0.栈总结.md

- 构造问题:栈&队列
  - [ ] 155 最小栈：minstack保留和栈相同长度的最小元素
  - [ ] 225 用队列实现栈：队列遍历不改变顺序push=append nwe +popleft others
  - [ ] 232 用栈实现队列：栈会改变顺序需要两个存储stackin, stackout，在pop的时候reverse
- 两两匹配问题
  - [ ] 20 有效的括号：左边括号入栈，右边括号出栈
  - [ ] 32 最长有效括号：通过右括号位置隔断每个有效括号子串
  - [ ] 150 逆波兰表达式求值：数值入栈，符号出栈，
  - [ ] 394 字符串解码: ']'出栈，isdigit停止出栈
  - [ ] 1047 删除字符串中的所有相邻重复项：不重复入栈，重复出栈
- DFS搜索
  - [ ] 130 被围绕的区域：定位所有边界O，DFS寻找所有相连O
  - [ ] 200 岛屿的数量：DFS深度搜索
  - [ ] 733 图像渲染: 注意通过修改当前值避免循环递归
- 单调/优先，栈/队列
  - [ ] 42 接雨水：单调栈，需要左中右三个元素来计算雨水
  - [ ] 215 数组中的第K个最大元素：heapq小根堆排序
  - [ ] 239 滑动窗口最大值：双向单调队列，队列内单调递减，pop head，push tail
  - [ ] 496 下一个更大元素 I：单调栈，哈希存储
  - [ ] 503 下一个更大元素 II：单调栈， 数组存储（数组有重复），循环用遍历2*len的方式实现
  - [ ] 703 数据流中的第 K 大元素： heapq小根堆排序, 注意原地排序，注意先push再pop因为可能原始nums为空
  - [ ] 剑指offer40 最小的k个数：小根堆只能解TopK，所以转换成负数

1047. 删除字符串中的所有相邻重复项.md

给出由小写字母组成的字符串 S，重复项删除操作会选择两个相邻且相同的字母，并删除它们。

在 S 上反复执行重复项删除操作，直到无法继续删除。

在完成所有重复项删除操作后返回最终的字符串。答案保证唯一。

示例：

输入："abbaca"
输出："ca"
解释：
例如，在 "abbaca" 中，我们可以删除 "bb" 由于两字母相邻且相同，这是此时唯一可以执行删除操作的重复项。之后我们得到字符串 "aaca"，其中又只有 "aa" 可以执行重复项删除操作，所以最后的字符串为 "ca"。

提示：

```
1 <= S.length <= 20000
s 仅由小写英文字母组成。
```

```python
class Solution:
    def removeDuplicates(self, s: str) -> str:
        stack = [s[0]]
        for i in s[1:]:
            if stack and i != stack[-1]:
                stack.append(i)
            elif not stack:
                stack.append(i)
            else:
                stack.pop()
        return ''.join(stack)
```

Tips

同样是两两比较题

130. 被围绕的区域.md

给你一个 m x n 的矩阵 board ，由若干字符 'X' 和 'O' ，找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充。

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：被围绕的区间不会存在于边界上，换句话说，任何边界上的 'O' 都不会被填充为 'X'。 任何不在边界上，或不与边界上的 'O' 相连的 'O' 最终都会被填充为 'X'。如果两个元素在水平或垂直方向相邻，则称它们是“相连”的。

示例 2：

输入：board = [["X"]]
输出：[["X"]]

提示：

```
m == board.length
n == board[i].length
1 <= m, n <= 200
board[i][j] 为 'X' 或 'O'
```

```
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        nrow = len(board)
        ncol = len(board[0])
        stack = []
        for i in range(ncol):
            if board[0][i] == 'O':
                stack.append((0, i))
            if board[nrow - 1][i] == 'O':
                stack.append((nrow - 1, i))
        for i in range(nrow):
            if board[i][0] == 'O':
                stack.append((i, 0))
            if board[i][ncol - 1] == 'O':
                stack.append(((i, ncol - 1)))
        while stack:
            node = stack.pop()
            board[node[0]][node[1]] = 'M'
            for nr,nc in [(node[0]+1, node[1]),(node[0]-1,node[1]),
(node[0],node[1]+1), (node[0],node[1]-1)]:
                if nr < nrow and nr>=0 and nc<ncol and nc>=0 and board[nr][nc]=='O':
                    stack.append((nr,nc))

        for i in range(nrow):
            for j in range(ncol):
                if board[i][j] =='M':
                    board[i][j] = 'O'
                elif board[i][j]=='O':
                    board[i][j] = 'X'
```

