Distinct Elements in each Window in C++

```cpp
#include <iostream>
#include <unordered_map> // for unordered_map

using namespace std;

void printDistinct(int arr[], int n, int k) {
   unordered_map<int, int> m; // Declaration of unordered_map to store element frequencies

   // Count frequencies of first window
   for (int i = 0; i < k; i++) {
      m[arr[i]]++;
   }

   // Print the size of the map for the first window
   cout << m.size() << " ";

   // Process subsequent windows
   for (int i = k; i < n; i++) {
      // Remove the element that is moving out of the window
      m[arr[i - k]]--;

      // Remove the element from map if its count becomes zero
      if (m[arr[i - k]] == 0) {
         m.erase(arr[i - k]);
      }

      // Add the new element to the map
      m[arr[i]]++;

      // Print the size of the map for the current window
      cout << m.size() << " ";
   }
}

int main() {
   int arr[] = {10, 10, 5, 3, 20, 5};
   int n = sizeof(arr) / sizeof(arr[0]); // Calculate the size of the array
   int k = 4; // Size of the window

   // Call the function to print distinct elements in every window of size k
   printDistinct(arr, n, k);

   cout << endl;

   return 0;
}
```

Input:

Array: {10, 10, 5, 3, 20, 5}
Window size k = 4

Step 1: Initialize the First Window (size k)

The function first processes the first k elements of the array and calculates their frequencies using an unordered map.

First Window (arr[0..3] = {10, 10, 5, 3}):

- m[10]++ → {10: 2}
- m[5]++ → {10: 2, 5: 1}
- m[3]++ → {10: 2, 5: 1, 3: 1}

Distinct elements count = **3** (keys: 10, 5, 3).

Output so far:

3

Step 2: Slide the Window

Now, we slide the window over the array. For each new element that comes into the window, we remove the element that goes out of the window and update the frequencies accordingly.

Second Window (arr[1..4] = {10, 5, 3, 20}):

1. Remove the element arr[0] (10) that goes out of the window:
   - m[10]-- → {10: 1, 5: 1, 3: 1}
   - Since m[10] == 0, remove 10 from the map.
   - Map becomes: {5: 1, 3: 1}
2. Add the new element arr[4] (20):
   - m[20]++ → {5: 1, 3: 1, 20: 1}

Distinct elements count = **4** (keys: 5, 3, 20).

Output so far:

3 4

Third Window (arr[2..5] = {5, 3, 20, 5}):

1. Remove the element arr[1] (10) that goes out of the window:
   - m[10]-- → {5: 1, 3: 1, 20: 1} (no change since 10 was already removed).
2. Add the new element arr[5] (5):

<table>
<tr><td></td><td>

- m[5]++ → {5: 2, 3: 1, 20: 1}

Distinct elements count = **3** (keys: 5, 3, 20).

Output so far:

3 4 3
</td></tr>
<tr><td colspan="2">Output:
3 4 3</td></tr>
</table>

Frequency in C++	
`#include <iostream>` `#include <unordered_map> // for unordered_map` `using namespace std;` `void countFreq(int arr[], int n) {` `   unordered_map<int, int> hmp; // Declaration of unordered_map to store element frequencies` `   // Count frequencies of each element in the array` `   for (int i = 0; i < n; i++) {` `      int key = arr[i];` `      if (hmp.find(arr[i]) != hmp.end()) {` `         hmp[arr[i]]++;` `      } else {` `         hmp[arr[i]] = 1;` `      }` `   }` `   // Print the frequencies` `   for (auto itr = hmp.begin(); itr != hmp.end(); itr++) {` `      cout << itr->first << " " << itr->second << endl;` `   }` `}` `int main() {` `   int arr[] = {3112102, 3112500, 3112501, 3112700, 3112800};` `   int n = sizeof(arr) / sizeof(arr[0]);` `   countFreq(arr, n);` `   return 0;` `}`	**Input** Array: {3112102, 3112500, 3112501, 3112700, 3112800} Size of array (n) = 5. **Step 1: Initialize an unordered_map** We use an unordered_map named hmp to store element frequencies, where the **key** is the array element, and the **value** is its count. Initially, hmp is empty: hmp = {} **Step 2: Traverse the Array to Count Frequencies** **Iteration 1 (i = 0):** • key = arr[0] = 3112102 • hmp.find(3112102) == hmp.end() (not found in the map). • Add 3112102 to the map with a frequency of 1: hmp = {3112102: 1} **Iteration 2 (i = 1):** • key = arr[1] = 3112500 • hmp.find(3112500) == hmp.end() (not found in the map). • Add 3112500 to the map with a frequency of 1: hmp = {3112102: 1, 3112500: 1} **Iteration 3 (i = 2):** • key = arr[2] = 3112501 • hmp.find(3112501) == hmp.end() (not found in the map). • Add 3112501 to the map with a frequency of 1: hmp = {3112102: 1, 3112500: 1, 3112501: 1} **Iteration 4 (i = 3):** • key = arr[3] = 3112700

	• hmp.find(3112700) == hmp.end() (not found in the map). • Add 3112700 to the map with a frequency of 1: yaml Copy code hmp = {3112102: 1, 3112500: 1, 3112501: 1, 3112700: 1} **Iteration 5 (i = 4):** • key = arr[4] = 3112800 • hmp.find(3112800) == hmp.end() (not found in the map). • Add 3112800 to the map with a frequency of 1: hmp = {3112102: 1, 3112500: 1, 3112501: 1, 3112700: 1, 3112800: 1} **Step 3: Print the Frequencies** Now, we iterate through hmp and print each key-value pair: 1. 3112102 1 2. 3112500 1 3. 3112501 1 4. 3112700 1 5. 3112800 1
Output: 3112800 1 3112102 1 3112500 1 3112700 1 3112501 1	

Get Common elements in C++

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

using namespace std;

void getCommonElements(int a1[], int a2[], int n1, int n2) {
   unordered_map<int, int> hm; // HashMap to store element frequencies from a1

   // Count frequencies of elements in a1
   for (int i = 0; i < n1; i++) {
      hm[a1[i]]++;
   }

   // Find common elements and print them
   vector<int> commonElements;
   for (int i = 0; i < n2; i++) {
      if (hm.find(a2[i]) != hm.end() && hm[a2[i]] > 0) {
         commonElements.push_back(a2[i]);
         hm[a2[i]]--; // Decrement the count in HashMap
      }
   }

   // Print the common elements
   for (int elem : commonElements) {
      cout << elem << " ";
   }
   cout << endl;
}

int main() {
   int a1[] = {5, 5, 9, 8, 5, 5, 8, 0, 3};
   int a2[] = {9, 7, 1, 0, 3, 6, 5, 9, 1, 1, 8, 0, 2, 4, 2, 9, 1, 5};

   int n1 = sizeof(a1) / sizeof(a1[0]);
   int n2 = sizeof(a2) / sizeof(a2[0]);

   getCommonElements(a1, a2, n1, n2);

