*ALGORITHM*

*PROJECT REPORT*

* National School of Cyber Security
* END OF FIRST SEMESTER PROJECT

**Github:**

* **0xFAROUK-CYBER**
* **yznix-mamer**
* **URL:**
* https://github.com/0xFAROUK-CYBER/LibraryProject-repo

***Group members***

* HADJ HACENE FAROUK “A3”
* MAAMERI YANIS “B4”

**UNDER THE SUPERVISION OF**

* MR. BERGHOUT .
* MR. NAIT HAMOUD .
* MR.SOUALMI .

**NumberOfDigits**

**Function Purpose:**

* The NumberOfDigits(int num) function calculates the number of digits in a given integer number.
* It is useful in tasks that involve handling digits individually.

**Algorithm Design:**

* **Special Case:** If the input is 0, the function directly returns 1, as 0 has only one digit.
* **Loop to Count Digits:**
  1. Initialize a variable digits to 0.
  2. Iterate through a loop until the num becomes 0.
  3. In each iteration:
     + Increment the digits counter by 1.
     + Remove the last digit of the num by dividing it by 10 using integer division.
* **Return:** Return the final value of digits, which represents the total count of digits in the input number.

**Edge Cases Handled:**

* **Zero:** If the input is 0, the function handles this case specifically by returning 1.
* **Negative Numbers:** The function doesn't explicitly handle negative numbers. It will work as expected for negative inputs, as the loop iterates until the input becomes 0, regardless of its sign. However, it's important to note that it treats negative numbers as positive during the digit counting process.

**Combination**

**Function Purpose :**

* The Combination(int n, int r) function calculates the number of ways to choose *r* items from a set of *n* items. This is also known as the binomial coefficient.

**Algorithm Design :**

* The function first checks for invalid input:
  + r should not exceed n (i.e., r > n is invalid)
  + r and n should not be negative
* If the input is invalid, the function returns 0.
* Otherwise, it calculates the combination using the formula:
  + C(n, r) = n! / (r! \* (n - r)!)
  + This formula represents the number of ways to choose *r* items from *n* items.

**Steps :**

1. Check for invalid input.
2. If input is valid, calculate the combination using the provided formula.
3. Return the calculated combination.

**Edge Cases Handled :**

* **Invalid Input:** The function handles invalid input by returning 0. This ensures that the function does not return incorrect results.
* **Negative Input:** The function handles negative input by returning 0. This prevents the function from returning incorrect results for negative inputs.

**Absolute**

* This function calculates the absolute value of an integer.

**Steps :**

1. Check if the input number is negative.
2. If it's negative, multiply it by -1 to make it positive.
3. Return the absolute value.

**Edge Cases Handled :**

* **Negative Input:** The function handles negative input by converting it to a positive value.
* **Zero Input:** The function returns 0 for a zero input, which is the absolute value of zero.

**Sum of Digits**

**Function Purpose:**

* The sumOfDigits(int num) function calculates the sum of all digits of a given integer num.
* This function is useful in numerical analysis, checksum calculations, and digital root computations.

**Algorithm Design:**

* The function iteratively extracts each digit of the given number by repeatedly using the modulus (%) and division (/) operators.
* The extracted digit is added to a cumulative sum until the number is reduced to zero.

**Steps:**

1. Initialize a variable sum to 0.
2. Use a loop to extract the last digit of the number (num % 10).
3. Add the extracted digit to sum.
4. Update the number by removing the last digit (num/10).
5. Repeat steps 2-4 until the number becomes 0.
6. Return the computed sum.

**Edge Cases Handled:**

* **Negative Numbers:** The function converts negative input into a positive value.
* **Zero:** If the input is 0, the function directly returns.

**ReverseNumber**

**Function Purpose:**

* The ReverseNumber(int num) function calculates the reverse of a given number.
* The function returns -1 if the input is negative as the reverse of a negative number is ambiguous.

**Algorithm Design:**

* The function uses a loop to iterate through the digits of the given number.
* In each iteration, the function extracts the last digit of the number using the modulus operator (%).
* This extracted digit is then appended to the reversed number, which is initialized to 0.
* The last digit is removed from the original number by dividing it by 10 using integer division (/).
* The loop continues until the original number becomes 0.
* The function returns the reversed number.

**Steps:**

* **Initialization:** Initialize variables digit and Reverse to 0.
* **Input Validation:** Check if the input number is negative. If yes, return -1.
* **Iteration:** Use a loop to iterate while the original number is greater than 0.
  + **Extract Digit:** Extract the last digit of the original number using the modulus operator (%).
  + **Append to Reverse:** Multiply the current Reverse value by 10 and add the extracted digit to it.
  + **Remove Last Digit:** Divide the original number by 10 using integer division (/) to remove the last digit.
* **Return:** After the loop completes, return the Reverse value, which now contains the reversed number.

**Edge Cases Handled :**

* **Negative Numbers:** The function returns -1 if the input is negative, as the definition of a negative number's reverse is ambiguous.

**Palindrome**

**Function Purpose:**

* The isPalindrome(int num) function determines whether a given integer is a palindrome.
* It checks if the number is identical when reversed.

