**Name:**M.VINAY

**Reg no:**192311146

**Course code:**CSA0678

**Course name:**DAA

**1.You are given a string s, and an array of pairs of indices in the string pairs where pairs[i] = [a, b] indicates 2 indices(0-indexed) of the string.You can swap the characters at any pair of I ndices in the given pairs any number of times. Return the lexicographically smallest string that s can be changed to after using the swaps.**

**Program:**

from collections import defaultdict

def smallestStringWithSwaps(s, pairs):

# Helper function for DFS

def dfs(node, visited, graph, component):

stack = [node]

while stack:

v = stack.pop()

if v not in visited:

visited.add(v)

component.append(v)

for neighbor in graph[v]:

if neighbor not in visited:

stack.append(neighbor)

# Build the graph

graph = defaultdict(list)

for a, b in pairs:

graph[a].append(b)

graph[b].append(a)

visited = set()

components = []

# Find all connected components using DFS

for i in range(len(s)):

if i not in visited:

component = []

dfs(i, visited, graph, component)

components.append(component)

# For each component, sort the indices and the corresponding characters

s = list(s)

for component in components:

indices = sorted(component)

chars = sorted(s[i] for i in indices)

for index, char in zip(indices, chars):

s[index] = char

return ''.join(s)

# Example usage:

s = "dcab"

pairs = [[0, 3], [1, 2]]

print(smallestStringWithSwaps(s, pairs))

**Output:** "bacd"

**2.Given two strings: s1 and s2 with the same size, check if some permutation of string s1 can break some permutation of string s2 or vice-versa. In other words s2 can break s1 or vice-versa. A string x can break string y (both of size n) if x[i] >= y[i] (in alphabetical order) for all i between 0 and n**

**Program:**

def can\_break(s1, s2):

# Sort both strings

sorted\_s1 = sorted(s1)

sorted\_s2 = sorted(s2)

# Check if sorted\_s1 can break sorted\_s2

can\_s1\_break\_s2 = all(c1 >= c2 for c1, c2 in zip(sorted\_s1, sorted\_s2))

# Check if sorted\_s2 can break sorted\_s1

can\_s2\_break\_s1 = all(c2 >= c1 for c2, c1 in zip(sorted\_s2, sorted\_s1))

# Return True if either condition is met

return can\_s1\_break\_s2 or can\_s2\_break\_s1

# Example usage:

s1 = "abc"

s2 = "xya"

print(can\_break(s1, s2))

**Output:** True

**3.** **You are given a string s. s[i] is either a lowercase English letter or '?'. For a string t having length m containing only lowercase English letters, we define the function cost(i) for an index i as the number of characters equal to t[i] that appeared before it, i.e. in the range [0, i - 1]. The value of t is the sum of cost(i) for all indices i. For example, for the string t = "aab":**

**cost(0) = 0**

**cost(1) = 1**

**cost(2) = 0**

**Hence, the value of "aab" is 0 + 1 + 0 = 1. Your task is to replace all occurrences of '?' in s with any lowercase English letter so at the value of s is minimized.**

**Program:**

from collections import defaultdict

def minimize\_string\_value(s):

# Initialize the frequency dictionary

frequency = defaultdict(int)

# Convert the string to a list for easier manipulation

s = list(s)

# Iterate through the string

for i in range(len(s)):

if s[i] == '?':

# Find the character with the minimum frequency

min\_freq\_char = min(frequency, key=frequency.get, default='a')

min\_freq = frequency[min\_freq\_char]

# Find the least used character from 'a' to 'z'

for char in 'abcdefghijklmnopqrstuvwxyz':

if frequency[char] < min\_freq:

min\_freq\_char = char

min\_freq = frequency[char]

# Replace '?' with this character

s[i] = min\_freq\_char

# Update the frequency counter

frequency[s[i]] += 1

return ''.join(s)

# Example usage:

s = "ab??ac?"

