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

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

string addStrings(string num1, string num2) {
    string res = "";

    int i = num1.length() - 1;
    int j = num2.length() - 1;
    int carry = 0;

    while (i >= 0 || j >= 0 || carry != 0) {
        int ival = i >= 0 ? num1[i] - '0' : 0;
        int jval = j >= 0 ? num2[j] - '0' : 0;

        int sum = ival + jval + carry;
        res = to_string(sum % 10) + res;
        carry = sum / 10;

        i--;
        j--;
    }

    return res;
}

int main() {
    string n1 = "123";
    string n2 = "23";
    string res = addStrings(n1, n2);
    cout << res << endl; // Output should be 146

    return 0;
}
```

Input:

```
n1 = "123"
n2 = "23"
```

🔁 Dry Run Table:

Step	i	j	num1[i]	num2[j]	ival	jval	carry (before)	sum = ival + jval + carry	res	carry (after)
1	2	1	'3'	'3'	3	3	0	6	"6"	0
2	1	0	'2'	'2'	2	2	0	4	"46"	0
3	0	-1	'1'	-	1	0	0	1	"146"	0

✅ Final Output:

```
"146"
```

Island Perimeter in C++

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

int perimeter(vector<vector<int>>& grid) {
    int p = 0;
    int rows = grid.size();
    int cols = grid[0].size();

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (grid[i][j] == 1) {
                p += 4;

                if (i > 0 && grid[i - 1][j] == 1) {
                    p -= 2;
                }
                if (j > 0 && grid[i][j - 1] == 1) {
                    p -= 2;
                }
            }
        }
    }
    return p;
}

int main() {
    vector<vector<int>> grid = {
        {1, 0, 0},
        {1, 1, 1},
        {0, 1, 0},
        {0, 1, 0}
    };

    int p = perimeter(grid);
    cout << p << endl;

    return 0;
}
```

Input Grid:

```
grid = {
    {1, 0, 0},
    {1, 1, 1},
    {0, 1, 0},
    {0, 1, 0}
};
```

Visualized:

```
1 0 0
1 1 1
0 1 0
0 1 0
```

🔁 Dry Run Strategy:

- Each land cell contributes +4 to perimeter.
- Each shared edge with another land cell subtracts 2.

🔍 Dry Run Table:

Cell (i,j)	grid[i][j]	+4	Top Neighbor = 1	Left Neighbor = 1	Net Contribution
(0,0)	1	4	✗	✗	4
(1,0)	1	4	✓ (0,0)	✗	2 (4-2)
(1,1)	1	4	✗	✓ (1,0)	2 (4-2)
(1,2)	1	4	✗	✓ (1,1)	2 (4-2)
(2,1)	1	4	✓ (1,1)	✗	2 (4-2)
(3,1)	1	4	✓ (2,1)	✗	2 (4-2)

✅ Total Perimeter:

= 4 + 2 + 2 + 2 + 2 + 2 = 14

✅ Output:

14

Max Avg. Subarray in C++

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

double solution(vector<int>& nums, int k) {
    int sum = 0;
    for (int i = 0; i < k; i++) {
        sum += nums[i];
    }

    int max_sum = sum;

    for (int i = k; i < nums.size(); i++) {
        sum += nums[i];
        sum -= nums[i - k];
        max_sum = max(max_sum, sum);
    }

    return static_cast<double>(max_sum) / k;
}

int main() {
    vector<int> nums = {-10, 5, -6, 8, -7, 2, -4, 8, -6, 7};
    int k = 3;
    cout << solution(nums, k) << endl;

    return 0;
}
```

Input:

nums = {-10, 5, -6, 8, -7, 2, -4, 8, -6, 7}
k = 3

🔍 Dry Run Table:

We'll track the sum of every window of size 3:

Window (Indexes)	Elements	Window Sum	max_sum
0–2	-10, 5, -6	-11	-11
1–3	5, -6, 8	7	7
2–4	-6, 8, -7	-5	7
3–5	8, -7, 2	3	7
4–6	-7, 2, -4	-9	7
5–7	2, -4, 8	6	7
6–8	-4, 8, -6	-2	7
7–9	8, -6, 7	9	**9**

✅ Final Output:

9 / 3 = 3.0

✓ Output: 3

3

Max Chunks to make array sorted in C++

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

int maxChunksToSorted(vector<int>& arr) {
    int max_val = 0;
    int count = 0;

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

        if (i == max_val) {
            count++;
        }
    }

    return count;
}

int main() {
    vector<int> arr = {4, 3, 2, 1, 0};
    int res = maxChunksToSorted(arr);
    cout << res << endl;

    return 0;
}
```

