算法基础作业 1

作者：LYZS　学习交流使用

1 Leetcode 题目

伪代码部分选择 $a \leftarrow b, a = b$ 表示将 b 的值赋值给 a，　$\triangleright$ 表示注释

1.1 第一次作业

1.1.1 两数之和

给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出和为目标值 target 的那两个整数，并返回它们的数组下标。假设每种输入只会对应一个答案，不能使用两次相同的元素。可以按任意顺序返回答案。

示例 1：

输入：$nums = [2, 7, 11, 15], target = 9$ 输出：$[0, 1]$

解释：因为 $nums[0] + nums[1] == 9$ ，返回 $[0, 1]$。

示例 2：

输入：$nums = [3, 2, 4], target = 6$ 输出：$[1, 2]$

Algorithm 1 两数之和

输入: 一个整数数组 $nums$ 和一个整数目标值 $target$

输出: 一个包含和为目标值 $target$ 的那两个整数的数组下标的数组

$sum\_nums = \{\}$　$\triangleright$ 创建一个空字典

for i from 0 to $len(nums) - 1$ **do**

　$num \leftarrow nums[i]$

　$num\_new \leftarrow target - num$

　if num_new 在 sum_nums 中 **then**

　　return $[i, sum\_nums[num\_new]]$　$\triangleright$ 返回这两个数的索引

　end if

　$sum\_nums[num] \leftarrow i$

end for

return 空列表

时间复杂度：$O(n)$

算法使用了一个哈希表（用字典表示）来存储遍历过程中的元素值和它们的索引，使用一个 for 循环遍历数组 $nums$ 中的每个元素。这个循环的时间复杂度是 $O(n)$，其中 n 是数组 $nums$ 的长度，时间复杂

度为 $O(n)$。

1.1.2 三数之和

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

注意：答案中不可以包含重复的三元组。

示例 1：

输入：$nums = [-1, 0, 1, 2, -1, -4]$ 输出：$[[-1, -1, 2], [-1, 0, 1]]$

解释：$nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0,$

$nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0,$

$nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0$

不同的三元组是 [-1,0,1] 和 [-1,-1,2] 。

注意，输出的顺序和三元组的顺序并不重要。

示例 2：

输入：$nums = [0, 1, 1]$

输出：[]

解释：唯一可能的三元组和不为 0 。示例 3：

输入：$nums = [0, 0, 0]$

输出：$[[0, 0, 0]]$

解释：唯一可能的三元组和为 0 。

Algorithm 2 三数之和

输入: 一个整数数组 $nums$

输出: 所有和为 0 且不重复的三元数组

```
nums.sort()                                         ▷ 对 numms 排序
len(nums)
result = []                                         ▷ 创建一个空列表
for i from 0 to length − 1 do
    if i > 0 and nums[i] = nums[i − 1] then
        继续
    end if
    left ← i + 1
    right ← length − 1
    while left < right do
        total ← nums[i] + nums[left] + nums[right]
        if total < 0 then
            left ← left + 1
        else if total > 0 then
            right ← right − 1
        else
            result. 添加 [nums[i], nums[left], nums[right]]
            while left < right and nums[left] = nums[left + 1] do
                left ← left + 1
            end while
            while left < right and nums[right] = nums[right − 1] do
                right ← right − 1
            end while
            left ← left + 1
            right ← right − 1
        end if
    end while
end for
return result
```

时间复杂度：$O(n^2)$

首先对输入数组 $nums$ 进行排序, 排序的时间复杂度通常为 $O(n * logn)$，使用一个 for 循环遍历排序后的数组 $nums$。这个循环的时间复杂度是 $O(n)$，对于每个数组元素 $nums[i]$，算法使用两个指针 left 和 right 分别指向 $i+1$ 和数组末尾，然后向中间移动以寻找和为零的三元组，时间复杂度是 $O(n)$，最内层的循环（双指针搜索）最多执行 n 次（每个元素都可能启动一次搜索），而每次搜索最多遍历 n/2 个元素（数组已排序，所以每次找到一个有效的三元组后，可以移动 left 和 right 指针），算法的总体时间复杂度是 $O(n^2)$。

1.1.3 交易逆序对的总数

在股票交易中，如果前一天的股价高于后一天的股价，则可以认为存在一个「交易逆序对」。请设计一个程序，输入一段时间内的股票交易记录 record，返回其中存在的「交易逆序对」总数。

示例 1:

输入：record = [9, 7, 5, 4, 6]

