[1 编程 3](#_Toc202721276)

[2 双指针 4](#_Toc202721277)

[2.1 三数之和 4](#_Toc202721278)

[2.2 5. 最长回文子串 ★★★ 5](#_Toc202721279)

[2.3 链表 6](#_Toc202721280)

[2.3.1 206. 反转链表 6](#_Toc202721281)

[2.3.2 142. 环形链表 II 7](#_Toc202721282)

[2.3.3 24. 两两交换链表中的节点 7](#_Toc202721283)

[2.3.4 19. 删除链表的倒数第 N 个结点 9](#_Toc202721284)

[2.3.5 203. 移除链表元素 9](#_Toc202721285)

[2.4 原地，27. 移除元素 10](#_Toc202721286)

[2.5 原地，26. 删除有序数组中的重复项， 10](#_Toc202721287)

[2.6 42. 接雨水 11](#_Toc202721288)

[2.7 LCR 039. 柱状图中最大的矩形 12](#_Toc202721289)

[3 栈/队列 13](#_Toc202721290)

[3.1 栈 13](#_Toc202721291)

[3.1.1 14. 最长公共前缀 ★★ 13](#_Toc202721292)

[3.1.2 20. 有效的括号 ★★ 14](#_Toc202721293)

[3.1.3 1614. 括号的最大嵌套深度 14](#_Toc202721294)

[3.1.4 LCR 038. 每日温度，单调栈|----(↘) 15](#_Toc202721295)

[3.1.5 496. 下一个更大元素 I，单调栈|----(↘) 16](#_Toc202721296)

[3.1.6 LCR 039. 柱状图中最大的矩形，|----(↗) 16](#_Toc202721297)

[3.2 队列-滑动窗口 17](#_Toc202721298)

[3.2.1 209. 长度最小的子数组  ★★ 18](#_Toc202721299)

[3.2.2 3. 无重复字符的最长子串  ★★ 18](#_Toc202721300)

[3.2.3 239. 滑动窗口最大值 19](#_Toc202721301)

[3.2.4 215. 数组中的第K个最大元素 21](#_Toc202721302)

[3.2.5 347. 前 K 个高频元素 22](#_Toc202721303)

[4 BFS & DFS 23](#_Toc202721304)

[4.1 二叉树 23](#_Toc202721305)

[4.1.1 94. 二叉树的中序遍历 23](#_Toc202721306)

[4.1.2 226. 翻转二叉树 24](#_Toc202721307)

[4.1.3 LCR 145. 对称二叉树 ★★ 24](#_Toc202721308)

[4.1.4 110. 平衡二叉树 25](#_Toc202721309)

[4.1.5 111. 二叉树的最小深度 26](#_Toc202721310)

[4.1.6 662. 二叉树最大宽度 27](#_Toc202721311)

[4.2 路径问题 28](#_Toc202721312)

[4.2.1 257. 二叉树的所有路径 28](#_Toc202721313)

[4.2.2 LCR 080. 组合 ★★ 29](#_Toc202721314)

[4.2.3 图遍历，200. 岛屿数量  ★★★☆ 29](#_Toc202721315)

[4.2.4 130. 被围绕的区域 31](#_Toc202721316)

[4.2.5 994. 腐烂的橘子 ★★★☆ 32](#_Toc202721317)

[5 动态规划 33](#_Toc202721318)

[5.1 674. 最长连续递增序列 33](#_Toc202721319)

[5.2 121. 买卖股票的最佳时机  ★★ 34](#_Toc202721320)

[5.3 64. 最小路径和  ★★ 35](#_Toc202721321)

[5.4 53. 最大子数组和  ★★ 35](#_Toc202721322)

[6 贪心 36](#_Toc202721323)

[6.1 122. 买卖股票的最佳时机 II 36](#_Toc202721324)

[6.2 435. 无重叠区间 37](#_Toc202721325)

[7 其它 38](#_Toc202721326)

[7.1 470. 用 Rand7() 实现 Rand10() ★★ 38](#_Toc202721327)

[7.2 59. 螺旋矩阵 II 38](#_Toc202721328)

[7.3 240. 搜索二维矩阵 II  ★ 39](#_Toc202721329)

v3.0—2025.0706

# 编程

|  |
| --- |
| 数字：  float('inf') float('-inf')  math.sqrt()  字符串：  ' '.join(numbers)  #  numbers必须为 str 列表  map(int/str, list)  #  列表元素转换  str.split() # 会按照所有的“空白字符（whitespace）”进行分割  ord(char)  #  字符转ascii  列表：  [[0] \* 3 for \_ in range(5)]  for i in range(start, stop, step)  list.append() 只会改变list, 但是注意该操作的**返回值是 None**  list.pop(index)  # 默认弹出最后一个（O1），其它O(n)  copy.deepcopy(list)  # 深拷贝  list.sort()  #  list 会改， reverse=True  a = sorted(b)  # 有返回值，不改变原先  sorted(freq\_dict.items(), key=lambda x: -x[1]) #正序or 倒序  队列  from collections import deque # Python 中是用**双向链表**实现  q=deque() / deque([]) / deque([1, 2, 3, 4]) # 初始化  q.popleft()  #  O(1)  q.append() #  O(1)  import heapq  heapq.heappush(min\_heap[], (freq, num)) # 只有小顶堆，大顶堆取相反数  heapq.heappop(min\_heap[])    字典  dic.get('key', 0) # 设置字典默认值      dic[num] = dic.get(num, 0) +1  dic.**setdefault**('a', []) # 列表默认值，会自动添加该默认值进字典  dic.setdefault(‘a’,[]).append(..) # 默认列表append |

华为：

|  |
| --- |
| 模板：  while True:      try:          a = input()          print(a, '!!')  ………      except Exception as e:          print(f'Error: {e}') |

|  |
| --- |
| s = input()   #  输入字符串  a= int(input()) # 输入int  b= float(input()) # 输入 float  # 在一行同时输入a，b，以空格分开  a,b= map(int, input().split())  or ： nums = list( map(int, input().split()) )  # m\*n矩阵          m,n = map(int,input().split())          dp = []          for i in range(m):              dp.append(list(map(int, input().split())))          print(dp) |

# 双指针

## 三数之和

给你一个整数数组 nums ，判断是否存在三元组 [nums[i], nums[j], nums[k]] 满足 i != j、i != k 且 j != k ，同时还满足 nums[i] + nums[j] + nums[k] == 0 。请你返回所有和为 0 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例 1：

输入：nums = [-1,0,1,2,-1,-4]

输出：[[-1,-1,2],[-1,0,1]]

注意，输出的顺序和三元组的顺序并不重要。

|  |
| --- |
| class Solution:      """      思路：      1，排序，有序数组      2，遍历，两数之和      3， 既然，去重，使用set()      """      special\_char = '#' # 使用 Solution.special\_char引用      def \_\_get\_sum\_index(self, nums, target):          """          双指针法， 求给定target 的数组元素          这里为了去重，改返回为 ：['-1#2', '0#1']          """          l = 0          r= len(nums)-1          res = []          while l< r:              sum\_ = nums[l] + nums[r]              if sum\_ == target:                  res\_ = list( map(str, [nums[l] , nums[r]] ) )                  res.append('#'.join(res\_))                  # 更新不要忘记                  l += 1                  r -= 1              elif sum\_ < target:                  l += 1              else:                  r -= 1          return res      def threeSum(self, nums: List[int]) -> List[List[int]]:          nums.sort()          res\_set = set()            for ind, val in enumerate(nums):              #  加速              if ind >0  and nums[ind-1] == nums[ind]: continue              target = 0 -val              target\_indexs = self.\_\_get\_sum\_index(nums[ind+1:], target)              for i in target\_indexs:                  # 去重前: ["-1#-1#2","-1#0#1","-1#0#1"]                  res\_set.add(str(val)+'#'+i)  # 结果去重          return [ list(              map(int, i.split('#')              )) for i in res\_set] |

