[1 双指针 3](#_Toc201563642)

[1.1 三数之和 3](#_Toc201563643)

[1.2 189. 轮转数组 4](#_Toc201563644)

[1.3 206. 反转链表 5](#_Toc201563645)

[1.4 19. 删除链表的倒数第 N 个结点 6](#_Toc201563646)

[1.5 203. 移除链表元素 7](#_Toc201563647)

[1.6 27. 移除元素， 原地★★ 8](#_Toc201563648)

[1.7 26. 删除有序数组中的重复项 8](#_Toc201563649)

[1.8 977. 有序数组的平方 9](#_Toc201563650)

[1.9 3. 无重复字符的最长子串  ★★★ 10](#_Toc201563651)

[1.10 209. 长度最小的子数组  ★★★ 10](#_Toc201563652)

[1.11 5. 最长回文子串 ★★★ 11](#_Toc201563653)

[2 动态规划 12](#_Toc201563654)

[2.1 674. 最长连续递增序列 12](#_Toc201563655)

[2.2 121. 买卖股票的最佳时机  ★★ 13](#_Toc201563656)

[2.3 122. 买卖股票的最佳时机 II 14](#_Toc201563657)

[2.4 64. 最小路径和  ★★ 14](#_Toc201563658)

[2.5 53. 最大子数组和  ★★ 15](#_Toc201563659)

[3 二分查找 16](#_Toc201563660)

[3.1 704. 二分查找  ★ 16](#_Toc201563661)

[3.2 240. 搜索二维矩阵 II  ★ 17](#_Toc201563662)

[4 字符串 18](#_Toc201563663)

[4.1 14. 最长公共前缀 ★★ 18](#_Toc201563664)

[5 图论 19](#_Toc201563665)

[5.1 回溯， LCR 080. 组合 ★★ 19](#_Toc201563666)

[5.2 图遍历，200. 岛屿数量  ★★★☆ 20](#_Toc201563667)

[5.3 994. 腐烂的橘子 ★★★☆ 21](#_Toc201563668)

[6 栈 23](#_Toc201563669)

[6.1 20. 有效的括号 ★★ 23](#_Toc201563670)

[6.2 1614. 括号的最大嵌套深度 24](#_Toc201563671)

[7 二叉树 25](#_Toc201563672)

[7.1 LCR 145. 判断对称二叉树 ★★ 25](#_Toc201563673)

[7.2 662. 二叉树最大宽度 26](#_Toc201563674)

[8 其它 27](#_Toc201563675)

[8.1 470. 用 Rand7() 实现 Rand10() ★★ 27](#_Toc201563676)

v1.1—2025.0623

典型写法：

|  |
| --- |
| [[0] \* 3 for \_ in range(5)]  float('inf') float('-inf')  math.sqrt()  ord(char)  #  字符转ascii  dic.get('key', 0) # 设置字典默认值  list.sort()  #  list 会改， reverse=True  a = sorted(b)  # 有返回值，不改变原先  ' '.join(numbers)  #  numbers必须为 str 列表  map(int/str, list)  #  列表元素转换    list.pop(index)  # 默认弹出最后一个  from collections import deque  deque.popleft()  #  O(1)    copy.deepcopy(list)  # 深拷贝 |

# 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      """      def \_\_twosum(self, nums, target):          """          双指针法， 求给定target 的数组元素          """          # print(f'{target}, nums: {nums} !!')          l = 0          r= len(nums)-1          res = []          while l< r:              # print(l, r, '--')              sum\_ = nums[l] + nums[r] #  注意写在while 内              if sum\_ == target:                  res.append([nums[l] , nums[r]])  # 去重，注意先写 l<r                  while l<r and nums[l]==nums[l+1]:                      l += 1                  while l < r and nums[r] == nums[r - 1]:                      r -= 1                  # 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 = []          for ind, val in enumerate(nums):              #  去重              if ind >0  and nums[ind-1] == nums[ind]: continue              target = 0 - val              res\_ = self.\_\_twosum(nums[ind+1:], target)              if res\_:                  # print(val, res\_, "???")                  for i in res\_:                      # print(i, '^^^') #  列表                      res.append([val]+i) # 构造新列表[val]+i， 用于返回                      # res.append(i.append(val)) # 写法错误！！list.append() 是原地操作，返回 None，不能直接用于 res.append()。          # print(res, "!!!!!!!")          return res |

