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}

Step-by-Step Execution:

1. **Find Minimum and Maximum**:
   - minVal=1
   - maxVal=37
2. **Calculate Common Difference**:

   d=(maxVal−minVal)/(size−1)=37−1/10−1=4
   Check Arithmetic Sequence:

   - Construct sequence using minVal+i·d
   - {1,5,9,13,17,21,25,29,33,37}
   - All values exist in the hash set: 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;
}
```

Step 1: Calculate Remainders

Dry Run

Input:

- Array: {22, 12, 45, 55, 65, 78, 88, 75}
- Divisor (k): 7

Step 1: Calculate Remainders
For each element in the array, calculate the remainder rem = val % k:

Element (val)	Remainder (val % k)
22	22 % 7 = 1
12	12 % 7 = 5
45	45 % 7 = 3
55	55 % 7 = 6
65	65 % 7 = 2
78	78 % 7 = 1
88	88 % 7 = 4
75	75 % 7 = 5

Remainder Frequency Map:
{1: 2, 5: 2, 3: 1, 6: 1, 2: 1, 4: 1}

Step 2: Validate Pairing Conditions
Iterate through the array and validate the conditions for pairing:

1. **For rem = 1:**
    - Frequency of 1: freq[1] = 2
    - Frequency of k - 1 (6): freq[6] = 1
    - Mismatch found: freq[1] ≠ freq[6].
    - **Condition failed.**

Since the pairing condition fails for rem = 1, we conclude that the array cannot be divided into valid pairs.

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}`

Character (`ch`)	Action Taken	Updated `map`
`'p'`	Found in `map`, decrement `map['p']`	`{'p': 1, 'e': 1}`
`'e'`	Found in `map`, decrement `map['e']`	`{'p': 1}`
`'p'`	Found in `map`, decrement `map['p']`	`{}`

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}

Explanation of Each Iteration:

- **At i = 0**:
  - arr[0] = 0
  - Treat 0 as -1.
  - sum = -1.
  - map[sum] = map[-1] = 0, so we add 0 to ans.
  - We update map[-1] = 1.
- **At i = 1**:
  - arr[1] = 0
  - Treat 0 as -1.
  - sum = -2.
  - map[sum] = map[-2] = 0, so we add 0 to ans.

	○ We update map[-2] = 1. • **At i = 2**: ○ arr[2] = 1 ○ sum = -1. ○ map[sum] = map[-1] = 1, so we add 1 to ans. ○ We update map[-1] = 2. • **At i = 3**: ○ arr[3] = 0 ○ Treat 0 as -1. ○ sum = -2. ○ map[sum] = map[-2] = 1, so we add 1 to ans. ○ We update map[-2] = 2. • **At i = 4**: ○ arr[4] = 1 ○ sum = -1. ○ map[sum] = map[-1] = 2, so we add 2 to ans. ○ We update map[-1] = 3. • **At i = 5**: ○ arr[5] = 0 ○ Treat 0 as -1. ○ sum = -2. ○ map[sum] = map[-2] = 2, so we add 2 to ans. ○ We update map[-2] = 3. • **At i = 6**: ○ arr[6] = 1 ○ sum = -1. ○ map[sum] = map[-1] = 3, so we add 3 to ans. ○ We update map[-1] = 4. • **At i = 7**: ○ arr[7] = 1 ○ sum = 0. ○ map[sum] = map[0] = 2, so we add 2 to ans. ○ We update map[0] = 3. • **At i = 8**: ○ arr[8] = 0 ○ Treat 0 as -1. ○ sum = -1. ○ map[sum] = map[-1] = 4, so we add 4 to ans. ○ We update map[-1] = 5. • **At i = 9**: ○ arr[9] = 0 ○ Treat 0 as -1. ○ sum = -2. ○ map[sum] = map[-2] = 3, so we add 3 to ans. ○ We update map[-2] = 4. • **At i = 10**: ○ arr[10] = 1 ○ sum = -1. ○ map[sum] = map[-1] = 5, so we add 5 to ans. ○ We update map[-1] = 6.

	• **At i = 11**:
	  ○ arr[11] = 1
	  ○ sum = 0.
	  ○ map[sum] = map[0] = 3, so we add 3 to ans.
	  ○ We update map[0] = 4.
	• **At i = 12**:
	  ○ arr[12] = 1
	  ○ sum = 1.
	  ○ map[sum] = map[1] = 0, so we add 0 to ans.
	  ○ We update map[1] = 1.
	
	Final Result:
	
	• The total count of subarrays whose sum is 0 is **24**.

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";

    findCount(map);

    return 0;
```

