本文档用来记录刷题过程中遇到的之前没有见过的算法或者不熟悉的算法。

1 摩尔投票算法

找出输入数组中重复出现超过一半以上的元素。如果没有这种元素，则会随意输出一个元素。所以要是没办法保证存在该元素，需要最后遍历一下验证该元素是否是满足条件的元素。

假设投票是这样的，[A, C, A, A, B]，ABC是指三个候选人。

第一张票与第二张票进行对坑，如果票不同则互相抵消掉；

接着第三票与第四票进行对坑，如果票相同，则增加这个候选人的可抵消票数；

这个候选人拿着可抵消票数与第五张票对坑，如果票不同，则互相抵消掉，即候选人的可抵消票数-1。

算法：

Cnt = 0

Res = 0

For I in range(len(nums)):

If cnt == 0:

Res = nums[i]

If nums[i] == res:

Cnt += 1

Else:

Cnt -= 1

Return res

最终得到的res是该元素。

需要做一次循环验证该元素次数是否大于一半。

如果最后得到的抵消票数不为0的话，那说明他可能希望的，这是我们需要一个阶段来验证这个候选人的票数是否超过一半——计数阶段。

所以摩尔投票法分为两个阶段：抵消阶段和计数阶段。

抵消阶段：两个不同投票进行对坑，并且同时抵消掉各一张票，如果两个投票相同，则累加可抵消的次数；

计数阶段：在抵消阶段最后得到的抵消计数只要不为0，那这个候选人是有可能超过一半的票数的，为了验证，则需要遍历一次，统计票数，才可确定。

摩尔投票法经历两个阶段最多消耗O(2n)的时间，也属于O(n)的线性时间复杂度，另外空间复杂度也达到常量级。

摩尔投票进阶算法：找出输入数组中m个重复出现超过(1/(m+1))的元素。

我们可以这样理解，假设投票是这样的[A, B, C, A, A, B, C]，ABC是指三个候选人。

第1张票，第2张票和第3张票进行对坑，如果票都不同，则互相抵消掉；

第4张票，第5张票和第6张票进行对坑，如果有部分相同，则累计增加他们的可抵消票数，如[A, 2]和[B, 1]；

接着将[A, 2]和[B, 1]与第7张票进行对坑，如果票都没匹配上，则互相抵消掉，变成[A, 1]和[B, 0]。

解决方法：使用2个变量记录当前投票最多的选手及其票数。设为one=[选手1，1],two=[选手1，0]。

For I=1 to n-1:

If num[i]==one[0]:

One[1]++

elIf num[i]==two[0]:

Two[1]++

Else:

If one[1]==0:

One[0]=num[i], one[1]=1

Elif two[1]==0:

Two[0]=num[i], two[1]=1

Else:

One[1]--, two[1]--

代码：

if len(nums) == 0:

            return []

        one = [nums[0], 1]

        two = [nums[0], 0]

        for i in range(1, len(nums)):

            if nums[i] == one[0]:

                one[1] += 1

            elif nums[i] == two[0]:

                two[1] += 1

            else:

                if one[1] == 0:

                    one[0] = nums[i]

                    one[1] = 1

                elif two[1] == 0:

                    two[0] = nums[i]

                    two[1] = 1

                else:

                    one[1] -= 1

                    two[1] -= 1

        ans = []

        if one[1] > 0:

            freq = 0

            for i in range(len(nums)):

                if nums[i] == one[0]:

                    freq += 1

            if freq > len(nums)//3:

                ans.append(one[0])

        if two[1] > 0:

            freq = 0

            for i in range(len(nums)):

                if nums[i] == two[0]:

                    freq += 1

            if freq > len(nums)//3:

                ans.append(two[0])

        return ans

2. 荷兰国旗算法：

给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

此题中，我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

注意:

不能使用代码库中的排序函数来解决这道题。

示例:

输入: [2,0,2,1,1,0]

输出: [0,0,1,1,2,2]

算法：

双指针l,r分别记录当前左侧0和右侧2应该记录的位置，也就是在右侧出现的0和左侧出现的2应该换到的位置。

然后使用一个指针cur记录当前位置

if nums[cur] == 0:

cur, l互换，因为0应该排到左边

cur++; l++因为这两个位置均已经处理过，且满足l<=cur，同时前面的遍历可以保证cur左侧的元素不存在2

elif nums[cur] == 2:

cur, r互换

r--。注意此处cur不应该修改，因为有可能互换之后cur的位置为0，和0的情况处理有差异

else:

cur++

代码：

if len(nums) == 1:

            return

        l, r = 0, len(nums) - 1

        i = 0

        while i <= r:

            if nums[i] == 0:

                nums[l], nums[i] = nums[i], nums[l]

                l += 1

                i += 1

            elif nums[i] == 2:

                nums[r], nums[i] = nums[i], nums[r]

                r -= 1

            else:

                i += 1

生成子集算法：

给定一组**不含重复元素**的整数数组 nums，返回该数组所有可能的子集（幂集）。

**说明：**解集不能包含重复的子集。

'''

