Table Of Contents :

National Higher School of Cybersecurity
Date: 19/01/2025
Module: Algorithms and Data Structures
Prepared by: Laib Tarek | Group: B4
End-Semester Project Report

Introduction

In the field of programming, efficient data processing plays a crucial role in developing robust and scalable software systems. This project focuses on the development of a comprehensive C library that provides a wide variety of functions for processing and manipulating numbers, strings, arrays, and matrices. The primary goal of the project is to create modular, reusable code that can serve as a foundation for solving complex computational problems. Through the design of this library, students will deepen their understanding of algorithms, static data structures, and modular programming, while fostering creativity through the inclusion of user-defined functions.

Objective

The objective of this project is to develop a modular C library that includes efficient algorithms and functions for handling numbers, strings, arrays, and matrices. This project aims to:

- **Enhance programming skills,**
- **Strengthen algorithmic thinking, and**
- **Encourage creativity by allowing students to implement and expand upon the proposed functions.**

Operations on Numbers

This module contains a range of functions designed to perform both basic and advanced numerical operations. These functions are designed to improve efficiency and provide reusable components for various numerical computations. Below is a list of the operations that have been implemented in this module:

Function Purpose	**The `power` function calculates the result of raising an integer `base` to an integer `exponent`. It handles positive, negative, and zero exponents.**

Algorithm Design	1. Check if the `exponent` is `0`. If true, return `1` (any number raised to power `0` is `1`). 2. Handle the special case where both `base` and `exponent` are `0` by returning `0`. 3. For negative exponents, compute the reciprocal of the `base` and multiply iteratively for the absolute value of the exponent. 4. For positive exponents, iteratively multiply the `base` by itself. 5. Return the computed `result`.
Edge Cases Handled	- **Zero Exponent**: Returns `1` regardless of the `base`. - **Base and Exponent Zero**: Returns `0` for undefined cases. - **Negative Exponent**: Handles by computing the reciprocal of the `base` raised to the absolute value of the exponent.
Function Signature	`double power(int base, int exponent)` **Parameters**: - `base` (integer): The base number. - `exponent` (integer): The power to which the base is raised. **Returns**: A `double` representing the computed power.

Pseudocode

```
if exponent = 0 then
   return 1
end if

if base = 0 and exponent = 0 then
   return 0
end if

result ← base
if exponent < 0 then
   result ← 1.0 / base
   for i ← -1 to exponent do
      result ← result / base
   end for
else
   for i ← 1 to exponent - 1 do
      result ← result * base
   end for
end if

return result
```

Function Purpose	**The `swap` function exchanges the values of two integers in memory without using a temporary variable. This is achieved through XOR bitwise operations.**
Algorithm Design	1. Check if the two integers pointed to by `num1` and `num2` are not equal. 2. Perform XOR operations in the following order: - `num1 = num1 ^ num2` - `num2 = num1 ^ num2` - `num1 = num1 ^ num2` 3. Return, with the integers swapped in place.
Edge Cases Handled	- No operation is performed if the integers are already equal.
Function Signature	`void swap(int *num1, int *num2)` **Parameters**: - `num1` (pointer to integer): Pointer to the first integer. - `num2` (pointer to integer): Pointer to the second integer. **Returns**: No value (void).
Pseudocode	

```
Function swap(num1: pointer to integer, num2: pointer to integer)
   if *num1 ≠ *num2 then
      *num1 ← *num1 XOR *num2
      *num2 ← *num1 XOR *num2
      *num1 ← *num1 XOR *num2
   end if
End Function
```

Function Purpose	**The `sumOfDigits` function calculates the sum of all digits in a given integer. Negative integers are handled by converting them to positive.**
Algorithm Design	1. Convert the input number to its absolute value if it is negative. 2. Initialize a variable `sum` to `0`. 3. Use a loop to extract each digit of the number: - Add the digit to `sum`. - Remove the last digit from the number by integer division. 4. Repeat until the number becomes `0`. 5. Return the computed `sum`.
Edge Cases Handled	- Handles negative numbers by taking their absolute value. - Returns `0` for input `0`.