**Algorithm Design:**

* Negative Number Check: The function begins by checking if the input num is negative. If it is, the function immediately returns false because negative numbers cannot be palindromes.
* Number Reversal: The function calls the ReverseNumber(num) function to reverse the input number.
* Palindrome Check: The original number (num) is compared to its reversed version.
  + If they are equal, the function returns true indicating that the number is a palindrome.
  + If they are not equal, the function returns false indicating the number is not a palindrome.

**Edge Cases Handled :**

* Negative Numbers: The function explicitly handles negative numbers by returning false as they cannot be palindromes.
* Zero: While not explicitly mentioned, the function implicitly handles zero as it's a palindrome (0 reversed is still 0).

**Prime Number**

**Function Purpose:**

* The isPrime(int num) function determines if a given integer num is a prime number.

**Algorithm Design:**

1. The function begins by checking if the input num is less than 2. If it is, it returns false since prime numbers must be greater than 1.
2. The function iterates through a loop starting from 2 and continuing up to the square root of num.
3. For each number i within this range, the function checks if num is divisible by i.
4. If num is divisible by any number i, it returns false, indicating that num is not prime.
5. If the loop completes without finding any divisors, it returns true, indicating that num is a prime number.

**Edge Cases Handled:**

* **Negative Numbers:** The function handles negative numbers by returning false as prime numbers are always positive.
* **Zero and One:** The function returns false for 0 and 1, as they are not considered prime numbers.

**GRATEST COMMON DIVISOR**

**Function Purpose**

* The gcd(int a, int b) function calculates the Greatest Common Divisor (GCD) of two integers a and b using the Euclidean algorithm.
* The GCD is the largest positive integer that divides both a and b without leaving a remainder.

**Algorithm Design :**

* The function first ensures that a is the smaller number for efficiency. If not, it swaps the values.
* The function uses a while loop to repeatedly compute the remainder of a divided by b until b becomes 0.
* In each iteration:
  + If a is completely divisible by b, the function returns b as the GCD.
  + Otherwise, it calculates the remainder (GCD = a % b), assigns the previous value of b to a, and assigns the remainder to b.
* When b reaches 0, a will contain the GCD, which is returned.

**Steps :**

1. **Initialization:**
   * Declare variables GCD and t to store temporary values.
2. **Swapping:**
   * If a is greater than b, swap their values to ensure a is the smaller number.
3. **Euclidean Algorithm:**
   * Enter a while loop that continues as long as b is not equal to 0.
   * Inside the loop:
     + If a is completely divisible by b, return b as the GCD.
     + Otherwise, calculate the remainder (GCD = a % b), assign b to a, and assign the remainder (GCD) to b.
4. **Return GCD:**
   * When the loop terminates (i.e., b becomes 0), return a as the GCD.

**Edge Cases Handled :**

* This function doesn't explicitly handle edge cases like negative numbers, zero, or if a and b are both zero. However, the Euclidean algorithm inherently handles these cases.
* If either a or b is zero, the function will directly return the non-zero value as the GCD.
* If both a and b are zero, the function will likely return

**LEAST COMMON MULTIPLYER**

**Function Purpose:**

* The lcm(int a, int b) function calculates the Least Common Multiple (LCM) of two integers a and b.

**Algorithm Design:**

* The function uses the formula: LCM(a, b) = (a \* b) / GCD(a, b)
* It calls the gcd(a, b) function to compute the Greatest Common Divisor (GCD) of a and b.
* It then multiplies a and b and divides the product by the GCD to obtain the LCM.

**Steps:**

1. Compute the GCD of a and b using the gcd(a, b) function.
2. Multiply a and b.
3. Divide the product from step 2 by the GCD from step 1.
4. Return the result of the division.

**Edge Cases Handled:**

* **Zero or Negative Inputs**: The function does not explicitly handle negative or zero inputs. The formula used is valid for positive numbers. The user should ensure that valid positive inputs are provided.

**Factorial**

**Function Purpose:**

* The Factorial(int num) function calculates the factorial of a given number num using recursion.
* Factorial is the product of all positive integers less than or equal to a given number.

**Algorithm Design:**

* Base Case: If the input number is 0 or 1, the factorial is defined as 1.
* Recursive Case: If the input number is greater than 1:
  + The function recursively calculates the factorial of num-1.
  + It multiplies the current number num with the factorial of num-1 and returns the result.

**Steps:**

1. **Initialization:** 
   * A variable Fact is initialized to 1 to store the factorial result.
2. **Base Case Check:** 
   * If the input num is 0 or 1, the function returns 1 directly.
3. **Invalid Input Check:** 
   * If the input num is less than 0, the function returns -1 to indicate an error.
4. **Recursive Call:** 
   * If the input num is greater than 1, the function calls itself recursively with num-1.
   * The function then multiplies the current number num with the result of the recursive call and returns the product.

**Edge Cases Handled :**

* **Negative Numbers**: The function handles negative input by returning -1 to signal an error.
* **Zero and One:** The function correctly returns 1 for the base cases of 0 and 1.

**Even Number**

**Function Purpose:**

* The isEven(int num) function checks whether the given integer num is even or not.