   return 0;
}
```

Input

Array 1: a1 = {5, 5, 9, 8, 5, 5, 8, 0, 3}
Size (n1) = 9

Array 2: a2 = {9, 7, 1, 0, 3, 6, 5, 9, 1, 1, 8, 0, 2, 4, 2, 9, 1, 5}
Size (n2) = 18

Step 1: Populate the HashMap

We iterate through a1 and populate the unordered_map (hm) with the count of each element in a1.

Iteration Over a1:

Index	Element	HashMap (hm)
0	5	{5: 1}
1	5	{5: 2}
2	9	{5: 2, 9: 1}
3	8	{5: 2, 9: 1, 8: 1}
4	5	{5: 3, 9: 1, 8: 1}
5	5	{5: 4, 9: 1, 8: 1}
6	8	{5: 4, 9: 1, 8: 2}
7	0	{5: 4, 9: 1, 8: 2, 0: 1}
8	3	{5: 4, 9: 1, 8: 2, 0: 1, 3: 1}

Step 2: Find Common Elements

Now, iterate through a2. For each element in a2, check if it exists in hm with a count greater than 0. If yes:

1. Add it to the commonElements list.
2. Decrement its count in hm.

Iteration Over a2:

Index	Element	Found in hm?	Updated hm	Common Elements
0	9	Yes	{5: 4, 9: 0, 8: 2, 0: 1, 3: 1}	[9]
1	7	No	{5: 4, 9: 0,	[9]

Index	Element	Found in hm?	Updated hm	Common Elements
			8: 2, 0: 1, 3: 1}	
2	1	No	{5: 4, 9: 0, 8: 2, 0: 1, 3: 1}	[9]
3	0	Yes	{5: 4, 9: 0, 8: 2, 0: 0, 3: 1}	[9, 0]
4	3	Yes	{5: 4, 9: 0, 8: 2, 0: 0, 3: 0}	[9, 0, 3]
5	6	No	{5: 4, 9: 0, 8: 2, 0: 0, 3: 0}	[9, 0, 3]
6	5	Yes	{5: 3, 9: 0, 8: 2, 0: 0, 3: 0}	[9, 0, 3, 5]
7	9	No	{5: 3, 9: 0, 8: 2, 0: 0, 3: 0}	[9, 0, 3, 5]
8	1	No	{5: 3, 9: 0, 8: 2, 0: 0, 3: 0}	[9, 0, 3, 5]
9	1	No	{5: 3, 9: 0, 8: 2, 0: 0, 3: 0}	[9, 0, 3, 5]
10	8	Yes	{5: 3, 9: 0, 8: 1, 0: 0, 3: 0}	[9, 0, 3, 5, 8]
11	0	No	{5: 3, 9: 0, 8: 1, 0: 0, 3: 0}	[9, 0, 3, 5, 8]
12	2	No	{5: 3, 9: 0, 8: 1, 0: 0, 3: 0}	[9, 0, 3, 5, 8]
13	4	No	{5: 3, 9: 0, 8: 1, 0: 0, 3: 0}	[9, 0, 3, 5, 8]
14	2	No	{5: 3, 9: 0, 8: 1, 0: 0, 3: 0}	[9, 0, 3, 5, 8]

Index	Element	Found in hm?	Updated hm	Common Elements
15	9	No	{5: 3, 9: 0, 8: 1, 0: 0, 3: 0}	[9, 0, 3, 5, 8]
16	1	No	{5: 3, 9: 0, 8: 1, 0: 0, 3: 0}	[9, 0, 3, 5, 8]
17	5	Yes	{5: 2, 9: 0, 8: 1, 0: 0, 3: 0}	[9, 0, 3, 5, 8, 5]

Step 3: Output the Common Elements

The commonElements list is:

[9, 0, 3, 5, 8, 5]

Output:
9 0 3 5 8 5

Highest Frequency Char in C++	
(code)	(explanation)

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

using namespace std;

char getHighestFrequencyChar(string str) {
    unordered_map<char, int> hm; // HashMap to store character frequencies

    // Count frequencies of characters in the string
    for (char ch : str) {
        hm[ch]++;
    }

    char mfc = str[0]; // Initialize most frequent character with the first character

    // Find the character with the highest frequency
    for (auto it = hm.begin(); it != hm.end(); ++it) {
        if (it->second > hm[mfc]) {
            mfc = it->first;
        }
    }

    return mfc;
}

int main() {
    string str = "zmszeqxllzvheqwrofgcuntypejcxovtaqbnqyqlmrwitc";

    char highestFreqChar = getHighestFrequencyChar(str);

    cout << highestFreqChar << endl;

    return 0;
}
```

Input

String: "zmszeqxllzvheqwrofgcuntypejcxovtaqbnqyqlmrwitc"

Step 1: Count Character Frequencies

We iterate through the string str and populate the unordered_map (hm) with the count of each character.

Character Frequency Count:

Character	Count
z	3
m	3
s	2
e	4
q	4
x	2
l	3
v	2
h	1
w	2
r	2
o	2
f	1
g	1
c	2
u	1
n	2
t	2
y	3
p	1
j	1
a	1
b	1

i	1

Step 2: Find the Character with the Highest Frequency

We iterate through the unordered_map (hm) and keep track of the character with the maximum frequency (mfc). Initially, mfc is set to the first character of the string, z.

Iteration Over HashMap:

Current Character	Frequency	hm[mfc]	Update mfc?	Updated mfc
z	3	3	No	z
m	3	3	No	z
s	2	3	No	z
e	4	3	Yes	e
q	4	4	No	e
x	2	4	No	e
l	3	4	No	e
v	2	4	No	e
h	1	4	No	e
w	2	4	No	e
r	2	4	No	e
o	2	4	No	e
f	1	4	No	e
g	1	4	No	e
c	2	4	No	e
u	1	4	No	e
n	2	4	No	e
t	2	4	No	e
y	3	4	No	e
p	1	4	No	e
j	1	4	No	e
a	1	4	No	e
b	1	4	No	e

Current Character	Frequency	hm[mfc]	Update mfc?	Updated mfc
i	1	4	No	e

Step 3: Output

The character with the highest frequency is **q**, appearing **4 times** in the string.

Output

Output:
q

K-Largest Elements in C++

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

void solve(int n, vector<int>& arr, int k) {
   priority_queue<int, vector<int>, greater<int>>
pq; // Min-heap

   for (int i = 0; i < arr.size(); ++i) {
      if (i < k) {
         pq.push(arr[i]);
      } else {
         if (arr[i] > pq.top()) {
            pq.pop();
            pq.push(arr[i]);
         }
      }
   }

   vector<int> result;
   while (!pq.empty()) {
      result.push_back(pq.top());
      pq.pop();
   }

   for (int j = result.size() - 1; j >= 0; --j) {
      cout << result[j] << " ";
   }
   cout << endl;
}

int main() {
   vector<int> num = {44, -5, -2, 41, 12, 19, 21, -6};
   int k = 2;
   solve(num.size(), num, k);