Tips

不被围绕的区域就是4条边界上的O，以及和他们相恋的部分。所以先遍历4个边界找到所有的O，然后放入栈，依次pop，寻找和他们直接相连的部分入栈，所有这些O都标记成M。最终只需要把不是M的保留，是M的变成X即可

150. 逆波兰表达式求值.md

根据 逆波兰表示法，求表达式的值。

有效的算符包括 +、-、*、/ 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

说明：

> 整数除法只保留整数部分。
> 给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 `0` 的情况。

示例 1：

输入：tokens = ["2","1","+","3","*"]
输出：9
解释：该算式转化为常见的中缀算术表达式为：((2 + 1) * 3) = 9

示例 2：

输入：tokens = ["4","13","5","/","+"]
输出：6
解释：该算式转化为常见的中缀算术表达式为：(4 + (13 / 5)) = 6

示例 3：

输入：tokens = ["10","6","9","3","+","-11","”,”/”,”","17","+","5","+"]
输出：22
解释：
该算式转化为常见的中缀算术表达式为：
  ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22

提示：

```
1 <= tokens.length <= 104
tokens[i] 要么是一个算符（"+"、"-"、"*" 或 "/"），要么是一个在范围 [-200, 200] 内的整数
```

逆波兰表达式：

逆波兰表达式是一种后缀表达式，所谓后缀就是指算符写在后面。

> 平常使用的算式则是一种中缀表达式，如 (1 + 2) * (3 + 4) 。
> 该算式的逆波兰表达式写法为 ((1 2 +) (3 4 +) *) 。

逆波兰表达式主要有以下两个优点：

> 去掉括号后表达式无歧义，上式即便写成 1 2 + 3 4 + * 也可以依据次序计算出正确结果。
> 适合用栈操作运算：遇到数字则入栈；遇到算符则取出栈顶两个数字进行计算，并将结果压入栈中。

```python
class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        stack = []
        for token in tokens:
            if token not in ['+','-','*','/']:
                stack.append(int(token))
            else:
                b = stack.pop()
                a = stack.pop()
                if token=='+':
                    stack.append(a+b)
                elif token=='-':
                    stack.append(a-b)
                elif token =='*':
                    stack.append(a*b)
                else:
                    if a*b<0:
                        stack.append(-(abs(a)//abs(b)))
                    else:
                        stack.append(a//b)
        return int(stack.pop())
```

Tips

用栈来存放数值，当碰到计算符的时候pop出来进行四则运算

155. 最小栈.md

设计一个支持 push ， pop ， top 操作，并能在常数时间内检索到最小元素的栈。

```
push(x) —— 将元素 x 推入栈中。
pop() —— 删除栈顶的元素。
top() —— 获取栈顶元素。
getMin() —— 检索栈中的最小元素。
```

示例:

输入：
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

输出：
[null,null,null,null,-3,null,0,-2]

解释：
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); --> 返回 -3.
minStack.pop();
minStack.top();   --> 返回 0.
minStack.getMin(); --> 返回 -2.

