Day 15 / 100 :

Topic - Array, String

1 Problem statement: Substring with largest variance (Hard)

The variance of a string is defined as the largest difference between the number of occurrences of any 2 characters present in the string. Note the two characters may or may not be the same.
Given a string s consisting of lowercase English letters only, return the largest variance possible among all substrings of s.
A substring is a contiguous sequence of characters within a string.

Example 1:

Input: s = "aababbb"
Output: 3
Explanation:
All possible variances along with their respective substrings are listed below:
- Variance 0 for substrings "a", "aa", "ab", "abab", "aababb", "ba", "b", "bb", and "bbb".
- Variance 1 for substrings "aab", "aba", "abb", "aabab", "ababb", "aababbb", and "bab".
- Variance 2 for substrings "aaba", "ababbb", "abbb", and "babb".
- Variance 3 for substring "babbb".
Since the largest possible variance is 3, we return it.
Example 2:

Input: s = "abcde"
Out

Solutions :

Approach 1 - Brute Force

put: 0
Explanation:
No letter occurs more than once in s, so the variance of every substring is 0.

1. The maxVarianceSubstring function takes a string s as input and returns the maximum variance among all substrings of s.
2. In the main function, we define a string s = "aababbb" as an example.
3. Inside the maxVarianceSubstring function, we initialize the maxVariance variable to 0.
4. We use two nested loops to iterate through all possible substrings of s.

5. The outer loop starts at index i and the inner loop starts at index j (from i to the end of the string).
6. In each iteration of the inner loop, we calculate the frequency count of each character in the current substring.
7. We use a vector freq of size 26 (one for each lowercase English letter) to store the frequency count of each character.
8. To calculate the frequency count, we increment the count of the corresponding character in freq.
9. After updating the frequency count, we find the maximum and minimum frequency using the std::max_element and std::min_element functions, respectively.
10. We calculate the variance as the difference between the maximum and minimum frequencies.
11. Finally, we update the maxVariance with the maximum value seen so far using std::max.
12. Once all substrings have been considered, we return the maximum variance.

```
    int maxVariance = 0;

    // Iterate through all possible substrings
    for (int i = 0; i < n; i++) {
        std::vector<int> freq(26, 0);  // Frequency count of each
character

        // Calculate frequencies for substring starting at index i
        for (int j = i; j < n; j++) {
            freq[s[j] - 'a']++;  // Increment the frequency of
character s[j]
            int maxFreq = *std::max_element(freq.begin(),
freq.end());
            int minFreq = *std::min_element(freq.begin(),
freq.end());
            int variance = maxFreq - minFreq;
            maxVariance = std::max(maxVariance, variance);
        }
    }

    return maxVariance;
```

Solutions :

Approach 2 - Kadane's Algorithm

Intuition:

Upon reviewing the code, my initial thoughts on solving this problem are as follows:

1. The code appears to be finding the largest variance between two letters in a given string. Variance here refers to the absolute difference in the count of occurrences of two letters, while maintaining the order of the letters.
2. The code utilizes a nested loop to iterate through all possible pairs of letters (i and j) in the range 'a' to 'z'. It then calculates the variance between these two letters by counting their occurrences in the string.
3. The code uses two counters, c1 and c2, to keep track of the occurrences of letters i and j while iterating through the string.
4. The maximum variance found so far is stored in the variable ans and is updated whenever a larger variance is found.
5. The code also reverses the string s after each iteration of the innermost loop, allowing it to calculate the variance in both the forward and backward directions.
6. The final result, the largest variance found, is returned by the function.

Overall, the code employs a brute-force approach by checking all possible pairs of letters and calculating the variance between them. It iterates through the string twice, considering both forward and backward directions. However, without additional context or problem statement, it is difficult to determine the exact purpose or intended use of this code.

Approach:

Based on the code provided, the approach to solving the problem seems to involve the following steps:

1. Create a vector arr of size 26 to store the frequency of each letter from 'a' to 'z' in the input string s.

2. Iterate through each character x in the string s and increment the corresponding frequency count in the arr vector using x - 'a' as the index.

3. Initialize a variable ans to store the maximum variance found.

4. Iterate through each pair of letters i and j from 'a' to 'z' using nested loops. Skip the iteration if i and j are the same or if either of their frequency counts in arr is zero.

