DATE:7/6/24

LAB-3

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 indices in the given pairs any number of times. Return the lexicographically smallest string that s can be changed to after using the swap

CODE:

```
class Graph:
def __init__(self, V):
# Number of vertices in the graph
self.V = V
# Adjacency list to represent the graph
self.adj = [[] for i in range(V)]

def addPair(self, v, w):
# Adjust indices to be 0-based
v -= 1
w -= 1
# Add a relation (edge) between v and w
self.adj[v].append(w)
# Also add the reverse relation (undirected graph)
self.adj[w].append(v)

def disjointSets(self):
# Create a list to track visited vertices
visited = [False] * self.V
# List to store groups of connected vertices
all_sets = []

def countAll(v, group):
# Mark the current vertex as visited
visited[v] = True
```

```
# Add the vertex to the current group (adjust index)
group.append(v + 1)

# Recursively explore neighbors of the current vertex
for neighbor in self.adj[v]:
if not visited[neighbor]:
countAll(neighbor, group)

for i in range(self.V):
# If the vertex hasn't been visited,
# it's the start of a new group
if not visited[i]:
# Create a new group
group = []
# Find all vertices connected to this one
countAll(i, group)
# Add the group to the list of all groups
all_sets.append(group)

return all_sets

# Driver Code

# Input string
Str = 'zcxfbe'
# Number of elements
N = 6
# Pairs of indices indicating relationships
Pairs = [[0, 1], [0, 2], [3, 5]]

# Create a graph with N vertices
g = Graph(N)
```

```
# Add relations based on pairs
for i in Pairs:
g.addPair(i[0], i[1])

# Find groups formed by the relations
groups = g.disjointSets()

key = [] # List to store indices of characters
value = [] # List to store characters

# Sort characters within each group and store the results
for i in range(len(groups)):
# Temporary list to store characters within a group
semians = []
for j in groups[i]:
# Extract characters from the input string
semians.append(Str[j])
# Sort the characters lexicographically
semians.sort()
# Sort the indices
groups[i].sort()

# Add sorted characters to the value list
value.extend(semians)
# Add sorted indices to the key list
key.extend(groups[i])

# Initialize a list to reconstruct the final string
ans = [""] * N

# Reconstruct the final string
```

based on sorted characters and indices

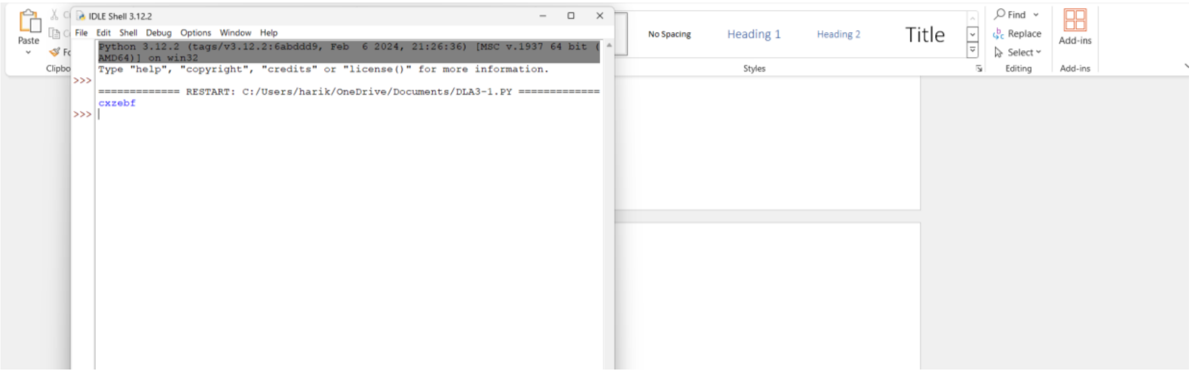
for i in range(N):

ans[key[i]] = value[i]

Print the rearranged string

print(''.join(ans))

OUTPUT:



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-1.