Function Signature	`int sumOfDigits(int num)` **Parameters**: - `num` (integer): The input number whose digits are to be summed. **Returns**: An integer representing the sum of the digits.
Pseudocode	

```
Function sumOfDigits(num: integer): integer
  if num < 0 then
    num ← -num
  end if

  sum ← 0
  while num ≠ 0 do
    sum ← sum + (num MOD 10)
    num ← num DIV 10
  end while

  return sum
End Function
```

Function Purpose	**The `reverseNumber` function computes the reverse of a given integer by reversing the order of its digits. Negative integers are converted to positive during the process.**
Algorithm Design	1. Convert the input number to its absolute value if it is negative. 2. Initialize a variable `reversed` to `0`. 3. Use a loop to extract each digit of the number: - Multiply `reversed` by `10` and add the last digit of the number. - Remove the last digit from the number by integer division. 4. Repeat until the number becomes `0`. 5. Return the computed `reversed`.
Edge Cases Handled	- Handles negative numbers by taking their absolute value. - Returns `0` for input `0`.
Function Signature	`int reverseNumber(int num)` **Parameters**: - `num` (integer): The input number to be reversed. **Returns**: An integer representing the reversed number.
Pseudocode	

```
Function reverseNumber(num: integer): integer
   if num < 0 then
      num ← -num
   end if

   reversed ← 0
   while num ≠ 0 do
      reversed ← reversed * 10 + (num MOD 10)
      num ← num DIV 10
   end while

   return reversed
End Function
```

Function Purpose	**The `isPalindrome` function checks whether a given integer is a palindrome, meaning it reads the same forwards and backwards. Negative integers are handled by converting them to positive.**
Algorithm Design	1. Convert the input number to its absolute value if it is negative. 2. Use the `reverseNumber` function to compute the reverse of the number. 3. Compare the original number with its reversed value. 4. Return `true` if they are equal, `false` otherwise.
Edge Cases Handled	- Handles negative numbers by taking their absolute value. - Returns `true` for single-digit numbers.
Function Signature	`bool isPalindrome(int num)` **Parameters**: - `num` (integer): The input number to check. **Returns**: A boolean indicating whether the number is a palindrome.
Pseudocode	

```
Function isPalindrome(num: integer): boolean
   if num < 0 then
      num ← -num
   end if

   return (num = reverseNumber(num))
End Function
```

Function Purpose	**The `isPrime` function determines whether a given integer is a prime number. A prime number is greater than 1 and divisible only by 1 and itself.**
Algorithm Design	1. Check if the input number is less than or equal to `1`. If true, return `false`. 2. Initialize a loop from `2` to the square root of the number. - Check if the number is divisible by any value in this range. - If a divisor is found, return `false`. 3. If no divisors are found, return `true`.
Edge Cases Handled	- Returns `false` for numbers less than or equal to `1`. - Correctly identifies `2` as a prime number.
Function Signature	`bool isPrime(int num)` **Parameters**: - `num` (integer): The input number to check for primality. **Returns**: A boolean indicating whether the number is prime.
Pseudocode	

```
Function isPrime(num: integer): boolean
   if num ≤ 1 then
      return false
   end if

   for i ← 2 to sqrt(num) do
      if num MOD i = 0 then
         return false
      end if
   end for

   return true
End Function
```

Function Purpose	**The `gcd` function calculates the greatest common divisor (GCD) of two integers using the Euclidean algorithm.**

Algorithm Design	1. Convert both numbers to their absolute values. 2. Handle the special case where both numbers are `0`, returning `1` as a placeholder. 3. Ensure `num1` is greater than or equal to `num2`. 4. Use a loop to compute the GCD by updating `num1` and `num2` with the remainder until `num2` becomes `0`. 5. Return `num1` as the GCD.
Edge Cases Handled	- Handles negative inputs by converting them to positive values. - Returns `1` for input `(0, 0)`.
Function Signature	`int gcd(int num1, int num2)` **Parameters**: - `num1` (integer): The first input number. - `num2` (integer): The second input number. **Returns**: An integer representing the GCD.
Pseudocode	

```
Function gcd(num1: integer, num2: integer): integer
   if num1 < 0 then
      num1 ← -num1
   end if

   if num2 < 0 then
      num2 ← -num2
   end if

   if num1 = 0 and num2 = 0 then
      return 1
   end if

   if num2 > num1 then
      swap(num1, num2)
   end if

   while num2 ≠ 0 do
      mod ← num1 MOD num2
      num1 ← num2
      num2 ← mod
   end while

   return num1
End Function
```

