Course Name:	Competitive Programming Lab	Semester:	IV
Date of Performance:	19/01/2026	DIV/ Batch No:	B2
Student Name:	Ashwera Hasan	Roll No:	16010124107

Experiment No: 1

Title: Implementation of Frequency Analysis of elements using Arrays in C++

Aim of the Experiment: To understand and implement the concept of frequency counting using in C++, and to develop problem-solving skills required for competitive programming.

Objective of the Experiment:

After completing this experiment, the student will be able to:

1. **Understand** the use of arrays for storing and processing large sets of data.
2. **Apply** frequency counting techniques to solve common competitive programming problems such as:
   - counting occurrences of elements,
   - identifying duplicates,
   - finding the most/least frequent elements.
3. **Design and implement** efficient C++ programs using arrays for data analysis.
4. **Analyze** the time and space complexity of frequency-based solutions.
5. **Develop logical thinking** and optimization skills required for competitive programming challenges.
6. **Enhance coding proficiency** by solving real-world problem scenarios involving array manipulation and frequency computation.

COs to be achieved:

CO 1 Applying various problem-solving paradigms, enabling them to create and implement efficient algorithms for real-world challenges.

Theory:

In competitive programming, efficient data processing is essential for solving problems within strict time limits. One of the most fundamental techniques used in problem solving is frequency counting, which determines how many times each element occurs in a given dataset. This technique is widely used in problems related to duplicate detection, mode finding, element comparison, pattern recognition, and optimization problems.

Arrays in C++

An **array** is a linear data structure that stores elements of the same data type in contiguous memory locations. Arrays provide fast access to elements using index values and are ideal for handling large amounts of input data in competitive programming.

Arrays support:

- Constant-time access: O(1)

- Efficient iteration
- Compact memory representation

Frequency Counting Technique

Frequency counting is the process of counting the number of occurrences of each element in a dataset. This is implemented using a frequency array, where the index represents the element and the value stored represents its count.

```
int freq[1000] = {0};
freq[x]++;
```

This method improves efficiency from **O(n²)** (brute force) to **O(n)**.

Importance in Competitive Programming

Frequency-based techniques are used to solve:

- Duplicate detection problems
- Permutation and anagram checking
- Finding the most and least frequent elements
- Constraint satisfaction problems
- Statistical analysis problems

Time and Space Complexity

Operation	Complexity
Input processing	O(n)
Frequency update	O(1)
Total algorithm	**O(n)**
Space complexity	O(k), where k is the maximum possible value of elements

Applications

1. **Duplicate Detection:** Identifying repeated elements in an array.
2. **Anagram Checking:** Verifying if two strings are permutations of each other.
3. **Mode Calculation:** Finding the most frequent element in a dataset.
4. **Data Analysis:** Summarizing large datasets quickly.
5. **Pattern Recognition:** Used in text processing and bioinformatics.
6. **Competitive Programming Problems:** Solving constraint-based problems efficiently.

Advantages

- **High Efficiency:** Reduces time complexity significantly.
- **Simple Implementation:** Easy to code and debug.
- **Fast Lookup:** Direct access to frequency values.
- **Low Overhead:** Uses basic arrays without complex data structures.
- **Scalable:** Handles large datasets efficiently.
- **Memory Predictability:** Memory usage is known in advance.

Problem Statements:

Difficulty (as per lab)	Problem	LeetCode mapping
Easy	Duplicate within distance K	LC 219 – Contains Duplicate II
Medium	3sum	https://leetcode.com/problems/3sum/description/
Difficult	Meet-in-the-middle subset sum (N ≤ 30)	LC 1755 – Closest Subsequence Sum

Code :

1. easy

C++ Auto

```cpp
class Solution {
public:
    bool containsNearbyDuplicate(vector<int>& v, int k) {
        int n = v.size();
        if(n<2) return false;
        map<int,int>seen;
        int l=0,r=0;
        // seen the left element
        // expand r till condition
        // shrink l and reduce freq
        seen[v[l]]++;
        while(r+1<n&&l<n){
            if(abs(r+1-l)<=k){
                r++;
                seen[v[r]]++;
                if(seen[v[r]] == 2){
                    return true;
                }
            }
            else{
                seen[v[l]]--;
                l++;
            }
        }
        return false;
    }
};
auto init = atexit([]() { ofstream("display_runtime.txt") << "0";});

