**Problem Statement**: Determine whether a given integer is even or odd.

**Approach**: The code uses a bitwise operation to check the least significant bit (LSB) of the integer.

**Steps**:

1. Accept an integer 'a' as input.
2. Use a bitwise AND operation between 'a' and 1 (**a & 1**).
3. If the result of the bitwise operation is 1, it indicates that the LSB is 1, which means 'a' is an odd number.
4. If the result is not 1, it means 'a' is an even number.
5. Print the result accordingly.

**Code**

void evenOdd(int a)

{

    if (a & 1 == 1)

        cout << "The given number " << a << " is Odd. " << endl;

    else

        cout << "The given number " << a << " is Even. " << endl;

}

The provided C++ code defines a function `swap` that exchanges the values of two integer variables `a` and `b` without using a temporary variable. It uses bitwise XOR (^) to perform the swap operation. Here's an explanation of the code:

\*\*Approach\*\*:

The code leverages the XOR (^) operator to swap the values of `a` and `b` without using a temporary variable. XOR is used to manipulate the bits of `a` and `b` in a way that the original values can be reconstructed.

\*\*Steps\*\*:

1. The first XOR operation `a = a ^ b` combines the bits of `a` and `b` into `a`.

2. The second XOR operation `b = a ^ b` combines the bits of the updated `a` with the original bits of `b`, effectively putting the original value of `a` into `b`.

3. The final XOR operation `a = a ^ b` combines the bits of the updated `a` with the original bits of `b`, effectively putting the original value of `b` into `a`.

void swap(int &a, int &b)

{

    a = a ^ b;

    b = a ^ b;

    a = a ^ b;

}

The provided C++ code defines a function `findBit` that determines the value of a specific bit at a given position in an integer `a`. It uses bitwise operations and bit shifting to check the bit at the specified position. Here's an explanation of the code:

\*\*Approach\*\*:

The code uses a bitwise right shift operation (`>>`) to move the bit of interest to the least significant position (bit 0). It then applies a bitwise AND operation to isolate the value of that bit (1 or 0).

\*\*Steps\*\*:

1. The input integer `a` and the position of the bit to be checked, `pos`, are given as arguments to the function.

2. A bitmask is created by right-shifting `a` by `pos` positions, effectively moving the bit of interest to the least significant position (`int bitmask = a >> pos`).

3. The least significant bit of the bitmask (`bitmask & 1`) is checked using a bitwise AND operation.

4. If the least significant bit is 1, it indicates that the bit at position `pos` in `a` is 1. Otherwise, it's 0.

void findBit(int a, int pos)

{

    int bitmask = a >> pos;

    if ((bitmask & 1) == 1)

        cout << "The " << pos << " bit of " << a << " is 1." << endl;

    else

        cout << "The " << pos << " bit of " << a << " is 0." << endl;

}

The provided C++ code defines a function `setbit` that sets a specific bit at a given position in an integer `a`. It uses bitwise operations to manipulate the specified bit. Here's an explanation of the code:

\*\*Approach\*\*:

The code uses bitwise operations to set the bit at the given position `pos` to 1 while leaving the other bits in `a` unchanged.

\*\*Steps\*\*:

1. The input integer `a` and the position of the bit to be set, `pos`, are given as arguments to the function.

2. A bitmask is created by left-shifting the value 1 by `pos` positions, creating a bit with a 1 at the desired position (`int masking = 1 << pos`).

3. The bitwise OR operation is used to combine the bitmask with the original value `a`. This operation sets the bit at position `pos` to 1 while leaving other bits unchanged (`masking = masking | a`).

4. The result, stored in `masking`, represents the integer with the specified bit set to 1.

5. The function then prints the modified value of `masking`, indicating that the bit at position `pos` has been set to 1.

void setbit(int a, int pos)

{

    int masking = 1 << pos;

    masking = masking | a;

    cout << "The " << pos << " bit set of " << masking << "." << endl;

}

The provided C++ code defines a function `clearBit` that clears a specific bit at a given position in an integer `a`. It uses bitwise operations to manipulate the specified bit. Here's an explanation of the code:

\*\*Approach\*\*:

The code uses bitwise operations to clear (set to 0) the bit at the given position `pos` while leaving the other bits in `a` unchanged.

