部分
18     int = Math.floor(int / 10);
19   }
20   // 异常值
21   if (num >= MAX || num <= MIN) {
22     return 0;
23   }
24   if (x < 0) {
25     return num * -1;
26   }
27   return num;
28 };
```

复杂度分析：

- 时间复杂度： $O(n)$

代码中使用 for 循环，次数为 n ，即整数的长度，因此时间复杂度为 $O(n)$ 。

- 空间复杂度： $O(1)$

算法中只用到常数个变量，因此空间复杂度为 $O(1)$ 。

有效的字母异位词

给定两个字符串 s 和 t，编写一个函数来判断 t 是否是 s 的字母异位词。

示例1

```
1 输入: s = "anagram", t = "nagaram"  
2 输出: true
```

示例2

```
1 输入: s = "rat", t = "car"  
2 输出: false
```

方法一 利用数组sort()方法

思路

首先，对字符串字母进行排序，然后，比较两字符串是否相等。

详解

1. 首先，将字符串转为数组。
2. 利用数组 sort 方法进行排序。
3. 然后，转为字符串进行比较，如果相等返回 true，反之返回 false。

代码

```
1 const isAnagram = (s, t) => {
2   const sArr = s.split('');
3   const tArr = t.split('');
4   const sortFn = (a, b) => {
5     return a.charCodeAt() - b.charCodeAt();
6   };
7   sArr.sort(sortFn);
8   tArr.sort(sortFn);
9   return sArr.join('') === tArr.join('');
10};
```

复杂度分析

- 时间复杂度： $O(n \log n)$

`JavaScript` 的 `sort` 方法的实现原理，当数组长度小于等于 10 的时候，采用插入排序，大于 10 的时候，采用快排，快排的平均时间复杂度是 $O(n \log n)$ 。

- 空间复杂度： $O(n)$ 算法中申请了 2 个数组变量用于存放字符串分割后的字符串数组，所以数组空间长度跟字符串长度线性相关，所以为 $O(n)$ 。

方法二 计数累加方法

思路

声明一个对象记录字符串每个字母的个数，另外一个字符串每项与得到的对象做匹配，最后，根据计数判断是否相等。

详解

- 首先，声明一个变量，遍历其中一个字符串 `s` 或 `t`，对每个字母出现的次数进行累加。
- 然后，遍历另一个字符串，使每一个字母在已得到的的对象中做匹配，如果匹配则对象下的字母个数减 1，如果匹配不到，则返回 `false`，如果最后对象中每个字母个数都为 0，则表示两字符串相等。

代码

```
1 const isAnagram = (s, t) => {
2   if (s.length !== t.length) {
3     return false;
4   }
5   const hash = {};
6   for (const k of s) {
```

```
7     hash[k] = hash[k] || 0;
8     hash[k] += 1;
9 }
10    for (const k of t) {
11        if (!hash[k]) {
12            return false;
13        }
14        hash[k] -= 1;
15    }
16    return true;
17 };
```

复杂度分析

- 时间复杂度： $O(n)$

算法中使用了 2 个单层循环，因此，时间复杂度为 $O(n)$ 。

- 空间复杂度： $O(1)$

申请的变量 `hash` 最大长度为 256，因为 Ascii 字符最多 256 种可能，因此，考虑为常量空间，即 $O(1)$ 。

字符串转换整数

`atoi` (表示 ascii to integer) 是把字符串转换成整型数的一个函数，实现一个 `atoi` 函数，使其能将字符串转换成整数。

首先，该函数会根据需要丢弃无用的开头空格字符，直到寻找到第一个非空格的字符为止。

当我们寻找到的第一个非空字符为正或者负号时，则将该符号与之后面尽可能多的连续数字组合起来，作为该整数的正负号；假如第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形成整数。

该字符串除了有效的整数部分之后也可能会存在多余的字符，这些字符可以被忽略，它们对于函数不应该造成影响。

注意：假如该字符串中的第一个非空格字符不是一个有效整数字符、字符串为空或字符串仅包含空白字符时，则你的函数不需要进行转换。

在任何情况下，若函数不能进行有效的转换时，请返回 0。

示例

示例 1:

```
1 输入: "42"
2 输出: 42
```

示例 2:

```
1 输入: "-42"
2 输出: -42
```

示例 3:

```
1 输入: "4193 with words"
2 输出: 4193
```

示例 4:

```
1 输入: "words and 987"
2 输出: 0
```

示例 5:

```
1 输入: "-91283472332"
2 输出: -2147483648
```

解释: 数字 "-91283472332" 超过 32 位有符号整数范围。因此返回 INT_MIN (-2147483648)。

说明 :

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。

方法一 正则匹配

思路

第一步，使用正则提取满足条件的字符，`/^(-|\+)?\d+/g`，`(-|\+)?` 表示第一位是-或+或都不是，`\d+` 表示匹配多个数字

```
const result = str.trim().match(/^(-|\+)?\d+/g);
```

第二步，判断目标是否超过 Int 整形最大值或最小值

```
1 return result
2 ? Math.max(Math.min(Number(result[0]), Math.pow(2,31)-1), -Math.pow(2,31))
3 : 0;
```

代码

```
1 /**
2  * @param {string} str
3  * @return {number}
4 */
5 const myAtoi = function (str) {
6   // 提取需要的字符
7   const result = str.trim().match(/^(-|\+)?\d+/g);
8   return result
9   ? Math.max(Math.min(Number(result[0]), Math.pow(2, 31) - 1), -Math.pow(2, 31))
10  : 0;
11};
```

复杂度分析

- 时间复杂度： $O(1)$

上述解法中，代码在执行的时候，它消耗的时间并不随着某个变量的增长而增长，因此时间复杂度为 $O(1)$

- 空间复杂度： $O(1)$

上述解法中，额外所分配的空间都不随着处理数据量变化，所以空间复杂度为 $O(1)$

方法二 逐个判断

思路

第一步，去除字符串之中的空格

```
const news = str.trim();
```

第二步，通过执行 parseInt 判断是否为数字，不是数字返回 0，是数组继续解析

```
1 if(parseInt(news)){
2     return retrunNum(parseInt(news));
3 } else {
4     return 0;
5 }
```

第三步，判断目标是否超过 Int 整形最大值或最小值

```
1 const retrunNum = function (num) {
2     if (num >= -Math.pow(2, 31) && num <= Math.pow(2, 31) - 1) {
3         return num;
4     } else {
5         return num > 0 ? Math.pow(2, 31) - 1 : -Math.pow(2, 31);
6     }
7};
```

代码

```
1 /**
2  * @param {string} str
3  * @return {number}
4 */
5 const myAtoi = function (str) {
6     const news = str.trim();
7     if (parseInt(news)) {
8         return retrunNum(parseInt(news));
9     } else {
10        return 0;
```

```
11      }
12  };
13 const retrunNum = function (num) {
14   if (num >= -Math.pow(2, 31) && num <= Math.pow(2, 31) - 1) {
15     return num;
16   } else {
17     return num > 0 ? Math.pow(2, 31) - 1 : -Math.pow(2, 31);
18   }
19};
```

复杂度分析

- 时间复杂度： $O(1)$

上述解法中，代码在执行的时候，它消耗的时间并不随着某个变量的增长而增长，因此时间复杂度为 $O(1)$

- 空间复杂度： $O(1)$ 上述解法中，额外所分配的空间都不随着处理数据量变化，所以空间复杂度为 $O(1)$

报数、反转字符串和字符串中的第一个唯一字符

报数

报数序列是一个整数序列，按照其中的整数的顺序进行报数，得到下一个数。其前五项如下：

```
1 1. 1
2 2. 11
3 3. 21
4 4. 1211
5 5. 111221
```

1 被读作 "one 1" ("一个一")，即 11。 11 被读作 "two 1s" ("两个一")，即 21。 21 被读作 "one 2"，"one 1" ("一个二", "一个一")，即 1211。

给定一个正整数 n ($1 \leq n \leq 30$)，输出报数序列的第 n 项。

注意：整数顺序将表示为一个字符串。 **示例**

```
1 输入: 1
2 输出: "1"
```

```
1 输入: 4
2 输出: "1211"
```

方法一 递归

想要获取第 n 项的结果，需要先获取到第 $n-1$ 项的结果，然后报出第 $n-1$ 项的结果做为第 n 项的结果。所以可以采用递归调用法。

```
1 const countAndSay = function (n) {
2   if (n === 1) {
3     return '1';
4   }
5   const preResult = countAndSay(n - 1); // 获取第 n-1 项的结果。
```

```
6     /**
7      * \d 匹配一个数字
8      * \1 匹配前面第一个括号内匹配到的内容
9      * (\d)\1* 匹配相邻数字相同的内容
10     * 使用replace方法将匹配到的内容处理为长度 + 内容的第一个字符
11     * 结果为所求报数
12   */
13   return preResult.replace(/(\d)\1*/g, item => `${item.length}${item[0]}`);
14 };
```

复杂度分析

- 时间复杂度： $O(n)$
本算法涉及递归，代码的调用次数为 n 次。故而时间复杂度为 $O(n)$ 。
- 空间复杂度： $O(n)$
递归算法，调用次数随 n 增加而成线性增加，每次调用申明变量数相同。故而空间复杂度为 $O(n)$ 。

方法二 循环法

递归法是由 n 到 1 计算相应的值并层层返回的，循环法正好相反，循环法由 1 计算到 n 。然后将最终值返回。

```
1 const countAndSay = function (n) {
2   let result = '1'; // 第一个数为'1'
3   for (let i = 1; i < n; i++) { // 循环获取知道第 n 项。
4     // 同方法一
5     result = result.replace(/(\d)\1*/g, item => `${item.length}${item[0]}`);
6   }
7   return result;
8 };
```

复杂度分析

- 时间复杂度： $O(n)$
本算法代码的调用次数为 n 次。故而时间复杂度为 $O(n)$ 。
- 空间复杂度： $O(1)$
申明对象数量为固定值。空间复杂度为常量 $O(1)$ 。

反转字符串

编写一个函数，其作用是将输入的字符串反转过来。输入字符串以字符数组 `char[]` 的形式给出。

不要给另外的数组分配额外的空间，你必须原地修改输入数组、使用 $O(1)$ 的额外空间解决这一问题。

你可以假设数组中的所有字符都是 ASCII 码表中的可打印字符。

示例1

```
1 输入 : ["h", "e", "l", "l", "o"]
2 输出 : ["o", "l", "l", "e", "h"]
```

示例2

```
1 输入 : ["H", "a", "n", "n", "a", "h"]
2 输出 : ["h", "a", "n", "n", "a", "H"]
```

方法一 首尾替换法

思路

首尾替换法，逐位遍历，进行交换

详解

1. 设置变量 `i = 0`；
2. 替换字符串的第`i`位和倒数第 `i` 位，替换方式：使用es6的解构赋值进行变量的交换；
3. 变量 `i + 1`，继续替换替换字符串的第 `i` 位和倒数第 `i` 位；
4. 直到 `i` 大于字符串s的长度的中位数，完成整个字符串的反转

```
1 /**
2  * @param {character[]} s
3  * @return {void} Do not return anything, modify s in-place instead.
4 */
```

```
5 const reverseString = function (s) {
6   for (let i = 0; i < s.length / 2; i++) {
7     [s[i], s[s.length - 1 - i]] = [s[s.length - 1 - i], s[i]];
8   }
9   return s;
10};
```

复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$
没有新开辟的内存空间

方法二 中间变量首尾替换法

思路

中间变量首尾替换法，逐位遍历，进行交换

详解

1. 设置变量 `i = 0`；
2. 替换字符串的第`i`位和倒数第`i`位，替换方式：设置一个中间变量，替换两个字符串的值；
3. 变量 `i + 1`，继续替换替换字符串的第`i`位和倒数第`i`位；
4. 直到`i`大于字符串`s`的长度的中位数，完成整个字符串的反转

```
1 /**
2  * @param {character[]} s
3  * @return {void} Do not return anything, modify s in-place instead.
4 */
5 const reverseString = function (s) {
6   for (let i = 0; i < s.length / 2; i++) {
7     const a = s[i];
8     s[i] = s[s.length - i - 1];
9     s[s.length - i - 1] = a;
10  }
11};
```

复杂度分析

- 时间复杂度： $O(n)$

- 遍历次数：如果字符串长度为 n ， n 是偶数，遍历次数位 $n/2$ ，如果 n 是奇数，遍历次数为 $(n + 1)/2$
- 空间复杂度： $O(1)$
 - 1个临时变量

字符串中的第一个唯一字符

给定一个字符串，找到它的第一个不重复的字符，并返回它的索引。如果不存在，则返回 -1。

示例

```
1 s = "leetcode"
2 返回 0.
3
4 s = "loveleetcode",
5 返回 2.
```

注意事项：您可以假定该字符串只包含小写字母。

方法一 库函数

思路

某个字符从头开始开始的索引和从尾开始找的索引如果相等，就说明这个字符只出现了一次

详解

- 从头到尾遍历一遍字符串；
- 判断每个位置的字符的 `index()` 和 `lastIndexOf()` 的结果是否相等；

代码

```
1 /**
2  * @param {string} s
3  * @return {number}
4  */
5 const firstUniqChar = function (s) {
6   for (let i = 0; i < s.length; i += 1) {
7     if (s.indexOf(s[i]) === s.lastIndexOf(s[i])) {
8       return i;
9     }
10  }
```

```
9      }
10     }
11     return -1;
12   };
```

复杂度分析

- 时间复杂度： $O(n^2)$

外层遍历，时间复杂度为 $O(n)$ ，调用 `indexOf` 的复杂度为 $O(n)$ ，得出总的时间复杂度为 $O(n^2)$

- 空间复杂度： $O(1)$

因为除了临时变量 `i`，没有开辟额外的空间

方法二 哈希

思路

遍历两次。第一次遍历，用一个哈希对象记录所有字符的出现次数；第二次遍历，找出哈希对象中只出现一次的字符的下标

详解

- 第一次遍历，用一个哈希对象记录所有字符的出现次数；
- 第二次遍历，找出哈希对象中只出现一次的字符的下标；

代码

```
1  /**
2   * @param {string} s
3   * @return {number}
4   */
5 const firstUniqChar = function (s) {
6   const hash = {};
7   for (let i = 0; i < s.length; i += 1) {
8     if (!hash[s[i]]) {
9       hash[s[i]] = 1;
10    } else {
11      hash[s[i]] += 1;
12    }
13  }
14  for (let i = 0; i < s.length; i += 1) {
15    if (hash[s[i]] === 1) {
16      return i;
```

```
17      }
18  }
19  return -1;
20 };
```

复杂度分析

- 空间复杂度： $O(1)$

因为变量只有 `hash` 和 `i`，开辟空间大小不随输入的变量变化

- 时间复杂度： $O(n)$

因为有两次遍历，且每次遍历都只有一层没有嵌套，所以遍历的次数只和入参字符串 `s` 的长度线性正相关

验证回文字符串、实现 strStr()、最长公共前缀和最长回文子串

验证回文串

给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。

说明：本题中，我们将空字符串定义为有效的回文串。

示例 1：

- ```
1 输入: "A man, a plan, a canal: Panama"
2 输出: true
```

### 示例 2：

- ```
1 输入: "race a car"
2 输出: false
```

方法一

思路

首先，去除字符串中的非字母和数字，再将字符串转换为数组，再对数组首尾一一比较，即可得出结果。

详解

1. 将传入的字符串，利用 `toLowerCase()` 方法统一转化为小写，再利用正则表达式 `/[^A-Za-z0-9]/g` 在字符串中去除非字母和数字，最后将字符串转换为数组。
2. 转换数组后，利用循环一一比较元素，先比较第一个和最后一个，再比较第二个和倒数二个，依次类推，若中间有不相等则不是回文串，反之，则是回文串。

代码

```

1  /**
2   * @param {string}
3   * @return {boolean}
4  */
5  const isPalindrome = (s) => {
6    // 将传入的字符串,统一转化为小写,同时去除非字母和数字,在转换为数组
7    const arr = s.toLowerCase().replace(/\[^A-Za-z0-9]/g, '').split('');
8    let i = 0;
9    let j = arr.length - 1;
10   // 循环比较元素
11   while (i < j) {
12     // 从首尾开始,一一比较元素是否相等
13     if (arr[i] === arr[j]) {
14       // 若相等,即第二个元素和倒数第二个元素继续比较,依次类推
15       i += 1;
16       j -= 1;
17     } else {
18       // 只要有一个相对位置上不相等,既不是回文串
19       return false;
20     }
21   }
22   // 是回文串
23   return true;
24 };

```

复杂度分析

- 时间复杂度： $O(n)$ 该解法中 while 循环最多执行 $n/2$ 次，即回文时，因此，时间复杂度为 $O(n)$ 。
- 空间复杂度： $O(n)$ 该解法中，申请了 1 个大小为 n 的数组空间，因此，空间复杂度为 $O(n)$ 。

方法二

思路

首先，去除字符串中的非字母和数字，然后，利用数组将字符串翻转，再和原字符串进行比较，即可得到结果。

详解

- 将传入的字符串，利用 `toLowerCase()` 方法统一转化为小写，再利用正则表达式 `/[^A-Za-z0-9]/g` 在字符串中去除非字母和数字，得到字符串 `arr`。
- 将字符串 `arr` 转换为数组，利用数组的方法反转数组，再将数组转为字符串 `newArr`。

3. 将字符串 arr 和字符串 newArr 进行比较，相等即为回文串，不相等则不为回文串。

代码

```
1  /**
2   * @param {string} s
3   * @return {boolean}
4   */
5  const isPalindrome = (s) => {
6    // 方便比较,统一转化为小写,并去除非字母和数字
7    const arr = s.toLowerCase().replace(/[^A-Za-z0-9]/g, '');
8    // 将新字符串转换为数组,利用数组的方法获得反转的字符串
9    const newArr = arr.split('').reverse().join('');
10   // 将2个字符进行比较得出结果
11   return arr === newArr;
12 };
```

复杂度分析

- 时间复杂度： $O(n)$

该解法中，`toLowerCase()`，`replace()`，`split()`，`reverse()`，`join()` 的时间复杂度都为 $O(n)$ ，且都在独立的循环中执行，因此，总的时间复杂度依然为 $O(n)$ 。

- 空间复杂度： $O(n)$

该解法中，申请了 1 个大小为 n 的字符串和 1 个大小为 n 的数组空间，因此，空间复杂度为 $O(n * 2)$ ，即 $O(n)$ 。

实现`strStr()`

给定一个 haystack 字符串和一个 needle 字符串，在 haystack 字符串中找出 needle 字符串出现的第一个位置 (从0开始)。如果不存在，则返回 -1。

以下称 haystack 字符串为匹配字符串，needle 字符串为查找字符串

示例

```
1  给定 haystack = 'hello world', needle = 'll'
2
3  返回2
```

说明:

当 needle 是空字符串时，我们应当返回什么值呢？这是一个在面试中很好的问题。

对于本题而言，当 needle 是空字符串时我们应当返回 0。这与C语言的 strstr() 以及 Java 的 indexOf() 定义相符。

方法一 遍历截取字符串对比

思路

截取字符串对比的思路很简单，从匹配字符串 haystack 中截取出与需查找字符串 needle 长度相等的内容后，对比截取的内容与匹配字符串是否相等，如果相等返回开始截取的下标。

详解

首先处理几个特殊场景

1. needle 的长度为0，直接返回0
2. needle 的字符串长度大于 haystack，肯定不匹配
3. needle 的字符串长度等于 haystack，判断是否相等，相等则匹配否则不匹配

剩下的就是 needle 字符串长度小于 haystack 的情况，遍历 haystack

此处需要注意的是，当 haystack 剩余字符串长度小于 needle 长度时，肯定是不相等，无需再次比较。

在遍历中判断 将要截取的字符串的首位与 needle 字符串的首位是否相同，如果不相同也就不需要后续截取、比较，跳过该次循环

代码

```
1 const strStr = function (haystack, needle) {  
2     const hayLen = haystack.length;  
3     const nedLen = needle.length;  
4  
5     if (!needle) {  
6         return 0;  
7     } if (nedLen > hayLen) {  
8         return -1;  
9     } else if (nedLen === hayLen) {  
10        return haystack === needle ? 0 : -1;  
11    } else {
```

```

12     for (let index = 0; index <= hayLen - nedLen; index++) {
13         if (haystack[index] !== needle[0]) {
14             continue;
15         }
16         if (haystack.substring(index, index + nedLen) === needle) {
17             return index;
18         }
19     }
20 }
21 return -1;
22 };

```

复杂度分析：

- 时间复杂度： $O(n)$

遍历长度可能从 1 到 $n - 1$ ，假设不同长度出现的概率均等，那么时间复杂度为 $(n - 1 + 1)/2$ 时间复杂度即为 $O(n)$ 。

- 空间复杂度： $O(1)$

使用 2 个额外存储空间。

方法二 双层循环对比字符

思路

循环对比字符串思路也很简单，从匹配字符串 haystack 的不同位置开始遍历，判断其中是否含有查找字符串 needle。

如：haystack 为 hello，needle 为 ll，依次判断 he、el、ll、lo 是否完全和 ll 相等，相等即返回对应字符串在 haystack 中的下标。

详解

首先处理特殊边际情况，这块与第一种方法相同，就不再赘述。

以下为算法步骤：

1. 设置最外层循环，遍历次数为 0 - haystack 长度减去 needle 的长度。剩余字符串长度小于 needle 长度时，肯定不匹配
2. 判断匹配字符串 haystack 中该次循环使用到的字符串首尾字母是否与查找字符串 needle 首尾字母相同。
 - 不相等，直接跳过继续遍历。
 - 相等，执行第三步。

3. 判断查找字符串 needle 的长度

- 长度为 1，表明匹配成功，直接返回当前长字符串下标即可
- 长度大于 1，执行第四步

4. 遍历对比字符串，循环判断匹配字符串 haystack 不同位置的字符是否与匹配字符串 needle 对应位置的字符相等

- 不相等时，跳出循环，进行下次循环。
- 到最后一位还未跳出循环表明完全匹配，返回当前遍历次数（即查找字符串在匹配字符串中首次出现的位置）

代码

```
1 const strStr = function (haystack, needle) {
2     const hayLen = haystack.length;
3     const nedLen = needle.length;
4
5     if (!needle) {
6         return 0;
7     } if (nedLen > hayLen) {
8         return -1;
9     } else if (nedLen === hayLen) {
10        return haystack === needle ? 0 : -1;
11    } else {
12        for (let hasIndex = 0; hasIndex <= hayLen - nedLen; hasIndex++) {
13            if (
14                haystack[hasIndex] === needle[0] &&
15                haystack[hasIndex + nedLen - 1] === needle[nedLen - 1]
16            ) {
17                if (nedLen === 1) {
18                    return hasIndex;
19                }
20                for (let nedIndex = 1; nedIndex < nedLen; nedIndex++) {
21                    if (haystack[hasIndex + nedIndex] !== needle[nedIndex]) {
22                        break;
23                    }
24                    if (nedIndex === nedLen - 1) {
25                        return hasIndex;
26                    }
27                }
28            }
29        }
30    }
31    return -1;
32};
```

复杂度分析

- 时间复杂度： $O(n^2)$

假设长字符串长度为无限大的 n ，那么对比字符串长度最大为 $n - 1$ ，那么就需要对比 $(n - 1) * n = n^2 - n$ 次。当 n 趋近无限大时， n^2 要远远大于 n ，因此忽略减数 n ，那么时间复杂度为 $O(n^2)$

- 空间复杂度： $O(1)$

使用 2 个额外存储空间

最长公共前缀

编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串 ""。

示例

```
1 输入: ["flower", "flow", "flight"]
2 输出: "fl"
```

方法一 递归迭代

思路

查找 n 个字符串的最长公共前缀，可以拆分成两步：1. 查找前 $n-1$ 个字符串的最长公共前缀 m 2. 查找 m 与最后一个字符串的公共前缀

因此，我们可以得到递归公式：

```
$longestCommonPrefix([S1, S2, ..., Sn]) = findCommPrefix(longestCommonPrefix([S1, S2, ..., Sn-1]), Sn)$
```

我们只需要实现 `findCommPrefix` 方法，然后遍历数组即可。

详解

- 获取数组中第一个字符串，当做最长公共前缀保存到变量 `commonPrefix`
- 从数组中取出下一个字符串，与当前的最长公共前缀 `commonPrefix` 对比，得到新的最长公共前缀存到 `commonPrefix`

3. 重复第 2 步遍历完整个字符串，最后得到的即使数组中所有字符串的最长公共前缀

代码

```
1  /**
2   * @param {string[]} strs
3   * @return {string}
4  */
5 const longestCommonPrefix = function (strs) {
6   function findCommonPrefix (a, b) {
7     let i = 0;
8     while (i < a.length && i < b.length && a.charAt(i) === b.charAt(i)) {
9       i++;
10    }
11    return i > 0 ? a.substring(0, i) : '';
12  }
13  if (strs.length > 0) {
14    let commonPrefix = strs[0];
15    for (let i = 1; i < strs.length; i++) {
16      commonPrefix = findCommonPrefix(commonPrefix, strs[i]);
17    }
18    return commonPrefix;
19  }
20  return '';
21};
```

复杂度分析

- 时间复杂度： $O(n)$

最坏的情况下，所有字符串都是相同的。那么会将所有字符串的所有字符串都遍历比较一次这样就会进行 n 次字符比较，其中 n 是输入数据中所有字符数量。最好的情况下，所有的字符串都不一样，那么每个字符串只会访问一次，复杂度是 n , n 即数组长度。

- 空间复杂度： $O(1)$ ，除了保存当前公共前缀外无需其他存储空间。

方法二 循环迭代

思路

最长公共前缀一定是数组中所有数组都包含的前缀子串，我们可以将任意字符串的前缀作为公共前缀，从长度 0 到 n (n 为该字符串长度)，横向扫描数组中的所有字符串，看是否都有该前缀，直到找到不满足的为止。

详解

- 先假设最长公共子串的长度为 1，存到变量 i 。以第一个字符串为基准，取它的第 i 个字符与数组中其他所有的字符串第 i 个字符进行比较，如果都相等，那么将最长公共子串的长度加 1，否则停止查找，已找到最长公共前缀的长度，设置完成匹配标记 $flag$ 为 $false$
- 重复第 1 步，直到 i 等于第一个字符串的长度，或者匹配标记 $flag$ 为 $false$
- 返回第一个字符串的前 i 个字符，即为当前数组的最长公共前缀

代码

```
1  /**
2   * @param {string[]} strs
3   * @return {string}
4  */
5 const longestCommonPrefix = function (strs) {
6  if (strs.length === 0) {
7    return '';
8  }
9  let i = 0;
10 let flag = true;
11 while (flag) {
12  if (strs[0].length > i) {
13    const char = strs[0].charAt(i);
14    for (let j = 1; j < strs.length; j++) {
15      if (strs[j].length <= i || strs[j].charAt(i) !== char) {
16        flag = false;
17        break;
18      }
19    }
20  } else {
21    flag = false;
22  }
23  i++;
24 }
25 return strs[0].substring(0, i - 1);
26};
```

复杂度分析：

- 时间复杂度： $O(n)$
 - 空间复杂度： $O(1)$
- 整个过程只需要存储匹配标志。

最长回文子串

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

示例

- ```
1 输入: "babad"
2 输出: "bab"
3 注意: "aba" 也是一个有效答案。
```

## 方法一 动态规划法

### 思路

动态规划的思想，是希望把问题划分成相关联的子问题；然后从最基本的子问题出发来推导较大的子问题，直到所有的子问题都解决。

根据字符串的长度，建立一个矩阵 dp，通过不同情况的判断条件，通过  $dp[i][j]$  表示  $s[i]$  至  $s[j]$  所代表的子串是否是回文子串。

### 详解

1. 建立矩阵 dp
2. 循环遍历字符串，取得不同长度的子串
3. 不同长度的子串，根据不同的条件进行判断是否为回文子串
  - (1) 长度为 1，一定回文
  - (2) 长度为 2 或 3，判断首尾是否相同： $s[i] === s[j]$
  - (3) 长度 > 3，首尾字符相同，且去掉首尾之后的子串仍为回文： $(s[i] === s[j]) \&& dp[i + 1][j - 1]$
4. 取得长度最长的回文子串

```
1 /**
2 * @param {string} s
3 * @return {string}
4 */
5 const longestPalindrome = function (s) {
6 const dp = [];
7 for (let i = 0; i < s.length; i += 1) {
8 dp[i] = [];
9 }
10 let max = -1; let str = '';
11 for (let l = 0; l < s.length; l += 1) {
12 // l 为所遍历的子串长度 - 1，即左下标到右下标的长度
13 for (let i = 0; i + l < s.length; i += 1) {
```

```

14 const j = i + l;
15 // i为子串开始的左下标，j为子串开始的右下标
16 if (l === 0) {
17 // 当子串长度为1时，必定是回文子串
18 dp[i][j] = true;
19 } else if (l <= 2) {
20 // 长度为2或3时，首尾字符相同则是回文子串
21 if (s[i] === s[j]) {
22 dp[i][j] = true;
23 } else {
24 dp[i][j] = false;
25 }
26 } else {
27 // 长度大于3时，若首尾字符相同且去掉首尾之后的子串仍为回文，则为回文子串
28 if ((s[i] === s[j]) && dp[i + 1][j - 1]) {
29 dp[i][j] = true;
30 } else {
31 dp[i][j] = false;
32 }
33 }
34 if (dp[i][j] && l > max) {
35 max = l;
36 str = s.substring(i, j + 1);
37 }
38 }
39 }
40 return str;
41 };

```

## 复杂度分析

- 时间复杂度： $O(n^2)$  遍历次数取决于字符串的长度，因为是两层循环嵌套，所以遍历的最大次数为  $n^2$ 。
- 空间复杂度： $O(n)$  需要申请空间为字符串长度  $n$  的数组来记录不同长度子串的情况。

## 方法二 中心扩展

### 思路

回文子串一定是对称的，所以我们可以每次选择一个中心，然后从中心向两边扩展判断左右字符是否相等。

中心点的选取有两种情况：

当长度为奇数时，以单个字符为中心；

当长度为偶数时，以两个字符之间的空隙为中心。

## 详解

- 1.循环遍历字符串取得不同长度的子串
- 2.通过定义好的中心扩展方法，选取奇数对称和偶数对称的中心
- 3.通过比较选择出两种组合较大的回文子串长度，然后对比之前的长度，判断是否更新起止位置
- 4.全部遍历完成后，根据最后的起止位置的值，截取最长回文子串

```
1 /**
2 * @param {string} s
3 * @return {string}
4 */
5 const longestPalindrome = function (s) {
6 if (s == null || s.length < 1) {
7 return '';
8 }
9 let start = 0; let end = 0;
10 // 从中心向两边扩展
11 const expandFromCenter = (s, left, right) => {
12 while (left >= 0 && right < s.length && s[left] === s[right]) {
13 left -= 1;
14 right += 1;
15 }
16 return right - left - 1;
17 };
18 for (let i = 0; i < s.length; i += 1) {
19 // 中心的两种选取（奇对称和偶对称）
20 const len1 = expandFromCenter(s, i, i);
21 const len2 = expandFromCenter(s, i, i + 1);
22 // 两种组合取最大的回文子串长度
23 const len = Math.max(len1, len2);
24 // 如果此位置为中心的回文数长度大于之前的长度，则进行处理
25 if (len > end - start) {
26 start = i - Math.floor((len - 1) / 2);
27 end = i + Math.floor(len / 2);
28 }
29 }
30 return s.substring(start, end + 1);
31 };
```

## 复杂度分析

- 时间复杂度： $O(n^2)$

遍历次数取决于字符串的长度，因为是两层循环嵌套，所以遍历的最大次数为  $n^2$ 。

- 空间复杂度： $O(1)$

只使用到常数个临时变量，与字符串长度无关。

# 数学

在做算法时，可能需要运用一些数学知识。数学也是计算机的基础，这部分章节的内容相对于后面的章节比较轻松。在实际的面试中，用到的数学知识大纲为初中，极少可能超纲到高中，所以大家不用过于担心。

本章节分为 3 个部分：

- Part 1
  - 罗马数字转整数
  - Fizz Buzz
  - 计数质数
- Part 2
  - 3的幂
  - Excel表序列号
  - 快乐数
  - 阶乘后的零
- Part 3
  - Pow(x, n)
  - 两数相除
  - 分数到小数
  - x的平方根

# 罗马数字转整数、Fizz Buzz和计数质数

## 罗马数字转整数

罗马数字包含以下七种字符：I，V，X，L，C，D 和 M。

分别对应的数值为：1，5，10，50，100，500，1000。

例如，罗马数字 3 写做 III，即为三个并列的 1。12 写做 XII，即为 X+II。26 写做 XXVI，即为 XX+V+I。

通常情况下，不能出现超过连续三个相同的罗马数字并且罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况：

- 1 I 可以放在 V(5) 和 X(10) 的左边，来表示 4 和 9。
- 2 X 可以放在 L(50) 和 C(100) 的左边，来表示 40 和 90。
- 3 C 可以放在 D(500) 和 M(1000) 的左边，来表示 400 和 900。

给定一个罗马数字，将其转换成整数。输入确保在 1 到 3999 的范围内。

## 示例

- ```
1 输入："III"
2 输出：3
3
4 输入："IV"
5 输出：4
6
7 输入："LVIII"
8 输出：58
9
10 输入："MCMXCIV"
11 输出：1994
```

方法一 遍历

思路

先遍历特殊值，如果有特殊值，先累加特殊值，然后用正则去掉特殊值，再遍历剩余的数字。

代码

```
1 const romanToIntOne = function (num) {
2   const roman = {
3     IV: 4,
4     IX: 9,
5     XL: 40,
6     XC: 90,
7     CD: 400,
8     CM: 900
9   };
10  const list = {
11    I: 1,
12    V: 5,
13    X: 10,
14    L: 50,
15    C: 100,
16    D: 500,
17    M: 1000
18  };
19  let result = 0;
20  // 先遍历特殊值
21  for (const key in roman) {
22    // 检测输入值是否含有特殊值
23    if (num.includes(key)) {
24      // 用正则去掉特殊值
25      const reg = new RegExp(key);
26      num = num.replace(reg, '');
27      result += roman[key];
28    }
29  }
30  for (const i of num) {
31    // 累加正常罗马数
32    result += list[i];
33  }
34  return result;
35};
```

复杂度分析

- 时间复杂度： $O(n)$

上述解法中，每个数字都被遍历了一次，时间复杂度跟数字的个数 n 线性相关，因此为 $O(n)$ 。

- 空间复杂度： $O(1)$

方法二 switch + includes 法

思路

先遍历所有罗马数字进行累加，对于特殊数字的循环，比如： $5+1=6$ ，而实际是 4，相差 2，所以需要在结果上减去 2，以此类推。

代码

```
1 const romanToIntTwo = function (num) {
2     let result = 0;
3     for (const c of num) {
4         switch (c) {
5             case 'I':
6                 result += 1;
7                 break;
8             case 'V':
9                 result += 5;
10                break;
11            case 'X':
12                result += 10;
13                break;
14            case 'L':
15                result += 50;
16                break;
17            case 'C':
18                result += 100;
19                break;
20            case 'D':
21                result += 500;
22                break;
23            case 'M':
24                result += 1000;
25                break;
26        }
27    }
28    // 减去特殊组合
29    if (num.includes('IV') || num.includes('IX')) result -= 2;
30    if (num.includes('XL') || num.includes('XC')) result -= 20;
31    if (num.includes('CD') || num.includes('CM')) result -= 200;
32    return result;
33};
```

复杂度分析

- 时间复杂度： $O(n)$

- 空间复杂度： $O(1)$

Fizz Buzz

写一个程序，输出从 1 到 n 数字的字符串表示。

1. 如果 n 是 3 的倍数，输出“Fizz”；
2. 如果 n 是 5 的倍数，输出“Buzz”；
3. 如果 n 同时是 3 和 5 的倍数，输出 “FizzBuzz”。

示例

```
1  n = 15,
2
3  返回:
4  [
5      "1",
6      "2",
7      "Fizz",
8      "4",
9      "Buzz",
10     "Fizz",
11     "7",
12     "8",
13     "Fizz",
14     "Buzz",
15     "11",
16     "Fizz",
17     "13",
18     "14",
19     "FizzBuzz"
20 ]
```

方法一 遍历

思路

很简单，只需要判断 1 - n 的每个数字是否能被 3、5、15 整除，输出对应的字符串即可。

详解

1. 第一步，申请一个数组 arr，用于存放每个数字转换后字符串。

2. 第二步，循环遍历 1-n 的每个数字。如果该数字能被15整除（即取余为0），则该数字对应的字符串为 "FizzBuzz"；如果能被3整除，则为 "Fizz"；如果能被5整除，则为 "Buzz"；否则，为该数字即可。

代码

```
1  /**
2   * @param {number} n
3   * @return {string[]}
4  */
5  const fizzBuzz = (n) => {
6    const arr = [];
7    for (let i = 1; i <= n; i += 1) {
8      if (i % 15 === 0) { // 被15整除
9        arr.push('FizzBuzz');
10     } else if (i % 3 === 0) { // 被3整除
11       arr.push('Fizz');
12     } else if (i % 5 === 0) { // 被5整除
13       arr.push('Buzz');
14     } else {
15       arr.push(i.toString());
16     }
17   }
18   return arr;
19};
```

复杂度分析

- 时间复杂度： $O(n)$

上述解法中，每个数字都被遍历了一次，时间复杂度跟数字的个数 n 线性相关，因此为 $O(n)$ 。

- 空间复杂度： $O(n)$

上述解法中，申请了大小为 n 的数组空间，空间复杂度跟数字的个数 n 线性相关，因此为 $O(n)$ 。

方法二：字符串累加

思路

这种方法也很简单，因为 15 的倍数输出 `FizzBuzz`，正好是 3 的倍数输出的 `Fizz` 拼接上 5 的倍数输出的 `Buzz`，所以只需要单独写 2 个 if 判断，将字符串拼接即可。

详解

1. 第一步，申请一个数组 arr，用于存放每个数字转换后字符串。
2. 第二步，循环遍历 1-n 的每个数字。定义一个空字符串 str，用于临时存放字符串拼接的结果。
如果能被3整除，则 str 追加字符串 "Fizz"；如果能被5整除，则 str 追加字符串 "Buzz"；同时能被3和5整除的话，str 的值就为 "FizzBuzz" 了；否则，为该数字即可。

代码

```
1  /**
2   * @param {number} n
3   * @return {string[]}
4  */
5 const fizzBuzz = (n) => {
6   const arr = [];
7   for (let i = 1; i <= n; i += 1) {
8     let str = '';
9     if (i % 3 === 0) {
10       str += 'Fizz';
11     }
12     if (i % 5 === 0) {
13       str += 'Buzz';
14     }
15     if (i % 3 !== 0 && i % 5 !== 0) {
16       str += i;
17     }
18     arr.push(str);
19   }
20   return arr;
21 };
```

复杂度分析

- 时间复杂度： $O(n)$

上述解法中，每个数字都被遍历了一次，时间复杂度跟数字的个数 n 线性相关，因此为 $O(n)$ 。

- 空间复杂度： $O(n)$

上述解法中，申请了大小为 n 的数组空间，空间复杂度跟数字的个数 n 线性相关，因此为 $O(n)$ 。

计数质数

统计所有小于非负整数 n 的质数的数量。

示例

```
1 输入: 10
2 输出: 4
3 解释: 小于 10 的质数一共有 4 个, 它们是 2, 3, 5, 7 。
```

方法一 暴力法

思路

首先回顾质数的定义，质数是指在大于 1 的自然数中，除了 1 和它本身以外不再有其他因数的自然数。所以，我们可以根据定义直接从 2 开始直到 n 根据定义判断每一个数字是否为质数。

详解

1. 首先我们定义一个方法 `isPrime` 用于判断一个自然数是否为质数，根据乘法交换律，判断其是否有因子的边界为 n 的平方根即可。
2. 循环从 2 到 n 判断是否为质数，将数量存入 `count` 计数器中。

代码

```
1 function isPrime (n) {
2     // 判断是否为质数
3     if (n === 2 || n === 3) {
4         return true;
5     }
6     if (n % 6 !== 1 && n % 6 !== 5) {
7         return false;
8     }
9     const sqrtN = Math.sqrt(n); // 根据乘法交换律，判断边界为平方根即可
10    for (let i = 3; i <= sqrtN; i += 2) {
11        if (n % i === 0) {
12            return false;
13        }
14    }
15    return true;
16 }
17
18 function countPrimes (n) { // 返回质数数量
19     let count = 0;
20     for (let i = 2; i < n; i++) {
```

```
21     if (isPrime(i)) { // 循环判断
22         count++;
23     }
24 }
25 return count;
26 }
```

复杂度分析

- 时间复杂度： $O(n * \sqrt{n})$ 外层需要判断 n 个数是否质数，判断为质数是需要进行 \sqrt{n} 次计算
- 空间复杂度： $O(1)$ 使用了常数个变量，即为 $O(1)$ 。

方法二 埃拉托斯特尼筛法

思路

给出要筛选数值的范围 n ，找出 \sqrt{n} 以内的素数 p_1, p_2, \dots, p_k 。先用 2 去筛，即把 2 留下，把 2 的倍数剔除掉；再用下一个素数，也就是 3 筛，把 3 留下，把 3 的倍数剔除掉；接下去用下一个素数 5 筛，把 5 留下，把 5 的倍数剔除掉；不断重复下去……不断的剔除不需要比对的元素；每计算一个数，都要把它的倍数去掉。到了 n ，数一下留下了几个数。

详解

- `Uint8Array` 数组类型表示一个8位无符号整型数组，创建时内容被初始化为 0。创建完后，可以以对象的方式或使用数组下标索引的方式引用数组中的元素；
- `arr` 用来记录“已经找过的数的倍数”。内层循环中，一次把找过数的倍数，对应的 `arr` 下标元素设置为 `true`，这样外循环时不会计数；
- 外层循环用来计数，如果 `arr` 数组对应值是 `false`，即表示为质数，则计数器 `count` 加一，最终获取所有质数数量。

代码

```
1 const countPrimes = function (n) {
2     let count = 0;
3     const arr = new Uint8Array(n);
4     for (let i = 2; i < n; i++) {
5         if (!arr[i - 1]) {
6             count++;
```

```
7         for (let j = i * i; j <= n; j += i) {
8             arr[j - 1] = true;
9         }
10    }
11 }
12 return count;
13 };
```

复杂度分析

- 时间复杂度： $O(n \log \log n)$

对每一个 i ，要划掉 n/i 个数，要进行 n/i 次运算，全部加起来，就是 n (从 1 到 \sqrt{n} 之间的 $1/i$ 之和)，简单讲就是 (从 1 到 n 之间的 $1/i$ 之和) 约等于 \log (对所有 k 从 1 到 n 之间的 $1/k$ 之和)，后者是 $\log \log n$ ，所以前者就是 $\$ \log \log n \$$ ；最外层需要判断 n 次；所以最终时间复杂度为 $O(n \log \log n)$ 。

- 空间复杂度： $O(n)$

上述解法中，申请了大小为 n 的数组空间，空间复杂度跟数字的个数 n 线性相关，因此为 $O(n)$ 。

3的幂、Excel表列序号、快乐数和阶乘后的零

3的幂

给定一个整数，写一个函数来判断它是否是 3 的幂次方。

进阶：你能不使用循环或者递归来完成本题吗？

示例 1：

```
1    输入: 27  
2    输出: true
```

示例 2：

```
1    输入: 0  
2    输出: false
```

示例 3：

```
1    输入: 9  
2    输出: true
```

示例 4：

```
1    输入: 45  
2    输出: false
```

题目分析

- 3 的幂，顾名思义，需要判断当前数字是否可以一直被 3 整除
- 特殊情况：如果 `n == 1`，即 3 的 0 次幂的情况，应输出 `true`

方法一 循环求解

思路

基本想法，可以利用循环解决。排除特殊情况后，用待确定的数字 n ，循环除以 3，看是否能被 3 整除。

详解

- 1、判断特殊情况，若待定值 n 小于 1 则直接返回 `false`
- 2、循环判断待定值 n 是否可以被 3 整除
- 3、若不可以被 3 整除则返回 `false`，若可以则将该数字除以 3，直至循环结束
- 4、其余情况则返回 `true`

代码

```
1  /**
2   * @param {number} n
3   * @return {boolean}
4   */
5  const isPowerOfThree = function (n) {
6    if (n < 1) {
7      return false;
8    }
9    while (n > 1) {
10      // 如果该数字不能被 3 整除，则直接输出 false
11      if (n % 3 !== 0) {
12        return false;
13      } else {
14        n = n / 3;
15      }
16    }
17    return true;
18  };
```

复杂度分析

- 时间复杂度： $O(n)$

该解法中，`while` 循环耗时 $O(n)$ ，其余为 $O(1)$ ，因此总时间复杂度为 $O(n)$ 。

- 空间复杂度： $O(1)$

该解法中，未申请额外的空间，因此空间复杂度为 $O(1)$ 。

解法二 递归求解

思路

或许，我们可以考虑使用递归的方法实现。递归的思路类似于循环，只不过将循环体改为方法的递归调用。

详解

1、判断特殊情况 $n == 1$ 时，直接返回 `true`

2、判断特殊情况 $n \leq 0$ 时，直接返回 `false`

3、若待定值 n 可以被 3 整除，则开始递归

4、若不满足上述条件，则返回 `false`

代码

```
1  /**
2   * @param {number} n
3   * @return {boolean}
4   */
5  const isPowerOfThree = function (n) {
6    // n === 1，即 3 的 0 次幂，返回 true
7    if (n === 1) {
8      return true;
9    }
10   if (n <= 0) {
11     return false;
12   }
13   if (n % 3 === 0) {
14     // 递归调用 isPowerOfThree 方法
15     return isPowerOfThree(n / 3);
16   }
17   return false;
18 };
```

复杂度分析

- 时间复杂度： $O(n)$

该解法中，递归耗时 $O(n)$ ，普通条件判断耗时 $O(1)$ ，整个算法时间复杂度为 $O(n)$

- 空间复杂度： $O(n)$

该解法中，由于递归依赖于栈，因此其空间复杂度为 $O(n)$ 。

方法三 神奇的解法

思路

进阶！既无循环又无递归。

既然要判断输入值是否为 3 的幂，我们可以巧妙的依赖它是否能被 3 的幂的极大值整除来作为判断依据。因此首先要找到 3 的最大次幂。

计算机中最大的整数是 2147483647，转换成 16 进制为 0x7fffffff。

`Math.log(x) / Math.log(y)` 方法可以求出以 `y` 为底，`x` 的对数，即 `y` 的多少次幂的值是 `x`，我们称之为 `maxPow`。由于该值不能被整除，此处 `maxPow` 只需取整数部分。最后，我们可以利用 `Math.pow` 求出 3 的幂的极大值 `maxValue`，并检查该值是否能整除待确定的输入值。

详解

1、判断特殊情况 `n <= 0` 时，直接返回 `false`

2、求计算机允许情况下 3 的最大次幂，记为 `maxPow`

3、求 3 的 `maxPow` 次幂值

4、判断 3 的 `maxPow` 次幂值是否能整除待定值 `n`

代码

```
1  /**
2   * @param {number} n
3   * @return {boolean}
4   */
5  const isPowerOfThree = function (n) {
6    if (n <= 0) {
7      return false;
8    }
9    // 求 3 的最大次幂
10   const maxPow = parseInt((Math.log(0x7fffffff) / Math.log(3)));
```

```
11 // 求 3 的 maxPow 次幂值
12 const maxValue = Math.pow(3, maxPow);
13 // 判断该值是否能整除待定值 n
14 return (maxValue % n === 0);
15 };
```

复杂度分析

- 时间复杂度： $O(1)$

一般情况下，`Math.pow` 的时间复杂度为 $O(1)$ ，取整和除法的复杂度也为 $O(1)$ ，因此该解法的时间复杂度为 $O(1)$ 。

- 空间复杂度： $O(1)$

该解法中，仅申请了 2 个常量级的额外空间，因此空间复杂度为 $O(1)$ 。

Excel表列序号

给定一个Excel表格中的列名称，返回其相应的列序号。

示例

```
1 A -> 1
2 B -> 2
3 C -> 3
4 ...
5 Z -> 26
6 AA -> 27
7 AB -> 28
8 输入: "A",
9 输出: 1
10 输入: "AB",
11 输出: 28
```

方法一

思路

从末尾开始取得每一个字符对应的数 `cur = c.charCodeAt() - 64`（因为 A 的`charCode`为 64） 因为有 26 个字母，所以相当于 26 进制，每 26 个数则向前进一位 数字总和 `sum +=` 当前数 `进制位数 / 26`，初始化进制位数 `carry = 1`

详解

1. 创建临时变量 sum 和初始化进制位数 carry
2. 循环数组
3. 数字总和 sum += 当前数 * 进制位数
4. 进制位数 *= 26

```
1  /**
2   * @param {string} s
3   * @return {number}
4  */
5 const titleToNumber = function (s) {
6   let sum = 0;
7   let i = s.length - 1;
8   let carry = 1;
9   while (i >= 0) {
10     const cur = s[i].charCodeAt() - 64;
11     sum += cur * carry;
12     carry *= 26;
13     i--;
14   }
15   return sum;
16 };
```

复杂度分析

- 时间复杂度： $O(n)$

对于每个元素，通过一次遍历数组的其余部分来寻找它所对应的目标元素，这将耗费 $O(n)$ 的时间。

- 空间复杂度： $O(1)$

由于算法中临时变量得个数与循环次数无关，所以空间复杂度为 $O(1)$

方法二

思路

因为有26个字母，相当于 26 进制转 10 进制

详解

1. 26 进制 转化 10 进制公式， $ans = ans * 26 + num$
2. 比如： $AB = 126 * 2 + 28 = 28, ZY = 2626 + 25 = 701$

```

1  /**
2   * @param {string} s
3   * @return {number}
4  */
5 const titleToNumber = function (s) {
6   const arr = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
7   const len = s.length;
8   let sum = 0;
9   for (let i = 0; i < len; i++) {
10     sum = (arr.indexOf(s[i]) + 1) * Math.pow(26, len - 1 - i) + sum;
11   }
12   return sum;
13 };

```

复杂度分析

- 时间复杂度： $O(n)$

对于每个元素，通过一次遍历数组的其余部分来寻找它所对应的目标元素，这将耗费 $O(n)$ 的时间。

- 空间复杂度： $O(1)$

由于算法中临时变量得个数与循环次数无关，所以空间复杂度为 $O(1)$

快乐数

编写一个算法来判断一个数是不是“快乐数”。

一个“快乐数”定义为：对于一个正整数，每一次将该数替换为它每个位置上的数字的平方和，然后重复这个过程直到这个数变为 1，也可能是无限循环但始终变不到 1。如果可以变为 1，那么这个数就是快乐数。

示例

```

1 输入: 19
2 输出: true
3 解释:
4 1^2 + 9^2 = 82
5 8^2 + 2^2 = 68
6 6^2 + 8^2 = 100
7 1^2 + 0^2 + 0^2 = 1

```

方法一 尾递归

思路

根据示例来看，函数的执行过程是一个可递归的过程，首先，我们先写一个递归函数来模拟这个执行过程，然后按照示例 输入 19 来验证编写函数正确性，然后输入任意数字（比方说 99999），这时，会发现报内存溢出的错误，那这道题就变成了如何解决堆栈溢出的问题：首先，我们要考虑的是，为什么会内存溢出？从题目中，我们可以看到“也可能是无限循环但始终变不到 1”，是“无限循环”导致内存溢出，那我们就应该想一个方式去终结这个“死循环”。首先我们要找到这个循环的规律，怎么找？把递归内容打印（console.log）出来。这时，你会发现一个有规律的死循环。那么，我们只要用一个变量（once）记录已经输入过的值，一旦出现第二次相同输入，就终止递归，并返回“非快乐数”的结果（false）。

详解

1. 申请一个变量来存放已经执行过函数的“输入”，如果出现重复输入，则说明进入了死循环，从“示例”来看：{19:true,82:true,100:true}
2. 将输入（19）转化为数组（[1,9]）
3. 将[1,9]进行平方和运算（ $1^2 + 9^2 = 82$ ）
4. 判断平方和的结果是不是等于 1，如果是，则为“快乐数”，否则，则继续执行 fn 函数
5. 直到平方和等于 1 或者判定为死循环。

```
1 const fn = function (n, once) {
2   if (once[n]) {
3     return false;
4   }
5   const list = n.toString().split('');
6   let result = 0;
7   once[n] = true;
8   list.forEach(val => {
9     result += Math.pow(parseInt(val, 10), 2);
10  });
11  if (result === 1) {
12    return true;
13  } else {
14    return fn(result, once);
15  }
16 };
17
18 /**
19  *
20  * @param {number} n
21  * @return {boolean}
22  */
23 const isHappy = function (n) {
24   const once = {};
```

```
25     return fn(n, once);
26 }
```

\$ 复杂度分析

- 时间复杂度： $O(\log(n))$
- 空间复杂度： $O(1)$

方法二 尾递归

思路

其实和方法一类似，只不过终止循环的条件有所不同，输入 99，观察“函数一”执行时的输出可得，如果函数执行过程中出现 4,16,37,58,89,145,42,20 就不是快乐数，为了稍微提高一下函数执行效率，你也可以简单的枚举以下特殊的快乐数，比方说：1，10，100。

```
1 // 函数一
2 const isHappy = function (n) {
3   console.log('n = ', n);
4   let result = 0;
5   const list = n.toString().split('');
6   list.forEach(val => {
7     result += Math.pow(parseInt(val, 10), 2);
8   });
9   if (result === 1) {
10     return true;
11   } else {
12     return isHappy(result);
13   }
14 }
```

详解

- 已知非快乐数[4,16,37,58,89,145,42,20]
- 已知快乐数：1
- 将输入 (19) 转化为数组 ([1,9])
- 将[1,9]进行平方和运算 ($1^2 + 9^2 = 82$)
- 判断平方和的结果是不是等于 1，若果是，则为“快乐数”，否，则继续执行 fn 函数
- 直到平方和等于 1 或者判定为非快乐数。

```
1 const isHappy = function (n) {
2   const unHappy = [4, 16, 37, 58, 89, 145, 42, 20];
3   if (n === 1) {
4     return true;
5   }
6   if (unHappy.indexOf(n) > -1) {
7     return false;
8   }
9   let result = 0;
10  const list = n.toString().split('');
11  list.forEach(val => {
12    result += Math.pow(parseInt(val, 10), 2);
13  });
14  if (result === 1) {
15    return true;
16  } else {
17    return isHappy(result);
18  }
19};
```

判断优化

可以从 unHappy 取任意一个数作为判断条件，可以减少 indexOf 函数带来的时间消耗。

```
1 const isHappy = function (n) {
2   if (n === 1) {
3     return true;
4   }
5   if (n === 4) {
6     return false;
7   }
8   let result = 0;
9   const list = n.toString().split('');
10  list.forEach(val => {
11    result += Math.pow(parseInt(val, 10), 2);
12  });
13  if (result === 1) {
14    return true;
15  } else {
16    return isHappy(result);
17  }
18};
```

复杂度分析

- 时间复杂度： $O(\log(n))$

- 空间复杂度： $O(1)$

事先通过规律已经找到了所有非快乐数，unHappy 的内存空间的开辟是固定的，所以空间复杂度是 $O(1)$ 。

阶乘后的零

给定一个整数 n ，返回 $n!$ 结果尾数中零的数量。

示例1

```
1 输入: 3
2 输出: 0
3 解释: 3! = 6, 尾数中没有零。
```

示例2

```
1 输入: 5
2 输出: 1
3 解释: 5! = 120, 尾数中有 1 个零。
```

方法一 暴力法

思路

1. 尾数中有 0 必定是 10 的倍数
2. 尾数中有多少个 0 就是整个数能有多少个因子 10
3. 因子 10 又可以拆成 2 5，因此就是找整个数字可以拆分成多少个 2 5
4. 因为在因子中 2 的数量一定比 5 多，所以实际上我们只要找到因子 5 的个数就可以找到尾数中 0 的个数了，所以这个问题就可以转换成找因子 5 的个数。

详解

1. 循环 $1 \sim n$ ，找出能被 5 整除的数字
2. 找到能被 5 整除的数字，找该数字能被拆分成多少个因子 5
3. 所有的个数相加就是尾数 0 的个数

```

1 const trailingZeroes = function(n) {
2     let count = 0;
3     for(let i = 1; i <= n; i++) {
4         let num = i;
5         if (num % 5 === 0) {
6             while(num % 5 === 0 && num !== 0) {
7                 count += 1;
8                 num = parseInt(num / 5);
9             }
10        }
11    }
12    return count;
13 }

```

复杂度分析

- 时间复杂度为: $O(n^2)$

因为要进行两次循环，所以时间复杂度为 $O(n^2)$ ，当数字比较大的时候有性能问题

- 空间复杂度 : $O(1)$

没有申请额外空间，所以空间复杂度为 $O(1)$

方法二

思路

整体思路和方法一基本一致，都是找因子5的个数，只是方法二是在找因子5的个数时做文章，用耗时更少的方法来找5的个数

详解

1. $n!$ 这些乘数中，每隔 5 个数，肯定会有 1 个数至少能拆出一个 5 因子。所以 $n / 5 =$ 至少会出现的 5 的个数。
2. 因为 $n / 5$ 并不能完全算出 5 因子的个数，比如若某个数 $25 = 5 * 5$ ，分解后得到的 5 也算一个，所以能被 25 因式分解相当于会出现 2 个 5 因子，而第一步中除以 5 算个数的时候已经算了一个了，所以相当于比之前会多一个 5 因子
3. 依此类推，能被 $25 * 5 = 125$ 因式分解的相当于比之前按 25 因式分解的时候又多出一个 5 因子。能被 $125 * 5 = 625$ 因式分解的相当于比按 125 因式分解时又多出一个 5 因子。还有 $625 * 5 \dots$

所以 $n!$ 的结果可以拆分为多少个 5 因子呢：

$n/5 + n/25 + n/125 + n/625 + \dots$

```
1  function trailingZeroes(n) {  
2      let count = 0;  
3      while (n > 0) {  
4          n = parseInt(n / 5);  
5          count += n;  
6      }  
7      return count;  
8  }
```

复杂度分析：

- 时间复杂度： $O(\log(n))$ ，遍历次数为 $n/5^x = 1$ ；即 $x = \log_5(n)$ ，所以时间复杂度为 $O(\log(n))$
- 空间复杂度： $O(1)$
没有申请额外空间，所以空间复杂度为 $O(1)$

Pow(x, n)、两数相除、分数到小数和x的平方根

pow(x, n)

实现 `pow(x, n)`，即计算 x 的 n 次幂函数

示例 1：

```
1 输入: 2.00000, 10  
2 输出: 1024.00000
```

示例 2：

```
1 输入: 2.10000, 3  
2 输出: 9.26100
```

示例 3：

```
1 输入: 2.00000, -2  
2 输出: 0.25000  
3 解释: 2^-2 = 1/2^2 = 1/4 = 0.25
```

方法一 二分法

思路

看到题目首先想到可以用暴力计算，如果 n 为整数，则做 n 次底数 x 的累乘，如果 n 为负数，则做 n 次 底数($1 / x$) 的累乘，于是有了如下代码：

```
1 function myPow (x, n) {  
2     // 考虑 n 为 0 的边界情况  
3     if (n === 0) {  
4         return 1;  
5     }  
6 }
```

```

7   const base = n > 0 ? x : 1 / x; // 通过正负号，确认参与幂运算的底数
8   let result = 1;
9
10  for (let i = 1; i <= Math.abs(n); i++) {
11    result *= base;
12  }
13
14  return result;
15 };

```

但是暴力计算会在指数较大时超时，这时我们发现比如计算 2^{20} ， $2^{20} = 2^{10} \cdot 2^{10}$ ，我们使用分治法，只需要计算出 2^{10} 的值做相乘，便可以得出 2^{20} 的值。那么计算 2^{10} 的值，又可以拆解为 $2^5 \cdot 2^5$ ，以此类推.....

详解

我们可以使用折半计算，每次把 n 缩小一半，通过递归，最终获取 x 的 n 次幂，递推公式如下：

$$x^n = x^{n/2} \cdot x^{n/2} \quad (\text{当 } n \text{ 为偶数时})$$

$$x^n = x^{n/2} + x^{n/2} \quad (\text{当 } n \text{ 为奇数时})$$

1. 边界情况：当 n 为 0 时，返回 1，当 n 为 1 或者 -1 时 分别返回 x 与 $1 / x$
2. 其他情况：当 n 为奇数时，需要多乘一次 x 的值

判断 n 是正数还是负数，如果是正数，则直接以 x 作为底数计算；如果是负数，则以 $1 / x$ 作为底数计算。

代码

```

1  function myPow (x, n) {
2    // 考虑 n 为 0, 1, -1 的边界情况
3    if (n === 0) {
4      return 1;
5    } else if (n === 1) {
6      return x;
7    } else if (n === -1) {
8      return 1 / x;
9    }
10
11  const base = n > 0 ? x : 1 / x; // 通过正负号，确认参与幂运算的底数
12  const half = parseInt(n / 2, 10); // 将 n 的值缩小一半
13  const result = myPow(x, half); // 保存折半计算的值，避免重复计算
14

```

```

15     if (n % 2) { // 如果 n 是奇数，则需要额外乘以一次底数
16         return base * result * result;
17     }
18     // 如果 n 是偶数，则直接返回折半计算的乘积
19     return result * result;
20 };

```

复杂度分析

- 时间复杂度： $O(\log_2 n)$

每次递归时，指数 n 减小一半，即 $n, n/2, n/4, \dots, n/2^k$ ，最终趋近于1，令 $n/2^k = 1$ ，计算得出计算次数 $k = \log_2 n$ ，所以时间复杂度为 $O(\log_2 n)$ 。

- 空间复杂度： $O(\log_2 n)$

每一次递归计算，都需要变量保存底数、下一次计算的幂值以及每次递归计算的结果三个常量，递归深度为 $\log_2 n$ ，所以空间复杂度为 $\log_2 n$ 。

方法二 快速幂

思路

我们继续在指数 n 上做文章，将 n 看做数列之和，使得 $n = a_1 + a_2 + \dots + a_n$ ；

那么由 $x^{(a+b)} = x^a x^b$ ，可得 $x^n = x^{a_1} x^{a_2} \dots x^{a_n}$ ；如果能够快速计算出 x^{a_i} 的值，那么即可求得 x^n 。

详解

有了以上分析，我们通过对幂值进行分解来简化计算，例如：

当 $n = 13$ 时，可转化为二进制表示法：1101，那么 $13 = 2^3 + 2^2 + 2^0$ ，即：

$$x^n = x^{2^3} x^{2^2} x^{2^0} ;$$

1 二进制：	1	1	0	1
2 权重：	x^8	x^4	x^2	x^1

可通过移位运算，当该位不为0时，乘以对应位上的权重，循环累乘，得到最终计算结果，对应位上的权重则可以通过低位的权重计算所得，如：

```
1 x^1 = x;
2 x^2 = (x^1) * (x^1);
3 x^4 = (x^2) * (x^2);
4 x^8 = (x^4) * (x^4).
```

我们拿 $x = 5$, $n = 13$ 举例，需要进行 $\log_2 13$ 次计算，即 4 次计算。

将 13 转化为二进制表示法，对应值为 1101，使用 result 来保存计算中间值，使用 base 来保存计算到第 i 位需要乘的权重

result 初始位 1，base 初始值为 5^1

1. 第一轮：从低位取数 $13 \& 1 = 1$ ， $result = result \ base = 1 \ 5^1$ ，更新 base 的值: $base *= base$ 即 base 为 5^2
2. 第二轮：13 右移 1 位后二进制表示为 110， $(13 >> 1) \& 1 = 0$ ，则跳过不予计算 result，更新 base 的值: $base *= base$ 即 base 为 5^4
3. 第三轮：13 右移 2 位后二进制表示为 11， $(13 >> 2) \& 1 = 1$ ， $result = result \ base = 5^1 \ 5^4$ ，更新 base 的值: $base *= base$ 即 base 为 5^8
4. 第四轮：13 右移 3 位后二进制表示为 1， $(13 >> 3) \& 1 = 1$ ， $result = result \ base = 5^1 \ 5^4 * 5^8$

移位运算完毕，得到最终结果 $result = 5^1 \ 5^4 \ 5^8 = 5^{(1+4+8)} = 5^{13}$ 。

代码

```
1 function myPow (x, n) {
2     if (n === 0) {
3         return 1;
4     } else if (n === 1) {
5         return x;
6     } else if (n === -1) {
7         return 1 / x;
8     }
9
10    let pow = Math.abs(n); // 取幂值绝对值，防止 -1 向右移位结果永远是 -1 的情况
11    let result = 1; // 计算的最终结果
12    let base = x; // 初始值为  $x^1$ 
13
14    while (pow) {
15        if (pow & 1 === 1) { // 判断当前位是 0 还是 1
16            result = result * base;
17        }
18    }
19
20    return result;
21}
```

```
19     base *= base; // 更新第 n 位上的权重值
20     pow = pow >> 1; // 向右移位
21 }
22
23 return n > 0 ? result : 1 / result;
24 };
```

复杂度分析

- 时间复杂度： $O(\log_2 n)$

循环计算的次数为二进制的位数，将 n 转换为二进制共有 $O(\log_2 n)$ 位，所以时间复杂度为 $O(\log_2 n)$ 。

- 空间复杂度： $O(1)$

我们使用了 3 个变量来保存中间值，所以空间复杂度为常数级别。

两数相除

给定两个整数，被除数 dividend 和除数 divisor。将两数相除，要求不使用乘法、除法和 mod 运算符。

返回被除数 dividend 除以除数 divisor 得到的商。

示例

```
1 输入: dividend = 10, divisor = 3
2 输出: 3
3 示例 2:
4
5 输入: dividend = 7, divisor = -3
6 输出: -2
7 说明:
```

被除数和除数均为 32 位有符号整数。除数不为 0。假设我们的环境只能存储 32 位有符号整数，其数值范围是 $[-2^{31}, 2^{31} - 1]$ 。本题中，如果除法结果溢出，则返回 $2^{31} - 1$ 。

方法一 利用累加的方式寻找商

思路

我们先让除数 `divisor` 左移直到大于被除数之前得到一个最大的 n 的值，说明被除数 `dividend` 至少包含 2^n 个 `divisor`，然后减去这个数，再一次找到多少个 $n - 1$ 、 $n - 2$

详解

1. 以 $100 / 4$ 为例， $4 * 2^4 = 64 \leq 100$ ，得到 100 里最少有 2^4 个 4
2. 将 $100 - 64$ 得到 36 ， $4 * 2^3 = 32 \leq 36$ ，此时得到 100 里最少有 $2^4 + 2^3$ 个 4
3. 将 $36 - 32$ 得到 4 ， $4 * 2^1 = 4 \leq 4$ ，得出商为 $2^4 + 2^3 + 2^1 = 25$

