leetcode精选100

leetcode精选100题

1. 给定一个整数数组 `nums` 和一个目标值 `target`，请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。
2. 你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

```
class Solution:
    def twoSum1(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
        return [[i,j] for i in range(len(nums)) for j in range(len(nums)) if nums[i]+nums[j] == target and i != j][0]

    def twoSum(self, nums, target):
        assist = {}
        for i, num in enumerate(nums):
            other = target - num
            if other not in assist:
                assist[num] = i
            else:
                return [i, assist[other]]
```

```
给出两个 非空 的链表用来表示两个非负的整数。其中，它们各自的位数是按照 逆序 的方式存储的，并且它们的每个节点只能存储 一位 数字。
如果，我们将这两个数相加起来，则会返回一个新的链表来表示它们的和。
您可以假设除了数字 0 之外，这两个数都不会以 0 开头。
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        """
        分析：
        由于链表是按照逆序的方式存储的，所有最开始的就是各位，因此直接按照按位计算加法即可
        1. 需要计算进位
        2. 需要将最后长的那一个链表加进来
        3. 题目要求返回一个新的链表来表示他们的和，但是如果允许修改原链表的话，可以不申请新的内存
        """
        head = ListNode(0)
        cur = head
        carry = 0
        while l1 or l2:
            data1 = l1.val if l1 else 0
            data2 = l2.val if l2 else 0
            s = data1+data2+carry
            carry = s // 10
            s %= 10
            cur.next = ListNode(s)
            cur = cur.next
            if l1:
                l1 = l1.next
            if l2:
                l2 = l2.next

        if carry:
            cur.next = ListNode(1)
        return head.next
```

```
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。
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        """
        滑窗法:
        设置两个指针，start和end，每次end+=1，记录当前点访问的次数
        """
        res, start, end = 0, 0, 0
        count_dict ={}
        for c in s:
            end += 1
            count_dict[c] = count_dict.get(c,0)+1
            while count_dict[c] > 1:
                count_dict[s[start]] -= 1
                start += 1
            res = max(res, end-start)
        return res
```

```
1. 给定两个大小为 m 和 n 的有序数组 nums1 和 nums2。
2. 请你找出这两个有序数组的中位数，并且要求算法的时间复杂度为 O(log(m + n))。
3. 你可以假设 nums1 和 nums2 不会同时为空。
4. class Solution:
5.     def findMedianSortedArrays(self, nums1, nums2):
6.         """
7.         :type nums1: List[int]
8.         :type nums2: List[int]
9.         :rtype: float
10.        """
11.        nums = nums1+nums2
12.        nums.sort()
13.        if len(nums)%2 == 0:
14.            return (nums[(len(nums)-1)//2]+nums[(len(nums)-1)//2+1])/2.0
15.        else:
16.            return(nums[(len(nums)-1)//2])
```

```
给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。
class Solution:
    def __init__(self):
        self.lens = None
        self.startIndex = None
    def longestPalindrome(self, s: str) -> str:
        '''
        # 动态规划，状态转移法
        if not s or len(s) <=1:
            return s
        n = len(s)
        # 状态转移矩阵
        historyRecord = [[0] * n for _ in range(n)]

        self.lens = 1
        self.startIndex = 0

        # 初始化长度为1的回文字符串信息
        for i in range(n):
            historyRecord[i][i] = 1

        # 初始化长度为2的回文字符串信息
        for i in range(n-1):
            if s[i] == s[i+1]:
                historyRecord[i][i+1] = 1
                self.startIndex = i
                self.lens = 2
        # 从3开始遍历长度为k的回文字符串信息
        for k in range(3, n+1):
            i = 0
            while i < n-k+1:
                j = i+k-1
                if s[i] == s[j] and historyRecord[i+1][j-1] == 1:
                    historyRecord[i][j] = 1
                    self.startIndex = i
                    self.lens = k
                i += 1
        # print(self.startIndex, self.lens)
        return s[self.startIndex:self.startIndex+self.lens]
        '''
        '''
        # 中心扩展法
```

```
43.         if not s and len(s)<=1:
44.             return s
45.         self.lens = 1
46.         self.startIndex = 0
47.         for i in range(len(s)-1):
48.             # 当回文数中心为一个数时
49.             self.expandBothSide(s, i, i)
50.             # 当回文数中心为两个数时
51.             self.expandBothSide(s, i, i+1)
52.         return s[self.startIndex:self.startIndex+self.lens]
53.
54.     def expandBothSide(self, s, c1, c2):
55.         n = len(s)
56.         while c1 >= 0 and c2 < n and s[c1] == s[c2]:
57.             c1 -= 1
58.             c2 += 1
59.         tmpStartIndex = c1 + 1
60.         tmpLens = c2-c1-1
61.         if tmpLens > self.lens:
62.             self.lens = tmpLens
63.             self.startIndex = tmpStartIndex
64.     '''
65.         # manacher算法...
```

```
给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 '.' 和 '*' 的正则表达式匹配。
class Solution:
    def __init__(self):
        self.memo = dict()
    def isMatch(self, s: str, p: str) -> bool:
        """
        依次遍历s和p的每一个字符:
        1. 如何p[j]不为'*'和'.', 此时判断s[i] == p[j],相等i++, j++, 不相等返回False
        2. 如果p[j] == '.',直接i++, j++
        3. 如果p[j+1] == '*'
            a. 如果s[i] != s[j], j += 2,继续判断
            b. 如果s[i] == p[j] or p[j] == '.', 分为三种情况, 1,  *匹配0个, 1个和多个
        """
        # 这里引入备忘录，记录中间结果，减小计算量
        memo = dict()
        def dp(i, j):
            if (i, j) in memo: return memo[(i,j)]
            if j == len(p): return i == len(s)
            first_match = i < len(s) and p[j] in [s[i], '.']
            if j <= len(p)-2 and p[j+1] == '*':
                ans = dp(i, j+2) or (first_match and dp(i+1, j))
            else:
                ans = first_match and dp(i+1, j+1)
            memo[(i,j)] = ans
            return ans
        return dp(0,0)
```

```
11. 盛最多水的容器
—
class Solution:
    def maxArea(self, height: List[int]) -> int:
        # 双指针法，每次记录最大面积，指向短的那一条边向中间移动
        max_area = 0
        i = 0
        j = len(height) - 1
        while i < j:
            max_area = max_area if max_area > (j-i)*min(height[i], height[j]) else (j-i)*min(height[i], height[j])
            if height[i] < height[j]:
                i += 1
            else:
                j -= 1
        return max_area
```

```
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。
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        """
        1. 先进行排序，时间负责度为O（nlogn）
        2. 选定一个作为参照点，（不是中心点），左右各需一人，且加起来为0，如果小于0，左边++，大于0，右边--
        3. 边界条件优化：
            ①：整个数组同号则误解，nums[0] <= 0 and nums[-1]>=0
            ②：三元组中的左边数为正数，则无解，这里的参照点就是最左边点
            ③：
        """
        length = len(nums)
        resultList = []
        nums.sort()
        for i in range(0,length-2):
            if i>0 and nums[i] == nums[i-1]:
                continue
            if nums[i]>0:
                break
            j = i + 1
            k = length - 1
            while j < k:
                sum0 = nums[i] + nums[j] + nums[k]
                if sum0 == 0:
                    resultList.append([nums[i], nums[j], nums[k]])
                    while j < k and nums[j] == nums[j+1]:
                        j += 1
                    while j < k and nums[k]==nums[k-1]:
                        k -= 1
                    j +=1
                    k -=1

                elif sum0 < 0:
                    j +=1
                else:
                    k -=1

        return resultList
```

```
17. 电话号码的字母组合
—
class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        """
        # 非常直观的递归方法
        if not digits:
            return []
        num_char_dict = {'2':"abc",'3':"def",'4':"ghi",'5':"jkl",'6':"mno",'7':"pqrs",'8':"tuv",'9':"wxyz"}
        res = []
        length = len(digits)
        def find_char(i, tmp):
            if i == length:
                res.append(tmp)
                return
            for char in num_char_dict[digits[i]]:
                find_char(i+1, tmp+char)
        find_char(0, "")
        return res
        """
        # DFS
        if not digits:
            return []
        num_char_dict = {'2':"abc",'3':"def",'4':"ghi",'5':"jkl",'6':"mno",'7':"pqrs",'8':"tuv",'9':"wxyz"}
        res = ['']
        for i in digits:
            res = [k+j for k in res for j in num_char_dict[i]]
        return res
```

```
20. 有效的括号

class Solution:
    def isValid(self, s: str) -> bool:
        # 模拟入栈，出栈，最后栈为空即可
        pair_dict = {"(":")", "[":"]", "{":"}"}
        stack = []
        length = len(s)
        i = 0
        while length > i:
            if stack and pair_dict.get(stack[-1], 0) == s[i]:
                stack.pop()
            else:
                stack.append(s[i])
            i += 1
        if not stack:
            return True
        else:
            return False
```

```
21. 合并两个有序链表

class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        '''
        # 使用递归操作
        if not l1:
            return l2
        if not l2:
            return l1
        if l1.val < l2.val:
            l1.next = self.mergeTwoLists(l1.next, l2)
            return l1
        else:
            l2.next = self.mergeTwoLists(l1, l2.next)
            return l2
        '''
        # 使用迭代法
        prehead = ListNode(0)
        new_p = prehead
        while l1 and l2:
            if l1.val <= l2.val:
                new_p.next = l1
                l1 = l1.next
            else:
                new_p.next = l2
                l2 = l2.next
            new_p = new_p.next
        new_p.next = l1 if l1 else l2

        return prehead.next
```

```
给出 n 代表生成括号的对数，请你写出一个函数，使其能够生成所有可能的并且有效的括号组合。
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        ans = []
        def backtrack(S="", left =0 ,right = 0):
            if len(S) == 2*n:
                ans.append(S)
                return
            if left < n:
                backtrack(S+"(", left+1, right)
            if right < left:
                backtrack(S+")", left, right+1)
        backtrack()
        return ans
```

```
合并 k 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。
class Solution:
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        # 将K个链表两两结合，这里可以采用分治的思想，因此只需要执行logk次合并
        length = len(lists)
        interval = 1
        while interval < length:
            for i in range(0, length-interval, 2*interval):
                lists[i] = self.merge2Lists(lists[i], lists[i+interval])