\*\*Steps\*\*:

1. The input integer `a` and the position of the bit to be cleared, `pos`, are given as arguments to the function.

2. A bitmask is created by left-shifting the value 1 by `pos` positions, creating a bit with a 1 at the desired position (`int masking = 1 << pos`).

3. The bitwise XOR operation is used to toggle the bit at position `pos` from 1 to 0 or from 0 to 1 in the bitmask. This operation clears the specified bit while leaving other bits unchanged (`masking = masking ^ a`).

4. The result, stored in `masking`, represents the integer with the specified bit cleared (set to 0).

5. The function then prints the modified value of `masking`, indicating that the bit at position `pos` has been cleared.

void clearBit(int a, int pos)

{

    int masking = 1 << pos;

    masking = masking ^ a;

    cout << "The " << pos << " bit clear of " << masking << "." << endl;

}

The provided C++ code defines a function `changeBitSetClear` that toggles (sets to 1 if it's 0, and clears to 0 if it's 1) a specific bit at a given position in an integer `a`. It uses a combination of bitwise operations and calls to the `setbit` and `clearBit` functions. Here's an explanation of the code:

\*\*Approach\*\*:

The code uses bitwise operations and function calls to toggle the bit at the given position `pos` while leaving the other bits in `a` unchanged.

\*\*Steps\*\*:

1. The input integer `a` and the position of the bit to be toggled, `pos`, are given as arguments to the function.

2. A bitmask is created by right-shifting `a` by `pos` positions, effectively moving the bit of interest to the least significant position (`int masking = a >> pos`).

3. The least significant bit of the bitmask (`masking & 1`) is checked using a bitwise AND operation.

4. If the least significant bit is 1, it indicates that the bit at position `pos` in `a` is 1, and the `clearBit` function is called to clear it.

5. If the least significant bit is 0, it indicates that the bit at position `pos` in `a` is 0, and the `setbit` function is called to set it to 1.

void changeBitSetClear(int a, int pos)

{

    int masking = a >> pos;

    if ((masking & 1) == 1)

        clearBit(a, pos);

    else

        setbit(a, pos);

}

The provided C++ code defines a function `totalSetBit` that counts the number of set (1) bits in an integer `a` by iteratively checking each bit. It uses bitwise operations to perform this count. Here's an explanation of the code:

\*\*Approach\*\*:

The code uses a `while` loop to iterate through each bit of the input integer `a`. It shifts the bits to the right one at a time and checks whether the least significant bit is set (1) using bitwise AND (`&`) operation. If it's set, it increments the count.

\*\*Steps\*\*:

1. The input integer `a` is given as an argument to the function.

2. A bitmask `masking` is initialized with the value of `a`.

3. A variable `cont` (short for "count") is initialized to 0 to keep track of the number of set bits.

4. The code enters a `while` loop that continues as long as `masking` is not equal to 0.

5. Within the loop, the least significant bit of `masking` (`masking & 1`) is checked using a bitwise AND operation. If it's set (equal to 1), the `cont` variable is incremented.

6. After checking the least significant bit, the `masking` value is right-shifted by 1 position (`masking = masking >> 1`).

7. The loop continues to the next bit until all bits of `a` have been checked.

8. Once the loop exits, the function prints the count of set bits and returns this count as the result.

int totalSetBit(int a)

{

    int masking = a;

    int cont = 0;

    while (masking != 0)

    {

        if (masking & 1 == 1)

        {

            cont++;

        }

        masking = masking >> 1;

    }

    cout << "The set bit of " << a << " is " << cont << ".\n";

    return cont;

}

The provided C++ code defines a function `formula` that calculates the number of set bits (1s) in the binary representation of an integer `n` using a bit manipulation technique. Here's an explanation of the code:

\*\*Approach\*\*:

The code uses bitwise operations to clear (set to 0) the least significant set bit in the integer `n` repeatedly. It counts how many times it needs to clear a bit before `n` becomes 0. This count represents the number of set bits in the binary representation of `n`.

