Zengla mohammed abdel illah

C library

Documentation

Algorithms and Data Structures

National Higher School of Cubersecurity

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Purpose of the report

The primary purpose of this report is to document the design, implementation, and testing of a comprehensive library of numerical functions. This library is designed to provide a wide range of functionalities, from basic arithmetic operations to advanced mathematical computations, which can be reused in various programming scenarios. This report also serves as a technical reference, offering detailed explanations of each function, its algorithm, and its usage. Additionally, it highlights the coding standards, methodologies, and testing strategies adopted during development.

1.2. Objectives

The objectives of this project and its associated report are as follows:

1. To develop a robust, reusable library of numerical functions using the C programming language.
2. To ensure the correctness and efficiency of each function through rigorous testing.
3. To provide clear and comprehensive documentation for each function, including its purpose, algorithm, and usage examples.
4. To demonstrate good programming practices by adhering to coding standards and modular design principles.
5. To deliver a structured report that facilitates easy understanding of the library for academic evaluation and practical use.

1.3. Overview of the Library

This library is a collection of 30+ numerical functions categorized based on their complexity and purpose:

1. Basic Operations on Numbers: These include functions for fundamental tasks such as digit sum computation, prime checking, and factorial calculation.

1. INTRODUCTION

1.Intermediate Operations on Numbers: These cover slightly more complex tasks, such as prime factorization, Armstrong number verification, and Fibonacci sequence generation.

2.Advanced Operations on Numbers: These encompass sophisticated algorithms, such as calculating Catalan numbers, generating Pascal's triangle, and determining if a number is a Kaprekar or Smith number. The library is modularly designed, allowing for the easy addition of new functions. It is implemented in a single header file, making it lightweight and simple to integrate into other C projects. Each function is tested using sample cases to ensure reliability and accuracy.

1.3. Description of the Functions

Basic Operations on Numbers

- Sum of Digits (int sumOfDigits(int num)): Calculates the sum of all digits in a number. Reverse Number (int reverseNumber(int num)): Reverses
- the digits of a given number while preserving its sign. Palindrome (bool isPalindrome(int num)): Checks if a number reads the same forwards
- and backwards. Prime (bool isPrime(int num)): Determines if a number is prime using optimized logic. Greatest Common Divisor (GCD) (int gcd(int
- a, int b)): Computes the GCD of two numbers using the Euclidean algorithm, which repeatedly divides the larger number by the smaller
- one until a remainder of zero is reached. Least Common Multiple (LCM) (int lcm(int a, int b)): Calculates the LCM of two numbers, which is the smallest number that is divisible by both a and b. Factorial (long factorial(int num)): Computes the factorial of a number, which is the
- product of all positive integers up to that number. This includes handling potential overflow when computing large factorials. Even/Odd (bool isEven(int num)): Returns TRUE if the number is even (i.e., divisible by 2), and FALSE otherwise.
-

-

0. INTRODUCTION

Intermediate Operations on Numbers

- Prime Factorization (void primeFactors(int num)): Prints all prime factors of a number by dividing it by prime numbers starting from 2 and continuing until the number becomes 1.
- Armstrong Number (bool isArmstrong(int num)): Checks if a number is an Armstrong number, where the sum of its digits raised to the power of the number of digits equals the number itself (e.g., 153 because 13+53+33=1531^3 + 5^3 + 3^3 = 15313+53+33=153).
- Fibonacci Sequence (void fibonacciSeries(int n)): Generates the Fibonacci sequence up to the nth term, where each term is the sum of the two preceding ones, starting from 0 and 1.
- Sum of Divisors (int sumDivisors(int num)): Calculates the sum of all divisors of a number, which are the numbers that divide it without a remainder.
- Perfect Number (bool isPerfect(int num)): Checks if a number is perfect, meaning the sum of its proper divisors (excluding the number itself) equals the number (e.g., 6 because 1+2+3=61 + 2 + 3 = 61+2+3=6).
- Magic Number (bool isMagic(int num)): Checks if the sum of the digits of a number, when recursively summed until a single digit is obtained, equals 1 (e.g., 19 because 1+9=101 + 9 = 101+9=10 and 1+0=11 + 0 = 11+0=1).
- Automorphic Number (bool isAutomorphic(int num)): Checks if a number is automorphic, meaning its square ends with the number itself (e.g., 25 because 252=62525^2 = 625252=625, and the last two digits are 25).
-

Advanced Operations on Numbers

- Binary Conversion (void toBinary(int num)): Converts a number to its binary representation by repeatedly dividing the number by 2 and collecting the remainders. Narcissistic Number (bool isNarcissistic(int
- num)): Checks if a number is narcissistic, where the number equals the sum of its digits raised to the power of the number of digits (e.g., 370 because 33+73+03=3703^3 + 7^3 + 0^3 = 37033+73+03=370).

0. INTRODUCTION

- Square Root Calculation (double sqrtApprox(int num)): Calculates the square root of a number using the Babylonian method (also known as Heron's method), which iteratively approximates the square root.
- Exponentiation (double power(int base, int exp)): Calculates the power of a base raised to an exponent, using the formula baseexp\text{base}^{\text{exp}}baseexp. Happy Number (bool
- isHappy(int num)): Checks if a number is happy by repeatedly summing the squares of its digits. If the process eventually leads to 1, the number is happy. Abundant Number (bool isAbundant(int num)): Checks if a
- number is abundant, meaning the sum of its proper divisors exceeds the number itself. Deficient Number (bool isDeficient(int num)): Checks if a number is deficient, meaning the sum of its proper divisors is less than
- the number itself. Sum of Fibonacci Even Numbers (int sumEvenFibonacci(int n)): Calculates the sum of even Fibonacci numbers up to the nth term in the Fibonacci sequence. Harshad Number
- (bool isHarshad(int num)): Checks if a number is divisible by the sum of its digits (also known as a Niven number). Catalan Number Calculation (unsigned long catalanNumber(int n)): Computes the nth Catalan
- number, which is used in combinatorics to count the number of possible binary search trees that can be formed with n nodes. Pascal Triangle
- (void pascalTriangle(int n)): Generates the first n rows of Pascal's Triangle, a triangular array of binomial coefficients. Bell Number (unsigned long bellNumber(int n)): Computes the nth Bell number, which counts the number of ways to partition a set of n elements. Kaprekar Number (bool isKaprekar(int num)): Checks if a number is Kaprekar,
- meaning the square of the number can be split into two parts that sum to the original number (e.g., 45 because 452=202545^2 = 2025452=2025
- and 20+25=4520 + 25 = 4520+25=45). Smith Number (bool isSmith(int num)): Checks if a number is a Smith number, which is a non-prime number whose sum of digits equals the sum of the digits of its prime
- factors. Sum of Prime Numbers (int sumOfPrimes(int n)): Calculates the sum of all prime numbers less than or equal to n by checking each number for primality and summing the primes.