            interval *= 2
        return lists[0] if length >0 else lists

    def merge2Lists(self, l1, l2):
        if not l1:
            return l2
        if not l2:
            return l1
        if l1.val < l2.val:
            l1.next = self.merge2Lists(l1.next, l2)
            return l1
        else:
            l2.next = self.merge2Lists(l1, l2.next)
            return l2
```

```
实现获取下一个排列的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列。
class Solution:
    def nextPermutation(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        #1. 从后面进行遍历，依次比较相邻数字大小，直到nums[i] > nums[i-1]
        #2. 将该数插入到后面有序数组的相应位置
        #3. 对后面数组进行翻转
        i = len(nums)-2
        while i >= 0 and nums[i] >= nums[i+1]:
            i -= 1
        if i >= 0:
            j = len(nums)-1
            while j >= 0 and nums[j] <= nums[i]:
                j -= 1
            nums[i], nums[j] = nums[j], nums[i]
        # 对后面的数组进行翻转
        start = i+1
        end = len(nums)-1
        while start < end:
            nums[start], nums[end] = nums[end], nums[start]
            start += 1
            end -= 1
        print(nums)
```

```
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。
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        '''
        采用动态规划方法
        1. 创建一个dp数组，dp[i]表示以第i个字符为结束符的有效括号的长度
        2. 对每两个字符检查一次。
            a. 如果s[i] = ')',s[i-1] = "(", 那么dp[i] = dp[i-2]+2
            b. 如果s[i] = ')',s[i-1] = ')',并且s[i-dp[i-1]-1],那么dp[i] = dp[i-1]+dp[i-dp[i-1]-2]+2
        '''
        if not s:
            return 0
        dp = [0 for i in range(len(s))]
        for i in range(1, len(s)):
            if s[i] == ")":
                if s[i-1] == "(":
                    dp[i] = dp[i-2]+2 if i >= 2 else 2
                elif (i - dp[i-1] - 1) >= 0 and s[i-dp[i-1]-1] == "(":
                    print(i)
                    dp[i] = (dp[i-1] + dp[i-dp[i-1]-2]+2) if (i-dp[i-1]) >= 2 else (dp[i-1]+2)
        print(dp)
        return max(dp)
```

```
33. 搜索旋转排序数组
—
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        """
        logN的题大部分都采用二分法的方式做
        1. 先找出旋转数组的最小值，然后再判断是在左边查找还是右边查找，这里采用二分法查找
        """
        if not nums:
            return -1
        n = len(nums)
        left = 0
        right = n - 1
        while left < right:
            mid = left + (right-left)//2
            if nums[mid] > nums[right]:
                left = mid+1
            else:
                right = mid
        min_index = left
        left = 0
        right = n - 1
        while left <= right:
            mid = (left+right)//2
            realmid = (mid + min_index)%n
            if nums[realmid] == target:
                return realmid
            elif nums[realmid] > target:
                right = mid-1
            else:
                left = mid+1
        return -1
```

```
34. 在排序数组中查找元素的第一个和最后一个位置
—
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        res = []
        if not nums:
            return [-1,-1]
        # 查找最左边的target
        low = 0
        high = len(nums)-1
        while low <= high:
            mid = (low+high)//2
            if nums[mid] >= target:
                high = mid-1
            else:
                low = mid+1
        if 0 <= low and low <= len(nums)-1 and nums[low] == target:
            res.append(low)
        else:
            res.append(-1)
        # 查找最右边的target
        low = 0
        high = len(nums)-1
        while low <= high:
            mid = (low+high)//2
            if nums[mid] <= target:
                low = mid +1
            else:
                high = mid -1
        if 0 <= high and high <= len(nums)-1 and nums[high] == target:
            res.append(high)
        else:
            res.append(-1)
        return res
```

```
1. 给定一个无重复元素的数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。
2. class Solution:
3.     def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
4.         # 该题为求排列模式的回溯算法，通常采用递归实现
5.         candidates.sort()
6.         n = len(candidates)
7.         res = []
8.         def backtrack(i, tmp_sum, tmp):
9.             if tmp_sum > target or i == n:
10.                return
11.            if tmp_sum == target:
12.                res.append(tmp)
13.                return
14.            for j in range(i, n):
15.                if tmp_sum +candidates[j] > target:
16.                    break
17.                backtrack(j, tmp_sum+candidates[j], tmp+[candidates[j]])
18.        backtrack(0,0,[])
19.        return res
```

```
给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。
class Solution:
    def trap(self, height: List[int]) -> int:
        """
        #对于每一块，找到该块左边和右边最大的块，用次大的高度减去当前高度即为存储水的高度
        #*:  提前通过遍历找到每一个块左边最大的块高度和右边对大块高度

        if not height:
            return 0
        ans = 0
        n = len(height)
        left_max = [0 for _ in range(n)]
        right_max = [0 for _ in range(n)]
        left_max[0] = height[0]
        right_max[-1] = height[-1]
        for i in range(1, n):
            left_max[i] = max(height[i], left_max[i-1])

        for i in range(n-2,-1, -1):
            right_max[i] = max(height[i], right_max[i+1])

        for i in range(1, n-1):
            ans += min(left_max[i], right_max[i]) - height[i]
        return ans
        """
        # 采用栈的做法
        if not height:
            return 0
        n = len(height)
        stack = []
        res = 0
        for i in range(n):
            while stack and height[stack[-1]] < height[i]:
                tmp = stack.pop()
                if not stack:
                    break
                res += (min(height[i], height[stack[-1]]) - height[tmp]) * (i - stack[-1]-1)
            stack.append(i)
        return res
```

```
给定一个没有重复数字的序列，返回其所有可能的全排列。
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        # 直接用回溯的方法
        if not nums:
            return []
        res = []
        n = len(nums)
        def backtrack(nums, tmp):
            if len(tmp) == n:
                res.append(tmp)
                return
            for i in range(len(nums)):
                backtrack(nums[:i]+nums[i+1:], tmp+[nums[i]])
        backtrack(nums, [])
        return res
    """
    # 使用库函数
    from itertools import permutations
    return list(itertools.permutations(nums))
    """
```

```
1.  给定一个 n × n 的二维矩阵表示一个图像。将图像顺时针旋转 90 度。
2.  class Solution:
3.      def rotate(self, matrix):
4.          """
5.          :type matrix: List[List[int]]
6.          :rtype: void Do not return anything, modify matrix in-place instead.
7.          """
8.          # 先转置，再反向
9.          l = len(matrix)
10.         for i in range(l):
11.             for j in range(i+1, l):
12.                 matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
13.
14.         for i in range(l):
15.             matrix[i] = matrix[i][::-1]
```

```
给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。
from collections import defaultdict
class Solution(object):
    def groupAnagrams(self, strs):
        """
        :type strs: List[str]
        :rtype: List[List[str]]
        """
        # 将每个字符串排序后存入dict中
        res = defaultdict(list)
        for s in strs:
            res[tuple(sorted(s))].append(s)
        return res.values()
```

```
给定一个整数数组 nums ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。
class Solution:
    def maxSubArray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        max = -float('Inf')
        sum = 0
        for i in nums:
            sum = sum+i
            if sum > max:
                max = sum
            if sum <0:
                sum = 0
        return max
```

```
给定一个非负整数数组，你最初位于数组的第一个位置。数组中的每个元素代表你在该位置可以跳跃的最大长度。判断你是否能够到达最后一个位置。
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        '''
        超时~!
        # 排列组合的问题，溯源法
        if not nums:
            return False
        n = len(nums)
        already_point = []
        def backtrace(i):
            if i == n-1:
                return True
            if i > n-1 or nums[i] == 0:
                return False
            flag = False
            for j in range(1, nums[i]+1):
                if i+j > n-1:
                    break
                if i+j in already_point:
                    continue
                already_point.append(i+j)
                flag = flag or backtrace(i+j)
            return flag
        return backtrace(0)
        '''
        # 贪心法解题
        lastPos = len(nums)-1
        i = len(nums)-1
        while i >= 0:
            if i+nums[i] >= lastPos:
                lastPos = i
            i -= 1
        return lastPos == 0
```

```
1.  给出一个区间的集合，请合并所有重叠的区间。
2.  class Solution:
3.      def merge(self, intervals: List[List[int]]) -> List[List[int]]:
4.          intervals = sorted(intervals)
5.          res = []
6.          n = len(intervals)
7.          i = 0
8.          while i < n:
9.              left = intervals[i][0]
10.             right = intervals[i][1]
11.             while i < n-1 and intervals[i+1][0] <= right:
12.                 i += 1
13.                 right = max(right, intervals[i][1])
14.             res.append([left, right])
15.             i+= 1
16.         return res
```

```
一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为“Start” ）。机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。问总共有多少条不同的路径?
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        '''
        # 经典的回溯法
        self.res = 0
        dp = [[-1 for _ in range(n+1)] for _ in range(m+1)]

        def backtrack(i, j):
            if i == m and j == n:
                return 1
            if dp[i][j] != -1:
                return dp[i][j]
            ans = 0
            if i+1 <= m:

                ans += backtrack(i+1, j)

            if j+1 <= n:
                ans += backtrack(i, j+1)
            dp[i][j] = ans
            return ans

        return backtrack(1,1)
        '''
        # 使用动态规划
        dp = [[-1 for _ in range(n)] for _ in range(m)]


        # 初始化数据
        for i in range(m):
            dp[i][0] = 1

        for j in range(n):
            dp[0][j] = 1

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