## 189. 轮转数组

给定一个整数数组 nums，将数组中的元素向右轮转 k 个位置，其中 k 是非负数。

原地实现

示例 1:

输入: nums = [1,2,3,4,5,6,7], k = 3

输出: [5,6,7,1,2,3,4]

解释:

向右轮转 1 步: [7,1,2,3,4,5,6]

向右轮转 2 步: [6,7,1,2,3,4,5]

向右轮转 3 步: [5,6,7,1,2,3,4]

|  |
| --- |
| class Solution:      def \_\_rotate\_all(self, my\_list,  left, right):          # 原地交换（双指针）          while left < right:              my\_list[left], my\_list[right] = my\_list[right], my\_list[left]  # 交换              left += 1              right -= 1      def rotate(self, nums: List[int], k: int) -> None:          """          Do not return anything, modify nums in-place instead.            先实现所有元素反转,          注意，这里不能使用切片，切片不修改原列表          """          n = len(nums)          k %= n  # 处理 k > n 的情况          self.\_\_rotate\_all(nums, 0, n - 1)  # 全部反转          self.\_\_rotate\_all(nums, 0, k - 1)  # 前 k 个反转          self.\_\_rotate\_all(nums, k, n - 1)  # 剩余部分反转 |

## [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 = cur, cur\_next            return prev  # 最后 prev 是新的头节点 |

方法2： 使用哨兵节点

|  |
| --- |
| class Solution:      def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:          """          假设输入链表为：1 → 2 → 3 → None          执行过程如下：          guard → None          插入 1：guard → 1 → None          插入 2：guard → 2 → 1 → None          插入 3：guard → 3 → 2 → 1 → None          """          guard = ListNode()            while head:              temp = head.next    # 保存下一个节点                head.next = guard.next              guard.next = head                head = temp            # 移动到下一个节点            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

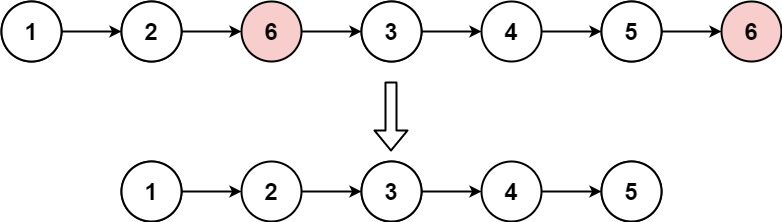
**输出：**[1,2,3,5]

|  |
| --- |
| # 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()  # 哨兵节点，注意使用          guard.next = head          f = guard          s = guard          step = 0          while step < n:              f  =f.next              step += 1            while f.next: #  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          # print(s.val, '????')          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 #  确保f 始终比s快1          while f:              if f.val==val:                  f= f.next                  s.next = f              else:                  f= f.next                  s= s.next          return guard.next |

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

**示例 1：**

**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）。

|  |
| --- |
| class Solution:      def removeElement(self, nums: List[int], val: int) -> int:          # 快慢指针，需要不停地赋值          # 类似解法：26. 删除有序数组中的重复项，  https://leetcode.cn/problems/remove-duplicates-from-sorted-array/submissions/632753488/          f =s= 0  # 快慢指针          len\_ = len(nums)          while f < len\_:              # slow 用来收集最终结果，如果 fast 对应值不等于 val，则替换slow              if  nums[f]==val:                  f += 1              else:                  nums[s] = nums[f] # 先赋值再更新                  s += 1                  f += 1          return s |