**Algorithm Design**

* It uses the modulus operator % to find the remainder when num is divided by 2.
* If the remainder is 0, the number is even and true is returned.
* Otherwise, the number is odd and false is returned.

**Steps:**

1. Check if the remainder of num divided by 2 is equal to 0 (num % 2 == 0).
2. If the remainder is 0, return true.
3. Otherwise, return false.

**Edge Cases Handled:**

* None, the function handles all integer inputs correctly.

**Prime Factors**

**Function Purpose:**

* The function takes an integer input (num) and identifies its prime factors.
* It iteratively checks for divisibility by prime numbers, dividing the input number repeatedly until it reaches 1.
* The prime factors are printed in order.

**Algorithm Design:**

* **Initialization:** 
  + Starts with i set to 2, representing the first prime number.
  + Checks if num is less than or equal to 1, in which case it prints a message indicating that there are no prime factors and returns.
* **Iterative Division**:
  + Enters a loop that continues until num is reduced to 1.
    - **Prime Check**: Calls the isPrime(i) function to check if i is a prime number.
    - **Divisibility Check**: If i is prime and num is divisible by i, it prints i as a prime factor and divides num by i.
    - **Increment**: Moves to the next number (i += 1) after checking for divisibility by the current prime number.
* **Output**:
  + Prints the prime factors separated by spaces.
  + Prints a newline character to separate the output from other potential output.

**Edge Case Handled:**

* **Invalid Input**: The function handles input values less than or equal to 1 by printing a message stating that there are no prime factors.

**Armstrong Number**

**Purpose:**

* The isArmstrong(int num) function takes an integer num as input and checks if it's an Armstrong number.

**Algorithm:**

1. **Initialization:**
   * armstrong: This variable will store the sum of the cubes of the digits of the input number. It is initialized to 0.
   * temp: This variable stores a copy of the original input number (num) to be used later for comparison.
2. **Digit Cube Calculation:**
   * A while loop iterates until the original number (num) becomes 0.
     + Inside the loop:
       - The last digit of num is extracted using the modulo operator (num % 10).
       - The cube of the extracted digit is calculated using pow(num % 10, 3) and added to the armstrong variable.
       - The last digit is removed from num using integer division (num /= 10).
3. **Comparison:**
   * After the loop completes, the original input number is restored from the temp variable.
   * The function compares armstrong with the restored num.
   * If they are equal, it means the number is an Armstrong number, and the function returns true.
   * Otherwise, it returns false.

**Edge Cases:**

* The function handles negative numbers by converting them to positive values before performing the calculations.
* The function handles zero correctly, as it is considered an Armstrong number.

**Fibonacci Series**

**Function Purpose:**

* The fibonacciSeries(int num) function calculates and prints the Fibonacci sequence up to a given number num.

**Algorithm Design:**

* **Initialization:**
  + A variable fibonacci is declared as a long type to store the calculated Fibonacci numbers.
* **Looping:**
  + A for loop iterates through numbers from 0 to num to calculate each Fibonacci term.
* **Fibonacci Calculation:**
  + Inside the loop, Binet's formula is used to calculate the Fibonacci number at position i:
    - fibonacci = (1 / sqrt(5)) \* ((pow(((1 + sqrt(5)) / 2), i)) - pow(((1 - sqrt(5)) / 2), i))
* **Printing:**
  + After each calculation, the Fibonacci number is printed along with its corresponding position i using printf.

**Edge Cases Handled:**

* **Binet's Formula:** The function uses Binet's formula to calculate the Fibonacci numbers. This formula is generally suitable for larger numbers, but it might have rounding issues for smaller numbers due to floating-point arithmetic.
* **Data Type:** The fibonacci variable is declared as long to handle potential overflow issues with larger Fibonacci numbers.
* **Negative Numbers:** The function handles negative input by printing an error message and exiting the function.

**Sum of Divisors Function**

**Function Purpose:**

* The sumDivisors(int num) function calculates the sum of all divisors of a given integer num.

**Algorithm Design:**

* The function iterates through numbers from 1 to num/2.
* For each number i in the loop, it checks if i is a divisor of num.
* If i is a divisor, it is added to the sum.
* The function returns the total sum of divisors.

**Steps:**

1. Initialize a variable sum with the value of num.
2. Iterate through numbers from 1 to num/2.
3. For each i, check if i is a divisor of num using the modulo operator (num%i == 0).
4. If i is a divisor, add it to sum.
5. Return the total sum sum.

**PERFECT NUMBER**

**Logic:**

* A perfect number is a positive integer that equals the sum of its proper divisors (divisors excluding the number itself).
* The function calculates the sum of the number's proper divisors using the sumDivisors function.
* It subtracts the input number itself from this sum to obtain the sum of proper divisors.
* If the sum of proper divisors equals the input number, the function returns true indicating a perfect number.
* Otherwise, it returns false indicating the number is not perfect.

**Steps:**

1. **Calculate the sum of divisors:** Use the sumDivisors function to obtain the sum of all divisors of the input number, including the number itself.
2. **Subtract the input number:** Subtract the input number from the sum of divisors to get the sum of proper divisors.
3. **Compare and return:**
   * If the sum of proper divisors equals the input number, return true.
   * Otherwise, return false.

