Experiment 7

Student Name: Karan Goyal **UID:-22BCS15864**

Branch: BE-CSE **Group-Ntpp_IOT_602-A**

Semester: 5TH **Dateof Performance:7-3-25**

Subject Name:-Advance Programming lab-2 **Subject Code:-22CSP-351**

Problem 1

Aim:- You are assigned to put some amount of boxes onto **one truck**. You are given a 2D array boxTypes, where boxTypes[i] = [numberOfBoxes$_i$, numberOfUnitsPerBox$_i$]:

- numberOfBoxes$_i$ is the number of boxes of type i.
- numberOfUnitsPerBox$_i$ is the number of units in each box of the type i.

You are also given an integer truckSize, which is the **maximum** number of **boxes** that can be put on the truck. You can choose any boxes to put on the truck as long as the number of boxes does not exceed truckSize.

Return *the **maximum** total number of **units** that can be put on the truck.*

.

Objective:- The objective is to maximize the total number of units loaded onto a truck with a limited capacity. Given different box types, each with a specific number of boxes and units per box, the goal is to select and load boxes efficiently without exceeding the truck's size constraint, prioritizing boxes with the highest units per box.

Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

Algorithm for Maximum Units on a Truck

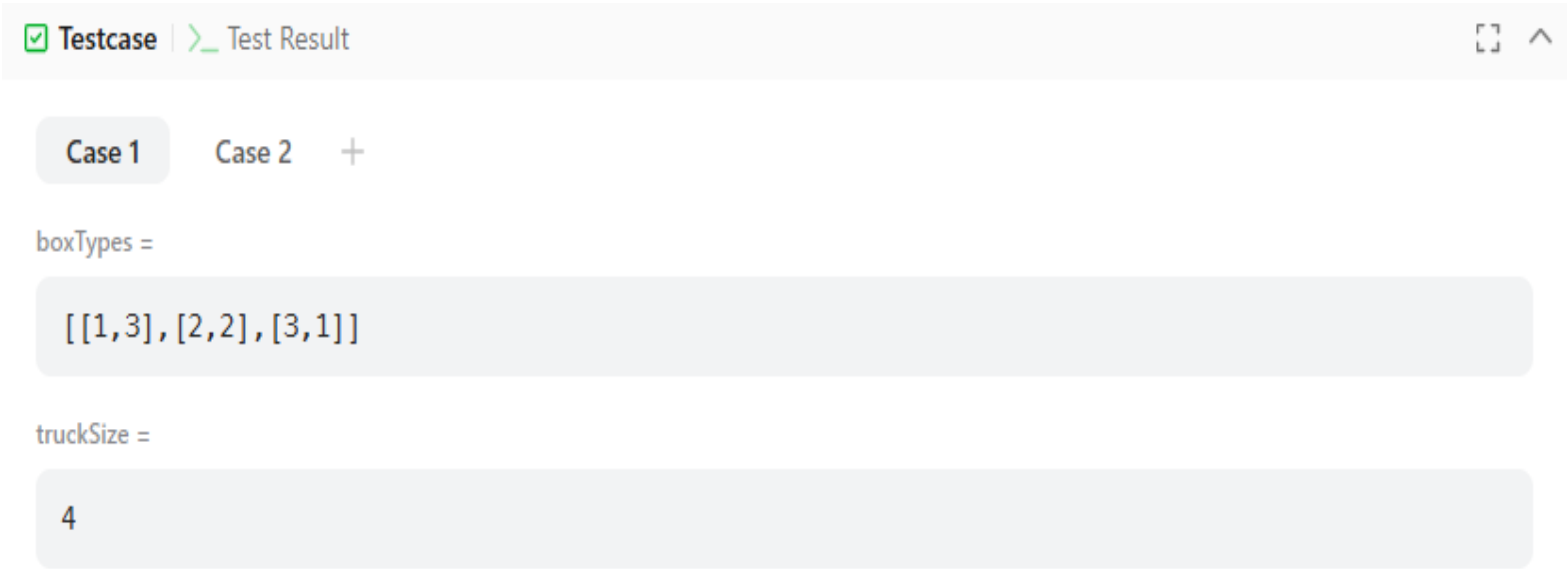
1. **Sort the Box Types**
   - Sort the boxTypes array in descending order based on the number of units per box.
2. **Initialize Variables**
   - ans = 0 (to store the maximum units that can be loaded).
   - truckSize (remaining capacity of the truck).
3. **Iterate Through the Sorted Box List**
   - For each box type:
     - If the truck can accommodate all boxes of the current type, load them completely and update ans.
     - Otherwise, load only as many boxes as the truck can hold and update ans.
     - If the truck is full, stop.
4. **Return the Maximum Units**
   - Output the total number of units loaded on the truck.