## [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。

**示例 2：**

**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。

|  |
| --- |
| class Solution:      def removeDuplicates(self, nums: List[int]) -> int:          # 快慢指针, 与27. 移除元素， 类似          s= f=0          len\_ = len(nums)            while f < len\_:              if nums[s] == nums[f]:                  f +=1              else:                  s +=1                  nums[s] = nums[f]                  f += 1          return s+1 |

## [977. 有序数组的平方](https://leetcode.cn/problems/squares-of-a-sorted-array/)

给你一个按 **非递减顺序** 排序的整数数组 nums，返回 **每个数字的平方** 组成的新数组，要求也按 **非递减顺序** 排序。

 时间复杂度为 O(n)

**示例 1：**

**输入：**nums = [-4,-1,0,3,10]

**输出：**[0,1,9,16,100]

**解释：**平方后，数组变为 [16,1,0,9,100]

排序后，数组变为 [0,1,9,16,100]

|  |
| --- |
| class Solution:      def sortedSquares(self, nums: List[int]) -> List[int]:          """          求解思路，使用双指针，另外开辟数组res          左右指针比较，平方值谁大，就填到新的数组里          同时更新 i          """          l =0          r = len(nums) -1          i = len(nums) -1  #  新数组res 要填进去数值的序号          res= nums.copy()            while l <= r :              if abs(nums[l]) > nums[r]:                  res[i] = nums[l] \*\* 2                  l += 1              else:                  res[i] = nums[r] \*\* 2                  r -= 1              i -= 1  #  res 待补充值的数组元素序号          return res |

## [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:          """          滑动窗口位置： start----end          当dict 中命中时候，更新窗口的起始位置          # """          start = 0          dict\_str= {}  #  记录字符最新出现位置          res = 0  #  最大长度          for end,val in enumerate(s):              #  and dict\_str[val] >= start:              # --> 含义，上一个重复的字符在窗口内              # tmqmzuxt, 比如这的 qm的m有效, uxt 的t 无效              #（因为窗口起始位置已经改变，窗口已经不包含该重复字符了）              if val in dict\_str and dict\_str[val]>=start:                  start = dict\_str[val] +1 #  窗口起始位置更新                res = max(res, end-start+1)              dict\_str[val] = end          return res |

## [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 = 4, nums = [1,4,4]

输出：1

示例 3：

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, 注意遍历是 end          sum\_ 是滑动窗口的核心变量，动态维护当前窗口的和。          """          sum\_ = 0 # 当前窗口的sum和,  下标为[s,...., end]          start = 0  # 窗口左端点          res = float('inf') # 记录最小长度， 初始化最小长度为无穷大            # 遍历 窗口右端点          for end in range(len(nums)):              sum\_ += nums[end]              #  如果sum\_ > target, 窗口左端点右移动,同时更新sum\_              while sum\_ >= target:                  res= min(res, end- start +1)                  # 移动 s                  sum\_ -= nums[start]                  start += 1                  # print(f'res: {res} ??')          return res if res != float('inf') else 0  # 如果没找到，返回 0 |

## [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              Palindrome\_string= new\_s[l+1:r].replace(special\_s, "")  #当前的最大回文串              if len(Palindrome\_string) > len(res\_s):                  res\_s = Palindrome\_string          return res\_s |

# 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。

示例 1：

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。

示例 2：

输入：nums = [2,2,2,2,2]

输出：1

解释：最长连续递增序列是 [2], 长度为1。

|  |
| --- |
| 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/)

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* 。

**示例 1：**

**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 。

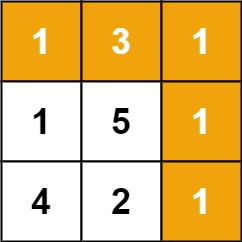
|  |
| --- |
| class Solution:      def maxProfit(self, prices: List[int]) -> int:          pass          """          比较相邻两天的价格，只要升值就交易          121题：只能交易一次，找最大单次利润。  122题：可以交易多次，只要今天比昨天贵就卖出（贪心）。          """          res = 0          len\_ = len(prices)          for i in range(1, len\_):          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