**Example:**

Consider the input number 6:

* The proper divisors of 6 are 1, 2, and 3.
* The sum of its proper divisors is 1 + 2 + 3 = 6.
* Therefore, 6 is a perfect number, and the isPerfect function would return true.

**MAGIC NUMBER**

**Function Purpose:**

* The function aims to identify magic numbers.
* A magic number is defined as a number that eventually reduces to 1 when repeatedly calculating the sum of its digits until a single-digit number is obtained.

**Algorithm Design:**

1. Loop for Digits Sum: The code uses a while loop to repeatedly calculate the sum of digits of the input number.
   * The loop continues as long as the number is greater than or equal to 10.
   * Inside the loop, the SumOfDigits function (which is assumed to be defined elsewhere) is called to calculate the sum of digits of the current number.
   * The resulting sum is then assigned back to the number variable. This process effectively reduces the number by repeatedly summing its digits.
2. Single Digit Check: Once the loop completes, it means the number has been reduced to a single digit.
   * The code checks if the final number is equal to 1.
   * If it is, the number is a magic number, and the function returns true.
   * If it is not 1, the number is not a magic number, and the function returns false.

**Example:**

* If the input number is 123:
  + The loop calculates the sum of digits: 1 + 2 + 3 = 6
  + The number is updated to 6.
  + The loop continues as 6 is still greater than or equal to 10.
  + The sum of digits of 6 is 6.
  + The number is updated to 6.
  + The loop exits as 6 is less than 10.
  + The final single-digit number is 6, which is not 1.
  + The function returns false, indicating that 123 is not a magic number.

**Automorphic Number**

**Purpose:**

* The function isAutomorphic(int num) determines if a given integer (num) is an automorphic number.

**Algorithm:**

1. **Calculate the Square:** Calculate the square of the input number (num \* num) and store it in the variable square.
2. **Determine Place Value:**
   * Initialize a variable digit to 1, which will be used to determine the place value of the input number (powers of 10).
   * Repeatedly multiply digit by 10 until the division of num by digit is not zero. This ensures that digit represents the place value of the most significant digit in num.
3. **Compare Last Digits:**
   * Calculate the remainder when square is divided by digit. This gives us the last few digits of the square, representing the same number of digits as in num.
   * Compare this remainder with the original input number (num).
4. **Return Result:**
   * If the remainder is equal to the original number, return true, indicating that the number is automorphic.
   * Otherwise, return false.

**Edge Cases Handled:**

* The function does not explicitly handle any specific edge cases.
* However, the logic is designed to work for both positive and negative integers.
* It also gracefully handles scenarios where the input number is 0, as in this case, the square will also be 0, and the comparison will be successful, returning true.

**Decimal To Binary**

**purpos**

* The toBinary(int num) function converts a given integer to its binary representation.
* The function handles negative input by printing an error message and exiting.

**Algorithm Design :**

* **Initialization:**
  + **Bin**: Initialized to 0, this variable will store the calculated binary representation.
  + **i**: Initialized to 1, this variable acts as a multiplier to position the binary digits correctly.
* **Input Validation:**
  + If the input num is negative, the function prints an error message "ERROR." and exits using return;.
* **Binary Conversion Loop:**
  + The function iterates using a while loop until the input num becomes 0.
  + **Inside the loop:**
    - **Bin = Bin + (num % 2) \* i**: Extracts the last binary digit (num % 2) and adds it to Bin, multiplied by the appropriate multiplier (i) for its correct place value.
    - **num /= 2**: Divides the number by 2 to shift to the next binary digit.
    - **i \*= 10**: Multiplies the multiplier (i) by 10 to move to the next position in the binary representation.
* **Output:** The binary representation stored in Bin is printed to the console.

**Edge Cases Handled:**

* **Negative Numbers:** The function handles negative input by printing an error message and exiting the function.
* **Zero:** The function will correctly produce the binary representation of 0, which is 0.

**NARCISSISTIC**

* The isNarcissistic(int num) function checks if a given integer is a narcissistic number.
* A narcissistic number is a number that is equal to the sum of the digits of that number raised to the power of the number of digits in the number. For example, 153 is a narcissistic number because 153 = 1^3 + 5^3 + 3^3.

**Function Steps:**

1. **Calculate the number of digits in the number:**
   * The function uses a helper function NumberOfDigits(num) (not provided in the code snippet) to determine the number of digits in the input number (num).
   * This value is stored in the variable digit.
2. **Initialize variables:**
   * temp: A temporary variable to store the original value of num to be used later.
   * Narcissistic: A variable to store the sum of the digits raised to the power of the number of digits.
3. **Iterate through the digits:**
   * The function enters a while loop that continues as long as num is not equal to 0.
   * Inside the loop:
     + The last digit of num is extracted using num % 10.
     + This digit is raised to the power of digit using the pow function.
     + The result of the power calculation is added to Narcissistic.
     + The last digit is removed from num using num /= 10.
4. **Restore the original number:**
   * After the loop completes, the original value of num is restored from temp.
5. **Check for Narcissistic number:**
   * The function compares the calculated Narcissistic value with the original num.
   * If they are equal, it returns true, indicating that the number is narcissistic.
   * Otherwise, it returns false, indicating that the number is not narcissistic.

