第一部分

1.1.二分搜索

34. 在排序数组中查找元素的第一个和最后一个位置

```python
# 34. 在排序数组中查找元素的第一个和最后一个位置
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        return self.findZone(nums, target, 0, len(nums)-1)

    def findZone(self, nums, target, start, end):
        if start > end:
            return [-1, -1]
        else:
            mid = int((start + end)/2)
            if nums[mid] == target:
                left = mid
                while left >= 0 and nums[left] == target :
                    left -= 1
                right = mid
                while right < len(nums) and  nums[right] == target:
                    right += 1
                return [left+1, right-1]
            elif nums[mid] > target:
                return self.findZone(nums, target, start, mid-1)
            elif nums[mid] < target:
                return self.findZone(nums, target, mid+1, end)

# 非递归的方式
class Solution:

    def searchRange(self, n_list, target):
        return [self.leftBound(n_list, target), self.rightBound(n_list, target)]

    def leftBound(self, n_list, target):
        low, hight = 0, len(n_list)-1
        while low <= hight:
            mid = (low + hight)//2
            if n_list[mid] > target:
                hight = mid -1
            elif n_list[mid] < target:
                low = mid + 1
            elif n_list[mid] == target:
                hight = mid -1
        if low >= len(n_list):
            return -1
        elif low < len(n_list):
            return  low if n_list[low] == target else -1
```

```
    def rightBound(self, n_list, target):
        low, hight = 0, len(n_list) - 1
        while low <= hight:
            mid = (low + hight)//2
            if n_list[mid] > target:
                hight = mid-1
            elif n_list[mid] < target:
                low = mid + 1
            elif n_list[mid] == target:
                low = mid + 1
        if hight < 0:
            return -1
        else:
            return hight if n_list[hight] == target else -1
```

704. 二分查找

```
# 704. 二分查找
from typing import List

class Solution:
    def search(self, nums: List[int], target: int) -> int:
        return self.findN(nums, target, 0, len(nums)-1)

    def findN(self, nums, target, s, e):
        if s > e:
            return -1
        mid = int((s+e)/2)
        if nums[mid] == target:
            return mid
        elif nums[mid] > target:
            return self.findN(nums, target, s, mid-1)
        elif nums[mid] < target:
            return self.findN(nums, target, mid+1, e)
```

35. 搜索插入位置

```
# 35. 搜索插入位置
from typing import List

class Solution:
    def searchInsert(self, nums: List[int], target: int) -> int:
        return self.findN(nums, target, 0, len(nums)-1)

    def findN(self, nums, target, s, e):
        if s > e:
            return s
```

```
        mid = int((s + e)/2)
        if nums[mid] == target:
            return mid
        elif nums[mid] > target:
            return self.findN( nums,target, s, mid-1)
        elif nums[mid] < target:
            return self.findN(nums, target, mid+1, e)
```

354. 俄罗斯套娃信封问题

```
# 354. 俄罗斯套娃信封问题
from functools import cmp_to_key
from typing import List
class Solution:
    def maxEnvelopes(self, envelopes: List[List[int]]) -> int:
        def sort_item(item1, item2):
            if item1[0] == item2[0]:
                return item2[1] - item1[1]
            else:
                return item1[0] - item2[0]
        envelopes_list = sorted(envelopes, key=cmp_to_key(sort_item))
        height_list = [item[1] for item in envelopes_list]
        # O(N*2)的方式记录dp会超时，但是这个方式确实太fancy了
        return self.lengthOfLIS(height_list)

    def lengthOfLIS(self, nums):
        piles, n = 0, len(nums)
        top = [0 for _ in range(n)]
        for i in range(n):
            poker = nums[i]
            left, right = 0, piles
            while left < right:
                mid = int((left + right)/2)
                if top[mid] >= poker:
                    right = mid
                else:
                    left = mid + 1
            if left == piles:
                piles += 1
            top[left] = poker
        return piles
```

300. 最长递增子序列

```
# 300. 最长递增子序列
from typing import List

class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
```

```
        dp = [1 for _ in nums]
        for i in range(1, len(nums)):
            for j in range(i):
                if nums[i] > nums[j]:
                    dp[i] = max(dp[i], dp[j]+1)
        return max(dp)

class Solution1:
    def lengthOfLIS(self, nums):
        maxL = 0
        # 存放当前的递增序列的潜在数据
        dp = [0 for _ in nums]
        for num in nums:
            lo, hi = 0, maxL
            # 二分查找，并替换，这一步维护dp的本质是维护一个潜在的递增序列。非常trick
            while lo < hi:
                mid = lo + (hi-lo)/2
                if dp[mid] < num:
                    lo = mid + 1
                else:
                    hi = mid
            dp[lo] = num
            # 若是接在最后面，则连续递增序列张长了
            if lo == maxL:
                maxL += 1
        return maxL
```

392. 判断子序列

```
# 392. 判断子序列
class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        i,j = 0, 0
        while i < len(s) and j < len(t):
            if s[i] == t[j]:
                i += 1
                j += 1
            else:
                j += 1
        if i == len(s):
            return True
        else:
            return False
```

793. 阶乘函数后 K 个零

```
# 793. 阶乘函数后 K 个零
import sys
class Solution:
```

```
    def preimageSizeFZF(self, k: int) -> int:
        return self.rightBound(k) - self.leftBound(k) + 1

    def trailingZeros(self, n):
        res = 0
        d = n
        while d //5 > 0:
            res += d//5
            d = d//5
        return res

    def leftBound(self, target):
        low, hight = 0 , sys.maxsize
        while low < hight:
            mid = int((low + hight)/2)
            mid_zero = self.trailingZeros(mid)
            if mid_zero < target:
                low = mid+1
            elif mid_zero > target:
                hight = mid
            else:
                hight = mid
        return low

    def rightBound(self, target):
        low, hight = 0, sys.maxsize
        while low < hight:
            mid = int((low+hight)//2)
            mid_zero = self.trailingZeros(mid)
            if mid_zero < target:
                low = mid+1
            elif mid_zero > target:
                hight = mid
            else:
                low = mid+1
        return low-1
```

172. 阶乘后的零

```
# 172. 阶乘后的零
class Solution:
    def trailingZeroes(self, n: int) -> int:
        res = 0
        divisor = 5
        while divisor <= n:
            res += n // divisor
            divisor *= 5
        return res
```

875. 爱吃香蕉的珂珂

```python
# 875. 爱吃香蕉的珂珂
from typing import List

class Solution:
    def minEatingSpeed(self, piles: List[int], h: int) -> int:
        max_val = max(piles)
        if max_val <= 0:
            return 0
        return self.minHelp(piles, 1, max_val, h)

    def minHelp(self, piles: List[int], s: int, e: int, h: int):
        if s == e:
            return s
        mid = (s+e)//2
        if self.eatCount(piles, mid) > h:
            return self.minHelp(piles, mid+1, e, h)
        else:
            return self.minHelp(piles, s, mid, h)

    def eatCount(self, piles: List[int], k: int):
        counter = 0
        for n in piles:
            if n % k == 0:
                counter += n // k
            else:
                counter += n // k + 1
        return counter
```