print(minimize\_string\_value(s))

**Output** : "abcbaca" or other with minimal value

**4.**

**You are given a string s. Consider performing the following operation until s becomes empty: For every alphabet character from 'a' to 'z', remove the first occurrence of that character in s (if it exists). For example, let initially s = "aabcbbca". We do the following operations: Remove the underlined characters s = "aabcbbca". The resulting string is s = "abbca". Remove the underlined characters s = "abbca". The resulting string is s = "ba". Remove the underlined characters s = "ba". The resulting string is s = "". Return the value of the string s right before applying the last operation. In the example above, answer is "ba".**

**PROGRAM:**

def last\_before\_empty(s):

while s:

previous\_s = s # Keep track of the string before this round of deletions

for char in 'abcdefghijklmnopqrstuvwxyz':

index = s.find(char)

if index != -1:

s = s[:index] + s[index+1:]

if not s: # If the string is empty after this round, return the previous state

return previous\_s

s = "aabcbbca"

print(last\_before\_empty(s))

Output: "ba"

**5.** **Given an integer array nums, find the subarray with the largest sum, and return its sum.**

**PROGRAM:**

def max\_subarray\_sum(nums):

# Initialize current\_sum to 0 and max\_sum to a very small number

current\_sum = 0

max\_sum = float('-inf')

for num in nums:

# Calculate current\_sum by either adding the current number or starting fresh from current number

current\_sum = max(num, current\_sum + num)

# Update max\_sum if the current\_sum is greater

max\_sum = max(max\_sum, current\_sum)

return max\_sum

nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]

print(max\_subarray\_sum(nums))

**Output:** 6

**6.** **You are given an integer array nums with no duplicates. A maximum binary tree can be built recursively from nums using the following algorithm: Create a root node whose value is the maximum value in nums. Recursively build the left subtree on the subarray prefix to the left of the maximum value. Recursively build the right subtree on the subarray suffix to the right of the maximum value. Return the maximum binary tree built from nums.**

**PROGRAM:**

# Definition for a binary tree node.

class TreeNode:

def \_\_init\_\_(self, val=0, left=None, right=None):

self.val = val

self.left = left

self.right = right

def construct\_maximum\_binary\_tree(nums):

if not nums:

return None

# Find the index of the maximum element in the current array (or subarray)

max\_index = nums.index(max(nums))

# Create the root node with the maximum element

root = TreeNode(nums[max\_index])

# Recursively build the left and right subtrees

root.left = construct\_maximum\_binary\_tree(nums[:max\_index])

root.right = construct\_maximum\_binary\_tree(nums[max\_index + 1:])

return root

# Example usage:

nums = [3, 2, 1, 6, 0, 5]

root = construct\_maximum\_binary\_tree(nums)

# Function to print the binary tree in order

def print\_tree(node):

if not node:

return

print\_tree(node.left)

print(node.val)

print\_tree(node.right)

print\_tree(root)

**7.** **Given a circular integer array nums of length n, return the maximum possible sum of a non-empty subarray of nums.A circular array means the end of the array connects to the beginning of the array. Formally, the next element of nums[i] is nums[(i + 1) % n] and the previous element of nums[i] is nums[(i - 1 + n) % n].A subarray may only include each element of the fixed buffer nums at most once. Formally, for a subarray nums[i], nums[i + 1], ..., nums[j], there does not exist i <= k1, k2 <= j with k1 % n == k2 % n.**

**PROGRAM:**

def max\_subarray\_sum\_circular(nums):

def kadane\_max(nums):

max\_ending\_here = max\_so\_far = nums[0]

for x in nums[1:]:

max\_ending\_here = max(x, max\_ending\_here + x)

max\_so\_far = max(max\_so\_far, max\_ending\_here)

return max\_so\_far

def kadane\_min(nums):

min\_ending\_here = min\_so\_far = nums[0]

for x in nums[1:]:

min\_ending\_here = min(x, min\_ending\_here + x)

min\_so\_far = min(min\_so\_far, min\_ending\_here)

return min\_so\_far

total\_sum = sum(nums)

max\_sum = kadane\_max(nums)

min\_sum = kadane\_min(nums)