输出：8

解释：交易中的逆序对为 (9, 7), (9, 5), (9, 4), (9, 6), (7, 5), (7, 4), (7, 6), (5, 4)。时间复杂度：$O(n^2)$

Algorithm 3 交易数组中的逆序对

输入: 一个数组 $nums$

输出: 逆序对的总个数

```
n ← len(nums)
total_count ← 0
for i from 0 to n − 1 do                          ▷ 遍历数组中的每个元素
    for j from i + 1 to n − 1 do                  ▷ 遍历数组中每个 nums[i] 之后的所有元素
        if nums[i] > nums[j] then
            total_count ← total_count + 1
        end if
    end for
end for
return total_count
```

外层循环遍历数组 $nums$ 中的每个元素。这个循环从索引 0 到 n-1，其中 n 是数组 $nums$ 的长度，对于索引 i 的元素，内层循环从索引 $i+1$ 到 $n-1$, 内层循环的总执行次数是所有索引 i 到 $n-1$ 的和，即 $(n-1)+(n-2)+...+1=n(n-1)/2$

显然由于时间复杂度过高，在 leetcode 上不能成功提交, 所以需要下面的改进。

Algorithm 4 交易数组中的逆序对 (第一部分)

输入: 一个数组 $record$

输出: 逆序对的总个数

$temp_a rr = []$ ▷ 创建一个长度与 $record$ 相同的临时数组，并初始化为 0

$count \leftarrow 0$

$count \leftarrow$ MergeSort$(record, 0, len(record) - 1, temp_arr)$

return $count$

Algorithm 5 交易数组中的逆序对 (第二部分)

function MergeSort$(nums, start, end, temp_arr)$▷ 将数组分成两半，直到每一半只有一个元素或没有, 并调用 $Merge$ 函数对数组做合并排序，计算合并过程中产生的逆序对数量

    if $start \geq end$ **then**

        return 0

    end if

    $mid \leftarrow (start + end)//2$

    $count \leftarrow 0$

    $count \leftarrow count+$ MergeSort$(nums, start, mid, temp_arr)$

    $count \leftarrow count+$ MergeSort$(nums, mid + 1, end, temp_arr)$

    $count \leftarrow count+$ Merge$(nums, start, mid, end, temp_arr)$

    return $count$

end function

第二部分的算法采用归并排序的思想。

Algorithm 6 交易数组中的逆序对 (第三部分)

```
function MERGE(nums, start, mid, end, temp_arr)    ▷ 合并两个已排序的数组部分
    i, j, k ← start, mid + 1, 0 ▷ 初始化三个指针 i、j 和 k，分别指向左子数组的起始位置、右子数组的起始位置和临时数组的当前位置
    count ← 0
    while i ≤ mid and j ≤ end do
        if nums[i] ≤ nums[j] then ▷ 如果左子数组的元素小于等于右子数组的元素，将左子数组的元素复制到临时数组
            temp_arr[k] ← nums[i]
            i ← i + 1
        else    ▷ 否则将右子数组的元素复制到临时数组，并更新逆序对计数器
            temp_arr[k] ← nums[j]
            count ← count + (mid − i + 1)
            j ← j + 1
        end if
        k ← k + 1
    end while
    while i ≤ mid do
        temp_arr[k] ← nums[i]
        i ← i + 1
        k ← k + 1
    end while
    while j ≤ end do
        temp_arr[k] ← nums[j]
        j ← j + 1
        k ← k + 1
    end while
    nums[start : end + 1] ← temp_arr[0 : k]
    return count
end function
```

时间复杂度：$O(nlogn)$

归并排序的时间复杂度是 $O(nlogn)$, 在合并过程中，每次发现一个逆序对，都会增加计数。在最坏的情况下，每次合并操作都可能发现 $O(n)$ 个逆序对。

1.1.4 合并 K 个升序链表

给定一个链表数组，每个链表都已经按升序排列。

请将所有链表合并到一个升序链表中，返回合并后的链表。

示例 1：

输入：lists = [[1,4,5],[1,3,4],[2,6]]

输出：[1,1,2,3,4,4,5,6]

解释：链表数组如下：[1->4->5, 1->3->4, 2->6] 将它们合并到一个有序链表中得到。[1->1->2->3->4->4->5->6]

示例 2：

输入：lists = []

输出：[]

示例 3：

输入：lists = [[]]

输出：[]