```
1  /**
2   * @param {number} dividend
3   * @param {number} divisor
4   * @return {number}
5   */
6  const divide = function (dividend, divisor) {
7    const MIN_VALUE = -2147483648;
8    const MAX_VALUE = 2147483647;
9
10   const positive = (dividend ^ divisor) >= 0;
11
12   let d = Math.abs(dividend);
13   const b = Math.abs(divisor);
14   let res = 0;
15   while (d >= b) {
16     let tmp = b;
17     let p = 1;
18     // 寻找有多少个 b
19     while (d >= tmp << 1 && tmp < 1073741823) { // 1073741823 考虑溢出的情况
20       tmp <<= 1;
21       p <<= 1;
22     }
23     d -= tmp;
24     res += p;
25   }
26
27   if (positive) {
28     return res > MAX_VALUE ? MAX_VALUE : res;
29   }
30   return res < MIN_VALUE ? MIN_VALUE : -res;
31 };
```

复杂度分析

- 时间复杂度： $O(\log_2(n))$

首先找出最大的 n

`dividend` 减去 2^n 个 `divisor`，再作为被除数，去找到多少个 $n - 1$ 、 $n - 2$

$$O(n) = \log_2(n) + \log_2(n - 1) + \dots = O(\log_2(n))$$

- 空间复杂度分析： $O(1)$

方法二 转化为减法

思路

任何一个整数都可以表示成以2的幂为底的一组基的线性组合，即

$$num = a_0 2^0 + a_1 2^1 + a_2 2^2 + \dots + a_n 2^n$$

题目要求不能用除法，要求商，可以转化为减法，能减多少次，商就是多少。但是光做减法的效率太低了，由于计算机擅长做移位计算，我们可以用移位。

详解

把被除数 `dividend` 除以 2^n ，根据题干要求 n 最大为 31，不断去尝试，当 $dividend/2^n \geq divisor$ 时，将 `dividend` 减去 $divisor * 2^n$ ，以此类推。

以 $10 / 3$ 为例， $10/2^1 = 5 > 3$ ，将 $10 - 2 * 3 = 4 > 3$ ， $res = 2^1 = 2$ 暂存

$4/2^0 = 4 > 3$ ， $4 - 1 * 3 = 1$ ， $res = 2 + 2^0 = 3$ ，一共减去了 $2^1 + 2^0$ 个 3，因此答案是 3

代码

```
1  /**
2   * @param {number} dividend
3   * @param {number} divisor
4   * @return {number}
5   */
6  const divide = function (dividend, divisor) {
7    const MIN_VALUE = -2147483648;
8    const MAX_VALUE = 2147483647;
9
10   const positive = (dividend ^ divisor) >= 0;
11
```

```
12 let t = Math.abs(dividend);
13 const d = Math.abs(divisor);
14
15 let res = 0;
16 for (let i = 31; i >= 0; i--) {
17     // 加绝对值是考虑溢出的情况，会变成负数
18     if (Math.abs(t >> i) >= d) {
19         res += Math.abs(1 << i);
20         t -= Math.abs(d << i);
21     }
22 }
23 if (positive) {
24     return res > MAX_VALUE ? MAX_VALUE : res;
25 }
26 return res < MIN_VALUE ? MIN_VALUE : -res;
27 };
```

复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$

分数到小数

给定两个整数，分别表示分数的分子 numerator 和分母 denominator，以字符串形式返回小数。

如果小数部分为循环小数，则将循环的部分括在括号内。

示例

```
1 输入: numerator = 1, denominator = 2
2 输出: "0.5"
3
4 输入: numerator = 2, denominator = 3
5 输出: "0.(6)"
```

解这道题只需要数学的基本知识 长除法 的运算规则。长除法的流程如下图：

$$\begin{array}{r}
 0.16 \\
 6) \overline{1.00} \\
 0 \\
 1 0 \\
 \underline{-6} \\
 40 \\
 \underline{-36} \\
 4
 \end{array}$$

... 余数为 1，标记 1 出现在位置 0。
 ... 余数为 4，标记 4 出现在位置 1。
 ... 余数为 4，在位置 1 出现过，所以循环节从位置 1 开始，为 1(6)。

由图中可知当余数出现循环的时候，对应的商也会循环，所以，当同一个余数出现两次时，我们就找到了循环位置

但是在计算过程中有诸多细节问题需要注意：

- 分母 `denominator` 为 0 时，应当抛出异常，这里为了简单起见不考虑
- 分子 `numerator` 为 0 时，结果为 0
- 分母 `denominator` 和分子 `numerator` 中存在一个负数时，结果为负数

方法一 递归

思路

用递归的方法实现。

维护一个记录余数的数组和记录余数第一次出现的位置的map，如果map中存在当前循环计算出的余数，则表示结果开始进入循环部分

详解

1. 先进行边界判断
2. 分别计算出整数和余数部分
3. 对余数部分使用 长除法 计算出新的余数
4. 若map中存在新计算出的余数，则说明出现循环，合并计算结果
5. 若map中不存在新计算出的余数，则将计算出的余数记录下来
6. 重复第3步

```

1 const fun = (map, remainder, remainders, denominator) => {
2   if (!remainder) {
3     return remainders;

```

```

4    }
5    let num = 0;
6    if (map.has(remainder)) {
7        remainders.splice(map.get(remainder), 0, '(');
8        remainders.push(')');
9        return remainders;
10   } else {
11       map.set(remainder, remainders.length);
12       remainder *= 10;
13       num = Math.floor(remainder / denominator);
14       remainder %= denominator;
15       remainders.push(num);
16       return fun(map, remainder, remainders, denominator);
17   }
18 };
19 const fractionToDecimal = function (numerator, denominator) {
20     // 判断边界
21     if (denominator === 0) {
22         return '';
23     }
24     // 判断边界
25     if (numerator === 0) {
26         return '0';
27     }
28     let result = '';
29     // 只存在1位负数
30     if ((denominator < 0) ^ (numerator < 0)) {
31         result += '-';
32         denominator = Math.abs(denominator);
33         numerator = Math.abs(numerator);
34     }
35     // 整数部分
36     const integer = Math.floor(numerator / denominator);
37     result += integer;
38     // 余数部分
39     const remainder = numerator % denominator;
40     if (remainder) {
41         result += '.';
42     }
43     let remainders = [];
44     const map = new Map();
45     if (remainder) {
46         remainders = fun(map, remainder, remainders, denominator);
47     }
48     result += remainders.join('');
49     return result;
50 };

```

复杂度分析

- 时间复杂度： $O(n)$

计算量与结果长度成正比，是线性的。

- 空间复杂度： $O(1)$

该解法中，申请了常数个变量，因此，空间复杂度为 $O(1)$ 。

方法二 迭代

思路

使用迭代方法实现

维护一个记录以{remainder: position}形式记录每一个余数出现的位置的哈希对象，如果存在当前循环计算出的余数，则表示结果开始进入循环部分

详解

1. 先进行边界判断
2. 分别计算出整数和余数部分
3. 对余数部分使用 长除法 计算出新的余数
4. 若 `remainders` 中存在新计算出的余数，则说明出现循环，合并计算结果
5. 若 `remainders` 中不存在新计算出的余数，则将计算出的余数和改余数出现的位置记录下来
6. 重复第3步

```
1 const fractionToDecimal = function (numerator, denominator) {
2     // 判断边界
3     if (denominator === 0) {
4         return '';
5     }
6     // 判断边界
7     if (numerator === 0) {
8         return '0';
9     }
10    let result = '';
11    // 只存在1位负数
12    if ((denominator < 0) ^ (numerator < 0)) {
13        result += '-';
14        denominator = Math.abs(denominator);
15        numerator = Math.abs(numerator);
16    }
17    // 整数部分
18    const integer = Math.floor(numerator / denominator);
19    result += integer;
20    // 余数部分
21    let remainder = numerator % denominator;
22    if (remainder) {
23        result += '.';
```

```
24    }
25    // 小数部分
26    let decimal = '';
27    let index = 0;
28    const remainders = {};
29    while (remainder) {
30        const target = remainders[remainder];
31        if (!isNaN(target)) {
32            decimal = `${decimal.substring(0, target)}${decimal.substring(target)}`;
33            break;
34        }
35        remainders[remainder] = index++;
36        remainder *= 10;
37        const num = Math.floor(remainder / denominator);
38        decimal = `${decimal}${num}`;
39        remainder = remainder % denominator;
40    }
41    result += decimal;
42    return result;
43};
```

复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$

该解法中，申请了常数个变量，因此，空间复杂度为 $O(1)$ 。

x的平方根

实现 `int sqrt(int x)` 函数。即计算并返回 x 的平方根，其中 x 是非负整数，由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

示例1

```
1 输入: 4
2 输出: 2
```

示例2

```
1 输入: 8
2 输出: 2
```

3 解释：8 的平方根是 2.82842...，由于返回类型是整数，小数部分将被舍去。

方法一 顺序查找

思路

解题思路：从数字1开始找，一旦找到平方值等于x的数字i，直接返回i。如果找到平方值大于x的数字i，需要返回i - 1。

详解

1. 对于任一给定数字 x，我们进行循环；
2. 对于每次循环的数字 i，将 $i * i$ 与 x 的值进行比较；
3. 如果前者大于后者，则说明 i 比我们需要的值大，又因为需要返回的是整数，所以 $i-1$ 就是我们要找的值；
4. 如果两者值正好相等，那么 i 就是我们要找的值；
5. 如果前者小于后者，说明 i 与自身的乘积还没有达到要求，需要继续循环下去。

代码

```
1  /**
2   * @param {number} n - a positive integer
3   * @return {number}
4  */
5  const mySqrt= function(x) {
6      for (let i = 1; i <= x; i++) {
7          if(i * i > x) {
8              return (i - 1);
9          }else if(i * i == x) {
10              return i;
11          }
12      }
13      return 0;
14 }
```

复杂度分析

- 时间复杂度： $O(\sqrt{x})$

对于每个元素，我们试图通过遍历数组的其余部分来寻找它所对应的目标元素，故需要耗费 $O(\sqrt{x})$ 的时间。

- 空间复杂度： $O(1)$

上述解法不需要申请额外的空间，故空间复杂度为 $O(1)$ 。

方法二 二分查找法

思路

解题思路：采用 二分查找 的思想，因为x是非负整数，那么当x是0的时候平方根为0，x为1时平方根为1，只有当x大于1时才需要计算。因为当 $x > 1$ 时，x 的平方根 y 肯定在 1 到 x 之间 即 $1 < y < x$ ，知道了数值的范围，我们可以每次把划分区间分为3部分 $[(0, mid), mid, (mid, end)]$ ，从而不断缩小空间，即：在区间 $[0, end]$ 取中间值mid,判断 $mid * mid$ 与 x 的大小关系。

当 $mid * mid > x$ 时 表示, mid mid 大了，那么接下来在 $[1, mid-1]$ 取中间值再判断。

当 $mid * mid < x$ 时 表示, mid mid 小了，那么接下来在 $[mid + 1, x]$ 取中间值再判断。

详解

1. 设置 3 个变量分别为 start、mid、end，并将 start 赋值为 1，end 赋值为所求目标 x 的一半的最正整数；
2. 因为存在 0，这样的特殊情况，开始时循环条件就不成立，我们直接返回 end 即可；
3. 只要 start 不大于 end，就进行循环操作，同时给 mid 赋值，这样把划分区间分为3部分 $[(start, mid), mid, (mid, end)]$ ；
4. 计算 mid 与自身的乘积 并且与 x 进行比较，前者大于后者时，则需要为 end 重新赋值；
5. 如果前者小于后者，我们需要更改最小值，则需要为 start 重新赋值；
6. 这样不断地缩小取值的范围，如果 mid 乘积 与 x 两者相同，mid 值就是我们需要的值。

代码

```
1  /**
2   * @param {number} x
3   * @return {number}
4  */
5 const mySqrt = function(x) {
6   let start = 1;
7   let end = Math.floor(x/2) +1;
8   let mid;
9   while(start <= end) {
10     mid = Math.floor((start + end) / 2);
11     if (mid * mid > x) {
12       // 更改最大值，继续取中间值
```

```
13         end = mid - 1
14     } else if (mid * mid < x) {
15         // 更改最小值，继续取中间值
16         start = mid + 1
17     } else {
18         return mid
19     }
20 }
21 return end
22};
```

复杂度分析

- 时间复杂度： $O(\log(n))$

二分法的时间复杂度是对数级别的,故时间复杂度为 $O(\log(n))$ 。

- 空间复杂度： $O(1)$

使用了常数个数的辅助空间用于存储和比较,不需要申请额外的空间 ,故空间复杂度为 $O(1)$ 。

数组

[Arrays] JS

在计算机中，数组是最常见的数据结构，对于一些非科班出身的同学，可能对数据结构的认识就到数组为止了。数组是编程语言中的内建类型，一般来说效率很高。

定义

数组是一个储存元素的线性集合(collection)。元素可以通过索引来任意存取，这个索引通常来说是数字，用来计算元素之间的存储位置的偏移量。与其他编程语言不同，JavaScript 中数组的长度可随时改变，并且其数据在内存中也可以不连续。刚刚说了通常这个索引是数字，聪明的你已经发现了，在 Javascript 中，通过字符串依然可以访问对应的元素。

```
1 const a = [1, 2, 3];
2
3 a['1'] // 2
```

实际上，Javascript 中的数组是一种比较特殊的对象，用来计算元素之间偏移量的索引是该对象的属性，索引可能是整数。因为在 Javascript 中，对象的属性名必须是字符串，这些数字索引被转化为字符串类型。

```
Object.keys(a); // ["0", "1", "2"]
```

因此，Javascript 中的数组，严格地来说应该被称作对象，因此在日常的开发中，它有很多属性和方法可以在编程时使用，譬如上面使用的 Object.keys。

创建数组

```
1 const a = new Array(3);
2 const b = ['A', 'B', 'C'];
3 b.length // 3
```

通常来说第二种比较常用。和其他编程语言一样，JS 中的数组也是有序的，并且从 0 开始编号，通过对对应的索引可以直接访问到对应的元素。

```
1 b[0] // A
2 b[2] // C
```

通过索引来替换相应的元素

```
1 b[0] = 'D'
2 b[0] // D
```

JS 数组中的元素可以是任何类型

```
1 const a = ['A', { name: 'B' }, () => 'C']
2
3 a[0] // A
4 a[1].name // B
5 a[2]() // C
```

操作数组

- push

在尾部插入元素

```
1 const a = ['A', 'B'];
2 a.push('C');
3 a // ['A', 'B', 'C']
```

- `pop`

取出最后一个元素并返回

```
1 const a = ['A', 'B'];
2 a.pop(); // B
3 a // ['A']
```

- `shift` 取出第一个元素并返回

```
1 const a = ['A', 'B'];
2 a.shift(); // A
3 a // ['B']
```

- `unshift`

在头部添加元素

```
1 const a = ['B'];
2 a.unshift('A');
3 a // ['A', 'B']
```

通过对比 `push`、`pop` 和 `shift`、`unshift` 我们发现，`push` 和 `pop` 是作用于数组尾部的方法，而 `shift` 和 `unshift` 是作用于数组头部的方法。

不同之处

上面提到，JS 的数组和其他编程语言的不太一样，它是一种特殊的对象，是对象的扩展，提供了特殊的方法来处理有序的数据集合以及 `length` 属性。但从本质上来说，它仍然是一个对象。

```
1 const a = [];
2 a.age = 1;
3 a.age // 1
```

因为数组是基于对象来实现，所以在对象可以做的事情，在数组上也能搞。但是这个 JS 引擎会发现，我们在像使用常规对象一样使用数组，那么针对数组的优化就不再适用了，可能会带来性能问题。

下面是关于使用 JS 数组的教科书式的反例：

- 随意给数组添加属性：`arr.name = 'Arr'`
- 制造断层

```
1 const a = [];
2 a[0] = 1;
3 a[10] = 10;
```

所以虽然 JS 中的数组可以是不连续的结构，但是我们在使用的时候还是要按照常规套路出牌，不然建议直接使用常规对象 `{}`。

关于性能

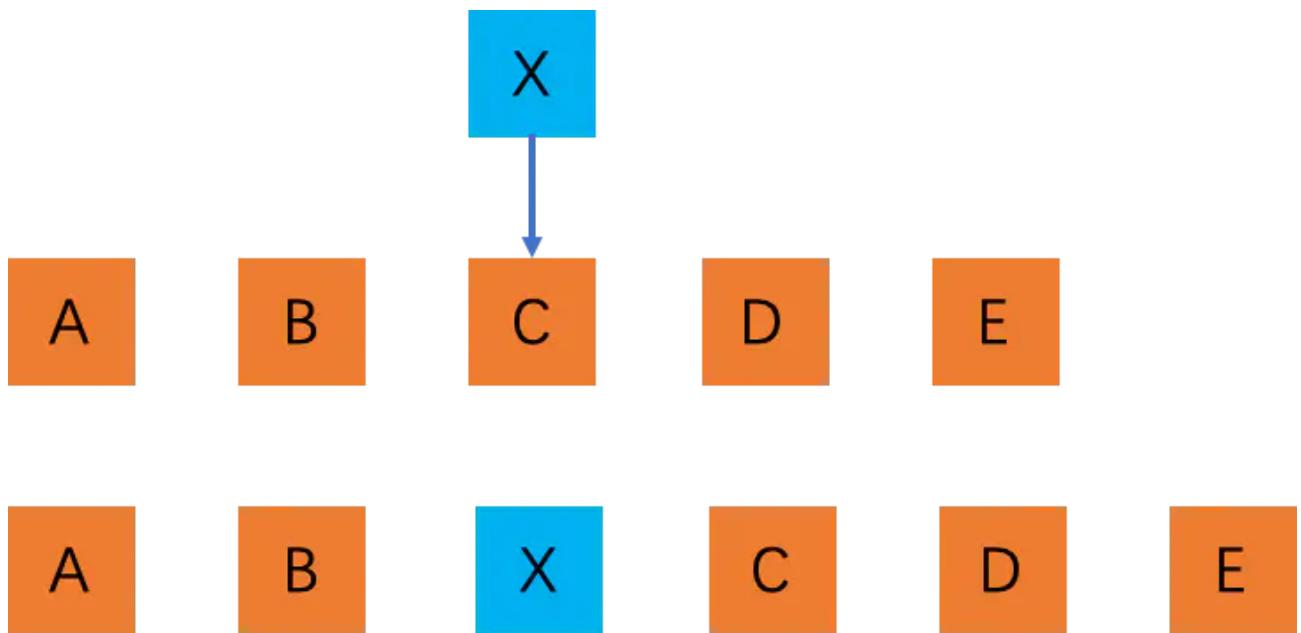
上面说到在使用数组的时候，要保证数组内容的连续性，会导致插入和删除的效率比较低。

我们先来看下插入的时候发生了什么。

假设数组的长度为 n ，现在我们需要将一个元素插入到第 i 个位置，有新元素要加入，其他的元素就要挪地方，我们需要把第 $i \sim n$ 的元素都顺序地往后挪一位。那插入操作的时间复杂度是多少呢？

如果在数组的尾部插入，那么其他元素都不用动，时间复杂度就是 $O(1)$ 。如果运气比较差，刚好要从头部插入，这样一来所有的元素都要往后挪，此时的时间复杂度为 $O(n)$ 。得到平均的时间复杂度为 $(1 + 2 + 3 + \dots + n)/n = O(n)$

数组的元素越多，插入的位置越靠前，也就意味着需要花费更多的时间。



再来看删除操作，和插入类似，如果删除数组末尾的元素，时间复杂度为 $\mathcal{O}(1)$ 。如果删除头部的元素，时间复杂度为 $\mathcal{O}(n)$ ，平均的时间复杂度也为 $\mathcal{O}(n)$ 。

综上，数组最大的特点就是支持随机访问，但插入、删除操作的效率也因此受到影响，平均情况时间复杂度为 $\mathcal{O}(n)$ 。

如何实现

先来看 V8 中关于数组的源码：

src/objects/js-array.h

```
1 // The JSArray describes JavaScript Arrays
2 // Such an array can be in one of two modes:
3 //   - fast, backing storage is a FixedArray and length <= elements.length();
4 //     Please note: push and pop can be used to grow and shrink the array.
5 //   - slow, backing storage is a HashTable with numbers as keys.
6 class JSArray: public JObject {
7 public:
8     // [length]: The length property.
9     DECL_ACCESSORS(length, Object)
10
11    // Number of element slots to pre-allocate for an empty array.
12    static const int kPreallocatedArrayElements = 4;
13};
```

观察代码我们发现，JS 的数组，继承了 `JSONObject`，所以说 JS 的数组是一个特殊的对象。因此也不难解释，JS 的数组有那么的“骚操作”了，可以存放任意不同类型的值，它的内部也是 `k-v` 的储存形式。

从源码中的注释来看，数组有两种储存模式：`Fast` 和 `Slow`。

- `Fast`：快速的存储结构是 `FixedArray`，并且数组长度 $\leq \text{elements.length}()$ ；快速的根据索引来直接定位，`push` 和 `pop` 操作会对数组进行动态的扩容和缩容
- `Slow`：慢速的基于 Hash 表来实现

下面我们来具体看下 `FastElements` 和 `SlowElements` 如何实现。

- 快数组(`FastElements`)

`FixedArray` 是一种线性的存储方式。创建的新空数组，默认的存储方式是快数组，在 `push` 和 `pop` 时会进行对应的扩容和缩容。

- 慢数组(`SlowElements`)

慢数组以哈希表的形式来储存在内存中，不用开辟大块连续的存储空间，节省了内存，它的效率比快数组要低。

快与慢之间如何转变

- 快 \rightarrow 慢

首先来看源码中判断是否需要转为慢数组部分的代码：

```
// If the fast-case backing storage takes up much more memory than a dictionary
// backing storage would, the object should have slow elements.
// static
static inline bool ShouldConvertToSlowElements(uint32_t used_elements,
                                                uint32_t new_capacity) {
    uint32_t size_threshold = NumberDictionary::kPreferFastElementsSizeFactor *
        NumberDictionary::ComputeCapacity(used_elements) *
        NumberDictionary::kEntrySize;
    return size_threshold <= new_capacity;
}

static inline bool ShouldConvertToSlowElements(JSObject object,
                                              uint32_t capacity,
                                              uint32_t index,
                                              uint32_t* new_capacity) {
    STATIC_ASSERT(JSObject::kMaxUncheckedOldFastElementsLength <=
        JSObject::kMaxUncheckedFastElementsLength);
    if (index < capacity) {
        *new_capacity = capacity;
        return false;
    }
    if (index - capacity >= JSObject::kMaxGap) return true;
    *new_capacity = JSObject::NewElementsCapacity(index + 1);
    DCHECK_LT(index, *new_capacity);
    // TODO(ulan): Check if it works with young large objects.
    if (*new_capacity <= JSObject::kMaxUncheckedOldFastElementsLength ||
        (*new_capacity <= JSObject::kMaxUncheckedFastElementsLength &&
         ObjectInYoungGeneration(object))) {
        return false;
    }
    return ShouldConvertToSlowElements(object.GetFastElementsUsage(),
                                       *new_capacity);
}
```

从代码中可以看出，当出现以下情况时，会转变为慢数组：

- 1 插入的 `index - 当前的capacity >= kMaxGap(1024)` 时，也就是至少有了 1024 个断层，会转变为慢数组
- 2 新容量 $\geq 3 \times$ 扩容后的容量 $\times 2$

- 慢 \rightarrow 快

我们知道了什么时候快数组会转为慢数组，也大概可以猜到什么时候慢数组会转为快数组，为了严谨起见，我们还是看一下源码。

各有千秋

快数组以空间换时间，申请了连续内存，提高效率，但是比较占内存。

慢数组以时间换空间，不需要申请连续的空间，节省了内存，但是效率较低。

本章节分为 4 个部分：

- Part 1
 - 翻转整数
 - 只出现一次的数字
 - 两数之和
 - 旋转图像
- Part 2
 - 从排序数组中删除重复项
 - 加一
 - 买股票的最佳时机
 - 移动零
- Part 3
 - 两个数组的交集
 - 一周中的第几天
 - 有效的数独
 - 除资深以外数组的乘积
 - 存在重复元素
- Part 4
 - 字谜分组
 - 三数之和
 - 无重复字符的最长子串
 - 矩阵置零
 - 递增的三元子序列

阅读完本章节，你将有以下收获：

- 熟悉 `JavaScript` 中数组的常见操作，插入、删除、迭代、排序等。
- 能够解决一些数组相关的算法题。

旋转数组、只出现一次的数字、两数之和、旋转图像

旋转数组

给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数。

示例

```
1 输入: [1,2,3,4,5,6,7] 和 k = 3
2 输出: [5,6,7,1,2,3,4]
3 解释:
4 向右旋转 1 步: [7,1,2,3,4,5,6]
5 向右旋转 2 步: [6,7,1,2,3,4,5]
6 向右旋转 3 步: [5,6,7,1,2,3,4]
```

方法一

思路

数据元素向右移动 1 个位置，相当于将数组元素的最后一项截取，然后放到第一项的位置，因此向右移动 k 个位置，就是循环执行上述操作 k 次。而当 k 为数组长度的倍数时，实际相当于没有移动，所以实际需要循环操作的次数为 $k \% l$ 。

详解

- 首先计算出需要循环移动的次数；
- 通过数组的 `unshift()` 和 `pop()` 方法实现旋转，循环执行 k 次。

`unshift()` 方法将把它的参数插入数组的头部，并将已经存在的元素顺次地移到较高的下标处，该方法不会创建新数组，而是直接修改原数组。

`pop()` 方法将删除数组的最后一个元素，把数组长度减 1，并且返回它删除的元素的值。

```
1 /**
2  * @param {number[]} nums
3  * @param {number} k
4  * @return {void} Do not return anything, modify nums in-place instead.
```

```
5  */
6 const rotate = function (nums, k) {
7   const l = nums.length;
8   k = k % l;
9   for (let i = 0; i < k; i++) {
10     nums.unshift(nums.pop());
11   }
12};
```

复杂度分析

- 时间复杂度： $O(n)$

循环遍历的次数取决于k的值，与k值呈线性关系，因此复杂度为 $O(n)$ 。

- 空间复杂度： $O(1)$

没有申请额外的空间，因此复杂度为 $O(1)$ 。

方法二

思路

方法一是将数组中的元素一个一个移动，我们还可以一次性将最后的 $k \% l$ 项全部截取，通过扩展运算符'...'将截取的值放到数组的前边，实现旋转。

详解

- 首先还是计算出需要截取的数组元素的长度；
- 通过数组的 `splice()` 方法截取需要移动的元素，然后使用扩展运算符'...'将截取的元素当作参数，通过 `unshift()` 方法将截取的元素放到数组的前边。

`splice()` 方法可删除从 `index` 处开始的零个或多个元素，然后返回被删除的项目。

数组的扩展运算符 `...` 相当于将数组展开,主要的使用场景是用于数组复制、合并等。

`unshift()` 方法的第一个参数将成为数组的 `index` 为0的新元素，如果还有第二个参数，它将成为 `index` 为1的新元素，以此类推。

```
1 /**
2  * @param {number[]} nums
3  * @param {number} k
4  * @return {void} Do not return anything, modify nums in-place instead.
5 */
```

```
6 const rotate = function (nums, k) {  
7   const l = nums.length;  
8   k = k % l;  
9   nums.unshift(...nums.splice(l - k, k));  
10};
```

复杂度分析：

- 时间复杂度： $O(1)$ 。采用一次性截取，所有方法都只执行了1次，因此复杂度为 $O(1)$ 。
- 空间复杂度： $O(1)$ 。没有申请额外的空间，因此复杂度为 $O(1)$ 。

方法三

思路

先将原数组的所有元素整体往后移动 k 个位置，给需要旋转的元素预留出位置，然后通过替换和删除，实现数组的旋转。

详解

- 先将原数组原有的元素从最后一位开始，依次移动到（原下标 + k ）的位置；
- 然后再从改变后的新数组的下标为 $(k - 1)$ 的元素开始，依次将最后一位赋值给新数组下标为 $(k - 1)$ 的元素，然后删除掉最后一位元素。

```
1 /**
2  * @param {number[]} nums
3  * @param {number} k
4  * @return {void} Do not return anything, modify nums in-place instead.
5 */
6 const rotate = function (nums, k) {
7   const l = nums.length;
8   k = k % l;
9   for (let i = l - 1; i >= 0; i--) {
10     nums[i + k] = nums[i];
11   }
12   for (let j = k - 1; j >= 0; j--) {
13     nums[j] = nums[l + j];
14     nums.pop();
15   }
16};
```

复杂度分析：

- 时间复杂度： $O(n)$

循环遍历的次数取决于数组长度，与数组长度呈线性关系，因此复杂度为 $O(n)$ 。

- 空间复杂度： $O(n)$

数组进行了扩充，申请了n个空间，因此复杂度为 $O(n)$ 。

只出现一次的数字

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

示例 1:

```
1 输入: [2,2,1]
2 输出: 1
```

示例 2:

```
1 输入: [4,1,2,1,2]
2 输出: 4
```

方法一 分组查询法

思路

将数组遍历，并通过过滤的方法，将值相同的数归集为数组的一个元素，由于除了一个元素，其他元素都会出现两次，所有只要找到过滤的集合的长度为1的那个集合，该集合第一个元素即是该元素。

详解

1. 遍历数组，由于需要返回值，这里使用map方法
2. 使用过滤函数，过滤数组中值与当前遍历的元素的值相同的元素
3. 现在得到了一个存在多个集合的数组，而数组中唯一值的那个元素的集合肯定值存在它自己
4. 查询这个集合中长度只有1的集合，再取这个集合的第一个元素，即是只出现一次的数字

代码

```
1 const singleNumber = (nums) => {
2   const numsGroup = nums.map(num => nums.filter(v => v === num));
3   return numsGroup.find(num => num.length === 1)[0];
4 };
```

复杂度分析

- 时间复杂度： $O(n^2)$

使用了 `map` 和 `filter`，嵌套遍历，故为 $O(n^2)$ 。

- 空间复杂度： $O(n)$

`map` 方法创建了一个长度为 n 的数组，占用了 n 大小的空间。

方法二 异或比较法

思路

异或运算符可以将两个数字比较，由于有一个数只出现了一次，其他数皆出现了两次，类似乘法法则无论先后顺序，最后相同的数都会异或成0，唯一出现的数与0异或就会得到其本身，该方法是最优解，直接通过比较的方式即可得到只出现一次的数字。

详解

- 将数组的一个元素与下一个元素做异或比较，直接使用`reduce`方法
- 两两异或最后与所有元素都不相同，最后返回的值即是只出现一次的数字。

代码

```
1 const singleNumber = (nums) => {
2   return nums.reduce((accumulator, currentValue) => accumulator ^ currentValue);
3 };
```

复杂度分析

- 时间复杂度： $O(n)$

仅用 `reduce` 方法遍历，一层遍历，故为 $O(n)$ 。

- 空间复杂度： $O(1)$

空间复杂度为常量，占用空间没有随数据量 n 的大小发生改变，故为 $O(1)$ 。

两数之和

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。

示例

```
1 给定 nums = [2, 7, 11, 15], target = 9
2
3 因为 nums[0] + nums[1] = 2 + 7 = 9
4 所以返回 [0, 1]
```

方法一 暴力法

思路

遍历每个元素 `x`，并查找是否存在一个值与 `target - x` 相等的目标元素。

详解

1. 遍历每个元素 `x`
2. 并查找是否存在一个值与 `target - x` 相等的目标元素。

代码

```
1 const twoSum = function (nums, target) {
2   for (let i = 0; i < nums.length; i++) {
3     for (let j = i + 1; j < nums.length; j++) {
4       if (nums[j] === target - nums[i]) {
5         return [i, j];
6       }
7     }
8   }
9};
```

复杂度分析

- 时间复杂度： $O(n^2)$

对于每个元素，通过遍历数组的其余部分来寻找它所对应的目标元素，这将耗费 $O(n^2)$ 的时间。

- 空间复杂度： $O(1)$
没有申请额外的空间，所以空间复杂度为 $O(1)$ 。

方法二 利用 map

思路

将每个元素的值和它的索引加到表中，检查每个元素所对应的目标元素 ($\text{target} - \text{nums}[i]$) 是否存在于表中

详解

1. 将每个元素的值和它的索引添加到表中
2. 检查每个元素所对应的目标元素 ($\text{target} - \text{nums}[i]$) 是否存在于表中

代码

```
1 const twoSum = function (nums, target) {
2   const lookup = {};
3   let res = [];
4   nums.some((v, i) => {
5     if (lookup[target - v]) {
6       res = [lookup[target - v], i];
7       return true;
8     } else {
9       lookup[v] = i;
10      return false;
11    }
12  });
13  return res;
14};
```

复杂度分析

- 时间复杂度： $O(n)$
我们只遍历了包含有 n 个元素的列表一次，在 `map` 中进行的每次查找只花费 $O(1)$ 的时间，因此总的复杂度为 $O(n)$
- 空间复杂度： $O(n)$

上述解法中，申请了大小为 n 的空间，空间复杂度跟数字的个数 n 线性相关，因此为 $O(n)$ 。

旋转图像

给定一个 $n \times n$ 的二维矩阵表示一个图像，将图像顺时针旋转 90 度。

必须在原地旋转图像，需要直接修改输入的二维矩阵，不要使用另一个矩阵来旋转图像。

示例

```
1 给定 matrix =
2 [
3   [1,2,3],
4   [4,5,6],
5   [7,8,9]
6 ],
7 原地旋转输入矩阵，使其变为：
8 [
9   [7,4,1],
10  [8,5,2],
11  [9,6,3]
12 ]
```

```
1 给定 matrix =
2 [
3   [ 5, 1, 9,11],
4   [ 2, 4, 8,10],
5   [13, 3, 6, 7],
6   [15,14,12,16]
7 ],
8
9 原地旋转输入矩阵，使其变为：
10 [
11   [15,13, 2, 5],
12   [14, 3, 4, 1],
13   [12, 6, 8, 9],
14   [16, 7,10,11]
15 ]
```

方法一 四角旋转

思路

找到一个矩阵中对应点的旋转90度需要涉及到的四个位置的数字，然后将对应的四个数字按照顺时针90度的方法交换即可。

详解

1. 根据输入得到矩阵的维数；
2. 以左上角为 $(0, 0)$ 点，从左向右递增，从上向下递增建立矩阵；
3. 将需要矩阵上需要交换的四个数字存放在一个临时数组中；
4. 将原本位置上的数据按照数组中的数据进行替换。

代码

```
1 function rotate (matrix) {  
2     const n = matrix.length; // n维矩阵  
3     for (let i = 0; i < n; i++) {  
4         for (let j = i; j < n - i - 1; j++) {  
5             const a = [matrix[i][j], matrix[j][n - 1 - i], matrix[n - 1 - i][n - 1 - j]  
6             matrix[i][j] = a[3];  
7             matrix[j][n - 1 - i] = a[0];  
8             matrix[n - 1 - i][n - 1 - j] = a[1];  
9             matrix[n - 1 - j][i] = a[2];  
10        }  
11    }  
12 }
```

复杂度分析

- 时间复杂度： $O(n^2)$
将矩阵进行遍历，二维数组需循环遍历两次，则复杂度为 $O(n^2)$
- 空间复杂度： $O(1)$
所占用空间不会随矩阵的维数而有所变化

方法二 正向对称

思路

先将矩阵沿左上角到右下角的对角线进行对称，然后将矩阵沿垂直中线对称即可。

详解

1. 根据输入得到矩阵的维数；
2. 以左上角为 (0, 0) 点，从左向右递增，从上向下递增建立矩阵；
3. 将矩阵沿左上角到右下角的对角线进行对称，以题目例子 1 为例即：

```
1  [
2    [1,4,7],
3    [2,5,8],
4    [3,6,9]
5  ]
```

1. 再将矩阵沿垂直中线进行对称。即：

```
1  [
2    [7,4,1],
3    [8,5,2],
4    [9,6,3]
5  ]
```

代码

```
1  function rotate (matrix) {
2    const n = matrix.length;
3    // 调换对角线元素
4    for (let i = 0; i < n; i++) {
5      for (let j = i; j < n; j++) {
6        const temp = matrix[i][j];
7        matrix[i][j] = matrix[j][i];
8        matrix[j][i] = temp;
9      }
10    }
11    // 调换每行的左右元素
12    for (let k = 0; k < n; k++) {
13      for (let l = 0; l < Math.floor(n / 2); l++) {
14        const temp = matrix[k][l];
15        matrix[k][l] = matrix[k][n - 1 - l];
16        matrix[k][n - 1 - l] = temp;
17      }
18    }
19  }
```

复杂度分析

- 时间复杂度： $O(n^2)$
将矩阵进行遍历，二维数组需循环遍历两次，则复杂度为 $O(n^2)$
- 空间复杂度： $O(1)$
所占用空间不会随矩阵的维数而有所变化

方法三 反向对称

思路

先将矩阵沿右上角到左下角的对角线进行对称，然后将矩阵沿水平中线对称即可。

详解

1. 根据输入得到矩阵的维数；
2. 将矩阵沿右上角到左下角的对角线进行对称，以题目例子 1 为例即：

```
1  [
2    [9,6,3],
3    [8,5,2],
4    [7,4,1]
5  ]
```

1. 将矩阵沿水平中线进行对称。即：

```
1  [
2    [7,4,1],
3    [8,5,2],
4    [9,6,3]
5  ]
```

代码

```
1  function rotate (matrix) {
2    const n = matrix.length;
3    // 按对角线调换数字
4    for (let i = 0; i < n; i++) {
5      for (let j = 0; j < n - 1 - i; j++) {
6        const temp = matrix[i][j];
7        matrix[i][j] = matrix[n - 1 - j][n - 1 - i];
```

```
8         matrix[n - 1 - j][n - 1 - i] = temp;
9     }
10    }
11
12    // 从头到尾交换每一行
13    for (let k = 0; k < Math.floor(n / 2); k++) {
14        const temp = matrix[k];
15        matrix[k] = matrix[n - 1 - k];
16        matrix[n - 1 - k] = temp;
17    }
18 }
```

复杂度分析

- 时间复杂度： $O(n^2)$

将矩阵进行遍历，二维数组需循环遍历两次，则复杂度为 $O(n^2)$

- 空间复杂度： $O(n)$

空间与矩阵维数正相关

从排序数组中删除重复项、加一、买股票的最佳时机和移动零

从排序数组中删除重复项

给定一个排序数组，你需要在原地删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在原地修改输入数组并在使用 O(1) 额外空间的条件下完成。

示例1

```
1 给定数组 nums = **[1,1,2],**
2
3 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。**
4
5 你不需要考虑数组中超出新长度后面的元素。
```

示例2

```
1 给定 nums = **[0,0,1,1,1,2,2,3,3,4],**
2
3 函数应该返回新的长度 **5**, 并且原数组 nums 的前五个元素被修改为 **0, 1, 2, 3, 4**。
4
5 你不需要考虑数组中超出新长度后面的元素。
```

说明

为什么返回数值是整数，但输出的答案是数组呢？

请注意，输入数组是以“引用”方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下：

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

```
3
4 // 在函数里修改输入数组对于调用者是可见的。
5 // 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。
6 for (let i = 0; i < len; i++) {
7     print(nums[i]);
8 }
```

方法一

思路

我们先遍历数组，若发现相同的相邻项，将该元素删除。此时数组的长度也会发生变化，我们需要把 $i - 1$ ，保证遍历顺序不会出错。最后，再返回数组的长度。

详解

1. 遍历数组里的元素；
2. 判断该元素的相邻项是否与之相同；
3. 若相同，则删除该元素，同时将数组长度减一，继续遍历；
4. 待遍历结束时，返回数组的新长度。

代码

```
1 const removeDuplicates = function (nums) {
2     // 遍历数组
3     for (let i = 1; i < nums.length; i++) {
4         // 若该元素的相邻项与之相同，则删除该元素
5         if (nums[i - 1] === nums[i]) {
6             nums.splice(i - 1, 1);
7             // 因删除该元素后，数组长度会减一，故 i 也需要减一
8             i--;
9         }
10    };
11    return nums.length;
12 };
```

复杂度分析

- 时间复杂度： $O(n)$
共执行了 n 次，因此时间复杂度为 $O(n)$ 。
- 空间复杂度： $O(1)$

没有申请额外的空间，因此空间复杂度为 $O(1)$ 。

方法二

思路

我们用 count 来记录不重复的下标数量，第一个数必定不是重复的，即 $\text{nums}[0]$ 肯定是不重复的，所以从第二项（即 $\text{nums}[1]$ ）开始，遍历数组，判断该下标的值跟不重复的数组最后一个元素 $\text{nums}[\text{count}]$ 是否相同，如果不相同，将该元素值赋值给 $\text{nums}[\text{count} + 1]$ ，然后 $\text{count}++$ ，继续遍历。待遍历结束时，我们可以通过 count 数量来判断不重复元素个数，因为 count 是从 0 开始的，故返回的新数组的长度为 $\text{count} + 1$ 。

比如原数组为 $[0,0,1,1,1,2,2,3,4]$ ，经过遍历会变成类似 $[0,1,2,3,4,x,x,x,x]$ 的结构，此时 $\text{count} = 4$ ，返回新数组长度 $\text{count} + 1 = 5$ 。

详解

1. 创建字段 count，用来记录不重复的下标数量，初始值为 0；
2. 因数组的第一个元素（即 $\text{nums}[\text{count} = 0]$ ）必定是不重复的，故从数组第二项开始（即 $\text{nums}[1]$ ）开始，遍历数组里的元素；
3. 判断数组当前元素是否与 $\text{nums}[\text{count}]$ 相等，若不同，得知当前元素并未重复，将该元素值赋值给 $\text{nums}[\text{count} + 1]$ ，然后 $\text{count}++$ ，继续遍历；
4. 待遍历结束，我们可以通过 count 数量来判断不重复元素个数，因为 count 是从 0 开始计数的，故返回的新数组的长度为 $\text{count} + 1$ 。

代码

```
1 const removeDuplicates = function (nums) {
2     let count = 0;
3     // 遍历数组
4     for (let i = 1; i < nums.length; i++) {
5         // 若该下标的值跟不重复的数组最后一个元素不相同，则该值添加到不重复数组后一位
6         if (nums[count] !== nums[i]) {
7             nums[count + 1] = nums[i];
8             count++;
9         }
10    };
11    // 因为 count 是从 0 开始的，故返回的数组长度加一
12    return count + 1;
13};
```

复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$

加一

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]
2 输出: [1,2,4]
3 解释: 输入数组表示数字 123.
```

## 方法一 奇技淫巧

### 思路

我们可以把数组转化成数字加一，然后再转成数组

### 详解

利用数组的 `join` 方法，转成数字。

在提交代码时发现有 `6145390195186705543` 溢出的情况，使用 `BigInt` 来解决。

加一之后使用 `toString` 方法，转换成字符，最后 `split` 成数组。

`BigInt` 是 `JavaScript` 中的一个新的原始类型，可以用任意精度表示整数。使用 `BigInt`，即使超出`JavaScript Number`的安全整数限制，也可以安全地存储和操作大整数。

### 代码

```
1 /**
2 * @param {number[]} digits
3 * @return {number[]}
4 */
5 const plusOne = function (digits) {
6 return (BigInt(digits.join('')) + 1n).toString().split('');
7 };
```

## 复杂度分析

- 时间复杂度： $O(n)$   
复杂度与输入数组的长度线性相关，为  $O(n)$
- 空间复杂度： $O(1)$   
没有申请额外空间，复杂度为  $O(1)$

## 方法二 进位相加

### 思路

我们可以模拟加法的操作

```
1 1 2 3
2 +
3 -----
4 1 2 4
```

### 详解

在实际情况中，加一有且只有以下两种情况：

- 9 加一进位
- 其他数字加一

先把这个位的数加一，如果没有进位就直接推出循环。

如果进位了，再把十位的数加一，个位数设置为0，如此循环，，直到判断没有再进位就退出循环返回结果。

```
1 9 9
2 + 1
3 -----
4 1 0 0
```

对于 9、99 这种情况，需要进行补位

## 代码

```
1 const plusOne = function (digits) {
2 for (let i = digits.length - 1; i >= 0; i--) {
3 digits[i]++;
4 digits[i] = digits[i] % 10; // 9 + 1 = 10 ---> 0 判断有没有进位
5 if (digits[i] === 0) {
6 return digits; // 最后没有进位直接返回
7 }
8 }
9 digits.splice(0, 0, 1);
10 return digits;
11};
```

## 复杂度分析

- 时间复杂度： $O(n)$   
复杂度与输入数组的长度线性相关，复杂度为  $O(n)$
- 空间复杂度： $O(1)$   
没有申请额外空间，复杂度为  $O(1)$

## 买卖股票的最佳时机 II

给定一个数组，它的第  $i$  个元素是一支给定股票第  $i$  天的价格。

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

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

### 示例 1：

```
1 输入: [7,1,5,3,6,4]
2 输出: 7
3 解释: 在第 2 天 (股票价格 = 1) 的时候买入，在第 3 天 (股票价格 = 5) 的时候卖出，这笔交易所
4 随后，在第 4 天 (股票价格 = 3) 的时候买入，在第 5 天 (股票价格 = 6) 的时候卖出，这笔
```

## 示例 2：

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

## 示例 3：

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

## 方法一

### 思路

定一个滑动窗口，滑动窗口内所有的价格从后向前是递减，如若不是则调整下标，满足该条件。前一个下标从后向前遍历，两个下标初始位置为数组最后一个。如果遇到价格不是递减，则相减得出利润差，并移动两个下标到新的起点。如此遍历直到第一日。

### 详解

1. 初始化滑动窗口的两个下标，前下标从后向前遍历。
2. 如果遇到价格不是递减的情况，则相减得出利润差，并移动两个下标到新的起点。
3. 如此遍历直到第一日。

### 代码

```
1 const maxProfit = (prices) => {
```

```
2 // 总利润
3 let num = 0;
4 // 滑动窗口后一个下标
5 let aftOff = prices.length - 1;
6 // 滑动串口前一个下标
7 let offset = prices.length - 1;
8 while (offset > 0) {
9 // 价格递减则移动前一个下标，否则计算出利润差并移动两个下标到新的起点
10 if (prices[offset] > prices[offset - 1]) {
11 offset -= 1;
12 } else {
13 num += prices[aftOff] - prices[offset];
14 offset -= 1;
15 aftOff = offset;
16 }
17 }
18 // 价格递减到第一日情况的逻辑补充
19 if (aftOff !== offset) {
20 num += prices[aftOff] - prices[offset];
21 }
22 return num;
23 };
```

## 复杂度分析

- 时间复杂度： $O(n)$

假设 \$n\$ 为给定数组的长度，while 循环执行了  $n$  次，因此，时间复杂度为  $O(n)$ 。

- 空间复杂度： $O(1)$  该解法中，申请了常数个变量，因此，空间复杂度为  $O(1)$ 。

## 方法二

### 思路

遍历每日的价格，从第二天起，如果每日的价格都比前一日的价格高，则相减得出利润差，累加在总利润上，直到遍历到最后一日。

### 详解

- 从前向后遍历数组，从第二个开始
- 如果当前日的价格比前一日的价格高，则相减得出利润差，累加在总利润上。
- 遍历到最后一日，退出循环，返回总利润

### 代码

```
1 const maxProfit = function (prices) {
2 // 总收益
3 const totalBenefit = 0;
4 // 当前日下标
5 const offset = 1;
6 const len = prices.length;
7 while (offset <= len - 1) {
8 // 如果当日价格比前一天价格高，则相减得出收益
9 const curPrice = prices[offset];
10 const prePrice = prices[offset - 1];
11 if (curPrice > prePrice) {
12 totalBenefit += curPrice - prePrice;
13 }
14 offset += 1;
15 }
16 return totalBenefit;
17};
```

## 复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$

## 移动零

给定一个数组 `nums`，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。

### 示例

```
1 输入: [0,1,0,3,12]
2 输出: [1,3,12,0,0]
```

### 说明

- 必须在原数组上操作，不能拷贝额外的数组。
- 尽量减少操作次数。

### 注意事项

- 由于题目要求必须在原数组上操作，数组的 `filter` API 或是 `sort` 算法，都是不必考虑的。
- 切记不要边遍历数组边修改数组长度，如：`splice`，`push`，`pop` 等。

## 方法一 双指针

### 思路

创建两个指针 `i` 和 `j`，`j` 来记录非零元素的下标。遍历遇到一个非零元素时，就把 `j` 往右挪。遍历结束后，`j` 指向了最后一个非零元素的下标，再把剩余地址填充 0 即可。

### 详解

1. `i` 用于遍历数组，`j` 来记录非零元素的下标
2. 当发现 `num[i] !== 0` 时，说明找到非零元素，把第 `j` 和 `i` 指向的两个元素交互位置，再把 `j` 往右挪
3. 遍历结束把剩余的地址填充 0

```
1 const moveZeroes = function (nums) {
2 let j = 0;
3 for (let i = 0; i < nums.length; i++) {
4 if (nums[i] !== 0) {
5 nums[j] = nums[i];
6 j++;
7 }
8 }
9 // 遍历完了，把尾部的元素填充 0 即可
10 nums.fill(0, j, nums.length);
11};
```

### 复杂度分析

- 时间复杂度： $O(n)$   
算法包括一次遍历，运行次数与数组长度一致，所以时间复杂度为  $O(n)$
- 空间复杂度： $O(1)$   
额外空间包括若干个常量

## 方法二 双指针优化

### 思路

方法一进行了两次循环，还能进一步优化，只循环一次。

### 详解

思路与方法 1 极其相似，依次用非零元素与零元素交换即可，优点在于一步到位，不用再次填充 0。1. 用 `j` 记录非零元素的下标，初始为 0 2. 依次用非零元素与 `j` 对应元素交换位置，每次交换后，`j` 加 1，`j` 对应元素就被替换成 0（除初始值）3. 零元素不用处理，会被替换或者保持不变。

```
1 const moveZeroes = function (nums) {
2 let j = 0;
3 let temp = '';
4 for (let i = 0; i < nums.length; i++) {
5 if (nums[i] !== 0) {
6 temp = nums[j];
7 nums[j] = nums[i];
8 nums[i] = temp;
9 j++;
10 }
11 }
12};
```

### 复杂度分析：

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$

# 两个数组的交集、一周中的第几天、有效的数独、除资深以外数组的乘积和存在重复元素

## 两个数组的交集

给定两个数组，计算数组交集。

1. 输出结果中每个元素出现的次数，应与元素在两个数组中出现的次数一致。
2. 我们可以不考虑输出结果的顺序。

### 示例

```
1 输入: nums1 = [1,2,2,1], nums2 = [2,2]
2 输出: [2,2]
```

```
1 输入: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
2 输出: [4,9]
```

### 方法一 模拟哈希

#### 思路

遍历第一个数组，将第一个数组的值、该值出现的次数，以(key:value)的形式存储下来，接着遍历第二个数组，判断是否在(key:value)中存在，存在则 value 减去 1，继续。

#### 详解

1. 定义模拟哈希的对象 hashObject、定义 result 数组存放最终符合条件的结果；
2. for 循环遍历第一个数组，将数组中每个值作为 key、出现次数作为 value 存到 hashObject 对象中，第一次出现 value 为 1，再次出现 value 加 1；
3. for 循环遍历第二个数组，判断第二个数组中每个值是否在 hashObject 中存在，即在 hashObject 作为 key 对应的 value 为 1 或者大于 1，如果存在将该值 push 到 result 数组中，并将该值对应的 value 减去 1；
4. 返回 result 即可；

## 代码

```
1 const intersect = function (nums1, nums2) {
2 const hashObject = {};
3 for (let i = 0; i < nums1.length; i++) {
4 if (hashObject[nums1[i]]) {
5 hashObject[nums1[i]] += 1;
6 } else {
7 hashObject[nums1[i]] = 1;
8 }
9 }
10 const result = [];
11 for (let j = 0; j < nums2.length; j++) {
12 if (hashObject[nums2[j]]) {
13 result.push(nums2[j]);
14 hashObject[nums2[j]] -= 1;
15 }
16 }
17 return result;
18};
```

## 复杂度分析

- 时间复杂度： $O(n)$

分别 for 循环遍历两个数组，每个耗费时间都是  $O(n)$ ，总的时间复杂度  $O(n)$

- 空间复杂度： $O(n)$

定义了一个 `hashObject` 对象存储第一个数组每个值、每个值出现的次数，空间大小最大为  $O(n)$ 、定义一个 `result` 数组存放最终符合条件的结果，空间大小最大为  $O(n)$ ，两者一起空间大小最大为  $2n$ ，所以空间复杂度为  $O(n)$

## 方法二 长短数组

### 思路

找出两个数组中的长短数组，遍历短数组，判断值是否存在于长数组中，如果存在，记录并且删除长数组中的该值，继续。

### 详解

- 定一个 `longerArr` 存放两个数组中较长的数组、`shorterArr` 存放两个数组中较短的数组，定义 `result` 数组存放最终结果；

2. for 循环遍历 shorterArr 数组，如果 shorterArr 数组中当前元素在 longerArr 数组中存在，就将该值 push 到 result 数组中，并且在 longerArr 数组中删除对应当前值的元素；
3. 返回 result 数组即可；

```
1 const intersect = function (nums1, nums2) {
2 const longerArr = nums1.length > nums2.length ? nums1 : nums2;
3 const shorterArr = nums1.length > nums2.length ? nums2 : nums1;
4 const result = [];
5 for (let i = 0; i < shorterArr.length; i++) {
6 if (longerArr.indexOf(shorterArr[i]) > -1) {
7 result.push(shorterArr[i]);
8 longerArr.splice(longerArr.indexOf(shorterArr[i]), 1);
9 }
10 }
11 return result;
12};
```

## 复杂度分析：

- 时间复杂度： $O(n)$   
只对长度较小的数组进行一次 for 循环遍历，时间复杂度为  $O(n)$
- 空间复杂度： $O(n)$   
需要额外的空间分别存储长短数组，该表最多需要存储  $2n$  个元素，故空间复杂度为  $O(n)$   
。

## 一周中的第几天

给你一个日期，请你设计一个算法来判断它是对应一周中的哪一天。

输入为三个整数：day、month 和 year，分别表示日、月、年。

您返回的结果必须是这几个值中的一个 {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"}。

说明：给出的日期一定是在 1971 到 2100 年之间的有效日期。

### 示例 1：

```
1 输入：day = 31, month = 8, year = 2019
```

```
2 输出："Saturday"
```

## 示例 2:

```
1 输入：day = 18, month = 7, year = 1999
2 输出："Sunday"
```

## 示例 3:

```
1 输入：day = 15, month = 8, year = 1993
2 输出："Sunday"
```

## 方法一 直接求解法

### 思路

说明中指出了：给出的日期一定是在 1971 到 2100 年之间的有效日期。那么可以先算出给出的日期与 1970 年 12 月 31 日之间一共有多少天，然后取模 7 的余数即可得到星期几。

### 详解

1. 先算出 1970 年 12 月 31 日距今一共有多少天
2. 然后对得到的天数模 7 取余数，得到一个数字如 5，表示当前是星期五
3. 最后根据得到的数字输出英文的星期几

### 代码

```
1 const dayOfTheWeek = function (day, month, year) {
2 const Month = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31];
3 const Week = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
4 // 1970年12月31日为星期四，即初始值为4
5 let count = 4;
6 // 算上此年前每年的日期，都先当365天算
7 count += (year - 1970 - 1) * 365;
8 // 算上此月前每个月的日期
9 for (let i = 1; i < month; i++) {
10 count += Month[i - 1];
```

```

11 }
12 // 算上此月的日期
13 count += day;
14 // 加上今年之前的闰年天数
15 for (let y = 1971; y <= year - 1; y++) {
16 if ((y % 4 === 0 && y % 100 !== 0) || y % 400 === 0) {
17 count++;
18 }
19 }
20 if ((year % 4 === 0 && year % 100 !== 0) || year % 400 === 0) {
21 if (month > 2) {
22 count++;
23 }
24 }
25 return Week[count % 7];
26 };

```

## 复杂度分析

- 时间复杂度： $O(n)$  对于任意一个年月日，计算此月前每个月的日期里的循环体需要执行  $n$  次，计算加上今年之前的闰年天数里的循环体需要执行  $n$  次，其它计算需要各执行 1 次。因此函数执行次数为  $f(n) = 2n + x$ ，其中  $x$  为正整数常数。问题规模属于线性阶，故时间复杂度为  $O(n)$ 。
- 空间复杂度： $O(1)$   
此算法需要分配的空间主要为 Month 和 Week 数组，分别需要存放 12 和 7 个常量节点。空间占用属于常数阶，故空间复杂度为  $O(1)$ 。

## 方法二 巧用 JS 库函数法

### 思路

JS 库提供了一系列的函数。借助以下函数，我们可以得到任意一个符合题目说明的日期是星期几。

```

1 Date.parse(datestring);
2 // 解析一个日期时间字符串(datestring)，并返回 1970/1/1 午夜距离该日期时间的毫秒数。
3 var Date = new Date();
4 // 初始化当前的日期和时间
5 Date.getDay();
6 // 获取当前星期X(0-6，0代表星期天)

```

### 详解

1. 先调用 `Date.parse()` 方法得到给定日期的时间戳
2. 再调用 `new Date()` 方法得到给定时间戳的日期和时间
3. 再调用 `Date.getDay()` 方法得到一个数字如 5，表示当前是星期五
4. 最后输出英文的星期几

## 代码

```
1 const dayOfTheWeek = function (day, month, year) {
2 const date = new Date(Date.parse(` ${year}/${month}/${day}`));
3 const Week = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'
4 return Week[date.getDay()];
5 };
```

## 复杂度分析

- 时间复杂度： $O(1)$
- 空间复杂度： $O(1)$

## 有效的数独

判断一个  $9 \times 9$  的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。  
1. 数字 `1-9` 在每一行只能出现一次。  
2. 数字 `1-9` 在每一列只能出现一次。  
3. 数字 `1-9` 在每一个以粗实线分隔的  $3 \times 3$  宫内只能出现一次。

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 |   |   | 7 |   |   |   |   |
| 6 |   |   | 1 | 9 | 5 |   |   |   |
|   | 9 | 8 |   |   |   |   | 6 |   |
| 8 |   |   |   | 6 |   |   |   | 3 |
| 4 |   |   | 8 |   | 3 |   |   | 1 |
| 7 |   |   |   | 2 |   |   |   | 6 |
|   | 6 |   | 4 | 1 | 9 | 2 | 5 |   |
|   |   |   |   | 8 |   |   |   | 5 |
|   |   |   |   |   |   |   | 7 | 9 |

上图是一个部分填充的有效的数独。

数独部分空格内已填入了数字，空白格用'.'表示。

## 示例

```

1 输入:
2 [
3 ["5","3",".",".","7",".",".",".","."],
4 ["6",".",".","1","9","5",".",".","."],
5 [".","9","8",".",".",".","6","."],
6 ["8",".",".","6",".",".",".","3"],
7 ["4",".",".","8",".","3",".",".","1"],
8 ["7",".",".","2",".",".",".","6"],
9 [".","6",".",".","2",".",".","."],
10 [".",".","4","1","9",".",".","5"],
11 [".",".","8",".","7","9"]
12]
13 输出: true
14
15 输入:
16 [

```

```

17 ["8","3",".",".","7",".",".",".","."],
18 ["6",".",".","1","9","5",".",".","."],
19 [".","9","8",".",".",".","6","."],
20 ["8",".",".","6",".",".",".","3"],
21 ["4",".",".","8",".","3",".",".","1"],
22 ["7",".",".","2",".",".",".","6"],
23 [".","6",".",".",".","2","8","."],
24 [".",".",".","4","1","9",".",".","5"],
25 [".",".",".","8",".",".","7","9"]
26]
27 输出: false

```

解释: 以上两个例子中，除了第一行的第一个数字从 5 改为 8 以外，空格内其他数字均与示例1相同。但由于位于左上角的 3x3 宫内有两个 8 存在，因此这个数独是无效的。

说明:

- 一个有效的数独（部分已被填充）不一定是可解的。
- 只需要根据以上规则，验证已经填入的数字是否有效即可。
- 给定数独序列只包含数字 1-9 和字符 ' .' 。
- 给定数独永远是 9x9 形式的。

## 方法一 利用哈希表

### 思路

用一个哈希对象记录每一个数字是否在当前行 / 列 / 子数独已存在，若已存在，则为无效数独

### 详解

1. 维护三个记录数独单元格元素的哈希对象
2. 以行 / 列 / 子数独 三种方式遍历数据每一个单元格，
3. 如果出现重复，返回 false。
4. 如果没有，则保留此值以进行进一步跟踪。

```

1 const isValidSudoku = function (board) {
2 for (let i = 0; i < 9; i++) {
3 // 行
4 const rowMap = {};
5 // 列
6 const colMap = {};
7 // 子数独
8 const sqreMap = {};

```

```

9 for (let j = 0; j < 9; j++) {
10 const rowEle = board[i][j];
11 const colEle = board[j][i];
12 // 行内是否存在重复
13 if (rowEle !== '.') {
14 if (rowMap[rowEle]) {
15 return false;
16 }
17 rowMap[rowEle] = 1;
18 }
19 // 列内是否存在重复
20 if (colEle !== '.') {
21 if (colMap[colEle]) {
22 return false;
23 }
24 colMap[colEle] = 1;
25 }
26 // 每一个子数独内是否存在重复
27 const R = Math.floor(i / 3) * 3 + Math.floor(j / 3);
28 const C = Math.floor(3 * (i % 3) + j % 3);
29 const sqreEle = board[R][C];
30 if (sqreEle !== '.') {
31 if (sqreMap[sqreEle]) {
32 return false;
33 }
34 sqreMap[sqreEle] = 1;
35 }
36 }
37 }
38 return true;
39 };

```

## 复杂度分析

- 时间复杂度： $O(1)$   
因为只对 81 个单元格进行了一次迭代。
- 空间复杂度： $O(1)$

## 方法二 位运算

### 思路

使用一个9位二进制数判断数字是否被访问，第k位数为1代表k已存在，为0代表k不存在

### 详解

更新方式(记九位数为 `row`，传入的数字为 `rowEle`)：

- 判断是否加入：将 `row` 右移位 `rowEle` 位，与1进行与运算
  - 结果为0：未加入，将 `rowEle` 加入 `row`
  - 结果为1：已加入，返回 `false`
- 将传入的数字加入 `row`：将1左移位 `rowEle` 位，与 `row` 异或
 

例子：对于数字 `1010101000`，其第3,5,7,9位为1，表示当前3,5,7,9已经存在
- 新来数字为2：
  - 将 `1010101000` 右移2位得到 `10101010`，与1进行与运算，结果为0，2不存在。
  - 将1左移2位得到 `100`，异或后得到 `1010101100`
- 新来数字为4：
  - 将 `1010101000` 右移3位得到 `1010101`，与1进行与运算，结果为1，3已经存在。
  - 返回 `false`

列，子数独同上

```

1 const isValidSudoku = function (board) {
2 let row = 0;
3 let col = 0;
4 let sqre = 0;
5 for (let rIndex = 0; rIndex < 9; rIndex++) {
6 row = col = sqre = 0;
7 for (let cIndex = 0; cIndex < 9; cIndex++) {
8 const rowEle = board[rIndex][cIndex];
9 const colEle = board[cIndex][rIndex];
10 const R = Math.floor(rIndex / 3) * 3 + Math.floor(cIndex / 3);
11 const C = Math.floor(3 * (rIndex % 3) + cIndex % 3);
12 const sqreEle = board[R][C];
13 if (!isNaN(rowEle)) {
14 row = ((row >> rowEle) & 1) === 1 ? -1 : row ^ (1 << rowEle);
15 }
16 if (!isNaN(colEle)) {
17 col = ((col >> colEle) & 1) === 1 ? -1 : col ^ (1 << colEle);
18 }
19 if (!isNaN(sqreEle)) {
20 sqre = ((sqre >> sqreEle) & 1) === 1 ? -1 : sqre ^ (1 << sqreEle);
21 }
22 if (row === -1 || col === -1 || sqre === -1) {
23 return false;
24 }
25 }
26 }
27 return true;
28 };

```

复杂度分析

- 时间复杂度： $O(1)$
- 空间复杂度： $O(1)$

## 除本身之外的数组之积

给定长度为  $n$  的整数数组 `nums`，其中  $n > 1$ ，返回输出数组 `output`，其中 `output[i]` 等于 `nums` 中除 `nums[i]` 之外其余各元素的乘积。

### 示例

```
1 输入: [1,2,3,4]
2 输出: [24,12,8,6]
3
4 说明: 请不要使用除法,且在 O(n) 时间复杂度内完成此题。
5
6 进阶:
7 你可以在常数空间复杂度内完成这个题目吗? (出于对空间复杂度分析的目的,输出数组不被视为额外)
```

### 方法一

#### 思路

看完题目最直接的一个想法是，先遍历一遍数组求出所有数字之乘积，然后除以对应位置的上的数字即可。但是，这道题的关键在于，不能使用除法，并且要求时间复杂度为  $O(n)$ 。所以，我们稍微调整下思路，将乘积分成 2 部分，先计算出当前数字的左侧数字乘积，然后计算出当前数字的右侧数字乘积，最后将 2 部分乘积相乘即可。

#### 详解

1. 第一步，定义一个数组 `leftProduct`，用于存放当前数字的左侧数字乘积。通过循环遍历，依次相乘，计算出当前数字的左侧数字乘积。
2. 第二步，定义一个数组 `rightProduct`，用于存放当前数字的右侧数字乘积。同理，依次计算出当前数字的右侧数字乘积。
3. 第三步，将每个数字的左侧乘积和右侧乘积对应相乘，即可得到最终的结果。

#### 代码

```
1 /**
```

```

2 * @param {number[]} nums
3 * @return {number[]}
4 */
5 const productExceptSelf = (nums) => {
6 const len = nums.length;
7 const result = [];
8 const leftProduct = [];// 存储当前数字的左侧数字乘积
9 const rightProduct = [];// 存储当前数字的右侧数字乘积
10 leftProduct[0] = 1;
11 rightProduct[len - 1] = 1;
12 // 计算左侧数字的乘积
13 for (let i = 1; i < len; i += 1) {
14 leftProduct[i] = leftProduct[i - 1] * nums[i - 1];
15 }
16 // 计算右侧数字的乘积
17 for (let j = len - 2; j >= 0; j -= 1) {
18 rightProduct[j] = rightProduct[j + 1] * nums[j + 1];
19 }
20 // 左侧数字的乘积 * 右侧数字的乘积
21 for (let k = 0; k < len; k += 1) {
22 result[k] = leftProduct[k] * rightProduct[k];
23 }
24 return result;
25 };

```

## 复杂度分析

- 时间复杂度： $O(n)$

上述解法中，使用了 3 个单层 for 循环，时间复杂度和数组个数 n 线性相关，因此，时间复杂度为  $O(n)$ 。

- 空间复杂度： $O(n)$

上述解法中，申请了 3 个大小为 n 的数组空间，因此，空间复杂度为  $O(n * 3)$ ，即  $O(n)$ 。

## 方法二

### 思路

对于第一种解法，我们可以再进行下优化，将后面 2 个循环合并下。先计算出左侧数字的乘积，直接放到结果数组 result 中，然后用变量 right 存储每个数字右侧的乘积，并且进行累积相乘，就可以得到最终的结果了。

### 详解

- 第一步，定义一个数组 result，用于存放当前数字的左侧数字乘积。通过循环遍历，依次相乘，计算出当前数字左侧的乘积。

2. 第二步，定义一个变量 `right`，存储每个数字右侧的乘积，并且进行累积相乘，即可得到最终结果。

## 代码

```
1 /**
2 * @param {number[]} nums
3 * @return {number[]}
4 */
5 const productExceptSelf = (nums) => {
6 const len = nums.length;
7 const result = [1];
8 let right = 1;
9 // 计算左侧数字的乘积，存到 result 中
10 for (let i = 1; i < len; i += 1) {
11 result[i] = result[i - 1] * nums[i - 1];
12 }
13 // 用变量 right 存储每个数字右侧的乘积，并且进行累积相乘
14 for (let j = len - 1; j >= 0; j -= 1) {
15 result[j] *= right;
16 right *= nums[j];
17 }
18 return result;
19 };
```

