Distinct Elements in each Window in C++

```
#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

```
arr[] = {10, 10, 5, 3, 20, 5}
n = 6
k = 4
```

Dry Run Table (Sliding Window)

Window Index	Elements in Window	Frequencies Map (unordered_map)	Distinct Count
[0–3]	10, 10, 5, 3	{10: 2, 5: 1, 3: 1}	3
[1–4]	10, 5, 3, 20	{10: 1, 5: 1, 3: 1, 20: 1}	4
[2–5]	5, 3, 20, 5	{5: 2, 3: 1, 20: 1}	3

Final Output

```
3 4 3
```

Output:
3 4 3

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[] = {4,4,5,2,3,1,6,7,6};

    int n = sizeof(arr) / sizeof(arr[0]);

    countFreq(arr, n);

    return 0;
}
```

Dry Run of `countFreq(arr, n)`

Input:

```
arr = {4, 4, 5, 2, 3, 1, 6, 7, 6};
n = 9;
```

Step 1: Initialize `unordered_map<int, int> hmp`

- `hmp` is empty at the beginning.

Step 2: Count Frequencies of Elements

Iteration	`arr[i]`	`hmp` (after processing `arr[i]`)
i = 0	4	{4: 1}
i = 1	4	{4: 2}
i = 2	5	{4: 2, 5: 1}
i = 3	2	{4: 2, 5: 1, 2: 1}
i = 4	3	{4: 2, 5: 1, 2: 1, 3: 1}
i = 5	1	{4: 2, 5: 1, 2: 1, 3: 1, 1: 1}
i = 6	6	{4: 2, 5: 1, 2: 1, 3: 1, 1: 1, 6: 1}
i = 7	7	{4: 2, 5: 1, 2: 1, 3: 1, 1: 1, 6: 1, 7: 1}
i = 8	6	{4: 2, 5: 1, 2: 1, 3: 1, 1: 1, 6: 2, 7: 1}

Step 3: Print Frequencies

```
4 2
5 1
2 1
3 1
1 1
6 2
7 1
```

Output:

```
4 2
5 1
2 1
3 1
1 1
6 2
7 1
```

Get Common elements in C++

```
#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++

```
#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 =
"zmszeqxllzvheqwrofgcuntypejcxovtaqbnqyqlmrwit
c";

    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

Current Character	Frequency	hm[mfc]	Update mfc?	Updated mfc
b	1	4	No	e
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;
}
```

Dry Run of `solve(n, arr, k)`

Input:

```
arr = {44, -5, -2, 41, 12, 19, 21, -6};
k = 2;
```

Step 1: Initialize Min-Heap (`priority_queue`)

- Min-heap stores the **top `k` largest** elements.
- **Initial heap (empty):** `pq = {}`

Step 2: Process First `k` Elements (`k = 2`)

Iteration	`arr[i]`	Heap After Push (`pq`)
`i = 0`	`44`	`{44}`
`i = 1`	`-5`	`{-5, 44}`

Step 3: Process Remaining Elements

Iteration	`arr[i]`	Compare With `pq.top()`	Action Taken	Heap After Update
`i = 2`	`-2`	`-5 < -2`	Pop `-5`, Push `-2`	`{-2, 44}`
`i = 3`	`41`	`-2 < 41`	Pop `-2`, Push `41`	`{41, 44}`
`i = 4`	`12`	`41 > 12`	No Change	`{41, 44}`
`i = 5`	`19`	`41 > 19`	No Change	`{41, 44}`
`i = 6`	`21`	`41 > 21`	No Change	`{41, 44}`
`i = 7`	`-6`	`41 > -6`	No Change	`{41, 44}`

Step 4: Extract Elements from Min-Heap

- Extract elements in ascending order: `{41, 44}`
- Reverse order to print in descending: **`44 41`**

Output:
44 41

Merge k sorted elements in C++

```
#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}
    };
```

Dry Run of mergeKSortedLists(lists)

Input:

```
lists = {
    {10, 20, 30, 40, 50},
    {5, 7, 9, 11, 19, 55, 57},
    {1, 2, 3}
};
```

Step 1: Initialize Min-Heap (priority_queue)

- Min-heap stores **(value, list index, data index)** for sorting.
- **Insert the first element of each list:**
  - (10, 0, 0) from list **0** ({10, 20, 30, 40, 50})
  - (5, 1, 0) from list **1** ({5, 7, 9, 11, 19, 55, 57})
  - (1, 2, 0) from list **2** ({1, 2, 3})

Step 2: Extract Minimum & Insert Next Element

Step	Extracted (Min)	Insert Next	Updated Heap
1	(1, 2, 0)	(2, 2, 1)	{(2,2,1), (5,1,0), (10,0,0)}
2	(2, 2, 1)	(3, 2, 2)	{(3,2,2), (5,1,0), (10,0,0)}
3	(3, 2, 2)	None (End)	{(5,1,0), (10,0,0)}
4	(5, 1, 0)	(7, 1, 1)	{(7,1,1), (10,0,0)}
5	(7, 1, 1)	(9, 1, 2)	{(9,1,2), (10,0,0)}
6	(9, 1, 2)	(11, 1, 3)	{(10,0,0), (11,1,3)}
7	(10, 0, 0)	(20, 0, 1)	{(11,1,3), (20,0,1)}
8	(11, 1, 3)	(19, 1, 4)	{(19,1,4), (20,0,1)}
9	(19, 1, 4)	(55, 1, 5)	{(20,0,1), (55,1,5)}
10	(20, 0, 1)	(30, 0, 2)	{(30,0,2), (55,1,5)}
11	(30, 0, 2)	(40, 0, 3)	{(40,0,3), (55,1,5)}
12	(40, 0, 3)	(50, 0, 4)	{(50,0,4), (55,1,5)}
13	(50, 0, 4)	None (End)	{(55,1,5)}
14	(55, 1, 5)	(57, 1, 6)	{(57,1,6)}
15	(57, 1, 6)	None (End)	{}

Final Merged List:

{1, 2, 3, 5, 7, 9, 10, 11, 19, 20, 30, 40, 50, 55, 57}

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

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

    return 0;
}
```

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;
}
```

Dry Run of `ZeroSumSubarray(arr)`

Input:

```
arr = {5, 3, 9, -4, -6, 7, -1};
```

Step 1: Initialize Variables

- **Prefix Sum (`prefix_sum`)** = `0`
- **Hash Set (`us`)** = `{0}` (We insert 0 initially to handle cases where the prefix sum itself is zero)

Step 2: Iterating Over the Array

Iteration	`arr[i]`	`prefix_sum` (cumulative)	`us` (hash set)	Check if `prefix_sum` exists in `us`
1	5	0 + 5 = 5	{0, 5}	No
2	3	5 + 3 = 8	{0, 5, 8}	No
3	9	8 + 9 = 17	{0, 5, 8, 17}	No
4	-4	17 - 4 = 13	{0, 5, 8, 17, 13}	No
5	-6	13 - 6 = 7	{0, 5, 8, 17, 13, 7}	No
6	7	7 + 7 = 14	{0, 5, 8, 17, 13, 7, 14}	No
7	-1	14 - 1 = 13	{0, 5, 8, 17, 13, 7, 14}	**Yes (13 exists in set!)**

Step 3: Return Result

- Since `prefix_sum = 13` **already exists in**

	us, it means there exists a subarray with sum `0`. • **Return `1` (True)**.
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;
}
```

Dry Run of `isSum()` Function

Input:

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

Step 1: Initialize Variables

- **Prefix Sum (`pre_sum`)** = `0`
- **Hash Set (`s`)** = `{}` (Empty initially)

Step 2: Iterating Over the Array

Iteration	`arr[i]`	`pre_sum` (cumulative)	`pre_sum - sum`	Check if `pre_sum - sum` exists in set	Update Hash Set
1	5	0 + 5 = 5	5 - 22 = -17	No	{5}
2	8	5 + 8 = 13	13 - 22 = -9	No	{5, 13}
3	6	13 + 6 = 19	19 - 22 = -3	No	{5, 13, 19}
4	13	19 + 13 = 32	32 - 22 = 10	No	{5, 13, 19, 32}
5	3	32 + 3 = 35	35 - 22 = 13	**Yes** (13 exists in set)	{5, 13, 19, 32, 35}
6	-1	35 + (-1) = 34	34 - 22 = 12	No	{5, 13, 19, 32, 35, 34}

Step 3: Return Result

- At iteration 5, when `pre_sum = 35`, `pre_sum - sum = 13` is found in the hash set, which means there exists a subarray with a sum of `22`.
- **Return `true`**.

Output: Subarray with sum 22 exists.	