Input:

vector<int> arr = {4, 3, 2, 1, 0};

🔍 Dry Run Table:

Let's walk through the loop step-by-step and record values:

i	arr[i]	max_val (max so far)	i == max_val?	count
0	4	4	✗	0
1	3	4	✗	0
2	2	4	✗	0
3	1	4	✗	0
4	0	4	✓	1

✅ Output:

1

1

Max product of three in C++

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

int maxProduct(vector<int>& nums) {
    int min1 = INT_MAX, min2 = INT_MAX;
    int max1 = INT_MIN, max2 = INT_MIN,
max3 = INT_MIN;

    for (int val : nums) {
        if (val > max1) {
            max3 = max2;
            max2 = max1;
            max1 = val;
        } else if (val > max2) {
            max3 = max2;
            max2 = val;
        } else if (val > max3) {
            max3 = val;
        }

        if (val < min1) {
            min2 = min1;
            min1 = val;
        } else if (val < min2) {
            min2 = val;
        }
    }

    return max(min1 * min2 * max1, max1 *
max2 * max3);
}

int main() {
    vector<int> nums = {2, 4, 6, 7};
    int result = maxProduct(nums);
    cout << result << endl;
    return 0;
}
```

Input:

nums = {2, 4, 6, 7}

🔍 Variables Tracked:

Iteration	val	max1	max2	max3	min1	min2
1	2	2	INT_MIN	INT_MIN	2	INT_MAX
2	4	4	2	INT_MIN	2	4
3	6	6	4	2	2	4
4	7	7	6	4	2	4

✔ Computed Products:

- min1 * min2 * max1 = 2 * 4 * 7 = 56
- max1 * max2 * max3 = 7 * 6 * 4 = 168

🧠 Output:

return max(56, 168); // → 168

168

No of subarrays with odd sum in C++

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

int nos(int arr[], int n) {
    long long ans = 0;
    int even = 0;
    int odd = 0;
    int sum = 0;

    for (int i = 0; i < n; i++) {
        sum += arr[i];
        if (sum % 2 == 0) {
            ans += odd;
            even++;
        } else {
            ans += 1 + even;
            odd++;
        }
    }
    return ans % 1000000007;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

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

    return 0;
}
```

Input:

arr = {1, 2, 3, 4, 5, 6, 7}

🔍 Key Variables Tracked:

- sum → cumulative sum from start to current index
- even → count of prefix sums that are even so far
- odd → count of prefix sums that are odd so far
- ans → count of subarrays with odd sum

🧮 Dry Run Table:

i	arr[i]	sum	sum%2	Action	ans	even	odd
0	1	1	1 (odd)	Add 1 + even (0) → ans += 1	1	0	1
1	2	3	1 (odd)	Add 1 + even (0) → ans += 1	2	0	2
2	3	6	0 (even)	Add odd (2) → ans += 2	4	1	2
3	4	10	0 (even)	Add odd (2) → ans += 2	6	2	2
4	5	15	1 (odd)	Add 1 + even (2) → ans += 3	9	2	3
5	6	21	1 (odd)	Add 1 + even (2) → ans += 3	12	2	4
6	7	28	0 (even)	Add odd (4) → ans += 4	16	3	4

✅ Final Output:

16

16

Reverse Vowel of String in C++

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
bool isVowel(char ch) {
    return (ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'U' ||
            ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u');
}

string reverseVowel(string s) {
    int left = 0;
    int right = s.length() - 1;

    while (left < right) {
        while (left < right && !isVowel(s[left])) {
            left++;
        }

        while (left < right && !isVowel(s[right])) {
            right--;
        }

        if (left < right) {
            swap(s[left], s[right]);
            left++;
            right--;
        }
    }

