**let arr=read\_line().split(" ")**

**let n=arr[0]**

**let arr1=[]**

**while(n--){**

**arr1.push(read\_line()) //带有空格的字符串**

**}**

**console.log(arr1[0].split(" ")[1],arr1[2].split(" ")[2])**

**动态规划总结：**

1. **斐波那契数：**该数列由 0 和 1 开始，后面的每一项数字都是前面两项数字的和。

Dp[i]为第i个值，初始化dp[0]=0,dp[1]=1,dp[i]=dp[i-1]+dp[i-2]

1. **爬楼梯 :**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？

Dp[i]为几种方式  **dp[i]=dp[i-1]+dp[i-2]**

初始化：

第一节有一种方法dp[1]=1

第二阶有两种方法：dp[2]=2

**完全背包的解法：**

求几种方法，组合dp[j]+=dp[j-n[i]] 需要考虑顺序，先遍历背包（需要判断是否装得下）

创建物品列表nums=[1,2]

**746. 使用最小花费爬楼梯:**数组的每个下标作为一个阶梯，第 i 个阶梯对应着一个非负数的体力花费值 cost[i]（**下标从 0** 开始）。每当你爬上一个阶梯你都要花费对应的体力值，一旦支付了相应的体力值，你就可以选择向上爬一个阶梯或者爬两个阶梯。你可以选择从下标为 0 或 1 的元素作为初始阶梯。

请你找出达到楼层顶部的最低花费。

Dp[i]为花费的最少体力  **dp[i]=min(dp[i-1],dp[i-2])+n[i]**  （加上当前体力值，在最后一阶不需要支付体力值）dp[len]=min(dp[len-1],dp[len-2])

初始化：dp[0]=n[0] dp[1]=n[0]

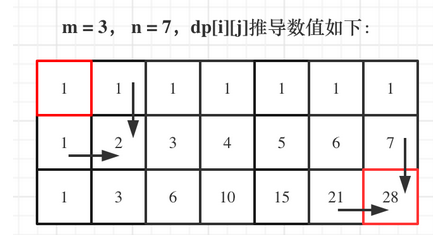
**62.不同路径 :**一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为 “Start” ）。机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为 “Finish” ）。

问总共有多少条不同的路径？

Dp[i][j]为不同的路径 **dp[i][j]=dp[i-1][j]+dp[i][j-1]**

初始化：i和j直行的路径数

Dp[i][0]=1 dp[0][j]=1



1. **不同路径 II :**考虑网格中有障碍物。那么从左上角到右下角将会有多少条不同的路径？

如果遇到障碍，则之后的路径为0

Dp[i][j]为不同的路径 dp[i][j]=dp[i-1][j]+dp[i][j-1]

初始化：（与上题不同）

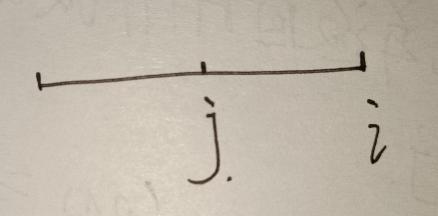
For(let i=1;i<m **&& ob[i][0]===0**;i++)

1. **整数拆分 :**给定一个正整数 n，将其拆分为至少两个正整数的和，并使这些整数的乘积最大化。 返回你可以获得的最大乘积。

数值为i，Dp[i]为最大乘积

**For(i=4,i<=n)**

**For(j=1;j<i)**



判断i不拆：,dp[i]

拆：j不拆：j\*(i-j) j拆：dp[j]\*(i-j)

（再判断j的部分是拆，还是不拆最大化）

**dp[i]=max(j\*(i-j),dp[j]\*(i-j),dp[i])**

初始化：

dp[2]=1 dp[3]=2

1. **不同的二叉搜索树 ：**给定一个整数 n，求以 1 ... n 为节点组成的二叉搜索树有多少种？

i个节点，其中j为头节点，左子树的种类：dp[j-1] 右子树的种类：dp[i-j]

初始化：dp[0]=1

递归公式：**dp[i] += dp[以j为头结点左子树节点数量] \* dp[以j为头结点右子树节点数量]**

1. **分割等和子集 :**给定一个只包含正整数的非空数组。是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

背包容量为sum/2,每个元素只取一次（01背包：取或不取）

滚动数组的遍历：外层遍历物品，内层遍历背包（剩余容量）

**dp[j]表示 背包总容量是j，最大可以凑成j的子集总和为dp[j]**。

**Dp[j]=max(dp[j],dp[j-nums[i]]+w[i]) //背包容量减少，背包重量增大**

**1049. 最后一块石头的重量 II ：**有一堆石头，每块石头的重量都是正整数。

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

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

**分成两份的粉末，返回分成两份的差值** return (sum-dp[n])-dp[n]

**Dp[j]=max(dp[j],dp[j-nums[i]]+w[i])**

**494. 目标和 ：**给定一个非负整数数组，a1, a2, ..., an, 和一个目标数，S。现在你有两个符号 + 和 -。对于数组中的任意一个整数，你都可以从 + 或 -中选择一个符号添加在前面。

返回可以使最终数组和为目标数 S 的所有添加符号的方法数。

正数的总和-（sum-正数的总和）=目标数 负数的总和=sum-正数的总和

正总和=（sum+s）/2

所以要求正数的总和（的组合数）

**求组合（每增加一个元素，需要叠加之前的状态：dp[j]+=dp[j-w]）**

**累加需要初始化dp[0]=1**

**474.一和零 :**给你一个二进制字符串数组 strs 和两个整数 m 和 n 。

请你找出并返回 strs 的最大子集的大小，该子集中 最多 有 m 个 0 和 n 个 1 。

如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的 子集 。

示例 1：

输入：strs = ["10", "0001", "111001", "1", "0"], m = 5, n = 3 输出：4

**01背包问题，装哪个价值最大化（取还是不取）（两个背包）**

**Dp[i]为最多的字串数**

**取：Dp[i][j]=dp[i-o][j-l]+1 不取（个数保持）：dp[i][j]=dp[i][j]**

1. **零钱兑换 II：**给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

无限个，完全背包（区别在于遍历，背包容量由小到大，初始化为w[i]）

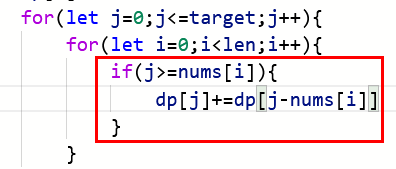
**求组合数：dp[i]+=dp[i-w[i]] 累加需要初始化dp[0]=1**

**求个数（要还是不要）：dp[j]=max(dp[j],dp[j-w[i]]+1)**

1. **组合总和 Ⅳ ：**给定一个由正整数组成且不存在重复数字的数组，找出和为给定目标正整数的组合的个数。顺序不同的序列被视作不同的组合。

**需要注意顺序，则先遍历背包(背包初始化为0，则每次加入需要判断是否装得下)**

Dp为组合数 dp[j]+=dp[j-w[i]]



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。

**求个数；dp[j]=min(dp[j],dp[j-n[i]]+1)**

首先凑足总金额为0所需钱币的个数一定是0，那么dp[0] = 0;

其他下标对应的数值呢？

考虑到递推公式的特性，dp[j]必须初始化为一个最大的数，否则就会在min(dp[j - coins[i]] + 1, dp[j])比较的过程中被初始值覆盖。

**所以下标非0的元素都是应该是最大值。**

**279.完全平方数：**给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。你需要让组成和的完全平方数的**个数最少。**

给你一个整数 n ，返回和为 n 的完全平方数的 最**少数量** 。

**求个数；dp[j]=min(dp[j],dp[j-n[i]]+1)**

**求最小数，初始值赋最大值Infinity**

**求最大数，初始值赋值0**

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

背包最好从0开始遍历背包内的内容s，所以先遍历背包（**需要判断容量是否放得下）**

截取s当前字段判断是否等于字典中的某个字符**(s.slice(j-w[i].length,j)==w[i])**

且前一个字符串为true，当前总的字符串才能为true

**dp[j-w[i].length]==true**

**198.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 不触动警报装置的情况下 ，一夜之内能够偷窃到的**最高金额。**

**第i间累积的金额：**

**当前房间不偷，延续前一间的金额 dp[i-1]**

**当前房间偷，前提是前一间不偷，dp[i-2]+n[i]**

初始化dp[0]和dp[1]

1. **打家劫舍II :**这道题目和[198.打家劫舍 (opens new window)](https://programmercarl.com/0198.%E6%89%93%E5%AE%B6%E5%8A%AB%E8%88%8D.html" \t "https://www.programmercarl.com/_blank)是差不多的，唯一区别就是成环了。

选头不选尾，或者选尾不选头

**Dd(0,len-2) dd(1,len-1) 需要排除len为0，1，2的情况**

1. **打家劫舍 III ：**除了“根”之外，每栋房子有且只有一个“父“房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。 如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

采用后序

//不偷(不管子节点偷不偷，取子节点最大的值)

let val1=Math.max(left[0],left[1])+Math.max(right[0],right[1])

//偷（取当前节点的值，和子节点不偷的值）

let val2=root.val+left[0]+right[0]

**121. 买卖股票的最佳时机 ：**给定一个数组 prices ，它的第 i 个元素 prices[i] 表示一支给定股票第 i 天的价格。

你只能选择 某一天 买入这只股票，并选择在 未来的某一个不同的日子 卖出该股票。设计一个算法来计算你所能获取的最大利润。

返回你可以从这笔交易中获取的**最大利润**。如果你不能获取任何利润，返回 0 。

**只买卖一次**

**定义两个状态，**

**买：今日买，或之前已买 ：dp[i][0]=Math.max(dp[i-1][0],-p[i])**

**卖: 今日卖，或之前已卖： dp[i][1]=Math.max(dp[i-1][0]+p[i],dp[i-1][1])**

**需要初始化第0天的买入和卖出的情况**

1. **买卖股票的最佳时机II ：可以多次买卖**

**定义两个状态，**

**买：今日买（前一天已经卖出），或之前已买 ：dp[i][0]=Math.max(dp[i-1][0],dp[i-1][1]-p[i])**

**卖: 今日卖(前一天已经买入)，或之前已卖： dp[i][1]=Math.max(dp[i-1][0]+p[i],dp[i-1][1])**

**需要初始化第0天的买入和卖出的情况**

1. **买卖股票的最佳时机III ：**最多可以完成两笔交易。

**定义五个状态：**

无操作：dp[i][0]=dp[i-1][0]

第一次买入：dp[i][1]=max(-p[i],dp[i-1][1])

第一次卖出：dp[i][2]=max(dp[i-1][1]+p[i],dp[i-1][2])

第二次买入：dp[i][3]=max(dp[i-1][2]-p[i],dp[i-1][3])

第二次卖出：dp[i][4]=max(dp[i-1][3]+p[i],dp[i-1][4])

初始化第0天的买入金额

dp[0][1]=-p[0]

dp[0][3]=-p[0]

返回最后一天的状态4

1. **买卖股票的最佳时机IV ：**最多可以完成 k 笔交易。

· 0 表示不操作

· 1 第一次买入

· 2 第一次卖出

· 3 第二次买入

· 4 第二次卖出 。。。。。

所以状态j的数量为 2\*k+1

**第二次买入：dp[i][3]=max(dp[i-1][2]-p[i],dp[i-1][3])**

**第二次卖出：dp[i][4]=max(dp[i-1][3]+p[i],dp[i-1][4])**

参照上一题的思路：定义两个状态：dp[i][j]

状态j为奇数表示买入，j为偶数表示卖出 (**j<2\*k;每次递增+2**)

买入：dp[i][j+1]=max(dp[i-1][j]-p[i]，dp[i-1][j+1])

卖出：dp[i][j+2]=max(dp[i-1][j+1]+p[i],dp[i-1][j+2])

初始化：第0天，k次买入 :  for(let **i=1;i<2\*k;i+=2**)

**309.最佳买卖股票时机含冷冻期 :**给定一个整数数组，其中第 i 个元素代表了第 i 天的股票价格 。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）:

* 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）。
* 卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。

定义四个状态：

0：买入 （之前已经买入，今天买入（前一天不是冷冻期，或者前一天是冷冻期））

dp[i][0]=max(dp[i-1][0],dp[i-1][1]-p[i],dp[i-1][3]-p[i])

1：保持卖出（前一天不是冷冻期，或者前一天是冷冻期）

dp[i][1]=max(dp[i-1][1],dp[i-1][3])

1. 卖出（今天卖出(之前已经买入)）

Dp[i][2]=dp[i-1][0]+p[i]

1. 冷冻期(前一天卖出)

Dp[i][3]=dp[i-1][2]

**714.买卖股票的最佳时机含手续费 ：**给定一个整数数组 prices，其中第 i 个元素代表了第 i 天的股票价格 ；非负整数 fee 代表了交易股票的手续费用。

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

返回获得利润的**最大值**。

**定义两个状态：**

**0持有：之前持有，或今天买入**

**Dp[i][0]=max(dp[i-1][0],dp[i-1][1]-p[i])**

**1不持有：之前不持有，今天卖出**

**Dp[i][1]=max(dp[i-1][1],dp[i-1][0]+p[i]-fee)**

需要注意p的长度为0，1，2的情况

1. **最长递增子序列 ：**给你一个整数数组 nums ，找到其中最长严格递增子序列的长度。

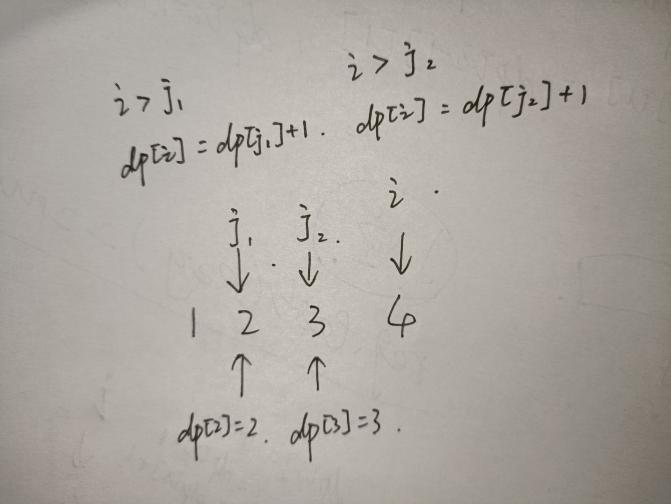
示例 1： 输入：nums = [10,9,2,5,3,7,101,18] 输出：4 解释：最长递增子序列是 [2,3,7,101]，因此长度为 4 。

