**DAY: 3 LAB PROGRAMS**

**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 swaps.**

class Solution:

def union(self, a, b):

self.parent[self.find(a)] = self.find(b)

def find(self, a):

if self.parent[a] != a:

self.parent[a] = self.find(self.parent[a])

return self.parent[a]

def smallestStringWithSwaps(self, s: str, pairs: List[List[int]]) -> str:

self.parent = list(range(len(s)))

for a, b in pairs:

self.union(a, b)

group = defaultdict(lambda: ([], []))

for i, ch in enumerate(s):

parent = self.find(i)

group[parent][0].append(i)

group[parent][1].append(ch)

res = [''] \* len(s)

for ids, chars in group.values():

ids.sort()

chars.sort()

for ch, i in zip(chars, ids):

res[i] = ch

return ''.join(res)

**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.**

def can\_break(s1, s2):

count\_s1 = [0] \* 26

count\_s2 = [0] \* 26

for ch in s1:

count\_s1[ord(ch) - ord('a')] += 1

for ch in s2:

count\_s2[ord(ch) - ord('a')] += 1

cumulative\_s1 = 0

cumulative\_s2 = 0

for i in range(26):

cumulative\_s1 += count\_s1[i]

cumulative\_s2 += count\_s2[i]

if cumulative\_s1 < cumulative\_s2 and cumulative\_s2 != len(s2):

return False

if cumulative\_s2 < cumulative\_s1 and cumulative\_s1 != len(s1):

return False

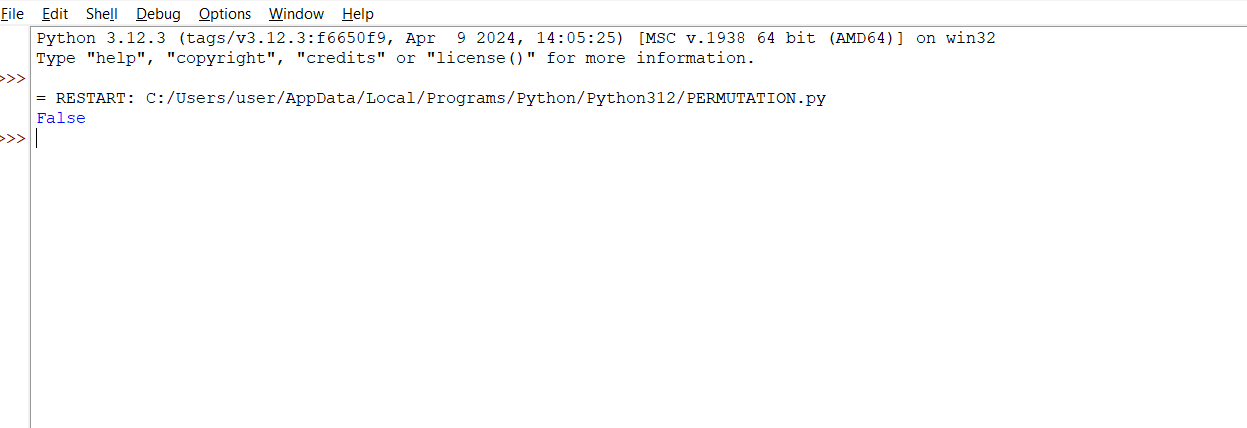
return True

s1 = "abc"

s2 = "xya"

print(can\_break(s1, s2))

TIME COMPLEXITY: O(n)



**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":**

def minimize\_string(s):

n = len(s)

cost = 0

result = ''

for i in range(n):

if s[i] == '?':

current\_cost = min(result.count(chr(ord('a') + j)) for j in range(26))

cost += current\_cost

result += chr(ord('a') + current\_cost)

else:

result += s[i]

