**Algorithm Notes**

**Hints0：审题！当代码没问题时，一定是理解错了题意。**

**Hints1: 学会提取问题的本质，代码逻辑冗余往往浮于表象。**

**Hints2：摆脱顺序思考的惯性，学会还原问题、分解问题、处理特殊情况。**

**Hints3: 遇到复杂的边界情况，尽量分步处理，且边界选择需方便后续处理。**

**Hints4: 优美的代码结最重要的不是精简，而是清晰的逻辑**

**Hints: 虽python自动处理边界、数据转化等，但为严谨依据c++处理**

目录

[Python 易错语法 4](#_Toc49068013)

[1 Search 4](#_Toc49068014)

[1.1 Binary Search 4](#_Toc49068015)

[4 Median of two sorted array 5](#_Toc49068016)

[153 find minimum in rotated sorted array 6](#_Toc49068017)

[410 \*Split Array Largest Sum 7](#_Toc49068018)

[454 4sum II 8](#_Toc49068019)

[1.2 Breadth-first Search 9](#_Toc49068020)

[126. \*Word Ladder II 9](#_Toc49068021)

[207 \*课程表 11](#_Toc49068022)

[1.3 DFS 12](#_Toc49068023)

[46 全排列 13](#_Toc49068024)

[面试题38 字符串的排列 14](#_Toc49068025)

[207 课程表 15](#_Toc49068026)

[399 除法求值（？并查集解法） 15](#_Toc49068027)

[1367 二叉树中的列表 16](#_Toc49068028)

[1.4 滑动窗口 17](#_Toc49068029)

[3 无重复字符的最长子串 17](#_Toc49068030)

[438 找出字符串中所有字母异位词 18](#_Toc49068031)

[1.5 字符串匹配 19](#_Toc49068032)

[KMP(实现Str()str()) 19](#_Toc49068033)

[2 Dynamic Programming 20](#_Toc49068034)

[2.1 Base 21](#_Toc49068035)

[72 最小编辑距离 21](#_Toc49068036)

[152. Maximum Product Subarray 22](#_Toc49068037)

[337 \*打家劫舍III 23](#_Toc49068038)

[1074 \*Number of Submatrices That Sum to Target 24](#_Toc49068039)

[2.2 状态表示 25](#_Toc49068040)

[\*1349 参加考试的最大学生数 26](#_Toc49068041)

[5337 每个元音包含偶数次的最长子串 27](#_Toc49068042)

[2.3 股票问题全解 28](#_Toc49068043)

[123 Best Time to Buy and Sell Stock III 29](#_Toc49068044)

[2.4 背包问题 30](#_Toc49068045)

[416 分割等和子集 30](#_Toc49068046)

[2.5 数位DP 31](#_Toc49068047)

[233 数字1的个数 32](#_Toc49068048)

[3 分治法(Divide and Conquer) 33](#_Toc49068049)

[932 \*Beautiful Array 33](#_Toc49068050)

[312 戳气球 34](#_Toc49068051)

[4 排序 35](#_Toc49068052)

[4.1 快排 35](#_Toc49068053)

[4.2 堆（优先队列） 36](#_Toc49068054)

[295 数据流的中位数 38](#_Toc49068055)

[347 前k个高频元素 39](#_Toc49068056)

[480 滑动窗口中位数 40](#_Toc49068057)

[1353 最多可以参加会议的数目 41](#_Toc49068058)

[4.3 桶排序（计数排序） 43](#_Toc49068059)

[347 前k个高频元素 43](#_Toc49068060)

[4.4 循环排序 43](#_Toc49068061)

[41 first missing positive 43](#_Toc49068062)

[5 Tree 44](#_Toc49068063)

[5.1 二叉树 44](#_Toc49068064)

[线索二叉树（莫里斯遍历） 44](#_Toc49068065)

[236 二叉树的最近公公祖先 45](#_Toc49068066)

[297 二叉树的序列化与反序列化 47](#_Toc49068067)

[5346 二叉树中的列表 49](#_Toc49068068)

[5.2 并查集 50](#_Toc49068069)

[200 岛屿数量(并查集) 50](#_Toc49068070)

[5.3 线段树 52](#_Toc49068071)

[315 计算右侧小于当前元素的个数 53](#_Toc49068072)

[6 Stack 53](#_Toc49068073)

[42 Trapping Rain Water 53](#_Toc49068074)

[394 字符串解码 54](#_Toc49068075)

[7 Greedy 54](#_Toc49068076)

[45 \*Jump Game 2 55](#_Toc49068077)

[406 根据身高重建队列 55](#_Toc49068078)

[1353 最多可以参加会议的数目 56](#_Toc49068079)

[5359 最大的团队表现值 56](#_Toc49068080)

[8 Pointer 57](#_Toc49068081)

[8.1 双向指针/指针数组 58](#_Toc49068082)

[15 3 sum 58](#_Toc49068083)

[42 Trapping Rain Water 58](#_Toc49068084)

[8.2 滑动窗口 60](#_Toc49068085)

[76 Minimum Window Substring 60](#_Toc49068086)

[239 滑动窗口最大值 61](#_Toc49068087)

[8.3 快慢指针 62](#_Toc49068088)

[234 回文链表 62](#_Toc49068089)

[287 寻找重复数 63](#_Toc49068090)

[9 Concepts and Observations 64](#_Toc49068091)

[9.1 辅助栈 64](#_Toc49068092)

[面试题30. 包含min函数的栈 64](#_Toc49068093)

[5 Longest Palindromic Substring 65](#_Toc49068094)

[32 最长有效括号 66](#_Toc49068095)

[146 \*LRU缓存机制 68](#_Toc49068096)

[148 排序链表 71](#_Toc49068097)

[189 Rotate Array 72](#_Toc49068098)

[229 Majority Element II 74](#_Toc49068099)

[260 只出现一次的数字 74](#_Toc49068100)

[计算右侧小于当前元素的个数 75](#_Toc49068101)

[795 Number of Subarrays with Bounded Maximum 79](#_Toc49068102)

[1238 循环码列 80](#_Toc49068103)

[1401 圆和矩形是否重叠 81](#_Toc49068104)

[10 Math 82](#_Toc49068105)

[31 Next Permutation 83](#_Toc49068106)

[136. 只出现一次的数字 84](#_Toc49068107)

[160 相交链表 84](#_Toc49068108)

[264 Ugly Number II 85](#_Toc49068109)

[335 Self Crossing 86](#_Toc49068110)

[372 超级次方 87](#_Toc49068111)

[523 Continuous Subarray Sum 88](#_Toc49068112)

[650 2 Keys Keyboard 89](#_Toc49068113)

[866 Prime Palindrome 90](#_Toc49068114)

[1354 多次求和构造目标数组 91](#_Toc49068115)

[5383 给Nx3网格涂色的方案数 92](#_Toc49068116)

[面试题62 约瑟夫环 93](#_Toc49068117)

# Python 易错语法

1. str 对象不支持item assignment，如s=’123’;s[1]=’4’是错误的。
2. 最大（小）堆的末尾并不保证是最小（大）的
3. “e”split(‘e’)后为[“”,“”]
4. None在python中也是一个对象，可以作为字典的键
5. s = ‘love’, s[3:3]不等于空也不等于None，而是空字符串“”
6. 3 and 4 语句的结果为4，0 and 4语句的结果为0。

7 list和dict都是对象，其一对一的赋值都是引用，会造成变量更改混乱（

list\_a=list\_b

list\_a.append(‘d’)

然后b也会添加’d‘

8． 12//10=1， -12//10=-2

又在python中 a%b = a-b\*(a//b)，所以-12%10 = 8。

所以一般负数整数可以变换为： -12整除10=-（-12//-10）=-1或int(float(-12)/10)

对应的取余：-12对10取余可以转变为：-12%-10=-2。

# 计算机结构

1 取模比乘法、除法节约时间

# Search

**1 在搜索数列中所有可能的组合情况时，无论是深度优先搜索还是广度优先搜索，都只是有效的拓扑结构遍历方式，并不能保证每一步搜索到必要性，为了节约时间，可以结合动态规划。**

**2 DFS 和 BFS 都是常用来遍历搜索树或图的算法。二叉树中的前序、中序和后序遍历都属于DFS，层次遍历属于BFS。一般能用DFS解决的问题也能用BFS解决。**

**3 只要用递归，涉及到对结果重用时，都是先深入，再处理。（如果先处理再深入，那么处理时没有深入的结果去重用）**

**4 对数列操作时，统一下标处理的逻辑，是0～n-1还是1～n**

## Binary Search

**对于基于数列需要递归的二分法，为了节约内存，最好是传递划分数列的下标，而不是子数列。**

### 4 Median of two sorted array

Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

**Example 1:**

nums1 = [1, 3]

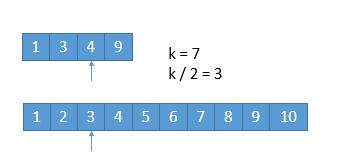
nums2 = [2]

The median is 2.0

**核心思想：** 本质是一个找两个序列中第k大的数的问题，但注意：这里是有序序列， 所以通过比较两个序列k/2处的值，能够判断必有一个序列的前k/2的子序列不包含第k大的数（如下图），去除掉该子序列，问题转变为找新的两个有序序列中第k-k/2的值。

特殊情况下，如果两个序列的长度和为偶数，那么还需要找第k/2+1的数，对此，只要在找到k/2大的数后在两个数列中寻找最近的数即可。

此外，为了方便判断，需提前使得第一个数列的长度不大于第二个数列。



**核心代码：**

m, n = len(nums1), len(nums2)

if m>n:

    return self.findMedianSortedArrays(nums2, nums1)

mid = int((m+n+1)/2)

def df(s1, s2, k, flag):

    if s1==m:

        return nums2[s2+k-1] if not flag else sum(nums2[s2+k-1:s2+k+1])/2

    if k==1:

        if flag:

            if nums1[s1]<nums2[s2]:

                one = nums1[s1]

                second = min(nums1[s1+1], nums2[s2]) if s1<m-1 else nums2[s2]

            else:

                one = nums2[s2]

                second = min(nums2[s2+1], nums1[s1]) if s2<n-1 else nums1[s1]

            return (one+second)/2

        else:

            return min(nums1[s1], nums2[s2])

    else:

        k1 = min(m-s1, int(k/2))

        k2 = k-k1

        if nums1[s1+k1-1]<nums2[s2+k2-1]:

            return df(s1+k1, s2, k-k1, flag)

        else:

            return df(s1, s2+k2, k-k2, flag)

return df(0, 0, mid, (m+n)%2==0)

### 153 find minimum in rotated sorted array

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

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

请找出其中最小的元素。

你可以假设数组中不存在重复元素。

示例 1:

输入: [3,4,5,1,2]

输出: 1

示例 2:

输入: [4,5,6,7,0,1,2]

输出: 0

**核心思想：**

这个问题的本质是升序，基于此二分法，如果某个子序列不是升序，那一定包含因为旋转所产生的首尾断点，如果是升序，那么可能包含首元素即为最小值，也可能在另一个子序列中，为了方便首元素和另一个子序列的合并处理，将判断对象设定为右子序列。

**核心代码：**

 def findMin(self, nums: List[int]) -> int:

        n = len(nums)

        mi, ma = 0, n-1

        while mi<ma:

            mid = (mi+ma)//2

            if nums[mid]>nums[ma]:

                mi = mid+1

            else:

                ma = mid

        return nums[mi]

进一步思考，如果有重复元素？

问题本质由升序变为非降序，其他分析不变，如果子序列降序，则一定包含首尾断点，如果升序，如果相等，减去另一个相等的元素即可。

while mi<ma:

            mid = (mi+ma)//2

            if nums[mid]<nums[ma]:

                ma = mid

            elif nums[mid]==nums[ma]:

                ma = ma-1

            else:

                mi = mid+1

### 410 \*Split Array Largest Sum

Given an array which consists of non-negative integers and an integer *m*, you can split the array into *m* non-empty continuous subarrays. Write an algorithm to minimize the largest sum among these *m*subarrays.

**Note:**  
If *n* is the length of array, assume the following constraints are satisfied:

* 1 ≤ *n* ≤ 1000
* 1 ≤ *m* ≤ min(50, *n*)

**Examples:**

Input:

**nums** = [7,2,5,10,8]

**m** = 2

Output:

18

**核心思想：**对max output可能的范围二分搜索，然后对每一个二分的output检验。

**二分不仅可以搜索数列，还可以搜索目标变量的范围。**

**核心代码：**

lo, hi = max(nums), sum(nums)

while lo < hi:

mid, sm, cnt = lo + (hi-lo)//2, 0, 1

for n in nums:

sm, cnt = (n,cnt+1) if sm+n>mid else (sm+n,cnt)

lo, hi = (mid+1,hi) if cnt>m else (lo,mid)

return lo

### 454 4sum II

Given four lists A, B, C, D of integer values, compute how many tuples (i, j, k, l) there are such that A[i] + B[j] + C[k] + D[l] is zero.

**Input:**

A = [ 1, 2]

B = [-2,-1]

C = [-1, 2]

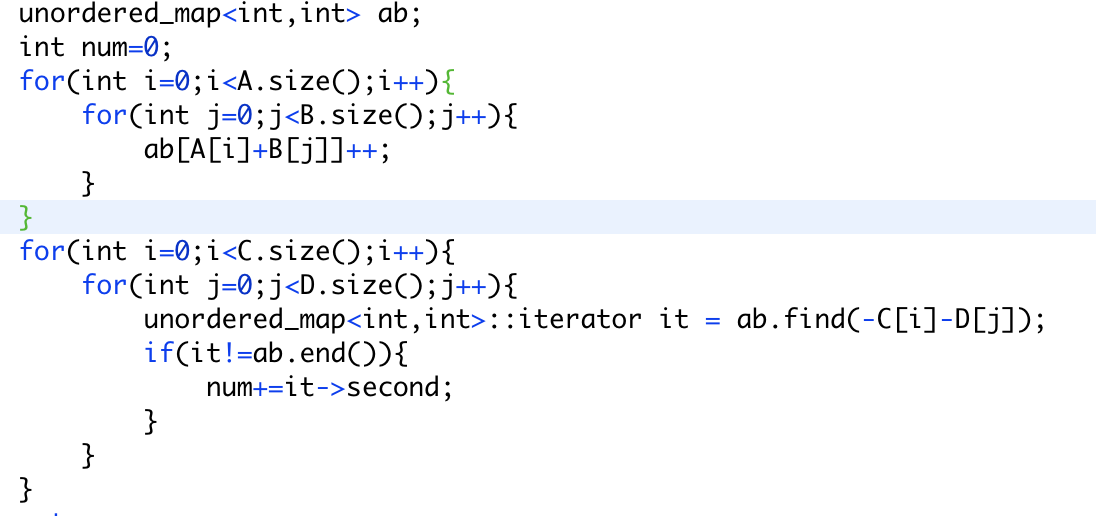
D = [ 0, 2]

**核心思想：**

第一反应是用动态规划（本质是层次迭代的方法），但是时间复杂度为n的4次方。

进一步思考，此问题的本质是加和，且是有目标值的加和，那么知道其中一子组合的值，剩余子组合的目标值（相减）也就知道了，所以该问题的数列可以一分为二，先求其中两个子序列的所有可能值，存为字典（键为可能值，值为出线次数），然后对另外两个序列的所有可能值，判断与目标值相减后的差是否在字典中，如果在，提取出对应数量即可。

扩展：由于给定目标值的加和问题，只有在加和因子数量为2时才能一一对应，所以数列只能二分一次，即如果数列数量继续增加，该方法并不能有效减少问题复杂度。



## 1.2 Breadth-first Search

**1 BFS常用队列来实现。**

### 126. \*Word Ladder II

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) from *beginWord* to *endWord*, such that:

1. Only one letter can be changed at a time
2. Each transformed word must exist in the word list. Note that *beginWord* is *not* a transformed word.

**Note:**

* Return an empty list if there is no such transformation sequence.
* All words have the same length.
* All words contain only lowercase alphabetic characters.
* You may assume no duplicates in the word list.
* You may assume *beginWord* and *endWord* are non-empty and are not the same.