```
给定一个包含非负整数的 m x n 网格，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。
class Solution:
    def minPathSum(self, grid: List[List[int]]) -> int:
        '''
        # 如果将路径和的值卸载grid中，则可以不申请grid数组
        if not grid:
            return 0
        # 使用动态规划
        height = len(grid)
        width = len(grid[0])
        dp = [[-1 for _ in range(width)] for _ in range(height)]
        # 初始化第一行
        dp[0][0] = grid[0][0]
        for i in range(1, width):
            dp[0][i] = dp[0][i-1]+grid[0][i]
        # 初始化第一列
        for i in range(1, height):
            dp[i][0] = dp[i-1][0]+grid[i][0]
        for i in range(1, height):
            for j in range(1, width):
                dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]
        return dp[-1][-1]
        '''
        # 由于只能是往右或者往下移动，因此可以只用一维数组来存储状态
        if not grid:
            return 0
        height = len(grid)
        width = len(grid[0])
        dp = [-1 for _ in range(width)]
        for i in range(height - 1, -1, -1):
            for j in range(width - 1, -1, -1):
                if j == width - 1 and i != height - 1:
                    dp[j] = dp[j] + grid[i][j]
                elif i == height - 1 and j != width - 1:
                    dp[j] = dp[j+1] + grid[i][j]
                elif i != height - 1 and j != width - 1:
                    dp[j] = min(dp[j + 1], dp[j]) + grid[i][j]
                else:
                    dp[j] = grid[i][j]
        print(dp[0])
        return dp[0]
```

```
70. 爬楼梯

class Solution:
    def climbStairs(self, n: int) -> int:
        # f(1) = 1 f(2)=2 f(3) = f(1)+f(2) ----> f(n) = f(n-1)+f(n-2)
        if n <= 0:
            return 0
        if n == 1:
            return 1
        if n == 2:
            return 2
        f = [0 for _ in range(n)]
        f[0] = 1
        f[1] = 2
        for i in range(2, n):
            f[i] = f[i-2]+f[i-1]
        return f[-1]
```

```
72. 编辑距离

class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        # 动态规划，ed二维数组，ed[i][j]表示word1长度为i的子串到word2长度为j的子串之间的编辑距离
        '''
        1. 建立一个2为数组D，表示将s1的i子串变为s2的j子串所需要的编辑距离
            a. i = 0时，D[i][j] = j
            b. j = 0 时， D[i][j] = i
        2. 增：如果计算出了D(i, j-1)的值，那么D(i, j) 等于D(i,j-1)+1
        3. 删：如果计算出了D(i-1, j)的值，那么D (i, j) 等于D (i-1, j) +1
        4. 改：如果计算出了D (i-1, j-1) 的值，如果s1[i]==s2[j],则D (i,j)  = D(i-1, j-1),否则，D (i,j)  = D(i-1, j-1)+1：将s2[j]改为s1[i]
        '''
        if not word1:
            return len(word2)
        if not word2:
            return len(word1)

        len1 = len(word1)
        len2 = len(word2)
        # 这里长度必须加1，因为可以从“”开始变，这里的长度为0，一共n+1个长度
        et = [[-1 for _ in range(len2+1)] for _ in range(len1+1)]
        # 初始化部分值
        for i in range(len1+1):
            et[i][0] = i
        for j in range(len2+1):
            et[0][j] = j

        for i in range(1, len1+1):
            for j in range(1, len2+1):
                # 这里之所以要减1，是由于i和j为1的时候对应第一个字符，即0，所以要减1
                if word1[i-1] == word2[j-1]:
                    et[i][j] = min(et[i-1][j]+1, et[i][j-1]+1, et[i-1][j-1])
                else:
                    et[i][j] = min(et[i-1][j]+1, et[i][j-1]+1, et[i-1][j-1]+1)
        return et[-1][-1]
```

```
1. 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。
2. class Solution:
3.     def sortColors(self, nums: List[int]) -> None:
4.         """
5.         Do not return anything, modify nums in-place instead.
6.         """
7.         # 设置三个指针pre, cur, end, pre始终指向 0, end指向2, cur指向当前值
8.         # 初始化: pre、cur为0,  end为n-1
9.         # 1. 当cur指向0，则cur与pre的值交换，两者都右移;  2. 当end指向2时,  则cur与end的值交换，end向左移; 3. 知道cur==end返回。
10.        length = len(nums)
11.        pre = cur = 0
12.        end = length - 1
13.        while cur <= end:
14.            if nums[cur] == 0:
15.                nums[pre], nums[cur] = nums[cur], nums[pre]
16.                pre += 1
17.                cur += 1
18.            elif nums[cur] == 2:
19.                nums[end], nums[cur] = nums[cur], nums[end]
20.                end -= 1
21.            else:
22.                cur += 1
```

```
给你一个字符串 S、一个字符串 T，请在字符串 S 里面找出：包含 T 所有字母的最小子串。
from collections import Counter
class Solution:
    def minWindow(self, s: str, t: str) -> str:
        # 使用滑窗法
        dict_t = Counter(t)
        required = len(dict_t)
        l = r = 0
        formed = 0
        window_counts = {}

        ans = (float("Inf"), None, None)

        while r < len(s):
            char = s[r]
            window_counts[char] = window_counts.get(char, 0)+1
            # 这里子串要包含T中所有的字母，且重复的字符也算进去
            if char in dict_t and window_counts[char] == dict_t[char]:
                formed += 1

            # 窗口满足条件，开始收缩窗口
            while l <= r and formed == required:
                char = s[l]
                if r-l+1 < ans[0]:
                    ans = (r-l+1, l, r)

                window_counts[char] -= 1
                if char in dict_t and window_counts[char] < dict_t[char]:
                    formed -= 1
                l += 1
            r += 1
        return "" if ans[0] == float("Inf") else s[ans[1]:ans[2]+1]
```

```
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）。
from itertools import combinations
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        '''
        # 全排列问题
        res = []
        def permute(nums, tmp):
            res.append(tmp)
            if not nums:
                return
            for j in range(len(nums)):
                permute(nums[j+1:], tmp+[nums[j]])
        permute(nums, [])
        return res
        '''
        # 使用库函数
        res = []
        for i in range(len(nums)+1):
            for tmp in itertools.combinations(nums, i):
                res.append(tmp)
        return res
```

```
给定一个二维网格和一个单词，找出该单词是否存在于网格中。
class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        # dfs+回溯算法
        row = len(board)
        col = len(board[0])

        def backtrace(i, j, k, visited):
            if k == len(word):
                return True
            # 四个方向均可查看
            for x, y in [(-1, 0), (1, 0), (0, 1), (0, -1)]:
                tmp_i = i+x
                tmp_j = j+y
                if 0 <= tmp_i < row and 0 <= tmp_j < col and visited[tmp_i][tmp_j] != 1 and board[tmp_i][tmp_j] == word[k]:
                    visited[tmp_i][tmp_j] = 1
                    if backtrace(tmp_i, tmp_j, k+1, visited):
                        return True
                    visited[tmp_i][tmp_j] = 0
            return False

        visited = [[0 for _ in range(col)] for _ in range(row)]
        for i in range(row):
            for j in range(col):
                if board[i][j] == word[0]:
                    visited[i][j] = 1
                    if backtrace(i, j, 1, visited):
                        return True
                    visited[i][j] = 0
        return False
```

```
给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。求在该柱状图中，能够勾勒出来的矩形的最大面积。
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        '''
        # 该方法超时!
        # 使用分治法，先找出最低的柱子，则：1. 确定了柱子的高度，矩形宽为整个的宽度；2. 在最矮柱子的左边；3. 在右边
        return self.calculateArea(heights, 0, len(heights)-1)

    def calculateArea(self, heights, start, end):
        if start > end:
            return 0
        minindex = start
        for i in range(start, end+1):
            if heights[i] < heights[minindex]:
                minindex = i
        return max([heights[minindex]*(end-start+1), self.calculateArea(heights, start, minindex-1), self.calculateArea(heights, minindex+1, end)])
    '''
        # 使用栈的方法
        stack = []
        heights = [0] + heights + [0]
        res = 0
        for i in range(len(heights)):
            while stack and heights[stack[-1]] > heights[i]:
                tmp = stack.pop()
                res = max(res, heights[tmp]*(i - stack[-1]-1))
            stack.append(i)
        return res
```

```
给定一个仅包含 0 和 1 的二维二进制矩阵，找出只包含 1 的最大矩形，并返回其面积。
class Solution(object):
    def maximalRectangle(self, matrix):
        """
        :type matrix: List[List[str]]
        :rtype: int
        """
        maxarea = 0
        dp = [[0] * len(matrix[0]) for _ in range(len(matrix))]
        for i in range(len(matrix)):
            for j in range(len(matrix[0])):
                if matrix[i][j] == '0':
                    continue
                # 计算当前点作为矩形右下角点的宽度
                width = dp[i][j] = dp[i][j-1] + 1 if j else 1

                for k in range(i, -1, -1):
                    # 求出矩形的宽度
                    width = min(width, dp[k][j])
                    maxarea = max(maxarea, width*(i-k+1))

        return maxarea
```

```
给定一个二叉树，返回它的中序 遍历。
class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]:
        '''
        # 先使用递归的方法完成
        res = []
        def inorder_traversal(root):
            if not root:
                return
            inorder_traversal(root.left)
            res.append(root.val)
            inorder_traversal(root.right)

        inorder_traversal(root)
        return res
        '''
        # 使用栈的方法
        res = []
        stack = []
        cur = root
        while cur or stack:
            while cur:
                stack.append(cur)
                cur = cur.left
            cur = stack.pop()
            res.append(cur.val)
            cur = cur.right
        return res
```