-

-

0. INTRODUCTION

1.3. Coding Standards and Guidelines

The development of the library follows strict coding standards to ensure high-quality code that is easy to read, understand, and maintain. 1.5.1 Code Formatting

- Indentation: Consistent use of 4 spaces per indentation level.
- Naming Conventions:
  - Functions: CamelCase (e.g., isPrime, reverseNumber).
  - Variables: Descriptive camelCase (e.g., num, reversed).
  - Constants: Uppercase with underscores (e.g., MAX_VALUE).
- Comments: Inline and block comments are used to explain the purpose and logic of the code.

1.5.2 Error Handling

- Functions handle invalid or edge-case inputs, such as negative values or overflow scenarios.
- Logical checks are included to prevent undefined behavior, ensuring robustness.

1.5.3 Optimization

- Algorithms are optimized for performance (e.g., skipping unnecessary checks in prime number detection).
- Efficient looping and conditional structures reduce computational overhead.

1.5.4 Examples of Functions Each function includes detailed comments, proper error handling, and illustrative examples to demonstrate its usage and output. Refer to Section 2 for full implementation and examples of the library's functions.

-

1. OPERATION ON NUMBERS

01 sumOfDigits(int num)

1)Purpose

This function calculates the sum of digits ina number. For example,123 gives 1+2+3 = 6.It works for both positive and negative nums.

2)Inputs / Outputs

- Input: `int num`: Any positive or negative whole number.
- Output: Returns the sum of all digits in the input number.

3)Logic and Steps

1. If the number is negative, make it pos.
   - Example: -78 becomes 78.
2. Create a variable `sum` and set it to 0.
3. Use a loop to process each digit:
   - Find the last digit with `num % 10`.
   - Add it to `sum`.
   - Remove the last digit using `num / 10`.
4. Repeat the loop until `num` becomes 0.
5. Return the final value of `sum`.

4)Edge Cases

- If num is 0, the output will be 0.
- Negative numbers are handled by converting them to positive.
- Single digit numbers return themselves, e.g. input = 5, result = 5.

1. OPERATION ON NUMBERS

02 reverseNumber(int num)

1)Purpose

This function reverses the digits of a number. For example, if input is 123, the output is 321.It handles both positive and negative numbers.

2)Inputs / Outputs

- Input: `int num`: A whole number, can be positive or negative.
- Output: Returns the number with its digits reversed.

3)Logic and Steps

1. Check if the number is negative.
   - If yes, make it positive and set `positive` to `false`.
2. Create a variable `reversed` to store the reversed number, starting from 0.
3. Use a loop to reverse the digits:
   - Multiply `reversed` by 10 and add the last digit of `num` using `num % 10`.
   - Remove the last digit of `num` by dividing it by 10.
4. Repeat the loop until `num` becomes 0.
5. If the number was negative, return the reversed number as negative, otherwise return it as positive.

4)Edge Cases

- If the number is 0, the result is also 0.
- Single-digit numbers remain the same, e.g. input = 7, output = 7.
- Negative numbers are handled correctly, e.g. input = -123, output = -321.

1. OPERATION ON NUMBERS

03 isPalindrome(int num)

1)Purpose

This function checks if a number is a palindrome. A number is considered a palindrome if it reads the same forward and backward.

2)Inputs / Outputs

- Input: `int num`: A whole number, which can be positive or negative.
- Output: Returns `true` if the number is a palindrome, `false` otherwise.

3)Logic and Steps

1. The function compares the number with its reversed version.
2. It calls `reverseNumber(num)` to reverse the digits of `num`.
3. If `num` is equal to its reversed version, the number is a palindrome, and the function returns `true`.
4. If not, the function returns `false`.

4)Edge Cases

- If `num` is 0, it is considered a palindrome.
- Negative numbers are handled by considering their absolute value, e.g. -121 is treated as 121.
- Single-digit numbers are always palindromes.

1. OPERATION ON NUMBERS

04 isPrime(int num))

1)Purpose

This function checks if a number is prime.A prime number is a number greater than 1 that can only be divided by 1 and itself. For example,5 is prime, but 4 is not.

2)Inputs / Outputs

- Input: `int num`: A whole number to check.
- Output: Returns `true` if the number is prime, otherwise `false`.

3)Logic and Steps

1. Handle edge cases:
   - If `num <= 1`, return `false` (not prime).
   - If `num <= 3`, return `true` (prime).
2. If `num` is divisible by 2 or 3, return `false`.
3. Start a loop from `i = 5` and check if `num` is divisible by `i` or `i + 2`.
   - The loop only checks up to `sqrt(num)`(i.e., `i * i <= num`).
   - `i` increments by 6 in each iteration (`i += 6`).
4. If `num` is divisible by any `i` or `i + 2`, return `false`.
5. If no divisors are found, return `true`.

4)Edge Cases

- Numbers less than or equal to 1 are not prime.
- 2 and 3 are prime numbers.
- Large prime numbers are handled correctly by checking up to the square root of `num`.

1. OPERATION ON NUMBERS

05 gcd(int a, int b)

1)Purpose

This function calculates the Greatest Common Divisor (GCD) of two numbers. The GCD is the largest number that divides both `a` and `b` without leaving a remainder.For example, GCD of 12 and 18 is 6.

2)Inputs / Outputs

- Input: `int a, int b`: Two whole numbers to find the GCD of.
- Output: Returns the GCD of `a` and `b`.

3)Logic and Steps

1. If either `a` or `b` is negative, change the to positive using the ternary operator.
2. If `b` is greater than `a`, swap the values of `a` and `b`.
3. Use the Euclidean algorithm:
   - While `b` is not zero, do the following:
     - Set `a` to `b`.
     - Set `b` to the remainder of `a % b`.
4. When `b` becomes 0, `a` holds the GCD value.
5. Return `a`.

4)Edge Cases

- If `a` or `b` is 0, the GCD is the absolute value of the other number.
- If both `a` and `b` are 0, the result is undefined,but the function might return 0.

1. OPERATION ON NUMBERS

06 lcm(int a, int b)

1)Purpose

This function calculates the Least Common Multiple (LCM) of two numbers. The LCM is the smallest numberthat both `a` and `b` can divide evenly into. For example, the LCM of 4 and 5 is 20.

2)Inputs / Outputs

- Input: `int a, int b`: Two whole numbers to find the LCM of.
- Output: Returns the LCM of `a` and `b`.

3)Logic and Steps

1. If either `a` or `b` is negative, change them to positive using the ternary operator.
2. Use the formula:
   - `LCM(a, b) = (a * b) / GCD(a, b)`
   - The function calls `gcd(a, b)` to get the greatest common divisor.
3. Return the result of `(a * b) / gcd(a, b)`.

4)Edge Cases

- If either `a` or `b` is 0, the LCM is 0.
- If both numbers are 0, the result is undefined,but the function might return 0.