**Edge Cases:**

* The code does not explicitly handle negative numbers. The pow function may not return an expected result for negative numbers.
* The code does not explicitly handle zero as an input. However, the loop would terminate immediately in this case, and the function would return false .

**SQUARE ROOT**

**Function Description :**

* The sqrtApprox function approximates the square root of a given integer num using the Newton-Raphson method.

**Function Parameters :**

* **num**: An integer representing the number for which the square root is to be approximated.

**Function Body :**

1. **Precision Tolerance:**
   * Defines a variable epsilon with a value of 0.0001 to determine the desired accuracy of the approximation.
2. **Initial Values:**
   * Initializes two variables:
     + x: Represents the current guess for the square root, initialized to 0.0.
     + xPrev: Represents the previous guess for the square root, initialized to 0.0.
3. **Special Case Handling:**
   * **If num is 0:** Sets x to 0, as the square root of 0 is 0.
   * **If num is 1:** Sets x to 1, as the square root of 1 is 1.
   * **If num is negative:** Sets x to -1, indicating that the square root of a negative number is undefined.
4. **Iterative Approximation:**
   * **If num is greater than 1:**
     + Initializes x to num/2 as an initial guess for the square root.
     + Iteratively refines the guess using the Newton-Raphson formula:
       - x = (xPrev + num/xPrev)/2;
       - The previous guess (xPrev) is updated with the current guess (x).
       - This iteration continues until the absolute difference between the current guess (x) and the previous guess (xPrev) falls below the defined precision tolerance (epsilon).
5. **Return Value:**
   * The function returns the approximated square root (x).

**Function Behavior :**

The sqrtApprox function implements the Newton-Raphson method, which iteratively refines a guess for the square root until it reaches a desired level of accuracy. It handles special cases for input values of 0, 1, and negative numbers. The function ensures that the returned value is within the specified precision tolerance.

**Power**

**Purpose:**

* This function calculates the power of a given base raised to a given exponent.
* It handles different scenarios, including:
  + Base = 0
  + Negative exponents

**Algorithm:**

1. **Initialization:**
   * pow: A double variable to store the result of the power calculation. Initialized to 1.
   * T: An integer flag to mark if the exponent was initially negative. Initialized to 1.
2. **Base = 0 Handling:**
   * If the base is 0:
     + If the exponent is positive, set pow to 0 (any positive power of 0 is 0).
     + If the exponent is negative or zero, set pow to -1 (to indicate an undefined result).
3. **General Power Calculation:**
   * If the base is not 0:
     + If the exponent is negative, convert it to positive and set T to 0 to indicate the original negative exponent.
     + For a positive exponent:
       - Use a loop to multiply the base exp times to calculate the power.
     + If the exponent was initially negative, compute the reciprocal of the calculated power.
4. **Return:**
   * Return the calculated power.

**Edge Cases:**

* **Base 0:**
  + If the base is 0, the function handles special cases for positive and non-positive exponents to determine the result.
* **Negative Exponent:**
  + The function converts the negative exponent to positive, calculates the power, and then takes the reciprocal to get the correct result for the negative exponent.

**HAPPY NUMBER**

**Function Purpose:**

* The isHappy(int num) function determines if a given integer is a "happy number".
* A happy number is a number that eventually reaches 1 when replaced by the sum of the squares of its digits repeatedly.

**Algorithm Design:**

* **Initialization:**
  + A variable sum is initialized to 0 to store the sum of the squares of the digits.
* **Single Digit Check:**
  + If the input number is less than 10, it is considered a single digit and its square is calculated using the power() function.
* **Iterative Process:**
  + A while loop iterates until the number becomes a single digit.
  + Within the loop:
    - The sum variable is reset to 0.
    - Another while loop calculates the sum of the squares of the digits of the current number:
      * The last digit of the current number is extracted using the modulo operator (%).
      * The extracted digit is squared using the power function and added to sum.
      * The last digit is removed from the current number using integer division (/).
    - The current number is updated with the calculated sum.
* **Result Check:**
  + Once the number becomes a single digit, the function checks if it equals 1.
  + If the single-digit number is 1, the function returns true, indicating the input number is a happy number.
  + Otherwise, the function returns false, indicating the input number is not a happy number.

**Edge Cases Handled:**

* **Single Digit:** If the input number is less than 10, it is directly squared.
* **Zero:** The function does not handle zero specifically as it is not considered a happy number.

**ABUNDANT NUMBER**

**Function Purpose:**

* The isAbundant(int num) function determines if a given integer num is an abundant number.

**Algorithm Design:**

1. The function uses the sumDivisors(num) function to calculate the sum of all the proper divisors (excluding the number itself) of the input number num.
2. It then compares the sum of divisors (sumDivisors(num)) with the input number num.
3. If the sum of divisors is greater than the number itself (sumDivisors(num) - num > num), it returns true, indicating the number is abundant.
4. Otherwise, it returns false.

**Edge Cases Handled:**

* The code handles cases where the input number is 0 or negative .

**Deficient Number**

**Purpose:**

* The isDeficient(int num) function determines if a given integer (num) is a deficient number.