Function Purpose	**The `lcm` function computes the least common multiple (LCM) of two integers based on their GCD.**
Algorithm Design	1. Calculate the GCD of the two numbers using the `gcd` function. 2. Compute the LCM using the formula `lcm = (num1 * num2) / gcd(num1, num2)`. 3. Return the absolute value of the LCM.
Edge Cases Handled	- Handles negative inputs by returning a positive LCM.
Function Signature	`int lcm(int num1, int num2)` **Parameters**: - `num1` (integer): The first input number. - `num2` (integer): The second input number. **Returns**: An integer representing the LCM.
Pseudocode	

```
Function lcm(num1: integer, num2: integer): integer
   gcd_value ← gcd(num1, num2)
   lcm_value ← (num1 * num2) / gcd_value

   if lcm_value < 0 then
      lcm_value ← -lcm_value
   end if

   return lcm_value
End Function
```

Function Purpose	**The `fact` function calculates the factorial of a non-negative integer. Factorial is the product of all positive integers up to the given number.**
Algorithm Design	1. Handle the special case where the input number is negative, returning `-1` as an error indicator. 2. Initialize a variable `fact` to `1`. 3. Use a loop to multiply `fact` by each decrementing value of the number until it reaches `1`. 4. Return the computed `fact`.
Edge Cases Handled	- Returns `-1` for negative input as factorial is undefined for negative numbers. - Returns `1` for input `0` or `1`.

Function Signature	`long int fact(int num)` **Parameters**: - `num` (integer): The input number. **Returns**: A long integer representing the factorial.

Pseudocode

```
Function fact(num: integer): long integer
   if num < 0 then
      return -1
   end if

   fact ← 1
   while num > 1 do
      fact ← fact * num
      num ← num - 1
   end while

   return fact
End Function
```

Function Purpose	**The `isEven` function checks whether a given integer is even.**
Algorithm Design	1. Compute the modulus of the number with `2`. 2. Return `true` if the result is `0`; otherwise, return `false`.
Edge Cases Handled	- Handles all integer inputs, including negative numbers.
Function Signature	`bool isEven(int num)` **Parameters**: - `num` (integer): The input number to check. **Returns**: A boolean indicating whether the number is even.

Pseudocode

```
Function isEven(num: integer): boolean
   return (num MOD 2 = 0)
End Function
```

Function Purpose	**The `primeFact` function computes the prime factorization of a given integer, storing the prime factors in an array.**
Algorithm Design	1. Handle special cases for non-positive numbers by marking the array with `0` to indicate an error. 2. Initialize the smallest prime factor `i` as `2`. 3. Loop through possible factors while the number is greater than `1`: - Check if `i` is a prime factor of the number. - If true, store `i` in the factors array and divide the number by `i`. - If not, increment `i` to the next potential factor. 4. Mark the end of the factors array with `0`.
Edge Cases Handled	- Handles non-positive numbers by indicating an error in the factors array.
Function Signature	`void primeFact(int num, int factors[])` **Parameters**: - `num` (integer): The input number to factorize. - `factors` (array): The output array to store prime factors. **Returns**: Nothing (void).
Pseudocode	

```
Function primeFact(num: integer, factors: array): void
  if num ≤ 0 then
    factors[0] ← 0
    return
  end if

  i ← 2
  while num > 1 do
    if isPrime(i) and num MOD i = 0 then
      factors ← i
      factors++
      num ← num DIV i
    else
      i ← i + 1
    end if
  end while

  factors ← 0
End Function
```

Function Purpose	**The `isArmstrong` function determines whether a given integer is an Armstrong number. An Armstrong number is a number equal to the sum of its digits raised to a power.**
Algorithm Design	1. Handle negative input by returning `false`. 2. Initialize variables to store the original number and the result of computations. 3. For each digit in the number, compute its cube and add to the result. 4. Compare the result to the original number and return `true` if equal, otherwise `false`.
Edge Cases Handled	- Returns `false` for negative numbers.
Function Signature	`bool isArmstrong(int num)` **Parameters**: - `num` (integer): The input number to check. **Returns**: A boolean indicating whether the number is an Armstrong number.
Pseudocode	

```
Function isArmstrong(num: integer): boolean
  if num < 0 then
    return false
  end if

  temp ← num
  result ← 0
  while temp > 0 do
    result ← result + power((temp MOD 10), 3)
    temp ← temp DIV 10
  end while

  return (result = num)
End Function
```

Function Purpose	**The `fib` function generates the Fibonacci sequence up to the specified number of terms and stores it in an array.**
Algorithm Design	1. Handle non-positive input by returning early. 2. Initialize the first two Fibonacci numbers as `1`. 3. Use a loop to compute and store subsequent Fibonacci numbers in the array.

