Problem 56:

区间合并：

给出一个区间的集合，请合并所有重叠的区间。

示例 1:

输入: [[1,3],[2,6],[8,10],[15,18]]

输出: [[1,6],[8,10],[15,18]]

解释: 区间 [1,3] 和 [2,6] 重叠, 将它们合并为 [1,6].

示例 2:

输入: [[1,4],[4,5]]

输出: [[1,5]]

解释: 区间 [1,4] 和 [4,5] 可被视为重叠区间。

题解：对输入序列进行以第一个元素为key值的排序操作，并以第一个区间作为初始序列，然后遍历序列，判断i+1个区间和第i个生成区间的关系，即i+1.begin和i.end之间的大小，若begin<=end，则合并，更新生成区间。否则将区间放入结果，重新构造该区间为当前区间。到最后一个区间判断完成后加入到结果序列中。

代码：

class Solution(object):

    def merge(self, intervals):

        """

        :type intervals: List[List[int]]

        :rtype: List[List[int]]

        """

        if intervals == []:

            return []

        ans = []

        intervals.sort(key = lambda num:num[0])

        ss = intervals

        start = ss[0][0]

        end = ss[0][1]

        tmp = [start]

        for i in range(1, len(ss), 1):

            if ss[i][0] <= end:

                end = max(end, ss[i][1])

            else:

                tmp.append(end)

                ans.append(tmp)

                tmp = [ss[i][0]]

                end = ss[i][1]

        tmp.append(end)

        ans.append(tmp)

        return ans

57插入区间

给出一个无重叠的 ，按照区间起始端点排序的区间列表。

在列表中插入一个新的区间，你需要确保列表中的区间仍然有序且不重叠（如果有必要的话，可以合并区间）。

示例 1:

输入: intervals = [[1,3],[6,9]], newInterval = [2,5]

输出: [[1,5],[6,9]]

示例 2:

输入: intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]

输出: [[1,2],[3,10],[12,16]]

解释: 这是因为新的区间 [4,8] 与 [3,5],[6,7],[8,10] 重叠。

题解：解法一：直接合并区间到原区间，使用56的题解方法。

解法二：找到新区间所在的区间。即找到原区间序列中end<=newI.start和begin>=newI.end的区间，然后合并min(最左边)和max(最右边)之间的所有区间。其余区间不变直接加入。最终排序。

class Solution(object):

    def insert(self, intervals, newInterval):

        """

        :type intervals: List[List[int]]

        :type newInterval: List[int]

        :rtype: List[List[int]]

        """

        if intervals == []:

            return [newInterval]

        ans = []

        start = newInterval[0]

        end = newInterval[1]

        for i in range(len(intervals)):

            if intervals[i][1] < start or intervals[i][0] > end:

                ans.append(intervals[i])

            elif intervals[i][1] >= start:

                start = min(start, intervals[i][0])

                end = max(end, intervals[i][1])

            elif intervals[i][0] <= end:

                start = min(start, intervals[i][0])

                end = max(end, intervals[i][1])

        ans.append([start, end])

        ans.sort(key = lambda x:x[0])

        return ans

75颜色分类

给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

此题中，我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

注意:

不能使用代码库中的排序函数来解决这道题。

示例:

输入: [2,0,2,1,1,0]

输出: [0,0,1,1,2,2]

进阶：

一个直观的解决方案是使用计数排序的两趟扫描算法。

首先，迭代计算出0、1 和 2 元素的个数，然后按照0、1、2的排序，重写当前数组。

你能想出一个仅使用常数空间的一趟扫描算法吗？

题解：一趟扫描，首先解题的关键在于理解，通过互换0和2的位置可以得到最终解。然后维护两个指针，p0指向当前数组中0的最右边界，即从左起不是0的第一个位置，p2指向2的最左边界。当前指针为i，

while I < p2: //大于p2时表示右侧均为2，不需要移动

if nums[i] == 2: swap(nums[i], nums[p2]), p2-- //将2置换到右侧，但换完后nums[i]扔有可能为2，所以i不移动

if nums[i] == 0: swap(nums[i], nums[p0]), p0++, i++ //将0置换到左侧，且此时p0左侧均为0，i左侧均为1或0，所以i右移，

if nums[i] == 1: i++

这个算法也叫做3-way-partition，伪代码如下：

**procedure** three-way-partition(A : array of values, mid : value):

i ← 0

j ← 0

n ← **size of** A - 1

**while** j ≤ n:

**if** A[j] < mid:

**swap** A[i] and A[j]

i ← i + 1

j ← j + 1

**else if** A[j] > mid:

**swap** A[j] and A[n]

n ← n - 1

**else**:

j ← j + 1

代码：

class Solution(object):

    def sortColors(self, nums):

        """

        :type nums: List[int]

        :rtype: None Do not return anything, modify nums in-place instead.

        """

        if nums == []:

            return nums

        length = len(nums)

        index\_0 = 0

        index\_2 = length-1

        i = 0

        while i <= index\_2:

            if nums[i] == 2:

                nums[i] = nums[index\_2]

                nums[index\_2] = 2

                index\_2 -= 1

            elif nums[i] == 0:

                nums[i] = nums[index\_0]

                nums[index\_0] = 0

                index\_0 += 1

                i += 1

            else:

                i += 1

147

对链表进行插入排序。

插入排序的动画演示如上。从第一个元素开始，该链表可以被认为已经部分排序（用黑色表示）。

每次迭代时，从输入数据中移除一个元素（用红色表示），并原地将其插入到已排好序的链表中。

插入排序算法：

插入排序是迭代的，每次只移动一个元素，直到所有元素可以形成一个有序的输出列表。

每次迭代中，插入排序只从输入数据中移除一个待排序的元素，找到它在序列中适当的位置，并将其插入。

重复直到所有输入数据插入完为止。

示例 1：

输入: 4->2->1->3

输出: 1->2->3->4

示例 2：

输入: -1->5->3->4->0

输出: -1->0->3->4->5

题解：主要是对链表使用的考察。可以生成虚拟head用于遍历，每次找到要插入位置之前的元素，即第一个大于要插入元素值的元素，然后插入即可。可以使用pre表示当前节点的父节点降低插入难度。

# Definition for singly-linked list.

# class ListNode(object):

#     def \_\_init\_\_(self, x):

#         self.val = x

#         self.next = None

class Solution(object):

    def insertionSortList(self, head):

        """

        :type head: ListNode

        :rtype: ListNode

        """

        if head == None:

            return None

        tmp = ListNode(0)

        tmp.next = head

        pre, cur = head, head.next

        while cur is not None:

            if cur.val > pre.val:

                cur = cur.next

                pre = pre.next

            else:

                tmpNode = cur.next

                insert\_tmp = tmp

                while insert\_tmp.next.val < cur.val:

                    insert\_tmp = insert\_tmp.next

                cur.next = insert\_tmp.next

                insert\_tmp.next = cur

                cur = tmpNode

                pre.next = cur

        return tmp.next

179

给定一组非负整数，重新排列它们的顺序使之组成一个最大的整数。

示例 1:

输入: [10,2]

输出: 210

示例 2:

输入: [3,30,34,5,9]

输出: 9534330

说明: 输出结果可能非常大，所以你需要返回一个字符串而不是整数。

题解：1.将数字转换为字符串之后，对字符串进行降序排序。需要考虑到的特殊情况为，首数字相同时，将长度较短的字符串补全首数字为最长长度。因为3，34，30的顺序应该为34，3，30得到最大数字，短字符的位置与首数字相关。通过记录补全后排序的数字位置，生成结果时引用原数组构建。此外，对于生成最长字串存在相等的情况，做如下判断：

以12，121为例，第二位字符大于第一位且12长度更短，所以12在121之前。即

if num[1]>num[0]：

len(num\_x)<len(num\_y):

return 1 //不需要交换

else:

return -1//交换

else:

情况与上述相反

代码：

class Solution(object):

    def largestNumber(self, nums):