1. OPERATION ON NUMBERS

07 factorial(int num)

1)Purpose

This function calculates the factorial of a number. The factorial of a number `n` is the product of all positive integers less than or equal to `n`. For example,factorial of 5 is 5 * 4 * 3 * 2 * 1 = 120.

2)Inputs / Outputs

- Input: `int num`: The number for which the factorial is calculated.
- Output: Returns the factorial of `num` as an unsigned long long integer, or `-1` if overflow occurs.

3)Logic and Steps

1. If `num` is negative, return 0, as factorial is not defined for negative numbers.
2. Initialize `fact` to 1, as the factorial starts at 1.
3. Use a loop to multiply each integer from 2 to `num` with `fact`.
4. After each multiplication, check if the result exceeds the maximum value of an unsigned long long integer (`LLONG_MAX`). If it does, return `-1` to indicate an overflow.
5. Return the final value of `fact`.

4)Edge Cases

- If `num` is 0, the factorial is 1, as `0! = 1`.
- Negative numbers return 0 because factorial is undefined for them.
- If the factorial exceeds the maximum value, `-1` is returned.

Folder: 08 Even Odd Check

1. OPERATION ON NUMBERS

08 isEven(int num)

1)Purpose

This function checks if a number is even. A number is even if its least significant bit (LSB) is 0. The functionworks for both positive and negative numbers.

2)Inputs / Outputs

- Input: `int num`: A whole number to check.
- Output: Returns `true` if the number is even, otherwise returns `false`.

3)Logic and Steps

1. The function uses the bitwise AND operator (`&`) to check the least significant bit (LSB) of the number.
2. If the LSB is 0 (i.e., `num & 1 == 0`), the number is even, and the function returns `true`.
3. If the LSB is 1 (i.e., `num & 1 != 0`), the number is odd, and the function returns `false`.

4)Edge Cases

- If `num` is 0, the result is `true` because 0 is considered even.
- The function works for both positive and negative numbers, as the LSB is checked directly using bitwise operations.

1. OPERATION ON NUMBERS

09 primeFactors(int num)

1)Purpose

This function prints all the prime factors of a number.A prime factor is a prime number that divides the numberexactly without leaving a remainder. For example, the prime factors of 18 are 2 and 3.

2)Inputs / Outputs

- Input: `int num`: The number whose prime factors need to be found.
- Output: Prints the prime factors of `num` to the console.

3)Logic and Steps

1. If `num` is less than or equal to 1, print a message that the number has no prime factors.
2. Print a message saying "Prime factors of `num` are :".
3. Use a loop starting from 2 to check divisibility:
   - For each `i` (from 2 to `num`), if `num` is divisible by `i`, print `i` as a prime factor.
   - Keep dividing `num` by `i` until `num` is no longer divisible by `i` to ensure the prime factor is counted only once.
4. Use a `first` flag to control the formatting of the printed prime factors (to avoid printing a comma before the first factor).
5. After all factors are printed, print a newline.

4)Edge Cases

- If `num` is 0 or 1, the function will print "has no prime factors".
- If `num` is a prime number, it will print the number itself as its only prime factor.
- If `num` is a product of prime numbers (e.g., 18), all the prime factors will be printed.

1. OPERATION ON NUMBERS

10 isArmstrong(int num)

1)Purpose

This function checks if a number is an Armstrong number. An Armstrong number is a number that is equal to the sum of its own digits each raised to the power of the number of digits.For example, 153 is an Armstrong number because 1^3 + 5^3 + 3^3 = 153.

2)Inputs / Outputs

- Input: `int num`: A number to check.
- Output: Returns `true` if the number is Armstrong, otherwise returns `false`.

3)Logic and Steps

1. If `num` is negative, return `false` as Armstrong numbers are non-negative.
2. Count the number of digits in `num`.
3. Initialize `sum` to 0, which will hold the sum of each digit raised to the power of `digits`.
4. For each digit in `num`, calculate `digit^digits` and add it to `sum`.
5. If the sum equals the original number, return `true`, otherwise return `false`.

4)Edge Cases

- Negative numbers return `false` as they cannot be Armstrong.
- Single-digit numbers are always Armstrong numbers (e.g., 5 is Armstrong).
- If the sum of digits raised to the power of their count equals the number, it is Armstrong (e.g., 153).

1. OPERATION ON NUMBERS

11 fibonacciSeries(int n)

1)Purpose

This function prints the Fibonacci series up to a given number `n`. The Fibonacci series starts with0 and 1, and each subsequent number is the sum of the previous two. For example, if `n` is 10, the series will be 0, 1, 1, 2, 3, 5, 8.

2)Inputs / Outputs

- Input: `int n`: The limit up to which the Fibonacci series is printed.
- Output: Prints the Fibonacci series up to `n`.

3)Logic and Steps

1. If `n` is negative, print "invalid input!!".
2. If `n` is 0, print "0" since the Fibonacci series for 0 only contains 0.
3. If `n` is greater than 0, start printing the series starting with 0 and 1.
4. Use two variables `n1` and `n2` to hold the two previous numbers in the series (starting with 0 and 1).
5. Use a while loop to keep adding the numbers in the series until `n2` is greater than `n`.
6. After each addition, print the current number, update `n1` and `n2`, and calculate the next number.
7. Print the series and finish with a newline.

4)Edge Cases

- If `n` is negative, the function prints an error message.
- If `n` is 0, the function prints "0".
- The function prints Fibonacci numbers up to and including `n` if they are part of the series.

1. OPERATION ON NUMBERS

12 sumDivisors(int num)

1)Purpose

This function calculates the sum of divisors of a given number. A divisor is a number that can divide `num` without leaving a remainder. For example, the divisors of 6 are 1, 2, 3, and 6,and their sum is 12.

2)Inputs / Outputs

- Input: `int num`: A number to calculate divisorsfor.
- Output: Returns the sum of divisors of `num`, or `-1` if `num` is negative.

3)Logic and Steps

1. If `num` is negative, return `-1` to indicate an error.
2. Initialize `sum` to `num` as the number itself is always a divisor.
3. Use a loop to check divisibility for each number from 1 to `num / 2`.
4. If `i` divides `num` exactly, add `i` to `sum`.
5. After the loop, return the total sum of divisors.

4)Edge Cases

- If `num` is 0, the sum of divisors is undefined
  but the function may return a large number since
  0 is divisible by all integers.
- Negative numbers return `-1` as an error.
- Single-digit numbers will only include small divisors.

1. OPERATION ON NUMBERS

13 isPerfect(int num)

1)Purpose

This function checks if a number is a perfect number.A perfect number is a number that is equal to the sum of its proper divisors (divisors excluding the number itself).For example, 6 is a perfect number because its divisors(1, 2, 3) add up to 6.