**不连续的递增序列：**

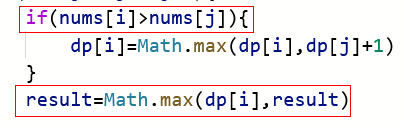
**dp[i]表示i之前包括i的最长上升子序列的长度**。

i与0到（i-1）的值比较，如果i比j大，就在dp[j](j的最长上升序列)加1，

然后取所有dp[i]的最大值



**注意：大于才加入，且要保存全局的最大值**



1. **最长连续递增序列 ：**给定一个未经排序的整数数组，找到最长且 **连续递增**的子序列，并返回该序列的长度。

**dp[i]表示i之前包括i的最长上升子序列的长度**。

**If(n[i]>n[i-1]) Dp[i]=dp[i-1]+1**

初始值为1，保存全局最大值

**718. 最长重复子数组 ：**给两个整数数组 A 和 B ，返回两个数组中公共的、长度最长的子数组的长度。

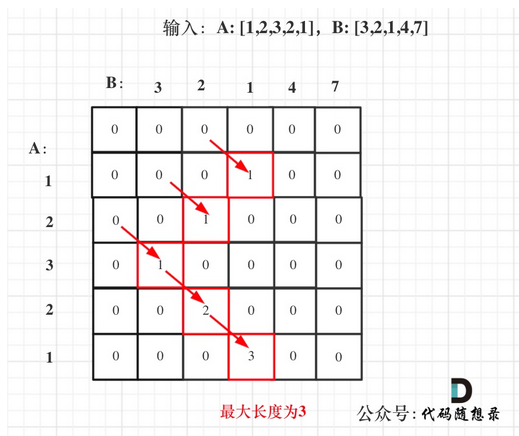
示例：输入： A: [1,2,3,2,1] B: [3,2,1,4,7] 输出：3 解释： 长度最长的公共子数组是 [3, 2, 1] 。

**两个数组，找连续相同**

dp[i][j] ：以下标i - 1为结尾的A，和以下标j - 1为结尾的B，最长重复子数组长度为dp[i][j]。

二维数组中，当前两个元素相同，且前一个元素也必须相同

**Dp[i][j]=dp[i-1][j-1]+1**



1. **最长公共子序列 :**给定两个字符串 text1 和 text2，返回这两个字符串的最长公共子序列的长度。

**两个数组，找不连续的相同字符串**

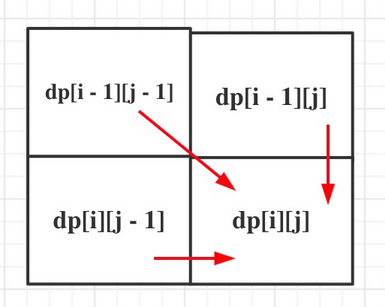
dp[i][j]：长度为[0, i - 1]的字符串text1与长度为[0, j - 1]的字符串text2的最长公共子序列为dp[i][j]

如果i-1与j-1相等，则直接相同的字符串数目加1

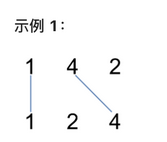
**Dp[i][j]=dp[i-1][j-1]+1**

如果不相等，则取[0,i-1]与[0,j-2]或[0,i-2]与[0,j-1]的最大公共序列

**Dp[i][j]=max(dp[i][j-1],dp[i-1][j])**



1. **不相交的线 :**

 与上一题一致，找不连续的相同字串

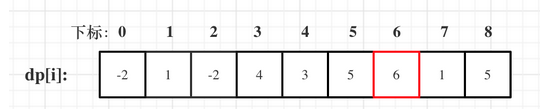
**53. 最大子序和 ：**给定一个整数数组 nums ，找到一个具有最大和的**连续子数组**（子数组最少包含一个元素），返回其最大和。

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

**dp[i]：包括下标i之前的最大连续子序列和为dp[i]**。

如果dp[i-1]为负值，则直接取当前值为开始序列：

**Dp[i]=max(dp[i-1]+n[i],n[i])**



初始化dp[0]=n[0]

**392.判断子序列 :**给定字符串 s 和 t ，判断 s 是否为 t (源字符串)的子序列。

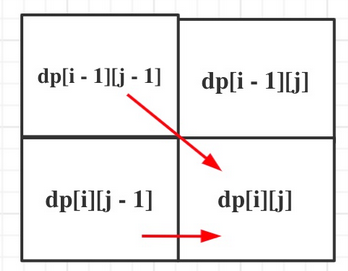
字符串的一个子序列是原始字符串删除一些（也可以不删除）字符而不改变剩余字符相对位置形成的新字符串。（例如，"ace"是"abcde"的一个子序列，而"aec"不是）。

**对比不同，其中一个字串跳过当前位置，继承上一个状态**

转换问题：找到连续的公共子序列长度，判断是否等于s的长度

如果相等，则dp[i][j]=dp[i-1][j-1]+1

如果当前t[j-1]不等于是s[i-1],则等于t串的前一个状态值，dp[i][j]=dp[i][j-1]

 横坐标为t源字符串（j）,纵坐标为s（i）

1. **不同的子序列 ：**给定一个字符串 s 和一个字符串 t ，计算在 s（源字符串）的子序列中 t 出现的**个数**。

dp[i][j]：以i-1为结尾的s子序列中出现以j-1为结尾的t的个数为dp[i][j]。

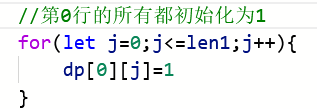
**对比不同，则取s前一个状态值：dp[i][j]=dp[i][j-1]**

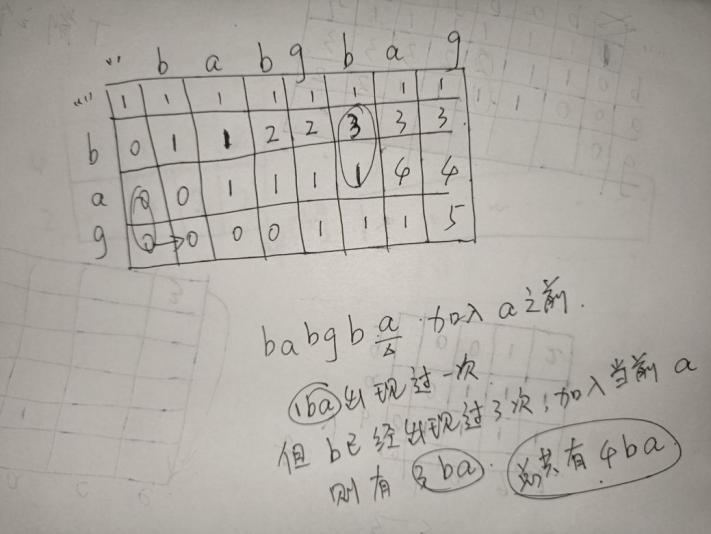
**对比相等，**

**取:dp[i]=dp[i-1][j-1]**

**不取当前s[j] :dp[i][j]=dp[i][j-1]**

初始化：t为空串的时候，s的任意子序列都有空串，所以i=0时初始化值为1





横坐标为源字符串s（j）,纵坐标为t(i）

1. **两个字符串的删除操作 ：**给定两个单词 word1 和 word2，找到使得 word1 和 word2 相同**所需的最小步数**，每步可以删除任意一个字符串中的一个字符。

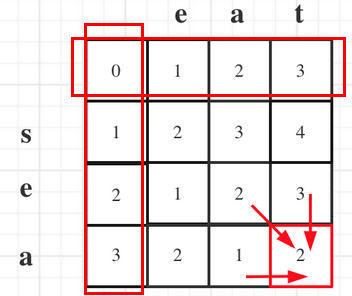
**相同的时候，dp[i][j]=dp[i-1][j-1]**

**不相同的时候：w1删，或者w2删，或者w1和w2一起删**

**Dp[i][j]=min(dp[i-1][j]+1,dp[i][j-1]+1,dp[i-1][j-1]+2)**

初始化：i=0时，表示w1为空，则w2需要删除多少次才能为空

j=0时，表示w2为空，则w1需要删除多少次才能为空



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

**相等：dp[i][j]=dp[i-1][j-1]**

**不相等：w1删除或w2删除，或其中一个字符串替换 (都是增加一步操作)**

**Dp[i][j]=min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1**

初始化如上题

**647. 回文子串 ：**给定一个字符串，你的任务是计算这个字符串中有多少个回文子串。具有不同开始位置或结束位置的子串，即使是由相同的字符组成，也会被视作不同的子串。**求回文子串的个数**

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

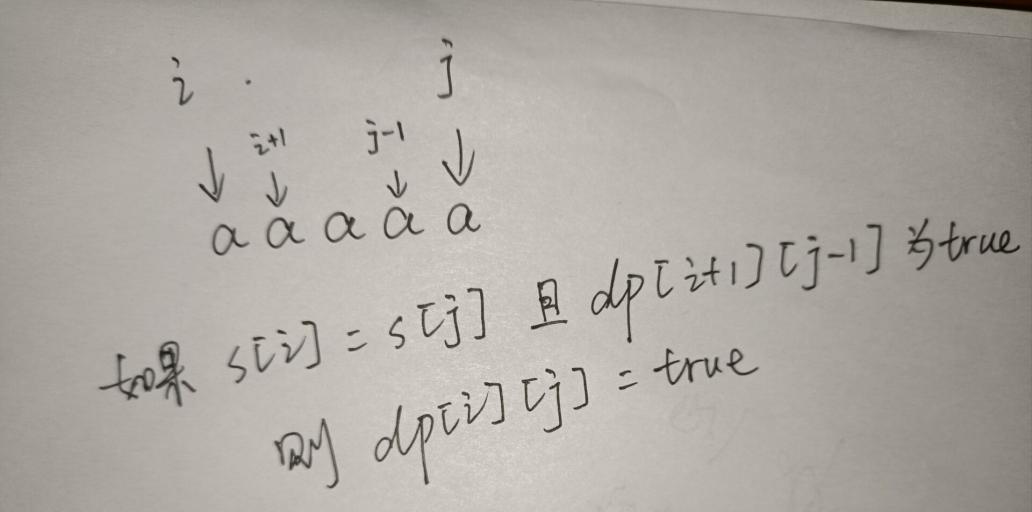
Dp[i][j]表示[i,j]是否为回文字串

如果s[i]不等于s[j],则一定不是回文字串 dp[i][j]=false

如果s[i]等于s[j]:

1,aa ;j-i=0 aba: j-i=1 都是回文字串

1. aaabaaa :j-i>1的时候需要考虑，[i+1,j-1]是回文，dp[i][j]才是true



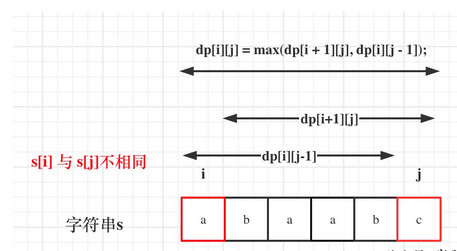
遍历顺序：j从0开始，i也是从0开始

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

**求不连续的回文子串的长度**

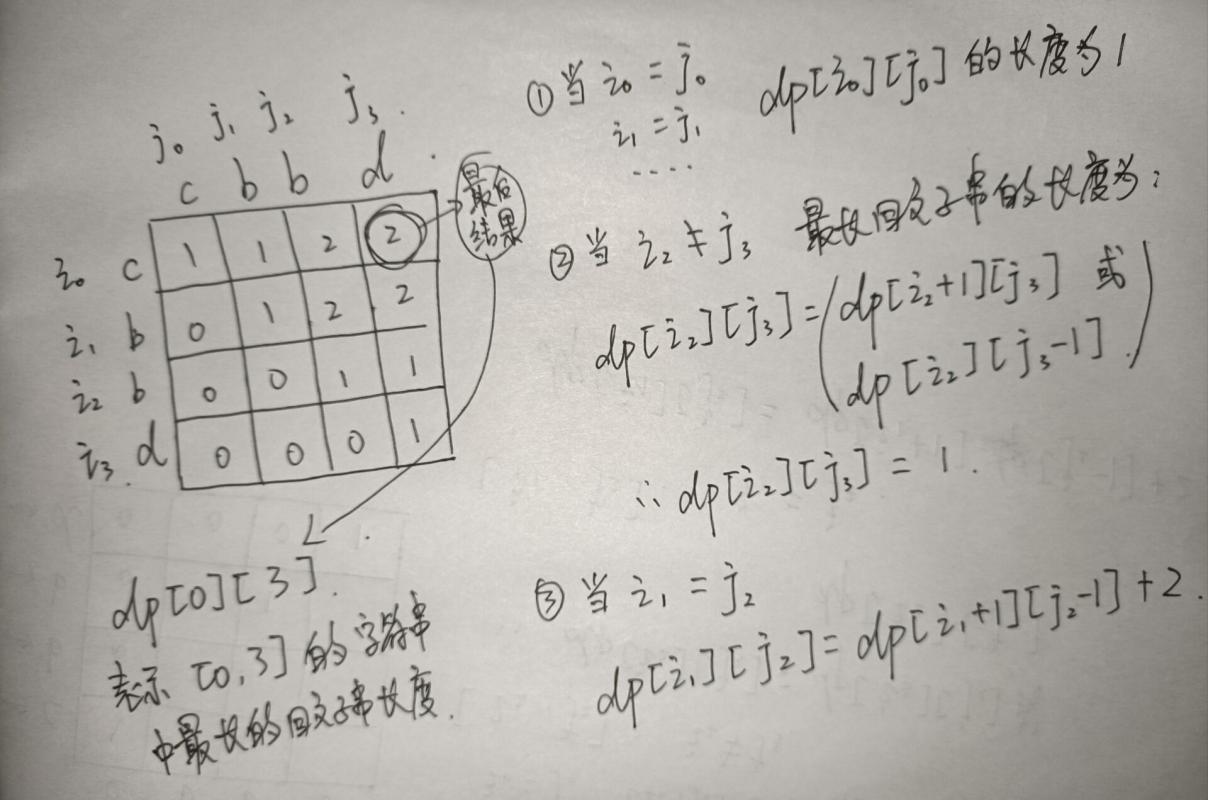
示例 1: 输入: "bbbab" 输出: 4 一个可能的最长回文子序列为 "bbbb"。

**dp[i][j]：字符串s在[i, j]范围内最长的回文子序列的长度为dp[i][j]**。



如果s[i]==s[j] 则长度加2 dp[i][j]=dp[i+1][j-1]+2

不相等（取s[i]或s[j]）:dp[i][j]=max(dp[i+1][j],dp[i][j-1])