```
给定一个整数 n，求以 1 ... n 为节点组成的二叉搜索树有多少种?
class Solution:
    def numTrees(self, n: int) -> int:
        # 二叉搜索树又称为二叉排序树，其左子树上的所有点均小于它根节点的值，右子树上的所有点据大于根节点的值
        # 假设n个节点存在二叉排序树为G(n),令f(i)为根节点为i的二叉排序树的个数，则G(n) = f(1)+f(2)+...+f(n)
        # f(i) = G(i-1)*G(n-i)
        # G(n) = G(0)*G(n-1) + G(1)*G(n-2)+ ... + G(n-1)*G(0): **又名卡特兰数**

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

        for i in range(2, n+1):
            for j in range(1, i+1):
                dp[i] += dp[j-1]*dp[i-j]
        return dp[n]
```

```
给定一个二叉树，判断其是否是一个有效的二叉搜索树。
class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
        # 二叉搜索树的中序遍历是否是一个升序数组
        self.pre = -float("inf")
        def judge_bst(root):
            if not root:
                return True
            l = judge_bst(root.left)
            if root.val <= self.pre:
                return False
            self.pre = root.val
            r = judge_bst(root.right)
            return l and r
        return judge_bst(root)
```

```
1.  给定一个二叉树，检查它是否是镜像对称的。
2.  class Solution:
3.      def isSymmetric(self, root: TreeNode) -> bool:
4.          # 1. 分别按照左中右和右中左的顺序，看遍历是否相等
5.          # 2. 将其根节点分为两份，形成两个树，判断两个树是否镜像对称
6.          def isMirror(root1, root2):
7.              if not root1 and not root2:
8.                  return True
9.              if not root1 or not root2:
10.                 return False
11.             return root1.val == root2.val and isMirror(root1.left, root2.right) and
    isMirror(root1.right, root2.left)
12. 
13.         return isMirror(root,root)
```

```
给定一个二叉树，返回其按层次遍历的节点值。 （即逐层地，从左到右访问所有节点）。
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        '''
        # 利用队列实现层次遍历
        if not root:
            return []
        res = []
        stack = []
        tmp = [root]
        stack.append(tmp)
        while stack:
            layer = stack.pop(0)
            tmp = []
            res_l = []
            while layer:
                node = layer.pop(0)
                res_l.append(node.val)
                if node.left:
                    tmp.append(node.left)
                if node.right:
                    tmp.append(node.right)
            if tmp:
                stack.append(tmp)
            if res_l:
                res.append(res_l)
        return res
        '''
        # 利用递归的方法
        levels = []
        if not root:
            return []
        def helper(root, level):
            if len(levels) == level:
                levels.append([])
            levels[level].append(root.val)
            if root.left:
                helper(root.left, level+1)
            if root.right:
                helper(root.right, level+1)
        helper(root, 0)
```

```
        return levels
```

```
104. 二叉树的最大深度

class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        self.max_depth = 0
        def pre_traversal(root, d):
            if not root:
                self.max_depth = max(self.max_depth, d)
                return
            pre_traversal(root.left, d+1)
            pre_traversal(root.right, d+1)

        pre_traversal(root, 0)
        return self.max_depth
```

```
1. 根据一棵树的前序遍历与中序遍历构造二叉树。
2. class Solution:
3.     def buildTree(self, preorder: List[int], inorder: List[int]) -> TreeNode:
4.         if len(preorder) > 0:
5.             root = TreeNode(preorder[0])
6.             root_id = inorder.index(preorder[0])
7.             root.left = self.buildTree(preorder[1:1+root_id], inorder[:root_id])
8.             root.right = self.buildTree(preorder[1+root_id:], inorder[root_id+1:])
9.             return root
```

```
给定一个二叉树，原地将它展开为链表。
class Solution:
    def __init__(self):
        self.last = None
    def flatten(self, root: TreeNode) -> None:
        """
        Do not return anything, modify root in-place instead.
        """
        if not root:
            return
        if self.last:
            self.last.left = None
            self.last.right = root
        self.last = root
        # 复制当前right的节点，防止在left的分支将本身的right修改所造成无法访问的问题
        copy_right = root.right
        self.flatten(root.left)
        self.flatten(copy_right)
```

```
给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        min_price = float("inf")
        max_profit = 0
        for i in range(len(prices)):
            if prices[i] < min_price:
                min_price = prices[i]
            elif prices[i] - min_price > max_profit:
                max_profit = prices[i] - min_price
        return max_profit
```

```
给定一个未排序的整数数组，找出最长连续序列的长度。
class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        # 这里使用hash表空间换时间
        res = 0
        num_set = set(nums)
        for num in nums:
            # 从序列中最小的数开始判断
            if num-1 not in num_set:
                cur_len = 1
                cur_num = num
                while cur_num +1 in num_set:
                    cur_len += 1
                    cur_num += 1
                res = max(res, cur_len)
        return res
```

```
1.  给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。
2.  from functools import reduce
3.  class Solution:
4.      def singleNumber(self, nums):
5.          """
6.          :type nums: List[int]
7.          :rtype: int
8.          """
9.          # 只有一个数字不同，那么直接采用异或运算，得到的值即为落单的数字
10.         # 使用高级特性reduce
11.         return reduce(lambda x, y: x ^ y, nums)
```

```
给定一个非空字符串 s 和一个包含非空单词列表的字典 wordDict，判定 s 是否可以被空格拆分为一个或多个在字典中出现的单词。
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        '''
        # 带有记忆的回溯法，也超时了
        if not s:
            return False
        memo = [None for _ in range(len(s))]
        def traceback(s, start):
            if not s:
                return True

            if memo[start]:
                return memo[start]

            for i in range(1, len(s)+1):
                if s[:i] in wordDict:
                    if traceback(s[i:], i):
                        memo[start] = True
                        return True
            memo[start] = False
            return False
        return traceback(s, 0)
        '''
        # maxlen记录字典中最长字符串
        maxlen = 0
        for word in wordDict:
            if len(word) > maxlen:
                maxlen = len(word)
        res = [0]*len(s)
        for i in range(len(s)):
            p = i
            while (p>=0 and i-p<=maxlen):
                # 两个条件
                if (res[p] == 1 and s[p+1:i+1] in wordDict) or (p == 0 and s[p:i+1] in wordDict):
                    res[i] = 1
                    break
                p -= 1
        return res[-1] == 1
```

```
给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。
class Solution(object):
    def detectCycle(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        # 先判断是否有环，如果有找出相遇点meet_node，然后再创建两个指针，p位于头结点，q位于相遇点，然后同时移动，直到相遇，即是入口点
        if not head or not head.next:
            return None
        slow = fast = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                break
        if slow == fast:
            meet_node = slow
        else:
            return None

        # 接下来寻找入口点
        pre = head
        while pre != meet_node:
            pre = pre.next
            meet_node = meet_node.next
        return pre
```

```
LRU缓存机制

class ListNode:
    def __init__(self, key=None, value = None):
        self.key = key
        self.value = value
        self.pre = None
        self.next = None

class LRUCache:

    def __init__(self, capacity: int):
        self.capacity = capacity

        self.hashmap = {}
        # 新建两个节点head和tail
        self.head = ListNode()
        self.tail = ListNode()

        # 初始化链表为head<->tail
        self.head.next = self.tail
        self.tail.pre = self.head

    def move_node_to_tail(self, key):
        # 由于将中间节点移动到末尾比较麻烦，因此写一个函数单独处理
        node = self.hashmap[key]
        node.pre.next = node.next
        node.next.pre = node.pre
        # 上面两步相当于将node的前后连接起来了， 下面将node插入到尾结点
        node.pre = self.tail.pre
        node.next = self.tail
        self.tail.pre.next = node
        self.tail.pre = node

    def get(self, key: int) -> int:
        if key in self.hashmap:
            self.move_node_to_tail(key)
        res = self.hashmap.get(key, -1)
        if res == -1:
            return -1
        else:
            return res.value

```

```
    def put(self, key: int, value: int) -> None:
        if key in self.hashmap:
            # 更新值
            self.hashmap[key].value = value
            self.move_node_to_tail(key)
        else:
            if len(self.hashmap) == self.capacity:
                # 超过容量，删除头结点最近的节点
                self.hashmap.pop(self.head.next.key)
                self.head.next = self.head.next.next
                self.head.next.pre = self.head
            # 直接添加到尾结点
            new = ListNode(key, value)
            self.hashmap[key] = new
            new.pre = self.tail.pre
            new.next = self.tail
            self.tail.pre.next = new
            self.tail.pre = new

# 写入数据是一个key-value的形式，因此利用字典存储数据。然后还需要将该结构放入类似与队列的数据结构中，可以实现头节点删除，尾结点加入，当访问旧节点，将旧节点移动到尾结点
# 采用hash表和双向链表实现

'''
# 采用python库函数OrderedDict实现
class LRUCache(object):

    def __init__(self, capacity):
        self.od, self.cap = collections.OrderedDict(), capacity

    def get(self, key):
        if key not in self.od: return -1
        self.od.move_to_end(key)
        return self.od[key]

    def put(self, key, value):
        if key in self.od: del self.od[key]
        elif len(self.od) == self.cap: self.od.popitem(False)
        self.od[key] = value
'''
```