2)Inputs / Outputs

- Input: `int num`: A number to check if it's perfect.
- Output: Returns `true` if the number is perfect,otherwise returns `false`

3)Logic and Steps

1. If `num` is less than or equal to 0, return `false`since perfect numbers must be positive.
2. Initialize `sum` to 0, which will store the sum ofdivisors of `num`.
3. Use a loop to check for divisors of `num` from 1 to`num / 2`.
4. If `i` divides `num`, add `i` to `sum`.
5. After the loop, compare `sum` with `num`. If they areequal, return `true`, otherwise return `false`.

4)Edge Cases

- If `num` is 0 or negative, the function returns `false`.
- Small numbers like 1 will return `false` as they have no proper divisors.
- Numbers like 6, 28, and 496 are perfect numbers.

1. OPERATION ON NUMBERS

14 isMagic(int num)

1)Purpose

This function checks if a number is a magic number.A magic number is defined as a number whose digital root (the sum of its digits repeated iteratively) equals 1.
For example, 19 is a magic number because 1 + 9 = 10 and1 + 0 = 1.

2)Inputs / Outputs

- Input: `int num`: A number to check if it's magic.
- Output: Returns `true` if the number is magic,otherwise returns `false`

3)Logic and Steps

1. If `num` is negative, return `false` since we considernegative numbers as non-magic.
2. Check if `num % 9 == 1`. This checks if the digital rootof `num` equals 1.
3. If `num % 9 == 1`, return `true`, otherwise return `false`

4)Edge Cases

- If `num` is 0, the function returns `false` because 0 is not considered a magic number.
- For numbers like 19, 28, 37, etc., the function will return `true`.

1. OPERATION ON NUMBERS

15 isAutomorphic(int num)

1)Purpose

This function checks if a number is an automorphic number.An automorphic number is a number whose square ends with the number itself. For example, 25 is an automorphic number because 25^2 = 625, and 625 ends with 25.

2)Inputs / Outputs

- Input: `int num`: A number to check if it's automorphic.
- Output: Returns `true` if the number is automorphic,otherwise returns `false`

3)Logic and Steps

1. If `num` is negative, return `false` becauseautomorphicnumbers are non-negative.
2. Initialize `factor` to 1, which will be used to isolatethe last digits of the square of `num`.
3. Use a loop to calculate the number of digits in `num`. The loop divides `num` by 10 repeatedly, and `factor` is multiplied by 10 in each iteration.
4. After the loop, `factor` will be equal to 10 raised tothe power of the number of digits in `num`.
5. Check if the square of `num` modulo `factor` equals `num`.If true, return `true`, otherwise return `false`

4)Edge Cases

- If `num` is 0, return `true` because 0 is an automorphic number.
- Single-digit numbers are automorphic if their square ends in the same number.
- Larger numbers can also be automorphic if the square ends with the number itself.

1. OPERATION ON NUMBERS

16 toBinary(int num)

1)Purpose

This function converts a decimal number to itsbinary representation and prints the result.For example, 5 will be converted to 101.

2)Inputs / Outputs

- Input: `int num`: An integer number to convert to binary.
- Output: Prints the binary representation of `num` to the console.

3)Logic and Steps

1. Initialize `binary` to 0 to store the binary result, and `i` to 0 for the place value.
2. While `num` is greater than 0, do the following
   - Find the remainder when `num` is divided by 2 (the least significant bit).
   - Add this bit to `binary`, shifting it to the correct place by multiplying by `10^i`.
   - Divide `num` by 2 to process the next bit.
   - Increment `i` to move to the next place.
3. Print the binary result.

4)Edge Cases

- If `num` is 0, the output will be `0`.
- the function assumes `num` is a positive integer; negative numbers are not handled. .

1. OPERATION ON NUMBERS

17 bool isNarcissistic(int num)

1)Purpose

This function checks if a number is a narcissisticnumber, which is the same as an Armstrong number.A narcissistic number is equal to the sum of its digits each raised to the power of the number of digits.For example, 153 is a narcissistic number because1^3 + 5^3 + 3^3 = 153.

2)Inputs / Outputs

- Input: `int num`: A number to check if it's narcissistic.
- Output: Returns `true` if the number is narcissistic, otherwise returns `false`.

3)Logic and Steps

1. The function calls `isArmstrong(num)` to check if the number is an Armstrong number, as both concepts are mathematically the same.
2. Return the result from `isArmstrong(num)`.

4)Edge Cases

- If `num` is 0 or a single-digit number, the function will return `true` because single-digit numbers are narcissistic.
- Negative numbers are not considered narcissistic.

1. OPERATION ON NUMBERS

18 sqrtApprox(int num)

1)Purpose

This function calculates the square root of a number using Newton's method (also known as the Babylonian method) for approximation. It continues until the difference between successive guesses is within a certain accuracy. For example, for `num = 16`, the function will return approximately `4.0`.

2)Inputs / Outputs

- Input: `int num`: A number to find the square root of.
- Output: Returns a `double` that approximates the square root of `num`.

3)Logic and Steps

1. Initialize `x0` as half of `num`, which is the first guess.
2. Set `accuracy` to `0.00001` to define the desired precision.
3. In the `while` loop, apply the Babilonian formula:
   `x1 = 0.5 *(x0 +(num /x0))`.
4. Check if the relative difference between `x1` and `x0` is within the accuracy limit. If it is, exit the loop.
5. Update `x0` to `x1` and repeat until the accuracy condition is met.
6. Return `x1` as the square root approximation.

4)Edge Cases

- If `num` is 0, the square root is 0.
- The function assumes `num` is a positive integer; negative numbers are not handled.
- The function may return incorrect results for very large or very small numbers due to precision limits.

1. OPERATION ON NUMBERS

19 power(int base, int exp)

1)Purpose

This function calculates the power of a base raised to an exponent (i.e., `base^exp`). It handles both positive and negative exponents. For example, `2^3` equals 8, and `2^-3` equals 0.125.

2)Inputs / Outputs

- Input: `int base`: The base number.
- Input: `int exp`: The exponent to raise the base to.
- Output: Returns a `double` representing the result of `base^exp`.

3)Logic and Steps

1. If both `base` and `exp` are 0, return 1 as it's mathematically undefined, but conventionally set to 1
2. If `base` is 0 and `exp` is negative, return 0 to avoid division by zero.
3. If `exp` is 0, return 1, as any number raised to 0 is 1.
4. Initialize `result` to 1.0, which will hold the result.
5. If `exp` is negative, make `abs_exp` the absolute value of `exp`.
6. Use a loop to multiply `result` by `base` `abs_exp` times.
7. If `exp` is negative, take the reciprocal of `result`.
8. Return `result` as the final result.

4)Edge Cases

- If `base` is 0 and `exp` is 0, it returns 1, which is a special case.
- If `base` is 0 and `exp` is negative, it returns 0, which avoids division by zero.
- If `exp` is 0, the result is always 1, regardless of the base.
- Negative exponents are handled by taking the reciprocal.