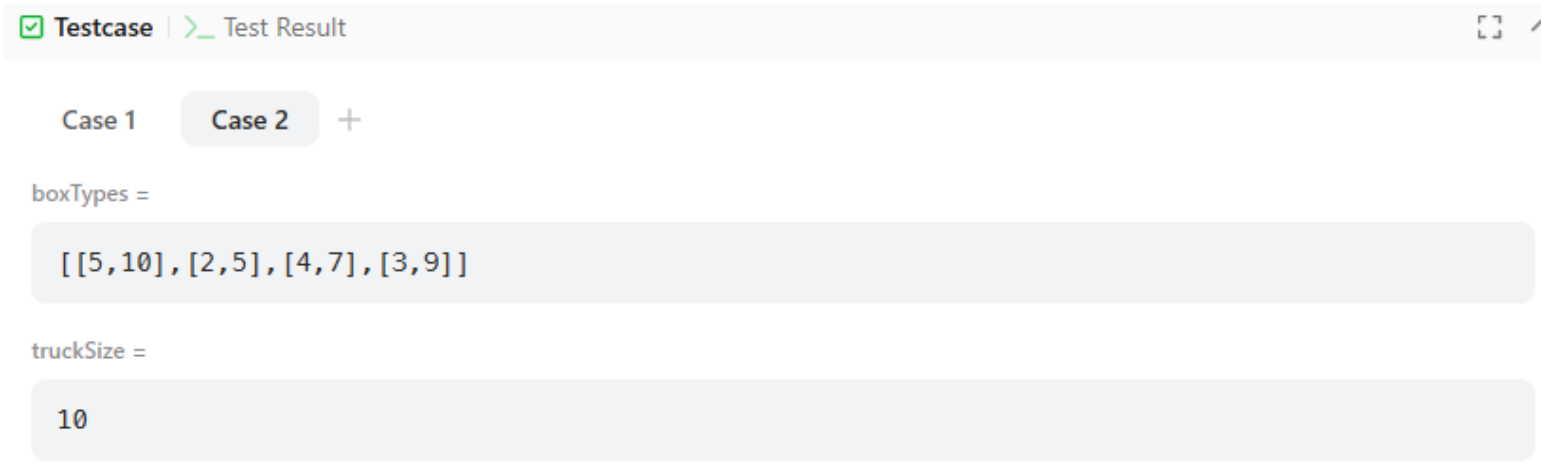
Code:

```
Class Solution {
public:
    static bool cmp(vector<int>&x , vector<int>&y){
          return x[1] > y[1];
    }

    int maximumUnits(vector<vector<int>>& boxTypes, int truckSize) {
        int n=boxTypes.size();
        sort(boxTypes.begin(),boxTypes.end(),cmp);
        int ans = 0;
        for(int i=0;(i<n) && truckSize ;i++){
            if( boxTypes[i][0] <= truckSize ){
                ans = ans + boxTypes[i][0] * boxTypes[i][1];
                truckSize = truckSize - boxTypes[i][0];
            }
            else{
                ans = ans+ boxTypes[i][1] * truckSize ;
                break;
            }
        }
        return ans;

    }
};
```

Time Complexity: O(n log n) (due to sorting)
Space Complexity: O(1) (in-place sorting and constant extra space)

Output- All the test cases passed





Here are five brief learning outcomes:

1. **Understand Business Failures** – Identify key reasons businesses fail, including poor planning, lack of adaptation, and financial mismanagement.
2. **Importance of Patents** – Learn how patents and technical/business plans help engineers promote innovation within companies.
3. **Risk in Market Expansion** – Recognize that expanding into new markets with new products carries the highest risk.
4. **Startup Challenges** – Analyze why startups often fail, with cash flow issues being the most common reason.
5. **Proactive Issue Resolution** – Understand the importance of addressing product defects early to maintain customer trust and business reputation.

