Easy 1:

Code:

def length\_of\_last\_word(s):

if not (1 <= len(s) <= 104):

return "Invalid input: String length should be between 1 and 104 characters."

if not all(ch.isalpha() or ch == ' ' for ch in s):

return "Invalid input: String should consist only of English letters and spaces."

words = s.split()

return len(words[-1])

s = input("Enter a string: ")

result = length\_of\_last\_word(s)

print("Length of the last word:", result)

Logic explanation

Check if the string length is within the allowed range. Verify that the string contains only English letters and spaces. Split the validated string into words using spaces as separators.Remove any empty strings resulting from consecutive spaces.Retrieve the last word from the list of words obtained after splitting.calculate and return the length of this last word.

Medium 2

def majority\_elements(nums): count1, count2, candidate1, candidate2 = 0, 0, None, None for num in nums: if candidate1 == num: count1 += 1 elif candidate2 == num: count2 += 1 elif count1 == 0: candidate1, count1 = num, 1 elif count2 == 0: candidate2, count2 = num, 1 else: count1 -= 1 count2 -= 1 count1 = count2 = 0 for num in nums: if num == candidate1: count1 += 1 elif num == candidate2: count2 += 1 n = len(nums) result = [] if count1 > n // 3: result.append(candidate1) if count2 > n // 3: result.append(candidate2) return resultinputs = input("Enter the numbers separated by spaces: ")input\_nums = list(map(int, inputs.split()))output = majority\_elements(input\_nums)print("Elements appearing more than ⌊ n/3 ⌋ times:", output)

Logic Explanation:

The algorithm employs the Boyer-Moore Majority Vote algorithm to find potential candidates that might appear more than ⌊ n/3 ⌋ times in the array. It iterates through the array to identify two potential candidates that might satisfy the given condition.It employs a voting mechanism to maintain these candidates based on their counts and decrementing counts when different candidates are encountered. It checks if these candidates appear more than ⌊ n/3 ⌋ times in the array and adds them to the result list accordingly. Returns the elements that appear more than ⌊ n/3 ⌋ times in the given array, adhering to the constraints provided.

Hard 2

def shortest\_palindrome(s):

rev\_s = s[::-1

full\_str = s + "#" + rev\_s # Use a special character as separator

lps = [0] \* len(full\_str)

j = 0

for i in range(1, len(full\_str)):

while j > 0 and full\_str[i] != full\_str[j]:

j = lps[j - 1]

if full\_str[i] == full\_str[j]:

j += 1

lps[i] = j

return rev\_s[:len(s) - lps[-1]] + s

user\_input = input("Enter a string: ")

result = shortest\_palindrome(user\_input)

print("Shortest palindrome:", result)

Logic Explanation

Generate full\_str by concatenating s, a special separator character (here, #), and rev\_s. Use the KMP algorithm to create the Longest Prefix Suffix array (lps) for the full\_str. This array stores the length of the longest prefix that matches the suffix at each index. The shortest palindrome can be obtained by appending a portion of rev\_s (the reversed string) to s.Determine the portion by excluding the common prefix obtained from the LPS array. Return the concatenated string, consisting of the appropriate portion of rev\_s followed by the original s, forming the shortest palindrome achievable by adding characters in front of s.