**Example 1:**

**Input:**

beginWord = "hit",

endWord = "cog",

wordList = ["hot","dot","dog","lot","log","cog"]

**Output:**

[

["hit","hot","dot","dog","cog"],

  ["hit","hot","lot","log","cog"]

]

**核心思想：**

1 由于涉及到最短的路径，所以使用广度优先搜索，不然会有冗余计算

2 由于每一层的搜索涉及到多个结果的变换，所以为了避免搜索的不同层的混淆，需要在每层搜索前指定界限。对应于核心代码第4行。

**核心代码：**

lenWord = len(beginWord)

neighbor\_dict = collections.defaultdict(list)

for word in wordList:

for i in range(lenWord):

neighbor\_dict[word[:i]+'\*'+word[(i+1):]].append(word)

q = collections.deque()

q.append([beginWord])

found = False

res = []

visited = set([beginWord])

while q and not found:

cur\_len = len(q)

local\_visited = set()

for i in range(cur\_len):

path = q.popleft()

cur\_word = path[-1]

for j in range(lenWord):

for word in neighbor\_dict[cur\_word[:j]+'\*'+cur\_word[(j+1):]]:

if endWord == word:

res.append(path+[endWord])

found = True

elif word not in visited:

q.append(path+[word])

local\_visited.add(word)

visited = visited.union(local\_visited)

return res

注意：这个题的关键是证明BFS每一层不会加入同一个单词。

可能会加入同一个单词对应俩种情况：

1 如果是对应位置不同的两个上一步单词：bfs, bas，可能相同的情况也只能是b\*s的情况（因为同时与两个单词只相差一个字母就相同的情况只能是b\*s）,但是明显，产生bfs、bas就模式只能是b\*a，也就是说b\*s模式已经被用过，其对应的单词都已经被visit过

2.befa baca 对应的模式b\*fa 与ba\*a可能对应同一个单词bafa。但是由一个初始单词b\*\*a发展为有两个不同字母的单词对应最少需要两步，而这两步每一步都会包含bafa的情况，所以bafa肯定也被visit过

总结起来，主要因为bfs是寻找单词到单词之间变换的最短路径，如果两个不同单词对应的模式会产生相同单词，那么初始单词到这个相同单词的路径肯定小于等于这两个不同的单词，那么这个相同单词肯定被visit过。

### 207 \*课程表

现在你总共有 n 门课需要选，记为 0 到 n-1。

在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们: [0,1]

给定课程总量以及它们的先决条件，判断是否可能完成所有课程的学习？

示例 1:

输入: 2, [[1,0]]

输出: true

解释: 总共 2 门课程。学习课程 1 之前，你需要完成课程 0。所以是可能的。

示例 2:

输入: 2, [[1,0],[0,1]]

输出: false

解释: 总共有 2 门课程。学习课程 1 之前，你需要先完成​课程 0；并且学习课程 0 之前，你还应先完成课程 1。这是不可能的。

**核心思想：**

本质是一个判断DAG中是否存在环的问题，两种判断方法，bfs, dfs，这里讲述bfs, dfs在[后面dfs](#_207_课程表_1)小节中再讲。

bfs解法的思路主要在于从入读为0的节点（不可能在环中）开始判断每个子节点的入度，当遍历到子节点时，子节点入度减1， 然后如果子节点入度为0，则加入到队列，后续继续深入，如果不为0，不加入队列，可能有两种情况1）在环中，且为环的入口，2）有另外的父节点指向它，如果是前者，那么该环的所有节点不会再被遍历到，于是BFS最后会剩下节点，从而判断；如果是后者，等之后的父节点遍历过来即可。

**简单而言基本思想为：DAG中每个节点的入度都有迹可循，即每个入度都能由没有入度的根结点便利得到， 如果不是，则一定有环，不是DAG。**

**核心代码：**

        indegrees = [0 for \_ in range(numCourses)]

        adjacency = [[] for \_ in range(numCourses)]

        queue = []

        for cur, pre in prerequisites:

            indegrees[cur] += 1

            adjacency[pre].append(cur)

        for i in range(len(indegrees)):

            if not indegrees[i]: queue.append(i)

        while queue:

            pre = queue.pop(0)

            numCourses -= 1

            for cur in adjacency[pre]:

                indegrees[cur] -= 1

                if not indegrees[cur]: queue.append(cur)

        return not numCourses

## 1.3 DFS

**1. DFS两种实现思路：**

**1）使用全局变量+回溯（即该变量在深度优先搜索之前更新，之后恢复），此为由浅往深对结果更新（能避免将记录变量作为参数传递）。**

**2）也可以传递子变量+返回子结果，此时无需用回溯。**

一般每个DFS问题都能同时由这两种思路解决(只有对于不放回的搜索不能用第二种思路)，不过随具体问题不同，两种问题难易不一。

**DFS有两种实现方式：递归和栈，**或者有的问题所有子问题的结构都呈现出来，如棋盘、数独会给出对应数组，依照数组搜索、回溯即可，无序递归。**对于树用栈实现DFS会损失掉父亲节点信息，**如果需要，要使用另外的数据结构存储父节点信息**。**

**2 在有严格层级关系（不同层之间的状态不互相包含）的DFS中，设计一般可以严格的按层级搜索，不会重复**（如果重复，往往代码逻辑有冗余，见5346），反之，如果没有严格的层级关系，往往会重复搜索，这时需要使用缓存避免重复的搜索。

**3．一层DFS中，能同时处理的情况越多，相对而言越节约时间和空间。**而一层处理状态的数量由设计模式决定，如对于一个数列，设计模式使得DFS的一层状态由当前元素的有无决定，那么就只有两个状态，而如果设计模式使得DFS中一层状态包含当前数列中所有可选元素，那么一层就有n个状态。

### 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]

]

**核心代码：**

def permute(self, nums: List[int]) -> List[List[int]]:

    n = len(nums)

    res = []

# 解法1 回溯（基于的直觉是通过调换， 使得每个位置在当前已出现的数字基础上，剩下每个数都会出现）

    def back(first):

        if first == len(nums):

            res.append(nums.copy())

        for i in range(first, len(nums)):

            nums[first], nums[i] = nums[i], nums[first]

            back(first+1)

            nums[first], nums[i] = nums[i], nums[first]

    back(0)

    return res

# 解法2 子变量子结果，由于是在原始序列上一个个依序深入，没有冗余搜索，不需要cache

    if n==0:

        return [[]]

    def df(idx):

        if idx == n-1:

            return [[nums[idx]]]

        else:

            res = []

            cur\_num = nums[idx]

            nexts = df(idx+1)

            for items in nexts:

                for i in range(len(items)+1):

                    res.append(items[:i]+[cur\_num]+items[i:])

            return res

return df(0)

### 面试题38 字符串的排列

输入一个字符串，打印出该字符串中字符的所有排列。

你可以以任意顺序返回这个字符串数组，但里面不能有重复元素。

示例:

输入：s = "abc"

输出：["abc","acb","bac","bca","cab","cba"]

**核心思想：**

本题用dfs，回溯和子变量的思想都可行，但是由于需要返回的多个结果，子变量会占用较大的内存，所以使用回溯。

回溯即在每一层都要处理当前层的所有可能，对于本问题，当前层即在当前位置，所有可能出现的字符。采用与与该位置所有字符调换的方法实现。

**当字符串有重复字符时，只需让当前位置不重复出现相同字符即可（对应上一题中绿色注释部分）。**

**核心代码：**

    def permutation(self, s: str) -> List[str]:

        res = list()

        def dfs(i, tmp):

            if i==len(s)-1:

                res.append("".join(tmp))

            else:

                firsts = set()

                for j in range(i, len(s)):

                    if tmp[j] in firsts:

                        continue

                    firsts.add(tmp[j])

                    tmp[i], tmp[j] = tmp[j], tmp[i]

                    dfs(i+1, tmp)

                    tmp[i], tmp[j] = tmp[j], tmp[i]

        dfs(0, list(s))

        return res

### [207 课程表](#_207_课程表)

**核心思想：**

DFS, 基本思想为对每个节点做dfs, 判断其是否会回到本节点，如果会，证明有环，否则无环，借助flags数组，简化判断。

**核心代码：**

        def dfs(i, adjacency, flags):

            if flags[i] == -1: return True

            if flags[i] == 1: return False

            flags[i] = 1

            for j in adjacency[i]:

                if not dfs(j, adjacency, flags): return False

            flags[i] = -1

            return True

        adjacency = [[] for \_ in range(numCourses)]

        flags = [0 for \_ in range(numCourses)]

        for cur, pre in prerequisites:

            adjacency[pre].append(cur)

        for i in range(numCourses):

            if not dfs(i, adjacency, flags): return False

        return True

### 399 除法求值（？并查集解法）

给出方程式 A / B = k, 其中 A 和 B 均为代表字符串的变量， k 是一个浮点型数字。根据已知方程式求解问题，并返回计算结果。如果结果不存在，则返回 -1.0。

示例 :

给定 a / b = 2.0, b / c = 3.0

问题: a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ?

返回 [6.0, 0.5, -1.0, 1.0, -1.0 ]

基于上述例子，输入如下：

equations(方程式) = [ ["a", "b"], ["b", "c"] ],

values(方程式结果) = [2.0, 3.0],

queries(问题方程式) = [ ["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"] ].

输入总是有效的。你可以假设除法运算中不会出现除数为0的情况，且不存在任何矛盾的结果。

**核心思想：**

1.先建图（表示每个节点所对应的子节点），再使用dfs或者bfs，其中python只建图可以直接使用嵌套字典。

### 1367 二叉树中的列表

给你一棵以 root 为根的二叉树和一个 head 为第一个节点的链表。

如果在二叉树中，存在一条一直向下的路径，且每个点的数值恰好一一对应以 head 为首的链表中每个节点的值，那么请你返回 True ，否则返回 False 。

一直向下的路径的意思是：从树中某个节点开始，一直连续向下的路径。

**核心思想：**

本题主要用来展示冗余的代码逻辑，在有严格层级关系（树）的结构中，DFS往往是可以严格深入而不用重复搜索的，除非没有理清代码逻辑。

本题相当于拿着整条链表从头开始与二叉树比较，如果不等就拿整条链表与二叉树的子节点比较

**展示代码：**

        def dfs(p, node):

            if p is None:

                return True

            if node is None:

                return False

            flag=False

            if p.val==node.val:

                flag = flag or dfs(p.next, node.left)

                flag = flag or dfs(p.next, node.right)

            if not flag and p==head:

                flag = flag or dfs(head, node.left)

                flag = flag or dfs(head, node.right)

            return flag

        return dfs(head, root)

## 1.4 滑动窗口

滑动窗口的算法思路：

1、我们在字符串 S 中使用双指针中的左右指针技巧，初始化 left=right=0，把索引闭区间 [left, right] 称为一个「窗口」。

2、我们先不断地增加 right 指针扩大窗口 [left, right]，直到窗口中的字符串符合要求（包含了 T 中的所有字符）。

3、此时，我们停止增加 right，转而不断增加 left 指针缩小窗口 [left, right]，直到窗口中的字符串不再符合要求（不包含 T 中的所有字符了）。同时，每次增加 left，我们都要更新一轮结果。

4、重复第 2 和第 3 步，直到 right 到达字符串 S 的尽头。

这个思路其实也不难，第 2 步相当于在寻找一个「可行解」，然后第 3 步在优化这个「可行解」，最终找到最优解。左右指针轮流前进，窗口大小增增减减，窗口不断向右滑动。

**其中滑动窗口内的判断往往可以用字典+标记变量实现。**

### 3 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:

输入: "abcabcbb"

输出: 3

解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

示例 2:

输入: "bbbbb"

输出: 1

解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。

**核心思想：**滑动窗口，用一前一后两个指针表示，前指针指向最前面的一个不重复字符。

**核心代码：**

**O(2n)+set:**

        unique = set()

        n = len(s)

        res, i, j = 0, 0, 0

        while j<n:

            if s[j] not in unique:

                unique.add(s[j])

                j=j+1

                res = max(j-i, res)

            else:

                unique.remove(s[i])

                i=i+1

        return res

**O(n)+dict:**

unique = dict()

        n = len(s)

        res, i, j = 0, -1, 0

        while j<n:

            if s[j] in unique:

                i = max(unique[s[j]], i)

            unique[s[j]] = j

            res = max(res, j-i)

            j+=1

return res

### 438 找出字符串中所有字母异位词

给定一个字符串 s 和一个非空字符串 p，找到 s 中所有是 p 的字母异位词的子串，返回这些子串的起始索引。

字符串只包含小写英文字母，并且字符串 s 和 p 的长度都不超过 20100。

说明：

字母异位词指字母相同，但排列不同的字符串。

不考虑答案输出的顺序。

示例 1:

输入:

s: "cbaebabacd" p: "abc"

输出:

[0, 6]

输入:

s: "abab" p: "ab"

输出:

[0, 1, 2]

**核心代码：**

        p\_dict = collections.Counter(p)

        window = {}

        left, right, match = 0, 0, 0

        res =[]

        while right<len(s):

            cr = s[right]

            if cr not in p\_dict:

                window = {}

                match = 0

                left = right+1

            else:

                window[cr] = window.get(cr, 0)+1

                if window[cr]==p\_dict[cr]:

                    match+=1

                while match==len(p\_dict):

                    if right-left+1 == len(p):

                        res.append(left)

                    cl = s[left]

                    if cl in p\_dict:

                        window[cl]-=1

                        if window[cl]<p\_dict[cl]:

                            match-=1

                    left+=1

            right+=1

        return res

## 1.5 字符串匹配

### KMP(实现Str()str())

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。

示例 1:

输入: haystack = "hello", needle = "ll"

输出: 2

示例 2:

输入: haystack = "aaaaa", needle = "bba"

输出: -1

**核心思想：  
KMP的核心在于fail数组，即匹配字符串的当前元素的下一个元素不匹配时，应该回溯到的位置，即在以当前元素为最后一个元素的子字符串中为后缀的最长前缀的下一个位置。**

**核心代码：**

    def strStr(self, haystack: str, needle: str) -> int:

        m, n = len(haystack), len(needle)

        if n==0:

            return 0

        fail = [0]\*n

        for i in range(1, n):

            j = fail[i-1]

            while j>0 and needle[j]!=needle[i]:

                j = fail[j-1]

            if needle[j]==needle[i]:

                j+=1

            fail[i] = j

        j = 0

        for i in range(m):

            while j>0 and needle[j]!=haystack[i]:

                j = fail[j-1]

            if needle[j]==haystack[i]:

                j+=1

            if j==n:

                return i-n+1

        return -1

# 2 Dynamic Programming

 动态规划解法是用递推模拟回归过程的方法，在缓存中逐渐推演，通过一步步的解决小问题来得到最终问题的解。

具有最优子结构性质和无后效性质的问题可以通过动态规划求解。

**最优子结构**

• 如果一个问题的最优解包含其子问题的最优解，我们就称此问题具有最优子结构

• 一个问题具有最优子结构，也可能使用递归，决定动态规划优势还有另一个条件：无后效性。

**无后效性：**

简单而言一句话，当前状态是唯一决定下一步优化的因素，与当前状态来源无关。

由此，无后效性使得动态规划的状态更新只基于上一步的状态，而不用关心上一步状态是怎么得来的，可以理解为只关注结果不关注过程。（是否有无后效性的对比见337打家劫舍的两种不同解法）

此外，要体现动态规划算法优势还需要问题优化过程有冗余，即重叠子问题，动态规划本质上是空间换时间，如果没有时间的冗余，也就没必要用动态规划。

**重叠子问题**

• 子问题空间必须重复，即在反复求解相同的子问题，而不是每次都产生新的子问题。

• 如果递归算法反复求解相同的子问题，我们就称最优化问题具有重叠子问题性质

**1** 动态规划有的依托数组实现，有的依托**指针**实现（但指针并不只用在动态规划问题中，此外，一般是动态规划中的**每种情况只出现一次**（不反复使用）且逐步更新时采用指针）。

**2** 动态规划一般都是自底向上（循环遍历思路），而递归是自顶向下（纯碎递归的复杂度往往是n的平方（因为要递归出所有的情况）**，递归+缓存并不是动态规划，将自顶向下的递归转换为自底向上的递推再加上缓存才是动态规划。**

3 动态规划的关键是1）找到状态转换公式2）构建缓存变量并初始化3）由初始情况递推。

4 动态规划的过程中有些中间状态（如从后往前的动态规划中正则匹配中的\*）是为了其他状态服务的，本身并没有意义，此时没必要重新设计代码逻辑判定，直接设定为否定状态即可。

## 2.1 Base

### 72 最小编辑距离

给定两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

插入一个字符

删除一个字符

替换一个字符

示例 1:

输入: word1 = "horse", word2 = "ros"

输出: 3

解释:

horse -> rorse (将 'h' 替换为 'r')

rorse -> rose (删除 'r')

rose -> ros (删除 'e')

**核心思想：**动态规划，字符串的动态规划实现一般为对子字符串逐渐的添加字符实现。此外，本题中不同操作的转换方程不同数组元素巧妙实现。

**核心代码：**

m, n = len(word1), len(word2)

        dp = [[0]\*(n+1) for \_ in range(m+1)]

        for i in range(m+1):

            dp[i][0] = i

        for i in range(n+1):

            dp[0][i] = i

        for i in range(1, m+1):

            for j in range(1, n+1):

                if word1[i-1] == word2[j-1]:

                    dp[i][j] = dp[i-1][j-1]

                else:

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

        return dp[m][n]

### 152. Maximum Product Subarray

Given an integer array nums, find the contiguous subarray within an array (containing at least one number) which has the largest product.