Step-by-Step Walkthrough of the Example:

Input:

```
unordered_map<string, string> map;
map["A"] = "C";
map["B"] = "C";
map["C"] = "F";
map["D"] = "E";
map["E"] = "F";
map["F"] = "F";
```

- A, B, and C report to C
- D reports to E, and E reports to F
- C reports to F and F reports to F (CEO)

Building the Tree:

- **Manager-Employee Tree Structure:**

  F: {C, E} (F manages C and E)
  E: {D} (E manages D)
  C: {A, B} (C manages A and B)
  D: {} (D has no subordinates)
  A: {} (A has no subordinates)
  B: {} (B has no subordinates)

- **CEO (F)** has employees C and E.
- **E** manages D.
- **C** manages A and B.

Dry Run:

Call getSize(tree, "F", result):

- Starting with **F**:
  - F has two direct subordinates: **C** and **E**.
  - Recursively calculate the size for **C** and **E**:
    - **C** has two direct subordinates: **A** and **B**.
      - Both **A** and **B** have no subordinates (base case).
      - Size of **C** = 2 (A and B).
    - **E** has one direct subordinate: **D**.
      - **D** has no subordinates (base case).
      - Size of **E** = 1 (D).
  - Total size of **F** = Size of C (2) + Size of E (1) = 5.

Final Sizes:

- **F**: 5 (Subordinates: C, E, A, B, D)

}	• **E**: 1 (Subordinate: D) • **C**: 2 (Subordinates: A, B) • **A**: 0 (No subordinates) • **B**: 0 (No subordinates) • **D**: 0 (No subordinates) **Output:** F 5 E 1 B 0 A 0 D 0 C 2
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;
            } else {
                break;
            }
```

Input:

arr = {2, 1, 3, 2, 3}

We are looking for subarrays with **all unique elements** in the array.

Initial Setup:

- arr = {2, 1, 3, 2, 3}
- We initialize an unordered_set called set to store the unique elements of the array.
- We calculate k = set.size(), which is the number of unique elements in the array.
    - set = {2, 1, 3} -> k = 3 (3 unique elements).

Algorithm Steps:

1. **Initialization:**
    - i = -1, j = -1 (start indices for the sliding window).
    - map = {} (tracks the frequency of elements in the current window).
    - ans = 0 (tracks the number of valid subarrays).
2. **Start the outer while (true) loop:**
    - The outer loop keeps expanding and shrinking the window until we can no longer process subarrays.

First pass through the outer loop:

- **Expand the window (moving i):**
    - Initially i = -1, we move i to 0 (i.e., arr[i] = 2).
    - Add arr[i] = 2 to the map:
        - map = {2: 1}
    - Now, i = 0, and map.size() = 1 (we have only one unique element, so we move on).
    - Move i to 1 (i.e., arr[i] = 1).
    - Add arr[i] = 1 to the map:
        - map = {2: 1, 1: 1}
    - Now, i = 1, and map.size() = 2 (we still have not covered all unique elements, so continue expanding).
    - Move i to 2 (i.e., arr[i] = 3).
    - Add arr[i] = 3 to the map:
        - map = {2: 1, 1: 1, 3: 1}
    - Now, i = 2, and map.size() = 3 (we have all 3 unique elements).
    - At this point, we have a valid subarray from arr[0] to arr[2]. All unique elements are covered.
    - **Count valid subarrays:**
        - Subarrays ending at i = 2:
            - The number of subarrays is

```
        }

        // 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;
}
```

calculated as arr.size() - i = 5 - 2 = 3.
- ans += 3 → ans = 3.

- **Shrink the window (moving j):**
  - Now we start shrinking the window by moving the left pointer (j).
  - Move j to 0 (i.e., arr[j] = 2):
    - Since map[arr[j]] = 1, we remove arr[j] from the map:
      - map = {1: 1, 3: 1}
  - Move j to 1 (i.e., arr[j] = 1):
    - Since map[arr[j]] = 1, we remove arr[j] from the map:
      - map = {3: 1}
  - Now, map.size() = 1, which is less than k. We stop shrinking.

Second pass through the outer loop:

- Now, we repeat the steps again by expanding and shrinking the window, but the map.size() will no longer match k (the number of unique elements).

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;
}
```

Dry Run:

Input:

s1 = "abc"
s2 = "cad"

Step 1 - Check Length:

- First, we check if the lengths of s1 and s2 are the same. Both are of length 3, so we proceed.