Edge Cases Handled - Handles non-positive input by returning early.

Function Signature `void fib(int num, int fib[])` **Parameters**: - `num` (integer): The number of Fibonacci terms to generate. - `fib` (array): The output array to store the sequence. **Returns**: Nothing (void).

Pseudocode

```
Function fib(num: integer, fib: array): void
   if num ≤ 0 then
      return
   end if

   fib0 ← 1
   fib1 ← 1
   for i ← 0 to num - 1 do
      fib[i] ← fib0
      fib2 ← fib1 + fib0
      fib0 ← fib1
      fib1 ← fib2
   end for
End Function
```

Function Purpose **The `sumDivisors` function computes the sum of all divisors of a given number.**

Algorithm Design 1. Handle special cases for `0`, `1`, and negative numbers. 2. Initialize the sum with the input number plus `1`. 3. Loop through potential divisors up to `num / 2` and add divisors to the sum. 4. Return the total sum of divisors.

Edge Cases Handled - Handles special cases for `0`, `1`, and negative numbers.

Function Signature `int sumDivisors(int num)` **Parameters**: - `num` (integer): The input number. **Returns**: An integer representing the sum of the divisors.

Pseudocode

```
Function sumDivisors(num: integer): integer
   if num = 0 then
      return 0
   end if

   if num = 1 then
      return 1
   end if

   if num < 0 then
      return 0
   end if

   sum ← num + 1
   for i ← 2 to num DIV 2 do
      if num MOD i = 0 then
         sum ← sum + i
      end if
   end for

   return sum
End Function
```

Function Purpose	**The `isPerfect` function checks whether a given integer is a perfect number. A perfect number is equal to the sum of its proper divisors.**
Algorithm Design	1. Compute the sum of divisors of the number using the `sumDivisors` function. 2. Subtract the number from the sum of its divisors. 3. Return `true` if the result equals the original number, otherwise `false`.
Edge Cases Handled	- Handles all positive integer inputs.
Function Signature	`bool isPerfect(int num)` **Parameters**: - `num` (integer): The input number to check. **Returns**: A boolean indicating whether the number is perfect.
Pseudocode	

```
Function isPerfect(num: integer): boolean
  return (sumDivisors(num) - num = num)
End Function
```

Function Purpose	**The `isMagic` function checks whether a given integer is a magic number. A magic number repeatedly sums its digits until a single digit is obtained, and that digit is `1`.**
Algorithm Design	1. Handle non-positive input by returning `false`. 2. Use a loop to repeatedly compute the sum of digits of the number until it becomes a single digit. 3. Check if the final digit is `1`.
Edge Cases Handled	- Returns `false` for non-positive numbers.
Function Signature	`bool isMagic(int num)` **Parameters**: - `num` (integer): The input number to check. **Returns**: A boolean indicating whether the number is a magic number.
Pseudocode	

```
Function isMagic(num: integer): boolean
  if num ≤ 0 then
    return false
  end if

  do
    num ← sumOfDigits(num)
  while num ≥ 10

  return (num = 1)
End Function
```

Function Purpose	**The `numOfDigits` function computes the number of digits in a given integer.**

Algorithm Design	1. Initialize a counter to `0`. 2. Use a loop to repeatedly divide the number by `10`, incrementing the counter with each iteration until the number becomes `0`. 3. Return the counter as the number of digits.
Edge Cases Handled	- Handles all integer inputs, including negative numbers.
Function Signature	`int numOfDigits(int num)` **Parameters**: - `num` (integer): The input number. **Returns**: An integer representing the number of digits.
Pseudocode	

```
Function numOfDigits(num: integer): integer
  count ← 0
  while num ≠ 0 do
    num ← num DIV 10
    count ← count + 1
  end while

  return count
End Function
```

Function Purpose	**The `isAutomorphic` function checks whether a given integer is an automorphic number. An automorphic number's square ends with the number itself.**
Algorithm Design	1. Handle negative input by returning `false`. 2. Compute the rank of the number using `10` raised to the power of its number of digits. 3. Check if the square of the number modulo its rank equals the number itself.
Edge Cases Handled	- Returns `false` for negative numbers.
Function Signature	`bool isAutomorphic(int num)` **Parameters**: - `num` (integer): The input number to check. **Returns**: A boolean indicating whether the number is automorphic.