1011. 在 D 天内送达包裹的能力

```python
from typing import List

# 1011. 在 D 天内送达包裹的能力
class Solution:
    def shipWithinDays(self, weights: List[int], days: int) -> int:
        max_val = sum(weights)
        min_val = max(weights)
        return self.carrayWeight(weights, min_val, max_val, days)

    def carrayWeight(self, weights, s, e, days):
        if s == e:
            return s
        mid = (s + e) // 2
        if self.carrayDays(weights, mid) > days:
            return self.carrayWeight(weights, mid + 1, e, days)
        else:
            return self.carrayWeight(weights, s, mid, days)

    def carrayDays(self, weights, limitWeight):
        days = 0
```

```
        cumWeight = 0
        for w in weights:
            if cumWeight + w > limitWeight:
                days += 1
                cumWeight = w
            elif cumWeight + w == limitWeight:
                days += 1
                cumWeight = 0
            else:
                cumWeight += w
        if cumWeight != 0:
            days += 1
        return days

if __name__ == "__main__":
    s = Solution()
    weights = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    limitWeight = 11
    print(s.carrayDays(weights, limitWeight))
```

1.2.滑动窗口

3. 无重复字符的最长子串

```
# 3. 无重复字符的最长子串
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        idx_dict = dict()
        cur_idx = 0
        max_len = 0
        for j, alpha in enumerate(s):
            if idx_dict.get(s[j]) is None:
                idx_dict[s[j]] = [j]
                max_len = max(max_len, j-cur_idx+1)
            else:
                pre_idx = idx_dict.get(s[j])[-1]
                if cur_idx > pre_idx:
                    max_len = max(max_len, j-cur_idx+1)
                    idx_dict[s[j]].append(j)
                else:
                    max_len = max(max_len, j-pre_idx)
                    cur_idx = pre_idx + 1
                    idx_dict[s[j]].append(j)
        return max_len
```

76. 最小覆盖子串

```


from collections import defaultdict
```

```
class Solution(object):
    def minWindow(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: str
        """
        mem = defaultdict(int)
        for char in t:
            mem[char]+=1
        t_len = len(t)

        minLeft, minRight = 0,len(s)
        left = 0

        for right,char in enumerate(s):
            if mem[char]>0:
                t_len-=1
            mem[char]-=1

            if t_len==0:
                while mem[s[left]]<0:
                    mem[s[left]]+=1
                    left+=1

                if right-left<minRight-minLeft:
                    minLeft,minRight = left,right

                mem[s[left]]+=1
                t_len+=1
                left+=1
        return '' if minRight==len(s) else s[minLeft:minRight+1]
```

438. 找到字符串中所有字母异位词

```
# 438. 找到字符串中所有字母异位词
from collections import defaultdict
class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        if len(s) < len(p):
            return []
        p_dict, s_dict = dict(), dict()
        left, right = 0, -1
        res_list = []
        for a in p:
            if p_dict.get(a) is None:
                p_dict[a] = 0
            p_dict[a] += 1
            right += 1
            if s_dict.get(s[right]) is None:
                s_dict[s[right]] = 0
```

```
            s_dict[s[right]] += 1
        if self.cmp_dict(p_dict, s_dict):
            res_list.append(left)
        right += 1
        while right < len(s):
            if s_dict.get(s[right]) is None:
                s_dict[s[right]] = 0
            s_dict[s[right]] += 1
            if s_dict.get(s[left]) is None:
                s_dict[s[left]] = 0
            s_dict[s[left]] -= 1
            left += 1
            if self.cmp_dict(p_dict, s_dict):
                res_list.append(left)
            right += 1
        return res_list

    def cmp_dict(self, p1_dict, p2_dict):
        for k, v in p1_dict.items():
            if v == 0:
                continue
            if p2_dict.get(k) != v:
                return False
        for k, v in p2_dict.items():
            if v == 0:
                continue
            if p1_dict.get(k) != v:
                return False
        return True

s = "cbaebabacd"
p = "abc"
slo = Solution()
res = slo.findAnagrams(s, p)
print(res)
print(s[6:9])
```

567. 字符串的排列

```
# 567. 字符串的排列
class Solution:
    def checkInclusion(self, s1: str, s2: str) -> bool:
        if len(s2) < len(s1):
            return False

        s2_dict, s1_dict = dict(), dict()
        left, right = 0, 0
        for s in s1:
            if s1_dict.get(s) is None:
                s1_dict[s] = 0
            s1_dict[s] += 1
```

```
            if s2_dict.get(s2[right]) is None:
                s2_dict[s2[right]] = 0
            s2_dict[s2[right]] += 1
            right += 1
        if self.cmp_dict(s1_dict, s2_dict):
            return True
        print(s2_dict)
        print(right)
        while right < len(s2):
            if s2_dict.get(s2[right]) is None:
                s2_dict[s2[right]] = 0
            s2_dict[s2[right]] += 1
            if s2_dict.get(s2[left]) is None:
                s2_dict[s2[left]] = 0
            s2_dict[s2[left]] -= 1
            left += 1
            if self.cmp_dict(s2_dict, s1_dict):
                return True
            right += 1
            print(s2_dict)
        return False

    def cmp_dict(self, t1_dict, t2_dict):
        for k, v in t1_dict.items():
            if v == 0:
                continue
            if t2_dict.get(k) != v:
                return False
        for k, v in t2_dict.items():
            if v == 0:
                continue
            if t1_dict.get(k) != v:
                return False
        return True

s1 = "adc"
s2 = "dcda"

sol = Solution()
res = sol.checkInclusion(s1, s2)
print(res)
```

239.滑动窗口最大值

```
from typing import List

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        win, ret = [], []
        for i, v in enumerate(nums):
            if i >= k and win[0] <= i - k:
```

```
                win.pop(0)
            while win and nums[win[-1]] <= v:
                win.pop()
            win.append(i)
            if i >= k - 1:
                ret.append(nums[win[0]])
        return ret
```

1.3.其他问题

26. 删除有序数组中的重复项

```
# 26. 删除有序数组中的重复项
from typing import List

class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        if len(nums) <= 0:
            return 0
        s_p = 1
        pre_val = nums[0]
        for f_p in range(1, len(nums)):
            if pre_val != nums[f_p]:
                nums[s_p]= nums[f_p]
                s_p += 1
            pre_val = nums[f_p]
        return s_p

if __name__ == "__main__":
    nums = [1,1,2,2,2,3,3]
    sol = Solution()
    ret = sol.removeDuplicates(nums)
    print(ret)
```

83. 删除排序链表中的重复元素

```
# 83. 删除排序链表中的重复元素
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def deleteDuplicates(self, head: ListNode) -> ListNode:
        if head is None:
            return head
        s_p = head
        pre_val = head.val
```

```
        f_p = head
        while f_p is not None:
            if pre_val != f_p.val:
                s_p.next = f_p
                s_p = s_p.next
            pre_val = f_p.val
            f_p = f_p.next
        s_p.next = None
        return head
```

27.移除元素

```
# 27.移除元素
from typing import List

class Solution:
    def removeElement(self, nums: List[int], val: int) -> int:
        s_p = 0
        for f_p, n in enumerate(nums):
            if nums[f_p] != val:
                nums[s_p] = nums[f_p]
                s_p += 1
        return s_p
```

283. 移动零

```
# 283. 移动零
from typing import List

class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        s_p = 0
        for i, n in enumerate(nums):
            if n != 0:
                nums[s_p] = nums[i]
                s_p += 1
        while s_p < len(nums):
            nums[s_p] = 0
            s_p += 1
        return nums
```