1. OPERATION ON NUMBERS

20 isHappy(int num)

1)Purpose

This function checks if a number is a happy number.
A happy number is a number where, if you repeatedly sum the squares of its digits, you eventually get 1. If the number falls into a cycle that doesn't include 1, it's not a happy number. For example, 19 is a happy number.

2)Inputs / Outputs

- Input: `int num`: The number to check if it's happy.
- Output: Returns `true` if the number is happy, otherwise returns `false`.

3)Logic and Steps

1. If `num` is less than or equal to 0, return `false` since happy numbers must be positive.
2. Initialize an array `seen` of size 1000 to track the sums encountered during the process.
3. While `num` is not equal to 1, repeat the following:
   - Calculate the sum of the squares of the digits of `num`.
   - If this sum has been seen before, return `false` as we're in a cycle.
   - Update `num` to the sum calculated.
4. If `num` eventually becomes 1, return `true` indicating the number is happy.

4)Edge Cases

- If `num` is 0 or negative, the function returns `false` as negative numbers and 0 cannot be happy numbers.
- The function assumes a limit on the size of the `seen` array and will not work correctly for very large numbers beyond the size of 1000.

1. OPERATION ON NUMBERS

21 isAbundant(int num)

1)Purpose

This function checks if a number is abundant.
An abundant number is a number for which the sum of its proper divisors (divisors excluding the number itself) is greater than the number. For example, 12 is an abundant number because the sum of its divisors (1, 2, 3, 4, 6) is 16, which is greater than 12.

2)Inputs / Outputs

- Input: `int num`: The number to check if it's abundant.
- Output: Returns `true` if the number is abundant, otherwise returns `false`.

3)Logic and Steps

1. If `num` is less than or equal to 0, return `false` because abundant numbers must be positive.
2. Call `sumDivisors(num)` to get the sum of the divisors of `num`, including `num` itself.
3. Subtract `num` from the sum to get the sum of the proper divisors.
4. If the sum of proper divisors is greater than `num`, return `true`. Otherwise, return `false`.

4)Edge Cases

- If `num` is 0 or negative, the function returns `false` because negative numbers and 0 are not abundant.

1. OPERATION ON NUMBERS

22 isDeficient(int num)

1)Purpose

This function checks if a number is deficient.
A deficient number is a number for which the sum of its proper divisors (divisors excluding the number itself) is less than the number. For example, 8 is a deficient number because the sum of its divisors (1, 2, 4) is 7, which is less than 8.

2)Inputs / Outputs

- Input: `int num`: The number to check if it's deficient
- Output: Returns `true` if the number is deficient, otherwise returns `false`.

3)Logic and Steps

1. If `num` is less than or equal to 0, return `false` because deficient numbers must be positive.
2. Call `sumDivisors(num)` to get the sum of the divisors of `num`, including `num` itself.
3. Subtract `num` from the sum to get the sum of the proper divisors.
4. If the sum of proper divisors is less than `num`, return `true`. Otherwise, return `false`.

4)Edge Cases

- If `num` is 0 or negative, the function returns `false` because negative numbers and 0 are not deficient.

1. OPERATION ON NUMBERS

23 sumEvenFibonacci(int n)

1)Purpose

This function calculates the sum of all even Fibonacci numbers up to a given number `n`. The Fibonacci sequence is generated by adding the two previous numbers together, starting with 0 and 1.For example, for `n = 10`, the even Fibonacci numbers are 2, 8, andtheir sum is 10.

2)Inputs / Outputs

- Input: `int n`: The upper limit for Fibonacci numbers to sum
- Output: Returns the sum of even Fibonacci numbers up to `n`.

3)Logic and Steps

1. If `n` is less than 2, return 0 since no Fibonacci even number exist (except 0).
2. Initialize `n1` and `n2` as the first two Fibonacci numbers (0 and 1).
3. Initialize `sum` to 0, which will store the sum of even Fibonacci numbers.
4. Loop from 0 to `n-2` to generate Fibonacci numbers.
5. For each Fibonacci number `n2`, check if it's even.
   - If it is, add it to `sum`.
6. Return the calculated sum after the loop finishes.

4)Edge Cases

- If `n` is less than 2, the function returns 0.
- The function works only for non-negative integers.
- If `n` is very large, the loop may take significant time.

1. OPERATION ON NUMBERS

24 isHarshad(int num)

1)Purpose

This function checks if a number is a Harshad number (or Niven number). A Harshad number is an integer that is divisible by the sum of its digits. For example, 18 is a Harshad number because the sum of its digits is 1 + 8 = 9, and 18 is divisible by 9.

2)Inputs / Outputs

- Input: `int num`: The number to check if it's Harshad.
- Output: Returns `true` if the number is a Harshad number, otherwise returns `false`.

3)Logic and Steps

1. If `num` is less than or equal to 0, return `false` because Harshad numbers must be positive integers.
2. Call the `sumOfDigits(num)` function to get the sum of the digits of `num`.
3. Check if `num` is divisible by the sum of its digits.
   - If it is, return `true` indicating it's a Harshad number.
   - Otherwise, return `false`.

4)Edge Cases

- If `num` is 0 or negative, the function returns `false`.
- The function assumes that `sumOfDigits` works correctly and returns the sum of the digits.

1. OPERATION ON NUMBERS

25 catalanNumber(int n)

1)Purpose

This function calculates the nth Catalan number using a recursive formula. The nth Catalan number is used in combinatorics and represents the number of distinct binary trees, valid parenthesis expressions, and other combinatorial structures. For example, the 3rd Catalan number is 5.

2)Inputs / Outputs

- Input: `int n`: The index of the Catalan number to calculate.
- Output: Returns the nth Catalan number as an unsigned long.

3)Logic and Steps

1. If `n` is less than or equal to 0, return 0 since the nth Catalan number
   is not defined for non-positive values of `n`.
2. If `n` is 1, return 1, as the base case for the first Catalan number (C0 = 1).
3. Initialize the `catalan` variable to 1, which will store the nth Catalan number.
4. Loop from 0 to `n-1` and apply the Catalan number formula:
   - `C(n) = (2 * (2*i + 1) * C(i)) / (i + 2)`
5. Return the calculated `catalan` number.

4)Edge Cases

- If `n` is 0 or negative, the function returns 0, as the Catalan number is not defined for non-positive `n`.
- If `n` is 1, the function returns 1, as the first Catalan number is 1.

1. OPERATION ON NUMBERS

26_pascalTriangle(int n)

1)Purpose

-Pascal's Triangle is a triangular array where each number is the sum of the two numbers directly above it.
-For example, the 4th row is `1 3 3 1`.

2)Inputs / Outputs

- Input: `int n`: The number of rows in the Pascal's Triangle to print.
- Output: Prints the Pascal's Triangle up to `n` rows.