Pseudocode

```
Function isAutomorphic(num: integer): boolean
    if num < 0 then
        return false
    end if

    rank ← power(10, numOfDigits(num))
    return ((num * num) MOD rank = num)
End Function
```

Function Purpose	**The `toBinary` function converts a non-negative integer to its binary representation.**
Algorithm Design	1. Handle special cases for `0` and negative numbers by returning early. 2. Use an array to store binary digits. 3. Repeatedly compute binary digits by dividing the number by `2` and store them in reverse order. 4. Rebuild the binary number from the binary digits array.
Edge Cases Handled	- Returns early for `0` and negative numbers.
Function Signature	`void toBinary(int *num)` **Parameters**: - `num` (pointer to integer): Pointer to the input number to convert. **Returns**: Nothing (void).

Pseudocode

```
Function toBinary(num: pointer to integer): void
    if *num ≤ 0 then
        return
    end if

    bin ← array of size *num
    i ← 0

    while *num > 0 do
        bin[i] ← *num MOD 2
        *num ← *num DIV 2
```

```
    i ← i + 1
  end while

  *num ← 0
  for j ← i - 1 to 0 do
    *num ← *num * 10 + bin[j]
  end for
End Function
```

Function Purpose	**The `isNarcissistic` function checks whether a given integer is a narcissistic number. A narcissistic number is equal to the sum of its digits raised to the power of the number of digits.**
Algorithm Design	1. Compute the sum of the digits of the number. 2. Raise the sum to the power of the number of digits. 3. Check if the result equals the original number.
Edge Cases Handled	- Handles all integer inputs, including negative numbers.
Function Signature	`bool isNarcissistic(int num)` **Parameters**: - `num` (integer): The input number to check. **Returns**: A boolean indicating whether the number is narcissistic.
Pseudocode	

```
Function isNarcissistic(num: integer): boolean
  return (power(sumOfDigits(num), numOfDigits(num)) = num)
End Function
```

Function Purpose	**The `sqrtApprox` function computes an approximate square root of a given number using the Babylonian method (Newton's method).**
Algorithm Design	1. Handle negative input by returning `-1` as an error value. 2. For `0`, return `0` as the square root. 3. Use the Babylonian method to iteratively refine the square root approximation until the difference is less than `0.001`.

Edge Cases Handled	- Returns `-1` for negative numbers. - Returns `0` for input `0`.
Function Signature	`double sqrtApprox(int num)` **Parameters**: - `num` (integer): The input number to approximate the square root. **Returns**: A double representing the approximate square root.
Pseudocode	

```
Function sqrtApprox(num: integer): double
  if num < 0 then
    return -1
  end if

  if num = 0 then
    return 0
  end if

  sq ← num * 2
  sq1 ← num
  while sq - sq1 > 0.001 do
    sq ← sq1
    sq1 ← 0.5 * (sq + num / sq)
  end while

  return sq1
End Function
```

Function Purpose	**The `isAbundant` function checks whether a given integer is an abundant number. An abundant number is one whose sum of proper divisors exceeds the number itself.**
Algorithm Design	1. Use the `sumDivisors` function to compute the sum of all divisors of the number. 2. Subtract the number from the sum of divisors. 3. Check if the result is greater than or equal to the number.
Edge Cases Handled	- Handles all positive integers.

Function Signature	`bool isAbundant(int num)` **Parameters**: - `num` (integer): The input number to check. **Returns**: A boolean indicating whether the number is abundant.
Pseudocode	

```
Function isAbundant(num: integer): boolean
  return (sumDivisors(num) - num ≥ num)
End Function
```

Function Purpose	**The `isDeficient` function checks whether a given integer is a deficient number. A deficient number is one whose sum of proper divisors is less than the number itself.**
Algorithm Design	1. Use the `isAbundant` function to check if the number is not abundant. 2. Return `true` if the number is deficient.
Edge Cases Handled	- Handles all positive integers.
Function Signature	`bool isDeficient(int num)` **Parameters**: - `num` (integer): The input number to check. **Returns**: A boolean indicating whether the number is deficient.
Pseudocode	

```
Function isDeficient(num: integer): boolean
  return NOT isAbundant(num)
End Function
```

Function Purpose	**The `sumEvenFibonacci` function computes the sum of even Fibonacci numbers up to a specified number of terms.**
Algorithm Design	1. Handle non-positive input by returning `0`. 2. Generate the Fibonacci sequence using the `fib` function. 3. Iterate through the sequence and sum up the even numbers.
Edge Cases Handled	- Returns `0` for non-positive input.