11. 盛最多水的容器

```python
# 11. 盛最多水的容器
from typing import List

class Solution:
    def maxArea(self, height: List[int]) -> int:
        i, j = 0, len(height)-1
        area = 0
        while i < j:
            cur_area = min(height[i], height[j]) * (j - i)
            area = max(cur_area, area)
            if height[i] < height[j]:
                i += 1
            else:
                j -= 1
        return area

height = [1,8,6,2,5,4,8,3,7]
sol = Solution()
assert 49 == sol.maxArea(height)
```

15.三数之和

```python
from typing import List

#15.三数之和
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        nums = sorted(nums)
        pair_list = []
        for idx, n in enumerate(nums):
            tmp_nums = nums[idx+1:]
            if nums[idx] > 0:
                break
            if idx >= 1 and nums[idx] == nums[idx-1]:
                continue
            pair_list.extend(self.twoSum(-n, tmp_nums))
        return pair_list

    def twoSum(self, target, nums):
        i, j = 0, len(nums)-1
        pair_list = []
        while i < j:
            if nums[i] + nums[j] > target:
                j -= 1
            elif nums[i] + nums[j] < target:
                i += 1
            elif nums[i] + nums[j] == target:
                pair_list.append([-target, nums[i], nums[j]])
                while i < j and nums[i] == nums[i+1]:
                    i += 1
```

```
                while i < j and nums[j] == nums[j-1]:
                    j -= 1
                i += 1
                j -= 1
        return pair_list
```

18.四数之和

```
from typing import List

# 18.四数之和
class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        nums = sorted(nums)
        return self.nSums(target, 4, nums)

    def nSums(self, target, n, nums):
        if n == 2:
            return self.twoSum(target, nums)
        else:
            res_list = []
            for i, val in enumerate(nums):
                tmp_nums = nums[i+1:]
                if i >= 1 and nums[i] == nums[i-1]:
                    continue
                sub_list = self.nSums(target-val, n-1, tmp_nums)
                for sub in sub_list:
                    sub.append(val)
                    res_list.append(sub)
            return res_list

    def twoSum(self, target, nums):
        i, j = 0, len(nums)-1
        res_list = []
        while i < j:
            if nums[i] + nums[j] == target:
                res_list.append([nums[i], nums[j]])
                while i< j and nums[i] == nums[i+1]:
                    i += 1
                while i< j and nums[j] == nums[j-1]:
                    j -= 1
                i += 1
                j -= 1
            elif nums[i] + nums[j] < target:
                i += 1
            elif nums[i] + nums[j] > target:
                j -= 1
        return res_list

if __name__ == "__main__":
    nums = [1, 0, -1, 0, -2, 2]
```

```
    target = 0
    sol = Solution()
    ret = sol.fourSum(nums, target)
    print(ret)
```

870. 优势洗牌

```
# 870. 优势洗牌
from typing import List


class Solution:
    def advantageCount(self, nums1: List[int], nums2: List[int]) -> List[int]:
        if len(nums1) != len(nums2):
            return []
        # 排序
        nums1_s = sorted(nums1)
        nums2_s = sorted(nums2)
        n2_dict = dict()
        # nums2的词典记录索引位置
        for i, n in enumerate(nums2):
            if n2_dict.get(n) is None:
                n2_dict[n] = []
            n2_dict[n].append(i)
        i, j = 0, 0
        nums1_tmp = [None for _ in nums1]
        # 排序后双指针对比
        while i < len(nums1) and j < len(nums2):
            if nums1_s[i] <= nums2_s[j]:
                i += 1
            else:
                nums1_tmp[j] = nums1_s[i]
                nums1_s[i] = None
                i += 1
                j += 1

        nums1_s = list(filter(lambda x: x != None, nums1_s))
        # 补数
        while j < len(nums2):
            nums1_tmp[j] = nums1_s.pop()
            j += 1
        # 还原
        for i, n in enumerate(nums1_tmp):
            n2_idx = n2_dict.get(nums2_s[i]).pop()
            nums1[n2_idx] = n
        return nums1
```

42.接雨水

```python
#42.接雨水
from typing import List

class Solution:
    def trap(self, height: List[int]) -> int:
        if len(height) <= 2:
            return 0
        right_idx_list = sorted(range(len(height)), key=lambda i: height[i],
reverse=True)
        left_idx_list = []
        water = 0
        right_idx_list.remove(0)
        for i in range(1, len(height)-1):
            left_idx_list = self.update_idx_list(left_idx_list, i-1, height)
            right_idx_list.remove(i)
            left_max, right_max = height[left_idx_list[0]],
height[right_idx_list[0]]
            h = min(left_max, right_max) - height[i]
            print("left:", left_idx_list, "right:", right_idx_list, "cur_idx:", i)
            if h > 0:

                water += h
        return water

    def update_idx_list(self, idx_list, idx, height):
        if len(idx_list) == 0:
            idx_list.append(idx)
            return idx_list
        for i, cur_idx in enumerate(idx_list):
            if height[cur_idx] < height[idx]:
                idx_list.insert(i, idx)
                break
        return idx_list

class Solution1:
    def trap(self, height: List[int]) -> int:
        if len(height) <= 2:
            return 0
        water = 0
        for i in range(1, len(height)-1):
            h = min(max(height[:i]), max(height[i+1:])) - height[i]
            if h > 0:
                water += h
        return water

class Solution2:
    def trap(self, height: List[int]) -> int:
        ans = 0
        h1 = 0
        h2 = 0
        for i in range(len(height)):
            h1 = max(h1,height[i])
            h2 = max(h2,height[-i-1])
```

```
            ans = ans + h1 + h2 -height[i]
        return  ans - len(height)*h1

if __name__ == "__main__":
    height = [0,1,0,2,1,0,1,3,2,1,2,1]
    sol = Solution()
    ret = sol.trap(height)
    ret1 = sol.trap(height)
    print(ret)
    print(ret1)
```

1.4.区间问题

986.区间列表的交集

```
# 986.区间列表的交集
from typing import List

class Solution:
    def intervalIntersection(self, firstList: List[List[int]], secondList:
List[List[int]]) -> List[List[int]]:
        i,j = 0, 0
        result_list = []
        while i < len(firstList) and j < len(secondList):
            if firstList[i][0] > secondList[j][0]:
                if firstList[i][0] > secondList[j][1]:
                    j += 1
                else:
                    if firstList[i][1] < secondList[j][1]:
                        result_list.append([firstList[i][0], firstList[i][1]])
                        i += 1
                    else:
                        result_list.append([firstList[i][0], secondList[j][1]])
                        j += 1
            else:
                if secondList[j][0] > firstList[i][1]:
                    i += 1
                else:
                    if secondList[j][1] > firstList[i][1]:
                        result_list.append([secondList[j][0], firstList[i][1]])
                        i += 1
                    else:
                        result_list.append([secondList[j][0], secondList[j][1]])
                        j += 1
        return result_list
```