**Logic:**

1. **Sum of Proper Divisors Calculation:**
   * It calls the sumDivisors(num) function (assumed to be defined elsewhere) to calculate the sum of all proper divisors (excluding the number itself) of the given integer.
2. **Deficiency Check:**
   * It checks if the sum of proper divisors is less than or equal to twice the number itself (sumDivisors(num) <= 2 \* num) and if the number is not zero (num != 0).
   * If both conditions are true, the number is considered deficient, and the function returns true.
   * Otherwise, the function returns false, indicating the number is not deficient.

**Edge Case:**

* The code explicitly checks for a zero input (num != 0). This is important because the sum of proper divisors of zero is undefined.

**Sum of Even Fibonacci**

**Function Purpose :**

* The sumEvenFibonacci(int num) function calculates the sum of all even Fibonacci numbers up to a given number, num.
* It first checks if the input is non-negative. If not, it returns -1.
* If the input is non-negative, it initializes a variable sumfib to 0.
* Then, it iterates through the Fibonacci sequence up to the given number and adds each even Fibonacci number to sumfib.
* Finally, it returns the calculated sum of even Fibonacci numbers.

**Algorithm Design :**

* The function uses Binet's formula to calculate the ith Fibonacci number.
* It uses a for loop to iterate through the Fibonacci sequence, starting from 1 and going up to num.
* In each iteration:
  + The current Fibonacci number is calculated using Binet's formula.
  + The function checks if the current Fibonacci number is even using the modulo operator (%).
  + If the number is even, it is added to the sumfib variable.
* The sumfib variable is returned at the end of the loop.

**Edge Cases Handled :**

* **Negative Input:** The function checks for negative input. If the input is negative, it returns -1.
* **Zero Input:** The function handles zero input correctly. It returns 0 if the input is 0.

**Harshad Number**

**Function Purpose:**

* The isHarshad(int num) function checks if a given integer num is a Harshad number.
* A Harshad number is a number that is divisible by the sum of its digits.

**Algorithm Design:**

* The function calculates the sum of digits of the number using the SumOfDigits(num) function (assuming it's already defined).
* It checks if the number is divisible by the sum of its digits, using the modulus operator (%).
* If the number is divisible by the sum of its digits and greater than 0, or if the number is 0, the function returns true indicating the number is a Harshad number.
* Otherwise, the function returns false, indicating that the number is not a Harshad number.

**Edge Cases Handled:**

* **Zero:** The function correctly handles the edge case where the input is 0. 0 is considered a Harshad number.
* **Negative Numbers:** The function handles negative numbers correctly be checking num > 0 to return ‘**true**’ ,this is because negative numbers are not included in the Harshad numbers.

**Catalan Number**

**Function Purpose:**

* Computes the nth Catalan number.
* The Catalan numbers appear in a variety of combinatorial structures, including:
  + Number of ways to divide a convex polygon with n + 2 sides into triangles by connecting vertices.
  + Number of possible binary search trees with n nodes.
  + Number of ways to parenthesize a product of n + 1 factors.

**Algorithm Design:**

* The function uses the following formula to calculate the nth Catalan number:

**Catalan(n) = Factorial(2 \* n) / (Factorial(n + 1) \* Factorial(n))**

where Factorial(x) represents the factorial of x.

**Steps:**

1. **Input Validation:** The function checks if the input n is less than 0. If it is, the function returns 0.
2. **Calculation:** The function calculates the nth Catalan number using the formula mentioned above. It uses the Factorial function to calculate the factorial of the required numbers.
3. **Return:** The function returns the calculated Catalan number.

**Edge Cases Handled:**

* **Invalid Input:** The function handles invalid input by returning 0 if n is less than 0.
* **Overflow:** The function uses unsigned long data type to handle large numbers. However, it's important to note that this function can still potentially cause overflow for very large values of n.

**Pascal's Triangle**

**Function Purpose:**

* The function printPascalTriangle(int n) generates and prints the first n rows of Pascal's Triangle.

**Algorithm Design:**

* The function utilizes nested loops to calculate and print the triangle's elements.

**Steps:**

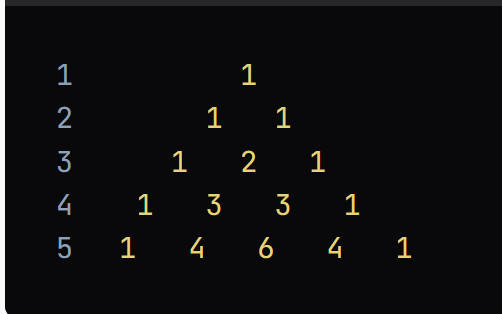
1. **Outer Loop:** Iterates n times, representing each row of the triangle.
   * **Inner Loop (Spaces):** Iterates to print spaces for alignment, creating the pyramid shape. The number of spaces decreases for each subsequent row.
   * **Inner Loop (Combinations):** Iterates to calculate and print the combination values for each row using the Combination(i, j) function.
   * **Print Newline:** After printing each row, a newline character is printed to move to the next row.