        """

        :type nums: List[int]

        :rtype: str

        """

        tmp = [str(num) for num in nums]

        maxLength = max([len(num) for num in tmp])

        ans = []

        for i in range(len(tmp)):

            node = Node(tmp[i], tmp[i], i)

            cur\_len = len(tmp[i])

            for \_ in range(maxLength - cur\_len):

                node.val += tmp[i][0]

            ans.append(node)

        # ans.sort(key=lambda x:x.val, reverse=True)

        ans = sorted(ans, key=cmp\_to\_key(cmp))

        n = [s.index for s in ans]

        res = ""

        for i in n:

            res += tmp[i]

        res = str(int(res))

        return res

class Node(object):

    def \_\_init\_\_(self, x, origin, index):

        self.val = x

        self.index = index

        self.origin = origin

def cmp\_to\_key(mycmp):

    'Convert a cmp= function into a key= function'

    class K(object):

        def \_\_init\_\_(self, obj, \*args):

            self.obj = obj

        def \_\_lt\_\_(self, other):

            return mycmp(self.obj, other.obj) < 0

        def \_\_gt\_\_(self, other):

            return mycmp(self.obj, other.obj) > 0

        def \_\_eq\_\_(self, other):

            return mycmp(self.obj, other.obj) == 0

        def \_\_le\_\_(self, other):

            return mycmp(self.obj, other.obj) <= 0

        def \_\_ge\_\_(self, other):

            return mycmp(self.obj, other.obj) >= 0

        def \_\_ne\_\_(self, other):

            return mycmp(self.obj, other.obj) != 0

    return K

def cmp(x, y):

    if x.val > y.val:

        return -1

    elif x.val < y.val:

        return 1

    else:

        if len(x.val) > 1:

            if x.val[1] < x.val[0]:

                if len(x.origin) > len(y.origin):

                    return -1

                else:

                    return 1

            else:

                if len(x.origin) > len(y.origin):

                    return 1

                else:

                    return -1

242

给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。

示例 1:

输入: s = "anagram", t = "nagaram"

输出: true

示例 2:

输入: s = "rat", t = "car"

输出: false

说明:

你可以假设字符串只包含小写字母。

进阶:

如果输入字符串包含 unicode 字符怎么办？你能否调整你的解法来应对这种情况？

题解：字符串拆开排序。

class Solution(object):

    def isAnagram(self, s, t):

        """

        :type s: str

        :type t: str

        :rtype: bool

        """

        s = [s[i] for i in range(len(s))]

        t = [t[i] for i in range(len(t))]

        s.sort()

        t.sort()

        if s == t:

            return True

        else:

            return False

220

给定一个整数数组，判断数组中是否有两个不同的索引 i 和 j，使得 nums [i] 和 nums [j] 的差的绝对值最大为 t，并且 i 和 j 之间的差的绝对值最大为 ķ。

示例 1:

输入: nums = [1,2,3,1], k = 3, t = 0

输出: true

示例 2:

输入: nums = [1,0,1,1], k = 1, t = 2

输出: true

示例 3:

输入: nums = [1,5,9,1,5,9], k = 2, t = 3

输出: false

题解：k最大到1万，暴力不行。但t是个小值，可以先对数组排序，然后在num[i]-num[i]+t的范围内判断索引绝对值是否不大于k。

代码：

class Solution(object):

    def containsNearbyAlmostDuplicate(self, nums, k, t):

        """

        :type nums: List[int]

        :type k: int

        :type t: int

        :rtype: bool

        """

        length = len(nums)

        ans = [Node(nums[i], i) for i in range(length)]

        ans.sort(key=lambda x:x.val)

        for i in range(length):

            j = i+1

            while j < length:

                if ans[j].val - ans[i].val <= t:

                    if abs(ans[j].index - ans[i].index) <= k:

                        return True

                    else:

                        j += 1

                else:

                    break

        return False

class Node(object):

    def \_\_init\_\_(self, x, index):

        self.val = x

        self.index = index

148 排序链表

在 O(n log n) 时间复杂度和常数级空间复杂度下，对链表进行排序。

示例 1:

输入: 4->2->1->3

输出: 1->2->3->4

示例 2:

输入: -1->5->3->4->0

输出: -1->0->3->4->5

题解：O(nlgn)一看就是归并排序。但是链表的操作不是很熟悉。记录一下。

归并排序操作：

1. 将序列均分为两个子序列
2. 对子序列继续划分，直到序列为空或者只剩一个元素不需要排序
3. 对左右子序列按顺序合并

合并操作：申请新的数组，比较子序列大小，加入到新数组中，最后剩下的子序列直接加到新数组后。

链表的归并排序：关键点在序列中点的查找。可使用快慢指针。

fast,slow = head.next, slow，

while fast is not None and fast.next is not None:

fast, slow = fast.next.next, slow.next

循环结束时，slow指向中点元素。

子序列的划分：mid=slow.next表示中点的后一个，slow.next=None使序列分为两个子序列。

然后对head和mid进行递归划分。

合并：通过构建h=空节点方便合并。left, right中按顺序合并，最后将非空序列合并到h后。

代码：

# Definition for singly-linked list.

# class ListNode(object):

#     def \_\_init\_\_(self, x):

#         self.val = x

#         self.next = None

class Solution(object):

    def sortList(self, head):

        """

        :type head: ListNode

        :rtype: ListNode

        """

        if not head or not head.next:

            return head

        fast = head.next

        slow = head

        while fast is not None and fast.next is not None:

            fast = fast.next.next

            slow = slow.next

        mid, slow.next = slow.next, None

        left, right = self.sortList(head), self.sortList(mid)

        h = res = ListNode(0)

        while left and right:

            if left.val < right.val:

                h.next = left

                left = left.next

            else:

                h.next = right

                right = right.next

            h = h.next

        h.next = left if left else right

        return res.next

164 最大间距

给定一个无序的数组，找出数组在排序之后，相邻元素之间最大的差值。

如果数组元素个数小于 2，则返回 0。

示例 1:

输入: [3,6,9,1]

输出: 3

解释: 排序后的数组是 [1,3,6,9], 其中相邻元素 (3,6) 和 (6,9) 之间都存在最大差值 3。

示例 2:

输入: [10]

输出: 0

解释: 数组元素个数小于 2，因此返回 0。

说明:

你可以假设数组中所有元素都是非负整数，且数值在 32 位有符号整数范围内。

请尝试在线性时间复杂度和空间复杂度的条件下解决此问题。

题解：线性时间复杂度O(n)，想到是基数排序。对于序列，首先对个位数进行排序，然后对十位数、百位数…，得到的结果即为有序。

代码：

class Solution(object):

    def maximumGap(self, nums):

        """

        :type nums: List[int]

        :rtype: int

        """

        if len(nums) < 2:

            return 0

        maxNum = max(nums)

        mode = 1

        while maxNum > 0:

            nums.sort(key=lambda x:(x/mode)%10)

            mode \*= 10

            maxNum //= 10

        ans = 0

        for i in range(len(nums)-1):

            ans = max(ans, nums[i+1]-nums[i])

        return ans

349 交集

给定两个数组，编写一个函数来计算它们的交集。

示例 1:

输入: nums1 = [1,2,2,1], nums2 = [2,2]

输出: [2]

示例 2:

输入: nums1 = [4,9,5], nums2 = [9,4,9,8,4]

输出: [9,4]

说明:

输出结果中的每个元素一定是唯一的。

我们可以不考虑输出结果的顺序。

题解：数组转Set减少数据量，然后使用字典进行hash。

class Solution(object):

    def intersection(self, nums1, nums2):

        """

        :type nums1: List[int]

        :type nums2: List[int]

        :rtype: List[int]

        """

        d = {}

        ans = []

        nums1 = set(nums1)

        nums2 = set(nums2)

        for num in nums1:

            d[num] = 1

        for num in nums2:

            if d.get(num) == 1:

                ans.append(num)

        return ans

350 交集2

给定两个数组，编写一个函数来计算它们的交集。

示例 1:

输入: nums1 = [1,2,2,1], nums2 = [2,2]

输出: [2,2]

示例 2:

输入: nums1 = [4,9,5], nums2 = [9,4,9,8,4]

输出: [4,9]

说明：

输出结果中每个元素出现的次数，应与元素在两个数组中出现的次数一致。

我们可以不考虑输出结果的顺序。

进阶:

如果给定的数组已经排好序呢？你将如何优化你的算法？

如果 nums1 的大小比 nums2 小很多，哪种方法更优？

如果 nums2 的元素存储在磁盘上，磁盘内存是有限的，并且你不能一次加载所有的元素到内存中，你该怎么办？

题解：去掉上题的set操作改为计数。

class Solution(object):

    def intersect(self, nums1, nums2):

        """

        :type nums1: List[int]

        :type nums2: List[int]

        :rtype: List[int]

        """

        d = {}

        ans = []

        for num in nums1:

            d[num] = d.get(num, 0) + 1

        for num in nums2:

            if d.get(num, 0) >= 1:

                ans.append(num)

                d[num] -= 1

        return ans

274

给定一位研究者论文被引用次数的数组（被引用次数是非负整数）。编写一个方法，计算出研究者的 h 指数。

h 指数的定义: “h 代表“高引用次数”（high citations），一名科研人员的 h 指数是指他（她）的 （N 篇论文中）有 h 篇论文分别被引用了至少 h 次。（其余的 N - h 篇论文每篇被引用次数不多于 h 次。）”

示例:

输入: citations = [3,0,6,1,5]

输出: 3

解释: 给定数组表示研究者总共有 5 篇论文，每篇论文相应的被引用了 3, 0, 6, 1, 5 次。

  由于研究者有 3 篇论文每篇至少被引用了 3 次，其余两篇论文每篇被引用不多于 3 次，所以她的 h 指数是 3。

说明: 如果 h 有多种可能的值，h 指数是其中最大的那个。

题解：

1. H篇论文至少被引用了h次，将引用次数降序排序，如果存在nums[i]>=i+1，则说明前i篇论文（共有i+1篇）的引用次数至少为i+1，（因为数组索引是从0开始），则结果为i+1

class Solution(object):

    def hIndex(self, citations):

        """

        :type citations: List[int]

        :rtype: int

        """

        if len(citations) == 0:

            return 0

        ans = 0

        citations.sort(reverse=True)

        for i in range(len(citations)-1, -1, -1):

            if citations[i] > i:

                return i + 1

        return ans

1. O(n)解法，引用最多的论文次数大于论文数量时，将其修改为论文数量不影响最终结果。修改完成后使用计数排序，使用k表示引用数量，countk表示num中k的个数，Sk表示论文引用次数大于k的论文数，Sk=counti的和，i>=k。所以最大的满足k<=Sk的k即为结果h。

倒序搜索找到第一个满足Sk>=k的k即为结果。

class Solution(object):

    def hIndex(self, citations):

        """

        :type citations: List[int]

        :rtype: int

        """

        if len(citations) == 0:

            return 0

        length = len(citations)

        ans = [0 for \_ in range(length+1)]

        for num in citations:

            ans[min(num, length)] += 1

        k = length

        sk = ans[k]

        while sk < k:

            k -= 1

            sk += ans[k]

        return k

324 摆动排序

给定一个无序的数组 nums，将它重新排列成 nums[0] < nums[1] > nums[2] < nums[3]... 的顺序。

示例 1:

输入: nums = [1, 5, 1, 1, 6, 4]

输出: 一个可能的答案是 [1, 4, 1, 5, 1, 6]

示例 2:

输入: nums = [1, 3, 2, 2, 3, 1]

输出: 一个可能的答案是 [2, 3, 1, 3, 1, 2]

说明:

你可以假设所有输入都会得到有效的结果。

进阶:

你能用 O(n) 时间复杂度和 / 或原地 O(1) 额外空间来实现吗？

题解：

1. 普通解法：根据要求很容易看出来做法是对数组排序，然后将数组分成两半穿插重组。但是存在重复元素的问题。如果某一重复元素的个数>=(length+1)/2的情况，那么这个重复元素至少会在一个地方相邻。当然也就不存在重复解。但当[1,1,2,2,3,3]如果穿插后是[1,3,1,3,2,2]不符合，所以需要对拆分后的两部分进行逆序重组，得到[2,1,1],[3,3,2]，然后得到[2,3,1,3,1,2]。如果是奇数个数的话，则需要把逆序后的最后一个元素放到数组的最后。因为用到了排序，所以复杂度为O(nlgn)

代码：

class Solution(object):

    def wiggleSort(self, nums):

        """

        :type nums: List[int]

        :rtype: None Do not return anything, modify nums in-place instead.

        """

        if len(nums) == 1:

            return nums

        nums.sort()

        if len(nums) % 2 == 0:

            mid = len(nums) // 2

            x = nums[:mid]

            y = nums[mid:]

            for i in range(mid):

                nums[2\*i] = x[mid-i-1]

                nums[2\*i+1] = y[mid-i-1]

        else:

            mid = len(nums) // 2

            x = nums[:mid+1]

            y = nums[mid+1:]

            for i in range(mid):

                nums[2\*i] = x[mid-i]

                nums[2\*i+1] = y[mid-i-1]

            nums[-1] = x[0]

        return nums

也可以使用降序排列，直接交叉。

nums.sort(reverse=True)

mid = len(nums) // 2

nums[1::2],nums[0::2] = nums[:mid], nums[mid:]

利用python的list操作。

1. 数组的拆分并不需要排序，只需要找到中位数并且使的左侧数据小于中位数，而右侧大于，即可得到上述的效果。所以找到中位数后，将数组拆分后反序插入合并，时间复杂度为O(n)。使用快速搜索算法找到数组的中位数。

代码如下：

#搜索完成后，中位数在数组中点处，数组左侧<=中位数，右侧>=中位数。但是左右两侧的中位数位置不确定。

quickSearch(self, nums, begin, end, k):

        t = nums[end-1]

        i, j = begin, begin

        while j < end:

            if nums[j] <= t:

                tmp = nums[i]

                nums[i] = nums[j]

                nums[j] = tmp

                i += 1

                j += 1

            else:

                j += 1

        if i <= k:

            self.quickSearch(nums, i, end, k)

        elif i - 1 > k:

            self.quickSearch(nums, begin, i-1, k)

然后对数组进行三分算法（荷兰国旗）进行调整，使数组为左侧小于中位数，中间为等于中位数的数字，右侧大于中位数。最后进行逆序交叉组合。但是python实现的算法超时，还没有自带的sort函数快。

代码：

class Solution(object):

    def wiggleSort(self, nums):

        """

        :type nums: List[int]

        :rtype: None Do not return anything, modify nums in-place instead.

        """

        if len(nums) == 1:

            return nums

        # nums.sort()

        self.quickSearch(nums, 0, len(nums), len(nums)//2)

        mid = nums[len(nums)//2]

        i, j = 0,0

        end = len(nums)-1

        while j <= end:

            if nums[j] < mid:

                nums[i], nums[j] = nums[j], nums[i]

                i += 1

                j += 1

            elif nums[j] > mid:

                nums[j], nums[end] = nums[end], nums[j]

                end -= 1

            else:

                j += 1

        if len(nums) % 2 == 0:

            mid = len(nums) // 2

            x = nums[:mid]

            y = nums[mid:]

            for i in range(mid):

                nums[2\*i] = x[mid-i-1]

                nums[2\*i+1] = y[mid-i-1]

        else:

            mid = len(nums) // 2

            x = nums[:mid+1]

            y = nums[mid+1:]

            for i in range(mid):

                nums[2\*i] = x[mid-i]

                nums[2\*i+1] = y[mid-i-1]

            nums[-1] = x[0]

        return nums

    def quickSearch(self, nums, begin, end, k):

        t = nums[end-1]

        i, j = begin, begin

        while j < end:

            if nums[j] <= t:

                nums[i], nums[j] = nums[j], nums[i]

                i += 1

                j += 1

            else:

                j += 1

        if i <= k:

            self.quickSearch(nums, i, end, k)

        elif i - 1 > k:

            self.quickSearch(nums, begin, i-1, k)

1. 上述算法空间复杂度均为O(n)，因为要复制数组。可以使用虚拟地址将复杂度降为O(1)。通过定义A(i)=nums[(1+2\*i)%(n|1)]作为虚拟地址。

代码：

class Solution(object):

    def wiggleSort(self, nums):

        """

        :type nums: List[int]

        :rtype: None Do not return anything, modify nums in-place instead.