1288.删除被覆盖区间

```python
# 1288.删除被覆盖区间
from functools import cmp_to_key
from typing import List

class Solution:
    def removeCoveredIntervals(self, intervals: List[List[int]]) -> int:
        def cmp(t1, t2):
            if t1[0] != t2[0] :
                return t1[0] - t2[0]
            else:
                return t2[1] - t1[1]
        intervals = sorted(intervals, key=cmp_to_key(cmp))
        print(intervals)
        j = 0
        while j < len(intervals) -1:
            if intervals[j][0] == intervals[j+1][0]:
                del intervals[j+1]
            else:
                if intervals[j][1] >= intervals[j+1][1]:
                    del intervals[j+1]
                else:
                    j += 1
        print(intervals)
        return len(intervals)


if __name__ == "__main__":
    # intervals = [[1,4],[3,6],[2,8]]
    intervals = [[34335,39239],[15875,91969],[29673,66453],[53548,69161],
[40618,93111]]
    sol = Solution()
    ret = sol.removeCoveredIntervals(intervals)
    print(ret)
```

56.合并区间

```python
# 56.合并区间
from functools import cmp_to_key
from typing import List

class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        def cmp(t1, t2):
            if t1[0] != t2[0]:
                return t1[0] - t2[0]
            else:
                return t2[1] - t1[1]
        intervals = sorted(intervals, key=cmp_to_key(cmp))
        i = 0
        while i < len(intervals) -1:
```

```
            if intervals[i][0] == intervals[i+1][0]:
                del intervals[i+1]
            else:
                if intervals[i][1] >= intervals[i+1][1]:
                    del intervals[i+1]
                else:
                    if intervals[i][1] < intervals[i+1][0]:
                        i += 1
                    elif intervals[i][1] < intervals[i+1][1]:
                        intervals[i] = [intervals[i][0], intervals[i+1][1]]
                        del intervals[i+1]
        return intervals
```

435.无重叠区间

```
# 435.无重叠区间
from functools import cmp_to_key
from typing import List

class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        intervals = sorted(intervals, key=lambda x: x[0])
        i = 0
        counter = 0
        while i <= len(intervals) - 2:
            if intervals[i][0] == intervals[i+1][0]:
                if intervals[i][1] > intervals[i+1][1]:
                    i += 1
                else:
                    intervals[i], intervals[i+1] = intervals[i+1], intervals[i]
                    i += 1
                counter += 1
            else:
                if intervals[i][1] >= intervals[i+1][1]:
                    i += 1
                    counter += 1
                else:
                    if intervals[i][1] <= intervals[i+1][0]:
                        i += 1
                    elif intervals[i][1] < intervals[i+1][1]:
                        intervals[i], intervals[i+1] = intervals[i+1],
intervals[i]
                        i += 1
                        counter += 1
        return counter
```

452.用最少数量的箭引爆气球

```python
from typing import List

# 452.用最少数量的箭引爆气球
class Solution:
    def findMinArrowShots(self, points: List[List[int]]) -> int:
        if len(points) <= 0:
            return 0
        points = sorted(points, key=lambda x: x[1])
        counter = 0
        i =  0
        piv = points[i]
        while i < len(points):
            counter += 1
            while i < len(points) and piv[1] >= points[i][0]:
                i += 1
            if i < len(points):
                piv = points[i]
            else:
                piv = None
        return counter if piv is None else counter + 1

if __name__ == "__main__":
    points = [[1,2]]
    sol = Solution()
    ret = sol.findMinArrowShots(points)
    print(ret)
```

1024.视频拼接

```python
from typing import List

# 1024.视频拼接
class Solution:
    def videoStitching(self, clips: List[List[int]], time: int) -> int:
        clips = sorted(clips, key=lambda x: x[0])
        i = 0
        while i < len(clips) -1 and clips[i][0] == clips[i+1][0]:
            if clips[i][1] > clips[i+1][1]:
                clips[i], clips[i+1] = clips[i+1], clips[i]
            i += 1
        piv = clips[i]
        counter = 1
        if piv[0] == 0 and piv[1] >= time:
            return counter

        while i < len(clips):
            if piv[1] >= clips[i][0]:
                # print("cur_id:", i)
                if i < len(clips) -1:
                    while i < len(clips) -1 and clips[i+1][0] <= piv[1]:
```

```
                        if clips[i][1] > clips[i+1][1]:
                            clips[i], clips[i+1] = clips[i+1], clips[i]
                        i += 1
                    piv = [piv[0], clips[i][1]]
                    counter += 1
                    if piv[0] == 0 and piv[1] >= time:
                        return counter
                    i += 1
                else:
                    piv = [piv[0], clips[i][1]]
                    counter += 1
                    i += 1
            else:
                return -1
        print(piv)
        if piv[0] == 0 and piv[1] >= time:
            return counter
        else:
            return -1

if __name__ == "__main__":
    # clips = [[0,1],[6,8],[0,2],[5,6],[0,4],[0,3],[6,7],[1,3],[4,7],[1,4],[2,5],
[2,6],[3,4],[4,5],[5,7],[6,9]]
    clips = [[5,7],[1,8],[0,0],[2,3],[4,5],[0,6],[5,10],[7,10]]
    print(len(clips))
    time = 5
    sol = Solution()
    ret = sol.videoStitching(clips, time)
    print(ret)
```

1.5.链表双指针

2.两数相加

```
#2.两数相加

# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        l1_len = self.getLength(l1)
        l2_len = self.getLength(l2)
        if l2_len > l1_len:
            l1, l2 = l2, l1
        p = 0
        pre_node = None
        l1_head = l1
        while l1 is not None:
```

```
            l2_val = 0 if l2 is None else l2.val
            cur_val = l1.val + l2_val + p
            if cur_val // 10 == 1:
                p = 1
                cur_val = cur_val % 10
            else:
                p = 0
            l1.val = cur_val
            pre_node = l1
            l1 = l1.next
            l2 = None if l2 is None else l2.next
        if p != 0:
            pre_node.next = ListNode(val=p)
        return l1_head

    def getLength(self, list_node):
        counter = 0
        while list_node is not None:
            counter += 1
            list_node = list_node.next
        return counter
```

19.删除链表的倒数第N个节点

```
# Definition for singly-linked list.
# 19.删除链表的倒数第N个节点
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:
        p_head = ListNode(next=head)
        p_head_bk = p_head
        length = self.getLength(head)
        if length == n and n == 1:
            return None
        i = 0
        while i < length -n:
            p_head = p_head.next
            i += 1
        p_head_next = p_head.next
        if p_head_next is not None:
            p_head.next = p_head_next.next
        else:
            p_head.next = None
        return p_head_bk.next

    def getLength(self, head):
        counter = 0
        while head is not None:
```

```
            counter += 1
            head = head.next
        return counter
```

21.合并两个有序链表

```
# Definition for singly-linked list.
# 21.合并两个有序链表
from typing import Optional


class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
class Solution:
    def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode])
-> Optional[ListNode]:
        head = ListNode()
        p_head = head
        while list1 is not None and list2 is not None:
            if list1.val < list2.val:
                head.next = list1
                list1 = list1.next
            else:
                head.next = list2
                list2 = list2.next
            head = head.next
        if list1 is not None:
            head.next = list1
        if list2 is not None:
            head.next = list2
        return p_head.next
```