## [5. 最长回文子串](https://leetcode.cn/problems/longest-palindromic-substring/) ★★★

给你一个字符串 s，找到 s 中最长的 回文 子串。

**示例 1：**

**输入：**s = "babad"

**输出：**"bab"

**解释：**"aba" 同样是符合题意的答案。

**示例 2：**

**输入：**s = "cbbd"

**输出：**"bb"

|  |
| --- |
| class Solution:      def longestPalindrome(self, s: str) -> str:          """          讲解 ： https://www.youtube.com/watch?v=47159E0ZAwI          回文字符串分为2种：为了统一，中间塞入特殊字符串              aba    -- >  a#b#a#              abba    -- > a#b#b#a#          遍历，然后从中间， 向两边使用双指针          """          res\_s = ''  #  记录结果          special\_s = "#" # 这里字符串收尾均添加特殊字符，为了后面数组边界处理方便          new\_s = ''          for i in s:              new\_s += i+special\_s          len\_ = len(new\_s)          # print(new\_s, '?????')  #  构造新的字符串，b#a#b#a#d#          for ind,val in enumerate(new\_s):              l=r=ind              while l>=0 and r <len\_ and new\_s[l]==new\_s[r]:                  l -= 1                  r+= 1              #当前的最大回文串              Palindrome\_string= new\_s[l+1:r].replace(special\_s, "")              if len(Palindrome\_string) > len(res\_s):                  res\_s = Palindrome\_string          return res\_s |

## 链表

### [206. 反转链表](https://leetcode.cn/problems/reverse-linked-list/)

给你单链表的头节点 head ，请你反转链表，并返回反转后的链表。

**示例 1：**



|  |
| --- |
| # Definition for singly-linked list.  # class ListNode:  #     def \_\_init\_\_(self, val=0, next=None):  #         self.val = val  #         self.next = next  class Solution:      def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:          """解决的其实是这样的问题：          prev                     cur          o                     1-->2 -->3 -->o          o <-- 1               2 -->3 -->o          o <-- 1<-- 2         3 -->o          o <-- 1<-- 2<-- 3     o          """          prev = None          cur = head            while cur:              cur\_next = cur.next   # 保存下一个节点              cur.next = prev        # 当前节点指向前一个节点              prev = cur # 更新              cur = cur\_next          return prev  # 最后 prev 是新的头节点 |

### [142. 环形链表 II](https://leetcode.cn/problems/linked-list-cycle-ii/)

链表环判断，有的话，反环的入口点

**示例 1：**



**输入：**head = [3,2,0,-4], pos = 1

**输出：**返回索引为 1 的链表节点

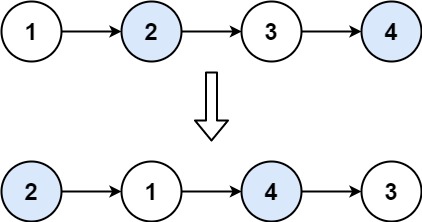
**解释：**链表中有一个环，其尾部连接到第二个节点。

|  |
| --- |
| # Definition for singly-linked list.  # class ListNode:  #     def \_\_init\_\_(self, x):  #         self.val = x  #         self.next = None  class Solution:      def detectCycle(self, head: Optional[ListNode]) -> Optional[ListNode]:          # f-->2步, s->1步; slow,fast 出发位置要一样          s = f = head          #  判断是否有环          while f and f.next:              f = f.next.next              s = s.next              if f == s:  # 相遇了                  break          else:  # 正常结束循环说明无环              return None          f = head  #  回到出发点起始位置          while f != s:              f, s = f.next, s.next  # 同步步伐          # print(fast.val, '????')          return f |

### [24. 两两交换链表中的节点](https://leetcode.cn/problems/swap-nodes-in-pairs/)

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。你必须在不修改节点内部的值的情况下完成本题（即，只能进行节点交换）。

**示例 1：**



**输入：**head = [1,2,3,4]

**输出：**[2,1,4,3]

|  |
| --- |
| # Definition for singly-linked list.  # class ListNode:  #     def \_\_init\_\_(self, val=0, next=None):  #         self.val = val  #         self.next = next  class Solution:      def swapPairs(self, head: Optional[ListNode]) -> Optional[ListNode]:          """          prev-->n0-->n1-->....          """          guard = ListNode() # 哨兵节点          guard.next = head          prev = guard          # 只要还能两两交换就继续          while prev.next and prev.next.next:              node0 = prev.next              node1 = prev.next.next              # 交换两个节点              tmp = node1.next              node1.next= node0              node0.next = tmp              prev.next = node1              # 移动 prev 到新的位置              prev = node0          return guard.next |

### [19. 删除链表的倒数第 N 个结点](https://leetcode.cn/problems/remove-nth-node-from-end-of-list/)



**输入：**head = [1,2,3,4,5], n = 2

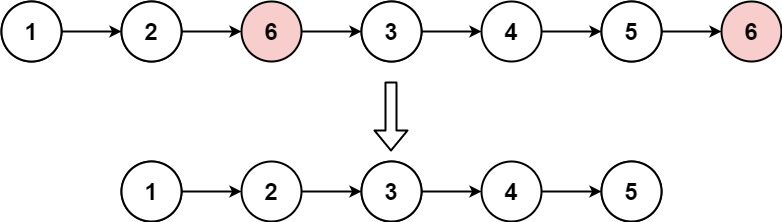
**输出：**[1,2,3,5]

|  |
| --- |
| # Definition for singly-linked list.  # class ListNode:  #     def \_\_init\_\_(self, val=0, next=None):  #         self.val = val  #         self.next = next  class Solution:      def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:          """          关键是找到 slow, s.next =del          """          guard = ListNode()          guard.next = head          s= guard          f= head          while n>0:              f= f.next              n -= 1          while f: #结束后，s 对应的为待删除节点的前一个节点              f = f.next              s = s.next          s.next = s.next.next          return guard.next |

### [203. 移除链表元素](https://leetcode.cn/problems/remove-linked-list-elements/)

给你一个链表的头节点 head 和一个整数 val ，请你删除链表中所有满足 Node.val == val 的节点，并返回 **新的头节点** 。

**示例 1：**



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

**输出：**[1,2,3,4,5]

|  |
| --- |
| guard = ListNode()          guard.next= head          f=head          s=guard #  注意这里 f初始比s快          while f:              if f.val==val:                  f= f.next                  s.next = f              else:                  f= f.next                  s= s.next          return guard.next |

## 原地，[27. 移除元素](https://leetcode.cn/problems/remove-element/)

**输入：**nums = [3,2,2,3], val = 3

**输出：**2, nums = [2,2,\_,\_]

**解释：**你的函数函数应该返回 k = 2, 并且 nums中的前两个元素均为 2。

你在返回的 k 个元素之外留下了什么并不重要（因此它们并不计入评测）。

## 原地，[26. 删除有序数组中的重复项](https://leetcode.cn/problems/remove-duplicates-from-sorted-array/)，

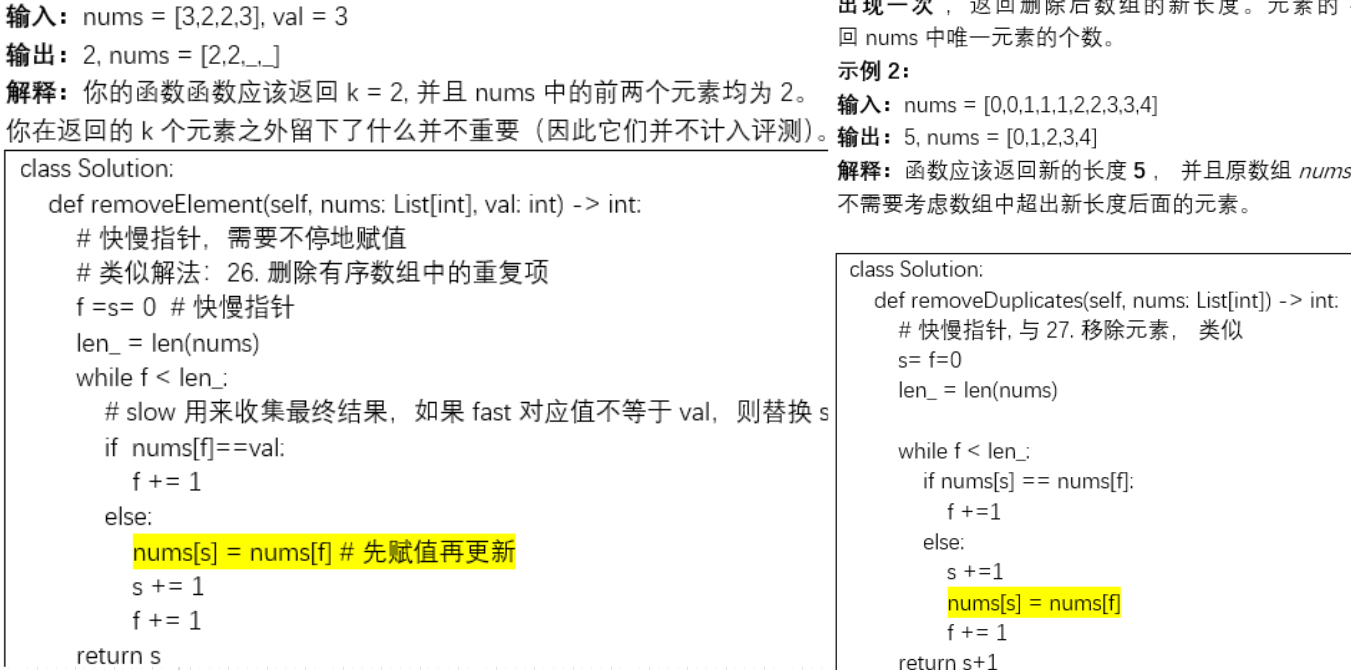
给你一个 **非严格递增排列** 的数组 nums ，请你[**原地**](http://baike.baidu.com/item/%E5%8E%9F%E5%9C%B0%E7%AE%97%E6%B3%95) 删除重复出现的元素，使每个元素 **只出现一次** ，返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 nums 中唯一元素的个数。