\*\*Steps\*\*:

1. The input integer `n` is given as an argument to the function.

2. Two variables are initialized: `cont` to 0 (to count the set bits) and `masking` to the value of `n`.

3. The function enters a `while` loop that continues as long as `masking` is greater than or equal to 1.

4. Within the loop, the least significant set bit of `masking` is counted by incrementing `cont`.

5. The same least significant bit is then cleared (set to 0) in `masking` using a bitwise AND operation with `(masking - 1)`. This operation removes the least significant set bit.

6. The loop continues until all set bits in `n` have been counted and cleared.

7. The function returns the count `cont`, which represents the number of set bits in the binary representation of `n`.

int formula(int n)

{

    int cont = 0;

    int masking = n;

    while (masking >= 1)

    {

        cont++;

        masking = masking & (masking - 1);

    }

    return cont;

}

Finding the Non-Repeated Element in an Array Using Bitwise XOR

\*\*Approach\*\*:

- This algorithm utilizes the properties of bitwise XOR (^) to efficiently find the non-repeated element in an array. XOR is chosen because it cancels out duplicate elements, leaving only the unique element.

\*\*Example in Mathematics\*\*:

- Consider a set of positive integers: {3, 5, 3, 7, 5}.

- We can find the non-repeated element using XOR:

- 3 ^ 5 ^ 3 ^ 7 ^ 5 = 7.

- The non-repeated element in this set is 7.

\*\*Steps\*\*:

1. Initialize a variable `result` to 0. This variable will store the result.

2. Determine the size of the input array and store it in `n`.

3. Iterate through the array using a loop with a variable `i` ranging from 0 to `n-1`.

4. Within the loop, apply the XOR operation between `result` and the current element in the array (`arr[i]`) and assign the result back to `result`.

- `result = result ^ arr[i]`.

- This step cancels out duplicate elements.

5. After the loop, the `result` will contain the non-repeated element.

6. Return `result` as the non-repeated element.

int findNonRepeated(vector<int> arr)

{

    int result = 0;

    int n = arr.size();

    for (int i = 0; i < n; i++)

    {

        result = result ^ arr[i];

        cout << i << " , ";

    }

    cout << endl;

    return result;

}

\*\*Title\*\*: "Finding Two Non-Repeated Elements in an Array Using Bitwise XOR"

\*\*Approach\*\*:

- This algorithm finds two non-repeated elements in an array using bitwise XOR. It assumes that all other elements in the array occur twice except for the two unique elements.

- The algorithm first calculates the XOR of all elements in the array, which results in a value that represents the XOR of the two unique elements.

- It then determines the rightmost set bit in this XOR result to split the elements into two groups.

- Finally, it applies XOR operations to find the two unique elements in each group.

\*\*Example in Mathematics\*\*:

- Consider an array: {3, 5, 3, 7, 5}.

- Calculate the XOR of all elements:

- 3 ^ 5 ^ 3 ^ 7 ^ 5 = 7.

- Find the rightmost set bit in 7 (binary: 0111) at position 1.

- Separate the elements into two groups based on the rightmost set bit:

- Group A (bit at position 1 is 0): {5, 5} with XOR result 5.

- Group B (bit at position 1 is 1): {3, 3, 7} with XOR result 7.

- The two unique elements are 5 and 7.

\*\*Steps\*\*:

1. Initialize `allXor` to 0. This variable will store the XOR result of all elements in the array.

2. Iterate through the array using a loop and calculate the XOR of all elements by applying the XOR operation between `allXor` and the current element.

3. Determine the rightmost set bit in `allXor` and find its position.

4. Initialize `a` as the same value as `allXor`.

5. Iterate through the array again. For each element, check whether the bit at the determined position is set. If it is set, apply XOR with `a`.