23.合并K个升序链表

```
# 23.合并K个升序链表
# Definition for singly-linked list.
from typing import List
import heapq

class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    # 当成n-1个两个升序列表的合并
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
```

```
        if len(lists) == 0:
            return None
        p1 = lists.pop()
        for p2 in lists:
            p1 = self.mergeListNode(p1, p2)
        return p1

    def mergeListNode(self, p1, p2):
        p_head = ListNode()
        p_head_bk = p_head
        while p1 is not None and p2 is not None:
            if p1.val < p2.val:
                p_head.next = p1
                p1 = p1.next
            else:
                p_head.next = p2
                p2 = p2.next
            p_head = p_head.next
        if p1 is not None:
            p_head.next = p1
        if p2 is not None:
            p_head.next = p2
        return p_head_bk.next

class Solution1:
    # 排序利用最小堆替代就是时间复杂度OK的K个升序的列表的合并
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        p_head = ListNode()
        p_head_bk = p_head
        lists = list(filter(lambda node : node is not None, lists))
        if len(lists) == 0:
            return None

        heap_list = sorted(lists, key=lambda node:node.val)
        while len(heap_list) > 0:
            node = heap_list.pop(0)
            p_head.next = node
            p_head = p_head.next

            node = node.next
            if node is not None:
                heap_list.append(node)
                heap_list = sorted(heap_list, key=lambda node: node.val)
        return p_head_bk.next
```

141.环形链表

```
# Definition for singly-linked list.
from typing import Optional

# 141.环形链表
```

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    def hasCycle(self, head: Optional[ListNode]) -> bool:
        slow, fast = head, head
        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                return True
        return False
```

142.环形链表2

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None
# 142.环形链表2
class Solution:
    def detectCycle(self, head: ListNode) -> ListNode:
        slow, fast = head, head
        meet = None
        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                meet = fast
                break
        if meet is None:
            return None
        slow = head
        while slow != fast:
            slow = slow.next
            fast = fast.next
        return slow
```

160.链表相交

```
# Definition for singly-linked list.
# 160.链表相交
class ListNode:
    def __init__(self, x):
        self.val = x
```

```
        self.next = None

class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:
        p1, p2 = headA, headB
        while p1 != p2:
            if p1 is None:
                p1 = headB
            else:
                p1 = p1.next
            if p2 is None:
                p2 = headA
            else:
                p2 = p2.next
        return p1
```

876.链表的中间结点

```
# Definition for singly-linked list.
# 876. 链表的中间结点
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
class Solution:
    def middleNode(self, head: ListNode) -> ListNode:
        slow, fast = head, head
        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
        return slow
```

25. K 个一组翻转链表

```
# Definition for singly-linked list.
from typing import Optional

# 25. K 个一组翻转链表
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
class Solution:
    def reverseKGroup(self, head: Optional[ListNode], k: int) ->
Optional[ListNode]:
        a, b = head, head
        for i in range(k):
            if b == None:
                return head
```

```
            b = b.next
        new_head = self.reverse(a, b)
        a.next = self.reverseKGroup(b, k)
        return new_head

    def reverse(self, a, b):
        pre, cur, nxt = None, a, a
        while cur != b:
            nxt = cur.next
            cur.next = pre
            pre = cur
            cur = nxt
        return pre
```

92.反转链表2

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
# 92.反转链表2
class Solution:
    def reverseBetween(self, head: ListNode, left: int, right: int) -> ListNode:
        p_head = head
        counter = 1
        pre_node = None
        while counter < left :
            counter += 1
            pre_node = p_head
            p_head = p_head.next
        next_node = self.reverse(p_head, right-left+1)
        if pre_node is None:
            return next_node
        else:
            pre_node.next = next_node
            return head

    def reverse(self, head, right):
        if right == 1:
            return head
        p_head, pre_node = head, None
        counter = 1
        while counter <= right:
            counter += 1
            next_node = p_head.next
            p_head.next = pre_node
            pre_node = p_head
            p_head = next_node
```

```
        head.next = p_head
        return pre_node
```

234.回文链表

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
# 234.回文链表
class Solution:
    def isPalindrome(self, head: ListNode) -> bool:
        n_list = []
        while head is not None:
            n_list.append(head.val)
            head = head.next
        print(n_list)
        i, j = 0, len(n_list)-1
        status = True
        while i < j:
            if n_list[i] != n_list[j]:
                status = False
                break
            else:
                i +=1
                j -= 1
        return status

class Solution1:
    def isPalindrome(self, head: ListNode) -> bool:
        p_head = head
        pre_node = ListNode(p_head.val)
        p_head = p_head.next
        while p_head is not None:
            node = ListNode(p_head.val)
            node.next = pre_node
            pre_node = node
            p_head = p_head.next

        status = True
        while pre_node is not None and head is not None:
            if pre_node.val == head.val:
                pre_node = pre_node.next
                head = head.next
            else:
                status = False
                break
        return status
```

1.6.前缀和问题

303. 区域和检索 - 数组不可变

```
# 303. 区域和检索 - 数组不可变
class NumArray:

    def __init__(self, nums: List[int]):
        self.acc_list = []
        acc = 0
        for n in nums:
            acc += n
            self.acc_list.append(acc)

    def sumRange(self, left: int, right: int) -> int:
        if left > 0:
            return self.acc_list[right] - self.acc_list[left-1]
        else:
            return self.acc_list[right]
```

304.二位区域和检索-矩阵不可变

```
# 304.二位区域和检索-矩阵不可变
from typing import List

class NumMatrix:

    def __init__(self, matrix: List[List[int]]):
        self.acc_matrix = [self.getAccListt(matrix[0])]
        for i in range(1, len(matrix)):
            acc_line = 0
            acc_list = []
            for j, n in enumerate(matrix[i]):
                acc_line += n
                acc_list.append(acc_line + self.acc_matrix[i-1][j])
            self.acc_matrix.append(acc_list)

    def getAccListt(self, n_list):
        acc = 0
        tmp = []
        for n in n_list:
            acc += n
            tmp.append(acc)
        return tmp

    def sumRegion(self, row1: int, col1: int, row2: int, col2: int) -> int:
        if col1 > 0 and row1 > 0:
            return self.acc_matrix[row2][col2] - self.acc_matrix[row2][col1-1] -
self.acc_matrix[row1-1][col2] + self.acc_matrix[row1-1][col1-1]
        elif col1 == 0  and row1 > 0:
```

```
            return self.acc_matrix[row2][col2] - self.acc_matrix[row1-1][col2]
        elif col1 > 0 and row1 == 0:
            return self.acc_matrix[row2][col2] - self.acc_matrix[row2][col1-1]
        elif col1 == 0 and row1 == 0:
            return self.acc_matrix[row2][col2]
```

560. 和为 K 的子数组

```
# 560. 和为 K 的子数组
from typing import List

class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        acc_list, acc_dict = self.getAccList(nums)
        base = acc_dict[k] if acc_dict.get(k) is not None else 0
        count = 0
        for n in acc_list[::-1]:
            acc_dict[n] -= 1
            if acc_dict.get(n - k) is not None and acc_dict.get(n-k) > 0:
                count += acc_dict.get(n-k)
        return count + base

    def getAccList(self, nums):
        acc_dict = {}
        acc_list = []
        for n in nums:
            val = acc_list[-1] + n if len(acc_list) > 0 else  n
            acc_list.append(val)
            if acc_dict.get(val) is None:
                acc_dict[val] = 1
            else:
                acc_dict[val] += 1
        return acc_list, acc_dict

class Solution1:
    def subarraySum(self, nums: List[int], k: int) -> int:
        preSumDict = dict()
        preSumDict[0] = 1
        res, sum0_i = 0, 0
        for i in range(len(nums)):
            sum0_i += nums[i]
            sum0_j = sum0_i - k
            if preSumDict.get(sum0_j) is not None:
                res += preSumDict.get(sum0_j)
            if preSumDict.get(sum0_i) is None:
                preSumDict[sum0_i] = 1
            else:
                preSumDict[sum0_i] += 1
        return res
```