遍历顺序为：i从i2到i0 ,j从（i+1）到len-1

# 贪心算法

**455.分发饼干 ：**假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。

对每个孩子 i，都有一个胃口值 g[i]，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 j，都有一个尺寸 s[j] 。如果 s[j] >= g[i]，我们可以将这个饼干 j 分配给孩子 i ，这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

**先满足胃口最大的小孩，再逐渐满足胃口小的（从小到大排序）**

**遍历小孩（从后往前），**如果 s[j] >= g[i]，**饼干长度自减**

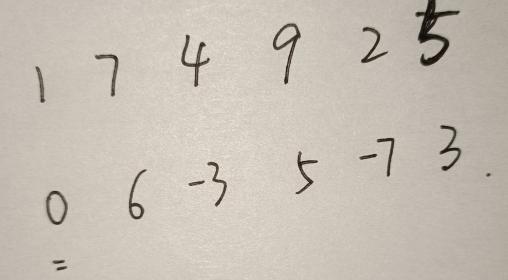
**376. 摆动序列 ：**如果连续数字之间的差严格地在正数和负数之间交替，则数字序列称为摆动序列。第一个差（如果存在的话）可能是正数或负数。少于两个元素的序列也是摆动序列。

例如， [1,7,4,9,2,5] 是一个摆动序列，因为差值 (6,-3,5,-7,3) 是正负交替出现的。给定一个整数序列，返回作为摆动序列的**最长子序列的长度**。

**整体最优：整个序列有最多的局部峰值，从而达到最长摆动序列**。

**记录当前差值，和前一个差值 两个差值，一个大于0，一个小于0，则存在一个峰值**

**第一个差值取0，之后的差值与当前值相比**

 存在一个峰值，长度为2，所以result初值1

 if((c1>=0&&c2<0)||(c1<=0&&c2>0)){

            result++

            c1=c2

        }

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

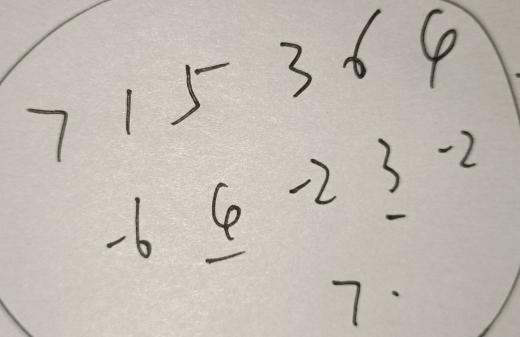
**如果当前和小于0，重置sum=0**

**求最大和，则result初值为最小值-Infinity**

**122.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）。

**累计所有正利润**



**55. 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**。

**遍历能到达的最远距离end（初始化为n[0]）,记录在这区间最大的距离max,**

**如果max>len-1，则可以到达终点，**

**当i==end时，更新结束位置end=max**

for(let i=0;i<=end;i++){

        max=Math.max(max,i+nums[i])

        if(max>=nums.length-1){

            return true

        }

        if(i==end){

            end=max

        }

    }

**45.跳跃游戏II ：**给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用**最少的跳跃次数**到达数组的最后一个位置。

**遍历能到达的最远距离end（初始化为n[0]）,记录在这区间最大的距离max,**

**当i==end时，但是max<len-1,需要再跳一次ans++，更新结束位置end=max**

for(let i=0;i<=end;i++){

     max=Math.max(max,i+nums[i])

     if(i==end && i<nums.length-1){

         ans++

         end=max

            }

        }

**1005.K次取反后最大化的数组和 :**给定一个整数数组 A，我们只能用以下方法修改该数组：我们选择某个索引 i 并将 A[i] 替换为 -A[i]，然后总共重复这个过程 K 次。（我们可以多次选择同一个索引 i。）

以这种方式修改数组后，返回数组可能的最大和。

**绝对值从大到小排序，判断其是否为负数且k>0，把前面的负数反转(k--)**

**如果没有负数，k大于0且为奇数,把绝对值小的正数反转（后置k=0）**

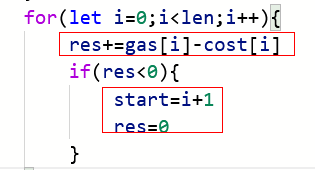
**134. 加油站:**在一条环路上有 N 个加油站，其中第 i 个加油站有汽油 gas[i] 升。

你有一辆油箱容量无限的的汽车，从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。你从其中的一个加油站出发，开始时油箱为空。

如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

**先总体判断油量是否能跑完一圈：耗油量大于存量，返回-1**

**找到出发点：如果剩余的油量小于0，取下一个节点为初始值，并重置剩余油量=0**



**135. 分发糖果：** 老师想给孩子们分发糖果，有 N 个孩子站成了一条直线，老师会根据每个孩子的表现，预先给他们评分。

你需要按照以下要求，帮助老师给这些孩子分发糖果：

* 每个孩子至少分配到 1 个糖果。
* 相邻的孩子中，评分高的孩子必须获得更多的糖果。

那么这样下来，老师至少需要准备多少颗糖果呢？

**从左向右遍历（基数为1，如果比左边的元素大，加1），**

**再从右向左遍历（如果比右边的元素大取max(arr[i+1]+1,arr[i])），**

**860.柠檬水找零 :**在柠檬水摊上，每一杯柠檬水的售价为 5 美元。

顾客排队购买你的产品，（按账单 bills 支付的顺序）一次购买一杯。

每位顾客只买一杯柠檬水，然后向你付 5 美元、10 美元或 20 美元。你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 5 美元。

注意，一开始你手头没有任何零钱。

如果你能给每位顾客正确找零，返回 true ，否则返回 false 。

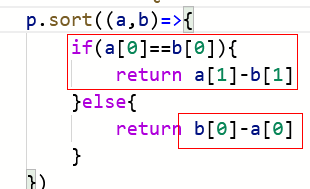
**记录5，10的个数**

**收5：f++ 收10:(t++,f--) 收20：（t--,f--）或（f-3）先花10，再花5**

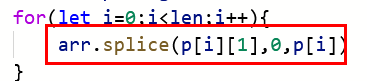
**406.根据身高重建队列 ：**假设有打乱顺序的一群人站成一个队列，数组 people 表示队列中一些人的属性（不一定按顺序）。每个 people[i] = [hi, ki] 表示第 i 个人的身高为 hi ，前面 正好 有 ki 个身高大于或等于 hi 的人。

请你重新构造并返回输入数组 people 所表示的队列。返回的队列应该格式化为数组 queue ，其中 queue[j] = [hj, kj] 是队列中第 j 个人的属性（queue[0] 是排在队列前面的人）。

**先按照身高hi从大到小排序，如果身高相等，按ki从小到大排序**



**然后按顺序，把元素插入到第ki个位置**



**452. 用最少数量的箭引爆气球 ：**一支弓箭可以沿着 x 轴从不同点完全垂直地射出。在坐标 x 处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart，xend， 且满足  xstart ≤ x ≤ xend，则该气球会被引爆。可以射出的弓箭的数量没有限制。 弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。

给你一个数组 points ，其中 points [i] = [xstart,xend] ，返回引爆所有气球所必须射出的最小弓箭数。

**按起点排序，每次加入的起点要比最小终点小**

**更新最小终点**

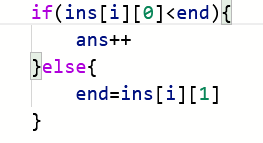
end=Math.min(end,points[i][1])

**435. 无重叠区间 ：**给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。

注意: 可以认为区间的终点总是大于它的起点。 区间 [1,2] 和 [2,3] 的边界相互“接触”，但没有相互重叠。

**以终点排序，下一个起点小于当前终点，则需要删除，**

**否则更新当前第i个元素的终点**

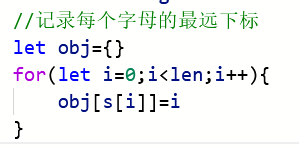


**763.划分字母区间 ：**字符串 S 由小写字母组成。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。返回一个表示每个字符串片段的长度的列表。

示例：

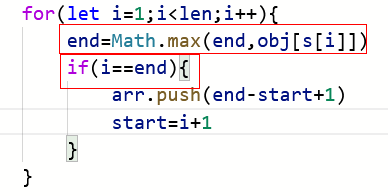
* 输入：S = "ababcbacadefegdehijhklij"
* 输出：[9,7,8] 解释： 划分结果为 "ababcbaca", "defegde", "hijhklij"。 每个字母最多出现在一个片段中。 像 "ababcbacadefegde", "hijhklij" 的划分是错误的，因为划分的片段数较少。

**找到每个字母出现的最远的下标，用对象保存，字母为键，下标为值**



**遍历当前范围，如果当前范围内没有超出，累计最远范围**

**如果下标到达end,则输出end-start+1，并更新start**



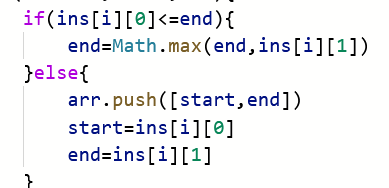
**56. 合并区间 :**给出一个区间的集合，请合并所有重叠的区间。

示例 1:

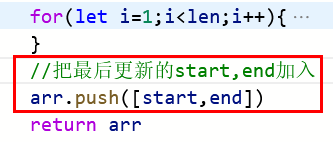
* 输入: intervals = [[1,3],[2,6],[8,10],[15,18]]
* 输出: [[1,6],[8,10],[15,18]]
* 解释: 区间 [1,3] 和 [2,6] 重叠, 将它们合并为 [1,6].

**当前i的起点小于等于end，则重叠，更新end**

**如果不重叠，加入数组，并更新起始结束点**



**！！！把最后更新的起点结束加入**

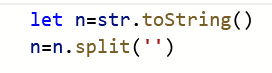


**738.单调递增的数字 ：**给定一个非负整数 N，找出小于或等于 N 的最大的整数，同时这个整数需要满足其各个位数上的数字是单调递增。

（当且仅当每个相邻位数上的数字 x 和 y 满足 x <= y 时，我们称这个整数是单调递增的。）

* 输入: N = 332
* 输出: 299

**转化为字符串（toString()），分割成数组split(‘’),输出是把数组转化为字符串n.join(‘’),再转化为数字+n**



**标记变成9的位置，然后遍历把之后的都变成9！！**

**从后往前遍历，当n[i]比n[i-1]小，则n[i]=9,n[i-1]-1**

**968.监控二叉树:**

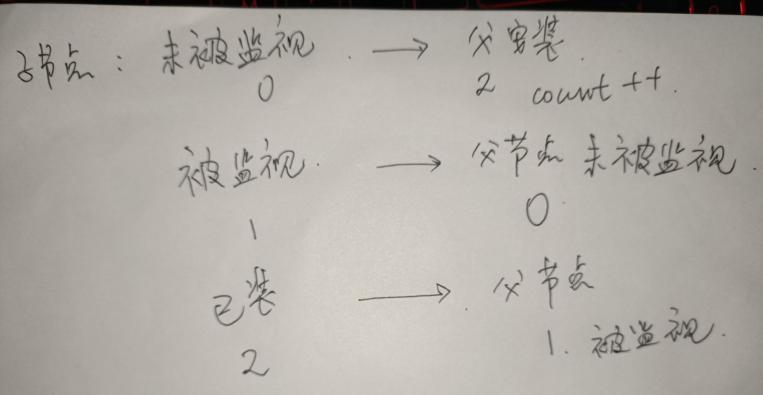
第一步：确定节点3个状态，

0代表未被监视；

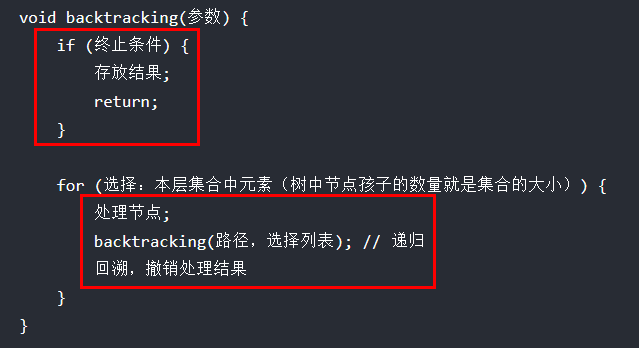
1代表被监视；

2代表已安装摄像头；

第二步：后序遍历；先处理2个子节点再处理父节点；



**回溯算法**



for循环是**横向遍历**N叉树  
递归是**纵向遍历**N叉树

## for循环起点是index还是0？

**在递归过程中，下次层是否可以选择集合里的所有元素**

**可选范围是（index,len）还是（0，len）**

当通过**一个集合**求组合（或分割或子集），需要start，防止重复；

当通过**多个集合**求组合（或分割或子集），不需要start，因为各个集合都不同不会相互影响；

如果是**排列问题**的话，就不需要start，因为每到新的一列都需要重新使用之前使用的元素；

## 如果有start，什么时候递归时i加1？

**看下一层中，当前元素是否可重复使用**

**下一层的可选范围：（i，len）还是（i+1.len）**

当一个集合中**每个元素只能使用一次**时，i + 1；

当一个集合中每个元素**可以重复使用**时，不用加1；

**设置for的起点，和递归设置i的原理是一样的，用于决定下一层的可选范围**

**不同的是，如果for设置的起点为0，说明下一层的可选范围是整个集合**

**递归中i不加1，说明下一层的可选范围是从i（包括i）开始（i可重复使用）**

## 需要去重类的题目：“树层去重”和“树枝去重”

树枝去重  
**used[i] == true，说明**同一树枝**元素 i 使用过**

数层（同一层）去重 **方法一：used[i] == false，说明**同一树层**元素 i 使用过**

**方法二：排序去重**if(i>index && cs[i]==cs[i-1])

如果为false，说明递归已经结束，已经通过回溯回到了false，所以此时该遍历同一层的下一个元素，

如果是true，则说明是**这一个树枝**上已经使用的元素，一般这里去重**出现在排列中**；

## 什么时候需要 树层去重 什么时候需要 树枝去重

1，如果**一个集合**里面会**出现重复的元素**，结果还不能有重复，那么就需要**树层去重**  
2，树枝去重一般就会出现在排列问题中，其他问题暂时还没有遇到

**第77题. 组合 :**给定两个整数 n 和 k，返回范围 [1, n] 中所有可能的 k 个数的组合。

你可以按 任何顺序 返回答案。

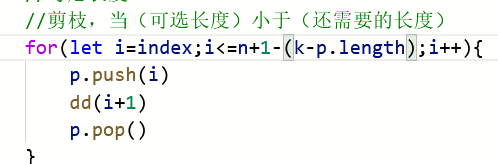
输入：n = 4, k = 2

输出：[[2,4],[3,4],[2,3],[1,2],[1,3],[1,4],]

