**DSA – ASSIGNMENT 19**

💡 1. **Merge k Sorted Lists**

You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.

*Merge all the linked-lists into one sorted linked-list and return it.*

**Example 1:**

Input: lists = [[1,4,5],[1,3,4],[2,6]]

Output: [1,1,2,3,4,4,5,6]

Explanation: The linked-lists are:

[

1->4->5,

1->3->4,

2->6

]

merging them into one sorted list:

1->1->2->3->4->4->5->6

**Example 2:**

Input: lists = []

Output: []

**Example 3:**

Input: lists = [[]]

Output: []

**Constraints:**

* k == lists.length
* 0 <= k <= 10000
* 0 <= lists[i].length <= 500
* -10000 <= lists[i][j] <= 10000
* lists[i] is sorted in **ascending order**.
* The sum of lists[i].length will not exceed 10000.

**Solution. :-**

* Create an empty min-heap or priority queue.
* Iterate over each linked list in the given array lists and add the head node of each linked list to the min-heap along with the list index to keep track of the original list.
* Initialize an empty dummy node and a current pointer.
* While the min-heap is not empty:
  + Remove the minimum element from the min-heap, which gives us the smallest node among all the heads of the linked lists.
  + Append the removed node to the current pointer of the dummy node.
  + Move the current pointer to the next node.
  + If the removed node has a next node, add the next node to the min-heap.
* Return the next pointer of the dummy node, which points to the merged sorted list.

**import heapq**

**class ListNode:**

**def \_\_init\_\_(self, val=0, next=None):**

**self.val = val**

**self.next = next**

**def mergeKLists(lists):**

**# Custom comparator for the min-heap**

**# Compare nodes based on their values**

**def compare(node1, node2):**

**return node1.val - node2.val**

**# Create a min-heap**

**min\_heap = []**

**heapq.heapify(min\_heap)**

**# Add the heads of each linked list to the min-heap**

**for i in range(len(lists)):**

**if lists[i]:**

**heapq.heappush(min\_heap, (lists[i].val, i))**

**lists[i] = lists[i].next**

**# Initialize a dummy node and a current pointer**

**dummy = ListNode()**

**curr = dummy**

**# Merge the sorted lists using the min-heap**

**while min\_heap:**

**val, index = heapq.heappop(min\_heap)**

**curr.next = ListNode(val)**

**curr = curr.next**

**if lists[index]:**

**heapq.heappush(min\_heap, (lists[index].val, index))**

**lists[index] = lists[index].next**

**return dummy.next**

**# Create linked lists**

**list1 = ListNode(1, ListNode(4, ListNode(5)))**

**list2 = ListNode(1, ListNode(3, ListNode(4)))**

**list3 = ListNode(2, ListNode(6))**

**lists = [list1, list2, list3]**

**# Merge the sorted lists**

**result = mergeKLists(lists)**

**# Print the merged sorted list**

**while result:**

**print(result.val, end=" ")**

**result = result.next**

💡 2. **Count of Smaller Numbers After Self**

Given an integer array nums, return *an integer array* counts *where* counts[i] *is the number of smaller elements to the right of* nums[i].

**Example 1:**

Input: nums = [5,2,6,1]

Output: [2,1,1,0]

Explanation:

To the right of 5 there are2 smaller elements (2 and 1).

To the right of 2 there is only1 smaller element (1).

To the right of 6 there is1 smaller element (1).

To the right of 1 there is0 smaller element.

**Example 2:**

Input: nums = [-1]

Output: [0]

**Example 3:**

Input: nums = [-1,-1]

Output: [0,0]

**Constraints:**

* 1 <= nums.length <= 100000
* -10000 <= nums[i] <= 10000

**Solution. :-**

* Create a helper function mergeSort() that takes an input array and returns the sorted array along with the count of smaller elements to the right of each element.
* The base case of the recursive merge sort is when the input array contains only one element. In this case, return the array itself and set the count to zero.
* Split the input array into two halves, and recursively call mergeSort() on each half.
* Merge the sorted halves using the two-pointer technique and keep track of the count of smaller elements.
  + Initialize two pointers, i and j, at the beginning of the left and right halves.
  + Compare the elements at i and j. If left[i] is smaller than or equal to right[j], move i to the next element in the left half.
  + If left[i] is greater than right[j], it means that there are mid - i elements in the left half that are greater than right[j]. Increment the count by mid - i and move j to the next element in the right half.
  + Repeat this process until either the left or right half is fully merged.
* After merging the two halves, return the merged array and the total count of smaller elements.