1.7.差分数组

1094. 拼车

```python
# 1094. 拼车
from typing import List

class Solution:
    def carPooling(self, trips: List[List[int]], capacity: int) -> bool:
        max_stop = max([trip[2] for trip in trips])
        delta_cap_list = [0 for _ in range(max_stop+1)]
        for trip in trips:
            cap, start, stop = trip[0], trip[1], trip[2]
            delta_cap_list[start] += cap
            delta_cap_list[stop] -= cap
        cap_list = []
        status = True
        for delta in delta_cap_list:
            cap = cap_list[-1] + delta if len(cap_list) > 0 else delta
            if cap > capacity:
                status = False
                break
            else:
                cap_list.append(cap)
        return status
```

1109. 航班预订统计

```python
# 1109. 航班预订统计
from typing import List

# 差分数组
class Solution:
    def corpFlightBookings(self, bookings: List[List[int]], n: int) -> List[int]:
        delta_seat_list = [0 for _ in range(n+1)]
        for booking in bookings:
            first, last, seats = booking[0], booking[1], booking[2]
            delta_seat_list[first] += seats
            if last+1 < len(delta_seat_list):
                delta_seat_list[last+1] -= seats
        all_seat_list = []
        for delta in delta_seat_list[1:]:
            alls = all_seat_list[-1] + delta  if len(all_seat_list) > 0 else delta
            all_seat_list.append(alls)
        return all_seat_list
```

370.区间加法

```python
# 370.区间加法
from typing import List
from winreg import REG_RESOURCE_LIST

class Solution:
    def zoneAdd(self, updates: List[List[int]], length: int) -> int:
        # max_val = max([item[1] for item in updates])
        delta_list = [0 for _ in range(length)]
        for item in updates:
            start, end , inc = item[0], item[1], item[2]
            delta_list[start] += inc
            if end + 1 < len(delta_list):
                delta_list[end+1] -= inc
        res_list = []
        for delta in delta_list:
            val = res_list[-1] + delta if len(res_list) > 0 else delta
            res_list.append(val)
        return res_list

if __name__ == "__main__":
    updates = [[1,3,2],[2,4,3],[0,2,-2]]
    length = 5
    s = Solution()
    ret = s.zoneAdd(updates, length)
    print(ret)
```

1.8.队列和栈

20.有效的括号

```python
# 20.有效的括号
class Solution:
    def isValid(self, s: str) -> bool:
        n_list = []
        a_dict = {"(": ")", "[":"]", "{": "}"}
        status = True
        for a in s:
            if a not in [")", "}", "]"]:
                n_list.append(a)
            else:
                if len(n_list) > 0:
                    left = n_list.pop()
                    if a_dict.get(left) != a:
                        status = False
                        break
                else:
                    status = False
                    break
        if len(n_list) > 0:
```

```
            status = False
        return status
```

921. 使括号有效的最少添加

```
#921. 使括号有效的最少添加

# 骚操作
class Solution:
    def minAddToMakeValid(self, s: str) -> int:
        while  "()" in s :
            s = s.replace("()", "")
        return len(s)

# 正常版本
class Solution1:
    def minAddToMakeValid(self, s: str) -> int:
        left = []
        for a in s:
            if a == "(":
                left.append(a)
            elif a == ")":
                if len(left) > 0:
                    if left[-1] == "(":
                        left.pop()
                    else:
                        left.append(a)
                else:
                    left.append(a)
        return len(left)
```

1541.平衡括号字符串的最少插入次数

```
# 1541.平衡括号字符串的最少插入次数
class Solution:
    def minInsertions(self, s: str) -> int:
        left = []
        counter = 0
        i = 0
        while i < len(s):
            if s[i] == "(":
                left.append(s[i])
                i += 1
            elif s[i] == ")":
                if len(left) > 0:
                    left.pop()
                    if i+1 < len(s):
                        if s[i+1] == "(":
                            counter += 1
```

```
                            i += 1
                        elif s[i+1] == ")":
                            i += 2
                    else:
                        counter += 1
                        i += 1
                else:
                    counter += 1
                    if i+1 < len(s):
                        if s[i+1] == ")":
                            i += 2
                        elif s[i+1] == "(":
                            counter += 1
                            i += 1
                    else:
                        counter += 1
                        i += 1

        return counter + len(left) * 2

def is_exists(s):
    while "())" in s:
        s = s.replace("())", "")
        print(s)

if __name__ == "__main__":
    s = "((()))(()))()(()))"
    sol = Solution1()
    ret = sol.minInsertions(s)
    print(ret)
    print(is_exists(s))
```

32. 最长有效括号

```
# 32. 最长有效括号
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        left_idx_list = []
        dp = [0 for i in range(len(s) +1)]
        for i, a in enumerate(s):
            if a == "(":
                left_idx_list.append(i)
                dp[i+1] = 0
            elif a == ")":
                if len(left_idx_list) > 0:
                   left_idx = left_idx_list.pop()
                   length = i - left_idx + 1 + dp[left_idx]
                   dp[i+1] = length
                   i += 1
                else:
```

```
                    dp[i+1] = 0
        return max(dp)
```

225.用队列实现栈

```
import queue

# 225.用队列实现栈
def print_queue(que):
    n_list = []
    while not que.empty():
        n_list.append(que.get())
    print(n_list)

class MyStack:

    def __init__(self):
        self.que = queue.Queue()
        self.top_num = None

    def push(self, x: int) -> None:
        self.que.put(x)
        self.top_num = x

    def pop(self) -> int:
        rev_que = queue.Queue()
        q_size = self.que.qsize()
        if q_size > 2:
            while q_size > 2:
                cur = self.que.get()
                rev_que.put(cur)
                q_size -= 1
            self.top_num = self.que.get()
            pop_val = self.que.get()

            rev_que.put(self.top_num)
            self.que = rev_que
        elif q_size == 2:
            self.top_num = self.que.get()
            pop_val = self.que.get()
            self.que.put(self.top_num)
        elif q_size == 1:
            self.top_num = None
            pop_val = self.que.get()
        elif q_size == 0:
            pop_val = None
        return pop_val

    def top(self) -> int:
        return self.top_num
```

```
    def empty(self) -> bool:
        return self.que.empty()

# Your MyStack object will be instantiated and called as such:
# obj = MyStack()
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.top()
# param_4 = obj.empty()
```

232.用栈实现队列

```
from collections import deque
# 232.用栈实现队列
class MyQueue:

    def __init__(self):
        self.stack = deque()
        self.top = None

    def push(self, x: int) -> None:
        if len(self.stack) == 0:
            self.top = x
        self.stack.append(x)

    def pop(self) -> int:
        if len(self.stack) == 0:
            return None
        stack_bak = deque()
        while len(self.stack) > 0:
            stack_bak.append(self.stack.pop())
        pop_val = stack_bak.pop()

        if len(stack_bak) > 0:
            self.top = stack_bak.pop()
            self.stack.append(self.top)
            while len(stack_bak) > 0:
                self.stack.append(stack_bak.pop())
        else:
            self.top = None
        return pop_val

    def peek(self) -> int:
        return self.top


    def empty(self) -> bool:
        return len(self.stack) == 0

# Your MyQueue object will be instantiated and called as such:
# obj = MyQueue()
```

```
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.peek()
# param_4 = obj.empty()
```