3)Logic and Steps

1. Loop through rows from 0 to `n-1`.
2. In each row:
   - Initialize the first value in the row as 1.
   - Print spaces to align the numbers in a triangular shape.
3. Calculate each subsequent value in the row using the formula:
   - `value = value * (i - j + 1) / (j + 1)`. This calculates the binomial coefficient for the current position.
4. Print each value in the row followed by a space.
5. After printing the row, move to the next line for the next row.

4)Edge Cases

- If `n` is less than or equal to 0, no output is printed.
- The function may not handle very large values of `n` well due to the size of numbers in Pascal's Triangle.

1. OPERATION ON NUMBERS

27 bellNumber(int n)

1)Purpose

This function calculates the nth Bell number, which represents the number of ways to partition a set of `n` elements into non-empty subsets.

2)Inputs / Outputs

- Input: `int n`: The index of the Bell number to calculate.
- Output: Returns the nth Bell number as an unsigned long.

3)Logic and Steps

1. If `n` is 0, return 1 as the base case (B0 = 1).
2. Initialize a `bell` variable to 0, which will store the calculated Bell number.
3. Loop through values of `k` from 0 to `n-1` and apply the recursive formula:
   - `B(n)=Σ(arrangement(n-1,k)*B(k))`.
4. In the formula, `arrangement(n-1, k)` represents the number of ways to arrange `k` elements from `n-1`.
   This is calculated by the `arrangement` function
5. Add the results of the recursive calls and store the sum in `bell`.
6. Return the calculated `bell` number.

4)Edge Cases

- If `n` is 0, the function correctly returns 1.
- If `n` is negative, the function may behave incorrectly. The function assumes `n` is non-negative.
- This function relies on the `arrangement` function working correctly.

1. OPERATION ON NUMBERS

28 isKaprekar(int num)

1)Purpose

This function checks if a given number is a Kaprekar number.
A Kaprekar number is a non-negative integer for which the sum of the parts of its square equals the number itself.
For example, 45 is a Kaprekar number because $45^2 = 2025$, and $20 + 25 = 45$.

2)Inputs / Outputs

- Input: int num`: The number to check if it is a Kaprekar number.
-Output: Returns `true` if the number is a Kaprekar number, otherwise returns `false`.

3)Logic and Steps

1. If `num` is negative, return `false` because Kaprekar numbers are non-negative.
2. Compute the square of the number (`n2 = num*num`).
3. Count the number of digits in the square (`digitsNum2`) using a loop.
4. Loop through all possible ways to split the square:
   - Use a divisor based on powers of 10 to separate the square into left and right parts.
   - Calculate `left` as the integer division of `n2` by the divisor and `right` as the modulus of `n2` by the divisor.
   - If the sum of `left` and `right` equals `num`, return `true`.
5. If no valid split is found, return `false`.

4)Edge Cases

- If `num` is negative, the function immediately returns `false`.
- A `num` of 0 will be correctly identified as a non-Kaprekar number.
- For larger numbers, we already use the `long long int`

1. OPERATION ON NUMBERS

29 isSmith(int num)

1)Purpose

This function checks if a given number 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 (including multiplicity). For example, 202 is a Smith number because 2 + 0 + 2 = 2 + 0 + 1 + 0 + 1 (sum of digits of 2, 101).

2)Inputs / Outputs

-Input :`int num`: The number to check if it is a Smith number.
- Output ;Returns `true` if the number is a Smith number, otherwise returns `false`.

3)Logic and Steps

1. If `num` is less than or equal to 1, return `false` because Smith numbers are positive and greater than 1.
2. If `num` is a prime number, return `false` because prime numbers cannot be Smith numbers.
3. Compute the sum of the digits of `num` using `sumOfDigits`.
4. Factorize `num` and calculate the sum of the digits of its prime factors.
   - Start from `i = 2` and divide `num` by `i` as long as it is divisible.
   - For each factor `i`, add the sum of its digits to a running total `sumPrimesDigits`.
5. Compare the sum of the digits of `num` (`numDigits`) with the sum of the digits of its prime factors (`sumPrimesDigits`). If they are equal, return `true`.
6. Otherwise, return `false`.

4)Edge Cases

- A `num` of 1 or any prime number returns `false`

1. OPERATION ON NUMBERS

30 sumOfPrimes(int n)

1)Purpose

This function calculates the sum of all prime numbers less than or equal to a given number `n`. Prime numbers are numbers greater than 1 that have no divisors other than 1 and themselves.

2)Inputs / Outputs

- Input:`int n`: The upper limit for finding prime numbers.
- Output: Returns the sum of all prime numbers less than or equal to `n`.

3)Logic and Steps

1. If `n` is less than or equal to 1, return `0` because there are no prime numbers in this range.
2. If `n` is equal to 2, return `2` because 2 is the only even prime number.
3. Initialize the sum as `2`, since `n > 2` means the number 2 is included in the sum.
3. Use a loop starting from `3` to `n`, incrementing by `2` to skip even numbers (only odd numbers need to be checked after 2).
   - Check if each number is prime using the `isPrime` function.
   - If the number is prime, add it to the sum.
5. Return the final sum.

4)Edge Cases

- If `n = 0` or `n = 1`, the function returns `0`.
- Handles the smallest prime number, 2, separately.

2. OPERATION ON ARRAYS

01 initializeArray(int arr[], int size, int value)

1)Purpose

The function initializes all elements of an array to a specific value.

2)Inputs / Outputs

- Inputs:
  `int arr[]`: The array to be initialized.
  `int size`: The number of elements in the array.
  `int value`: The value to assign to all elements.
- Outputs:
  No return value. The function modifies the array directly.

3)Logic and Steps

1. Check if the size is less than or equal to `0`. If true,
   exit the function because there's no valid array to modify.
2. Use a loop to iterate from `0` to `size - 1`.
3. For each index `i`, assign the value `value` to `arr[i]`.

4)Edge Cases

- Empty Array:
  If `size` is `0` or negative, the function does nothing.
- Large Arrays:
  For very large arrays the function will work correctly.
- Value as Zero:
  Works correctly if `value` is `0`, initializing the array
  to all zeros.

2. OPERATION ON ARRAYS

02 printArray(int arr[], int size)

1)Purpose

The function prints all the elements of an array, each on a new line

2)Inputs / Outputs

- Inputs:
  arr[]: The array to be printed.
  size: The number of elements in the array.
- Outputs:
  No return value. The function outputs the array elements to the console.

3)Logic and Steps

1. Check if the size is less than or equal to `0`. If true,
   exit the function because there's no valid array to modify.
2. Use a loop to iterate from `0` to `size - 1`.
3. For each index `i`, assign the value `value` to `arr[i]`.