**Example 1:**

**Input:** [2,3,-2,4]

**Output:** 6

**Explanation:** [2,3] has the largest product 6.

**核心思想：**由于会出现负数，所以乘积可能为负，而负的乘积乘以负数会得到正的乘积，所以需要考虑最小的负的乘积，所以动态规划以最大的正乘积与最小的负乘积为依托随着时间进行。属于用变量设计动态规划。

**核心代码：**

def maxProduct(self, nums):

prev\_max, prev\_min = nums[0], nums[0]

res = nums[0]

for num in nums[1:]:

cur\_max = max(num, num\*prev\_max, num\*prev\_min)

cur\_min = min(num, num\*prev\_max, num\*prev\_min)

res = max(res, cur\_max)

prev\_max = cur\_max

prev\_min = cur\_min

return res

**错误的思想：**最初的设计中对于乘积的讨论过多使用if, else， 其实用max，min就能很好的概括，所以学会概括性思维。

### 337 \*打家劫舍III

在上次打劫完一条街道和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。 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。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

示例 1:

输入: [3,2,3,null,3,null,1]

3

/ \

2 3

\ \

3 1

输出: 7

解释: 小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = 7.

**核心思想1:**

递归+记忆。

**核心代码1:**    dp = {}

    def rob(self, root: TreeNode) -> int:

        if root is None:

            return 0

        if root not in self.dp:

            money = root.val

            if root.left:

                money += (self.rob(root.left.left)+self.rob(root.left.right))

            if root.right:

                money += (self.rob(root.right.left)+self.rob(root.right.right))

            self.dp[root] = max(money, self.rob(root.left)+self.rob(root.right))

        return self.dp[root]

**核心思想2:**

上述代码在计算当前节点时同时涉及到子节点与孙子节点，这样在计算子节点时，优惠重复计算孙子节点，有重复，即使有记忆，仍需要多余的遍历，所以关键是减少这种重复。

一层一层的计算， 而不用跨层。

下面的方法虽然没有单独的动态规划变量，但由于设计的严格层次，所以每一层的结果实际都是动态规划一层层传递的值。

**核心代码2:**

def get\_res(root):

            res = [0, 0]

            if not root:

                return res

            left\_res = get\_res(root.left)

            right\_res = get\_res(root.right)

            res[0] = max(left\_res)+max(right\_res)

            res[1] = root.val + left\_res[0] + right\_res[0]

            return res

        res = get\_res(root)

        return max(res)

**可取之处：关键在于如何将错乱的递归顺序通过**状态变量**转换为有序的层次结构**

注：「[740 删除与获得点数](https://leetcode-cn.com/problems/delete-and-earn/)」中数组的前后相邻元素不同时操作也类似于打家劫舍中对父节点、子节点的排除

### 1074 \*Number of Submatrices That Sum to Target

Given a matrix, and a target, return the number of non-empty submatrices that sum to target.

A submatrix x1, y1, x2, y2 is the set of all cells matrix[x][y]with x1 <= x <= x2 and y1 <= y <= y2.

Two submatrices (x1, y1, x2, y2) and (x1', y1', x2', y2')are different if they have some coordinate that is different: for example, if x1 != x1'.

**Example 1:**

**Input: matrix = [[0,1,0],[1,1,1],[0,1,0]], target = 0**

**Output: 4**

**Explanation: The** four 1x1 submatrices that only contain 0.

**核心思想：**肯定要使用动态规划，但是单纯的对（x1, x2, y1, y2）动态规划也有四层遍历，所以需要使用**累计存储结合前缀相减**的技巧。

**累计存储结合前缀相减**：求一个序列中满足和为目标值的连续子序列，可以对子序列的每个数累计相加，然后用后面的累计结果与之前累计结果的差是否等于目标值，判断其间的子序列和是否满足要求。为了存储之前的累计结果并方便索引，可以使用字典。

**核心代码：**

def num\_for\_one\_row(nums): #累加相减求子序列和

prev = {}

prev[0] = 1

cur\_sum = 0

ans = 0

for num in nums:

cur\_sum += num

if cur\_sum - target in prev:

ans += prev[cur\_sum - target]

if cur\_sum not in prev:

prev[cur\_sum] = 1

else:

prev[cur\_sum] += 1

return ans

res = 0

m, n = len(matrix), len(matrix[0])

for i in range(m):

nums = [0]\*n

for j in range(i,m):

for k in range(n):

nums[k]+=matrix[j][k]

res += num\_for\_one\_row(nums)

return res

## 2.2 状态表示

同机器学习一样，动态规划可分为两步：第一是表示，第二学习。

**表示**即是如何提取并表示一个状态。学习是如何找到状态之间的转换关系，完成状态的递推。

大部分简单的动态规划问题，一个状态只涉及一个元素，但是更复杂的会涉及多个元素之间的排列组合，即多个元素表示一个状态，那么不同状态之间的转换也是多个元素之间的转换。

那么如何有效表示多个元素组成的状态，举个简单例子，如果每个元素只有是、否两种情况，那么状态可由二进制数表示，几个元素即是几位二进制。

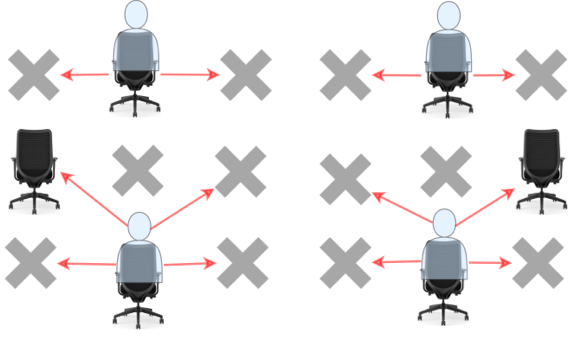
### \*1349 参加考试的最大学生数

给你一个m \* n的矩阵 seats 表示教室中的座位分布。如果座位是坏的（不可用），就用 '#' 表示；否则，用 '.' 表示。

学生可以看到左侧、右侧、左上、右上这四个方向上紧邻他的学生的答卷，但是看不到直接坐在他前面或者后面的学生的答卷。请你计算并返回该考场可以容纳的一起参加考试且无法作弊的最大学生人数。

学生必须坐在状况良好的座位上。

示例 1：



输入：seats = [["#",".","#","#",".","#"],

  [".","#","#","#","#","."],

  ["#",".","#","#",".","#"]]

输出：4

解释：教师可以让 4 个学生坐在可用的座位上，这样他们就无法在考试中作弊。

**核心思想：**

明显，本题不能按照输入的size构建状态数组，因为该情况下的每个状态不能代表一个合理的子问题，需要找到合适的能代表最优子问题的状态。

所以本题在动态规划类问题中的启示为：动态规划中的状态并不一定与输入数组的粒度对应的（简单而言一个元素对应一个状态），有可能**多个元素对应一个状态。**

本题即是多个元素对应一个状态，对于一排座位，任意一种对于学生的安排都是一个状态，当然由题意，这个安排需要满足左右不相邻，且在没坏的椅子上，并且上一排的左上、右上座位没有被安排。

可知，一排n个座位有2\*\*n种安排，即有2\*\*n个状态，有m层这样的状态。但是对每个状态，如何表示状态的有效性。

本题的第二个要点，即是位运算。

**核心代码：**

m, n = len(seats), len(seats[0]),

dp = [[0]\*(1 << n) for \_ in range(m+1)]  # 状态数组 dp

a = []

for i in range(m):

    s = "".join(seats[i])

    s = s.replace('.', '0')

    s = s.replace('#', '1')

    a.append(int(s, 2))

for row in range(1, m+1):

    for j in range(1 << n):

        if not j & j<<1 and not j&j>>1 and not j & a[row-1]: # j & a[row]代表该位置可以坐人，j & j<<1 and not j&j>>1 表示该位置左右没人可以坐的

            for k in range(1 << n):

                if not j&k<<1 and not j&k>>1: # j状态的左上和右上没有人

                    dp[row][j] = max(dp[row][j], dp[row-1][k] + bin(j).count('1'))

# print(dp)

return max(dp[m])

### 5337 每个元音包含偶数次的最长子串

给你一个字符串 s ，请你返回满足以下条件的最长子字符串的长度：每个元音字母，即 'a'，'e'，'i'，'o'，'u' ，在子字符串中都恰好出现了偶数次。

示例 1：

输入：s = "eleetminicoworoep"

输出：13

解释：最长子字符串是 "leetminicowor"，它包含 e，i，o 各2个，以及0个 a，u 。

**核心思想：**

本题第一反应是使用滑动窗口，但滑动窗口是单向的，而奇偶是反复的，如何对滑动窗口已经扫过的区域再处理？首先需要合理**表示已经扫过的状态。**

五个元音字母，每个字母出现奇数次或偶数次，总共有2\*\*5=32种状态，假设0表示偶，1表示奇，可知只有一段字符串的状态全为0时才符合要求。

在从左到右遍历时，**在已知现有字符串状态的情况下，如何判断从现有位置到之前某一位置之间的字符串是满足条件的呢**？如此便需要利用到**状态相减**。（之前碰到过的累加相减即是如此）。现有字符串i的状态减去之前某个位置的字符串j的状态得到的即是j~i之间的状态k，判断状态k是否满足要求，即可解决问题。

回到本题，状态k需要全0, 所以只要i和j的状态相同，如此可使用hash表存储已经遇到过的状态，键的形式为[0,0,0,0,0]，值为状态出现的最早下标。

进一步思考，因为每个状态只有0，1两种，所以可以用二进制表示，如此键的表示为[0, 31]之间的一个数，因为数值范围小，易于索引，所以可以用数组替代hash表，进一步简化问题。

最后一个子问题，如何更新状态，很简单，在0，1之间转换，使用抑或操作（**抑或操作在涉及到两种状态转换时很好用）。**

**核心代码：**

        records = [-1 for \_ in range(32)]

        chars\_dict = {'a':1, 'e':2, 'i':4, 'o':8, 'u':16}

        cur = 0

        ans = 0

        for i,c in enumerate(s):

            if c in chars\_dict:

                cur ^= chars\_dict[c]

            if cur!=0 and records[cur]==-1:

                records[cur]=i

            else:

                ans = max(ans, i-records[cur])

        return ans

## 2.3 股票问题全解

这里涉及动态规划，扩展讲一些。

动态规划的本质是穷举，进一步的如何**有效的穷举**。

**1）股票问题**需要穷举的「状态」有三个，第一个是天数，第二个是允许交易的最大次数，第三个是当前的持有状态（即之前说的 rest 的状态，我们不妨用 1 表示持有，0 表示没有持有）。然后我们用一个三维数组就可以装下这几种状态的全部组合：

dp[i][k][0 or 1] ，0 <= i <= n-1, 1 <= k <= K

n 为天数，大 K 为最多交易数

此问题共 n × K × 2 种状态，全部穷举就能搞定。

for 0 <= i < n:

for 1 <= k <= K:

for s in {0, 1}:

dp[i][k][s] = max(buy, sell, rest)

但是在上述的三层循环中存在冗余计算，不是每个i, k, [0,1]都是有效的，且三维数组过于浪费空间（实际上由于在天数上严格的上层依赖，需要的数组只是二维，不需要数组在天数维度展开）。

**2）那么，如何对每个k, [0, 1]更新呢**：

dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])

max(选择 rest ,选择 sell )

解释：今天我没有持有股票，有两种可能：

要么是我昨天就没有持有，然后今天选择 rest，所以我今天还是没有持有；

要么是我昨天持有股票，但是今天我 sell 了，所以我今天没有持有股票了。

dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])

max( 选择 rest , 选择 buy )

解释：今天我持有着股票，有两种可能：

要么我昨天就持有着股票，然后今天选择 rest，所以我今天还持有着股票；

要么我昨天本没有持有，但今天我选择 buy，所以今天我就持有股票了。

于是加上i=1时的基本情况，即可完成。

此外，在k为1和无穷时，可以省略k层的循环，在1时只有一层，而在无穷时k,k-1对上层依赖、对下层影响的状态是一样的（因为没有买入、卖出的限制，只需要去最优化），而k为其他整数则不同，只是有限的最优化。

3）对于每次 sell 之后要等一天才能继续交易。只要把这个特点融入上一题的状态转移方程即可：

dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])

dp[i][1] = max(dp[i-1][1], dp[i-2][0] - prices[i])

4）每次交易要支付手续费，只要把手续费从利润中减去即可。改写方程：

dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])

dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i] - fee)

**5）相关题**

123 Best Time to Buy and Sell Stock III

309 最佳买卖股票时期含冷冻期

### 123 Best Time to Buy and Sell Stock III

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most two transactions.

**Note:**You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

**Example 1:**

**Input:** [3,3,5,0,0,3,1,4]

**Output:** 6

**Explanation:** Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3. Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.

**核心思想：**1 注意购买股票的多种状态，第一次购入，售出，第二次购入，售出

2 双重动态规划，一重是对股票之后的操作与之前的操作的动态规划， 一重是随着时间的动态规划。注意如果只使用一重动态规划会造成操作的冗余，消耗时间。

**核心代码：**

res, max1, max1\_hold, min\_price = 0, 0, -prices[0], prices[0]

for price in prices:

res = max(res, max1\_hold+price)

max1\_hold = max(max1\_hold, max1-price)

max1 = max(max1, price-min\_price)

min\_price = min(min\_price, price)

res = max(res, max1)

## 2.4 背包问题

个人对背包问题的理解为给定一个数列和一个目标值，求1）能否用数列中的数相加得到目标值或者2）最少需要数列中的多少个数才能达到目标值。

第一个问题：动态规划，数列长度x目标值大小 的**状态**（bool）数组，但往往状态数组可以使用大小为目标值的一维数组，从而节约内存。

第二个问题：可以使用动态规划，但也可以使用限定范围的BFS。

### 416 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。

注意:

每个数组中的元素不会超过 100

数组的大小不会超过 200

示例 1:

输入: [1, 5, 11, 5]

输出: true

解释: 数组可以分割成 [1, 5, 5] 和 [11].

**核心思想：**

背包问题，建立len x target的动态规划，每个状态在前一状态的基础上分为用或者不用当前节点。

为了节约内存，可以只用一维状态数组，但由于dp[i][j]会依赖于之前的状态dp[i]j-k], 如果顺序遍历，会使状态在当前层已更新的基础上再次更新，为此可以使用**反序遍历**。

此外，还有剪枝，即提前判断是否达到目标。

**核心代码：**

dp = [False]\*(half+1)

        dp[nums[0]] = True

        for i in range(1, n):

            for j in range(half, -1, -1):

                if nums[i] == j:

                    dp[j] = True

                elif nums[i]<j:

                    dp[j] = dp[j] or dp[j-nums[i]]

                else:

                    break

            if dp[half]:

                return True

        return dp[half]

**注意：由于target范围一般较大，j有很多无效遍历，会使时间复杂度较高（往往大于bfs）**

## 2.5 数位DP

数位DP应用问题：寻找小于某个数B的满足一定条件的数值。

基本思路：

依次遍历每一位：

1. 按位构基于地位建dp缓存
2. 再按dp缓存计算当前位满足条件的数量

第2步展开：

对每一位的数从0（或1）开始遍历至B对应位的数减1：

不受限的寻找满足条件的数。（不同位之间满足条件的数可以用dp求得，一般从小到大实现dp）

对B对应位的数，等价于更低一位满足条件的数

数位dp的关键点在于处理每一数位的边界情况，如球小于4320的数所包含的1的个数，当第一位等于0，1，2，3时都很好处理，但当第一位等于4时，此时满足条件的数量其实决定于320，所以边界情况取决于更低位，这也是真正需要用dp的地方。

### 233 数字1的个数

给定一个整数 n，计算所有小于等于 n 的非负整数中数字 1 出现的个数。

示例:

输入: 13

输出: 6

解释: 数字 1 出现在以下数字中: 1, 10, 11, 12, 13 。

**核心代码：**

        digits, nums\_1 = list(map(int, str(n))), [0]

        l = len(digits)

        for i in range(l-1):

            num\_1 = 10\*nums\_1[i]+10\*\*i

            nums\_1.append(num\_1)

        res = 0

        for i,num in enumerate(digits):

            for j in range(0, num):

                res+=nums\_1[-i-1]

                if j==1:

                    res+=10\*\*(l-i-1)

            if num==1:

                res += n%(10\*\*(l-i-1))+1

        return res

902 最大为N的数字组合

我们有一组排序的数字 D，它是  {'1','2','3','4','5','6','7','8','9'} 的非空子集。（请注意，'0' 不包括在内。）

现在，我们用这些数字进行组合写数字，想用多少次就用多少次。例如 D = {'1','3','5'}，我们可以写出像 '13', '551', '1351315' 这样的数字。

返回可以用 D 中的数字写出的小于或等于 N 的正整数的数目。

示例 1：

输入：D = ["1","3","5","7"], N = 100

输出：20

解释：

可写出的 20 个数字是：

1, 3, 5, 7, 11, 13, 15, 17, 31, 33, 35, 37, 51, 53, 55, 57, 71, 73, 75, 77.

