Fast and Last Index in C++

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

void findFirstAndLastIndex(int arr[], int n,
int d) {
   int low = 0;
   int high = n - 1;
   int firstIndex = -1;
   int lastIndex = -1;

   // Finding the first occurrence
   while (low <= high) {
      int mid = low + (high - low) / 2;
      if (d > arr[mid]) {
         low = mid + 1;
      } else if (d < arr[mid]) {
         high = mid - 1;
      } else {
         firstIndex = mid;
         high = mid - 1;
      }
   }

   // Finding the last occurrence
   low = 0;
   high = n - 1;
   while (low <= high) {
      int mid = low + (high - low) / 2;
      if (d > arr[mid]) {
         low = mid + 1;
      } else if (d < arr[mid]) {
         high = mid - 1;
      } else {
         lastIndex = mid;
         low = mid + 1;
      }
   }

   cout << "First Index: " << firstIndex <<
endl;
   cout << "Last Index: " << lastIndex <<
endl;
}

int main() {
   int arr[] = {1, 5, 10, 15, 22, 33, 33, 33, 33,
33, 40, 42, 55, 66, 77, 33};
   int n = sizeof(arr) / sizeof(arr[0]);
   int d = 33;

   findFirstAndLastIndex(arr, n, d);

   return 0;
}
```

Dry Run Example (on sorted array):

Sorted version of the array:

{1, 5, 10, 15, 22, 33, 33, 33, 33, 33, 33, 40, 42, 55, 66, 77}

We want to find **first and last index of 33**.

First Occurrence:

Iteration	low	high	mid	arr[mid]	firstIndex	high (updated)
1	0	15	7	33	7	6
2	0	6	3	15		
3	4	6	5	33	5	4
4	4	4	4	22		

➡ First index = **5**

Last Occurrence:

Iteration	low	high	mid	arr[mid]	lastIndex	low (updated)
1	0	15	7	33	7	8
2	8	15	11	40		
3	8	10	9	33	9	10
4	10	10	10	33	10	11

➡ Last index = **10**

🖶 **Final Output:**

First Index: 5
Last Index: 10

First Index: 5 Last Index: 10	

IsSorted in C++

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

bool isSortedEff(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        if (arr[i] < arr[i - 1]) {
            return false;
        }
    }
    return true;
}

bool isSorted(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[i]) {
                return false;
            }
        }
    }
    return true;
}

int main() {
    int arr1[] = {1, 2, 3, 4, 5, 6};
    int arr2[] = {11, 2, 3, 4, 5, 6};
    int n1 = sizeof(arr1) / sizeof(arr1[0]);
    int n2 = sizeof(arr2) / sizeof(arr2[0]);

    cout << boolalpha; // Print boolean values as true/false
    cout << isSortedEff(arr1, n1) << endl;
    cout << isSortedEff(arr2, n2) << endl;

    cout << isSorted(arr1, n1) << endl;
    cout << isSorted(arr2, n2) << endl;

    return 0;
}
```

Check if an array is **sorted in non-decreasing order** (each element is ≤ the next).

🔍 Difference between isSortedEff and isSorted:

Function	Approach	Time Complexity
isSortedEff	Linear scan (compare adjacent)	**O(n)**
isSorted	Brute force (nested loops)	**O(n²)**

✅ Dry Run with Sample Arrays

Array 1: {1, 2, 3, 4, 5, 6} (Sorted)

isSortedEff(arr1, n1):

i	arr[i-1]	arr[i]	Comparison	Result
1	1	2	2 ≥ 1	✅
2	2	3	3 ≥ 2	✅
3	3	4	4 ≥ 3	✅
4	4	5	5 ≥ 4	✅
5	5	6	6 ≥ 5	✅
→ All passed → **Returns: true**				

isSorted(arr1, n1): Checks every pair (i, j) where j > i:

- For every arr[i] <= arr[j] → all OK → **Returns: true**

Array 2: {11, 2, 3, 4, 5, 6} (Not sorted)

isSortedEff(arr2, n2):

i	arr[i-1]	arr[i]	Comparison	Result
1	11	2	2 < 11 ❌	●
→ Early exit → **Returns:**				