提示：

```
pop、top 和 getMin 操作总是在 非空栈 上调用。
```

```python
class MinStack:

    def __init__(self):
        self.stack = []
        self.minstack = [math.inf]

    def push(self, val: int) -> None:
        self.stack.append(val)
        self.minstack.append(min(self.minstack[-1], val))
```

```
    def pop(self) -> None:
        self.stack.pop()
        self.minstack.pop()

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


    def getMin(self) -> int:
        return self.minstack[-1]



# Your MinStack object will be instantiated and called as such:
# obj = MinStack()
# obj.push(val)
# obj.pop()
# param_3 = obj.top()
# param_4 = obj.getMin()
```

Tips

因为要同时保留入栈顺序和最小顺序，所以需要用额外O(n)的空间来换取常熟时间，多加一个最小栈，每次入栈是当前的最小值和新val更小的那一个，这样最小栈中可能会出现多个min，但是保证了最小栈和栈的数量相同，可以一起pop

20. 有效的括号.md

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。

示例 1：

输入：s = "()"
输出：true

示例 2：

输入：s = "()[]{}"
输出：true

示例 3：

输入：s = "(]"
输出：false

示例 4：

输入：s = "([)]"
输出：false

示例 5：

输入：s = "{[]}"
输出：true

提示：

```
1 <= s.length <= 104
s 仅由括号 '()[]{}' 组成
```

```python
class Solution:
    def isValid(self, s: str) -> bool:
        stack = []
        for i in s:
            if i=='(':
                stack.append(')')
            elif i =='[':
                stack.append(']')
            elif i =='{':
                stack.append('}')
            else:
                if stack and stack[-1] == i:
                    stack.pop()
                else:
                    return False
        if stack:
            return False
        else:
            return True
```

Tips:

1. 前后顺序匹配问题，转化成FILO的堆栈问题
2. 虽然if else很蠢，但是这种写法对内存占用最少

200. 岛屿数量.md

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

示例 2：

输入：grid = [
 ["1","1","0","0","0"],
 ["1","1","0","0","0"],
 ["0","0","1","0","0"],
 ["0","0","0","1","1"]
]
输出：3

提示：

```
m == grid.length
n == grid[i].length
1 <= m, n <= 300
grid[i][j] 的值为 '0' 或 '1'
```

```
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        nrow = len(grid)
        ncol = len(grid[0])

        def dfs(r,c):
            nonlocal grid, nrow, ncol
            grid[r][c] = '2'
            for x,y in ([r-1,c], [r+1,c],[r, c-1],[r,c+1]):
                if 0<=x< nrow and 0<=y< ncol and grid[x][y]=='1':
```

```
                dfs(x,y)
            return
        counter = 0
        for r in range(nrow):
            for c in range(ncol):
                if grid[r][c]=='1':
                    counter+=1
                    dfs(r,c)
        return counter
```

Tips

深度优先搜索，和Tree的区别是图会存在重复遍历的问题，解决方案是每个遍历过的点都改成'2'避免重复遍历

215. 数组中的第K个最大元素.md

给定整数数组 nums 和整数 k，请返回数组中第 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。

示例 1:

输入: [3,2,1,5,6,4] 和 k = 2
输出: 5

示例 2:

输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
输出: 4

提示：

```
1 <= k <= nums.length <= 104
-104 <= nums[i] <= 104
```

```python
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        heap = [x for x in nums[:k]]
        heapq.heapify(heap)
        n=len(nums)
        for i in range(k, n):
            if nums[i]> heap[0]:
                heapq.heappop(heap)
                heapq.heappush(heap,nums[i])
        return heap[0]
```

Tips

和703题相同，用小根堆数据结构来保存最大的K个元素，这时root就是第K大的元素

225. 用队列实现栈.md

请你仅使用两个队列实现一个后入先出（LIFO）的栈，并支持普通栈的全部四种操作（push、top、pop 和 empty）。

实现 MyStack 类：

```
void push(int x) 将元素 x 压入栈顶。
int pop() 移除并返回栈顶元素。
int top() 返回栈顶元素。
boolean empty() 如果栈是空的，返回 true ；否则，返回 false 。
```

注意：

```
你只能使用队列的基本操作 —— 也就是 push to back、peek/pop from front、size 和 is empty 这些操作。
你所使用的语言也许不支持队列。 你可以使用 list （列表）或者 deque（双端队列）来模拟一个队列 , 只要是标准的队列操作即可。
```