**核心代码：**

    def atMostNGivenDigitSet(self, D, N):

        S = str(N)

        K = len(S)

        dp = [0] \* K + [1]

        for i in range(K-1, -1, -1):

            for d in D:

                if d < S[i]:

                    dp[i] += len(D) \*\* (K-i-1)

                elif d == S[i]:

                    dp[i] += dp[i+1]

        return dp[0] + sum(len(D) \*\* i for i in range(1, K))

# 3 分治法(Divide and Conquer)

分治算法往往将n2或者n!的问题分解为(n/2)2或者(n/2)!的子问题，但不是所有问题都能用分治算法，分治算法要求子问题**不相互关联**。

### 932 \*Beautiful Array

For some fixed N, an array A is beautiful if it is a permutation of the integers 1, 2, ..., N, such that:

For every i < j, there is **no** k with i < k < j such that A[k] \* 2 = A[i] + A[j].

Given N, return **any** beautiful array A.  (It is guaranteed that one exists.)

**Example 1:**

**Input:** 4

**Output:** [2,1,4,3]

**核心思想：**

1 为了不满足A[k]\*2=A[i]+A[j]， 让右边的A[i]+A[j]不为偶数即可，为此需要一个为奇数，一个为偶数，放在数列中，一半为奇数，一半为偶数，如此divide and conquer递归处理即可。

2 难点在于全为偶数或奇数的半个数列如何再划分奇偶？引入另一个性质：**若x1,x2,..,xn满足条件，则a\*x1+b, a\*x2+b,…, a\*xn+b也满足**，如此可将划分为奇偶的数列先变为连续的数列再处理。最后再用a，b的线性变换恢复即可。

**核心代码：**

memo = {1: [1]}

def f(N):

if N not in memo:

odds = f(int((N+1)/2))

evens = f(int(N/2))

memo[N] = [2\*x-1 for x in odds] + [2\*x for x in evens]

return memo[N]

return f(N)

**小技巧：**递归时为了节约内存和时间，可以使用缓存结果（memo变量）

### 312 戳气球

有 n 个气球，编号为0 到 n-1，每个气球上都标有一个数字，这些数字存在数组 nums 中。

现在要求你戳破所有的气球。每当你戳破一个气球 i 时，你可以获得 nums[left] \* nums[i] \* nums[right] 个硬币。 这里的 left 和 right 代表和 i 相邻的两个气球的序号。注意当你戳破了气球 i 后，气球 left 和气球 right 就变成了相邻的气球。

求所能获得硬币的最大数量。

说明:

你可以假设 nums[-1] = nums[n] = 1，但注意它们不是真实存在的所以并不能被戳破。

0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100

示例:

输入: [3,1,5,8]

输出: 167

解释: nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []

coins = 3\*1\*5 + 3\*5\*8 + 1\*3\*8 + 1\*8\*1 = 167

**核心思想：**

分治，问题可分解为子问题，如抽取(i，j)中间的k， 然后处理(i,k)+(k,j)+nums[k-1]\*nums[k]\*nums[k+1], 但是顺序处理的过程中，(i，k)与（k,j）互相依赖，不方便处理，可以反过来处理，假设最后一个取的是k。然后处理(i, k) + (k, j) + nums[i]\*nums[k]\*nums[j]。

分治的基本套路是递归，但为了节约内存，可以使用动态规划，动态规划设计也是从后往前。

**核心代码：**

        nums = [1, \*nums, 1]

        n = len(nums)

        dp = [[0]\*n for i in range(n)]

#开 始 dp：i为 begin，j为 end，k为 在 i、j区 间 划 分 子 问 题 时 的 边 界

        for i in range(n-3, -1, -1):

            for j in range(i+2, n):

                cur\_max = 0

                for k in range(i+1, j):

                    tmp = dp[i][k]+dp[k][j]+nums[i]\*nums[j]\*nums[k]

                    cur\_max = max(tmp, cur\_max)

                dp[i][j] = cur\_max

        return dp[0][n-1]

# 4 排序

## 4.1 快排

def partition(a, l, r):

v =a[r]

idx = l

i = l

while i<r:

if a[i]<v:

a[i], a[idx] = a[idx], a[i]

idx += 1

i += 1

a[r], a[idx] = a[idx], a[r]

return idx

def quick\_sort(a, l, r):

if l>=r:

return

q = partition(a, l, r)

quick\_sort(a, l, max(l,q-1))

quick\_sort(a, q+1, r)

return

此外，用partition函数去逼近能在线性时间内找到一个数列中第k大的值。

## 4.2 堆（优先队列）

堆的特点是只关注最大（小）的，进一步的堆中的有序性只体现在父节点与子节点之间，不同的父子节点之间并不保证有序性，这种“偷懒“在全部排序时不能体现优势，但相对的，在只需要”**局部有序**“时，优势明显。

如提取一个数列中前k大的数，使用普通的排序时间复杂度为O(NlogN)，最差为O(N\*\*2)，但是通过维持一个大小为k的最小堆，时间复杂度为O(Nlogk)，原因在于堆只保证前k个元素的最小值大于不在k中的所有元素，而不管堆中元素的全部顺序和其他元素的顺序。

此外，堆能在动态的添加元素时，通过较小的时间复杂度保证堆的性质，如求在动态添加元素的数列中第k大的元素时，普通的时间复杂度最少为O(N),

但是通过大、小堆（即一个前k大的最小堆和一个后N-k大的最小堆）共同维护，时间复杂度为O(logN)。

对于会删除元素的动态数列，堆虽然不便于删除元素，但只要能保证大、小堆的性质（相互之间的有序性，堆的大小），也能通过“延迟删除”解决删除的问题。

#关于堆：

堆是完全树，基于数组实现，数组下标由1开始，子节点的下标除以2取整为父节点的下标，由此实现子节点与父节点的寻找。

堆有两个关键操作，上浮，下沉。其他一切都基于这两个操作。如

插入操作：将元素放在数组尾部，然后对该元素使用上浮。

弹出操作：取数组下标为1的元素，然后将数组尾部的元素放在数组下标为1的位置，删除数组尾部元素，然后对数组下标1使用下沉操作。

排序：对给定数组的前n//2个元素(即所有父节点)逆序使用下沉操作，然后执行n-1次弹出操作即可。

    def insert(self, x):

        self.nums.append(x)

        self.shift\_up(len(self.nums)-1)

    def pop(self):

        min\_num = self.top()

        self.nums[1] = self.nums[-1]

        self.nums.pop()

        self.shift\_down(1)

        return min\_num

    def shift\_up(self, idx):

        while idx>1 and self.nums[idx]<self.nums[idx//2]:

            self.nums[idx], self.nums[idx//2] = self.nums[idx//2], self.nums[idx]

            idx = idx//2

    def shift\_down(self, idx):

        while 2\*idx<len(self.nums):

            j = 2\*idx

            if j+1<len(self.nums) and self.nums[j+1]<self.nums[j]:

                j = j+1

            if self.nums[j]<self.nums[idx]:

                self.nums[idx], self.nums[j] = self.nums[j], self.nums[idx]

                idx = j

            else:

                break

    def sort(self, data):

        l = len(data)

        self.nums = [-1]+data

        if l<2:

            return self.nums[1:]

        for i in range(l//2, 0, -1):

            self.shift\_down(i)

        res = []

        for i in range(l):

            res.append(self.pop())

        return res

### 295 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：

addNum(1)

addNum(2)

findMedian() -> 1.5

addNum(3)

findMedian() -> 2

**核心思想：**

大小堆，即最大堆和最小堆，最小堆用来维护大的一半数，最大堆用来维护小的一半数。并且保证两个堆的大小相等（奇数时最小堆多一个）。

为了保证最小堆中元素为数列中最大的一半，即最小堆的的最小值>=最大堆中的最大值，需要将所有的元素都放进最小堆中，然后最小堆再将最小的元素弹出，送到最大堆。最后还需要判断两个堆的元素数量是否相等，并调整。

**核心代码：**

class MedianFinder:

    def \_\_init\_\_(self):

        self.max\_h = []

        self.min\_h = []

    def addNum(self, num: int) -> None:

        heapq.heappush(self.min\_h, num)

        heapq.heappush(self.max\_h, -heapq.heappop(self.min\_h))

        if len(self.max\_h)>len(self.min\_h):

            heapq.heappush(self.min\_h, -heapq.heappop(self.max\_h))

    def findMedian(self) -> float:

        if len(self.min\_h)>len(self.max\_h):

            return self.min\_h[0]

        else:

            return (self.min\_h[0]-self.max\_h[0])/2

### 347 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:

输入: nums = [1,1,1,2,2,3], k = 2

输出: [1,2]

示例 2:

输入: nums = [1], k = 1

输出: [1]

说明：

你可以假设给定的 k 总是合理的，且 1 ≤ k ≤ 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。

**核心思想：**

先遍历，统计每个数字出现的次数， 然后

**1** 维护大小为k的最小堆， 堆的比较依据出现的次数

**2** 桶排序，因为次数有限，可以设计长为最大出现次数的数组，将对应的元素依次放入，然后从后往前选取k个元素即可。

**核心代码：**

**#** 基本处理，计数统计

cache = dict()

        max\_count = 0

        for num in nums:

            cache[num] = cache.get(num, 0)+1

            max\_count = max(max\_count, cache[num])

# 解法1，维护最小堆

        k\_top = []

        for key, val in cache.items():

            if len(k\_top)<k:

                heapq.heappush(k\_top, (cache[key], key))

            else:

                if cache[key]>k\_top[0][0]:

                    heapq.heappop(k\_top)

                    heapq.heappush(k\_top, (cache[key], key))

        res = []

        while k\_top:

            res.append(k\_top.pop()[1])

**#** 解法2，桶排序

#见桶排序部分

### 480 滑动窗口中位数

中位数是有序序列最中间的那个数。如果序列的大小是偶数，则没有最中间的数；此时中位数是最中间的两个数的平均数。

给出一个数组 nums，有一个大小为 k 的窗口从最左端滑动到最右端。窗口中有 k 个数，每次窗口向右移动 1 位。你的任务是找出每次窗口移动后得到的新窗口中元素的中位数，并输出由它们组成的数组。

提示：

你可以假设 k 始终有效，即：k 始终小于输入的非空数组的元素个数。

与真实值误差在 10 ^ -5 以内的答案将被视作正确答案。

**核心思想：**

本质与285一致，使用大小堆来维护，但是滑动窗口会动态的移除和添加元素，所以堆也需要移除元素和添加元素，但在经典堆中删除一个元素却非常复杂，为此可以使用“延迟删除”

所谓“延迟删除”即当元素在堆中不删除，直到出现在堆顶中再删除，那元素在堆中不会有影响吗？

重新思考最大堆、最小堆维护中位数的本质：

1 保证最大堆中的元素是最小的一半，最小堆的元素是最大的一半

2 最大堆的元素数量和最小堆的元素数量相等

延迟删除不影响第一个性质，但会影响第二个性质，如何保证第二个性质？

进一步想，如果滑动窗口移除的元素没有在相应的堆中删除，然后新包含的元素正好添加在相同的堆里，那么第二性质依然保证。那么如果添加在不同的堆里，只要再在堆之间转移相应元素即可, 转移后使用“延迟删除”删除堆顶被标记为删除的元素，然后求两堆的中位数。

**核心技巧：**

     hi, lo = [], []

        remove = collections.defaultdict(int) #存的是index

        odd = k&1

        for i in range(len(nums)):

            if i<k:

                heapq.heappush(hi, nums[i])

                heapq.heappush(lo, -heapq.heappop(hi))

                if len(hi)<len(lo):

                    heapq.heappush(hi, -heapq.heappop(lo))

            else:

                out\_num = nums[i-k]

                in\_num = nums[i]

                flag = -1 if out\_num<=-lo[0] else 1

                remove[out\_num] += 1

                if in\_num<=-lo[0]:

                    heapq.heappush(lo, -in\_num)

                    flag+=1

                else:

                    heapq.heappush(hi, in\_num)

                    flag-=1

                if flag>0:

                    heapq.heappush(hi, -heapq.heappop(lo))

                if flag<0:

                    heapq.heappush(lo, -heapq.heappop(hi))

                while remove[-lo[0]]>0:

                    remove[-lo[0]]-=1

                    heapq.heappop(lo)

                while remove[hi[0]]>0:

                    remove[hi[0]]-=1

                    heapq.heappop(hi)

            if i>=k-1:

                res.append(hi[0] if odd else (hi[0]-lo[0])/2)

        return res

### 1353 最多可以参加会议的数目

给你一个数组 events，其中 events[i] = [startDayi, endDayi] ，表示会议 i 开始于 startDayi ，结束于 endDayi 。

你可以在满足 startDayi <= d <= endDayi 中的任意一天 d 参加会议 i 。注意，一天只能参加一个会议。

请你返回你可以参加的 最大 会议数目。

示例 1：

输入：events = [[1,2],[2,3],[3,4]]

输出：3

解释：你可以参加所有的三个会议。

安排会议的一种方案如上图。

第 1 天参加第一个会议。

第 2 天参加第二个会议。

第 3 天参加第三个会议。

**核心思想：**

本质上是贪心问题，贪心的原则是参加最早结束的会议，基于此，有两种实现方法。

1 遍历所有可能的参加会议时间并使用最小堆对会议进行管理。

2 按照贪心原则（结束的时间）排序后依次遍历会议并尽可能早地安排，使用集合存储安排信息。

**核心代码:**

# 解法1

def maxEvents(self, events: List[List[int]]) -> int:

res = 0

ends = []

events\_dict = collections.defaultdict(list)

for event in events:

events\_dict[event[0]].append(event[1])

for i in range(10\*\*5+1):

if i in events\_dict:

for e in events\_dict[i]:

heapq.heappush(ends, e)

while ends:

cur = heapq.heappop(ends)

if cur>=i:

res+=1

break

return res

#解法2：

def maxEvents(self, events: List[List[int]]) -> int:

events = sorted(events, key = lambda x: x[1])

arrange = set()

for event in events:

for i in range(event[0], event[1]+1):

if i not in arrange:

arrange.add(i)

break

return len(arrange)

## 4.3 桶排序（计数排序）

### [347 前k个高频元素](#_347_前k个高频元素)

**核心代码：**

counts\_array = [[] for \_ in range(max\_count)]

        for key, val in cache.items():

            counts\_array[val-1].append(key)

        res = []

        while len(res)<k:

            res.extend(counts\_array.pop())

## 4.4 循环排序

基本思想是下面一句话的迭代：

把当前位置的元素安排到对应的位置，把占据对应位置的元素拉回来。

### 41 first missing positive

Given an unsorted integer array, find the smallest missing positive integer.

**Example 1:**

Input: [1,2,0]

Output: 3

**Example 2:**

Input: [3,4,-1,1]

Output: 2

Your algorithm should run in *O*(*n*) time and uses constant extra space.