	<table><tr><th>i</th><th>arr[i-1]</th><th>arr[i]</th><th>Comparison</th><th>Result</th></tr><tr><td>false</td><td></td><td></td><td></td><td></td></tr></table> **isSorted(arr2, n2)**: • (0,1) → 2 < 11 → ✗ → **Returns: false** 🖨 Output: true false true false
true false true false	

Leaders Array in C++

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

void leaders(int arr[], int n) {
    int curr = arr[n - 1];
    cout << curr << " ";

    for (int i = n - 2; i >= 0; i--) {
        if (arr[i] > curr) {
            curr = arr[i];
            cout << curr << " ";
        }
    }
}

int main() {
    int arr[] = {7, 10, 4, 10, 6, 5, 2};
    int n = sizeof(arr) / sizeof(arr[0]);

    leaders(arr, n);
    cout << endl;

    return 0;
}
```

Dry Run Table

Input array: {7, 10, 4, 10, 6, 5, 2}

We process from **right to left**:

Index	arr[i]	Current Leader (curr)	Is arr[i] > curr?	Print Leader?	Updated curr
6	2	2	-	✓	2
5	5	2	✓	✓	5
4	6	5	✓	✓	6
3	10	6	✓	✓	10
2	4	10	✗	✗	10
1	10	10	✗	✗	10
0	7	10	✗	✗	10

✓ **Output (Printed from right to left):**

2 5 6 10

2 5 6 10

Majority element in C++

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

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

    count = 0;
    for (int i = 0; i < n; i++) {
        if (arr[res] == arr[i]) {
            count++;
        }
    }

    if (count <= n / 2) {
        res = -1;
    }
    return res;
}

int main() {
    int arr[] = {6, 8, 4, 8, 8};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << majority(arr, n) << endl;

    return 0;
}
```

Array Given:

arr[] = {6, 8, 4, 8, 8}
n = 5

We need to find the element (if any) that appears **more than 5 / 2 = 2** times.

🔁 Moore's Voting Algorithm Dry Run

We'll go step-by-step through the first for loop which finds a *candidate*.

i	arr[i]	arr[res]	count	Explanation
0	6	6	1	Initial candidate at index 0
1	8	6	0	8 ≠ 6 → count--
		8	1	count = 0 → new candidate at index 1
2	4	8	0	4 ≠ 8 → count--
		4	1	count = 0 → new candidate at index 2
3	8	4	0	8 ≠ 4 → count--
		8	1	count = 0 → new candidate at index 3
4	8	8	2	8 == 8 → count++

Candidate Index: res = 3, arr[3] = 8

✅ Second loop: Confirm the candidate

We check how many times 8 appears in the array.

```
count = 0;
for (int i = 0; i < n; i++) {
    if (arr[i] == 8) count++;
}
```

8 appears **3 times** (at indices 1, 3, and 4).

Since 3 > 2, it **is** the majority element.

✅ Final Output

3

	That’s the index of the majority element 8.
3	

Max Subarray sum in C++

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

int maxsub(int arr[], int n) {
    int res = arr[0];
    int maxEnding = arr[0];
    for (int i = 1; i < n; i++) {
        maxEnding = max(maxEnding + arr[i], arr[i]);
        res = max(res, maxEnding);
    }
    return res;
}

int main() {
    int arr[] = {-3, 8, -2, 4, -5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << maxsub(arr, n) << endl;
    return 0;
}
```

Input:

arr[] = {-3, 8, -2, 4, -5, 6}
n = 6

📓 **Variables:**

- res: Stores the **maximum subarray sum found so far**
- maxEnding: Stores the **maximum subarray sum ending at the current index**

🔁 **Dry Run Table:**

i	arr[i]	maxEnding = max(maxEnding + arr[i], arr[i])	res = max(res, maxEnding)
0	-3	maxEnding = -3	res = -3
1	8	max(-3 + 8, 8) = 8	res = 8
2	-2	max(8 - 2, -2) = 6	res = 8
3	4	max(6 + 4, 4) = 10	res = 10
4	-5	max(10 - 5, -5) = 5	res = 10
5	6	max(5 + 6, 6) = 11	res = 11

✅ **Final Output:**

11

11

Tapping Rain Water in C++

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

int getWater(int arr[], int n) {
    int res = 0;
    for (int i = 0; i < n; i++) {
        int lmax = arr[i];
        for (int j = 0; j < i; j++) {
            lmax = max(arr[j], lmax);
        }
        int rmax = arr[i];
        for (int j = i + 1; j < n; j++) {
            rmax = max(arr[j], rmax);
        }

        res += min(lmax, rmax) - arr[i];
    }
    return res;
}

int main() {
    int arr[] = {3, 0, 1, 2, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << getWater(arr, n) << endl;
    return 0;
}
```