## 复杂度分析

- 时间复杂度： $O(n)$

上述解法中，使用了 2 个单层 for 循环，时间复杂度和数组个数  $n$  线性相关，因此，时间复杂度为  $O(n)$ 。

- 空间复杂度： $O(n)$

上述解法中，申请了大小为  $n$  的数组空间，空间复杂度和数组个数  $n$  线性相关，因此，空间复杂度为  $O(n)$ 。

## 存在重复元素

给定一个整数数组，判断是否存在重复元素。

如果任何值在数组中出现至少两次，函数返回 `true`。如果数组中每个元素都不相同，则返回 `false`。

## 示例1

```
1 输入: [1,2,3,1]
2 输出: true
```

## 示例2

```
1 输入: [1,2,3,4]
2 输出: false
```

## 示例3

```
1 输入: [1,1,1,3,3,4,3,2,4,2]
2 输出: true
```

## 方法一 暴力循环法

### 思路

暴力循环法很简单，排序数组，判断前后两个数字是否相等，只要有相等的情况，那就返回 true ，直到循环全部数组，没有就返回 false

### 详解

1. 对原数组进行排序，按照从小到大排序
2. 设置默认结果是 false
3. 遍历数组，当遍历到的数据和下一个将要遍历的数据做比较，判断是否相同，相同则返回 true , 否则返回 false

```
1 /**
2 * @param {number[]} nums
3 * @return {boolean}
4 */
5 const containsDuplicate = function (nums) {
6 // 按照从小到大的顺序进行排序
7 nums.sort((a, b) => a - b);
```

```
8 let res = false;
9 nums.forEach((i, index) => {
10 // 防止数组下标越界，因为要取第index + 1位的数据
11 if (index < nums.length - 1) {
12 // 进行或运算，当有相等的时候，res设置为true
13 res = res || (i === nums[index + 1]);
14 }
15 });
16 return res;
17};
```

## 复杂度分析

- 时间复杂度： $O(n * \log n)$

对于每个元素，首先通过排序确定数组从小到大的顺序，构造一个从小到大展示的数组，比如 [1,1,2,3]，再通过对比回前后两位的数字是否相等，这将耗费  $O(n * \log n)$  的时间。

- 空间复杂度： $O(1)$

对于数组，无论循环多少遍，空间占用永远只有一个 res，所以空间复杂度是  $O(1)$

## 方法二 转化Set法

### 思路

通过 js 的 API 转化为 Set，然后比较原数组和Set的长度来判断是否有相等的数字

### 详解

- 1.对原数组转化为 Set 去重
- 2.再把Set转化成 Array
- 3.比较两个 Array 的长度是否相等，相等就说明没有重复的数据

```
1 /**
2 * @param {number[]} nums
3 * @return {number}
4 */
5 const containsDuplicate = function (nums) {
6 // 转化为Set来去重
7 const newArr = Array.from(new Set(nums));
8 return newArr.length !== nums.length;
9};
```

## 复杂度分析

- 时间复杂度： $O(n)$

会有1次遍历数组，所以最终的时间复杂度是  $O(n)$

- 空间复杂度： $O(n)$

因为要一个数组的辅助存储空间，所以空间复杂度是  $O(n)$

# 字谜分组、三数之和、无重复字符的最长子串、矩阵置零和递增的三元子序列

## 字谜分组

给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。

### 示例

```
1 输入: ["eat", "tea", "tan", "ate", "nat", "bat"],
2 输出:
3 [
4 ["ate","eat","tea"],
5 ["nat","tan"],
6 ["bat"]
7]
```

### 方法一 排序分类

#### 思路

我们先遍历数组，把每个字母异位词都进行排序。再将排序后的字符串作为 key，将 key 值一样的字母异位词置于同一个数组中。最后，再按照题目所要求的格式返回数组。

#### 详解

1. 创建个空对象 obj，用于后续将字母异位词分类储存；
2. 创建个空数组 arr，用与后续返回结果；
3. 遍历数组里的元素；
4. 将每个字母异位词进行排序，并将排序后的字符串作为 key，可知 key 值一样的即为字母异位词，将他们置于同一个数组中（即 obj["aet"] = ["ate", "eat", "tea"]）；
5. 待上述遍历结束，再遍历 obj，将 obj 的每一个值，push 到 arr 中。

#### 代码

```
1 const groupAnagrams = function (strs) {
2 const obj = {};
3 const arr = [];
```

```

4 // 遍历数组
5 for (let i = 0; i < strs.length; i++) {
6 // 将每个字母异位词进行排序，并将排序后的字符串作为 key
7 const unit = Array.from(strs[i]).sort().join('');
8 // 将 key 值一样的字母异位词置于同一个数组中
9 if (!obj[unit]) {
10 obj[unit] = [];
11 }
12 obj[unit].push(strs[i]);
13 }
14 for (const i in obj) {
15 arr.push(obj[i]);
16 }
17 return arr;
18 };

```

## 复杂度分析

- 时间复杂度： $O(nk \log k)$

外层 `strs` for 循环为  $O(n)$ ，里面根据字符数据进行排序，查资料得排序时间复杂度为  $O(k \log k)$ ， $k$  为字符串长度最大值，即整个时间复杂度为  $O(nk \log k)$ 。

- 空间复杂度： $O(nk)$

代码运行时创建了 `obj` 对象用于存储 `strs` 中的字符串，空间与 `strs` 长度与字符串长度成正比关系，所以空间复杂度为  $O(nk)$ 。

附 [Array.sort排序算法链接](#)

## 方法二 计数分类

### 思路

我们先遍历数组，每次都创建一个长度为 26，元素全是 0 的数组，用于记录每个单词中每个字符出现的次数；然后将其转化为字符串作为 key，将 key 值一样的字母异位词置于同一个数组中。最后，再按照题目所要求的格式返回数组。

### 详解

1. 创建个空对象 `obj`，用于后续将字母异位词分类储存；

2. 创建个空数组 `arr`，用与后续返回结果；

3. 遍历数组里的元素；

4.每次遍历都创建一个长度为 26 (跟 26 个英文字母对应) ,元素全是 0 的数组，用于记录每个单词中每个字符出现的次数，然后将该数组转化的字符串作为 key ,可知 key 值一样的即为字母异位词，将他们置于同一个数组中；

5.待上述遍历结束，再遍历 obj ,将 obj 的每一个值，push 到 arr 中。

## 代码

```
1 const groupAnagrams = function (strs) {
2 const obj = {};
3 const arr = [];
4 // 遍历数组
5 for (let i = 0; i < strs.length; i++) {
6 // 都创建一个长度为 26 ,元素全是 0 的数组，用于记录每个单词中每个字符出现的次数
7 const unit = new Array(26).fill(0);
8 for (let j = 0; j < strs[i].length; j++) {
9 const index = strs[i].charCodeAt(j) - 97;
10 unit[index] += 1;
11 }
12 // 将每个数组转化的字符串作为 key
13 const newUnit = JSON.stringify(unit);
14 // 将 key 值一样的字母异位词置于同一个数组中
15 if (!obj[newUnit]) {
16 obj[newUnit] = [];
17 }
18 obj[newUnit].push(strs[i]);
19 }
20 for (const i in obj) {
21 arr.push(obj[i]);
22 }
23 return arr;
24};
```

## 复杂度分析

- 时间复杂度： $O(nk)$

外层 strs for 循环为  $O(n)$  ,内层为对字符串进行 for 循环，时间复杂度为  $O(k)$  , k 为字符串长度最大值，即整个时间复杂度为  $O(nk)$  。

- 空间复杂度： $O(nk)$

代码运行时创建了 obj 对象用于存储 strs 中的字符串，空间与 strs 长度与字符串长度成正比关系，所以空间复杂度为  $O(nk)$  。

## 三数之和

给定一个包含  $n$  个整数的数组  $\text{nums}$ ，判断  $\text{nums}$  中是否存在三个元素  $a, b, c$ ，使得  $a + b + c = 0$ ？找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

## 示例

```
1 例如，给定数组 nums = [-1, 0, 1, 2, -1, -4] ,
2
3 满足要求的三元组集合为：
4 [
5 [-1, 0, 1],
6 [-1, -1, 2]
7]
```

## 方法一 暴力法

思路

我们首先想到的是三重循环，题目中要求三元组不能重复，因此对每次插入数组对三元组做一下简单的记录。

詳解

直接对数组进行3重遍历，当三个数的和等于 0 时，插入数组，同时对已经插入对数据做一下记录

```
1 const threeSum = function (nums) {
2 const res = [];
3 const uniqueMap = {};
4 nums.sort();
5 for (let i = 0; i < nums.length - 2; i++) {
6 for (let j = i + 1; j < nums.length - 1; j++) {
7 for (let k = j + 1; k < nums.length; k++) {
8 if (nums[i] + nums[j] + nums[k] === 0) {
9 const item = [nums[i], nums[j], nums[k]];
10 if (!uniqueMap[item.join(',')]) {
11 res.push(item);
12 uniqueMap[item.join(',')] = 1;
13 }
14 }
15 }
16 }
17 }
18}
```

```
16 }
17 }
18 return res;
19 };
```

## 复杂度分析

- 时间复杂度： $O(n^3)$   
解法虽然简单，套三重循环，时间复杂度高。
- 空间复杂度： $O(n)$   
额外申请了uniqueMap的空间，复杂度为  $O(n)$

## 方法二 双指针

### 思路

首先对数组进行排序，便于在插入的时候去重，进行双指针遍历时，遇到重复的数就可以很方便得跳过。

### 详解

先将数组排序，令左指针  $L = i + 1$ ，右指针  $R = n - 1$ ，当  $L \leq R$  时，进行循环

- 当  $nums[i] + nums[L] + nums[R] == 0$  时，将三个数插入数组，同时判断  $nums[L]$  和  $nums[L + 1]$  是否重复，去重复解之后，同时将  $L$  和  $R$  移到下一个位置
- 若  $sum$  小于 0，说明  $nums[L]$  太小， $L$  需要右移， $L++$
- 若  $sum$  大于 0，说明  $nums[R]$  太大， $R$  需要左移， $R--$

```
1 const threeSum = function (nums) {
2 const res = [];
3 nums.sort((a, b) => a - b);
4 const length = nums.length;
5
6 for (let i = 0; i < length; i++) {
7 let left = i + 1;
8 let right = length - 1;
9 while (left < right) {
10 const sum = nums[i] + nums[left] + nums[right];
11 if (sum === 0) {
12 res.push([nums[i], nums[left], nums[right]]);
13 }
14 const leftValue = nums[left];
```

```

15
16 // 这两步是为了去重
17 while (left < length && nums[left] === leftValue) {
18 left++;
19 }
20 const rightValue = nums[right];
21 while (right > left && nums[right] === rightValue) {
22 right--;
23 }
24 } else if (sum < 0) {
25 // 小于 0 说明太小了，需要向右移动
26 left++;
27 } else {
28 // 太大了，把右边的指针向左移动
29 right--;
30 }
31 }
32 while (i + 1 < nums.length && nums[i] === nums[i + 1]) {
33 i++;
34 }
35 }
36 return res;
37 };

```

## 复杂度分析

- 时间复杂度： $O(n^2)$   
数组遍历  $O(n)$ ，双指针遍历  $O(n)$ ，因此复杂度为  $O(n) * O(n)$  为  $O(n^2)$
- 空间复杂度： $O(1)$   
指针使用常数大小的额外空间

## 无重复字符的最长子串

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

```

1 输入: "abcabcbb"
2 输出: 3
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

```
1 输入: "bbbbbb"
2 输出: 1
3 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

```
1 输入: "pwwkew"
2 输出: 3
3 解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
4 请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。
```

### 方法一

维护一个数组，数组里从前到后存放着字符。在遍历过程中，维持数组中字符不重复，如果重复，则从数组中 `shift` 出字符，直到移除那个重复字符为止，并记录下数组的最长长度。

### 思路

用一个数组来储存当前字符，在遍历过程中不断地存入不重复字符，遇到重复字符则整理数组达到字符不重复的条件。

### 详解

初始化一个数组和最大值，从前向后遍历字符串，如果该字符不在数组中，则把字符 `push` 到数组中，并且比较记录下当前最大值。否则就从头部向外 `pop` 字符直到该重复字符被移除，如此循环直到结束。

```
1 const lengthOfLongestSubstring = function (s) {
2 let current = 0;
3 let max = 0;
4 const list = [];
5 const len = s.length;
6 for (; current < len; current++) {
7 if (list.indexOf(s[current]) === -1) {
8 list.push(s[current]);
9 } else {
10 do {
11 list.shift();
12 } while (list.indexOf(s[current]) !== -1);
13 list.push(s[current]);
14 }
15 }
16 return max;
17}
```

```
14 }
15 max = Math.max(list.length, max);
16 }
17 return max;
18 };
```

## 复杂度分析

- 时间复杂度： $O(n)$

这样子需要从前向后遍历一遍长度为  $n$  的字符串，需要进行  $n$  次字符是否重复的比较。

- 空间复杂度： $O(n)$

在计算比较过程中，数组最长有可能存放  $n$  个不重复字符串。

## 方法二

从前到后遍历字符串，维护一个string，记录着不重复字符子串。每当遇到重复的字符时候，找到 string中重复的字符，并截断。循环往复直到遍历最后一个字符.

## 思路

记录当前正在遍历的不重复字串的子集 string，在遍历过程中不断地添加不重复字符，遇到重复字符则截断 string 达到 string 内补字符不重复的条件。

## 详解

1. 初始化一个 string 和最大值，
2. 从前向后遍历字符串，
3. 如果该字符不在 string 中，则添加字符到 string 中，并记录下最大值。
4. 否则就截断字符串，如此循环直到结束。

```
1 const lengthOfLongestSubstring = function (s) {
2 let num = 0;
3 let max = 0;
4 let subString = '';
5 for (char of s) {
6 if (subString.indexOf(char) === -1) {
7 subString += char;
8 num++;
9 max = max < num ? num : max;
10 } else {
11 subString += char;
12 subString = subString.slice(subString.indexOf(char) + 1);
13 num = subString.length;
14 }
15 }
16 return max;
17}
```

```
14 }
15 }
16 return max;
17 };
```

## 复杂度分析

- 时间复杂度： $O(n)$

我们只遍历了包含有  $n$  个元素的字符串一次。

- 空间复杂度： $O(n)$

所需的额外空间取决于子串的长度，子串始终小于等于传入字符串的长度，该子串最多需要存储  $n$  个元素。

## 矩阵置零

给定一个  $m \times n$  的矩阵，如果一个元素为 0，则将其所在行和列的所有元素都设为 0。请使用[原地](#)算法。

### 示例 1

```
1 输入:
2 [
3 [1,1,1],
4 [1,0,1],
5 [1,1,1]
6]
7 输出:
8 [
9 [1,0,1],
10 [0,0,0],
11 [1,0,1]
12]
```

### 示例 2

```
1 输入:
2 [
3 [0,1,2,0],
4 [3,4,5,2],
5 [1,3,1,5]
```

```
6]
7 输出:
8 [
9 [0,0,0,0],
10 [0,4,5,0],
11 [0,3,1,0]
12]
```

## 进阶

- 一个直接的解决方案是使用  $O(mn)$  的额外空间，但这并不是一个好的解决方案。
- 一个简单的改进方案是使用  $O(m + n)$  的额外空间，但这仍然不是最好的解决方案。
- 你能想出一个常数空间的解决方案吗？

## 注意事项

1. 注意边遍历边修改原数组，以免未遍历元素受 0 值的影响。

### 方法一 记录 0 的位置

#### 思路

申请额外空间去记录需要置为 0 的行号和列号

#### 详解

1. 申请  $O(m + n)$  的额外空间，声明两个数组，所占空间最大值分别为  $m$ ， $n$ ，分别存放水平方向、垂直方向应该重置为零的元素下标
2. 从上到下，从左到右依次遍历原数组，记录元素为 0 的横坐标与纵坐标到提前声明的两个需置零的数组
3. 待遍历结束后，再次遍历原数组，按照两个需置零的数组把元素置为零。

```
1 const setZeroes = function (matrix) {
2 const len = matrix.length;
3 const width = matrix[0].length;
4 const vertical = [];
5 const horizontal = [];
6 for (let i = 0; i < len; i++) {
7 for (let j = 0; j < width; j++) {
8 if (!matrix[i][j]) {
9 vertical.push(j);
10 horizontal.push(i);
11 }
12 }
13 }
14 for (let i = 0; i < vertical.length; i++) {
15 for (let j = 0; j < horizontal.length; j++) {
16 matrix[horizontal[j]][vertical[i]] = 0;
17 }
18 }
19}
```

```
12 }
13 }
14 for (let i = 0; i < len; i++) {
15 if (horizontal.indexOf(i) > -1) {
16 matrix[i].fill(0, 0, width);
17 }
18 for (let j = 0; j < vertical.length; j++) {
19 matrix[i][vertical[j]] = 0;
20 }
21 }
22 };
```

## 复杂度分析：

- 时间复杂度： $O(m * n)$

算法包括两个遍历，运行次数为是  $m * n$ ，所以时间复杂度是  $O(m * n)$

- 空间复杂度： $O(m + n)$

额外空间包括若干个常量和两个长度分别为  $m$ ， $n$  的数组

## 方法二 原地算法

### 思路

我们不能再额外创建数组来记录 0 的位置，从原数组本身找突破点，用原数组的第一行和第一列记录该行或该列需不需要置零，再根据首行首列的标识对元素置零

### 参考图



image

## 详解

- 首先从左到右，从上到下遍历数组中每一个元素（列遍历从第二列开始），若该元素为0，则同时设置改行首列元素、首行该列元素为0，若首列存在为0元素，则设置标识，意为首列元素会被全部置零，此目的是区分行元素与首列元素置零的标识，
- 然后，从右到左，从下到上遍历数组，若首行该列或该行首列元素为0，则置该元素为0，若存在标识，则置首列元素为0，为什么不选择从左到右，从上到下遍历？是因为首行首列的先根据标志置零，新的零元素会影响后面的数据。

```

1 const setZeroes = function (matrix) {
2 const len = matrix.length;
3 const width = matrix[0].length;
4 let flag = false;
5 for (let i = 0; i < len; i++) {
6 if (!matrix[i][0]) {
7 flag = true;
8 }
9 for (let j = 1; j < width; j++) {
10 if (!matrix[i][j]) {

```

```

11 matrix[i][0] = 0;
12 matrix[0][j] = 0;
13 }
14 }
15 }
16 for (let i = len - 1; i >= 0; i--) {
17 for (let j = width - 1; j > 0; j--) {
18 if (!matrix[0][j] || !matrix[i][0]) {
19 matrix[i][j] = 0;
20 }
21 }
22 if (flag) {
23 matrix[i][0] = 0;
24 }
25 }
26 };

```

## 复杂度分析：

- 时间复杂度： $O(mn)$

算法包括两个遍历，运行次数都是  $mn$ ，所以时间复杂度是  $O(mn)$

- 空间复杂度： $O(1)$

额外空间包括若干个常量，与  $m$ 、 $n$  大小无关

## 递增的三元子序列

给定一个未排序的数组，判断这个数组中是否存在长度为 3 的递增子序列。

数学表达式如下：

如果存在这样的  $i, j, k$ , 且满足  $0 \leq i < j < k \leq n-1$ ，使得  $\text{arr}[i] < \text{arr}[j] < \text{arr}[k]$ ，返回 true；否则返回 false。

说明：要求算法的时间复杂度为  $O(n)$ ，空间复杂度为  $O(1)$ 。

示例 1：

```

1 输入: [1, 2, 3, 4, 5];
2 输出: true;

```

示例 2：

```
1 输入: [5, 4, 3, 2, 1];
2 输出: false;
```

## 方法一 不连续的递增

### 思路

使用 one 和 two 两个数。保证 one 小于 two，若遍历到一个数大于 two，则满足3个数递增。返回 true

### 详解

1. 设置两个变量，one 代表三元子序列的第一个，two 代表三元子序列的第二个。

例如：

- 第一个数作为 one，第二个数若比第一个数大，这两个数可以成三元子序列中的前两个，于是可以赋值给two；
- 第三个数比第二个数小，说明三元子序列可能会从这个数开始  
one、two 的初始值为 undefined，任意数与undefined做比较均为 false。
- 现在，开始循环遍历 num，若：
  - num > two，说明可以构成三元子序列了，返回 true
  - num > one，说明 num 比 two 小（或等于），比 one 大，可以将 two 更新为此 num，
  - num < one，则这个 num 可以成为三元子序列的最小者，更新 one 为 num。

```
1 /**
2 * @param {number[]} nums
3 * @return {boolean}
4 */
5 const increasingTriplet = function (nums) {
6 if (nums.length < 3) return false;
7 let one,
8 two;
9 for (const num of nums) {
10 if (num > two) {
11 return true;
12 } else if (num > one) {
13 two = num;
14 } else {
15 one = num;
16 }
17 }
```

```
18 return false;
19 };
```

## 复杂度分析

- 时间复杂度： $O(n)$ ，遍历了1次含n个元素的空间
- 空间复杂度： $O(1)$ ，遍历过程没有用到新的空间存储数据

## 方法二 贪心算法

### 思路

循环遍历数组，不断更新数组内出现的最小值与最大值，如果出现的一个大于最大值的数，则表示存在长度为3的递增子序列。

### 详解

- 若目标数组 nums 存在递增的三元子序列，设这三个数为 a1,a2,a3, 则  $a_3 > a_2 > a_1$ 。
- 可以先定义两个变量 small, big (small < big) 分别用于存放最小的两个数字，在js中使用 `Number.MAX_SAFE_INTEGER` 常量表示最大的安全整数 (maximum safe integer) ( $2^{53} - 1$ )。
- 遍历数组，实时捕获当前最小的两个数，同时判断在这两个数后方是否存在一个数字  $a_3 > small \&& a_3 > big$ , 若存在，即该数组存在长度为3的递增的子序列。

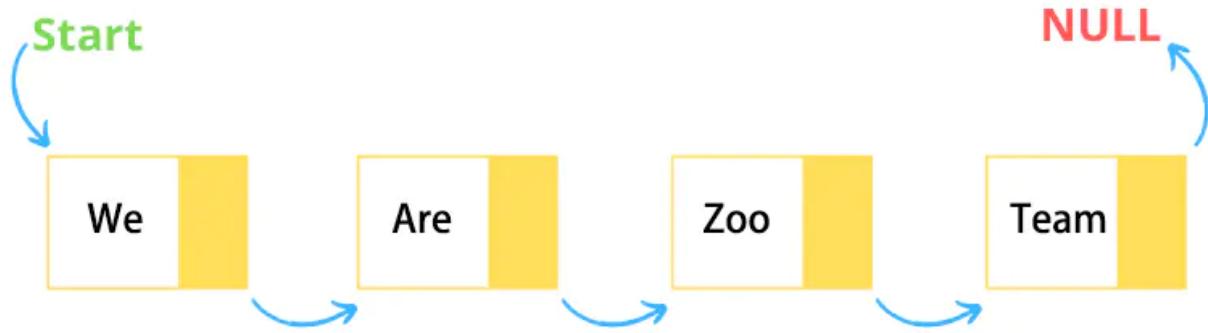
```
1 /**
2 * @param {number[]} nums
3 * @return {boolean}
4 */
5 const increasingTriplet = function (nums) {
6 let small = Number.MAX_SAFE_INTEGER;
7 let big = Number.MAX_SAFE_INTEGER;
8 for (let i = 0; i < nums.length; i++) {
9 if (nums[i] <= small) {
10 small = nums[i];
11 } else if (nums[i] <= big) {
12 big = nums[i];
13 } else {
14 return true;
15 }
16 }
17 return false;
18 };
```

## 复杂度分析

- 时间复杂度： $O(n)$   
遍历了 1 次含  $n$  个元素的空间
- 空间复杂度： $O(1)$   
遍历过程没有用到新的空间存储数据

# 链表

## Linked list



相比数组，链表( `Linked List` )是一种稍微复杂一点的数据结构，掌握起来也要比数组稍难一些。链表是通过“指针”将一组零散的内存块串联起来使用。数组的线序是由数组的下标来决定的，而链表的顺序是由各个对象中的指针来决定。

在多数编程语言中，数组的长度是固定的，一旦被填满，要再加入数据将会变得非常困难。在数组中，添加和删除元素也比较麻烦，因为需要把数组中的其他元素向前或向后移动。

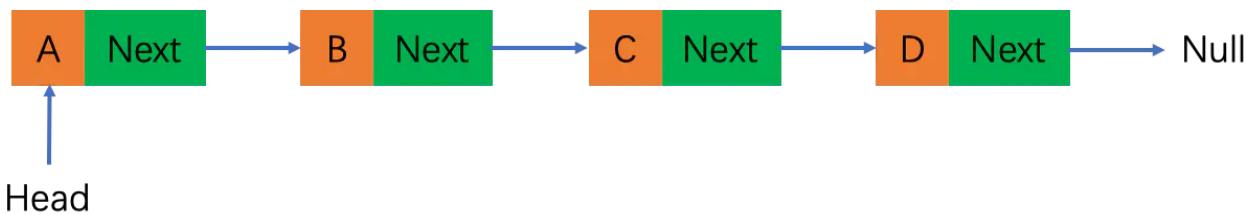
前面介绍数组的章节说过，JavaScript 的数组被实现成了对象，与 Java 相比，效率偏低。

在实际开发中，不能单靠复杂度就决定使用哪个数据结构，没有一种数据结构是完美的，否则其他的数据结构不都被淘汰了。

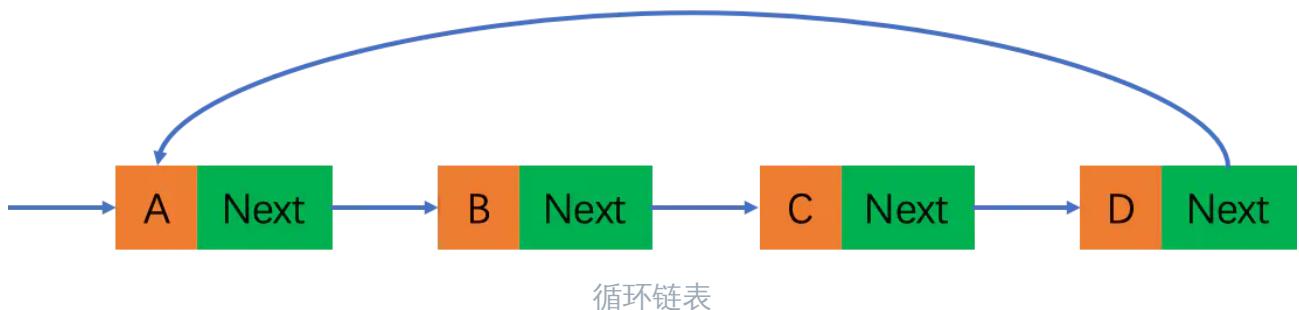
上面的图中， `Are` 跟在 `We` 的后面，而不是说 `Are` 是这个链表的第二个元素。遍历该链表，就是顺着指针，从链表的首元素走到尾元素。

链表的结构可以由很多种，它可以是单链表或双链表，也可以是已排序的或未排序的，环形的或非环形的。如果一个链表是单向的，那么链表中的每个元素没有指向下一个元素的指针。已排序的和未排序的链表较好理解。

### 单链表



单链表



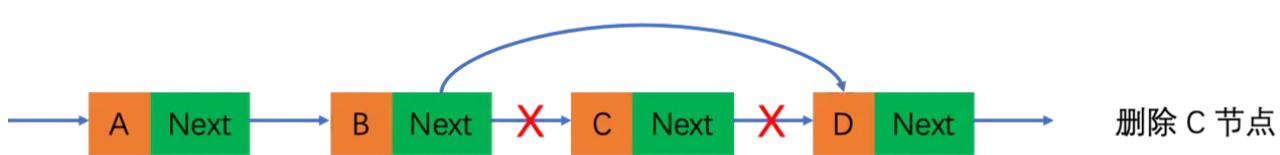
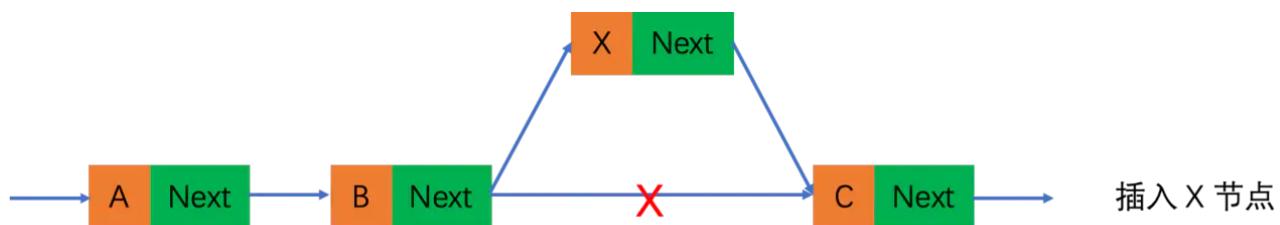
循环链表

循环链表和单向链表的区别在于，单链表的尾元素指向的是 Null，而循环链表的尾元素是指向链表的头部元素。

和数组一样，链表也支持数据的查找、插入和删除。

由于链表是非连续的，想要访问第  $i$  个元素就没数组那么方便了，需要根据指针一个结点一个结点地依次遍历，直到找到相应的结点。

数组在插入或删除元素时，为了保证数据的连续性，需要对原有的数据进行挪动。然而链表在插入或删除时，不要挪动原来的数据，因为链表的数据本身就是非连续的空间，因此在链表中插入、删除数据是非常快的。



如何设计一个链表

我们设计的链表包含两个类。`Node` 类用来表示节点，`LinkedList` 类提供节点插入、删除和查找。

- `Node`

```
1 class Node {
2 constructor(el) {
3 this.el = el;
4 this.next = null;
5 }
6 }
```

- `LinkedList`

```
1 class LinkedList {
2 constructor() {
3 this.head = new Node('head');
4 }
5
6 // 用于查找
7 find() {
8
9 }
10
11 // 插入节点
12 insert() {
13
14 }
15
16 // 删除节点
17 remove() {
18
19 }
20 }
```

`LinkedList` 类提供了所有对链表进行操作的方法。在构造函数中，我们用一个 `Node` 对象来保存该链表的头节点。

头节点 `head` 的 `next` 属性被初始化为 `null`，每当调用 `insert` 方法时，`next` 就会指向新的元素。

## 插入新节点

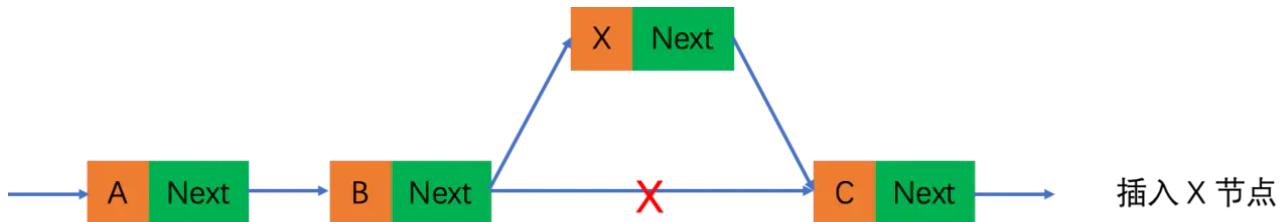
如果要在链表中插入新节点，需要说明在哪个节点前或后插入。先考虑单链表的情况，在后面插入，只需要知道在哪里插入和插入的元素是啥。

在已知的节点后面插入元素，需要先找到这个节点的位置，因此也需要提供一个 `find` 方法来遍历链表，查找数据。

```
1 function find() {
2 // 从链表的头节点开始遍历
3 let currentNode = this.head;
4 while (currentNode && currentNode.el !== item) {
5 currentNode = currentNode.next;
6 }
7 return currentNode;
8 }
```

上面的方法演示了如何在链表上查找元素，如果没找到就把当前指针往后移，找到了就返回改元素，找不到就算了直接返回 `null`。

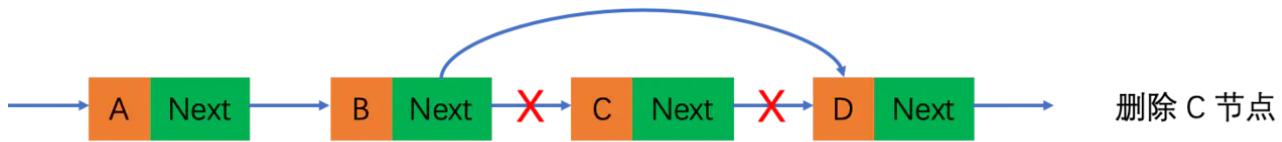
找到元素之后，就这个把新元素插入到链表当中去了。如下图所示，我们先创建一个新节点(X)，把新节点的 `next` 指针指向“后面”的节点的 `next` 指针对应的节点(可能有点绕，看图)，再把“后面”节点的 `next` 指针指向该新节点，一个完美的插入就完成了。



```
1 function insert(el, item) {
2 const newNode = new Node(el);
3 const currentNode = this.find(item);
4 // 将 X 的 next 指向 B 的 next
5 newNode.next = currentNode.next;
6 currentNode.next = newNode;
7 }
```

## 删除节点

从链表中删除节点时，和插入节点类似，首先需要找到相应的节点前一个节点，找到节点之后，让他的 next 指针不再指向待删除的节点，而是指向待删除节点的下一个节点。



如图所示，我们需要删除的是 C 节点，让 B 的 next 指向 C 的下一个节点，也就是 D，这样就完成了一个删除的操作。现在来看下代码如何实现。

注意：我们这次找的是待删除节点的前一个节点，所以先来定义一个 `findPrev` 方法

```
1 function findPrev(item) {
2 let node = this.head;
3 while (node.next !== null && node.next.el !== item) {
4 node = node.next;
5 }
6 return node;
7 }
```

接下来实现 `remove`

```
1 function remove(item) {
2 const prevNode = this.findPrev(item);
3 if (prevNode.next !== null) {
4 // 指向下一个元素，这行代码很关键
5 prevNode.next = prevNode.next.next;
6 }
7 }
```

- 完整代码

```
1 // 定义单个节点
2 class Node {
3 constructor(el) {
4 this.el = el;
5 this.next = null;
6 }
7 }
```

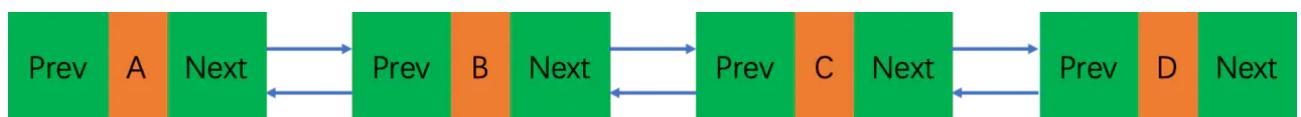
```

8
9 class LinkedList {
10 constructor() {
11 this.head = new Node('head');
12 }
13
14 // 用于查找
15 find(item) {
16 let node = this.head;
17 while (node !== null && node.el !== item) {
18 node = node.next;
19 }
20 return node;
21 }
22
23 findPrev() {
24 let node = this.head;
25 while (node.next !== null && node.next.el !== item) {
26 node = node.next;
27 }
28 return node;
29 }
30
31 // 插入节点
32 insert(el, item) {
33 const newNode = new Node(el);
34 const currentNode = this.find(item);
35 newNode.next = currentNode.next;
36 currentNode.next = newNode;
37 }
38
39 // 删除节点
40 remove(item) {
41 const prevNode = this.findPrev(item);
42 if (prevNode.next !== null) {
43 // 指向下一个元素，这行代码很关键
44 prevNode.next = prevNode.next.next;
45 }
46 }
47 }

```

## 双向链表

双向链表看字面意思，它有两个方向，除了 next 指针指向下一个元素之外，还多了一个 prev 的指针用来指向前面的元素。



我们发现，单链表尽管可以通过前一个元素可以找到下一个元素，但是后面的元素不知道自己前面的是谁。那么如何来解决这个问题呢？其实很简单，只需要再给 Node 添加一个 `prev` 的指针就可以解决问题了。

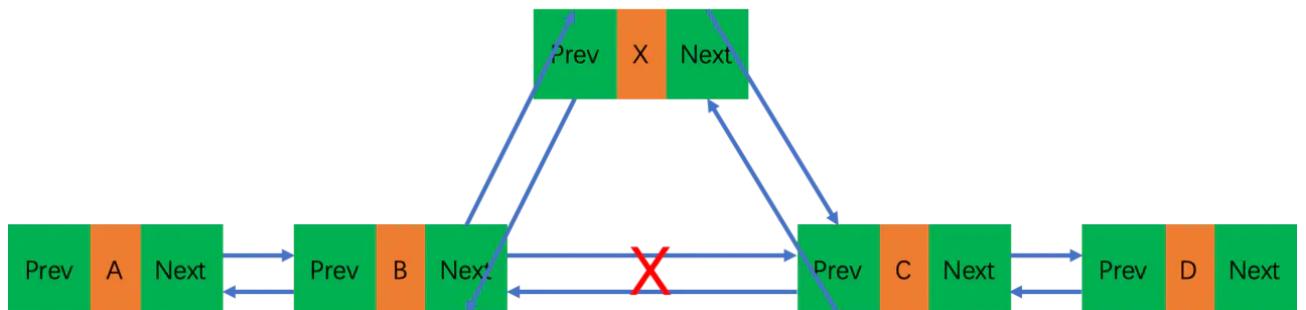
储存同样的数据，双向链表要比单链表占用更多的内存空间。我们知道空间可以换时间，双向链表往往比单链表更加灵活。

## 如何实现一个双链表

和单链表类似，在原有的 `Node` 基础上，增加一个 `prev` 指针。

```
1 class Node {
2 constructor(el) {
3 this.el = el;
4 this.prev = null;
5 this.next = null;
6 }
7 }
```

双向链表的查找和单链表的查找一样，这边不做重复的说明。



实现双向链表的插入和单链表非常类似，多了一个 `prev` 指针，我们对之前的代码稍作修改：

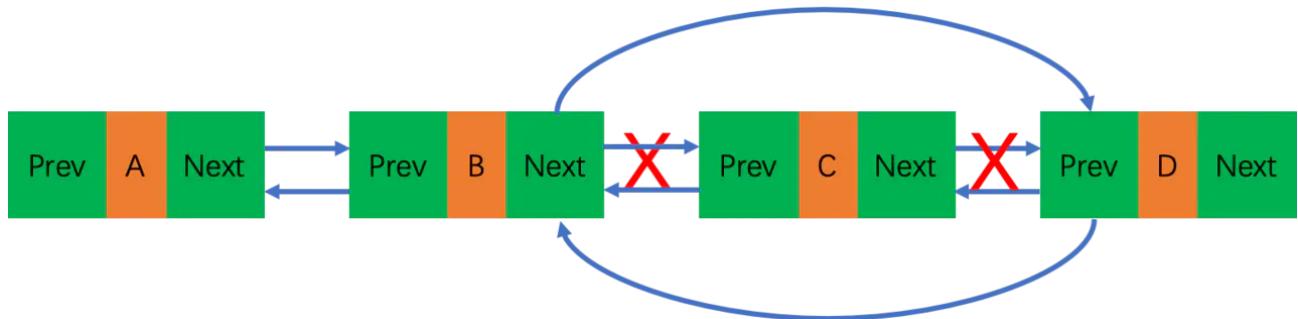
```
1 function insert(el, item) {
2 const newNode = new Node(el);
3 const currentNode = this.find(item);
4 // 将 X 的 next 指向 B 的 next (C)
5 newNode.next = currentNode.next;
6 // 将 X 的 prev 指向 B
7 nextNode.prev = currentNode;
8 // 将 B 的 next 指向 X
```

```

9 currentNode.next = newNode;
10 // 将 C 的 prev 指向 X
11 currentNode.next.prev = newNode;
12 }

```

双向链表的删除比单链表更加简单，不需要再定义 `findPrev` 方法，话不多说，看图看代码。



```

1 function remove(item) {
2 const node = this.find(item);
3 node.prev.next = node.next;
4 node.next.prev = node.prev;
5 node.next = null;
6 node.prev = null;
7 }

```

在实际面试过程中，还需要留意**边界**的情况。写完代码后，问自己 4 个问题：

- 如果链表为空，代码会不会报错？
- 如果只有一个元素，是否会抛异常？
- 两个元素呢？
- 在头部和尾部插入或删除，代码是否还可以正常工作？

下面的表列出零数组和链表在使用时的复杂度差异，在实际应用时，可以根据实际情况来选择。一般来说应用于两种场景：

- 插入和删除非常频繁，并且想要改善
- 不知道有多少元素在，每当有新的元素就链到表的最后面

| 时间复杂度 | 数组     | 链表     |
|-------|--------|--------|
| 插入、删除 | $O(n)$ | $O(1)$ |

---

随机访问

$O(1)$

$O(n)$

---

本章节分为 3 个部分：

- Part 1
  - 回文链表
  - 环形链表
  - 删除链表中的节点
- Part 2
  - 反转链表
  - 删除链表的倒数第N个节点
  - 合并两个有序链表
  - 两数相加
- Part 3
  - 排序链表
  - 相交链表
  - 奇偶链表

阅读完本章节，你将有以下收获：

- 了解各种链表的结构
- 可以在链表中插入、删除、移动元素
- 能够解决一些经典的链表问题

# 回文链表、环形链表、删除链表中的节点

## 回文链表

请判断一个链表是否为回文链表。

### 示例

```
1 输入: 1->2
2 输出: false
3
4 输入: 1->2->2->1
5 输出: true
```

进阶：你能否用  $O(n)$  时间复杂度和  $O(1)$  空间复杂度解决此题？

### 题目分析

- $O(n)$  的时间复杂度意味着只能遍历一趟链表；
- $O(1)$  的空间复杂度意味着只能使用常数个变量（也就是不能使用数组、集合等变量）

### 方法一 字符串拼接比较

#### 思路

通过正向、反向将链表节点值拼接成字符串，最后比较正向、反向字符串是否相同。

#### 详解

1. 定义两个临时变量，存储正、反两个拼接的字符串
2. 遍历链表，进行字符串拼接
3. 比较正、反字符串是否相同

#### 代码

```
1 function isPalindrome (head) {
2 let positiveStr = '';
3 let reverseStr = '';
```

```
4 while (head) {
5 const nodeVal = head.val;
6 // 正向字符串拼接
7 positiveStr += nodeVal;
8 // 反向字符串拼接
9 reverseStr = nodeVal + reverseStr;
10 // 下一个节点
11 head = head.next;
12 }
13
14 return positiveStr === reverseStr;
15 }
```

## 复杂度分析

- 时间复杂度： $O(n)$  过程中只会遍历一遍链表，因此，时间复杂度为  $O(n)$ 。
- 空间复杂度： $O(1)$  过程中产生 2 个临时变量存储，因此，空间复杂度为  $O(1)$ 。

## 方法二 递归解法

### 思路

通过递归的方式逆序遍历链表，同时定义一个全局变量 `pointer` 从前往后正序遍历链表，如果正序和逆序遍历出来的值相等，则为回文链表。

### 详解

1. 首先，定义一个全局变量 `pointer` 指针，初始化值为头部节点 `head`，用于正序遍历。
2. 然后，调用递归函数进行链表的逆序遍历，递归出口为 `head` 为 `null`，表示链表遍历结束，返回 `true`。
3. 如果正序遍历的节点值全部都等于逆序遍历的节点值，那么 `res` 的值一直为 `true`，则递归结束最后的返回值也为 `true`，即为回文链表。反之，则 `res` 为 `false`，不是回文链表。

### 代码

```
1 let pointer;
2
3 function reverseLinkList (head) {
4 if (!head) return true;
5 // 递归逆序遍历
6 const res = reverseLinkList(head.next) && (pointer.val === head.val);
7 // pointer 指针不断向后指，进行正序遍历
8 pointer = pointer.next;
9 }
```

```
9 return res;
10 }
11
12 function isPalindrome (head) {
13 pointer = head;
14 return reverseLinkList(head);
15 }
```

## 复杂度分析

- 时间复杂度： $O(n)$  过程中只对链表进行了一次遍历，因此，时间复杂度为  $O(n)$ 。
- 空间复杂度： $O(1)$  过程中只申请了一个全局变量 `pointer`，因此，空间复杂度为  $O(1)$
- 。

## 方法三 快慢指针

### 思路

找到链表的中间节点，将前半部分的链表反转，与后半部分链表数据进行比较。为了找到中间位置，采用两个引用，步调速度相差 1，当快的引用到达最终节点时，慢的正好在中间。

### 详解

- 分别定义快、慢指针，及前半部分的指针存储
- 遍历链表，快指针走 2 步，慢指针走 1 步，同时将慢指针对应的前半部分链表进行反转
- 链表结束后，慢指针指向中间，与前半部分反转的链表进行逐个比较

### 代码

```
1 function isPalindrome (head) {
2 // 空或者单节点
3 if (!head || !head.next) {
4 return true;
5 }
6
7 let slowRef = head; // 慢指针
8 let fastRef = head; // 快指针
9 let reverseRef; // 反转前半部分
10 let reversePreRef; // 反转前一个节点
11 // 连续 2 个节点都存在
12 while (fastRef && fastRef.next) {
13 // 快指针前进 2 步
14 fastRef = fastRef.next.next;
15 }
```

```

16 reverseRef = slowRef;
17 // 慢指针前进 1 步
18 slowRef = slowRef.next;
19
20 // 反转链表
21 reverseRef.next = reversePreRef;
22 // 记录上一个节点
23 reversePreRef = reverseRef;
24 }
25
26 // 奇数场景
27 if (fastRef) {
28 // 中间值不用比较，慢指针直接前进一步
29 slowRef = slowRef.next;
30 }
31
32 while (reverseRef && slowRef) {
33 // 链表逐个值比较
34 if (reverseRef.val !== slowRef.val) {
35 return false;
36 }
37 reverseRef = reverseRef.next;
38 slowRef = slowRef.next;
39 }
40 return true;
41 }

```

## 复杂度分析

- 时间复杂度： $O(n)$

我们只遍历了包含有  $n$  个元素的列表一次，因此，时间复杂度为  $O(n)$ 。

- 空间复杂度： $O(1)$

产生 4 个临时变量存储，因此，空间复杂度为  $O(1)$ 。

## 环形链表

给定一个链表，判断链表中是否有环。为了表示给定链表中的环，我们使用整数  $pos$  来表示链表尾连接到链表中的位置（索引从 0 开始）。如果  $pos$  是 -1，则在该链表中没有环。

### 示例 1

```

1 输入：head = [3,2,0,-4], pos = 1
2 输出：true
3 解释：链表中有一个环，其尾部连接到第二个节点。

```

## 示例 2

```
1 输入 : head = [1], pos = -1
2 输出 : false
3 解释 : 链表中没有环。
```

## 思路

环形链表一般分为两种，一是头尾相连的循环链表，二是尾部与中间节点相连的6字型链表。我们如何判断一个链表是否是环形链表呢？不管是哪一种环形链表，循环遍历的时候一定是死循环的，那么在链表循环过程中，如果我们不止一次的遇到同一个节点，这个链表就肯定是环形链表。

最常见的有三种方法判断：1. 双指针法 2. 哈希表法 3. 利用 Symbol 的特性

### 方法一 双指针

#### 详解

双指针，即快慢指针。我们定义两个指针，初始位置都是在链表的头部，两个指针同时出发，快指针一次可以前进两步，而慢指针一次只能前进一步。会出现以下几种情况 1. 链表为空，肯定不是环形链表； 2. 链表不为空，快指针走到了链表的结尾，也可以判断不是环形链表； 3. 链表不为空，快指针和慢指针相遇，则证明此链表是环形链表

就像 A 和 B 两个人跑步，A 跑步速度是 B 跑步速度的两倍。两人同时从同一起点开始跑。排除跑道长度为 0 的情况：1. A 跑到了跑道的尽头，此时 B 在跑道的中间，那么这个跑道肯定不是一个环形跑道。2. A 和 B 相遇。在 A、B 速度不同的情况下，如果除起点外还可以再次相遇，那么这个跑道，不管是圆形还是6字型，肯定是环形跑道。

#### 代码

```
1 /**
2 * @param {ListNode} head
3 * @return {boolean}
4 */
5
6 const hasCycle = function (head) {
7 if (!head) return false;
8 let fast = head;
9 let slow = head;
10 while (fast && fast.next) {
11 fast = fast.next.next;
```

```
12 slow = slow.next;
13 if (fast === slow) return true;
14 }
15 return false;
16 };
```

## 复杂度分析

- 时间复杂度： $O(n)$  对于含有  $n$  个元素的单向链表，指针移动到尾部的时间为  $O(n)$ ；对于含有  $n$  个元素的环形链表，指针移动的时间为 第一次指针移动到尾部的时间 + 指针再次移动到两指针相遇的节点的时间 =  $O(n + m)$ ，即  $O(n)$ ；
- 空间复杂度： $O(1)$

## 方法二 哈希表

### 详解

创建一个空 Map 对象并遍历链表中的所有节点，每遍历一个节点，就像空对象里插入一条组键值对为 { 当前节点: 1 }。1. 如果遍历完成，该 Map 对象中不存在相同节点，那么不是环形链表。2. 遍历中，发现该 Map 对象中存在相同节点且值为 1，即该节点已经遍历过了，那么链表是环形链表

### 代码

```
1 /**
2 * @param {ListNode} head
3 * @return {boolean}
4 */
5 const hasCycle = function (head) {
6 if (!head) return false;
7 const newData = new Map();
8 while (head) {
9 if (newData.has(head)) return true;
10 newData.set(head, 1);
11 head = head.next;
12 }
13 return false;
14};
```

## 复杂度分析

- 时间复杂度： $O(n)$  添加一个结点到哈希表中只需要花费  $O(1)$  的时间。对于含有  $n$  个元素的链表，将节点添加到哈希表中至少有  $n$  次，这将耗费  $O(n)$  的时间。
- 空间复杂度： $O(n)$  所需的额外空间取决于哈希表中存储的元素数量，该表最多需要存储  $n$  个元素。

### 方法三 Symbol

#### 详解

Symbol，表示独一无二的值。ES6中新引入的一种数据类型。

和哈希表不一样的是，哈希表是将遍历过节点存到一个 Map 对象中，若循环到一个节点，且对象中存在该节点，则证明为环形链表。而这个方法是将当前节点的 val 值改为用 Symbol 创建的一个独一无二的值，若链表循环过程中存在节点的 val 全等于这个值，那么证明当前不是第一次循环到该节点，即链表为环形链表，反之不是。

#### 代码

```
1 /**
2 * @param {ListNode} head
3 * @return {boolean}
4 */
5 const hasCycle = function (head) {
6 if (!head) return false;
7 const newData = Symbol('');
8 while (head) {
9 if (head.val === newData) return true;
10 head.val = newData;
11 head = head.next;
12 }
13 return false;
14};
```

#### 复杂度分析

- 时间复杂度： $O(n)$  对于含有  $n$  个元素的链表，循环判断并赋值，这将耗费  $O(n)$  的时间。
- 空间复杂度： $O(1)$

### 删除链表中的节点

请编写一个函数，使其可以删除某个链表中给定的（非末尾）节点，你将只被给定要求被删除的节点。

现有一个链表 – head = [4,5,1,9]，它可以表示为：

4 -> 5 -> 1 -> 9

### 示例 1：

```
1 输入: head = [4,5,1,9], node = 5
2 输出: [4,1,9]
3 解释: 给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9
```

### 示例 2：

```
1 输入: head = [4,5,1,9], node = 1
2 输出: [4,5,9]
3 解释: 给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9
```

### 说明:

- 链表至少包含两个节点。
- 链表中所有节点的值都是唯一的。
- 给定的节点为非末尾节点并且一定是链表中的一个有效节点。
- 不要从你的函数中返回任何结果。

### 方法一

#### 思路

这道题通俗地讲就是要删除给定节点。在获取当前节点后，可以将下一个节点的值赋给当前节点，然后将当前节点指向下一个节点，完成删除。

#### 详解

1. `node.value = node->next.value`
2. `node->next=node->next->next`

## 代码

```
1 /**
2 * Definition for singly-linked list.
3 * function ListNode(val) {
4 * this.val = val;
5 * this.next = null;
6 * }
7 */
8 /**
9 * @param {ListNode} node
10 * @return {void} Do not return anything, modify node in-place instead.
11 */
12 const deleteNode = function (node) {
13 node.val = node.next.val;
14 node.next = node.next.next;
15 };
```

## 复杂度分析

- 时间复杂度： $O(1)$   
耗时与输入数据大小无关
- 空间复杂度： $O(1)$   
耗空间与输入数据大小无关

## 方法二

### 思路

利用 Js 的 `Object.assign()` 方法。

`Object.assign()` 方法用于将所有可枚举属性的值从一个或多个源对象复制到目标对象。它将返回目标对象。简单来说，`Object.assign(a,b)` 能合并两个对象(a和b)，并覆盖到第一个参数(a)所指的地址上。

由于 Js 的对象都是内存引用，也就是说 `node` 这个变量里面，只保存了内存地址。因此可以用 `Object.assign()`，使得 `node.next` 覆盖 `node`.

### 详解

将 `node` 和 `node.next` 合并，并保存到 `node` 所指的内存地址上。

## 代码

```
1 /**
2 * Definition for singly-linked list.
3 * function ListNode(val) {
4 * this.val = val;
5 * this.next = null;
6 * }
7 */
8 /**
9 * @param {ListNode} node
10 * @return {void} Do not return anything, modify node in-place instead.
11 */
12 const deleteNode = function (node) {
13 Object.assign(node, node.next);
14 };
```

## 复杂度分析

- 时间复杂度： $O(1)$   
删除节点只改变指针的指向，与数据大小无关。
- 空间复杂度： $O(1)$   
只对原有的参数空间进行了操作。

# 反转链表、删除链表的倒数第N个节点、合并两个有序链表和两数相加

## 反转链表

反转一个单链表。

### 示例

```
1 输入: 1->2->3->4->5->NULL
2 输出: 5->4->3->2->1->NULL
```

### 方法一

#### 思路

用迭代的方法实现。

#### 详解

1. 先判断链表是否为空或者只有一个元素，是的话直接返回；
2. 若链表不仅有一个元素，先开辟一个空间存头指针，再把旧的头指针变为新的尾指针；
3. 遍历链表，申请一个新的临时空间用于前后元素交换即可。

```
1 const reverseList = function (head) {
2 if (head === null || head.next === null) {
3 return head;
4 }
5 let p = head.next;
6 head.next = null; // 旧的头指针是新的尾指针，next需要指向null
7 while (p !== null) {
8 const temp = p.next; // 先保留下一个step要处理的指针
9 p.next = head; // 然后p和head进行反向
10 head = p; // 指针后移
11 p = temp; // 指针后移
12 }
13 return head;
14};
```

## 复杂度分析

- 时间复杂度： $O(n)$

上述解法中，遍历了一次链表，这将耗费  $O(n)$  的时间。

- 空间复杂度： $O(1)$

上述解法中，申请了两个额外的临时存储空间，这将耗费  $O(1)$  的空间。

## 方法二

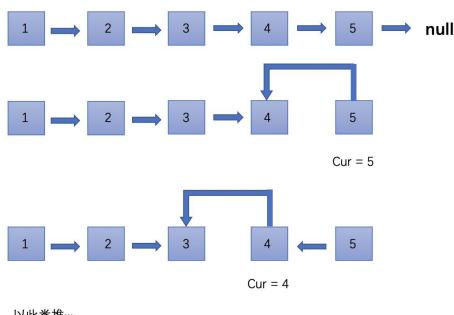
### 思路

用递归的方法实现。

### 详解

- 先判断链表是否为空或只有一个元素，是的话直接返回；
- 若链表不仅有一个元素，则递归的调用链表反转方法。

递归演示：



```
1 const reverseList = function (head) {
2 if (head === null || head.next === null) {
3 return head;
4 }
5 //这里的cur就是最后一个节点，也就是反转后的头节点
6 const newHead = reverseList(head.next); // 反转后的头节点
7 //这里请配合动画演示理解
8 //如果链表是 1->2->3->4->5，那么此时的cur就是5
9 //而head是4，head的下一个是5，下一个是空
10 //所以head.next.next 就是5->4
11 head.next.next = head; // 将反转后的链表的尾结点与当前节点相连
12 //防止链表循环，需要将head.next设置为空
13 head.next = null;
```

```
14 //每层递归函数都返回cur，也就是最后一个节点
15 return newHead;
16 }
```

## 复杂度分析

- 时间复杂度： $O(n)$

上述解法中，遍历了一次链表，这将耗费  $O(n)$  的时间。

- 空间复杂度： $O(n)$

上述解法中，使用了递归的方法，这将耗费  $O(n)$  的空间。

## 删除链表的倒数第N个节点

给定一个链表，删除链表的倒数第  $n$  个节点，并且返回链表的头结点。

### 示例

```
1 给定一个链表：1->2->3->4->5，和 n = 2。
2
3 当删除了倒数第二个节点后，链表变为 1->2->3->5.
```

### 方法一 双指针法

#### 思路

先用 `first` 指针前进  $n$ ，然后让 `second` 从 `head` 开始和 `first` 一起前进，直到 `first` 到了最后，此时 `second` 的下一个节点就是要删除的节点；如果 `first` 一开始前进  $n$  就已经不在链表中了，说明要删除的节点正是 `head` 节点，那么直接返回 `head` 的下一个节点。

#### 详解

1. 指针 `first` 指向头节点，然后，让其向后移动  $n$  步。
2. 指针 `second` 指向头结点，并和 `first` 一起向后移动。当 `first` 的 `next` 指针为 `null` 时，`second` 即指向了要删除节点的前一个节点。
3. 指针 `first` 的 `next` 指向要删除节点的下一个节点。

```

1 const removeNthFromEnd = (head, n) => {
2 let first = head;
3 let second = head;
4 while (n > 0) {
5 first = first.next;
6 n -= 1;
7 }
8 if (!first) return head.next; // 如果first为null，则要删除的节点是首节点，直接返回head
9 while (first.next) {
10 first = first.next;
11 second = second.next;
12 }
13 second.next = second.next.next;
14 return head;
15 };

```

## 复杂度分析

- 时间复杂度： $O(n)$  该算法对含有  $n$  个结点的列表进行了单层遍历，因此，时间复杂度为  $O(n)$
- 空间复杂度： $O(1)$  该算法只用了常量级的额外空间。故空间复杂度为  $O(1)$

## 方法二 单向链表成为双向链表

### 思路

可以遍历一次，让单向链表成为双向链表，先找到其尾节点，然后遍历整个链表同时  $n$  做递减操作，相当于从最后一个节点向前查找，直到  $n=1$  时，此时的节点就是我们要找的节点，然后直接删除即可。

### 详解

1. 指针  $cur$  指向头节点，并定义  $cur.prev$ 、 $cur.next$  使其成为双向链表。
2. 找到其尾节点，当  $cur.next$  不存在时，则当前节点  $cur$  为尾节点。
3. 遍历链表同时向前推进。 $n$  做递减，当  $n=1$  时就是我们要删除的节点位置，否则，就让节点向前推进一个节点，直到  $n=1$  删除当前节点。（同时要考虑要删除的节点位置刚好是头节点的时候）

```

1 const removeNthFromEnd = (head, n) => {
2 let cur = head;
3 while (cur.next) {
4 cur.next.prev = cur;

```

```

5 cur = cur.next;
6 }
7 if (n === 1) {
8 // 删除最后一个节点
9 if (!cur.prev) { // 若是头节点则直接返回null
10 return null;
11 } else {
12 cur.prev.next = null;
13 return head;
14 }
15 }
16 while (n > 0 && cur) {
17 if (n === 1) {
18 if (!cur.prev) {
19 // 删除第一个节点
20 cur.next.prev = null;
21 return cur.next;
22 } else {
23 cur.prev.next = cur.next;
24 cur.next.prev = cur.prev;
25 return head;
26 }
27 }
28 cur = cur.prev;
29 n -= 1;
30 }
31 };

```

## 复杂度分析

- 时间复杂度： $O(n)$  该算法对含有  $n$  个结点的列表进行了单层遍历，因此，时间复杂度为  $O(n)$ 。
- 空间复杂度： $O(1)$  该算法只用了常量级的额外空间，因此，空间复杂度为  $O(1)$ 。

## 合并两个有序链表

将两个有序链表合并为一个新的有序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

### 示例

```

1 输入：1->2->4, 1->3->4
2 输出：1->1->2->3->4->4

```

## 方法一 递归法

### 思路

用递归的方式，依次比较两个链表中首项的大小，保留数值小的为链表当前值，直到一个链表参数为空则结束。

### 详解

1. 递归处理两个入参链表
2. 若两个链表中有一个链表为空，则返回另一个链表
3. 两个链表都不为空时比较两个链表中第一个节点的值，保留较小者（相同则均可）
4. 继续递归执行去掉该节点的链表和另一个链表直至其中一个链表为空

### 代码

```
1 const mergeTwoLists = function (l1, l2) {
2 if (l1 === null) {
3 return l2;
4 }
5 if (l2 === null) {
6 return l1;
7 }
8 if (l1.val <= l2.val) {
9 l1.next = mergeTwoLists(l1.next, l2);
10 return l1;
11 } else {
12 l2.next = mergeTwoLists(l1, l2.next);
13 return l2;
14 }
15};
```

### 复杂度分析

- 时间复杂度： $O(n + m)$   
 $n$  和  $m$  分别是两个链表的长度，这将耗费  $O(n + m)$  的时间。
- 空间复杂度： $O(n + m)$   
每次比较值大小的时候，都会递归调用一次，耗费  $O(1)$  的栈空间，因此空间复杂度为  $O(n + m)$

## 方法二 双指针法

## 思路

创建一个新链表，通过判断两个链表当前值，将较小值放到新链表的下个节点，较小值的链表重新赋值为其下一节点，直到参数链表都为空时结束。

## 详解

1. 创建一个新链表
2. 当两个链表不都为空时执行以下循环
3. 判断两个链表的第一个节点，取较小值放入新链表，原链表去掉该节点
4. 直到两个链表都为空时循环结束，返回新链表

## 代码

```
1 const mergeTwoLists = function (l1, l2) {
2 const prevHead = new ListNode(-1);
3 let prevNode = prevHead;
4 while (l1 !== null && l2 !== null) {
5 if (l1.val <= l2.val) {
6 prevNode.next = l1;
7 l1 = l1.next;
8 } else {
9 prevNode.next = l2;
10 l2 = l2.next;
11 }
12 prevNode = prevNode.next;
13 }
14 prevNode.next = l1 || l2;
15 return prevHead.next;
16};
```

## 复杂度分析

- 时间复杂度： $O(n + m)$   
 $n$  和  $m$  分别是两个链表的长度，时间复杂度为  $O(n + m)$
- 空间复杂度： $O(1)$   
用申请了一个指针空间，其空间复杂度为  $O(1)$ 。

## 两数相加

给出两个非空的链表用来表示两个非负的整数。其中，它们各自的位数是按照 逆序 的方式存储的，并且它们的每个节点只能存储一位 数字。

如果，我们将这两个数相加起来，则会返回一个新的链表来表示它们的和。

您可以假设除了数字 0 之外，这两个数都不会以 0 开头。

## 示例

```
1 输入 : (2 -> 4 -> 3) + (5 -> 6 -> 4)
2 输出 : 7 -> 0 -> 8
3 原因 : 342 + 465 = 807
```

## 方法一 游标法

### 思路

我们使用游标变量来跟踪进位，并从包含最低有效位的表头开始模拟逐位相加的过程。

### 详解

1. 设置游标变量 pointer1 和 pointer2，分别指向需要相加的两个链表 L1 和 L2；新建一个链表为 sumListNode，存储 L1 和 L2 逐位相加的结果。
2. 逐步移动 pointer1 和 pointer2，对 L1 和 L2 的每个节点相加得到两数之和 sumListNode 的链表节点

```
1 /**
2 * Definition for singly-linked list.
3 */
4 function ListNode (val) {
5 this.val = val;
6 this.next = null;
7 }
8 /**
9 * @param {ListNode} l1
10 * @param {ListNode} l2
11 * @return {ListNode}
12 */
13 const addTwoNumbers = function (l1, l2) {
14 // 两数之和联表
15 const sumListNode = new ListNode(0);
16 // 指针1 指向 链表1
17 let pointer1 = l1;
18
19 // 指针2 指向 链表2
20 let pointer2 = l2;
```

```

21
22 // current 指向 两树之和的链表
23 let current = sumListNode;
24
25 // 逐位相加的进位
26 let carry = 0;
27
28 // 如果指针 pointer1、pointer2 还未移动结束
29 while (pointer1 || pointer2) {
30 // 如果 pointer1 已经移动到链表 l1 的末尾，当前值为0
31 const num1 = pointer1 ? pointer1.val : 0;
32
33 // 如果 pointer1 已经移动到链表 l1 的末尾，当前值为0
34 const num2 = pointer2 ? pointer2.val : 0;
35
36 // sum为当前移动位的两数之和
37 const sum = carry + num1 + num2;
38
39 // 存储进位
40 carry = Math.floor(sum / 10);
41
42 // sumListNode添加一个当前 l1 和 l2 相加的node ，值为 sum的个位数
43 current.next = new ListNode(sum % 10);
44
45 // current 指针后移一位
46 current = current.next;
47
48 // 如果 pointer1 未移动到 l1 的结尾，继续后移一位
49 if (pointer1) {
50 pointer1 = pointer1.next;
51 }
52 // 如果 pointer2 未移动到 l2 的结尾，继续后移一位
53 if (pointer2) {
54 pointer2 = pointer2.next;
55 }
56 }
57 //如果有进位，两数之和的联表，加一个进位的 node
58 if (carry > 0) {
59 current.next = new ListNode(carry);
60 }
61
62 return sumListNode.next;
63 };

```

## 复杂度分析

- 时间复杂度： $O(n)$   
假设  $m$  和  $n$  分别表示  $L1$  和  $L2$  的长度，上面的算法最多重复  $\max(m, n)$  次。
- 空间复杂度： $O(n)$   
新列表的长度最多为  $\max(m, n)$ 。

## 方法二 数字相加法

### 思路

本题目为链表模拟的两个数字相加，那么可以现将链表转换为数字，数字相加，最后把结果转换为链表。由于javascript 数字范围为  $-2^{53} \sim 2^{53}$ ，考虑存在过大的数字相加，此处数字使用BigInt类型 [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/BigInt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt)。

### 详解

两数相加：1. 将 L1 转换为数字 num1，将 L2 转换为数字 num2，2. 将数字 num1 和 num2 相加得出两数之和 sumNum 3. 最后将 sumNum 转换为链式结构

```
1 /**
2 * Definition for singly-linked list.
3 */
4 function ListNode (val) {
5 this.val = val;
6 this.next = null;
7 }
8 /**
9 * @description 将链表转换为数字
10 * @param {ListNode} listNode
11 * @return {BigInt}
12 */
13 const listNodeToNum = function (listNode) {
14 let numString = '';
15 let currentNode = listNode;
16 while (currentNode) {
17 numString = currentNode.val + numString;
18 currentNode = currentNode.next;
19 }
20 // eslint-disable-next-line no-undef
21 return BigInt(numString);
22 };
23 /**
24 * @description 数字转为链表
25 * @param {number} num 数字
26 * @return {ListNode}
27 */
28 const numToListNode = function (num) {
29 let listNode = null;
30 const numString = num.toString();
31 for (let i = 0; i < numString.length; i++) {
32 const newNode = new ListNode(numString[i]);
33 newNode.next = listNode;
34 listNode = newNode;
35 }
36 }
```

```
37 return listNode;
38 };
39
40 const addTwoNumbers = function (l1, l2) {
41 return numToListNode(listNodeToNum(l1) + listNodeToNum(l2));
42 };
```

## 复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(n)$

# 排序链表、相交链表和奇偶链表

## 排序链表

在  $O(n * \log(n))$  时间复杂度和常数级空间复杂度下，对链表进行排序。

示例1：

```
1 输入: 4->2->1->3
2 输出: 1->2->3->4
```

示例2：

```
1 输入: -1->5->3->4->0
2 输出: -1->0->3->4->5
```

分析：

题目看完就两个要求：

1. 时间复杂度： $O(n * \log(n))$
2. 空间复杂度： $O(1)$

看到这里如果你脑海里没有各种排序的时空间复杂度表，就看下图。这道题的难点在于选择什么算法，先考虑时间复杂度，满足条件的只有堆排序、快排和归并排序。

此外，本题的对象是链表。当数列以链表的形式存储的时候，归并排序就不需要额外申请 $O(n)$ 级别的空间。此时它的空间复杂度是  $O(1)$ 。而快排和堆排序虽然都是速度很快的排序，但在链表中不是很合适。所以这道题优先选择归并排序来做。