**思路的重点：在nums对应的下标存数值等于下标+1的值，但注意不要覆盖原来的数（可以一直交换直到不满足交换条件，见标粗代码）。**

**核心代码：**

for(i=0;i<nums.size();i++)

while(nums[i]>0&&nums[i]<=nums.size()&&**nums[i]!=nums[nums[i]-1]**)

swap(nums[i],nums[nums[i]-1]);

for(i=0;i<nums.size();i++)

if(nums[i]!=i+1)return i+1;

# 5 Tree

## 5.1 二叉树

### 线索二叉树（莫里斯遍历）

**核心思想：**

由二叉树转换而来，可用于二叉树的中序遍历，流程如下：

若current没有左子节点

a. 将current添加到输出

b. 进入右子树，亦即, current = current.right

否则

a. 在current的左子树中，令current成为最右侧节点的右子节点

b. 进入左子树，亦即，current = current.left

**核心代码：**

TreeNode curr = root;

TreeNode pre;

while (curr != null) {

if (curr.left == null) {

res.add(curr.val);

curr = curr.right; // move to next right node

} else { // has a left subtree

pre = curr.left;

while (pre.right != null) { // find rightmost

pre = pre.right;

}

pre.right = curr; // put cur after the pre node

TreeNode temp = curr; // store cur node

curr = curr.left; // move cur to the top of the new tree

temp.left = null; // original cur left be null, avoid infinite loops

}

}

不改变原有树结构的实现：

res = []

  cur = root

  while cur is not None:

    if not cur.left:

      res.append(cur.val)

      cur = cur.right

    else:

      pre = cur.left

      while pre.right and pre.right!=cur:

        pre = pre.right

# 第一次先建立一边线索树

        if not pre.right:

          pre.right = cur

          cur = cur.left

# 第二次回到该点时，由于已经建立后回溯的right节点，所以最终会重复到自身。

        else:

          res.append(cur.val)

          pre.right = None

          cur = cur.right

### 236 二叉树的最近公公祖先

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。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

例如，给定如下二叉树:  root = [3,5,1,6,2,0,8,null,null,7,4]



示例 1:

输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

输出: 3

解释: 节点 5 和节点 1 的最近公共祖先是节点 3。

示例 2:

输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

输出: 5

**核心思想1：**如果能往上回溯，解决该类父节点的问题就非常简单，可是原始树的设计就没有父指针，但是可以通过遍历得到父指针（用辅助的字典实现）

**核心代码1:**

        stack = [root]

        parent = {root: None}

        while p not in parent or q not in parent:

            node = stack.pop()

            if node.left:

                parent[node.left] = node

                stack.append(node.left)

            if node.right:

                parent[node.right] = node

                stack.append(node.right)

        ancestors = set()

        while p:

            ancestors.add(p)

            p = parent[p]

        while q not in ancestors:

            q = parent[q]

        return q

**核心思想2：**如果不使用回溯，那么就要对每个当前节点往下判断，判断其是否是最深父节点，往下深入的过程可以用递归实现。但最终判断还是要回溯。

**核心代码2:**

        flag = True #标记是否找到公有父节点，由于向下递归，第一个找到肯定最深

        parent\_node = None

        def bi(node):

            nonlocal flag, parent\_node

            if not flag or not node:

                return 0

            count = 0

            if node.val==p.val or node.val==q.val:

                count+=1

            count += bi(node.left)

            count += bi(node.right)

            if flag and count==2:

                parent\_node = node

                flag = False

            return count

        bi(root)

        return parent\_node

### 297 二叉树的序列化与反序列化

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。

**核心思想：**

序列化肯定需要先遍历，提供两种遍历方式。

1 先序遍历。

2层次遍历（使用队列）。

无论哪种遍历方式，先遍历树得到序列化字符串，然后在同样的方式遍历构建树。

**核心代码：**

1先序遍历

def serialize(self, root):

"""

Encodes a tree to a single string.

:type root: TreeNode

:rtype: str

"""

res = ""

if not root:

return ""

def dfs(node, res):

if not node:

res+="# "

else:

res+=str(node.val)+" "

res = dfs(node.left, res)

res = dfs(node.right, res)

return res

return dfs(root, res)

def deserialize(self, data):

"""

Decodes your encoded data to tree.

:type data: str

:rtype: TreeNode

"""

if data == "":

return None

data = data.strip().split(" ")

data = data[::-1]

def build\_tree(data):

num1 = data.pop()

if num1 == '#':

return None, data

node = TreeNode(int(num1))

node.left, data = build\_tree(data)

node.right, data = build\_tree(data)

return node, data

return build\_tree(data)[0]

1. 层次遍历

def serialize(self, root):

res = ""

if not root:

return ""

q = collections.deque([root])

res += str(root.val)+" "

while q:

node = q.pop()

if node.left:

res += str(node.left.val)+' '

q.appendleft(node.left)

else:

res += "# "

if node.right:

res += str(node.right.val)+' '

q.appendleft(node.right)

else:

res += "# "

return res

def deserialize(self, data):

if data == "":

return None

data = data.strip().split(" ")

data = data[::-1]

root = TreeNode(data.pop())

q = collections.deque([root])

while q:

node = q.pop()

num1 = data.pop()

if num1 != '#':

node.left = TreeNode(int(num1))

q.appendleft(node.left)

num2 = data.pop()

if num2 != '#':

node.right = TreeNode(int(num2))

q.appendleft(node.right)

return root

### 5346 二叉树中的列表

给你一棵以 root 为根的二叉树和一个 head 为第一个节点的链表。

如果在二叉树中，存在一条一直向下的路径，且每个点的数值恰好一一对应以 head 为首的链表中每个节点的值，那么请你返回 True ，否则返回 False 。

一直向下的路径的意思是：从树中某个节点开始，一直连续向下的路径。

**核心思想：**

除去使用DFS，本题还可以先存取所有的树节点，将每个树节点与头指针的值比较，如果相等，存取其子节点至下一次要比较的树节点列表中。把所有当前树节点比较完后，将头指针下移，比较更新后的树节点列表。

**本题重点的思想是将树节点提取出来进行遍历，而不用递归。**

**核心代码：**

tree = [root]

x = 0

# 取出所有树节点

while x < len(tree):

    if tree[x].left != None:

        tree.append(tree[x].left)

    if tree[x].right != None:

        tree.append(tree[x].right)

    x += 1

# 等价于从所有树节点开始匹配，每次失配的丢掉，匹配的留下，如果没有可匹配的，则返回false，否则循环结束返回true

while head != None:

    new\_tree = []

    for v in tree:

        if v != None and v.val == head.val:

            new\_tree.append(v.left)

            new\_tree.append(v.right)

    if len(new\_tree) == 0:

        return False

    tree = new\_tree

    head = head.next

return True

## 5.2 并查集

### 200 岛屿数量(并查集)

给定一个由 '1'（陆地）和 '0'（水）组成的的二维网格，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:

输入:

11110

11010

11000

00000

输出: 1

**核心思想：**

该类问题属于flood fill问题，使用并查集（用来查询给定的森林中独立的树的数量），即需要实现一个UnionFind类，该类初始每个节点的parent为自身，rank为1，然后依次对每个节点搜索，判断其他元素与其是否有相同的parent，并更新parent与rank（本问题中的其他元素主要指其右边与下边的元素）。最后返回并查集中独立的树的数量。

此外，本题还可以对每个节点依次使用dfs和bfs实现。

**核心代码：**

class UnionFind:

    def \_\_init\_\_(self, n):

        self.parent = list(range(n))

        self.rank = [1]\*n

        self.count = n

    def get\_count(self):

        return self.count

    def find(self, p):

        while p != self.parent[p]:

            self.parent[p] = self.parent[self.parent[p]]

            p = self.parent[p]

        return p

    def union(self, p, q):

        root1 = self.find(p)

        root2 = self.find(q)

        if root1 == root2:

            return

        if self.rank[root1] > self.rank[root2]:

            self.parent[root2] = root1

            self.rank[root1]+=1

        elif self.rank[root1] < self.rank[root2]:

            self.parent[root1] = root2

            self.rank[root2]+=1

        else:

            self.parent[root2] = root1

            self.rank[root1] += 1

        self.count -= 1

row = len(grid)

if row == 0:

    return 0

col = len(grid[0])

def get\_index(x, y):

    return x \* col + y

directions = [(1, 0), (0, 1)]

# 多开一个空间，把水域 "0" 都归到这个虚拟的老大上

dummy\_node = row \* col

uf = UnionFind(dummy\_node + 1)

for i in range(row):

    for j in range(col):

        if grid[i][j] == '0':

            uf.union(get\_index(i, j), dummy\_node)

        if grid[i][j] == '1':

            for direction in directions:

                new\_x = i + direction[0]

                new\_y = j + direction[1]

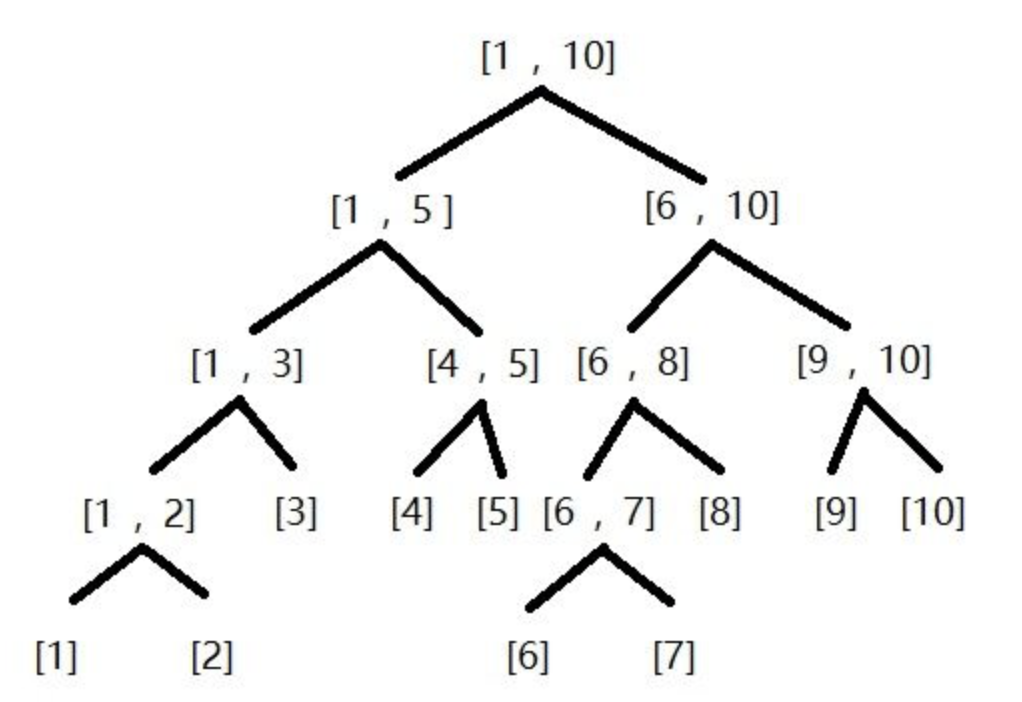
                if new\_x < row and new\_y < col and grid[new\_x][new\_y] == '1':

                    uf.union(get\_index(i, j), get\_index(new\_x, new\_y))

# 不要忘记把那个虚拟结点减掉

return uf.get\_count() – 1

## 5.3 线段树



### 315 [计算右侧小于当前元素的个数](#_315_计算右侧小于当前元素的个数)

## 5.4 trie树

### 208 Trie树

实现一个 Trie (前缀树)，包含 insert, search, 和 startsWith 这三个操作。

**核心思想：**trie树每个节点有一个索引表，索引表中每个索引对应下一个节点。此外，每个节点有一个标识，标识到这个节点为止是否存在单词。

**核心代码：**

class Trie:

  def \_\_init\_\_(self):

        self.children = {}

        self.isword = False

  def insert(self, word: str) -> None:

        cur = self

        for c in word:

            if c not in cur.children:

                cur.children[c] = Trie()

            cur = cur.children[c]

        cur.isword = True

  def search(self, word: str) -> bool:

        cur = self

        for c in word:

            if c not in cur.children:

                return False

            cur = cur.children[c]

        return cur.isword

 def startsWith(self, prefix: str) -> bool:

        cur = self

        for c in prefix:

            if c not in cur.children:

                return False

            cur = cur.children[c]

        return True

# 6 Stack

**Stack处理，每一次入栈、出栈的对象要表示一个完整的操作。**

### 42 Trapping Rain Water

Given *n* non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.



**核心思想：**

**往栈中放wall,如果新的wall比栈顶wall小，不能判断每个位置能装的水，只有在新的wall比栈顶wall大，才能判断。**

经典的stack使用方法，即遍历，将每一格墙的高度存入stack，遇到比stack最顶层的墙高的墙时，stack弹出最顶层的墙，并计算两墙之间的容量，直到stack为空或者顶层的墙比当前墙高，然后继续遍历。

注意：

在遇到比stack顶层墙高的墙时的迭代需要有一个基准变量ground，防止容量重复累积

核心代码：

stack<int> walls;

int ground = 0, sum=0;

for(int i=0;i<height.size();i++){

while(walls.size()!=0 && height[walls.top()]<=height[i]){

sum+=(i-walls.top()-1)\*(height[walls.top()]-ground);

ground = height[walls.top()];

walls.pop();

}

# 最后删除的一个比当前点矮的列上的雨水还没有计算，所以需# # 要加如下判断

if(walls.size()!=0)sum+=(i-walls.top()-1)\*(height[i]-ground);

walls.push(i);

ground = 0;

}

### 394 字符串解码

给定一个经过编码的字符串，返回它解码后的字符串。

编码规则为: k[encoded\_string]，表示其中方括号内部的 encoded\_string 正好重复 k 次。注意 k 保证为正整数。

你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。

此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 k ，例如不会出现像 3a 或 2[4] 的输入。

示例:

s = "3[a]2[bc]", 返回 "aaabcbc".

s = "3[a2[c]]", 返回 "accaccacc".

s = "2[abc]3[cd]ef", 返回 "abcabccdcdcdef".

**核心思想：**辅助栈的使用，字符串基本的结构是一个数字负责一个括号内的字符串，所以要将数字与括号内的字符串对应起来。于是，最好的是将数字与字符串同时做栈的操作。

**核心代码：**

stack, res, multi = [], "", 0

        for c in s:

            if c == '[':

                stack.append([multi, res])

                res, multi = "", 0

            elif c == ']':

                cur\_multi, last\_res = stack.pop()

                res = last\_res + cur\_multi \* res

            elif '0' <= c <= '9':

                multi = multi \* 10 + int(c)

            else:

                res += c

        return res

# 7 Greedy

1 一般在数据没有明显结构或者状态转换之类，又有一定优先级需要安排同时又无法划分子问题进行分治时，可以考虑贪心的思想。

2 贪心算法的重点在于确定贪心的依据。

**3 贪心的有效性建立在整体的最优决定于局部的最优，如果没有这样的约束，局部的贪心并不能得到整体的最优，此时应该使用动态规划。**

### 45 \*Jump Game 2

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

**Example:**

**Input:** [2,3,1,1,4]

**Output:** 2

**Explanation:** The minimum number of jumps to reach the last index is 2.

Jump 1 step from index 0 to 1, then 3 steps to the last index.

核心思想：

每一个元素都有一个可jump的范围，记录在这个范围内的元素能跳到的最远的位置，即为下一个范围，如此不断往前，每个范围的更新，即为一步。也就是贪心的思想，记录最大的。

核心代码：

int curfathest = 0, curend=0, step = 0;

for(int i=0;i<nums.size()-1;i++){

curfathest = max(curfathest,nums[i]+i);

if(i==curend){

step++;

curend=curfathest;

}

}

### 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。 每个人由一个整数对(h, k)表示，其中h是这个人的身高，k是排在这个人前面且身高大于或等于h的人数。 编写一个算法来重建这个队列。

注意：

总人数少于1100人。

示例

输入:

[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

输出:

[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]

**核心思想：**

让我们从最简单的情况下思考，当队列中所有人的 (h,k) 都是相同的高度 h，只有 k 不同时，解决方案很简单：每个人在队列的索引 index = k。

即使不是所有人都是同一高度，这个策略也是可行的。因为个子矮的人相对于个子高的人是 “看不见” 的，所以可以先安排个子高的人。

**核心代码：**

        people.sort(key=lambda x: (-x[0], x[1]))

        res = []

        for p in people:

            res.insert(p[1], p)

        return res

### 1353 [最多可以参加会议的数目](#_1353_最多可以参加会议的数目)

### 5359 最大的团队表现值

公司有编号为 1 到 n 的 n 个工程师，给你两个数组 speed 和 efficiency ，其中 speed[i] 和 efficiency[i] 分别代表第 i 位工程师的速度和效率。请你返回由最多 k 个工程师组成的 **​​​​​​最大团队表现值** ，由于答案可能很大，请你返回结果对 10^9 + 7 取余后的结果。

**团队表现值** 的定义为：一个团队中「所有工程师速度的和」乘以他们「效率值中的最小值」。

**示例 1：**

**输入：**n = 6, speed = [2,10,3,1,5,8], efficiency = [5,4,3,9,7,2], k = 2

**输出：**60

**核心思想：**

本题的贪心算法是有**两个贪心目标**的问题，对于这种情况，只考虑其中一种的优先级是违背问题的，但是如何同时考虑两个问题的优先级呢？

**在一个贪心目标的优先级的基础上，考虑另一个优先级。**

例如本题中，可以以效率的优先级为基础，从高到低，依次处理当前效率下最大的速度和（注意这里的当前效率就是整个团队的最小效率，而不是整个团队大于等于该效率）。当然，也可以先考虑速度。

此外，为了算法的效率，用堆来维护随效率动态变化的最大速度集合。

**核心代码：**

res = 0

sp\_k = []

sum\_sp = 0

ef\_idxs = list(zip(efficiency, speed))

ef\_idxs = sorted(ef\_idxs, reverse=True)

for i, ef\_idx in enumerate(ef\_idxs):

ef, sp = ef\_idx

if i<k:

heapq.heappush(sp\_k, sp)

sum\_sp+=sp

else:

if sp<=sp\_k[0]:

continue

else:

sum\_sp+=sp

sum\_sp-=heapq.heappop(sp\_k)

heapq.heappush(sp\_k, sp)

res = max(res, sum\_sp\*ef)

return res%(10\*\*9+7)

# 8 Pointer

**Pointer可分为单指针、多指针、指针数组，**

**单指针往往用于记录某种全局信息，如最大值、最小值。**

**多指针，可分为双向指针、双值指针、快慢指针，窗口指针。**

双向指针用于记录左右边界，也叫左右指针，分别从左右逼近目标值

双值指针，可以理解为两个单指针，所以除了双值指针还是多值指针。

快慢指针，

窗口指针，两份指针分别表示窗口的左右边界。往往右指针用于满足条件，左指针用于缩小满足条件的范围。

**指针数组表示对目标数列的每个位置都建立一个特定意义的指针。**

## 8.1 双向指针/指针数组

在**有序**的前提下寻找符合特定条件的两个元素时，相对于暴力的两层遍历叠加，可以使用一头一尾两个指针，分别从前后逼近查找。

### 15 3 sum

给定一个包含 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 ？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。

核心代码：

nums.sort()

for i in range(0, n-2):

    if nums[i]>0:

        return res

    if i>0 and nums[i]==nums[i-1]:

        continue

    l, r = i+1, n-1

    while l<r:

        if nums[l]+nums[r]==-nums[i]:

            res.append([nums[i], nums[l], nums[r]])

            while l<r-1 and nums[l]==nums[l+1]:

                l+=1

            l+=1

            while r>l+1 and nums[r]==nums[r-1]:

                r-=1

            r-=1

        elif nums[l]+nums[r]>-nums[i]:

            r-=1

        else:

            l+=1

return res

### 42 Trapping Rain Water

Given *n* non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.