4)Edge Cases

- Empty Array:
  If `size` is `0` or negative, the function does nothing.
- Large Arrays:
  For very large arrays the function will work correctly.
- Value as Zero:
  Works correctly if `value` is `0`, initializing the array to all zeros.

2. OPERATION ON ARRAYS

03 findMax(int arr[], int size)

1)Purpose

The function finds and returns the maximum value in an array of integers. It compares each element and keeps track of the largest one.

2)Inputs / Outputs

- Inputs:
  arr[]: The array of integers.
  size: The number of elements in the array.
- Outputs:
  The maximum integer in the array, or `INT_MIN` if the array is empty or invalid.

3)Logic and Steps

1. Check if the size is less than or equal to 0. If true, print "Invalid size" and return `INT_MIN` as an error value.
2. Set the first element of the array as the current maximum.
3. Loop through the rest of the array, comparing each element with the current maximum. If a larger element is found, update the maximum.
4. Return the maximum value found.

4)Edge Cases

- If the array size is 0 or negative, the function prints an error message and returns `INT_MIN`.

2. OPERATION ON ARRAYS

04 findMin(int arr[], int size)

1)Purpose

The function finds and returns the minimum value in an array of integers. It compares each element and keeps track of the smallest one.

2)Inputs / Outputs

- Inputs:
  arr[]: The array of integers.
  size: The number of elements in the array.
- Outputs:
  The minimum integer in the array, or `INT_MAX` if the array is empty or invalid.

3)Logic and Steps

1. Check if the size is less than or equal to 0. If true, print "Invalid size" and return `INT_MAX` as an error value.
2. Set the first element of the array as the current minimum.
3. Loop through the rest of the array, comparing each element with the current minimum. If a smaller element is found, update the minimum.
4. Return the minimum value found.

4)Edge Cases

- If the array size is 0 or negative, the function prints an error message and returns `INT_MAX`.

2. OPERATION ON ARRAYS

05 sumArray(int arr[], int size)

1)Purpose

The function calculates and returns the sum of all the elements in an array of integers.

2)Inputs / Outputs

- Inputs:
  arr[]: The array of integers.
  size: The number of elements in the array.
- Outputs:
  The sum of the elements in the array, or `0` if the array is empty or invalid.

3)Logic and Steps

1. Check if the size is less than or equal to 0. If true, print "Invalid size" and return `0` as an error value.
2. Initialize a variable `sum` to 0.
3. Loop through the array, adding each element to the `sum`.
4. Return the total `sum`.

4)Edge Cases

- If the array size is 0 or negative, the function prints an error message and returns `0`.

2. OPERATION ON ARRAYS

06 averageArray(int arr[], int size)

1)Purpose

The function calculates and returns the average (mean) of all elements in an array of integers. The average is the sum of the elements divided by the number of elements.

2)Inputs / Outputs

- Inputs:
  arr[]: The array of integers.
  size: The number of elements in the array.
- Outputs:
  The average of the elements in the array, or `0` if the array is empty or invalid.

3)Logic and Steps

1. Check if the size is less than or equal to 0. If true, print "Invalid size" and return `0` as an error value.
2. Call the `sumArray` function to get the sum of the elements.
3. Divide the sum by the size of the array and return the result as a double (to handle decimal values).

4)Edge Cases

- If the array size is 0 or negative, the function prints an error message and returns `0`.

2. OPERATION ON ARRAYS

07 isSorted(int arr[], int size)

1)Purpose

The function checks if the elements of an array are sorted in ascending order. An array is considered sorted if every element is less than or equal to the next element.

2)Inputs / Outputs

- Inputs:
  arr[]: The array of integers.
  size: The number of elements in the array.
- Outputs:
  Returns `true` if the array is sorted, otherwise returns `false`.
  If the size is invalid, it prints "Invalid size" and returns `false`.

3)Logic and Steps

1. Check if the size is less than or equal to 0. If true,
   print "Invalid size" and return `false`.
2. Iterate through the array and compare each element with the next.
3. If any element is greater than the next element, return `false`.
4. If no violations are found, return `true` indicating the array is sorted.

4)Edge Cases

- If the array size is 0 or negative, the function prints
  an error message and returns

2. OPERATION ON ARRAYS

08 reverseArray(int arr[], int size)

1)Purpose

The function reverses the elements of an array in place, meaning it modifies the original array to have its elements in the opposite order.

2)Inputs / Outputs

- Inputs:
  arr[]: The array of integers.
  size: The number of elements in the array.
- Outputs:
  The array is reversed. If the size is invalid, the function prints "Invalid size" and does nothing.

3)Logic and Steps

1. Check if the size is less than or equal to 0. If true, print "Invalid size" and return without modifying the array.
2. Use a loop to swap the elements at the start and end of the array, gradually going the center.
3. Each pair of elements is swapped until the middle of the array is reached.

4)Edge Cases

- If the array size is 0 or negative, the function prints an error message and does nothing.

2. OPERATION ON ARRAYS

07 reverseArray(int arr[], int size)

1)Purpose

The function reverses the elements of an array in place, meaning it modifies the original array to have its elements in the opposite order.

2)Inputs / Outputs

- Inputs:
  arr[]: The array of integers.
  size: The number of elements in the array.
- Outputs:
  The array is reversed. If the size is invalid, the function prints "Invalid size" and does nothing.

3)Logic and Steps

1. Check if the size is less than or equal to 0. If true, print "Invalid size" and return without modifying the array.
2. Use a loop to swap the elements at the start and end of the array, gradually going the center.
3. Each pair of elements is swapped until the middle of the array is reached.

4)Edge Cases

- If the array size is 0 or negative, the function prints an error message and does nothing.

2. OPERATION ON ARRAYS

09 countEvenOdd(int arr[], int size, int* evenCount, int* oddCount)

1)Purpose

The function counts the number of even and odd numbers in an array and stores the results in the provided variables.

2)Inputs / Outputs

- Inputs:
  arr[]: The array of integers.
  size: The number of elements in the array.
  evenCount: A pointer to an integer where the count of even numbers will be stored.
  oddCount: A pointer to an integer where the count of odd numbers will be stored.
- Outputs: The function updates the `evenCount` and `oddCount` values with the respective counts of even and odd numbers in the array.

3)Logic and Steps

1. Check if the size is less than or equal to 0. If true, print "Invalid size" and return without modifying the counts.
2. Initialize both `evenCount` and `oddCount` to 0.
3. Loop through the array and check each element:
   - If an element is even (`arr[i] % 2 == 0`), increment `evenCount`.
   - Otherwise, increment `oddCount`.
4. At the end, the values pointed to by `evenCount` and `oddCount` will hold the respective counts.

4)Edge Cases

- If the array size is 0 or negative, the function prints an error message and does nothing.

2. OPERATION ON ARRAYS

10 secondLargest(int arr[], int size)

1)Purpose

The function finds and returns the second largest element in the array.