**当前元素不能重复选择：dd（i+1）**

**剪枝:**（可选长度）>=（还需要的长度）

**起点+（k-p.lrngth）-1=n index<=n+1-(k-p.length)**



**216.组合总和III :**找出所有相加之和为 n 的 k 个数的组合。组合中只允许含有 1 - 9 的正整数，并且每种组合中不存在重复的数字。

示例 1: 输入: k = 3, n = 7 输出: [[1,2,4]]

示例 2: 输入: k = 3, n = 9 输出: [[1,2,6], [1,3,5], [2,3,4]]

**当前元素不能重复使用:dd(i+1)**

**长度等于k的时候，如果sum=n,则加入arr**

**17.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。

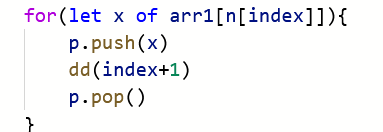
给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。



示例: 输入："23" 输出：["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

**用一个数组保存字典，下标是对应的数字，输入的n是字符串，**

**字典的下标===n[i]**



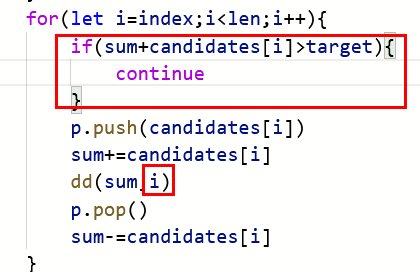
**39. 组合总和 ：**给定一个无重复元素的数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的数字可以**无限制重复被选取**。 **递归的起始起点不用加1**

示例 2： 输入：candidates = [2,3,5], target = 8, 所求解集为： [   [2,2,2,2],   [2,3,3],   [3,5] ]

**当前元素可重复使用：dd(i)**

**剪枝：在加入之前判断是否会大于目标值，如果会大于，直接continue**



**40.组合总和II :**给定一个数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的每个数字在每个组合中只能使用一次。递归下标 i+1

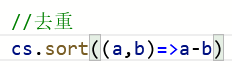
说明： 所有数字（包括目标数）都是正整数。 解集不能包含重复的组合。

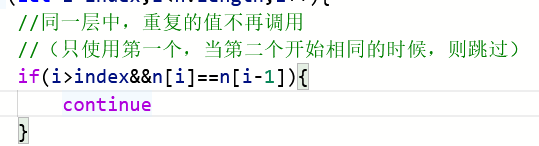
示例 1: 输入: candidates = [10,1,2,7,6,1,5], target = 8, 所求解集为: [ [1, 7], [1, 2, 5], [2, 6], [1, 1, 6] ]

**当前元素不能重复使用：dd(i+1)**

**解集不能有重复的，需要进行去重（相同的同层不能使用）**

**采用排序，元素值相同的则直接剪枝**





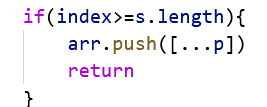
**131.分割回文串 ：**给定一个字符串 s，将 s 分割成一些子串，使每个子串都是回文串。

返回 s 所有可能的分割方案。

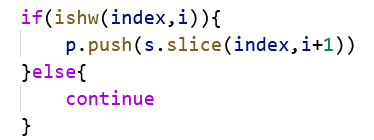
示例: 输入: "aab" 输出: [ ["aa","b"], ["a","a","b"] ]

**同一个组合中，元素不能重复使用：dd(i+1)**

**（1）退出递归的条件：当某一个路径切割到字符串尾部，则结束**

****

**（2）如果符合回文字串则加入p数组，如果不符合直接跳过当前递归**

****

1. **判断回文字串，首尾指针对比**

**93.复原IP地址 :**给定一个只包含数字的字符串，复原它并返回所有可能的 IP 地址格式。

有效的 IP 地址 正好由四个整数（每个整数位于 0 到 255 之间组成，且不能含有前导 0），整数之间用 '.' 分隔。

例如："0.1.2.201" 和 "192.168.1.1" 是 有效的 IP 地址，但是 "0.011.255.245"、"192.168.1.312" 和 "192.168@1.1" 是 无效的 IP 地址。

**递归：Dd(i+1) 4<Len<12**

**退出条件：遍历结束（index=len），且p的长度为4**

**截取的单个字符串长度大于3或值大于255；不能以0开头，break**

if(str.length>3 || +str>255 ||(str.length>1&&str[0]=='0'))

**78.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: 输入: nums = [1,2,3] 输出: [ [3],   [1],   [2],   [1,2,3],   [1,3],   [2,3],   [1,2],   [] ]

**遍历整棵树，不需要退出条件，不重复dd(i+1)**

**90.子集II :**给定一个可能包含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。

说明：解集不能包含重复的子集。

示例:

* 输入: [1,2,2]
* 输出: [ [2], [1], [1,2,2], [2,2], [1,2], [] ]

**集合中有重复的元素，但是组合要求不能重复，需要去重（sort）**

**遍历整棵树，不需要退出条件，组合不重复dd(i+1)**

**Use的用法：没有赋值之前为undefined，撤销标记为false**

**当use的值为true的时候，退出此次循环**

**标记本层：在for之前初始化，把用过的标记为true(不用撤销标记，因为同一层的不涉及递归)**

**标记树枝：在递归的父节点就传入use,用于标记整个树枝，涉及递归，所以当回溯的时候，也需要撤销标记**

**491.递增子序列 :**给定一个整型数组, 你的任务是找到所有该数组的递增子序列，递增子序列的长度至少是2。

示例:

* 输入: [4, 6, 7, 7]
* 输出: [[4, 6], [4, 7], [4, 6, 7], [4, 6, 7, 7], [6, 7], [6, 7, 7], [7,7], [4,7,7]]

**一个组合中同一个元素不能重复使用:递归i加1**

**集合有重复的值，但是组合不能重复，需要去重，**

**且本题需要保留元素的顺序，不能用排序去重**

**采用：used[i]==false判断同一层中，是否使用过值相同的元素**

**Used的下标是nums[i]的值**

**递增序列，用当前元素与p中最后一个元素比较**

**！！！use初始化的位置:记录同层的元素是否使用过（不涉及递归），所以在进入每层的遍历前，就需要初始化**

**46.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:

* 输入: [1,2,3]
* 输出: [ [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1] ]

集合没有重复的数字，不需要数层的去重

**全排列，下一层的范围是整个集合（除了当前i）**

**所以：for从0开始，递归中需要标记（一整个树枝用过的元素）**

**！！！use的初始化，放在递归中（跟随遍历整个数枝）**

**Use==true,则表示在本组和中用过**

**47.全排列 II ：**给定一个可包含重复数字的序列 nums ，按任意顺序 返回所有不重复的全排列。

示例 1：

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

**集合有重复的数字，需要数层的去重，**

**且在每次递归中需要去掉用过的值，所以需要树枝去重**

**树枝去重，用过标记为true,撤销标记为false**

**树层去重，没用过的值应该为undefined,如果为false表示用过了**

**需要进行排序如果和前一个相等，use标记的是下标**

**且use值为false,表示用过**

 for(let i=0;i<len;i++){

            //数层去重

            if(i>0&&nums[i]==nums[i-1]&&use[i-1]==false){

                continue

            }

            //树枝去重

            if(use[i]){

                continue

            }

            use[i]=true

            p.push(nums[i])

            dd(use)

            p.pop()

            use[i]=false

        }

**二叉树**

****什么时候需要返回值？****

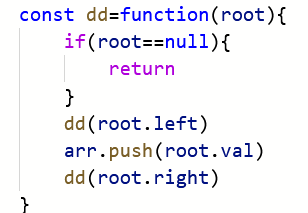
**递归是重复调用函数自身实现循环。迭代是函数内某段代码实现循环，**

****二叉树的递归遍历 ：****

1. ****二叉树的中序遍历：****

**遍历整棵树，把每个节点加入arr,所以不需要返回值**

**退出递归的条件，空节点则返回**



****二叉树的统一迭代法：****

****把访问节点放入栈中，处理节点也放入，随后加入标记null****

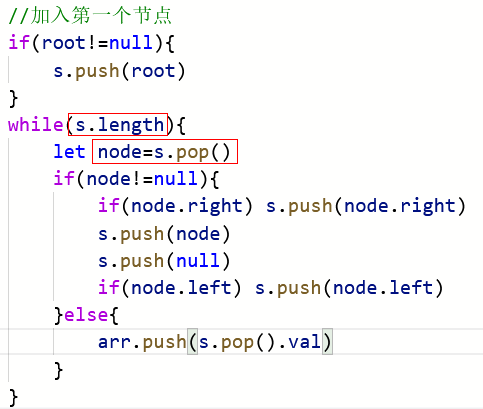
****加入栈的顺序要与加入结果的顺序相反****

****递归的条件，栈不为空****

****判断弹出的末尾节点是null还是节点值****

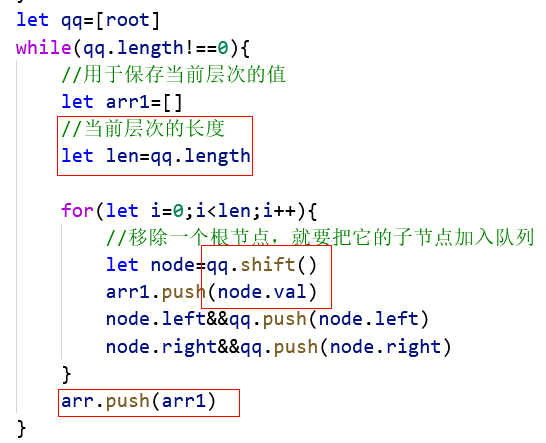
****不为空，取出该节点，再按顺序加入节点和其左右子节点（空节点不入栈）****

****为null,说明此节点为处理节点：取出需要处理的节点，加入结果数组arr****



**二叉树层序遍历 ：**

**在栈中，如果移除第一个（shift）节点，就要把其非空的子节点放进队列**

**一层一个数组；所以需要记录每层的节点数，根据数量遍历**

**只用一个循环实现层序遍历：**

**用arr接收子节点，当s遍历结束时，把子节点数组赋值给s**

**注：arr1,arr2要在while前初始化**

var levelOrderBottom = function(root) {

    let arr=[]  //保存结果

    let s=[]   //栈

    if(root==null){

        return arr

    }

    s.push(root)

    let arr1=[]  //本层

    let arr2=[]     //子节点数组，用于和s交换

    while(s.length){

        let node=s.shift()

        arr1.push(node.val)

        node.left&&arr2.push(node.left)

        node.right&&arr2.push(node.right)

        if(s.length===0){

            arr.push(arr1)

            s=[...arr2]

            arr1=[]

            arr2=[]

        }

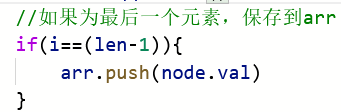
    }

    return arr.reverse()

};

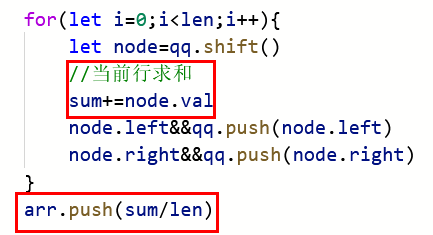
1. **二叉树的右视图 ：**给定一个二叉树的 **根节点** root，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。

**把每层最后一个节点加入结果数组**



1. **二叉树的层平均值 ：**给定一个非空二叉树的根节点 root , 以数组的形式返回每一层节点的平均值。与实际答案相差 10-5 以内的答案可以被接受。

把每层的值累加除以len,把结果加入数组

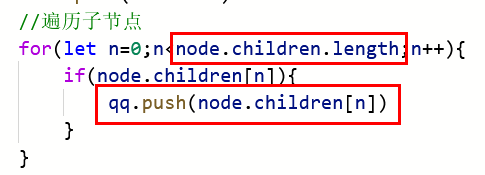


**429. N 叉树的层序遍历 ：**给定一个 N 叉树，返回其节点值的层序遍历。（即从左到右，逐层遍历）。

树的序列化输入是用层序遍历，每组子节点都由 null 值分隔（参见示例）。

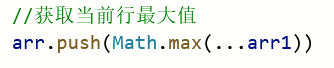
**与二叉树的区别在于，不知道子节点的长度，需要遍历的加入子节点**

**子节点的长度为node.children.length**



1. **在每个树行中找最大值 :**给定一棵二叉树的根节点 root ，请找出该二叉树中每一层的最大值。

**找到每层最大的值，再加入结果数组**



**116. 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：

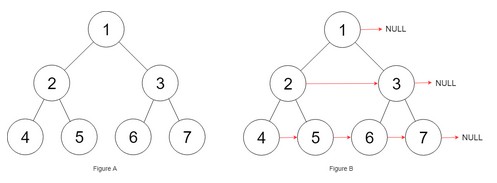
struct Node {

int val;Node \*left;Node \*right; Node \*next;

}

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 NULL。

初始状态下，所有 next 指针都被设置为 NULL。



**每退出一个就把它的next指向栈s中的第一个元素s[0]**

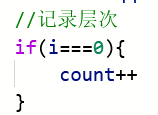
**如果是该层的最后一个，则指向空**



**104. 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。

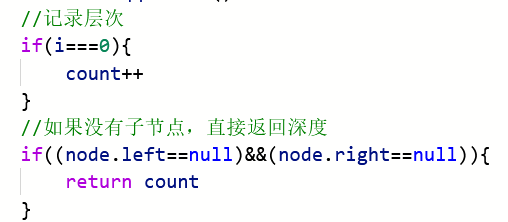
**记录层数**



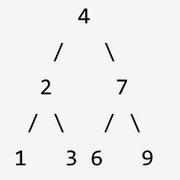
**111. 二叉树的最小深度：**给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

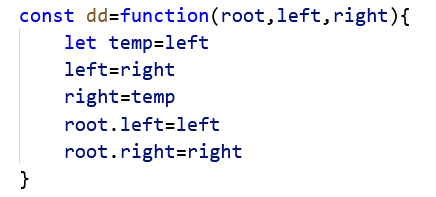
**记录层数，当没有子节点就返回**



1. **翻转二叉树 ：**给你一棵二叉树的根节点 root ，翻转这棵二叉树，并返回其根节点。

 ----》

**左右子树交换的函数：**



**法一递归**：返回值是根节点，交换左右子树

var invertTree = function(root) {

    if(root==null){

        return root

    }

**dd(root,root.left,root.right)**

    invertTree(root.left)

    invertTree(root.right)