## 方法一

### 思路

使用归并排序的方法实现。

## 详解

1. 先判断是否只有一个元素，若只有一个元素，直接返回；
2. 若不只有一个元素，首先找到链表的中间节点；
3. 然后递归的对前半部分链表和后半部分链表分别进行递归排序；
4. 最后对两个子链表进行归并操作。

```
1 const sortList = function (head) {
2 // 只有一个元素
3 if (head === null || head.next === null) {
4 return head;
5 }
6 // 快慢双指针
7 let slow = head; let fast = head;
8 while (slow.next && fast.next && fast.next.next) {
9 slow = slow.next;
10 fast = fast.next.next;
11 }
12 const middle = slow.next;
13 slow.next = null;
14 // 一分为二
15 const left = head;
16 const right = middle;
17 return merge(sortList(left), sortList(right));
18 };
19 const merge = function (left, right) {
20 const tmp = new ListNode(null);
21 let p1 = left; let p2 = right; let p = tmp;
22 while (p1 && p2) {
23 if (p1.val < p2.val) {
24 const s = p1;
25 p1 = p1.next;
26 s.next = null;
27 p.next = s;
28 p = s;
29 } else {
30 const s = p2;
31 p2 = p2.next;
32 s.next = null;
33 p.next = s;
34 p = s;
35 }
36 }
37 if (p1) p.next = p1;
38 if (p2) p.next = p2;
39 return tmp.next;
40 };
41 function ListNode (val) {
42 this.val = val;
43 this.next = null;
```

```
44 }
```

## 复杂度分析

- 时间复杂度： $O(n \log n)$

上述解法中，采用了归并排序的方法，归并排序时间复杂度  $O(n \log n)$ 。

- 空间复杂度： $O(1)$

上述解法中，申请了两个额外的临时存储空间，这将耗费  $O(1)$  的空间。

## 方法二

### 思路

借助数组实现，方法取巧。

### 详解

- 先判断是否只有一个元素，若只有一个元素，直接返回；
- 若不只有一个元素，首先把链表转为数组；
- 然后把数组排序后重建链表，方法取巧。

```
1 const sortList = function (head) {
2 // 只有一个元素
3 if (head === null || head.next === null) {
4 return head;
5 }
6 let cur = head; let index = 0; const arr = [];
7 // 链表转化为数组
8 while (cur !== null) {
9 arr[index] = cur.val;
10 cur = cur.next;
11 index += 1;
12 }
13 arr.sort((a, b) => a - b); // 数组升序排序
14 cur = head;
15 index = 0;
16 // 重建链表
17 while (cur !== null) {
18 cur.val = arr[index];
19 index += 1;
}
```

```
20 cur = cur.next;
21 }
22 return head;
23 };
```

## 复杂度分析

- 时间复杂度： $O(n \log n)$

上述解法中，遍历两遍链表，时间复杂度  $O(n \log n)$ 。

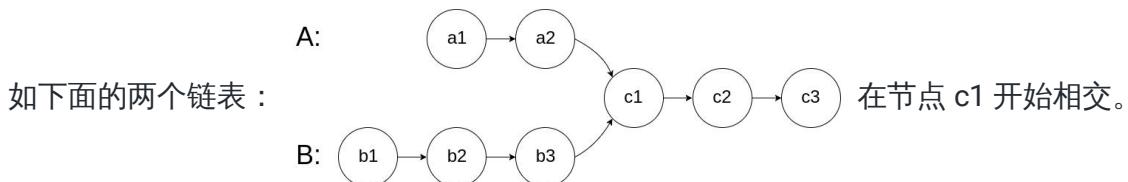
- 空间复杂度： $O(n)$

上述解法中，申请了两个额外的数组空间，耗费的空间数量取决于链表的长度  $n$ ，空间复杂度  $O(n)$ 。

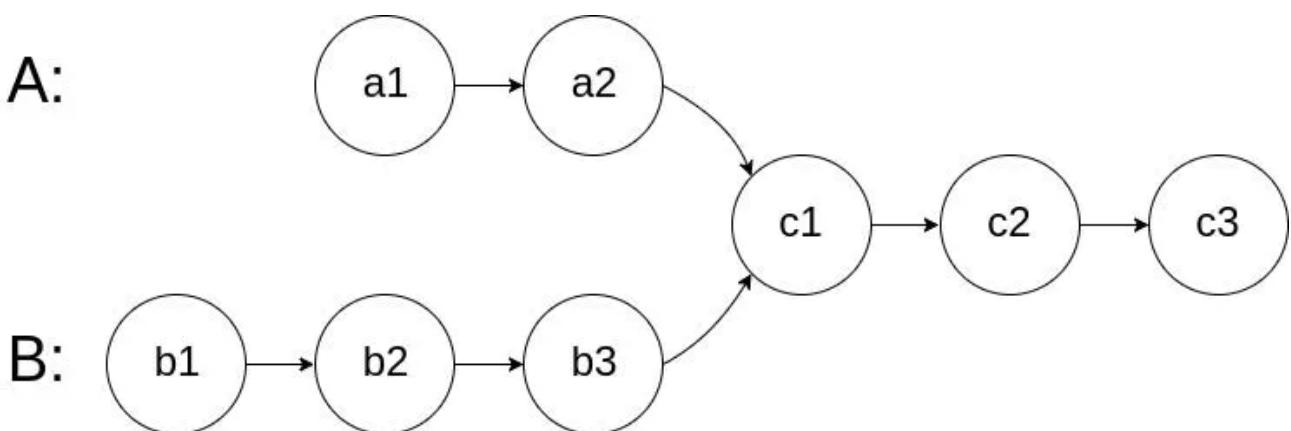
## 相交链表

### 示例

编写一个程序，找到两个单链表相交的起始节点。



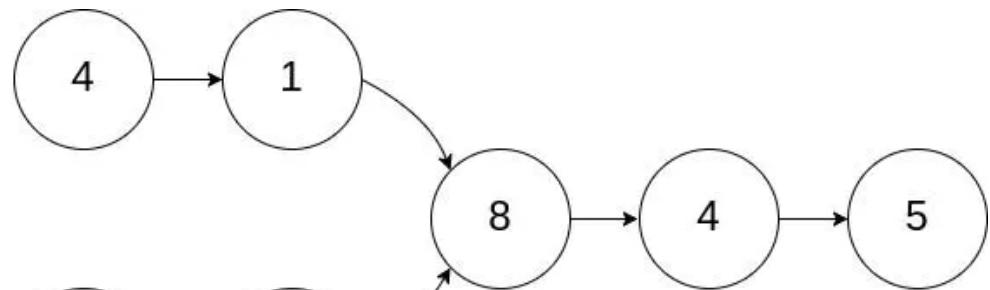
### 示例 1：



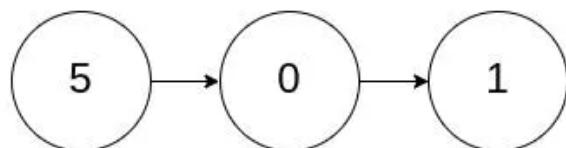
```
1 输入 : intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
2 输出 : Reference of the node with value = 8
3 输入解释 : 相交节点的值为 8 (注意, 如果两个列表相交则不能为 0)。从各自的表头开始算起, 链表
```

示例 2 :

A:



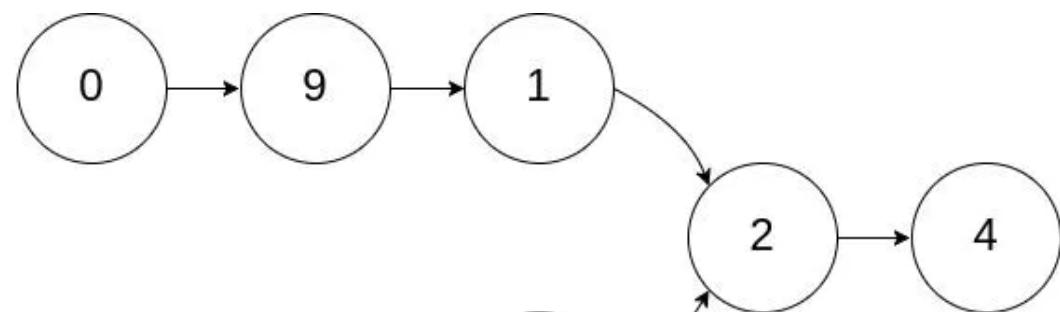
B:



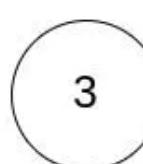
```
1 输入 : intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 2
2 输出 : Reference of the node with value = 2
3 输入解释 : 相交节点的值为 2 (注意, 如果两个列表相交则不能为 0)。从各自的表头开始算起, 链表
```

示例 3 :

A:



B:



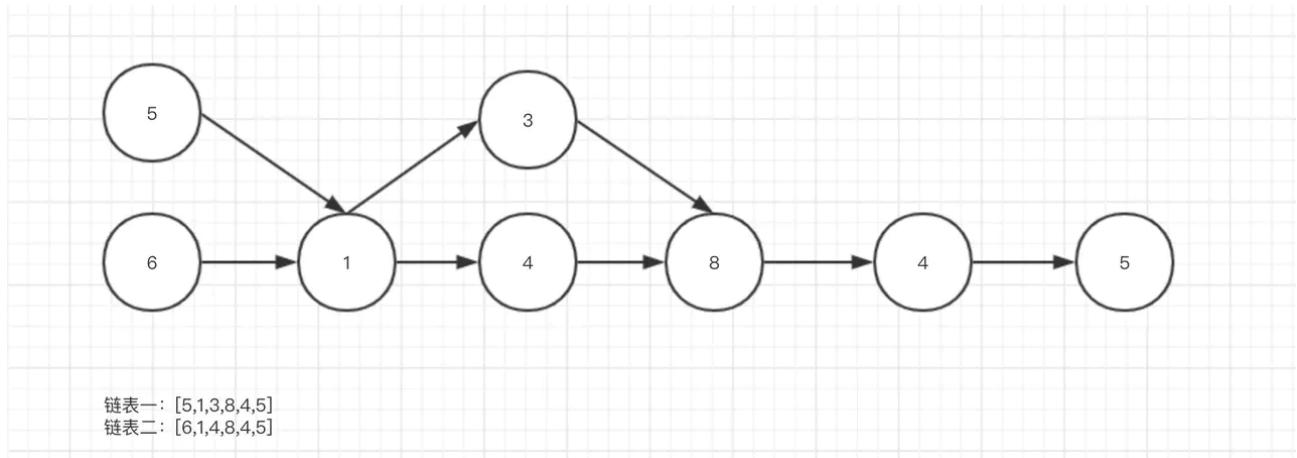
```
1 输入 : intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
2 输出 : null
3 输入解释 : 从各自的表头开始算起, 链表 A 为 [2,6,4] , 链表 B 为 [1,5]。由于这两个链表不相交
4 解释 : 这两个链表不相交, 因此返回 null。
```

注意：

- 如果两个链表没有交点，返回 `null` .
- 在返回结果后，两个链表仍须保持原有的结构。
- 可假定整个链表结构中没有循环。
- 程序尽量满足  $O(n)$  时间复杂度，且仅用  $O(1)$  内存。

问题分析：

- 两条链表相交，交点开始必定所有的节点值都相等。
- 链表相交不仅仅是链表的值相同，而是链表的引用都相同，所以只要某个节点开始相等，就会一直相等。所以下图这种情况不存在。



## 方法一 暴力破解法

### 思路

如果给的数据结构是双向链表，很容易得到解法，两条链表从末尾开始遍历，直到链表同一个位置的两个值不相等即可。既然数据结构定了单向链表这种方法就不考虑了。

如果两条链表是一样的长度也很好得到解法，节点逐一比较，直到末尾节点值都是相等的就说明是相交点。我们可以按照此思路，先将两条链表处理成相同的长度在进行比较。

### 详解

1. 计算链表长度
2. 将较长的那条链表的长度调整为较短的那条的长度
3. 继续遍历找出相交点

```
1 const getIntersectionNode = function (headA, headB) {
2 if (headA === null || headB === null) return null;
3
4 let pA = headA;
5 let pB = headB;
6
7 // 第一步：计算链表的长度
8 let lenA = 0;
9 let lenB = 0;
10 while (pA !== null) {
11 lenA += 1;
12 pA = pA.next;
13 }
14 while (pB !== null) {
15 lenB += 1;
16 pB = pB.next;
17 }
18 let lenDiff = lenA - lenB;
19
20 // 第二步：将较长的那条链表的长度调整为较短的那条的长度
21 // 若链表a比较长，需要调整a链表
22 pA = headA;
23 pB = headB;
24 if (lenDiff > 0) {
25 while (lenDiff !== 0) {
26 pA = pA.next;
27 lenDiff -= 1;
28 }
29 } else {
30 // 若链表b比较长，需要调整b链表
31 while (lenDiff !== 0) {
32 pB = pB.next;
33 lenDiff += 1;
34 }
35 }
36
37 // 第三步：继续遍历找出相交点
38 while (pA !== null) {
39 if (pA === pB) {
40 return pA;
41 }
42 pB = pB.next;
43 pA = pA.next;
44 }
45 return null;
46};
```

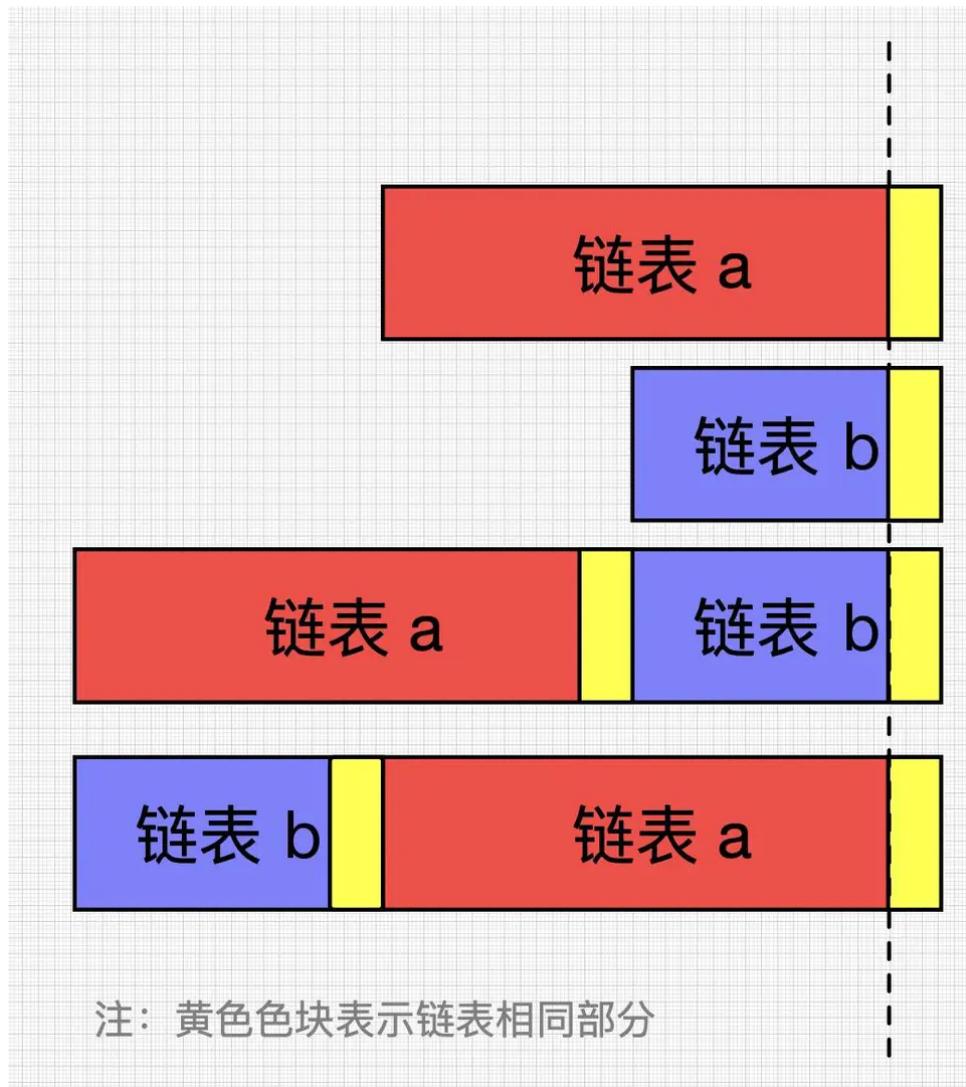
## 复杂度分析

- 时间复杂度： $O(n)$ ，因为最差情况会完整遍历一遍链表，时间复杂度与链表长度呈线性正相关
- 空间复杂度： $O(1)$ ，因为只开辟了固定个数的变量空间，与输入值无关

## 方法二 双指针法

### 思路

通过加法的手段消除长度差。将两链表首尾相接形成 ab 和 ba 链表，此时我们构建了两条长度相同的链表，若 a 和 b 相交，则 ab 和 ba 也必定相交。



### 详解

1. 定义两个指针 pA 和 pB；
2. pA 从链表 a 的头部开始走，走完后再从链表 b 的头部开始走；
3. pB 从链表 b 的头部开始走，走完后再从链表 a 的头部开始走；

#### 4. 如果存在相交结点，则两个指针必会相遇

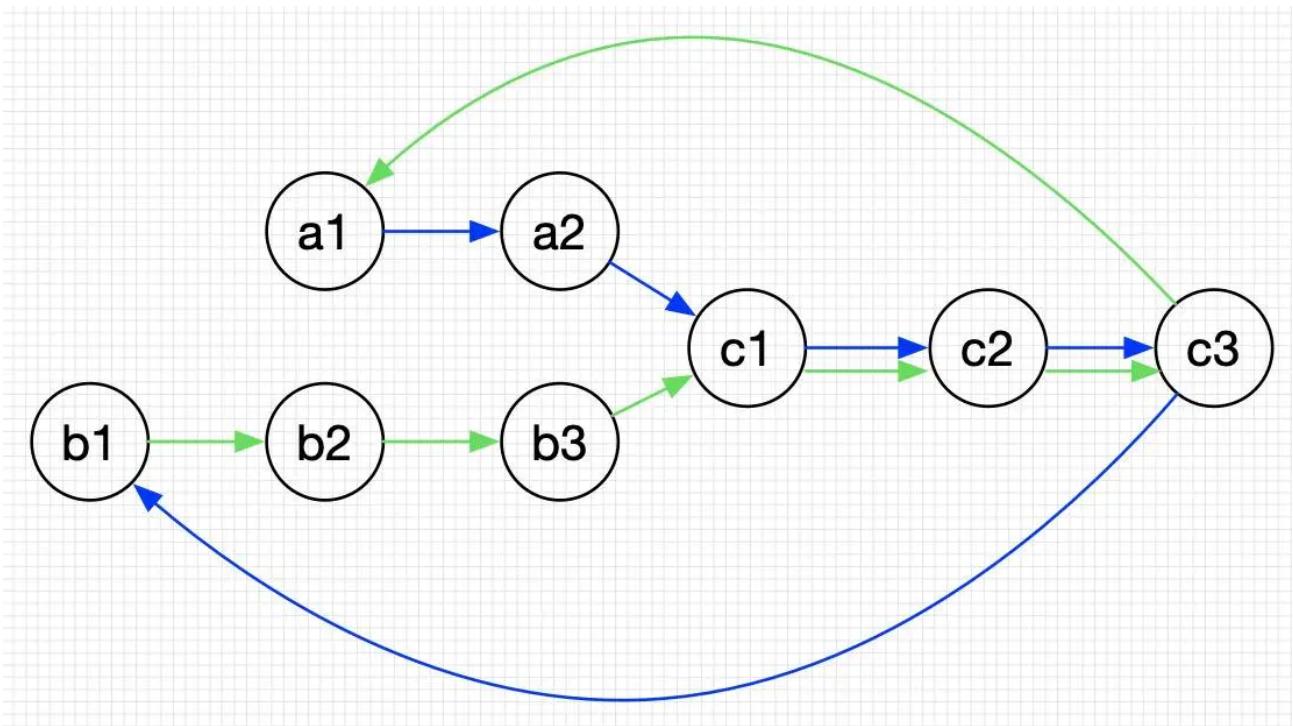
```
1 const getIntersectionNode = function (headA, headB) {
2 if (headA === null || headB === null) return null;
3
4 let pA = headA;
5 let pB = headB;
6 while (pA !== pB) {
7 pA = pA === null ? headB : pA.next;
8 pB = pB === null ? headA : pB.next;
9 }
10 return pA;
11};
```

### 复杂度分析

- 时间复杂度： $O(n)$   
因为会完整遍历一遍链表，时间复杂度与链表长度呈线性关系
- 空间复杂度： $O(1)$   
因为只开辟了固定个数的变量空间，与输入值无关

### 奇偶链表

给定一个单链表，把所有的奇数节点和偶数节点分别排在一起。请注意，这里的奇数节点和偶数节点指的是节点编号的奇偶性，而不是节点的值的奇偶性。请尝试使用原地算法完成。你的算法的空间复杂度应为  $O(1)$ ，时间复杂度应为  $O(nodes)$ ，nodes 为节点总数。



- 应当保持奇数节点和偶数节点的相对顺序。
- 链表的第一个节点视为奇数节点，第二个节点视为偶数节点，以此类推。

## 示例

```

1 输入: 1->2->3->4->5->NULL
2 输出: 1->3->5->2->4->NULL

```

```

1 输入: 2->1->3->5->6->4->7->NULL
2 输出: 2->3->6->7->1->5->4->NULL

```

## 方法一 奇偶链表分离法

### 思路

将链表中所有元素按照奇数位置、偶数位置划分为两个链表：odd 链表、event 链表，遍历结束，直接将偶数链表挂在奇数链表之后。

### 详解

1. 如果链表中节点个数为 0、1、2 个时，链表自身已满足奇偶链表，直接返回 head 节点即可；

2. 定义 odd 变量指向头节点、even 和 evenHeadPointer 变量指向链表的第二个节点，其中 head 即代表奇数链表的头节点、evenHeadPointer 即代表偶数链表的头节点；
3. while 循环遍历链表（利用 odd、even 变量遍历），利用原链表中奇数位置节点的子节点应该挂到偶链表中、偶数位置节点的子节点应该挂到奇链表中交叉遍历赋值，odd、even 变量永远指向奇链表、偶链表最后一个节点；
4. 奇链表最后一个节点 odd 的子节点指向偶链表的头节点 evenHeadPointer；
5. 返回 head 头节点即可；

```

1 /**
2 * Definition for singly-linked list.
3 * function ListNode(val) {
4 * this.val = val;
5 * this.next = null;
6 * }
7 */
8 /**
9 * @param {ListNode} head
10 * @return {ListNode}
11 */
12 const oddEvenList = function (head) {
13 if (head === null || head.next === null || head.next.next === null) {
14 return head;
15 }
16 let odd = head;
17 let even = head.next;
18 const evenHeadPointer = head.next;
19 while (even != null && even.next != null) {
20 odd.next = even.next;
21 odd = odd.next;
22 even.next = odd.next;
23 even = even.next;
24 }
25 odd.next = evenHeadPointer;
26 return head;
27 };

```

## 复杂度分析

- 时间复杂度： $O(n)$   
while 循环遍历一次链表，以第一个节点作为奇数链表的头节点，第二个节点作为偶数链表的头节点，交叉串联起奇数、偶数链表，时间复杂度为  $O(n)$
- 空间复杂度： $O(1)$  新增三个变量用于存储链表头节点，空间复杂度： $O(1)$

## 方法二 数组暂存法

## 思路

遍历链表并利用数组暂存链表节点，然后在数组中对奇数、偶数位置的节点进行串联；

## 详解

1. 如果链表中节点个数为 0、1、2 个时，链表自身已满足奇偶链表，直接返回 head 节点即可；
2. 定义一个数组暂存链表节点；
3. while 循环遍历链表（利用 head 变量遍历），将每一个节点 push 到数组中，并且从第三个节点开始，将第三个节点作为子节点挂到第一个节点上，第四个节点作为子节点挂到第二个节点上，以此类推；
4. 遍历到最后一个节点时，倒数第二个节点的子节点赋值为 null；
5. 如果数组长度为偶数个，则数组的倒数第二个元素是奇链表的最后一个节点，如果数组长度为奇数个，则数组的最后一个元素是奇链表的最后一个节点；
6. 数组的第二个元素是偶链表的头节点；
7. 串联奇偶链表，直接将奇链表的最后一个节点的 next 指向偶链表的头节点；
8. 返回数组的第一个元素即可；

```
1 /**
2 * Definition for singly-linked list.
3 * function ListNode(val) {
4 * this.val = val;
5 * this.next = null;
6 * }
7 */
8 /**
9 * @param {ListNode} head
10 * @return {ListNode}
11 */
12 const oddEvenList = function (head) {
13 // 如果链表中元素个数少于2个，直接返回链表
14 if (head === null || head.next === null || head.next.next === null) {
15 return head;
16 }
17 // 为了防止链表节点丢失，利用一个数组暂存链表
18 const linkArr = [];
19 while (head != null) {
20 linkArr.push(head);
21 const len = linkArr.length;
22 // 从第三个节点开始处理next
23 if (len > 2) {
24 linkArr[len - 3].next = linkArr[len - 1];
25 }
26 head = head.next;
27 if (head === null) {
28 linkArr[len - 2].next = null;
29 }
30 }
31 }
```

```
30 const isOdd = len % 2 !== 0;
31 if (!isOdd) {
32 linkArr[len - 2].next = linkArr[1];
33 } else {
34 linkArr[len - 1].next = linkArr[1];
35 }
36 }
37 return linkArr[0];
38 };
```

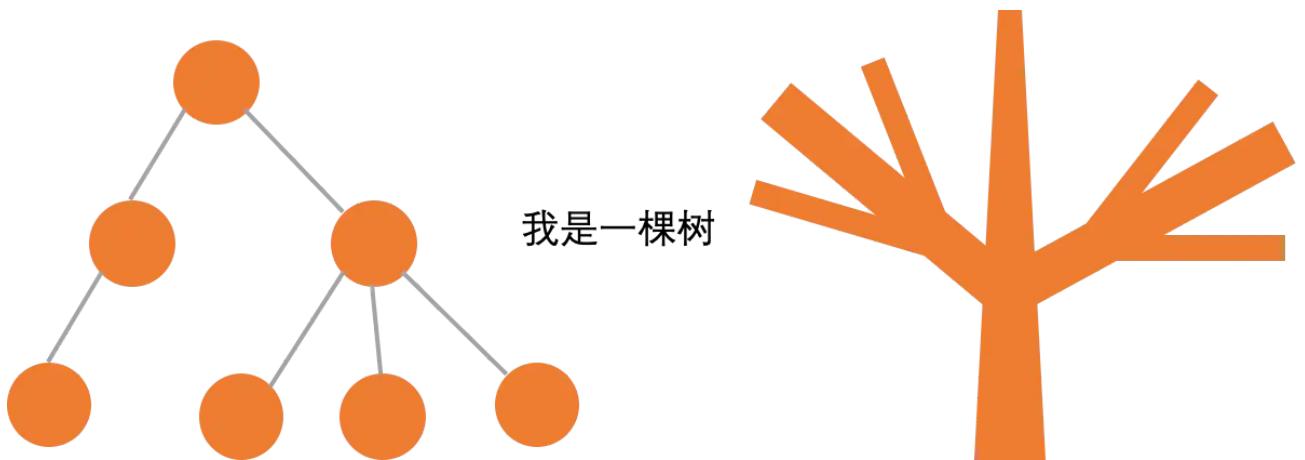
## 复杂度分析

- 时间复杂度： $O(n)$   
while 循环遍历一次链表，从第三个节点开始处理 `next`，时间复杂度： $O(n)$
- 空间复杂度： $O(n)$   
借用一个跟链表等长的数组暂存链表元素，空间复杂度为  $O(n)$

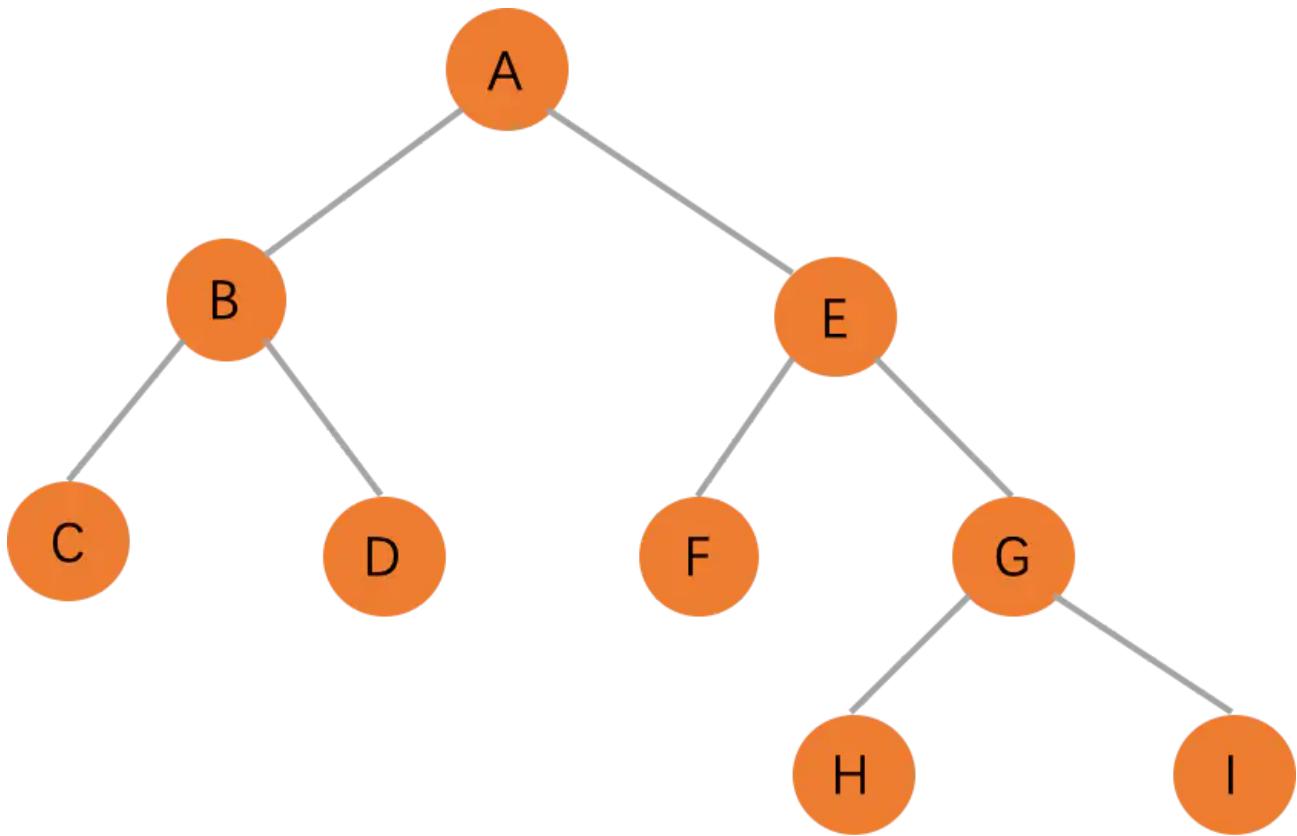
# 二叉树

## 树(Tree)

树是计算机中经常用到的一种数据结构，与列表不同，它是一种非线性的数据结构，以分层的方式来储存数据。像公司的组织架构，就可以理解成一棵树。



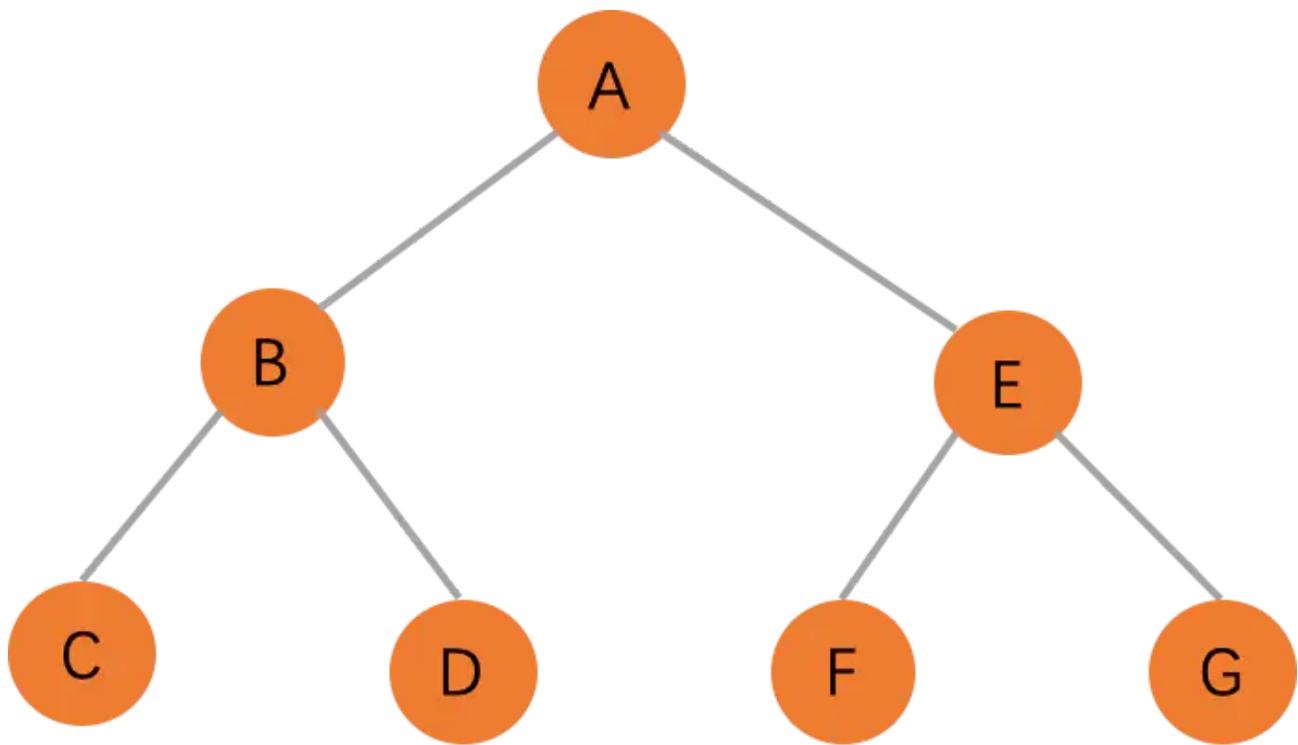
一棵树最上面的节点被称为根节点，在下图中，`A` 就是根节点。如果一个节点下面连接多个节点，该节点被称为父节点，它下面的节点被成为子节点，一个节点可以有0、1或多个子节点，没有子节点的节点被称为叶子节点。`A` 是 `B` 的父节点，`B` 是 `A` 的子节点。`C` 和 `D` 属于同一个父节点 `B`，他们直接互相称之为兄弟，即互相为兄弟节点。`C`、`D`、`F`、`H` 和 `I` 都是叶子节点。



## 二叉树(Binary Tree)

二叉树是一种特殊的树，它的子节点个数不超过两个。上面的图就是一个二叉树。

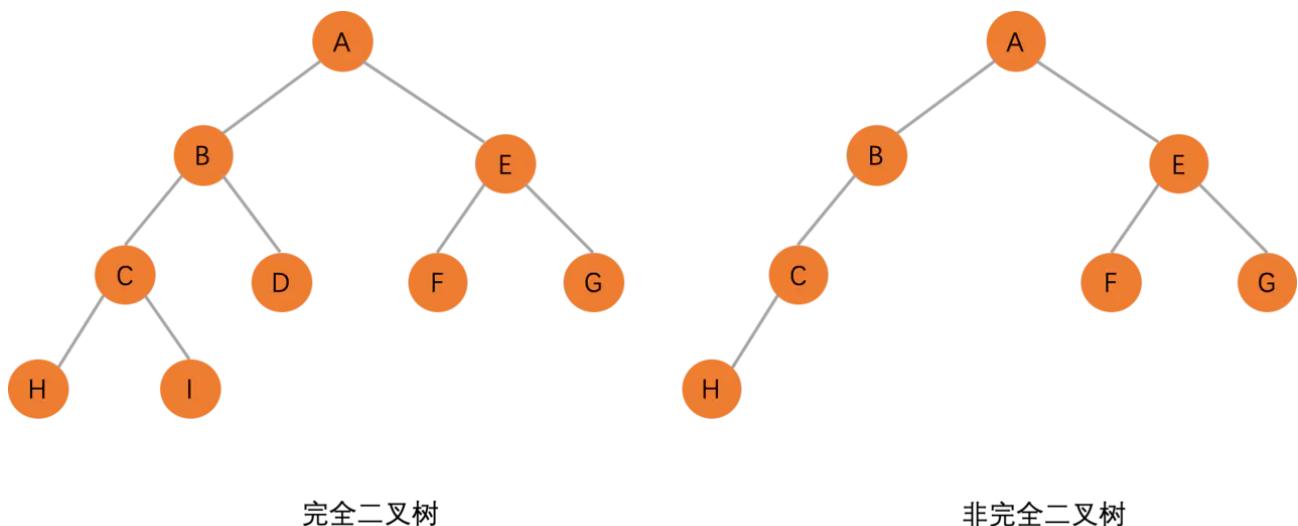
一个二叉树，如果每一个层的结点数都达到最大值，则这个二叉树就是满二叉树。也就是说，如果一个二叉树的层数为  $k$ ，且结点总数是  $2^k - 1$ ，则它就是**满二叉树**。[来自百度百科]。



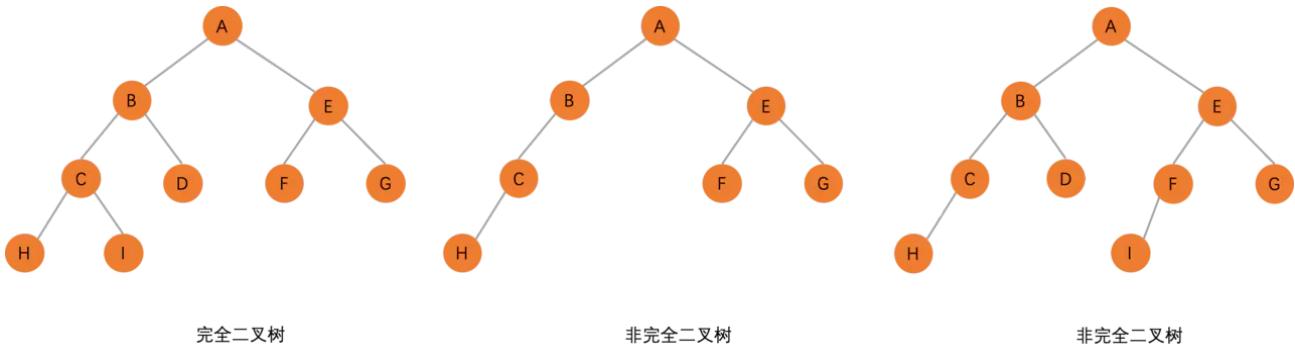
### 满二叉树(Full Binary Tree)

我们发现，满二叉树的所有叶子节点都在最底层，而且除了叶子节点，其他的节点都有左右两个子节点。

看完满二叉树，来认识一下由其引出来的完全二叉树，可能光看字面意思不是很好理解。若设二叉树的深度为  $h$ ，除第  $h$  层外，其它各层 ( $1 \sim h-1$ ) 的结点数都达到最大个数，第  $h$  层所有的结点都连续集中在最左边，这就是完全二叉树。



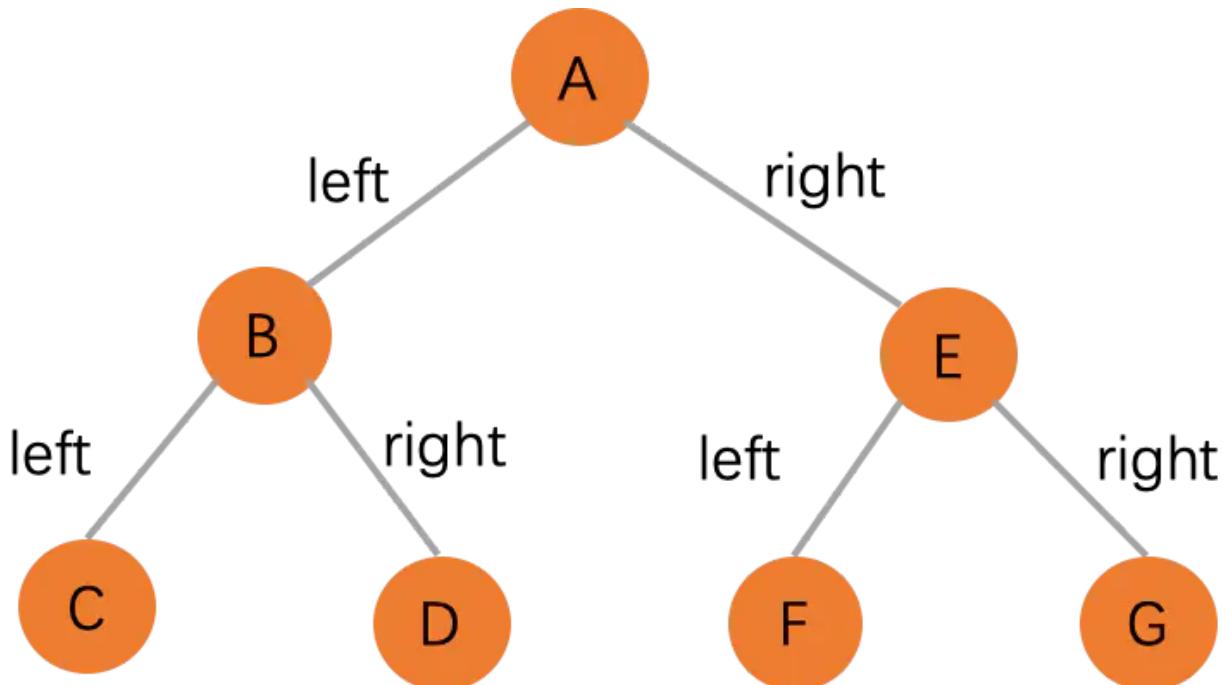
对于大多数同学来说，满二叉树很好理解，但是到了完全二叉树可能就迷糊了，下面的图给出了几个完全二叉树和非完全二叉树的例子，相信你看了应该就明白是怎么回事了。



## 二叉树的存储结构

存储一棵二叉树，有两种方法，一种是基于指针或者引用的**二叉链式存储**，一种是基于数组的**顺序存储**。

先来看大家都比较熟悉、更加直观的链式存储结构。链式存储结构中，每一个结点包含三个关键属性：指向左子节点的指针，数据，指向右子节点的指针。



- 结构定义

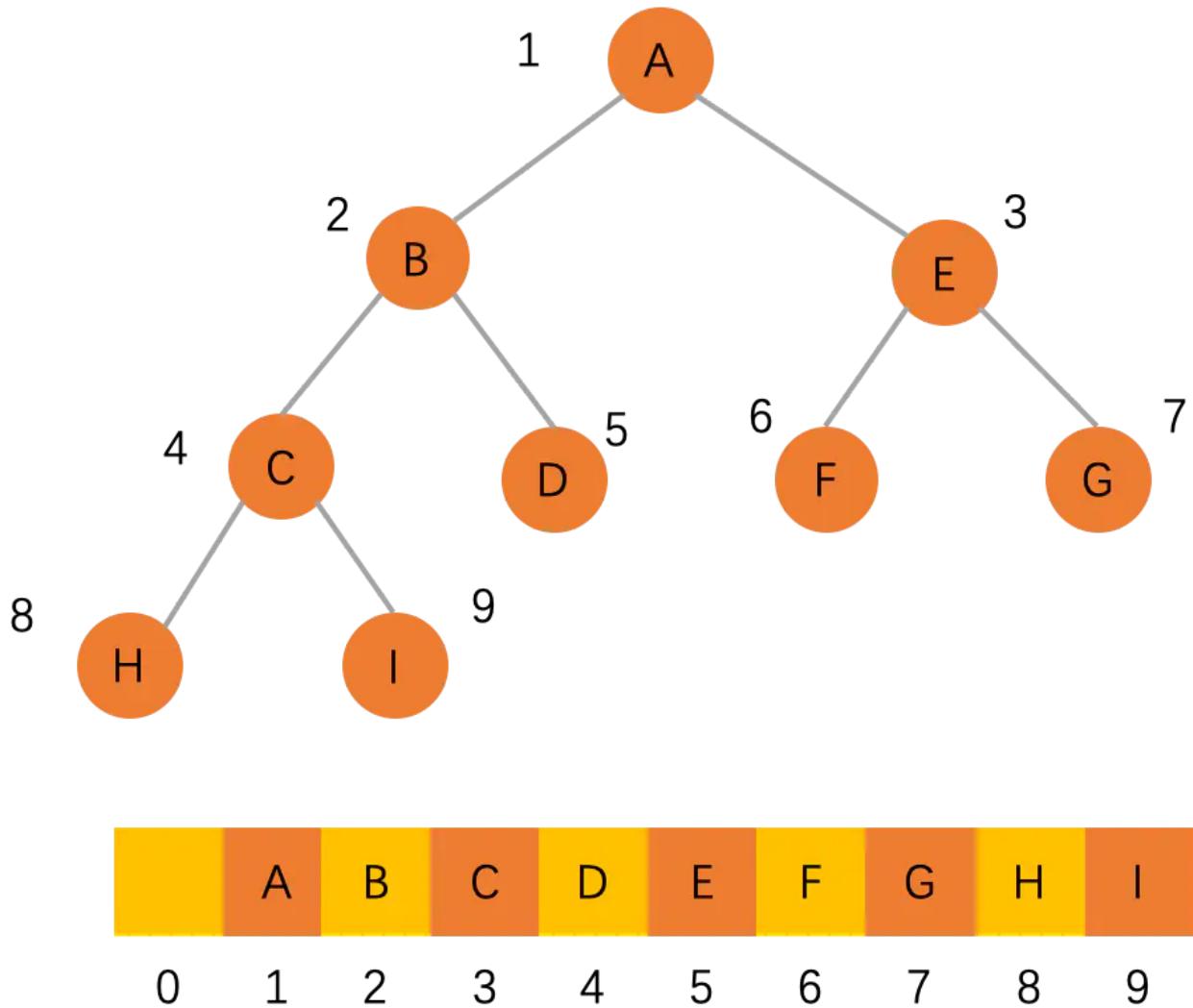
```

1 class Node {
2 public data: any;
3 public left?: Node;
4 public right?: Node;
5 constructor({ data, left, right }) {
6 this.data = data;
7 this.left = left;
8 this.right = right;
9 }
10 }

```

```
8 this.right = right;
9 }
10 }
```

再来看基于数组的顺序存储，为了帮助大家更好地理解，这里拿了一张完全二叉树的图。使用顺序存储，完全二叉树是非常合适的，可以自上而下，从左到右来顺序存储  $n$  个结点的完全二叉树。



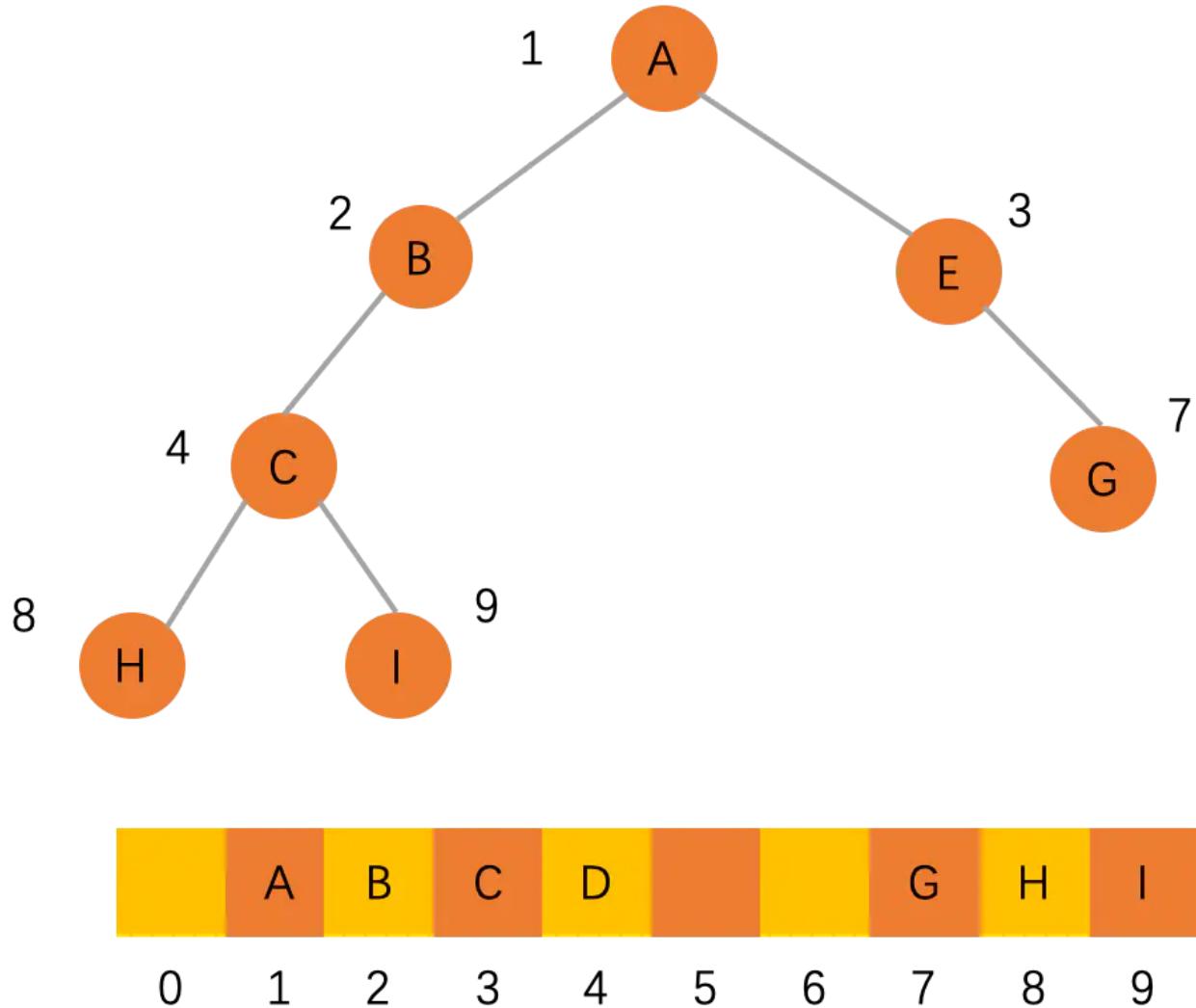
我们把根节点放到  $i = 1$ ，那么根节点的左子节点存储在下标  $2 * i = 2$  的位置，右子节点存储在  $2 * i + 1 = 3$  的位置。依此类推，B 节点的左子节点存储在  $2 * i = 2 * 2 = 4$  的位置，右子节点存储在  $2 * i + 1 = 2 * 2 + 1 = 5$  的位置。

完全二叉树的这种存储结构，有以下特点：

- 非根节点的父节点对应数组下标为是  $i/2$
- 节点的左子节点的序号是  $2 * i$ ,如果  $2 * i > n$  ,则左子节点不存在

- 节点的右子节点的序号是  $2 * i + 1$  , 如果  $2 * i > n + 1$  , 则右子节点不存在

如果不是完全二叉树，用这种结构来存储会浪费比较多的空间，可以看下面的图。



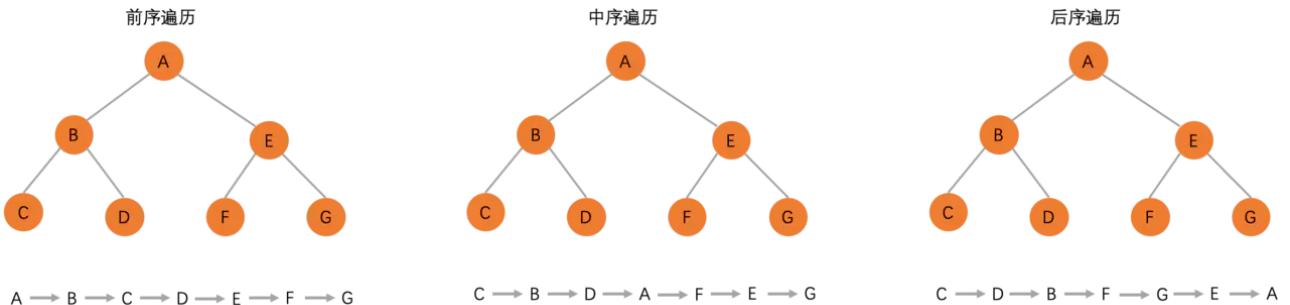
因此，如果一棵树刚好是完全二叉树，采用顺序存储是最节省内存空间的，不需要额外再提供左右两个指针。

## 二叉树的遍历

二叉树的常见的遍历方法有 3 种：前序遍历、中序遍历、和后序遍历，依据节点和它的左右子树的遍历顺序不同来划分。

- 前序遍历：对于二叉树中的任意节点，先访问该节点，再去访问当前节点的左子树，然后是右子树，若当前节点无左子树，则访问当前节点的右子树；
- 中序遍历：对于二叉树中的任意节点，先访问该节点的左子树，再去访问当前节点，然后是右子树；

- 后序遍历：对于二叉树中的任意节点，先访问该节点的左子树，再去访问当前节点的右子树，然后是当前节点；



二叉树是一种递归形式的数据结构，因此对于二叉树的 3 种遍历，我们就可以借助其自身的特性，通过递归实现。

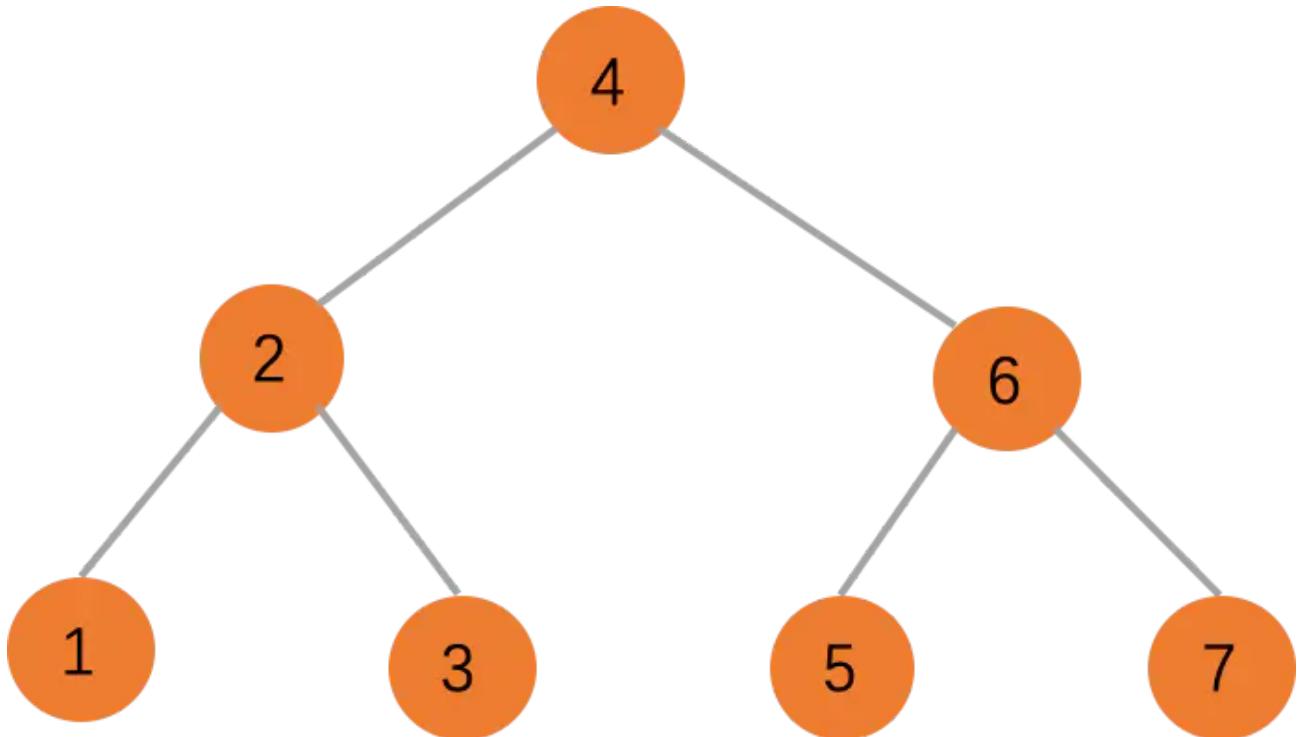
```

1 // 前序遍历
2 function preTraversal(node) {
3 if (node === null) {
4 return;
5 }
6 console.log(node.data);
7 preTraversal(node.left);
8 preTraversal(node.right);
9 }
10
11 // 中序遍历
12 function inTraversal(node) {
13 if (node === null) {
14 return;
15 }
16 inTraversal(node.left);
17 console.log(node.data);
18 inTraversal(node.right);
19 }
20
21 // 后序遍历
22 function postTraversal(node) {
23 if (node === null) {
24 return;
25 }
26 postTraversal(node.left);
27 postTraversal(node.right);
28 console.log(node.data);
29 }
```

可以看到，使用递归实现二叉树的遍历十分简单。

## 二叉查找树(Binary Search Tree)

二叉查找树(BST)是一种特殊的二叉树，较小的值保存在左子树中，较大的值保存在右子树中，这一特性使得查找的效率很高。因此它又有另外一个名字，叫二叉排序树(Binary Sort Tree)。看了图是不是瞬间明白二叉查找树是个啥。



## 二叉查找树的查找

对于 BST 通常有以下 3 种类型的查找：

- 找给定值
- 找最小值
- 找最大值

根据二叉查找树的属性，查找最小值和最大值比较简单。因为较小的值总是在左子节点上，要找到最小值，只需要遍历左子树，直到找到最后一个节点。

```
1 function getMin() {
2 let currentNode = this.tree;
3 while (currentNode.left !== null) {
4 currentNode = currentNode.left;
5 }
6 return currentNode.data;
7 }
```

查找最大值也是同理，只需遍历右子树，直到找到最后一个节点即可。

```
1 function getMax() {
2 let currentNode = this.tree;
3 while (currentNode.right !== null) {
4 currentNode = currentNode.right;
5 }
6 return currentNode.data;
7 }
```

在 BST 上查找给定值，通过比较该值和当前节点上的值的大小，就能够确定向左还是向右遍历。

```
1 function find(data) {
2 let node = this.tree;
3 while (node !== null) {
4 if (node.data === data) {
5 return node;
6 } else if (data < node.data) {
7 node = node.left;
8 } else {
9 node = node.right;
10 }
11 }
12 }
```

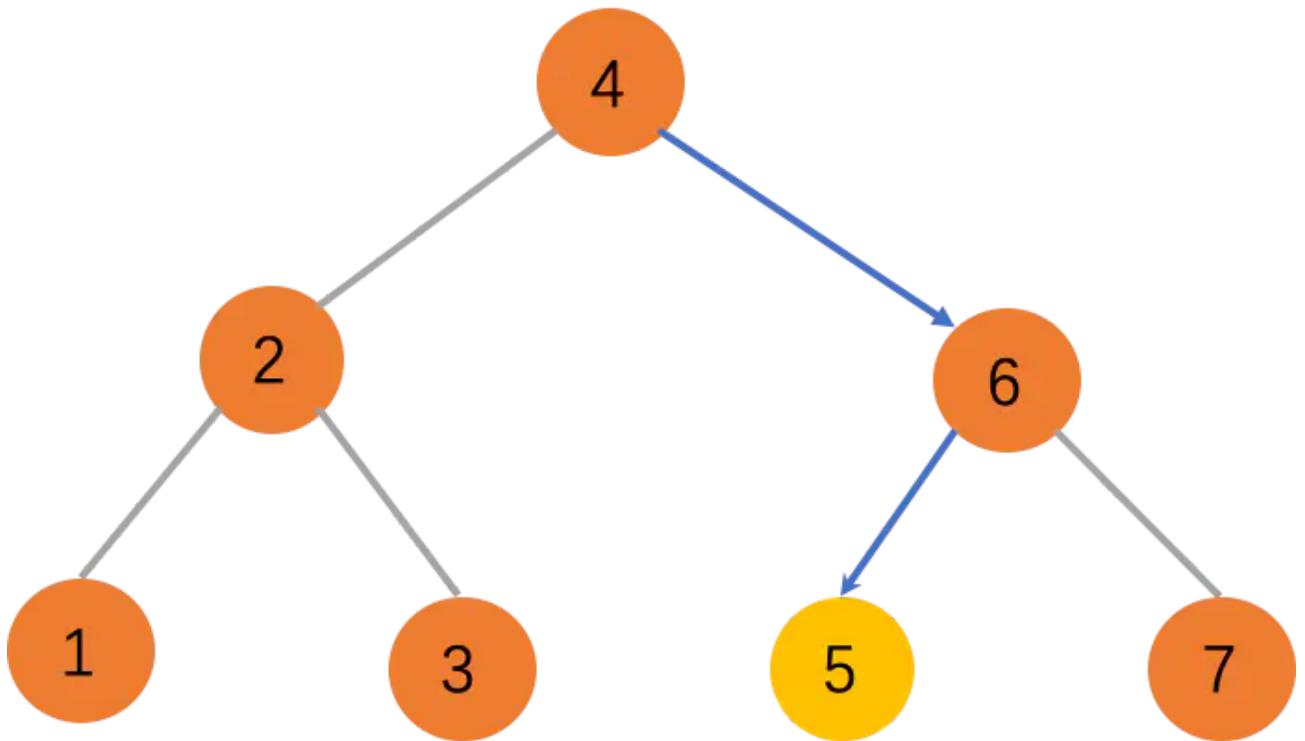
## 二叉查找树的插入

在二叉查找树中插入元素比较复杂，分为以下几种情况：

- 首先需要检查 BST 是否有根节点，如果没有，插入的节点就是根节点，就完事了
- 如果待插入的节点不是根节点，那么就需要遍历整棵树，找到合适的位置

如果要插入的数据大于当前节点，并且当前节点的右子树为空，就将新数据插入到右子节点的位置；如果不为空，就再递归遍历右子树，查找插入位置。同理，如果要插入的数据小于当前节点，并且节点的左子树为空，就将新数据插入到左子节点的位置；如果不为空，就再递归遍历左子树，查找插入位置。

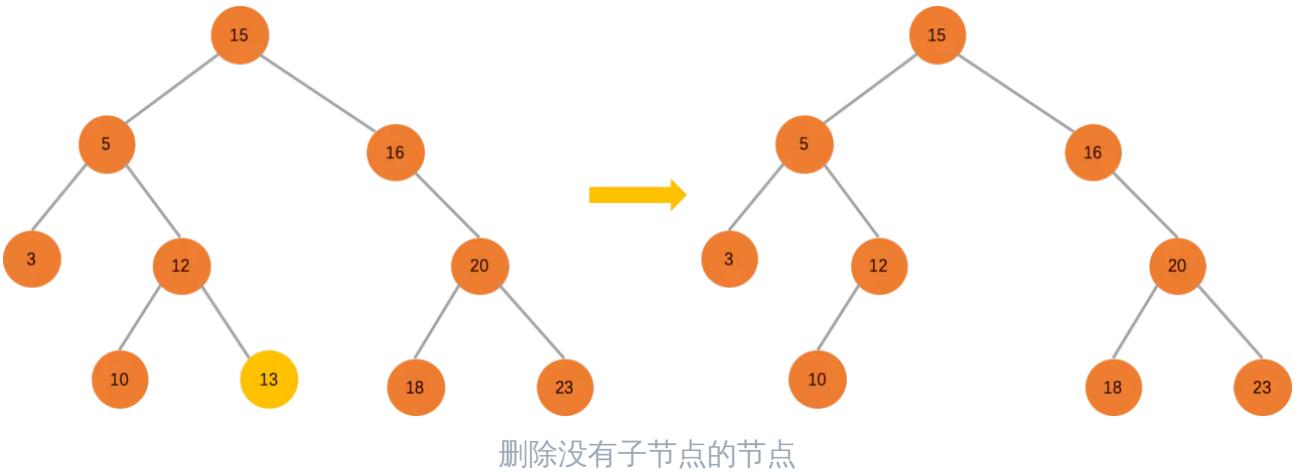
## 插入5



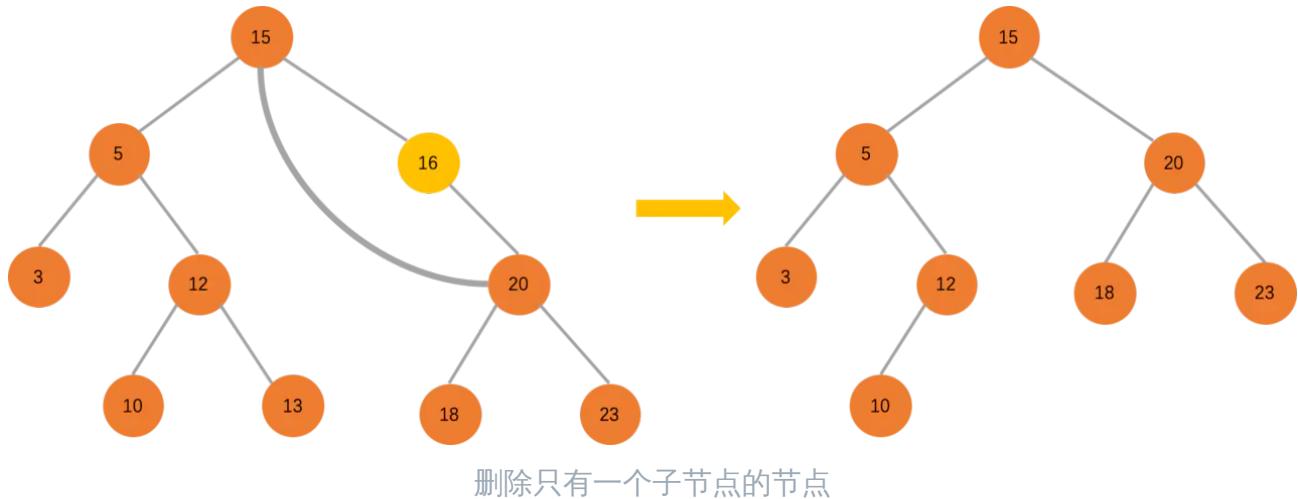
```
1 function insert(data) {
2 const n = new Node(data);
3 if (this.tree === null) {
4 this.tree = n;
5 return;
6 }
7
8 let node = this.tree;
9 while (node !== null) {
10 if (data > node.data) {
11 if (node.right === null) {
12 node.right = n;
13 return;
14 }
15 node = node.right;
16 } else {
17 if (data < node.data) {
18 if (node.left === null) {
19 node.left = n;
20 return;
21 }
22 node = node.left;
23 }
24 }
25 }
26}
27}
```

## 二叉查找树的删除

在二叉查找树上删除节点的操作最复杂，牵一发而动全身，节点直接存在相互关联。如果删除的是没有子节点的节点，那就直接删掉就完事了。稍微麻烦的是删除节点只有一个子节点的情况，如果删除的节点包含两个子节点，那就是最麻烦的了。