239.滑动窗口最大值

```
from typing import List

# 239.滑动窗口最大值
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        win, ret = [], []
        for i, v in enumerate(nums):
            if i >= k and win[0] <= i - k:
                win.pop(0)
            while win and nums[win[-1]] <= v:
                win.pop()
            win.append(i)
            if i >= k - 1:
                ret.append(nums[win[0]])
        return ret
```

1.9.二叉堆

23.合并K个升序链表

```
# 23.合并K个升序链表
# Definition for singly-linked list.
from typing import List
import heapq

class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    # 当成n-1个两个升序列表的合并
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        if len(lists) == 0:
            return None
        p1 = lists.pop()
        for p2 in lists:
            p1 = self.mergeListNode(p1, p2)
        return p1

    def mergeListNode(self, p1, p2):
        p_head = ListNode()
```

```
        p_head_bk = p_head
        while p1 is not None and p2 is not None:
            if p1.val < p2.val:
                p_head.next = p1
                p1 = p1.next
            else:
                p_head.next = p2
                p2 = p2.next
            p_head = p_head.next
        if p1 is not None:
            p_head.next = p1
        if p2 is not None:
            p_head.next = p2
        return p_head_bk.next


class Solution1:
    # 排序利用最小堆替代就是时间复杂度OK的K个升序的列表的合并
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        p_head = ListNode()
        p_head_bk = p_head
        lists = list(filter(lambda node : node is not None, lists))
        if len(lists) == 0:
            return None

        heap_list = sorted(lists, key=lambda node:node.val)
        while len(heap_list) > 0:
            node = heap_list.pop(0)
            p_head.next = node
            p_head = p_head.next

            node = node.next
            if node is not None:
                heap_list.append(node)
                heap_list = sorted(heap_list, key=lambda node: node.val)
        return p_head_bk.next
```

215.数组中的第K大元素

```
from typing import List
import heapq
# 215.数组中的第K大元素
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        class SmallHeap():
            def __init__(self, k: int):
                self.k = k
                self.small_heap = []

            def add(self, n: int):
```

```
                if len(self.small_heap) == k:
                    val = heapq.heappop(self.small_heap)
                    heapq.heappush(self.small_heap, max(val, n))
                else:
                    heapq.heappush(self.small_heap, n)

            def getTop(self):
                return self.small_heap[0]

        s_heap = SmallHeap(k)
        for n in nums:
            s_heap.add(n)
        print(s_heap.small_heap)
        return s_heap.getTop()

if __name__ == "__main__":
    nums = [3,2,1,5,6,4]
    k = 2
    sol = Solution()
    ret = sol.findKthLargest(nums, k)
    print(ret)
```

295.数据流的中位数

```
import heapq
# 295.数据流的中位数
class MedianFinder:
    def __init__(self):
        class SmallHeap:
            def __init__(self):
                self.small_heap = []

            def add(self, n):
                heapq.heappush(self.small_heap, n)

            def getTop(self):
                if self.num() > 0:
                    return self.small_heap[0]
                else:
                    return None

            def pop(self):
                if self.num() > 0:
                    return heapq.heappop(self.small_heap)
                else:
                    return None

            def num(self):
                return len(self.small_heap)

        class BigHeap:
```

```
            def __init__(self):
                self.big_heap = []

            def add(self, n):
                heapq.heappush(self.big_heap, n * (-1))

            def getTop(self):
                if self.num() > 0:
                    return self.big_heap[0] * -1
                else:
                    return None

            def pop(self):
                if self.num() > 0:
                    return heapq.heappop(self.big_heap) * (-1)
                else:
                    return None

            def num(self):
                return len(self.big_heap)

        self.s_heap = SmallHeap()
        self.b_heap = BigHeap()

    def addNum(self, num: int) -> None:
        small_top, big_top = self.s_heap.pop(), self.b_heap.pop()
        small_count, big_count = self.s_heap.num(), self.b_heap.num()
        sorted_list = []
        if small_top is not None:
            sorted_list.append(small_top)
        if big_top is not None:
            sorted_list.append(big_top)
        sorted_list.append(num)
        sorted_list = sorted(sorted_list)
        if small_count > big_count:
            if len(sorted_list) == 3:
                self.s_heap.add(sorted_list[-1])
                self.b_heap.add(sorted_list[0])
                self.b_heap.add(sorted_list[1])
            else:
                self.s_heap.add(sorted_list[1])
                self.b_heap.add(sorted_list[0])
        elif small_count < big_count:
            if len(sorted_list) == 3:
                self.s_heap.add(sorted_list[-2])
                self.s_heap.add(sorted_list[-1])
                self.b_heap.add(sorted_list[0])
            else:
                self.s_heap.add(sorted_list[1])
                self.b_heap.add(sorted_list[0])
        elif small_count == big_count:
            if len(sorted_list) == 1:
                self.s_heap.add(sorted_list[0])
            elif len(sorted_list) == 2:
```

```
                self.s_heap.add(sorted_list[1])
                self.b_heap.add(sorted_list[0])
            else:
                self.s_heap.add(sorted_list[-2])
                self.s_heap.add(sorted_list[-1])
                self.b_heap.add(sorted_list[0])

    def findMedian(self) -> float:
        small_count, big_count = self.s_heap.num(), self.b_heap.num()
        small_top, big_top = self.s_heap.getTop(), self.b_heap.getTop()
        if big_count > small_count:
            return big_top
        elif big_count < small_count:
            return small_top
        elif big_count == small_count:
            if small_count == 0:
                return None
            else:
                return (small_top + big_top)/2.0


# Your MedianFinder object will be instantiated and called as such:
# obj = MedianFinder()
# obj.addNum(num)
# param_2 = obj.findMedian()
```

```
# 703. 数据流中的第 K 大元素
from typing import List
import heapq

class KthLargest:
    def __init__(self, k: int, nums: List[int]):
        class SmallHeap:
            def __init__(self, k):
                self.k = k
                self.small_heap = []

            def add(self, val):
                if len(self.small_heap) == k:
                    top_val = heapq.heappop(self.small_heap)
                    heapq.heappush(self.small_heap, max(val, top_val))
                else:
                    heapq.heappush(self.small_heap, val)
                return self.small_heap[0]

        self.s_heap = SmallHeap(k)
        for n in nums:
            self.s_heap.add(n)

    def add(self, val: int) -> int:
```

```
        return self.s_heap.add(val)

# Your KthLargest object will be instantiated and called as such:
# obj = KthLargest(k, nums)
# param_1 = obj.add(val)
```