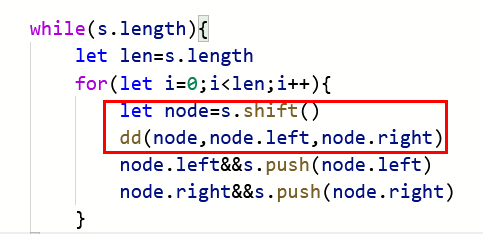
    return root

};

**统一迭代法：当退出的节点为空时，需要处理节点（交换节点）**

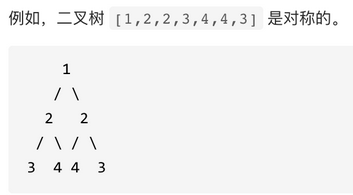


**层序遍历法：交换左右子树后再加入栈s**



**小结：**

1. **递归：交换当前节点左右子树，再递归处理左节点和右节点**
2. **迭代：在root=null时，处理节点（交换左右子树）**
3. **层序：移除节点(shift())，交换了左右子节点，再加入栈s**

**101. 对称二叉树 :**给定一个二叉树，检查它是否是镜像对称的。

root.left.left=root.right.right

root.left.right=root.right.left

法一：

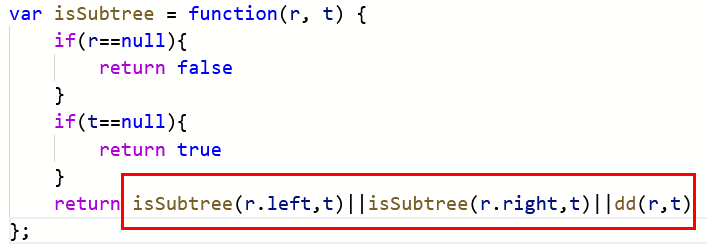
**递归：判断两个节点是否相等（两个都为空，一个为空，两个的值不等），再判断其左右子节点是否相等**

**法二：迭代法（类似层序遍历），一个栈，遍历两个子树（取前两个节点），判断两个节点是否相等（两个为空则结束循环，一个为空或值不等返回false），然后按照特定的顺序，加入其子节点**

**572. 另一棵树的子树：**给你两棵二叉树 root 和 subRoot 。检验 root 中是否包含和 subRoot 具有相同结构和节点值的子树。如果存在，返回 true ；否则，返回 false 。

二叉树 tree 的一棵子树包括 tree 的某个节点和这个节点的所有后代节点。tree 也可以看做它自身的一棵子树。

**先找到相等的父节点，如果父节点相等再调用dd（）再判断其子树是否相等**

判断子树是否相等的函数与上题一致

**104.二叉树的最大深度**

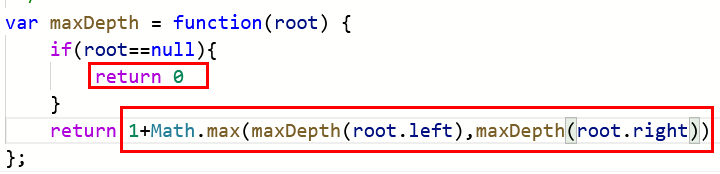
**559.n叉树的最大深度**

层序遍历，for循环中，当i=0，count++

1. **二叉树的最小深度**

层序遍历，for循环中，当i=0，count++，当其中一个无子节点，立马返回count

**以上两题的递归解法：**

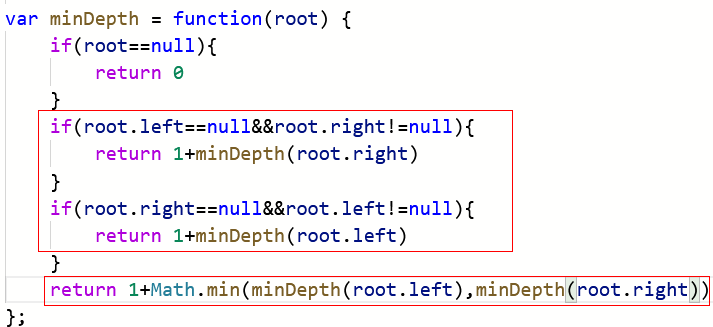
**最大深度：**

**最小深度：**

这就重新审题了，题目中说的是：**最小深度是从根节点到最近叶子节点的最短路径上的节点数量。**，注意是**叶子节点**。

什么是叶子节点，左右孩子都为空的节点才是叶子节点

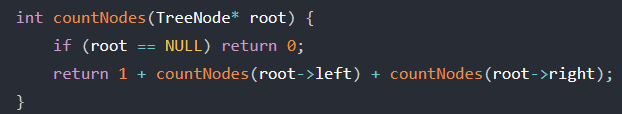
**增加一个判断，当左右节点中有一个为空，则返回不为空的子树的最小深度**



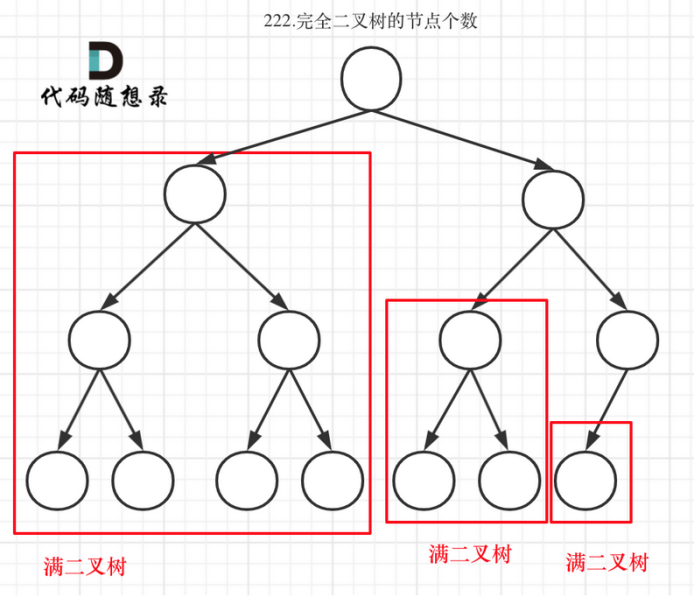
**普通二叉树求结点数：**

**层序遍历累计节点个数，**

**或者递归**



1. **完全二叉树的节点个数 ：**给出一个完全二叉树，求出该树的节点个数。

完全二叉树只有两种情况，情况一：就是满二叉树（2^k - 1 ）

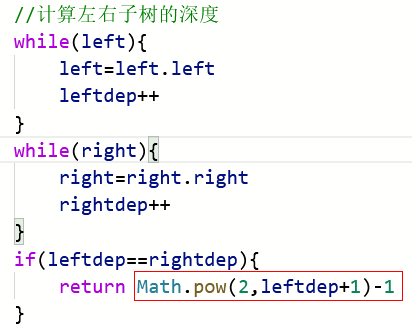
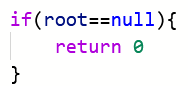
一棵叉树的子节点数=1+left+right

递归左子树和右子树的深度，如果深度相等，则为满二叉树（返回节点个数2^k-1）

注意初始化：

左子树和右子树的高度初始化为0，

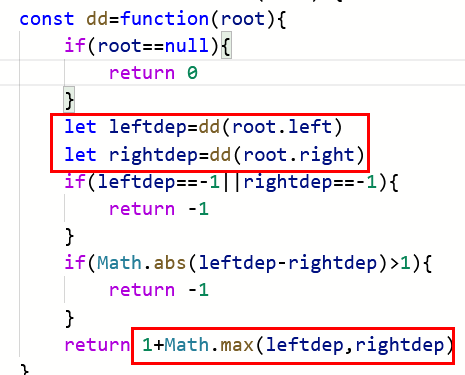
当高度相等时返回**Math.pow(2,leftdep+1)-1**



**110.平衡二叉树 ：**给定一个二叉树，判断它是否是高度平衡的二叉树。

本题中，一棵高度平衡二叉树定义为：一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过1。

**递归判断左右子树的高度差（左右子树差绝对值大于1，返回-1，返回大的高度）**

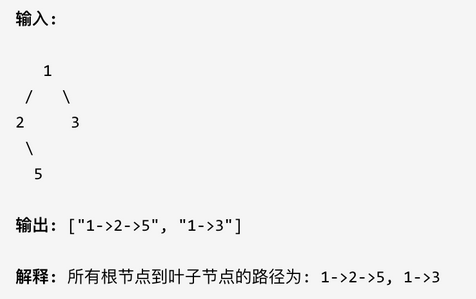


**！！！ 高频考点**

**257. 二叉树的所有路径 ：**给定一个二叉树，返回所有从根节点到叶子节点的路径。

说明: 叶子节点是指没有子节点的节点。

**（遍历所有路径，不需要处理返回值，遇到叶子节点就加入结果，不是叶子节点则多加入“->”，且递归子节点）**



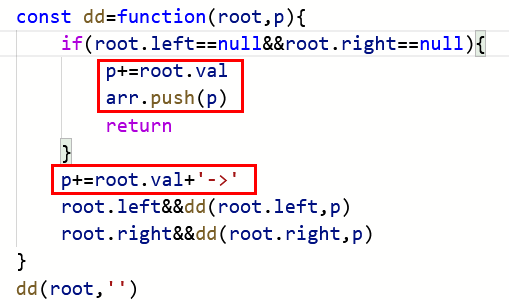
类似于回溯算法：

**遍历整棵树，不需要返回值：**

**退出递归的条件：**

**没有叶子节点则把当前值加入路径（** p+=root.val**），并把路径加入结果**

**当前值加入路径（**p+=root.val+'->'**），递归左右子节点**



**再来看返回值，递归函数什么时候需要返回值？什么时候不需要返回值？这里总结如下三点：**

**1，如果需要搜索整棵二叉树且不用处理递归返回值，递归函数就不要返回值。（这种情况就是本文下半部分介绍的113.路径总和ii）**

**2，如果需要搜索整棵二叉树且需要处理递归返回值，递归函数就需要返回值。 （这种情况我们在[236. 二叉树的最近公共祖先](https://programmercarl.com/0236.%E4%BA%8C%E5%8F%89%E6%A0%91%E7%9A%84%E6%9C%80%E8%BF%91%E5%85%AC%E5%85%B1%E7%A5%96%E5%85%88.html" \t "https://www.programmercarl.com/_blank)中介绍）**

**3，如果要搜索其中一条符合条件的路径，那么递归一定需要返回值，因为遇到符合条件的路径了就要及时返回。（112）**

**为什么sun不需要回溯但是path需要回溯呢？**

传入数组，通过push操作，会改变原数组，所以需要回溯，但是sum是基本数据，当作参数传入函数，不会影响原来的值

1. **路径总和 ：**给定一个二叉树和一个目标和，判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和。

**（有返回值，就需要接收其递归过程中子节点产生的返回值）**

1. **当前为叶子节点，有两种情况，和为sum,（返回true）,和不等于sum（返回false）**
2. **当前不是叶子节点，处理子节点，递归处理子节点（两个子节点有一个为true，那么结果就是true）**



**迭代法：采用栈的方式（s.pop()）**

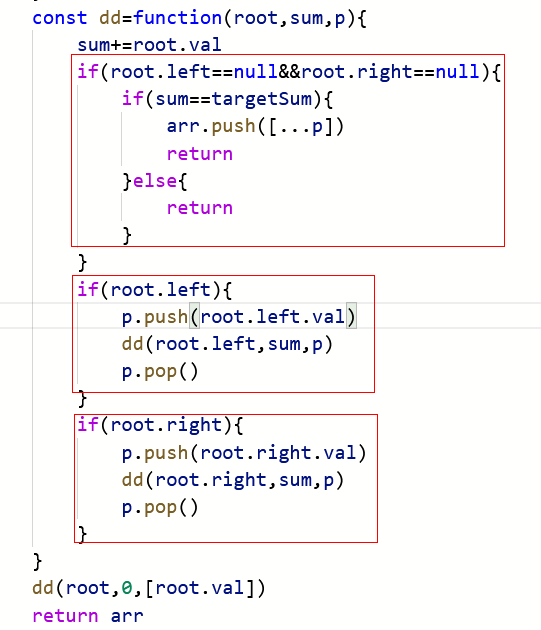
**如果是叶子节点，sum==target,则返回true**

**不是叶子节点，取出一个节点就把其子节点加入进去，并累加当前节点的值，**

**113. 路径总和ii :**给定一个二叉树和一个目标和，找到所有从根节点到叶子节点路径总和等于给定目标和的路径。

说明: 叶子节点是指没有子节点的节点。

**遍历整棵树，不需要返回值，把符合条件的路径加入结果数组，与上一题的区别在于需要记录路径，单独处理左右节点的回溯**



1. **左叶子之和 ：**计算给定二叉树的所有左叶子之和。

**左子树的左叶子+右子树的左叶子（叶子：没有子节点）**

**如果左节点不为空，且左节点没有左右孩子，那么这个节点就是左叶子**

**root.left!==null&&root.left.left==null&&root.left.right==null**

1. **找树左下角的值 ：**给定一个二叉树，在树的最后一行找到最左边的值。

**层序遍历：取第一个（一直替换）**

1. **从中序与后序遍历序列构造二叉树：**给定两个整数数组 inorder 和 postorder ，其中 inorder 是二叉树的中序遍历， postorder 是同一棵树的后序遍历，请你构造并返回这颗 二叉树 。

**1，取后序最后一个字符(pop())，在中序中找到对应的下标（indexof），建立根节点new TN,划分中序**

**2，后序数组就可以按照左中序数组的大小来切割，切成左后序数组和右后序数组**。

3,左子树区间slice（0,index）右子树slice(index+1) //中序

4,slice(0,index) slice(index) //后序

**654.最大二叉树 :**给定一个不含重复元素的整数数组。一个以此数组构建的最大二叉树定义如下：

* 二叉树的根是数组中的最大元素。
* 左子树是通过数组中最大值左边部分构造出的最大二叉树。
* 右子树是通过数组中最大值右边部分构造出的最大二叉树。

通过给定的数组构建最大二叉树，并且输出这个树的根节点。

**需要注意左右下标会不会越界，所以建立一个递归函数传入左后端点**

 const dd=function(arr,left,right){

        if(left>right){

            return null

        }。。。。。。

**1,找到最大值的下标，最大值建立节点，划分左右区间，左右分别建树**

**108.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。

**如上题需要判断是否越界，，向下取整中间的值为头节点**

let index=Math.floor(left+(right-left)/2)