6. Calculate `b` as the XOR of `allXor` and `a`.

7. Print the values of `a` and `b`, which are the two non-repeated elements.

void findTwoNonRepeated(vector<int> arr, int n)

{

    int allXor = 0;

    for (int i = 0; i < n; i++)

    {

        allXor = allXor ^ arr[i];

    }

    int pos = 0;

    int temp = allXor;

    while (!(temp & 1))

    {

        temp = temp >> 1;

        pos++;

    }

    int a = allXor;

    for (int i = 0; i < n; i++)

    {

        if (arr[i] >> pos == 1)

            a = a ^ arr[i];

    }

    int b = allXor ^ a;

    cout << a << " " << b;

}

\*\*Title\*\*: "Calculate Factorial of an Integer (with Constraint)"

\*\*Approach\*\*:

- This function calculates the factorial of an integer `n` with a constraint to handle values of `n` greater than or equal to 13. For `n >= 13`, the function returns 0, as the result exceeds the range of a 32-bit integer.

- For values of `n` that are within the allowed range, the function uses a loop to calculate the factorial.

\*\*Examples in Mathematics\*\*:

- The factorial of a non-negative integer `n` (denoted as `n!`) is the product of all positive integers from 1 to `n`.

- For example, 5! = 5 x 4 x 3 x 2 x 1 = 120.

\*\*Steps\*\*:

1. Check if `n` is greater than or equal to 13. If so, return 0 due to the constraint.

2. Initialize a variable `res` to 1 to store the result of the factorial.

3. Start a loop with `i` initialized to `n`.

4. In each iteration of the loop, multiply the current value of `res` by `i`.

5. Decrement `i` by 1 in each iteration.

6. Continue the loop as long as `i` is greater than 0 and not equal to 0.

7. After the loop, return the calculated `res` as the factorial of `n`.

int factorial(int n)

{

    if (n >= 13)

        return 0;

    int res = 1;

    for (int i = n; i > 0 && i != 0; i--)

    {

        res \*= i;

    }

    return res;

}

\*\*Title\*\*: "Count Trailing Zeros in the Factorial of an Integer"

\*\*Approach\*\*:

- This function calculates the number of trailing zeros in the factorial of an integer `n` by counting the occurrences of factors of 5.

- Trailing zeros result from pairs of 2 and 5 in the factorial. Since factors of 2 are abundant, the algorithm focuses on counting the factors of 5.

\*\*Examples in Mathematics\*\*:

- To count the trailing zeros in 10! (10 factorial), we need to find the number of factors of 5 in the product of all positive integers from 1 to 10.

- The factors of 5 are 5, 10, and 15. Therefore, the number of trailing zeros in 10! is 2.

\*\*Steps\*\*:

1. Initialize a variable `result` to 0 to store the count of trailing zeros.

2. Start a loop with `i` initialized to 5, which represents the first factor of 5.

3. In each iteration of the loop, calculate how many multiples of `i` (5, 25, 125, etc.) are present in the range from 5 to `n`.

4. Add this count to the `result`.

5. Multiply `i` by 5 to consider the next power of 5 (25, 125, 625, etc.).

6. Continue the loop until `i` exceeds `n`.

7. Return the calculated `result` as the count of trailing zeros in the factorial of `n`.

int trailingZerosFactorial(int n)

{

    int result = 0;

    for (int i = 5; i <= n; i \*= 5)

    {

        result += n / i;

    }

    return result;

}

\*\*Title\*\*: "Count Trailing Zeros in the Factorial of an Integer"

\*\*Approach\*\*:

- This function calculates the number of trailing zeros in the factorial of an integer `n` by counting the occurrences of factors of 5.

- Trailing zeros result from pairs of 2 and 5 in the factorial. Since factors of 2 are abundant, the algorithm focuses on counting the factors of 5.

\*\*Examples in Mathematics\*\*:

- To count the trailing zeros in 10! (10 factorial), we need to find the number of factors of 5 in the product of all positive integers from 1 to 10.

- The factors of 5 are 5, 10, and 15. Therefore, the number of trailing zeros in 10! is 2.

\*\*Steps\*\*:

1. Initialize a variable `result` to 0 to store the count of trailing zeros.

2. Start a loop with `i` initialized to 5, which represents the first factor of 5.

3. In each iteration of the loop, calculate how many multiples of `i` (5, 25, 125, etc.) are present in the range from 5 to `n`.