   return 0;
}
```

Input:

Array: {44, -5, -2, 41, 12, 19, 21, -6}
k = 2

Step 1: Initialize Min-Heap (pq)

We use a priority_queue<int, vector<int>, greater<int>> to create a min-heap.

Step 2: Process Array

- **Iteration 0 (i = 0):**
  - Push 44 into the heap.
    Min-Heap: {44}
- **Iteration 1 (i = 1):**
  - Push -5 into the heap.
    Min-Heap: {-5, 44}
- **Iteration 2 (i = 2):**
  - Compare -2 with the heap's top (-5):
    -2 > -5, so:
    - Pop -5 from the heap.
    - Push -2 into the heap.
      Min-Heap: {-2, 44}
- **Iteration 3 (i = 3):**
  - Compare 41 with the heap's top (-2):
    41 > -2, so:
    - Pop -2 from the heap.
    - Push 41 into the heap.
      Min-Heap: {41, 44}
- **Iteration 4 (i = 4):**
  - Compare 12 with the heap's top (41):
    12 < 41, so we skip this element.
    Min-Heap remains unchanged: {41, 44}
- **Iteration 5 (i = 5):**
  - Compare 19 with the heap's top (41):
    19 < 41, so we skip this element.
    Min-Heap remains unchanged: {41, 44}
- **Iteration 6 (i = 6):**
  - Compare 21 with the heap's top (41):
    21 < 41, so we skip this element.
    Min-Heap remains unchanged: {41, 44}
- **Iteration 7 (i = 7):**
  - Compare -6 with the heap's top (41):
    -6 < 41, so we skip this element.
    Min-Heap remains unchanged: {41,

	44} **Step 3: Extract and Store Result** The min-heap contains the top k largest elements: {41, 44}. Extract them and reverse the order to print the largest first. Result: [44, 41]
Output: 44 41	

Merge k sorted elements in C++

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

struct Pair {
    int li;   // List index
    int di;   // Data index (current index in the list)
    int val;  // Value at current index in the list

    Pair(int li, int di, int val) {
        this->li = li;
        this->di = di;
        this->val = val;
    }

    bool operator>(const Pair& other) const {
        return val > other.val;
    }
};

vector<int> mergeKSortedLists(vector<vector<int>>& lists) {
    vector<int> rv;

    // Min-heap priority queue
    priority_queue<Pair, vector<Pair>, greater<Pair>> pq;

    // Initialize the priority queue with the first element from each list
    for (int i = 0; i < lists.size(); ++i) {
        if (!lists[i].empty()) {
            pq.push(Pair(i, 0, lists[i][0]));
        }
    }

    while (!pq.empty()) {
        Pair p = pq.top();
        pq.pop();

        // Add the current value to result vector
        rv.push_back(p.val);

        // Move to the next element in the same list
        p.di++;
        if (p.di < lists[p.li].size()) {
            p.val = lists[p.li][p.di];
            pq.push(p);
        }
    }

    return rv;
}

int main() {
    vector<vector<int>> lists = {
        {10, 20, 30, 40, 50},
        {5, 7, 9, 11, 19, 55, 57},
        {1, 2, 3}
    };
```

Step-by-Step Execution:

1. **Initialization:**
    - We declare a vector rv to store the merged result.
    - We use a **min-heap priority queue** to manage the smallest elements of each list.
    - The Pair struct stores:
        - li: Index of the list from which the element comes.
        - di: Index of the element in that list.
        - val: The value of the element.
2. **Inserting the First Elements into the Min-Heap:** We push the first element of each list into the min-heap:
    - From list 0: {10, 20, 30, 40, 50}, push 10.
    - From list 1: {5, 7, 9, 11, 19, 55, 57}, push 5.
    - From list 2: {1, 2, 3}, push 1.

   At this point, the priority queue (min-heap) looks like this:

   {1, 5, 10}

3. **Processing the Min-Heap:**
    - Pop the smallest element (1) from the heap, add it to the result list rv.
    - Push the next element from the same list (list 2) into the heap, which is 2.

   Now, the heap looks like:

   {2, 5, 10}

    - Pop the smallest element (2), add it to rv.
    - Push the next element from list 2 into the heap, which is 3.

   Now, the heap looks like:

   {3, 5, 10}

    - Pop the smallest element (3), add it to rv.
    - No more elements left in list 2.

   Now, the heap looks like:

   {5, 10}

    - Pop the smallest element (5), add it

```
    vector<int> mlist = mergeKSortedLists(lists);

    for (int val : mlist) {
        cout << val << " ";
    }
    cout << endl;

    return 0;
}
```

to rv.

- Push the next element from list 1 into the heap, which is 7.

Now, the heap looks like:

{7, 10}

- Pop the smallest element (7), add it to rv.
- Push the next element from list 1 into the heap, which is 9.

Now, the heap looks like:

{9, 10}

- Pop the smallest element (9), add it to rv.
- Push the next element from list 1 into the heap, which is 11.

Now, the heap looks like:

{10, 11}

- Pop the smallest element (10), add it to rv.
- Push the next element from list 0 into the heap, which is 20.

Now, the heap looks like:

{11, 20}

- Pop the smallest element (11), add it to rv.
- Push the next element from list 1 into the heap, which is 19.

Now, the heap looks like:

{19, 20}

- Pop the smallest element (19), add it to rv.
- Push the next element from list 1 into the heap, which is 55.