示例：

输入：
["MyStack", "push", "push", "top", "pop", "empty"]
[[], [1], [2], [], [], []]
输出：
[null, null, null, 2, 2, false]

解释：
MyStack myStack = new MyStack();
myStack.push(1);
myStack.push(2);
myStack.top(); // 返回 2
myStack.pop(); // 返回 2
myStack.empty(); // 返回 False

提示：

```
1 <= x <= 9
最多调用100 次 push、pop、top 和 empty
每次调用 pop 和 top 都保证栈不为空
```

进阶：你能否实现每种操作的均摊时间复杂度为 O(1) 的栈？换句话说，执行 n 个操作的总时间复杂度 O(n) ，尽管其中某个操作可能需要比其他操作更长的时间。你可以使用两个以上的队列。

```python
class MyStack:

    def __init__(self):
        self.queue = collections.deque()

    def push(self, x: int) -> None:
        n = len(self.queue)
        self.queue.append(x)
        for _ in range(n):
            self.queue.append(self.queue.popleft())

    def pop(self) -> int:
        return self.queue.popleft()


    def top(self) -> int:
        return self.queue[0]
```

```
    def empty(self) -> bool:
        return not self.queue
```

Tips

1. python deque是双向队列，优化了list在双端的操作，append pop 在0和-1都是O(1)的操作
2. 时间复杂度pop，top，empty是O(1), push因为每次要移动前n个元素到末尾所以是O(n)
3. 注意队列FIFO的出入顺序，不会改变元素顺序

232. 用栈实现队列.md

请你仅使用两个栈实现先入先出队列。队列应当支持一般队列支持的所有操作（push、pop、peek、empty）：

实现 MyQueue 类：

```
void push(int x) 将元素 x 推到队列的末尾
int pop() 从队列的开头移除并返回元素
int peek() 返回队列开头的元素
boolean empty() 如果队列为空，返回 true ；否则，返回 false
```

说明：

```
你只能使用标准的栈操作 —— 也就是只有 push to top, peek/pop from top, size, 和 is empty 操作是合法的。
你所使用的语言也许不支持栈。你可以使用 list 或者 deque（双端队列）来模拟一个栈，只要是标准的栈操作即可。
```

进阶：

```
你能否实现每个操作均摊时间复杂度为 O(1) 的队列？换句话说，执行 n 个操作的总时间复杂度为 O(n) ，即使其中一个操作可能花费较长时间。
```

```
class MyQueue:

    def __init__(self):
        self.stackin = [] # python list is stack
        self.stackout = []
```

```
    def push(self, x: int) -> None:
        self.stackin.append(x)

    def pop(self) -> int:
        if not self.stackout:
            while self.stackin:
                self.stackout.append(self.stackin.pop())
        return self.stackout.pop()

    def peek(self) -> int:
        if not self.stackout:
            while self.stackin:
                self.stackout.append(self.stackin.pop())
        return self.stackout[-1]

    def empty(self) -> bool:
        return not self.stackin and not self.stackout
```

Tips

1. python list本身就是stack LIFO, 维护两个list，一个入栈一个出栈
2. 时间复杂度复杂度push, empty O(1), peek和pop虽然会遍历入栈反向写入出栈，但是均摊到每个元素还是O(1)。空间复杂度每个元素要么在入要么在出O(n)
3. 注意栈FILO的出入顺序，每循环一遍所有元素的顺序会reverse

239. 滑动窗口最大值.md

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

示例 2：

输入：nums = [1], k = 1
输出：[1]

示例 3：

输入：nums = [1,-1], k = 1
输出：[1,-1]

示例 4：

输入：nums = [9,11], k = 2
输出：[11]

示例 5：

输入：nums = [4,-2], k = 2
输出：[4]

提示：

```
1 <= nums.length <= 105
-104 <= nums[i] <= 104
1 <= k <= nums.length
```