Problem-2

Aim:- You are given a **0-indexed** integer array piles, where piles[i] represents the number of stones in the i^{th} pile, and an integer k. You should apply the following operation **exactly** k times:

- Choose any piles[i] and **remove** floor(piles[i] / 2) stones from it.

Notice that you can apply the operation on the **same** pile more than once.

Return *the **minimum** possible total number of stones remaining after applying the* k *operations.*

floor(x) is the **greatest** integer that is **smaller** than or **equal** to x (i.e., rounds x down).

Objective- The objective is to minimize the total number of stones remaining after performing exactly **k** operations on a given array of stone piles. In each operation, half of the stones from any chosen pile are removed. The goal is to strategically reduce the largest piles first to achieve the minimum possible total stone count.

Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

Algorithm

1. **Initialize a Max-Heap (Priority Queue)**
   - Create a max-heap to efficiently access the largest stone pile.
   - Initialize a variable ans to store the sum of all stones.
2. **Insert All Elements into the Max-Heap**
   - Iterate through the array nums, pushing each element into the max-heap.
   - Add the element's value to ans to keep track of the total sum of stones.
3. **Perform k Operations**
   - Repeat k times (or until the heap is empty):
     - Extract the largest element (top) from the max-heap.
     - Compute removed = top / 2, which represents half of the largest pile (floor division).
     - Subtract removed from ans since these stones are removed.
     - Insert top - removed (the remaining stones) back into the max-heap.
4. **Return the Final Sum**
   - After k operations, return ans, which represents the minimum possible sum of stones remaining.