**输入：**nums = [0,0,1,1,1,2,2,3,3,4]

**输出：**5, nums = [0,1,2,3,4]

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

二者代码一样几乎：



## [42. 接雨水](https://leetcode.cn/problems/trapping-rain-water/)

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。



**输入：**height = [0,1,0,2,1,0,1,3,2,1,2,1]

**输出：**6

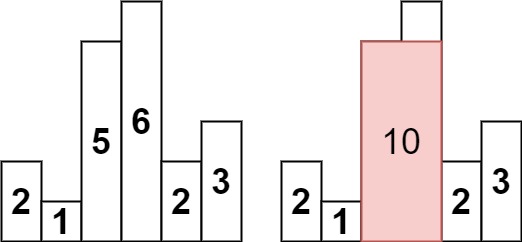
**解释：**上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情

|  |
| --- |
| class Solution:      def trap(self, height: List[int]) -> int:          # 如果数组为空 或者 柱子数量小于3，无法形成凹槽，直接返回0          if not height or len(height) < 3:              return 0            # 双指针初始化          left = 0          right = len(height) - 1            #left\_max：表示左指针左侧（包括左指针本身）的最高柱子。          #right\_max：表示右指针右侧（包括右指针本身）的最高柱子          left\_max = 0          right\_max = 0          res = 0  # 最终结果，记录能接的雨水总量          while left < right: # <= 均可              # 更新当前看到的左/右边最高柱子              if height[left] > left\_max:                  left\_max = height[left]              if height[right] > right\_max:                  right\_max = height[right]              # 核心逻辑：              # 哪边的 max 小，说明哪边是瓶颈，水只能由它决定              # res 是总积水量，每次循环累加当前柱子能存储的水。              # 列计算雨水面积！！              #            \_\_              #    \_\_     |  |              #    |  | ↑ |  |              #    |\_\_| ↑ |\_|              if left\_max < right\_max:                  res += left\_max - height[left]                  left += 1              else:                  res += right\_max - height[right]                  right -= 1          # print(left, right, '@@')  # 停在了最高的柱子处          return res |

## [LCR 039. 柱状图中最大的矩形](https://leetcode.cn/problems/0ynMMM/)

给定非负整数数组 heights ，数组中的数字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。



**输入：**heights = [2,1,5,6,2,3]

**输出：**10

**解释：**最大的矩形为图中红色区域，面积为 10

|  |
| --- |
| 方法1：单调递增栈，【栈/队列章节】 [LCR 039. 柱状图中最大的矩形，|----(↗)](#_LCR_039._柱状图中最大的矩形，|----(↗))  方法2：双指针      def largestRectangleArea(self, heights: List[int]) -> int:          """          找出柱状图中能组成的最大矩形面积          思路：          - 对于每个柱子 i，向左和向右扩展，找到第一个比它矮的位置 left 和 right          - 那么以 heights[i] 为高的矩形宽度为 (right - left - 1)          - 面积 = height \* width，更新 max\_area          """          max\_area = 0          n = len(heights)          for i in range(n):              left = i - 1              right = i + 1              cur = heights[i]              if n \* heights[i] > max\_area: #  剪枝加速                  while left >= 0 and heights[left] >= cur:                      left -= 1                  while right <= n-1 and heights[right] >=cur:                      right += 1                  max\_area = max(max\_area, (right-left-1)\*cur)          return max\_area |

# 栈/队列

## 栈

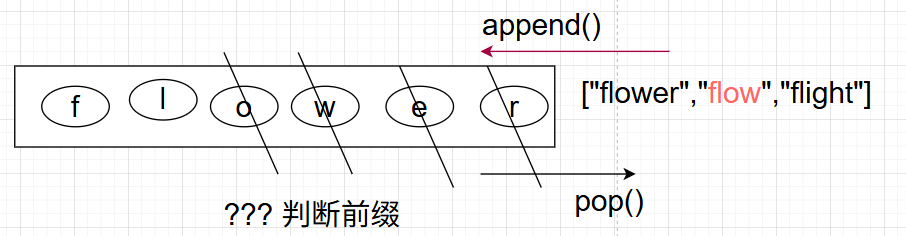
### [14. 最长公共前缀](https://leetcode.cn/problems/longest-common-prefix/) ★★

**示例 1：**

**输入：**strs = ["flower","flow","flight"]

**输出：**"fl"

|  |
| --- |
| class Solution:      def \_\_judge\_prefix(self, str\_long, strs):          #  str\_long : 长字符串          # 检查 strs 是否是 str\_long 的前缀（即 str\_long 是否以 strs 开头）          len\_ = len(strs)          return str\_long[:len\_] == strs      def longestCommonPrefix(self, strs: List[str]) -> str:          # 使用栈, 不是前缀则弹出          stack = []          for i in strs[0]:              stack.append(i)            for str\_ in strs:              while stack and not self.\_\_judge\_prefix(str\_, ''.join(stack)):                  stack.pop()          return ''.join(stack) |



### [20. 有效的括号](https://leetcode.cn/problems/valid-parentheses/description/) ★★

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串 s ，判断字符串是否有效。

输入：s = "()[]{}"

输出：true

|  |
| --- |
| class Solution:      def isValid(self, s: str) -> bool:          """  [] 模拟栈，append(), 与 pop()  ---------------  | ( (           ....   ?? ) ]  ---------------          """          dic = { '(':')', '{':'}', '[':']'}          stack = []          for i in s:              if i in dic:  #  左括号，入栈                  stack.append(i)              else: # i 为 】】】】                  if not stack: return False                  tmp = stack.pop()                  if i != dic[tmp]:                      return  False          return not stack #  布尔运算 |

类似：[**1047. 删除字符串中的所有相邻重复项**](https://leetcode.cn/problems/remove-all-adjacent-duplicates-in-string/)

**输入：**"abbaca"

**输出：**"ca"

### [1614. 括号的最大嵌套深度](https://leetcode.cn/problems/maximum-nesting-depth-of-the-parentheses/)

给定 有效括号字符串 s，返回 s 的 嵌套深度。嵌套深度是嵌套括号的 最大 数量。

输入：s = "(1+(2\*3)+((8)/4))+1"

输出：3

解释：数字 8 在嵌套的 3 层括号中。

|  |
| --- |
| class Solution:      def maxDepth(self, s: str) -> int:          stack = []          max\_ = 0          for i in s:              if i=="(":                  stack.append(i)                  current\_len = len(stack)                  max\_ = max(max\_, current\_len)              elif i==")":                  stack.pop()          return max\_ |

### [LCR 038. 每日温度](https://leetcode.cn/problems/iIQa4I/)，单调栈|----(↘)

请根据每日 气温 列表 temperatures ，重新生成一个列表，要求其对应位置的输出为：要想观测到更高的气温，至少需要等待的天数。如果气温在这之后都不会升高，请在该位置用 0 来代替。