**617.合并二叉树 ：**给定两个二叉树，想象当你将它们中的一个覆盖到另一个上时，两个二叉树的一些节点便会重叠。

你需要将他们合并为一个新的二叉树。合并的规则是如果两个节点重叠，那么将他们的值相加作为节点合并后的新值，否则不为 NULL 的节点将直接作为新二叉树的节点。

**迭代：**

**两棵树一起放入一个队列层序遍历（shift）,**

**当前节点累加（root1.val+=root2.val）**

**返回的是树1，只有两个相同的节点都不为空才需要（加入队列）累加，有一个为空，直接赋值到树1的节点上**

**递归：**

**如果有一个为空，返回非空的树**

**都不为空，累加在树1的节点上（root1.val+=root2.val）**

**再递归处理树1的左右子树**

**二叉搜索树BST（中序遍历）**

1. **二叉搜索树中的搜索 ：**给定二叉搜索树（BST）的根节点和一个值。 你需要在BST中找到节点值等于给定值的节点。 返回以该节点为根的子树。 如果节点不存在，则返回 NULL。

**处理当前节点：root为空，或者root.val==val,返回节点**

if(root==null||root.val==val){

        return root

    }

**递归处理子节点：**

**如果val>root.val,找右子树**

**如果val<root.val,找左子树**

**98.验证二叉搜索树 ：**给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

* 节点的左子树只包含小于当前节点的数。
* 节点的右子树只包含大于当前节点的数。
* 所有左子树和右子树自身必须也是二叉搜索树。

**递归法：**

**法1：中序遍历所有节点加入数组，判断数组是否是递增序列（遍历数组）**

**法2：中序遍历，如果前一个值大于当前值，则为false，（记得更新p指针）**

if(pre!==null&&pre.val>=root.val){

            return false

        }

        //记录当前值

        pre=root

1. **二叉搜索树的最小绝对差：**给你一棵所有节点为非负值的二叉搜索树，请你计算树中任意两节点的差的绝对值的最小值。

**中序遍历所有节点加入数组，最小值设置为Infinity,遍历数组的时候求最小差值**

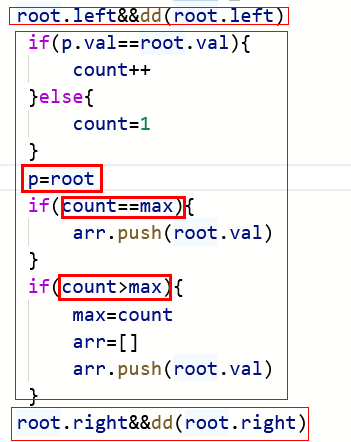
1. **二叉搜索树中的众数:**给定一个有相同值的二叉搜索树（BST），找出 BST 中的**所有众数**（出现频率最高的元素）。

**与98题法二类似，在递归过程中处理，（累计次数之后，更新p指针指向当前节点）**

**1，前一个值与当前值相等，则次数加1,与前一个值不相等，初始化为1（记得更新p节点）**

**2，如果当前次数等于最大值，加入arr（因为可能存在多个众数）**

**3,当前次数大于等于记录最大次数，更新最大值，清空数组，加入当前值**



**！！高频**

1. **二叉树的最近公共祖先 ：**给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

**自底向上寻找祖先，采用后序的方式**

**判断当前节点的子树中是否含有一个目标节点**

**推出递归：如果当前节点为空或为目标值，则返回节点**

**处理节点：**

**如果当前的左右子树的返回值都不是null(则是目标值)，则返回当前节点**

**如果一个为空，则返回另一个子树**

var lowestCommonAncestor = function(root, p, q) {

    if(root==null||root==p||root==q){

        return root

    }

    let left=lowestCommonAncestor(root.left,p,q)

    let right=lowestCommonAncestor(root.right,p,q)

**if(left&&right){**

**return root**

**}**

    return left||right

};

1. **二叉搜索树的最近公共祖先 ：**给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

**如果当前节点值在p和q之间，则当前节点为公共父节点**

**如果两个目标值在同一个子树上，递归继续查找**

**如果不在同一个子树上，直接返回节点**

var lowestCommonAncestor = function(root, p, q) {

    if(root==null){

        return root

    }

    if(**p.val>root.val&&q.val>root.val**){

        let right=lowestCommonAncestor(root.right,p,q)

        return right

    }

    if(**p.val<root.val&&q.val<root.val**){

        let left=lowestCommonAncestor(root.left,p,q)

        return left

    }

**return root**

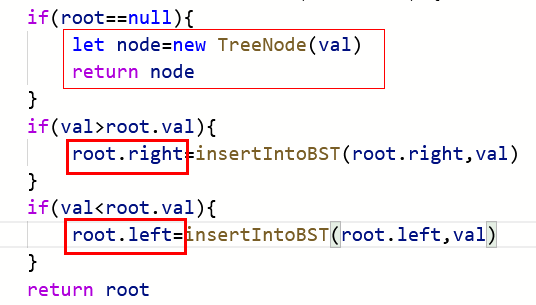
};

**701.二叉搜索树中的插入操作 ：**给定二叉搜索树（BST）的根节点和要插入树中的值，将值插入二叉搜索树。 返回插入后二叉搜索树的根节点。 输入数据保证，新值和原始二叉搜索树中的任意节点值都不同。

注意，可能存在多种有效的插入方式，只要树在插入后仍保持为二叉搜索树即可。 你可以返回任意有效的结果。

**插入操作都是在叶子节点后添加**

**如果比当前节点大，插在右子树，反之则插在左子树**



1. **删除二叉搜索树中的节点：**给定一个二叉搜索树的根节点 root 和一个值 key，删除二叉搜索树中的 key 对应的节点，并保证二叉搜索树的性质不变。返回二叉搜索树（有可能被更新）的根节点的引用。

**返回值用当前节点的左右子树接收删除过后的子树，所以返回的是除去当前节点后剩余的子树**

当前节点：

找不到root=null,直接返回，大于或小于，递归查找其子树

相等：

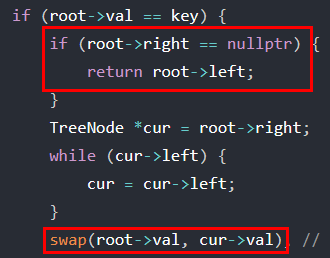
如果没有子节点删除节点，返回null

如果有一个节点，删除本节点，返回其子节点

如果有两个节点，把左子节点连接在右子节点的左节点上

**如果是删除普通二叉树：**

把找到的节点与右子树最左的叶子节点上，然后通过遍历把右子树的叶子节点删掉，只返回左子节点



1. **修剪二叉搜索树：**给定一个二叉搜索树，同时给定最小边界L 和最大边界 R。通过修剪二叉搜索树，使得所有节点的值在[L, R]中 (R>=L) 。你可能需要改变树的根节点，所以结果应当返回修剪好的二叉搜索树的新的根节点。

**如果当前节点值小于最低范围，则直接返回修剪后的右子树**

**如果当前节点值大于最大范围，直接返回修剪后的左子树**

**递归处理左右子节点**

1. **把二叉搜索树转换为累加树 ：**给出二叉 搜索 树的根节点，该树的节点值各不相同，请你将其转换为累加树（Greater Sum Tree），使每个节点 node 的新值等于原树中大于或等于 node.val 的值之和。

**一个有序数组[2, 5, 13]，求从后到前的累加数组，也就是[20, 18, 13]**

**二叉搜索树从后往前遍历（右中左）然后依次把后面的值往前累加**

**Let p记录前一个节点的值**

## 二叉树的遍历方式

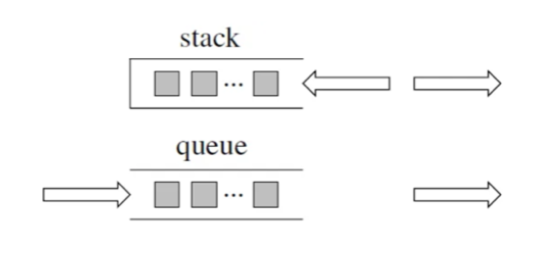
**1,递归**

**2，迭代遍历（pop(),判断栈中最后一个元素是否为null,如果是，则为下一个需要处理的节点，不是则继续入栈）（只加入非空节点）**

1. **层序遍历（shift(),取头节点，则把子节点放入队列中（for遍历当前层次的头节点））（只加入非空节点）**

**栈与队列**

**栈：先进后出 队列：先进先出**



# **232.用栈实现队列 ：**请你仅使用两个栈实现先入先出队列。队列应当支持一般队列支持的所有操作（push、pop、peek、empty）：

**使用push和pop()**

两个栈**一个输入栈（in），一个输出栈(out)**

1. **加入 in.push(x)**
2. **删除 先把out栈内的元素输出（pop()）,再把in栈的元素（pop）放入out 栈(push)，返回pop()的值**
3. **返回第一个元素值，调用方法二（this.pop()）取得返回值，再加入out栈 （push）**
4. **判断队列是否为空 如果输入栈和输出栈都为空，则队列为空**

return !this.in.length&&!this.out.length

**225. 用队列实现栈 ：请你仅使用两个队列实现一个后入先出（LIFO）的栈，并支持普通栈的全部四种操作（push、top、pop 和 empty）。**

**使用Push,shift**

**一个队列在模拟栈弹出元素的时候只要将队列头部的元素（除了最后一个元素外） 重新添加到队列尾部，此时在去弹出元素就是栈的顺序了。**

1. **加入 push**
2. **删除 每次都把队首（shift）放到队尾（push）,除了第一个，返回shift**
3. **栈顶元素，调用this.pop(),再push到队列中**
4. **返回！队列的长度**

**20. 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。

**把括号对保存在map中，遍历字符串，**

**把左括号保存在arr中，**

**如果不是左括号，就用来匹配数组最后一个值（pop()）**

**最后的返回值决定于arr的长度，为空才是true**

**1047. 删除字符串中的所有相邻重复项 :**给出由小写字母组成的字符串 S，重复项删除操作会选择两个相邻且相同的字母，并删除它们。

在 S 上反复执行重复项删除操作，直到无法继续删除。

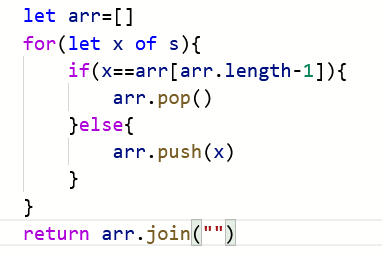
在完成所有重复项删除操作后返回最终的字符串。答案保证唯一。

**用数组arr接收结果，**

**遍历字符串**

**如果与前一个相同，则删除前一个（arr.pop），**

**如果不相等则加入arr.push()**



**150. 逆波兰表达式求值:**根据 逆波兰表示法，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。

**波兰式：前缀 逆波兰式：后缀**

**遇到数字则入栈；**s.push(parseInt(x))

**遇到算符则取出栈顶两个数字进行计算，并将结果压入栈中。**

**239. 滑动窗口最大值：**给定一个数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

**队列用于遍历数组i，队列第一个值是当前窗口的最大值的下标**

**当队列第一个元素下标小于当前窗口，则剔除**s[0]<=i-k

**当前元素比队尾元素大，则推出队尾元素所有小于的值，再加入当前值（while）**

**当下标大于窗口大小时，把队列第一个元素加入结果数组** i>=k-1

1. **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。

**用小顶堆，因为要统计最大前k个元素，只有小顶堆每次将最小的元素弹出，最后小顶堆里积累的才是前k个最大元素。**

添加：在最后一个节点添加该元素，然后**和父节点比较**大小，如果比父节点小，则父节点交换位置（从下往上移动）

删除：删除头节点，然后把最后一个子节点放置根节点，然后**与较小的子节点比较**，如果大于子节点，则交换位置（从上往下移动）

需要交换 [node1,node2]=[node2,node1]

需要获得父节点， (i-1)>>1

获得左子节点 i\*2+1

获得右子节点 i\*2+2

**把大的往下移，把小的往上移**

**上移**操作：

如果是第一个元素，不需要上移（index==0）

如果父节点存在,且小于父节点，与父节点交换位置，递归调用上移函数

**下移**操作：

获得左右子节点的下标，先判断是否小于左右子节点(**左右子节点存在**)，判断左右子节点的大小（**小的节点存在**），与较小的节点交换，再递归操作下移函数

添加：加入队尾（push）,再调用上移函数（s.length-1）

删除，把最后一个节点赋值给头节点 this.s[0]=this.s.pop()，再调用下移函数(0)

获取堆的大小 this.s.length

主函数：

**用map记录字符和次数，创建小顶堆，遍历map，把元素添加（函数）到小顶堆中，如果堆的长度大于k，就弹出（pop()）**

**双指针**

**27. 移除元素 ：给你一个数组 nums 和一个值 val，你需要 原地 移除所有数值等于 val 的元素，并返回移除后数组的新长度。**

**不要使用额外的数组空间，你必须仅使用 O(1) 额外空间并 原地 修改输入数组。**

**元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。**

**快指针遍历数组，如果快指针的值不等于val,赋值nums[s]=nums[k],且慢指针自增**

for(let k=0;k<nums.length;k++){

        if(nums[k]!=val){

            nums[s]=nums[k]

            s++

        }

    }

#### ****26. 删除有序数组中的重复项 ：****

给你一个 **升序排列** 的数组 nums ，请你 [原地](http://baike.baidu.com/item/%E5%8E%9F%E5%9C%B0%E7%AE%97%E6%B3%95) 删除重复出现的元素，使每个元素 **只出现一次** ，返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。

不要使用额外的空间，你必须在 [原地](https://baike.baidu.com/item/%E5%8E%9F%E5%9C%B0%E7%AE%97%E6%B3%95) **修改输入数组** 并在使用 O(1) 额外空间的条件下完成。