Step 2 - Initialize Maps:

- map1 (for mapping characters of s1 to s2) is an empty map initially.
- map2 (to track characters already used in s2) is also an empty map initially.

Step 3 - Iterate Over the Strings:

Now we iterate over each character in s1 and s2 simultaneously:

1. **First iteration** (i = 0):
    - ch1 = s1[0] = 'a' and ch2 = s2[0] = 'c'
    - 'a' is not mapped yet, and 'c' is not used yet in map2.
    - So, we create a mapping 'a' -> 'c' in map1 and mark 'c' as used in map2.
2. **Second iteration** (i = 1):
    - ch1 = s1[1] = 'b' and ch2 = s2[1] = 'a'
    - 'b' is not mapped yet, and 'a' is not used yet in map2.
    - So, we create a mapping 'b' -> 'a' in map1 and mark 'a' as used in map2.
3. **Third iteration** (i = 2):
    - ch1 = s1[2] = 'c' and ch2 = s2[2] = 'd'
    - 'c' is not mapped yet, and 'd' is not used yet in map2.
    - So, we create a mapping 'c' -> 'd' in map1 and mark 'd' as used in map2.

Step 4 - Check Mapping Consistency:

After the loop ends, the mappings in map1 are:

map1 = { 'a' -> 'c', 'b' -> 'a', 'c' -> 'd' }

Since all characters in s1 have been mapped to distinct characters in s2, and no character in s2 has been mapped by more than one character from

	s1, the strings are isomorphic. **Final Output:** Since the mappings are valid and consistent, the function returns 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;
}
```

Dry Run Example:

Input Data:

```
unordered_map<string, string> map;
map["Chennai"] = "Banglore";
map["Bombay"] = "Delhi";
map["Goa"] = "Chennai";
map["Delhi"] = "Goa";
```

1. **psrc Mapping**:
    - Initially, all cities are marked as potential sources (true).
    - Iterating over map:
        - "Chennai" is a source (because it's not in the destination list).
        - "Bombay" is a source.
        - "Goa" is a destination, so it's marked as false.
        - "Delhi" is a destination, so it's marked as false.
    - Final psrc will be:

      cpp
      Copy code
      psrc = { "Bombay" = true, "Delhi" = false, "Goa" = false, "Chennai" = false }

2. **Finding the Source City**:
    - The first city with true in psrc is "Bombay".
    - Set src = "Bombay".
3. **Building the Itinerary**:
    - Starting from "Bombay":
        - "Bombay" -> "Delhi"
        - "Delhi" -> "Goa"
        - "Goa" -> "Chennai"
        - "Chennai" -> "Banglore"
    - Print "Bombay -> Delhi -> Goa -> Chennai -> Banglore."

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;
        }
    }
```

Dry Run:

Input:

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

Brute Force Approach (largest2):

- The outer loop starts from i = 0 and the inner loop starts from j = i to calculate the sum of subarrays.
- It checks if the sum becomes zero and keeps track of the maximum length of subarrays where the sum is zero.

For example:

- i = 0 to j = 5, sum = 0, length = 6, so max_len = 6.
- i = 1 to j = 7, sum = 0, length = 7, so max_len = 7.
- Continue the same till the end.

Optimized Approach (largest3):

- The map stores the cumulative sum at each index.
- It checks if the cumulative sum has been encountered before. If yes, then the subarray sum between those two indices is zero.

For example:

- At i = 0, cumulative sum = 2, map stores 2: 0.
- At i = 1, cumulative sum = 10, map stores 10: 1.
- At i = 2, cumulative sum = 7, map stores 7: 2.
- At i = 3, cumulative sum = 2, found 2 at index 0, so subarray length = 3.

Final Approach (largestSubarrayWithZeroSum):

- The logic here is very similar to the optimized approach. It uses the unordered map for efficiency. The result is calculated as the maximum length of subarrays with zero sum.