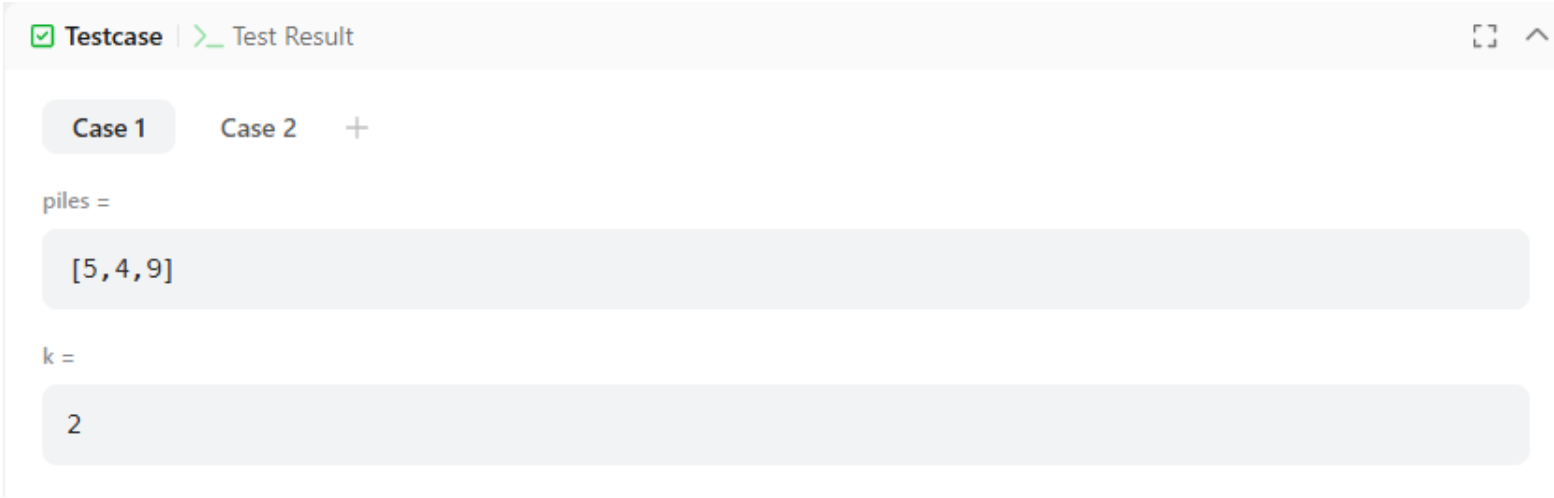
Code-

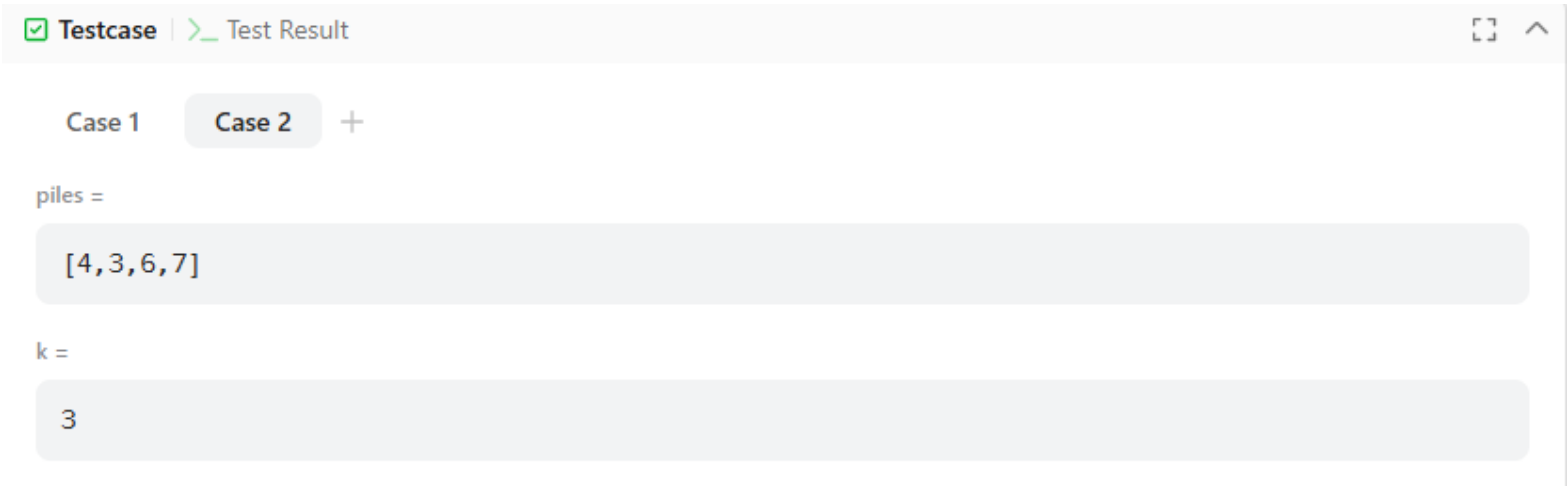
```
class Solution {
public:
   int minStoneSum(vector<int>& nums, int k) {
      priority_queue<int> maxHeap;
      int ans = 0;

      for (int num : nums) {
         maxHeap.push(num);
         ans += num;
      }
```

```
        while (k-- && !maxHeap.empty()) {
            int top = maxHeap.top();
            maxHeap.pop();
            int removed = top / 2 ;
            ans -= removed;
            maxHeap.push(top - removed);
        }

        return ans;
    }
};
```

Time Complexity: O(n + k log n) (heap operations)
Space Complexity: O(n) (priority queue storage)

Result-All test cases passes





Learning Outcomes:

1. Understand the use of heaps (priority queues) for greedy optimization problems.
2. Learn how to efficiently remove the largest elements first to minimize a sum.
3. Gain experience in implementing floor division operations in iterative processes.
4. Analyze the time and space complexity of heap-based solutions.
5. Improve problem-solving skills in greedy algorithms and heap manipulations.

Problem-3

Aim- A You are given a string s and two integers x and y. You can perform two types of operations any number of times.Remove substring "ab" and gain x points.For example, when removing "ab" from "cabxbae" it becomes "cxbae".

Remove substring "ba" and gain y points.For example, when removing "ba" from "cabxbae" it becomes "cabxe".
Return *the maximum points you can gain after applying the above operations on* s.

Objective- The goal is to maximize the points gained by removing substrings from a given string **s**. You can remove "ab" for **x** points and "ba" for **y** points, performing these operations any number of times. The challenge is to determine the optimal order of removals to achieve the highest possible score efficiently.

Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

Algorithm

Define a Helper Function getCount

- It takes a string str, a target substring sub, and two reference counters cnt1 and cnt2.
- It searches for occurrences of the substring sub and removes them while increasing cnt1.
- Then, it searches for occurrences of sub in reverse order and removes them while increasing cnt2.