4. Add this count to the `result`.

5. Multiply `i` by 5 to consider the next power of 5 (25, 125, 625, etc.).

6. Continue the loop until `i` exceeds `n`.

7. Return the calculated `result` as the count of trailing zeros in the factorial of `n`.

bool isPalindromeNumber(int n)

{

    int temp = n;

    int reversNumber = 0;

    while (temp > 0)

    {

        if (reversNumber == 0)

            reversNumber = (temp % 10);

        else

            reversNumber = (reversNumber \* 10) + (temp % 10);

        temp /= 10;

    }

    if (reversNumber == n)

        return true;

    else

        return false;

}

\*\*Title\*\*: "Sieve of Eratosthenes for Prime Number Generation"

\*\*Approach\*\*:

- This function uses the Sieve of Eratosthenes algorithm to generate a list of prime numbers up to a specified integer `n`.

- The Sieve of Eratosthenes is an efficient algorithm for finding all prime numbers up to a given limit. It works by iteratively marking the multiples of each prime number as composite, leaving only the prime numbers unmarked.

\*\*Examples in Mathematics\*\*:

- The Sieve of Eratosthenes can be used to generate prime numbers up to a certain limit, such as finding all prime numbers up to 100.

\*\*Steps\*\*:

1. Create a vector `isPrime` of boolean values, with `n+1` elements, initialized to `true`. These values will indicate whether the corresponding index is prime or not.

2. Set `isPrime[0]` and `isPrime[1]` to `false`, as 0 and 1 are not prime numbers.

3. Start a loop with an integer `i` ranging from 2 to the square root of `n`.

4. Within the loop, start an inner loop with an integer `j` initialized to `2\*i` and incrementing by `i` in each iteration.

5. In the inner loop, mark all multiples of `i` as `false` in the `isPrime` vector to eliminate non-prime numbers.

6. Continue these loops until all multiples of primes up to the square root of `n` have been marked as composite.

7. Return the `isPrime` vector, where `true` values represent prime numbers, and `false` values represent composite numbers.

vector<bool> sieveOfEratoSthenes(int n)

{

    vector<bool> isPrime(n + 1, true);

    isPrime[0] = false;

    isPrime[1] = false;

    for (int i = 2; i \* i <= n; i++)

    {

        for (int j = 2 \* i; j <= n; j += i)

        {

            isPrime[j] = false;

        }

    }

    return isPrime;

}

\*\*Title\*\*: "Calculate Greatest Common Divisor (GCD) Using Recursion"

\*\*Approach\*\*:

- This function calculates the Greatest Common Divisor (GCD) of two integers `a` and `b` using recursion.

- The GCD of two numbers is the largest positive integer that divides both of them without leaving a remainder.

\*\*Examples in Mathematics\*\*:

- For example, the GCD of 8 and 12 is 4 because it is the largest number that divides both 8 and 12 evenly.

\*\*Steps\*\*:

1. The function takes two integer parameters, `a` and `b`.

2. Inside the function, it checks whether `a` is divisible by `b` with no remainder (i.e., `a % b == 0`).

3. If the condition is met, it means that `b` is the GCD, so the function returns `b`.

4. If the condition is not met, the function makes a recursive call to itself with the parameters `b` and `a % b`. This is done to find the GCD of the remaining part of the division, `a % b`.

5. The recursion continues until `a % b` becomes zero, at which point `b` is the GCD, and it is returned.

int gcd(int a, int b)

{

*// GCD and HCF both same*

    return (a % b == 0) ? b : gcd(b, a % b);

}

\*\*Title\*\*: "Fast Power Calculation Using Bit Manipulation and Modulo Arithmetic"

\*\*Approach\*\*:

- This function calculates the power of an integer `a` raised to the exponent `b` using a fast exponentiation technique with modulo arithmetic.

- It employs bitwise operations to optimize the calculation, and it ensures that the result is obtained modulo a large number.

\*\*Examples in Mathematics\*\*:

- Fast exponentiation with modulo arithmetic is used in number theory and cryptography to efficiently calculate powers of numbers and ensure results remain within a specified range.