        """

        if len(nums) == 1:

            return nums

        # nums.sort()

        length = len(nums)

        self.quickSearch(nums, 0, length, length//2)

        mid = nums[length//2]

        i, j = 0,0

        end = length-1

        while j <= end:

            if nums[(1+2\*j)%(length|1)] > mid:

                nums[(1+2\*i)%(length|1)], nums[(1+2\*j)%(length|1)] = nums[(1+2\*j)%(length|1)], nums[(1+2\*i)%(length|1)]

                i += 1

                j += 1

            elif nums[(1+2\*j)%(length|1)] < mid:

                nums[(1 + 2 \* j) % (length | 1)], nums[(1 + 2 \* end) % (length | 1)] = nums[(1 + 2 \* end) % (length | 1)], \

                                                                                     nums[(1 + 2 \* j) % (length | 1)]

                end -= 1

            else:

                j += 1

        return nums

    def quickSearch(self, nums, begin, end, k):

        t = nums[end-1]

        i, j = begin, begin

        while j < end:

            if nums[j] <= t:

                nums[i], nums[j] = nums[j], nums[i]

                i += 1

                j += 1

            else:

                j += 1

        if i <= k:

            self.quickSearch(nums, i, end, k)

        elif i - 1 > k:

            self.quickSearch(nums, begin, i-1, k)

493 反转对

给定一个数组 nums ，如果 i < j 且 nums[i] > 2\*nums[j] 我们就将 (i, j) 称作一个重要翻转对。

你需要返回给定数组中的重要翻转对的数量。

示例 1:

输入: [1,3,2,3,1]

输出: 2

示例 2:

输入: [2,4,3,5,1]

输出: 3

注意:

给定数组的长度不会超过50000。

输入数组中的所有数字都在32位整数的表示范围内。

题解：基础做法是逆序对。逆序对为i>j但是nums[i]<nums[j]的一对。求逆序对的个数可以在归并排序上进行改进。将数组划分为两个部分，分别统计左半部分、右半部分以及左右的逆序对。当右侧出一个元素时，左边剩余的所有元素和该元素形成逆序对，因为左侧元素大于它。

I=begin, j=mid+1

For k in range(begin, end+1):

If I > mid or j > end:

表示左半部分或右半部分已经为空，将剩余半部分合并过去

If nums[i]<nums[j]:

I += 1

Else:

J += 1

Count += (mid-i+1)

代码：

class Solution(object):

    def reversePairs(self, nums):

        """

        :type nums: List[int]

        :rtype: int

        """

        if len(nums) == 0:

            return 0

        self.inverse\_pair(nums)

    def inverse\_pair(self, nums):

        l = len(nums)

        self.merge\_sort(nums, 0, l-1)

        print(nums)

    def merge\_sort(self, nums, begin, end):

        if begin >= end:

            return

        mid = begin + (end - begin) // 2

        self.merge\_sort(nums, begin, mid)

        self.merge\_sort(nums, mid+1, end)

        self.merge(nums, begin, mid, end)

        return

    def merge(self, nums, begin, mid, end):

        n1 = mid - begin + 1

        n2 = end - mid

        a = [0 for i in range(0, n1, 1)]

        b = [0 for i in range(0, n2, 1)]

        for i in range(0, n1, 1):

            a[i] = nums[i + begin]

        for i in range(0, n2, 1):

            b[i] = nums[i + 1 + mid]

        i = 0

        j = 0

        for k in range(begin, end+1, 1):

            if j >= n2 or (i < n1 and a[i] <= b[j]):

                nums[k] = a[i]

                i += 1

            else:

                nums[k] = b[j]

                j += 1

本题题解：在归并排序的基础上，进行翻转对的判断。归并排序在得到有序结果后，可以在合并阶段之前，对左半部分和右半部分进行统计，得到翻转对的数量之后，再进行合并。可以理解为翻转对的数量只是归并排序的副产物。

代码：

class Solution(object):

    def reversePairs(self, nums):

        """

        :type nums: List[int]

        :rtype: int

        """

        if len(nums) == 0:

            return 0

        return self.inverse\_pair(nums)

    def inverse\_pair(self, nums):

        l = len(nums)

        return self.merge\_sort(nums, 0, l-1)

    def merge\_sort(self, nums, begin, end):

        if begin >= end:

            return 0

        mid = begin + (end - begin) // 2

        count = self.merge\_sort(nums, begin, mid) + self.merge\_sort(nums, mid+1, end)

        i = begin

        j = mid + 1

        while i <= mid and j <= end:

            if nums[i] > 2 \* nums[j]:

                j += 1

                count += (mid - i + 1)

            else:

                i += 1

        self.merge(nums, begin, mid, end)

        return count

    def merge(self, nums, begin, mid, end):

        n1 = mid - begin + 1

        n2 = end - mid

        a = [0 for i in range(0, n1, 1)]

        b = [0 for i in range(0, n2, 1)]

        for i in range(0, n1, 1):

            a[i] = nums[i + begin]

        for i in range(0, n2, 1):

            b[i] = nums[i + 1 + mid]

        i = 0

        j = 0

        for k in range(begin, end+1, 1):

            if j >= n2 or (i < n1 and a[i] <= b[j]):

                nums[k] = a[i]

                i += 1

            else:

                nums[k] = b[j]

                j += 1

题解2：树状数组（待学）

315

给定一个整数数组 nums，按要求返回一个新数组 counts。数组 counts 有该性质： counts[i] 的值是  nums[i] 右侧小于 nums[i] 的元素的数量。

示例:

输入: [5,2,6,1]

输出: [2,1,1,0]

解释:

5 的右侧有 2 个更小的元素 (2 和 1).

2 的右侧仅有 1 个更小的元素 (1).

6 的右侧有 1 个更小的元素 (1).

1 的右侧有 0 个更小的元素.

题解：看到数后面元素小于当前元素的数量，可以联想到通过归并排序可以做到。

If nums[left] <= nums[right]:

左侧元素更小，那么左侧出列的时候，记录右侧有多少出列的元素就是右侧更小的元素数量

Right++

Res[index[i]] += right – mid – 1

Else: 右侧出列不需要管

Left++

关键在于通过索引数组记录原数组的索引，这样在排序完成后原数组的数字不变，通过索引数组就可以拿到排好序的数组。记录索引后，在每次出列的时候不更新数组元素，而是更新数组索引。

class Solution(object):

    def countSmaller(self, nums):

        """

        :type nums: List[int]

        :rtype: List[int]

        """

        if len(nums) == 0:

            return []

        if len(nums) == 1:

            return [0]

        return self.inverse\_pair(nums)

    def inverse\_pair(self, nums):

        l = len(nums)

        tmp = [0 for \_ in range(l)]

        index = [i for i in range(l)]

        res = [0 for i in range(l)]

        self.merge\_sort(nums, 0, l-1, tmp, index, res)

        return res

    def merge\_sort(self, nums, begin, end, tmp, index, res):

        if begin >= end:

            return 0

        mid = begin + (end - begin) // 2

        self.merge\_sort(nums, begin, mid, tmp, index, res)

        self.merge\_sort(nums, mid+1, end, tmp, index, res)