        1.递归解法。从空数组开始，每次遍历数组元素时，将该元素加入到已有的子集中作为新的子集

        '''

        # ans = [[]]

        # for num in nums:

        #     ans += [cur + [num] for cur in ans]

        # return ans

        '''

        回溯解法。

        每次填充固定长度的数组，并且每个元素放进去后，应该再拿出来实现回溯。

        对于回溯，在确定回溯规则后，首先确定初始化状态，比如初始化应该数组为空，长度为0.然后就是对递归结束条件的考虑，此题中得到相应长度的解后返回。最后是递归过程的实现，长度应该为0-你，需要一层循环，然后遍历数组，每个元素有使用和不使用两个状态，首先放入，继续递归一直有该元素，等发现所有该元素的解后再拿出该元素。

        '''

        ans = []

        def dfs(index, length, res):

            if len(res) == length:

                ans.append(res.copy())

                return

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

                res.append(nums[i])

                dfs(i+1, length, res)

                res.pop(-1)

        for i in range(len(nums)+1):

            dfs(0, i, [])

        return ans

'''

        3.移位运算。

        对于长度为n的数组，子集个数为2^n，而使用n位二进制编码刚好得到相同数量的解，所以可以通过000...111表示对数组[1,2,3]的每个元素的选择。

        下面是如何生成二进制表示，因为每个二进制对应0-2^n之内的一个数，所以将2^n中的每个数拆分成二进制即可。需要注意的点是如何补全前置0，因为1应该表示为001而不是1.

        通过循环将num向右位移一位，可以得出该num是否应该加上当前位的数字。

        '''

        ans = []

        length = len(nums)

        for i in range(2\*\*length):

            index = length - 1

            tmp = []

            while index > -1:

                if i % 2 == 1:

                    tmp.append(nums[index])

                index -= 1

                i = i >> 1

            ans.append(tmp)

        return ans

快慢指针算法：

使用快慢指针找出环。

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的整数。假设只有一个重复的整数，找出这个重复的数。

采用快慢指针算法。

        证明：假设i，vi表示数组的索引及其对应的数字，如果数组中存在重复元素，那么必定有不同的索引x,y满足nums[x]==nums[y]。问题转换为找到x,y。假设重复元素出现在xx,yy位置，那么从xx->nums[xx]->nums[nums[xx]]->yy一定是一个环，对于环来说，使用双指针，一个指针的移动距离是另一个指针的二倍，一定会在某一点相遇，这个相遇点在环中。

        上述找到环后，如果两个指针的位置不同，那么该元素就是重复元素，如果位置相同，则需要指针1从0开始，而指针2从相遇点开始，找到环的入口（元素相同的位置）。

slow, fast = nums[0], nums[nums[0]]

        while fast != slow:

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

        p1, p2 = 0, slow

        while p1 != p2:

            p1, p2 = nums[p1], nums[p2]

        return p1

除此之外还有线性表找中点也需要快慢指针，和环类似。

根据中序和前序遍历恢复二叉树：

递归算法：前序遍历的第一个节点是根节点，从而找到中序遍历中的根节点，其左侧是左子树，右侧是右子树。

if len(preorder) == 0:

            return None

        root = TreeNode(preorder[0])

        index = -1

        for i in range(len(inorder)):

            if inorder[i] == preorder[0]:

                index = i

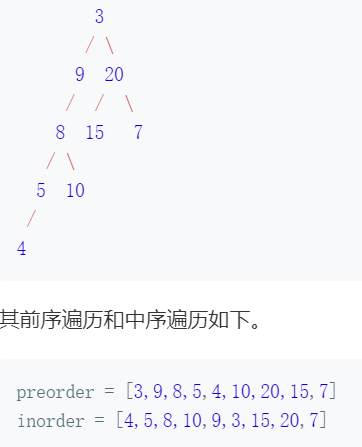
                break

        root.left = self.buildTree(preorder[1:index+1], inorder[:index])

        root.right = self.buildTree(preorder[index+1:], inorder[index+1:])

        return root

迭代法：



做法：通过栈记录当前节点值。遍历前序序列，并将栈顶元素（上一个元素）与中序的当前元素比较。

设index指向中序的当前元素。

If 前序元素==中序[index]：弹出并index++，直到栈顶元素和中序元素不同或者栈不为空，令当前元素为最后一个相等元素的右节点。

Else：则当前元素作为上一个元素的左子节点，并将当前元素压入栈中。

关键点：1.不相等则置为左子节点。因为前序遍历的一路将左子节点遍历到底的顺序，如果没有出现和中序相等的元素，则当前路径始终在左子节点上，那么必定是上个元素的左子节点。