Now, the heap looks like:

{20, 55}

- Pop the smallest element (20), add it to rv.
- Push the next element from list 0

	into the heap, which is 30. Now, the heap looks like: {30, 55} - Pop the smallest element (30), add it to rv. - Push the next element from list 0 into the heap, which is 40. Now, the heap looks like: {40, 55} - Pop the smallest element (40), add it to rv. - Push the next element from list 0 into the heap, which is 50. Now, the heap looks like: {50, 55} - Pop the smallest element (50), add it to rv. - No more elements left in list 0. Now, the heap looks like: {55} - Pop the smallest element (55), add it to rv. - Push the next element from list 1 into the heap, which is 57. Now, the heap looks like: {57} - Pop the smallest element (57), add it to rv. - No more elements left in list 1. **Final Output:** The merged result stored in rv is: 1 2 3 5 7 9 10 11 19 20 30 40 50 55 57s
Output: 1 2 3 5 7 9 10 11 19 20 30 40 50 55 57	

Subarray with 0 sum in C++	
`` #include <iostream> #include <unordered_set> #include <vector> using namespace std; int ZeroSumSubarray(vector<int>& arr) { unordered_set<int> us; int prefix_sum = 0; us.insert(0); // Insert 0 initially to handle cases where the prefix_sum itself is zero for (int i = 0; i < arr.size(); ++i) { prefix_sum += arr[i]; if (us.count(prefix_sum) > 0) return 1; // Found a subarray with sum zero us.insert(prefix_sum); } return 0; // No subarray with sum zero found } int main() { vector<int> arr = {5, 3, 9, -4, -6, 7, -1}; cout << ZeroSumSubarray(arr) << endl; return 0; } ```	**Input:** vector<int> arr = {5, 3, 9, -4, -6, 7, -1}; **Goal:** Check whether there exists a subarray whose sum is zero. **Key Concepts:** • **Prefix Sum**: It is the cumulative sum of elements up to the current index. • **Hash Set (unordered_set)**: Used to store the prefix sums encountered so far. If a prefix sum repeats, it means the sum of elements between these two indices is zero. **Step-by-Step Execution:** 1. **Initialization:** ○ We initialize an unordered set us to store the prefix sums, starting by inserting 0 into it (this helps in case the sum of elements from the start up to the current element is zero). ○ prefix_sum is initialized to 0. 2. **Loop through the array:** ○ We iterate over the array, computing the prefix sum at each step. **Iteration 1:** • i = 0: arr[i] = 5 • prefix_sum = 0 + 5 = 5 • Check if prefix_sum = 5 exists in the set. It doesn't, so we insert 5 into the set. Set us: {0, 5} **Iteration 2:** • i = 1: arr[i] = 3 • prefix_sum = 5 + 3 = 8 • Check if prefix_sum = 8 exists in the set. It doesn't, so we insert 8 into the set. Set us: {0, 5, 8}

	Iteration 3: • i = 2: arr[i] = 9 • prefix_sum = 8 + 9 = 17 • Check if prefix_sum = 17 exists in the set. It doesn't, so we insert 17 into the set. Set us: {0, 5, 8, 17} **Iteration 4:** • i = 3: arr[i] = -4 • prefix_sum = 17 + (-4) = 13 • Check if prefix_sum = 13 exists in the set. It doesn't, so we insert 13 into the set. Set us: {0, 5, 8, 13, 17} **Iteration 5:** • i = 4: arr[i] = -6 • prefix_sum = 13 + (-6) = 7 • Check if prefix_sum = 7 exists in the set. It doesn't, so we insert 7 into the set. Set us: {0, 5, 7, 8, 13, 17} **Iteration 6:** • i = 5: arr[i] = 7 • prefix_sum = 7 + 7 = 14 • Check if prefix_sum = 14 exists in the set. It doesn't, so we insert 14 into the set. Set us: {0, 5, 7, 8, 13, 14, 17} **Iteration 7:** • i = 6: arr[i] = -1 • prefix_sum = 14 + (-1) = 13 • Check if prefix_sum = 13 exists in the set. It **does** exist (it was added in iteration 4). Since prefix_sum = 13 is found in the set, it means there is a subarray between index 4 and index 6 whose sum is zero. Therefore, we return 1.