Problem Explanation: Trapping Rain Water

At each index i, the amount of water it can hold is:

water_at_i = min(lmax, rmax) - arr[i]

Where:

- lmax: Max height to the left of i (including i)
- rmax: Max height to the right of i (including i)
- If min(lmax, rmax) - arr[i] > 0, it adds to total water trapped.

📊 Dry Run Table

Array: {3, 0, 1, 2, 5}

i	arr[i]	lmax (max left)	rmax (max right)	min(lmax, rmax)	Water at i = min(lmax, rmax) - arr[i]	res
0	3	3	5	3	0	0
1	0	3	5	3	3	3
2	1	3	5	3	2	5
3	2	3	5	3	1	6
4	5	5	5	5	0	6

✅ Final Output:

6

Output:
6

Remove Duplicates in C++

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

int removeDup(int arr[], int n) {
    int res = 1;
    for (int i = 1; i < n; i++) {
        if (arr[i] != arr[res - 1]) {
            arr[res] = arr[i];
            res++;
        }
    }
    return res;
}

int main() {
    int arr[] = {2, 2, 3, 4, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    int p = removeDup(arr, n);

    cout << "After Removal" << endl;

    for (int i = 0; i < p; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Dry Run Table

i	arr[i]	arr[res - 1]	Condition Met (!=)	Action	arr[res]	res
1	2	2	✗ No	Skip	-	1
2	3	2	✓ Yes	arr[1] = 3	3	2
3	4	3	✓ Yes	arr[2] = 4	4	3
4	5	4	✓ Yes	arr[3] = 5	5	4
5	6	5	✓ Yes	arr[4] = 6	6	5

✓ **Final Values:**

- res = 5 → means 5 unique elements.
- Modified array (first res elements):

  arr[] = {2, 3, 4, 5, 6}

```
After Removal
2 3 4 5 6
```

Rotate Array in C++

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

void rotate(int arr[], int d, int n) {
    int temp[d];
    for (int i = 0; i < d; i++) {
        temp[i] = arr[i];
    }

    for (int i = d; i < n; i++) {
        arr[i - d] = arr[i];
    }

    for (int i = 0; i < d; i++) {
        arr[n - d + i] = temp[i];
    }

    for (int i = 0; i < n; i++) {
        cout << " " << arr[i];
    }
    cout << endl;
}

int main() {
    int arr[] = {1, 3, 6, 2, 5, 4, 3, 2, 4};
    int n = sizeof(arr) / sizeof(arr[0]);
    rotate(arr, 5, n);
    return 0;
}
```

Input:

arr[] = {1, 3, 6, 2, 5, 4, 3, 2, 4}
d = 5
n = 9

🔁 Step-by-step Breakdown:

1. Store first d elements in temp

temp = {1, 3, 6, 2, 5}

i	temp[i]
0	1
1	3
2	6
3	2
4	5

2. Shift remaining n - d elements to the left

arr[0] = arr[5] → 4
arr[1] = arr[6] → 3
arr[2] = arr[7] → 2
arr[3] = arr[8] → 4

i	arr[i] (after shift)
0	4
1	3
2	2
3	4

3. Copy temp back to the end

arr[4] = temp[0] = 1
arr[5] = temp[1] = 3
arr[6] = temp[2] = 6
arr[7] = temp[3] = 2
arr[8] = temp[4] = 5

i	arr[i] (final state)
4	1

i	arr[i] (final state)
5	3
6	6
7	2
8	5

Final Output:

4 3 2 4 1 3 6 2 5

4 3 2 4 1 3 6 2 5