**Easy 1**

Given a string s consisting of words and spaces, return *the length of the* ***last*** *word in the string.*

A **word** is a maximal

substring consisting of non-space characters only.

#include <iostream>

#include <string>

class Solution {

public:

int lengthOfLastWord(std::string s) {

int c = 0;

for (int i = s.size() - 1; i >= 0; i--) {

if (s[i] == ' ' && c == 0)

continue;

else if (s[i] != ' ')

c++;

else

break;

}

return c;

}

};

int main() {

Solution solution;

std::string inputString = " fly me to the moon ";

int result = solution.lengthOfLastWord(inputString);

std::cout << "Length of the last word: " << result << std::endl;

return 0;

}

The provided C++ code defines a class Solution with a function lengthOfLastWord that calculates and returns the length of the last word in a given string. The main function demonstrates the functionality by creating an instance of the class and applying it to a specific input string.

Approach:

Initialize Counter (c) to 0: Start with a counter set to 0, which will be used to track the length of the last word.

Iterate Backward Through String (s): Traverse the characters of the input string in reverse order, starting from the end.

Skip Leading Spaces: Ignore leading spaces until the first non-space character is encountered.

Count Characters of Last Word: Begin counting characters as soon as a non-space character is found.

Stop at First Space after Last Word: Continue counting until the first space after the last word is encountered.

Return Length of Last Word (c): The function returns the count, representing the length of the last word.

Example:

For the input string " fly me to the moon ", the last word is "moon," and its length is 4. Therefore, the output of the program is "Length of the last word: 4."

Key Point:

The approach ensures efficiency by iterating backward and avoiding unnecessary computations by skipping leading spaces and stopping once the last word is processed.

**Medium 2**

Given an integer array of size n, find all elements that appear more than ⌊ n/3 ⌋ times.

#include <bits/stdc++.h>

using namespace std;

vector<int> majorityElement(vector<int> v) {

int n = v.size();

vector<int> ls;

map<int, int> mpp;

int mini = int(n / 3) + 1;

for (int i = 0; i < n; i++) {

mpp[v[i]]++;

if (mpp[v[i]] == mini) {

ls.push\_back(v[i]);

}

if (ls.size() == 2) break;

}

return ls;

}

int main()

{

vector<int> arr={3,2,3};

vector<int> ans = majorityElement(arr);

cout << "The majority elements are: ";

for (auto it : ans)

cout << it << " ";

cout << "\n";

return 0;

}

The provided C++ code aims to find majority elements in a given vector. A majority element is defined as an element that appears more than ⌊n/3⌋ times, where n is the size of the vector.

Function majority Element:

Input: A vector v containing integers.

Output: A vector ls containing the majority elements.

Approach:

Uses a map (mpp) to store the count of each element.

Calculates a minimum threshold (mini) based on the vector size.

Iterates through the vector, updating the count in the map.

Checks if the count of an element meets the majority condition.

Adds the element to the result vector (ls) if it is a majority element.

Breaks the loop if two majority elements are found (as there can be at most 2).

Output: Returns the vector ls containing the majority elements.

main Function:

Initializes an input vector arr.

Calls the majorityElement function on the input vector.

Prints the majority elements found.

Key Points:

The code uses a map to efficiently count element occurrences.

The threshold ensures that an element is considered a majority element if its count is more than ⌊n/3⌋.

The loop breaks early if two majority elements are found, optimizing the algorithm.

Example Output:

For the input vector {3, 2, 3}, the majority element is 3

**Hard 2**

You are given a string s. You can convert s to a

palindrome by adding characters in front of it.

Return *the shortest palindrome you can find by performing this transformation*.

#include <iostream>

#include <vector>

#include <algorithm>

#define all(var) var.begin(), var.end()

using namespace std;

class Solution {

public:

string shortestPalindrome(string t) {

reverse(all(t));

string rev = t;

reverse(all(rev));

string s = rev + '#' + t;

int n = s.size();

vector<int> kmpp(n + 1);

kmpp[0] = -1;

int i = 0, j = -1;

while (i < n) {

while (j > 0 && s[i] != s[j]) j = kmpp[j];

j++;

i++;

kmpp[i] = j;

}

int req = t.size() - kmpp[n];

string add = "";

for (int i = 0; i < req; i++) {

add += t[i];

}

reverse(all(t));

return add + t;

}

};

int main() {

Solution solution;

string inputString = "aacecaaa";

string result = solution.shortestPalindrome(inputString);

cout << "Shortest Palindrome: " << result << endl;

return 0;

}

Approach:

1.String Reversal: The input string t is reversed using the reverse function.

2.String Concatenation (s):

A new string s is created by concatenating:The reversed string rev.

The character '#'.

The original string t.

3.KMP Algorithm:he Knuth–Morris–Pratt (KMP) algorithm is used to find the length of the longest proper prefix that is also a suffix. This information is stored in the kmpp array.

The KMP algorithm involves iterating through the characters of s and updating the kmpp array based on the pattern matching.

1. Calculate Characters to be Added (req):The number of characters required to be added to the beginning of the string is determined by subtracting the length of the longest proper prefix (found using KMP) from the total length of the original string t.

The variable req represents the count of characters that need to be added.

1. Construct the Shortest Palindrome:A string add is created by taking the first req characters from the original string t.The original string t is reversed again.The shortest palindrome is constructed by concatenating the add string with the reversed original string t.
2. Return the Result:The constructed shortest palindrome is returned.

Example:

For the input string "aacecaaa":

After reversal, t becomes "aaacecaa".

The concatenated string s becomes "aaacecaa#aaacecaaa".

The KMP algorithm is applied to find the length of the longest proper prefix that is also a suffix.

The KMP result is used to calculate that req (number of characters to be added) is 1.

The constructed shortest palindrome is "aaacecaaa".

Key Points:

Reversing the string and using the KMP algorithm efficiently finds the length of the longest proper prefix that is also a suffix.

The KMP result helps identify the portion of the string that is already a palindrome.

The algorithm then calculates the characters needed to be added to make the entire string a palindrome.

Result:

The result is the shortest palindrome that can be obtained by adding characters to the beginning of the input string.