        # i = begin

        # j = mid + 1

        self.merge(nums, begin, mid, end, tmp, index, res)

    def merge(self, nums, begin, mid, end, tmp, index, res):

        for i in range(begin, end+1):

            tmp[i] = index[i]

        i = begin

        j = mid + 1

        for k in range(begin, end+1, 1):

            if i > mid: #左侧数据全部出列

                index[k] = tmp[j]

                j += 1

            elif j > end: #右侧数据全部出列，此时左边剩下的所有元素都比右侧大，要加上右侧所有的数量

                index[k] = tmp[i]

                i += 1

                res[index[k]] += (end - mid)

            elif nums[tmp[i]] <= nums[tmp[j]]: #左侧出列，出列的该元素比右侧已出列的所有元素都大，记录数量

                index[k] = tmp[i]

                i += 1

                res[index[k]] += (j - mid - 1)

            else:

                index[k] = tmp[j]

                j += 1

题解2：线段数（待学）

922

给定一个非负整数数组 A， A 中一半整数是奇数，一半整数是偶数。

对数组进行排序，以便当 A[i] 为奇数时，i 也是奇数；当 A[i] 为偶数时， i 也是偶数。

你可以返回任何满足上述条件的数组作为答案。

示例：

输入：[4,2,5,7]

输出：[4,5,2,7]

解释：[4,7,2,5]，[2,5,4,7]，[2,7,4,5] 也会被接受。

提示：

2 <= A.length <= 20000

A.length % 2 == 0

0 <= A[i] <= 1000

class Solution(object):

    def sortArrayByParityII(self, A):

        """

        :type A: List[int]

        :rtype: List[int]

        """

        l = len(A)

        if l == 0:

            return []

        ans = [0 for \_ in range(l)]

        index\_even = 0

        index\_odd = 1

        for i in range(l):

            if A[i] % 2 == 0:

                ans[index\_even] = A[i]

                index\_even += 2

            else:

                ans[index\_odd] = A[i]

                index\_odd += 2

        return ans

976

给定由一些正数（代表长度）组成的数组 A，返回由其中三个长度组成的、**面积不为零**的三角形的最大周长。

如果不能形成任何面积不为零的三角形，返回 0。

题解：三角形a,b,c满足c>b>a，则构成三角形面积大于0的条件是c<b+a(c-b<a)，所以只要降序排列后找到第一个满足条件的三角形。并且c,b,a不能满足条件的时候，任何小于a的边长都不能构成三角形，所以需要向后移。

class Solution(object):

    def largestPerimeter(self, A):

        """

        :type A: List[int]

        :rtype: int

        """

        A.sort(reverse=True)

        for i in range(0, len(A)-2, 1):

            if A[i] < A[i+1] + A[i+2]:

                return A[i] + A[i+1] + A[i+2]

        return 0

973

我们有一个由平面上的点组成的列表 points。需要从中找出 K 个距离原点 (0, 0) 最近的点。

（这里，平面上两点之间的距离是欧几里德距离。）

你可以按任何顺序返回答案。除了点坐标的顺序之外，答案确保是唯一的。

示例 1：

输入：points = [[1,3],[-2,2]], K = 1

输出：[[-2,2]]

解释：

(1, 3) 和原点之间的距离为 sqrt(10)，

(-2, 2) 和原点之间的距离为 sqrt(8)，

由于 sqrt(8) < sqrt(10)，(-2, 2) 离原点更近。

我们只需要距离原点最近的 K = 1 个点，所以答案就是 [[-2,2]]。

示例 2：

输入：points = [[3,3],[5,-1],[-2,4]], K = 2

输出：[[3,3],[-2,4]]

（答案 [[-2,4],[3,3]] 也会被接受。）

题解：求出距离，排序得到解。

class Solution(object):

    def kClosest(self, points, K):

        """

        :type points: List[List[int]]

        :type K: int

        :rtype: List[List[int]]

        """

        nodes = [node(x[0], x[1]) for x in points]

        nodes.sort(key=lambda x:x.dis)

        ans = [[nodes[i].x, nodes[i].y] for i in range(K)]

        return ans

class node(object):

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

        self.dis = self.get\_dis(x, y)

    def get\_dis(self, x, y):

        return math.sqrt(x\*\*2 + y\*\*2)

767

给定一个字符串S，检查是否能重新排布其中的字母，使得两相邻的字符不同。

若可行，输出任意可行的结果。若不可行，返回空字符串。

示例 1:

输入: S = "aab"

输出: "aba"

示例 2:

输入: S = "aaab"

输出: ""

注意:

S 只包含小写字母并且长度在[1, 500]区间内。

题解：对字符出现的次数降序排列，拆分成左右两部分交叉重组。如果出现最多的次数>(length+1)/2，那么不会有结果。

class Solution(object):

    def reorganizeString(self, S):

        """

        :type S: str

        :rtype: str

        """

        l = len(S)

        nums = {}

        for i in range(l):

            nums[S[i]] = nums.get(S[i], 0) + 1

        x = sorted(nums.items(), key=lambda x:x[1], reverse=True)

        if x[0][1] > (l + 1) // 2:

            return ""

        ans = ""

        for key in x:

            for i in range(key[1]):

                ans += key[0]

        # print(ans)

        res = ""

        mid = l // 2

        if l % 2 == 0:

            for i in range(mid):

                res = res + ans[i] + ans[i+mid]

        else:

            for i in range(mid):

                res = res + ans[i] + ans[i+mid+1]

            res += ans[mid]

        return res

853

N  辆车沿着一条车道驶向位于 target 英里之外的共同目的地。

每辆车 i 以恒定的速度 speed[i] （英里/小时），从初始位置 position[i] （英里） 沿车道驶向目的地。

一辆车永远不会超过前面的另一辆车，但它可以追上去，并与前车以相同的速度紧接着行驶。

此时，我们会忽略这两辆车之间的距离，也就是说，它们被假定处于相同的位置。

车队 是一些由行驶在相同位置、具有相同速度的车组成的非空集合。注意，一辆车也可以是一个车队。

即便一辆车在目的地才赶上了一个车队，它们仍然会被视作是同一个车队。

会有多少车队到达目的地?

示例：

输入：target = 12, position = [10,8,0,5,3], speed = [2,4,1,1,3]

输出：3

解释：

从 10 和 8 开始的车会组成一个车队，它们在 12 处相遇。

从 0 处开始的车无法追上其它车，所以它自己就是一个车队。

从 5 和 3 开始的车会组成一个车队，它们在 6 处相遇。

请注意，在到达目的地之前没有其它车会遇到这些车队，所以答案是 3。

提示：

0 <= N <= 10 ^ 4

0 < target <= 10 ^ 6

0 < speed[i] <= 10 ^ 6

0 <= position[i] < target

所有车的初始位置各不相同。

题解：追击问题中，相遇只存在于speed[i]>speed[j]但position[i]<position[j]，但是知道speed其实并没有太大帮助，反而更重要的是到达目的地的time，因为还需要通过time来判断是否在到达终点之前就相遇。所以可以记录每辆车到终点的时间和当前位置。对位置降序排列，只有位置小但是需要时间更短的车才会相遇，并且相遇后保持匀速，不会第二次相遇。所以查找出time序列的递降序列个数就是结果。

class Solution:

    def carFleet(self, target: int, position: List[int], speed: List[int]) -> int:

        l = len(position)

        if l == 0:

            return 0

        if l == 1:

            return 1

        cars = [car(position[i], (target - position[i]) / speed[i]) for i in range(l)]

        cars.sort(key=lambda x:x.p, reverse=True)

        # for c in cars:

            # print(c.t)

            # print(c.p)

        res = 1

        cur\_t = cars[0].t

        # cur\_p = cars[0].p

        for i in range(1, l, 1):

            if cur\_t >= cars[i].t:

                pass

            else:

                cur\_t = cars[i].t

                res += 1

        # if cars[-1].t > cars[-2].t:

        #     res += 1

        return res

class car(object):

    def \_\_init\_\_(self, p, t):

        self.p = p

        self.t = t

524

给定一个字符串和一个字符串字典，找到字典里面最长的字符串，该字符串可以通过删除给定字符串的某些字符来得到。如果答案不止一个，返回长度最长且字典顺序最小的字符串。如果答案不存在，则返回空字符串。