**快指针遍历数组，如果当前值与前一个值不等，赋值nums[s]=nums[k],且慢指针自增**

 for(let k=0;k<nums.length;k++){

        if(nums[k]!=nums[k-1]){

            nums[s]=nums[k]

            s++

        }

**283. 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。

**请注意** ，必须在不复制数组的情况下原地对数组进行操作。

**快指针遍历数组，如果当前值不等于0，，赋值nums[s]=nums[k],且慢指针自增**

**最后补零**

for(let i=s;i<nums.length;i++){

        nums[i]=0

    }

**844. 比较含退格的字符串 ：给定 s 和 t 两个字符串，当它们分别被输入到空白的文本编辑器后，如果两者相等，返回 true 。# 代表退格字符。**

**注意：如果对空文本输入退格字符，文本继续为空。**

**两个字符串，用两个指针遍历，分别找到字符就比较是否相等**

**记录#号的个数，有多少个指针就回退几个**

**从后往前遍历，while（s>=0||t>=0）找到两个指针的值**

**遍历字符串时有三种情况：**

**如果是#，指针自减，标记a++,**

**不是#，但是标记大于零，a>0 ,标记自减，指针自减**

**不是以上两种情况，退出循环，break**

1. **有序数组的平方 ：**给你一个按**递增** 排序的整数数组 nums，返回 **每个数字的平方** 组成的新数组，要求也按 **递增** 排序。

**首尾定义一个指针，比较其平方大小，大的加入结果arr,指针进1，**

**从arr后往前填充**

if(n>m){

            arr[i]=n

            l++

        }else{

            arr[i]=m

            r--

        }

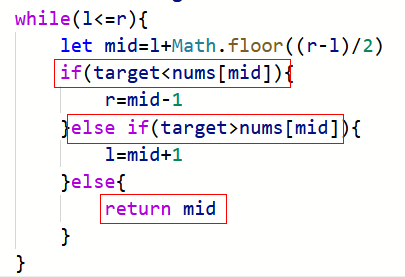
        i--

**二分查找**

1. **二分查找；**给定一个 n 个元素有序的（升序）整型数组 nums 和一个目标值 target  ，写一个函数搜索 nums 中的 target，如果目标值存在返回下标，否则返回 -1。

**这道题目的前提是数组为有序数组，同时题目还强调数组中无重复元素，**

**区间为闭区间[left,right] 则遍历的时候while (left <= right)**



**35.搜索插入位置 :**给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回**它将会被按顺序插入的位置**。

你可以假设数组中无重复元素。

示例 1:

* 输入: [1,3,5,6], 5
* 输出: 2

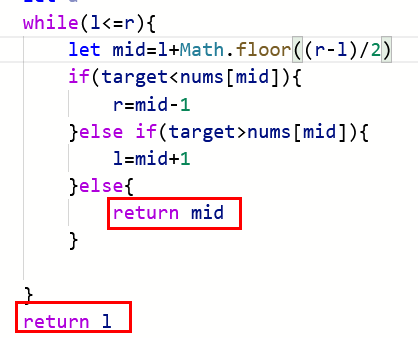
**有序数组，无重复元素，采用二分查找**

**跳出循环的两种方式，找到mid=target,返回mid**

**找不到，则依靠left和right交错越界，输出left**

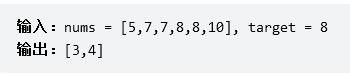
**如果target小于最小的值，left一直不动，right逐渐向left靠近，直到越界，left都为0，所以返回left**

**如果target大于最大的值，right一直不动，left向right靠近，直到越界，left为数组范围的后一位，返回left**



**34. 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。

如果数组中不存在目标值 target，返回 [-1, -1]。



**查找左边界，如果mid=target,**

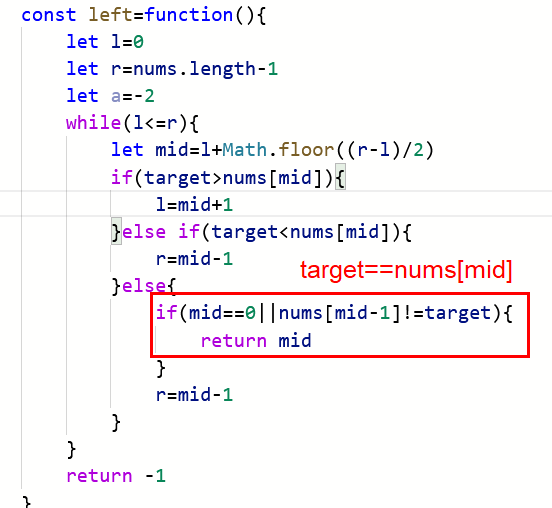
**如果mid=0，或者nums[mid-1]!=target,则当前值为第一个，return mid**

**如果不是第一个，继续查找左区间，right=mid-1**

**查找右边界，如果mid=target,**

**如果mid=nums.length-1,或者nums[mid+1]!=target,则当前值为最后一个，return mid**

**如果不是最后一个，继续查找右区间，left=mid+1**



#### **69. x 的平方根 ：**

给你一个非负整数 x ，计算并返回 x 的 **算术平方根** 。

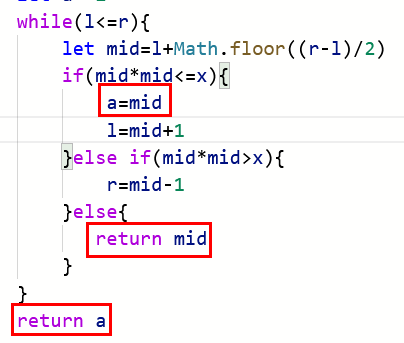
由于返回类型是整数，结果只保留 **整数部分** ，小数部分将被 **舍去 。**

**注意：**不允许使用任何内置指数函数和算符，例如 pow(x, 0.5) 或者 x \*\* 0.5 。

**如果这个整数的平方 恰好等于 输入整数，那么我们就找到了这个整数；**

**如果这个整数的平方 严格大于 输入整数，那么这个整数肯定不是我们要找的那个数；**

**如果这个整数的平方 严格小于 输入整数，那么这个整数 可能 是我们要找的那个数**



**a可能是我们要找的值，标记a，如果找不到相等的mid，就返回a**

**367. 有效的完全平方数 :**给定一个 正整数 num ，编写一个函数，如果 num 是一个完全平方数，则返回 true ，否则返回 false 。

进阶：不要 使用任何内置的库函数，如 sqrt 。

**如果找得到mid\*mid=target,返回true**

**27. 移除元素 :**给你一个数组 nums 和一个值 val，你需要 原地 移除所有数值等于 val 的元素，并返回移除后数组的新长度。

不要使用额外的数组空间，你必须仅使用 $O(1)$ 额外空间并**原地**修改输入数组。

元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

**快指针遍历数组，如果快指针!=val,则nums[s]=nums[k] s++**

1. **删除排序数组中的重复项 :**

**快指针遍历数组，如果nums[k]!=nums[k-1],则nums[s]=nums[k],s++**

1. **移动零**

**快指针遍历数组，如果nums[k]!=0,则nums[s]=nums[k],s++,**

**最后补零，s到nums.length-1**

1. **比较含退格的字符串 :**

**比较两个字符串找到的字符，**

**找字符：**

**如果为#，累计个数，指针下移**

**如果不是#，但个数大于0，个数减少，指针下移**

**除了以上两个情况，break**

**977，有序数组的平方 :**含有正负值，且从小到大排序

**定义两个指针，在数组的头和尾，较大的值（平方和）填在结果数组的后面**

**209，长度最小的子数组 ：**给定一个含有 n 个正整数的数组和一个正整数 target 。

找出该数组中满足其和 ≥ target 的长度最小的 **连续子数组** [numsl, numsl+1, ..., numsr-1, numsr] ，并返回其长度。如果不存在符合条件的子数组，返回 0 。

**连续子数组，采用滑动窗口,定义两个指针，起始指针start和滑动指针i**

**如果窗口总和小于target,扩大窗口,i++**

**如果窗口总和大于等于target，（while）移动start指针，窗口总值减少**

**本题要取的是最小的长度，初始定义最大值，之后覆盖，**

**窗口长度为滑动指针-起始位置+1**

#### **904. 水果成篮 ：**

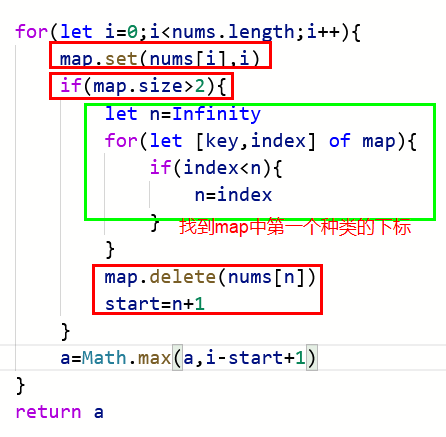
**一个数组，只能取两个不同的值，且必须连续，求连续的最大数目**

**采用滑动窗口思想**

**Map：key为nums[i],value为最后出现的下标**

**起始指针start为0，滑动指针i遍历数组，**

**（if）当种类（map.size）大于2,删除第一个种类，更新起始指针**



**76. 最小覆盖子串 ：**给你一个字符串 s 、一个字符串 t 。返回 s 中涵盖 t 所有字符的**最小子串**。如果 s 中不存在涵盖 t 所有字符的子串，则返回空字符串 "" 。

注意：

对于 t 中重复字符，我们寻找的子字符串中该字符数量必须不少于 t 中该字符数量。

如果 s 中存在这样的子串，我们保证它是唯一的答案。



**1，整理目标字符串map字符(key)出现的次数(val)，并记录种类(map.size)**

**2，定义起始指针start=0，滑动指针i遍历源字符串，累计窗口内出现的字符，**

**如果nums[i]在map中存在，则相应的次数减1，当满足次数，则种类减一**

**3，（while）当种类==0的时候，获得滑动窗口长度，**

**如果移动的值在t中有(has)，则次数+1，**

**如果map.get>0,则种类加1，**

**移动起始指针**

**需要记录最佳窗口的起始点**

var minWindow = function(s, t) {

    let map=new Map()

    for(let x of t){

        map.set(x,(map.get(x)||0)+1)

    }

    let count=map.size

    let start=0

**let a=Infinity  //长度**

**let flag=Infinity  //标记起始点**

    for(let i=0;i<s.length;i++){

        if(map.has(s[i])){

            map.set(s[i],map.get(s[i])-1)  //单个字符次数减1

        }

        if(map.get(s[i])==0){

            count--        //种类减1

        }

        while(count==0){

            //记录长度，和起始点

            let len=i-start+1

            if(a>len){

                a=len

**flag=start**

            }

            if(map.has(s[start])){

                map.set(s[start],map.get(s[start])+1)  //起始位置的是map中的字符，次数+1

            }

            if(map.get(s[start])>0){  //单词次数大于0.种类count加1

                count++

            }

            start++

        }

    }

    return s.substr(flag,a)

};

**！！！！高频**

1. **螺旋矩阵II ：**给定一个正整数 n，生成一个包含 1 到 n^2 所有元素，且元素按顺时针顺序螺旋排列的正方形矩阵。

示例:

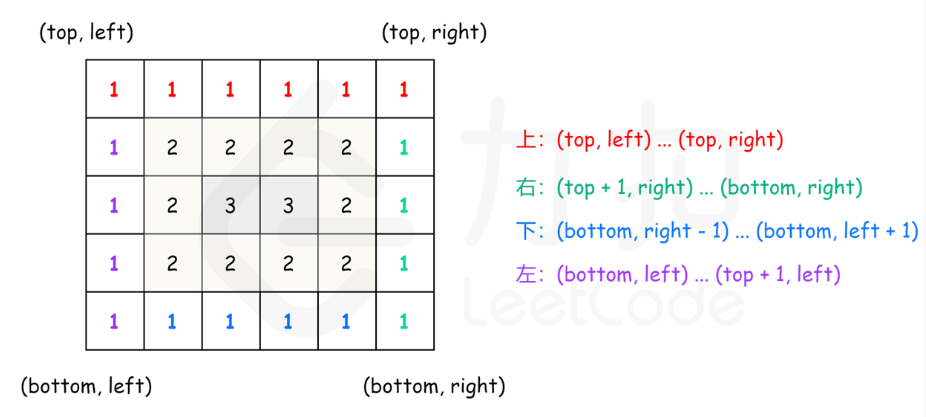
输入: 3 输出: [ [ 1, 2, 3 ], [ 8, 9, 4 ], [ 7, 6, 5 ] ]

**54. 螺旋矩阵**

给你一个 m 行 n 列的矩阵 matrix ，请按照 顺时针螺旋顺序 ，返回矩阵中的所有元素。

**剑指 Offer 29. 顺时针打印矩阵**

输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字。



**链表**

**203.移除链表元素 ：**题意：删除链表中等于给定值 val 的所有节点。

示例 1：  
输入：head = [1,2,6,3,4,5,6], val = 6  
输出：[1,2,3,4,5]

**Temp.next=temp.next.next**

**防止删除的是头节点，建立一个头节点指向链表头**

1. **设计链表 ：**

get(index)：获取链表中第 index 个节点的值。如果索引无效，则返回-1。

**index大于链表长度，或者是头节点或尾节点 ，需要单独处理**

**如果是1到this.size-2范围内的，需要找到index, while(i<index)**

addAtHead(val)：在链表的第一个元素之前添加一个值为 val 的节点。插入后，新节点将成为链表的第一个节点。

**如果链表为空：head和tail都指向当前节点，长度加1**

**链表不为空：创建节点，连接到原头节点，更新头节点，长度加1**

addAtTail(val)：将值为 val 的节点追加到链表的最后一个元素。

**如果链表为空：同上**

**链表不为空：创建新节点，链表尾指针指向该节点，并更新尾指针和长度**

addAtIndex(index,val)：在链表中的第 index 个节点之前添加值为 val 的节点。如果 index 等于链表的长度，则该节点将附加到链表的末尾。如果 index 大于链表长度，则不会插入节点。如果index小于0，则在头部插入节点。

**超出链表长度，返回false**

**小于等于链表长度，插入头节点（调用**addAtHead(val)**）**

**Index=size，插入尾节点（调用**addAtTail(val)**）**

**找到要插入的前一个节点，**while(i<index-1)，**创建新节点，连接，更新长度**

deleteAtIndex(index)：如果索引 index 有效，则删除链表中的第 index 个节点。

超出范围，return

删除头节点，更新头节点，长度减1 ，return

找到要删除的前一个节点while(i<index-1)  temp.next=temp.next.next

如果是尾节点，还要更新tail，长度减1

**206.反转链表**

题意：反转一个单链表。

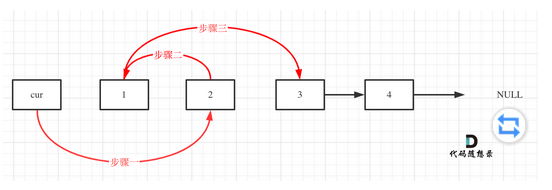
示例: 输入: 1->2->3->4->5->NULL 输出: 5->4->3->2->1->NULL