**核心思想**：

水池中每一格能够装的水，取决于其左右最长的墙中较小的一个，由此有两种方法：

1. 使用指针数组记录每个位置的leftmax, rightmax.
2. 在1的基础上利用two pointer代替指针数组进行实现。

**核心代码**：

int ans = 0, size = height.size();

vector<int> left\_max(size), right\_max(size);

left\_max[0] = height[0];

for (int i = 1; i < size; i++)

left\_max[i] = max(height[i], left\_max[i - 1]);

right\_max[size - 1] = height[size - 1];

for (int i = size - 2; i >= 0; i--)

right\_max[i] = max(height[i], right\_max[i + 1]);

for (int i = 1; i < size - 1; i++)

ans += min(left\_max[i], right\_max[i]) - height[i];

int left = 0, right = height.size() - 1;

int ans = 0;

int left\_max = 0, right\_max = 0;

while (left < right) {

if (height[left] < height[right]) {

height[left] >= left\_max ? (left\_max = height[left]) : ans += (left\_max - height[left]);

++left;

}

else {

height[right] >= right\_max ? (right\_max = height[right]) : ans += (right\_max - height[right]);

--right;

}

}

## 8.2 滑动窗口

### 76 Minimum Window Substring

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

**Example:**

**Input: S** = "ADOBECODEBANC", **T** = "ABC"

**Output:** "BANC"

**注意：t可能包含重复的字母**

**核心思想：**滑动窗口法，使用左指针、右指针两个指针。

**核心代码：**

def minWindow(self, s: str, t: str) -> str:

window\_count = dict(zip(list(t), [0]\*len(t)))

chars\_count = collections.Counter(t)

chars\_num = 0

start\_idx = 0

min\_len = float('inf')

min\_window = ""

for i, c in enumerate(s):

if c in chars\_count:

window\_count[c] += 1

if window\_count[c] == chars\_count[c]:

chars\_num += 1

while (chars\_num == len(chars\_count)) and start\_idx <= i:

if i-start\_idx+1 < min\_len:

min\_len = i-start\_idx+1

min\_window = s[start\_idx:i+1]

if s[start\_idx] in window\_count:

cur\_c = s[start\_idx]

window\_count[cur\_c] -= 1

if window\_count[cur\_c] < chars\_count[cur\_c]:

chars\_num -= 1

start\_idx += 1

return min\_window

### 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。

在线性时间复杂度内返回滑动窗口中的最大值。

示例:

输入: nums = [1,3,-1,-3,5,3,6,7], 和 k = 3

输出: [3,3,5,5,6,7]

解释:

滑动窗口的位置 最大值

--------------- -----

[1 3 -1] -3 5 3 6 7 3

1 [3 -1 -3] 5 3 6 7 3

1 3 [-1 -3 5] 3 6 7 5

1 3 -1 [-3 5 3] 6 7 5

1 3 -1 -3 [5 3 6] 7 6

1 3 -1 -3 5 [3 6 7] 7

提示： 你可以假设 k 总是有效的，在输入数组不为空的情况下，1≤k≤输入数组的大小。

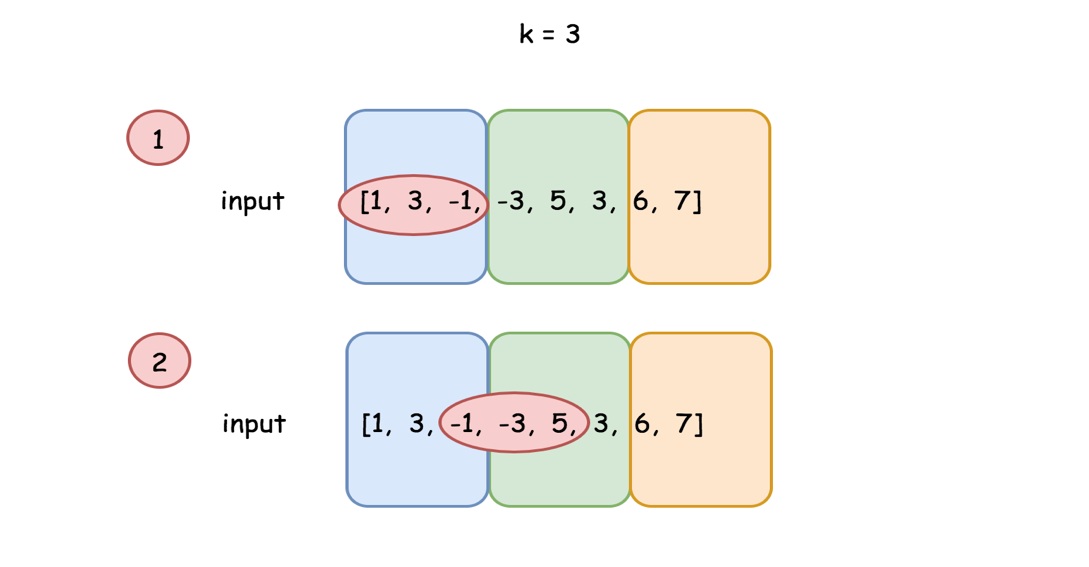
**核心思想：**

这道题解法虽然最终归结到双向队列，但是双向队列的每个元素本质还是滑动窗口左右指针的思想，一个指向向前满足条件的最相邻元素，一个指向向后的满足条件的最相邻元素，只不过由于滑动窗口的单向性，需要两次遍历，此外由于滑动窗口固定的长度，指针所囊括的范围也被分段化，如此连续的指针所组成的正是双向队列。

1 为了获得滑动窗口的最大值，第一想法是遍历数组，但遍历只能获得当前元素为止的最大元素， 而不是在一定窗口内，为了实现在一定窗口内，必须限定范围，如果模拟滑动窗口的移动限定范围，则每一次移动都要观察窗口去除的值、添加的值与原本最大值的关系，无法在O(n)时间内解决，那么在限定范围的基础上肯定不能移动。

**关键**在于**这限定范围的基础上不移动，**那么假设以k为块划分数列，求在该块内这个元素之前所有元素的最大值， 但这样在移动过程中一个块并不能与当前窗口契合，如下情况2， 5只能保证是往前绿色部分的最大值，蓝色部分的最大值呢？于是便想到需要一个反向的最大值块统计，能保证-1位置是从-1开始往后蓝色部分所有元素的最大值，于是便用到双向队列。

**简而言之：双向队列通过拼接两个队列中对应块的记录解决固定块无法任意匹配范围的问题。**



**核心代码：**

        l, r = 0, n-1

        left\_max, right\_max = [0]\*n, [0]\*n

        left\_max[0], right\_max[-1] = nums[0], nums[-1]

        for i in range(1, n):

            if i%k==0:

                left\_max[i] = nums[i]

            else:

                left\_max[i] = max(left\_max[i-1], nums[i])

            j = n-1-i

            if j%k==0:

                right\_max[j] = nums[j]

            else:

                right\_max[j] = max(right\_max[j+1], nums[j])

        res = []

        for i in range(0, n-k+1):

            res.append(max(left\_max[i+k-1], right\_max[i]))

        return res

## 8.3 快慢指针

### 234 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

**输出:** false

**示例 2:**

**输入:** 1->2->2->1

**输出:** true

**核心思想：**

**快慢指针**

**核心代码：**

        prev = None

        slow, fast = head, head

        while fast and fast.next:

            fast = fast.next.next

            tmp = slow.next

            slow.next = prev

            prev = slow

            slow = tmp

        if fast:

            slow = slow.next

        while prev and slow:

            if prev.val != slow.val:

                return False

            prev = prev.next

            slow = slow.next

        return True

### 287 寻找重复数

给定一个包含 n + 1 个整数的数组 nums，其数字都在 1 到 n 之间（包括 1 和 n），可知至少存在一个重复整数。假设只有一个重复整数，找出这个重复数。

示例 1:

输入: [1,3,4,2,2]

输出: 2

示例 2:

输入: [3,1,3,4,2]

输出: 3

说明：

不能更改原数组（假设数组是只读的）。

只能使用额外的 O(1) 的空间。

时间复杂度小于 O(n2) 。

数组中只有一个重复的数字，但它可能不止重复出现一次。

**核心思想：  
 快慢指针法**（循环查找法），因为取值在1～n之间，所以从头开始以当前的值作为索引寻找下一个值必然存在环，环的入口（即环与线相交的点）即为重复的数，使用快慢指针a、b，a每次走两步，b每次走一步，必然会在环中相遇。

重新声明一个指针c从头开始，b继续从相遇点开始，每次各走一步，由快慢指针步数相差两倍可知当c和b必然会在相遇点再次相遇，而在该相遇之前，a、c两个指针首先在环的入口相遇。

**核心代码：**

slow, fast = 0, 0

        slow, fast = nums[slow], nums[nums[fast]]

        while slow != fast:

            slow, fast = nums[slow], nums[nums[fast]]

        head = 0

        while head != slow:

            head, slow = nums[head], nums[slow]

        return head

# 9 Concepts and Observations

**为了获得一个数列中累和、累积或者其他操作结果满足要求额连续子序列，可以使用累计存储结合前缀相减， 如accum(m,n) = accumm(n)-accum(m) n>m.**

## 9.1 辅助栈

### 面试题30. 包含min函数的栈

定义栈的数据结构，请在该类型中实现一个能够得到栈的最小元素的 min 函数在该栈中，调用 min、push 及 pop 的时间复杂度都是 O(1)。

**核心思想：关键在于将min函数的时间复杂度由O(n)降为O(1)，min函数在这里的含义为当前为止所有数值的最小数，关键在于当前为止，随着pop（）和push()最小值都会跟着pop（）和push（），那么需要建立一个最小值的栈，表示当前为止的最小值，该栈存储的数列是非严格递减的数 列。**

**核心代码：**

class MinStack:

    def \_\_init\_\_(self):

        self.A = []

        self.B = []

    def push(self, x: int) -> None:

        self.A.append(x)

        if not self.B or self.B[-1]>=x:

            self.B.append(x)

    def pop(self) -> None:

        x = self.A.pop()

        if self.B[-1]==x:

            self.B.pop()

    def top(self) -> int:

        return self.A[-1]

    def min(self) -> int:

        return self.B[-1]

### 32 最长有效括号

给定一个只包含 '(' 和 ')' 的字符串，找出最长的包含有效括号的子串的长度。

示例 1:

输入: "(()"

输出: 2

解释: 最长有效括号子串为 "()"

示例 2:

输入: ")()())"

输出: 4

解释: 最长有效括号子串为 "()()"

**核心思想：**

从左往右实现动规，当前字符为’(‘时肯定无效，当前字符为‘)’时分两种情况：

具体见代码

**核心代码：**

动规思想：for (int i = 1; i < s.length(); i++) {

if (s.charAt(i) == ')') {

if (s.charAt(i - 1) == '(') {

dp[i] = (i >= 2 ? dp[i - 2] : 0) + 2;

} else if (i - dp[i - 1] > 0 && s.charAt(i - dp[i - 1] - 1) == '(') {

dp[i] = dp[i - 1] + ((i - dp[i - 1]) >= 2 ? dp[i - dp[i - 1] - 2] : 0) + 2;

}

maxans = Math.max(maxans, dp[i]);

}

}

递归思想：

    stack, max\_ans = [-1], 0

    for i, c in enumerate(s):

        if c=='(':

            stack.append(i)

        else:

            idx = stack.pop()

            if len(stack)==0:

                stack.append(i)

            else:

                max\_ans = max(max\_ans, i-stack[-1])

return max\_ans

指针思想：

基于两条原则：

1 当左括号数等于右括号数时，必定是有效序列，可以更新

2 当左括号数小于右括号数时，必定无效，可以归零

由此从左往右寻找，但会有一个问题，到最后的情况如果右括号数小于左括号数，会漏判，于是需要再从右往左寻找一次。

int left = 0, right = 0, maxlength = 0;

for (int i = 0; i < s.length(); i++) {

if (s.charAt(i) == '(') {

left++;

} else {

right++;

}

if (left == right)maxlength = Math.max(maxlength, 2 \* right);

else if (right >= left) left = right = 0;

}

left = right = 0;

for (int i = s.length() - 1; i >= 0; i--) {

if (s.charAt(i) == '(') {

left++;

} else {

right++;

}

if (left == right) maxlength = Math.max(maxlength, 2 \* left);

else if (left >= right)left = right = 0;

}

### 146 \*LRU缓存机制

设计和实现一个  LRU (最近最少使用) 缓存机制。它应该支持以下操作： 获取数据 get 和 写入数据 put 。

获取数据 get(key) - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。

写入数据 put(key, value) - 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的数据值，从而为新的数据值留出空间。

进阶:

你是否可以在 O(1) 时间复杂度内完成这两种操作？

示例:

LRUCache cache = new LRUCache( 2 /\* 缓存容量 \*/ );

cache.put(1, 1);

cache.put(2, 2);

cache.get(1); // 返回 1

cache.put(3, 3); // 该操作会使得密钥 2 作废

cache.get(2); // 返回 -1 (未找到)

cache.put(4, 4); // 该操作会使得密钥 1 作废

cache.get(1); // 返回 -1 (未找到)

cache.get(3); // 返回 3

cache.get(4); // 返回 4

**核心思想：**

可以用OrderedDict实现，更基本的可以用哈希表+双向链表实现OrderedDict。

为什么要用双向链表？

因为删除节点时需要知道前驱，只有双向链表才能在O(1)时间内找到前驱。

**核心代码：**

**方法一：**

class LRUCache(OrderedDict):

def \_\_init\_\_(self, capacity):

self.capacity = capacity

def get(self, key):

if key not in self:

return - 1

self.move\_to\_end(key)

return self[key]

def put(self, key, value):

if key in self:

self.move\_to\_end(key)

self[key] = value

if len(self) > self.capacity:

self.popitem(last = False)

**方法二：**

class DLinkedNode():

    def \_\_init\_\_(self):

        self.key = 0

        self.value = 0

        self.prev = None

        self.next = None

class LRUCache():

    def \_\_init\_\_(self, capacity):

        self.cache = {}

        self.size = 0

        self.capacity = capacity

        self.head, self.tail = DLinkedNode(), DLinkedNode()

        self.head.next = self.tail

        self.tail.prev = self.head

    def \_add\_node(self, node):

        node.prev = self.head

        node.next = self.head.next

        self.head.next.prev = node

        self.head.next = node

    def \_remove\_node(self, node):

        prev = node.prev

        new = node.next

        prev.next = new

        new.prev = prev

    def \_move\_to\_head(self, node):

        self.\_remove\_node(node)

        self.\_add\_node(node)

    def \_pop\_tail(self):

        res = self.tail.prev

        self.\_remove\_node(res)

        return res

    def get(self, key):

        node = self.cache.get(key, None)

        if not node:

            return -1

        self.\_move\_to\_head(node)

        return node.value

    def put(self, key, value):

        node = self.cache.get(key)

        if not node:

            newNode = DLinkedNode()

            newNode.key = key

            newNode.value = value

            self.cache[key] = newNode

            self.\_add\_node(newNode)

            self.size += 1

            if self.size > self.capacity:

                tail = self.\_pop\_tail()

                del self.cache[tail.key]

                self.size -= 1

        else:

            node.value = value

            self.\_move\_to\_head(node)

### 148 排序链表

在 O(nlogn) 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:

输入: 4->2->1->3

输出: 1->2->3->4

示例 2:

输入: -1->5->3->4->0

输出: -1->0->3->4->5

**核心思想：**归并+链表的**cut、merge**+**链表头指针**的使用

**核心代码：**

def sortList(self, head: ListNode) -> ListNode:

        l = 0

        p = head

        while p:

            l+=1

            p = p.next

        dummy\_head = ListNode(0)

        dummy\_head.next = head

        step = 1

        while step<=l:

            tail = dummy\_head

            cur = dummy\_head.next

            while cur:

                left = cur

                right = self.cut(cur, step)

                cur = self.cut(right, step)

                tail.next = self.merge(left, right)

                while tail.next:

                    tail = tail.next

            step \*= 2

        return dummy\_head.next

    def cut(self, cur, step):

        p = cur

        while p and step-1:

            step -= 1

            p = p.next

        if not p:

            return None

        tmp = p.next

        p.next = None

        return tmp

    def merge(self, left, right):

        dummy\_head = ListNode(0)

        p = dummy\_head

        while left and right:

            if left.val <= right.val:

                p.next = left

                p = p.next

                left = left.next

            else:

                p.next = right

                p = p.next

                right = right.next

        p.next = left if left else right

return dummy\_head.next

### 189 Rotate Array

Given an array, rotate the array to the right by k steps, where k is non-negative.