return cost, result

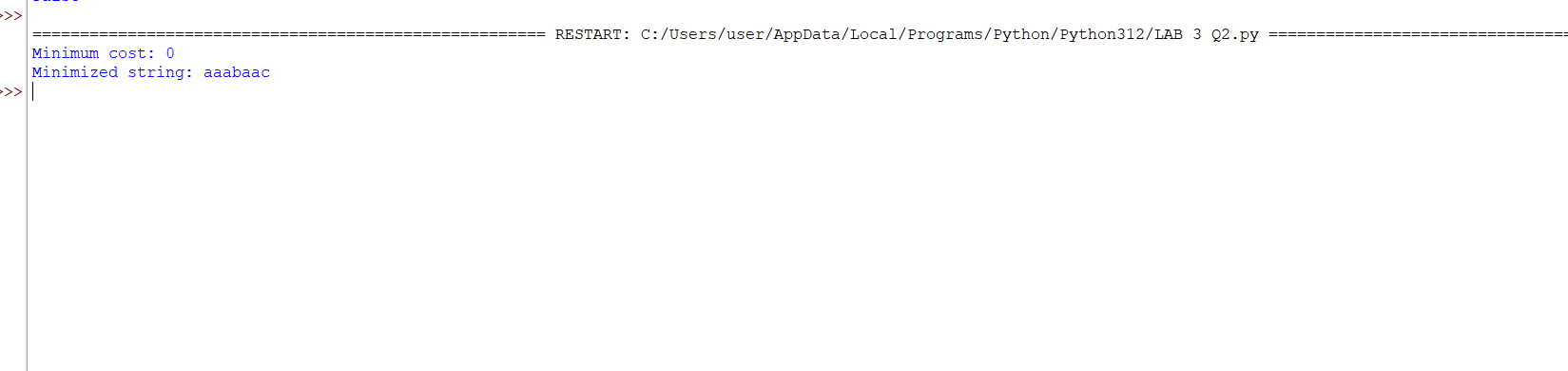
s = "a??b??c"

min\_cost, minimized\_s = minimize\_string(s)

print("Minimum cost:", min\_cost)

print("Minimized string:", minimized\_s)

TIME COMPLEXITY = O(n\*26) = O(n)



**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".**

def last\_string\_before\_empty(s):

char\_index = {chr(ord('a') + i): -1 for i in range(26)}

for i, ch in enumerate(s):

if ch in char\_index:

char\_index[ch] = i

last\_removed = min(char\_index.values())

result = ''.join(ch for i, ch in enumerate(s) if i < last\_removed)

return result

s = "aabcbbca"

print(last\_string\_before\_empty(s))

TIME COMPLEXITY : O(n)

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

**Example 1:**

**Input: nums = [-2,1,-3,4,-1,2,1,-5,4]**

**Output: 6**

**Explanation: The subarray [4,-1,2,1] has the largest sum 6.**

def max\_subarray\_sum(nums):

max\_sum = nums[0]

current\_sum = nums[0]

for num in nums[1:]:

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

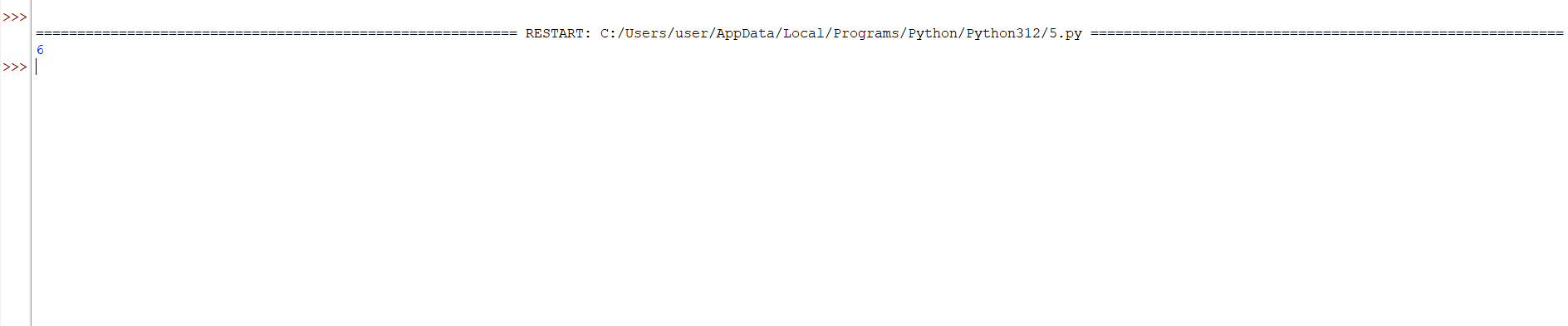
max\_sum = max(max\_sum, current\_sum)

return max\_sum

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

print(max\_subarray\_sum(nums))

TIME COMPLEXITY: O(n)



**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.**

class TreeNode:

def \_\_init\_\_(self, val):

self.val = val

self.left = None

self.right = None

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

if not nums:

return None

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

root = TreeNode(nums[max\_index])

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

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

return root

def inorder\_traversal(root):

if root is None:

return []