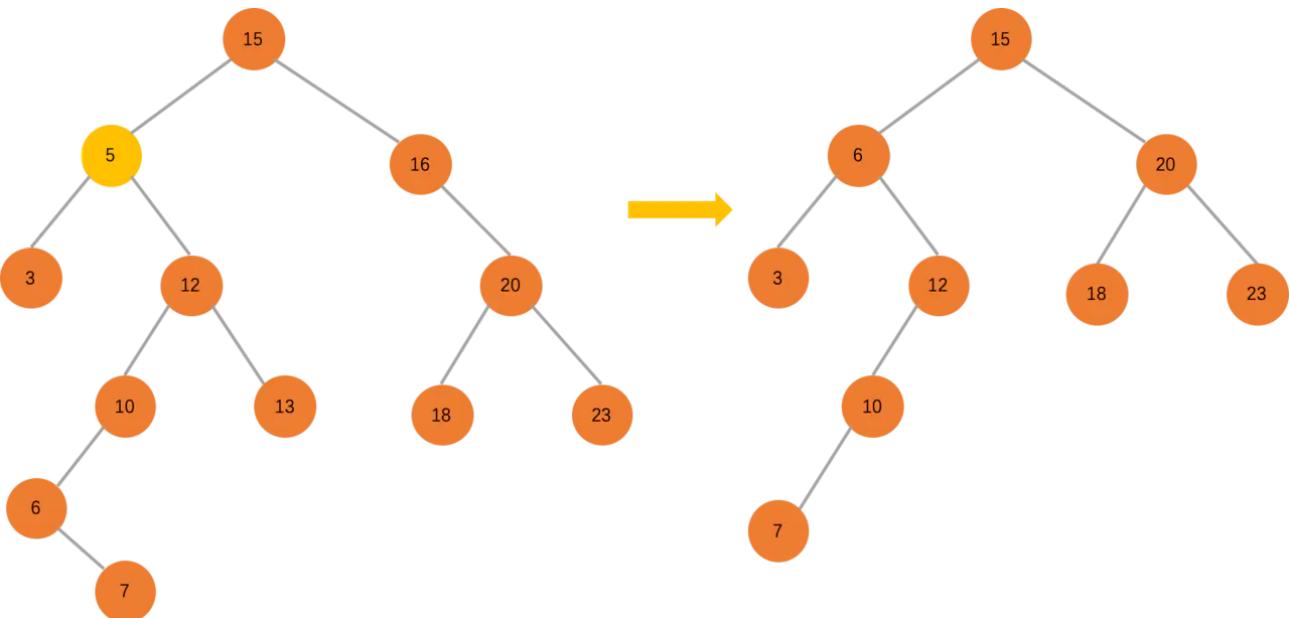


如果没有子节点，直接把要删除的节点干掉就行



只有一个子节点只需要把更新父节点指向要删除节点的指针，指向要删除节点的子节点就可以了。比如图中删除节点的节点是 16，把 15 指向 13 的指针指向 20 即可。

现在来看最复杂的情况：



删除有两个子节点的节点

有两个子节点，我们需要查找右子树上的最小值，把它拿到待删除的节点上，然后再删除这个最小的节点。删除这个最小的节点，又回到了上面讲的两条规则，这个最小节点不可能有两个子节点，如果有左子节点，就不是最小节点了。

```

1 function remove(data) {
2 let node = this.tree;
3 let parentNode;
4 while (node !== null && node.data != data) {
5 parentNode = node;
6 if (data > node.data) {
7 node = node.right;
8 } else {
9 node = node.left;
10 }
11 }
12
13 if (node === null) {
14 return; // 没找着
15 }
16
17 // 我们采用非递归的方式，先处理要删除的节点有两个子节点的情况
18 if (node.left !== null && node.right !== null) {
19 // 查找右子树中最小节点
20 let minNodeParent = node;
21 let minNode = minNodeParent.right;
22 while (minNodeParent.left !== null) {
23 minNodeParent = minNode;
24 minNode = minNode.left;
25 }
26 node.data = minNode.data;
27 node = minNode; // 下面就变成删除 minNode 了

```

```

28 parentNode = minNodeParent;
29 }
30
31 // 删除节点是叶子节点或者只有一个子节点
32 let childNode = null;
33 if (node.left !== null) {
34 childNode = node.left;
35 } else if (node.right !== null) {
36 childNode = node.right;
37 }
38
39 if (parentNode === null) { // 父节点是 null，说明删除的是根节点
40 this.tree = childNode;
41 // 第二种情况，把父节点的指向删除节点的指针指向删除节点的子节点
42 // 删除的是 node，把 parentNode 指向 node 的指针，指向 childNode
43 } else if (parentNode.left === node) {
44 parentNode.left = childNode;
45 } else {
46 parentNode.right = childNode;
47 }
48 }

```

本章节分为 4 个部分：

- Part 1
  - 最小栈
  - Shuffle an Array
  - 将有序数组转换为二叉搜索树
- Part 2
  - 对称二叉树
  - 二叉树的最大深度
  - 验证二叉搜索树
- Part 3
  - 二叉树的层次遍历
  - 二叉树的序列化与反序列化
- Part 4
  - 中序遍历二叉树
  - 从前序与中序遍历序列构造二叉树
  - 二叉搜索树中第 K 小的元素
- Part 5
  - 填充每个节点的下一个右侧节点指针
  - 岛屿数量
  - 二叉树的锯齿形层次遍历

阅读完本章节，你将有以下收获：

- 熟悉栈的数据结构，可以解决基本栈相关问题
- 掌握二叉树的概念
- 会用不同的方法去遍历二叉树，解决相关问题

# 最小栈、Shuffle an Array和将有序数组转换为二叉搜索树

## 最小栈

设计一个支持 push , pop , top 操作，并能在常数时间内检索到最小元素的栈。

- push(x) – 将元素 x 推入栈中。
- pop() – 删除栈顶的元素。
- top() – 获取栈顶元素。
- getMin() – 检索栈中的最小元素。

## 示例

```
1 MinStack minStack = new MinStack();
2 minStack.push(-2);
3 minStack.push(0);
4 minStack.push(-3);
5 minStack.getMin(); --> 返回 -3.
6 minStack.pop();
7 minStack.top(); --> 返回 0.
8 minStack.getMin(); --> 返回 -2.
```

## 方法一 最小元素栈

### 思路

根据数组的性质，push、pop、top都能在常数时间内完成，而此题关键是常数时间内检索最小元素，此解法是开辟一个数组存储，push数据同时，存储当前栈中最小元素，pop数据的同时pop最小元素栈栈顶数据；

### 详解

1. 创建最小元素栈时，开辟 stack 及 minStack 数组，stack 用于存储压栈元素，minStack 用于存储最小元素序列；
2. push操作执行元素 x 压栈：
3. 如果 minStack 数组为空时，添加元素 x 到 minStack 数组。

4. 否则比较元素 x 与数组末尾元素，取最小值添加到 minStack 数组末尾。表示当前栈中元素的最小值为x。
5. pop操作:
6. 删除 stack 数组末尾元素的同时，删除数组 minStack 末尾元素。
7. getMin操作:
8. 直接调用 pop 方法，返回 minStack 数据中末尾元素即可。

```
1 const MinStack = function () {
2 this.stack = [];
3 this.minStack = [];
4 };
5
6 /**
7 * @param {number} x
8 * @return {void}
9 */
10 MinStack.prototype.push = function (x) {
11 this.stack.push(x);
12 if (this.minStack.length === 0) {
13 this.minStack.push(x);
14 } else {
15 const min = Math.min(this.minStack[this.minStack.length - 1], x);
16 this.minStack.push(min);
17 }
18 };
19
20 /**
21 * @return {void}
22 */
23 MinStack.prototype.pop = function () {
24 this.minStack.pop();
25 return this.stack.pop();
26 };
27
28 /**
29 * @return {number}
30 */
31 MinStack.prototype.top = function () {
32 return this.stack[this.stack.length - 1];
33 };
34
35 /**
36 * @return {number}
37 */
38 MinStack.prototype.getMin = function () {
39 return this.minStack[this.minStack.length - 1];
40 };
```

## 复杂度分析

- 时间复杂度： $O(1)$

`push`、`pop`、`top`、`getMin` 操作都是基于数组索引，耗费  $O(1)$  的时间。

- 空间复杂度： $O(n)$

每次 `push` 一个元素到 `stack` 栈的同时将 `stack` 栈中最小元素 `push` 到 `minStack` 栈，耗费  $O(n)$  的栈空间

## 方法二 差值存储

### 思路

用一个 `min` 变量保存最小值，每次 `push` 操作压栈时，保存的是入栈的值和最小值 `min` 的差值，而不是入栈的值；`pop` 出栈时，通过 `min` 值和栈顶的值得到；不过此算法有一个缺陷，两数差值有溢出风险。

### 详解

1. 创建最小元素栈时，开辟 `stack` 数组，用于存储入栈元素 `x` 与最小元素 `min` 的差值；同时定义变量 `min`，用于存储最小值，初始值为 `Number.MAX_VALUE`。
2. `push` 操作执行元素 `x` 入栈：：
3. 取元素 `x` 与最小元素 `min` 的差值，压入 `stack` 数组：
4. 比较元素 `x` 与 `min` 值，取其最小值赋值给 `min` 变量。
5. `pop` 操作：
6. 弹出 `stack` 数组末尾元素值 `value`，如果值 `value > 0`，说明 `push` 操作的值大于 `min`，返回 `value + min` 值；如果值 `value <= 0`，说明 `push` 操作的值小于等于原 `min` 值，恢复最小值 `min`，同时返回 `min` 值即可。
7. `top` 操作：
8. 取 `stack` 数组末尾元素值 `value`，如果值 `value > 0`，说明 `push` 操作的值大于 `min`，返回 `value + min` 值；如果值 `value <= 0`，说明 `push` 操作的值小于等于原 `min` 值，返回 `min` 值即可。
9. `getMin` 操作：
10. 返回 `min` 元素即可。

```
1 const MinStack = function () {
2 this.stack = [];
3 this.min = Number.MAX_VALUE;
4 };
5
6 /**
7 * @param {number} x
8 * @return {void}
*/
```

```

9 */
10 MinStack.prototype.push = function (x) {
11 const min = this.min;
12 this.stack.push(x - min);
13 if (x < min) {
14 this.min = x;
15 }
16 };
17
18 /**
19 * @return {void}
20 */
21 MinStack.prototype.pop = function () {
22 const value = this.stack.pop();
23 const min = this.min;
24 if (value > 0) {
25 return value + min;
26 } else {
27 this.min = min - value;
28 return min;
29 }
30 };
31
32 /**
33 * @return {number}
34 */
35 MinStack.prototype.top = function () {
36 const value = this.stack[this.stack.length - 1];
37 if (value > 0) {
38 return value + this.min;
39 } else {
40 return this.min;
41 }
42 };
43
44 /**
45 * @return {number}
46 */
47 MinStack.prototype.getMin = function () {
48 return this.min;
49 };

```

## 复杂度分析

- 时间复杂度： $O(1)$
- 空间复杂度： $O(1)$

每次 push 一个元素到 stack 栈的同时只需要一个元素空间存储最小值，耗费  $O(1)$  的栈空间

## Shuffle an Array

打乱一个没有重复元素的数组。

## 示例

```
1 // 以数字集合 1, 2 和 3 初始化数组。
2
3 int[] nums = {1,2,3};
4
5 Solution solution = new Solution(nums);
6
7 // 打乱数组 [1,2,3] 并返回结果。任何 [1,2,3] 的排列返回的概率应该相同。
8
9 solution.shuffle();
10
11 // 重设数组到它的初始状态[1,2,3]。
12
13 solution.reset();
14
15 // 随机返回数组[1,2,3]打乱后的结果。
16
17 solution.shuffle();
```

## 方法一

### 思路

- `reset` 函数：缓存传入的原始数据，用于在函数调用时返回。但值得一提的是，进行缓存原始数据时，必须进行浅拷贝，因为原始数据为数组，普通的赋值会导致引用对象传递，一旦变更了 `this.nums` 的数组内容，缓存的数组也将同步变更
- `shuffle` 函数：我们模拟一个这样的场景，有  $n$  个标着数的球，我们把这  $n$  个球放入一个看不见的袋子中，每次从中摸一个球出来，并按照摸出的顺序，直到摸空袋子。具体的操作，我们把原始数组复制一份为 `nums`，每次根据 `nums` 的长度随机生成一个下标从 `nums` 中取一个数出来，将其放入新数组 `ary` 中，并删除 `nums` 中对应下标的数

### 详解

1. 定义 `this.nums` 存储传入的数据
2. 定义 `this.original` 存储 `nums` 的克隆数组
3. 定义重置 `reset` 方法，将 `this.nums` 重制为 `this.original` 的克隆数组，并将 `this.original` 重新克隆一遍（因数组为引用对象，不重新缓存新数组会导致 `this.original` 和 `this.nums` 同步变化）

4. 定义打乱 `shuffle` 方法，根据 `this.nums` 的长度进行循环，每次从根据 `this.nums` 长度通过 `Math.random()` 随机生成一个下标
5. 根据随机生成的下标，将值存入 `ary` 数组中
6. 最后返回 `ary` 数组

## 代码

```

1 /**
2 * @param {number[]} nums
3 */
4 const Solution = (nums) => {
5 this.nums = nums;
6 this.original = nums.slice(0);
7 };
8 /**
9 * 重置数组并返回
10 * @return {number[]}
11 */
12 Solution.prototype.reset = () => {
13 this.nums = this.original;
14 this.original = this.original.slice(0);
15 return this.original;
16 };
17 /**
18 * 返回一个随机重排的数组
19 * @return {number[]}
20 */
21 Solution.prototype.shuffle = () => {
22 const ary = [];
23 const nums = this.nums.slice(0);
24 const len = nums.length;
25 for (let i = 0; i < len; i += 1) {
26 const targetIndex = Math.floor(Math.random() * nums.length);
27 ary[i] = nums[targetIndex];
28 nums.splice(targetIndex, 1);
29 }
30 return ary;
31 };

```

## 复杂度分析

- 时间复杂度： $O(n^2)$
- js 中 `splice` 方法的时间复杂度为  $O(n)$ ，因为当你在中间删除一个元素后，在此位置之后的所有元素都需要整体向前移动一个位置，因此导致遍历所有  $n$  个元素，又因为 `splice` 方法于 `for` 循环中需执行  $n$  遍，因此为  $O(n^2)$ 。

- 空间复杂度： $O(n)$

为实现重置功能，原始数组需保存一份，因此为  $O(n)$ 。

## 方法二

### 思路

- `reset` 函数：同方法一
- `shuffle` 函数：为降低方法一中的时间复杂度，我们可以让数组中的元素进行互换，从而减少 `splice` 方法所需执行的时间。具体的操作，我们从数组的最后往前迭代，生成一个范围在 0 到当前遍历下标之间的随机整数，和当前遍历下标的元素进行互换。这跟方法一中的模拟摸球是一样的，每次被摸出的球便不能再被摸出来。

### 详解

1. 定义 `this.nums` 存储传入的数据
2. 定义 `this.original` 存储 `nums` 的克隆数组
3. 定义重置 `reset` 方法，将 `this.nums` 重制为 `this.original` 的克隆数组，并将 `this.original` 重新克隆一遍（因数组为引用对象，不重新缓存新数组会导致 `this.original` 和 `this.nums` 同步变化）
4. 定义打乱 `shuffle` 方法，根据 `this.nums` 的长度进行倒序循环，每次从根据当前下标 `i` 通过 `Math.floor(Math.random() * (i + 1))` 随机生成一个下标（Knuth-Durstenfeld Shuffle 算法）
5. 根据随机生成的下标，和当前下标，进行数据互换
6. 最后返回 `nums` 数组

### 代码

```
1 /**
2 * @param {number[]} nums
3 */
4 const Solution = (nums) => {
5 this.nums = nums;
6 this.original = nums.slice(0);
7 };
8 /**
9 * 重置数组并返回
10 * @return {number[]}
11 */
12 Solution.prototype.reset = () => {
13 this.nums = this.original;
14 this.original = this.original.slice(0);
```

```

15 return this.original;
16 };
17 /**
18 * 返回一个随机重排的数组
19 * @return {number[]}
20 */
21 Solution.prototype.shuffle = () => {
22 const nums = this.nums.slice(0);
23 const len = nums.length;
24 for (let i = len - 1; i > 0; i -= 1) {
25 const targetIndex = Math.floor(Math.random() * (i + 1));
26 const tmp = nums[i];
27 nums[i] = nums[targetIndex];
28 nums[targetIndex] = tmp;
29 }
30 return nums;
31 };

```

## 复杂度分析

- 时间复杂度： $O(n)$

上述算法中，for 循环内操作都是常数时间复杂度，因此为  $O(n)$ 。

- 空间复杂度： $O(n)$

为实现重置功能，原始数组需保存一份，因此为  $O(n)$ 。

## 将有序数组转换为二叉搜索树

将一个按照升序排列的有序数组，转换为一棵高度平衡二叉搜索树。本题中，一个高度平衡二叉树是指一个二叉树每个节点的左右两个子树的高度差的绝对值不超过 1。

### 示例

```

1 给定有序数组：[-10,-3,0,5,9]，
2 一个可能的答案是：[0,-3,9,-10,null,5]，它可以表示下面这个高度平衡二叉搜索树：
3 0
4 / \
5 -3 9
6 / \ / \
7 -10 null 5 null

```

### 方法一 二分 + 递归法

## 思路

由于数组是按照递增有序排列的，并且高度平衡二叉搜索树（一个高度平衡二叉树是指一个二叉树每个节点的左右两个子树的高度差的绝对值不超过1），可以使用二分法从数组的中间开始查找数据。

## 详解

1. 找出数组的中间坐标 (mid) 对应元素，作为当前二叉树节点 (root) 的 value
2. root 的左节点 是  $0 \rightarrow mid$  的中间坐标对应元素
3. root 的右节点 是  $mid \rightarrow (arr.length - 1)$  的中间坐标对应元素
4. root 的左右节点 按照 1 - 3 步骤生成新的左右节点，直到数组遍历完，算法终止

## 代码

```
1 const sortedArrayToBST = (arr) => {
2 return initTreeNodes(arr, arr.length - 1, 0);
3 };
4
5 const initTreeNodes = (arr, end, start) => {
6 if (start <= end) {
7 const mid = start + parseInt((end - start) / 2, 10);
8 const root = Node(arr[mid]);
9 root.left = initTreeNodes(arr, mid - 1, start);
10 root.right = initTreeNodes(arr, end, mid + 1);
11 return root;
12 } else {
13 return null;
14 }
15};
```

## 复杂度分析

- 时间复杂度： $O(n)$   
每次调用需要遍历数组，因此，时间复杂度是  $O(n)$ 。
- 空间复杂度： $O(1)$   
算法在运行过程中，只申请了常量大小内存，因此，空间复杂度是  $O(1)$ 。

## 方法二 数组模拟队列

## 思路

可以先将提示数组按照二分法的查找顺序，一次推入数组中。然后，按照数组顺序通过生成二叉树的一般算法生成目标树。由于在题目的二叉树中，节点的左子节点的值要小于节点的值，节点的右子节点的值要大于节点的值。因此，数组从中点按照 [root, 左, 右, 左, 右...] 接收节点的值，并按照生成二叉树的一般算法，即可生成目标树。

## 详解

1. 首先，使用二分法，先找到数组中间位置 (mid) 元素，将该元素 push 进目标数组 queue (模拟队列)
2. 将数组分成两部分 0 -> mid, (mid + 1) -> arr.length
3. 将获得两个新数组，按照 1、2 步骤重复，直到数组元素全部遍历完成
4. 然后，按顺序遍历数组，arr[0] 为根节点值
5. 当 queue[i]，小于 arr[0] 时会被分配到左子节点，大于 arr[0] 时会被分配到右子节点
6. 当左子节点已经有值时，会比较左子节点的值与 queue[i]，按照值得大小 分配到 左子节点的左子节点或者右子节点
7. 当右子节点已经有值时，会比较右子节点的值与 queue[i]，按照值得大小 分配到 左子节点的左子节点或者右子节点
8. 遍历 queue，重复执行 6、7 步骤，直到 queue 被全部遍历

## 代码

```
1 const sortedArrayToBST = (arr) => {
2 if (!arr.length) {
3 return null;
4 }
5 const queue = [];
6 // 二分法重新排列数组 queue
7 initNodeValueArr(arr, 0, arr.length, queue);
8 // 根节点
9 const root = new TreeNode(queue[0]);
10 for (let i = 1; i < queue.length; i += 1) {
11 // 插入节点数据
12 insertNode(root, queue[i]);
13 }
14 return root;
15 };
16
17 const insertNode = (node, value) => {
18 // 生成左子节点
19 if (value < node.val) {
20 if (!node.left) {
21 node.left = new TreeNode(value);
22 } else {
23 insertNode(node.left, value);
24 }
25 } // 生成右子节点
```

```
26 } else if (!node.right) {
27 node.right = new TreeNode(value);
28 } else {
29 insertNode(node.right, value);
30 }
31 };
32
33 const initNodeValueArr = (arr, start, end, queue) => {
34 const mid = start + parseInt((end - start) / 2, 10);
35 queue.push(arr[mid]);
36 // 左节点数值
37 if (start < mid) {
38 initNodeValueArr(arr, start, mid, queue);
39 }
40 // 右节点数值
41 if (mid + 1 < end) {
42 initNodeValueArr(arr, mid + 1, end, queue);
43 }
44};
```

## 复杂度分析

- 时间复杂度： $O(n)$

每次调用需要遍历数组，因此，时间复杂度是  $O(n)$ 。

- 空间复杂度： $O(n)$

算法在运行过程中，申请了  $n$  长度的数组 `queue`，因此，空间复杂度是  $O(n)$

# 对称二叉树、二叉树的最大深度和验证二叉搜索树

## 对称二叉树

给定一个二叉树，要判定是否是镜像对称的

### 示例

```
1 例如，二叉树 [1,2,2,3,4,4,3] 是对称的。
2
3 1
4 / \
5 2 2
6 / \ / \
7 3 4 4 3
8
9 但是下面这个 [1,2,2,null,3,null,3] 则不是镜像对称的：
10
11 1
12 / \
13 2 2
14 / \
15 3 3
```

### 方法一 递归判断

#### 思路

判断二叉树是不是对称的，主要是看二叉树左边和右边的节点是不是各自相等。所以我们可以通过递归，去判断左树的左节点和右树的右节点是不是相同。如果两个节点都为空，则表示递归到树的底部了；如果有一边空了另外一半没空，说明有一边的节点没了，另外一半还在，肯定不是对称的树；如果两边对称，继续递归节点的左右节点，直到递归完全或者发现不对称。

#### 详解

1. 我们判断当前的树结构还是否为空，为空则该树是对称的，不为空则递归判断左子树的左子树和右子树的右子树是否相等
2. 如果左节点或右节点为空时，则判断对应的右节点或左节点是否为空，为空则返回 `true`，不为空则返回 `false`

3. 如果左右节点都不为空时，则判断左节点的左节点和右节点的右节点是否相等
4. 如果相等，则继续传入该节点的子节点去判断；不相等则直接返回 `false`

## 代码

```
1 /**
2 * 如果输入一个空对象，则直接返回true
3 * @param {TreeNode} root
4 * @return {boolean}
5 */
6 const isSymmetric = function (root) {
7 if (!root) {
8 // 若根节点为空，则返回true
9 return true;
10 }
11 // 递归函数
12 return isSameTree(root.left, root.right);
13 };
14
15 /**
16 * 判断二叉树是否对称，直接去判断根节点的左子树和右子树是否相同
17 * @param {TreeNode} root
18 */
19 const isSameTree = (r, l) => {
20 // 若左节点或右节点为空，则判断对应的右节点或左节点是否为空
21 // 为空时，则返回true，不为空则返回false
22 if (r === null) return l === null;
23 if (l === null) return r === null;
24
25 // 判断左节点的左节点和右节点的右节点是否相等
26 // 不相等时，则该树不对称，相等时则继续递归判断
27 if (r.val !== l.val) return false;
28
29 return isSameTree(r.left, l.right) && isSameTree(r.right, l.left);
30};
```

## 复杂度分析

- 时间复杂度： $O(n)$   
我们需要对这个树中的每个节点都要进行遍历
- 空间复杂度： $O(n)$   
当树是线性时，由栈上的递归调用造成的空间复杂度为  $O(n)$

## 方法二 迭代判断

## 思路

迭代的思路就是不断的把待处理的节点入队，每次都将本级的节点进行判断是否对称，如果不对称则返回false，对称则将下一级的节点全部入栈，然后再依次出队判断处理，再获取新的待处理节点入队，直到结束。

## 详解

1. 最开始将根节点入列，然后新建一个出队的队列 level，我们对 level 进行判断处理。
2. 我们一次取出 level 队列中下标为 i 和下标为 (level.length - i - 1) 的两个元素进行比较；
3. 若不相等，则返回false；若相等，则将下下一级的节点全部入列，然后在将下一级的节点全部出列进行判断；
4. 重复第 3、4 步，当队列为空时，则方法停止。

## 代码

```
1 /**
2 * 如果输入一个空对象，则直接返回true
3 * @param {TreeNode} root
4 * @return {boolean}
5 */
6 const isSymmetric = function (root) {
7 if (!root) return true;
8 const queue = [root.left, root.right];
9 while (queue.length) {
10 let len = queue.length;
11 const level = [];
12 while (len) {
13 // 出列
14 const pop = queue.shift();
15 level.push(pop);
16 if (pop) {
17 // 入列
18 queue.push(pop.left);
19 queue.push(pop.right);
20 }
21 len--;
22 }
23 // 判断该层级的所有节点是否是对称的
24 for (let i = 0, l = level.length; i < l / 2; i++) {
25 // 一个为null，一个不为null的则该节点不对称
26 // 返回fasle
27 if (level[i] === null && level[l - i - 1] !== null) return false;
28 if (level[i] !== null && level[l - i - 1] === null) return false;
29
30 // 两个都不是null的情况
31 // 节点不相等 返回false
```

```
32 if (level[i] != null && level[l - i - 1] != null) {
33 if (level[i].val != level[l - i - 1].val) return false;
34 }
35 }
36 }
37 return true;
38 };
```

## 复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(n)$

当树是线性时，我们可能需要向队列中插入  $O(n)$  个节点

## 二叉树的最大深度

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]，

```
1 3
2 / \
3 9 20
4 / \\
5 15 7
```

返回它的最大深度 3。

### 方法一 递归查询排序

### 思路

遍历所有节点的深度，记录所有子节点的深度，然后筛选出最大的深度

## 详解

1. 创建一个空数组用来保存所有节点深度
2. 判断二叉树是否为空，空的直接返回 0，结束，非空二叉树继续
3. 遍历二叉树节点，没有左右子节点的，直接将当前 level 塞入之前定义的深度数组
4. 有左右子节点的就继续递归查询查询子节点的深度，传入的 level 也加 1 传递，直到二叉树遍历结束
5. 对深度数组排序返回最大值，也就是二叉树最大深度

## 代码

```
1 const maxDepth = function (root) {
2 // 创建保存节点深度的空数组
3 const levelList = [];
4 // 判断二叉树是否为空
5 if (root === null) {
6 return 0;
7 } else {
8 loop(root, 1);
9 return sort(levelList);
10 }
11 // 遍历二叉树子节点
12 function loop (node, level) {
13 // 没有子节点的，将当前深度保存进深度数组，有节点的继续遍历
14 if (node.left === null && node.right === null) {
15 levelList.push(level);
16 } else if (node.left !== null) {
17 loop(node.left, level + 1);
18 } if (node.right !== null) {
19 loop(node.right, level + 1);
20 }
21 }
22 // 对数组进行排序，返回最大深度
23 function sort (arr) {
24 arr.sort((a, b) => {
25 return b - a;
26 });
27 return arr[0];
28 }
29}
30};
```

## 复杂度分析

- 时间复杂度： $O(n^2)$

- 空间复杂度： $O(n)$

## 方法二 递归

### 思路

递归二叉树的节点，获取左子树和右子树的最大深度，比较后，返回最大深度

### 详解

1. 判断二叉树是否为空，空的直接返回 0，结束，非空二叉树继续
2. 分别递归计算左右子树的最大深度
3. 根据返回两者的两者数字，比较后的返回二叉树的最大深度

### 代码

```
1 const maxDepth = function (root) {
2 if (root === null) {
3 return 0;
4 } else {
5 const leftDepth = maxDepth(root.left);
6 const rightDepth = maxDepth(root.right);
7 const childDepth = leftDepth > rightDepth ? leftDepth : rightDepth;
8 return 1 + childDepth;
9 }
10};
```

### 复杂度分析

- 时间复杂度： $O(n)$   
通过递归的方式查询了数的所有子节点。查询花费  $O(n)$  的时间。
- 空间复杂度： $O(n)$   
每次递归都需要创建新的临时空间，空间复杂度  $O(n)$

# 二叉树的层次遍历、二叉树的序列化与反序列化 和常数时间内插入删除、获得随机数

## 二叉树的层次遍历

给定一个二叉树，返回其按层次遍历的节点值。 (即逐层地，从左到右访问所有节点)。

### 示例

```
1 给定二叉树: [3,9,20,null,null,15,7]
2
3 3
4 / \
5 9 20
6 / \
7 15 7
8
9 返回其层次遍历结果：
10 [
11 [3],
12 [9,20],
13 [15,7]
14]
```

### 方法一 递归

#### 思路

最简单的解法就是递归，首先确认树非空，然后调用递归函数 `helper(node, level)`，参数是当前节点和节点的层次。

#### 详解

1. 输出列表称为 `levels`，当前最高层数就是列表的长度 `len(levels)`。比较访问节点所在的层次 `level` 和当前最高层次 `len(levels)` 的大小，如果前者更大就向 `levels` 添加一个空列表;
2. 将当前节点插入到对应层的列表 `levels[level]` 中;
3. 递归非空的孩子节点：`helper(node.left / node.right, level + 1)`。

```
1 /**
```

```

2 * Definition for a binary tree node.
3 * function TreeNode(val) {
4 * this.val = val;
5 * this.left = this.right = null;
6 * }
7 */
8 /**
9 * @param {TreeNode} root
10 * @return {number[][]}
11 */
12 const levelOrder = function (root) {
13 const levels = [];
14 if (!root) {
15 return levels;
16 }
17 const helper = function (node, level) {
18 if (levels.length === level) {
19 levels.push([]);
20 }
21 levels[level].push(node.val);
22 if (node.left) {
23 helper(node.left, level + 1);
24 }
25 if (node.right) {
26 helper(node.right, level + 1);
27 }
28 };
29 helper(root, 0);
30 return levels;
31 };

```

## 复杂度分析

- 时间复杂度： $O(n)$   
因为每个节点恰好会被运算一次。
- 空间复杂度： $O(n)$   
保存输出结果的数组包含  $n$  个节点的值。

## 方法二 迭代

### 思路

我们将树上顶点按照层次依次放入队列结构中，队列中元素满足 FIFO（先进先出）的原则。使用了 js 中的 push 和 shift 方法

第 0 层只包含根节点 root，算法实现如下：

初始化队列只包含一个节点 root 和层次编号 0 : level = 0。当队列非空的时候：

- 在输出结果 levels 中插入一个空列表，开始当前层的算法。
- 计算当前层有多少个元素：等于队列的长度。
- 将这些元素从队列中弹出，并加入 levels 当前层的空列表中。
- 将他们的孩子节点作为下一层压入队列中。
- 进入下一层 level++。

```
1 /**
2 * Definition for a binary tree node.
3 * function TreeNode(val) {
4 * this.val = val;
5 * this.left = this.right = null;
6 * }
7 */
8 /**
9 * @param {TreeNode} root
10 * @return {number[][]}
11 */
12 const levelOrder = function (root) {
13 const levels = [];
14 if (!root) {
15 return levels;
16 }
17
18 let level = 0;
19 const queue = [root];
20 while (queue.length) {
21 // 开始当前层级的循环
22 levels.push([]);
23 // 获取当前level的长度
24 const levelLength = queue.length;
25
26 for (let i = 0; i < levelLength; i++) {
27 const node = queue.shift();
28 // 给当前level添加值
29 levels[level].push(node.val);
30
31 // 给当前level添加子节点
32 // 并加入队列，给下一个level使用
33 if (node.left) {
34 queue.push(node.left);
35 }
36 if (node.right) {
37 queue.push(node.right);
38 }
39 }
40
41 // 进入下一个level
42 level += 1;
43 }
44 }
```

```
43 }
44
45 return levels;
46 }
```

## 复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(n)$

## 二叉树的序列化与反序列化

序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。

请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

### 示例

```
1 你可以将以下二叉树：
2
3 1
4 / \
5 2 3
6 / \
7 4 5
8
9 序列化为 "[1, 2, 3, null, null, 4, 5]"
```

### 方法一 深度优先遍历(DFS)

#### 思路

遍历二叉树：L、D、R 分别表示遍历左子树、访问根结点和遍历右子树。先序遍历二叉树的顺序是 DLR，中序遍历二叉树的顺序是 LDR，后序遍历二叉树的顺序是 LRD。深度优先遍历包含先序、中序、后序遍历。先序遍历的顺序：先访问根节点，再遍历左节点，最后遍历右节点。为了能够反序列化，在遍历的时候需要将 null 也保存进去。伪代码如下：

```
1 visit(node)
2 print node.value
3 if node.left != null then visit(node.left)
4 if node.right != null then visit(node.right)
```

示例中的二叉树以本算法先序遍历后的结果为：[1,2,null,null,3,4,null,null,5,null,null] 反序列化：将先序遍历的结果按根节点、左子树、右子树顺序还原，伪代码如下：

```
1 structure();
2 node = new TreeNode();
3 node.left = structure();
4 node.right = structure();
```

## 详解

1. 首先序列化二叉树，定义一个遍历方法，先访问根节点，再遍历左节点，最后遍历右节点，将 null 也保存进数组中
2. 反序列化二叉树，将数组还原回二叉树，因为数组是先序遍历的结果，遍历数组，然后按照根节点、左子树、右子树顺序还原二叉树

```
1 /**
2 * 二叉树节点构造函数
3 * @param {string} val 节点值
4 */
5 function TreeNode (val) {
6 this.val = val;
7 this.left = this.right = null;
8 }
9
10 /**
11 * 序列化二叉树，将二叉树转成数组
12 * @param {TreeNode} root
13 */
14 const serialize = function (root) {
15 const result = [];
16 // 遍历节点
17 function traverseNode (node) {
18 if (node === null) {
19 result.push(null);
20 } else {
21 // 先序遍历，先访问根节点，再遍历左节点，最后遍历右节点
22 result.push(node.val);
23 traverseNode(node.left);
```

```

24 traverseNode(node.right);
25 }
26 }
27
28 traverseNode(root);
29 return result;
30 };
31
32 /**
33 * 反序列化二叉树，将数组还原回二叉树
34 * @param {string} data
35 */
36 const deserialize = function (data) {
37 const length = data.length;
38 if (length === 0) {
39 return null;
40 }
41 let i = 0;
42 // data 结构化成树
43 function structure () {
44 if (i >= length) {
45 return null;
46 }
47 const val = data[i];
48 i++;
49 if (val === null) {
50 return null;
51 }
52 const node = new TreeNode(val);
53 node.left = structure();
54 node.right = structure();
55
56 return node;
57 }
58 return structure();
59 };

```

## 复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(n)$   
 $\text{serialize}$  方法为  $\text{result}$  分配了空间，递归中没有产生新的对象，空间复杂度为  $O(1)$   
 $\text{deserialize}$  方法每次遍历会  $\text{new}$  一个对象，空间复杂度为  $O(n)$

## 方法二 广度优先遍历(BFS)

示例中的二叉树以本算法广度优先(层序)遍历后的结果为：[1,2,3,null,null,4,5]

## 思路

序列化：广度优先遍历序列化二叉树是按层级从上往下将每层节点从左往右依次遍历，用队列来处理遍历，先将根节点入队，然后根节点出队，再左子树和右子树入队，递归遍历即可。反序列化：从序列化好的数组中取第一个元素，生成根节点。将根节点放入队列。循环队列，根节点的左右子树分别放入队列，循环此操作，直到队列为空。

## 详解

序列化：1. 定义一个 result 数组存放序列化结果 1. 定义一个 queue 数组，作为队列 2. 将根节点入队 3. 循环队列，队列中的第一个元素(节点)出队，将此节点值 push 进 result 数组。分别将此节点左右节点入队 4. 当队列为空时，跳出循环 5. 返回 result

反序列化：1. 从 result 取出第一个节点值，生成根节点放到队列中 2. 循环队列，队列中第一个元素(节点)出队，从 result 取出下一个值还原左节点，将此左节点入队，从 result 取出下一个值还原右节点，将此右节点入队 3. 当 result 或队列为空时，跳出循环 4. 返回反序列化好的节点(根节点)

```
1 /**
2 * 二叉树节点构造函数
3 * @param {string} val 节点值
4 */
5 function TreeNode (val) {
6 this.val = val;
7 this.left = this.right = null;
8 }
9
10 /**
11 * 序列化二叉树，将二叉树转成数组
12 * @param {TreeNode} root
13 */
14 const serialize = function (root) {
15 if (!root) {
16 return [];
17 }
18 const result = [];
19 const queue = [];
20 queue.push(root);
21 let node;
22 while (queue.length) {
23 node = queue.shift();
24 result.push(node ? node.val : null);
25 if (node) {
26 queue.push(node.left);
27 queue.push(node.right);
28 }
29 }
30 return result;
```

```

31 };
32
33 /**
34 * 反序列化二叉树，将数组还原回二叉树
35 * @param {string} data
36 */
37 const deserialize = function (data) {
38 const length = data.length;
39 if (length === 0) {
40 return null;
41 }
42 // 取出第一个节点值，生成根节点放到队列中
43 const root = new TreeNode(data.shift());
44 const queue = [root];
45 while (queue.length) {
46 // 取出将要复原的节点
47 const node = queue.shift();
48 // 节点遍历完，跳出循环
49 if (data.length === 0) {
50 break;
51 }
52 // 先还原左节点
53 const leftVal = data.shift();
54 if (leftVal === null) {
55 node.left = null;
56 } else {
57 node.left = new TreeNode(leftVal);
58 // 将生成的左节点放入队列，下次循环会复原此左节点的子节点
59 queue.push(node.left);
60 }
61 // 节点遍历完，跳出循环
62 if (data.length === 0) {
63 break;
64 }
65 // 还原右节点
66 const rightVal = data.shift();
67 if (rightVal === null) {
68 node.right = null;
69 } else {
70 node.right = new TreeNode(rightVal);
71 // 将生成的右节点放入队列，下次循环会复原此左节点的子节点
72 queue.push(node.right);
73 }
74 }
75
76 return root;
77 };

```

## 复杂度分析

- 时间复杂度： $O(n)$

`serialize` 方法中每个节点遍历一次，时间复杂度为  $O(n)$  `deserialize` 方法遍历  $n$  次， $n$  为 `data` 的长度，时间复杂度为  $O(n)$

- 空间复杂度： $O(n)$

`serialize` 方法为 `result` 分配了空间，递归中没有产生新的对象，空间复杂度为  $O(1)$

`deserialize` 方法每次遍历会 `new` 一个对象，空间复杂度为  $O(n)$

## 常数时间内插入删除、获得随机数 □

设计一个支持在平均 时间复杂度  $O(1)$  下，执行以下操作的数据结构。

`insert(val)`：当元素 `val` 不存在时，向集合中插入该项。  
`remove(val)`：元素 `val` 存在时，从集合中移除该项。  
`getRandom`：随机返回现有集合中的一项。每个元素应该有相同的概率被返回。

### 示例

```
1 // 初始化一个空的集合。
2 RandomizedSet randomSet = new RandomizedSet();
3
4 // 向集合中插入 1 。返回 true 表示 1 被成功地插入。
5 randomSet.insert(1);
6
7 // 返回 false ，表示集合中不存在 2 。
8 randomSet.remove(2);
9
10 // 向集合中插入 2 。返回 true 。集合现在包含 [1,2] 。
11 randomSet.insert(2);
12
13 // getRandom 应随机返回 1 或 2 。
14 randomSet.getRandom();
15
16 // 从集合中移除 1 ，返回 true 。集合现在包含 [2] 。
17 randomSet.remove(1);
18
19 // 2 已在集合中，所以返回 false 。
20 randomSet.insert(2);
21
22 // 由于 2 是集合中唯一的数字，getRandom 总是返回 2 。
23 randomSet.getRandom();
```

### 方法一 Array + Object(Map)

#### 思路

使用动态数组存储元素值，对象存储值到索引的映射；有索引可以实现常数时间的 insert 和 getRandom。remove 的常数时间则使用：总是删除最后一个元素，将要删除元素和最后一个元素交换，然后将最后一个元素删除来实现。

## 详解

1. insert: 添加元素到动态数组，并在对象中添加值到索引的映射关系。
2. remove: 在对象中查找要删除元素的索引，将要删除元素与最后一个元素交换，删除最后一个元素，更新对象中的对应关系。
3. getRandom: 借助 random 函数，获取下标范围内的整数，然后取出数组对应下标元素。

```
1 /**
2 * Initialize your data structure here.
3 */
4 const RandomizedSet = function () {
5 this.arr = [];
6 this.values = {};
7 };
8
9 /**
10 * Inserts a value to the set. Returns true if the set did not already contain t
11 * @param {number} val
12 * @return {boolean}
13 */
14 RandomizedSet.prototype.insert = function (val) {
15 if (this.values[val] >= 0) {
16 return false;
17 } else {
18 this.values[val] = this.arr.length;
19 this.arr.push(val);
20 return true;
21 }
22 };
23
24 /**
25 * Removes a value from the set. Returns true if the set contained the specified
26 * @param {number} val
27 * @return {boolean}
28 */
29 RandomizedSet.prototype.remove = function (val) {
30 const i = this.values[val];
31 const l = this.arr.length;
32 if (i >= 0) {
33 this.values[this.arr[l - 1]] = i;
34 this.values[val] = -1;
35 this.arr.splice(i, 1, this.arr[l - 1]);
36 this.arr.pop();
37 return true;
38 } else {
```

```
39 return false;
40 }
41 }
42
43 /**
44 * Get a random element from the set.
45 * @return {number}
46 */
47 RandomizedSet.prototype.getRandom = function () {
48 const l = this.arr.length;
49 const i = Math.floor(Math.random() * l);
50 return this.arr[i];
51 };
52
53 /**
54 * Your RandomizedSet object will be instantiated and called as such:
55 * var obj = new RandomizedSet()
56 * var param_1 = obj.insert(val)
57 * var param_2 = obj.remove(val)
58 * var param_3 = obj.getRandom()
59 */

```

## 复杂度分析

- 时间复杂度： $O(1)$   
getRandom 时间复杂度为  $O(1)$ ，insert 和 remove 平均时间复杂度为  $O(1)$
- 空间复杂度： $O(n)$   
存储了 n 个元素信息

# 中序遍历二叉树、从前序与中序遍历序列构造二叉树和二叉搜索树中第 K 小的元素

## 中序遍历二叉树

给定一个二叉树，返回它的中序遍历。

### 示例

```
1 输入: [1,null,2,3]
2 树结构:
3 TreeNode: {
4 val: 1,
5 right: {
6 val: 2,
7 right: null,
8 left: {
9 val: 3,
10 right: null,
11 left: null,
12 },
13 },
14 left: null
15 }
16 图解:
17 1
18 \
19 2
20 /
21 3
22 输出: [1,3,2]
```

### 方法一 递归算法

#### 思路

对于查询一棵二叉树的所有子节点，由于其层级深且多，使用递归是最直接的方法，加上题目需要中序遍历的条件，我们需要查询完根节点后，将左侧的子节点全部查询出来，递归方法能帮助我们一直查询子节点下是否还有子节点，查询到处于左边的子节点时，插入结果数组，直到查到所有的节点的左侧节点之后，再查询右侧节点，这样查询就实现了中序遍历二叉树。

#### 详解

1. 传入根节点，查询根节点的值插入结果数组，并且查询根节点是否存在左子节点
2. 存在左子节点，将该节点作为参数重新传入递归函数，并将其值插入结果数组
3. 递归函数将会把步骤1和2重复进行，直到当前左边的所有节点都插入了结果数组
4. 当前最左边的子节点都查询完毕后，之前的右子节点也将作为参数重新传入递归函数
5. 此时递归函数还是会先从传入的节点的左边开始查询，直到步骤1和2重复查询到没有左边的子节点为止
6. 所有的右子节点的左子节点也查询结束后，再次查询该节点的右子节点，如此重复直到所有节点查询结束即完成查询

```

1 const inorderTraversal = (root) => {
2 const result = [];
3 const middleSequence = (root) => {
4 if (!root) return;
5 const { left, right, val } = root;
6 left && middleSequence(left);
7 val && result.push(val);
8 right && middleSequence(right);
9 };
10 middleSequence(root);
11 return result;
12 };

```

## 复杂度分析

- 时间复杂度： $O(n)$   
递归函数  $T(n) = 2 * T(n/2) + 1$ ，因此时间复杂度为  $O(n)$
- 空间复杂度： $O(\log n)$

## 方法二 迭代算法

### 思路

迭代算法的思路比较巧妙，结合之前的递归算法，我们知道了要一直先查询左边的节点，那我们不如先将所有的左边的子节点存下来，通过类似堆栈的方式，将左边的子节点都插入一个临时的数组，直到所有的左边子节点都查询完后，我们再将其从临时数组中按插入顺序的倒序移除，即后进先出，并将其值插入结果数组，所有左子节点查询完后，右节点再重复一样的操作，即可实现中序遍历二叉树。

### 详解

1. 声明一个临时数组，传入根节点，并查询该根节点的左子节点并将下一次的迭代对象赋值为该节点，同时插入临时数组
2. 左节点传入迭代函数后，将重复步骤1，一直到查询不到左子节点为止
3. 当第一次传入的左子节点和其以下的所有左子节点被查询完后，迭代的对象会变成undefined
4. 此时我们开始处理这段时间插入的所有左子节点，此时增加一个判断条件，临时数组里是否存在未处理的节点
5. 将最后插入的子节点，也是最深层的子节点的值插入结果数组，并移除出临时数组，再将该节点的右节点作为下一次迭代对象
6. 查询该节点下的所有左子节点，重复步骤1和2，直到查询结束为止，再重复步骤3和4，一直往树的根节点查询，直到结束

```

1 const inorderTraversal = (root) => {
2 const result = [];
3 const stack = [];
4 let node = root;
5 while (node || stack.length > 0) {
6 if (node) {
7 stack.push(node);
8 node = node.left;
9 continue;
10 }
11 node = stack.pop();
12 result.push(node.val);
13 node = node.right;
14 }
15 return result;
16 };

```

## 复杂度分析

- 时间复杂度： $O(n)$   
查询所有节点需要  $O(n)$  的时间
- 空间复杂度： $O(n)$   
创建了长度为n的数组。

## 从前序与中序遍历序列构造二叉树

根据一棵树的前序遍历与中序遍历构造二叉树。

可假设树中没有重复的元素。

## 示例

```
1 给出
2 前序遍历 preorder = [3,9,20,15,7]
3 中序遍历 inorder = [9,3,15,20,7]
4
5 返回如下的二叉树
6 3
7 / \
8 9 20
9 / \
10 15 7
```

## 名词解释

**前序遍历**：首先访问根结点，然后遍历左子树，最后遍历右子树。在遍历左、右子树时，仍然先访问根结点，然后遍历左子树，最后遍历右子树。

示例解析：二叉树的前序遍历先访问根结点为3，然后遍历左子树9，最后遍历右子树；右子树为一棵树，先访问根结点为20，再遍历左子树为15，最后遍历右子树为7。则示例中二叉树的前序遍历为[3, 9, 20, 15, 7]。

**中序遍历**：首先遍历左子树，然后访问根结点，最后遍历右子树。在遍历左、右子树时，仍然先遍历左子树，然后访问根结点，最后遍历右子树。

示例解析：二叉树的中序遍历，先遍历左子树为9，然后访问根结点3，最后遍历右子树；右子树为一棵树，先遍历左子树为15，再访问根结点为20，最后遍历右子树为7。则示例中二叉树的中序遍历为[9, 3, 15, 20, 7]。

## 方法一 递归法

### 思路

前序遍历中首先出现的结点均为根结点，结合中序遍历中对应结点及距离上（下）一结点的中间的数字可确定此结点的左（右）子树中的结点值，在左（右）子树中的结点值确定后，再结合前序遍历的数组可得出左（右）子树的根结点，再根据中序遍历中的左（右）子树的根结点得出左（右）子树的左右子树……如此递归，则可得出完整的二叉树。

根据题目中的例子，前序遍历首先出现的数（即3）为根结点，3在中序遍历中的位置之前的结点9必定为根结点的左子树；在前序遍历中9之后的结点为根结点的右子树；同样的方法确定前序遍历中的20为右子树的根结点，15为右子树的左结点，7为右子树的右结点。

## 详解

1. 根据前序遍历数组找到该树的根结点；

```
const root = preorder[preStart];
```

preorder 为前序遍历的数组，preStart 为前序遍历数组的下标。初始化时置为 0，为二叉树的根结点；之后每次递归均为该子树的根结点的下标。

1. 若中序遍历中根结点的位置之前有值，则证明该根结点有左子树，然后从第一步开始递归，拿到该根结点的左子树；

```
1 // 存在左子树
2 if (inOrderIndex - inStart >= 1) {
3 rootNode.left = constructTreeNode(preorder, inorder, preStart + 1, preStart +
4 }
```

1. 若中序遍历中根结点的位置之后有值，则证明该根结点有右子树，然后从第一步开始递归，拿到该根结点的右子树。

```
1 //存在右子树
2 if (inLength - inOrderIndex >= 1) {
3 rootNode.right = constructTreeNode(preorder, inorder, preStart + (inOrderIndex -
4 }
```

## 代码

```
1 function TreeNode (val) {
2 this.val = val;
3 this.left = this.right = null;
4 }
5
6 function buildTree (preorder, inorder) {
7 if (preorder.length === 0 || inorder.length === 0) {
8 return null;
9 }
10 return constructTreeNode(preorder, inorder, 0, preorder.length, 0, inorder.length);
}
```

```

11 }
12
13 function findInorderIndex (list, target) {
14 if (list.length === 0) {
15 return undefined;
16 }
17
18 let index;
19 list.forEach((item, i) => {
20 if (item === target) {
21 index = i;
22 }
23 });
24 return index;
25 }
26
27 function constructTreeNode (preorder, inorder, preStart, preLength, inStart, inLength) {
28 const root = preorder[preStart];
29 const inOrderIndex = findInorderIndex(inorder, root);
30 // 中序遍历根结点左边为左子树，右边为右子树
31 const rootNode = new TreeNode(root);
32 // 存在左子树
33 if (inOrderIndex - inStart >= 1) {
34 rootNode.left = constructTreeNode(preorder, inorder, preStart + 1, preStart + inOrderIndex - inStart);
35 }
36
37 // 存在右子树
38 if (inLength - inOrderIndex >= 1) {
39 rootNode.right = constructTreeNode(preorder, inorder, preStart + (inOrderIndex - inStart), preStart + inLength);
40 }
41 return (root || root === 0) ? rootNode : null;
42 }

```

## 复杂度分析

- 时间复杂度： $O(n)$

由于每次递归 inorder 和 preorder 的总数都会减 1，因此需要递归  $n$  次，故时间复杂度为  $O(n)$ ，其中  $n$  为结点个数

- 空间复杂度： $O(n)$

所用空间与树本身存储空间正相关

## 方法二 遍历法

### 思路

前序遍历 preorder = [3,9,20,15,7] 中序遍历 inorder = [9,3,15,20,7]

借用了栈的数据结构，先将根结点放入，然后前序遍历数组，若在中序遍历的节点与前序遍历的节点相等（即找到了已遍历节点的右子树的根结点）则从匹配到的节点到该根节点出栈。如此，遍历完前序数组后则可唯一确定一颗完整的二叉树。【重点在于判断何时找到了右子树的根节点，在中序遍历中的节点与前序遍历的当前节点相等时则可确定，原因在于前序遍历中遍历到右子树的根结点且与中序遍历中的节点相等时，必然可以确定匹配节点的左子节点已经遍历完毕；同时确定该子树是哪个根结点的右子树（示例解析中已说明），问题即可得到解决】

### 示例解析：

若当前需要确定的二叉树为示例所示：

```
1 3
2 / \
3 9 20
4 / \
5 15 7
```

首先，假设只有前序遍历的数组preorder = [3,9,20,15,7]，需要确定上面的唯一一颗二叉树，可以做什么？

我们首先可以确定这棵树的根结点为【3】，然后是节点【9】，按照前序遍历的原则，先遍历根结点，再遍历左子节点，然后再遍历右子节点，可以确定的是【9】是【3】的子树的根结点，但是无法确定是左子树的根结点，还是右子树的根结点；

此时需要结合中序遍历的数组inorder = [9,3,15,20,7]来进行判断。可以先假设【9】是【3】的右子树的根结点，根据中序遍历的特点，先遍历左子树，再访问根结点，再遍历右子树可得出中序遍历的顺序为[3,9]，与既定的中序遍历数组的顺序不符，所以可确定【9】是【3】的左子树的根结点；

同时，注意到了此时的前序遍历的【9】和中序遍历的【9】相等了，说明【9】是没有左子节点的，从中序遍历的特点以及【9】在中序遍历中所处的位置可以得出。则前序遍历的下一个节点必定是【20】必定为右子树的根结点，那么【20】是【3】的右子树的根结点，还是【9】的右子树的根结点呢？

假设【20】是【3】的右子树的根结点，那么中序遍历的数组顺序应当是[9,3,20]；假设【20】是【9】的右子树的根结点，那么中序遍历的数组顺序应当是[9,20,3]；由此可确定【20】是【3】的右子树的根结点。

此时我们的中序遍历的数组是[9,3,15,20,7]，【9】匹配，【3】匹配，最后一次匹配是【3】，所以【20】是【3】的右子树。

```
1 3
2 / \
3 9 20
```

综上所述，可用一个栈来记录已经遍历过的节点，遍历前序遍历的数组，作为当前节点的左子树的根结点，直到前序遍历的当前节点与中序遍历的节点相匹配，即找到了遍历过的某个节点的右子树的根结点，则从该匹配节点到最后一个匹配节点出栈，并将当前节点作为最后一个匹配节点的右子树的根结点。如此，将前序与中序数组遍历完毕，便可确定唯一的一颗二叉树。

## 详解

1、根据前序遍历确定该树的根结点，将其放入栈中；

```
treeNodeList.push(root);
```

2、进行前序遍历，若中序遍历中当前值为正好为栈中最后一个根结点时，该根结点出栈，当前前序遍历的结点为该根结点的右子根结点；否则当前前序遍历的结点为该根结点的左子根结点，并将当前值放入栈中，继续遍历，重复第二个步骤。

## 代码

```
1 function TreeNode (val) {
2 this.val = val;
3 this.left = this.right = null;
4 }
5
6 const buildTree = function (preorder, inorder) {
7 if (preorder.length === 0) {
8 return null;
9 }
10 const treeNodeList = [];
11 const root = new TreeNode(preorder[0]);
12 treeNodeList.push(root);
13 // j从0开始，为中序遍历的序数
14 let j = 0;
15 for (let i = 1; i < preorder.length; i++) {
16 const current = new TreeNode(preorder[i]);
17 let curParent;
18
19 // 中序遍历至当前根结点时，右子树根结点确定，当前根结点出栈
20 while (treeNodeList.length !== 0 && treeNodeList[treeNodeList.length - 1].va
```

```

21 curParent = treeNodeList[treeNodeList.length - 1];
22 treeNodeList.length--;
23 j++;
24 }
25 if (curParent) {
26 curParent.right = current;
27 } else {
28 treeNodeList[treeNodeList.length - 1].left = current;
29 }
30 treeNodeList.push(current);
31 }
32 return root;
33 };

```

## 复杂度分析

- 时间复杂度： $O(n)$   
前序遍历与中序遍历的序数变动基本一致
- 空间复杂度： $O(n)$   
所用空间与二叉树的左子树的深度正相关为  $O(\log_2 n)$ ，树本身存储空间为  $O(n)$ ，取大者

## 二叉搜索树中第 K 小的元素

给定一个二叉搜索树，编写一个函数 `kthSmallest` 来查找其中第  $k$  个最小的元素。

**说明** 你可以假设  $k$  总是有效的， $1 \leq k \leq$  二叉搜索树元素个数。

### 示例 1：

```

1 输入: root = [3,1,4,null,2], k = 1
2 3
3 / \
4 1 4
5 \
6 2
7 输出: 1

```

### 示例 2：

```

1 输入: root = [5,3,6,2,4,null,null,1], k = 3

```

```
2 5
3 / \
4 3 6
5 / \
6 2 4
7 /
8 1
9 输出: 3
```

进阶：如果二叉搜索树经常被修改（插入/删除操作）并且你需要频繁地查找第  $k$  小的值，你将如何优化  $kthSmallest$  函数？

## 方法一 递归查找

根据二叉搜索树的特性：

若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值

## 思路

我们只需要中序遍历输入的树，然后输出第  $k$  个元素即可以得到第  $k$  个最小的元素。最简单的办法就是写一个递归函数进行遍历然后将遍历结果保存到数组中。

## 详解

中序遍历的原则是：先遍历左子树，然后访问根节点，最后遍历右边子树，当发现已经找到第  $k$  个元素，提前中止遍历

## 代码

```
1 /**
2 * Definition for a binary tree node.
3 * function TreeNode(val) {
4 * this.val = val;
5 * this.left = this.right = null;
6 * }
7 */
8 /**
9 * @param {TreeNode} root
10 * @param {number} k
11 * @return {number}
12 */
13 const kthSmallest = function (root, k) {
```

```

14 const result = [];
15
16 function travel (node) {
17 // 当已经找到 k 个元素时提前中止遍历
18 if (result.length >= k) return;
19 if (node.left) {
20 // 遍历左子树
21 travel(node.left);
22 }
23 // 保存根节点
24 result.push(node.val);
25 if (node.right) {
26 // 遍历右子树
27 travel(node.right);
28 }
29 }
30 travel(root);
31 return result[k - 1];
32 }

```

## 复杂度分析

- 时间复杂度： $O(n)$

时间复杂度与节点个数相关， $n$  个节点最多需要递归查找  $n$  次，所以时间复杂度为  $O(n)$

。

- 空间复杂度： $O(n)$

空间复杂度与调用堆栈有关，调用栈需要记住每个节点的值，所以空间复杂度为  $O(n)$ 。

## 方法二 循环查找

### 思路

还是一样采用中序遍历，只不过我们不使用递归，改为循环的方式实现

### 详解

见代码注释

### 代码

```

1 /**
2 * Definition for a binary tree node.
3 * function TreeNode(val) {
4 * this.val = val;

```

```

5 * this.left = this.right = null;
6 * }
7 */
8 /**
9 * @param {TreeNode} root
10 * @param {number} k
11 * @return {number}
12 */
13 const kthSmallest = function (root, k) {
14 const result = []; let current = root; const stack = [];
15 while (result.length < k && (current || stack.length > 0)) {
16 if (current) {
17 // 有左孩子，表示有比当前元素更小的，继续查找
18 if (current.left) {
19 // 把当前节点暂存到堆栈
20 stack.push(current);
21 // 继续查找左子树
22 current = current.left;
23 } else {
24 // 没有左孩子表示当前元素是目前最小，存入数组
25 result.push(current.val);
26 // 左子树查找完后开始查找右子树
27 current = current.right;
28 }
29 } else {
30 // 已经遍历到叶子节点，需要回溯，从节点堆栈中弹出一个节点
31 current = stack.pop();
32 // 由于左子树已经查找完成，那么当前节点是目前最小的节点
33 result.push(current.val);
34 // 然后继续查找右子树
35 current = current.right;
36 }
37 }
38 return result[k - 1];
39 };

```

## 复杂度分析

- 时间复杂度： $O(n)$

时间复杂度与节点个数相关， $n$  个节点最多需要循环查找  $n$  次，所以时间复杂度为  $O(n)$

。

- 空间复杂度： $O(n)$

由于需要一个需要辅助栈来记住节点的值，最坏情况下所有节点都会存进辅助栈，所以空间复杂度为  $O(n)$ 。

# 填充每个节点的下一个右侧节点指针、岛屿数量和二叉树的锯齿形层次遍历

## 填充每个节点的下一个右侧节点指针

给定一个**完美二叉树**，其所有叶子节点都在同一层，每个父节点都有两个子节点。二叉树定义如下：

```
1 struct Node {
2 int val;
3 Node *left;
4 Node *right;
5 Node *next;
6 }
```

填充它的每个 `next` 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 `next` 指针设置为 `NULL`。初始状态下，所有 `next` 指针都被设置为 `NULL`。

### 示例

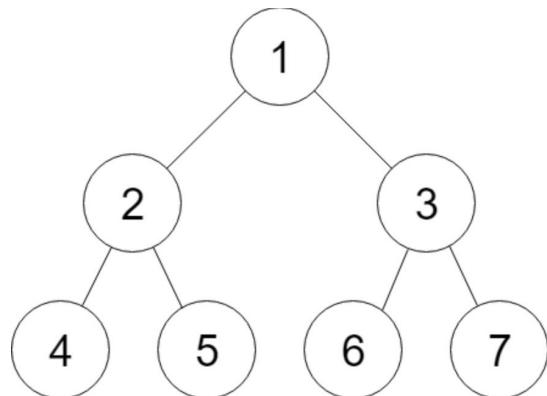


Figure A

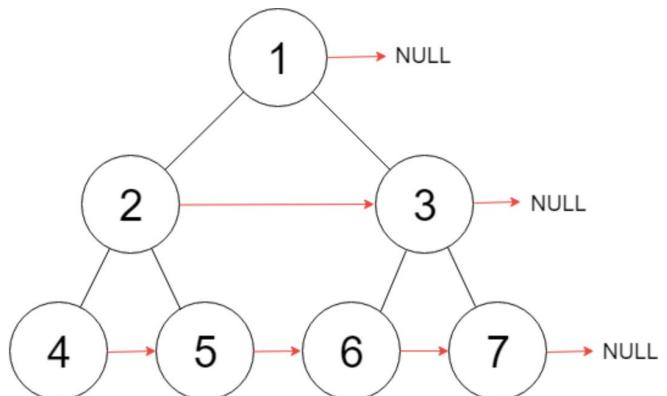


Figure B

```
1 输入：{"$id": "1", "left": {"$id": "2", "left": {"$id": "3", "left": null, "next": null, "right": null}, "left": null, "next": null, "right": null}
2
3 输出：{"$id": "1", "left": {"$id": "2", "left": {"$id": "3", "left": null, "next": {"$id": "4", "left": null, "next": null, "right": null}, "right": null}, "left": null, "next": null, "right": null}
4
5 解释：给定二叉树如图 A 所示，你的函数应该填充它的每个 next 指针，以指向其下一个右侧节点，如
```

## 提示

- 你只能使用常量级额外空间。
- 使用递归解题也符合要求，本题中递归程序占用的栈空间不算做额外的空间复杂度。

## 方法一 递归法

### 思路

使用树的先序遍历（递归方式）。这里需要解决两种情形，第一，同一父节点的左右节点的连接，如图中的 2、3 节点；第二，相近但非同一直系父节点的节点相连，如图中的 5、6 节点。

### 详解

1. 第一步，同一父节点的左右节点的连接：先将同一父节点的左节点的 `next` 指向右节点，以图例中的节点 1 为例，其左节点 2 的 `next` 指向右节点 3。
2. 第二步，相近但非同一直系父节点的节点相连：考虑到不同直系父节点的同一层节点也需要相连，以图例中的节点 5、6 为例，5 的 `next` 指向 6。我们可以通过 5 的父节点 2 的 `next` 节点找到 3，再通过节点 3，找到节点 6，即访问节点 3 的左节点。以此类推，无论下面还有多少层，都可以通过这种方法，连接相近但不是同一直系父节点的两个子节点。

### 代码

```
1 /**
2 * // Definition for a Node.
3 * function Node(val, left, right, next) {
4 * this.val = val === undefined ? null : val;
5 * this.left = left === undefined ? null : left;
6 * this.right = right === undefined ? null : right;
7 * this.next = next === undefined ? null : next;
8 * };
9 */
10 /**
11 * @param {Node} root
12 * @return {Node}
13 */
14 const connect = (root) => {
15 // 递归出口
16 if (root === null) {
17 return root;
18 }
19 // 同一父节点的左右节点相连，如图中 2、3 节点相连
20 if (!!root.left && !!root.right) {
```

```

21 root.left.next = root.right;
22 }
23 // 相近但非同一直系父节点的节点相连，如图中 5、6 节点相连
24 if (!!root.right && root.next && root.next.left) {
25 root.right.next = root.next.left;
26 }
27 connect(root.left);
28 connect(root.right);
29 return root;
30 };

```

## 复杂度分析

- 时间复杂度： $O(n)$

上述解法中，树的每个节点只访问了一次，时间复杂度跟树的节点个数线性相关，因此为  $O(n)$ 。

- 空间复杂度： $O(1)$

由于题目提示中声明，递归程序占用的栈空间不算做额外的空间复杂度，因此为  $O(1)$ 。

## 方法二 层序遍历法（队列方式）

### 思路

使用层序遍历的方法，每一层按从左到右进行遍历，把遍历出来的节点依次连接起来，但在连接的时候需要注意，每层的最后一个节点的 `next` 需要置为 `null`，而不是和前一个节点相连。

### 详解

- 第一步，进行二叉树的层序遍历，每一层按从左到右进行遍历，把遍历出来的节点依次连接起来。**补充说明**：如何进行二叉树的层序遍历？核心思想是使用队列先进先出的特性。以上述示例图中的二叉树为例。首先初始化，将根节点 1 推入队列，然后，首位（即节点 1）出队列，并将其左子节点 2 和右子节点 3 推入队列，然后，首位（即节点 2）出队列，并将其左子节点 4 和右子节点 5 推入队列，接下去，节点 3 出队列，节点 4 5 入队列，依次类推，直到队列为空，二叉树遍历结束。
- 第二步，在连接的时候，判断当前遍历到的节点是否为该层的最右节点。具体判断方法如下：因为层序遍历过程中，队列的 `size` 即为当前层节点的总个数，使用 `for` 循环进行当前层的遍历，循环执行的最后一次就是该层最后一个节点。

```

1 /**
2 * // Definition for a Node.
3 * function Node(val, left, right, next) {

```

```

4 * this.val = val === undefined ? null : val;
5 * this.left = left === undefined ? null : left;
6 * this.right = right === undefined ? null : right;
7 * this.next = next === undefined ? null : next;
8 * };
9 */
10 /**
11 * @param {Node} root
12 * @return {Node}
13 */
14 const connect = (root) => {
15 const arr = [];
16 if (root === null) {
17 return root;
18 }
19 arr.push(root);
20 while (arr.length) {
21 // size 为每一层节点的总个数。 prevNode 记录前一个节点。
22 const size = arr.length;
23 let prevNode;
24 let node;
25 // 遍历每一层的节点
26 for (let i = 0; i < size; i += 1) {
27 node = arr.shift(); // 出队列
28 // 每一层最后一个节点的 next 需要置为 null
29 // 因此，当前节点不是当前层的最后一个节点的话，将当前节点与前一节点连接
30 if (prevNode && i < size) {
31 prevNode.next = node;
32 }
33 if (node.left) {
34 arr.push(node.left); // 左节点入队列
35 }
36 if (node.right) {
37 arr.push(node.right); // 右节点入队列
38 }
39 prevNode = node;
40 }
41 }
42 return root;
43 };

```

## 复杂度分析

- 时间复杂度： $O(n)$

上述解法中，树的每个节点只访问了一次，时间复杂度跟树的节点个数线性相关，因此为  $O(n)$ 。

- 空间复杂度： $O(n)$

由于解法中使用到队列，且在访问最下面一层叶子节点时，空间占用达到最大，即存储了  $(n + 1)/2$  个节点，因此为  $O(n)$ 。

## 方法三 层序遍历法（指针方式）

### 思路

使用 2 个指针 `start` 和 `current` 来进行层序遍历。其中 `start` 用于标记每一层的第一个节点，`current` 用来遍历该层的其他节点。

### 详解

1. 第一步，□ 定义一个 `start` 指针□，用于标记每一层的第一个节点，`start` 指针的初始化值为 `root` 节点，只需要 `start = start.left`，就能获取到每一层的第一个节点。
2. 第二步，定义一个 `current` 指针，用来遍历该层的其他节点。然后，将同一父节点的左右节点的连接，即 `current.left.next = current.right`，如图中的 2、3 节点；将相近但非同一直系父节点的节点相连，即 `current.right.next = current.next.left`，如图中的 5、6 节点。