**Input:** [1,2,3,4,5,6,7] and k = 3

**Output:** [5,6,7,1,2,3,4]

**Explanation:**

rotate 1 steps to the right: [7,1,2,3,4,5,6]

rotate 2 steps to the right: [6,7,1,2,3,4,5]

rotate 3 steps to the right: [5,6,7,1,2,3,4]

**核心思想1**：用O(1)的空间复杂度，说明只能依序移动，用临时变量存储替换时被覆盖的值，注意由于是依序，不是交换，所以需要两个变量，一个存取上一个被覆盖的值，一个存取将要覆盖的值，此外，依序移动时，可能会陷入死循环，**观察得需要的迭代次数与k与nums.size()的最大公约数a有关**，。

int a = nums.size(),b = k,tmp1,tmp2;

while(a!=b){

if(a<b){ tmp1 = a; a = b; b = tmp1; }

else if(a>b) a = a-b;

}

b = nums.size()/a;

for(int i=0;i<a;i++){

tmp1 = nums[i];

tmp2 = nums[(k+i)%nums.size()];

for(int j=0;j<b;j++){

nums[((j+1)\*k+i)%nums.size()] = tmp1;

tmp1 = tmp2;

tmp2 = nums[((j+2)\*k+i)%nums.size()];

}

}

**核心思想2：**

当我们旋转数组 k 次， k%n 个尾部元素会被移动到头部，剩下的元素会被向后移动。

我们首先将所有元素反转。然后反转前 k 个元素，再反转后面 n−k 个元素，就能得到想要的结果。

**核心代码：**

**public void rotate(int[] nums, int k) {**

**k %= nums.length;**

**reverse(nums, 0, nums.length - 1);**

**reverse(nums, 0, k - 1);**

**reverse(nums, k, nums.length - 1);**

**}**

**public void reverse(int[] nums, int start, int end) {**

**while (start < end) {**

**int temp = nums[start];**

**nums[start] = nums[end];**

**nums[end] = temp;**

**start++;**

**end--;**

**}**

**}**

### 229 Majority Element II

Given an integer array of size *n*, find all elements that appear more than ⌊ n/3 ⌋ times.

**Note:**The algorithm should run in linear time and in O(1) space.

核心思想：使用Boyer Moore majority vote算法，最多可能出现几个满足条件的结果，就用几个标记量，**需要注意**，**遍历完成后剩下的标记量不一定满足条件，可能正好是最后插进来的数，所以需要再遍历一次，判断其出现的次数。**

核心代码：

int a,b,num\_a=0,num\_b=0;

for(int i=0;i<nums.size();i++){

if(num\_a!=0&&nums[i]==a)num\_a++;

else if(num\_b!=0&&nums[i]==b)num\_b++;

else if(num\_a==0){a=nums[i];num\_a=1;}

else if(num\_b==0){b=nums[i];num\_b=1;}

else{num\_a--;num\_b--;}

}

vector<int> res;

if(num\_a!=0&&ct(nums,a)>nums.size()/3)res.push\_back(a);

if(num\_b!=0&&ct(nums,b)>nums.size()/3)res.push\_back(b);

### 260 只出现一次的数字

给定一个整数数组 nums，其中恰好有两个元素只出现一次，其余所有元素均出现两次。 找出只出现一次的那两个元素。

示例 :

输入: [1,2,1,3,2,5]

输出: [3,5]

**核心思想：**

首先如果只有一个只出现一次的数字，那么对整个数组异或，就能找到。

但这道题，要寻找两个数字，全部异或后不是我们所要的结果，而题目要求线性时间，常数空间，那么肯定还是要使用异或，那么能否想办法把数组分为两部分？ 使得只出现过一次的两个数字分别在两组里边。

那么通过什么把数组分成两组呢？

放眼到二进制，我们要找的这两个数字是不同的，所以它俩至少有一位是不同的，所以根据两个数的异或结果的这一位，把数组分成这一位都是 1 的一类和这一位都是 0 的一类，这样就把这两个数分到两组里了。

关于快速找到一个数最低位的1的小技巧：用到补码（计算机中数值的存储统一用补码）的知识，负数的补码为对应正数反码再加1， 如-10补码为10（1010）的反码（0101）加1（0110），正负整数相与：（1010）&（0110）= 0010，即找到了最低位为1并且其他位全置为0的数。

然后用上面找到的只有一位为1的树，与数组中的元素相与，结果为0的一组，不为0的为另一组，各自异或，得解。

核心代码：

def singleNumber(self, nums: List[int]) -> List[int]:

    bitmask = 0

    for num in nums:

        bitmask ^= num

    diff = bitmask & (-bitmask)

    x = 0

    for num in nums:

        if num & diff:

            x ^= num

    return [x, bitmask^x]

### 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:

输入: [5,2,6,1]

输出: [2,1,1,0]

解释:

5 的右侧有 2 个更小的元素 (2 和 1).

2 的右侧仅有 1 个更小的元素 (1).

6 的右侧有 1 个更小的元素 (1).

1 的右侧有 0 个更小的元素.

**核心思想：**

**1 归并排序+索引数组**

**2 二叉搜索树，倒序添加（基本解法）**

**3 线段树， 倒序添加**

**4 数状数组**

**核心代码：**

class TreeNode:

    def \_\_init\_\_(self, x):

        self.val = x

        self.count = 0

        self.left = None

        self.right = None

class Node():

    def \_\_init\_\_(self,begin,end):

        self.begin = begin

        self.end = end

        self.mid = (begin+end)//2

        self.count = 0

        self.left = None

        self.right = None

    def add(self,num):

        # 返回线段树中比num小的值的数量

        self.count += 1

        if self.begin == self.end:

            return 0

        else:

            if not self.left:

                self.left = Node(self.begin, self.mid)

            if not self.right:

                self.right = Node(self.mid + 1, self.end)

            if num <= self.mid:

                return self.left.add(num)

            else:

                return self.left.count + self.right.add(num)

class Solution:

    def countSmaller(self, nums: List[int]) -> List[int]:

        if not nums:

            return []

        n = len(nums)

    # 解法1: 归并排序+索引数组

        indexes = list(range(n))

        tmp = list(range(n))

        res = [0]\*n

        def helper(s, e):

            if s==e:

                return

            mid = int((s+e)/2)

            helper(s, mid)

            helper(mid+1, e)

            if nums[indexes[mid]] <= nums[indexes[mid + 1]]:

                return

            sort\_count(s, mid, e)

        def sort\_count(s, mid, e):

            for i in range(s, e+1):

                tmp[i] = indexes[i]

            l, r = s, mid+1

            for i in range(s, e+1):

                if l>mid:

                    indexes[i] = tmp[r]

                    r+=1

                elif r>e:

                    indexes[i] = tmp[l]

                    l+=1

                    res[indexes[i]] += e-mid

                elif nums[tmp[l]] <= nums[tmp[r]]:

                    indexes[i] = tmp[l]

                    l+=1

                    res[indexes[i]] += (r-mid-1)

                else:

                    indexes[i] = tmp[r]

                    r+=1

        helper(0, n-1)

        return res

    # 解法2: 平衡搜索树

        res = [0] \* n

        def insert(root, num, i):

            if not root:

                root = TreeNode(num)

                return root

            if num > root.val:

                res[i] += root.count+1

                root.right = insert(root.right, num ,i)

            else:

                root.count += 1

                root.left = insert(root.left, num, i)

            return root

        root = TreeNode(nums[-1])

        for i in range(len(nums)-2, -1, -1):

            insert(root, nums[i], i)

        return res

 # 解法3：线段树

        mn = min(nums)

        mx = max(nums)

        root = Node(mn,mx)

        res = []

        # 这里倒着计算，返回数组前面比自己小的个数

        for i in range(len(nums)-1,-1,-1):

            # 一边构建线段树一边计算答案

            res.append(root.add(nums[i]))

        return res[::-1]

# 解法4：树状数组

        class FenwickTree:

            def \_\_init\_\_(self, n):

                self.size = n

                self.tree = [0 for \_ in range(n + 1)]

            def \_\_lowbit(self, index):

                return index & (-index)

            def update(self, index, delta):

                while index <= self.size:

                    self.tree[index] += delta

                    index += self.\_\_lowbit(index)

            def query(self, index):

                res = 0

                while index > 0:

                    res += self.tree[index]

                    index -= self.\_\_lowbit(index)

                return res

        size = len(nums)

        if size == 0:

            return []

        # 去重，方便离散化

        s = list(set(nums))

        s\_len = len(s)

        # 离散化，借助堆

        import heapq

        heapq.heapify(s)

        rank\_map = dict()

        rank = 1

        for \_ in range(s\_len):

            num = heapq.heappop(s)

            rank\_map[num] = rank

            rank += 1

        fenwick\_tree = FenwickTree(s\_len)

        res = [None for \_ in range(size)]

        # 从后向前填表

        for index in range(size - 1, -1, -1):

            # 1、查询排名

            rank = rank\_map[nums[index]]

            # 2、在树状数组排名的那个位置 + 1

            fenwick\_tree.update(rank, 1)

            # 3、查询一下小于等于“当前排名 - 1”的元素有多少

            res[index] = fenwick\_tree.query(rank - 1)

        return res

### 795 Number of Subarrays with Bounded Maximum

Given an array A of positive integers, and two positive integers L and R (L <= R).

Return the number of (contiguous, non-empty) subarrays such that the value of the maximum array element in that subarray is at least L and at most R.

**Example :**

**Input:**

A = [2, 1, 4, 3]

L = 2

R = 3

**Output:** 3

**Explanation:** There are three subarrays that me

**核心思路：**

count(a~b) = count(b)-count(a)

**核心代码：**

class Solution(object):

def numSubarrayBoundedMax(self, A, L, R):

def count(bound):

ans = cur = 0

for x in A :

cur = cur + 1 if x <= bound else 0

ans += cur

return ans

return count(R) - count(L - 1)

### 1238 循环码列

给你两个整数 n 和 start。你的任务是返回任意 (0,1,2,,...,2^n-1) 的排列 p，并且满足：

p[0] = start

p[i] 和 p[i+1] 的二进制表示形式只有一位不同

p[0] 和 p[2^n -1] 的二进制表示形式也只有一位不同

示例 1：

输入：n = 2, start = 3

输出：[3,2,0,1]

解释：这个排列的二进制表示是 (11,10,00,01)

所有的相邻元素都有一位是不同的，另一个有效的排列是 [3,1,0,2]

示例 2：

输出：n = 3, start = 2

输出：[2,6,7,5,4,0,1,3]

解释：这个排列的二进制表示是 (010,110,111,101,100,000,001,011)

**核心思想：** 本质是格雷码，例如： 生成一个3位格雷码的过程

初始: 0,1

复制前一行，添加前缀0: 00, 01

逆序复制前一行，添加前缀1: 11, 10

于是得到 00, 01, 11, 10

复制前一行，添加前缀0: 000, 001, 011, 010

逆序复制前一行，添加前缀1: 110, 111, 101, 100

于是得到 000, 001, 011, 010, 110, 111, 101, 100

那么本题的思路为：

1 生成格雷码

2 找到start的位置

3 旋转数组

**核心代码：**

        res = [0, 1]

        i = 1

        while i<n:

            res += [x+len(res) for x in res[::-1]]

            i+=1

        for j in range(2\*\*n):

            if res[j] == start:

                i = j

                break

        res[:(i+1)] = res[:(i+1)][::-1]

        res[(i+1):] = res[(i+1):][::-1]

        return res

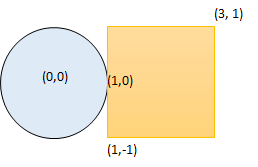
### 1401 圆和矩形是否重叠

给你一个以 (radius, x\_center, y\_center) 表示的圆和一个与坐标轴平行的矩形 (x1, y1, x2, y2)，其中 (x1, y1) 是矩形左下角的坐标，(x2, y2) 是右上角坐标。

如果圆和矩形有重叠的部分，请你返回 True ，否则返回 False 。

换句话说，请你检测是否 存在 点 (xi, yi) ，它既在圆上也在矩形上（两者都包括点落在边界上的情况）。

示例 1：

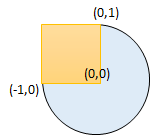


输入：radius = 1, x\_center = 0, y\_center = 0, x1 = 1, y1 = -1, x2 = 3, y2 = 1

输出：true

解释：圆和矩形有公共点 (1,0)

示例 2：



输入：radius = 1, x\_center = 0, y\_center = 0, x1 = -1, y1 = 0, x2 = 0, y2 = 1

输出：true

**核心思想：**

以矩形为参考，考虑圆的位置，而圆的位置最好由圆心代表。

相对位置分为两种情况，圆心在矩形外，圆心在矩形内。

1）圆心在矩形内，直接为true

2）圆心在矩形外，分为八种情况，分别对应矩形外围的八个区域，上下左右，上左，上右，下左，下右。

**核心代码：**

a = max(0, x1 - x\_center, x\_center - x2) \*\* 2

b = max(0, y1 - y\_center, y\_center - y2) \*\* 2

c = radius \*\* 2

return a + b <= c

# 10 Math

**观察问题的数学性质，这类题目是最考分析能力的，没有固定的范式，思考时思考其本质、考虑好边界条件，如果发现情况太复杂，尝试换一个角度。**

**1 多使用位运算**

### 31 Next Permutation

Implement **next permutation**, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as sorted in ascending order.

The replacement must be [**in-place**](http://en.wikipedia.org/wiki/In-place_algorithm) and use only constant extra memory.

1,2,3 → 1,3,2  
3,2,1 → 1,2,3  
1,1,5 → 1,5,1

核心思想：重点是要弄清楚排列组合的性质，正好比当前排列大一点的排列（或者叫下一个排列）即从后往前将所有nums[i]<nums[i-1]的序列倒置，直到找到nums[i]>nums[i-1]，然后将nums[i-1]插入到后面的序列使后面的序列最小，然后将后面数列正好大于nums[i-1]的数放到nums[i-1]原来的位置。

核心代码：

vector<int>::iterator i=nums.end()-1;

int tmp=\*i;

vector<int>::iterator loc,it;

while(i!=nums.begin()){

if(tmp<=\*(i-1)){

tmp = \*(i-1);

nums.erase(i-1);

nums.push\_back(tmp);

i=i-1;

}

else{

tmp = \*(i-1);

it = loc = nums.erase(i-1);

while(it!=nums.end()){

if(\*it>tmp){

nums.insert(it,tmp);

tmp = \*(it+1);

nums.erase(it+1);

nums.insert(loc,tmp);

return;

}

it = it+1;

}

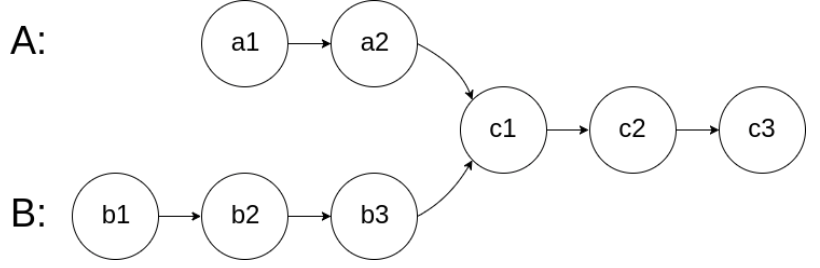
}

}

### 160 相交链表

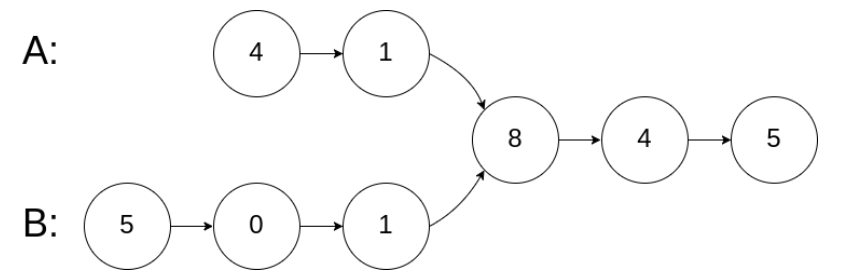
编写一个程序，找到两个单链表相交的起始节点。

如下面的两个链表**：**



在节点 c1 开始相交。

**示例 1：**



**核心思想：**

**a+all+b == b+all+a**

**核心代码：**

       p1, p2 = headA, headB

        if not p1 or not p2:

            return None

        while p1!=p2:

            p1 = p1.next

            p2 = p2.next

            if not p1 and not p2:

                return None

            if not p1:

                p1 = headB

            if not p2:

                p2 = headA

        return p1

### 264 Ugly Number II

Write a program to find the n-th ugly number.