**def countSmaller(nums):**

**def mergeSort(nums):**

**if len(nums) <= 1:**

**return nums, [0]**

**mid = len(nums) // 2**

**left, left\_count = mergeSort(nums[:mid])**

**right, right\_count = mergeSort(nums[mid:])**

**merged = []**

**count = [0] \* len(nums)**

**i, j = 0, 0**

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

**if left[i] <= right[j]:**

**merged.append(left[i])**

**count[left\_count[i]] += j**

**i += 1**

**else:**

**merged.append(right[j])**

**count[right\_count[j]] += len(left) - i**

**j += 1**

**merged.extend(left[i:])**

**merged.extend(right[j:])**

**count.extend(left\_count[i:])**

**count.extend(right\_count[j:])**

**return merged, count**

**\_, result = mergeSort(list(enumerate(nums)))**

**return result**

**nums = [5, 2, 6, 1]**

**output = countSmaller(nums)**

**print(output)**

💡 3. **Sort an Array**

Given an array of integers nums, sort the array in ascending order and return it.

You must solve the problem **without using any built-in** functions in O(nlog(n)) time complexity and with the smallest space complexity possible.

**Example 1:**

Input: nums = [5,2,3,1]

Output: [1,2,3,5]

Explanation: After sorting the array, the positions of some numbers are not changed (for example, 2 and 3), while the positions of other numbers are changed (for example, 1 and 5).

**Example 2:**

Input: nums = [5,1,1,2,0,0]

Output: [0,0,1,1,2,5]

Explanation: Note that the values of nums are not necessairly unique.

**Constraints:**

* 1 <= nums.length <= 5 \* 10000
* -5 \* 104 <= nums[i] <= 5 \* 10000

**Solution. :-**

* Create a helper function partition() that takes an input array, selects a pivot element, and partitions the array into two parts: elements smaller than the pivot and elements greater than the pivot.
  + Choose a pivot element. It can be the last element of the array.
  + Initialize a pointer i at the beginning of the array.
  + Iterate from the beginning to the second-to-last element of the array:
    - If the current element is smaller than the pivot, swap it with the element at index i and move i to the next element.
  + Finally, swap the pivot element with the element at index i and return i.
* Implement the recursive quickSort() function that takes an input array and sorts it using the QuickSort algorithm.
  + If the array has fewer than two elements, return the array.
  + Otherwise, call the partition() function to partition the array into two parts.
  + Recursively call quickSort() on the left part (elements smaller than the pivot) and the right part (elements greater than the pivot).
  + Finally, concatenate the sorted left part, the pivot element, and the sorted right part to get the sorted array.
* Call the quickSort() function on the input array and return the sorted array.

**def sortArray(nums):**

**def partition(nums, low, high):**

**pivot = nums[high]**

**i = low**

**for j in range(low, high):**

**if nums[j] < pivot:**

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

**i += 1**

**nums[i], nums[high] = nums[high], nums[i]**

**return i**

**def quickSort(nums, low, high):**

**if low < high:**

**pivot\_index = partition(nums, low, high)**

**quickSort(nums, low, pivot\_index - 1)**

**quickSort(nums, pivot\_index + 1, high)**

**quickSort(nums, 0, len(nums) - 1)**

**return nums**

**nums = [5, 2, 3, 1]**

**output = sortArray(nums)**

**print(output)**

💡 4. **Move all zeroes to end of array**

Given an array of random numbers, Push all the zero’s of a given array to the end of the array. For example, if the given arrays is {1, 9, 8, 4, 0, 0, 2, 7, 0, 6, 0}, it should be changed to {1, 9, 8, 4, 2, 7, 6, 0, 0, 0, 0}. The order of all other elements should be same. Expected time complexity is O(n) and extra space is O(1).

