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;``` ```}```	**Initial Call:** array11(arr, 0, 6) • **Condition:** index = 0, length = 6 → index < length is true. • **Value at nums[0]:** 1 (not equal to 11). • **Recursive Call:** array11(arr, 1, 6) **Second Call:** array11(arr, 1, 6) • **Condition:** index = 1, length = 6 → index < length is true. • **Value at nums[1]:** 11 (equal to 11). • **Recursive Call:** array11(arr, 2, 6) **Third Call:** array11(arr, 2, 6) • **Condition:** index = 2, length = 6 → index < length is true. • **Value at nums[2]:** 3 (not equal to 11). • **Recursive Call:** array11(arr, 3, 6) **Fourth Call:** array11(arr, 3, 6) • **Condition:** index = 3, length = 6 → index < length is true. • **Value at nums[3]:** 11 (equal to 11). • **Recursive Call:** array11(arr, 4, 6) **Fifth Call:** array11(arr, 4, 6) • **Condition:** index = 4, length = 6 → index < length is true. • **Value at nums[4]:** 11 (equal to 11). • **Recursive Call:**

array11(arr, 5, 6)

Sixth Call:

array11(arr, 5, 6)

- **Condition:** index = 5, length = 6 → index < length is true.
- **Value at nums[5]:** 11 (equal to 11).
- **Recursive Call:**

  array11(arr, 6, 6)

Base Case (Seventh Call):

array11(arr, 6, 6)

- **Condition:** index = 6, length = 6 → index >= length is true.
- **Action:** Return 0.

Backtracking and Return Values:

1. **Sixth Call:**
   - **Value at nums[5]:** 11 → Return 1 + 0 = 1.
2. **Fifth Call:**
   - **Value at nums[4]:** 11 → Return 1 + 1 = 2.
3. **Fourth Call:**
   - **Value at nums[3]:** 11 → Return 1 + 2 = 3.
4. **Third Call:**
   - **Value at nums[2]:** 3 → Return 0 + 3 = 3.
5. **Second Call:**
   - **Value at nums[1]:** 11 → Return 1 + 3 = 4.
6. **Initial Call:**
   - **Value at nums[0]:** 1 → Return 0 + 4 = 4.

Output:-
4

Check Palindrome in C++

```cpp
#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;
}
```

Step-by-Step Function Call Flow:

1. **Initial Call:**
   isStringPalindrome("abba", 0, 3)
   - s = 0, e = 3
   - input[s] = 'a' and input[e] = 'a' → They match → Continue with the next call:

   isStringPalindrome("abba", 1, 2)

2. **Second Call:**
   isStringPalindrome("abba", 1, 2)
   - s = 1, e = 2
   - input[s] = 'b' and input[e] = 'b' → They match → Continue with the next call:

   isStringPalindrome("abba", 2, 1)

3. **Third Call (Base Case):**
   isStringPalindrome("abba", 2, 1)
   - s = 2, e = 1
   - Since s < e + 1 condition fails (2 > 1), the function returns true.
4. **Backtracking:**
   The result true propagates back through all the recursive calls:
   - isStringPalindrome("abba", 1, 2) → true
   - isStringPalindrome("abba", 0, 3) → true

Output:-
true

Check sorted in C++

```cpp
#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;
}
```

Recursive Function Call Flow:

1. **Initial Call:**
   sorted(arr, 5)
   - arr[4] = 5 and arr[3] = 4 → 5 >= 4 → Continue checking with n = 4.
2. **Second Call:**
   sorted(arr, 4)
   - arr[3] = 4 and arr[2] = 3 → 4 >= 3 → Continue checking with n = 3.
3. **Third Call:**
   sorted(arr, 3)
   - arr[2] = 3 and arr[1] = 2 → 3 >= 2 → Continue checking with n = 2.
4. **Fourth Call:**
   sorted(arr, 2)
   - arr[1] = 2 and arr[0] = 1 → 2 >= 1 → Continue checking with n = 1.
5. **Base Case:**
   sorted(arr, 1)
   - n == 1 → Return true.