Function Signature	`int sumEvenFibonacci(int num)` **Parameters**: - `num` (integer): The number of Fibonacci terms. **Returns**: An integer representing the sum of even Fibonacci numbers.
Pseudocode	

```
Function sumEvenFibonacci(num: integer): integer
  if num ≤ 0 then
    return 0
  end if

  fibo ← array of size num
  sumFib ← 0

  fib(num, fibo)
  for i ← 0 to num - 1 do
    if fibo[i] MOD 2 = 0 then
      sumFib ← sumFib + fibo[i]
    end if
  end for

  return sumFib
End Function
```

Function Purpose	**The `isHarshad` function checks whether a given integer is a Harshad number. A Harshad number is divisible by the sum of its digits.**
Algorithm Design	1. Compute the sum of the digits of the number using `sumOfDigits`. 2. Check if the number is divisible by the sum of its digits.
Edge Cases Handled	- Handles all integer inputs, including negative numbers.
Function Signature	`bool isHarshad(int num)` **Parameters**: - `num` (integer): The input number to check. **Returns**: A boolean indicating whether the number is a Harshad number.
Pseudocode	

```
Function isHarshad(num: integer): boolean
   return (num MOD sumOfDigits(num) = 0)
End Function
```

Function Purpose	**The `isHappy` function checks whether a given integer is a happy number. A happy number eventually reduces to `1` when replaced by the sum of the squares of its digits.**
Algorithm Design	1. Handle non-positive input by returning `false`. 2. Use a loop to compute the sum of the squares of the digits repeatedly until the number becomes `1` (happy) or `4` (cycle detected).
Edge Cases Handled	- Returns `false` for non-positive numbers.
Function Signature	`bool isHappy(int num)` **Parameters**: - `num` (integer): The input number to check. **Returns**: A boolean indicating whether the number is happy.
Pseudocode	

```
Function isHappy(num: integer): boolean
   if num ≤ 0 then
      return false
   end if

   while num ≠ 1 and num ≠ 4 do
      sum ← 0
      while num ≠ 0 do
         digit ← num MOD 10
         sum ← sum + digit * digit
         num ← num DIV 10
      end while
      num ← sum
   end while

   return (num = 1)
End Function
```

Function Purpose	**The `catalanNumber` function computes the nth Catalan number using an iterative approach. Catalan numbers are used in combinatorics to solve various counting problems.**
Algorithm Design	1. Handle negative input by returning `-1` as an error value. 2. Return `1` for input `0` (base case). 3. Use an iterative formula to compute the Catalan number.
Edge Cases Handled	- Returns `-1` for negative input. - Returns `1` for input `0`.
Function Signature	`int catalanNumber(int num)` **Parameters**: - `num` (integer): The input number. **Returns**: An integer representing the nth Catalan number.

Pseudocode

```
Function catalanNumber(num: integer): integer
   if num < 0 then
      return -1
   end if

   if num = 0 then
      return 1
   end if

   catalan0 ← 1
   catalan1 ← 1
   for i ← 1 to num do
      catalan0 ← catalan1
      catalan1 ← (catalan0 * 2 * (2 * i + 1)) / (i + 2)
   end for

   return catalan0
End Function
```

Function Purpose	**The `pascalTriangle` function generates Pascal's Triangle up to the specified number of rows and stores it in a 2D array.**