2.如果相等，则表明当前元素是最左侧的节点元素，则中序遍历接下来的元素如果和栈顶相等就一定是之前左子节点的父节点，直到第一个不相等的元素，是上一个节点的右子节点。

所以关键就是，前序遍历刚开始是左子节点遍历到底，然后找到底之后回溯，再去寻找右子节点。

代码：

root = TreeNode(preorder[0])

        stack = [root]

        index = 0

        for i in range(1, len(preorder)):

            node = stack[-1]

            if node.val == inorder[index]:

                while stack and stack[-1].val == inorder[index]:

                    node = stack.pop()

                    index += 1

                node.right = TreeNode(preorder[i])

                stack.append(node.right)

            else:

                node.left = TreeNode(preorder[i])

                stack.append(node.left)

        return root

单调栈：

单调栈存放的数据应该是有序的，分为单调递增和递减，分别是栈底元素到栈顶元素的顺序是单调递增或递减。

单调递增栈：遍历数组时，如果栈为空或者入栈元素小于栈顶元素，则入栈，否则将所有比入栈元素小的元素全部出栈再放入入栈元素。

伪代码：

For num in nums:

If (栈空 or 栈顶元素大于等于num) 入栈

Else

While 栈不为空 and 栈顶元素小于num

出栈，并更新问题的解

Num入栈

单调栈例题：

1. 描叙：有n个人站队，所有的人全部向右看，个子高的可以看到个子低的发型，给出每个人的身高，问所有人能看到其他人发现总和是多少。  
   输入：4 3 7 1  
   输出：2  
   解释：个子为4的可以看到个子为3的发型，个子为7可以看到个子为1的身高，所以1+1=2

思路：当前数字向右查找第一个大于等于该数字之前有多少个比他小的数字，然后累加结果。使用一个单调递减的栈，如果遇到大于栈顶的元素，更新栈和结果。

For i in 0…len(nums)-1:

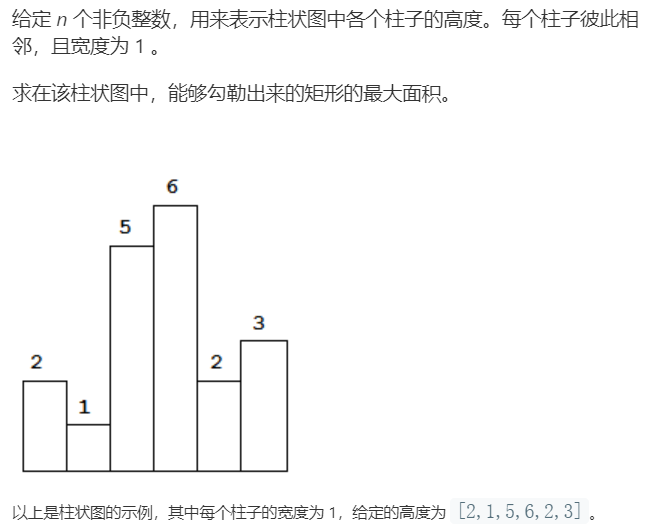
While stack and 栈顶 <= num[i]:

Top = stack.top()

Stack.pop()

Sum += (i-top-1) //这里多减一个1是因为最矮的那个人（最小的元素）看不到任何人,i表示num的索引位置

Stack.push(i)

1. 

对于高度序列来说，如果当前是升序，则最大面积一定是升序的最后一个高度（最高并且最宽）对应的答案，并且在连续出现的若干个柱状图中，面积是由最小的高度决定的。所以应该在找到每一段升序的下一个元素(A[i+1]<A[i])时更新答案，并且记录下一个答案所需的条件。而且在栈中记录的所有元素都是在当前序列起决定因素的元素，也就是最小高度。

使用单调递增栈记录当前序列的升序关系。单调递增栈内部元素按栈底到栈顶元素升序排列，并且弹出栈顶操作是因为找到了右侧数列中第一个小于栈顶元素的元素，所以需要更新答案。

对高度数组尾部插入0，表示结束，可以保证在最后一次遍历时计算最后一个柱状图为右侧边界时的面积。栈其实元素为-1，因为第一个柱状图也是需要计算面积的，宽度计算需要从-1开始，因为stack记录的是柱状图右侧边界为0开始，则访问左侧其实为-1.并且只是作为宽度的计算条件，不会涉及到访问高度，所以没有越界问题。

如果栈顶元素大于当前元素，弹出该元素，则此时矩阵面积的高为height[stack.top()]，是之前记录的决定当前序列高度的索引。矩阵的宽度，右侧边界是i-1，因为i是下一个元素，已经降序了。左侧边界是stack[-2]，，则stack[-1]是当前高度，但是因为柱状图的宽度为1，可以理解为stack记录的索引是柱状图的右侧，stack[-1]记录的是在从stack[-1]到i的区间内决定高度的索引，则stack[-2]是刚好到由stack[-1]高度决定的矩形位置的右侧，也就是stack[-2]对应的柱状图左侧边界就由height[stack[-2]]的高度决定了。更新ans。

stack = [-1]

        heights.append(0)

        ans = 0

        for i in range(len(heights)):

            while stack and heights[stack[-1]] > heights[i]:

                top = stack[-1]

                stack.pop()

                left = stack[-1] + 1

                right = i - 1

                area = (right - left + 1) \* heights[top]

                ans = max(ans, area)

            stack.append(i)

        return ans

设计动态规划算法的通用小技巧：

1. 每种动态规话方案都涉及网络
2. 单元格中的值通常是要优化的值。
3. 每个单元格都是一个子问题，应该考虑如何将问题分程子问题，这将有助于找出单元格的坐标轴。

最长公共子串（连续子序列）：if a[i] == b[j]: dp[i][j] = dp[i-1][j-1] + 1

Else: dp[i][j] = 0

最长公共子序列:if a[i] == b[j]:dp[i][j] = dp[i-1][j-1]+1

Else: dp[i][j] = max(dp[i-1][j], dp[i][j-1])

前缀和算法：

题目：如果一个由 '0' 和 '1' 组成的字符串，是以一些 '0'（可能没有 '0'）后面跟着一些 '1'（也可能没有 '1'）的形式组成的，那么该字符串是单调递增的。

我们给出一个由字符 '0' 和 '1' 组成的字符串 S，我们可以将任何 '0' 翻转为 '1' 或者将 '1' 翻转为 '0'。

返回使 S 单调递增的最小翻转次数。

示例 1：

输入："00110"

输出：1

解释：我们翻转最后一位得到 00111.

解：最终得到的满足条件的字符串的格式为[0...1...]，假设左边为x个0，右边为n-x个1，则相比于原数列，左边有若干个1，右边有若干个0需要反转。

        使用前缀和记录P[x] = P[0]+...+P[x-1]，则x的左边有p[x]个1，右边有(n-x) - (p[n]-p[x])个0，遍历P求出最小的遍历值。

n = len(S)

        p = [0] \* (n+1)

        for i in range(n):

            p[i+1] = p[i] + int(S[i])

        ans = min([p[x] + (n-x) - (p[n]-p[x]) for x in range(n+1)])

        return ans

'''

        动态规划算法。

        字符串的每个位置最终要么是0，要么是1，只有两个状态。

        设dp[i][0]和dp[i][1]分别是字符S[i]为0和为1所需的反转次数。

        if S[i] == '1':

            dp[i][0] = dp[i-1][0] + 1 #要变为0，将i转为0，多一次

            dp[i][1] = min(dp[i-1][0], dp[i-1][1]) #不管之前是0还是1，都不需要反转

        else:

            dp[i][0] = dp[i-1][0] #之前为0，不需要反转i

            dp[i][1] = min(dp[i-1][0]+1, dp[i-1][1]+1) #如果要让s[i]为1，需要把s[i]反转

        '''

        one, zero = 0, 0

        for s in S:

            if s == '1':

                one = min(zero, one)

                zero = zero + 1

            else:

                one = min(zero+1, one + 1)

        return min(one, zero)

Kadane算法：求最大字串和。对于一个包含负值的数字串array[1...n]，要找到他的一个子串array[i...j]（0<=i<=j<=n），使得在array的所有子串中，array[i...j]的和最大。

解法：从头到尾遍历目标数组，将数组分割为满足条件的子串，同时得到各子串的最大前缀和并比较，得到最大的答案。划分子串的方法是：如果当前子串和>0，则继续向右扩展，如果当前子串和<0，则将该子串划为一个子串，计算他的最大字串和（不计算使得子串和小于0的那个元素），继续遍历，下一个元素作为下一个子串的开头。

例如[-2,1,-3,4,-1,2,1,-5,4]划分为[-2],[1,-3],[4,-1,2,1,-5,4]。

代码：

Ans = -inf, sum = 0

for num in nums:

sum += num #计算当前前缀和

if sum > ans: #更新最终解

ans = sum

if sum < 0: #划分当前元素为子串结尾，从下一个元素开始找下一个子串

sum = 0

Python代码：

ans = cur = None

for x in A:

cur = x + max(cur, 0)

ans = max(ans, cur)

return ans