**Code Explanation:**

* **for (int i = 0; i < n; i++) {:** This loop iterates over each row of the triangle.
* **for (int space = 0; space < n - i - 1; space++) { ... }:** This loop prints spaces before each row to maintain the pyramid shape.
* **for (int j = 0; j <= i; j++) { ... }:** This loop calculates and prints the combination values for each row.
* **Combination(i, j):** This function is assumed to calculate the binomial coefficient (combination) for the given row and column.
* **printf("%d", Combination(i, j)) ;:** Prints the combination value.
* **printf("\n") ; :** Moves to the next line after printing each row.

**Example:**

If n = 5, the function would print:



**Bell Number**

**Function Purpose:**

* Calculates the nth Bell number, which represents the number of ways to partition a set of n elements.
* **Example:** bellNumber(3) returns 5, indicating there are 5 ways to partition a set of 3 elements.

**Algorithm Design**

* **Base Case:** For n = 0, the 0th Bell number is 1.
* **Recursive Calculation:** For n > 0, the nth Bell number is calculated recursively using the following formula:
  + B(n) = Σ (Combination(n-1, k) \* B(k)) for k = 0 to n-1

**Steps:**

1. **Initialization:** Initialize a variable sum to 0 to store the calculated sum.
2. **Base Case Handling:** If n is 0, return 1.
3. **Recursive Calculation:** Iterate from k = 0 to n - 1:
   * Calculate the combination of (n - 1) elements taken k at a time (Combination(n-1, k)).
   * Recursively calculate the kth Bell number (bellNumber(k)).
   * Multiply the combination and the kth Bell number and add it to the sum.
4. **Return:** Return the calculated sum representing the nth Bell number.

**Edge Cases :**

* **Negative Input:** The function handles negative input, as it returns -1 .
* **Base Case:** The base case for n = 0 is correctly implemented.
* **Large Values of n:** The function does not handle large values of n well due to the recursive nature, which can lead to stack overflow and performance issues.

**EXAMPLE :**

**Step 1: Understanding Bell Numbers**

The Bell numbers count the number of ways to partition a set of n elements. The first few Bell numbers are:

* B(0) = 1
* B(1) = 1
* B(2) = 2
* B(3) = 5
* B(4) = 15

**Step 2: Calculate B(3)**

We will call bellNumber(3).

1. **Base Case Check**:
   * Since n is not 0, we proceed to the loop.
2. **Initialize Sum**:
   * sum = 0
3. **Loop from k = 0 to n-1 (k = 0 to 2):**
   * For each k, we calculate Combination(n-1, k) and bellNumber(k).

**Iteration 1: k = 0**

* Calculate Combination(2, 0):
  + This is 1 (there's one way to choose 0 elements from 2).
* Calculate bellNumber(0):
  + This returns 1 (base case).
* Update sum:
  + sum += 1 \* 1 = 1

**Iteration 2: k = 1**

* Calculate Combination(2, 1):
  + This is 2 (there are two ways to choose 1 element from 2).
* Calculate bellNumber(1):
  + This returns 1 (base case).
* Update sum:
  + sum += 2 \* 1 = 2
  + Current sum = 1 + 2 = 3

**Iteration 3: k = 2**

* Calculate Combination(2, 2):
  + This is 1 (there's one way to choose 2 elements from 2).
* Calculate bellNumber(2):
  + We need to calculate bellNumber(2) now.

**Step 3: Calculate B(2)**

We will call bellNumber(2).

1. **Base Case Check**:
   * Since n is not 0, we proceed to the loop.
2. **Initialize Sum**:
   * sum = 0
3. **Loop from k = 0 to n-1 (k = 0 to 1)**:
   * For each k, we calculate Combination(n-1, k) and bellNumber(k).

**Iteration 1: k = 0**

* Calculate Combination(1, 0):
  + This is 1.
* Calculate bellNumber(0):
  + This returns 1.
* Update sum:
  + sum += 1 \* 1 = 1

**Iteration 2: k = 1**

* Calculate Combination(1, 1):
  + This is 1.
* Calculate bellNumber(1):
  + This returns 1.
* Update sum:
  + sum += 1 \* 1 = 1
  + Current sum = 1 + 1 = 2

**Step 4: Return B(2)**

* The sum for bellNumber(2) is 2, so we return 2.

**Step 5: Continue Calculating B(3)**

Now we return to our calculation of bellNumber(3).

* For k = 2, we have:
  + Combination(2, 2) = 1
  + bellNumber(2) = 2 (calculated above).
* Update sum:
  + sum += 1 \* 2 = 2
  + Current sum = 3 + 2 = 5

**Step 6: Return B(3)**

* The final sum for bellNumber(3) is 5, so we return 5.

**Conclusion**

Thus, the 3rd Bell number B(3) is 5, which matches the known value. The function works recursively, breaking down the problem into smaller subproblems until it reaches the base case.

**Kaprekar Number**

**Function Purpose**

* The isKaprekar(int num) function checks if the given number is a Kaprekar number. A Kaprekar number is a non-negative integer whose square can be split into two parts that add up to the original number.