# If all numbers are negative, min\_sum will be equal to total\_sum

if max\_sum < 0:

return max\_sum

return max(max\_sum, total\_sum - min\_sum)

# Example usage:

nums = [5, -3, 5]

print(max\_subarray\_sum\_circular(nums))

**Output**: 10

**8**.

**You are given an array nums consisting of integers. You are also given a 2D array queries, where queries[i] = [posi, xi].For query i, we first set nums[posi] equal to xi, then we calculate the answer to query i which is the maximum sum of a subsequence of nums where no two adjacent elements are selected. Return the sum of the answers to all queries. Since the final answer may be very large, return it modulo 109 + 7. A subsequence is an array that can be derived from another array by deleting some or no elements without changing the order of the remaining elements.**

**PROGRAM:**

def max\_sum\_subsequence\_no\_adjacent(nums):

if not nums:

return 0

n = len(nums)

if n == 1:

return nums[0]

include = nums[0]

exclude = 0

for i in range(1, n):

new\_exclude = max(include, exclude)

include = exclude + nums[i]

exclude = new\_exclude

return max(include, exclude)

def sum\_of\_answers(nums, queries):

MOD = 10\*\*9 + 7

total\_sum = 0

# Initial computation of the maximum sum subsequence

max\_sum = max\_sum\_subsequence\_no\_adjacent(nums)

for pos, x in queries:

# Update the element at pos

nums[pos] = x

# Recompute the maximum sum subsequence after the update

max\_sum = max\_sum\_subsequence\_no\_adjacent(nums)

# Add the result to the total sum

total\_sum = (total\_sum + max\_sum) % MOD

return total\_sum

nums = [2, 1, 3, 4]

queries = [[1, 5], [2, 2]]

print(sum\_of\_answers(nums, queries))

**Output:** the sum of answers to all queries

**9.Given an array of points where points[i] = [xi, yi] represents a point on the X-Y plane and an integer k, return the k closest points to the origin (0, 0).The distance between two points on the X-Y plane is the Euclidean distance (i.e., √(x1 - x2)2 + (y1 - y2)2). You may return the answer in any order. The answer is guaranteed to be unique (except for the order that it is in).**

**Program:**

import math

def kClosest(points, k):

distances = [(math.sqrt(point[0] \*\* 2 + point[1] \*\* 2), point) for point in points]

distances.sort(key=lambda x: x[0])

closest\_points = [distance[1] for distance in distances[:k]]

return closest\_points

points = [[1,3],[-2,2],[5,-1]]

k = 2

print(kClosest(points, k))

**Output:** [[-2, 2], [1, 3]]

**10.** **Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).**

**Program:**

def findMedianSortedArrays(nums1, nums2):

total\_length = len(nums1) + len(nums2)

if total\_length % 2 == 0:

return (findKthElement(nums1, nums2, total\_length // 2) + findKthElement(nums1, nums2, total\_length // 2 - 1)) / 2

else:

return findKthElement(nums1, nums2, total\_length // 2)

def findKthElement(nums1, nums2, k):

if not nums1:

return nums2[k]

if not nums2:

return nums1[k]

if k == 0:

return min(nums1[0], nums2[0])

length1 = len(nums1)

length2 = len(nums2)

index1 = min(length1 - 1, k // 2)

index2 = min(length2 - 1, k // 2)

if nums1[index1] > nums2[index2]:

return findKthElement(nums1, nums2[index2 + 1:], k - index2 - 1)

else:

return findKthElement(nums1[index1 + 1:], nums2, k - index1 - 1)

nums1 = [1, 3]

nums2 = [2]

print(findMedianSortedArrays(nums1, nums2))

**Output:** 2.0