Code:

def canBreak(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

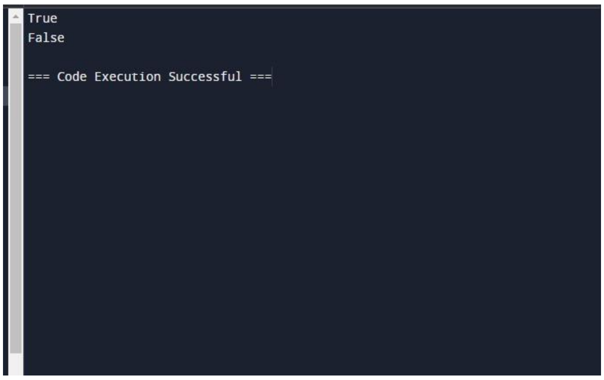
  can_s2_break_s1 = all(c2 >= c1 for c1, c2 in zip(sorted_s1, sorted_s2))

  # Return True if either can break the other

  return can_s1_break_s2 or can_s2_break_s1

```
# Example usage
s1 = "abc"
s2 = "xya"
print(canBreak(s1, s2))  # Output: True, since "abc" can be permuted to "cba" which can break "axy"

s1 = "abe"
s2 = "acd"
print(canBreak(s1, s2))  # Output: False, neither permutation can break the other
```

OUTPUT:



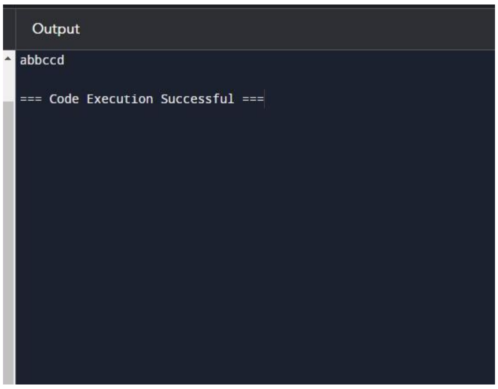
3.You are given a string s. s[i] is either a lowercase English letter or '?'. For a string t having length m containing only lowercase English letters, we define the function cost(i) for an index i as the number of characters equal to t[i] that appeared before it, i.e. in the range [0, i - 1]. The value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

Code:

```
def minimize_cost_string(s):
    # Dictionary to keep track of the frequency of each character
    frequency = {chr(i): 0 for i in range(ord('a'), ord('z') + 1)}

    result = []

    for char in s:
        if char == '?':
            # Find the character with the minimum frequency
```

```
        min_char = min(frequency, key=frequency.get)
        result.append(min_char)
        # Update the frequency of the chosen character
        frequency[min_char] += 1
      else:
        result.append(char)
        # Update the frequency of the current character
        frequency[char] += 1

  return ''.join(result)

# Example usage
s = "a?b?c?"
print(minimize_cost_string(s))  # Example output: "aabbbc"
```

OUTPUT:



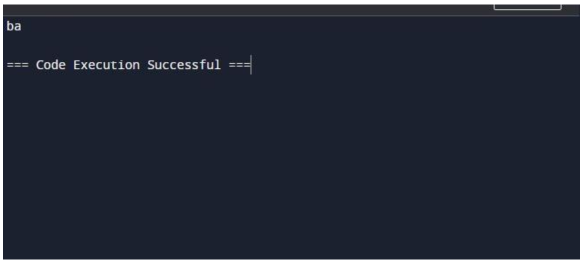
4.You are given a string s. Consider performing the following operation until s becomes empty: For every alphabet character from 'a' to 'z', remove the first occurrence of that character in s (if it exists).

CODE:

```
def last_state_before_empty(s):
  from string import ascii_lowercase

  while True:
    prev_s = s
    for char in ascii_lowercase:
```

```
            s = s.replace(char, '', 1)
        if s == "":
            return prev_s

# Example usage
s = "aabcbbca"
print(last_state_before_empty(s))
```

OUTPUT:



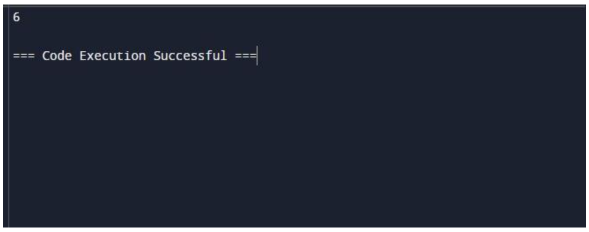
5.Given an integer array nums, find the subarray with the largest sum, and return its sum.