### 代码

```
1 /**
2 * // Definition for a Node.
3 * function Node(val, left, right, next) {
4 * this.val = val === undefined ? null : val;
5 * this.left = left === undefined ? null : left;
6 * this.right = right === undefined ? null : right;
7 * this.next = next === undefined ? null : next;
8 * };
9 */
10 /**
11 * @param {Node} root
12 * @return {Node}
13 */
14 const connect = (root) => {
15 if (root === null) {
16 return root;
17 }
18 let start = root;
19 let current = null;
20 // start 为每一层的第一个节点
21 while (start.left) {
22 // 每一层节点的遍历
23 current = start;
24 while (current) {
25 // 同一父节点的左右节点相连，如图中 2、3 节点相连
26 current.left.next = current.right;
27 // 相近但非同一直系父节点的节点相连，如图中 5、6 节点相连
28 if (current.next) {
```

```
29 current.right.next = current.next.left;
30 }
31 current = current.next;
32 }
33 start = start.left;
34 }
35 return root;
36 };
```

## 复杂度分析

- 时间复杂度： $O(n)$

本解法中，外层循环总共执行  $k$  次（其中  $k$  为树的最大深度），内层循环根据所在的层次不同而不同，第一层 1 次，第二层 2 次，第  $k$  层  $2^{k-1}$  次，而  $1 + 2 + \dots + (2^{k-1}) = n$ ，即所有节点数之和，因此时间复杂度为  $O(n)$ 。

- 空间复杂度： $O(1)$

由于解法中只申请了 2 个变量，空间复杂度与树的节点个数  $n$  无关，因此为  $O(1)$ 。

## 岛屿数量

给定一个由 '1' (陆地) 和 '0' (水) 组成的二维网格，计算岛屿的数量。一个岛被水包围，并且它是通过水平方向或垂直方向上相邻的陆地连接而成的。你可以假设网格的四个边均被水包围。

### 示例 1：

```
1 输入:
2 11110
3 11010
4 11000
5 00000
6
7 输出: 1
```

### 示例 2：

```
1 输入:
2 11000
3 11000
4 00100
5 00011
```

```
6
7 输出: 3
```

## 方法一 深度优先遍历

### 思路

遍历二维数组，当节点为陆地(1)时，对当前节点的上下左右四个方向启动深度优先遍历搜索，并将计数器加 1。同时在搜索过程中，遇到海水(0)节点便停止，遇到陆地(1)节点便标记为海水(0)节点。

### 详解

1. 定义岛屿数量计数变量 `landNum`
2. 对二维数组 `grid` 进行两层遍历
3. 遍历过程中，遇到为 `1` 的陆地则将 `landNum` 自增 1，然后进入传播函数，并传入当前的坐标 `i`、`j`
4. 根据传入的坐标，判断是否超出 `grid` 边界，并判断是否为 `0`
5. 若为超出边界，或为 `0`，则停止传播
6. 若为边界内的 `1`，则将该位置变为 `0`，并对此节点的上下左右节点继续递归传播，以此实现深度优先遍历
7. 传播结束后，便可根据 `landNum` 获得岛屿数量

### 代码

```
1 /**
2 * @param {character[][]} grid
3 * @return {number}
4 */
5 var numIslands = function(grid) {
6 let landNum = 0
7
8 for(let i = 0; i < grid.length; i++) {
9 const len = grid[i].length;
10 for(let j = 0; j < len; j++) {
11 const target = grid[i][j]
12 if (target === '1') {
13 spread(grid, i, j)
14 landNum++
15 }
16 }
17 }
18 return landNum;
```

```

19 };
20
21 /**
22 * @param {character[][]} grid
23 * @param {number} i
24 * @param {number} j
25 */
26 function spread(grid, i, j) {
27 if (i < 0 || j < 0 || i >= grid.length || j >= grid[i].length || grid[i][j] != '0')
28 return
29 }
30
31 grid[i][j] = '0'
32 spread(grid, i, j + 1);
33 spread(grid, i, j - 1);
34 spread(grid, i + 1, j);
35 spread(grid, i - 1, j);
36 }

```

## 复杂度分析

- 时间复杂度： $O(n)$   
 $n$  为传入的二维网络的节点个数
- 空间复杂度： $O(n)$   
最坏情况下为  $O(n)$ ，此时整个网格均为陆地

## 方法二 广度优先遍历

### 思路

遍历二维数组，当节点为陆地(1)时，启动广度优先遍历搜索，将节点坐标放入队列中，并将计数器加1。在搜索过程中，遇到陆地(1)节点便标记为海水(0)节点，迭代搜索队列中的每个结点，直到队列为空。

### 详解

1. 定义岛屿数量计数变量 `landNum`
2. 对二维数组 `grid` 进行两层遍历
3. 遍历过程中，遇到为 `1` 的陆地则将 `landNum` 自增 1，然后进入传播函数，并传入当前的坐标 `i`、`j`
4. 根据传入的坐标，构造成 `queue` 队列数组
5. 循环判断 `queue` 的数组长度
6. 若数组中存在坐标，则将末尾坐标从 `queue` 中 `pop` 取出

7. 判断取出的坐标是否为边界内的 1，若是，则将此坐标设置为 0，并将此坐标的上下左右坐标存入 queue 数组中，以此完成广度优先遍历
8. 遍历结束后，便可根据 landNum 获得岛屿数量

## 代码

```
1 /**
2 * @param {character[][]} grid
3 * @return {number}
4 */
5 var numIslands = function(grid) {
6 let landNum = 0
7
8 for(let i = 0; i < grid.length; i++) {
9 const len = grid[i].length;
10 for(let j = 0; j < len; j++) {
11 const target = grid[i][j]
12 if (target === '1') {
13 spread(grid, i, j)
14 landNum++
15 }
16 }
17 }
18
19 return landNum;
20 };
21
22
23 /**
24 * @param {character[][]} grid
25 * @param {number} i
26 * @param {number} j
27 */
28 function spread(grid, i, j) {
29 const queue = [[i, j]]
30
31 while (!queue.length) {
32 const [i, j] = queue.pop()
33 if (grid.length > i
34 && i >= 0
35 && grid[0].length > j
36 && j >= 0
37 && grid[i][j] === '1') {
38
39 grid[i][j] = '0'
40 queue.push([i - 1, j])
41 queue.push([i + 1, j])
42 queue.push([i, j + 1])
43 queue.push([i, j - 1])
44 }
45 }
46 }
```

## 复杂度分析

- 时间复杂度： $O(n)$   
n 为传入的二维网络的节点个数
- 空间复杂度： $O(n)$   
最坏情况下为  $O(n)$ ，此时整个网格均为陆地

## 二叉树的锯齿形层次遍历

给定一个二叉树，返回其节点值的锯齿形层次遍历。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

### 示例

给定二叉树 [3, 9, 20, null, null, 15, 7]，

```

1 3
2 / \
3 9 20
4 / \
5 15 7

```

返回锯齿形层次遍历如下：

```

1 [
2 [3],
3 [20,9],
4 [15,7]
5]

```

## 方法一 双栈法

### 思路

栈：是一种数据结构，先进后出原则。

双栈法：

1. 定义两个数组，一个接收奇数层元素，另一个接收偶数层元素。
2. 奇数层遍历完成之后，将下一层元素由左向右插入偶数层数组。
3. 先进后出原则，偶数列遍历时，取值顺序就变成了由右向左取值。
4. 偶数层遍历完成之后，将下一层元素由右向左插入奇数层数组。
5. 先进后出原则，奇数列遍历时，取值顺序就变成了由左向右取值。
6. 循环往复，形成锯齿形层次遍历

## 详解

1. 定义结果数组
2. 定义两个数组模拟栈(l2r、r2l)，一个由左向右，一个由右向左
3. 将要处理的数据 push 到 l2r，进行循环
4. 每层循环定义一个 临时数组，接收当前层的结果，最后需要 push 到结果数组。
5. 第一层 l2r 就一个根数据，直接就将当前值 push 到 临时数组。为了方便交替遍历，将 l2r 的下一层数据 由左向右 push 到 r2l
6. 第二层 r2l 的数据是由左向右，先进后出，所以我们循环取值是 由右向左，将当前结果 push 到 临时数组，然后将 r2l 的下一层数据 由右向左 push 到 l2r
7. 循环往复

## 代码

```
1 const zigzagLevelOrder = function (root) {
2 if (!root) return [];
3 const res = [];
4 const l2r = [];
5 const r2l = [];
6 l2r.push(root);
7 while (l2r.length || r2l.length) {
8 const temp = [];
9 if (l2r.length) {
10 while (l2r.length) {
11 const cur = l2r.pop();
12 temp.push(cur.val);
13 if (cur.left) r2l.push(cur.left);
14 if (cur.right) r2l.push(cur.right);
15 }
16 } else if (r2l.length) {
17 while (r2l.length) {
18 const cur = r2l.pop();
19 temp.push(cur.val);
20 if (cur.right) l2r.push(cur.right);
21 if (cur.left) l2r.push(cur.left);
22 }
23 }
24 res.push(temp);
25 }
26
27 const root = new TreeNode(3);
28 root.left = new TreeNode(9);
29 root.right = new TreeNode(20);
30 root.left.left = new TreeNode(5);
31 root.left.right = new TreeNode(15);
32 root.right.left = new TreeNode(7);
33 root.right.right = new TreeNode(10);
34
35 zigzagLevelOrder(root);
```

```
23 }
24 res.push(temp);
25 }
26 return res;
27 };
```

## 复杂度分析

- 时间复杂度： $O(n)$

每个节点都要进栈和出栈，所以时间复杂度为  $O(n)$

- 空间复杂度： $O(n)$

双栈每个节点的值也只记录一次，所以空间复杂度为  $O(n)$

## 方法二 递归

### 思路

采用递归，一层层遍历。每一层创建一个数组，奇数层元素从左向右插入数组，偶数层元素从右向左插入数组。

### & 与操作符 判断奇偶

### 详解

1. 先在最顶层定义返回结果数组

2. 然后看递归方法

1. 定义两个参数。第一个  $i$  层数减一，对应返回结果数组的索引值；第二个  $current$ ，当前处理对象；
2. 首先判断结果数组当前索引的是否为第一次创建，不是创建新数组
3. 索引为奇数，对应树形结构偶数行，从右向左插入
4. 索引为偶数，对应树形结构奇数行，从左向右插入
5. 然后不断递归

### 代码

```
1 const zigzagLevelOrder = function (root) {
2 const res = [];
3 dfs(0, root);
4 return res;
5 function dfs (i, current) {
```

```
6 if (!current) return;
7 // 首次进入该层递归，现在结果创建新数组用来接收结果。
8 if (!Array.isArray(res[i])) res[i] = [];
9 // 判断当前层数索引奇偶
10 // 奇数从前插入数组，偶数从后插入数组
11 if (i & 1) res[i].unshift(current.val);
12 else res[i].push(current.val);
13 // 左侧子二叉树进入递归
14 dfs(i + 1, current.left);
15 // 右侧子二叉树进入递归
16 dfs(i + 1, current.right);
17 }
18 };
```

## 复杂度分析

- 时间复杂度： $O(n)$

因为每个节点恰好会被运算一次

- 空间复杂度： $O(n)$

系统栈需要记住每个节点的值，所以空间复杂度为  $O(n)$

# 动态规划

使用递归去解决问题虽然代码简洁、简单，但是效率不高。很多用递归解决的算法题，如果用动态规范来解决，效率会更高。

动态规划(dynamic programming)是通过组合子问题的解决，从而解决整个问题的算法。英文中的 `programming`，指的是一种规划，而不是计算机代码。

动态规划适用于子问题不是独立的情况，对每个子问题只求解一次，使用数组来建立一张表格，来存放被分解成众多子问题的解，从而避免重复计算相同的子问题。

本章节分为 3 个部分：

- Part 1
  - 最大子序和
  - 爬楼梯
  - 买卖股票的最佳时机
- Part 2
  - 打家劫舍
  - 零钱兑换
  - 跳跃游戏
- Part 3
  - 不同路径
  - Longest Increasing Subsequence
  - 单词拆分

阅读完本章节，你将对动态规划算法有更加熟悉对了解。

# 最大子序和、爬楼梯和买卖股票的最佳时机

## 最大子序和

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

### 示例

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

### 方法一 暴力破解

#### 思路

从数组最左边开始于数组右边数据依次相加，将相加之后数据进行比较，比较之后最大值为最终结果

#### 详解

1. 创建临时变量 `sum` 和最大值 `maxNumber`
2. 从数组子序列左端开始遍历依次取数据
3. 从数组子序列右端开始遍历依次取数据和数组左边数据依次相加
4. 将相加之后值与最大值 `sum` 进行比较，大的值赋值与 `maxNumber`
5. 最终获得最大值

```
1 /**
2 * @param {number[]} nums
3 * @return {number}
4 */
5 const maxSubArray = function (nums) {
6 let sum = 0;
7 let maxNumber = 0;
8
9 for (let i = 0; i < nums.length; i++) { // 从数组子序列左端开始
10 for (let j = i; j < nums.length; j++) { // 从数组子序列右端开始
11 sum = 0;
12 for (let k = i; k <= j; k++) { // 暴力计算
13 sum += nums[k];
14 }
15 if (sum > maxNumber) {
16 maxNumber = sum;
17 }
18 }
19 }
20 return maxNumber;
21}
```

```

13 sum += nums[k];
14 }
15 if (sum > maxNumber) {
16 maxNumber = sum;
17 }
18 }
19 }
20 return maxNumber;
21 };

```

## 复杂度分析

- 时间复杂度： $O(n^3)$  对于每个元素，通过三次遍历数组的其余部分来寻找它所对应的目标元素，这将耗费  $O(n^3)$  的时间。
- 空间复杂度： $O(1)$

## 方法二 动态规划法

### 思路

数组从左端开始依次和右端数据相加，两数之和为最大数 sum。下一次相加之后和最大数进行比较，较大数赋值与 sum 由于有负数存在，如果两数相加之后为负数，则两数之和后的最大数为上一个数。

### 详解

- 从数组获取第一个值为最大值 sum 和中间值 n
- 遍历数组，如果中间值n大于0,则和中间值相加，相加结果和最大值比较，较大值赋值与 sum
- 如果中间值小于0，则将当前值作为中间值

```

1 /**
2 * @param {number[]} nums
3 * @return {number}
4 */
5 const maxSubArray = function (nums) {
6 let sum = nums[0];
7 let n = nums[0];
8 for (let i = 1; i < nums.length; i++) {
9 if (n > 0) n += nums[i]; // 判断中间值是否大于0
10 else n = nums[i];
11 if (sum < n) sum = n; // 相加后的值和最大值作比较
12 }
13 return sum;
14 };

```

## 复杂度分析

- 时间复杂度： $O(n)$

对于每个元素，通过遍历数组的其余部分来寻找它所对应的目标元素，这将耗费  $O(n)$  的时间。

- 空间复杂度： $O(1)$

## 爬楼梯

假设你正在爬楼梯。需要  $n$  阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定  $n$  是一个正整数。

### 示例一

```
1 输入： 2
2 输出： 2
3 解释： 有两种方法可以爬到楼顶。
4 1. 1 阶 + 1 阶
5 2. 2 阶
```

### 示例二

```
1 输入： 3
2 输出： 3
3 解释： 有三种方法可以爬到楼顶。
4 1. 1 阶 + 1 阶 + 1 阶
5 2. 1 阶 + 2 阶
6 3. 2 阶 + 1 阶
```

## 方法一 递归法

### 思路

1. 假设现在输入  $n = 10$ ，记作  $f(n)$ ，那么  $10$  个台阶 =  $9$  个台阶 + 走  $1$  步，此处记作  $f(n - 1)$ ，也可以是  $10$  个台阶 =  $8$  个台阶 + 走  $2$  步，记作  $f(n - 2)$
2. 步骤 1 的  $f(n - 1)$ ： $9$  个台阶 =  $8$  个台阶 + 走  $1$  步，也可以是  $9$  个台阶 =  $7$  个台阶 + 走  $2$  步
3. 步骤 1 的  $f(n - 2)$ ： $8$  个台阶 =  $7$  个台阶 + 走  $1$  步，也可以是  $8$  个台阶 =  $6$  个台阶 + 走  $2$  步
4. 以此类推，可以得出递归函数： $f(n) = f(n - 1) + f(n - 2)$

## 详解

从上述思路得出的"递归函数一"如下所示，但是由于执行效率低下，需要进行算法优化。通常情况下，我们提高执行速度的方式是用"空间换取时间"，顾名思义，占用更多的内存来减少计算时间，请看"递归函数二"：

1. 申请一个Object用于存放已经计算过的楼梯  $\$map[n]=num\$$
2. 每次函数执行前，先判断当前楼层是否已经被计算过，是，则直接从 map 中获得结果；否，则进入计算，并在 map 中记录计算结果

## 代码

```

1 // 递归函数一
2 const climbStairs = function (n) {
3 if (n <= 2) {
4 return n;
5 } else {
6 return climbStairs(n - 1) + climbStairs(n - 2);
7 }
8 };

```

```

1 // 递归函数二
2 const fn = (n, map) => {
3 if(n <= 2) {
4 return n
5 }
6 const result = map[n];
7 if(result) {
8 return result;
9 } else {
10 let num = fn(n - 1, map) + fn(n - 2, map);
11 map[n] = num;
12 return num
13 }
14 }
15
16 /**
17 * @param {number} n
18 * @return {number}

```

```
19 */
20 const climbStairs = (n) => {
21 const map = {};
22 return fn(n, map)
23 };
```

- 时间复杂度： $O(2^n)$

- 递归函数一，由于每个节点都需要计算，则时间复杂度就等于二叉树的节点数  $(2^n - 1)$
- 经过递归函数二的优化以后，避免了重复的运算，时间复杂度也就变成了二叉树的高度： $O(n)$

- 空间复杂： $O(n)$

## 方法二 动态规划

### 思路

我们先从给的示例入手，示例中说到：2个台阶有2种方法，3个台阶有3种方法，那么以此基础。4个台阶：3个台阶走一步或者2个台阶走2步，可以算出4个台阶有  $3 + 2 = 5$  种方法。5个台阶：4个台阶走一步或者3个台阶走2步，也就是  $5 + 3 = 8$  种方法。以此类推，我们可以发现，这就是经典的斐波那切数列： $f(n)=f(n-1)+f(n-2)$

### 详解

- 申明变量 *result*，记录一些已知结果，比方说：1个台阶1种解法，2个台阶2种解法
- 为了方便根据数组下标进行查找，在 *result* 中加一个0占位： $result = [0, 1, 2]$ ；
- 当输入 *n* 大于等于 3 时，开始循环计算并记录结果：  
 $result[i] = result[i - 1] + result[i - 2];$
- 循环结束后，输出 *result* 下标为 *n* 的结果。

### 代码

```
1 const climbStairs = function (n) {
2 const result = [0, 1, 2];
3 for (let i = 3; i <= n; i += 1) {
4 result[i] = result[i - 1] + result[i - 2];
5 }
6 return result[n];
7 };
```

## 复杂度分析

- 时间复杂度： $O(n)$   
对  $n$  进行了一次循环遍历，运行次数与输入  $n$  成正比
- 空间复杂度： $O(n)$   
创建了一个长度为  $n$  的空间，空间复杂度是  $O(n)$

## 买卖股票的最佳时机

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

### 示例 1：

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

### 示例 2：

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

## 方法一 穷举法

### 思路

首先，我们想到直观解法，即计算出第  $i$  天买入，后续所有可能卖出情况下的收益，取最大值即为第  $i$  天买入可获得的最大收益。

然后比较每一天的最大收益，取最大值，即可得出所有操作中能获取的最大收益。

### 详解

假设第  $i$  天当天买入，依次遍历第  $i$  天之后的所有可能卖出的情况，比较得出收益中的最大值  $\max$ 。假设  $\text{maxProfit}$  为当前可以获得的最大收益，初始值为 0，将第  $i$  天买入收益最大值  $\max$  与  $\text{maxProfit}$  比较，如果  $\max > \text{maxProfit}$  则更新  $\text{maxProfit}$  的值，依次进行，最终得到最大收益。

## 代码

```
1 /**
2 * @param {number[]} prices
3 * @return {number}
4 */
5 function maxProfit(prices) {
6 let maxProfit = 0;
7
8 function getMax(i) { // 获取第 i 天后股票价格中的最大值
9 let max = prices[i + 1];
10
11 for (let j = i + 1; j < prices.length; j++) {
12 if (prices[j] > max) {
13 max = prices[j];
14 }
15 }
16
17 return max;
18 }
19
20 for (i = 0; i < prices.length - 1; i++) {
21 const max = getMax(i) - prices[i]; // 记录第 i 天买入后续合理时间卖出可获得的最大
22 maxProfit = Math.max(maxProfit, max); // 比较当前已经获取到的最大收益与当天最大收
23 }
24
25 return maxProfit;
26 }
```

## 复杂度分析

- 时间复杂度： $O(n^2)$

我们使用了双重循环计算，内层循环求第  $i$  天买入后的日子里股票的最高价格，外层循环比较计算最大收益  $\text{maxProfit}$ 。那么第一天需要比较  $n - 1$  次才能求出后续最大价格，第二天比较  $n - 2$  次，以此类推... 根据等差数列求和公式，最后比较的总次数为  $n * (n - 1)/2$ ，所以最终得出时间复杂度为  $O(n^2)$ 。

- 空间复杂度： $O(n)$

使用长度为  $n$  的额外数组保存每日可获得的最大收益，即空间复杂度为  $O(n)$

## 方法二 求最大差值

### 思路

再次理解题意，每一天的股票价格组成一个数组，本质上我们只需要寻找一个数组中下标大的数值减去下标小的数值的最大差值即可，而这个差值即为最大收益。

### 详解

我们先假设最大利润为 maxProfit 和最小成本为 minPrice，令 minPrice 为数组中第一个元素，然后开始遍历数组。

当遍历到某一元素 prices[i] 时： 1. 如果 prices[i] 小于 minPrice，将 prices[i] 的值赋给 minPrice 2. 否则比较 prices[i] - minPrice（此时为非负数）与 maxProfit 的大小 3. 若 prices[i] - minPrice 的值大于 maxProfit，则把新的最大收益值赋给 maxProfit，否则不予处理

最终遍历一次，即可获得最大利润 maxProfit。

### 代码

```
1 function maxProfit(prices) {
2 let minPrice = 0;
3 let maxProfit = 0;
4
5 prices.forEach((price, index) => {
6 if (index === 0) { // 初始化最小价格为第一个元素
7 minPrice = price;
8 } else if (price < minPrice) { // 遍历过程中发现最小价格，则重新赋值
9 minPrice = price;
10 } else if (price - minPrice > maxProfit) { // 比较当日卖出收益与当前已获取的最大收益
11 maxProfit = price - minPrice;
12 }
13 });
14
15 return maxProfit;
16 }
```

### 复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$

在算法中，我们使用两个公共变量保存最大收益以及最小卖出价格，所以空间复杂度为常数级。

### 解法三 动态规划

#### 思路

有了解法二的参考，我们还可以利用差分数组连续求和来得出最大收益。第  $i$  天买入股票，第  $i + 1$  天卖出，那么我们可以获得的收益为第二天价格与第一天价格相减的差值。

如果差值为正则意味股票在上涨，如果差值为负则意味股票在下跌，我们可以将每日股票的收益转化为差分数组，求出此数组中连续子序列和的最大值，即为最大收益。

#### 详解

根据上述分析，我们用  $\text{profits}[i]$  表示第  $i$  天进行一笔交易能获得的最大收益，那么第  $i$  天会产生两种决策：

1. 第  $i$  天当天买入，此时收益为 0
2. 第  $i$  天之前买入，第  $i$  天卖出，此时可获得最大收益为第  $i - 1$  天的最大收益  $\text{profits}[i - 1]$  加上今天股票价格  $\text{prices}[i]$  与昨天价格  $\text{prices}[i - 1]$  的差值

那么第  $i$  天可以获得的最大收益为这两种情况的最大值，即： $\text{profits}[i] = \max(0, \text{profits}[i - 1] + (\text{prices}[i] - \text{prices}[i - 1]))$ ，

我们只需根据以上公式递推，即可得到每日可获取最大收益数组  $\text{profits}$ 。

我们通过一个变量  $\text{maxProfit}$  来保存已获取的最大收益，然后在计算每日最大收益的过程中与  $\text{maxProfit}$  做比较，最终计算出最大收益。

#### 代码

```
1 /**
2 * @param {number[]} prices
3 * @return {number}
4 */
5 function maxProfit(prices) {
6 let maxProfit = 0; // 最大收益
7 let profits = [0]; // 每日最大收益存入数组，第一天初始化为 0
8
9 for (i = 1; i < prices.length; i++) {
10 // 计算每日可获取的最大收益值
11 profits[i] = Math.max(0, profits[i - 1] + (prices[i] - prices[i - 1]));
12 if (profits[i] > maxProfit) { // 比较该日最大收益与已获取最大收益
13 maxProfit = profits[i];
14 }
15 }
16 return maxProfit;
17}
```

```
13 maxProfit = profits[i];
14 }
15 }
16
17 return maxProfit;
18 };
```

## 复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(n)$

# 打家劫舍、零钱兑换和跳跃游戏

## 打家劫舍

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你在不触动警报装置的情况下，能够偷窃到的最高金额。

### 示例

```
1 输入: [1,2,3,1]
2 输出: 4
3 解释: 偷窃 1 号房屋 (金额 = 1) , 然后偷窃 3 号房屋 (金额 = 3)。
4 偷窃到的最高金额 = 1 + 3 = 4 。
5
6 输入: [2,7,9,3,1]
7 输出: 12
8 解释: 偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9), 接着偷窃 5 号房屋 (金额 = 1)。
9 偷窃到的最高金额 = 2 + 9 + 1 = 12 。
```

### 方法一 用迭代的方式遍历计算

#### 思路

由于不可以相邻的房屋闯入，所以在当前位置 n 房屋可盗窃的最大值，要么就是 n-1 房屋可盗窃的最大值，要么就是 n-2 房屋可盗窃的最大值加上当前房屋的值，二者之间取最大值 动态规划方程： $dp[n] = \text{MAX}( dp[n-1], dp[n-2] + num )$

#### 详解

1. 获取房间的个数，如果为 0，就直接返回 0，
2. 如果为 1，就直接返回数组第一个值，
3. 设置三个变量 sumTemp |临时求和值，sumBefore n-2 总和，sumAfter n-1 总和
4. 初始化 3 个变量的值，
5. 循环 len-2 次求动态规划的值 `nums[i]` 为当前房间值

```

1 const rob = function (nums) {
2 const len = nums.length;
3 if (len === 0) return 0;
4 if (len === 1) {
5 return nums[0];
6 }
7 if (len === 2) {
8 return Math.max(nums[0], nums[1]);
9 }
10
11 let sumTemp = 0;
12 let sumBefore = nums[0];
13 let sumAfter = Math.max(nums[0], nums[1]);
14 let i = 2;
15 while (i < len) {
16 sumTemp = Math.max(sumAfter, sumBefore + nums[i]);
17 sumBefore = sumAfter;
18 sumAfter = sumTemp;
19 i++;
20 }
21 return sumAfter;
22 };

```

## 复杂度分析：

- 时间复杂度： $O(n)$   
只需要单循环  $n$  长度的数组  $O(n - 2)$ ，故时间复杂度为  $O(n)$
- 空间复杂度： $O(1)$

## 方法二

### 思路

由于不可以相邻的房屋闯入，所以在当前位置  $n$  房屋可盗窃的最大值，要么就是  $n-1$  房屋可盗窃的最大值，要么就是  $n-2$  房屋可盗窃的最大值加上当前房屋的值，二者之间取最大值 动态规划方程： $dp[n] = \text{MAX}(dp[n-1], dp[n-2] + num)$  总体的思路是一样的，方法一中，数组长度为 0, 1, 2 中单独处理，切换设计的求和变量过多，6 个可以利用数组变量优化。

### 详解

1. 获取房间的个数，如果为 0，就直接返回
2. 设置一个  $len+1$  的数组变量，初始化数组中的第一个和第二个对象，这边就可以不用单独处理数组长度为 1 和 2 的情况
3. 每次的循环求和的结果都记录在对于长度的数组对象中，不必声明多个变量暂存。

#### 4. 然后利用动态规划公式查找 n 个最大的数组和的值

```
1 const rob = function (nums) {
2 const len = nums.length;
3 if (len === 0) return 0;
4 const dp = new Array(len + 1);
5 dp[0] = 0;
6 dp[1] = nums[0];
7 for (let i = 2; i <= len; i++) {
8 dp[i] = Math.max(dp[i - 1], dp[i - 2] + nums[i - 1]);
9 }
10 return dp[len];
11};
```

### 复杂度分析

- 时间复杂度： $O(n)$   
只需要单循环  $n$  长度的数组  $O(n - 2)$ ，故时间复杂度为  $O(n)$
- 空间复杂度： $O(1)$

### 零钱兑换

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

#### 示例1

```
1 输入: coins = [1, 2, 5], amount = 11
2 输出: 3
3 解释: 11 = 5 + 5 + 1
```

#### 示例2

```
1 输入: coins = [2], amount = 3
2 输出: -1
```

### 解法一 动态规划法

## 思路

1. 由于需要找到最少的硬币，所以需要列举出所有可能的结果（如题）
2. 要凑够 amount 元硬币，可以从最  $i \rightarrow amount$  之前进行枚举 (counter) (下列出 最少次数)

amount = 1 ; counter[1] = 1 coins[1] = 1

amount = 2 ; counter[2] = 1 coins[2] = 2

amount = 3 ; counter[3] = 2 coins[1] + coins[2]

amount = 4 ; counter[4] = 2 coins[2] + coins[2]

amount = 5 ; counter[5] = 3 coins[2] + coins[2] + coins[1]

amount = 6 ; counter[6] = 3 coins[2] + coins[2] + coins[2]

amount = 7 ; counter[7] = 2 coins[5] + coins[2]

amount = 8 ; counter[8] = 3 coins[5] + coins[2] + coins[1]

amount = 9 ; counter[9] = 3 coins[5] + coins[2] + coins[2]

amount = 10 ; counter[10] = 2 coins[5] + coins[5]

amount = 11 ; counter[11] = 1 coins[5] + coins[5] + coins[1]

3. 递增  $i$  并且记录下能组合成  $i$  的硬币数量  $counter[i]$
4. 遍历 coins 数组，从 counter 中找到对应剩余金额 ( $i - coins[j]$ ) 的最小次数 + 1

## 详解

1. 需要计算出每一步的最佳次数，因此可以将每一步的次数保存在 counter 数组中
2. counter 中每一项的最大值应该是  $amount / 1$  次，此处默认填充  $(amount / 1) + 1$  次
3. 遍历 amount 数组 依次递增，在 coins 数组中找到满足  $i$  的最少硬币数，保存在 counter 中
4. 若  $counter[i]$  已经保存有最少的值，比较此次计算的最小次数，取两者较小
5. 重复 3, 4 步骤 直到 amount 遍历结束

```
1 const coinChange = (coins, amount) => {
```

```

2 const counter = Array(amount + 1);
3 counter.fill(amount + 1);
4 counter[0] = 0;
5 for (let i = 1; i <= amount; i++) {
6 for (let j = 0; j < coins.length; j++) {
7 if (i - coins[j] >= 0) {
8 // i - coins[j] 能凑成 i 的上一步的 最小硬币数量
9 counter[i] = Math.min(counter[i], counter[i - coins[j]] + 1);
10 }
11 }
12 }
13 // 最坏结果应该是 counter[amount] = amount
14 return counter[amount] > amount ? -1 : counter[amount];
15 }

```

## 复杂度分析

- 时间复杂度： $O(Sn)$

时间复杂度是  $O(Sn)$ ， $S$  是 `amount` 大小，需要迭代  $Sn$  次

- 空间复杂度： $O(S)$

每次迭代时没有增加新的资源， $O(1)$

## 解法二 递归

### 思路

由于要获取最小奖金币次数，实质上就是能拿到的满足 `amount` 的面值尽可能大的银币的个数。例如：

`coins = [11, 12, 5, 3]` `amount = 121;`

则次数刚好是  $121 / 11 = 11$ ，其他的取法均大于 11 次。

若 `amount = 124` 则有次数  $(121 / 11) + (3 / 3) = 12$  次

另外，在考虑最多次数时，应当满足有  $amount / 1 = amount$  次。

### 详解

1. `amount` 是总金额，要用最少的硬币数，应当是从最大的硬币开始拿起
2. 大的硬币可以多次拿取，就有  $amount \% coins[i] == 0$  成立
3. 可以保存一个最少硬币数 `mincounter` 的状态，按 `coins` 数组最大的开始，依次向最小的硬币循环

4. 按照可以使用的最大硬币次数作为循环起始条件，依次减1直至为0，递归调用
5. 当剩余 amount / coins[n] > 当前次数 + mincounter 则直接退出循环
6. 当amount为0时，即可得到最优结果

```

1 const coinChange = (coins, amount) => {
2 // 假设 最少次数 最多不会超过 amount 次
3 let minCount = amount + 1;
4 let coinsTemp = coins.sort((a, b) => b - a);
5 // 防止有超过 amount 面值的硬币出现
6 const maxValueIndex = coinsTemp.findIndex(v => v <= amount);
7 // 已经计算的次数，剩余的金额，coins，当前硬币位置
8 const calculateCountes = (count, amount, coins, index) => {
9 if (amount === 0) {
10 if (count < minCount) {
11 // 每次递归的所有可能结果进行保存
12 minCount = count;
13 }
14 return;
15 }
16 let maxCountatIndex = parseInt((amount / coins[index]), 10);
17 // 执行到超出数组边界 或者 预计最小次数大于已有 minCount 时 直接退出递归
18 if((index === coins.length) || maxCountatIndex + count >= minCount) {
19 return;
20 }
21 // amount 最少是 amount / coins[index] 次 coins[index] 的 和
22 for (let j = maxCountatIndex; j >= 0 ; j --) {
23 // 累计次数，剩余amount，银币数组，到达的coins数组下标
24 calculateCountes(count + j, amount - (coins[index] * j), coins, index + 1)
25 }
26 }
27 calculateCountes(0, amount, coinsTemp, maxValueIndex);
28 return minCount === amount + 1 ? -1 : minCount;
29 }
```

## 复杂度分析

- 时间复杂度： $O(Sn)$
- 空间复杂度： $O(n)$   
 $n$  为递归调用的最大深度，即需要  $O(n)$  空间的递归堆栈。

## 跳跃游戏

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

### 示例1:

输入: [2,3,1,1,4] 输出: true 解释: 我们可以先跳 1 步 , 从位置 0 到达位置 1, 然后再从位置 1 跳 3 步到达最后一个位置。

### 示例2:

输入: [3,2,1,0,4] 输出: false 解释: 无论怎样 , 你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0 , 所以你永远不可能到达最后一个位置。

## 方法一 贪心算法

### 思路

贪心算法的思路就是每到一个位置 , 都跳跃到当前位置可以跳跃的最大距离。当最后跳跃的最远距离等于或大于最后一个位置的时候 , 我们就认为可以到达最后一个位置 , 返回true

### 详解

1. 首先我们初始化最远位置为0 , 然后遍历数组 ;
2. 如果当前位置能到达 , 并且当前位置+跳数>最远位置 , 就更新最远位置 ;
3. 每次都去比较当前最远位置和当前数组下标 , 如果最远距离小于等于当前下标就返回false。

```
1 const canJump = function (nums) {
2 let max = 0; // 跳到最远的距离
3 for (let i = 0; i < nums.length - 1; i += 1) {
4 // 找到能跳的最远的的距离
5 if (i + nums[i] > max) {
6 max = i + nums[i];
7 }
8 // 如果跳的最远的小于当前脚标 , 返回false
9 if (max <= i) {
10 return false;
11 }
12 }
13 return true;
14 };
```

### 复杂度分析

- 时间复杂度： $O(n)$   
只需要访问 `nums` 数组一遍，共  $n$  个位置， $n$  是 `nums` 数组的长度。
- 空间复杂度： $O(1)$   
在 `max` 变量分配内存情况下，内存不会随着遍历有增长趋势，不需要额外的空间开销。

## 方法二 动态规划

### 思路

我们遍历数组，每到一个点  $i$ ，我们就去判断是否可以到达当前点；如果可以，就记录 `true`，否则为 `false`，最后判断是否可以到达(`nums.length - 1`)；

### 详解

1. 遍历数组 `nums`，每到一个点  $i$ ，我们就判断时刻可以到达当前点；
2. 如果  $i$  之前某点  $j$  本身是可以到达的，并且与当前点可达，表示点  $i$  是可达的；
3. 我们遍完成后，直接判断(`nums.length - 1`)是否可以达到。

```

1 const canJump = function (nums) {
2 // 定义一个数组，用来记录nums的点是否是可以达到的
3 const list = [nums.length];
4 // 遍历nums
5 for (let i = 1; i < nums.length; i++) {
6 // 遍历list
7 for (let j = 0; j < i; j++) {
8 // 如果j点是可以到达的，并且j点是可以达到i点的
9 // 则表示i点也是可以达到的
10 if (list[j] && nums[j] + j >= i) {
11 list[i] = true;
12 // 如果i点可以达到，则跳出当前循环
13 break;
14 }
15 }
16 }
17 return list[nums.length - 1];
18 };

```

### 复杂度分析

- 时间复杂度： $O(n^2)$  对于每个元素，通过两次遍历数组的其余部分来寻找它所对应的目标元素，这将耗费  $O(n^2)$  的时间
- 空间复杂度： $O(n)$  对于每次循环都需要给  $j$  重新分配空间，所以空间复杂度  $O(n)$

## 方法三 回溯

### 思路

我们模拟从第一个位置跳到最后位置的所有方案。从第一个位置开始，模拟所有可以跳到的位置，然后判断当前点是否可以到达(nums.length - 1);当没有办法继续跳的时候，就回溯。

### 详解

1. 我们每次传入一个下标 p，并且判断 p 是否可以达到最后的下标；
2. 如果传入的 p 等于(nums.length - 1),则表示可以到达，如果不行，则继续循环判断；
3. 如果存在 p 等于 (nums.length - 1) , 则返回true , 不存在则返回false。

```
1 const canJump = function (nums) {
2 return checkJumpPosition(0, nums); ;
3 };
4
5 function checkJumpPosition (p, nums = []) {
6 // 定义p点可以到达的最远距离
7 let jump = p + nums[p];
8 // 如果p点可以到达nums.length - 1，则返回true
9 if (p === nums.length - 1) {
10 return true;
11 }
12 // 如果最远距离大于(nums.length - 1),我们就将(nums.length - 1),设为最远距离
13 if (p + nums[p] > nums.length - 1) {
14 jump = nums.length - 1;
15 }
16 // 我们从p + 1开始到最远距离中间，找到(nums.length - 1)
17 // 如果可以，则返回true，找不到则返回false
18 for (let i = p + 1; i <= jump; i += 1) {
19 if (checkJumpPosition(i, nums)) {
20 return true;
21 }
22 }
23 return false;
24 }
```

### 复杂度分析

- 时间复杂度： $O(2^n)$  因为从第一个位置到最后一个位置的跳跃方式最多有  $2^n$  种，所以最多的耗时是  $O(2^n)$
- 空间复杂度： $O(n)$  对于每次循环都需要给  $i$  重新分配空间，最大的长度是  $\text{nums.length}$ ，所以空间复杂度  $O(n)$

# 不同路径、Longest Increasing Subsequence和单词拆分

## 不同路径

一个机器人位于一个  $m \times n$  网格的左上角，机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角，问总共有多少条不同的路径？

### 示例 1

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

### 示例 2

```
1 输入: m = 7, n = 3
2 输出: 28
```

## 思路

```
1 A B C D
2 E F G H
```

从点  $(x = 0, y = 0)$  出发，每次只能向下或者向右移动一步，因此下一点的坐标为  $(x + 1, y)$  或者  $(x, y + 1)$ ，一直到  $(x = m, y = n)$ 。在上图中，H 只能从 G 或者 D 达到，因此从 A 到 H 的路径数等于从 A 到 D 的路径与从 A 到 G 的路径之和。得出路径数量

$$T(m, n) = T(m - 1, n) + T(m, n - 1)。$$

我们又发现，当 $m = 1$  或  $n = 1$  时（只能一直往下或往右走），路径数量为1，这里得出跳出递归的条件。

## 方法一 递归

### 详解

由上面的分析可得，到达  $(m, n)$  的路径数量为  $(m, n - 1)$  坐标的路径数量与  $(m - 1, n)$  坐标的路径数量之和。可以使用最简单粗暴的递归方法

### 代码

```
1 /**
2 * @param {number} m
3 * @param {number} n
4 * @return {number}
5 */
6 const uniquePaths = function (m, n) {
7 if (m === 1 || n === 1) {
8 return 1;
9 }
10 return uniquePaths(m - 1, n) + uniquePaths(m, n - 1);
11};
```

## 方法二 动态规划

### 详解

根据以上思路，可以推出状态转移方程为

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

|   |   |   |    |    |    |    |
|---|---|---|----|----|----|----|
| 1 | 1 | 1 | 1  | 1  | 1  | 1  |
| 1 | 2 | 3 | 4  | 5  | 6  | 7  |
| 1 | 3 | 6 | 10 | 15 | 21 | 28 |

### 代码

```

1 /**
2 * @param {number} m
3 * @param {number} n
4 * @return {number}
5 */
6 const uniquePaths = function (m, n) {
7 const dp = new Array(m);
8 for (let i = 0; i < m; i++) {
9 dp[i] = new Array(n);
10 }
11 for (let i = 0; i < m; i++) {
12 for (let j = 0; j < n; j++) {
13 if (i === 0 || j === 0) {
14 dp[i][j] = 1;
15 } else {
16 dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
17 }
18 }
19 }
20 return dp[m - 1][n - 1];
21 };

```

## 复杂度分析

- 时间复杂度： $O(m * n)$

上述解法中，对  $m$  和  $n$  进行了双重循环，时间复杂度跟数字的个数线性相关，即为  $O(m * n)$

- 空间复杂度： $O(m * n)$

申请了大小为  $m * n$  的二维数组

## 方法三 动态规划优化

减少空间复杂度

### 详解

我们观察表格发现，下一个值等于当前值加上一行的值，利用这个发现，可以来压缩空间，用一维数组来实现

```

1 const uniquePaths = function (m, n) {
2 const dp = new Array(n).fill(1);

```

```

3 for (let i = 1; i < m; i++) {
4 for (let j = 1; j < n; j++) {
5 dp[j] = dp[j - 1] + dp[j];
6 }
7 }
8 return dp[n - 1];
9 };

```

## 复杂度分析

- 时间复杂度： $O(m * n)$
- 空间复杂度： $O(n)$

## 方法四 排列组合

### 详解

其实这是个高中数学问题。因为机器人只能向右或者向下移动，那么不论有多少中路径，向右和向下走的步数都是一样的。当  $m = 3, n = 2$  时，机器人向下走了一步，向右走了两步即可到达终点。所以我们可以得到

路径 = 从右边开始走的路径总数 + 从下边开始走的路径总数，转化为排列组合问题

不包括起点和终点，共移动  $N = m + n - 2$ ，向右移动  $K = m - 1$ ，将  $N$  和  $K$  代入上述公式，可得

因此得出答案  $C_{m+n-2}^{m-1}$

```

1 /**
2 * @param {number} m
3 * @param {number} n
4 * @return {number}
5 */
6 const uniquePaths = function (m, n) {
7 const N = m + n - 2;
8 const K = n - 1;
9 let num = 1;
10 for (let i = 1; i <= K; i++) {
11 num = num * (N - K + i) / i;
12 }
13 return num;
14};

```

## 复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$

## Longest Increasing Subsequence □ □

给定一个无序的整数数组，找到其中最长上升子序列的长度。

### 示例

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

方法一 动态规划

思路

- 状态定义： $\text{res}[i]$ 表示以 $\text{nums}[i]$ 为当前最长递增子序列尾元素的长度
- 转移方程：通过方程 $\text{res}[i] = \max(\text{res}[i], \text{nums}[i] > \text{nums}[j] ? \text{res}[j] + 1 : 1)$ ，动态计算出各上升子序列的长度。
- 倒序取值： res 数组进行倒序，第一个即为最大长度的值。

详解

- 如果给定数组长度小于等于 1，则最长上升子序列的长度等于数组长度。
- 初始化一个长度等于给定数组的长度，且元素都为 1 的数组 res 。
- 当 $\text{nums}[i] > \text{nums}[j]$ 时， $\text{nums}[i]$ 可以作为前一个最长的递增子序列 $\text{res}[j]$ 新的尾元素，而组成新的相对于 $\text{res}[i]$ 能够拼接的更长的递增子序列 $\text{res}[i] = \text{res}[j] + 1$ ，因为新的 $\text{res}[i]$ 能够拼接的最大长度取决于 $\text{nums}[i]$ 这个新的尾元素，而这个 $\text{nums}[i]$ 不一定大于 $\text{nums}[j]$ ，所以也不一定大于 $\text{res}[j]$ ，那么在 $i \sim j$ 之间，最大的递增子序列为 $\max(\text{res}[i], \text{res}[j]+1)$ ；当 $\text{nums}[i] \leq \text{nums}[j]$ ，长度为元素本身，即为 1。所以得出方程 $\text{res}[i] = \max(\text{res}[i], \text{nums}[i] > \text{nums}[j] ? \text{res}[j] + 1 : 1)$ ，通过转移方程收集各上升子序列的长度。
- 通过 `sort` 函数对 res 倒序排列，第一元素值 $\text{res}[0]$ 就是最长的上升子序列长度。

```

1  /**
2  * @param {number[]} nums
3  * @return {number}
4  */
5 const lengthOfLIS = function (nums) {
6  const len = nums.length;
7  if (len <= 1) {
8    return len;
9  }
10 // 初始化默认全为1
11 const res = new Array(len).fill(1);
12 for (let i = 1; i < len; i++) {
13   for (let j = 0; j < i; j++) {
14     // 转移方程
15     res[i] = Math.max(res[i], nums[i] > nums[j] ? res[j] + 1 : 1);
16   }
17 }
18 // 倒序
19 res.sort((a, b) => b - a);
20 return res[0];
21 };

```

复杂度分析

- 时间复杂度： $O(n^2)$
- 空间复杂度： $O(n)$

方法二 二分查找

思路

- 当前遍历元素大于前一个递增子序列的尾元素时 ($\text{nums}[i] > \text{tail}[\text{end}]$)，将当前元素追加到 tail 后面，这里解法其实和方法一中 $\text{nums}[i] > \text{nums}[j]$ 的解法一样。
- 当 $\text{nums}[i] \leq \text{tail}[\text{end}]$ 时，寻找前一个递增子序列第一个大于当前值的元素，替换为当前值，查找用二分，最后左边的元素即为查找到的需要被替换的结果元素。

详解

1、如果给定数组长度小于等于 1，则最长上升子序列的长度等于数组长度。2、初始化一个长度等于给定数组的长度，且第一个元素值等于给定数组的第一个元素值的数组 tail ， tail 用来存储最长递增子序列的元素。3、循环给定的数组，当前遍历元素大于前一个递增子序列的尾元素时 ($\text{nums}[i] > \text{tail}[\text{end}]$)，将当前元素追加到 tail 后面，这里解法其实和方法一中 $\text{nums}[i] > \text{nums}[j]$ 的解法一样；当 $\text{nums}[i] \leq \text{tail}[\text{end}]$ 时，寻找前一个递增子序列第一个大于当前值的元素，替换为当前值，

查找用二分，最后左边的元素即为查找到的需要被替换的结果元素。 4、循环完之后， $\text{end} + 1$ 即为最长的上升子序列长度。

```
1  /**
2   * @param {number[]} nums
3   * @return {number}
4  */
5  const lengthOfLIS = function (nums) {
6    const len = nums.length;
7    if (len <= 1) {
8      return len;
9    }
10   const tail = new Array(len);
11   tail[0] = nums[0];
12   let end = 0;
13   for (let i = 1; i < len; i++) {
14     if (nums[i] > tail[end]) {
15       end += 1;
16       tail[end] = nums[i];
17     } else {
18       let left = 0;
19       let right = end;
20       // 二分查找
21       while (left < right) {
22         // 位运算，右移一位
23         const mid = left + ((right - left) >> 1);
24         if (tail[mid] < nums[i]) {
25           left = mid + 1;
26         } else {
27           right = mid;
28         }
29       }
30       tail[left] = nums[i];
31     }
32   }
33   return end + 1;
34 };
```

复杂度分析

- 时间复杂度： $O(n \log N)$

外层一个数组循环遍历，里面嵌套一个二分查找，所以是 $O(n \log N)$

- 空间复杂度： $O(n)$

创建的数组 tail 占用空间大小为 n ，循环遍历中并没有分配新的空间

单词拆分

示例

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

说明：

- 拆分时可以重复使用字典中的单词。
- 你可以假设字典中没有重复的单词。
- 注意你可以重复使用字典中的单词。

示例 1：

```
1 输入: s = "leetcode", wordDict = ["leet", "code"]
2 输出: true
3 解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。
```

示例 2：

```
1 输入: s = "applepenapple", wordDict = ["apple", "pen"]
2 输出: true
3 解释: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
```

示例 3：

```
1 输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
2 输出: false
```

方法一 暴力破解法

思路

把字符串 s 的前缀从短到长拆出来进行判断是否在单词字典中，若在字典中则把前缀截取掉继续递归，直到字符串的长度为 0。在递归中若遇到字符串任何长度的前缀都无法匹配到字典中的单词，则回溯到上层递归。

详解

- 1、检查字典中是否有字符串的前缀；
- 2、若有的话，将字符串去掉这个前缀后继续遍历，重复步骤 1、2；
- 3、若某次调用发现整个字符串都已拆分并且都在字典内则返回 true；

输入: `s='catsandog', wordDict=['cats', 'dog', 'sand', 'and', 'cat']`

c a t s a n d o g 从第一个字母开始遍历

c a t s a n d o g 在字典中找到匹配的单词 ‘cat’，截取前缀后继续递归

c a t s a n d o g 在字典中找到匹配的单词 ‘sand’，截取前缀后继续递归

c a t s a n d o g 在字典中找不到匹配的单词，回溯到上层递归

c a t s a n d o g 在字典中找不到匹配的单词，回溯到上层递归

c a t s a n d o g 在字典中找到匹配的单词 ‘cats’，截取前缀后继续递归

c a t s a n d o g 在字典中找到匹配的单词 ‘and’，截取前缀后继续递归

c a t s a n d o g 在字典中找不到匹配的单词，回溯到上层递归

.

.

c a t s a n d o g 在字典中找不到匹配的单词，结束 返回false

方法一

```
1 const wordBreak = function (s, wordDict) {
2   if (s.length === 0) {
3     return true;
4   }
5   for (let i = 0; i < wordDict.length; i += 1) {
6     const startIndex = s.indexOf(wordDict[i]);
7     if (startIndex === 0) {
8       // 将字符串去掉这个匹配到的前缀后继续遍历
9       const temp = s.slice(startIndex + wordDict[i].length);
10      if (wordBreak(temp, wordDict) === true) {
11        return true;
12      }
13    }
14  }
15  return false;
16};
```

- 时间复杂度：最坏情况下是 $O(n^n)$ ，因为考虑

`s = 'aaaaaaaaaaaaaaaaaa'`，`wordDict = ['a']`，每一个字符都在字典中，此时递归的时间复杂度会达到 $O(n^n)$ ，妥妥超时

- 空间复杂度： $O(1)$ ，循环中申请了 3 个临时变量，与输入的字符串的长度无关，空间占用属于常数阶，故空间复杂度为 $O(1)$ 。

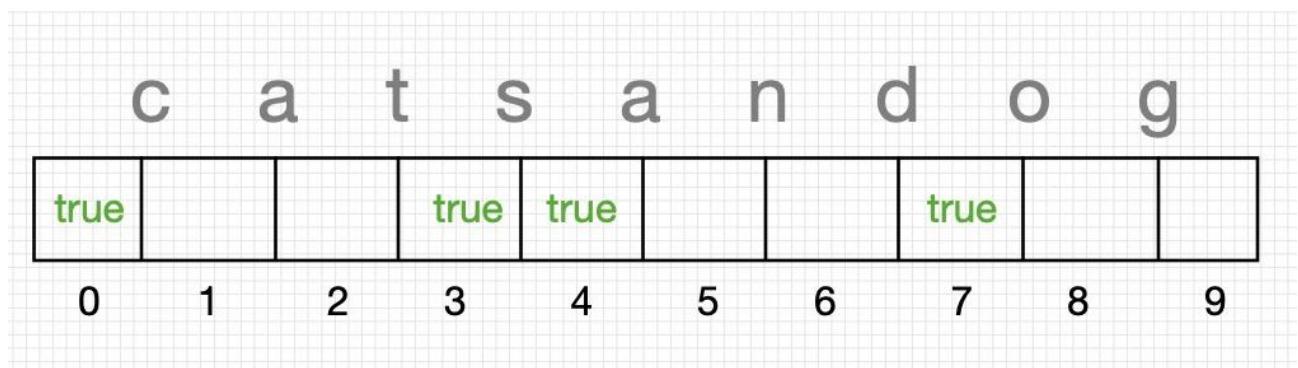
方法二 动态规划

思路

`dp[i]` 表示字符串 `s` 从开始到 `i` 位置是否可以由 `wordDict` 组成。使用 `j` 从头开始遍历，若 `dp[i]` 可由 `wordDict` 组成，并且 `i` 到 `j` 之间的单词可以在 `wordDict` 中找到，则说明 `dp[i] = true`。

详解

- 第一层遍历：用 `i` 从头到尾遍历字符串；
- 第二层遍历：用 `j` 从头到 `i` 遍历字符串；
- 若 `dp[j] = true` 而且字典中存在字符串 `s[i~j]`，则说明 `dp[i] = true`；
- 继续步骤 1、2，直到整个字符串都遍历一遍；
- 若 `dp[s.length()] = true`，则说明字符可由字段中的单词组合而成；



方法二-详解

代码

```

1 const wordBreak = function (s, wordDict) {
2   const len = s.length;
3   const dp = new Array(len + 1).fill(false);
4   dp[0] = true;
5   for (let i = 1; i <= len; i++) {
6     for (let j = 0; j < i; j++) {
7       if (dp[j] && wordDict.includes(s.substring(j, i))) {
8         dp[i] = true;
9         break;
10      }
}

```

```
11     }
12 }
13 return dp[len];
14 };
```

复杂度分析

- 时间复杂度： $O(n^2)$

因为有两层循环，每层循环都从头遍历到尾。

- 空间复杂度： $O(n)$

因为只开辟了一个 n 长度的数组。

方法三 动态规划（优化版）

思路

第二层遍历中不用每次遍历 i 的长度，只要遍历字典中最长单词的长度 \maxStep 即可。

详解

- 第一层遍历：用 i 从头到尾遍历字符串；
- 第二层遍历：用 j 从 $i - \maxStep$ 到 i 遍历字符串；
- 若 $dp[j] = \text{true}$ 而且字典中存在字符串 $s[i \sim j]$ ，则说明 $dp[i] = \text{true}$ ；
- 继续步骤 1、2，直到整个字符串都遍历一遍；
- 若 $dp[s.length()] = \text{true}$ ，则说明字符可由字段中的单词组合而成；

```
1 const wordBreak = function (s, wordDict) {
2     const len = s.length;
3     const dp = new Array(len + 1).fill(false);
4     dp[0] = true;
5     // 计算单词的最长长度
6     let maxStep = 0;
7     for (let i = 0; i < wordDict.length; i++) {
8         if (wordDict[i].length > maxStep) {
9             maxStep = wordDict[i].length;
10        }
11    }
12    for (let i = 1; i <= len; i++) {
13        const startOfJ = i - maxStep > 0 ? i - maxStep : 0;
14        for (let j = startOfJ; j < i; j++) {
15            if (dp[j] && wordDict.includes(s.substring(j, i))) {
16                dp[i] = true;
17                break;
18            }
19        }
20    }
21    return dp[len];
22 }
```

```
18         }
19     }
20 }
21 return dp[len];
22 };
```

复杂度分析

- 时间复杂度： $O(n^2)$

因为有两层循环，每层循环都从头遍历到尾。

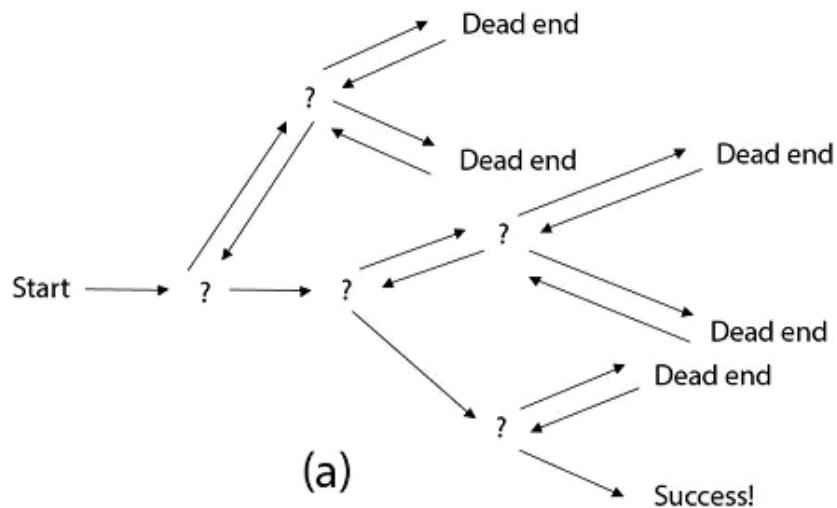
- 空间复杂度： $O(n)$

因为最长只开辟了一个 n 长度的数组。

回溯算法

回溯算法(back tracking)是一种类似尝试算法，按选优条件向前搜索，主要是在搜索尝试过程中寻找问题的解，以达到目标，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。换句话说，找到一条路往前走，能走就继续往前，不能走就算了，掉头换条路。相对于动态规划，这部分的内容相对于简单些。

回溯的处理思想，和枚举搜索有点类似，通过枚举找到所有满足期望的值。为了有规律地枚举所有的解，可以把问题拆解为多个小问题。每个小问题，我们都会面对一个岔路口，选择一条发现此路不通的时，就往回走，走到另一个岔路口。



本章节分为 2 个部分，选取了比较经典的算法题，希望可以帮助到同学们学会解决回溯相关的算法题。

- Part 1
 - 括号生成
 - 子集
 - 电话号码的字母组合
- Part 2
 - 全排列
 - 单词搜索

括号生成、子集和电话号码的字母组合

括号生成

给出 n 代表生成括号的对数，请你写出一个函数，使其能够生成所有可能的并且有效的括号组合。

示例

```
1 例如，给出 n = 3，生成结果为：  
2  
3  [  
4      "((()))",  
5      "(()())",  
6      "(()())",  
7      "()(())",  
8      "()()()",  
9  ]
```

方法一 回溯法（实现一）

思路

我们知道，有且仅有两种情况，生成的括号序列不合法

- 我们不停的加左括号，其实如果左括号超过 n 的时候，它肯定不是合法序列了。因为合法序列一定是 n 个左括号和 n 个右括号。
- 如果添加括号的过程中，如果右括号的总数量大于左括号的总数量了，后边不论再添加什么，它都不可能是合法序列了。

基于上边的两点，我们只要避免它们，就可以保证我们生成的括号一定是合法的了。

详解

1. 采用回溯法，即把问题的解空间转化成了图或者树的结构表示，然后，使用深度优先搜索策略进行遍历，遍历的过程中记录和寻找所有可行解或者最优解。
2. 如果自定义的字符串长度足够且走到这里，那么就说明这个组合是合适的，我们把它保存。
3. 否则，递归执行添加左括号，添加右括号的操作。
4. 递归结束条件为结果字符串长度等于左右括号的总个数 ($2n$)，则返回最终结果。

代码

```

1  /**
2  * @param {number} n
3  * @return {string[]}
4  */
5 const generateParenthesis = n => {
6  const res = [];
7  const gen = (str = '', l = 0, r = 0) => {
8    // 如果自定义的字符串长度足够且走到这里，那么就说明这个组合是合适的
9    if (str.length === 2 * n) {
10      res.push(str);
11      return;
12    }
13    // 下面的逻辑：左括号必须出现在右括号的前面
14    // 只有在 l >= n 的时候，才不能添加左括号，其他都可添加
15    if (l < n) {
16      gen(` ${str}(`, l + 1, r);
17    }
18    // 如果右括号没有左括号多，我们就可以添加一个右括号
19    if (r < l) {
20      gen(` ${str})`, l, r + 1);
21    }
22  };
23  gen();
24  return res;
25};

```

复杂度分析

我们的复杂度分析依赖于理解 `generateParenthesis(n)` 中有多少个元素。这个分析需要更多的背景来解释，但事实证明这是第 n 个卡塔兰数 $\frac{1}{n+1} \binom{2n}{n}$ ，这是由 $\frac{4^n}{n\sqrt{n}}$ 渐近界定的。

- 时间复杂度： $O(\frac{4^n}{\sqrt{n}})$ ，在回溯过程中，每个有效序列最多需要 n 步。
- 空间复杂度： $O(\frac{4^n}{\sqrt{n}})$ ，如上所述，并使用 $O(n)$ 的空间来存储序列。

方法二 回溯法（实现二）

思路

整体的实现思路也是回溯法，也是递归来实现该思路，唯一不同的是，递归的结束条件是左右括号都消费尽。

详解

1. 采用回溯法，即把问题的解空间转化成了图或者树的结构表示，然后，使用深度优先搜索策略进行遍历，遍历的过程中记录和寻找所有可行解或者最优解。
2. 首先，先从左括号开始填充。
3. 然后，填充右括号，保证两类括号数目一致，平衡。
4. 递归结束条件为左右括号均消费尽，则输出结果。

代码

```

1  /**
2   * @param {number} n
3   * @return {number}
4   */
5 const generateParenthesis = n => {
6   const res = [];
7   // left :左括号个数， right:右括号个数
8   function helper (left, right, max, str) {
9     if (left === max && right === max) {
10       res.push(str);
11       return;
12     }
13     // 先从左括号开始填充
14     if (left < max) {
15       helper(left + 1, right, max, `${str}(`);
16     }
17     // 保证两类括号数目一致
18     if (left > right) {
19       helper(left, right + 1, max, `${str})`);
20     }
21   }
22   helper(0, 0, n, '');
23   return res;
24 };

```

复杂度分析

本方法也是使用回溯法，只是具体实现方式不同，因此，复杂度也是一样的。

- 时间复杂度： $O(\frac{4^n}{\sqrt{n}})$

在回溯过程中，每个有效序列最多需要 n 步。

- 空间复杂度： $O(\frac{4^n}{\sqrt{n}})$

如上所述，并使用 $O(n)$ 的空间来存储序列。

子集

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

```
1 输入: nums = [1,2,3]
2 输出:
3 [
4   [3],
5   [1],
6   [2],
7   [1,2,3],
8   [1,3],
9   [2,3],
10  [1,2],
11  []
12 ]
```

方法一 回溯算法

思路

设置初始二维数组`[]`，依次把每个数加入到数组中的每一个元素中，并保留原来的所有元素。

详解

1. 初始化二维数组来存储所有子集；
2. 使用双层遍历，外层遍历 `nums` 数组，内层遍历二维数组，将每个元素与二维数组每项合并，并保留二维数组原有的元素
3. 并将得到的新子集与二维数组元素合并，最后得到所有子集；

例如：输入 `nums = [1, 2, 3]` 初始化二维数组存储所有子集 `result = []` 然后遍历 `nums`, 1 添加到`[]`, 结果为 `[[], [1]]`; 2 添加到`[[], [1]]`, 结果为 `[[], [1], [2], [1, 2]]`; 3 添加到`[[], [1], [2], [1, 2]]`, 结果为 `[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]`;

代码

```

1 const subsets = (nums) => {
2   let result = [[]]; // 初始化二维数组
3   for (let i of nums) {
4     const temp = [];
5     for (let k of result) {
6       temp.push(k.concat(i)); // 依次把每个元素添加到数组中
7     }
8     result = result.concat(temp);
9   }
10  return result;
11 }

```

复杂度分析

- 时间复杂度： $O(2^n)$

根据上述算法，每次循环 `result` 数组的次数为 $2^{(n-1)}$ ，则计算总数为

$1 + 2^1 + 2^2 + \dots + 2^{(n-1)}$ ，根据等比数列计算公式得到循环总次数为 $2^n - 1$ ，所以时间复杂度为 $O(2^n)$ 。

- 空间复杂度： $O(2^n)$

根据排列组合原理，子集包含一个数字的情况所耗费的存储空间为 $C_n^1 * 1$ ，包含两个数字所耗费的存储空间为 $C_n^2 * 2$ ，根据算法得出共需要

$C_n^0 + C_n^1 + 2 * C_n^2 + \dots + (n-1) * C_n^{n-1} + n * C_n^n$ 个存储空间，根据排列组合公式求和可得需要 $n * 2^{n-1} + 1$ 个额外存储空间，所以算法空间复杂度为 $O(2^n)$ 。

方法二 二进制表示法

思路

从数学角度看，求子集其实是一个排列组合问题，比如 `nums = [1, 2, 3]`，存在四种情况：

- 都不选，情况共有 $C(3,0) = 1$ 种
- 只选 1 个数，情况共有 $C(3,1) = 3$ 种
- 选 2 个数，情况共有 $C(3,2) = 3$ 种
- 全选，情况共有 $C(3,3) = 1$ 种

落到数组中的每个数字，都有两种可能，选与不选，我们用 1 标识选，用 0 标识不选。则 `[]` 表示为 000，`[1, 2, 3]` 表示为 111，我们通过转化为二进制表示法，遍历 000 - 111 的所有组合，即可求

出所有子集。

详解

- 根据上述分析，我们得出加入数组为 `nums`，则针对每位存在选与不选两种情况，那么所有组合数为 $2^{\text{(nums 长度)}}$ 个。
- 针对每一种组合情况，我们可以取出该二进制表示法中的每一位，如 110，我们分别通过向右移位并和 1 求与，判断最低位的值为 0 或者 1。
- 如果得到结果为 1，那么表示该位表示的数字在原数组中被选中，存入暂存数组，一轮遍历后即可获得该组子集的数字组合。将所有子集数字组合一起存入结果数组，即求出所有子集。

代码

```
1  const subsets = (nums) => {
2      // 子集的数量
3      const len = nums.length;
4      const setNums = Math.pow(2, len);
5      const result = [];
6
7      for (let i = 0; i < setNums; i++) {
8          let temp = [];
9          // 判断该二进制表示法中，每一个位是否存在
10         for (let j = 0; j < len; j++) {
11             if (1 & (i >> j)) { // 如果该位为 1，则存入组合数组
12                 temp.push(nums[j]);
13             }
14         }
15
16         result.push(temp);
17     }
18
19     return result;
20 };
```

复杂度分析

- 时间复杂度： $O(2^n)$

一共包含 2^n 个组合需要 $n * 2^n$ 次计算，所以时间复杂度为 $O(2^n)$ 。

- 空间复杂度： $O(2^n)$

电话号码的字母组合

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。

示例

```
1 输入："23"
2
3 输出：["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].
4  {
2   const map = {
3     2: ['a', 'b', 'c'],
4     3: ['d', 'e', 'f'],
5     4: ['g', 'h', 'i'],
6     5: ['j', 'k', 'l'],
7     6: ['m', 'n', 'o'],
8     7: ['p', 'q', 'r', 's'],
9     8: ['t', 'u', 'v'],
10    9: ['w', 'x', 'y', 'z']
11  };
12}
```

```

13  if (!digits) {
14      return [];
15  }
16
17  const res = [''];
18  for (let i = 0; i < digits.length; i++) {
19      const letters = map[digits[i]];
20      const size = res.length;
21      for (let j = 0; j < size; j++) {
22          const temp = res.shift(0); // 取出第一个元素
23          for (let k = 0; k < letters.length; k++) {
24              res.push(`$${temp}${letters[k]}`);
25          }
26      }
27  }
28  return res;
29 };

```

复杂度分析

- 时间复杂度： $O(3^m * 4^n)$

通过数组的 push 我们发现，循环的次数和最后数组的长度是一样的，然而数组的长度有和输入多个3个字母的数目、4个字母的数目有关，得出时间复杂度为 $O(3^m * 4^n)$
其中 m 为3个字母的数目，n 为4个字母的数目