5. For each pair of letters i and j, perform the following steps twice (forward and backward):

   a. Initialize two variables, A1 and A2, to represent the counters for letters i and j, respectively.

   b. Iterate through each character x in the string s.

   c. If x is equal to i, increment A1. If x is equal to j, increment A2.

   d. If A2 becomes greater than A1, reset both A1 and A2 to zero.

   e. If both A1 and A2 are greater than zero, update ans with the maximum difference between A1 and A2 so far.

   f. Reverse the string s using reverse(s.begin(), s.end()) to perform the same calculation in the backward direction.

After all iterations, return the maximum variance ans.

The code follows a brute-force approach by checking all possible pairs of letters and calculating the variance between them while considering the order of the letters in the string. However, without additional context or a specific problem statement, it is challenging to determine the exact purpose or optimal approach for solving this problem.

Complexity:
Time complexity: O(N)
Space complexity: O(1)

```
class Solution {
public:
    int largestVariance(string s) {
        vector<int> arr(26); // Array to store the frequency of each
letter (a-z)
        for(auto x : s) {
```

```
            arr[x - 'a']++; // Increment the count for the
corresponding letter
        }
        int ans = 0; // Variable to store the maximum variance found
        for(char i = 'a'; i <= 'z'; i++) { // Iterate through each
letter i
            for(char j = 'a'; j <= 'z'; j++) { // Iterate through
each letter j
                // Skip if j is the same as i or either i or j has a
frequency of 0
                if(j == i || arr[i - 'a'] == 0 || arr[j - 'a'] == 0)
                    continue;
                for(int k = 1; k <= 2; k++) { // Perform the
calculation twice (forward and backward)
                    int A1 = 0; // Counter for letter i
                    int A2 = 0; // Counter for letter j
                    for(auto x : s) {
                        if(x == i)
                            A1++; // Increment A1 if the current
letter is i
                        if(x == j)
                            A2++; // Increment A2 if the current
letter is j
                        if(A2 > A1) {
                            A1 = 0;
                            A2 = 0; // Reset A1 and A2 if A2 becomes
greater than A1
                        }
                        if(A1 > 0 && A2 > 0)
                            ans = max(ans, A1 - A2); // Update the
maximum variance if A1 - A2 is greater
                    }
                    reverse(s.begin(), s.end()); // Reverse the
string for the next iteration
                }
            }
        }
        return ans; // Return the maximum variance found
    }};
```

2 Problem statement: Roman to Integer (Easy)

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

For example, 2 is written as II in Roman numeral, just two ones added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

I can be placed before V (5) and X (10) to make 4 and 9.
X can be placed before L (50) and C (100) to make 40 and 90.
C can be placed before D (500) and M (1000) to make 400 and 900.
Given a roman numeral, convert it to an integer.

Example 1:

Input: s = "III"
Output: 3
Explanation: III = 3.
Example 2:

Input: s = "LVIII"
Output: 58
Explanation: L = 50, V= 5, III = 3.

Solutions :

Approach 2 - Hashmap + Maths

Intuition:
We can solve this problem using strings + Hash Table + Math.

Approach:
We can easily understand the approach by seeing the code, which is easy to understand with comments. Roman numeral
Complexity:
Time complexity:
Time Complexity : O(1), The maximum length of the string(s) can be 15 (as per the Constraint), therefore, the worst case time complexity can be O (15) or O(1).

Space complexity:
Space Complexity : O(1), We are using unordered_map(map) to store the Roman symbols and their corresponding integer values, but there are only 7 symbols hence the worst case space complexity can be O(7) which is equivalent to O(1).

Code

```
/*

    Time Complexity : O(1), The maximum length of the string(s) can be 15
(as per the Constraint), therefore, the
    worst case time complexity can be O(15) or O(1).

    Space Complexity : O(1), We are using unordered_map(map) to store the
Roman symbols and their corresponding
    integer values but there are only 7 symbols hence the worst case space
complexity can be O(7) which is
    equivalent to O(1).

    Solved using String + Hash Table + Math.

*/

class Solution {
public:
    int romanToInt(string s) {
        unordered_map<char, int> map;

        map['I'] = 1;
        map['V'] = 5;
        map['X'] = 10;
        map['L'] = 50;
        map['C'] = 100;
        map['D'] = 500;
        map['M'] = 1000;
```

```
        int ans = 0;

        for(int i=0; i<s.length(); i++){
            if(map[s[i]] < map[s[i+1]]){
                ans -= map[s[i]];
            }
            else{
                ans += map[s[i]];
            }
        }
        return ans;
    }
};
```