**定义一个空指针p，把头节点的next指向p,及时更新两个指针**

**24. 两两交换链表中的节点：**给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

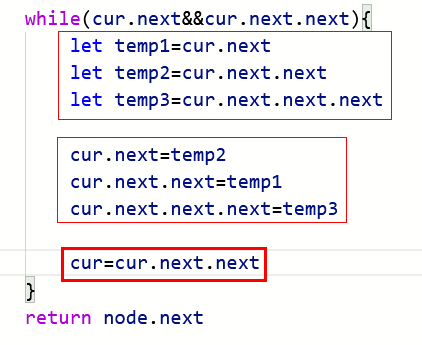
**改变指针的指向即可**



**需要交换的两个节点不能为空**

**先把虚拟节点加上，先保存原顺序的三个节点，在按照规定顺序赋值**

**注意：更新cur节点，是要在更改之后的链表上后移两位 cur=cur.next.next**



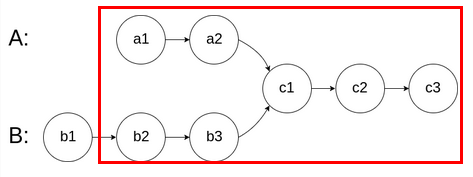
**19.删除链表的倒数第N个节点 :**给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

进阶：你能尝试使用一趟扫描实现吗？

**建立虚拟头节点，防止删除的是头节点**

如果要删除倒数第n个节点，**让fast移动n+1步** while(i<n+1)，然后让fast和slow同时移动，**直到fast指向链表末尾** while(cur)。删掉slow所指向的下一个节点就可以了。

**面试题 02.07. 链表相交 ：**给你两个单链表的头节点 headA 和 headB ，请你找出并返回两个单链表相交的起始节点。如果两个链表没有交点，返回 null 。

相等的长度

**遍历两个链表，获得长度while(cur1.next)和长度差a，让长的链表先走while(i<a)**

**然后两个链表比较，如果节点相等，返回该节点**

**遍历链表，如果有虚拟头节点while(cur.next)**

**没有虚拟头节点while(cur)**

1. **环形链表II:**题意： 给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。

**循环的条件（链表不为空，且快指针的next不为空）**

**定义快慢指针，如果相遇，则有环**

**找入口：从头节点和相遇节点定义两个指针，遍历while(p1!==p2)，返回相等的节点**

**242.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: 输入: s = "anagram", t = "nagaram" 输出: true

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

**说明:** 你可以假设字符串只包含小写字母

**建立一个数组，用于保存其字符出现的次数，**

**一个字符串累计arr[x.charCodeAt()-a]++**

**一个减少 arr[x.charCodeAt()-a]--，最后为0**

charCodeAt() //获得字符的编码

a的编码为0，其余字符的编码-a的码数，保存在数组中

1. **赎金信：**

给你两个字符串：ransomNote 和 magazine ，判断 ransomNote 能不能由 magazine 里面的字符构成。

如果可以，返回 true ；否则返回 false 。

magazine 中的每个字符只能在 ransomNote 中使用一次。

**遍历m，累计在arr中，遍历r,递减，如果存在小于0的值，为false**

**源字符串累加，目标字符串减少**

1. **字母异位词分组**

给你一个字符串数组，请你将 字母异位词 组合在一起。可以按任意顺序返回结果列表。

字母异位词 是由重新排列源单词的字母得到的一个新单词，所有源单词中的字母通常恰好只用一次。

**1,遍历每个字符串，转化为数组排序之后再转化为字符串（得到排序后的字符 串key）**key=Array.from(str).sort().toString()

**2,排序后的字符作为key,在map中查找是否有相同的，**

  list=map.get(key)?map.get(key):new Array()

**3,再把排序之前的字符串保存到list中，重新把list保存在map中，键为key, 值为list**

**438.找到字符串中所有字母异位词**

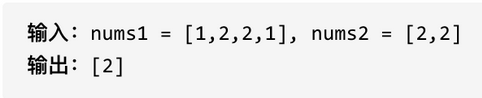
给定两个字符串 s 和 p，找到 s 中所有 p 的 异位词 的子串，返回这些子串的起始索引。不考虑答案输出的顺序。

异位词 指由相同字母重排列形成的字符串（包括相同的字符串）。

**1,定义窗口大小为目标字符串t的长度，**

**2，遍历字符串s，while(s[k]),统计t的字符和次数，截取s字符串，用于匹配**

1. **两个数组的交集：**题意：给定两个数组，编写一个函数来计算它们的交集。相同的值，只保存一次，用map



**遍历其中一个字符串，加入map,遍历另一个字符串，检查map.has(),如果存在则加入set（）中**

#### [350. 两个数组的交集 II](https://leetcode-cn.com/problems/intersection-of-two-arrays-ii/)

给你两个整数数组 nums1 和 nums2 ，请你以数组形式返回两数组的交集。返回结果中每个元素出现的次数，应与元素在两个数组中都出现的次数一致（如果出现次数不一致，则考虑取较小值）。可以不考虑输出结果的顺序。

**考虑次数（较小值）**

**遍历短数组，保存字符和出现的次数在map中**

map.set(x,(map.get(x)||0)+1)

**遍历长数组，如果map.has()&&map.get>0,则加入结果数组（**记得次数减1**）**

if(map.has(x)&&map.get(x)>0){

            arr.push(x)

             map.set(x,(map.get(x)||0)-1)

        }

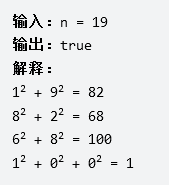
**第202题. 快乐数：编写一个算法来判断一个数 n 是不是快乐数。**

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。

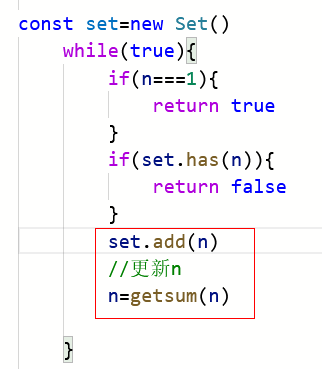
然后重复这个过程直到这个数变为 1，也可能是 **无限循环** 但始终变不到 1。

如果这个过程 结果为 1，那么这个数就是快乐数。

如果 n 是 快乐数 就返回 true ；不是，则返回 false 。



**退出循环的条件，结果为1，或者出现重复的值set.has()**



1. **两数之和**

给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出 和为目标值 target 的那 两个 整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。

你可以按任意顺序返回答案。

**遍历数组，如果数组中有target-map.get(x),**

**返回**[i,map.get(target-nums[i])]**，**

**没有则加入map**

**map的key为nums[i],value为下标**

**第454题.四数相加II ：**给你四个整数数组 nums1、nums2、nums3 和 nums4 ，数组长度都是 n ，请你计算有多少个元组 (i, j, k, l) 能满足：

0 <= i, j, k, l < n

nums1[i] + nums2[j] + nums3[k] + nums4[l] == 0

map的key为sum,value为该sum出现的次数

两两为一组，求和，sum1保存在map中**，** map.set(sum1,(map.get(sum1)||0)+1)

判断map中是否含有（0-sum2）,如果含有，**次数加上（0-sum2）出现的次数**

if(map.has(0-sum2)){

      count+=map.get(0-sum2)||0

   }

**第15题. 三数之和 ：**给你一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a，b，c ，使得 a + b + c = 0 ？请你找出所有和为 0 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

**长度小于3，直接返回**

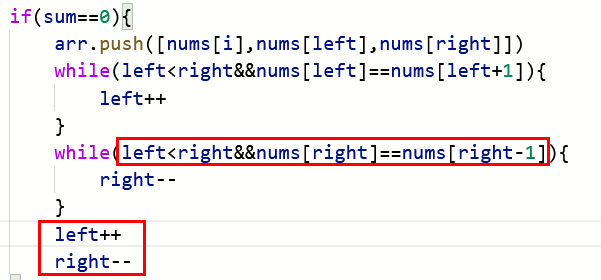
**定义三个指针，一个指针i遍历数组，另外两个指针遍历i之后的数组（首尾指针）**

**1,需要进行排序，因为是排序过的数组，遍历数组，如果nums[i]>0，则直接break**

**2，** while(left<right)**计算三个指针的和，**

**3，如果大于0，right指针左移，如果小于0，左指针右移**

**4，如果等于0，把结果加入数组arr,接着去重左右指针，再前进一位**



**两数之和 就不能使用双指针法，要求返回的是索引下标**

**如果1.两数之和要求返回的是数值的话，就可以使用双指针法了。**

**第18题. 四数之和：**题意：给定一个包含 n 个整数的数组 nums 和一个目标值 target，判断 nums 中是否存在四个元素 a，b，c 和 d ，使得 a + b + c + d 的值与 target 相等？找出所有满足条件且不重复的四元组。

**同一个数组，找出4个值之和等于target**

**定义4个指针，在三数之和的基础上再定义一个头指针**

for(let i=0;i<nums.length-3;i++){

        if(i>0&&nums[i]==nums[i-1]){

            continue

        }

        for(let x=i+1;x<nums.length-2;x++){

            if(x>i+1&&nums[x]==nums[x-1]){

                continue

            }

**344.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 s 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。

**定义两个指针指向首尾交换字符（while(l<r)）**

#### **541. 反转字符串II :**

给定一个字符串 s 和一个整数 k，从字符串开头算起，每计数至 2k 个字符，就反转这 2k 字符中的前 k 个字符。

* 如果剩余字符少于 k 个，则将剩余字符全部反转。
* 如果剩余字符小于 2k 但大于或等于 k 个，则反转前 k 个字符，其余字符保持原样。

**for循环，每次跳转2k个，定义头尾指针，如果尾指针的选择r=min(i+k,len)-1**

**剑指Offer 05.替换空格 :**请实现一个函数，把字符串 s 中的每个空格替换成"%20"。

示例 1： 输入：s = "We are happy."  
输出："We%20are%20happy."

**遍历字符串累计空格数，扩大数组arr长度,**arr.length+count\*2-1

**定义两个指针从队尾开始填充**

 while(t>=0){

        if(arr[t]==' '){

            arr[p--]="0"

            arr[p--]='2'

            arr[p--]="%"

            t--

        }else{

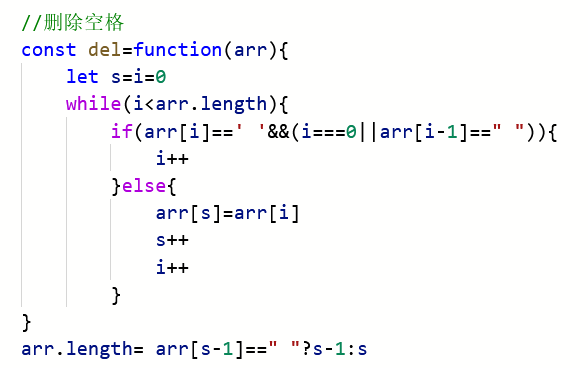
            arr[p--]=arr[t--]

        }

**151.翻转字符串里的单词 ：**给定一个字符串，逐个翻转字符串中的每个单词。

示例 1：  
输入: "the sky is blue"  
输出: "blue is sky the"

**删除字符串前的空格和字符串之间只留一个空格：删除字符串末尾的空格：**



**反转整个字符串，反转单词**

//反转单词

    let start=0

    for(let i=0;i<=arr.length;i++){

        if(arr[i]==' '||i==arr.length){

            resever(arr,start,i-1)

            start=i+1

        }

}

**剑指Offer58-II.左旋转字符串:**字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。请定义一个函数实现字符串左旋转操作的功能。比如，输入字符串"abcdefg"和数字2，该函数将返回左旋转两位得到的结果"cdefgab"。

**反转前n个，反转n个之后的，再反转整个数组**

**28. 实现 strStr() ：**实现 strStr() 函数。

给定一个 haystack 字符串和一个 needle 字符串，在 haystack 字符串中找出 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。

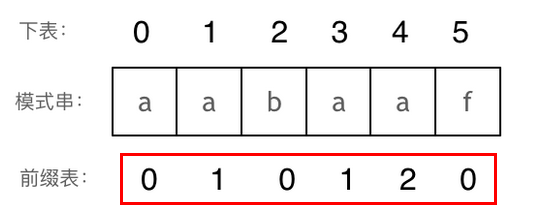
**理论**

**前缀是指不包含最后一个字符的所有以第一个字符开头的连续子串**。

**后缀是指不包含第一个字符的所有以最后一个字符结尾的连续子串**。

**前缀和后缀都是从前往后读，只是相应的忽略第一个或最后一个**

**填充前缀表：前缀表的值是当前字符串最大的共同前缀后缀的长度**



**比如aaba,前缀有：a,aa,aab 后缀有：aba,ba,a 相同的只有a，长度为1**

**匹配两个字符串：**



f不匹配，移动到b

**下标5之前这部分的字符串（也就是字符串aabaa）的最长相等的前缀 和 后缀字符串是 子字符串aa**

**aabaa在两个字符串中已经匹配，源字符串的后缀(aa)与目标字符串的前缀(aa)相同，所以目标字符串的前缀不用再匹配，直接对比目标字符串前缀之后的字符b**

**由图可知，当下标5的位置不匹配，所以把目标字符串的指针回退到前缀后一个位置（前缀表中的值）b**

KMP算法的时间复杂度是$O(n+m)$的。

**代码流程：**

**构建前缀表：**

定义两个指针i和j，j指向前缀起始位置，i指向后缀起始位置。

next[i] 表示 i（包括i）之前最长相等的前后缀长度（其实就是j）

j每次都从下标0开始，遍历i(遍历后缀的起始位置)

1. **初始化第一个字符的**最长相等的前后缀长度为0
2. 遍历后缀起始位置，
3. 如果nums[i]！=nums[j]，前缀指针j回退到前一个元素next[]值的下标，

  j=next[j-1]

**类似于前缀字符串和后缀字符串的匹配，如果当前值不相等，则前缀j回退**

1. 如果当前nums[i]=nums[j] 则前缀指针j++
2. 记录当前最长的值

**匹配两个字符串：**

1，定义两个指针分别指向两个字符串的首部，遍历源字符串

2，如果相等，指针前移，如果不相等，目标字符串指针需要回退，

3，如果目标字符串的指针位置等于目标字符串的长度，则匹配成功，返回起始位置源字符串的下标-长度+1

x-needle.length+1

**459.重复的子字符串 ：**给定一个非空的字符串，判断它是否可以由它的一个子串重复多次构成。给定的字符串只含有小写英文字母，并且长度不超过10000。

示例 1:  
输入: "abab"  
输出: True  
解释: 可由子字符串 "ab" 重复两次构成。

**构建next前缀表，next表的最后一个值不为0，如果这个周期可以被整除，就说明整个数组就是这个周期的循环。**

**一个周期的长度：数组长度减去最长相同前后缀的长度：**len-next[len-1]