```
在 O(n log n) 时间复杂度和常数级空间复杂度下，对链表进行排序。
class Solution(object):
    def sortList(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        # 使用归并排序
        if not head or not head.next:
            return head
        # 找到中点的位置
        slow = fast = head
        while fast.next and fast.next.next:
            slow = slow.next
            fast = fast.next.next
        mid = slow
        l = head
        r = mid.next
        mid.next = None
        return self.merge(self.sortList(l), self.sortList(r))

    def merge(self, p, q):
        # 合并两个子链表
        tmp = ListNode(0)
        h = tmp
        while p and q:
            if p.val < q.val:
                h.next = p
                p = p.next
            else:
                h.next = q
                q = q.next
            h = h.next
        if p:
            h.next = p
        if q:
            h.next = q
        return tmp.next
```

```
给定一个整数数组 nums ，找出一个序列中乘积最大的连续子序列（该序列至少包含一个数）。
class Solution(object):
    def maxProduct(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        res_max = -float("inf")
        imax = 1
        imin = 1
        for i in range(len(nums)):
            if nums[i] <0:
                imax, imin = imin, imax
            imax = max(imax*nums[i], nums[i])
            imin = min(imin*nums[i], nums[i])

            res_max = max(res_max, imax)

        return res_max
```

```
155. 最小栈

class MinStack(object):

    def __init__(self):
        """
        initialize your data structure here.
        """
        self.stack = []
        self.min_stack = []


    def push(self, x):
        """
        :type x: int
        :rtype: None
        """
        self.stack.append(x)
        if not self.min_stack:
            self.min_stack.append(x)
        elif self.min_stack[-1] >= x:
                self.min_stack.append(x)

    def pop(self):
        """
        :rtype: None
        """
        pop_num = self.stack.pop()
        if pop_num == self.min_stack[-1]:
            self.min_stack.pop()


    def top(self):
        """
        :rtype: int
        """
        return self.stack[-1]


    def getMin(self):
        """
        :rtype: int
        """
        return self.min_stack[-1]
```

```
45.
46.  # 用另一个辅助栈存储最小的元素
47.  '''
48.  1. push:当min_stack为None的时候，直接将数据push进去；当min_stack不为空时，比较min_stack的栈顶元素，如果当前数据小于栈顶元素则push，否则跳过
49.  2. pop:如果pop元素与min_stack的栈顶元素相等则同时pop，否则跳过
50.  '''
```

```
给定一个大小为 n 的数组，找到其中的众数。众数是指在数组中出现次数大于 ⌊ n/2 ⌋ 的元素。
class Solution(object):
    def majorityElement(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        '''
        # 摩尔投票
        res = nums[0]
        count = 1
        for i in range(1, len(nums)):
            if nums[i] == res:
                count += 1
            else:
                count -= 1
                if count == 0:
                    res = nums[i+1]
        return res
        '''
        unique_nums = set(nums)
        for num in unique_nums:
            if nums.count(num) > len(nums)//2:
                return num
```

```
1.  你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。给定一个代表每个房屋存放金额的非负整数数组，计算你在不触动警报装置的情况下，能够偷窃到的最高金额。
2.  class Solution(object):
3.      def rob(self, nums):
4.          """
5.          :type nums: List[int]
6.          :rtype: int
7.          """
8.          # 动态规划f(i) = max(f(i-2), f(i-3)) + nums[i], 最终res = max(f(n-1), f(n-2))
9.          n = len(nums)
10.         if n == 0:
11.             return 0
12.         if n == 1:
13.             return nums[-1]
14.         if n == 2:
15.             return max(nums)
16.         dp = [0 for _ in range(n)]
17.         dp[0] = nums[0]
18.         dp[1] = nums[1]
19.         dp[2] = nums[0] + nums[2]
20.         for i in range(3, n):
21.             dp[i] = max(dp[i-2], dp[i-3]) + nums[i]
22.         return max(dp[n-1], dp[n-2])
```

```
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。
class Solution(object):
    def numIslands(self, grid):
        """
        :type grid: List[List[str]]
        :rtype: int
        """
        # 创建一个辅助数组，用于记录是否已经访问过；从左往右，从上到下一次遍历，当该点为1且没有访问过，则开始向右边和下面递归遍历1，如果为1，在辅助数组中将其标记为1.
        # 这里可以不申请辅助数组，直接将1改为0，但是gird本身会被修改
        self.auxiliary = [[0 for _ in range(len(grid[0]))] for _ in range(len(grid))]
        res = 0
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if self.auxiliary[i][j] == 1:
                    continue
                elif grid[i][j] == '1':
                    print(i, j)
                    self.find_land(i,j, grid)
                    res += 1
        return res

    def find_land(self, i, j, grid):
        if grid[i][j] == '1':
            self.auxiliary[i][j] = 1
            if i+1 < len(grid) and self.auxiliary[i+1][j] == 0:
                self.find_land(i+1, j, grid)
            if j+1 < len(grid[0]) and self.auxiliary[i][j+1] == 0:
                self.find_land(i, j+1, grid)
            if j-1 > -1 and self.auxiliary[i][j-1] == 0:
                self.find_land(i, j-1, grid)
            if i-1 > -1 and self.auxiliary[i-1][j] == 0:
                self.find_land(i-1, j, grid)
```

```
1. 反转一个单链表。
2. class Solution(object):
3.     def reverseList(self, head):
4.         # 1. 使用递归的方法
5.         self.pointer = ListNode(None)
6.         res = self.pointer
7.         
8.         def reverse(root):
9.             if not root:
10.                 return
11.             reverse(root.next)
12.             self.pointer.next = root
13.             self.pointer = self.pointer.next
14.         reverse(head)
15.         self.pointer.next = None
16.         return res.next
17.         '''
18.         # 2. 使用迭代的方法
19.         if not head or not head.next:
20.             return head
21.         pre = head
22.         cur = head.next
23.         next_p = head.next.next
24.         while next_p:
25.             cur.next = pre
26.             pre = cur
27.             cur = next_p
28.             next_p = next_p.next
29.         cur.next = pre
30.         head.next = None
31.         return cur
```

1. 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?

```
2.  class Solution(object):
3.      def canFinish(self, numCourses, prerequisites):
4.          """
5.          :type numCourses: int
6.          :type prerequisites: List[List[int]]
7.          :rtype: bool
8.          """
9.          # 采用深度遍历
10.         clen = len(prerequisites)
11.         if clen == 0:
12.             return True
13.         # 三个状态，0：未访问， 1：已访问， 2：。。。
14.         visited = [0 for _ in range(numCourses)]
15.
16.         # 得到每门课的所有前继课程
17.         inverse_adj = [set() for _ in range(numCourses)]
18.         for second, first in prerequisites:
19.             inverse_adj[second].add(first)
20.
21.         for i in range(numCourses):
22.             if self.dfs(i, inverse_adj, visited):
23.                 return False
24.         return True
25.
26.     def dfs(self, i, inverse_adj, visited):
27.         # 遇到环
28.         if visited[i] == 2:
29.             return True
30.         if visited[i] == 1:
31.             return False
32.         # 如果后面递归发现visited[2] == 2，所以前继回来了，形成了环
33.         visited[i] = 2
34.         for precursor in inverse_adj[i]:
35.             if self.dfs(precursor, inverse_adj, visited):
36.                 return True
37.
38.         # 运行到此，说明当前没有找到环
39.         visited[i] = 1
40.         return False
```

```
实现一个 Trie (前缀树)，包含 insert, search, 和 startsWith 这三个操作。
# defaultdict有默认值，默认值类型需要自行传递
from collections import defaultdict

class TrieNode:
    def __init__(self):
        self.children = defaultdict(TrieNode)
        self.is_word = False

class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        """
        Inserts a word into the trie.
        """
        cur = self.root
        for letter in word:
            cur = cur.children[letter]
        cur.is_word = True


    def search(self, word: str) -> bool:
        """
        Returns if the word is in the trie.
        """
        cur = self.root
        for letter in word:
            cur = cur.children.get(letter)
            if not cur:
                return False
        return cur.is_word

    def startsWith(self, prefix: str) -> bool:
        """
        Returns if there is any word in the trie that starts with the given prefix.
        """
        cur = self.root
```

```
        for letter in prefix:
            cur = cur.children.get(letter)
            if not cur:
                return False
        return True
```

```
在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。
class Solution(object):
    def findKthLargest(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: int
        """
        if k > len(nums) or k <1:
            return None
        # 既然只需要找到第k大的元素，那么只需要排序到第k个数即可，采用类快排的方法
        return self.findSmallK(nums, 0, len(nums)-1, k)

    def findSmallK(self, arr, low, high, k):
        i = low
        j = high
        # 选择一个中间点
        splitElem = arr[i]

        while i < j:
            while i< j and arr[j] <= splitElem:
                j -= 1
            if i < j:
                arr[i] = arr[j]
                i += 1

            while i < j and arr[i] >= splitElem:
                i += 1
            if i < j:
                arr[j] = arr[i]
                j -= 1
        arr[i] = splitElem

        subArrayIndex = i-low
        if subArrayIndex == k - 1:
            return arr[i]
        elif subArrayIndex > k-1:
            # 第k小的数依然在左边
            return self.findSmallK(arr, low, i-1, k)
        else:
            # 在右边
```