Output:

- For each method, the result is calculated as follows:
    - **Brute Force (largest2)**: 8
    - **Optimized Approach (largest3)**: 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;
}
```

- **Final Approach (largestSubarrayWithZeroSum)**: 8

Final Output:

8
8
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;
}
```

Dry Run:

Input:

arr = {10, 12, 11}

Step 1 - Iterate Over the Array:

- We start by iterating over the array with two nested loops.
- The outer loop runs from i = 0 to i = n - 2 (where n is the size of the array).
- For each value of i, we initialize min_val and max_val with the value of arr[i] and set up a unordered_set to keep track of the distinct elements in the current contiguous subarray.

Step 2 - Inner Loop Iterations:

The inner loop runs from j = i + 1 to j = n - 1. In each inner loop iteration:

- We check if the current element arr[j] already exists in the set contiguous_set. If it does, we break out of the loop (this handles duplicates).
- We update min_val and max_val with the current element.
- We check if the condition max_val - min_val == j - i holds. If it does, we calculate the length of the subarray as j - i + 1. If the length is greater than the previous maximum length (ans), we update ans.

Detailed Execution:

First Outer Loop Iteration (i = 0):

- Initialize: min_val = arr[0] = 10, max_val = arr[0] = 10, contiguous_set = {10}.

Inner Loop Iterations for i = 0:

1. **First Inner Loop (j = 1)**:
    - arr[1] = 12
    - Add 12 to contiguous_set, update min_val = 10, max_val = 12.
    - max_val - min_val = 12 - 10 = 2, j - i = 1, so the condition max_val - min_val == j - i holds.
    - Subarray length = 1 - 0 + 1 = 2.
    - ans is updated to 2.
2. **Second Inner Loop (j = 2)**:
    - arr[2] = 11
    - Add 11 to contiguous_set, update min_val = 10, max_val = 12.
    - max_val - min_val = 12 - 10 = 2, j -

	i = 2, so the condition max_val - min_val == j - i holds. o Subarray length = 2 - 0 + 1 = 3. o ans is updated to 3. **Second Outer Loop Iteration (i = 1):** • Initialize: min_val = arr[1] = 12, max_val = arr[1] = 12, contiguous_set = {12}. **Inner Loop Iterations for i = 1:** 1. **First Inner Loop (j = 2)**: o arr[2] = 11 o Add 11 to contiguous_set, update min_val = 11, max_val = 12. o max_val - min_val = 12 - 11 = 1, j - i = 1, so the condition max_val - min_val == j - i holds. o Subarray length = 2 - 1 + 1 = 2. o ans remains 3. **Step 3 - Final Output:** • The longest valid subarray has a length of 3, so the function returns 3.
Output: 3	

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

```
#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;
    }
};
```

Step-by-Step Dry Run:

Initial state:

- str = "ddacbbaccdedacebb"
- k = 3
- ans = 0
- i = -1, j = -1
- map = {} (empty initially)

First iteration (expanding and contracting window):

Expand the window (while (i < str.length() - 1)):

1. i = 0, character is d, map = {d: 1} (1 unique character)
2. i = 1, character is d, map = {d: 2} (still 1 unique character)
3. i = 2, character is a, map = {d: 2, a: 1} (2 unique characters)
4. i = 3, character is c, map = {d: 2, a: 1, c: 1} (3 unique characters)
    - Window dacc has exactly 3 unique characters, so we calculate the length of this substring:
    - len = i - j = 3 - (-1) = 4. We update ans = 4.

At this point, the window size is valid (3 unique characters). Now we expand the window further.

5. i = 4, character is b, map = {d: 2, a: 1, c: 1, b: 1} (4 unique characters)
    - Since the number of unique characters exceeds k, we need to shrink the window from the left.

Shrink the window (while (j < i)):

1. j = 0, character is d, map = {d: 1, a: 1, c: 1, b: 1} (still 4 unique characters)
2. j = 1, character is d, map = {a: 1, c: 1, b: 1, d: 0} (map value for d becomes 0, we erase d)
    - Now, map = {a: 1, c: 1, b: 1} (3 unique characters)
    - The window acbb has 3 unique characters, and its length is i - j = 4 - 1 = 3. ans remains 4.

Now, we keep expanding again.

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

Second pass of the window expansion:

1. i = 5, character is b, map = {a: 1, c: 1, b: 2} (3 unique characters)
2. i = 6, character is a, map = {a: 2, c: 1, b: 2} (3 unique characters)
3. i = 7, character is c, map = {a: 2, c: 2, b: 2} (3 unique characters)
    - Now we have a valid window of acb. Its length is i - j = 7 - 1 = 7, so we update ans = 7.