**Algorithm Design**

The function works by:

1. **Calculating the square of the input number.**
2. **Determining the number of digits in the square.**
3. **Splitting the square into two parts based on the number of digits.**
4. **Checking if the sum of the two parts equals the original number.**

**Steps :**

1. **Calculate the square of the input number:** int NUM = power(num, 2);
2. **Calculate the number of digits in NUM and divide it by 2:** int K = NumberOfDigits(NUM) / 2;
3. **Initialize a variable i to 1 to determine the divisor for splitting NUM.**
4. **Check if the number of digits in NUM is odd:** If it is, the number is not a Kaprekar number, so the function returns false.
5. **Create the divisor i to split NUM based on the number of digits:** The loop iterates through each digit in the square, multiplying i by 10 for each digit. This is equivalent to moving the decimal point one place to the right for each iteration, effectively determining the divisor for the left part of the split square.
6. **Split NUM into two parts:**
   * **Left part:** NUM / i
   * **Right part:** NUM % i
7. **Check if their sum equals the original number:**
   * If the sum of the two parts equals the original number, the function returns true.
   * If the sum is not equal, the function returns false.

**Edge Cases Handled L**

* **Odd number of digits in the square:** The function handles the case where the square has an odd number of digits. Since the square cannot be divided into two equal parts in this case, it is not a Kaprekar number and the function returns false.

**Example**

**Input:** num = 9 **Output:** true

**Explanation:**

* The square of 9 is 81.
* 8+1 = 9

**Smith Number**

**Function Purpose**

* The isSmith(int num) function determines whether the given number is a Smith number.
* **Smith Number**: A Smith number is a composite number whose sum of digits is equal to the sum of the digits of its prime factors.

**Algorithm Design**

* **Initialize Variables:**
  + sumofprimefactors: Stores the sum of digits of the prime factors of the given number.
  + primefactors: Stores the product of all the prime factors of the given number.
  + i: Represents the current number being checked for prime factorization.
* **Handle Edge Cases:**
  + If the given number is less than or equal to 1 or a prime number, it is not a Smith number and the function returns false.
* **Factorization and Digit Sum Calculation:**
  + Iterate through potential prime factors (i) starting from 2.
  + If i is a prime factor of num, update sumofprimefactors by adding the sum of digits of i and update primefactors by multiplying it with i.
  + Divide num by i to continue factorization.
* **Final Check:**
  + After factorization, return true if sumofprimefactors is equal to the sum of digits of primefactors, indicating that the given number is a Smith number; otherwise, return false.

**Edge Cases Handled :**

* **Non-composite numbers:** The function returns false for 0, 1,negative and prime numbers, as they cannot be Smith numbers.

**Example:**

* Input: 1234
* Output: 10

**Steps:**

1. **Initialize sum to 0:** sum = 0
2. **Extract last digit:** last\_digit = 1234 % 10 = 4
3. **Add last digit to sum:** sum = 0 + 4 = 4
4. **Update number:** num = 1234 / 10 = 123
5. **Repeat steps 2-4:**
   * last\_digit = 123 % 10 = 3
   * sum = 4 + 3 = 7
   * num = 123 / 10 = 12
   * last\_digit = 12 % 10 = 2
   * sum = 7 + 2 = 9
   * num = 12 / 10 = 1
   * last\_digit = 1 % 10 = 1
   * sum = 9 + 1 = 10
   * num = 1 / 10 = 0
6. **Return sum:** 10

**Sum Of Primes**

**Function Purpose:**

* The sumOfPrimes(int n) function calculates the sum of all prime numbers up to a given integer n.

**Algorithm Design:**

* **Initialization:**
  + A variable sum is initialized to 0 to store the sum of prime numbers.
* **Iteration:**
  + The function iterates through all numbers from 2 to n using a for loop.
* **Prime Number Check:**
  + For each number i in the loop, the function checks if i is a prime number using the isPrime(i) function.
* **Sum Calculation:**
  + If i is a prime number, it is added to the sum.
* **Return:**
  + After the loop completes, the function returns the calculated sum.

**Edge Cases Handled:**

* **Input Validation:** The function implicitly handles negative inputs and invalid inputs (e.g., 0 or 1) by not adding them to the sum because they are not prime numbers. The function simply returns 0 if n is less than 2.

**Example:**

**Input:** 12345

**Steps:**

1. **Initialize sum to 0.**
2. **Extract the last digit:** 12345 % 10 = 5
3. **Add the extracted digit to sum:** sum = 0 + 5 = 5
4. **Update the number:** 12345 / 10 = 1234
5. **Repeat steps 2-4 until the number becomes 0:**
   * Extract the last digit: 1234 % 10 = 4
   * Add the extracted digit to sum: sum = 5 + 4 = 9
   * Update the number: 1234 / 10 = 123
   * Extract the last digit: 123 % 10 = 3
   * Add the extracted digit to sum: sum = 9 + 3 = 12
   * Update the number: 123 / 10 = 12
   * Extract the last digit: 12 % 10 = 2
   * Add the extracted digit to sum: sum = 12 + 2 = 14
   * Update the number: 12 / 10 = 1
   * Extract the last digit: 1 % 10 = 1
   * Add the extracted digit to sum: sum = 14 + 1 = 15
   * Update the number: 1 / 10 = 0
6. **Return the computed sum:** 15

**Output:** 15

THANK YOU FOR READING THE REPORT

**NOTE :**

* This project took a lot of effort from us so please mark our project fairly