Backtracking:

- The base case returns true and propagates this result through all the previous recursive calls:
  - sorted(arr, 2) → true
  - sorted(arr, 3) → true
  - sorted(arr, 4) → true
  - sorted(arr, 5) → true

Output:-
true

Count zeroes in C++

```cpp
#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 of the Function

Input:

input = 10034

Step-by-Step Execution

Initial Call:

countZerosRec(10034)

- input % 10 = 4 (last digit is not 0).
- Recursive call:

  countZerosRec(1003)

Second Call:

countZerosRec(1003)

- input % 10 = 3 (last digit is not 0).
- Recursive call:

  countZerosRec(100)

Third Call:

countZerosRec(100)

- input % 10 = 0 (last digit **is** 0).
- cnt++ → cnt = 1.
- Recursive call:

  countZerosRec(10)

Fourth Call:

countZerosRec(10)

- input % 10 = 0 (last digit **is** 0).
- cnt++ → cnt = 2.
- Recursive call:

  countZerosRec(1)

Fifth Call:

countZerosRec(1)

- input % 10 = 1 (last digit is not 0).
- Recursive call:

  countZerosRec(0)

Base Case (Sixth Call):

countZerosRec(0)

- input == 0, so return cnt.
- cnt = 2.

Backtracking and Final Output:

- **Fifth Call:** Returns 2.
- **Fourth Call:** Returns 2.
- **Third Call:** Returns 2.
- **Second Call:** Returns 2.
- **Initial Call:** Returns 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;
}
```

Step 1: Initial Call

- Function: fact(6)
- Input: n = 6
- Condition: n != 0 → **Not base case**
- Action: Call fact(5) and calculate 6 * fact(5)

Step 2: Call fact(5)

- Function: fact(5)
- Input: n = 5
- Condition: n != 0 → **Not base case**
- Action: Call fact(4) and calculate 5 * fact(4)

Step 3: Call fact(4)

- Function: fact(4)
- Input: n = 4
- Condition: n != 0 → **Not base case**
- Action: Call fact(3) and calculate 4 * fact(3)

Step 4: Call fact(3)

- Function: fact(3)
- Input: n = 3
- Condition: n != 0 → **Not base case**
- Action: Call fact(2) and calculate 3 * fact(2)

Step 5: Call fact(2)

- Function: fact(2)
- Input: n = 2
- Condition: n != 0 → **Not base case**
- Action: Call fact(1) and calculate 2 * fact(1)

Step 6: Call fact(1)

- Function: fact(1)
- Input: n = 1
- Condition: n != 0 → **Not base case**
- Action: Call fact(0) and calculate 1 * fact(0)

Step 7: Call fact(0)

- Function: fact(0)

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

Input: n = 0
Condition: n == 0 → Base case
Action: Return 1

Step 8: Return Values

Return to fact(1):

Calculation: 1 * fact(0) → 1 * 1 = 1
Return: 1

Return to fact(2):

Calculation: 2 * fact(1) → 2 * 1 = 2
Return: 2

Return to fact(3):

Calculation: 3 * fact(2) → 3 * 2 = 6
Return: 6

Return to fact(4):

Calculation: 4 * fact(3) → 4 * 6 = 24
Return: 24

Return to fact(5):

Calculation: 5 * fact(4) → 5 * 24 = 120
Return: 120

Return to fact(6):

Calculation: 6 * fact(5) → 6 * 120 = 720
Return: 720

</td>
</tr>
<tr>
<td colspan="2">Output:-
720</td>
</tr>
</table>

Min-Max in C++

```cpp
#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;
}
```

For the input array {12, 8, 45, 67, 9}, the program will execute the following steps:

Finding the Minimum:

1. getMin(arr, 0, 5) (array = {12, 8, 45, 67, 9}):
    - Compare arr[0] (12) with getMin(arr, 1, 4).
2. getMin(arr, 1, 4) (array = {8, 45, 67, 9}):
    - Compare arr[1] (8) with getMin(arr, 2, 3).
3. getMin(arr, 2, 3) (array = {45, 67, 9}):
    - Compare arr[2] (45) with getMin(arr, 3, 2).
4. getMin(arr, 3, 2) (array = {67, 9}):
    - Compare arr[3] (67) with getMin(arr, 4, 1).
5. getMin(arr, 4, 1) (base case, array = {9}):
    - Return arr[4] (9).
6. Now backtrack:
    - getMin(arr, 3, 2) returns min(67, 9) = 9.
    - getMin(arr, 2, 3) returns min(45, 9) = 9.
    - getMin(arr, 1, 4) returns min(8, 9) = 8.
    - getMin(arr, 0, 5) returns min(12, 8) = 8.

    Result: The minimum element is 8.

Finding the Maximum:

1. getMax(arr, 0, 5) (array = {12, 8, 45, 67, 9}):
    - Compare arr[0] (12) with getMax(arr, 1, 4).
2. getMax(arr, 1, 4) (array = {8, 45, 67, 9}):
    - Compare arr[1] (8) with getMax(arr, 2, 3).
3. getMax(arr, 2, 3) (array = {45, 67, 9}):
    - Compare arr[2] (45) with getMax(arr, 3, 2).
4. getMax(arr, 3, 2) (array = {67, 9}):
    - Compare arr[3] (67) with getMax(arr, 4, 1).
5. getMax(arr, 4, 1) (base case, array = {9}):
    - Return arr[4] (9).
6. Now backtrack:
    - getMax(arr, 3, 2) returns max(67, 9) = 67.
    - getMax(arr, 2, 3) returns max(45, 67) = 67.
    - getMax(arr, 1, 4) returns max(8, 67) = 67.
    - getMax(arr, 0, 5) returns max(12,

	67) = 67. **Result**: The maximum element is 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; } ```	**Initial Call** The function staircase(7) is called. • Base cases: ○ If n == 0, return 1 ○ If n == 1, return 1 ○ If n == 2, return 2 The recursive case is staircase(n-1) + staircase(n-2) + staircase(n-3). For n = 7, we call: staircase(7) = staircase(6) + staircase(5) + staircase(4) **Step 1: staircase(6)** • Call: staircase(6) = staircase(5) + staircase(4) + staircase(3) • Let's break it down: **Step 1.1: staircase(5)** • Call: staircase(5) = staircase(4) + staircase(3) + staircase(2) • Let's break it down: Step 1.1.1: staircase(4) • Call: staircase(4) = staircase(3) + staircase(2) + staircase(1) • Let's break it down: Step 1.1.1.1: staircase(3) • Call: staircase(3) = staircase(2) + staircase(1) + staircase(0) • Let's break it down: ○ staircase(2) = 2 ○ staircase(1) = 1 ○ staircase(0) = 1 So, staircase(3) = 2 + 1 + 1 = 4. Step 1.1.1.2: staircase(2) • Base case: staircase(2) = 2 Step 1.1.1.3: staircase(1) • Base case: staircase(1) = 1 So, staircase(4) = 4 + 2 + 1 = 7.

Step 1.2: staircase(3)

- We already calculated that staircase(3) = 4.

Step 1.3: staircase(2)

- Base case: staircase(2) = 2.

So, staircase(5) = 7 + 4 + 2 = 13.

Step 2: staircase(4)

We already calculated that staircase(4) = 7.

Step 3: staircase(3)

We already calculated that staircase(3) = 4.

So, staircase(6) = 13 + 7 + 4 = 24.

Final Calculation: staircase(7)

Now that we have the values for staircase(6), staircase(5), and staircase(4), we can calculate staircase(7):

staircase(7) = 24 + 13 + 7 = 44

Output:-
44

Subset Sum in C++

```cpp
#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;
}
```

Dry Run of subsetSums(arr, 0, 4, 0)

Let's dry run this code using the input array {5, 4, 3, 5, 4}.