Algorithm 7 合并 K 个升序链表

输入: 一个链表数组 $lists$

输出: 一个升序链表

$min\_heap \leftarrow$ 创建一个空的最小堆
for i from 0 to $len(lists) - 1$ **do**
    if $lists[i]$ is not null **then**
        HEAPPUSH$(min_heap, (lists[i].val, i, lists[i]))$
    end if
end for
$dummy$ ▷ 创建一个虚拟头节点，其值为 -1
$current \leftarrow dummy$
while min_heap is not empty **do**
    $(val, idx, node) \leftarrow$ HEAPPOP(min_heap) ▷ 从最小堆中弹出最小元素，即当前所有链表头节点中的最小值
    $current.next \leftarrow node$
    $current \leftarrow current.next$ ▷ 更新节点指针 current
    if $node.next$ is not null **then** ▷ 如果弹出的节点有后继节点则将后继节点推入最小堆中
        HEAPPUSH$(min_heap, (node.next.val, idx, node.next))$
    end if
end while
return $dummy.next$
function HEAPPUSH$(heap, value)$
    将 $value$ 推入 $heap$ 中，保持 $heap$ 的最小堆性质
end function
function HEAPPOP$(heap)$
    从 $heap$ 中弹出最小元素
    return 弹出的元素
end function

时间复杂度：$O(nlogk)$

由于每次从最小堆中弹出和推入操作的时间复杂度是 $O(logk)$，并且每个链表的每个节点都会被处理一次，所以总体时间复杂度是 $O(nlogk)$

1.1.5 查找和最小的 K 对数字

给定两个以升序排列的整数数组 $nums1$ 和 $nums2$, 以及一个整数 k 。

定义一对值 (u, v)，其中第一个元素来自 $nums1$，第二个元素来自 $nums2$ 。

请找到和最小的 k 个数对 $(u1, v1), (u2, v2)...(uk, vk)$ 。

示例 1:

输入: $nums1$ = [1,7,11], $nums2$ = [2,4,6], k = 3

输出: [1,2],[1,4],[1,6]

解释: 返回序列中的前 3 对数：[1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]

示例 2:

输入: $nums1$ = [1,1,2], $nums2$ = [1,2,3], k = 2

输出: [1,1],[1,1]

解释: 返回序列中的前 2 对数：[1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]

示例 3:

输入: $nums1$= [1,2], $nums2$ = [3], k = 3

输出: [1,3],[2,3]

解释: 也可能序列中所有的数对都被返回:[1,3],[2,3]

Algorithm 8 查找和最小的 K 对数字

输入: 两个升序排列的整数数组 $nums1$,$nums2$, 一个整数 k

输出: k 个数对形式的数组

$min\_heap \leftarrow$ 创建一个空的最小堆
for i from 0 to $len(nums1) - 1$ **do**
  HeapPush$(min_heap, (nums1[i] + nums2[0], i, 0))$ ▷ 遍历数组 $nums1$ 的每个元素，将每个元素与数组 $nums2$ 的第一个元素构成的数对（及其索引）推入最小堆中
end for
$result \leftarrow$ 创建一个空列表用于存储结果
while $len(result) < k$ **do**
  if min_heap is empty **then**
    break
  end if
  $(sum\_val, i, j) \leftarrow$ HeapPop(min_heap) ▷ 从最小堆中弹出最小的数对，这个数对是当前所有潜在数对中的最小值
  Append $[nums1[i], nums2[j]]$ to $result$ ▷ 将弹出的数对添加到结果列表 result 中
  if $j + 1 < len(nums2)$ **then**
    HeapPush$(min_heap, (nums1[i] + nums2[j + 1], i, j + 1))$ ▷ 如果弹出的数对中的 $nums2$ 元素有后继元素将当前 $nums1$ 元素与 $nums2$ 的下一个元素构成的数对推入最小堆中
  end if
end while
return $result$
function HeapPush$(heap, value)$
  将 $value$ 推入 $heap$ 中，保持 $heap$ 的最小堆性质
end function
function HeapPop$(heap)$
  从 $heap$ 中弹出最小元素
  return 弹出的元素
end function

时间复杂度：$O(k * logk)$

遍历数组 $nums1$ 并将数对推入最小堆的时间复杂度是 $O(n * logk)$，其中 n 是数组 $nums1$ 的长度, 在每次循环中，从最小堆中弹出和推入操作的时间复杂度是 $O(logk)$, 因为最多执行 k 次这样的操作，所以这部分的总时间复杂度是 $O(k * logk)$。