**示例 1：**

**输入：**temperatures = [73,71,74,75,71,69,72,76,73]

**输出：**[2,1,1,4,2,1,1,0,0]

|  |
| --- |
| class Solution:      def dailyTemperatures(self, temperatures: List[int]) -> List[int]:          """          单调栈          """          len\_ = len(temperatures)          stack = [(0, temperatures[0])]  # (序号，温度)          res= [0]\*len\_          for i in range(1,len\_):              cur = temperatures[i]              if cur <= stack[-1][1]:                  stack.append((i, temperatures[i]))              else:  #  cur > 栈顶温度                  while stack and cur > stack[-1][1]:                      ind = stack.pop()[0]  #  序号                      res[ind] = i - ind  #  记录结果                  stack.append((i, temperatures[i]))          # print(stack, '??')          return res |

### [496. 下一个更大元素 I](https://leetcode.cn/problems/next-greater-element-i/)，单调栈|----(↘)

nums1 中数字 x 的 **下一个更大元素** 是指 x 在 nums2 中对应位置 **右侧** 的 **第一个** 比 x大的元素。

给你两个**没有重复元素** 的数组 nums1 和 nums2 ，下标从 **0** 开始计数，其中nums1 是 nums2 的子集。

对于每个 0 <= i < nums1.length ，找出满足 nums1[i] == nums2[j] 的下标 j ，并且在 nums2 确定 nums2[j] 的 **下一个更大元素** 。如果不存在下一个更大元素，返 -1 。

返回一个长度为 nums1.length 的数组ans作为答案，满足ans[i]是如上所述的 **下一个更大元素** 。

**输入：**nums1 = [4,1,2], nums2 = [1,3,4,2].

**输出：**[-1,3,-1]

**解释：**nums1 中每个值的下一个更大元素如下所述：

- 4 ，用加粗斜体标识，nums2 = [1,3,**4**,2]。不存在下一个更大元素，所以答案是 -1 。

- 1 ，用加粗斜体标识，nums2 = [***1***,3,4,2]。下一个更大元素是 3 。

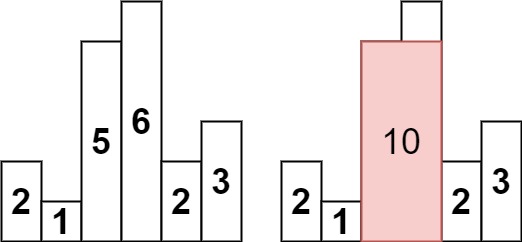
- 2 ，用加粗斜体标识，nums2 = [1,3,4,***2***]。不存在下一个更大元素，所以答案是 -1 。

|  |
| --- |
| # 暴力法： O(n \* m)  # 单调栈 + 哈希表： O(n + m)  class Solution:      def nextGreaterElement(self, nums1: List[int], nums2: List[int]) -> List[int]:          # Step 1: 单调栈预处理 nums2          stack = [nums2[0]]          mapping = {}          for num in nums2[1:]:              if num <= stack[-1]:                  stack.append(num)              else:                  while stack and num > stack[-1]:                      prev = stack.pop()                      mapping[prev] = num                  stack.append(num)          # 剩余在栈中的元素没有下一个更大元素          for num in stack:              mapping[num] = -1          # Step 2: 查询 nums1 对应的结果          res = [mapping[num] for num in nums1]          return res |

### [LCR 039. 柱状图中最大的矩形](https://leetcode.cn/problems/0ynMMM/)，|----(↗)

给定非负整数数组 heights ，数组中的数字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。



**输入：**heights = [2,1,5,6,2,3]

**输出：**10

**解释：**最大的矩形为图中红色区域，面积为 10

|  |
| --- |
| class Solution:        def largestRectangleArea(self, heights: List[int]) -> int:          """          使用单调栈方法找出柱状图中能组成的最大矩形面积            思路：          1. 维护一个单调递增栈 |----(↗)，存储的是柱子的索引          2. 当遇到比栈顶元素小的高度时，弹出栈顶并计算以该高度为高的矩形面积          3. 每个柱子只会入栈、出栈一次，整体时间复杂度为 O(n)          """          max\_area = 0          stack = []  # 栈中保存的是柱子的索引，按照高度单调递增          heights.append(0)  # 在末尾添加一个 0，确保所有柱子都会被弹出计算          for i, h in enumerate(heights):              if i==0 or h >= heights[stack[-1]]:                  stack.append(i)              else:                  while stack and h < heights[stack[-1]]:                      idx = stack.pop()         # 弹出栈顶柱子                      left = stack[-1] if stack else -1   # 左边第一个比它小的柱子                      right = i                             # 右边第一个比它小的柱子                        width = right - left - 1              # 宽度                      height = heights[idx]                      max\_area = max(max\_area, height \* width)                  stack.append(i)          return max\_area  方法2：双指针，[LCR 039. 柱状图中最大的矩形](#_LCR_039._柱状图中最大的矩形) |

## 队列-滑动窗口

涉及到窗口的都是区间连续

### [209. 长度最小的子数组](https://leetcode.cn/problems/minimum-size-subarray-sum/)  ★★

给定一个含有 n 个正整数的数组和一个正整数 target 。

找出该数组中满足其总和大于等于 target 的长度最小的 子数组 [numsl, numsl+1, ..., numsr-1, numsr] ，并返回其长度。如果不存在符合条件的子数组，返回 0 。

示例 1：

输入：target = 7, nums = [2,3,1,2,4,3]

输出：2

解释：子数组 [4,3] 是该条件下的长度最小的子数组。

示例 2：

输入：target = 11, nums = [1,1,1,1,1,1,1,1]

输出：0

|  |
| --- |
| class Solution:      def minSubArrayLen(self, target: int, nums: List[int]) -> int:          """          滑动窗口解 ，队列模拟，  sum\_ 是滑动窗口的和，只要满足要求就缩短窗口同时记录结果          """          from collections  import deque          q =deque()          sum\_ = 0 # 记录队列中元素sum          res\_len = float('inf')  # 记录结果          for i in nums:              q.append(i)              sum\_ += i              while sum\_ >= target:                  res\_len = min(res\_len, len(q))                    tmp = q.popleft()                  sum\_ -= tmp          return res\_len if res\_len != float('inf') else 0 |

### [3. 无重复字符的最长子串](https://leetcode.cn/problems/longest-substring-without-repeating-characters/description/)  ★★

给定一个字符串 s ，请你找出其中不含有重复字符的 最长 子串 的长度。

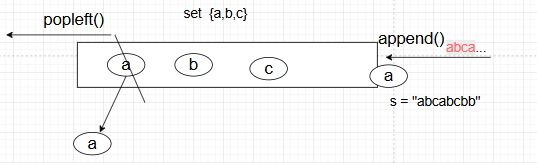
示例 1:

输入: s = "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。

|  |
| --- |
| class Solution:      def lengthOfLongestSubstring(self, s: str) -> int:          que = deque()  #  模拟窗口          set\_ = set()   #  查找窗口内是否有重复字符， 如果有不停弹出， 与que 保持同步          res = 0          for i in s:              if i in set\_:                  while i in set\_: #  不停弹出                      tmp = que.popleft()                      set\_.remove(tmp)                  #不在窗口，记录结果                  que.append(i)                  set\_.add(i)                  res = max(res, len(que))              else: #不在窗口，记录结果                  que.append(i)                  set\_.add(i)                  res = max(res, len(que))            return res |



图表 1队列解法，涉及到滑动窗口的均考虑deque

### [239. 滑动窗口最大值](https://leetcode.cn/problems/sliding-window-maximum/)

给你一个整数数组 nums，有一个大小为 k的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。

返回 *滑动窗口中的最大值*。

**示例 1：**

**输入：**nums = [1,3,-1,-3,5,3,6,7], 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**

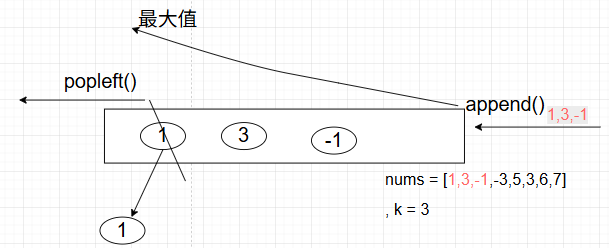
1 3 [-1 -3 5] 3 6 7  **5**

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**

|  |
| --- |
| class Solution:      def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:          from collections import deque  超时！！          res = []          queue =deque()          len\_ = len(nums)            for i in range(k):  #  赋初始值              queue.append(nums[i])          # print(queue, '???')          res.append(max(queue))          for i in nums[k:]:              queue.popleft()              queue.append(i)              res.append(max(queue)) # O(k)          return res  ------------------------------------------------------  优化： O(n)  class Solution:      def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:          if not nums or k == 0:              return []            dq = deque()  # 单调队列，维持dq[0] 最大          res = []          # 初始化第一个窗口（前k个元素）          for i in range(k):              current = nums[i]              # 维护单调递减队列, 只要目前值 > 尾部              while dq and current > nums[dq[-1]]:                  dq.pop()              dq.append(i)  # 初始值[3,-1]，参考示意图          res.append(nums[dq[0]]) #记录结果          #从第k个元素开始滑动窗口          for i in range(k, len(nums)):              current = nums[i]                # 最大值太远，紧接着会不在窗口内，需要先弹出              while dq and i-dq[0]+1 > k: # 隔的个数>窗口长度k                  dq.popleft()              # 维护单调递减队列              while dq and current >nums[dq[-1]]:                  dq.pop()              # 添加当前元素索引              dq.append(i)                res.append(nums[dq[0]]) #  记录结果          return res |