**Example:**

Input : arr[] = {1, 2, 0, 4, 3, 0, 5, 0};

Output : arr[] = {1, 2, 4, 3, 5, 0, 0, 0};

Input : arr[] = {1, 2, 0, 0, 0, 3, 6};

Output : arr[] = {1, 2, 3, 6, 0, 0, 0};

**Solution. :-**

* Initialize two pointers, i and j, both starting from the beginning of the array.
* Iterate through the array using the pointer i:
  + If the element at index i is non-zero, swap it with the element at index j and increment j by 1.
* After the iteration, all non-zero elements will be placed towards the beginning of the array, and the value of j will be the count of non-zero elements.
* Iterate from j to the end of the array and set all the elements to zero.
* The array will now have all the zeros placed at the end while maintaining the order of other elements.

**def moveZeroes(nums):**

**n = len(nums)**

**j = 0 # pointer to track the position of non-zero elements**

**for i in range(n):**

**if nums[i] != 0:**

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

**j += 1**

**while j < n:**

**nums[j] = 0**

**j += 1**

**return nums**

**nums = [1, 2, 0, 4, 3, 0, 5, 0]**

**output = moveZeroes(nums)**

**print(output)**

💡 5. **Rearrange array in alternating positive & negative items with O(1) extra space**

Given an **array of positive** and **negative numbers**, arrange them in an **alternate** fashion such that every positive number is followed by a negative and vice-versa maintaining the **order of appearance**. The number of positive and negative numbers need not be equal. If there are more positive numbers they appear at the end of the array. If there are more negative numbers, they too appear at the end of the array.

**Examples:**

Input:  arr[] = {1, 2, 3, -4, -1, 4} Output: arr[] = {-4, 1, -1, 2, 3, 4}

Input:  arr[] = {-5, -2, 5, 2, 4, 7, 1, 8, 0, -8} Output: arr[] = {-5, 5, -2, 2, -8, 4, 7, 1, 8, 0}

**Solution. :-**

* Initialize two pointers, i and j, both starting from the beginning of the array.
* Iterate through the array using the pointer i:
  + If the element at index i is positive, find the next negative element starting from index j using another loop.
  + Once a negative element is found at index k, shift all elements from j to i-1 one position to the right.
  + Place the negative element at index j and increment both i and j by 1.
* After the iteration, all positive elements will be placed at the beginning of the array, and negative elements will be placed alternately after them.

**def rearrangeArray(nums):**

**n = len(nums)**

**i = 0 # pointer to track positive elements**

**j = 0 # pointer to track negative elements**

**while i < n and j < n:**

**# Find next positive element**

**while i < n and nums[i] < 0:**

**i += 1**

**# Find next negative element**

**while j < n and nums[j] >= 0:**

**j += 1**

**if i < n and j < n:**

**# Shift elements from j to i-1 one position to the right**

**for k in range(i, j, -1):**

**nums[k] = nums[k-1]**

**# Place the negative element at index j**

**nums[j] = nums[i]**

**i += 1**

**j += 1**

**return nums**

**nums = [1, 2, 3, -4, -1, 4]**

**output = rearrangeArray(nums)**

**print(output)**

💡 **6. Merge two sorted arrays**

Given two sorted arrays, the task is to merge them in a sorted manner.

**Examples:**

Input: arr1[] = { 1, 3, 4, 5}, arr2[] = {2, 4, 6, 8}  Output: arr3[] = {1, 2, 3, 4, 4, 5, 6, 8}

Input: arr1[] = { 5, 8, 9}, arr2[] = {4, 7, 8} Output: arr3[] = {4, 5, 7, 8, 8, 9}

**Solution. :-**

* Create an empty array arr3 to store the merged result.
* Initialize three pointers, i for arr1, j for arr2, and k for arr3, all starting from the beginning of their respective arrays.
* Compare the elements at arr1[i] and arr2[j], and place the smaller element in arr3[k].
  + If arr1[i] is smaller, assign it to arr3[k] and increment both i and k.
  + If arr2[j] is smaller, assign it to arr3[k] and increment both j and k.
* Repeat step 3 until either arr1 or arr2 is fully processed (i.e., i reaches the end of arr1 or j reaches the end of arr2).
* If there are any remaining elements in arr1 or arr2 after the previous step, append them to the end of arr3.
* arr3 will now contain the merged sorted array.