```
42.             return self.findSmallK(arr, i+1, high, k-(i-low+1))
```

```
在一个由 0 和 1 组成的二维矩阵内，找到只包含 1 的最大正方形，并返回其面积。
class Solution(object):
    def maximalSquare(self, matrix):
        '''
        # 该方法超时
        maxarea = 0
        # 这道题和找到最大面积的矩形差不多，主要区别在于计算面积的时候要比较高和宽大小，取小的平方即为面积
        dp = [[0 for _ in range(len(matrix[0]))] for _ in range(len(matrix))]

        for i in range(len(matrix)):
            for j in range(len(matrix[0])):
                if matrix[i][j] == '0':
                    continue
                width = dp[i][j] = dp[i][j-1] + 1 if j else 1

                for k in range(i, -1, -1):
                    width = min(width, dp[k][j])
                    maxarea = max(maxarea, min(width, i-k+1)**2)
        return maxarea
        '''
        # 由于是计算最大的正方形，那么dp动态矩阵只需要记录当前点作为正方向右下角点时，该正方形的边长
        # if matrix[i][j] = 1: dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])
        if not matrix:
            return 0
        maxarea = 0
        dp = [[0 for _ in range(len(matrix[0]))] for _ in range(len(matrix))]
        # 初始化
        for i in range(len(matrix)):
            dp[i][0] = int(matrix[i][0])
            if dp[i][0] == 1:
                maxarea = 1
        for i in range(len(matrix[0])):
            print(i)
            dp[0][i] = int(matrix[0][i])
            if dp[0][i] == 1:
                maxarea = 1

        for i in range(1, len(matrix)):
            for j in range(1, len(matrix[0])):
                if matrix[i][j] == '0':
                    continue
```

```
                    dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])+1
                    maxarea = max(maxarea, dp[i][j]**2)
            print(dp)
            return maxarea
```

翻转一棵二叉树。

```
class Solution(object):
    def invertTree(self, root):
        """
        :type root: TreeNode
        :rtype: TreeNode
        """
        if not root:
            return root
        root.left, root.right = root.right, root.left
        self.invertTree(root.left)
        self.invertTree(root.right)
        return root
```

```
请判断一个链表是否为回文链表。
class Solution(object):
    def isPalindrome(self, head):
        """
        :type head: ListNode
        :rtype: bool
        """
        '''
        # 该题用递归会超出递归栈
        self.sequence = head
        def inverse(root):
            if not root:
                return True
            flag = inverse(root.next)
            if root.val != self.sequence.val:
                return False
            else:
                self.sequence = self.sequence.next
                return flag and True
        return inverse(head)

        '''
        # 该题用递归会超出递归栈，但是我们只需要递归到一半即可
        # 找出中点
        if not head or not head.next:
            return True
        slow = head
        fast = head
        while fast and fast.next and fast.next.next:
            slow = slow.next
            fast = fast.next.next
        self.middle= slow

        self.sequence = head
        def inverse(root):
            if not root:
                return True
            flag = inverse(root.next)
            if root.val != self.sequence.val:
                return False
            else:
```

```
                self.sequence = self.sequence.next
                return flag and True
        return inverse(slow.next)
        # 除此之外，还可以使用栈，访问两次链表，但是这里申请了新的内存
```

```
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。
class Solution(object):
    def lowestCommonAncestor(self, root, p, q):
        """
        :type root: TreeNode
        :type p: TreeNode
        :type q: TreeNode
        :rtype: TreeNode
        """
        # 采用后续遍历的方法
        if not root or root == p or root == q:
            return root
        l_node = self.lowestCommonAncestor(root.left, p, q)
        r_node = self.lowestCommonAncestor(root.right, p, q)
        # 如果左子树上没有指定的两个节点，那么
        if not l_node:
            return r_node
        elif not r_node:
            return l_node
        else:
            return root
```

```
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 k 内的数字。滑动窗口每次只向右移动一位。返回滑动窗口最大值。
from collections import deque
class Solution(object):
    def maxSlidingWindow(self, nums, k):
        # 使用双向链表来存储最大值，需要注意以下问题
        # 1. 首先需要添加一个窗口的数据
        # 2. 后面每遍历一个数据都要判断最大值下标是否已经超出了当前窗口
        # 3. 当前值大于原最大值则删除所有候选数据，当前值小于最大值，则判断是否能放在备选位置
        if not nums:
            return []
        res = []
        window = deque()
        for i in range(k):
            while window and nums[i] > nums[window[-1]]:
                window.pop()
            window.append(i)

        for i in range(k, len(nums)):
            res.append(nums[window[0]])
            while window and nums[i] > nums[window[-1]]:
                window.pop()
            if window and i - window[0] >= k:
                window.popleft()
            window.append(i)
        res.append(nums[window[0]])
        return res
```

```
给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。你需要让组成和的完全平方数的个数最少。
from collections import deque
class Solution(object):
    def numSquares(self, n):
        """
        :type n: int
        :rtype: int
        """
        '''
        # 1. 使用动态规划求解， dp[n] = min(dp[n-num1], dp[n-num2], ..., dp[n-numx])+1
        # num1, num2, ..., numx为小于n的所有平方数

        dp = [0 for _ in range(n+1)]
        dp[0] = 0
        dp[1] = 1
        for i in range(2, n+1):
            dp[i] = min(dp[i - j**2] for j in range(1, 1+int(i**0.5))) + 1
        return dp[-1]
        '''
        # 通过建立图的模型，将0到n每个数看做一个节点，如果节点之间相差一个平方数，那么连接两边；问题转化为在该图中找到一个0到n的最短路径
        square = [i**2 for i in range(1, int(n**0.5)+1)]
        r = 0
        seen = set()

        q = deque()
        q.append((0, r))

        # 进入队列循环
        while q:
            cur, step = q.popleft()
            step += 1

            # 放入周围元素
            for a in square:
                a += cur
                if a > n:
                    break
                if a == n:
                    return step
                if a < n and (a, step) not in seen:
                    seen.add((a, step))
```

```
                    q.append((a, step))
        return 0
```

```
给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。
class Solution:
    def moveZeroes1(self, nums):
        """
        :type nums: List[int]
        :rtype: void Do not return anything, modify nums in-place instead.
        """
        flag = []
        for i in range(len(nums)):
            if nums[i] == 0:
                flag.append(i)
        print(flag)
        j = 0
        for i in flag:
            nums.pop(i-j)
            j += 1
        nums.extend([0]*len(flag))

    def moveZeroes(self, nums):
        # 使用一个参数表示指向最前面0的位置，如果有不是0的值，则一次交换
        flag = 0
        for i in range(len(nums)):
            if nums[i]:
                nums[flag], nums[i] = nums[i], nums[flag]
                flag += 1
```

```
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。
class Solution:
    def findDuplicate(self, nums: List[int]) -> int:
        '''
        # 1. 数组映射法：但会修改数组本身，如果不修改数组本身，有需要申请额外的O（n）的内存
        index = 0
        while True:
            if nums[index] > 0:
                index = nums[index]
                nums[index] *= -1
            else:
            return -nums[index]
        '''
        # 2. 二分法
        length = len(nums)
        left = 1
        right = length-1

        while left < right:
            mid = left + (right-left+1) // 2

            # 统计当前数组小于mid的数字
            counter = 0
            for num in nums:
                if num < mid:
                    counter += 1

            # 和真正小于mid的数字数量相比，如果counter < mid,说明重复的数大于mid，反之小于mid
            if counter >= mid:
                right = mid-1
            else:
                left = mid
        return left
```

```
297. 二叉树的序列化与反序列化

class Codec:

    def serialize(self, root):
        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """
        # 序列化就是一个层次遍历即可
        if not root:
            return '[]'
        res = []
        queue = []
        queue.append(root)
        res.append(root.val)
        while queue:
            node = queue.pop(0)
            if node.left:
                res.append(node.left.val)
            else:
                res.append('#')
            if node.right:
                res.append(node.right.val)
            else:
                res.append('#')
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        while res[-1] == '#':
            res.pop()
        return str(res)


    def deserialize(self, data):
        """Decodes your encoded data to tree.

        :type data: str
        :rtype: TreeNode
        """
        data = eval(data)
```

```
44.         if not data:
45.             return None
46.         head = TreeNode(data[0])
47.         queue = [head]
48.         res = head
49.         i = 1
50.         while i < len(data):
51.             node = queue.pop(0)
52.             if data[i] != '#':
53.                 left_child = TreeNode(data[i])
54.                 node.left = left_child
55.                 queue.append(left_child)
56.             if i+1 < len(data) and data[i+1] != '#':
57.                 right_child = TreeNode(data[i+1])
58.                 node.right = right_child
59.                 queue.append(right_child)
60.             i += 2
61.         return res
```

```
给定一个无序的整数数组，找到其中最长上升子序列的长度。
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        '''
        # 1. 动态规划，转移方程dp[i] = max(dp[j]+1 if j < i and nums[i] > nums[j])
        # 时间复杂度：O（n**2）
        size = len(nums)
        if size <= 1:
            return size

        dp = [1 for _ in range(size)]

        for i in range(size):
            for j in range(i):
                if nums[i] > nums[j]:
                    dp[i] = max(dp[i], dp[j]+1)
        return max(dp)
        '''
        # 采用贪心算法（二分法），时间复杂度为O（nlogn）
        size = len(nums)
        if size <= 1:
            return size

        tail = []
        for num in nums:
            # 找到大于等于num的第一个数， 采用二分法
            left = 0
            right = len(tail)
            while left < right:
                mid = left + (right-left)//2
                if tail[mid] < num:
                    left = mid + 1
                else:
                    right = mid
            if left == len(tail):
                tail.append(num)
            else:
                tail[left] = num
        return len(tail)
```