图表 2单调队列初始值赋值情况【1.3,-1】

实现2（使用大顶堆）：O(n log k)

|  |
| --- |
| class Solution:      def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:          import heapq          max\_heap = []  # 大顶堆, （使用负数模拟最大堆）          result = []          for i in range(k): # 形成第一个窗口，初值              heapq.heappush(max\_heap, (-nums[i], i))          result.append(-max\_heap[0][0])          for i in range(k, len(nums)):              heapq.heappush(max\_heap, (-nums[i], i))              # 移除堆顶不在窗口内的元素,              while max\_heap and  i-max\_heap[0][1] +1 > k:   # 注意这是 while                  heapq.heappop(max\_heap)                result.append(-max\_heap[0][0])          return result |

### [215. 数组中的第K个最大元素](https://leetcode.cn/problems/kth-largest-element-in-an-array/)

**输入:** [3,2,1,5,6,4], k = 2

**输出:** 5

|  |
| --- |
| class Solution:      def findKthLargest(self, nums: List[int], k: int) -> int:          # nums.sort()          # # print(nums)          # return nums[-k]          #  使用小顶堆          import heapq          min\_heap = []          for i in nums[:k]: # 建堆              heapq.heappush(min\_heap, i)          for j in nums[k:]:              if j>min\_heap[0]: # 加判断，略微优化                  heapq.heappush(min\_heap, j) # push, pop 顺序无所谓                  heapq.heappop(min\_heap)          return min\_heap[0] |

### [347. 前 K 个高频元素](https://leetcode.cn/problems/top-k-frequent-elements/)

给你一个整数数组 nums 和一个整数 k ，请你返回其中出现频率前 k 高的元素。你可以按 **任意顺序** 返回答案。

**示例 1:**

**输入:** nums = [1,1,1,2,2,3], k = 2

**输出:** [1,2]

|  |
| --- |
| class Solution:      def topKFrequent(self, nums: List[int], k: int) -> List[int]:          # 步骤1：使用 dict 统计频率          freq\_dict = {}          for num in nums:              freq\_dict[num] = freq\_dict.get(num, 0) +1          # # ----------------------------- 方法1，O(n log n)          # # 步骤2：将字典按值排序（降序） {1: 3, 2: 2, 3: 1}          sorted\_items = sorted(freq\_dict.items(), key=lambda x: -x[1])          # print(sorted\_items, '？？？')  # [(1, 3), (2, 2), (3, 1)] ？？？          # 步骤3：取前k个元素的键          return [item[0] for item in sorted\_items[:k]]          # ----------------------------- or方法2， O(n log k)          min\_heap = []  # 初始化一个空堆, 会按照元组的顺序依次比较元素大小。          for num, freq in list(freq\_dict.items())[:k]:  # 建堆, [(1, 3), (2, 2), (3, 1)]              heapq.heappush(min\_heap, (freq, num))          for num, freq in list(freq\_dict.items())[k:]:              heapq.heappush(min\_heap, (freq, num))              heapq.heappop(min\_heap)          return [num for \_, num in min\_heap] |

桶排序： O(n)

|  |
| --- |
| class Solution:      def topKFrequent(self, nums: List[int], k: int) -> List[int]:          print(freq\_dict, '!!')  #{1: 3, 2: 2, 3: 1} !!          # 步骤2：构建桶，个数为最大频率 ，下标表示频率          max\_freq = max(freq\_dict.values())          buckets = [[] for \_ in range(max\_freq + 1)]          for num, freq in freq\_dict.items():              buckets[freq].append(num)          print(buckets, '?')          # 步骤3：从高频率往低频率收集结果，注意桶里存的是元素          result = []          for i in range(max\_freq, -1, -1): # 从max\_freq到0（降序）              for num in buckets[i]:                  result.append(num)                  if len(result) == k:                      return result |

# BFS & DFS

深搜：栈/递归, 先相信子问题能解决，再构建当前问题的答案 。

广搜：队列

## 二叉树

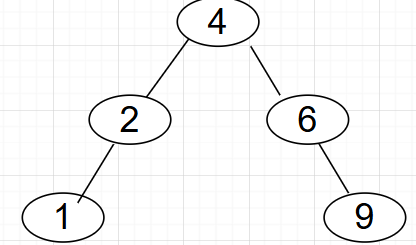
### [94. 二叉树的中序遍历](https://leetcode.cn/problems/binary-tree-inorder-traversal/)

深搜

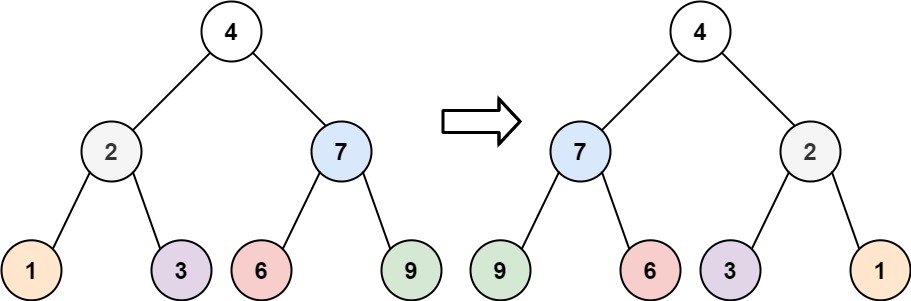
|  |
| --- |
| # Definition for a binary tree node.  # class TreeNode:  #     def \_\_init\_\_(self, val=0, left=None, right=None):  #         self.val = val  #         self.left = left  #         self.right = right  class Solution:      def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:          "左中右"          res= []          def dfs(node):              if not node: return              dfs(node.left)              res.append(node.val)              dfs(node.right)          dfs(root)          return res |

应用：**[98. 验证二叉搜索树](https://leetcode.cn/problems/validate-binary-search-tree/)**

BST 的中序遍历结果是一个严格递增的序列：[1,2,4,6,9]

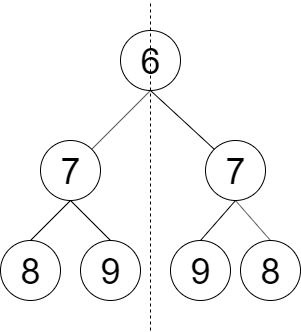


### [226. 翻转二叉树](https://leetcode.cn/problems/invert-binary-tree/)



|  |
| --- |
| class Solution:      def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:          def dfs(node):              if not node: return                node.left, node.right = node.right, node.left              dfs(node.left)              dfs(node.right)          dfs(root)          return  root |

### [LCR 145. 对称二叉树](https://leetcode.cn/problems/dui-cheng-de-er-cha-shu-lcof/) ★★

 对称

|  |
| --- |
| # Definition for a binary tree node.  # class TreeNode:  #     def \_\_init\_\_(self, val=0, left=None, right=None):  #         self.val = val  #         self.left = left  #         self.right = right  class Solution:      def checkSymmetricTree(self, root: Optional[TreeNode]) -> bool:          if not root: return True          def dfs(l,r): # 注意，这里是传入两个参数！              if not l and not r: return True              if not l and  r: return False              if l and not r: return False              if l.val != r.val:                  return False                res\_outside = dfs(l.left, r.right)              res\_inside = dfs(l.right, r.left)              return res\_outside and res\_inside            return dfs(root.left, root.right) |