\*\*Steps\*\*:

1. Initialize a variable `res` to 1. This variable will store the result of `a^b` modulo a large number.

2. Enter a `while` loop that continues as long as `b` is greater than 0.

3. Inside the loop, check if the least significant bit of `b` (i.e., `b & 1`) is set (i.e., not equal to 0). If it is set, multiply `res` by `a` because this bit contributes to the result.

4. Square `a` (i.e., `a = a \* a`) to consider the next power of `a`.

5. Right-shift `b` by one bit (i.e., `b = b >> 1`) to consider the next bit of the exponent.

6. Continue this process until all bits of the exponent `b` have been considered.

7. After the loop, return the calculated `res` as the result of `a^b` modulo a large number.

int fastPower(int a, int b)

{

*// Modulo arithmetics*

    int res = 1;

    while (b > 0)

    {

        if (b & 1 != 0)

            res = res \* a;

        a = a \* a;

        b = b >> 1;

    }

    return res;

}

\*\*Title\*\*: "Fast Power Calculation with Modulo Arithmetic and Bit Manipulation"

\*\*Approach\*\*:

- This function calculates the power of a long integer `a` raised to the exponent `b` while keeping the result within a modulo `n`.

- It uses a fast exponentiation technique with modulo arithmetic and bitwise operations to optimize the calculation.

\*\*Examples in Mathematics\*\*:

- This approach is often used in modular arithmetic and cryptography to efficiently compute large powers while ensuring the result remains within a specified range.

\*\*Steps\*\*:

1. Initialize a long variable `res` to 1. This variable will store the result of `a^b` modulo `n`.

2. Enter a `while` loop that continues as long as `b` is greater than 0.

3. Inside the loop, check if the least significant bit of `b` (i.e., `(b & 1) != 0`) is set (i.e., not equal to 0). If it is set, multiply `res` by `a` and apply modulo `n` to the result.

4. Square `a` and apply modulo `n` (i.e., `a = (a % n \* a % n) % n`) to consider the next power of `a`.

5. Right-shift `b` by one bit (i.e., `b = b >> 1`) to consider the next bit of the exponent.

6. Continue this process until all bits of the exponent `b` have been considered.

7. After the loop, return the calculated `res` as the result of `a^b` modulo `n`.

long fastPower(long a, long b, long n)

{

*// Modulo arithmetics*

    long res = 1;

    while (b > 0)

    {

        if ((b & 1) != 0)

            res = (res \* a % b) % n;

        a = (a % n \* a % n) % n;

        b = b >> 1;

        cout << "hi ";

    }

    return res;

}

\*\*Title\*\*: "Sum of Natural Numbers Using Recursion"

\*\*Solution Approach\*\*:

This code calculates the sum of the first `n` natural numbers using recursion. Here is the approach:

1. \*\*Base Case\*\*: The function defines a base case where if `n` is less than or equal to 0, it returns 0. This is because the sum of natural numbers from 1 to 0 is 0.

2. \*\*Recursion\*\*: For `n` greater than 0, the function calculates the sum of the first `n` natural numbers. It does this by adding `n` to the sum of the first `n-1` natural numbers. This is achieved by making a recursive call to the `sumOfNaturalNo` function with the argument `n-1`.

3. \*\*Generalizing the Relation\*\*: The recursive step generalizes the relation between the sum of the first `n` natural numbers and the sum of the first `n-1` natural numbers. It repeats this process until `n` reaches the base case.

The approach follows the three steps of recursion: defining the base case, finding the relation between the problem and subproblem, and generalizing the relation to solve the problem. This results in an efficient and elegant way to calculate the sum of natural numbers.

\*\*Approach\*\*:

- This function calculates the sum of the first `n` natural numbers using a recursive approach.

- It follows the three steps of recursion: defining the base case, finding the relation between the problem and subproblem, and generalizing the relation.

\*\*Examples in Mathematics\*\*:

- The sum of the first `n` natural numbers is a common mathematical problem that can be expressed as the sum of an arithmetic progression.

\*\*Steps\*\*:

1. The function takes an integer `n` as input.

2. It defines the base case: If `n` is less than or equal to 0, the function returns 0 because the sum of the natural numbers from 1 to 0 is 0.