```
删除最小数量的无效括号，使得输入的字符串有效，返回所有可能的结果。
class Solution:
    def removeInvalidParentheses(self, s: str) -> List[str]:
        # 该题与生成括号方式互为相反的过程，生成是我们需要记录左边和右边括号的个数，删除时我们也需要。
        # 1. 需要先找出不合法的左括号个数和右括号个数
        # 2. 利用dfs不断删除'('或者')'，直到合法个数为0
        # 3. 检验删除后的括号串是否合法

        def dfs(s):
            mi = calc(s)
            if mi == 0:
                return [s]
            ans = []
            for x in range(len(s)):
                if s[x] in ['(', ')']:
                    ns = s[:x] + s[x+1:]
                    if ns not in visited and calc(ns) < mi:
                        visited.add(ns)
                        ans.extend(dfs(ns))
            return ans

        def calc(s):
        # 统计字符串s中包含失配的括号数目
            a = b = 0
            for c in s:
                a += {'(':1, ')':-1}.get(c, 0)
                b += a < 0
                a = max(a, 0)
            return a + b

        visited = set([s])
        return dfs(s)
```

```
309. 最佳买卖股票时机含冷冻期

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # 动态规划求解，引入辅助数组sells[i], buys[i],代表第i天不持有股票所能获得最大利益和第i天持有股票所能获得最大利益

        # sells[0] = 0, sells[1] = max(0, prices[1] - price[0])
        # buys[0] = -prices[0], buys[1] = max(-prices[0], -prices[1])

        # 状态转移方程:
        # 1. sells[i] = max(sells[i-1], buys[i-1]+prices[i]), 第i天不持有股票，①：昨天就不持有；②：昨天持有，但是今天卖掉了
        # 2. buys[i] = max(buys[i-1], sells[i] - prices[i]), 第i天持有股票，①：i-1天就持有；②：第i-1天不持有股票，且i-2天也不持有股票
        length = len(prices)
        if length < 2:
            return 0
        buys = [None]*length
        sells = [None]*length
        sells[0] = 0
        sells[1] = max(0, prices[1] - prices[0])
        buys[0] = -prices[0]
        buys[1] = max(-prices[0], -prices[1])

        for x in range(2, length):
            sells[x] = max(sells[x-1], buys[x-1]+prices[x])
            buys[x] = max(buys[x-1], sells[x-2]-prices[x])

        return sells[-1]
```

```
312. 戳气球

class Solution:
    def maxCoins(self, nums: List[int]) -> int:
        coins = [1] + [i for i in nums if i >0] + [1]
        n = len(coins)
        max_coins = [[0 for _ in range(n)] for _ in range(n)]

        for k in range(2, n):
            # 上面这层循环是为了控制left和right之间的距离的，即戳破left到right之间的气球获取的最大硬币数量
            for left in range(n-k):
                right = left + k
                for i in range(left+1, right):
                    # 如果最后留下的是气球i还没有戳破
                    max_coins[left][right] = max(max_coins[left][right], coins[left]*coins[i]*coins[right]+max_coins[left][i]+max_coins[i][right])

        return max_coins[0][-1]
```

```
给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        # 和斐波那契数组类似，动态规划求解，递推公式dp[n] = min(dp[n-c1], dp[n-c2], ..., dp[n-cx])+1, ci为所有的零钱
        if amount == 0:
            return 0
        if amount < 0:
            return -1
        dp = [0 for _ in range(amount+1)]

        for i in range(1, amount+1):
            cost = float("inf")
            for c in coins:
                if i - c >= 0:
                    cost = min(cost, dp[i-c]+1)
            dp[i] = cost

        if dp[-1] == float("inf"):
            return -1
        else:
            return dp[-1]
```

```
在上次打劫完一条街道之后和一圈房屋后，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。
class Solution:
    def rob(self, root: TreeNode) -> int:
        # 递归解决，1：抢劫root节点，那么最高金额为root.val加上不抢劫root的left_child的左子树金额+不抢劫root的right_child的右子树金额
        # 2. 不抢劫root节点，那么最高金额max(抢劫root.left的最大金额， 不抢劫root的最大金额) + max(抢劫root.right的最大金额，不抢劫root的最大金额)
        def postorder(root):
            if not root:
                return (0, 0)
            l = postorder(root.left)
            r = postorder(root.right)
            return (root.val + l[1] + r[1], max(l[0], l[1])+max(r[0], r[1]))
        r = postorder(root)
        return max(r[0], r[1])
```

```
给定一个非负整数 num。对于 0 ≤ i ≤ num 范围中的每个数字 i ，计算其二进制数中的 1 的数目并将它们作为数组返回。
class Solution:
    def countBits(self, num: int) -> List[int]:
        # 分开奇数和偶数，每一个奇数都比它前面一个偶数多一个1
        # 每一个偶数的1的个数都是其右移一位之后的数相同
        result = [0 for _ in range(num+1)]
        for i in range(1, num+1):
            if i % 2 == 1:
                result[i] = result[i-1]+1
            else:
                result[i] = result[i>>1]
        return result
```

```
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。
# 默认的heapq为小顶堆
'''
堆：根节点最大的堆为大顶堆，反之为小顶堆
'''
import heapq
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        # 先用hash表统计词频，再建立大顶堆
        heap_max = []
        count_dict = {}
        res = []
        for i in nums:
            count_dict[i] = count_dict.get(i, 0) + 1

        for key, value in count_dict.items():
            # 这里value为负值，是因为heapq默认为最小堆
            heapq.heappush(heap_max, (-value, key))
        for j in range(k):
            p = heapq.heappop(heap_max)
            res.append(p[1])
        return res
```

```
给定一个经过编码的字符串，返回它解码后的字符串。s = "3[a]2[bc]", 返回 "aaabcbc".
class Solution:
    def decodeString(self, s: str) -> str:
        # 涉及到括号匹配的用栈就对了
        stack = []
        res = ''
        for i, char in enumerate(s):
            if char != ']':
                stack.append(char)
            else:
                string = ''
                while not stack[-1].isnumeric():
                    string = stack.pop() + string

                count = ''
                while stack and stack[-1].isnumeric():
                    count = stack.pop()+count
                if count:
                    string = string[1:]*int(count)

                if stack:
                    stack.append(string)
                else:
                    res += string
        return res + ''.join(stack)
```

```
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。
equations(方程式) = [ ["a", "b"], ["b", "c"] ],
values(方程式结果) = [2.0, 3.0],
class Solution:
    def calcEquation(self, equations: List[List[str]], values: List[float], queries: List[List[str]]) -> List[float]:
        # 非常明显的dfs寻找最短路径。先构图，再dfs
        # 利用dict创建图
        graph = {}
        for (x, y), value in zip(equations, values):
            if x in graph:
                graph[x][y] = value
            else:
                graph[x] = {y: value}
            if y in graph:
                graph[y][x] = 1/value
            else:
                graph[y] = {x: 1/value}

        # dfs寻找s到t的路径并返回叠成后的边权重成绩
        def dfs(s, t):
            if s not in graph:
                return -1
            if t == s:
                return 1
            # 遍历与点s相连的节点
            for node in graph[s].keys():
                if node == t:
                    return graph[s][node]
                elif node not in visited:
                    visited.add(node)
                    v = dfs(node, t)
                    if v != -1:
                        return graph[s][node]*v
            return -1

        res = []
        for qs, qt in queries:
            # visited很重要
            visited = set()
            res.append(dfs(qs, qt))
```

```
41.         return res
```

```
1. 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。
2. class Solution:
3.     def reconstructQueue(self, people: List[List[int]]) -> List[List[int]]:
4.         # 先排序再插入
5.         people = sorted(people, key = lambda p:p[1])
6.         people = sorted(people, key= lambda p:p[0], reverse=True)
7.
8.         res = []
9.         for h, k in people:
10.            res.insert(k, [h, k])
11.
12.        return res
```

```
1. 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。
2. class Solution:
3.     def canPartition(self, nums: List[int]) -> bool:
4.         target = sum(nums)
5.         # 如果
6.         if target % 2:
7.             return False
8.         target = target // 2
9.         sub_sum = set([0])
10.        for i in nums:
11.            tmp = sub_sum.copy()
12.            for j in sub_sum:
13.                tmp.add(i+j)
14.            sub_sum = tmp
15.        return target in sub_sum
```

```
给定一个二叉树，它的每个结点都存放着一个整数值。找出路径和等于给定数值的路径总数。
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None

class Solution:
    def pathSum(self, root: TreeNode, sum: int) -> int:
        # 使用两个递归，一个用于遍历每一个节点，另一个用于从该节点想下找存在的路径个数
        if not root:
            return 0
        def dfs(root, sum):
            count = 0
            if not root:
                return 0
            if root.val == sum:
                # 这里不能返回，因为后面还可能存在路径
                count += 1
            count += dfs(root.left, sum-root.val)
            count += dfs(root.right, sum-root.val)
            return count

        return dfs(root, sum)+self.pathSum(root.left, sum)+self.pathSum(root.right, sum)
```

```
438. 找到字符串中所有字母异位词，s: "cbaebabacd" p: "abc" out:[0, 6]

class Solution:
    def findAnagrams(self, s, p):
        """
        :type s: str
        :type p: str
        :rtype: List[int]
        """
        """
        :s和p都是字符串，在s中找到p的anagram（异位构词）
        """
        # 滑窗+对象对比
        # 如何比较异构词和模式词是否相等，可以采用Counter对象进行比较，这是关键
        res = []
        len_p = len(p)
        cp = collections.Counter(p)
        cs = collections.Counter()

        for i in range(len(s)):
            cs[s[i]] += 1
            # 当异构词数量超过模式词，将第一个字符次数减小1，如果为0，还要删除该对象
            if i >= len_p:
                cs[s[i-len_p]] -= 1
                if cs[s[i-len_p]] == 0:
                    del cs[s[i-len_p]]

            if cs == cp:
                res.append(i - len_p + 1)
        print(res)
        return res
```