Ugly numbers are**positive numbers** whose prime factors only include 2, 3, 5.

**Example:**

**Input:** n = 10

**Output:** 12

**Explanation:** 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 is the sequence of the first 10 ugly numbers.

**核心思想： 弄清楚因式分解的意义，大的数值都由小的数值与因子相乘得到。**

**核心代码：**

int c2 = 0,c3 =0,c5=0;

int n2 = 2,n3 = 3,n5 = 5;

int []dp = new int[n];

dp[0] = 1;

for(int i=1;i<n;i++){

int x = Math.min(n2,Math.min(n3,n5));

dp[i] = x;

if(dp[i]==n2) { n2 = 2\*dp[++c2];}

if(dp[i]==n3) { n3 = 3\*dp[++c3];}

if(dp[i]==n5) { n5 = 5\*dp[++c5];}

}

return dp[n-1];

### 335 Self Crossing

You are given an array *x* of n positive numbers. You start at point (0,0) and moves x[0] metres to the north, then x[1] metres to the west, x[2] metres to the south, x[3] metres to the east and so on. In other words, after each move your direction changes counter-clockwise.

Write a one-pass algorithm with O(1) extra space to determine if your path crosses itself or not.

**Example 1:**

**┌───┐**

**│   │**

**└───┼──>**

**│**

**Input:** [2,1,1,2]

**Output:** true

**Example 2:**

**┌──────┐**

**│      │**

**│**

**│**

**└────────────>**

**Input:** [1,2,3,4]

**Output:** false

**Example 3:**

**┌───┐**

**│   │**

**└───┼>**

**Input:** [1,1,1,1]

**Output:** true

**核心思想：  
学会概括性思维（从数学角度上，满足条件的是哪些情况），而不是仿真思维。**

**核心代码：**

**for pos in range(3, len(x)):**

**if x[pos - 1] <= x[pos - 3] and x[pos] >= x[pos - 2]: # general case**

**return True**

**# connected rectangle**

**if pos >= 4 and x[pos - 1] == x[pos - 3] and x[pos] + x[pos - 4] >= x[pos - 2]:**

**return True**

**# connected L-shape**

**if pos >= 5 and x[pos-1] <= x[pos-3] and x[pos-3] <= x[pos-1] + x[pos-5] and x[pos] + x[pos-4] >= x[pos-2] and x[pos-4] <= x[pos-2]:**

**return True**

### 372 超级次方

你的任务是计算 a\*\*b 对 1337 取模，a 是一个正整数，b 是一个非常大的正整数且会以数组形式给出。

示例 1:

输入: a = 2, b = [3]

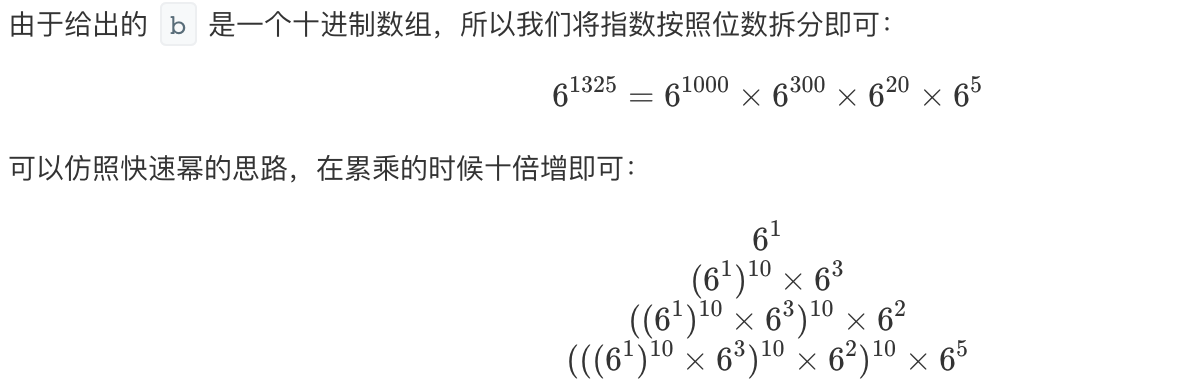
输出: 8

示例 2:

输入: a = 2, b = [1,0]

输出: 1024

**核心思想：**

****

关键点：如何对x的n次方求对m的余？

假设x为10， 二进制表示为1010， 则可以通过核心代码中的qpow函数计算

**核心代码：**

def qpow(self, x, n, m):

# 学会用模的思想

        ans = 1

        while n > 0:

            if n & 1 == 1:

                ans = ans \* x % m

            x = x \* x % m

            n >>= 1

        return ans

    def superPow(self, a: int, b: List[int]) -> int:

        res = 1

        for i in b:

            res = self.qpow(res, 10, 1337) \* self.qpow(a, i, 1337)

        return res % 1337

### 523 Continuous Subarray Sum

Given a list of **non-negative** numbers and a target **integer** k, write a function to check if the array has a continuous subarray of size at least 2 that sums up to a multiple of **k**, that is, sums up to n\*k where n is also an **integer**.

**Example 1:**

**Input:** [23, 2, 4, 6, 7], k=6

**Output:** True

**Explanation:** Because [2, 4] is a continuous subarray of size 2 and sums up to 6.

**核心思想：**

模k得r的整数n加上能被k整除的数m后，模k仍然得到r。

**核心代码：**

def checkSubarraySum(self, nums: List[int], k: int) -> bool:

s = 0

sum\_set = set()

pre = 0

for n in nums:

s += n

cur = s%k if k else s

if cur in sum\_set:

return True

sum\_set.add(pre)

pre = cur

return False

### 650 2 Keys Keyboard

Initially on a notepad only one character 'A' is present. You can perform two operations on this notepad for each step:

**Copy All**: You can copy all the characters present on the notepad (partial copy is not allowed).

**Paste:** You can paste the characters which are copied last time.

Given a number n. You have to get exactly n 'A' on the notepad by performing the minimum number of steps permitted. Output the minimum number of steps to get n 'A'.

**Example 1:**

**Input: 3**

**Output: 3**

**Explanation:**

**Intitally, we have one character 'A'.**

**In step 1, we use Copy All operation.**

**In step 2, we use Paste operation to get 'AA'.**

**In step 3, we use Paste operation to get 'AAA'.**

**核心思想：因式分解**

We can break our moves into groups of (copy, paste, ..., paste). Let C denote copying and P denote pasting. Then for example, in the sequence of moves CPPCPPPPCP, the groups would be [CPP][CPPPP][CP].

Say these groups have lengths g\_1, g\_2, .... After parsing the first group, there are g\_1 'A's. After parsing the second group, there are g\_1 \* g\_2 'A's, and so on. At the end, there are g\_1 \* g\_2 \* ... \* g\_n 'A's.

We want exactly N = g\_1 \* g\_2 \* ... \* g\_n. If any of the g\_i are composite, say g\_i = p \* q, then we can split this group into two groups (the first of which has one copy followed by p-1 pastes, while the second group having one copy and q-1 pastes).

**核心代码：**

def minSteps(self, n):

ans = 0

d = 2

while n > 1:

while n % d == 0:

ans += d

n /= d

d += 1

return ans

### 866 Prime Palindrome

Find the smallest prime(素数) palindrome greater than or equal to N.

* 1 <= N <= 10^8
* 答案肯定存在，且小于 2 \* 10^8。

**Example 1:**

**Input:** 6

**Output:** 7

**Example 2:**

**Input:** 8

**Output:** 11

**Example 3:**

**Input:** 13

**Output:** 101

**核心思想：**

**以回文数为基础遍历查找素数**。

关键在于如何有序构建回文数，利用到回文数的性质：回文数的左半为回文数的根，从而对称构建回文数.

**核心代码：** for length in range(1, 6):

for root in range(10\*\*(length - 1), 10\*\*length):

s = str(root)

x = int(s + s[-2::-1]) #eg. s = '123' to x = int('12321')

if x >= N and is\_prime(x):

return x

for root in range(10\*\*(length - 1), 10\*\*length):

s = str(root)

x = int(s + s[-1::-1]) #eg. s = '123' to x = int('123321')

if x >= N and is\_prime(x):

return x

### 1354 多次求和构造目标数组

给你一个整数数组 target 。一开始，你有一个数组 A ，它的所有元素均为 1 ，你可以执行以下操作：

令 x 为你数组里所有元素的和

选择满足 0 <= i < target.size 的任意下标 i ，并让 A 数组里下标为 i 处值为 x 。

你可以重复该过程任意次

如果能从 A 开始构造出目标数组 target ，请你返回 True ，否则返回 False 。

示例 1：

输入：target = [9,3,5]

输出：true

解释：从 [1, 1, 1] 开始

[1, 1, 1], 和为 3 ，选择下标 1

[1, 3, 1], 和为 5， 选择下标 2

[1, 3, 5], 和为 9， 选择下标 0

[9, 3, 5] 完成

示例 2：

输入：target = [1,1,1,2]

输出：false

解释：不可能从 [1,1,1,1] 出发构造目标数组。

**核心思想：**

逆向思维，当前target中的最大元素为上次加和之前所有元素的和。

**核心代码：**

pq = []

        cusum = 0

        for t in target:

            cusum += t

            heapq.heappush(pq, -t)

        while sum(pq)!=-len(target):

            cur = -heapq.heappop(pq)

            pre = 2\*cur-cusum

            if pre<=0:

                return False

            heapq.heappush(pq, -pre)

cusum = cur

return True

### 5383 给Nx3网格涂色的方案数

你有一个 n x 3 的网格图 grid ，你需要用 红，黄，绿 三种颜色之一给每一个格子上色，且确保相邻格子颜色不同（也就是有相同水平边或者垂直边的格子颜色不同）。

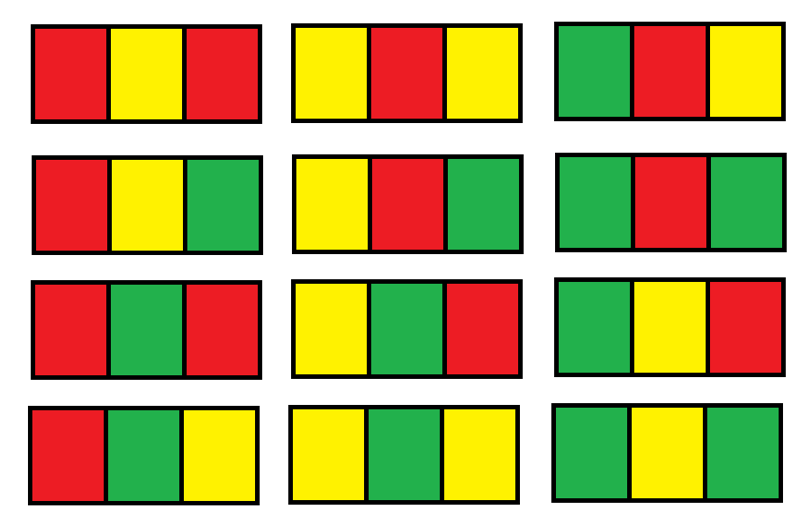
给你网格图的行数 n 。请你返回给 grid 涂色的方案数。由于答案可能会非常大，请你返回答案对 10^9 + 7 取余的结果。

示例 1：

输入：n = 1

输出：12

解释：总共有 12 种可行的方法：



**核心思想：**关键归纳出该问题的几个关键点

1: 每个颜色没有区别，ABC, CBA…是同一种情况，关键在于组合

2: 总共有两种组合： ABC, ABA

3: 每一层只受上一层限制，与更之前的层数没有关系。

基于此分析， ABC共有6种情况，ABA也有6种情况，那么对于当前层，在上一层为ABA时，当前层有5种组合（3种为ABA模式，2种ABC模式），在上一层为ABC时，当前层有4种组合（2种ABA模式，2种ABC模式）。

**核心代码：**

def numOfWays(self, n: int) -> int:

abc, aba = 6, 6

for i in range(n-1):

abc2 = abc\*2+aba\*2

aba2 = abc\*2+aba\*3

abc = abc2%(10\*\*9+7)

aba = aba2%(10\*\*9+7)

return (abc+aba)%(10\*\*9+7)

**启示：**学会归纳和简化问题。然后基于此再找规律。

### 面试题62 约瑟夫环

0,1,,n-1这n个数字排成一个圆圈，从数字0开始，每次从这个圆圈里删除第m个数字。求出这个圆圈里剩下的最后一个数字。

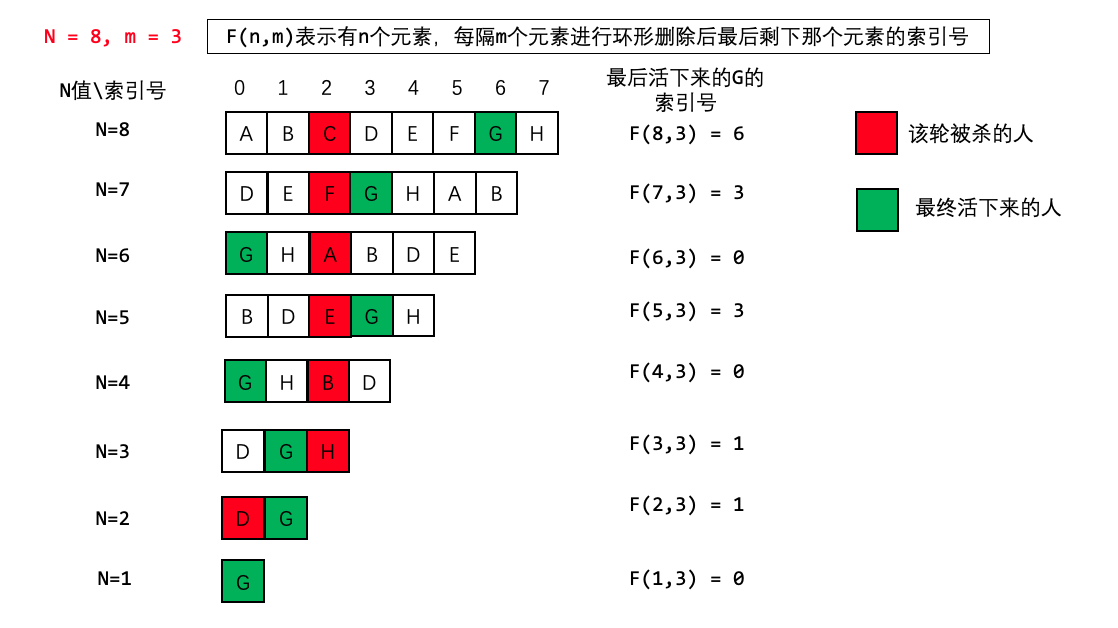
例如，0、1、2、3、4这5个数字组成一个圆圈，从数字0开始每次删除第3个数字，则删除的前4个数字依次是2、0、4、1，因此最后剩下的数字是3。

示例 1：

输入: n = 5, m = 3

输出: 3

**核心思想：**



由上可见，每次删除元素后的重新排列，可以看成是将删除后的数组左移m位，所以逆推时，右移m位即可（需要除模）

至于为什么由最后剩下的数的坐标（0）逆推，因为如果从前往后删除，肯定是over time的，自然想到由最后的数逆推，逆推出最后剩下数的坐标。

**核心代码：**

ans = 0

        for i in range(2, n+1):

            ans = (ans+m)%i

        return ans