	Final Output: 1
Output: 1	

Subarray with given sum in C++

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

bool isSum(int arr[], int n, int sum) {
    unordered_set<int> s;
    int pre_sum = 0;
    for (int i = 0; i < n; i++) {
        if (pre_sum == sum) {
            return true;
        }
        pre_sum += arr[i];
        if (s.find(pre_sum - sum) != s.end()) {
            return true;
        }
        s.insert(pre_sum);
    }
    return false;
}

int main() {
    int arr[] = {5, 8, 6, 13, 3, -1};
    int sum = 22;
    int n = sizeof(arr) / sizeof(arr[0]);

    if (isSum(arr, n, sum)) {
        cout << "Subarray with sum " << sum << "
exists." << endl;
    } else {
        cout << "No subarray with sum " << sum << "
exists." << endl;
    }

    return 0;
}
```

Input:

int arr[] = {5, 8, 6, 13, 3, -1};
int sum = 22;

Goal:

Check whether there exists a subarray whose sum equals 22.

Logic Explanation:

The isSum function tries to find a subarray with a sum equal to sum (in this case, 22). It uses a **prefix sum** approach combined with a **hash set** to track the cumulative sums encountered so far.

1. pre_sum: Keeps track of the cumulative sum of elements as we iterate through the array.
2. We check if the difference between the pre_sum and the sum (i.e., pre_sum - sum) has already been encountered. If it has, then there exists a subarray with the required sum.
3. We insert each cumulative sum into a set (s) to help with the lookup.

Step-by-Step Execution:

1. **Initialization:**
    - o We initialize pre_sum = 0 and an empty unordered set s.
2. **Iteration 1 (i = 0):**
    - o arr[i] = 5
    - o pre_sum = 0 + 5 = 5
    - o Check if pre_sum == sum. It's not (5 != 22).
    - o Check if pre_sum - sum = 5 - 22 = -17 is in the set. It is not.
    - o Insert pre_sum = 5 into the set.
    - o Set s = {5}
3. **Iteration 2 (i = 1):**
    - o arr[i] = 8
    - o pre_sum = 5 + 8 = 13
    - o Check if pre_sum == sum. It's not (13 != 22).
    - o Check if pre_sum - sum = 13 - 22 = -9 is in the set. It is not.
    - o Insert pre_sum = 13 into the set.
    - o Set s = {5, 13}
4. **Iteration 3 (i = 2):**

	- arr[i] = 6 - pre_sum = 13 + 6 = 19 - Check if pre_sum == sum. It's not (19 != 22). - Check if pre_sum - sum = 19 - 22 = -3 is in the set. It is not. - Insert pre_sum = 19 into the set. - Set s = {5, 13, 19} 5. **Iteration 4 (i = 3):** - arr[i] = 13 - pre_sum = 19 + 13 = 32 - Check if pre_sum == sum. It's not (32 != 22). - Check if pre_sum - sum = 32 - 22 = 10 is in the set. It is not. - Insert pre_sum = 32 into the set. - Set s = {5, 13, 19, 32} 6. **Iteration 5 (i = 4):** - arr[i] = 3 - pre_sum = 32 + 3 = 35 - Check if pre_sum == sum. It's not (35 != 22). - Check if pre_sum - sum = 35 - 22 = 13 is in the set. **It is!** - Since 13 exists in the set, it means that the sum of the subarray from index 2 to index 4 equals 22. We return true. **Conclusion:** The code returns true because a subarray with sum 22 exists, specifically the subarray {6, 13, 3}. **Final Output:** Subarray with sum 22 exists.
Output: Subarray with sum 22 exists.	