```

2. medium

Submit

Code

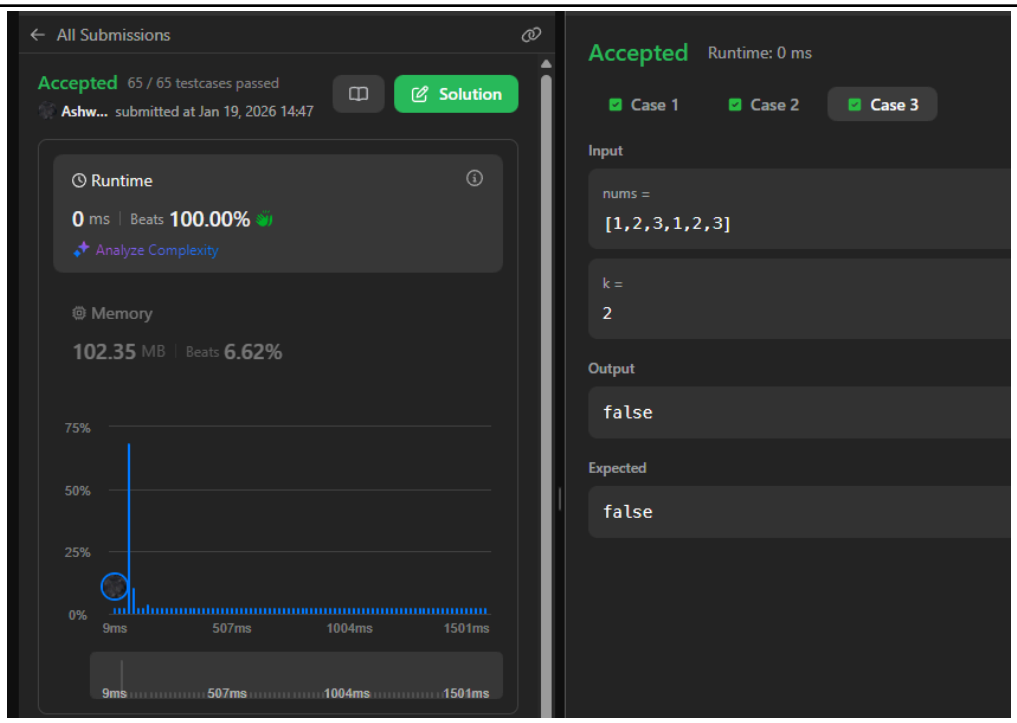
C++ Auto

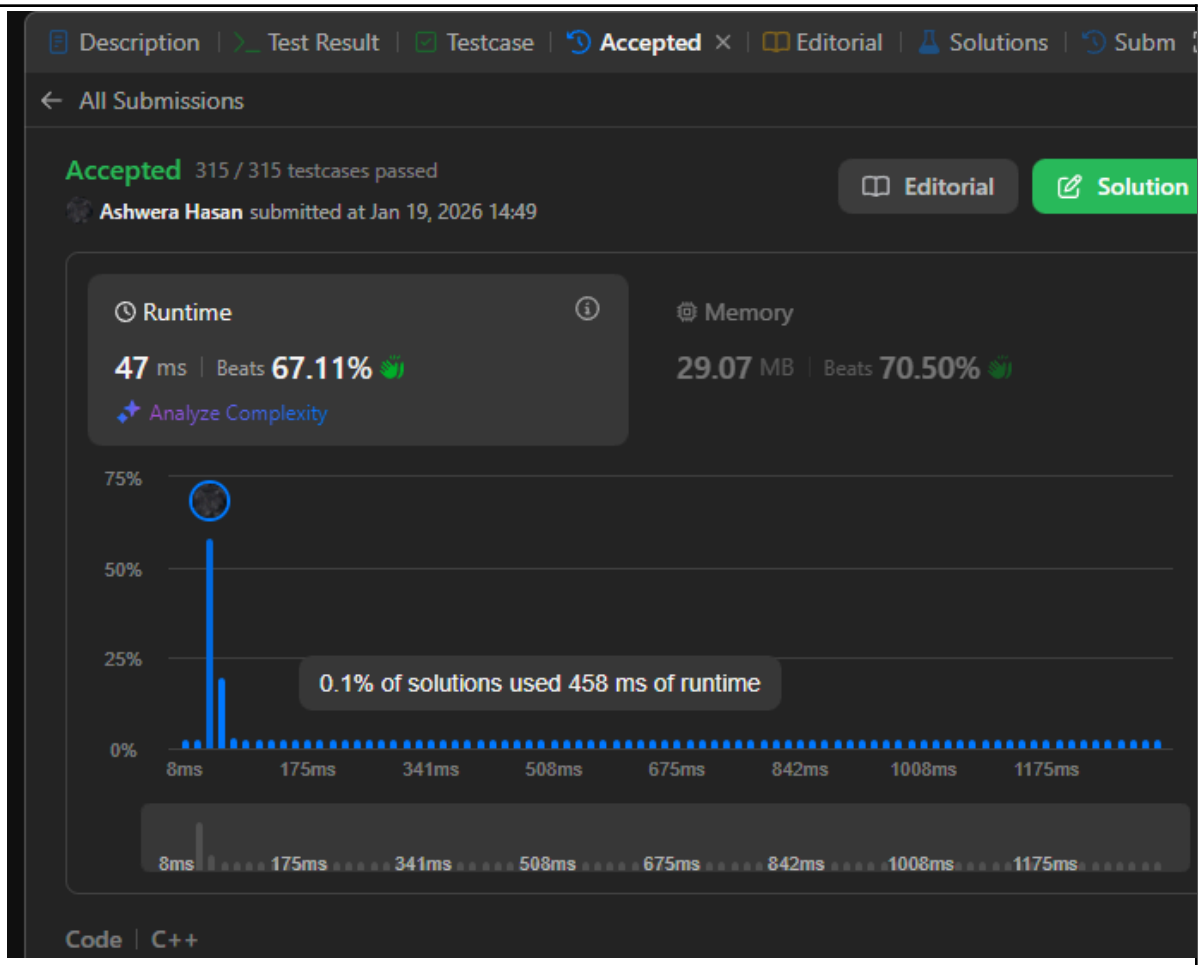
```cpp
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> result;
        int n = nums.size();

        sort(nums.begin(), nums.end());

        for (int i = 0; i < n - 2; ++i) {

            if (i > 0 && nums[i] == nums[i - 1]) continue;

            int target = -nums[i];
            int left = i + 1;
            int right = n - 1;

            while (left < right) {
                int sum = nums[left] + nums[right];
                if (sum == target) {
                    result.push_back({nums[i], nums[left], nums[right]});

                    while (left < right && nums[left] == nums[left + 1]) ++left;
                    while (left < right && nums[right] == nums[right - 1]) --right;

                    ++left;
                    --right;
                }
                else if (sum < target) {
                    ++left;
                }
                else {
                    --right;
                }
            }
        }
        return result;
    }
};