Code:

```
def max_subarray_sum(nums):
    # Initialize the maximum sum to be the first element
    max_current = max_global = nums[0]

    for num in nums[1:]:
        # Calculate the maximum sum of subarray ending at the current position
        max_current = max(num, max_current + num)

        # Update the global maximum sum if the current maximum sum is greater
        if max_current > max_global:
            max_global = max_current

    return max_global

# Example usage
nums = [-2,1,-3,4,-1,2,1,-5,4]
```

print(max_subarray_sum(nums))

OUTPUT:



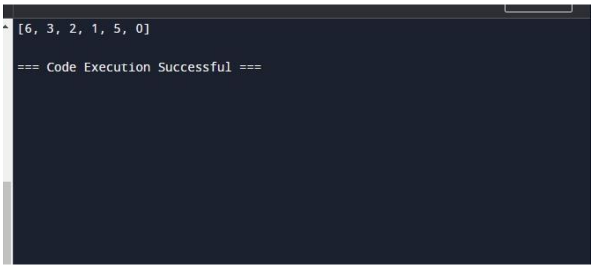
6. You are given an integer array nums with no duplicates. A maximum binary tree can be built recursively from nums using the following algorithm: Create a root node whose value is the maximum value in nums. Recursively build the left subtree on the subarray prefix to the left of the maximum value. Recursively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

Code:

```
class TreeNode:
    def _init_(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def constructMaximumBinaryTree(nums):
    if not nums:
        return None

    # Find the index of the maximum value in the current array
    max_index = nums.index(max(nums))

    # Create the root node with the maximum value
    root = TreeNode(nums[max_index])

    # Recursively build the left and right subtrees
    root.left = constructMaximumBinaryTree(nums[:max_index])
    root.right = constructMaximumBinaryTree(nums[max_index + 1:])
```

```
    return root

# Helper function to print the tree in a readable format (preorder traversal)
def preorderTraversal(root):
    if not root:
        return []
    return [root.val] + preorderTraversal(root.left) + preorderTraversal(root.right)

# Example usage
nums = [3, 2, 1, 6, 0, 5]
tree = constructMaximumBinaryTree(nums)
print(preorderTraversal(tree))
```

OUTPUT:



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.

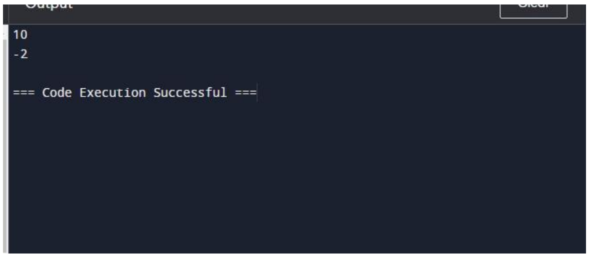
CODE:

```
def maxSubarraySumCircular(nums):
```

```
    # Helper function to perform Kadane's algorithm
    def kadane(array):
        max_current = max_global = array[0]
        for num in array[1:]:
            max_current = max(num, max_current + num)
            if max_current > max_global:
                max_global = max_current
        return max_global

    # Case 1: Max subarray sum for non-circular case
    max_kadane = kadane(nums)

    # Case 2: Max subarray sum for circular case
    total_sum = sum(nums)

    # Invert the nums array to find the minimum subarray sum
    nums_inverted = [-num for num in nums]

    # Find the maximum subarray sum for the inverted array
    # This is equivalent to finding the minimum subarray sum for the original array
    max_inverted_kadane = kadane(nums_inverted)

    max_circular = total_sum + max_inverted_kadane  # max_circular considers the wrap-
around part

    # Handle the edge case where all elements are negative
    if max_circular == 0:
        return max_kadane

    return max(max_kadane, max_circular)

# Example usage
nums = [5, -3, 5]
```

```
print(maxSubarraySumCircular(nums))


nums = [-3, -2, -3]
print(maxSubarraySumCircular(nums))
```

output:



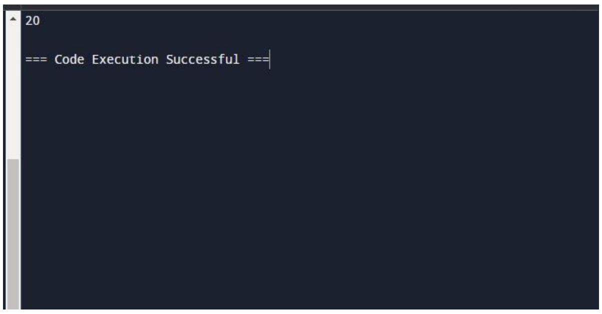
8.You are given an array nums consisting of integers. You are also given a 2D array queries, where queries[i] = [posi, xi].For query i, we first set nums[posi] equal to xi, then we calculate the answer to query i which is the maximum sum of a subsequence of nums where no two adjacent elements are selected. 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.

Code:

```
def max_non_adjacent_sum(nums):
  include = 0
  exclude = 0

  for num in nums:
    new_include = exclude + num
    new_exclude = max(include, exclude)
    include = new_include
    exclude = new_exclude

  return max(include, exclude)

def solve(nums, queries):
  MOD = 10**9 + 7
  result = 0
```

```
    for pos, val in queries:
        # Update the array
        nums[pos] = val

        # Calculate the maximum sum of the non-adjacent subsequence
        max_sum = max_non_adjacent_sum(nums)

        # Add to the result
        result = (result + max_sum) % MOD

    return result

# Example usage
nums = [1, 2, 3, 4]
queries = [[0, 2], [3, 5], [1, 1]]
print(solve(nums, queries))
```

OUTPUT:

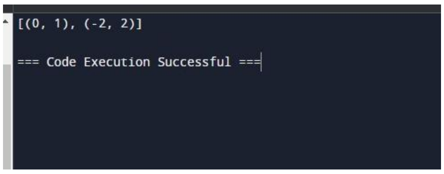


9. Given an array of points where points[i] = [xi, yi] represents a point on the X-Y plane and an integer k, return the k closest points to the origin (0, 0).The distance between two points on the X-Y plane is the Euclidean distance (i.e., √(x1 - x2)2 + (y1 - y2)2). You may return the answer in any order.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

CODE:

```
import heapq


def kClosest(points, k):
    # Create a min-heap
    heap = []

    for (x, y) in points:
        # Calculate the squared distance from the origin
        dist = x * x + y * y
        # Push the distance and the point onto the heap
        heapq.heappush(heap, (dist, (x, y)))

    # Extract the k closest points
    closest_points = [heapq.heappop(heap)[1] for _ in range(k)]

    return closest_points

# Example usage
points = [[1, 3], [-2, 2], [5, 8], [0, 1]]
k = 2
print(kClosest(points, k))
```

OUTPUT:



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)).

CODE:

```
def findMedianSortedArrays(nums1, nums2):
    # Ensure nums1 is the smaller array
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1

    m, n = len(nums1), len(nums2)
    imin, imax, half_len = 0, m, (m + n + 1) // 2

    while imin <= imax:
        i = (imin + imax) // 2
        j = half_len - i

        if i < m and nums1[i] < nums2[j - 1]:
            imin = i + 1
        elif i > 0 and nums1[i - 1] > nums2[j]:
            imax = i - 1
        else:
            # i is perfect
            if i == 0: max_of_left = nums2[j - 1]
            elif j == 0: max_of_left = nums1[i - 1]
            else: max_of_left = max(nums1[i - 1], nums2[j - 1])

            if (m + n) % 2 == 1:
                return max_of_left

            if i == m: min_of_right = nums2[j]
            elif j == n: min_of_right = nums1[i]
            else: min_of_right = min(nums1[i], nums2[j])

            return (max_of_left + min_of_right)
```

```
# Example usage
nums1 = [1, 3]
nums2 = [2]
print(findMedianSortedArrays(nums1, nums2))

nums1 = [1, 2]
nums2 = [3, 4]
print(findMedianSortedArrays(nums1, nums2))
```

OUTPUT:

```
2
2.5

=== Code Execution Successful ===
```