    return s;
}

int main() {
    string s = "hello";
    string result = reverseVowel(s);
    cout << result << endl; // Output should be "holle"
    return 0;
}
```

Input:

string s = "hello";

Vowels: e, o

🔁 Dry Run Table:

Step	left	right	s[left]	s[right]	Action	String After Change
1	0	4	h	o	h is not a vowel → left++	"hello"
2	1	4	e	o	Both are vowels → swap e and o	**"holle"**
3	2	3	l	l	No further vowel swap needed	"holle"

✅ Final Output:

holle

holle

Two Sum in C++

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

vector<vector<int>> twoSum(vector<int>
nums, int target) {
    vector<vector<int>> res;
    int n = nums.size();
    sort(nums.begin(), nums.end()); // Sorting
the array

    int left = 0, right = n - 1;
    while (left < right) {
        if (left > 0 && nums[left] == nums[left -
1]) { // Skip duplicates for left pointer
            left++;
            continue;
        }

        int sum = nums[left] + nums[right];

        if (sum == target) {
            res.push_back({nums[left],
nums[right]});
            left++;
            right--;

            // Skip duplicates for both left and
right pointers
            while (left < right && nums[left] ==
nums[left - 1]) left++;
            while (left < right && nums[right] ==
nums[right + 1]) right--;

        } else if (sum > target) {
            right--;
        } else {
            left++;
        }
    }

    return res;
}

int main() {
    vector<int> nums = {2, 2, 4, 3, 1, 6, 6, 7, 5,
9, 1, 8, 9};
    int target = 10;

    vector<vector<int>> res = twoSum(nums,
target);

    // Sorting each pair and then sorting all
pairs lexicographically
    sort(res.begin(), res.end(), [](const
vector<int>& a, const vector<int>& b) {
        return a[0] == b[0] ? a[1] < b[1] : a[0] <
b[0];
    });
```

Input:

nums = {2, 2, 4, 3, 1, 6, 6, 7, 5, 9, 1, 8, 9}
target = 10

After sorting:

nums = {1, 1, 2, 2, 3, 4, 5, 6, 6, 7, 8, 9, 9}

🔍 Step-by-step Table Dry Run:

Step	left	right	nums[left]	nums[right]	sum	Action	Result
1	0	12	1	9	10	Found a pair, store it	{1, 9}
	1	11	1	9	10	Skip duplicate left	
	2	11	2	9	11	Sum > target, move right--	
2	2	10	2	8	10	Found a pair, store it	{1, 9}, {2, 8}
	3	9	2	7	9	Skip duplicate left, move left++	
3	4	9	3	7	10	Found a pair, store it	{1,9},{2,8}, {3,7}
4	5	8	4	6	10	Found a pair, store it	{1,9},{2,8}, {3,7},{4,6}
5	6	7	5	6	11	Sum > target, move right--	
6	6	6	5	5	10	Stop (left >= right)	

✅ Final Result:

{ {1, 9}, {2, 8}, {3, 7}, {4, 6} }

```
    // Printing the result
    for (auto& pair : res) {
        for (int val : pair) {
            cout << val << " ";
        }
        cout << endl;
    }

    return 0;
}
```

```
1 9
2 8
3 7
4 6
```

All Subarray in C++

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

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

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

    return 0;
}
```

Input:

arr[] = {1, 2, 3, 4};

Loop Structure:

- sp: Start point of subarray
- ep: End point of subarray
- i: Index for printing elements from sp to ep

Dry Run Table:

sp	ep	Subarray Printed
0	0	1
0	1	1 2
0	2	1 2 3
0	3	1 2 3 4
1	1	2
1	2	2 3
1	3	2 3 4
2	2	3
2	3	3 4
3	3	4

Output:

```
1
1 2
1 2 3
1 2 3 4
2
2 3
2 3 4
3
3 4
4
```


```
1
1 2
1 2 3
1 2 3 4
2
2 3
2 3 4
3
3 4
4
```

Print Boundary in C++

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

void printBoundary(vector<vector<int>>& mat) {
    int n = mat.size();
    int m = mat[0].size();

    // Print top row
    for (int j = 0; j < m; j++) {
        cout << mat[0][j] << " ";
    }

    // Print right column (excluding the top and bottom elements already printed)
    for (int i = 1; i < n; i++) {
        cout << mat[i][m - 1] << " ";
    }

    // Print bottom row (excluding the bottom-right corner already printed)
    if (n > 1) {
        for (int j = m - 2; j >= 0; j--) {
            cout << mat[n - 1][j] << " ";
        }
    }

    // Print left column (excluding the top-left and bottom-left corners already printed)
    if (m > 1) {
        for (int i = n - 2; i > 0; i--) {
            cout << mat[i][0] << " ";
        }
    }
}

int main() {
    vector<vector<int>> mat = {
        {1, 2, 3, 4, 5},
        {6, 7, 8, 9, 10},
        {11, 12, 13, 14, 15},
        {16, 17, 18, 19, 20},
        {21, 22, 23, 24, 25}
    };

    printBoundary(mat);
    cout << endl;

    return 0;
}
```