1.10.数据结构设计

146. LRU 缓存

```
# 146. LRU 缓存
from collections import OrderedDict

class LRUCache:

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

    def get(self, key: int) -> int:
        if self.cache.get(key) is None:
            return -1
        self.makeRecentKey(key)
        return self.cache.get(key)

    def put(self, key: int, value: int) -> None:
        if self.cache.get(key) is not None:
            self.cache[key] = value
            self.makeRecentKey(key)
            return
        if len(self.cache) >= self.capacity:
            early_key = list(self.cache.keys())[0]
            del self.cache[early_key]
        self.cache[key] = value

    def makeRecentKey(self, key: int) -> None:
        val = self.cache.get(key)
        del self.cache[key]
        self.cache[key] = val



# Your LRUCache object will be instantiated and called as such:
# obj = LRUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)

def testCase():
    capacity = 2
    lru = LRUCache(capacity)
    # ret = [lru.put(1, 1) ,lru.put(2, 2), lru.get(1), lru.put(3, 3), lru.get(2),
```

```
lru.put(4, 4), lru.get(1), lru.get(3), lru.get(4)]
    ret1 = [lru.put(2, 1), lru.put(2, 2), lru.get(2) ,lru.put(1, 1), lru.put(4,
1), lru.get(2)]
    print(ret1)


if __name__ == "__main__":
    testCase()
```

341. 扁平化嵌套列表迭代器

```
# """
# This is the interface that allows for creating nested lists.
# You should not implement it, or speculate about its implementation
# """

# 341. 扁平化嵌套列表迭代器
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.nested_list = nestedList

    def next(self) -> int:
        return self.nested_list.pop(0).getInteger()

    def hasNext(self) -> bool:
        while len(self.nested_list) > 0 and not self.nested_list[0].isInteger():
            first_list = self.nested_list.pop(0).getList()
            for n in first_list[::-1]:
                self.nested_list.insert(0, n)
```

```
        return len(self.nested_list) > 0

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

380. O(1) 时间插入、删除和获取随机元素

```
# 380. O(1) 时间插入、删除和获取随机元素
import random
class RandomizedSet:

    def __init__(self):
        self.nums = []
        self.valToIdx = dict()

    def insert(self, val: int) -> bool:
        if self.valToIdx.get(val) is not None:
            return False
        self.valToIdx[val] = len(self.nums)
        self.nums.append(val)
        return True

    def remove(self, val: int) -> bool:
        if self.valToIdx.get(val) is None:
            return False

        idx = self.valToIdx.get(val)
        last_val = self.nums[-1]
        self.valToIdx[last_val] = idx
        self.nums[idx] = last_val

        self.nums.pop()
        del self.valToIdx[val]
        return True

    def getRandom(self) -> int:
        return self.nums[random.randint(0, len(self.nums)-1)]


# Your RandomizedSet object will be instantiated and called as such:
# obj = RandomizedSet()
# param_1 = obj.insert(val)
# param_2 = obj.remove(val)
# param_3 = obj.getRandom()

def testCase():
    rs_sol = RandomizedSet()
    ret = [rs_sol.insert(1), rs_sol.remove(2), rs_sol.insert(2),
rs_sol.getRandom(), rs_sol.remove(1), rs_sol.insert(2), rs_sol.getRandom()]
```

```
    ret = [rs_sol.insert(0), rs_sol.insert(1), rs_sol.remove(0), rs_sol.insert(2),
rs_sol.remove(1), rs_sol.getRandom()]

    print(ret)



if __name__ == "__main__":
    testCase()
```

460. LFU 缓存

```
from collections import OrderedDict
# 460. LFU 缓存
class LFUCache:
    def __init__(self, capacity: int):
        self.key_to_value = dict()
        self.key_to_freq = dict()
        self.freq_to_order_key = dict()
        self.capacity = capacity
        self.min_freq = 0

        self.cum_time = 0

    def get(self, key: int) -> int:
        if key not in self.key_to_value:
            return -1
        self.updateFreq(key)
        return self.key_to_value.get(key)

    def put(self, key: int, value: int) -> None:
        if self.capacity <= 0:
            return 0

        if self.key_to_value.get(key) is None:
            self.balanceFreq(key, value)
        else:
            self.key_to_value[key] = value
            self.updateFreq(key)


    def updateFreq(self, key):
        if self.key_to_freq.get(key) is None:
            self.key_to_freq[key] = 1
            if self.freq_to_order_key.get(1) is None:
                self.freq_to_order_key[1] = OrderedDict([(key, 1)])
            else:
                self.freq_to_order_key[1][key] = 1
            # 维护min_freq
            self.min_freq = 1
```

```
        else:
            # 更新freq, 删除老的freq
            cur_freq = self.key_to_freq[key]
            if self.freq_to_order_key.get(cur_freq) is not None:
                if key in self.freq_to_order_key.get(cur_freq):
                    del self.freq_to_order_key[cur_freq][key]
                    if len(self.freq_to_order_key.get(cur_freq)) == 0:
                        del self.freq_to_order_key[cur_freq]
                        # 维护min_freq
                        if cur_freq == self.min_freq:
                            self.min_freq = cur_freq + 1

            # 添加新的freq
            cur_freq += 1
            self.key_to_freq[key] = cur_freq
            if self.freq_to_order_key.get(cur_freq) is None:
                self.freq_to_order_key[cur_freq] = OrderedDict([(key, 1)])
            else:
                self.freq_to_order_key[cur_freq][key] = 1


    def balanceFreq(self, key, value):
        if len(self.key_to_value) == self.capacity:
            olden_key =
self.getOrderDictTopKey(self.freq_to_order_key.get(self.min_freq))
            del self.freq_to_order_key.get(self.min_freq)[olden_key]
            if len(self.freq_to_order_key.get(self.min_freq)) == 0:
                del self.freq_to_order_key[self.min_freq]
            del self.key_to_value[olden_key]
            del self.key_to_freq[olden_key]

        self.key_to_value[key] = value
        self.updateFreq(key)

    def getOrderDictTopKey(self, orderDict):
        for key in orderDict.keys():
            return key



# Your LFUCache object will be instantiated and called as such:
# obj = LFUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)


def testCase():
    cache = LFUCache(2)
    ret = [cache.put(1, 1), cache.put(2, 2), cache.get(1),
    cache.put(3, 3), cache.get(2), cache.get(3), cache.put(4,4),
    cache.get(1), cache.get(3), cache.get(4)]

    print(ret)
```

```
def testCase1():
    import time
    cache = LFUCache(10000)
    t1 = time.time()
    for i in range(100000):
        key, value = i, i * 5
        cache.put(key, value)
    for i in range(1000000):
        cache.get(key)
    t2 = time.time()
    delta = (t2 - t1) * 1000
    print(delta)
    print('cum_time:', cache.cum_time)
# [null,null,null,1,null,-1,3,null,-1,3,4]

    #285998.8663196564

    #25608.806371688843

if __name__ == "__main__":
    testCase1()
```

895. 最大频率栈

```
#895. 最大频率栈
class FreqStack:

    def __init__(self):
        self.valToFreq = dict()
        self.freqToValDict = dict()
        self.max_freq = 0


    def push(self, val: int) -> None:
        freq = self.valToFreq.get(val, 0) + 1
        self.valToFreq[val] = freq
        if self.freqToValDict.get(freq) is None:
            self.freqToValDict[freq] = list()
        self.freqToValDict[freq].append(val)
        self.max_freq = max(self.max_freq, freq)


    def pop(self) -> int:
        val = self.freqToValDict.get(self.max_freq).pop()
        freq = self.valToFreq.get(val) -1
        self.valToFreq[val] = freq
        if len(self.freqToValDict.get(self.max_freq)) == 0:
            del self.freqToValDict[self.max_freq]
            self.max_freq -= 1
        return val
```

```
# Your FreqStack object will be instantiated and called as such:
# obj = FreqStack()
# obj.push(val)
# param_2 = obj.pop()
```