3. For `n` greater than 0, the function calculates the sum of the first `n` natural numbers using recursion.

4. The recursive step involves adding `n` to the sum of the first `n-1` natural numbers, which is calculated by making a recursive call to the `sumOfNaturalNo` function with `n-1`.

5. The recursion continues until `n` reaches the base case, and the function returns the sum of the natural numbers.

int sumOfNaturalNo(int n)

{

*// Three steps of recursion*

*// 1, Find the base case*

*// 2, Find the relation between the problem and subproblem*

*// 3, Generalize the relation*

    return (n <= 0) ? 0 : n + sumOfNaturalNo(n - 1);

}

\*\*Solution Approach\*\*:

The provided code calculates the number of ways to reach the bottom-right corner of a grid with dimensions `m x n` by only moving right or down. This problem is a classic example of combinatorial mathematics and can be solved using recursion. Here's the approach:

1. \*\*Base Case\*\*: The base case is defined as follows: If the grid has only one row (`m == 1`) or one column (`n == 1`), there is only one way to reach the bottom-right corner, which is to move either right or down from the starting position.

2. \*\*Recursion\*\*: For grids with more than one row and one column, the number of ways to reach the bottom-right corner can be calculated by summing two possibilities:

- Moving one step down and continuing to find the number of ways in the smaller grid (with dimensions `m - 1` x `n`).

- Moving one step right and continuing to find the number of ways in the smaller grid (with dimensions `m` x `n - 1`).

3. \*\*Base Case Generalization\*\*: The recursive approach generalizes this process by repeating the same steps until it reaches the base case, where the number of ways is known.

\*\*Mathematical Proficiency\*\*:

- This problem is based on combinatorial mathematics, particularly on the concept of counting the number of possible paths in a grid.

- It can be related to the binomial coefficient, as the number of ways to reach the bottom-right corner is a combination of the number of ways to reach the corner from the top and from the left.

\*\*Steps\*\*:

1. Check if the base case is met:

- If `m == 1` or `n == 1`, return 1, as there is only one way to reach the corner.

2. For larger grids (`m > 1` and `n > 1`), recursively calculate the number of ways to reach the corner by considering two possibilities:

- Moving down: Call the function with dimensions `m - 1` x `n`.

- Moving right: Call the function with dimensions `m` x `n - 1`.

3. Sum the results of these recursive calls to get the total number of ways to reach the bottom-right corner.

\*\*Example\*\*:

Consider a grid with dimensions `3 x 3`. You want to find the number of ways to reach the bottom-right corner. Using the provided code, you can calculate this as follows:

- Starting at the top-left corner, you can either move down or move right.

- If you move down, you are left with a `2 x 3` grid.

- If you move right, you are left with a `3 x 2` grid.

- You can continue this process until you reach the base case, where there is only one row or one column left.

By summing the possibilities at each step, you can find the total number of ways to reach the bottom-right corner. In this example, for a `3 x 3` grid, there are 6 different paths.

int numberWaysMatrix(int m, int n)

{

    return (m == 1 || n == 1) ? 1 : numberWaysMatrix(m - 1, n) + numberWaysMatrix(m, n - 1);

}

\*\*Solution Approach\*\*:

The provided code calculates the position of the "survivor" in a game known as the Josephus problem, given the total number of people in a circle `n` and the step size `k`. The problem is to find the position of the last person remaining after repeatedly eliminating every `k`-th person. Here's the approach:

1. \*\*Base Case\*\*: The base case is defined as follows: If there is only one person in the circle (`n == 1`), that person is the survivor, and their position is 0 (assuming a 0-based index).

2. \*\*Recursion\*\*: For circles with more than one person (`n > 1`), the position of the survivor can be calculated using recursion. The survivor's position in a circle of `n` people can be found by considering the following:

- First, calculate the position of the survivor in a smaller circle with `n-1` people, but using the same step size `k`. This is done by making a recursive call with `jos(n - 1, k)`.

- The position found above is with respect to the smaller circle. To map this position back to the original circle, you need to add `k` and take the result modulo `n`. This is because after every `k`-th person is eliminated, the circle size reduces by 1 (from `n` to `n-1`), and you need to adjust the position accordingly.