Output:-
7

LongestSubstringWithNonRepeatingCharacters in C++

```
#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;
}
```

Step-by-Step Dry Run:

Initial state:

- str = "aabcbcdbca"
- ans = 0
- i = -1, j = -1
- map = {}

First pass:

1. **Expand window (while (i < str.length() - 1)):**
    - i = 0, character is a, map = {a: 1}
    - i = 1, character is a, map = {a: 2}
    - Since map[a] == 2, break the loop.
2. **Shrink window (while (j < i)):**
    - j = 0, character is a, map = {a: 1}
    - Now, map[a] == 1, break the loop.

At this point:

- ans = 1 because we found the substring "a" (length 1).

Second pass:

1. **Expand window (while (i < str.length() - 1)):**
    - i = 2, character is b, map = {a: 1, b: 1}
    - i = 3, character is c, map = {a: 1, b: 1, c: 1}
    - i = 4, character is b, map = {a: 1, b: 2, c: 1}
    - Since map[b] == 2, break the loop.
2. **Shrink window (while (j < i)):**
    - j = 1, character is a, map = {a: 0, b: 2, c: 1}
    - j = 2, character is b, map = {b: 1, c: 1}
    - map.size() = 2 so continue shrinking.
    - j = 3, character is c, map = {b: 1, c: 0}
    - Now map.size() = 1 and j = 3, break the loop.

At this point:

	• ans = 3 because the substring "abc" (length 3) was found. **Third pass:** 1. **Expand window (while (i < str.length() - 1)):** o i = 4, character is d, map = {b: 1, c: 1, d: 1} o i = 5, character is c, map = {b: 1, c: 2, d: 1} o Since map[c] == 2, break the loop. 2. **Shrink window (while (j < i)):** o j = 4, character is b, map = {b: 0, c: 2, d: 1} o j = 5, character is c, map = {c: 1, d: 1} o map.size() = 2 so continue shrinking. o j = 6, character is d, map = {d: 0, c: 1} o Now map.size() = 1 and j = 6, break the loop. At this point: • ans = 3 because the substring "bcd" (length 3) was found. **Fourth pass:** 1. **Expand window (while (i < str.length() - 1)):** o i = 7, character is b, map = {d: 0, c: 1, b: 1} o i = 8, character is c, map = {d: 0, c: 2, b: 1} o Since map[c] == 2, break the loop. 2. **Shrink window (while (j < i)):** o j = 6, character is d, map = {d: 0, c: 1, b: 1} o Now map.size() = 3 and we have found the largest substring "bcd" (length 3). At this point: • The function finishes and ans = 4.
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;
}
```

Dry Run:

Input:

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

1. **Initialization**:
    - set = {} (an empty unordered set)
    - Start iterating over the array.
2. **Iteration through the array**:
    - **For i = 0 (arr[0] = 2)**:
        - **For j = 1 (arr[1] = 9)**, sum = 2 + 9 = 11. Insert 11 into the set.
        - **For j = 2 (arr[2] = 3)**, sum = 2 + 3 = 5. Insert 5 into the set.
        - **For j = 3 (arr[3] = 5)**, sum = 2 + 5 = 7. Insert 7 into the set.
        - **For j = 4 (arr[4] = 8)**, sum = 2 + 8 = 10. Insert 10 into the set.
        - **For j = 5 (arr[5] = 6)**, sum = 2 + 6 = 8. Insert 8 into the set.
        - **For j = 6 (arr[6] = 4)**, sum = 2 + 4 = 6. Insert 6 into the set.
    - **For i = 1 (arr[1] = 9)**:
        - **For j = 2 (arr[2] = 3)**, sum = 9 + 3 = 12. Insert 12 into the set.
        - **For j = 3 (arr[3] = 5)**, sum = 9 + 5 = 14. Insert 14 into the set.
        - **For j = 4 (arr[4] = 8)**, sum = 9 + 8 = 17. Insert 17 into the set.
        - **For j = 5 (arr[5] = 6)**, sum = 9 + 6 = 15. Insert 15 into the set.
        - **For j = 6 (arr[6] = 4)**, sum = 9 + 4 = 13. Insert 13 into the set.
    - **For i = 2 (arr[2] = 3)**:
        - **For j = 3 (arr[3] = 5)**, sum = 3 + 5 = 8. 8 is already in the set, so return true.

Output:

Since a sum of 8 was found twice, the program outputs

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;
}
```