Initialize Counters

- mxABcnt: Count of "ab" removed first.
- mxBAcnt: Count of "ba" removed first.
- minBAcnt: Remaining "ba" after "ab" removals.
- minABcnt: Remaining "ab" after "ba" removals.

Call getCount to Count Substring Removals

- First, count occurrences of "ab", then "ba" in the remaining string.
- Then, count occurrences of "ba", then "ab" in the remaining string.

Compute Maximum Gain

- Two different approaches:
    - Removing "ab" first, then "ba": operation1 = mxABcnt * x + minBAcnt * y
    - Removing "ba" first, then "ab": operation2 = mxBAcnt * y + minABcnt * x
- Return the maximum of operation1 and operation2.

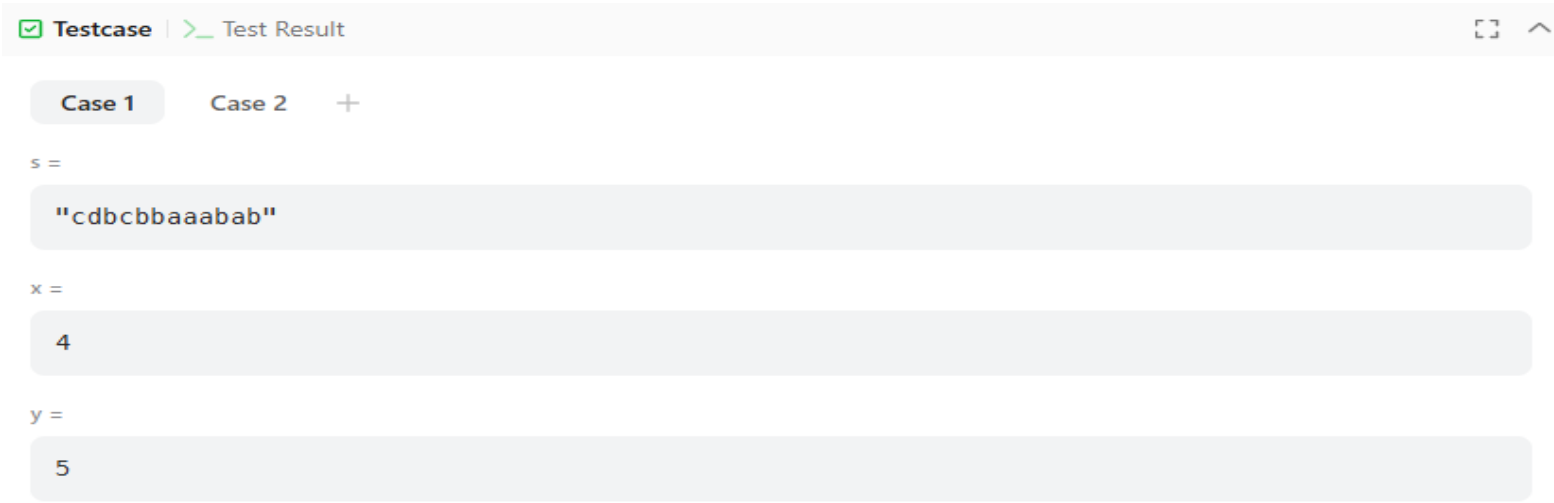
Code-

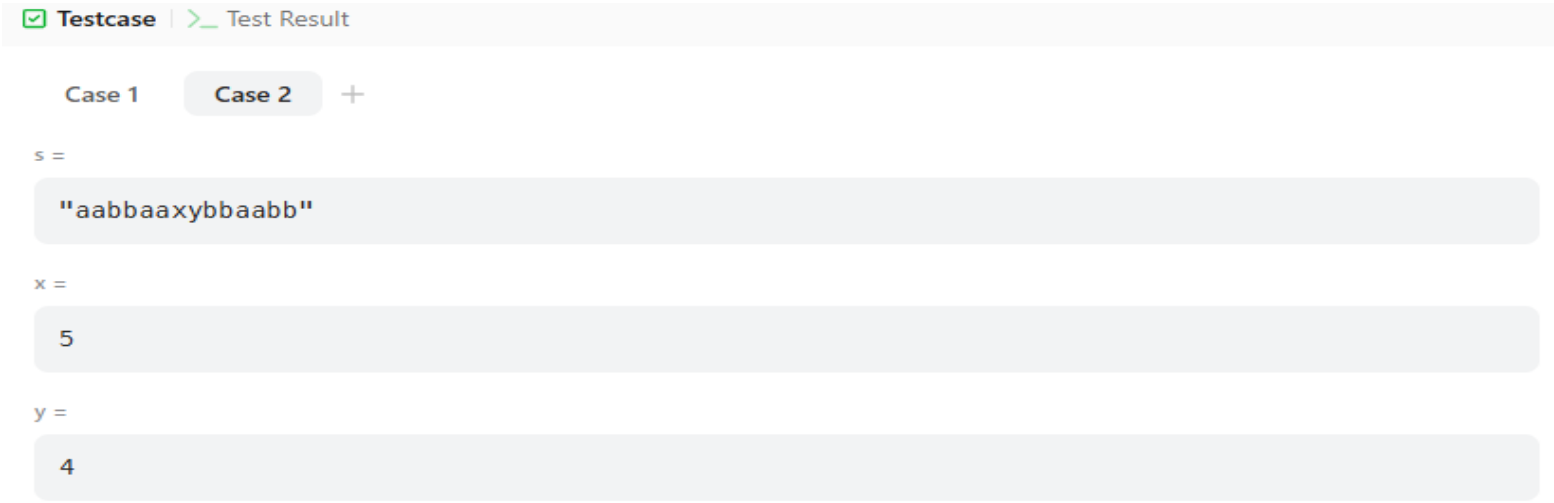
```
class Solution {
    void getCount(string str, string sub, int& cnt1, int& cnt2) {

        char first = sub[0], second = sub[1];
        int i = 1;
        while(i < str.length()) {
            if(i > 0 && str[i-1] == first && str[i] == second) {
                cnt1++;
                str.erase(i-1, 2);
                i--;
                continue;
            }
            i++;
        }

        i = 1;
        while(i < str.length()) {
            if(i > 0 && str[i-1] == second && str[i] == first) {
                cnt2++;
                str.erase(i-1, 2);
                i--;
                continue;
            }
            i++;
        }
        return;
    }
public:
    int maximumGain(string s, int x, int y) {

        int mxABcnt = 0;
        int mxBAcnt = 0;
        int minBAcnt = 0;
```

```
        int minABcnt= 0;

        getCount(s, "ab", mxABcnt, minBAcnt);
        getCount(s, "ba", mxBAcnt, minABcnt);

        int operation1 = mxABcnt * x + minBAcnt * y;
        int operation2 = mxBAcnt * y + minABcnt * x;
        return max(operation1, operation2);
    }
};
```

Time Complexity: O(n)
Space Complexity: O(n)