3. \*\*Result\*\*: The result of the recursive calculation represents the position of the survivor in the original circle of `n` people.

\*\*Mathematical Proficiency\*\*:

- The Josephus problem is a famous problem in mathematics and computer science that has applications in various areas, including number theory and combinatorics.

\*\*Steps\*\*:

1. Check if the base case is met:

- If `n == 1`, return 0, as the sole person in the circle is the survivor, and their position is 0.

2. For circles with more than one person (`n > 1`), calculate the survivor's position using recursion:

- Calculate the survivor's position in a smaller circle with `n-1` people and the same step size `k` by making a recursive call to `jos(n - 1, k)`.

- Adjust the position found in the smaller circle by adding `k` and taking the result modulo `n`. This maps the position back to the original circle.

3. Return the calculated result, which represents the position of the survivor in the original circle.

\*\*Example\*\*:

Suppose there are 7 people in a circle, and every 3rd person is eliminated until only one person remains. Using the provided code, you can find the position of the survivor as follows:

- For a circle of 7 people (`n = 7`) and a step size of 3 (`k = 3`), the survivor's position is calculated as `jos(7, 3)`, which returns 3. This means that the last person remaining is in the 4th position when using a 0-based index.

The Josephus problem is a classic example of a recursive solution and has applications in various scenarios, including game theory and algorithmic challenges.

int jos(int n, int k)

{

    return (n == 1) ? 0 : (jos(n - 1, k) + k) % n;

}

\*\*Solution Approach\*\*:

The provided code checks whether a given string `s` is a palindrome, which means it reads the same backward as forward. The approach is based on recursive comparisons of characters from both ends of the string. Here's the approach:

1. \*\*Base Case\*\*: The base case is defined as follows: If the left index `l` is greater than or equal to the right index `r`, then the string has been fully checked, and it is considered a palindrome. In this case, return `true`.

2. \*\*Recursion\*\*: For strings with more than one character to check (`l < r`), the code compares the characters at positions `l` and `r` to determine if they are the same. If they are the same, the code makes a recursive call to check the next pair of characters, moving the `l` index one step to the right and the `r` index one step to the left.

3. If the characters at positions `l` and `r` are not the same at any point during the recursive calls, the function returns `false`, indicating that the string is not a palindrome.

4. If all recursive calls pass and the function does not return `false`, it reaches the base case and returns `true`, indicating that the entire string is a palindrome.

\*\*Mathematical Proficiency\*\*:

- Palindromes are a common concept in mathematics and language. They have applications in fields like linguistics and computer science, especially in text processing and data validation.

\*\*Steps\*\*:

1. Check if the base case is met:

- If `l` is greater than or equal to `r`, return `true` because the string has been fully checked and is considered a palindrome.

2. For strings with more than one character to check (`l < r`), perform the following steps recursively:

- Compare the characters at positions `l` and `r` in the string `s`.

- If they are not the same, return `false` to indicate that the string is not a palindrome.

- If they are the same, make a recursive call with the `l` index moved one step to the right (i.e., `l + 1`) and the `r` index moved one step to the left (i.e., `r - 1`).

3. Continue the recursive comparisons until reaching the base case. If the function does not return `false` during any of the recursive calls, it eventually reaches the base case and returns `true`, indicating that the entire string is a palindrome.

\*\*Example\*\*:

Consider the string "racecar." Using the provided code, you can check if it is a palindrome as follows:

- Call `isPalindromeString("racecar", 0, 6)` to check if the entire string is a palindrome. This initial call compares the characters at positions `0` (the start) and `6` (the end).

- It proceeds with recursive calls, checking if "a" (position `1` and `5`) and "c" (position `2` and `4`) are the same.

- All recursive calls pass without returning `false`, and the base case is reached when `l >= r`.

- The function returns `true`, indicating that "racecar" is a palindrome.

bool isPalindromeString(string s, int l, int r)

{

    if (l >= r)

        return true;

    if (s[l] != s[r])

        return false;

    return isPalindromeString(s, l + 1, r - 1);

}