```

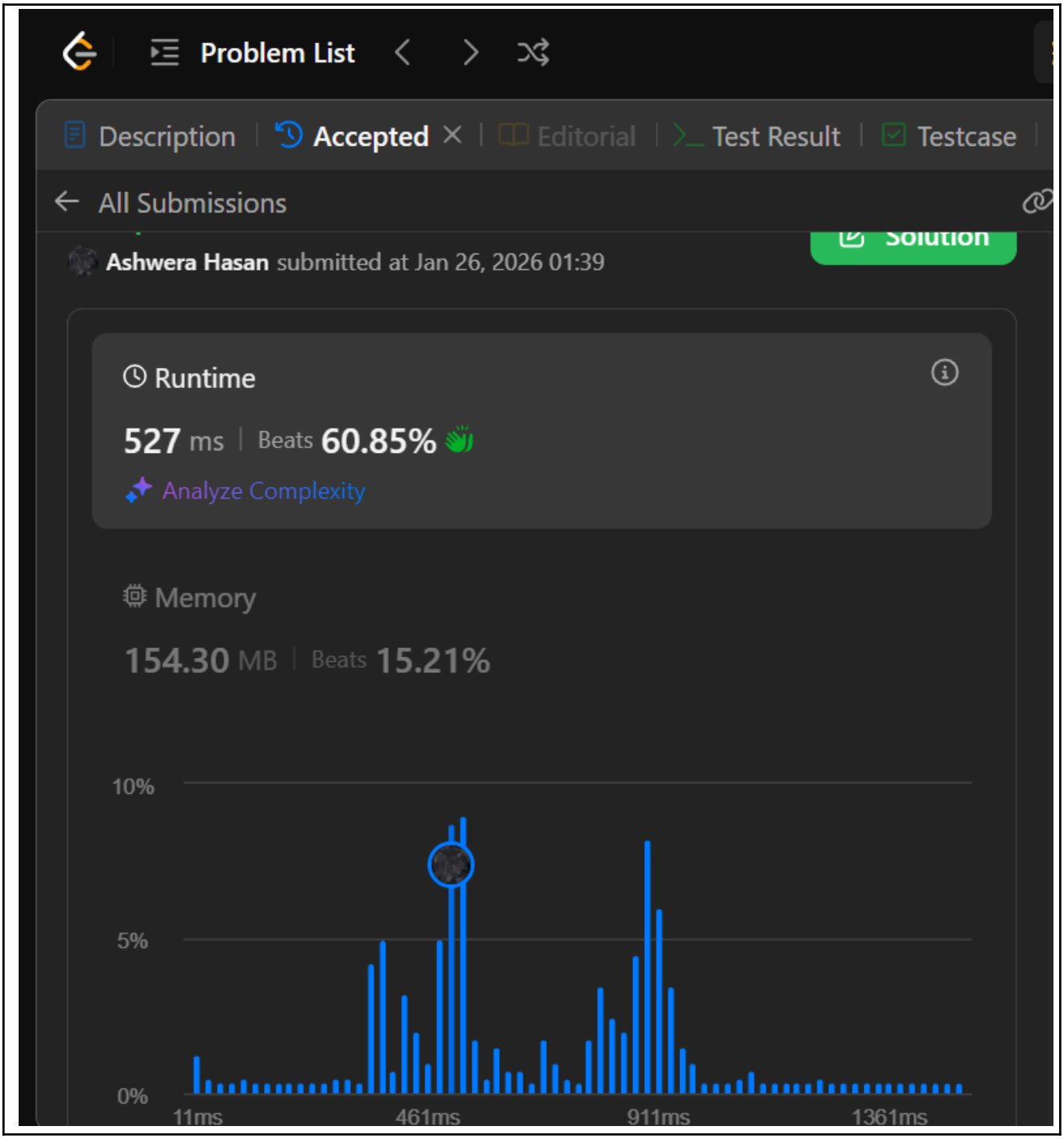
3.

```
public:
    void func(int idx, int end, vector<int>& nums, long long sum, vector<long long>& res) {
        if (idx == end) {
            res.push_back(sum);
            return;
        }
        func(idx + 1, end, nums, sum, res);
        func(idx + 1, end, nums, sum + nums[idx], res);
    }

    int minAbsDifference(vector<int>& nums, int goal) {
        int n = nums.size();
        vector<long long> leftSums, rightSums;

        func(0, n / 2, nums, 0, leftSums);
        func(n / 2, n, nums, 0, rightSums);

        sort(rightSums.begin(), rightSums.end());

        long long ans = llabs(goal);

        for (long long l : leftSums) {
            long long need = goal - l;
            auto it = lower_bound(rightSums.begin(), rightSums.end(), need);

            if (it != rightSums.end())
                ans = min(ans, llabs(l + *it - goal));
            if (it != rightSums.begin()) {
                --it;
                ans = min(ans, llabs(l + *it - goal));
            }
        }
        return (int)ans;
    }
```

Output:

1. 
All Submissions
Accepted 65 / 65 testcases passed
Ashw... submitted at Jan 19, 2026 14:47
Solution
Runtime
0 ms | Beats 100.00%
Analyze Complexity
Memory
102.35 MB | Beats 6.62%
75%
50%
25%
0%
9ms
507ms
1004ms
1501ms
Accepted Runtime: 0 ms
Case 1
Case 2
Case 3
Input
nums =
[1,2,3,1,2,3]
k =
2
Output
false
Expected
false

2. 
Description
Test Result
Testcase
Accepted
Editorial
Solutions
Subm
All Submissions
Accepted 315 / 315 testcases passed
Ashwera Hasan submitted at Jan 19, 2026 14:49
Editorial
Solution
Runtime
47 ms | Beats 67.11%
Analyze Complexity
Memory
29.07 MB | Beats 70.50%
75%
50%
25%
0%
0.1% of solutions used 458 ms of runtime
8ms
175ms
341ms
508ms
675ms
842ms
1008ms
1175ms
Code | C++
3.



Post Lab Subjective/Objective type Questions:

1. Why is frequency counting preferred over brute-force methods in competitive programming?

   Frequency counting helps maintain the number of times a number appears in a container in o(n) times. then, retrieving that data takes o(1) time per query. If we do not use the

frequency counter we will have to fetch the frequency for each query, leading to $O(n^2)$ time complexity.

2. Describe at least three applications of frequency counting in computer science.

   Frequency counting is helpful in:

   - The frequency counter pattern is used to check if two strings are anagrams of each other (contain the same characters with the same frequencies) or to identify duplicates within a dataset.
   - frequency counting is used to implement Counting Sort
   - in other greedy solutions, a frequency counter is helped to optimize solutions from O(n) lookup time to O(1).

Conclusion:

This experiment successfully helped learn the concept of sliding windows and sorting for CP questions.

Signature of faculty in-charge with Date: