Arithmetic Sequence in C++

```
#include <iostream>
#include <vector>
#include <unordered_set>
#include <algorithm>
#include <climits>

using namespace std;

bool isArithmeticSequence(const vector<int>& arr) {
    if (arr.size() <= 1) {
        return true;
    }

    int minVal = INT_MAX;
    int maxVal = INT_MIN;
    unordered_set<int> elements;

    for (int val : arr) {
        minVal = min(val, minVal);
        maxVal = max(val, maxVal);
        elements.insert(val);
    }

    int d = (maxVal - minVal) / (arr.size() - 1);

    for (size_t i = 0; i < arr.size(); ++i) {
        int ai = minVal + i * d;
        if (elements.find(ai) == elements.end()) {
            return false;
        }
    }

    return true;
}

int main() {
    vector<int> arr = {17, 9, 5, 29, 1, 25, 13, 37, 21, 33};
    cout << (isArithmeticSequence(arr) ? "true" :
"false") << endl;

    return 0;
}
```

Dry Run

Input:

arr = {17, 9, 5, 29, 1, 25, 13, 37, 21, 33}

Here is a step-by-step dry run of your C++ code, focusing on loop iterations and index-wise updates:

Step-by-Step Execution Table

First Loop (Finding `minVal`, `maxVal`, and Filling `unordered_set`)

Index (`i`)	Current `arr[i]`	Updated `minVal`	Updated `maxVal`	Updated `elements`
0	17	17	17	{17}
1	9	9	17	{9, 17}
2	5	5	17	{5, 9, 17}
3	29	5	29	{5, 9, 17, 29}
4	1	1	29	{1, 5, 9, 17, 29}
5	25	1	29	{1, 5, 9, 17, 25, 29}
6	13	1	29	{1, 5, 9, 13, 17, 25, 29}
7	37	1	37	{1, 5, 9, 13, 17, 25, 29, 37}
8	21	1	37	{1, 5, 9, 13, 17, 21, 25, 29, 37}
9	33	1	37	{1, 5, 9, 13, 17, 21, 25, 29, 33, 37}

- After this loop:
  - `minVal = 1`
  - `maxVal = 37`
  - `elements = {1, 5, 9, 13, 17, 21, 25, 29, 33, 37}`
  - `d = (37 - 1) / (10 - 1) = 4`

Second Loop (Verifying Arithmetic Sequence)

Index (`i`)	Expected Value `ai = minVal + i * d`	Check in `elements`	Result
0	`1 + 0*4 = 1`	✓ Found in `{1, 5, 9, 13, 17, 21, 25, 29, 33, 37}`	Continue
1	`1 + 1*4 = 5`	✓ Found	Continue
2	`1 + 2*4 = 9`	✓ Found	Continue
3	`1 + 3*4 = 13`	✓ Found	Continue
4	`1 + 4*4 = 17`	✓ Found	Continue
5	`1 + 5*4 = 21`	✓ Found	Continue
6	`1 + 6*4 =`	✓ Found	Continue

Index (`i`)	Expected Value `ai = minVal + i * d`	Check in `elements`	Result
	`25`		
7	`1 + 7*4 = 29`	✓ Found	Continue
8	`1 + 8*4 = 33`	✓ Found	Continue
9	`1 + 9*4 = 37`	✓ Found	Continue

- Since all expected values exist in `elements`, the function returns **`true`**.

Output:
true

Array Pair Divisible by K in C++

```
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;

void sol(const vector<int>& arr, int k) {
    unordered_map<int, int> remainderFreqMap;

    for (int val : arr) {
        int rem = val % k;
        remainderFreqMap[rem]++;
    }

    for (int val : arr) {
        int rem = val % k;

        if (rem == 0) {
            if (remainderFreqMap[rem] % 2 != 0) {
                cout << "false" << endl;
                return;
            }
        } else if (2 * rem == k) {
            if (remainderFreqMap[rem] % 2 != 0) {
                cout << "false" << endl;
                return;
            }
        } else {
            if (remainderFreqMap[rem] !=
remainderFreqMap[k - rem]) {
                cout << "false" << endl;
                return;
            }
        }
    }

    cout << "true" << endl;
}

int main() {
    vector<int> arr = {22, 12, 45, 55, 65, 78, 88, 75};
    int k = 7;
    sol(arr, k);
    return 0;
}
```

Dry Run of sol(arr, k)

arr = {22, 12, 45, 55, 65, 78, 88, 75};
k = 7;

Step 1: Compute Remainders and Store in remainderFreqMap

For each element in arr, compute rem = val % k and store it in the map:

Value (val)	rem = val % 7	remainderFreqMap (after insertion)
22	22 % 7 = 1	{1: 1}
12	12 % 7 = 5	{1: 1, 5: 1}
45	45 % 7 = 3	{1: 1, 5: 1, 3: 1}
55	55 % 7 = 6	{1: 1, 5: 1, 3: 1, 6: 1}
65	65 % 7 = 2	{1: 1, 5: 1, 3: 1, 6: 1, 2: 1}
78	78 % 7 = 1	{1: 2, 5: 1, 3: 1, 6: 1, 2: 1}
88	88 % 7 = 4	{1: 2, 5: 1, 3: 1, 6: 1, 2: 1, 4: 1}
75	75 % 7 = 5	{1: 2, 5: 2, 3: 1, 6: 1, 2: 1, 4: 1}

Final remainderFreqMap:

{1: 2, 5: 2, 3: 1, 6: 1, 2: 1, 4: 1}

Step 2: Validate Remainder Pairs

We check the conditions:

- If rem == 0, count should be even (not applicable here).
- If 2 * rem == k, count should be even (not applicable here).
- Otherwise, remainderFreqMap[rem] should match remainderFreqMap[k - rem].

Value (val)	rem = val % 7	Condition	Check
22	1	map[1] == map[6]	✗ 2 != 1

Since the condition fails, we print **"false"** and

Output: false	

Check anagram in C++

```
#include <iostream>
#include <unordered_map>
using namespace std;

bool solution(string s1, string s2) {
    unordered_map<char, int> map;

    // Count frequencies of characters in s1
    for (char ch : s1) {
        map[ch]++;
    }

    // Check characters in s2 against the frequency map
    for (char ch : s2) {
        if (map.find(ch) == map.end()) {
            return false; // Character not found in s1
        } else if (map[ch] == 1) {
            map.erase(ch); // Remove entry if frequency becomes zero
        } else {
            map[ch]--; // Decrement the count of the character
        }
    }

    // If map is empty, all characters from s1 and s2 match in frequency
    return map.empty();
}

int main() {
    string s1 = "pepcoding";
    string s2 = "codingpep";
    cout << boolalpha << solution(s1, s2) << endl; // Output: true

    return 0;
}
```

Dry Run for solution Function

Input:

- s1 = "pepcoding"
- s2 = "codingpep"

Step-by-Step Execution

Step 1: Count frequencies of characters in s1

Character (ch)	Frequency in map (map[ch])
'p'	2
'e'	1
'c'	1
'o'	1
'd'	1
'i'	1
'n'	1
'g'	1

Map after Step 1:

map = {'p': 2, 'e': 1, 'c': 1, 'o': 1, 'd': 1, 'i': 1, 'n': 1, 'g': 1}

Step 2: Process characters in s2

Character (ch)	Action Taken	Updated map
'c'	Found in map, decrement map['c']	{'p': 2, 'e': 1, 'o': 1, 'd': 1, 'i': 1, 'n': 1, 'g': 1}
'o'	Found in map, decrement map['o']	{'p': 2, 'e': 1, 'd': 1, 'i': 1, 'n': 1, 'g': 1}
'd'	Found in map, decrement map['d']	{'p': 2, 'e': 1, 'i': 1, 'n': 1, 'g': 1}
'i'	Found in map, decrement map['i']	{'p': 2, 'e': 1, 'n': 1, 'g': 1}
'n'	Found in map, decrement map['n']	{'p': 2, 'e': 1, 'g': 1}
'g'	Found in map, decrement map['g']	{'p': 2, 'e': 1}
'p'	Found in map, decrement map['p']	{'p': 1, 'e': 1}
'e'	Found in map, decrement	{'p': 1}

	<table><tr><th>Character (ch)</th><th>Action Taken</th><th>Updated map</th></tr><tr><td></td><td>map['e']</td><td></td></tr><tr><td>'p'</td><td>Found in map, decrement map['p']</td><td>{}</td></tr></table> **Step 3: Final Check** • Is map empty? Yes, map is empty, indicating all characters in s2 match the frequencies in s1. **Output:** true
Output: true	

Contiguous Array in C++

```
#include <iostream>
#include <unordered_map>
using namespace std;

int sol(int arr[], int n) {
    int ans = 0;
    unordered_map<int, int> map;
    map[0] = -1;
    int sum = 0;

    for (int i = 0; i < n; i++) {
        if (arr[i] == 0) {
            sum += -1;
        } else if (arr[i] == 1) {
            sum += +1;
        }

        if (map.find(sum) != map.end()) {
            int idx = map[sum];
            int len = i - idx;
            if (len > ans) {
                ans = len;
            }
        } else {
            map[sum] = i;
        }
    }
    return ans;
}

int main() {
    int arr[] = {0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << sol(arr, n) << endl; // Output: 10

    return 0;
}
```

Dry Run:

Given input:

```
int arr[] = {0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1};
int n = sizeof(arr) / sizeof(arr[0]);
```

Step-by-Step Breakdown:

Initial Values:

- ans = 0 (stores the longest subarray length)
- map = {0: -1} (maps cumulative sum to the first occurrence index)
- sum = 0 (initial cumulative sum)

Iteration by Iteration Walkthrough:

i	arr[i]	sum (cumulative sum)	map (sum -> index)	Length (len)	Updated ans
0	0	-1	{0: -1, -1: 0}	0 - (-1) = 1	1
1	0	-2	{0: -1, -1: 0, -2: 1}	1 - (-1) = 2	2
2	1	-1	{0: -1, -1: 0, -2: 1}	2 - 0 = 2	2
3	0	-2	{0: -1, -1: 0, -2: 1}	3 - 1 = 2	2
4	1	-1	{0: -1, -1: 0, -2: 1}	4 - 0 = 4	4
5	0	-2	{0: -1, -1: 0, -2: 1}	5 - 1 = 4	4
6	1	-1	{0: -1, -1: 0, -2: 1}	6 - 0 = 6	6
7	1	0	{0: -1, -1: 0, -2: 1}	7 - (-1) = 8	8
8	0	-1	{0: -1, -1: 0, -2: 1}	8 - 0 = 8	8
9	0	-2	{0: -1, -1: 0, -2: 1}	9 - 1 = 8	8
10	1	-1	{0: -1, -1: 0, -2: 1}	10 - 0 = 10	10
11	1	0	{0: -1, -1: 0,	11 - (-1) = 12	12

			-2: 1}		
12	1	1	{0: -1, -1: 0, -2: 1}	12 - (-1) = 14	14

Correct Analysis:

- The **longest subarray** with equal numbers of 0s and 1s spans from index 2 to 11 (inclusive), making the subarray length **12**.

Final Output:

12

Output:
12

Count of Subarrays Having Sum Equal to K in C++

```
#include <iostream>
#include <unordered_map>
#include <vector>

using namespace std;

int solution(vector<int>& arr, int target) {
    int ans = 0;
    unordered_map<int, int> map;
    map[0] = 1; // Initialize with sum 0 having
count 1
    int sum = 0;

    for (int i = 0; i < arr.size(); i++) {
        sum += arr[i];
        if (map.find(sum - target) != map.end()) {
            ans += map[sum - target];
        }
        map[sum]++;
    }

    return ans;
}

int main() {
    vector<int> arr = {1, 1, 1};
    int target = 2;
    cout << solution(arr, target) << endl; //
Output: 2

    return 0;
}
```

Dry Run for Input:

vector<int> arr = {1, 1, 1};
int target = 2;

Initial Values:

- ans = 0
- map = {0: 1} (since map[0] = 1 initially)
- sum = 0

Iteration Breakdown:

i	arr[i]	sum (cumulative sum)	sum - target	map[sum - target]	ans	map (updated)
0	1	1	1 - 2 = -1	Not found	0	{0: 1, 1: 1}
1	1	2	2 - 2 = 0	map[0] = 1 (found)	1	{0: 1, 1: 1, 2: 1}
2	1	3	3 - 2 = 1	map[1] = 1 (found)	2	{0: 1, 1: 2, 2: 1, 3: 1}

Explanation of each iteration:

- **At i = 0**:
  - arr[0] = 1
  - sum = 1
  - We check if sum - target = 1 - 2 = -1 is in map. It is **not**.
  - We update the map with map[1]++, so map = {0: 1, 1: 1}.
- **At i = 1**:
  - arr[1] = 1
  - sum = 2
  - We check if sum - target = 2 - 2 = 0 is in map. It **is** (map[0] = 1), so we add 1 to ans (i.e., ans += 1).
  - We update the map with map[2]++, so map = {0: 1, 1: 1, 2: 1}.
- **At i = 2**:
  - arr[2] = 1
  - sum = 3
  - We check if sum - target = 3 - 2 = 1 is in map. It **is** (map[1] = 1), so we add 1 to ans (i.e., ans += 1).
  - We update the map with map[3]++, so map = {0: 1, 1: 2, 2: 1, 3: 1}.

Final Output:

- The total number of subarrays whose sum equals target = 2 is **2**.

Output:
2

Count Of Subarrays With Equal 0 and 1 in C++

```
#include <iostream>
#include <unordered_map>
#include <vector>

using namespace std;

int solution(vector<int>& arr) {
    unordered_map<int, int> map;
    int ans = 0;
    map[0] = 1; // Initialize with sum 0 having
count 1
    int sum = 0;

    for (int val : arr) {
        // Treat 0 as -1 for sum calculation
        if (val == 0) {
            sum += -1;
        } else {
            sum += 1;
        }

        if (map.find(sum) != map.end()) {
            ans += map[sum];
            map[sum]++;
        } else {
            map[sum] = 1;
        }
    }

    return ans;
}

int main() {
    vector<int> arr = {0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1,
1, 1};
    cout << solution(arr) << endl; // Output the
result

    return 0;
}
```

Dry Run for Input:

vector<int> arr = {0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1};

Initial Values:

- ans = 0
- map = {0: 1}
- sum = 0

Iteration Breakdown:

i	arr[i]	sum (cumulative sum)	map[sum]	ans (after update)	map (updated)
0	0	-1	map[-1] = 0	0	{0: 1, -1: 1}
1	0	-2	map[-2] = 0	0	{0: 1, -1: 1, -2: 1}
2	1	-1	map[-1] = 1	1	{0: 1, -1: 2, -2: 1}
3	0	-2	map[-2] = 1	1	{0: 1, -1: 2, -2: 2}
4	1	-1	map[-1] = 2	3	{0: 1, -1: 3, -2: 2}
5	0	-2	map[-2] = 2	3	{0: 1, -1: 3, -2: 3}
6	1	-1	map[-1] = 3	6	{0: 1, -1: 4, -2: 3}
7	1	0	map[0] = 1	7	{0: 2, -1: 4, -2: 3}
8	0	-1	map[-1] = 4	11	{0: 2, -1: 5, -2: 3}
9	0	-2	map[-2] = 3	14	{0: 2, -1: 5, -2: 4}
10	1	-1	map[-1] = 5	19	{0: 2, -1: 6, -2: 4}
11	1	0	map[0] = 2	21	{0: 3, -1: 6, -2: 4}
12	1	1	map[1] = 0	24	{0: 3, -1: 6, -2: 4, 1: 1}

Output:
24

Count Of Zeros Sum Subarray in C++

```
#include <iostream>
#include <unordered_map>
#include <vector>

using namespace std;

int sol(const vector<int>& arr) {
    int count = 0;
    unordered_map<int, int> map;
    int sum = 0;
    map[0] = 1;

    for (int i = 0; i < arr.size(); ++i) {
        sum += arr[i];

        if (map.find(sum) != map.end()) {
            count += map[sum];
            map[sum]++;
        } else {
            map[sum] = 1;
        }
    }

    return count;
}

int main() {
    vector<int> arr = {2, 8, -3, -5, 2, -4, 6, 1, 2, 1,
-3, 4};
    int result = sol(arr);
    cout << result << endl;
    return 0;
}
```

Dry Run:

Initial Values:

- count = 0
- map = {0: 1}
- sum = 0

Iteration Breakdown:

i	arr[i]	sum (cumulative sum)	map[sum]	count (after update)	map (updated)
0	2	2	map[2] = 0	0	{0: 1, 2: 1}
1	8	10	map[10] = 0	0	{0: 1, 2: 1, 10: 1}
2	-3	7	map[7] = 0	0	{0: 1, 2: 1, 10: 1, 7: 1}
3	-5	2	map[2] = 1	1	{0: 1, 2: 2, 10: 1, 7: 1}
4	2	4	map[4] = 0	1	{0: 1, 2: 2, 10: 1, 7: 1, 4: 1}
5	-4	0	map[0] = 1	2	{0: 2, 2: 2, 10: 1, 7: 1, 4: 1}
6	6	6	map[6] = 0	2	{0: 2, 2: 2, 10: 1, 7: 1, 4: 1, 6: 1}
7	1	7	map[7] = 1	3	{0: 2, 2: 2, 10: 1, 7: 2, 4: 1, 6: 1}
8	2	9	map[9] = 0	3	{0: 2, 2: 2, 10: 1, 7: 2, 4: 1, 6: 1, 9: 1}
9	1	10	map[10] = 1	4	{0: 2, 2: 2, 10: 2, 7: 2, 4: 1, 6: 1, 9: 1}
10	-3	7	map[7] = 2	6	{0: 2, 2: 2, 10: 2, 7: 3, 4: 1, 6: 1, 9: 1}
11	4	11	map[11] = 0	6	{0: 2, 2: 2, 10: 2, 7: 3, 4: 1, 6: 1, 9: 1, 11: 1}

Final Values:

- **count = 6**
- **map = {0: 2, 2: 2, 10: 2, 7: 3, 4: 1, 6: 1, 9: 1, 11: 1}**

	Output: The total number of subarrays with sum equal to 0 is **6**. **Final Output:** 6
Output: 6	

Distinct Elements Window of Size K in C++

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <deque>

using namespace std;

vector<int> distinctElementsInWindow(const
vector<int>& arr, int k) {
    vector<int> result;
    unordered_map<int, int> frequencyMap;
    int n = arr.size();
    int i = 0;

    // Initialize the frequency map for the first window
    for (i = 0; i < k - 1; ++i) {
        frequencyMap[arr[i]]++;
    }

    for (int j = -1; i < n; ++i, ++j) {
        // Add the next element (i-th element) to the
frequency map
        frequencyMap[arr[i]]++;

        // Record the number of distinct elements in the
current window
        result.push_back(frequencyMap.size());

        // Remove the (j-th element) as the window slides
        if (j >= 0) {
            if (frequencyMap[arr[j]] == 1) {
                frequencyMap.erase(arr[j]);
            } else {
                frequencyMap[arr[j]]--;
            }
        }
    }

    return result;
}

int main() {
    vector<int> arr = {2, 5, 5, 6, 3, 2, 3, 2, 4, 5, 2, 2, 2, 2,
3, 6};
    int k = 4;
    vector<int> result =
distinctElementsInWindow(arr, k);

    for (int num : result) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Dry Run:

Initialize:

- **arr = [2, 5, 5, 6, 3, 2, 3, 2, 4, 5, 2, 2, 2, 2, 3, 6]**
- **k = 4**
- **frequencyMap** = {} (Empty at the start)
- **result** = [] (Empty at the start)

Step-by-Step Iteration:

i	arr[i]	frequencyMap (Updated)	Distinct Elements	result (after update)	j
0	2	{2: 1}	1	[]	-1
1	5	{2: 1, 5: 1}	2	[]	0
2	5	{2: 1, 5: 2}	2	[]	1
3	6	{2: 1, 5: 2, 6: 1}	3	[3]	2
4	3	{2: 1, 5: 1, 6: 1, 3: 1}	4	[3, 4]	3
5	2	{2: 2, 5: 1, 6: 1, 3: 1}	4	[3, 4, 4]	4
6	3	{2: 2, 5: 1, 6: 1, 3: 2}	3	[3, 4, 4, 3]	5
7	2	{2: 3, 5: 1, 6: 1, 3: 2}	3	[3, 4, 4, 3, 3]	6
8	4	{2: 3, 5: 1, 6: 1, 3: 2, 4: 1}	4	[3, 4, 4, 3, 3, 4]	7
9	5	{2: 3, 5: 2, 6: 1, 3: 2, 4: 1}	4	[3, 4, 4, 3, 3, 4, 4]	8
10	2	{2: 4, 5: 2, 6: 1, 3: 2, 4: 1}	3	[3, 4, 4, 3, 3, 4, 4, 3]	9
11	2	{2: 5, 5: 2, 6: 1, 3: 2, 4: 1}	2	[3, 4, 4, 3, 3, 4, 4, 3, 3]	10
12	2	{2: 6, 5: 2, 6: 1, 3: 2, 4: 1}	1	[3, 4, 4, 3, 3, 4, 4, 3, 3, 2]	11
13	2	{2: 7, 5: 2, 6: 1, 3: 2, 4: 1}	1	[3, 4, 4, 3, 3, 4, 4, 3, 3, 2, 2]	12
14	3	{2: 7, 5: 2, 6: 1, 3: 3, 4: 1}	2	[3, 4, 4, 3, 3, 4, 4, 3, 3, 2, 2,	13

				3]	
15	6	{2: 7, 5: 2, 6: 2, 3: 3, 4: 1}	3	[3, 4, 4, 3, 3, 4, 4, 3, 3, 2, 2, 3, 3]	14

Final Result:

The output is the list of distinct elements in each sliding window of size k as the window slides across the array:

Output:

3 4 4 4 3 3 4 4 3 3 2 2 3

Output:
3 4 4 4 3 3 4 4 3 3 2 2 3

Employees Under Manager in C++

```
#include <iostream>
#include <unordered_map>
#include <unordered_set>
#include <string>

using namespace std;

int getSize(unordered_map<string,
unordered_set<string>>& tree, const
string& manager, unordered_map<string,
int>& result) {
    if (tree.find(manager) == tree.end()) {
        result[manager] = 0;
        return 1;
    }
    int size = 0;
    for (const string& employee :
tree[manager]) {
        int currentSize = getSize(tree,
employee, result);
        size += currentSize;
    }
    result[manager] = size;
    return size + 1;
}

void findCount(unordered_map<string,
string>& map) {
    unordered_map<string,
unordered_set<string>> tree;
    string ceo = "";

    for (const auto& entry : map) {
        string employee = entry.first;
        string manager = entry.second;

        if (manager == employee) {
            ceo = manager;
        } else {
            tree[manager].insert(employee);
        }
    }

    unordered_map<string, int> result;
    getSize(tree, ceo, result);

    for (const auto& entry : result) {
        cout << entry.first << " " <<
entry.second << endl;
    }
}

int main() {
    unordered_map<string, string> map;
    map["A"] = "C";
    map["B"] = "C";
    map["C"] = "F";
    map["D"] = "E";
    map["E"] = "F";
    map["F"] = "F";
```

Step 1: Construct tree and Identify CEO

- Input mapping:

  A -> C
  B -> C
  C -> F
  D -> E
  E -> F
  F -> F (CEO identified)

- Constructing tree:

  C -> {A, B}
  F -> {C, E}
  E -> {D}

- **CEO Identified**: F

Step 2: Recursive Calls of getSize(tree, manager, result)

Function Call	Processing Employee Set	Recursive Calls	Result Updates (result[manager])	Return Value
getSize(tree, "F", result)	{C, E}	getSize(tree, "C"), getSize(tree, "E")	F → 5	6
getSize(tree, "C", result)	{A, B}	getSize(tree, "A"), getSize(tree, "B")	C → 2	3
getSize(tree, "A", result)	{} (Base Case)	-	A → 0	1
getSize(tree, "B", result)	{} (Base Case)	-	B → 0	1
getSize(tree, "E", result)	{D}	getSize(tree, "D")	E → 1	2
getSize(tree, "D", result)	{} (Base Case)	-	D → 0	1

Step 3: Output Values

Final result map:

mathematica
CopyEdit
A → 0
B → 0
C → 2

findCount(map);
 	
return 0;
}
	D → 0
	E → 1
	F → 5
	
	Final Output
	
	A 0
	B 0
	C 2
	D 0
	E 1
	F 5

Output: F 5 E 1 B 0 A 0 D 0 C 2	

Equivalent Subarrays in C++

```
#include <iostream>
#include <unordered_map>
#include <unordered_set>
#include <vector>

using namespace std;

int main() {
    int ans = 0;
    vector<int> arr = {2, 1, 3, 2, 3};
    unordered_set<int> set;

    // Insert unique elements into the set
    for (int i = 0; i < arr.size(); i++) {
        set.insert(arr[i]);
    }

    int k = set.size();
    int i = -1;
    int j = -1;
    unordered_map<int, int> map;

    while (true) {
        bool f1 = false;
        bool f2 = false;

        // Expand the window until all unique elements are covered
        while (i < arr.size() - 1) {
            f1 = true;
            i++;
            map[arr[i]] = map[arr[i]] + 1; // Add current element to the map
            if (map.size() == k) { // If all unique elements are covered
                ans += arr.size() - i; // Add the number of valid subarrays ending at index i
                break;
            }
        }

        // Slide the window to the right until the uniqueness condition is violated
        while (j < i) {
            f2 = true;
            j++;
            if (map[arr[j]] == 1) {
                map.erase(arr[j]); // Remove element from map if its count is reduced to 0
            } else {
                map[arr[j]] = map[arr[j]] - 1; // Decrease the count of the element
            }

            // If the map size matches k, add the number of valid subarrays again
            if (map.size() == k) {
                ans += arr.size() - i;
```

Step 1: Initializing Variables

- **Input Array**: `{2, 1, 3, 2, 3}`
- **Unique Elements (`set`)**:

  `{2, 1, 3} → k = 3 (total unique elements)`

- **Pointers**:

  ```
  i = -1, j = -1
  ans = 0
  map = {} (empty frequency map)
  ```

Step 2: Expanding the Window (Outer `while` Loop)

Expanding `i` Until `map.size() == k`

`i`	`arr[i]`	`map` (after update)	`map.size()`	Condition `map.size() == k`?
0	2	`{2: 1}`	1	✗
1	1	`{2: 1, 1: 1}`	2	✗
2	3	`{2: 1, 1: 1, 3: 1}`	3	✓ → Add `arr.size() - i = 5 - 2 = 3` to `ans`

- **`ans = 3`**

Step 3: Contracting `j` Until `map.size() < k`

`j`	`arr[j]`	`map` (after update)	`map.size()`	Condition `map.size() == k`?	`ans` Update
0	2	`{2: 0, 1: 1, 3: 1}` → removed 2	2	✗	**Break**

Step 4: Continue Expanding `i`

`i`	`arr[i]`	`map` (after update)	`map.size()`	Condition `map.size() == k`?	`ans` Update
3	2	`{1: 1, 3: 1, 2: 1}`	3	✓	Add `arr.size() - i = 5 - 3 = 2`
New `ans`	**`3 + 2 = 5`**				

```
            } else {
                break;
            }
        }

        // If both windows cannot be expanded or contracted further, break the loop
        if (!f1 && !f2) {
            break;
        }
    }

    // Print the total number of equivalent subarrays
    cout << ans << endl;

    return 0;
}
```

Step 5: Contracting `j` Again

`j`	`arr[j]`	`map` (after update)	`map.size()`	Condition `map.size() == k`?	`ans` Update
1	1	`{1: 0, 3: 1, 2: 1}` → removed 1	2	✗	**Break**

Step 6: Continue Expanding `i`

`i`	`arr[i]`	`map` (after update)	`map.size()`	Condition `map.size() == k`?	`ans` Update
4	3	`{3: 2, 2: 1}`	2	✗	No update

Final Output

```
5
```

Summary of Valid Subarrays

- The total number of subarrays containing all **3 distinct elements** `{1, 2, 3}` is **5**.

Output:-
0

First Non Repeating Character in C++

```
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

int sol(string s) {
    unordered_map<char, int> fmap;

    // Build frequency map
    for (char c : s) {
        fmap[c]++;
    }

    // Find first non-repeating character
    for (int i = 0; i < s.length(); i++) {
        char ch = s[i];
        if (fmap[ch] == 1) {
            return i;
        }
    }

    return -1; // If no non-repeating character found
}

int main() {
    string s = "abbcaddecfab";
    cout << sol(s) << endl;
    return 0;
}
```

Input:

s = "abbcaddecfab"

Step 1 - Build Frequency Map:

The frequency map (fmap) will look like this:

- 'a' → 2
- 'b' → 3
- 'c' → 2
- 'd' → 2
- 'e' → 2
- 'f' → 1

Step 2 - Find First Non-Repeating Character:

We now iterate through the string and check the frequency of each character:

1. For index 0: s[0] = 'a' → frequency of 'a' is 2 (repeated).
2. For index 1: s[1] = 'b' → frequency of 'b' is 3 (repeated).
3. For index 2: s[2] = 'b' → frequency of 'b' is 3 (repeated).
4. For index 3: s[3] = 'c' → frequency of 'c' is 2 (repeated).
5. For index 4: s[4] = 'a' → frequency of 'a' is 2 (repeated).
6. For index 5: s[5] = 'd' → frequency of 'd' is 2 (repeated).
7. For index 6: s[6] = 'd' → frequency of 'd' is 2 (repeated).
8. For index 7: s[7] = 'e' → frequency of 'e' is 2 (repeated).
9. For index 8: s[8] = 'c' → frequency of 'c' is 2 (repeated).
10. For index 9: s[9] = 'f' → frequency of 'f' is 1 (non-repeating).

Now, the first non-repeating character is 'f', which appears at index **7**, not index **9**.

Conclusion:

- The first non-repeating character in the string "abbcaddecfab" is 'f', which appears at **index 7**.

Output:
7

Isomorphic Strings in C++

```
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

bool iso(string s, string t) {
    if (s.length() != t.length()) {
        return false;
    }

    unordered_map<char, char> map1; // Maps characters from s to t
    unordered_map<char, bool> map2; // Tracks characters used in t

    for (int i = 0; i < s.length(); i++) {
        char ch1 = s[i];
        char ch2 = t[i];

        if (map1.count(ch1) > 0) { // If ch1 is already mapped
            if (map1[ch1] != ch2) { // Check if mapping is consistent
                return false;
            }
        } else { // ch1 has not been mapped yet
            if (map2.count(ch2) > 0) { // If ch2 is already mapped by another character in s
                return false;
            } else { // Create new mapping
                map1[ch1] = ch2;
                map2[ch2] = true;
            }
        }
    }

    return true;
}

int main() {
    string s1 = "abc";
    string s2 = "cad";
    cout << boolalpha << iso(s1, s2) << endl; // Output: true

    return 0;
}
```

Step 1: Initialize Variables

- **Input Strings**: `s = "abc"`, `t = "cad"`
- **Maps Used**:
  - `map1` → Stores mapping from `s` to `t`
  - `map2` → Tracks characters already mapped in `t`

Step 2: Iterating Through `s` and `t`

Index (`i`)	s[i]	t[i]	map1 (s → t)	map2 (used t characters)	Check for Conflict?	Result
0	`'a'`	`'c'`	`{ a → c }`	`{ c → true }`	No	Continue
1	`'b'`	`'a'`	`{ a → c, b → a }`	`{ c → true, a → true }`	No	Continue
2	`'c'`	`'d'`	`{ a → c, b → a, c → d }`	`{ c → true, a → true, d → true }`	No	Continue

Step 3: Return Result

- Since no conflicts were found, return `true`.

Final Output

`true`

Output:
true

Itinerary in C++

```
#include <iostream>
#include <unordered_map>
#include <string>

using namespace std;

int main() {
    unordered_map<string, string> map;
    map["Chennai"] = "Banglore";
    map["Bombay"] = "Delhi";
    map["Goa"] = "Chennai";
    map["Delhi"] = "Goa";

    // Create a hashmap to mark if a city is a potential
source
    unordered_map<string, bool> psrc;
    for (auto it = map.begin(); it != map.end(); ++it) {
        string src = it->first;
        string dest = it->second;

        psrc[dest] = false; // Destination city cannot be a
source
        if (psrc.find(src) == psrc.end()) {
            psrc[src] = true; // Source city if it is not a
destination in the map
        }
    }

    string src = "";
    for (auto it = psrc.begin(); it != psrc.end(); ++it) {
        if (it->second == true) {
            src = it->first;
            break;
        }
    }

    // Print the itinerary
    while (true) {
        if (map.find(src) != map.end()) {
            cout << src << " -> ";
            src = map[src];
        } else {
            cout << src << ". ";
            break;
        }
    }

    return 0;
}
```

Step 1: Initialize Data

- **Input Map (City Routes):**

```
Chennai  → Banglore
Bombay   → Delhi
Goa      → Chennai
Delhi    → Goa
```

- **Creating `psrc` (Potential Source Map)**:
  - **Initially Empty**

Step 2: Mark Potential Sources (`psrc` Construction)

Iteration	Source (`src`)	Destination (`dest`)	Updated `psrc` (Potential Source Map)
1	Chennai	Banglore	{ Banglore → false, Chennai → true }
2	Bombay	Delhi	{ Banglore → false, Chennai → true, Delhi → false, Bombay → true }
3	Goa	Chennai	{ Banglore → false, Chennai → false, Delhi → false, Bombay → true, Goa → true }
4	Delhi	Goa	{ Banglore → false, Chennai → false, Delhi → false, Bombay → true, Goa → false }

- **Final `psrc` Map:**

```
Bombay → true  (Only Source)
Banglore → false
Chennai → false
Delhi → false
Goa → false
```

Step 3: Find the Start City

- The only city with `true` in `psrc` is **"Bombay"**.
- **Start `src = "Bombay"`.**

Step 4: Print the Itinerary

Iteration	Current `src`	Next City (`map[src]`)	Printed Output
1	Bombay	Delhi	`Bombay ->`
2	Delhi	Goa	`Delhi ->`
3	Goa	Chennai	`Goa ->`
4	Chennai	Banglore	`Chennai ->`
5	Banglore	(Not Found)	`Banglore.`

Final Output

```
Bombay -> Delhi -> Goa -> Chennai -> Banglore.
```

Output:
Bombay -> Delhi -> Goa -> Chennai -> Banglore.

Largest Subarray with 0sum in C++

```
#include<bits/stdc++.h>

using namespace std;

int largest2(vector<int> arr, int n) {
    int max_len = 0;
    for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
            sum += arr[j];
            if (sum == 0) {
                max_len = max(max_len, j - i + 1);
            }
        }
    }

    return max_len;
}

int largest3(vector<int> arr, int n) {
    map<int, int> mapp;
    mapp[0]=-1;
    int sum=0;
    int ans=0;
    for (int i = 0; i < n; i++)
    {
       sum+=arr[i];
       if(mapp.find(sum)!=mapp.end()){
        auto it=mapp[sum];
        ans=max(ans,i- it);
       }
       else{
        mapp[sum]=i;
       }
    }
    return ans;
}



int largestSubarrayWithZeroSum(vector<int>& arr) {
    unordered_map<int, int> hm; // Maps sum to index
    int sum = 0;
    int max_len = 0;

    hm[0] = -1; // Initialize to handle the case where sum becomes 0 at the start

    for (int i = 0; i < arr.size(); i++) {
        sum += arr[i];

        if (hm.find(sum) != hm.end()) {
            int len = i - hm[sum];
            if (len > max_len) {
                max_len = len;
            }
        } else {
            hm[sum] = i;
```

Step 1: Understanding the Problem

- We need to find the **largest subarray with sum = 0**.
- The input array is:

  {2, 8, -3, -5, 2, -4, 6, 1, 2, 1, -3, 4}

- The program runs **three different implementations** for this:
  1. **largestSubarrayWithZeroSum()** → Optimized using unordered_map.
  2. **largest2()** → Brute-force approach.
  3. **largest3()** → Using map.

Step 2: Dry Run for largestSubarrayWithZeroSum() (Optimized Hashing Approach)

Index (i)	arr[i]	Sum	hm (Sum → Index)	Max Length (max_len)
0	2	2	{0:-1, 2:0}	0
1	8	10	{0:-1, 2:0, 10:1}	0
2	-3	7	{0:-1, 2:0, 10:1, 7:2}	0
3	-5	2	**Found 2 at index 0 → 3 - 0 = 3**	**3**
4	2	4	{0:-1, 2:0, 10:1, 7:2, 4:4}	3
5	-4	0	**Found 0 at index -1 → 5 - (-1) = 6**	**6**
6	6	6	{0:-1, 2:0, 10:1, 7:2, 4:4, 6:6}	6
7	1	7	**Found 7 at index 2 → 7 - 2 = 5**	6
8	2	9	{0:-1, 2:0, 10:1, 7:2, 4:4, 6:6, 9:8}	6
9	1	10	**Found 10 at index 1 → 9 - 1 = 8**	**8**
10	-3	7	**Found 7 at index 2 → 10 - 2 = 8**	8

```
        }
    }

    return max_len;
}

int main() {
    vector<int> arr = {2, 8, -3, -5, 2, -4, 6, 1, 2, 1, -3, 4};
    int max_length = largestSubarrayWithZeroSum(arr);
    cout << max_length << endl; // Output: 5

    int n=arr.size();
    int res=largest2(arr,n);
    cout<<res<<endl;

    int res3=largest3(arr,n);
    cout<<res3<<endl;

    return 0;
}
```

11	4	11	{0:-1, 2:0, 10:1, 7:2, 4:4, 6:6, 9:8, 11:11}	8

Final Output of **largestSubarrayWithZeroSum()** → **8**

Step 3: Dry Run for largest2() (Brute-force approach)

- **Time Complexity:** O(N²) → Iterates over all possible subarrays.
- Iterates over each possible subarray and calculates its sum.

i	j	Subarray	Sum	Max Length (max_len)
0	1	{2, 8}	10	0
0	2	{2, 8, -3}	7	0
0	3	{2, 8, -3, -5}	2	0
0	5	{2, 8, -3, -5, 2, -4}	**0**	**6**
1	5	{8, -3, -5, 2, -4}	**0**	**6**
3	9	{ -5, 2, -4, 6, 1, 2, 1 }	**0**	**7**
1	9	{ 8, -3, -5, 2, -4, 6, 1, 2, 1 }	**0**	**8**

Final Output of **largest2()** → **8**

Step 4: Dry Run for largest3() (Map-based approach)

- Similar to largestSubarrayWithZeroSum(), but uses map<int, int> instead of unordered_map<int, int>.

Index (i)	arr[i]	Sum	mapp (Sum → Index)	Max Length (ans)
0	2	2	{0:-1, 2:0}	0
1	8	10	{0:-1, 2:0, 10:1}	0
2	-3	7	{0:-1, 2:0, 10:1, 7:2}	0
3	-5	2	**Found 2 at index 0 → 3 - 0 = 3**	**3**
4	2	4	{0:-1, 2:0, 10:1, 7:2,	3

Index (i)	arr[i]	Sum	mapp (Sum → Index)	Max Length (ans)
			4:4}	
5	-4	0	**Found 0 at index -1 → 5 - (-1) = 6**	**6**
6	6	6	{0:-1, 2:0, 10:1, 7:2, 4:4, 6:6}	6
7	1	7	**Found 7 at index 2 → 7 - 2 = 5**	6
8	2	9	{0:-1, 2:0, 10:1, 7:2, 4:4, 6:6, 9:8}	6
9	1	10	**Found 10 at index 1 → 9 - 1 = 8**	**8**
10	-3	7	**Found 7 at index 2 → 10 - 2 = 8**	8
11	4	11	{0:-1, 2:0, 10:1, 7:2, 4:4, 6:6, 9:8, 11:11}	8

Final Output of largest3() → 8

Final Outputs

Function	Approach	Output
largestSubarrayWithZeroSum()	Hashing (unordered_map)	8
largest2()	Brute-force ($O(N^2)$)	8
largest3()	Hashing (map)	8

Output:
8
8
8

Largest Subarray With Contiguous Elements in C++

```
#include <iostream>
#include <unordered_set>
#include <vector>

using namespace std;

int solution(vector<int>&
arr) {
   int ans = 0;

   for (int i = 0; i < arr.size() -
1; i++) {
      int min_val = arr[i];
      int max_val = arr[i];
      unordered_set<int>
contiguous_set;

contiguous_set.insert(arr[i]);

      for (int j = i + 1; j <
arr.size(); j++) {
         if
(contiguous_set.find(arr[j]) !=
contiguous_set.end()) {
            break; // If
duplicate found, break the
loop
         }


contiguous_set.insert(arr[j]);
         min_val =
min(min_val, arr[j]);
         max_val =
max(max_val, arr[j]);

         if (max_val - min_val
== j - i) {
            int len = j - i + 1;
            if (len > ans) {
               ans = len;
            }
         }
      }
   }

   return ans;
}

int main() {
   vector<int> arr = {10, 12,
11};
   cout << solution(arr) <<
endl; // Output: 3

   return 0;
}
```

Understanding the Problem

- The function solution(arr) finds the length of the **longest contiguous subarray** where all elements are **distinct and consecutive**.
- A contiguous subarray is valid if:

  max_val - min_val = j - i

- **Example Input:** {10, 12, 11}
- **Expected Output:** 3 (as {10, 12, 11} forms a valid contiguous subarray)

Step-by-Step Dry Run

Outer Loop (i)	Inner Loop (j)	Subarray	min_val	max_val	max_val - min_val	j - i	Valid?	Current ans
0	0	{10}	10	10	0	0	✓	1
0	1	{10, 12}	10	12	2	1	✗	1
0	2	{10, 12, 11}	10	12	2	2	✓	**3**
1	1	{12}	12	12	0	0	✓	3
1	2	{12, 11}	11	12	1	1	✓	3
2	2	{11}	11	11	0	0	✓	3

Final Output: 3

Output:
3

Longest Substring With At Most K Unique Characters in C++

```cpp
#include <iostream>
#include <string>
#include <unordered_map>

class LongestSubstringWithAtMostKUniqueCharacters {
public:
    static int sol(const std::string& str, int k) {
        int ans = 0;
        int i = -1;
        int j = -1;
        std::unordered_map<char, int> map;

        while (true) {
            bool f1 = false;
            bool f2 = false;

            while (i < static_cast<int>(str.length()) - 1) {
                f1 = true;
                i++;
                char ch = str[i];
                map[ch]++;

                if (map.size() <= k) {
                    int len = i - j;
                    if (len > ans) {
                        ans = len;
                    }
                } else {
                    break;
                }
            }

            while (j < i) {
                f2 = true;
                j++;
                char ch = str[j];
                if (map[ch] == 1) {
                    map.erase(ch);
                } else {
                    map[ch]--;
                }

                if (map.size() > k) {
                    continue;
                } else {
                    int len = i - j;
                    if (len > ans) {
                        ans = len;
                    }
                    break;
                }
            }

            if (!f1 && !f2) {
                break;
            }
        }
        return ans;
    }
};
```

Understanding the Problem

- The function **`sol(str, k)`** finds the **longest substring** with at most `k` unique characters.
- Uses **two-pointer sliding window** technique (`i` and `j`) with an **unordered_map** to track character frequencies.
- Expands the window until the number of unique characters exceeds `k`, then shrinks the window.

Example Input

```
string str = "ddacbbaccdedacebb";
int k = 3;
```

Expected Output: `7`

Step-by-Step Dry Run

Step	i	j	Window (`str[j+1]` to `str[i]`)	Unique Chars	Max Length (ans)
1	0	-1	`d`	1	1
2	1	-1	`dd`	1	2
3	2	-1	`dda`	2	3
4	3	-1	`ddac`	3	4
5	4	-1	`ddacb`	4 (exceeds k)	4
6	4	0	`dacb`	3	4
7	5	0	`dacbb`	3	5
8	6	0	`dacbba`	3	6
9	7	0	`dacbbac`	3	7 ✓
10	8	0	`dacbbacc`	3	7
11	9	1	`acbbaccd`	4 (exceeds k)	7
12	9	2	`cbbaccd`	3	7
13	10	2	`cbbaccde`	4 (exceeds k)	7
14	10	3	`bbaccde`	3	7
15	11	3	`bbaccded`	4 (exceeds	7

```
int main() {
    std::string str = "ddacbbaccdedacebb";
    int k = 3;
    std::cout <<
LongestSubstringWithAtMostKUniqueCharacters::sol(str
, k) << std::endl;
    return 0;
}
```

				k)	
...	...	...	...	...	...

Final Output

✅ **Longest substring with at most `k = 3` unique characters: 7**

Output:-
7

LongestSubstringWithNonRepeatingCharacters in C++

```cpp
#include <iostream>
#include <string>
#include <unordered_map>

class LongestSubstringWithNonRepeatingCharacters {
public:
    static int solution(const std::string& str) {
        int ans = 0;
        int i = -1;
        int j = -1;

        std::unordered_map<char, int> map;
        while (true) {
            bool f1 = false;
            bool f2 = false;

            while (i < static_cast<int>(str.length()) - 1) {
                f1 = true;
                i++;
                char ch = str[i];
                map[ch]++;

                if (map[ch] == 2) {
                    break;
                } else {
                    int len = i - j;
                    if (len > ans) {
                        ans = len;
                    }
                }
            }

            while (j < i) {
                f2 = true;
                j++;
                char ch = str[j];
                map[ch]--;
                if (map[ch] == 1) {
                    break;
                }
            }

            if (!f1 && !f2) {
                break;
            }
        }

        return ans;
    }
};

int main() {
    std::string str = "aabcbcdbca";
    std::cout <<
LongestSubstringWithNonRepeatingCharacters::solution(str
) << std::endl;
    return 0;
}
```

Understanding the Problem

- The function `solution(str)` finds the **length of the longest substring with all distinct (non-repeating) characters**.
- Uses **two-pointer sliding window** (`i` and `j`) with an **unordered_map** to track character frequencies.
- Expands the window until a duplicate character is found, then contracts the window to remove duplicates.

Example Input

```
string str = "aabcbcdbca";
```

Expected Output: `4` (longest substring = `"bcdb"`)

Step-by-Step Dry Run

Step	i	j	Window (`str[j+1]` to `str[i]`)	Map	Max Length (ans)
1	0	-1	`a`	`{a:1}`	1
2	1	-1	`aa`	`{a:2}` (duplicate)	1
3	1	0	`a`	`{a:1}`	1
4	2	0	`ab`	`{a:1, b:1}`	2
5	3	0	`abc`	`{a:1, b:1, c:1}`	3
6	4	0	`abcb`	`{a:1, b:2, c:1}`	3
7	4	1	`bcb`	`{b:2, c:1}`	3
8	4	2	`cb`	`{b:1, c:1}`	3
9	5	2	`cbc`	`{b:1, c:2}`	3
10	5	3	`bc`	`{b:1, c:1}`	3
11	6	3	`bcd`	`{b:1, c:1, d:1}`	3
12	7	3	`bcdb`	`{b:2, c:1, d:1}`	**4** ✓
13	7	4	`cdb`	`{b:1,`	4

				`c:1, d:1}`	
14	8	4	`cdbc`	`{b:1, c:2, d:1}`	4
15	8	5	`dbc`	`{b:1, c:1, d:1}`	4
16	9	5	`dbca`	`{b:1, c:1, d:1, a:1}`	**4** ✓✓
17	10	6	`bca`	`{b:1, c:1, a:1}`	4

Final Output

✓ **Longest substring without repeating characters:** `4` (`"bcdb"` or `"dbca"`)

Output:-4

Pair with equal sum in C++

```
#include <iostream>
#include <unordered_set>
#include <vector>

using namespace std;

bool sol(vector<int>& arr) {
    unordered_set<int> set;

    for (int i = 0; i < arr.size(); i++) {
        for (int j = i + 1; j < arr.size(); j++) {
            int sum = arr[i] + arr[j];
            if (set.count(sum)) {
                return true;
            } else {
                set.insert(sum);
            }
        }
    }
    return false;
}

int main() {
    vector<int> arr = {2, 9, 3, 5, 8, 6, 4};
    bool ans = sol(arr);
    cout << boolalpha << ans << endl;
    return 0;
}
```

Input

```
arr = {2, 9, 3, 5, 8, 6, 4}
```

Dry Run Table

i	j	arr[i]	arr[j]	sum	Seen Sums Before	Is sum already in set?	Action
0	1	2	9	11	{}	No	Insert 11
0	2	2	3	5	{11}	No	Insert 5
0	3	2	5	7	{11, 5}	No	Insert 7
0	4	2	8	10	{11, 5, 7}	No	Insert 10
0	5	2	6	8	{11, 5, 7, 10}	No	Insert 8
0	6	2	4	6	{5, 7, 8, 10, 11}	No	Insert 6
1	2	9	3	12	{5, 6, 7, 8, 10, 11}	No	Insert 12
1	3	9	5	14	...	No	Insert 14
1	4	9	8	17	...	No	Insert 17
1	5	9	6	15	...	No	Insert 15
1	6	9	4	13	...	No	Insert 13
2	3	3	5	8	Already seen	✅ **Yes** → Return `true`	

✅ Output

```
true
```

Output:-
true

Subarray sum equals k in C++

```
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;
class SubarraySumEqualsK {
public:
    static int sol(const std::vector<int>& arr,
int target) {
        int ans = 0;
        std::unordered_map<int, int> map;
        map[0] = 1;
        int sum = 0;

        for (int i = 0; i < arr.size(); i++) {
            sum += arr[i];
            int rsum = sum - target;
            if (map.find(rsum) != map.end()) {
                ans += map[rsum];
            }
            map[sum]++;
        }
        return ans;
    }
};

int main() {
    vector<int> arr = {3, 9, -2, 4, 1, -7, 2, 6,
-5, 8, -3, -7, 6, 2, 1};
    int k = 5;
    cout << SubarraySumEqualsK::sol(arr,
k) << std::endl;
    return 0;
}
```

Example Input

```
vector<int> arr = {3, 9, -2, 4, 1, -7, 2, 6,
-5, 8, -3, -7, 6, 2, 1};
int k = 5;
```