```
给定一个范围在 1 ≤ a[i] ≤ n ( n = 数组大小 ) 的 整型数组，数组中的元素一些出现了两次，另一些只出现一次。找到所有在 [1, n] 范围之间没有出现在数组中的数字。
class Solution:
    def findDisappearedNumbers(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        """
        :找出[1,n]在nums中没有出现的元素，空间O（1），时间O（n）
        这道题由于数组长度为n，数组元素大小也在1和n之间，因此采用和找到数组中重复元素类似的方法，因为这里有多少个重复的数就有多少消失的数，且消失的数字刚好被重复的数占位。
        """
        len_n = len(nums)
        j = 0
        res = []
        while j < len_n:
            # 将元素放在自己的位置上
            if nums[nums[j]-1] != nums[j]:
                nums[nums[j]-1], nums[j] = nums[j], nums[nums[j]-1]
            else:
                j += 1
        # 和自己位置不相等的数就是消失的数
        for i in range(len(nums)):
            if i+1 != nums[i]:
                res.append(i+1)
        return res
```

```
两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。
class Solution:
    def hammingDistance(self, x, y):
        """
        :type x: int
        :type y: int
        :rtype: int
        """
        # 1. 先得到两个数的二进制表示，然后填充成同样的长度，最后比较不同位置求和
        # 2. 先异或再求1的个数
        return format(x^y, 'b').count('1')
```

```
给定一个非负整数数组，a1，a2，...，an，和一个目标数，S。现在你有两个符号 + 和 -。对于数组中的任意一个整数，你都可以从 + 或 -中选择一个符号添加在前面。
class Solution:
    def findTargetSumWays(self, nums: List[int], S: int) -> int:
        # 求排列和回溯的方法

        # dict由于是不可变向量，因此递归过程当中不会随递归深度而改变
        def backtrace(nums,i, S, tmp, dp={}):
            if i == len(nums):
                if S == tmp:
                    return 1
                return 0
            if (tmp, i) in dp:
                return dp[(tmp, i)]
            a = backtrace(nums, i+1, S, tmp+nums[i])
            b = backtrace(nums, i+1, S, tmp-nums[i])
            dp[(tmp, i)] = a+b
            return a + b

        return backtrace(nums,0, S, 0)
```

给定一个二叉搜索树（Binary Search Tree），把它转换成为累加树（Greater Tree），使得每个节点的值是原来的节点值加上所有大于它的节点值之和。

```
class Solution:
    def convertBST(self, root: TreeNode) -> TreeNode:
        # 采用右中左遍历求和即可
        
        self.sum = 0
        def mid_traversal(root):
            if not root:
                return root
            mid_traversal(root.right)
            self.sum += root.val
            root.val = self.sum
            mid_traversal(root.left)
            return root
        return mid_traversal(root)
```

```
1. 给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过根结点。\
2. class Solution:
3.     def diameterOfBinaryTree(self, root: TreeNode) -> int:
4.         # 递归
5.         self.distance = 0
6.         def recursion(root):
7.             if not root:
8.                 return 0
9.             left = recursion(root.left)
10.            right = recursion(root.right)
11.            self.distance = max(self.distance, left+right)
12.            return 1 + max(left, right)
13.        if not root:
14.            return 0
15.        recursion(root)
16.        return self.distance
```

```
给定一个整数数组和一个整数 k，你需要找到该数组中和为 k 的连续的子数组的个数。
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        '''超时
        # 使用dp[i][j]保存从i到j的子数组的和
        count = 0
        dp = [[0 for _ in range(len(nums))] for _ in range(len(nums))]
        for i in range(len(nums)):
            dp[i][i] = nums[i]
            if dp[i][i] == k:
                count += 1

        # 初始化dp
        for j in range(1, len(nums)):
            dp[0][j] = dp[0][j-1]+nums[j]
            if dp[0][j] == k:
                count += 1

        # 递推公式dp[i][j] = dp[0][j] - dp[0][i]
        for i in range(1, len(nums)):
            for j in range(i+1, len(nums)):
                dp[i][j] = dp[0][j] - dp[0][i-1]
                if dp[i][j] == k:
                    count += 1
        return count
        '''
        result, cur_sum = 0, 0
        sum_dict = {0:1}
        for num in nums:
            cur_sum += num
            if cur_sum - k in sum_dict:
                result += sum_dict[cur_sum - k]
            if cur_sum in sum_dict:
                sum_dict[cur_sum] += 1
            else:
                sum_dict[cur_sum] = 1
        return result
```

给定一个整数数组，你需要寻找一个连续的子数组，如果对这个子数组进行升序排序，那么整个数组都会变为升序排序。你找到的子数组应是最短的，请输出它的长度。

```
class Solution:
    def findUnsortedSubarray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        nums_1 = [i for i in nums]
        nums_1.sort()
        i = 0
        j = 0
        flag = 0
        left = 0
        right = len(nums)
        l = len(nums)
        while i < len(nums):
            if nums[i] != nums_1[i]:
                left = i
                flag = 1
                break
            i += 1
        if i == len(nums):
            if flag == 0:
                return 0
            else:
                return len(nums)
        while j < len(nums):
            if nums[l-j-1] != nums_1[l-j-1]:
                right = l - j
                break
            j += 1
        print(left, right)
        return right - left
```

```
给定两个二叉树，想象当你将它们中的一个覆盖到另一个上时，两个二叉树的一些节点便会重叠。
class Solution:
    def mergeTrees(self, t1, t2):
        """
        :type t1: TreeNode
        :type t2: TreeNode
        :rtype: TreeNode
        """
        if not t1 and not t2:
            return t1
        if not t1 and t2:
            return t2
        if t1 and not t2:
            return t1
        #将两颗树的值相加，形成新的树
        def frontBianli(t1, t2):
            if not t1 and not t2:
                return
            print(t1.val)
            t1.val = t1.val + t2.val
            print(t1.val,"p")
            if t1.left and t2.left:
                frontBianli(t1.left, t2.left)
            if not t1.left and t2.left:
                t1.left = TreeNode(0)
                frontBianli(t1.left, t2.left)
            if t1.left and not t2.left:
                t2.left = TreeNode(0)
                frontBianli(t1.left, t2.left)

            if t1.right and t2.right:
                frontBianli(t1.right, t2.right)
            if not t1.right and t2.right:
                t1.right = TreeNode(0)
                frontBianli(t1.right, t2.right)
            if t1.right and not t2.right:
                t2.right = TreeNode(0)
                frontBianli(t1.right, t2.right)
        frontBianli(t1, t2)
        return t1
```

```
给定一个用字符数组表示的 CPU 需要执行的任务列表。其中包含使用大写的 A - Z 字母表示的26 种不同种类的任务。任务可以以任意顺序执行，并且每个任务都可以在 1 个单位时间内执行完。CPU 在任何一个单位时间内都可以执行一个任务，或者在待命状态。
class Solution:
    def leastInterval(self, tasks: List[str], n: int) -> int:
        # 分析
        '''
        重点是找到了数学关系，可以按照最多的字母分组，比如最多的字母是A有a个，那么说明一定有a组，然后每组的数量是n+1个，但是最后一组不是，最后一组不用空闲时间补完，因此只要关心最后一组的数量 即可
        # 最后之所有要比较len(tasks)是因为，如果maxCount > n,必然存在不需要待命的最短时间的执行时间，就是len(tasks)
        '''
        if n < 1:
            return len(tasks)
        d = collections.Counter(tasks)
        print(d)
        max_count = 0
        for key in d:
            max_count = max(max_count, d[key])
        num_max_count = 0
        for key in d:
            if d[key] == max_count:
                num_max_count+= 1

        return max((max_count-1)*(n+1)+num_max_count, len(tasks))
```

```
给定一个字符串，你的任务是计算这个字符串中有多少个回文子串。
class Solution:
    def countSubstrings(self, s: str) -> int:
        # 利用中心扩展法，分为两种情况，一种是回文子串中心是一个字符，另一种是回文串中心是两个字符
        res = 0
        # 中心只有一个点的回文串
        for i in range(len(s)):
            res += 1
            res += (self.center_widen(s, i-1, i+1))

        # 中心有两个点的回文串
        for i in range(len(s)-1):
            if s[i] == s[i+1]:
                res += 1
                res += (self.center_widen(s, i-1, i+2))
        return res

    def center_widen(self, nums, i, j):
        count = 0
        while i> -1 and j < len(nums) and nums[i] == nums[j]:
            count += 1
            i -= 1
            j += 1
        return count
```

```
例如，给定一个列表 temperatures = [73, 74, 75, 71, 69, 72, 76, 73]，你的输出应该是 [1, 1, 4, 2, 1, 1, 0, 0]。
class Solution:
    def dailyTemperatures(self, T: List[int]) -> List[int]:
        # 维护一个递减栈，栈顶的元素总是最小的
        # 对比stack[-1]与当前元素d，如果d > stack[-1]，那么弹出stack[-1],此时stack[-1]对应的列表值为d-stack[-1]，然后继续比较，知道d< stack[-1],将d加入栈中
        stack = []
        res = [0 for _ in range(len(T))]

        for key, value in enumerate(T):

            while stack and T[stack[-1]] < value:
                res[stack[-1]] = key - stack[-1]
                stack.pop()
            stack.append(key)

        return res
```