### [110. 平衡二叉树](https://leetcode.cn/problems/balanced-binary-tree/)

平衡二叉树 是指该树所有节点的左右子树的高度相差不超过 1。

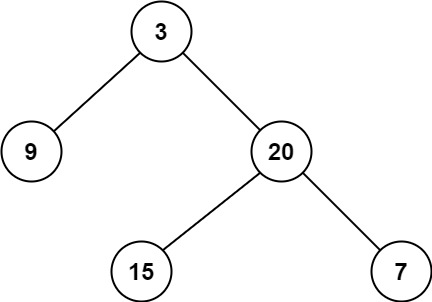
|  |
| --- |
| class Solution:      def isBalanced(self, root: Optional[TreeNode]) -> bool:          #           高度：   深度          #   O ---------4       1          #  / \          # O   O -------3       2          #      \          #       O ------2       3          #        \          #         O -----1     4          def dfs(node):          # 高度，使用后序遍历：左右中          # 如果左右子树不平衡， 返 -1              if not node: return 0              left\_height = dfs(node.left)              if left\_height==-1:                  return -1              right\_height = dfs(node.right)              if right\_height==-1:                  return -1                if abs(left\_height-right\_height)>1:                  return -1              else:                  return max(left\_height, right\_height) +1          return False if dfs(root)==-1 else True |

### [111. 二叉树的最小深度](https://leetcode.cn/problems/minimum-depth-of-binary-tree/)

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

说明：叶子节点是指没有子节点的节点。



**输入：**root = [3,9,20,null,null,15,7]

**输出：**2

|  |
| --- |
| # Definition for a binary tree node.  # class TreeNode:  #     def \_\_init\_\_(self, val=0, left=None, right=None):  #         self.val = val  #         self.left = left  #         self.right = right  class Solution:      def minDepth(self, root: Optional[TreeNode]) -> int:          "层序遍历"          if not root: return 0          q= deque([root])          layer = 0          while q:              layer += 1              for \_ in range(len(q)):                  tmp = q.popleft()                  if tmp and not tmp.left and not tmp.right: # 遍历到叶子节点，返回                      return layer                  if tmp and tmp.left: q.append(tmp.left)                  if tmp and tmp.right: q.append(tmp.right)    # ----------------------------          # 深度优先, 后序遍历            def get\_height(node):              if not node: return 0              # 递归获取左右子树的最小深度              l\_height = get\_height(node.left)  # 左              r\_height = get\_height(node.right)  # 右              # 中...              if not node.left and node.right:                  return 1+r\_height              if node.left and not node.right:                  return 1+l\_height              # 均不为空              return 1+min(l\_height, r\_height)          return get\_height(root) |

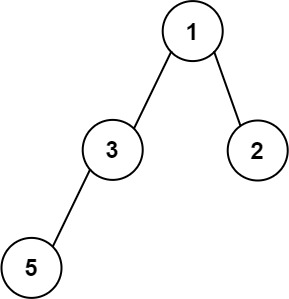
### [662. 二叉树最大宽度](https://leetcode.cn/problems/maximum-width-of-binary-tree/)

给你一棵二叉树的根节点 root ，返回树的 最大宽度 。

树的 最大宽度 是所有层中最大的 宽度 。

每一层的 宽度 被定义为该层最左和最右的非空节点（即，两个端点）之间的长度。将这个二叉树视作与满二叉树结构相同，两端点间会出现一些延伸到这一层的 null 节点，这些 null 节点也计入长度。

题目数据保证答案将会在 32 位 带符号整数范围内。



输入：root = [1,3,2,5]

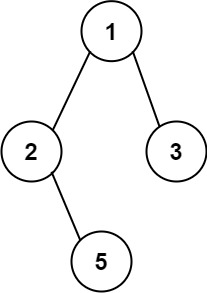
输出：2

解释：最大宽度出现在树的第 2 层，宽度为 2 (3,2) 。

|  |
| --- |
| # Definition for a binary tree node.  # class TreeNode:  #     def \_\_init\_\_(self, val=0, left=None, right=None):  #         self.val = val  #         self.left = left  #         self.right = right  from collections import deque  class Solution:      def widthOfBinaryTree(self, root: Optional[TreeNode]) -> int:          if not root: return 0            max\_width = 0          queue = deque( [(root, 0), ] )  # (node, index)          while queue:              level\_length = len(queue)              \_, first\_index = queue[0]  # 当前层第一个节点的索引              \_, last\_index = queue[-1]  # 当前层最后一个节点的索引              max\_width = max(max\_width, last\_index - first\_index + 1)              for \_ in range(level\_length):                  node, index = queue.popleft()                  if node.left:                      queue.append((node.left, 2 \* index + 1))                  if node.right:                      queue.append((node.right, 2 \* index + 2))          return max\_width |

## 路径问题

### [257. 二叉树的所有路径](https://leetcode.cn/problems/binary-tree-paths/)



**输入：**root = [1,2,3,null,5]

**输出：**["1->2->5","1->3"]

|  |
| --- |
| # Definition for a binary tree node.  class TreeNode:      def \_\_init\_\_(self, val=0, left=None, right=None):          self.val = val          self.left = left          self.right = right  class Solution:      def binaryTreePaths(self, root: Optional[TreeNode]) -> List[str]:          path = []          res = []          def dfs\_travel(node):              if not node: return              path.append(str(node.val))              if not node.left and not node.right:  # 为叶子节点                  res.append('->'.join(path))                  # return  注意这里不能加 return,会错， 将影响后文的 pop()回溯， 结合[LCR 080. 组合 ★★](#_LCR_080._组合) 一起研究              dfs\_travel(node.left)              dfs\_travel(node.right)                path.pop() #递归之后 pop          dfs\_travel(root)          return res |

### [LCR 080. 组合](https://leetcode.cn/problems/uUsW3B/" \t "_blank) ★★

**输入:** n = 4, k = 2

**输出:**

[

[2,4], [3,4],[2,3],

[1,2],[1,3],[1,4],

]

|  |
| --- |
| class Solution:      def combine(self, n: int, k: int) -> List[List[int]]:          path=[]          res=[]          def dfs(start\_index):              if (len(path)==k):                  res.append(path.copy())  # 注意是path 的当前状态的副本                  return #  注意这里return 并非关乎正确，而是加速（由于不影响pop）              for i in range(start\_index, n+1): # 遍历【1,2,3..,n】                  path.append(i)                  dfs(i+1)                  path.pop()          dfs(1)          return res |

### 图遍历，[200. 岛屿数量](https://leetcode.cn/problems/number-of-islands/" \t "_blank)  ★★★☆

给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

示例 1：

输入：grid = [

["1","1","1","1","0"],

["1","1","0","1","0"],

["1","1","0","0","0"],

["0","0","0","0","0"]

]

输出：1

|  |
| --- |
| class Solution:      def numIslands(self, grid: List[List[str]]) -> int:          rows = len(grid)          cols = len(grid[0])          count = 0          # ============================          def dfs(i, j):          # 如果当前位置 (i, j) 未超出网格边界， 且值是 '1'（陆地）。              if 0 <= i < rows and 0 <= j < cols and grid[i][j] == '1':                  grid[i][j] = '0'  # Mark as visited                  dfs(i + 1, j)                  dfs(i - 1, j)                  dfs(i, j + 1)                  dfs(i, j - 1)          # ============================          for i in range(rows):              for j in range(cols):                  if grid[i][j] == '1':                      dfs(i, j) #  深度搜索标记                      count += 1          return count  # --------------------------------------------------------------  # 广度优先实现  # --------------------------------------------------------------  from collections import deque  class Solution:      def numIslands(self, grid: List[List[str]]) -> int:          rows = len(grid)          cols = len(grid[0])          count = 0          queue = deque()            for i in range(rows):              for j in range(cols):                  if grid[i][j] == '1':                      # BFS, 循环遍历开始，queue 每次均为空deque([])                      queue.append((i, j))                      grid[i][j] = '0'  # Mark as visited                        while queue:                          x, y = queue.popleft()                          for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:                              nx, ny = x + dx, y + dy                              if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == '1':                                  grid[nx][ny] = '0'  # Mark as visited                                  queue.append((nx, ny))                      count += 1          return count |

### [130. 被围绕的区域](https://leetcode.cn/problems/surrounded-regions/)

给你一个 m x n 的矩阵 board ，由若干字符 'X' 和 'O' 组成，**捕获** 所有 **被围绕的区域**：

* **连接：**一个单元格与水平或垂直方向上相邻的单元格连接。
* **区域：连接所有**'O' 的单元格来形成一个区域。
* **围绕：**如果您可以用 'X' 单元格 **连接这个区域**，并且区域中没有任何单元格位于 board 边缘，则该区域被 'X' 单元格围绕。