Dry Run:

Input:

arr = {3, 9, -2, 4, 1, -7, 2, 6, -5, 8, -3, -7, 6, 2, 1}
k = 5

1. **Initialize**:
    - ans = 0
    - map = {0: 1} (We initialize with map[0] = 1 to handle the case where the subarray sum itself equals k).
    - sum = 0
2. **Iteration 1**: i = 0, arr[0] = 3
    - sum = 0 + 3 = 3
    - rsum = 3 - 5 = -2
    - map doesn't have -2, so ans remains 0.
    - map[sum]++: map[3] = 1
3. **Iteration 2**: i = 1, arr[1] = 9
    - sum = 3 + 9 = 12
    - rsum = 12 - 5 = 7
    - map doesn't have 7, so ans remains 0.
    - map[sum]++: map[12] = 1
4. **Iteration 3**: i = 2, arr[2] = -2
    - sum = 12 - 2 = 10
    - rsum = 10 - 5 = 5
    - map doesn't have 5, so ans remains 0.
    - map[sum]++: map[10] = 1
5. **Iteration 4**: i = 3, arr[3] = 4
    - sum = 10 + 4 = 14
    - rsum = 14 - 5 = 9
    - map doesn't have 9, so ans remains 0.
    - map[sum]++: map[14] = 1
6. **Iteration 5**: i = 4, arr[4] = 1
    - sum = 14 + 1 = 15
    - rsum = 15 - 5 = 10
    - map has 10 with count 1, so ans += 1.
    - ans = 1
    - map[sum]++: map[15] = 1
7. **Iteration 6**: i = 5, arr[5] = -7
    - sum = 15 - 7 = 8
    - rsum = 8 - 5 = 3
    - map has 3 with count 1, so ans += 1.
    - ans = 2
    - map[sum]++: map[8] = 1
8. **Iteration 7**: i = 6, arr[6] = 2
    - sum = 8 + 2 = 10
    - rsum = 10 - 5 = 5
    - map has 5 with count 1, so ans += 1.
    - ans = 3

	○ map[sum]++: map[10] = 2 9. **Iteration 8**: i = 7, arr[7] = 6 ○ sum = 10 + 6 = 16 ○ rsum = 16 - 5 = 11 ○ map doesn't have 11, so ans remains 3. ○ map[sum]++: map[16] = 1 10. **Iteration 9**: i = 8, arr[8] = -5 ○ sum = 16 - 5 = 11 ○ rsum = 11 - 5 = 6 ○ map doesn't have 6, so ans remains 3. ○ map[sum]++: map[11] = 1 11. **Iteration 10**: i = 9, arr[9] = 8 ○ sum = 11 + 8 = 19 ○ rsum = 19 - 5 = 14 ○ map has 14 with count 1, so ans += 1. ○ ans = 4 ○ map[sum]++: map[19] = 1 12. **Iteration 11**: i = 10, arr[10] = -3 ○ sum = 19 - 3 = 16 ○ rsum = 16 - 5 = 11 ○ map has 11 with count 1, so ans += 1. ○ ans = 5 ○ map[sum]++: map[16] = 2 13. **Iteration 12**: i = 11, arr[11] = -7 ○ sum = 16 - 7 = 9 ○ rsum = 9 - 5 = 4 ○ map doesn't have 4, so ans remains 5. ○ map[sum]++: map[9] = 1 14. **Iteration 13**: i = 12, arr[12] = 6 ○ sum = 9 + 6 = 15 ○ rsum = 15 - 5 = 10 ○ map has 10 with count 2, so ans += 2. ○ ans = 7 ○ map[sum]++: map[15] = 2 15. **Iteration 14**: i = 13, arr[13] = 2 ○ sum = 15 + 2 = 17 ○ rsum = 17 - 5 = 12 ○ map doesn't have 12, so ans remains 7. ○ map[sum]++: map[17] = 1 16. **Iteration 15**: i = 14, arr[14] = 1 ○ sum = 17 + 1 = 18 ○ rsum = 18 - 5 = 13 ○ map doesn't have 13, so ans remains 7. ○ map[sum]++: map[18] = 1 **Final Answer:** After processing all the elements, the number of subarrays whose sum equals 5 is **7**.