示例 1:

输入:

s = "abpcplea", d = ["ale","apple","monkey","plea"]

输出:

"apple"

示例 2:

输入:

s = "abpcplea", d = ["a","b","c"]

输出:

"a"

说明:

所有输入的字符串只包含小写字母。

字典的大小不会超过 1000。

所有输入的字符串长度不会超过 1000。

题解：对序列排序，使得按长度降序并且按字典序升序。然后判断是否存在s的子串。

使用第二种定义cmp函数的方式，即functools.cmp\_to\_key()

class Solution:

    def findLongestWord(self, s: str, d: List[str]) -> str:

        d.sort(key=functools.cmp\_to\_key(cmp))

        # print(d)

        for i in range(len(d)):

            if self.\_is\_sub\_str(s, d[i]):

                return d[i]

        return ""

    def \_is\_sub\_str(self, s, d):

        i, j, l1, l2 = 0, 0, len(s), len(d)

        while i < l1 and j < l2:

            if s[i] == d[j]:

                i += 1

                j += 1

            else:

                i += 1

        if j >= l2:

            return True

        else:

            return False

import functools

def cmp(a, b):

    if len(a) == len(b):

        if a > b:

            return 1

        else:

            return -1

    elif len(a) < len(b):

        return 1

    else:

        return -1

1030

给出 R 行 C 列的矩阵，其中的单元格的整数坐标为 (r, c)，满足 0 <= r < R 且 0 <= c < C。

另外，我们在该矩阵中给出了一个坐标为 (r0, c0) 的单元格。

返回矩阵中的所有单元格的坐标，并按到 (r0, c0) 的距离从最小到最大的顺序排，其中，两单元格(r1, c1) 和 (r2, c2) 之间的距离是曼哈顿距离，|r1 - r2| + |c1 - c2|。（你可以按任何满足此条件的顺序返回答案。）

示例 1：

输入：R = 1, C = 2, r0 = 0, c0 = 0

输出：[[0,0],[0,1]]

解释：从 (r0, c0) 到其他单元格的距离为：[0,1]

示例 2：

输入：R = 2, C = 2, r0 = 0, c0 = 1

输出：[[0,1],[0,0],[1,1],[1,0]]

解释：从 (r0, c0) 到其他单元格的距离为：[0,1,1,2]

[[0,1],[1,1],[0,0],[1,0]] 也会被视作正确答案。

示例 3：

输入：R = 2, C = 3, r0 = 1, c0 = 2

输出：[[1,2],[0,2],[1,1],[0,1],[1,0],[0,0]]

解释：从 (r0, c0) 到其他单元格的距离为：[0,1,1,2,2,3]

其他满足题目要求的答案也会被视为正确，例如 [[1,2],[1,1],[0,2],[1,0],[0,1],[0,0]]。

提示：

1 <= R <= 100

1 <= C <= 100

0 <= r0 < R

0 <= c0 < C

题解：求出距离排序，给出坐标即可。实现方法较多。

class Solution:

    def allCellsDistOrder(self, R: int, C: int, r0: int, c0: int) -> List[List[int]]:

        points = []

        for i in range(R):

            for j in range(C):

                points.append(point(i, j, r0, c0))

        points.sort(key=lambda x:x.dis)

        res = [[x.x, x.y] for x in points]

        return res

class point(object):

    def \_\_init\_\_(self, x, y, x0, y0):

        self.x = x

        self.y = y

        self.dis = abs(x-x0) + abs(y-y0)

也可使用字典，省空间

1122

给你两个数组，arr1 和 arr2，

arr2 中的元素各不相同

arr2 中的每个元素都出现在 arr1 中

对 arr1 中的元素进行排序，使 arr1 中项的相对顺序和 arr2 中的相对顺序相同。未在 arr2 中出现过的元素需要按照升序放在 arr1 的末尾。

示例：

输入：arr1 = [2,3,1,3,2,4,6,7,9,2,19], arr2 = [2,1,4,3,9,6]

输出：[2,2,2,1,4,3,3,9,6,7,19]

提示：

arr1.length, arr2.length <= 1000

0 <= arr1[i], arr2[i] <= 1000

arr2 中的元素 arr2[i] 各不相同

arr2 中的每个元素 arr2[i] 都出现在 arr1 中

题解：记录arr2顺序，构造相应个数序列，然后未出现的直接排序加载后面。

class Solution:

    def relativeSortArray(self, arr1: List[int], arr2: List[int]) -> List[int]:

        a1 = {}

        a2 = []

        nums = {}

        for i in arr2:

            a1[i] = 1

        for i in arr1:

            if a1.get(i) == 1:

                nums[i] = nums.get(i, 0) + 1

            else:

                a2.append(i)

        a2.sort()

        res = []

        for i in arr2:

            res.extend([i for \_ in range(nums[i])])

        res.extend(a2)

        return res

1329

给你一个 m \* n 的整数矩阵 mat ，请你将同一条对角线上的元素（从左上到右下）按升序排序后，返回排好序的矩阵。

输入：mat = [[3,3,1,1],[2,2,1,2],[1,1,1,2]]

输出：[[1,1,1,1],[1,2,2,2],[1,2,3,3]]

提示：

m == mat.length

n == mat[i].length

1 <= m, n <= 100

1 <= mat[i][j] <= 100

题解： row-col=一样的值时表示他们在同一个正对角线上，将所有对角线的元素存到相应字典之后进行排序展开，就可以得到结果。关键是对python一些迭代器的熟练使用。collections.defaultdict()后跟数据类型，比如list、int等，可以为字典的元素赋初始值。

class Solution:

    def diagonalSort(self, mat: List[List[int]]) -> List[List[int]]:

        row = len(mat)

        if row == 0:

            return []

        if row == 1:

            return mat

        col = len(mat[0])

        d = collections.defaultdict(list)

        for i in range(row):

            for j in range(col):

                d[i-j].append(mat[i][j])

        d = {k : iter(sorted(v)) for k,v in d.items()}

        for i in range(row):

            for j in range(col):

                mat[i][j] = next(d[i-j])

        return mat

1333

给你一个餐馆信息数组 restaurants，其中  restaurants[i] = [idi, ratingi, veganFriendlyi, pricei, distancei]。你必须使用以下三个过滤器来过滤这些餐馆信息。

其中素食者友好过滤器 veganFriendly 的值可以为 true 或者 false，如果为 true 就意味着你应该只包括 veganFriendlyi 为 true 的餐馆，为 false 则意味着可以包括任何餐馆。此外，我们还有最大价格 maxPrice 和最大距离 maxDistance 两个过滤器，它们分别考虑餐厅的价格因素和距离因素的最大值。

过滤后返回餐馆的 id，按照 rating 从高到低排序。如果 rating 相同，那么按 id 从高到低排序。简单起见， veganFriendlyi 和 veganFriendly 为 true 时取值为 1，为 false 时，取值为 0 。

示例 1：

输入：restaurants = [[1,4,1,40,10],[2,8,0,50,5],[3,8,1,30,4],[4,10,0,10,3],[5,1,1,15,1]], veganFriendly = 1, maxPrice = 50, maxDistance = 10

输出：[3,1,5]

解释：

这些餐馆为：

餐馆 1 [id=1, rating=4, veganFriendly=1, price=40, distance=10]

餐馆 2 [id=2, rating=8, veganFriendly=0, price=50, distance=5]

餐馆 3 [id=3, rating=8, veganFriendly=1, price=30, distance=4]

餐馆 4 [id=4, rating=10, veganFriendly=0, price=10, distance=3]

餐馆 5 [id=5, rating=1, veganFriendly=1, price=15, distance=1]