- 空间复杂度： $O(3^m * 4^n)$

最后结果的长度和输入的数字有关，得出复杂度为 $O(3^m * 4^n)$

方法二 回溯

思路

这道题有点排列组合的味道，我们可以穷举所有的可能性，找到所有的可能性

详解

- 如果还有数字需要被输入，就继续遍历数字对应的字母进行组合 `prefix + letter`
- 当发现没有数字输入时，说明已经走完了，得到结果

代码

```

1 const map = {
2     2: ['a', 'b', 'c'],
3     3: ['d', 'e', 'f'],
4     4: ['g', 'h', 'i'],

```

```

5   5: ['j', 'k', 'l'],
6   6: ['m', 'n', 'o'],
7   7: ['p', 'q', 'r', 's'],
8   8: ['t', 'u', 'v'],
9   9: ['w', 'x', 'y', 'z']
10 };
11
12 const letterCombinations = function (digits) {
13   const result = [];
14   function backtrack (prefix, next) {
15     // 发现没有字母需要输入时，就可以返回了
16     if (next.length === 0) {
17       result.push(prefix);
18     } else {
19       const digit = next[0];
20       const letters = map[digit]; // 获取对应的各个字母
21       for (let i = 0; i < letters.length; i++) {
22         backtrack(prefix + letters[i], next.substr(1));
23       }
24     }
25   }
26   if (digits.length !== 0) {
27     backtrack('', digits);
28   }
29   return result;
30 };

```

复杂度分析

- 时间复杂度： $O(3^m * 4^n)$
- 空间复杂度： $O(3^m * 4^n)$

实现数组的全排列和单词搜索

实现数组的全排列

给定一个没有重复数字的序列，返回其所有可能的全排列。

示例

```
1 给定 nums = [1,2,3];
2
3 返回
4 [
5   [1,2,3],
6   [1,3,2],
7   [2,1,3],
8   [2,3,1],
9   [3,1,2],
10  [3,2,1]
11 ]
```

方法一 回溯法

思路

回溯法通常是构造一颗生成树，生成树排列法的核心思想为：遍历需要全排列的数组，不断选择元素，将不同的数字生成一颗树，如果数组中待选择的结点数为 0，则完成一种全排列。

详解

从上面的思路中，我们可以抽象出全排列函数的步骤：

- 1 1. 遍历需要全排列的数组，取出不同位置的数字，创建以对应位置数字为根节点的树。
- 2 2. 遍历剩下的数组，选出一个数字，将该数字挂在生成的树上。
- 3 3. 重复第二步操作直到剩余数字数组的长度为0，表明完成了全排列，将生成的树存入排序结果数组中。

举个例子：输入数组为 [1, 2, 3]，首先选择 1 为根节点，剩余数组为 [2, 3]，继续选择 2 作为下一个结点，剩余数组为 [3]，那么选择 3 为最后一个结点，那么 [1, 2, 3] 组成一种全排列情况。我们回

溯到第二步剩余数组，选择 3 为第二个结点，那么剩余数组为 [2]，选择后完成 [1, 3, 2] 这种全排列情况。后续依次固定 2, 3 为根结点，列出所有可能。

从上述内容中可以看出，生成初始树后就是一个 截取数组 -> 设置节点，继续 截取节点 -> 设置节点 的递归过程了。那么我们是否可以将第一步的操作合并到我们的递归函数中呢？

既然我们递归的操作是截取数字，并将对应的数字与目标树结合。那么我们可以将第一步看为数字和空树结合。

那么我们递归函数的参数就确定为：

- 剩余的数组
- 现有的顺序树

递归函数的逻辑也可以收敛为：

1. 遍历需要全排列的数组，将不同位置的数字与目前树结合起来
2. 重复该操作直到需要全排列的数组长度为 0，即表明完成了全排列。

代码

```
1 const permute = function (numbs) {
2   const allSortResult = [] // 设置结果数组
3   function recursion (restArr, tempResult) {
4     for (let index = 0; index < restArr.length; index++) {
5       // 获取不同位置的数字
6       const insertNumb = restArr[index];
7
8       // 将不同位置的数字与现有的顺序树相结合
9       const nextTempResult = [...tempResult, insertNumb];
10
11      /**
12       * 判断传入的数组长度
13       * 大于1:
14       * 继续递归，参数分别为：
15       * 1. 除去当前数字外的数组
16       * 2. 新生成的树
17       * 等于1(不会出现小于1的情况):
18       * 表明已经结合到了最后一个节点，将生成的顺序树推送到结果数组中
19       */
20      if (restArr.length > 1) {
21        recursion(
22          restArr.filter(resetNumb => resetNumb !== insertNumb),
23          nextTempResult
24        );
25      } else {
```

```
26         allSortResult.push(nextTempResult);
27     }
28 }
29 }
30
31 // 调用递归方法，开始生成顺序树
32 recursion(nums, []);
33
34 // 返回全排列结果
35 return allSortResult;
36 };
```

复杂度分析

- 时间复杂度： $O(n^2)$
- 空间复杂度： $O(n^2)$

方法二 插值排序法

思路

插值排列法的核心思想为：遍历需要全排列的数组，将不同位置的数字抽离出来，插入到剩余数组的不同位置，即可得到该数字与另一个数组的全排列结果。将一个固定的数字，插入到另一个数组的全排列结果的不同位置，遍历需要全排列的数组，将不同的数字连接到不同的树上 继续全排列剩下的数组与生成的树，当剩余数组长度为0时，表明完成了全排列。

详解

从上面的思路中，我们可以抽象出插值排序全排列方法的具体实现：

首先处理特殊情况。如果传入排列的数组长度为1，直接返回该数组。否则进行下面的全排列方法

1. 截取传入数组的第一位数字，将剩余数组传入获取全排列方法函数，获取剩余数字的全排列结果
2. 遍历剩余数字的全排列结果数组并取出各个结果
3. 使用循环将截取的数字插入到不同结果(数组)的不同位置，生成新的全排列结果并保存
4. 返回全排列结果数组

代码

```
1 const permute = function (nums) {
2     // 设定保存结果变量
3     const result = [];
4
5     // 如果长度为1，只有一种排列，直接返回
6     if (nums.length === 1) {
7         return [nums];
8     } else {
9         // 长度大于1，获取除第一个数字外的数组的全排列结果
10        const allSortList = permute(nums.slice(1));
11
12        // 遍历剩余数字的全排列结果数组
13        for (let sortIndex = 0; sortIndex < allSortList.length; sortIndex++) {
14            // 取出不同的全排列结果
15            const currentSort = allSortList[sortIndex];
16
17            // 将第一个数字插入到不同的位置来生成新的结果，并保存
18            for (let index = 0; index <= currentSort.length; index++) {
19                const tempSort = [...currentSort];
20                tempSort.splice(index, 0, nums[0]);
21                result.push(tempSort);
22            }
23        }
24
25        // 返回全排列结果数组
26        return result;
27    }
28};
```

复杂度分析：

- 时间复杂度： $O(n^3)$ 时间复杂度由全排列函数 `permute` 提供。单个全排列函数中存在两层循环嵌套，因此单次调用的时间复杂度为 $O(n^2)$ 。当 `nums` 长度为 n 到 2 时调用全排列函数，即调用 $(n - 2)$ 次 $n^2(n - 2) = n^3 - 2n^2$ ；当 n 趋近于无限大时， n 可以忽略，因此时间复杂度为 $O(n^3)$
- 空间复杂度： $\$O(n^3)\$$ ，在循环体内使用4个变量，空间复杂度为 $\$O(4*n^3)\$$ ，当 n 趋近于无限大，忽略系数，即为 $\$O(n^3)\$$

单词搜索

给定一个二维网格和一个单词，找出该单词是否存在子网格中。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

示例

```
1 board =
2 [
3     ['A','B','C','E'],
4     ['S','F','C','S'],
5     ['A','D','E','E']
6 ]
7
8 给定 word = "ABCED"，返回 `true`。
9 给定 word = "SEE"，返回 `true`。
10 给定 word = "ABCB"，返回 `false`.
```

方法 回溯算法

思路

题目给的要求水平相邻或垂直相邻的单元格，这与回溯的思想非常相似，简单来说，上下左右都去走一遍，发现不符合要求立即返回。

以题目的示例为例子，从 A 开始，放四个方向试探，如果不匹配，立即换方向

详解

1. 从起点开始，枚举所有的可能性，递归搜索
2. 如果当前字符串匹配，再考虑上下左右 4 个方向，当发现超出边界或者不匹配时，立即结束当前方向的搜索

```
1 /**
2  * @param {character[][]} board
3  * @param {string} word
4  * @return {boolean}
5 */
6 const exist = function (board, word) {
7     const m = board.length;
8     const n = board[0].length;
9
10    for (let i = 0; i < m; i++) {
11        for (let j = 0; j < n; j++) {
12            if (wordSearch(i, j, 0)) {
```

```

13         return true;
14     }
15 }
16 }
17
18 function wordSearch (i, j, k) {
19     // 超出边界或者不匹配时，返回 false
20     if (i < 0 || j < 0 || i >= m || j >= n || word[k] !== board[i][j]) {
21         return false;
22     }
23
24     // 找到最后一个字符，返回 true，为递归的终止条件
25     if (k === word.length - 1) {
26         return true;
27     }
28
29     // 先占位
30     const temp = board[i][j];
31     board[i][j] = '-';
32     // 往四个方向递归搜索，有一个方向匹配就可以了
33     const res = wordSearch(i + 1, j, k + 1) ||
34     wordSearch(i - 1, j, k + 1) ||
35     wordSearch(i, j + 1, k + 1) ||
36     wordSearch(i, j - 1, k + 1);
37
38     // 四个方向搜索完了释放
39     board[i][j] = temp;
40
41     return res;
42 }
43
44 return false;
45 };

```

复杂度分析

- 时间复杂度： $O((m * n)^2)$

m 和 n 分别是矩阵的行数和列数。每个递归函数 `wordSearch` 的调用次数为 $m * n$ ，并且调用了4个递归函数，复杂度为 $O(4 * m * n)$ 在 for 循环下，时间复杂度为 $O(m * n)$ 。因此总的时间复杂度为 $O(4(m * n)^2) = O((m * n)^2)$

- 空间复杂度： $O(m * n)$

每个递归函数的递归深度为 $m * n$ ，空间复杂度为 $O(m * n)$ ，一共调用了4个，因此总的复杂度为 $O(4 * m * n) = O(m * n)$

排序与搜索

排序算法(sorting algorithm)是一种能将一串数据依照特定顺序进行排列的一种算法。

排序算法的一个指标是稳定性，稳定性即：如果只按照第一个数字排序的话，第一个数字相同而第二个数字不同的，第二个数字按照原有排序的就是稳定排序，不按照原有排序的就是不稳定排序。

算法复杂度

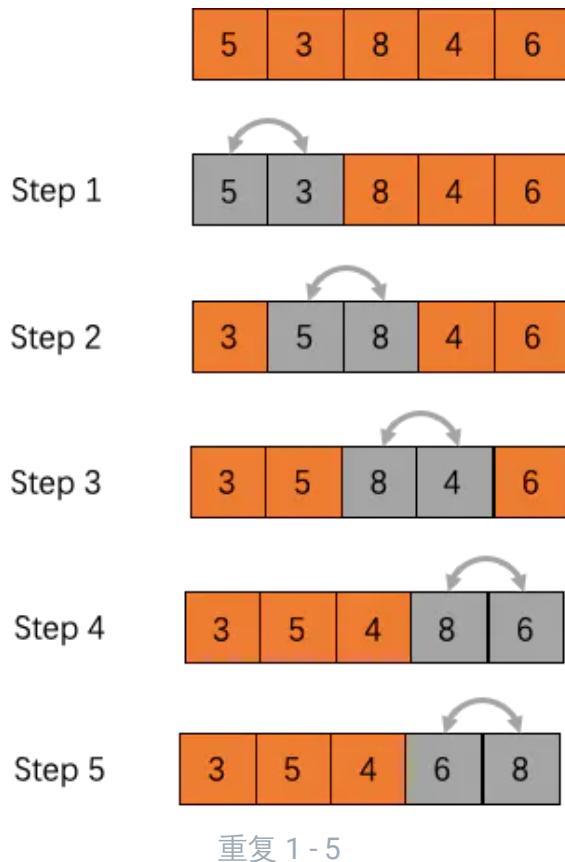
排序方法	时间复杂度(平均)	时间复杂度(最坏)	时间复杂度(最好)	空间复杂度	稳定性
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
快速排序	$O(n \log_2 n)$	$O(n^2)$	$O(n \log_2 n)$	$O(n \log_2 n)$	不稳定
归并排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n)$	稳定

冒泡排序(Bubble Sort)

我们先来了解一下冒泡排序算法，虽然比较容易实现，但是比较慢。之所以称之为冒泡排序是因为使用这种排序算法时，像气泡一样从数组的一端冒到另一端。

实现原理

- 每次比较，相邻的元素，如果第一个比第二个大，就交换两个元素的位置
- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该是最大的数；



代码

```

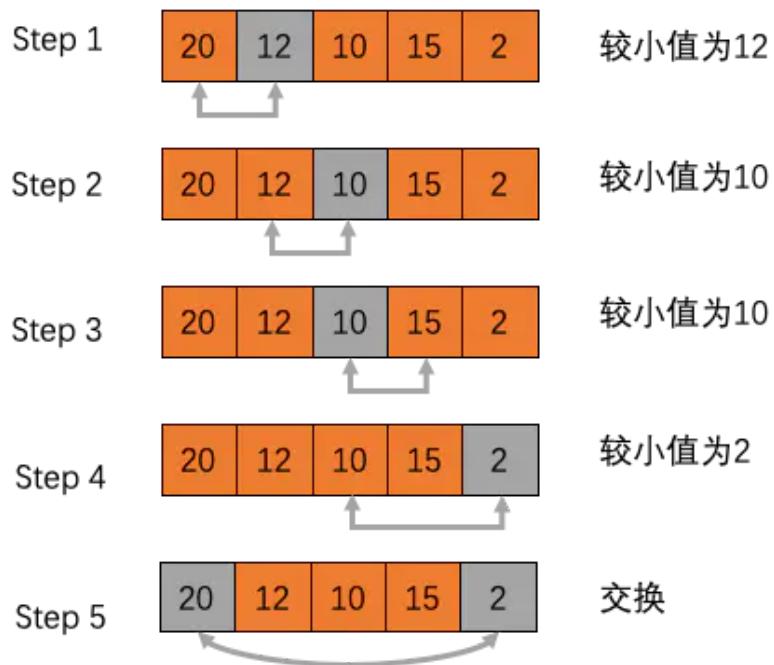
1  function bubbleSort(arr) {
2    const len = arr.length;
3    for (let i = 0; i < len - 1; i++) {
4      for (let j = 0; j < len - 1 - i; j++) {
5        if (arr[j] > arr[j+1]) {
6          const temp = arr[j+1];
7          arr[j+1] = arr[j];
8          arr[j] = temp;
9        }
10      }
11    }
12    return arr;
13  }

```

选择排序(Selection Sort)

选择排序是一种简单直观的排序算法。选择排序从数组的开头开始，将第一个元素和其他元素进行比较，检查完所有元素后最小的元素会被放到数组的第一个位置，然后从第二个元素开始继续。这

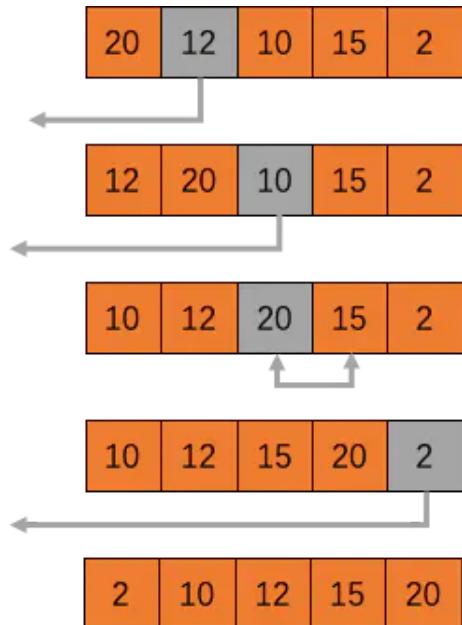
个过程一直进行到数组的倒数第二个位置。



```
1 function selectionSort(arr) {
2     const len = arr.length;
3     let minIndex;
4     let temp;
5     for (let i = 0; i < len - 1; i++) {
6         minIndex = i;
7         for (let j = i + 1; j < len; j++) {
8             if (arr[j] < arr[minIndex]) {
9                 minIndex = j; // 保存最小数的索引
10            }
11        }
12        temp = arr[i];
13        arr[i] = arr[minIndex];
14        arr[minIndex] = temp;
15    }
16    return arr;
17 }
```

插入排序(Insertion Sort)

插入排序类似于按首字母或者数字对数据进行排序。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。



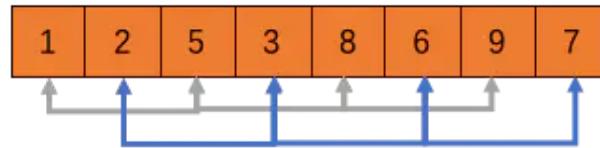
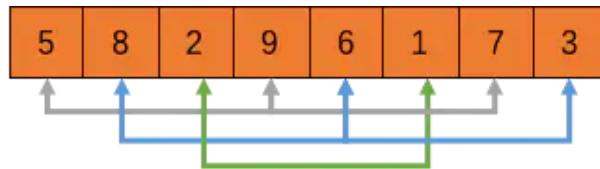
```

1  function insertionSort(arr) {
2    const len = arr.length;
3    let preIndex;
4    let current;
5    for (let i = 1; i < len; i++) {
6      preIndex = i - 1;
7      current = arr[i];
8      // 大于新元素，将该元素移到下一位置
9      while (preIndex >= 0 && arr[preIndex] > current) {
10        arr[preIndex + 1] = arr[preIndex];
11        preIndex--;
12      }
13      arr[preIndex + 1] = current;
14    }
15    return arr;
16  }

```

希尔排序(Shell Sort)

希尔排序之所以叫希尔排序，因为它就希老爷子(Donald Shell)创造的。希尔排序对插入做了很大的改善。核心理念与插入排序的不同之处在于，它会优先比较距离较远的元素，而不是相邻的元素。当开始用这个算法遍历数据集时，所有元素之间的距离会不断减少，直到处理到数据的末尾。



```

1  function shellSort(arr) {
2    const len = arr.length;
3    let gap = Math.floor(len / 2);
4
5    while (gap > 0) {
6      for (let i = gap; i < len; i++) {
7        const temp = arr[i];
8
9        let j = i;
10       while (j >= gap && arr[j - gap] > temp) {
11         arr[j] = arr[j - gap];
12         j -= gap;
13       }
14       arr[j] = temp;
15     }
16     gap = Math.floor(gap / 2);
17   }
18   return arr;
19 }
```

快速排序(Quick Sort)

快速排序一般用来处理大数据集，速度比较快。快速排序通过递归的方式，将数据依次分为包含较小元素和较大元素的不同子序列。

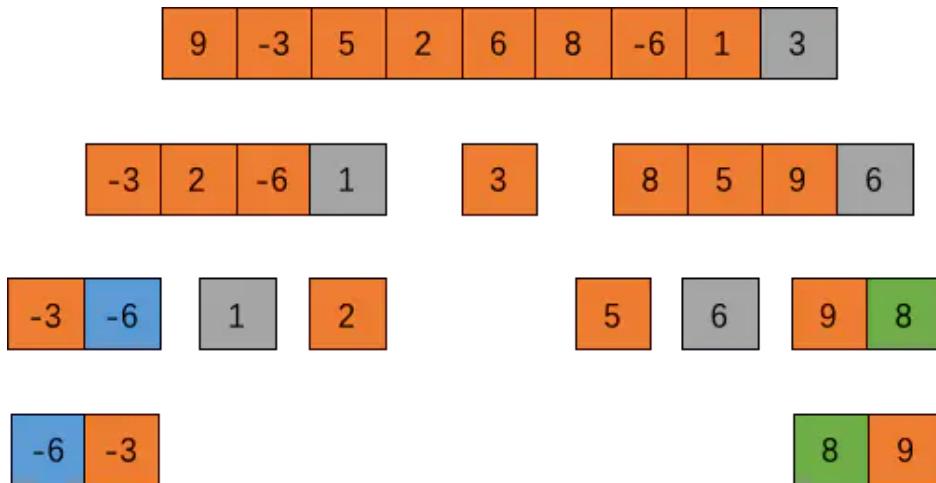
实现原理

这个算法首先要在列表中选择一个元素作为基准值，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面。这个基准值一般有 4 种取法：

- 无脑拿第一个元素

- 无脑拿最后一个元素
- 无脑拿中间的元素
- 随便拿一个

下面的解法基于取最后一个元素实现：



```

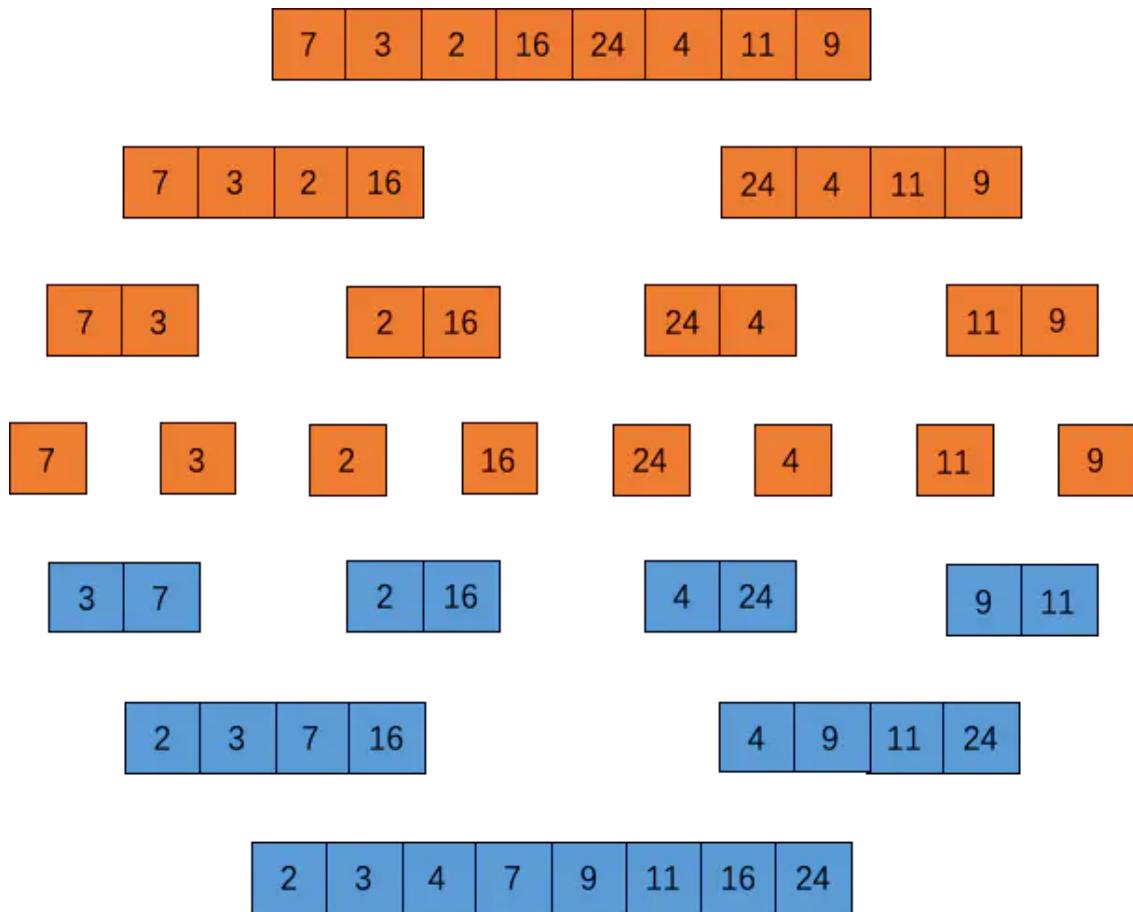
1  function partition(arr, low, high) {
2      let i = low - 1; // 较小元素的索引
3      const pivot = arr[high];
4
5      for (let j = low; j < high; j++) {
6          // 当前的值比 pivot 小
7          if (arr[j] < pivot) {
8              i++;
9              [arr[i], arr[j]] = [arr[j], arr[i]]
10         }
11     }
12     [arr[i + 1], arr[high]] = [arr[high], arr[i + 1]]
13     return i + 1;
14 }
15
16 function quickSort(arr, low, high) {
17     if (low < high) {
18         const pi = partition(arr, low, high)
19         quickSort(arr, low, pi - 1)
20         quickSort(arr, pi + 1, high)
21     }
22     return arr;
23 }
```

归并排序(Merge Sort)

归并排序是把一系列排好序的子序列合并成一个大的完整有序序列。

实现原理

把长度为 n 的输入序列分成两个长度为 $n / 2$ 的子序列，对这两个子序列分别采用归并排序，最后将两个排序好的子序列合并成一个最终的排序序列。



代码

```
1 function mergeSort(arr) {  
2     const len = arr.length;  
3     if (arr.length > 1) {  
4         const mid = Math.floor(len / 2); // 对半分  
5         const L = arr.slice(0, mid);  
6         const R = arr.slice(mid, len);  
7  
8         let i = 0;  
9         let j = 0;  
10        let k = 0;  
11  
12        mergeSort(L); // 对左边的进行排序  
13        mergeSort(R); // 对右边的进行排序  
14  
15        while (i < L.length && j < R.length) {  
16            if (L[i] < R[j]) {  
17                arr[k] = L[i];  
18                i++;  
19            } else {  
20                arr[k] = R[j];  
21                j++;  
22            }  
23            k++;  
24        }  
25        if (i < L.length) {  
26            for (let m = i; m < L.length; m++) {  
27                arr[m + k] = L[m];  
28            }  
29        } else {  
30            for (let m = j; m < R.length; m++) {  
31                arr[m + k] = R[m];  
32            }  
33        }  
34    }  
35}
```

```
17         arr[k] = L[i];
18         i++;
19     } else {
20         arr[k] = R[j];
21         j++;
22     }
23     k++;
24 }
25
26 // 检查是否有剩余项
27 while (i < L.length) {
28     arr[k] = L[i];
29     i++;
30     k++;
31 }
32
33 while (j < R.length) {
34     arr[k] = R[j];
35     j++;
36     k++;
37 }
38 }
39 return arr;
40 }
```

本章节将分为 3 个部分：

- Part 1
 - 合并两个有序数组 ◻
 - 第一个错误的版本 ◻
 - 搜索旋转排序数组 ◻◻
- Part 2
 - 在排序数组中查找元素的第一个和最后一个位置 ◻◻
 - 数组中的第K个最大元素 ◻◻
 - 颜色分类 ◻◻
- Part 3
 - 前K个高频元素 ◻◻
 - 寻找峰值 ◻◻
 - 合并区间 ◻◻
- Part 4
 - 搜索二维矩阵 || ◻◻
 - 计算右侧小于当前元素的个数 ◻◻

合并两个有序数组、第一个错误的版本和搜索旋转排序数组

合并两个有序数组

给定两个有序整数数组 `nums1` 和 `nums2`，将 `nums2` 合并到 `nums1` 中，使得 `num1` 成为一个有序数组。

说明:

初始化 `nums1` 和 `nums2` 的元素数量分别为 m 和 n 。你可以假设 `nums1` 有足够的空间（空间大小大于或等于 $m + n$ ）来保存 `nums2` 中的元素。

示例

```
1 输入:  
2 nums1 = [1,2,3,0,0,0], m = 3  
3 nums2 = [2,5,6], n = 3  
4  
5 输出: [1,2,2,3,5,6]
```

方法一 双指针 从前往后遍历

思路

先简化问题，从合并数组简化成合并两个元素。分别从两个数组中取出一个元素进行比较，比较完后将较小元素合并进结果数组，较大元素继续和另一个数组中取的下一个元素比较，如此循环，直到某个数组中的元素都被比较过时，剩下的数组中未被比较过的元素直接按顺序放到结果数组中。

详解

1. 定义两个指针 j 、 k ，分别指向当前 `nums1` 与 `nums2` 数组中第一个元素的下标，定义一个 `result` 数组存放合并结果
2. 比较 `nums1[j]` 和 `nums2[k]` 两个元素，将较小元素 `push` 进 `result` 中
 1. 指向较小元素的指针加 1，取出上次较大元素继续比较，循环第 2 步
 2. 当某个数组中的元素都被比较过了，将另一数组剩余元素直接 `push` 到 `result` 中，因为两个数组都是有序数组，剩下的肯定是较大值

代码

```
1  /**
2  * @param {number[]} nums1
3  * @param {number} m
4  * @param {number[]} nums2
5  * @param {number} n
6  * @return {void}
7  */
8 const merge = function (nums1, m, nums2, n) {
9    // 暂存 merge 结果
10   const result = [];
11   // 定义两个指针 j、k，分别指向当前 nums1 与 nums2 数组中正在比较值的数组下标，从前往后
12   let j = 0; let k = 0;
13   // 遍历 nums1 和 nums2 数组，遍历完一个数组后跳出循环
14   while (j < m && k < n) {
15     // 比较 nums1 中取的值与 nums2 中取的值，将较小值 push 到结果数组中
16     // 并将下标往后加一，下次循环取后一个值进行比较
17     if (nums1[j] > nums2[k]) {
18       result.push(nums2[k]);
19       k++;
20     } else {
21       result.push(nums1[j]);
22       j++;
23     }
24   }
25
26   // nums1 或 nums2 中有一个数组未遍历完全
27   if (result.length < m + n) {
28     // 如果 nums1 遍历完了，则说明 nums2 未遍历完全，
29     // 将 nums2 中剩余未比较的数据直接 push 到 merge 结果数组中
30     // 反之亦然
31     if (j === m) {
32       result.push(...nums2.slice(k, n));
33     } else {
34       result.push(...nums1.slice(j, m));
35     }
36   }
37   // 清空 nums1，将 merge 结果 push 到 nums1 中
38   nums1.splice(0, nums1.length);
39   nums1.push(...result);
40};
```

复杂度分析

- 时间复杂度： $O(m + n)$

最多遍历 $m + n - 1$ 次，所以时间复杂度为 $O(m + n)$

- 空间复杂度： $O(m)$

开辟新的空间存放 `nums1` 数组，所以空间复杂度为 $O(m)$

方法二 双指针 从后往前遍历

思路

先简化问题，从合并数组简化成合并两个元素。因为 `nums1` 数组长度可以存放最后排序好的元素，所以可以从后往前取两个数组的元素进行比较，从 `nums1` 数组的最后开始存放较大元素。较小值继续与新取出的元素进行比较，如此循环直到某个数组中的元素全部被比较过，可得最终结果。

详解

1. 定义一个指针 `p`，指向 `nums1` 数组最后一个位置($m + n - 1$)。
2. 比较 `nums1[m - 1]` 和 `nums2[n - 1]` 两个元素，将较大元素放到 `nums1[p]` 中
3. 指针 `p` 往前移动一位，，较大元素所在数组往前继续取出一个元素与上次较小元素进行比较，将较大元素放到 `nums1[p]` 中
4. 循环第 3 步，直到某个数组中的元素全部被比较过，因为 `nums1` 和 `nums2` 数组都是有序数组，所以另一数组未比较的元素肯定是较小的那部分元素，直接将剩余元素放到 `nums1` 的头部

代码

```
1  /**
2   * @param {number[]} nums1
3   * @param {number} m
4   * @param {number[]} nums2
5   * @param {number} n
6   * @return {void}
7  */
8 const merge = function (nums1, m, nums2, n) {
9  let currentInsertIndex = nums1.length - 1;
10 while (currentInsertIndex >= 0 && n > 0 && m > 0) {
11  if (nums1[m - 1] > nums2[n - 1]) {
12    nums1[currentInsertIndex--] = nums1[m - 1];
13    m--;
14  } else {
15    nums1[currentInsertIndex--] = nums2[n - 1];
16    n--;
17  }
18 }
19 // nums2 未遍历完成，将 nums2 中剩余未遍历的数据插入到 nums1 头部
20 // nums1 未遍历完成不用关心，已排序好了
21 if (n > 0) {
22  nums1.splice(0, n, ...nums2.slice(0, n));
23 }
24 };
25 };
```

复杂度分析

- 时间复杂度： $O(m + n)$
最多遍历 $m + n - 1$ 次，所以时间复杂度为 $O(m + n)$
- 空间复杂度： $O(1)$
不需要开辟新的空间，所以空间复杂度为 $O(1)$

方法三 利用 array.sort()方法

思路

直接合并两个数组并排序

详解

1. 将 `nums1` 后面的占位删除并将 `nums2` 合并 2. 用 `array.sort()` 方法排序

代码

```
1  /**
2   * @param {number[]} nums1
3   * @param {number} m
4   * @param {number[]} nums2
5   * @param {number} n
6   * @return {void}
7   */
8  const merge = function (nums1, m, nums2, n) {
9    // 两数组合并，将 nums1 后面的占位删除并放入 nums2
10   nums1.splice(m, n, ...nums2);
11   // 排序
12   nums1.sort((a, b) => a - b);
13 }
```

复杂度分析

- 时间复杂度： $O(n \log n)$
排序在 v8 引擎下的平均时间复杂度为 $O(n \log n)$
- 空间复杂度： $O(n \log n)$

排序在 v8 引擎下的平均空间复杂度为 $O(n \log n)$

第一个错误的版本

你是产品经理，目前正在带领一个团队开发新的产品。不幸的是，你的产品的最新版本没有通过质量检测。由于每个版本都是基于之前的版本开发的，所以错误的版本之后的所有版本都是错的。假设你有 n 个版本 $[1, 2, \dots, n]$ ，你想找出导致之后所有版本出错的第一个错误的版本。你可以通过调用 `bool isBadVersion(version)` 接口来判断版本号 `version` 是否在单元测试中出错。实现一个函数来查找第一个错误的版本。你应该尽量减少对调用 API 的次数。

示例

```
1 给定 n = 5，并且 version = 4 是第一个错误的版本。  
2  
3 调用 isBadVersion(3) -> false  
4 调用 isBadVersion(5) -> true  
5 调用 isBadVersion(4) -> true  
6  
7 所以，4 是第一个错误的版本。
```

方法一 暴力法[超出时间限制]

思路

直接for循环找第一个错误版本。

代码

```
1 const solution = function(isBadVersion) {  
2     return function firstBadVersion (n) {  
3         for (let i = 1; i < n; i++) {  
4             if (isBadVersion(i)) {  
5                 return i;  
6             }  
7         }  
8         return n;  
9     }  
10    };
```

复杂度分析

- 时间复杂度： $O(n)$

该方法需要遍历每一个元素，需要耗费 $O(n)$ 时间，当遇见版本特别多的时候 $O(n)$ 的时间，因此改方法时间复杂度为 $O(n)$ 。

- 空间复杂度： $O(1)$

该方法没有申请额外的空间，所以空间复杂度为 $O(1)$

方法二 二分法

思路

前一种方法需要遍历每一个元素，这样如果元素特别多的时候会耗时过多，这个时候通过二分法也就是折半法(有序数组中查找特定元素的搜索算法)来查找元素。

二分法思路：

- 首先，从数组的中间元素开始搜索，如果该元素正好是目标元素，则搜索过程结束，否则执行下一步。
- 如果目标元素大于/小于中间元素，则在数组大于/小于中间元素的那一半区域查找，然后重复步骤（1）的操作。
- 如果某一步数组为空，则表示找不到目标元素。

这样可以避免无差别遍历降低遍历耗时。

详解

- 确定数组左边边界值和右边边界值，找到边界值的中间值
- 比较中间值是否是错误版本，如果是则右边边界值=中间值-1，再找中间值比较。如果不是错误版本则左侧边界值=中间值+1，再找左侧值和右侧值之间的中间值比较，这样重复下去
- 当左侧边界值>右侧边界值得时候，说明右侧已经全是错误版本了，当前左侧的值就是临界值

代码

```

1 const solution = function(isBadVersion) {
2   return function firstBadVersion (n) {
3     let left = 1;
4     let right = n;
5     while (left <= right) {
6       const mid = Math.floor(left + (right - left) / 2);
7       if (isBadVersion(mid)) {
8         right = mid - 1;
9       } else {
10         left = mid + 1;
11       }
12     }
13     return left;
14   };
15 }
```

```
11         }
12     }
13     return left;
14 }
15 }
```

复杂度分析

- 时间复杂度为: $O(\log_2(n))$

对于n个元素的情况(去掉常数)

第一次二分 : $n/2$

第二次二分 : $n/2^2 = n/4$ 、.....

m次二分 : $n/(2^m)$

在最坏情况下是在排除到只剩下最后一个值之后得到结果，所以为 $n/(2^m) = 1$ ，得到

$$2^m = n$$

所以时间复杂度为 : $O(\log_2(n))$

- 空间复杂度 : $O(1)$ 该方法没有申请额外的空间，所以空间复杂度为 $O(1)$

搜索旋转排序数组

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])。

搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 -1。

你可以假设数组中不存在重复的元素。

你的算法时间复杂度必须是 $O(\log n)$ 级别。

示例 1:

```
1 输入: nums = [4,5,6,7,0,1,2], target = 0
2 输出: 4
```

示例 2:

```
1 输入: nums = [4,5,6,7,0,1,2], target = 3
2 输出: -1
```

方法一 二分查询最大最小值

思路

先算出 数组中最大最小值，利用 indexOf 计算之后要旋转位置，然后二分计算目标 target 位置

详解

1. 计算数组中的最大最小值
2. 定义变量，数组长度等
3. 目标值大于数组最后一位时，数组查询位置从 0 到数字中在最大位置
4. 目标值小于等于数组最后一位时，数组查询位置从数组中最小值的位置开始，到数组的最后一位，3.4 两部为了定位数组查询区间
5. 循环二分查询，计算定位数组的中间值，数组的值等于目标查询结束
6. 不等于的情况，如果目标大于中间值，则定位数组最小值等于中间值+1，目标小于中间值，则定位数组中最大值等于中间值-1，继续循环查询即可，知道定位数组查询完毕，没有结果的话，返回 -1 代表不存在

代码

```
1 const search = function (nums, target) {
2     const min = Math.min.apply(null, nums);
3     const max = Math.max.apply(null, nums);
4     const len = nums.length;
5     let pos;
6     let lo;
7     let hi;
8     let mid;
9
10    if (target > nums[len - 1]) {
11        pos = nums.indexOf(max);
12        lo = 0;
13        hi = pos;
14    } else {
15        pos = nums.indexOf(min);
16        lo = pos;
17        hi = len - 1;
18    }
19    while (lo <= hi) {
20        mid = Math.ceil((lo + hi) / 2);
21        if (nums[mid] === target) return mid;
```

```
22     if (nums[mid] < target) {  
23         lo = mid + 1;  
24     } else {  
25         hi = mid - 1;  
26     }  
27 }  
28 return -1;  
29};
```

复杂度分析：

- 时间复杂度： $O(\log(n))$
过程会最多遍历一遍数组
- 空间复杂度： $O(1)$
只产生一次临时变量存储

方法二 二分查询中间数

思路

根据数组的中间数和左右节点的大小对比，来确定升序部分的位置，然后用二分法查询目标节点在数组中的位置

详解

1. 计算数组长度，数组为0 直接返回-1
2. 定义左右值分别为数组第一个和最后一个的下标
3. 中间下标值为最大最小值的平均数
4. 如果数组中间数等于目标直接返回下标
5. 数组的中间值小于数组最后一个值，后半部分还处于升序，如果目标值在这部分数组中，则左下标等于中间值+1，代表目标值在后半部分数组，反着重新定义右下标为中间值-1，目标在前半数组
6. 数组中间值大于数组最后一个值，代表前半部分数组处于升序，如果目标在前半数组中，右标更新为中间值-1，反之，左下标更新为中间值+1
7. 二分查询到最后没找到目标值，则返回 -1 代表不存在

代码

```
1 const search = function(nums, target) {  
2     if(nums.length === 0){  
3         return -1;
```

```
4      }
5
6      let left = 0;
7      let right = nums.length - 1;
8      let mid;
9
10     while(left <= right){
11         mid = parseInt((left + right) / 2);
12         if(nums[mid] === target){
13             return mid;
14         } else if(nums[mid] < nums[right]) {
15             if(nums[mid] < target && target <= nums[right]) {
16                 left = mid + 1;
17             } else {
18                 right = mid - 1;
19             }
20         } else {
21             if(nums[left] <= target && target < nums[mid]){
22                 right = mid - 1;
23             } else {
24                 left = mid + 1;
25             }
26         }
27     }
28     return -1;
29 };
```

复杂度分析

- 时间复杂度： $O(\log(n))$

过程会最多遍历一遍数组

- 空间复杂度： $O(1)$

只产生一次临时变量存储

在排序数组中查找元素的第一个和最后一个位置、数组中的第K个最大元素和颜色分类

在排序数组中查找元素的第一个和最后一个位置

给定一个按照升序排列的整数数组 `nums`，和一个目标值 `target`。找出给定目标值在数组中的开始位置和结束位置。

你的算法时间复杂度必须是 $O(\log n)$ 级别。

如果数组中不存在目标值，返回 `[-1, -1]`。

示例

```
1 输入: nums = [5,7,7,8,8,10], target = 8
2 输出: [3,4]
3
4 输入: nums = [5,7,7,8,8,10], target = 6
5 输出: [-1,-1]
```

方法一 二分查找

思路

由于数组已经是升序排列，可直接根据二分查找，往左定位第一个位置，往右定位最后一个位置。二分查找的实现上可以使用循环或者递归。

详解

1. 根据二分查找，找到左边第一个不小于目标值的位置
2. 从上一步中的位置开始到最后，二分查找，确定右边最后一个符合条件值的位置
3. 得到结果

```
1 function getBinarySearchLowerBound (array, low, high, target) {
2     // 找到第一个不小于目标值的位置
3     while (low < high) {
4         const mid = Math.floor((low + high) / 2);
5         if (array[mid] < target) {
```

```

6         low = mid + 1;
7     } else {
8         high = mid;
9     }
10    }
11
12    // 如果相等，则匹配，否则不匹配
13    return array[low] === target ? low : -1;
14 }
15
16 function getBinarySearchUpperBound (array, low, high, target) {
17     // 找到第一个不大于目标值的位置
18     while (low < high) {
19         const mid = Math.ceil((low + high) / 2);
20         if (array[mid] > target) {
21             high = mid - 1;
22         } else {
23             low = mid;
24         }
25     }
26
27     // 如果相等，则匹配，否则不匹配
28     return array[high] === target ? high : -1;
29 }
30
31 const searchRange = function (nums, target) {
32     const size = nums.length;
33     const low = getBinarySearchLowerBound(nums, 0, size - 1, target);
34     if (low === -1) {
35         return [-1, -1];
36     }
37     // 从左边数字的位置开始
38     const high = getBinarySearchUpperBound(nums, low >= 0 ? low : 0, size - 1, target);
39     return [low, high];
40 };

```

复杂度分析

- 时间复杂度： $O(\log(n))$
过程中最差情况会遍历二遍数组
- 空间复杂度： $O(1)$
产生三个临时变量存储

数组中的第K个最大元素

在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

示例1：

```
1 输入: [3,2,1,5,6,4] 和 k = 2
2 输出: 5
```

示例2：

```
1 输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
2 输出: 4
```

说明：你可以假设 k 总是有效的，且 $1 \leq k \leq$ 数组的长度。

方法一

思路

首先通过快速排序的方法将数组升序排序，此时数组的头部为最小的元素，尾部为数组最大的元素。题目要求找到数组中的第 K 个最大的元素，即返回 $\text{length} - k$ 个元素即可。

详解

1. 本方法采用快速排序法；
2. 首先通过 `arr[Math.floor((start + end) / 2)]` 找到数组中间的元素作为主元；
3. 然后使用双指针，分别从数组的头部和尾部遍历数组；
4. 遍历过程中，把比主元小的数都放到主元的左边，比主元大的数都放到主元的右边，实现数组的升序排序；
5. 返回第 $\text{length} - k$ 个元素，即为数组中第 k 个最大的元素。

```
1 const findKthLargest = function (nums, k) {
2   return findK(nums, 0, nums.length - 1, nums.length - k);
3 };
4
5 function findK (arr, start, end, k) {
6   if (start === end) return arr[start];
7   // 主元
8   const pivot = arr[Math.floor((start + end) / 2)];
9   let i = start; let j = end;
10  while (i <= j) {
```

```

11     while (arr[i] < pivot) i++;
12     while (arr[j] > pivot) j--;
13     if (i <= j) {
14         swap(arr, i, j);
15         i++;
16         j--;
17     }
18 }
19 // 二分查到k位置
20 if (k >= (i - start)) {
21     return findK(arr, i, end, k - i + start);
22 } else {
23     return findK(arr, start, i - 1, k);
24 }
25 }
26 // 元素交换
27 function swap (arr, i, j) {
28     const temp = arr[i];
29     arr[i] = arr[j];
30     arr[j] = temp;
31 }

```

复杂度分析

- 时间复杂度： $O(n \log n)$

上述解法中，采用了快速排序的方法，快排的时间复杂度 $O(n \log n)$ 。

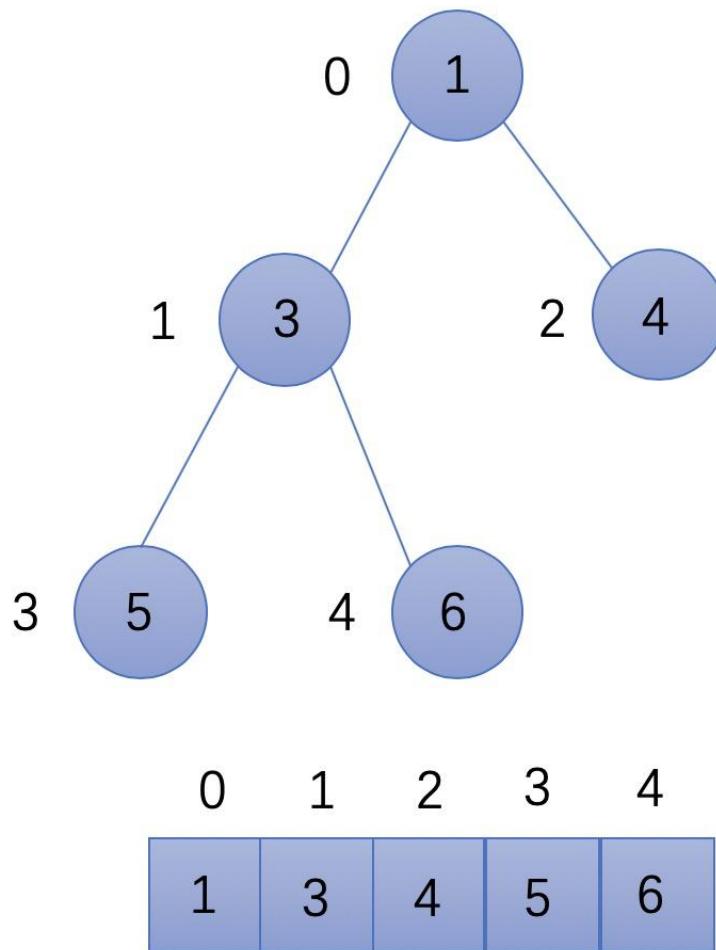
- 空间复杂度： $O(1)$

上述解法中，申请了四个额外的临时存储空间，这将耗费 $O(1)$ 的空间。

方法二

思路

首先通过最小堆排序的方法将数组升序排序，排序完的数组如下图所示：



D3D6F4FA-B956-444F-BEE5-DDCD55B65BD4.png

此时数组的头部为最小的元素，尾部为数组最大的元素。题目要求找到数组中的第 K 个最大的元素，即返回 $\text{length} - k$ 个元素即可。

详解

1. 本方法采用最小堆排序法；
2. 首先建立最小堆，将每个叶子结点视为一个堆，再将每个叶子结点与其父节点一起构成一个包含更多结点的堆；
3. 所以在构造堆的时候，首先需要找到最后一个结点的父节点，从这个节点开始构造最小堆，直到该节点前面的所有分支节点都处理完毕；
4. 然后返回第 $\text{length} - k$ 个，即为数组中第 k 个最大的元素。

```

1 const findKthLargest = function (nums, k) {
2   const size = nums.length;
3   // 建立堆

```

```

4   for (let i = parseInt(size / 2) + 1; i >= 0; i--) {
5     heapify(nums, i, size);
6   }
7   // 排序
8   for (let j = size - 1; j >= size - k; j--) {
9     // 得到本次的最大，将最大的与最后一个交换位子
10    swap(nums, 0, j);
11    heapify(nums, 0, j);
12  }
13  return nums[size - k];
14 };
15
16 function heapify (arr, x, length) {
17   // 左右两个子节点
18   const l = 2 * x + 1;
19   const r = 2 * x + 2;
20   let largest = x;
21   if (l < length && arr[l] > arr[largest]) {
22     largest = l;
23   }
24   if (r < length && arr[r] > arr[largest]) {
25     largest = r;
26   }
27   if (largest !== x) {
28     swap(arr, x, largest);
29     // 递归交换以下的是否也建好堆。
30     heapify(arr, largest, length);
31   }
32 }
33
34 function swap (arr, i, j) {
35   const temp = arr[i];
36   arr[i] = arr[j];
37   arr[j] = temp;
38 }

```

复杂度分析

- 时间复杂度： $O(n \log n)$

上述解法中，采用了堆排序的方法，堆排序的时间复杂度 $O(n \log n)$ 。

- 空间复杂度： $O(1)$

上述解法中，申请了四个额外的临时存储空间，这将耗费 $O(1)$ 的空间。

方法三

思路

首先通过冒泡排序的方法将数组升序排序，此时数组的头部为最小的元素，尾部为数组最大的元素。题目要求找到数组中的第 K 个最大的元素，即返回 $\text{length} - k$ 个元素即可。

详解

1. 本方法采用经典冒泡排序法；
2. 比较相邻的元素，如果第一个比第二个大，就交换他们两个；
3. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对；
4. 完成步骤 3 后，最后的元素会是最大的数，实现升序排序；
5. 返回第 $\text{len}-k$ 个元素，即为数组中第 k 个最大的元素。

```
1 const findKthLargest = function (nums, k) {
2   const len = nums.length;
3   for (let i = len - 1; i > 0; i--) {
4     // 冒泡排序
5     for (let j = 1; j <= i; j++) {
6       // 异或交换，详见题外话解析
7       if (nums[j - 1] > nums[j]) {
8         nums[j - 1] ^= nums[j];
9         nums[j] ^= nums[j - 1];
10        nums[j - 1] ^= nums[j];
11      }
12    }
13    if (i === (len - k)) {
14      return nums[i];
15    }
16  }
17  return nums[0];
18};
```

复杂度分析

- 时间复杂度： $O(n^2)$

上述解法中，内外两层循环，时间复杂度 $O(n^2)$ 。

- 空间复杂度： $O(1)$

上述解法中，最优的情况是开始时元素已经按顺序排好，空间复杂度为 0，最差的情况是开始时元素逆序排序，此时空间复杂度 $O(n)$ ，平均空间复杂度 $O(1)$ 。

复杂度分析：

- 时间复杂度： $O(n^2)$ ，内外两层循环，时间复杂度 $O(n^2)$
- 空间复杂度： $O(1)$ ，最优的情况是开始时元素已经按顺序排好，空间复杂度为0，最差的情况是开始时元素逆序排序，此时空间复杂度 $O(n)$ ，平均空间复杂度 $O(1)$

题外话

对于给定两个整数a,b，下面的异或运算可以实现a,b的交换，而无需借助第3个临时变量：

```
1  a = a ^ b;  
2  b = a ^ b;  
3  a = a ^ b;
```

这个交换两个变量而无需借助第3个临时变量过程，其实现主要是基于异或运算的如下性质：

1. 任意一个变量X与其自身进行异或运算，结果为0，即 $X \wedge X=0$
2. 任意一个变量X与0进行异或运算，结果不变，即 $X \wedge 0=X$
3. 异或运算具有可结合性，即 $a \wedge b \wedge c = (a \wedge b) \wedge c = a \wedge (b \wedge c)$
4. 异或运算具有可交换性，即 $a \wedge b = b \wedge a$

分析：

第一步： $a = a \wedge b;$

完成后 a 变量的结果为 $a \wedge b$

第二步： $b = a \wedge b;$

此时赋值号右边的 a 保存的是 $a \wedge b$ 的值，那么将赋值号右边的 a 用 $a \wedge b$ 替换，

得到 $(a \wedge b) \wedge b = a \wedge (b \wedge b) = a \wedge 0 = a,$

即经过第二步运算后 b 中的值为 a，即 $b=a$ ，将 a 换到了 b 里

第三步： $a = a \wedge b;$

此时赋值号右边的 a 保存的仍然是 $a \wedge b$ 的值，不变，而赋值号右边的 b 已经是 a 了，

将赋值号右边的 a,b 分别进行替换，

即此时赋值号右边 $a \wedge b = (a \wedge b) \wedge a = a \wedge b \wedge a = a \wedge b = 0 \wedge b = b$, 该值赋值给 a ,即 $a = b$

即经过第三步运算后 a 中的值为 b , 即 $a = b$, 将 b 换到了 a 里

这样经过如上的三步骤 , 完成了交换两个变量 a,b 而无需借助第 3 个临时变量过程。

颜色分类

给定一个包含红色、白色和蓝色，一共 n 个元素的数组，**原地**对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

此题中，我们使用整数 0、1 和 2 分别表示红色、白色和蓝色。

注意: 不能使用代码库中的排序函数来解决这道题。

示例

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

方法一 直接计算

思路

直接遍历整个数组，分别计算出红蓝白球的个数，然后按照红色、白色、蓝色顺序依次存入数组。

详解

1. 设定三个变量 red, white , blue 分别表示红球、白球和蓝球。
2. 遍历数组，遇到 0 则使 red 自增1，遇到 1 则使 white 自增1，遇到 2 则使 blue 自增1。
3. 根据红白蓝的个数，依次将 0 , 1 , 2 存入数组。

```
1 /**
2  * @param {number[]} nums
3  * @return {void} Do not return anything, modify nums in-place instead.
4 */
5 const sortColors = function (nums) {
```

```

6   let red = 0;
7   let blue = 0;
8   let white = 0;
9   for (let i = 0; i < nums.length; i++) {
10     if (nums[i] === 0) {
11       red++;
12     } else if (nums[i] === 1) {
13       blue++;
14     } else if (nums[i] === 2) {
15       white++;
16     }
17   }
18   let index = 0;
19   for (let i = 0; i < red; i++) {
20     nums[index++] = 0;
21   }
22   for (let i = 0; i < blue; i++) {
23     nums[index++] = 1;
24   }
25   for (let i = 0; i < white; i++) {
26     nums[index++] = 2;
27   }
28 };

```

复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(n)$

方法二 双指针遍历

思路

设定三个指针 `begin`, `end`, `i`，用 `i` 遍历数组，遇到 `0`, `1` 时分别将值与 `begin`, `end` 指向的值交换。这种方法相对于方法一的好处是只使用了一个常数空间。

详解

1. 设定一头一尾两个指针 `begin` 和 `end`，然后用一个指针 `i` 从头开始遍历数组。
2. 如果遇到 `0`，则将该数值与 `begin` 指向的值交换，并且使 `begin` 向后移一位。
3. 如果遇到 `2`，则将该数值与 `end` 指向的值交换，并且使 `end` 向前移一位，并且此时不需自加 `i`。
4. 如果遇到 `1`，则继续。
5. 最终得到新数组。

```

1  /**
2   * @param {number[]} nums
3   * @return {void} Do not return anything, modify nums in-place instead.
4  */
5 const sortColors = function (nums) {
6   let begin = 0;
7   let end = nums.length - 1;
8   let i = 0;
9   while (i <= end) {
10     if (nums[i] === 0) {
11       nums[i] = nums[begin];
12       nums[begin] = 0;
13       i++;
14       begin++;
15     } else if (nums[i] === 2) {
16       nums[i] = nums[end];
17       nums[end] = 2;
18       end--;
19     } else {
20       i++;
21     }
22   }
23 };

```

复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$

方法三 使用各种排序法

思路

本题的实质是将数字从小到大排序，可以使用各种排序法（冒泡排序法，选择排序法，快速排序法等），这里举一个冒泡排序法的例子。

```

1 /**
2  * @param {number[]} nums
3  * @return {void} Do not return anything, modify nums in-place instead.
4 */
5 const sortColors = function (nums) {
6   for (let i = 0; i < nums.length; i++) {
7     for (let j = 0; j < nums.length - i; j++) {
8       if (nums[j] > nums[j + 1]) {
9         const tem = nums[j];

```

```
10         nums[j] = nums[j + 1];
11         nums[j + 1] = tem;
12     }
13 }
14 }
15 };
```

复杂度分析

- 时间复杂度： $O(n^2)$
遍历了两次含n个元素的空间
- 空间复杂度： $O(1)$
排序过程没有用到新的空间存储数据

前 K 个高频元素、寻找峰值和合并区间

前 K 个高频元素

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：

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

示例 2：

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

说明：

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。

方法一

思路

这一题属于比较简单的算法题，整体的思路是需要开辟一个新的储存空间，对数组中数字的出现的次数进行记录，再对这个存储记录进行读取，进而得出前 K 个高频元素

详解

第一步，记录每个数出现的次数

```
1 let map = new Map();
2 for (let num of nums) { // 记录每个数出现的次数
3     map.set(num, (map.get(num) || 0) + 1);
4 }
```

第二步，设置一个数组 countOfNum，其第 i 个元素表示出现 i 次的元素数组

```
1 let countOfNum = Array(nums.length);
2 for (let key of map.keys()) {
3     let value = map.get(key);
4     if (countOfNum[value] === undefined) {
5         countOfNum[value] = [key]
6     } else {
7         countOfNum[value].push(key)
8     }
9 }
```

第三步，从 countOfNum 末尾开始往头遍历，将非空元素都加入结果数组中，直到结果数组的大小等于 K。

```
1 let res = [];
2 for (let i = countOfNum.length - 1; i >= 0 && res.length < k; i--) {
3     if (countOfNum[i] !== undefined) {
4         res = res.concat(countOfNum[i])
5     }
6 }
7 return res;
```

代码

```
1 /**
2  * @param {number[]} nums
3  * @param {number} k
4  * @return {number[]}
5  */
6 const topKFrequent = function(nums, k) {
7     let map = new Map();
8     for (let num of nums) {
9         map.set(num, (map.get(num) || 0) + 1);
10    }
11
12    let countOfNum = Array(nums.length);
13    for (let key of map.keys()) {
14        let value = map.get(key);
15        if (countOfNum[value] === undefined) {
16            countOfNum[value] = [key]
```

```

17     } else {
18         countOfNum[value].push(key)
19     }
20 }
21
22 let res = [];
23 for (let i = countOfNum.length - 1; i >= 0 && res.length < k; i--) {
24     if (countOfNum[i] !== undefined) {
25         res = res.concat(countOfNum[i])
26     }
27 }
28 return res;
29 };

```

复杂度分析

- 时间复杂度： $O(n)$
 - 上述解法中，我们使用了一层 for 循环，里面的代码会执行 n 遍，它消耗的时间是随着 n 的变化而变化的，因此时间复杂度是 $O(n)$
- 空间复杂度： $O(n)$
 - 上述解法中，我们额外声明的空间大小和输入规模成正比，所以空间复杂度为 $O(n)$

方法二

思路

整体思路同方法一，这一种方法我们使用 forEach 和 sort 简化了实现方案

详解

第一步，构建一个对象作为储存空间

```

1 const hashTable = {}
2 nums.forEach(item => {
3     if (hashTable[item] === undefined) {
4         hashTable[item] = 1
5     } else {
6         hashTable[item]++
7     }
8 })

```

第二步，根据存储的出现次数，对数组进行排序

```
1 hashTableArray = Object.keys(hashTable)
2 hashTableArray.sort((prev, next) => {
3   return hashTable[next] - hashTable[prev]
4 })
```

第三步，截取数去的前K项，得到想要的结果

```
return hashTableArray.slice(0, k)
```

代码

```
1 /**
2  * @param {number[]} nums
3  * @param {number} k
4  * @return {number[]}
5 */
6 var topKFrequent = function(nums, k) {
7   const hashTable = {}
8   nums.forEach(item => {
9     if (hashTable[item] === undefined) {
10       hashTable[item] = 1
11     } else {
12       hashTable[item]++
13     }
14   })
15   hashTableArray = Object.keys(hashTable)
16   hashTableArray.sort((prev, next) => {
17     return hashTable[next] - hashTable[prev]
18   })
19   return hashTableArray.slice(0, k)
20};
```

复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(n)$

寻找峰值

峰值元素是指其值大于左右相邻值的元素。给定一个输入数组 `nums`，其中 `nums[i] > nums[i+1]`，找到峰值元素并返回其索引。数组可能包含多个峰值，在这种情况下，返回任何一个峰值所在位置即可。你可以假设 `nums[-1] = nums[n] = -∞`。

示例

```
1 输入: nums = [1,2,3,1]
2 输出: 2
3 解释: 3 是峰值元素，你的函数应该返回其索引 2。
```

方法一 取最大值

思路

取最大值法思路最简单，最大值必然就是其峰值。因为这是算法题，我们就不用 `Math.max` 了。直接一次循环解决问题。

详解

1. 使用数组的 `reduce` 方法遍历数组，设其初始值为 0。
2. 当遍历到的值大于上一次返回的下标所对应的值时返回当前下标。
3. 遍历结束后所获取的值为最终答案。

代码

```
1 const findPeakElement = function (nums) {
2     return nums.reduce((index, cur, i) => {
3         if (nums[index] < cur) {
4             return i;
5         }
6         return index;
7     }, 0);
8 };
```

复杂度分析

- 时间复杂度： $O(n)$ 对于每个元素，通过遍历数组进行比较值的大小来寻找它所对应的目标元素，这将耗费 $O(n)$ 的时间。
- 空间复杂度： $O(1)$

算法执行的过程中，声明的变量并不与数组的长度 n 相关。故而空间复杂度为 $O(1)$ 。

方法二 暴力法

思路

暴力法很简单，遍历每个元素，并寻找到元素 n 大于元素 $n + 1$ 。则 n 就是我们要找的峰值。如果没有这个值，则数组的最后一个元素就是峰值。

详解

1. 遍历数组找出第 i 个元素大于 $i+1$ 个元素的元素的下标为结果。
2. 如果遍历完成没有找到，则取元素最后一个值的下标为结果。

代码

```
1 const findPeakElement = function (nums) {  
2     // 如果全部从小到大排列，则峰值为最后一个元素  
3     let result = nums.length - 1;  
4     // 否则只要找出第  $i$  个元素大于  $i+1$  个元素，则第  $i$  个元素就是峰值  
5     for (let i = 0; i < nums.length - 1; i += 1) {  
6         if (nums[i] > nums[i + 1]) {  
7             result = i;  
8             break;  
9         }  
10    }  
11    return result;  
12};
```

复杂度分析

- 时间复杂度： $O(n)$ 对于每个元素，通过遍历数组进行比较值的大小来寻找它所对应的目标元素，这将耗费 $O(n)$ 的时间。
- 空间复杂度： $O(1)$
算法执行的过程中，声明的变量并不与数组的长度 n 相关。故而空间复杂度为 $O(1)$ 。

方法三 二分法

思路

将数组从中间分成两个数组 L 和 R 。比较 L 最后一个值 l 与 R 的第一个值 r 的大小。如果 $l > r$ 则数组 L 必有一个及以上原数组的峰值，取数组 L 再做上一步。反之则数组 R 必有一个及以上原数组的

峰值，取数组 R 再做上一步。当数组长度为 1 时。其值为峰值。

详解

1. 将目标数组从中间分成两个数组 L 和 R。
2. 比较 L 最后一个值 l 与 R 的第一个值 r 的大小。
3. 如果 $l > r$ 则数组 L 必有一个及以上原数组的峰值，取数组 L 作为目标数组。
4. 反之则数组 R 必有一个及以上原数组的峰值，取数组 R 作为目标数组。
5. 目标数组长度为 1 则其值为峰值。
6. 目标数组长度大于 1 则重复 1, 2, 3, 4 步骤。直至目标数组长度为 1 为止。

代码

```
1 const findPeakElement = function (nums) {  
2     let lIndex = 0; // 虚拟数组第一个元素下标  
3     let rIndex = nums.length - 1; // 虚拟数组最后一个元素下标  
4     let mid; // 数组中间元素下标。  
5     while (lIndex < rIndex) { // 当数组长度不为 1 时。  
6         mid = Math.floor((lIndex + rIndex) / 2); // 取当前虚拟数组的中间元素的下标 (当数组  
7         if (nums[mid] > nums[mid + 1]) { // 比较左边数组最后一个元素与右边数组的第一个元素  
8             rIndex = mid; // 左边数组最后一个元素大、则取左边数组  
9         } else {  
10             lIndex = mid + 1; // 右边数组第一个元素大、则取右边数组  
11         }  
12     }  
13     return lIndex;  
14 };
```

复杂度分析：

- 时间复杂度： $O(\log n)$

每次循环数组的长度都除以 2。设 x 次循环之后 $lIndex === rIndex$ 。也就是说 2 的 x 次方等于 n ，那么 $x = \log_2 n$ 。也就是说当循环 $\log_2 n$ 次以后，代码运行结束。因此这个代码的时间复杂度为： $O(\log n)$ 。

- 空间复杂度： $O(1)$

算法执行的过程中，声明的变量并不与数组的长度 n 相关。故而空间复杂度为 $O(1)$ 。

合并区间

给出一个区间的集合，请合并所有重叠的区间。

示例

- ```
1 输入: [[1,3],[2,6],[8,10],[15,18]],
2 输出: [[1,6],[8,10],[15,18]]。
3 解释: 区间 [1,3] 和 [2,6] 重叠, 将它们合并为 [1,6].
```

## 方法一 合并重叠

思路：

从示例入手：1.[1, 3] [2, 6] 是否可以合并只要对比 [1, 3]的最大值 3 , [2, 6]的最小值 2 ,  $3 \leq 2$  , 则说明可以合并，否则不能。2.[1, 3] [2, 6] [8,10] , 从3个来看，如果 [2, 6]的最大值 $6 \leq [8,10]$ 的最小值 8 , 则说明[2, 6] 不能和 [8,10] , 因此 [1, 3] [2, 6] 执行合并操作，[8,10] 继续与下一个数组执行步骤一。

详解：

- ```
1 1.对区间进行排序 (升序)
2 2.从第一区间起取当前拟合并区间为a ,
3 3.取下一区间为b (如果没有b了则输出a , 退出)
4 4.如果a的尾 > b 的头 ,则合并为 a , 否则输出a , 把b作为a。
```

```
1 const merge = intervals => {
2   intervals.sort((a, b) => {
3     if (a[0] !== b[0]){
4       return a[0] - b[0];
5     }
6     return a[1] - b[1];
7   });
8   let len = intervals.length,
9   ans = [], // 定义新数组
10  start, end; // 遍历当前区间的最小值与最大值
11  for (let i = 0; i < len; i++) {
12    let s = intervals[i][0],
13    e = intervals[i][1];
14    if (start === undefined){
15      start = s, end = e;
16    } else if (s <= end){
17      end = Math.max(e, end);
18    } else {
19      let part = [start, end];
20      ans.push(part);
21    }
22  }
23  return ans;
24}
```

```

21         start = s;
22         end = e;
23     }
24 }
25 if (start !== undefined) {
26     let part = [start, end];
27     ans.push(part);
28 }
29 return ans;
30 };

```

复杂度分析

- 时间复杂度 : O(n) 该算法对含有n个结点的列表进行了一次遍历。因此时间复杂度为O(n)。
- 空间复杂度 : O(1) 该算法只用了常量级的额外空间。故空间复杂度为O(1)。

方法二 合并重叠

思路

从示例入手： 1.申明一个变量用于存储合并的结果 res = [] 2.第一个数组直接放入res中，res = [[1, 3]] 3.对比 res中的 [1, 3] 和下一个数组 [2 ,6] , [1,3] 的最大值 3 <= [2, 6] 的最小值2，则 res = [[1, 6]] 4.继续对比 res中的 [1, 6] 和下一个数组 [8, 10] ，则 res = [[1, 6], [8, 10]] 5.然后从 [8, 10] 开始继续执行步骤三

详解

1. 对区间进行排序 (升序)
2. 定义一个新的数组，用于存储新的数组区间。
3. 从第二个值开始遍历原数组，比较当前区间的最小值是否大于新数组最后一个区间的最大值，如果满足则push进入新的数组；又或者比较当前区间的最大值是否大于新数组的随后一个区间的最大值，若满足则将新数组的最后一个区间的最大值替换成当前区间的最大值。

```

1 const merge = function(intervals) {
2     if (!intervals || intervals.length === 0) {
3         return [];
4     }
5     let input = intervals.sort((a, b) => a[0] - b[0]);
6     let res = [];
7     res.push(input[0]);
8     for(let i = 1, len = input.length; i < len; i++) {
9         if (input[i][0] > res[res.length - 1][1]) {
10             res.push(input[i]);
11         } else if (input[i][1] > res[res.length - 1][1]){

```

```
12             res[res.length - 1][1] = input[i][1];
13         }
14     }
15     return res;
16 }
```

复杂度分析

- 时间复杂度： $O(n)$

该算法对含有 n 个结点的列表进行了一次遍历。因此时间复杂度为 $O(n)$ 。

- 空间复杂度： $O(n)$

该算法只开辟了额外空间用于存储结果，空间大小与数组长度 n 成正比。故空间复杂度为 $O(n)$ 。

搜索二维矩阵 II 和计算右侧小于当前元素的个数

搜索二维矩阵 II

编写一个高效的算法来搜索 $m \times n$ 矩阵 matrix 中的一个目标值 target。该矩阵具有以下特性：

每行的元素从左到右升序排列。每列的元素从上到下升序排列。

示例

```
1 现有矩阵 matrix 如下：  
2  
3 [  
4   [1, 4, 7, 11, 15],  
5   [2, 5, 8, 12, 19],  
6   [3, 6, 9, 16, 22],  
7   [10, 13, 14, 17, 24],  
8   [18, 21, 23, 26, 30]  
9 ]  
10  
11 给定 target = 5，返回 true。  
12  
13 给定 target = 20，返回 false。
```

方法一 暴力法

思路

题目只需找出二维数组中是否包含目标值，因此直接用两个 for 循环遍历二维矩阵的所有元素，找到目标元素则返回 true，遍历完仍未找到则返回 false。

详解

1. 第一层 for 循环取到二维数组中所有的一维数组
2. 第二层 for 循环中用两个索引值取到二维数组中每一个具体的值
3. 将取到的每一个值和目标值作比较，若相同则返回 true
4. 若直到所有循环结束还未找到与目标值相同的值则返回 false

代码

```
1 const searchMatrix = function (matrix, target) {
2     for (let i = 0; i < matrix.length; i++) {
3         for (let j = 0; j < matrix[i].length; j++) {
4             if (matrix[i][j] === target) {
5                 return true;
6             }
7         }
8     }
9     return false;
10};
```

复杂度分析

- 时间复杂度： $O(n^2)$ 通过遍历二维矩阵的所有元素来寻找它所对应的目标元素，这将耗费 $O(n^2)$ 的时间。
- 空间复杂度： $O(1)$

方法二 相邻比较法

思路

由于矩阵的行和列是排序的，从左到右递增，从上到下递增，所以对任意元素和目标值比较大小时，总可以去找相对较小（往左往上）或相对较大（往右往下）的值继续比较，直到找到目标值或找不到。

详解

- 直接取二维数组中左下角的值和目标值比较
- 若该值比目标值大，则向上查找（第一个索引减 1），若该值比目标值小，则向右查找（第二个索引加 1）
- 重复该操作，当查询值等于目标值时则返回 true
- 若直到查询值离开二维数组时还未找到目标值则返回 false

代码

```
1 const searchMatrix = function (matrix, target) {
2     let j = matrix.length - 1;
3     let i = 0;
4     while (j >= 0 && i < matrix[0].length) {
5         if (matrix[j][i] > target) {
6             j--;
7         } else if (matrix[j][i] < target) {
```

```
8         i++;
9     } else {
10        return true;
11    }
12 }
13 return false;
14 };
```

复杂度分析

- 时间复杂度： $O(n + m)$ 由于行只能运算 m 次，列只能运算 n 次，是两个变量之和，这将耗费 $O(n + m)$ 的时间。
- 空间复杂度： $O(1)$ 用到了两个变量，其空间复杂度为 $O(1)$

计算右侧小于当前元素的个数

给定一个整数数组 $nums$ ，按要求返回一个新数组 $counts$ 。数组 $counts$ 有该性质： $counts[i]$ 的值是 $nums[i]$ 右侧小于 $nums[i]$ 的元素的数量。

示例

```
1 输入: [5,2,6,1]
2 输出: [2,1,1,0]
3 解释:
4 5 的右侧有 2 个更小的元素 (2 和 1).
5 2 的右侧仅有 1 个更小的元素 (1).
6 6 的右侧有 1 个更小的元素 (1).
7 1 的右侧有 0 个更小的元素.
```

方法一 暴力法

思路

暴力法很简单，遍历每个元素 x ，并遍历查找在其后边并且比它小的元素，进行累加，最后将统计出来的个数 push 到新开辟的数组中。

详解

算法流程：

1. 声明一个数组用来存储最后的结果；
2. 第一层循环用来遍历所有的元素；
3. 循环体内先声明一个变量，初始值为 0，用来记录当前元素右边并且比当前元素小的个数；
4. 接下来通过第二个循环来计算当前元素右边比它小的数，由于只需要计算它右边的数，所以第二个循环从第 $i + 1$ 开始，遍历到数组的最后一个值即可。第二层循环体内通过 if 语句来判断比当前元素小的数，然后执行 $num++$ ；

代码

```
1 const countSmaller = (nums) => {
2     const newArr = [];
3     for (let i = 0, len = nums.length; i < len; i += 1) {
4         let num = 0;
5         for (let j = i + 1; j < nums.length; j += 1) {
6             if (nums[j] < nums[i]) {
7                 num += 1;
8             }
9         }
10        newArr.push(num);
11    }
12    return newArr;
13};
```

复杂度分析

- 时间复杂度： $O(n^2)$ 对于每个元素，通过遍历数组的其余部分来寻找它所对应的目标元素，所以时间复杂度为 $O(n^2)$ 。
- 空间复杂度： $O(n)$ 额外开辟了一个数组来存放结果，所以空间复杂度为 $O(n)$ 。

方法二 排序法

思路

首先将数组元素进行从小到大的排序，排序完成后，遍历原数组，找出原数组中当前元素在排序后的数组中的数组下标，即为该元素右侧比它小的个数，然后将排序后的数组中的这个元素删除。

例如：原数组 $nums = [5,2,6,1]$ ；排序后的数组为 $newArr = [1,2,5,6]$ ；对原数组 $nums$ 进行遍历，首先是元素 5，在 $newArr$ 中其数组下标为 2，其右侧比它小的元素有 2 个；然后将 $newArr$ 中的 5 这个元素删除， $newArr = [1,2,6]$ ；接下来是元素 2，在 $newArr$ 中的下标为 1，然后将 $newArr$ 中的 2 这个元素删除，其右侧比它小的元素有 1 个；以此类推...