## [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) |

# 二分查找

## [704. 二分查找](https://leetcode.cn/problems/binary-search/)  ★

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

你必须编写一个具有 O(log n) 时间复杂度的算法。

示例 1:

输入: nums = [-1,0,3,5,9,12], target = 9

输出: 4

解释: 9 出现在 nums 中并且下标为 4

|  |
| --- |
| class Solution:      def search(self, nums: List[int], target: int) -> int:          l = 0          r = len(nums)-1          while (l<= r):              mid = l + (r-l) // 2  # // 返回整数商,这里需要正确计算中间索引              if target == nums[mid]:                  return mid              elif target <nums[mid]:                  r = mid-1              else:                  l = mid +1          return -1 |

## [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 |

# 字符串

## [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:      """       关键是要判断\_\_judge\_index0， 是否为首字包含      [flow, flower, flight]      选择基准元素base: flow          1, flower--->  ?0, true， 继续下一个判断          2， flight              flow  ?0, false  , base\_strs = base\_strs[:-1]              flo   ?0, false  , base\_strs = base\_strs[:-1]              fl    ?0, true  --->  继续下一个判断      return  fl      """      def \_\_judge\_index0(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:          base\_strs = strs[0]          for str\_long in strs:              #  只要不是包含关系              while not self.\_\_judge\_index0(str\_long, base\_strs):              # if self.\_\_judge\_index0(str\_long, base\_strs):              #     continue              # else:                  base\_strs = base\_strs[:-1]          return base\_strs |

# 图论

注意 回溯 vs 图遍历 时候，for 循环的位置

| **范式** | **组合问题** | **岛屿问题** |
| --- | --- | --- |
| **算法类型** | 回溯（Backtracking） | 图遍历（Graph Traversal） |
| **状态转移** | 显式路径（**path**列表） | 隐式图（网格修改） |
| **终止条件** | 路径长度达到**k** | 越界或遇到非陆地 |
| **循环作用** | 分支枚举（横向扩展） | 入口点扫描（初始化DFS） |

* **回溯（组合问题）**：
  + **内部循环** + **递归** = 系统地枚举所有解空间。
  + 类似问题：排列、子集生成、N皇后。
* **图遍历（岛屿问题）**：
  + **外部循环** + **DFS/BFS** = 找到所有连通分量。
  + 类似问题：迷宫寻路、洪水填充、图连通性检测。

## 回溯， [LCR 080. 组合](https://leetcode.cn/problems/uUsW3B/) ★★

**示例 1：**

**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):              # print(f'递归start\_ind: {start\_index}')              if (len(path)==k):                  res.append(path.copy())  # 注意是path 的当前状态的副本                  return                for i in range(start\_index, n+1):                  # 遍历【1,2,3..,n】                  path.append(i)                  # print(path,'<-----')                  dfs(i+1)                  path.pop()                  # print(path,'----->')          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:          # if not grid:          #     return 0            rows = len(grid)          cols = len(grid[0])          count = 0          # ============================          def dfs(i, j):              #  dfs 终止条件：  #         如果当前位置 (i, j) 超出网格边界（即 i 或 j 为负数，或者 i >= rows 或 j >= cols），则直接返回。  #         如果当前位置的值不是 '1'（即不是陆地），也直接返回。              if i < 0 or j < 0 or i >= rows or j >= cols or grid[i][j] == '0': return              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:          if not grid:              return 0            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                      queue.append((i, j))                      grid[i][j] = '0'  # Mark as visited                        while queue:                          # 循环的主要目的就是 处理并标记当前岛屿的所有相连陆地                          x, y = queue.popleft()                          # Check four directions                          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 |