在按照 veganFriendly = 1, maxPrice = 50 和 maxDistance = 10 进行过滤后，我们得到了餐馆 3, 餐馆 1 和 餐馆 5（按评分从高到低排序）。

示例 2：

输入：restaurants = [[1,4,1,40,10],[2,8,0,50,5],[3,8,1,30,4],[4,10,0,10,3],[5,1,1,15,1]], veganFriendly = 0, maxPrice = 50, maxDistance = 10

输出：[4,3,2,1,5]

解释：餐馆与示例 1 相同，但在 veganFriendly = 0 的过滤条件下，应该考虑所有餐馆。

示例 3：

输入：restaurants = [[1,4,1,40,10],[2,8,0,50,5],[3,8,1,30,4],[4,10,0,10,3],[5,1,1,15,1]], veganFriendly = 0, maxPrice = 30, maxDistance = 3

输出：[4,5]

提示：

1 <= restaurants.length <= 10^4

restaurants[i].length == 5

1 <= idi, ratingi, pricei, distancei <= 10^5

1 <= maxPrice, maxDistance <= 10^5

veganFriendlyi 和 veganFriendly 的值为 0 或 1 。

所有 idi 各不相同。

题解：拿到所有餐馆后排序。注意python中，if a > b: return -1是降序。

class Solution:

    def filterRestaurants(self, restaurants: List[List[int]], veganFriendly: int, maxPrice: int, maxDistance: int) -> List[int]:

        nodes = []

        for res in restaurants:

            if (not veganFriendly or res[2] == veganFriendly) and res[3] <= maxPrice and res[4] <= maxDistance:

                nodes.append(node(res[0], res[1]))

        import functools

        nodes.sort(key=functools.cmp\_to\_key(cmp))

        ans = [x.id for x in nodes]

        return ans

def cmp(a, b):

    if a.rate == b.rate:

        if a.id > b.id:

            return -1

        else:

            return 1

    elif a.rate > b.rate:

        return -1

    else:

        return 1

class node(object):

    def \_\_init\_\_(self, id, rate):

        self.id = id

        self.rate = rate

1305

给你 root1 和 root2 这两棵二叉搜索树。

请你返回一个列表，其中包含 两棵树 中的所有整数并按 升序 排序。.

输入：root1 = [2,1,4], root2 = [1,0,3]

输出：[0,1,1,2,3,4]

示例 2：

输入：root1 = [0,-10,10], root2 = [5,1,7,0,2]

输出：[-10,0,0,1,2,5,7,10]

示例 3：

输入：root1 = [], root2 = [5,1,7,0,2]

输出：[0,1,2,5,7]

题解：前序遍历树可以得到升序序列，然后对两个数进行归并。

class Solution:

    def getAllElements(self, root1: TreeNode, root2: TreeNode) -> List[int]:

        res1 = []

        res2 = []

        self.f(root1, res1)

        self.f(root2, res2)

        ans = []

        i, j = 0, 0

        while i < len(res1) and j < len(res2):

            if res1[i] < res2[j]:

                ans.append(res1[i])

                i += 1

            else:

                ans.append(res2[j])

                j += 1

        while i < len(res1):

            ans.append(res1[i])

            i += 1

        while j < len(res2):

            ans.append(res2[j])

            j += 1

        return ans

    def f(self, root, res):

        if root is not None:

            self.f(root.left, res)

            res.append(root.val)

            self.f(root.right, res)

1054

在一个仓库里，有一排条形码，其中第 i 个条形码为 barcodes[i]。

请你重新排列这些条形码，使其中两个相邻的条形码 不能 相等。 你可以返回任何满足该要求的答案，此题保证存在答案。

示例 1：

输入：[1,1,1,2,2,2]

输出：[2,1,2,1,2,1]

示例 2：

输入：[1,1,1,1,2,2,3,3]

输出：[1,3,1,3,2,1,2,1]

提示：

1 <= barcodes.length <= 10000

1 <= barcodes[i] <= 10000

题解：相邻数字不能相等，并且保证有答案，那么出现最多次数小于序列的1/2+1，比如7个则最多4个，否则没有答案。然后按照出现次数排序得到新序列，拆分成两个部分穿插组成新数组即可。重要的是其中对python排序、list切割操作的使用。

class Solution:

    def rearrangeBarcodes(self, barcodes: List[int]) -> List[int]:

        d = collections.Counter(barcodes)

        a = sorted(d.items(), key=lambda x:x[1],reverse=True)

        ans = []

        for i in a:

            ans += [i[0]] \* i[1]

        res = [0 for \_ in range(len(barcodes))]

        res[0::2] = ans[:len(res[0::2])]

        res[1::2] = ans[len(res[0::2]):]

        return res

969

给定数组 A，我们可以对其进行煎饼翻转：我们选择一些正整数 k <= A.length，然后反转 A 的前 k 个元素的顺序。我们要执行零次或多次煎饼翻转（按顺序一次接一次地进行）以完成对数组 A 的排序。

返回能使 A 排序的煎饼翻转操作所对应的 k 值序列。任何将数组排序且翻转次数在 10 \* A.length 范围内的有效答案都将被判断为正确。

示例 1：

输入：[3,2,4,1]

输出：[4,2,4,3]

解释：

我们执行 4 次煎饼翻转，k 值分别为 4，2，4，和 3。

初始状态 A = [3, 2, 4, 1]

第一次翻转后 (k=4): A = [1, 4, 2, 3]

第二次翻转后 (k=2): A = [4, 1, 2, 3]

第三次翻转后 (k=4): A = [3, 2, 1, 4]

第四次翻转后 (k=3): A = [1, 2, 3, 4]，此时已完成排序。

示例 2：

输入：[1,2,3]

输出：[]

解释：

输入已经排序，因此不需要翻转任何内容。

请注意，其他可能的答案，如[3，3]，也将被接受。

提示：

1 <= A.length <= 100

A[i] 是 [1, 2, ..., A.length] 的排列

题解：由于每次都要翻转前k个数，原理和冒泡排序类似，首先找到当前序列中最大的数，翻转到第一个，然后放大最后的位置，然后继续找除该元素之外的序列中最大值，依次翻转到第一个然后到当前序列的最后一个。需要注意的是翻转的位置，第一个是最大数的索引，到第一个位子，然后翻转序列长度到最后。

class Solution:

    def pancakeSort(self, A: List[int]) -> List[int]:

        res = []

        l = len(A)-1

        if l == 0:

            return []

        while l > 0:

            maxN = 0

            maxIndex = -1

            for i in range(l+1):

                if A[i] > maxN:

                    maxN = A[i]

                    maxIndex = i

            res.append(maxIndex+1)

            res.append(l+1)

            A[:maxIndex+1] = A[maxIndex::-1]

            A[:l+1] = A[l::-1]

            l -= 1

        print(A)

        return res

327

给定一个整数数组 nums，返回区间和在 [lower, upper] 之间的个数，包含 lower 和 upper。

区间和 S(i, j) 表示在 nums 中，位置从 i 到 j 的元素之和，包含 i 和 j (i ≤ j)。

说明:

最直观的算法复杂度是 O(n2) ，请在此基础上优化你的算法。

示例:

输入: nums = [-2,5,-1], lower = -2, upper = 2,

输出: 3

解释: 3个区间分别是: [0,0], [2,2], [0,2]，它们表示的和分别为: -2, -1, 2。

题解：要找到区间和，首先要得到连续和S, Sj-Si表示区间i+1到j的和，这样只需要一次减法就可以拿到区间和。在得到S后，原题变为求lower<=Sj-Si<=upper的解的个数，其中i<j。回想之前逆序对的求法，在归并过程中，每次从左边取得数之后，记录右侧的出列个数，就是逆序对的个数，即找到Sj-Si<0。所以本题也可以转化为归并排序求解逆序对个数的扩展问题，将Sj-Si<0的条件转换为lower<=Sj-Si<=upper，但是不一样的是，对于左边每个出列的元素，因为具有下界，所以右侧是一个区间，而不是只有上限。所以应该从右侧序列中，找到第一个减去left[i]之后不小于lower的索引作为下界，找到右侧第一个减去left[i]之后大于upper的索引作为上届，区间内的个数就是left[i]所对应的满足区间和限制的个数。对于left中的每一个数字，都要进行上述查找求和。找到满足的结果之后，再进行归并排序，得到后续结果。连续和数组使用了前置0简化，因为前置0在后续的减操作中并不影响结果。