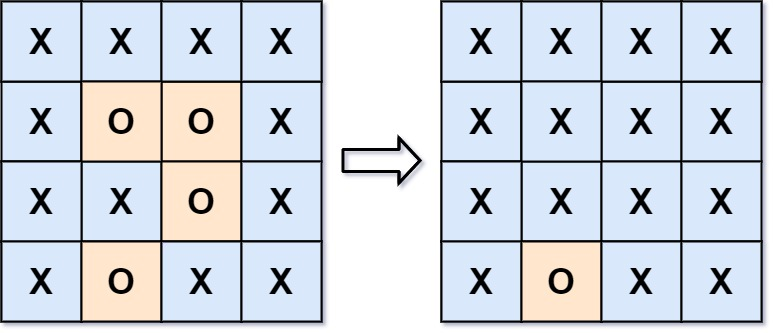
通过 **原地** 将输入矩阵中的所有 'O' 替换为 'X' 来 **捕获被围绕的区域**。你不需要返回任何值。

**示例 1：**

**输入：**board = [["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]]

**输出：**[["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","O","X","X"]]

**解释：**



在上图中，底部的区域没有被捕获，因为它在 board 的边缘并且不能被围绕。

|  |
| --- |
| class Solution:      def solve(self, board: List[List[str]]) -> None:          """          Do not return anything, modify board in-place instead.          """          # 思路：          # 1， 边界开始dfs搜索， 遇到相邻的o--> #, 保护起来          # 2, 结束后， 遍历 ， #-->o, o--> X          rows = len(board)          cols = len(board[0])          def dfs(i, j):              if 0 <= i < rows and 0 <= j < cols and board[i][j] == 'O':                  board[i][j] = '#'                  dfs(i-1, j)                  dfs(i, j-1)                  dfs(i+1, j)                  dfs(i, j+1)          for i in range(rows):              for j in [0, cols-1]:                      dfs(i, j)          for i in [0, rows-1]:              for j in range(cols):                  dfs(i, j)            for i in range(rows):              for j in range(cols):                  if board[i][j] == '#':                      board[i][j] = 'O'                  else:                      board[i][j] = 'X' |

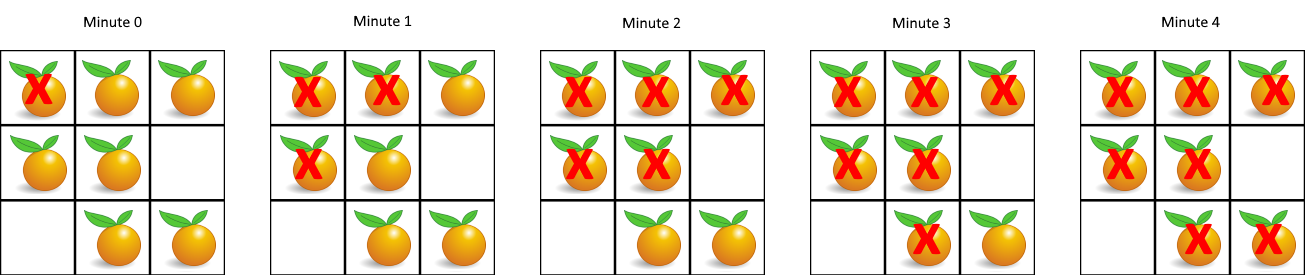
### [994. 腐烂的橘子](https://leetcode.cn/problems/rotting-oranges/) ★★★☆

在给定的 m x n 网格 grid 中，每个单元格可以有以下三个值之一：

值 0 代表空单元格；值 1 代表新鲜橘子；值 2 代表腐烂的橘子。

每分钟，腐烂的橘子 周围 4 个方向上相邻 的新鲜橘子都会腐烂。

返回 直到单元格中没有新鲜橘子为止所必须经过的最小分钟数。如果不可能，返回 -1 。



输入：grid = [[2,1,1],[1,1,0],[0,1,1]]

输出：4

示例 2：

输入：grid = [[2,1,1],[0,1,1],[1,0,1]]

输出：-1

解释：左下角的橘子（第 2 行， 第 0 列）永远不会腐烂，因为腐烂只会发生在 4 个方向上。

示例 3：

输入：grid = [[0,2]]

输出：0

解释：因为 0 分钟时已经没有新鲜橘子了，所以答案就是 0 。

|  |
| --- |
| # 本题，三种返回情况： -1，0以及>0  class Solution:      def orangesRotting(self, grid: List[List[int]]) -> int:          # 获取行数和列数          m, n = len(grid), len(grid[0])          # 初始化队列和新鲜橘子计数器          queue = deque()          fresh = 0            # 第一次遍历：统计新鲜橘子和初始腐烂橘子位置          for i in range(m):              for j in range(n):                  if grid[i][j] == 2:                      queue.append((i, j))  # 把腐烂橘子的位置加入队列                  elif grid[i][j] == 1:                      fresh += 1          if fresh == 0: # 如果没有新鲜橘子，直接返回0              return 0          # 四个方向：上、下、左、右          directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]          time = 0  # 时间从0开始          # 开始 BFS          while queue and fresh > 0:              len\_ = len(queue)  # # 当前这一分钟要处理的腐烂橘子数量, for 循环结束， time +1，体现出在不同位置的腐烂并发性，这里有一个动态的过程，类似层序遍历                for \_ in range(len\_): # 处理这一批次的腐烂                  x, y = queue.popleft()  # 取出当前腐烂橘子的位置                  for dx, dy in directions: # 向四个方向扩散                      nx, ny = x + dx, y + dy                      # 判断是否在边界内，且是一个新鲜橘子                      if 0 <= nx < m and 0 <= ny < n and grid[nx][ny] == 1:                          grid[nx][ny] = 2  # 被感染，变成腐烂橘子                          fresh -= 1        # 新鲜橘子减少                          queue.append((nx, ny)) # 加入队列，下一分钟继续传播              time += 1  # 每处理完一层，时间+1          # 最后判断是否全部感染完成          return time if fresh == 0 else -1 |

类似： algorithm-lab/huawei/test1.py [流浪地球发动机启动问题](https://github.com/sunnysky29/algorithm-lab/blob/main/huawei/test1.py)

# 动态规划

## 674. 最长连续递增序列

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

输入：nums = [1,3,5,4,7]

输出：3

解释：最长连续递增序列是 [1,3,5], 长度为3。

尽管 [1,3,5,7] 也是升序的子序列, 但它不是连续的，因为 5 和 7 在原数组里被 4 隔开。

|  |
| --- |
| class Solution:      def findLengthOfLCIS(self, nums: List[int]) -> int:          # dp[i] = dp[i-1] +1 or 1          len\_ = len(nums)          dp = [1]\* (len\_+1)          for i in range(1, len\_):              if nums[i]> nums[i-1]:                  dp[i] = dp[i-1] +1          return max(dp) |

## [121. 买卖股票的最佳时机](https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/)  ★★

给定一个数组 prices ，它的第 i 个元素 prices[i] 表示一支给定股票第 i 天的价格。

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

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 0 。

示例 1：

输入：[7,1,5,3,6,4]

输出：5

解释：在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。

注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。

|  |
| --- |
| class Solution:      def maxProfit(self, prices: List[int]) -> int:          """          121题：只能交易一次，找最大单次利润。122题：可以交易多次，只要今天比昨天贵就卖出（贪心）。          dp[i] 的本质：前 i 天中，一次买卖的最大利润。          dp[i] = max(dp[i-1], prices[i]-minprice)          """          len\_ = len(prices)          min\_ = prices[0]          dp = [0]\*len\_          for i in range(1,len(prices)):              dp[i] = max(dp[i-1], prices[i]-min\_)              if prices[i]<min\_:                  min\_ = prices[i]          return max(dp) |

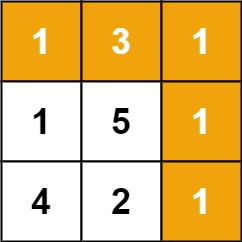
相关：[122. 买卖股票的最佳时机 II](https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/)（贪心）

## [64. 最小路径和](https://leetcode.cn/problems/minimum-path-sum/)  ★★

给定一个包含非负整数的 *m* x *n* 网格 grid ，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。