Expected Output: 5

Step-by-Step Dry Run

Step	i	arr[i]	sum (Prefix Sum)	rsum = sum - k	map[rsum] (if exists)	ans (count of subarrays)	map[sum] (updated)
1	0	3	3	-2	0	0	{0:1, 3:1}
2	1	9	12	7	0	0	{0:1, 3:1, 12:1}
3	2	-2	10	5	0	0	{0:1, 3:1, 12:1, 10:1}
4	3	4	14	9	0	0	{0:1, 3:1, 12:1, 10:1, 14:1}
5	4	1	15	10	✓1	1	{0:1, 3:1, 12:1, 10:1, 14:1, 15:1}
6	5	-7	8	3	✓1	2	{0:1, 3:1, 12:1, 10:1, 14:1, 15:1, 8:1}
7	6	2	10	5	0	2	{0:1, 3:1, 12:1, 10:2, 14:1, 15:1, 8:1}
8	7	6	16	11	0	2	{0:1,

							3:1, 12:1, 10:2, 14:1, 15:1, 8:1, 16:1}
9	8	-5	11	6	0	2	{0:1, 3:1, 12:1, 10:2, 14:1, 15:1, 8:1, 16:1, 11:1}
10	9	8	19	14	✓1	3	{0:1, 3:1, 12:1, 10:2, 14:1, 15:1, 8:1, 16:1, 11:1, 19:1}
11	10	-3	16	11	✓1	4	{0:1, 3:1, 12:1, 10:2, 14:1, 15:1, 8:1, 16:2, 11:1, 19:1}
12	11	-7	9	4	0	4	{0:1, 3:1, 12:1, 10:2, 14:1, 15:1, 8:1, 16:2, 11:1, 19:1, 9:1}
13	12	6	15	10	✓2	6	{0:1, 3:1, 12:1, 10:2, 14:1, 15:2, 8:1, 16:2, 11:1, 19:1,

							9:1}
14	13	2	17	12	✓1	7	{0:1, 3:1, 12:1, 10:2, 14:1, 15:2, 8:1, 16:2, 11:1, 19:1, 9:1, 17:1}
15	14	1	18	13	0	7	{0:1, 3:1, 12:1, 10:2, 14:1, 15:2, 8:1, 16:2, 11:1, 19:1, 9:1, 17:1, 18:1}

Final Output

✓ **Output:** 7

Output:-
7

Two Sum in C++

```
#include <iostream>
#include <unordered_map>
#include <vector>

using namespace std;

vector<int> twoSum(vector<int>& nums, int target)
{
   unordered_map<int, int> map; // Hash map to
store number and its index
   vector<int> result;

   for (int i = 0; i < nums.size(); i++) {
      int complement = target - nums[i];

      if (map.find(complement) != map.end()) {
         result.push_back(map[complement]);
         result.push_back(i);
         return result;
      }

      map[nums[i]] = i;
   }

   throw invalid_argument("No two sum solution");
}

int main() {
   vector<int> nums1 = {2, 7, 11, 15};
   int target1 = 9;

   vector<int> nums2 = {3, 2, 4};
   int target2 = 6;

   vector<int> result1 = twoSum(nums1, target1);
   vector<int> result2 = twoSum(nums2, target2);

   cout << "Output for nums1: [" << result1[0] << ", "
<< result1[1] << "]" << endl;
   cout << "Output for nums2: [" << result2[0] << ", "
<< result2[1] << "]" << endl;

   return 0;
}
```

Test Case 1

```
vector<int> nums1 = {2, 7, 11, 15};
int target1 = 9;
```

- We need to find two indices `i, j` such that `nums1[i] + nums1[j] = 9`.

Step	i	nums1[i]	Complement (target - nums1[i])	map (stored indices)	Match Found?
1	0	2	7	`{2:0}`	✗ No
2	1	7	2	`{2:0, 7:1}`	✓ Yes (`2` found at index `0`)

✓ **Output:** `[0, 1]` (because `nums1[0] + nums1[1] = 2 + 7 = 9`)

Test Case 2

```
vector<int> nums2 = {3, 2, 4};
int target2 = 6;
```

Step	i	nums2[i]	Complement (target - nums2[i])	map (stored indices)	Match Found?
1	0	3	3	`{3:0}`	✗ No
2	1	2	4	`{3:0, 2:1}`	✗ No
3	2	4	2	`{3:0, 2:1, 4:2}`	✓ Yes (`2` found at index `1`)

✓ **Output:** `[1, 2]` (because `nums2[1] + nums2[2] = 2 + 4 = 6`)

Output:-
Output for nums1: [0, 1]
Output for nums2: [1, 2]

Valid Anagram in C++

```
#include <iostream>
#include <string>
#include <unordered_map>

class ValidAnagrams {
public:
    static bool sol(const std::string& s1, const
std::string& s2) {
        std::unordered_map<char, int> map;
        for (char ch : s1) {
            map[ch]++;
        }

        for (char ch : s2) {
            if (map.find(ch) == map.end()) {
                return false;
            } else if (map[ch] == 1) {
                map.erase(ch);
            } else {
                map[ch]--;
            }
        }
        return map.empty();
    }
};

int main() {
    std::string s1 = "abbcaad";
    std::string s2 = "babacda";
    std::cout << (ValidAnagrams::sol(s1, s2) ? "true" :
"false") << std::endl;
    return 0;
}
```

Dry Run Table for `ValidAnagrams::sol(s1, s2)`

Input:

```
s1 = "abbcaad";
s2 = "babacda";
```

Step 1: Build Character Frequency Map (`s1`)

Iteration	Character (`ch`)	`map[ch]` (Updated)	`map` State
0	'a'	1	`{ 'a': 1 }`
1	'b'	1	`{ 'a': 1, 'b': 1 }`
2	'b'	2	`{ 'a': 1, 'b': 2 }`
3	'c'	1	`{ 'a': 1, 'b': 2, 'c': 1 }`
4	'a'	2	`{ 'a': 2, 'b': 2, 'c': 1 }`
5	'a'	3	`{ 'a': 3, 'b': 2, 'c': 1 }`
6	'd'	1	`{ 'a': 3, 'b': 2, 'c': 1, 'd': 1 }`

Final **map** after processing `s1`:

```
{ 'a': 3, 'b': 2, 'c': 1, 'd': 1 }
```

Step 2: Validate Using `s2`

Iteration	Character (`ch`)	Action	`map[ch]` (Updated)	`map` State
0	'b'	Decrement	1	`{ 'a': 3, 'b': 1, 'c': 1, 'd': 1 }`
1	'a'	Decrement	2	`{ 'a': 2, 'b': 1, 'c': 1, 'd': 1 }`

Iteration	Character (`ch`)	Action	`map[ch]` (Updated)	`map` State
2	'b'	Remove from `map`	—	`{ 'a': 2, 'c': 1, 'd': 1 }`
3	'a'	Decrement	1	`{ 'a': 1, 'c': 1, 'd': 1 }`
4	'c'	Remove from `map`	—	`{ 'a': 1, 'd': 1 }`
5	'd'	Remove from `map`	—	`{ 'a': 1 }`
6	'a'	Remove from `map`	—	`{}`

Final **map** state: **Empty `{}`**, meaning both strings are anagrams.

✅ **Output:** `"true"`

Output:-
true