Check number exists in array in C++

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

int array11(int nums[], int index, int length) {
    if (index >= length) {
        return 0;
    }
    int small = array11(nums, index + 1, length);
    if (nums[index] == 11) {
        return 1 + small;
    } else {
        return small;
    }
}

int main() {
    int arr[] = {1, 11, 3, 11, 11, 11};
    int length = sizeof(arr) / sizeof(arr[0]);
    cout << array11(arr, 0, length) << endl;
    return 0;
}
```

Input

```
arr = {1, 11, 3, 11, 11, 11}
```

Function Call Tree

```
array11(arr, 0, 6)
→ nums[0] == 1 → skip
→ array11(arr, 1, 6)
  → nums[1] == 11 → count +1
  → array11(arr, 2, 6)
    → nums[2] == 3 → skip
    → array11(arr, 3, 6)
      → nums[3] == 11 → count +1
      → array11(arr, 4, 6)
        → nums[4] == 11 → count +1
        → array11(arr, 5, 6)
          → nums[5] == 11 → count +1
          → array11(arr, 6, 6)
            → index >= length → return 0
```

Dry Run Table

Call	`index`	`nums[index]`	Matches 11?	Return Value
`array11(arr, 0, 6)`	0	1	✗	0 + 4 = 4
`array11(arr, 1, 6)`	1	11	✓	1 + 3 = 4
`array11(arr, 2, 6)`	2	3	✗	0 + 3 = 3
`array11(arr, 3, 6)`	3	11	✓	1 + 2 = 3
`array11(arr, 4, 6)`	4	11	✓	1 + 1 = 2
`array11(arr, 5, 6)`	5	11	✓	1 + 0 = 1
`array11(arr, 6, 6)`	6	N/A	N/A	0

Output

```
4
```

Output:-

```
4
```

Check Palindrome in C++

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

bool isStringPalindrome(const string& input, int s, int e) {
    // Base case: if start index equals end index, the string is a palindrome
    if (s == e) {
        return true;
    }
    // If the characters at the start and end do not match, it's not a palindrome
    if (input[s] != input[e]) {
        return false;
    }
    // If there are more characters to compare, call the function recursively
    if (s < e + 1) {
        return isStringPalindrome(input, s + 1, e - 1);
    }
    return true;
}

bool isStringPalindrome(const string& input) {
    int s = 0;
    int e = input.length() - 1;
    return isStringPalindrome(input, s, e);
}

int main() {
    cout << (isStringPalindrome("abba") ? "true" : "false") << endl;
    return 0;
}
```

Input

```
string = "abba"
```

🔍 Function Call Tree

```
isStringPalindrome("abba", 0, 3)
→ 'a' == 'a' ✓
→ isStringPalindrome("abba", 1, 2)
   → 'b' == 'b' ✓
   → isStringPalindrome("abba", 2, 1)
      → s > e → return true
```

📋 Dry Run Table

Call	s	e	input[s]	input[e]	Match?	Return
isStringPalindrome ("abba", 0, 3)	0	3	'a'	'a'	✓	✓
isStringPalindrome ("abba", 1, 2)	1	2	'b'	'b'	✓	✓
isStringPalindrome ("abba", 2, 1)	2	1	N/A	N/A	Base	✓

🟢 Output

```
true
```

Your program will print:

```
true
```

Output:-
true

Check sorted in C++

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

bool sorted(int arr[], int n) {
    if (n == 1 || n == 0) {
        return true;
    } else if (arr[n - 1] < arr[n - 2]) {
        return false;
    } else {
        return sorted(arr, n - 1);
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << boolalpha << sorted(arr, n) << endl;
    return 0;
}
```

Input

```
arr[] = {1, 2, 3, 4, 5}
n = 5
```

🔄 Recursive Calls

We check if the last two elements are in correct order (`arr[n-2] <= arr[n-1]`), and recursively reduce the array size.

📋 Dry Run Table

Call	n	arr[n-2]	arr[n-1]	Comparison	Result
`sorted(arr, 5)`	5	4	5	4 ≤ 5	✅
`sorted(arr, 4)`	4	3	4	3 ≤ 4	✅
`sorted(arr, 3)`	3	2	3	2 ≤ 3	✅
`sorted(arr, 2)`	2	1	2	1 ≤ 2	✅
`sorted(arr, 1)`	1	—	—	Base case	✅

✅ Output

```
true
```

Your program will print:

```
true
```

Output:-
true

Count zeroes in C++

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

int cnt = 0;

int countZerosRec(int input) {
    // Base case for initial input of 0
    if (input == 0 && cnt == 0) {
        return 1;
    }

    // Base case for recursion
    if (input == 0) {
        return cnt;
    }

    // Check if the current last digit is zero
    if (input % 10 == 0) {
        cnt++;
    }

    // Recursive call to process the next digit
    return countZerosRec(input / 10);
}

int main() {
    cout << countZerosRec(10034) << endl;
    return 0;
}
```

Dry Run for countZerosRec(10034)

Call	input	input % 10	is zero?	sum
countZerosRec(10034)	10034	4	✗	0 + next
countZerosRec(1003)	1003	3	✗	0 + next
countZerosRec(100)	100	0	✓	1 + next
countZerosRec(10)	10	0	✓	1 + next
countZerosRec(1)	1	-	✗	0

➞ Total = 1 + 1 = **2**

Output:-
2

Factorial in C++

```
#include <iostream>

using namespace std;

int fact(int n) {
    if (n == 0) {
        return 1;
    } else {
        int prev = fact(n - 1);
        return n * prev;
    }
}

int main() {
    cout << fact(6) << endl;
    return 0;
}
```