**说明：**每次只能向下或者向右移动一步。

**示例 1：**



**输入：**grid = [[1,3,1],[1,5,1],[4,2,1]]

**输出：**7

**解释：**因为路径 1→3→1→1→1 的总和最小。

|  |
| --- |
| class Solution:      def minPathSum(self, grid: List[List[int]]) -> int:          """  状态转移方程: dp[i][j]（ 到[i][j]的最小路径和） = min(dp[i-1][j],dp[i][j-1]) + grid[i][j]  ，          """          dp = grid          rows = len(grid)          colmns = len(grid[0])          for i in range(rows):              for j in range(colmns):                  if i+j==0: continue                  if i==0:                      dp[i][j] =dp[0][j-1] + grid[i][j]                  elif j==0:                      dp[i][j] =dp[i-1][j] + grid[i][j]                  else:                      dp[i][j] =  grid[i][j] + \                      min(grid[i-1][j], grid[i][j-1])          return dp[-1][-1] |

## [53. 最大子数组和](https://leetcode.cn/problems/maximum-subarray/)  ★★

给你一个整数数组 nums ，请你找出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。

子数组是数组中的一个连续部分。

示例 1：

输入：nums = [-2,1,-3,4,-1,2,1,-5,4]

输出：6

解释：连续子数组 [4,-1,2,1] 的和最大，为 6

|  |
| --- |
| class Solution:      def maxSubArray(self, nums: List[int]) -> int:          '''          注意 dp[i]含义：以nums[i]结尾的连续子数组最大和，          那么，dp[i]取最大值只有2情况：          dp[i] = max(nums[i], dp[i-1]+nums[i])          '''  len\_ = len(nums)          dp = [0]\* len\_          dp[0] = nums[0]          for i in range(1, len\_):              dp[i] = max(nums[i], dp[i-1]+nums[i])          return max(dp) |

# 贪心

## [122. 买卖股票的最佳时机 II](https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/)

给你一个整数数组 prices ，其中 prices[i] 表示某支股票第 i 天的价格。

在每一天，你可以决定是否购买和/或出售股票。你在任何时候 **最多** 只能持有 **一股** 股票。你也可以先购买，然后在 **同一天** 出售。

返回 *你能获得的****最大****利润* 。

**输入：**prices = [7,1,5,3,6,4]

**输出：**7

**解释：**在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出, 这笔交易所能获得利润 = 5 - 1 = 4。

随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出, 这笔交易所能获得利润 = 6 - 3 = 3。

最大总利润为 4 + 3 = 7 。

|  |
| --- |
| class Solution:      def maxProfit(self, prices: List[int]) -> int:          pass          """          比较相邻两天的价格，只要升值就交易          121题：只能交易一次，找最大单次利润。  122题：可以交易多次，只要今天比昨天贵就卖出（贪心）。          """          res = 0          len\_ = len(prices)          for i in range(1, len\_):              if prices[i]>prices[i-1]:                  res += prices[i]-prices[i-1]          return res |

## [435. 无重叠区间](https://leetcode.cn/problems/non-overlapping-intervals/)

给定一个区间的集合 intervals ，其中

 intervals[i] = [starti, endi] 。返回 *需要移除区间的最小数量，使剩余区间互不重叠*。

**注意** 只在一点上接触的区间是 **不重叠的**。例如 [1, 2] 和 [2, 3] 是不重叠的。

**输入:** intervals = [[1,2],[2,3],[3,4],[1,3]]

**输出:** 1

**解释:** 移除 [1,3] 后，剩下的区间没有重叠。

|  |
| --- |
| class Solution:      def eraseOverlapIntervals(self, intervals: list[list[int]]) -> int:          """          使用贪心策略结合栈来找出最多能保留多少个不重叠区间，          排序依据为区间的左边界（起始点）          局部最优：每次遇到重叠区间时，只比较当前区间和栈顶区间的结束时间，选择结束更早的（局部最优解）。          全局最优：通过局部最优的累积，最终得到最多的不重叠区间（全局最优解）。          """          if not intervals:              return 0          intervals.sort(key=lambda x: x[0])  #按左边界（起始点）从小到大排序， 排序后下面只与栈顶的比较          stack = [] # 栈里保存不重叠的          stack.append(intervals[0])          for i in range(1, len(intervals)):              current\_start, current\_end = intervals[i]              last\_start, last\_end = stack[-1]              if current\_start >= last\_end:   # 不重叠，保留当前区间                  stack.append(intervals[i])              else:  # 重叠，保留结束时间较早的区间                  if current\_end < last\_end:                      stack.pop()                      stack.append(intervals[i])          # 总数 - 栈的大小 = 要删除的最小区间数          return len(intervals) - len(stack) |

# 其它

## [470. 用 Rand7() 实现 Rand10()](https://leetcode.cn/problems/implement-rand10-using-rand7/) ★★

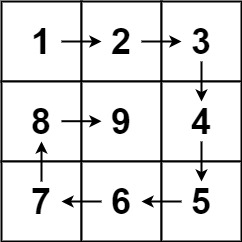
给定方法 rand7 可生成 [1,7] 范围内的均匀随机整数，试写一个方法 rand10 生成 [1,10] 范围内的均匀随机整数。

你只能调用 rand7() 且不能调用其他方法。请不要使用系统的 Math.random() 方法。

|  |
| --- |
| # The rand7() API is already defined for you.  # def rand7():  # @return a random integer in the range 1 to 7  class Solution:      def \_\_get\_rand49(self):          # 生成均匀的 [1,49]          num = (rand7() - 1) \* 7 + rand7()  # 生成1-49的随机数          return num      def rand10(self):          """          :rtype: int          概率拒绝采样，注意均匀          """          num = self.\_\_get\_rand49()          while num>40:              num = self.\_\_get\_rand49()          return num%10 +1 |

## [59. 螺旋矩阵 II](https://leetcode.cn/problems/spiral-matrix-ii/)

给你一个正整数 n ，生成一个包含 1 到 n2 所有元素，且元素按顺时针顺序螺旋排列的 n x n 正方形矩阵 matrix 。



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

|  |
| --- |
| class Solution:      def generateMatrix(self, n: int) -> List[List[int]]:          # 初始化 n x n 的矩阵          matrix = [[0] \* n for \_ in range(n)]          num = 1  # 当前填充的数字            # 定义当前处理的行的起始和结束，列的起始和结束          row\_start, row\_end = 0, n - 1          col\_start, col\_end = 0, n - 1            #  这个代码好理解，直接模拟循环过程，同时每次循环完都执行减缩操作          while row\_start <= row\_end and col\_start <= col\_end:              print(f'-'\*10)              # 从左到右填充当前行的最上方一行， 123              for i in range(col\_start, col\_end + 1):                  print(f'row\_start, i: {row\_start}-{i} 111')                  matrix[row\_start][i] = num                  num += 1              row\_start += 1  # 当前最上方一行处理完毕，向下移动一行              # 从上到下填充当前列的最右侧一列， 45              for i in range(row\_start, row\_end + 1):                  print(f'i, col\_end, i: {i}-{col\_end} 222')                  matrix[i][col\_end] = num                  num += 1              col\_end -= 1  # 当前最右侧一列处理完毕，向左移动一列              # 从右到左填充当前行的最下方一行， 67              for i in range(col\_end, col\_start - 1, -1):                  print(f'row\_end,i: {row\_end}-{i} 333')                  matrix[row\_end][i] = num                  num += 1              row\_end -= 1  # 当前最下方一行处理完毕，向上移动一行              # 从下到上填充当前列的最左侧一列， 8              for i in range(row\_end, row\_start - 1, -1):                  print(f'i,col\_start: {i}-{col\_start} 444')                  matrix[i][col\_start] = num                  num += 1              col\_start += 1  # 当前最左侧一列处理完毕，向右移动一列          return matrix |

## [240. 搜索二维矩阵 II](https://leetcode.cn/problems/search-a-2d-matrix-ii/)  ★

编写一高效的算法来搜索 m x n 矩阵 matrix 中的一个目标值 target 。该矩阵具有以下特性：

每行的元素从左到右升序排列。

每列的元素从上到下升序排列。

示例 1：

**输入：**matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]], target = 5

**输出：**true

|  |
| --- |
| class Solution:      def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:          """          从左下角开始，每次搜索时候，减去一行或一列          """          rows= len(matrix)          columns = len(matrix[0])          start\_row = rows-1          start\_column = 0          while start\_row>=0 and start\_column<columns:              cur\_val = matrix[start\_row][start\_column]              if cur\_val == target:                  return True              elif cur\_val < target:                  start\_column += 1              else: # 删行                  start\_row -= 1          return False |