Initial Call: subsetSums(arr, 0, 4, 0)

Call 1: subsetSums(arr, 0, 4, 0)

- We include arr[0] which is 5.
  - Next call: subsetSums(arr, 1, 4, 5)

Call 2: subsetSums(arr, 1, 4, 5)

- We include arr[1] which is 4.
  - Next call: subsetSums(arr, 2, 4, 9)

Call 3: subsetSums(arr, 2, 4, 9)

- We include arr[2] which is 3.
  - Next call: subsetSums(arr, 3, 4, 12)

Call 4: subsetSums(arr, 3, 4, 12)

- We include arr[3] which is 5.
  - Next call: subsetSums(arr, 4, 4, 17)

Call 5: subsetSums(arr, 4, 4, 17)

- We include arr[4] which is 4.
  - Next call: subsetSums(arr, 5, 4, 21) — **Base case reached**, prints 21.

Backtracking and Generating Other Subsets

Now, the recursion starts backtracking. The function will explore subsets where elements are **not** included.

Call 6: subsetSums(arr, 4, 4, 17) (skip arr[4])

- We **skip** arr[4] (i.e., do not add it to the subset).

- Next call: subsetSums(arr, 5, 4, 17) — **Base case reached**, prints 17.

Call 7: subsetSums(arr, 3, 4, 12) (skip arr[3])

- We **skip** arr[3] (i.e., do not add it to the subset).
  - Next call: subsetSums(arr, 4, 4, 12)

Call 8: subsetSums(arr, 4, 4, 12) (skip arr[4])

- We **skip** arr[4].
  - Next call: subsetSums(arr, 5, 4, 12) — **Base case reached**, prints 12.

Call 9: subsetSums(arr, 2, 4, 9) (skip arr[2])

- We **skip** arr[2] (i.e., do not add it to the subset).
  - Next call: subsetSums(arr, 3, 4, 9)

Call 10: subsetSums(arr, 3, 4, 9) (skip arr[3])

- We **skip** arr[3] (i.e., do not add it to the subset).
  - Next call: subsetSums(arr, 4, 4, 9)

Call 11: subsetSums(arr, 4, 4, 9) (skip arr[4])

- We **skip** arr[4].
  - Next call: subsetSums(arr, 5, 4, 9) — **Base case reached**, prints 9.

Call 12: subsetSums(arr, 1, 4, 5) (skip arr[1])

- We **skip** arr[1] (i.e., do not add it to the subset).
  - Next call: subsetSums(arr, 2, 4, 5)

Call 13: subsetSums(arr, 2, 4, 5) (skip arr[2])

<table>
<tr><td></td><td>
We skip arr[2].
Next call: subsetSums(arr, 3, 4, 5)
Call 14: subsetSums(arr, 3, 4, 5) (skip arr[3])
We skip arr[3].
Next call: subsetSums(arr, 4, 4, 5)
Call 15: subsetSums(arr, 4, 4, 5) (skip arr[4])
We skip arr[4].
Next call: subsetSums(arr, 5, 4, 5) — Base case reached, prints 5.
Final Output
The program will print the subset sums of the array {5, 4, 3, 5, 4}:
21 17 12 9 5
</td></tr>
<tr><td colspan="2">Output:-
21</td></tr>
</table>

Tiling in C++

```cpp
#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;
}
```

Step-by-step Calculation

1. tilingways(4):
   - tilingways(3) + tilingways(2)
2. **Recursive call**: tilingways(3):
   - tilingways(2) + tilingways(1)
3. **Recursive call**: tilingways(2):
   - tilingways(1) + tilingways(0)
4. **Base case reached**: tilingways(1) returns 1 (since there is 1 way to tile a 2x1 grid).
   - **Base case reached**: tilingways(0) returns 0 (no way to tile a 2x0 grid).
   - Result: tilingways(2) = 1 + 0 = 1
5. **Base case reached**: tilingways(1) returns 1.
   - Result: tilingways(3) = 1 + 1 = 2
6. **Recursive call**: tilingways(2):
   - tilingways(1) + tilingways(0)
   - tilingways(1) returns 1, tilingways(0) returns 0.
   - Result: tilingways(2) = 1 + 0 = 1
7. **Final Calculation**: tilingways(4) = 2 + 1 = 3

Output:-
3