## [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：

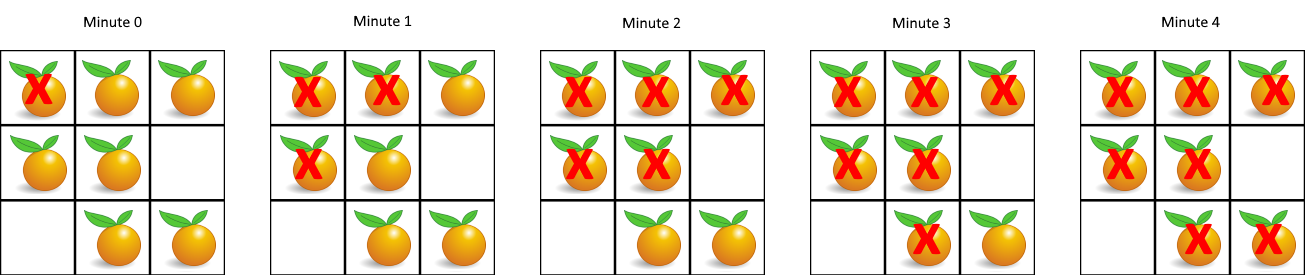
值 0 代表空单元格；

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 。

|  |
| --- |
| from collections import deque  from typing import List  class Solution:      def orangesRotting(self, grid: List[List[int]]) -> int:          """          计算所有新鲜橘子腐烂所需的最小分钟数            参数:              grid: 二维列表，表示橘子网格状态                  0 - 空单元格                  1 - 新鲜橘子                  2 - 腐烂橘子            返回:              int: 所有橘子腐烂所需分钟数，如果不可能则返回-1          """  # from collections import deque          # 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            # 新鲜橘子数量加1          # 如果没有新鲜橘子，直接返回0          if fresh == 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:              level\_size = 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                for \_ in range(level\_size): # 处理这一批次的腐烂                  x, y = queue.popleft()  # 取出当前腐烂橘子的位置                    # 向四个方向扩散                  for dx, dy in directions:                      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 |

# 栈

## [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。

有效字符串需满足：

左括号必须用相同类型的右括号闭合。

左括号必须以正确的顺序闭合。

每个右括号都有一个对应的相同类型的左括号。

示例 1：

输入：s = "()"

输出：true

示例 2：

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 |

## [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。

示例 1：

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。

**示例 2：**

**输入：**s = "(1)+((2))+(((**3**)))"

**输出：**3

**解释：**数字 3 在嵌套的 3 层括号中。

**示例 3：**

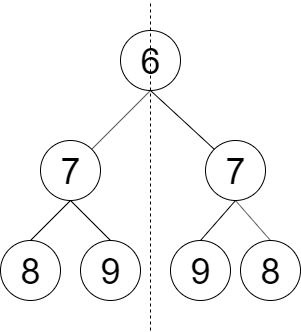
**输入：**s = "()(())((()()))"

**输出：**3

|  |
| --- |
| 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\_ |

# 二叉树

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



|  |
| --- |
| # 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:          def recur(L, R):              if not L and not R: return True              if not L or not R or L.val != R.val: return False              return recur(L.left, R.right) and recur(L.right, R.left)          return recur(root.left, root.right) if root else True |

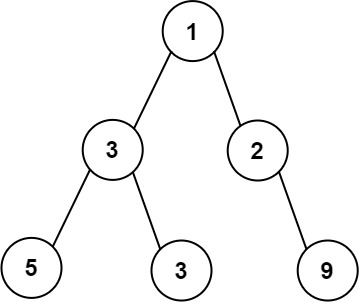
## [662. 二叉树最大宽度](https://leetcode.cn/problems/maximum-width-of-binary-tree/)

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。

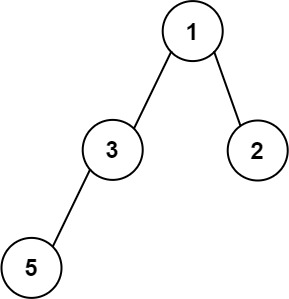
题目数据保证答案将会在 32 位 带符号整数范围内。



**输入：**root = [1,3,2,5,3,null,9]

**输出：**4

**解释：**最大宽度出现在树的第 3 层，宽度为 4 (5,3,null,9) 。



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  class SpecialTreeNode:      def \_\_init\_\_(self, val='#', 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 |

# 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 |