1. 存储数值的单调栈

```python
from collections import deque
class MyQueue(object):
    def __init__(self):
        self.queue = deque()

    def push(self, val):
        while self.queue and val > self.queue[-1]:
            self.queue.pop()
        self.queue.append(val)

    def pop(self, val):
        if self.queue and self.queue[0] == val:
            self.queue.popleft()


class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        mq = MyQueue()
        res =[]
        for i in range(k):
```

```
            mq.push(nums[i])
        res.append(mq.queue[0])
        for i in range(k, len(nums)):
            mq.push(nums[i])
            mq.pop(nums[i-k])
            res.append(mq.queue[0])
        return res
```

Tips

1. 单调队列, 时间复杂度O(n), 空间复杂度O(k)
2. 滚动窗口优化的核心在于你不知道最大值在上一个窗口的什么位置，滑动是否会影响当前最大值。所以需要一个queue，这个queue内部元素递减，队列top就是当前最大元素
    1. Push：保证queue内数组递减，所以逐一判断只保留比当前元素大的。这样只要没有删除当前元素，就不会改变有当前元素的窗口最大值
    2. Pop: 只用考虑队列top是否是窗口丢弃的元素，因为只有当丢弃元素是top时才会改变最大值

32. 最长有效括号.md

给你一个只包含 '(' 和 ')' 的字符串，找出最长有效（格式正确且连续）括号子串的长度。

示例 1：

输入：s = "(()"
输出：2
解释：最长有效括号子串是 "()"

示例 2：

输入：s = ")()())"
输出：4
解释：最长有效括号子串是 "()()"

示例 3：

输入：s = ""
输出：0

提示：

```
0 <= s.length <= 3 * 104
s[i] 为 '(' 或 ')'
```

```
class Solution:
    def longestValidParentheses(self, s: str) -> int:
```

```
        res = 0
        stack = [-1]
        for i in range(len(s)):
            if s[i]=='(':
                stack.append(i)
            else:
                stack.pop()
                if not stack:
                    stack.append(i)
                else:
                    res = max(res, i-stack[-1])
        return res
```

Tips

有效括号的升级版。计算最长的连续有效括号，计算括号是否有效通过左括号入栈，右括号弹出的操作可以实现，但是最长连续这里又一点tricky。其实是通过右括号也入栈来对有效的括号进行隔断。如果只有右括号，则右括号自动隔断前面的所有有效括号，这里stack=[-1]的初始位置是精髓

341. 扁平化嵌套列表迭代器.md

给你一个嵌套的整数列表 nestedList 。每个元素要么是一个整数，要么是一个列表；该列表的元素也可能是整数或者是其他列表。请你实现一个迭代器将其扁平化，使之能够遍历这个列表中的所有整数。

实现扁平迭代器类 NestedIterator：

```
NestedIterator(List<NestedInteger> nestedList) 用嵌套列表 nestedList 初始化迭代器。
int next() 返回嵌套列表的下一个整数。
boolean hasNext() 如果仍然存在待迭代的整数，返回 true ；否则，返回 false 。
```

你的代码将会用下述伪代码检测：

initialize iterator with nestedList
res = []
while iterator.hasNext()
    append iterator.next() to the end of res
return res

如果 res 与预期的扁平化列表匹配，那么你的代码将会被判为正确。

示例 1：

输入：nestedList = [[1,1],2,[1,1]]
输出：[1,1,2,1,1]
解释：通过重复调用 next 直到 hasNext 返回 false，next 返回的元素的顺序应该是: [1,1,2,1,1]。

示例 2：

输入：nestedList = [1,[4,[6]]]
输出：[1,4,6]
解释：通过重复调用 next 直到 hasNext 返回 false，next 返回的元素的顺序应该是: [1,4,6]。