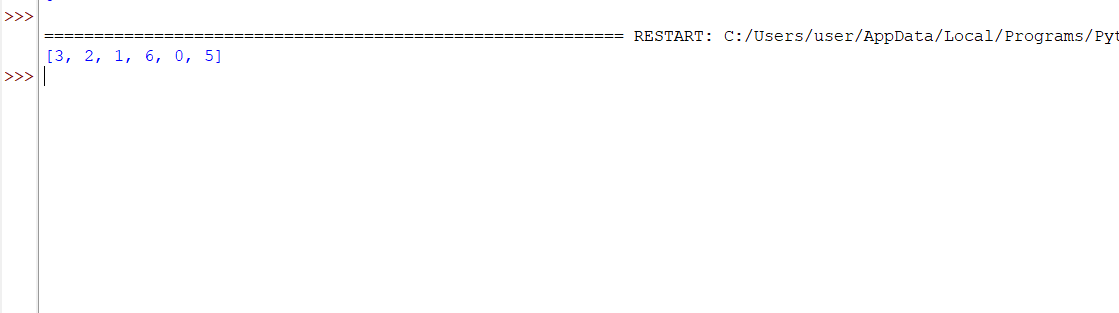
return inorder\_traversal(root.left) + [root.val] + inorder\_traversal(root.right)

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

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

print(inorder\_traversal(root))

TIME COMPLEXITY: O(nlogn)+O(n)



**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.**

def max\_subarray\_sum(nums):

max\_sum = float('-inf')

current\_sum = 0

for num in nums:

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

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

return max\_sum

def min\_subarray\_sum(nums):

min\_sum = float('inf')

current\_sum = 0

for num in nums:

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

min\_sum = min(min\_sum, current\_sum)

return min\_sum

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

n = len(nums)

max\_linear\_sum = max\_subarray\_sum(nums)

total\_sum = sum(nums)

min\_wrap\_sum = min\_subarray\_sum(nums)

max\_wrap\_sum = total\_sum - min\_wrap\_sum

if max\_wrap\_sum == 0:

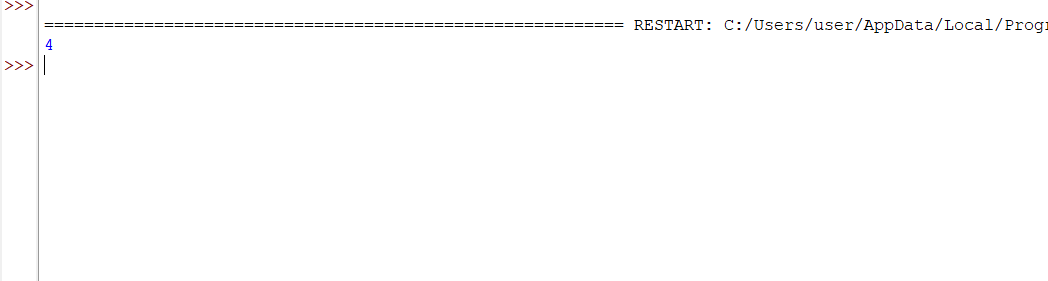
return max\_linear\_sum

return max(max\_linear\_sum, max\_wrap\_sum)

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

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

TIME COMPLEXITY: O(n)



**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**.

MOD = 10\*\*9 + 7

def maxSumAfterQueries(nums, queries):

max\_sum\_with\_last = 0

max\_sum\_without\_last = 0

for num in nums:

max\_sum\_with\_last = (max\_sum\_with\_last + max(num, 0)) % MOD

max\_sum\_without\_last = (max\_sum\_without\_last + max(num, 0)) % MOD

result = []

for posi, xi in queries:

prev\_num = nums[posi]

nums[posi] = xi

max\_sum\_with\_last = (max\_sum\_with\_last + max(0, xi) - max(0, prev\_num)) % MOD

max\_sum\_without\_last = (max\_sum\_without\_last + max(0, xi) - max(0, prev\_num)) % MOD

result.append(max(max\_sum\_with\_last, max\_sum\_without\_last))