Dry Run Table for fact(6):

Call Level	n	Recursive Call	Returned Value	Computation
1	6	6 * fact(5)	720	6 * 120
2	5	5 * fact(4)	120	5 * 24
3	4	4 * fact(3)	24	4 * 6
4	3	3 * fact(2)	6	3 * 2
5	2	2 * fact(1)	2	2 * 1
6	1	1 * fact(0)	1	1 * 1
7 (Base)	0	return 1	1	Base case hit

🔼 Final Output:

720

Output:-
720

Min-Max in C++

```
#include <iostream>
#include <climits> // for INT_MAX and INT_MIN
using namespace std;

int getMin(int arr[], int i, int n) {
    if (n == 1) {
        return arr[i];
    } else {
        return min(arr[i], getMin(arr, i + 1, n - 1));
    }
}

int getMax(int arr[], int i, int n) {
    if (n == 1) {
        return arr[i];
    } else {
        return max(arr[i], getMax(arr, i + 1, n - 1));
    }
}

int main() {
    int arr[] = {12, 8, 45, 67, 9};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Minimum element of array: " <<
getMin(arr, 0, n) << endl;
    cout << "Maximum element of array: " <<
getMax(arr, 0, n) << endl;
    return 0;
}
```

Dry Run Table for getMin(arr, 0, 5)

Call Level	i	arr[i]	Recursive Call	Returned Value	Computation
1	0	12	min(12, getMin(1, 4))	8	min(12, 8)
2	1	8	min(8, getMin(2, 3))	8	min(8, 9)
3	2	45	min(45, getMin(3, 2))	9	min(45, 9)
4	3	67	min(67, getMin(4, 1))	9	min(67, 9)
5 (base)	4	9	return arr[4]	9	Base case

Dry Run Table for getMax(arr, 0, 5)

Call Level	i	arr[i]	Recursive Call	Returned Value	Computation
1	0	12	max(12, getMax(1, 4))	67	max(12, 67)
2	1	8	max(8, getMax(2, 3))	67	max(8, 67)
3	2	45	max(45, getMax(3, 2))	67	max(45, 67)
4	3	67	max(67, getMax(4, 1))	67	max(67, 9)
5 (base)	4	9	return arr[4]	9	Base case

✔ Final Output:

Minimum element of array: 8
Maximum element of array: 67

Output:-

Minimum element of array: 8
Maximum element of array: 67

Stair Case in C++

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

// Function to calculate number of ways to reach nth step
int staircase(int n) {
    // Base cases
    if (n == 0 || n == 1) {
        return 1;
    }
    if (n == 2) {
        return 2;
    }
    // Recursive case
    return staircase(n-1) + staircase(n-2) + staircase(n-3);
}

int main() {
    // Test case
    int n = 7;
    cout << staircase(n) << endl;
    return 0;
}
```

Dry Run Table for `staircase(7)`

Track the **calls** and their **return values** from the bottom up (memoized-style for understanding):

`n`	`staircase(n)` Calculation	Result
0	1 (base case)	1
1	1 (base case)	1
2	2 (base case)	2
3	staircase(2) + staircase(1) + staircase(0)	2 + 1 + 1 = 4
4	staircase(3) + staircase(2) + staircase(1)	4 + 2 + 1 = 7
5	staircase(4) + staircase(3) + staircase(2)	7 + 4 + 2 = 13
6	staircase(5) + staircase(4) + staircase(3)	13 + 7 + 4 = 24
7	staircase(6) + staircase(5) + staircase(4)	24 + 13 + 7 = 44

✅ Final Output:

```
44
```

Output:-
44

Subset Sum in C++

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

// Function to calculate subset sums recursively
void subsetSums(int arr[], int l, int r, int sum) {
    // Base case: if l exceeds r, print the current sum
    if (l > r) {
        cout << sum << " ";
        return;
    }

    // Recursive case: include current element arr[l] in the subset sum
    subsetSums(arr, l + 1, r, sum + arr[l]);
}

int main() {
    // Initialize the array and its length
    int arr[] = {5, 4, 3, 5, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Call the function to calculate subset sums, starting with l=0, r=n-1, and initial sum=0
    subsetSums(arr, 0, n - 1, 0);

    return 0;
}
```

Input:

```
int arr[] = {5, 4, 3, 5, 4};
```

This adds:

```
5 + 4 + 3 + 5 + 4 = 21
```

And when `l > r`, it prints `sum`, which is `21`.

Dry Run Table (for your input):

Step	l	r	sum	Action
1	0	4	0	`sum = 0 + arr[0] = 5`
2	1	4	5	`sum = 5 + arr[1] = 9`
3	2	4	9	`sum = 9 + arr[2] = 12`
4	3	4	12	`sum = 12 + arr[3] = 17`
5	4	4	17	`sum = 17 + arr[4] = 21`
6	5	4	21	`l > r`, print `21` and return

Final Output:

```
21
```

Output:-
21

Tiling in C++

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

int tilingways(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return tilingways(n - 1) + tilingways(n - 2);
}

int main() {
    cout << tilingways(4) << endl;
    return 0;
}
```

Function Call	Returns	Reason
tilingways(4)	?	tilingways(3) + tilingways(2)
tilingways(3)	?	tilingways(2) + tilingways(1)
tilingways(2)	?	tilingways(1) + tilingways(0)
tilingways(1)	1	Base case
tilingways(0)	0 ✗	Wrong base case — it **should be 1**
tilingways(2)	1 + 0 = 1	
tilingways(1)	1	Base case
tilingways(3)	1 + 1 = 2	
tilingways(2)	1	Already computed
tilingways(4)	2 + 1 = 3 ✓	

Output:-
3