提示：

```
1 <= nestedList.length <= 500
嵌套列表中的整数值在范围 [-106, 106] 内
```

```
# """
# This is the interface that allows for creating nested lists.
# You should not implement it, or speculate about its implementation
# """
#class NestedInteger:
#    def isInteger(self) -> bool:
#        """
#        @return True if this NestedInteger holds a single integer, rather than a
nested list.
#        """
#
#    def getInteger(self) -> int:
#        """
#        @return the single integer that this NestedInteger holds, if it holds a single
integer
#        Return None if this NestedInteger holds a nested list
#        """
#
#    def getList(self) -> [NestedInteger]:
#        """
#        @return the nested list that this NestedInteger holds, if it holds a nested
list
#        Return None if this NestedInteger holds a single integer
#        """

class NestedIterator:
    def __init__(self, nestedList: [NestedInteger]):

        self.stack = nestedList[::-1]

    def next(self) -> int:
        return self.stack.pop().getInteger()

    def hasNext(self) -> bool:
        while self.stack:
            node = self.stack[-1]
```

```
            if node.isInteger():
                return True
            else:
                self.stack.pop()
                self.stack += node.getList()[::-1]
        return False

# Your NestedIterator object will be instantiated and called as such:
# i, v = NestedIterator(nestedList), []
# while i.hasNext(): v.append(i.next())
```

Tips

1. 本题正确的解法是一边遍历一遍进行展开。每次hasNext判断的时候对栈顶的元素进行展开
2. 记住栈的每一次入和出都会reverse顺序，所以想要正序返回，就需要倒序写入

394. 字符串解码.md

给定一个经过编码的字符串，返回它解码后的字符串。

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

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

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

示例 1：

输入：s = "3[a]2[bc]"
输出："aaabcbc"

示例 2：

输入：s = "3[a2[c]]"
输出："accaccacc"

示例 3：

输入：s = "2[abc]3[cd]ef"
输出："abcabccdcdcdef"

示例 4：

输入：s = "abc3[cd]xyz"
输出："abccdcdcdxyz"

```
class Solution:
    def decodeString(self, s: str) -> str:
```

```
        stack = []
        for i in s:
            if i !=']':
                stack.append(i)
            else:
                ss = ''
                while stack and stack[-1]!='[':
                    ss=stack.pop() + ss
                stack.pop() #discard [

                n =''
                while stack and stack[-1].isdigit():
                    n = stack.pop() + n
                stack.append(ss * int(n))
        return ''.join(stack)
```

Tips

两两匹配问题，这道题的特征更明显，在没碰到右边界时一直入栈，当碰到右边界]开始出栈进行匹配

42. 接雨水.md

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。

示例 1：

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，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

示例 2：

输入：height = [4,2,0,3,2,5]
输出：9

提示：

```
n == height.length
1 <= n <= 2 * 104
0 <= height[i] <= 105
```

```
class Solution:
    def trap(self, height: List[int]) -> int:
```

```
        stack = [0]
        res = 0
        for i in range(1, len(height)):
            while stack and height[i] > height[stack[-1]]:
                mid = stack.pop()
                if not stack:
                    break  #需要左中右三个元素
                left = stack[-1]
                w = i - left -1
                h = min(height[i], height[left]) - height[mid]
                res += w * h
            stack.append(i)
        return res
```

Tips

因为必须要右的柱子比中间的高，左边的柱子也比中间的高才能接雨水，所以元素i左边需要是递减的，右边需要是递增的，于是可以转化成单调栈问题

496. 下一个更大元素 I.md

给你两个 没有重复元素 的数组 nums1 和 nums2 ，其中nums1 是 nums2 的子集。

请你找出 nums1 中每个元素在 nums2 中的下一个比其大的值。

nums1 中数字 x 的下一个更大元素是指 x 在 nums2 中对应位置的右边的第一个比 x 大的元素。如果不存在，对应位置输出 -1 。

示例 1:

输入: nums1 = [4,1,2], nums2 = [1,3,4,2].
输出: [-1,3,-1]
解释:
  对于 num1 中的数字 4 ，你无法在第二个数组中找到下一个更大的数字，因此输出 -1 。
  对于 num1 中的数字 1 ，第二个数组中数字1右边的下一个较大数字是 3 。
  对于 num1 中的数字 2 ，第二个数组中没有下一个更大的数字，因此输出 -1 。

示例 2:

输入: nums1 = [2,4], nums2 = [1,2,3,4].
输出: [3,-1]
解释:
  对于 num1 中的数字 2 ，第二个数组中的下一个较大数字是 3 。
  对于 num1 中的数字 4 ，第二个数组中没有下一个更大的数字，因此输出 -1 。