	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;
}
```

Explanation of Code:

The given code is solving the classic "Two Sum" problem. The task is to find two indices in an array where the values at those indices add up to a specific target sum.

Step-by-Step Breakdown:

1. **Input**:
   - nums1 = {2, 7, 11, 15}, target1 = 9
   - nums2 = {3, 2, 4}, target2 = 6
2. **Core Logic**:
   - **twoSum function**:
     - A hash map (unordered_map) is used to store each element in the array and its corresponding index.
     - The idea is to check, for each element nums[i], whether its complement (target - nums[i]) already exists in the map. If it does, we've found the solution.
     - If not, we store the element nums[i] along with its index in the map for future reference.
3. **Detailed Steps**:
   - For nums1 = {2, 7, 11, 15} and target1 = 9:
     1. We start iterating through the array.
     2. For i = 0 (value 2), we calculate the complement: 9 - 2 = 7. The map is empty, so we store 2 in the map with its index: map = {2: 0}.
     3. For i = 1 (value 7), we calculate the complement: 9 - 7 = 2. Since 2 is already in the map (at index 0), we've found the solution: indices [0, 1].
     4. The result [0, 1] is returned.
   - For nums2 = {3, 2, 4} and target2 = 6:
     1. We start iterating through the array.
     2. For i = 0 (value 3), we calculate the complement: 6 - 3 = 3. The map is empty, so we store 3 in the map with its index: map = {3: 0}.
     3. For i = 1 (value 2), we calculate the complement: 6 - 2 = 4. Since 4 is not in the

	map, we store 2 with its index: map = {3: 0, 2: 1}. 4. For i = 2 (value 4), we calculate the complement: 6 - 4 = 2. Since 2 is already in the map (at index 1), we've found the solution: indices [1, 2]. 5. The result [1, 2] is returned. **Output:** Output for nums1: [0, 1] Output for nums2: [1, 2]
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;
}
```

Step-by-Step Breakdown:

1. **Input**:
   - `s1 = "abbcaad"`
   - `s2 = "babacda"`
2. **Core Logic**:
   - **`ValidAnagrams::sol` function**:
     - We create an **unordered_map** (`map`) to store the frequency of each character in `s1`.
     - Then, we iterate over the characters in `s2` and check if each character in `s2` is found in `map` (i.e., it should also exist in `s1` with the correct frequency).
     - If a character is found in `map`, we decrease its count. If the count reaches `1`, we remove that character from `map` entirely.
     - If `map` is empty at the end, it means that both strings are anagrams because they contain the same characters with the same frequencies.
     - If `map` is not empty at the end, it means the strings are not anagrams.
3. **Detailed Steps**:
   - **Input strings**: `s1 = "abbcaad"`, `s2 = "babacda"`
   - **First, we populate the frequency map** using `s1`:
     - For `s1`, the map will look like this:
       - `{'a': 3, 'b': 2, 'c': 1, 'd': 1}`
   - **Then, we iterate over the characters in `s2`**:
     - For `s2 = "babacda"`, the process proceeds as follows:
       - For `b`: map = `{'a': 3, 'b': 2, 'c': 1, 'd': 1}` → decrease `b` → map = `{'a': 3, 'b': 1, 'c':`

	`1, 'd': 1}`
	▪ For `a`: map = `{'a': 3, 'b': 1, 'c': 1, 'd': 1}` → decrease `a` → map = `{'a': 2, 'b': 1, 'c': 1, 'd': 1}`
	▪ For `b`: map = `{'a': 2, 'b': 1, 'c': 1, 'd': 1}` → decrease `b` → map = `{'a': 2, 'b': 0, 'c': 1, 'd': 1}` → remove `b`
	▪ For `a`: map = `{'a': 2, 'c': 1, 'd': 1}` → decrease `a` → map = `{'a': 1, 'c': 1, 'd': 1}`
	▪ For `c`: map = `{'a': 1, 'c': 1, 'd': 1}` → decrease `c` → map = `{'a': 1, 'd': 1}`
	▪ For `d`: map = `{'a': 1, 'd': 1}` → decrease `d` → map = `{'a': 0}` → remove `d`
	▪ For `a`: map = `{'a': 0}` → decrease `a` → map = `{}` (empty)
	4. **Conclusion**:
	o After processing all characters in `s2`, the map is empty, which indicates that the strings `s1` and `s2` are indeed anagrams of each other.
	Output:
	`true`

Output:- true	