Result- All test cases passes





Learning Outcomes:

1. **Efficient Substring Removal for Maximum Gain**
   - Learn how to remove specific substrings ("ab" and "ba") in a way that maximizes the total score.
   - Understand how different orders of removal can lead to different outcomes.
   - Apply a greedy strategy to determine which substring should be removed first for an optimal result.
2. **Understanding the Role of Greedy Algorithms in String Manipulation**
   - The problem is solved using a greedy approach, which prioritizes removing the substring that provides the highest gain first.
   - If x > y, removing "ab" first gives a better result; if y > x, removing "ba" first is preferred.
   - This ensures that the maximum possible score is achieved in an optimal order.
3. **Efficient String Modification and Iteration Techniques**
   - Gain experience in modifying a string while iterating over it without affecting performance.
   - Understand how erase() in C++ can be used to remove specific characters dynamically.
   - Learn how to traverse a string and modify it in O(n) time complexity instead of using nested loops.
4. **Time Complexity and Performance Optimization**
   - Analyze the O(n) time complexity, where n is the length of the input string.
   - Understand why using two passes through the string is sufficient for counting and removing substrings optimally.
   - Compare this approach with brute-force methods (O(n^2)) that would be inefficient for large strings.
5. **Problem-Solving in String Processing and Algorithm Design**
   - Develop skills in solving real-world string manipulation problems using algorithms.
   - Learn how to break down a complex problem into manageable subproblems (handling "ab" and "ba" separately).
   - Understand how to write modular code by implementing helper functions like getCount() to improve code reusability and clarity.