提示：

```
1 <= nums1.length <= nums2.length <= 1000
0 <= nums1[i], nums2[i] <= 104
nums1和nums2中所有整数 互不相同
nums1 中的所有整数同样出现在 nums2 中
```

进阶：你可以设计一个时间复杂度为 O(nums1.length + nums2.length) 的解决方案吗?

```python
class Solution:
    def nextGreaterElement(self, nums1: List[int], nums2: List[int]) -> List[int]:
        dic = {}
        stack = [nums2[0]]
        ans = []
        for i in nums2[1:]:
            while stack and (stack[-1] < i):
                dic[stack.pop()]=i
            stack.append(i)
        for i in nums1:
            ans.append(dic.get(i,-1))
        return ans
```

Tips

1. 暴力双指针解法是O(n^2)的复杂度
2. 单调栈解法，FILO的顺序，可以保证栈哪不是单调递增，或者单调递减。入栈保证栈内是单调递减，出栈是单调递增，是O(n)的复杂度
3. dic保存nums2里面每个元素右边第一个更大的元素

503. 下一个更大元素 II.md

给定一个循环数组（最后一个元素的下一个元素是数组的第一个元素），输出每个元素的下一个更大元素。数字 x 的下一个更大的元素是按数组遍历顺序，这个数字之后的第一个比它更大的数，这意味着你应该循环地搜索它的下一个更大的数。如果不存在，则输出 -1。

示例 1:

输入: [1,2,1]
输出: [2,-1,2]
解释: 第一个 1 的下一个更大的数是 2；
数字 2 找不到下一个更大的数；
第二个 1 的下一个最大的数需要循环搜索，结果也是 2。

注意: 输入数组的长度不会超过 10000。

```python
class Solution:
    def nextGreaterElements(self, nums: List[int]) -> List[int]:
        stack = [0]
        n = len(nums)
        ret = [-1] * n

        for i in range(2*n-1):
            while stack and nums[i%n] > nums[stack[-1]]:
                ret[stack.pop()]= nums[i%n]
            stack.append(i%n)
        return ret
```

Tips

1. 和496的差异就是数组是可以循环的，最直观的解法就是在搜索i位置右边更大的元素的时候把数组拉直，把i-1之前的元素拼到右边。这里可以用遍历2N，求mod的方式来进行隐形拼接
2. 注意因为数组可以循环，所以这里不能用dic来保存结果，因为和496不同的是这里没有说明数组中元素无重复，所以不同位置相同元素右边最大的值会不同

703. 数据流中的第 K 大元素.md

设计一个找到数据流中第 k 大元素的类（class）。注意是排序后的第 k 大元素，不是第 k 个不同的元素。

请实现 KthLargest 类：

```
KthLargest(int k, int[] nums) 使用整数 k 和整数流 nums 初始化对象。
int add(int val) 将 val 插入数据流 nums 后，返回当前数据流中第 k 大的元素。
```

示例：

输入：
["KthLargest", "add", "add", "add", "add", "add"]
[[3, [4, 5, 8, 2]], [3], [5], [10], [9], [4]]
输出：
[null, 4, 5, 5, 8, 8]

解释：
KthLargest kthLargest = new KthLargest(3, [4, 5, 8, 2]);
kthLargest.add(3);   // return 4
kthLargest.add(5);   // return 5
kthLargest.add(10); // return 5

kthLargest.add(9); // return 8
kthLargest.add(4); // return 8

提示：

```
1 <= k <= 104
0 <= nums.length <= 104
-104 <= nums[i] <= 104
-104 <= val <= 104
最多调用 add 方法 104 次
题目数据保证，在查找第 k 大元素时，数组中至少有 k 个元素
```

```python
class KthLargest:

    def __init__(self, k: int, nums: List[int]):
        self.k = k
        self.que = nums
        heapq.heapify(self.que)# heapify是原地把list变成小根堆

    def add(self, val: int) -> int:
        heapq.heappush(self.que, val)
        while len(self.que)> self.k:
            heapq.heappop(self.que)
        return self.que[0]
```