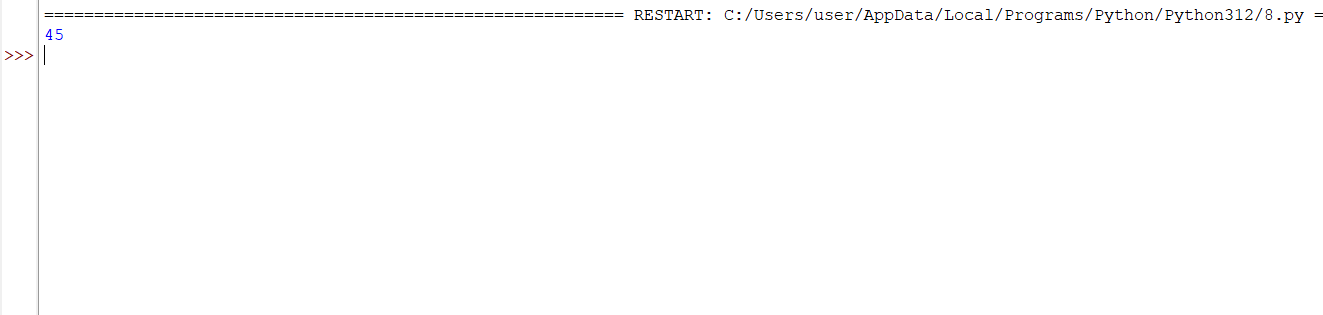
return sum(result) % MOD

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

queries = [[1, 2], [0, -2], [2, 5]]

print(maxSumAfterQueries(nums, queries))

TIME COMPLEXITY: O(n+m)



**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).**

import heapq

def kClosest(points, k):

def distance(point):

return point[0] \*\* 2 + point[1] \*\* 2

heap = []

for point in points:

dist = distance(point)

heapq.heappush(heap, (dist, point))

result = []

for \_ in range(k):

result.append(heapq.heappop(heap)[1])

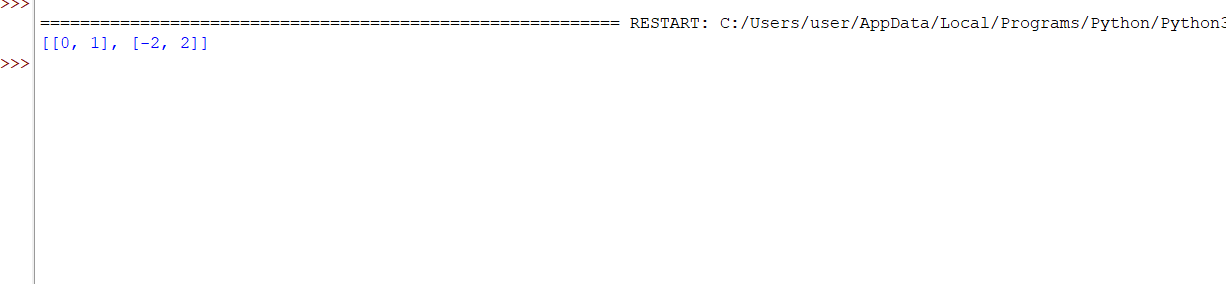
return result

points = [[1,3],[-2,2],[5,8],[0,1]]

k = 2

print(kClosest(points, k))

TIME COMPLEXITY: O(nlogn)



**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)).**

def findMedianSortedArrays(nums1, nums2):

if len(nums1) > len(nums2):

nums1, nums2 = nums2, nums1

m, n = len(nums1), len(nums2)

total\_length = m + n

left, right = 0, m

while left <= right:

partition\_nums1 = (left + right) // 2

partition\_nums2 = (total\_length + 1) // 2 - partition\_nums1

max\_left\_nums1 = float('-inf') if partition\_nums1 == 0 else nums1[partition\_nums1 - 1]

min\_right\_nums1 = float('inf') if partition\_nums1 == m else nums1[partition\_nums1]

max\_left\_nums2 = float('-inf') if partition\_nums2 == 0 else nums2[partition\_nums2 - 1]

min\_right\_nums2 = float('inf') if partition\_nums2 == n else nums2[partition\_nums2]

if max\_left\_nums1 <= min\_right\_nums2 and max\_left\_nums2 <= min\_right\_nums1:

if total\_length % 2 == 0:

return (max(max\_left\_nums1, max\_left\_nums2) + min(min\_right\_nums1, min\_right\_nums2)) / 2

else:

return max(max\_left\_nums1, max\_left\_nums2)

elif max\_left\_nums1 > min\_right\_nums2:

right = partition\_nums1 - 1

else:

left = partition\_nums1 + 1

nums1 = [1, 3]

nums2 = [2]

print(findMedianSortedArrays(nums1, nums2))

TIME COMPLEXITY: O(log(min(n,m))