Input Matrix (5x5):

```
[
 [ 1,  2,  3,  4,  5 ],
 [ 6,  7,  8,  9, 10 ],
 [11, 12, 13, 14, 15 ],
 [16, 17, 18, 19, 20 ],
 [21, 22, 23, 24, 25 ]
]
```

Step-by-step Dry Run Table:

Step	Indices	Printed Values
Top row	mat[0][0 to 4]	1 2 3 4 5
Right column	mat[1 to 4][4]	10 15 20 25
Bottom row	mat[4][3 to 0]	24 23 22 21
Left column	mat[3 to 1][0]	16 11 6

Dry Run Table

Phase	Loop Variable(s)	Value Printed
Top Row	j = 0 to 4	1 2 3 4 5
Right Col	i = 1 to 4	10 15 20 25
Bottom Row	j = 3 to 0 (reverse)	24 23 22 21
Left Col	i = 3 to 1 (reverse)	16 11 6

✅ Final Output:

1 2 3 4 5 10 15 20 25 24 23 22 21 16 11 6

1 2 3 4 5 10 15 20 25 24 23 22 21 16 11 6

First Missing Positive in C++

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

int firstMissingPositive(vector<int>& nums) {
    int n = nums.size();

    int i = 0;
    while (i < n) {
        if (nums[i] == i + 1) {
            i++;
            continue;
        }

        if (nums[i] <= 0 || nums[i] > n) {
            i++;
            continue;
        }

        int idx1 = i;
        int idx2 = nums[i] - 1;

        if (nums[idx1] == nums[idx2]) {
            i++;
            continue;
        }

        int temp = nums[idx1];
        nums[idx1] = nums[idx2];
        nums[idx2] = temp;
    }

    for (int j = 0; j < n; j++) {
        if (nums[j] != j + 1) {
            return j + 1;
        }
    }

    return n + 1;
}

int main() {
    vector<int> nums = {3, 4, -1, 1};
    int result = firstMissingPositive(nums);
    cout << "First missing positive: " << result << endl;
    return 0;
}
```

Input:

vector<int> nums = {3, 4, -1, 1};

Goal:

Find the **smallest positive integer** that is **missing** from the array.

Algorithm Insight:

You're trying to **place each positive integer x (1 ≤ x ≤ n)** at index x - 1 using cyclic swaps.

Dry Run Table:

While loop swaps

Step	i	nums[i]	Action	nums after
1	0	3	swap nums[0] with nums[2] (index 2 = 3 - 1)	{-1, 4, 3, 1}
2	0	-1	invalid (<= 0), move to i = 1	{-1, 4, 3, 1}
3	1	4	swap nums[1] with nums[3] (index 3 = 4 - 1)	{-1, 1, 3, 4}
4	1	1	swap nums[1] with nums[0] (index 0 = 1 - 1)	{1, -1, 3, 4}
5	1	-1	invalid, move to i = 2	{1, -1, 3, 4}
6	2	3	already at correct index (2 = 3 - 1)	no change
7	3	4	already at correct index (3 = 4 - 1)	no change

Final nums array after placements:

{1, -1, 3, 4}

Final Check:

Go through the array to find first j where nums[j] != j + 1:

j	nums[j]	j + 1	Match?
0	1	1	✓
1	-1	2	✗ → return 2

	📄 **Output:** First missing positive: 2
First missing positive: 2	

Range Sum in C++

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

vector<int> prefixSum;

void NumArray(vector<int>& nums) {
    prefixSum.resize(nums.size());
    prefixSum[0] = nums[0];
    for (int i = 1; i < nums.size(); i++) {
        prefixSum[i] = prefixSum[i - 1] + nums[i];
    }
}

int sumRange(int i, int j) {
    if (i == 0) {
        return prefixSum[j];
    }
    return prefixSum[j] - prefixSum[i - 1];
}

int main() {
    vector<int> arr = {1, 2, 3, 4};
    NumArray(arr);
    int res = sumRange(1, 2);
    cout << res << endl; // Output should be 5

    return 0;
}
```

Prefix Sum Table Construction in NumArray(arr)

Let's build prefixSum[] based on the input arr = {1, 2, 3, 4}.