Tips

优先队列算法（最小堆）

1. heapq是二叉树结构，特点是root<=children。所以有heap[k]<=heap[2K+1] & heap[k] <=heap[2K+2]的特点，且最小元素是root
2. heapq单次插入时间复杂度是log(k)，相比使用list每次add之后sort的时间复杂度是O(klog(k))
3. 永远只维护K个最大的元素，这样最小堆的root就是第K大元素
4. pop弹出最小元素，heap[0]最小元素

733. 图像渲染.md

有一幅以二维整数数组表示的图画，每一个整数表示该图画的像素值大小，数值在 0 到 65535 之间。

给你一个坐标 (sr, sc) 表示图像渲染开始的像素值（行 ，列）和一个新的颜色值 newColor，让你重新上色这幅图像。

为了完成上色工作，从初始坐标开始，记录初始坐标的上下左右四个方向上像素值与初始坐标相同的相连像素点，接着再记录这四个方向上符合条件的像素点与他们对应四个方向上像素值与初始坐标相同的相连像素点，……，重复该过程。将所有有记录的像素点的颜色值改为新的颜色值。

最后返回经过上色渲染后的图像。

示例 1:

输入:
image = [[1,1,1],[1,1,0],[1,0,1]]
sr = 1, sc = 1, newColor = 2
输出: [[2,2,2],[2,2,0],[2,0,1]]
解析:
在图像的正中间，(坐标(sr,sc)=(1,1)),
在路径上所有符合条件的像素点的颜色都被更改成2。
注意，右下角的像素没有更改为2，
因为它不是在上下左右四个方向上与初始点相连的像素点。

注意:

```
image 和 image[0] 的长度在范围 [1, 50] 内。
给出的初始点将满足 0 <= sr < image.length 和 0 <= sc < image[0].length。
image[i][j] 和 newColor 表示的颜色值在范围 [0, 65535]内。
```

```
class Solution:
    def floodFill(self, image: List[List[int]], sr: int, sc: int, newColor: int) ->
List[List[int]]:
        curval = image[sr][sc]
        nrow = len(image)
        ncol = len(image[0])
        if curval == newColor:
            return image
        image[sr][sc] = newColor
        stack = [(sr,sc)]
        while stack:
            pos = stack.pop()
            r = pos[0]
            c = pos[1]
            for row, col in [(r-1,c),(r,c-1), (r+1,c), (r,c+1)]:
                if row >=0 and row <nrow and col >=0 and col < ncol and image[row]
[col]==curval:
                    stack.append((row,col))
                    image[row][col] = newColor
        return image
```

Tips

DFS Search，和130解决方案一样。感觉自己要是面试做不出来这道题应该是语文不好，题没看明白。其实就是(sr,sc)相邻，任意上下左右能到到的位置上，对应位置的值和（sr,src)位置的值相同的就都改成newColor, 所以碰到对应的值入栈，然后么一步都出栈再便利山下左右四个位置，符合要求的再入栈

剑指 Offer 40. 最小的k个数.md

输入整数数组 arr ，找出其中最小的 k 个数。例如，输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。

示例 1：

输入：arr = [3,2,1], k = 2
输出：[1,2] 或者 [2,1]

示例 2：

输入：arr = [0,1,2,1], k = 1
输出：[0]

限制：

```
0 <= k <= arr.length <= 10000
0 <= arr[i] <= 10000
```

```python
class Solution:
    def getLeastNumbers(self, arr: List[int], k: int) -> List[int]:
        if k==0:
            return list()
        nums = [-i for i in arr[:k]]
        heapq.heapify(nums)
        for i in arr[k:]:
            if -i > nums[0]:
                heapq.heappop(nums)
                heapq.heappush(nums, -i)
        return [-x for x in nums]
```

Tips

大根堆问题，因为python只有小根堆的数据结构，所以需要number取负数，保留K个最大的负数->k个最小的正数