Algorithm Design	1. Handle non-positive input by exiting the function. 2. Initialize the first column and diagonal elements to `1`. 3. Use the formula to compute the inner elements of Pascal's Triangle.
Edge Cases Handled	- Handles non-positive input gracefully by exiting the function.
Function Signature	`void pascalTriangle(int num, int pascalTriangle[][100])` **Parameters**: - `num` (integer): The number of rows. - `pascalTriangle` (2D array): The output triangle.
Pseudocode	

```
Function pascalTriangle(num: integer, pascalTriangle: 2D array): void
  if num ≤ 0 then
    return
  end if

  for i ← 0 to num - 1 do
    pascalTriangle[i][0] ← 1
    pascalTriangle[i][i] ← 1
    for j ← 1 to i - 1 do
      pascalTriangle[i][j] ← pascalTriangle[i - 1][j - 1] + pascalTriangle[i - 1][j]
    end for
  end for
End Function
```

Function Purpose	**The `bellNumber` function computes the nth Bell number, which represents the number of ways to partition a set of `n` elements.**
Algorithm Design	1. Handle negative input by returning `-1` as an error value. 2. Initialize the base case with the first Bell number as `1`. 3. Use a dynamic programming approach to compute subsequent Bell numbers.
Edge Cases Handled	- Returns `-1` for negative input.

Function Signature	`int bellNumber(int num)` **Parameters**: - `num` (integer): The input number. **Returns**: An integer representing the nth Bell number.
Pseudocode	

```
Function bellNumber(num: integer): integer
  if num < 0 then
    return -1
  end if

  bell ← 2D array of size [num + 1][num + 1]
  bell[0][0] ← 1

  for i ← 1 to num do
    bell[i][0] ← bell[i - 1][i - 1]
    for j ← 1 to i do
      bell[i][j] ← bell[i - 1][j - 1] + bell[i][j - 1]
    end for
  end for

  return bell[num][0]
End Function
```

Function Purpose	**The `isKaprekar` function checks whether a given integer is a Kaprekar number. A Kaprekar number is one whose square can be split into two parts that sum to the original number.**
Algorithm Design	1. Handle non-positive input by returning `false`. 2. Compute the square of the number and split it into two parts based on its number of digits. 3. Check if the sum of the two parts equals the original number.
Edge Cases Handled	- Returns `false` for non-positive input.
Function Signature	`bool isKaprekar(int num)` **Parameters**: - `num` (integer): The input number to check. **Returns**: A boolean indicating whether the number is a Kaprekar number.
Pseudocode	

```
Function isKaprekar(num: integer): boolean
  if num ≤ 0 then
    return false
  end if

  cPoint ← power(10, numOfDigits(num))
  kap0 ← (num * num) MOD cPoint
  kap1 ← (num * num) DIV cPoint

  return (kap0 + kap1 = num)
End Function
```

Function Purpose	**The `isSmith` function checks whether a given integer is a Smith number. A Smith number is a composite number whose sum of digits equals the sum of the digits of its prime factors.**
Algorithm Design	1. Handle negative input by converting it to positive. 2. Return `false` if the number is prime. 3. Compute the sum of the digits of the number. 4. Find the prime factors of the number and compute the sum of their digits. 5. Check if the two sums are equal.
Edge Cases Handled	- Handles negative input by converting it to positive.
Function Signature	`bool isSmith(int num)` **Parameters**: - `num` (integer): The input number to check. **Returns**: A boolean indicating whether the number is a Smith number.
Pseudocode	

```
Function isSmith(num: integer): boolean
  if num < 0 then
    num ← -num
  end if

  if isPrime(num) then
    return false
```

```
    end if

    smith0 ← sumOfDigits(num)
    sFactors ← primeFact(num)
    smith1 ← 0

    for i ← 0 to size of sFactors do
        smith1 ← smith1 + sumOfDigits(sFactors[i])
    end for

    return (smith0 = smith1)
End Function
```

Function Purpose	**The `sumOfPrimes` function computes the sum of all prime numbers up to a given limit.**
Algorithm Design	1. Handle input less than `2` by returning `0`. 2. Loop through numbers from `2` to the input limit and check if each is prime. 3. Sum the prime numbers.
Edge Cases Handled	- Returns `0` for input less than `2`.
Function Signature	`int sumOfPrimes(int num)` **Parameters**: - `num` (integer): The upper limit. **Returns**: An integer representing the sum of prime numbers up to the limit.
Pseudocode	

```
Function sumOfPrimes(num: integer): integer
    if num < 2 then
        return 0
    end if

    sum ← 0
    for i ← 2 to num do
        if isPrime(i) then
            sum ← sum + i
        end if
    end for

    return sum
```

```
End Function
```