Index i	nums[i]	prefixSum[i] = prefixSum[i - 1] + nums[i]	prefixSum array
0	1	1	[1]
1	2	1 + 2 = 3	[1, 3]
2	3	3 + 3 = 6	[1, 3, 6]
3	4	6 + 4 = 10	[1, 3, 6, 10]

Final prefixSum = [1, 3, 6, 10]

🎁 **sumRange(1, 2) Execution**

We want to find sum from index 1 to 2 in original array (2 + 3 = 5).

Since i != 0, it uses:

prefixSum[2] - prefixSum[0] = 6 - 1 = 5

Expression	Value
prefixSum[2]	6
prefixSum[0]	1
Result	5

✅ Output printed: **5**

5

Rotate Image in C++

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

void rotate(vector<vector<int>>& matrix) {
    int n = matrix.size();
    int m = matrix[0].size();

    // Transpose the matrix
    for (int i = 0; i < n; i++) {
        for (int j = i; j < m; j++) {
            swap(matrix[i][j], matrix[j][i]);
        }
    }
    // Reverse each row
    for (int i = 0; i < n; i++) {
        int sp = 0;
        int ep = m - 1;

        while (sp < ep) {
            swap(matrix[i][sp], matrix[i][ep]);
            sp++;
            ep--;
        }
    }
}
void print2DArray(const vector<vector<int>>& array)
{
    for (size_t i = 0; i < array.size(); i++) {
        for (size_t j = 0; j < array[i].size(); j++) {
            cout << array[i][j] << " ";
        }
        cout << endl;
    }
}
int main() {
    vector<vector<int>> matrix = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
    cout << "Original matrix:" << endl;
    print2DArray(matrix);
    rotate(matrix);
    cout << "Rotated matrix:" << endl;
    print2DArray(matrix);
    return 0;
}
```

Input Matrix:

Original matrix:
1 2 3
4 5 6
7 8 9

🔄 Step 1: Transpose the matrix

Transposing means swapping matrix[i][j] with matrix[j][i] for j > i.

i	j	matrix[i][j]	matrix[j][i]	Action
0	1	2	4	Swap → 2 ↔ 4
0	2	3	7	Swap → 3 ↔ 7
1	2	6	8	Swap → 6 ↔ 8

🔁 After transpose:

1 4 7
2 5 8
3 6 9

🔄 Step 2: Reverse each row

Reverse each row of the transposed matrix:

Row Before	Row After
1 4 7	7 4 1
2 5 8	8 5 2
3 6 9	9 6 3

✅ Final Output:

Rotated matrix:
7 4 1
8 5 2
9 6 3

```
Original matrix:
1 2 3
4 5 6
7 8 9
Rotated matrix:
7 4 1
8 5 2
9 6 3
```

Running Sum in C++

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

vector<int> runningSum(vector<int>& nums) {
    int n = nums.size();
    vector<int> pre(n);
    pre[0] = nums[0];
    for (int i = 1; i < n; i++) {
        pre[i] = pre[i - 1] + nums[i];
    }
    return pre;
}

int main() {
    vector<int> arr = {1, 2, 3, 4};
    vector<int> res = runningSum(arr);

    for (int i = 0; i < res.size(); i++) {
        cout << res[i] << endl;
    }

    return 0;
}
```

Input:

```
vector<int> arr = {1, 2, 3, 4};
```

📋 Dry Run Table:

i	nums[i]	pre[i - 1]	pre[i] = pre[i - 1] + nums[i]	pre vector after iteration
0	1	-	pre[0] = 1	[1, _, _, _]
1	2	1	pre[1] = 1 + 2 = 3	[1, 3, _, _]
2	3	3	pre[2] = 3 + 3 = 6	[1, 3, 6, _]
3	4	6	pre[3] = 6 + 4 = 10	[1, 3, 6, 10]

✅ Final Output (printed one per line):

```
1
3
6
10
```

```
1
3
6
10
```