**def mergeArrays(arr1, arr2):**

**n1 = len(arr1)**

**n2 = len(arr2)**

**arr3 = [0] \* (n1 + n2)**

**i = 0 # pointer for arr1**

**j = 0 # pointer for arr2**

**k = 0 # pointer for arr3**

**while i < n1 and j < n2:**

**if arr1[i] <= arr2[j]:**

**arr3[k] = arr1[i]**

**i += 1**

**else:**

**arr3[k] = arr2[j]**

**j += 1**

**k += 1**

**# Append remaining elements of arr1, if any**

**while i < n1:**

**arr3[k] = arr1[i]**

**i += 1**

**k += 1**

**# Append remaining elements of arr2, if any**

**while j < n2:**

**arr3[k] = arr2[j]**

**j += 1**

**k += 1**

**return arr3**

**arr1 = [1, 3, 4, 5]**

**arr2 = [2, 4, 6, 8]**

**output = mergeArrays(arr1, arr2)**

**print(output)**

💡 7. **Intersection of Two Arrays**

Given two integer arrays nums1 and nums2, return *an array of their intersection*. Each element in the result must be **unique** and you may return the result in **any order**.

**Example 1:**

Input: nums1 = [1,2,2,1], nums2 = [2,2]

Output: [2]

**Example 2:**

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

Output: [9,4]

Explanation: [4,9] is also accepted.

**Constraints:**

* 1 <= nums1.length, nums2.length <= 1000
* 0 <= nums1[i], nums2[i] <= 1000

**Solution. :-**

* Convert both arrays, nums1 and nums2, into sets to remove any duplicate elements.
* Create an empty result set to store the intersection.
* Iterate over each element num in the first set, nums1.
  + If num is present in the second set, nums2, add it to the result set.
* Convert the result set into a list and return it as the intersection.

**def intersection(nums1, nums2):**

**set1 = set(nums1)**

**set2 = set(nums2)**

**result = set()**

**for num in set1:**

**if num in set2:**

**result.add(num)**

**return list(result)**

**nums1 = [1, 2, 2, 1]**

**nums2 = [2, 2]**

**output = intersection(nums1, nums2)**

**print(output)**

💡 8. **Intersection of Two Arrays II**

Given two integer arrays nums1 and nums2, return *an array of their intersection*. Each element in the result must appear as many times as it shows in both arrays and you may return the result in **any order**.

**Example 1:**

Input: nums1 = [1,2,2,1], nums2 = [2,2]

Output: [2,2]

**Example 2:**

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

Output: [4,9]

Explanation: [9,4] is also accepted.

**Constraints:**

* 1 <= nums1.length, nums2.length <= 1000
* 0 <= nums1[i], nums2[i] <= 1000

**Solution. :-**

* Create two dictionaries, dict1 and dict2, to store the count of each element in nums1 and nums2, respectively.
* Iterate over each element num in nums1 and update its count in dict1.
* Iterate over each element num in nums2 and update its count in dict2.
* Create an empty result list to store the intersection.
* Iterate over each key num in dict1.
  + If num is present in dict2, find the minimum count between dict1[num] and dict2[num].
  + Append num to the result list min\_count times.
* Return the result list.

**def intersect(nums1, nums2):**

**dict1 = {}**

**dict2 = {}**

**for num in nums1:**

**dict1[num] = dict1.get(num, 0) + 1**

**for num in nums2:**

**dict2[num] = dict2.get(num, 0) + 1**

**result = []**

**for num in dict1:**

**if num in dict2:**

**min\_count = min(dict1[num], dict2[num])**

**result.extend([num] \* min\_count)**

**return result**

**nums1 = [1, 2, 2, 1]**

**nums2 = [2, 2]**

**output = intersect(nums1, nums2)**

**print(output)**