详解

算法流程：

1. 将原数组从小到大排序，并声明一个新数组来存储；
2. 声明一个数组用来存储最后的结果；
3. 对原数组进行遍历；
4. 通过数组的方法 `indexOf` 找到遍历的当前元素在排序后的数组中的位置，并赋值给变量 `index`，该变量的值即为当前元素右侧比它小的个数；
5. 通过数组的方法 `splice`，把当前元素从原数组中删除；

代码

```
1 const countSmaller = (nums) => {
2   const newArr = [...nums].sort((a, b) => a - b);
3   const result = [];
4   nums.forEach((n) => {
5     const index = newArr.indexOf(n);
6     result.push(index);
7     newArr.splice(index, 1);
8   });
9   return result;
10};
```

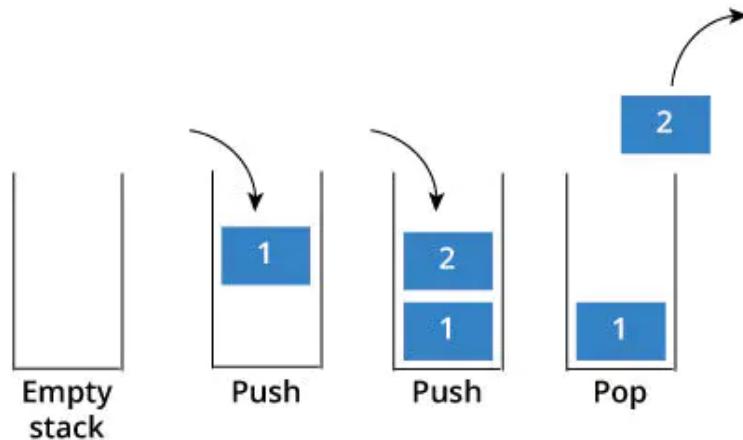
复杂度分析

- 时间复杂度： $O(n \log n)$ 代码中 `sort` 函数的时间复杂度为 $O(n \log n)$ ，`for` 循环的时间复杂度为 $O(n)$ ，因此整体的时间复杂度为 $O(n \log n)$ 。
- 空间复杂度： $O(n)$ 申请了 2 个大小为 n 的数组空间，因此空间复杂度为 $O(n)$ 。

栈和队列

栈(Stack)

栈是一种特殊的列表，限定仅在表尾进行插入和删除操作的线性表。这一端被称为栈顶，相对地，把另一端称为栈底。它按照**先进后出**的原则存储数据，先进入的数据被压入栈底，最后的数据在栈顶，需要读数据的时候从栈顶开始弹出数据。例如饭店里用来放盘子的，就是一种栈的结构。



由于栈具有**后入先出**的特点，因此只能访问在栈顶的元素。栈的操作主要有两种：入栈和出栈。上面的图演示了入栈和出栈的过程。

栈的实现

实现一个栈，可以用数组来实现，也可以用链表来实现。下面将用数组来实现。

```
1 class Stack {
2     constructor() {
3         this.data = [];
4         this.top = 0; // 记录栈顶位置，如果有元素进栈，该变量值随之变化
5     }
6
7     push(item) {
8         this.data[this.top++] = item;
9     }
10
11    pop() {
12        // 栈为空，则直接返回null
13        if (this.top === 0) {
14            return null;
15        }
16        // 返回下标为top - 1 的数组元素，并将栈中元素个数减一
17    }
18}
```

```
17     return this.data[--this.top];
18 }
19
20 clear() {
21     this.top = 0;
22 }
23
24 get length() {
25     return this.top;
26 }
27 }
```

那么栈的空间、时间复杂度又分别是多少？其实很简单，我们发现储存数据只需要长度为 n 的数组就够了，在入栈和出栈过程中，只需要一两个临时变量存储空间，所以空间复杂度是 $O(1)$ 。

注意 Δ ：算法的复杂度是指程序指除了原本的数据存储空间外，算法运行还需要额外的存储空间。

队列(Queue)

队列是一种**先进先出**的数据结构，这点和栈不一样。队列这个概念很好理解，可以想象成在食堂排队买饭吃，先到先得，后面来的排队，不允许插队。

队列的主要操作有两种：向队列中插入或删除新的元素。插入操作可以比喻成吃饭，删除操作可以比喻成(XX)。

和栈一样，队列可以用数组来实现，也可以用链表来实现。下面将用数组来实现一个队列。

```
1 class Queue {
2     constructor() {
3         this.data = [];
4         // head表示头部下标，tail表示尾部下标
5         this.head = 0;
6         this.tail = 0;
7     }
8
9     // 入队
10    enqueue(item) {
11        this.data[this.tail++] = item;
12    }
13
14    dequeue() {
15        // 如果head == tail 表示队列为空
16        if (this.head === this.tail) {
17            return null;
18        }
19        return this.data[this.head];
20    }
21}
```

```
18     }
19     return this.data[++this.head];
20   }
21
22   get length() {
23     return this.tail - this.head;
24   }
25 }
```

对于栈来说，一个指针就够了，但是队列需要两个指针，分别用于指向头部和尾部。

在栈和队列相关的题目之外，挑选了一些额外的题目。不知道看过前面章节的你掌握了多少？这部分的题目可以帮助大家测试以下算法掌握得怎么样了。建议先不要急着看答案，越能自主解决问题，自身的能力才提升得更快。

本章节分为 4 个部分：

- Part 1
 - 汉明距离
 - 位1的个数
 - 缺失的数字
- Part 2
 - 有效的括号
 - 帕斯卡三角形
 - 颠倒二进制位
- Part 3
 - 两整数之和
 - 数据流的中位数
 - 逆波兰表达式
- Part 4
 - Task Scheduler
 - 有序矩阵中第K小的元素
 - 多数元素

汉明距离、位 1 的个数、缺失数字

汉明距离

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目

给出两个整数 x 和 y ，计算它们之间的汉明距离

注意： $0 \leq x, y < 2^{31}$

示例

```
1 输入: x = 1, y = 4
2
3 输出: 2
4
5 解释:
6 1   (0 0 0 1)
7 4   (0 1 0 0)
8       ↑   ↑
9 上面的箭头指出了对应二进制位不同的位置
```

方法一

思路

题目要求比较二进制不同位，那我们就转化为二进制，再做比较

详解

1. 将两个数字转换为二进制字符串
2. 比较字符串长度，短的数组在前面补 0
3. 最后比较这两个数组的相同索引，不相同则 +1

代码

```
1 const hammingDistance = function (x, y) {
2     let xStr = Number(x).toString(2);
3     let yStr = Number(y).toString(2);
```

```

4  const xLen = xStr.length;
5  const yLen = yStr.length;
6  let counter = 0;
7
8  if (xLen > yLen) {
9    yStr = Array(xLen - yLen + 1).join('0') + yStr;
10 } else if (xLen < yLen) {
11   xStr = Array(yLen - xLen + 1).join('0') + xStr;
12 }
13
14 for (let i = 0; i < xStr.length; i++) {
15   if (xStr[i] !== yStr[i]) {
16     counter += 1;
17   }
18 }
19 return counter;
20 };

```

复杂度分析

- 时间复杂度: $O(n)$
与数字转为二进制的长度线性相关，复杂度为 $O(n)$
- 空间复杂度： $O(n)$
与数字转为二进制的长度线性相关，复杂度为 $O(n)$

方法二 异或

思路

求汉明距离，即求两数不同位的数量。那么将两数做异或运算，即可得到两数不同位的位置

异或 \wedge 运算法则：两位不同，结果为1，否则为0

代码

```

1 const hammingDistance = function (x, y) {
2   return ((x ^ y).toString(2).match(/1/g) || []).length;
3 };

```

复杂度分析

- 时间复杂度： $O(1)$
时间复杂度为常量
- 空间复杂度： $O(1)$
复杂度为常量，为 $O(1)$

方法三 异或

思路

异或取值，转为 string 后用正则匹配 1 的个数

详解

1. 先将两个数异或运算得到z，那么 z 里面 1 的个数就是结果
2. 如果 z 不为 0，那么 z 至少有一位是 1
3. 将 z - 1，会出现两种情况，最右边的 1 就会变为 0 或者 1 后面的所有 0 都会变成 1，并且 1 变成 0，其余的位不受影响
比如 $101 \rightarrow 100$ $101 \& 100 \rightarrow 100$ $100 \&& 011$
4. 右边这部分的 & 运算结果就为 0，然后循环

与 `&` 运算法则：两位同时为 1，结果才为 1，否则为 0

```
1 const hammingDistance = function (x, y) {
2     let z = x ^ y;
3     let counter = 0;
4
5     while (z) {
6         z = z & (z - 1);
7         counter++;
8     }
9     return counter;
10};
```

复杂度分析

- 时间复杂度： $O(n)$
与数字转为二进制的长度线性相关，复杂度为 $O(n)$
- 空间复杂度： $O(1)$
复杂度为常量，为 $O(1)$

位1的个数

编写一个函数，输入是一个无符号整数，返回其二进制表达式中数字位数为 '1' 的个数（也被称为汉明重量）。

在某些语言（如 Java）中，没有无符号整数类型。在这种情况下，输入和输出都将被指定为有符号整数类型，并且不应影响您的实现，因为无论整数是有符号的还是无符号的，其内部的二进制表示形式都是相同的。

示例 1：

```
1 输入：000000000000000000000000000000001011  
2 输出：3  
3 解释：输入的二进制串 000000000000000000000000000000001011 中，共有三位为 '1'。
```

示例 2：

```
1 输入：128  
2 输出：1  
3 解释：输入整数128的二进制串 0000000000000000000000000000000010000000 中，共有一位为 '1'。
```

示例 3：

```
1 输入：-3  
2 输出：1  
3 解释：输入-3的二进制串11111111111111111111111111111101 中，共有31位为 '1'。
```

方法一 替换法

思路

数字可能有各种形式的，正数、负数、二进制数等等，那么我们可以按照如下步骤，来完成位1个数的统计。
1.把各种数字转成二进制，并且转化为字符串；
2.然后我们可以使用字符串的方法
`replace`函数 把二进制中的0用空字符串 "" 来代替；
3.最后我们可以计算返回剩余的字符串的长度就是需要统计的位1的个数。

详解

1. 对于给定的目标数转换为二进制数并赋值给一个新的常量 seconds;
2. 设置一个新的变量 one 并赋值为 seconds 去除非 1 的新字符串;
3. 返回上述常量 one 的长度即为所求的位 1 的个数。

代码

```
1  /**
2   * @param {number} n - a positive integer
3   * @return {number}
4   */
5  const hammingWeight = function (n) {
6    // 对于任一数字n处理为2进制的字符串
7    const seconds = n.toString('2');
8    // 删除字符串中的“0”
9    const one = seconds.replace(/0/g, '');
10   return one.length;
11 };
```

复杂度分析

- 时间复杂度： $O(1)$

对于每个元素，通过把目标元素用空格替换来获取，这将耗费 $O(1)$ 的时间。

- 空间复杂度： $O(1)$

上述解法中只申请了一个变量空间，没有申请额外的空间，因此复杂度为 $O(1)$ 。

方法二 位操作技巧法

思路

解题思路：这里关键的想法是对于任意数字 n ，我们需要把二进制的每一位 1 转换为 0。那么，一个整数的二进制中有多少个 1，转化操作多少次，即位 1 的个数就是转化的次数。如何把二进制的每一位都转换为 0 呢？我们可以考虑用 n 和 $n-1$ 的二进制来理解。在二进制表示中，数字 n 中最低位的 1 总是对应 $n-1$ 中的 0。因此，将 n 和 $n-1$ 与运算总是能把 n 中最低位的 1 变成 0，并保持其他位不变。为了更形象，我们可以用以下图示：

n	...	0	0	1	0	1	1
n-1	...	0	0	1	0	1	0
n&(n-1)	...	0	0	1	0	1	0

n	...	0	0	1	0	1	0
n-1	...	0	0	1	0	0	1
n&(n-1)	...	0	0	1	0	0	0

n	...	0	0	1	0	0	0
n-1	...	0	0	1	1	1	1
n&(n-1)	...	0	0	1	0	0	0

...

详解

1. 设置一个变量 count ,用来记录 1 的个数；
2. 对任意不为 0 的数字 n 进行循环, 执行二进制的位操作方法 , 将数字 n 的每一位逐渐转换 为 0 ；
3. 当每执行一次循环 (也就是每转换一位 1 为 0) , 变量 count 就执行一次加 1 的操作 ；
4. 当数字 n 的每一位都为 0 的时候, 循环就结束了 ；
5. 此时 count 的值也就是我们所求的位1的个数。

代码

```

1 const hammingWeight = function (n) {
2     let count = 0;
3     while (n !== 0) {
4         count++;
5         n = n & (n - 1);
6     }
7     return count;
8 };

```

复杂度分析

- 时间复杂度 : $O(1)$

上述解法中，运行时间依赖于数字 n 中 1 的位数。由于这题中 n 是一个 32 位数，所以运行时间复杂度是 $O(1)$ ， n 在最坏情况下 32。

- 空间复杂度： $O(1)$

该表最多需要存储 1 个元素，没有申请额外的空间，所以空间复杂度为 $O(1)$ 。

缺失数字

给定一个包含 $0, 1, 2, \dots, n$ 中 n 个数的序列，找出 $0..n$ 中没有出现在序列中的那个数。

示例 1：

```
1 输入: [3,0,1]
2 输出: 2
```

示例 2：

```
1 输入: [9,6,4,2,3,5,7,0,1]
2 输出: 8
```

方法一 暴力法

思路

暴力法很简单，排序数组，遍历每个元素 x ，判断 $index$ 和当前元素是否相等，不相等的时候返回 $index$ ，对应的 $index$ 就是缺失的数

详解

1. 对原数组进行从小到大排序
2. 再对 Array 做处理，只有下标和当前值相同的时候，说明这个数值没有缺失
3. 用 reduce 方法返回仅缺失的那个数字

```
1 /**
2  * @param {number[]} nums
3  * @return {number}
4 */
```

```
5 const missingNumber = function (nums) {
6     // 对数组进行排序
7     return nums.sort((a, b) => a - b).reduce((arr, item, index) => {
8         // 判断当index和当前遍历到的值相同的时候，不对arr插入值
9         if (item !== index) {
10             // 只有不同的时候，才会往arr中塞值
11             arr.push(index);
12         }
13     return arr;
14     // 只会有一个数字，所以只取第一个数字
15 }, []);
16 };
```

复杂度分析

- 时间复杂度： $O(n * \log n)$

对于每个元素，首先通过排序确定数组从小到大的顺序，构造一个从小到大展示的数组，比如 [0,1,3]，再通过查找下标和当前数据不同的值，获取缺失的数字，这将耗费 $O(n * \log n)$ 的时间。

- 空间复杂度： $O(1)$

遍历的时候，并不会增加新的存储空间，所以空间复杂度是 $O(1)$

方法二 计算法

思路

计算法通过计算数据缺失的方式得到缺失的数字

详解

- 对原数组进行所有数据相加
- 再计算如果不缺失数字的时候，应该得到的相加数字的总和
- 用第二步计算出的数据减去第一步的数据就是缺失的数字

```
1 /**
2  * @param {number[]} nums
3  * @return {number}
4 */
5 const missingNumber = function (nums) {
6     let sum = 0;
7     // 计算数组的相加的和
8     nums.forEach((i) => { sum += i; });
9     const length = nums.length;
```

```
10 // 计算如果不缺失数字的时候，相加得到的值应该是多少
11 const termial = (1 + length) * length / 2;
12 // 相减得到缺失的值
13 return termial - sum;
14 };
```

复杂度分析

- 时间复杂度： $O(n)$

会有 2 次遍历数组，所以最终的时间复杂度是 $O(n)$

- 空间复杂度： $O(1)$

会有 2 个临时变量，不会随着入参数组的增加而增加，所以空间复杂度是 $O(1)$

方法三 按位异或 (XOR) 法

思路

可以先看看官方对异或的解释：“对于每一个比特位，当两个操作数相应的比特位有且只有一个 1 时，结果为 1，否则为 0。”，既然只缺失一位数字，那么当前元素和当前下标 index 进行异或运算的时候，不缺失的数字响应的比特一定是不止一个 1，所以可以用此方法算出对应缺失的值

详解

1. 对数组进行遍历
2. 对当前元素和当前下标做异或运算
3. 对上一步计算出的数据进行异或运算
4. 当遍历完成之后，得到的数值就是缺失的数字

```
1 /**
2  * @param {number[]} nums
3  * @return {number}
4 */
5 const missingNumber = function (nums) {
6   let res;
7   nums.forEach((i, index) => {
8     // 异或操作
9     res ^= i ^ (index + 1);
10  });
11  return res;
12 };
```

复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$

有效的括号、帕斯卡三角形和颠倒二进制位

有效的括号

给定一个只包括 '(', ')', '{', '}', '[', ']' 的字符串，判断字符串是否有效。

有效字符串需满足：

- 左括号必须用相同类型的右括号闭合。
- 左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

示例 1：

```
1 输入: '()';
2 输出: true;
```

示例 2：

```
1 输入: '()[]{}';
2 输出: true;
```

示例 3：

```
1 输入: '()';
2 输出: false;
```

示例 4：

```
1 输入: '([)]';
2 输出: false;
```

示例 5：

```
1 输入: '{[]}';  
2 输出: true;
```

方法一

思路

遍历 s，如果是左括号就压栈，如果是右括号就与栈顶元素进行匹配，匹配成功执行出栈操作 如果遍历完成后，栈中无元素，说明是有效字符串

详解

首先，左括号我们是无法判断是否合法的 因为所谓的不合法是当右括号与左括号不匹配的时候才发生的

1. 那么我们遇到左括号的时候，先把他推入栈中
2. 当我们遇到右括号的时候，刚好就可以与栈中的左括号进行匹配，如果正确，就把左括号出栈
3. 如果最终栈为空，说明全部匹配成功

代码

```
1  /**
2   * @param {string} s
3   * @return {boolean}
4   */
5 const isValid = function (s) {
6   const stack = [];
7   const map = {
8     ')': '(',
9     '}': '{',
10    ']': '['
11  };
12  for (const c of s) {
13    if (!(c in map)) {
14      stack.push(c);
15    } else if (!stack.length || map[c] !== stack.pop()) {
16      return false;
17    }
18  }
19 }
20 return !stack.length;
```

```
21 };
```

复杂度分析

- 时间复杂度： $O(n)$
遍历了 1 次含n个元素的空间
- 空间复杂度： $O(n)$

方法二

思路

遇见匹配的问题，最好的解决方案就是Stack结构，但是JS本身是没有栈结构的，JS可以用数组来实现栈。

详解

- 碰到左括号，push右括号，
- 不是左括号，判断栈是否为空或栈顶元素是否等当前元素
- 最终数组是否有元素判断 s 是否有效的括号

代码

```
1  /**
2   * @param {string} s
3   * @return {boolean}
4   */
5 const isValid = function (s) {
6   const rightSymbols = [];
7   for (let i = 0; i < s.length; i++) {
8     if (s[i] === '(') {
9       rightSymbols.push(')');
10    } else if (s[i] === '{') {
11      rightSymbols.push('}');
12    } else if (s[i] === '[') {
13      rightSymbols.push(']');
14    } else if (rightSymbols.pop() !== s[i]) {
15      return false;
16    }
17  }
18  return !rightSymbols.length;
19};
```

复杂度分析

- 时间复杂度： $O(n)$
遍历了 1 次含 n 个元素的空间
- 空间复杂度： $O(n)$
遍历过程没有用到新的空间存储数据

方法三

思路

与方法一类似，方法三使用 Map 数据类型。让代码更具可读性

详解

首先，左括号我们是无法判断是否合法的 因为所谓的不合法是当右括号与左括号不匹配的时候才发生的

1. 那么我们遇到左括号的时候，先把他推入栈中
2. 当我们遇到右括号的时候，刚好就可以与栈中的左括号进行匹配，如果正确，就把左括号出栈
3. 如果最终栈为空，说明全部匹配成功

代码

```
1 const isValid = function (s) {
2     const stack = [];
3     const map = new Map([
4         [')', '('],
5         ['}', '{'],
6         [']', '[']
7     ]);
8
9     for (const c of s) {
10        if (!map.has(c)) {
11            stack.push(c);
12        } else if (!stack.length || map.get(c) !== stack.pop()) {
13            return false;
14        }
15    }
16    return !stack.length;
17};
```

复杂度分析

- 时间复杂度： $O(n)$
遍历了 1 次含 n 个元素的空间
- 空间复杂度： $O(n)$
遍历过程没有用到新的空间存储数据

帕斯卡三角形

给定一个非负整数 $numRows$ ，生成杨辉三角的前 $numRows$ 行。

示例

```
1 输入: 5
2 输出:
3 [
4     [1],
5     [1,1],
6     [1,2,1],
7     [1,3,3,1],
8     [1,4,6,4,1]
9 ]
```

方法一 递归

思路

利用递归，除了第一个和最后一个元素之外，其他元素为它左上方的元素和正上方元素的和

详解

找出如何递归之后，我们需要找出递归终止的条件，当递归到 `index === rowNum` 或者 `index === 0` 时，即为第一个、最后一个元素时，跳出递归

代码

```
1 function combination (rowNum, index) {
2     if (index === 0 || rowNum === index) { // 最后一个数与第一个数始终为 1
3         return 1;
4     }
```

```

5   return combination(rowNum - 1, index - 1) + combination(rowNum - 1, index);
6 }
7
8 const generate = function (numRows) {
9   const result = [];
10  for (let i = 0; i < numRows; i++) {
11    const row = [];
12    for (let j = 0; j <= i; j++) {
13      row.push(combination(i, j));
14    }
15    result.push(row);
16  }
17  return result;
18 };

```

复杂度分析

- 时间复杂度： $O(2^n)$
递归的深度是 n , $O(n) = 2 * 2 * \dots * 2 = 2^n$
- 空间复杂度： $O(2^n)$
同理，每次需要额外申请 $O(2^n)$ 的堆栈内存，递归代码在紧凑易懂的同时，牺牲了执行速度，同时因为大量使用堆栈内存也牺牲了空间。

方法二 动态规划

思路

知道一行，就可以根据每对相邻的值，计算出下一行

详解

```

1  1          1
2  1 1      --->  1 1
3    \| 
4  1 X 1      1 2 1

```

每一列的头和尾都是 1，中间的每一个元素为上一列的当前索引和前一个索引之和

代码

```

1 const generate = function ( numRows ) {
2     if ( numRows ) {
3         const result = [[1]];
4         for ( let i = 1; i < numRows; i++ ) {
5             // 第一个元素始终为 1
6             const row = [1];
7             const prevRow = result[i - 1];
8
9             for ( let j = 1; j < i; j++ ) {
10                 row.push( prevRow[j] + prevRow[j - 1] );
11             }
12
13             // 最后一个也为 1
14             row.push(1);
15             result.push( row );
16         }
17         return result;
18     };
19     return [];
20 };

```

复杂度分析

- 时间复杂度： $O(n^2)$

在最外层循环需要运行 n 次，在里面一层需要循环 $s = 1 + 2 + \dots + n$ ，这里利用小学三年级的数学知识得到 $s = n(n + 1)/2 = O(n^2)$

- 空间复杂度： $O(n^2)$

需要在 `result` 中保存每个数字，与时间复杂度相同

颠倒二进制位

颠倒给定的 32 位无符号整数的二进制位。

示例

```

1 输入: 00000010100101000001111010011100
2 输出: 00111001011110000010100101000000
3 解释: 输入的二进制串 00000010100101000001111010011100 表示无符号整数 43261596 ,
4     因此返回 964176192，其二进制表示形式为 00111001011110000010100101000000。
5
6 输入 :11111111111111111111111111111101
7 输出 :10111111111111111111111111111111
8 解释 :输入的二进制串 111111111111111111111111111101 表示无符号整数 4294967293 ,

```

方法一

思路

将十进制转为二进制，补齐32位，翻转，在转换成十进制数。

详解

1. 换成 2 进制数字符串。
2. 不足 32 位的，前面补零
3. 转换成一位一位的数组
4. 数组翻转
5. 转换为字符串
6. 转换为十进制数

很简单的方法，但是执行时间非常长。

将数字转换成 2 进制数，然后不足 32 位的，前面补零。转换成一位一位的数组，颠倒，再转回字符串，再由 2 进制数转为普通数字。

```
1 const reverseBits = function (n) {
2   const temp = n.toString(2).padStart(32, 0).split('').reverse().join('');
3   return Number.parseInt(temp, 2);
4 };
```

复杂度分析

- 时间复杂度： $O(n)$
 - 空间复杂度： $O(1)$
- 并未申请额外空间，所以空间复杂度为 $O(1)$

方法二

思路

首先了解 10 进制数 转 2 进制数的算法。不断地除 2 取余，然后倒序拼接就是 2 进制数。

我们要颠倒 2 进制数，不断取余正序拼接就是我们想要的结果了。

详解

1. 在外面定义一个变量，接受结果。
2. 循环32次，每次判断是否能被 2 整除
 1. 能被 2 整除，变量拼接 '0'；然后 $n = n / 2$ 继续循环；
 2. 不能被 2 整除，变量拼接 '1'；然后 $n = \text{Math.floor}(n / 2)$ 继续循环

```
1 const reverseBits = function (n) {  
2     let str = '';  
3     while (str.length < 32) {  
4         if (n % 2 > 0) {  
5             str += '1';  
6             n = Math.floor(n / 2);  
7         } else {  
8             str += '0';  
9             n = n / 2;  
10        }  
11    }  
12    return parseInt(str, 2);  
13};
```

复杂度分析：

- 时间复杂度： $O(n)$
二进制数 32 位，所有位数循环一次时间复杂度为 $O(n)$
- 空间复杂度： $O(1)$
并未申请额外空间，所以空间复杂度为 $O(1)$

两整数之和、数据流的中位数和逆波兰表达式

两整数之和

不使用运算符 `+` 和 `-`，计算两整数 `a`、`b` 之和。

示例 1：

```
1 输入: a = 1, b = 2
2 输出: 3
```

示例 2：

```
1 输入: a = -2, b = 3
2 输出: 1
```

方法一 二进制位操作法

思路

既然不能直接使用运算符 `+` 和 `-`，可以考虑使用二进制相关的与、非和异或等位操作符来完成运算。

详解

- 与运算：二进制中的与运算是均为 1 才得 1，这样可以得到需要进位的地方，然后将其左移一位得到其进位后的结果。如：

```
1 a = 5 = 0101
2 b = 4 = 0100
3
4 a & b 如下：
5
6 0 1 0 1
7 0 1 0 0
8 -----
9 0 1 0 0
```

```
10  
11 结果左移一位后如下：  
12 1 0 0 0
```

- 异或运算：二进制中的异或运算是无进位加法，也就是两数异或得到所有不需要进位的结果。如：

```
1 a = 5 = 0101  
2 b = 4 = 0100  
3  
4 a ^ b 如下：  
5  
6 0 1 0 1  
7 0 1 0 0  
8 -----  
9 0 0 0 1
```

因此对于任意两个数，我们可以通过与运算后左移一位得到需要进位的结果，两数异或运算得到不需要进位的结果。循环操作，直到需要进位的为0，那么得到的不需要进位的结果就是两数之和。

- 解题步骤如下：
- 两整数 `a`、`b` 之和的问题可以拆分为 `a` 和 `b` 的进位结果 + `a` 和 `b` 的无进位结果
- 需要进位的地方进行与操作并将得到的结果左移一位得到 `a` 和 `b` 的进位结果
- 不需要进位的地方进行异或操作得到 `a` 和 `b` 的无进位结果
- 一直循环操作第 2 和 3 步，直到需要进位的为0

代码

```
1 const getSum = function (a, b) {  
2   while (a !== 0) {  
3     const t = (a & b) << 1; // a & b 将需要进位的地方进行与操作，并左移一位  
4     b = a ^ b; // a ^ b 将不需要进位的地方进行异或操作  
5     a = t; // 每次将需要进位的赋值给 a，继续执行上一步  
6   }  
7   return b;  
8 };
```

复杂度分析

- 时间复杂度： $O(n)$

对于任意两个数，循环体的代码需要执行 $3n$ 次。问题规模属于线性阶，故时间复杂度为 $O(n)$ 。

- 空间复杂度： $O(1)$

方法二 取对数法

思路

我们可以利用同底数幂的底数不变，指数相加的乘法规则，先取指数再取对数，等到答案。JS 中 Math 对象相关函数如下：

```
1 Math.log();  
2 // 用于返回一个数的自然对数  
3 Math.exp();  
4 // 用于返回一个数的指数  
5 Math.round();  
6 // 用于返回一个数四舍五入后最接近的整数
```

详解

1. 此方法主要借用了 JS 内置对象的 Math.log()、Math.exp() 和 Math.round() 三个函数
2. 先取两个整数的指数的乘积
3. 再取乘积结果的对数
4. 最后再对得到的对数四舍五入，得到最终答案
5. 注：引入了 10 的 10 次方是为了解决精度丢失的问题

参考代码

```
1 const getSum = function (a, b) {  
2     return Math.round(Math.log((Math.exp(a / 10e10) * Math.exp(b / 10e10))) * 10e1  
3 };
```

复杂度分析

- 时间复杂度： $O(1)$

对于任意两个数，只需要进行 1 次运算。问题规模属于常数阶，故时间复杂度为 $O(1)$ 。

- 空间复杂度： $O(1)$

此算法需要为 2 个常量 a 和 b 分配空间。空间占用属于常数阶，故空间复杂度为 $O(1)$ 。

数据流的中位数

中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。

例如，

[2,3,4] 的中位数是 3

[2,3] 的中位数是 $(2 + 3) / 2 = 2.5$

设计一个支持以下两种操作的数据结构：

- void addNum(int num) - 从数据流中添加一个整数到数据结构中。
- double findMedian() - 返回目前所有元素的中位数。

示例

```
1 addNum(1)
2 addNum(2)
3 findMedian() -> 1.5
4 addNum(3)
5 findMedian() -> 2
```

方法一 堆

思路

生成两个堆：small和large。用small堆存放数据中较小的一半元素，large堆存放数据中较大的一半元素

详解

建立两个堆，这两个堆需要满足：

1. 大顶堆元素都比小顶堆小；
2. 大顶堆元素不小于小顶堆，且最多比小顶堆多一个元素

然后，我们观察可以发现，如果，数据总数是偶数，那么大顶堆，和小顶堆，一边占一半元素，而且，还是有序的，很像二分法，这时，中位数为两堆顶平均值 如果数据个数为奇数，则，中位数出现在元素个数多的堆的堆顶中

代码

```
1 const MedianFinder = function () {
2     this.maxHeap = [];
3     this.minHeap = [];
4 };
5 function minHeapify () {
6     this.minHeap.unshift(null);
7     const a = this.minHeap;
8     for (let i = a.length - 1; i >> 1 > 0; i--) {
9         // 自下往上堆化
10        if (a[i] < a[i >> 1]) {
11            // 如果子元素更小，则交换位置
12            const temp = a[i];
13            this.minHeap[i] = a[i >> 1];
14            this.minHeap[i >> 1] = temp;
15        }
16    }
17    this.minHeap.shift(null);
18 }
19
20 function maxHeapify () {
21     this.maxHeap.unshift(null);
22     const a = this.maxHeap;
23     for (let i = a.length - 1; i >> 1 > 0; i--) {
24         // 自下往上堆化
25         if (a[i] > a[i >> 1]) {
26             // 如果子元素更大，则交换位置
27             const temp = a[i];
28             this.maxHeap[i] = a[i >> 1];
29             this.maxHeap[i >> 1] = temp;
30         }
31     }
32     this.maxHeap.shift(null);
33 }
34
35 MedianFinder.prototype.addNum = function (num) {
36     // 插入
37     if (num >= (this.minHeap[0] || Number.MIN_VALUE)) {
38         this.minHeap.push(num);
39     } else {
40         this.maxHeap.push(num);
41     }
42     // 使得大顶堆的数量最多大于小顶堆一个，且一定不小于小顶堆数量
43     if (this.maxHeap.length > this.minHeap.length + 1) {
44         // 大顶堆的堆顶元素移动到小顶堆
45         this.minHeap.push(this.maxHeap.shift());
46     }
47     if (this.minHeap.length > this.maxHeap.length + 1) {
48         this.maxHeap.push(this.minHeap.shift());
49     }
50 }
```

```

46    }
47
48    if (this.minHeap.length > this.maxHeap.length) {
49        // 小顶堆的堆顶元素移动到大顶堆
50        this.maxHeap.push(this.minHeap.shift());
51    }
52
53    // 调整堆顶元素
54    if (this.maxHeap[0] > this.minHeap[0]) {
55        const temp = this.maxHeap[0];
56        this.maxHeap[0] = this.minHeap[0];
57        this.minHeap[0] = temp;
58    }
59
60    // 堆化
61    maxHeapify.call(this);
62    minHeapify.call(this);
63 };
64
65 MedianFinder.prototype.findMedian = function () {
66     if ((this.maxHeap.length + this.minHeap.length) % 2 === 0) {
67         return (this.minHeap[0] + this.maxHeap[0]) / 2;
68     } else {
69         return this.maxHeap[0];
70     }
71 };

```

复杂度分析

- 时间复杂度： $O(n)$
- 空间复杂度： $O(n)$

方法二 二分查找插入法

思路

JS中操作数组非常方便，我们可以使用Array.splice这样的方法向数组的中间进行数据添加和删除。因此在JS中，二分法则成为了一个不错的选择

详解

难点在于每次插入后，保证数列按顺序排列，每次插入时可用二分法找出插入位置 1. 使用二分查找找出插入位置 2. 维护一个数组永远保持有序

代码

```

1  var MedianFinder = function() {
2      this.num = []
3  };
4
5  function binarySearch(a, target) {
6      target += 1;
7      var start = 0
8      var end = a.length - 1;
9
10     while(start <= end) {
11         var mid = ~~((start + end) >> 1);
12         if (a[mid] >= target)
13             end = mid - 1;
14         else
15             start = mid + 1;
16     }
17
18     return start;
19 }
20
21 MedianFinder.prototype.addNum = function(num) {
22     var index = binarySearch(this.num, num);
23     this.num.splice(index, 0, num);
24 };
25
26
27 MedianFinder.prototype.findMedian = function() {
28     var len = this.num.length;
29     if (len & 1)
30         return this.num[~~(len / 2)];
31     else return (this.num[len / 2] + this.num[len / 2 - 1]) / 2;
32 };

```

复杂度分析

- 时间复杂度： $O(n)$

上述解法中用二分查找将耗费 $O(\log n)$ 时间，插入时将耗费 $O(n)$ 的时间，所以时间复杂度为 $O(n)$

- 空间复杂度： $O(n)$

逆波兰表达式求值

根据[逆波兰表示法]

(<https://baike.baidu.com/item/%E9%80%86%E6%B3%A2%E5%85%B0%E5%BC%8F/128437>)，求表达式的值。

有效的运算符包括 `+, -, *, /`。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

说明：

整数除法只保留整数部分。给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

示例 1：

```
1 输入: ["2", "1", "+", "3", "*"]
2 输出: 9
3 解释: ((2 + 1) * 3) = 9
```

示例 2：

```
1 输入: ["4", "13", "5", "/", "+"]
2 输出: 6
3 解释: (4 + (13 / 5)) = 6
```

示例 3：

```
1 输入: ["10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"]
2 输出: 22
3 解释:
4   ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
5   = ((10 * (6 / (12 * -11))) + 17) + 5
6   = ((10 * (6 / -132)) + 17) + 5
7   = ((10 * 0) + 17) + 5
8   = (0 + 17) + 5
9   = 17 + 5
10  = 22
```

方法一 入栈出栈法

思路

根据示例中的逆波兰表达式转中序表达式的过程可知，`+`，`-`，`*`，`/` 运算符的两个操作数正好是运算符前两位，即逆波兰表达式 `ab+` 的运算符为 `+`、操作数分别为 `a` 和 `b`，转中序表达式就是 `a + b`；根据这个规则，顺序遍历表达式中各项值，遇到操作数时，操作数入栈，遇到运算符时，栈顶两个操作数分别出栈，并应用运算符完成两个操作数的计算；

详解

1. 定义 `*`，`+`，`-`，`/` 四个运算符与对应操作的映射关系 `ACTIONS`。
2. 定义 `stack` 数组，用于数据出栈入栈。
3. 遍历 `tokens` 数组，如遇到运算符时，依次弹出栈顶两个元素，并作为参数调用运算符对应的操作，将操作结果入栈。
4. 如遇上非运算符，则解析成 `Int` 类型压栈。
5. 返回 `stack` 数组栈顶元素作为逆波兰表达式结果。

代码

```
1  /**
2   * @param {string[]} tokens
3   * @return {number}
4  */
5  const ACTIONS = {
6    '*' : (a, b) => b * a,
7    '+' : (a, b) => b + a,
8    '-' : (a, b) => b - a,
9    '/' : (a, b) => parseInt(b / a)
10 };
11
12 const evalRPN = function (tokens) {
13   const stack = [];
14   tokens.forEach(token => {
15     const action = ACTIONS[token];
16     if (action) {
17       stack.push(action(stack.pop(), stack.pop()));
18     } else {
19       stack.push(parseInt(token));
20     }
21   });
22   return stack.shift();
23 };
```

复杂度分析

- 时间复杂度： $O(n)$

需要遍历逆波兰表达式数组，查找操作数及运算符，这将耗费 $O(n)$ 的时间。

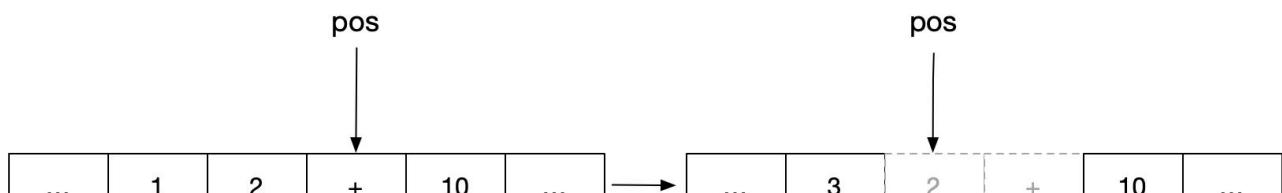
- 空间复杂度： $O(n)$

遍历逆波兰表达式数组时遇到操作数时，将操作数入栈，这将耗费 $O(n)$ 的栈空间。

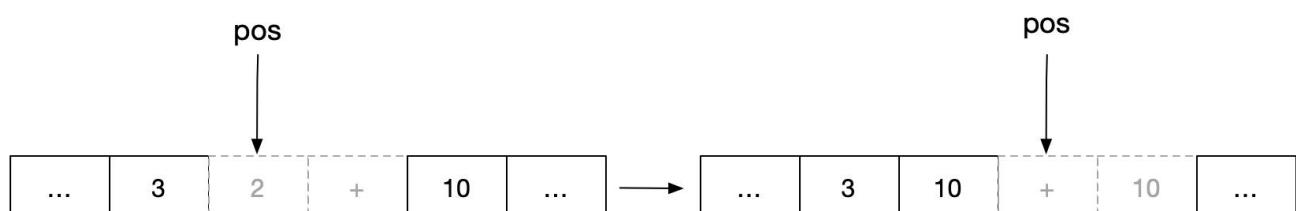
方法二 复用逆波兰表达式数组

思路

方法一是新开辟栈空间，在遍历时存储操作数，根据逆波兰表达式计算规则，运算符前两个地址空间分别为两个操作数，计算之后，两个操作数与运算符三个地址空间是可以复用存储计算结果的；



(1) 计算结果复用操作数空间



(2) 操作数前移

详解

1. 定义 `*`, `+`, `-`, `/` 四个运算符与对应操作的映射关系 ACTIONS。
2. 定义 pos 变量，用于保存指示当前操作数索引。
3. 遍历 tokens 数组，如遇到运算符时，依次取出索引位置为 pos-1 及 pos-2 两个元素，并作为参数调用运算符对应的操作。
4. 将操作结果存入数组 pos-2 索引空间，并且 pos 减 1，表示左移到上一个数组索引。
5. 如遇上非运算符，则解析成 Int 类型存入数组 pos 索引空间，并且 pos 加 1，表示右移到下一个索引。
6. 返回 tokens 数组 0 索引位置元素作为逆波兰表达式结果。

代码

```

1  /**
2   * @param {string[]} tokens
3   * @return {number}
4  */
5  const ACTIONS = {
6    '*' : (a, b) => b * a,
7    '+' : (a, b) => b + a,
8    '-' : (a, b) => b - a,
9    '/' : (a, b) => parseInt(b / a)
10 };
11
12 const evalRPN = function (tokens) {
13   let pos = 0;
14   tokens.forEach((token, index) => {
15     const action = ACTIONS[token];
16     if (action) {
17       tokens[pos - 2] = action(
18         parseInt(tokens[pos - 1]),
19         parseInt(tokens[pos - 2]))
20     );
21     pos = pos - 1;
22   } else {
23     if (pos !== index) {
24       tokens[pos] = tokens[index];
25     }
26     pos++;
27   }
28 });
29 return tokens[0];
30 };

```

复杂度分析：

- 时间复杂度： $O(n)$

需要遍历逆波兰表达式数组，查找操作数及运算符，这将耗费 $O(n)$ 的时间。

- 空间复杂度： $O(1)$

只需 1 个元素存储复用空间的地址索引 pos。

Task Scheduler、有序矩阵中第K小的元素和多数元素

Task Scheduler

给定一个用字符数组表示的 CPU 需要执行的任务列表。其中包含使用大写的 A - Z 字母表示的 26 种不同种类的任务。任务可以以任意顺序执行，并且每个任务都可以在 1 个单位时间内执行完。CPU 在任何一个单位时间内都可以执行一个任务，或者在待命状态。

然而，两个相同种类的任务之间必须有长度为 n 的冷却时间，因此至少有连续 n 个单位时间内 CPU 在执行不同的任务，或者在待命状态。

你需要计算完成所有任务所需要的最短时间。

示例

```
1 输入: tasks = ["A", "A", "A", "B", "B", "B"], n = 2
2 输出: 8
3 执行顺序: A -> B -> (待命) -> A -> B -> (待命) -> A -> B.
```

方法一 贪心选择法

思路

每一轮选择尽可能多的不超过 $n + 1$ 个任务执行，直到所有的任务被执行完。

详解

1. 对任务按照剩余的次数进行降序排序，从剩余次数最多的任务开始依次来执行
2. 每次判断任务是否可以执行，如果可以把对剩余的任务数量 -1
3. 一轮跑完之后，要重新对任务进行排列，保证任务的排序是降序

```
1 const leastInterval = function (tasks, n) {
2   const counts = Array(26).fill(0);
3   tasks.forEach(v => {
4     const index = v.charCodeAt() - 65; // 'A'.charCodeAt() 为 65
5     counts[index]++;
6   });
7 }
```

```

7   counts.sort((a, b) => (b - a)); // 直接从大到小排
8   let res = 0;
9   while (counts[0] > 0) {
10     let i = 0;
11     while (i < n + 1) { // 保证这一轮任务距离上次执行至少冷却 n 个单位的时间
12       if (counts[0] === 0) { // 当最大当任务数为 0，跳出循环
13         break;
14       }
15       if (i < 26 && counts[i] > 0) { // 判断任务是否可以执行
16         counts[i]--;
17       }
18       res++;
19       i++;
20     }
21     counts.sort((a, b) => (b - a)); // 每次任务执行完，保证最多的任务排在最前面
22   }
23   return res;
24 };

```

复杂度分析

- 时间复杂度： $O(\text{time})$

通过 `res++` 我们发现，最终的时间为多少，时间复杂度就是多少，这种解法的缺点在于，得出相当于是把每个任务执行了一遍

- 空间复杂度： $O(1)$

与 n 无关，申请数组的长度最大为26，复杂度为 $O(1)$

方法二 桶思想

思路

换一种形象的比喻，把间隔时间必做桶。

每个桶固定大小为 $n + 1$ ，最后一个除外，把相同的任务分在不同的桶中，那么桶的数量就由数量最多的任务的数量来决定

可以得到结果为 $(n + 1) * (\text{count} - 1) + \text{最后一个桶的任务数}$ ，至于说最后一个桶对任务数是多少，通过找规律发现，即为有多少个最多任务数

比如 `["A", "A", "A", "B", "B", "B"]`，即为 2，因为 A 和 B 的数量最多且一样

还有一种情况，当没有出现等待时，比如 `["A", "A", "B", "C", "D"]`，间隔 $n = 2$ 的时候，此时最长时间就是任务的长度明显是 4

但是发现上面的公式得出的值为1，不对了，此时我们只需要取两个数的最大值就可以了

详解

- 1、将任务按类型分组，把字母转为数字做数组的索引存在数组里
- 2、对数组进行排序，找出上面公式的 count 和最后一个桶的任务数
- 3、最后套公式就可以得出答案了

```
1 const leastInterval = function (tasks, n) {  
2   const counts = Array(26).fill(0);  
3   tasks.forEach(v => {  
4     const index = v.charCodeAt() - 65; // 'A'.charCodeAt() 为 65  
5     counts[index]++;  
6   });  
7   counts.sort((a, b) => (b - a)); // 直接从大到小排  
8   const max = counts[0];  
9   const maxCount = counts.filter(a => a === max).length; // 多少个最多任务数  
10  const res = (n + 1) * (max - 1) + maxCount;  
11  return res > tasks.length ? res : tasks.length;  
12};
```

复杂度分析：

- 时间复杂度： $O(n)$
 n 为任务的总数
- 空间复杂度： $O(1)$
与 n 无关，申请数组的长度最大为26，复杂度为 $O(1)$

有序矩阵中第K小的元素 □□

给定一个 $n \times n$ 矩阵，其中每行和每列元素均按升序排序，找到矩阵中第k小的元素。请注意，它是排序后的第k小元素，而不是第k个元素。

示例

```
1 matrix = [  
2   [ 1, 5, 9],  
3   [10, 11, 13],
```

```
4      [12, 13, 15]
5  ],
6  k = 8,
7
8 返回 13。
```

说明 你可以假设 k 的值永远是有效的, $1 \leq k \leq n^2$ 。

方法一

思路

将二维数组展开为一维升序数组，取第 $k-1$ 位。

详解

1. 将数组使用reduce展开为一维数组。
2. 使用数组的sort方法升序排列，注意这里是 `.sort((a, b) => a - b)`。
3. 取第 $k-1$ 个元素。

代码

```
1 /**
2  * @param {number[][]} matrix
3  * @param {number} k
4  * @return {number}
5 */
6 const kthSmallest = function (matrix, k) {
7   const flat = matrix.reduce((acc, item) => {
8     return acc.concat(item);
9   }, []).sort((a, b) => a - b);
10
11   return flat[k - 1];
12 };
```

复杂度分析

- 时间复杂度： $O(n^2)$
reduce 执行 n 次，时间复杂度为 $O(n)$ ，sort 排序 $O(n^2)$ ，所以时间复杂度取最大值 $O(n^2)$ 。
- 空间复杂度： $O(n^2)$

flat 的长度是 $n \times n$ ，所以空间复杂度为 $O(n^2)$ 。

方法二

思路

1. 找出二维矩阵中最小的数left，最大的数right，那么第k小的数必定在left~right之间。
2. $mid = (left+right) / 2$ ；在二维矩阵中寻找小于等于mid的元素个数count。
3. 若这个count小于k，表明第k小的数在右半部分且不包含mid，即left=mid+1, right=right，又保证了第k小的数在left~right之间。
4. 若这个count大于k，表明第k小的数在左半部分且可能包含mid，即left=left, right=mid，又保证了第k小的数在left~right之间。
5. 因为每次循环中都保证了第k小的数在left~right之间，当left==right时，第k小的数即被找出，等于right。

注意：这里的left mid right是数值，不是索引位置。

详解

1. 获取行数、列数，再取左上角的最小数、右下角最大数。
2. while循环判断left 小于 right，定义 $mid = \text{Math.floor}((left+right) / 2)$ ；在二维矩阵中寻找小于等于mid的元素个数count。
3. 若这个count小于k，left=mid+1, right=right。
4. 若这个count大于k，即left=left, right=mid。
5. 返回right。

代码

```
1 const kthSmallest = function (matrix, k) {
2     const row = matrix.length;
3     const col = matrix[0].length;
4     let left = matrix[0][0];
5     let right = matrix[row - 1][col - 1];
6
7     while (left < right) {
8         const mid = Math.floor((left + right) / 2);
9         const count = findNotBiggerThanMid(matrix, mid, row, col);
10        if (count < k) {
11            left = mid + 1;
12        } else {
13            right = mid;
14        }
15    }
}
```

```
16
17     return right;
18 }
19
20 function findNotBiggerThanMid (matrix, mid, row, col) {
21     let i = row - 1;
22     let j = 0;
23     let count = 0;
24
25     while (i >= 0 && j < col) {
26         if (matrix[i][j] <= mid) {
27             count += i + 1;
28             j++;
29         } else {
30             i--;
31         }
32     }
33
34     return count;
35 }
```

复杂度分析

- 时间复杂度： $O(n^2)$

两层循环，每个最多执行 n 次，时间复杂度是平方阶 $O(n^2)$ 。

- 空间复杂度： $O(1)$

虽然有两个循环，但没有分派多余的存储空间，所以空间复杂度为 $O(1)$ 。

多数元素

给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数大于 $\lfloor n/2 \rfloor$ 的元素。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

示例 1：

```
1    输入: [3,2,3]
2    输出: 3
```

示例 2：

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

题目分析

- $\lfloor n/2 \rfloor$ 意味着不大于 $n/2$ 的最大正整数
 - 可以通过 `Math.floor()` 或 `parseInt()` 取得
- 最直接的想法是，遍历数组中的每个元素，如果该元素在数组中出现的次数大于 $\lfloor n/2 \rfloor$ ，则将它输出。依据题目可知，在输入数组中，有且只会有 1 种数字符合输出条件，即输出的值只有一个。该解法的重点是，如何找到该元素在数组中出现的次数。如果直接继续用循环或 `filter` 方法，虽然可行，但是会因时间复杂度导致运行超时。因此，需要找到合理的解决办法。

方法一 统计数量遍历求解

思路

既然两层嵌套循环会运行超时，那无脑的想法便是将循环拆开处理。第一步，先遍历数组统计各元素出现的次数，并存入对象中；第二步，找到该对象中符合题目条件的元素。话不多说，简单粗暴的方法先来一个。

详解

- 1、创建变量 `number` 和 `statistics`，分别用于存储结果元素和元素统计结果
- 2、找到“多数元素”的边界值
- 3、遍历并统计给定数组中各元素的出现次数，存入 `statistics`
- 4、在 `statistics` 中找到符合边界值要求的元素
- 5、`return` 该元素

代码

```
1  /**
2   * @param {number[]} nums
3   * @return {number}
4  */
```

```

5  const majorityElement = function (nums) {
6    let number = 0;
7    // statistics 元素统计结果
8    const statistics = {};
9    if (nums.length) {
10      // 找到题目中要求的“多数元素”的边界值
11      const bound = Math.floor(nums.length / 2);
12      // 统计每个元素出现的次数
13      nums.map((num) => {
14        statistics[num] = statistics[num] ? statistics[num] + 1 : 1;
15      });
16      // 找到 statistics 中符合条件的元素
17      Object.keys(statistics).map((key) => {
18        if (statistics[key] > bound) {
19          number = key;
20        }
21      });
22    }
23    return number;
24  };

```

复杂度分析

- 时间复杂度： $O(n)$

该解法中，两个map循环遍历总次数最多为 $2n$ ，因此，时间复杂度为 $O(n)$ 。

- 空间复杂度： $O(n)$

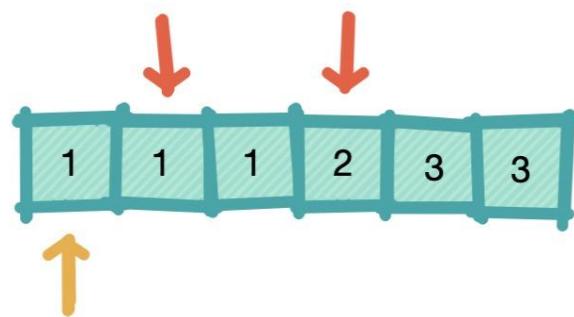
该解法中，考虑到众数的频率超过 $n/2$ ，因此，`statistics` 的占用空间最多 $n/2$ ，空间复杂度为 $O(n)$ 。

方法二 双指针求解

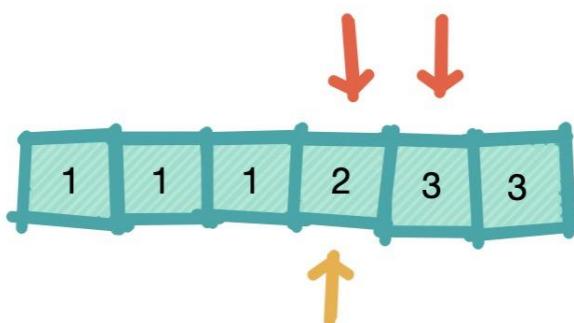
思路

假设输入的数组为 `nums=[3,1,1,2,1,3]`，我们可以考虑将其排序后，再利用两个指针查找目标值。排序后的数组如图所示，初始化时两个指针 i 和 j 分别指向数组的第一个元素（黄色箭头位置）和第二个元素（（左）红色指针位置）。

- 黄色指针 i 不动，红色指针 j 每次向右移动一个格子，直到当前元素和上一元素不同时暂停，即当前停在下标为 3 的位置上。
- 此时可知元素 1 出现的次数为 $j - i$ 次。比较该元素出现次数是否满足题目元素，若满足，则将其输出，若不满足则继续执行下一步。



- 将黄色指针 i 移动到当前指针 j 的位置，如图所示。之后，继续将红色指针 j 向右移动。



- 重复上述步骤，直至找到符合条件的元素

详解

1、创建变量 `number`，用于存储结果元素

2、对给定数组进行排序

3、找到“多数元素”的边界值

4、初始化两个指针并依据上文提到的思路进行遍历，找到出现次数大于边界值的元素

5、`return` 该元素

其中需考虑特殊情况，如 `number` 的初始值以及遍历中特殊情况的处理，详见代码。

代码

```

1  /**
2   * @param {number[]} nums
3   * @return {number}

```

```

4  */
5 const majorityElement = (nums) => {
6   let number = 0;
7   // 数组排序
8   nums.sort();
9   // 为 number 赋初始值
10  number = nums[0];
11  if (nums.length > 1) {
12    // 找到题目中要求的“多数元素”的边界值
13    const bound = Math.floor(nums.length / 2);
14    for (let i = 0, j = 1; j < nums.length; j += 1) {
15      if (nums[i] !== nums[j]) {
16        // 找到元素 i 及其出现次数
17        if (j - i > bound) {
18          number = nums[i];
19        } else {
20          // 若不满足条件则移动 i 指针至当前 j 指针的位置
21          i = j;
22        }
23      }
24      // 若指针 j 已经移动到最后一个元素位置时，仍未找到满足条件的元素，则当前元素为最终结果
25      if (j === nums.length - 1) {
26        number = nums[i];
27      }
28    }
29  }
30  return number;
31 };

```

复杂度分析

- 时间复杂度： $O(n \log n)$

上述解法中，使用了 `sort()` 方法，该方法在底层实现上依不同浏览器而有所差异。在 Chrome 浏览器中，使用了插入排序和快速排序结合的算法，具体情况与待排序数组的长度有关，一般情况下长度小于等于 10 的数组使用插入排序（可参考：[深入浅出 JavaScript 的 Array.prototype.sort 排序算法](#)）。假设此处考虑 Chrome 浏览器下使用快排作为排序算法的情况，则 `sort()` 耗时为 $O(n \log n)$ 。排序后 `for` 循环耗时 $O(n)$ 。因此，时间复杂度可记为 $O(n \log n)$ ，其最差可能为 $O(n^2)$ 。

- 空间复杂度： $O(\log n)$

该解法中，自主申请的变量为常量级别 $O(1)$ 。`sort()` 使用插入排序时空间复杂度为 $O(1)$ ，使用快排时为 $O(\log n)$ 。因此空间复杂度最好为 $O(1)$ ，最差为 $O(\log n)$ 。（此分析仅以 Chrome 浏览器为例，其余浏览器的情况可具体分析）

方法三 Boyer-Moore Vote Algorithm

思路

摩尔投票算法（多数投票算法），该算法由 Robert S.Boyer 和 J Strother Moore 发明，它可以高效的计算出序列中哪个元素占多数。

同样利用两个指针，初始时分别位于第一个元素和第二个元素位置。假定 `count = 1`，数组的每一轮遍历中，判断两个指针所指元素是否相等，若相等则 `count++`，否则 `count--`。若此时 `count === 0`，意味着相邻两元素值不相同，需将两指针分别向前移动一位并将 `count` 还原为初始值。

详解

1、初始化指针 `majorityIndex` 和 `count` 的值分别为 `0` 和 `1`

2、遍历给定数组，判断相邻两元素是否相等并依据判断结果修改 `count` 的值。若此时 `count` 为 `0`，则将 `majorityIndex` 指针移动到当前 `i` 指针的位置，并重新将 `count` 设置为 `1`

3、`return` 给定数组中下标为 `majorityIndex` 的元素

代码

```
1  /**
2   * @param {number[]} nums
3   * @return {number}
4  */
5 const majorityElement = (nums) => {
6   // 指针，指向符合输出条件的元素
7   let majorityIndex = 0;
8   // 计数器
9   let count = 1;
10  for (let i = 1; i < nums.length; i += 1) {
11    // 判断相邻两元素是否相等
12    nums[majorityIndex] === nums[i] ? count += 1 : count -= 1;
13    if (count === 0) {
14      majorityIndex = i;
15      count = 1;
16    }
17  }
18  return nums[majorityIndex];
19};
```

复杂度分析

- 时间复杂度： $O(n)$

该解法中，只需要遍历一次数组，其时间复杂度为 $O(n)$ 。

- 空间复杂度： $O(1)$

该解法中，仅申请了 2 个常量级的额外空间，因此空间复杂度为 $O(1)$ 。

结束篇

看到这里，小册子的内容也就基本结束了。首页非常感谢各位同学购买这本小册子。这是我们团队的第一本小册子，内容可能会存在一些瑕疵，在此接受大家的指正。

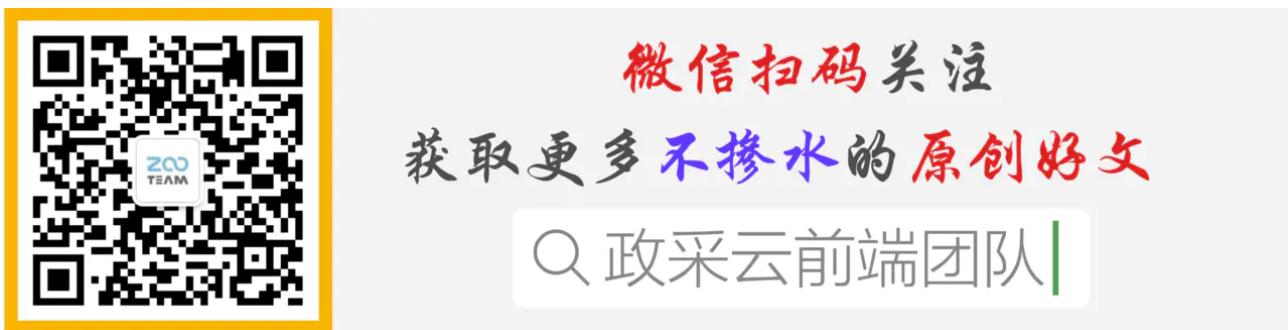
解决同样一个问题，往往可以有多种解法。在掌握了算法之后，就可以知道不同的算法在相应的场景下，孰优孰略。打个比方来说，排序算法有很多种，在数据量大的情况下，使用快速排序就会比冒泡排序高效很多。在数组中找到一个元素很简单，但是在一些场景下，二分查找的效率比线性查找的效率不知道高到哪里去了。所以说掌握了算法之后，你写的代码一定跑得比谁都快。

不忘初心

有的小伙伴可能接触算法比较晚，刚开始的过程多半是虐心的，这很正常。以至于看到别人的解法会叹为观止。网上的很多解法很多，让人眼花缭乱，但大多数讲得太散乱，对于初学者来说，解法缺少引导，让读者从看懂到自己真正会写这道题目还有很多的路要走，也是我们为什么要写这本小册子的原因，希望可以帮助到大家。

最后

在小册子的最后，放一下我们团队的微信公众号二维码，每周会推送前端及周边的热点、知识点，也欢迎大家来交流。



祝大家工作顺利，在变得更好的路上不断前进！