正确性的证明：对于S数组，Sj-Si得到区间和，如果是求逆序对，在求得逆序对数量之后，归并排序仅是当前半部分的顺序，不改变递归之后整个序列前后元素的相对位置（前后），所以可以归并。而在本题中，合并之前存在i<j的关系，i属于左边，j属于右边，归并排序完成后依旧存在i<j的关系，并且左侧和右侧完成排序之后，改变的只是i-j的关系，比如区间[1-5]符合条件，但是排序后该区间位置变成了[2-3]，位置改变之后依旧是唯一的解中的一个，不会重复，改变的只是现在得到的排序数列的区间位置，个数是不变的。

class Solution:

    def countRangeSum(self, nums: List[int], lower: int, upper: int) -> int:

        n = len(nums)

        if n == 0 or nums is None:

            return 0

        sums = [0 for i in range(n+1)]

        for i in range(n):

            sums[i+1] = sums[i] + nums[i]

        return self.merge\_sort(sums, lower, upper)

    def merge\_sort(self, nums, lower, upper):

        if len(nums) <= 1:

            return 0

        n = len(nums)

        mid = n // 2

        left = nums[:mid]

        right = nums[mid:]

        cnt = self.merge\_sort(left, lower, upper) + self.merge\_sort(right, lower, upper)

        i = 0

        low = 0

        up = 0

        while i < mid:

            while low < len(right) and right[low] - left[i] < lower:

                low += 1

            while up < len(right) and right[up] - left[i] <= upper:

                up += 1

            cnt += up - low

            i += 1

        i, j, k = 0, 0, 0

        while i < len(left) and j < len(right):

            if left[i] < right[j]:

                nums[k] = left[i]

                i += 1

            else:

                nums[k] = right[j]

                j += 1

            k += 1

        while i < len(left):

            nums[k] = left[i]

            k += 1

            i += 1

        while j < len(right):

            nums[k] = right[j]

            k += 1

            j += 1

        return cnt

题解中的low和up只需要赋初值，因为low和up在右侧区间只会不变或增加，不需要从头开始。

1235

你打算利用空闲时间来做兼职工作赚些零花钱。

这里有 n 份兼职工作，每份工作预计从 startTime[i] 开始到 endTime[i] 结束，报酬为 profit[i]。

给你一份兼职工作表，包含开始时间 startTime，结束时间 endTime 和预计报酬 profit 三个数组，请你计算并返回可以获得的最大报酬。

注意，时间上出现重叠的 2 份工作不能同时进行。

如果你选择的工作在时间 X 结束，那么你可以立刻进行在时间 X 开始的下一份工作。

输入：startTime = [1,2,3,3], endTime = [3,4,5,6], profit = [50,10,40,70]

输出：120

解释：

我们选出第 1 份和第 4 份工作，

时间范围是 [1-3]+[3-6]，共获得报酬 120 = 50 + 70。

题解：这道题是一个动态规划问题，从题中分析，若想得到最大报酬，要么是得到区间内的最多子区间，要么是得到工资最高的子区间。但是贪心在这里行不通，因为区间最多并不一定工资最高。但是是一个最优子结构问题，首先对数组排序，对于当前工作i，打工的最大值一定是结束时间小于i.start的所有工作的工资和加上i的工资。设dp[i]表示当前工资和的话，dp[i]=max(dp[0],…dp[j])+i.profit, j<i。表示dp[i]应该是i之前所有与i不冲突的最大工资加上i的工资，因为最大工资区间不确定，所以dp[n]不是最优解，最优解是max(dp[i]), 0<i<=n。

代码：

class Solution:

    def jobScheduling(self, startTime: List[int], endTime: List[int], profit: List[int]) -> int:

        l = len(startTime)

        h = sorted(zip(startTime, endTime, profit))

        dp = [0 for \_ in range(l)]

        res = 0

        s = 0

        pos = 0

        for i in range(l):

            for j in range(pos, i):

                if h[i][0] >= h[j][1]:

                    if j == pos:

                        pos += 1

                    s = max(s, dp[j])

            dp[i] = s + h[i][2]

            res = max(res, dp[i])

        return res

程序中的pos用来缩小搜索范围，不然会超时。因为j是与i不冲突的工作，如果j是当前搜索的第一个工作，那么dp[i]的结果是已经包含了dp[j]的最优结果，所以dp[j]不需要再次计算，也就是动态规划的最优子结构性质。而下一层循环会计算到i的结果，i中包含了所有小于i的工作且与第i个工作不冲突的结果，所以可以不再计算之前的所有工作。

710

给定一个包含 [0，n ) 中独特的整数的黑名单 B，写一个函数从 [ 0，n ) 中返回一个不在 B 中的随机整数。

对它进行优化使其尽量少调用系统方法 Math.random() 。

提示:

1 <= N <= 1000000000

0 <= B.length < min(100000, N)

[0, N) 不包含 N，详细参见 interval notation 。

示例 1:

输入:

["Solution","pick","pick","pick"]

[[1,[]],[],[],[]]

输出: [null,0,0,0]

示例 2:

输入:

["Solution","pick","pick","pick"]

[[2,[]],[],[],[]]

输出: [null,1,1,1]

示例 3:

输入:

["Solution","pick","pick","pick"]

[[3,[1]],[],[],[]]

Output: [null,0,0,2]

示例 4:

输入:

["Solution","pick","pick","pick"]

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

输出: [null,1,3,1]

输入语法说明：

输入是两个列表：调用成员函数名和调用的参数。Solution的构造函数有两个参数，N 和黑名单 B。pick 没有参数，输入参数是一个列表，即使参数为空，也会输入一个 [] 空列表。

题解：看懂题之后，需要做的是从不在黑名单中的所有整数中随机返回一个，并且有n个。但是n数值太大，不能使用遍历方法，时间空间都超。

因为黑名单B和白名单W合为0-n的集合，所以不在黑名单B的整数，即白名单的数组长度为N-B.length，并且和黑名单没有交集。考虑到黑白名单加起来是0-n的集合，所以可以任选出一个数字k属于[0-W.length-1]，一定存在如此的k，是白名单中第k大的数字。然后可以使用二分方法查找出白名单第k大的数字，只不过需要和B做对比，并且B[mid]-mid表示白名单中小于mid的个数，所以可以找到白名单第k大的数字。

If B[mid]-mid > k: 说明小于mid的数字个数大于k，所以表示第k大的数字在左侧

High = mid-1

Else:

Low = mid

查找完成后，会有两种情况。第一是B中最小的数都大于k，所以W[k]就是k，即白名单第k大的数是k；第二种是W[k]>B[low]，此时的W[k] = k + low + 1，low是找到的满足W[k]>B[low]的索引，此时k+low就是所有小于W[k]的数字个数，所以W[k]是k+low+1。

import random

class Solution:

    b = None

    n = 0

    def \_\_init\_\_(self, N, blacklist):

        self.n = N

        blacklist.sort()

        self.b = blacklist

    def pick(self):

        k = random.randint(0, self.n-len(self.b)-1)

        low = 0

        high = len(self.b) - 1

        while low < high:

            mid = (low + high + 1) // 2

            if self.b[mid] - mid > k:

                high = mid - 1

            else:

                low = mid

        if low == high and self.b[low] - low <= k:

            return k + low + 1

        else:

            return k

# Your Solution object will be instantiated and called as such:

# obj = Solution(N, blacklist)

# param\_1 = obj.pick()