2)Inputs / Outputs

- Inputs:
  arr[]: The array of integers.
  size: The number of elements in the array.
- Outputs:
  Returns the second largest element in the array. If the array has fewer than 2 elements, it returns `INT_MIN` to indicate an error.

3)Logic and Steps

1. Check if the size is less than or equal to 1. If true, print "Invalid size" and return `INT_MIN` to indicate an error (since the second largest number can't exist).
2. Initialize `max` as the first element of the array and `submax` as `INT_MIN` (to handle all possible cases).
3. Loop through the array starting from the second element:
   - If the current element is larger than `max`, update `submax` to hold the previous `max`, then update `max` with the current element.
   - If the current element is smaller than `max` but larger than `submax`, update `submax`.
4. After the loop, return `submax`, which holds the second largest value. If no second largest exists (e.g., the array has identical values), `submax` will still be `INT_MIN`.

4)Edge Cases

- If the array has fewer than 2 elements, the function will return `INT_MIN`, indicating an invalid array size for finding the second largest element.

2. OPERATION ON ARRAYS

11 elementFrequency(int arr[], int size)

1)Purpose

The function calculates and prints the frequency of each element in the array.

2)Inputs / Outputs

- Inputs:
  arr[]: The array of integers.
  size: The number of elements in the array.
- Outputs:
  Prints the frequency of each element in the array to the console.

3)Logic and Steps

1. Check if the size of the array is less than or equal to 0. If true, print "Invalid size" and return.
2. Initialize a `checked` array of the same size as the input array, where each element is initially set to `false`. This array will track which elements have already been counted to avoid double-counting.
3. Loop through the array:
   - For each element at index `i`, check if it has already been counted (i.e., `checked[i]` is `true`). If it has, skip to the next iteration.
   - If it hasn't been counted, initialize a `frequency` variable to 0. Then, loop through the array again to count how many times the element at index `i` appears in the array.
   - After counting, mark the indices that contain the element as checked by setting `checked[j] = true` for each occurrence.
   - Print the element and its frequency.

4)Edge Cases

- If the array size is 0 or less, the function will print an error message and exit without processing.

2. OPERATION ON ARRAYS

12 removeDuplicates(int arr[], int size)

1)Purpose

The function removes duplicates from an array and returns the new size of the array containing only unique elements.

2)Inputs / Outputs

- Inputs:
  arr[]: The array of integers.
  size: The number of elements in the array.
- Outputs:
  Returns the new size of the array after removing duplicates. Modifies the input array in place.

3)Logic and Steps

1. Check if the size of the array is less than or equal to 0. If true, print "Invalid size" and return 0.
2. Initialize a `checked` array of the same size as the input array. Each element is initially set to `false`, indicating that no element has been checked.
3. Loop through the array and check for duplicates. For each element `arr[i]`, loop through the remaining elements `arr[j]` starting from `i`. If `arr[i]` is equal to `arr[j]` and `i != j`, mark `checked[j]` as `true`. This step identifies the duplicate elements.
4. Count the unique elements by looping through the array and counting elements that are not marked as `checked`. Store the count in the variable `new_size`.
5. Create the new array with only unique elements by writing the elements that are not marked as duplicates (`checked[i]` is `false`) to the beginning of the array. The variable `i_write` keeps track of where the unique element should be placed.
6. Return the new size of the array after removing duplicates.

4)Edge Cases

- If the array size is 0 or less, the function will print "Invalid size" and exit without processing.
- If all elements in the array are the same, the resulting array will contain only one unique element.

2. OPERATION ON ARRAYS

13 binarySearch(int arr[], int size, int target)

1)Purpose

The function performs a binary search on a sorted array to find the index of a target value.

2)Inputs / Outputs

- Inputs:
  arr[]: The sorted array of integers.
  size: The number of elements in the array.
  target: The value to search for in the array.
- Outputs:
  Returns the index of the target if found, otherwise returns -1.

3)Logic and Steps

1. Check if the array size is less than or equal to 0. If true, print "Invalid size" and return -1.
2. Initialize two pointers: `start` (at index 0) and `end` (at the last index of the array).
3. Perform the binary search in a while loop until the start pointer is greater than the end pointer.
4. Calculate the middle index as `middle = start + (end - start) / 2`.
5. Compare the middle element with the target value:
   - If `arr[middle] == target`, return the middle index.
   - If `arr[middle] > target`, move the `end` pointer to `middle - 1` to search the left half of the array.
   - If `arr[middle] < target`, move the `start` pointer to `middle + 1` to search the right half of the array.
6. If the target is not found after the loop finishes, return -1 indicating that the target is not present in the array.

4)Edge Cases

- If the size of the array is less than or equal to 0, the function will print "Invalid size" and return -1.
- If the target is not found in the array, the function will return -1.

2. OPERATION ON ARRAYS

14 linearSearch(int arr[], int size, int target)

1)Purpose

The function performs a linear search on an array to find the index of a target value.

2)Inputs / Outputs

- Inputs:
  arr[]: The array of integers.
  size: The number of elements in the array.
  target: The value to search for in the array.
- Outputs:
  Returns the index of the target if found, otherwise returns -1.

3)Logic and Steps

1. Check if the array size is less than or equal to 0. If true, print "Invalid size" and return -1.
2. Iterate through each element in the array from index 0 to `size-1`:
   - If an element is equal to the target value (`arr[i] == target`), return the index `i`.
3. If the target is not found by the end of the loop, return -1 indicating that the target is not present in the array.

4)Edge Cases

- If the size of the array is less than or equal to 0, the function will print "Invalid size" and return -1.
- If the target is not found in the array, the function will return -1.

2. OPERATION ON ARRAYS

15 leftShift(int arr[], int size, int rotations)

1)Purpose

The function performs a left shift on an array by rotating its elements a specified number of times.

2)Inputs / Outputs

- Inputs:
  arr[]: The array of integers.
  size: The number of elements in the array.
  rotations: The number of positions to shift the array to the left.
- Outputs:
  Modifies the input array by shifting its elements to the left by the specified number of rotations.

3)Logic and Steps

3)Logic and Steps

1. Check if the array size is less than or equal to 0. If true, print "Invalid size" and return.
2. Calculate the effective number of rotations needed by using `rotations %size`. This ensures that the number of rotations is within the array bounds (since shifting by the array's length results in the same array).
3. For each rotation:
   - Save the first element of the array (`arr[0]`).
   - Shift each element of the array to the left by one position, starting from the second element.
   - Place the saved first element at the end of the array (`arr[size-1]`).

4)Edge Cases

- If the array size is less than or equal to 0, the function will print "Invalid size" and not perform any operations.
- If the number of rotations is a multiple of the array size (e.g., 5 rotations for a size 5 array), the array will remain unchanged.

2. OPERATION ON ARRAYS

16 rightShift(int arr[], int size, int rotations)

1)Purpose

The function performs a right shift on an array by rotating its elements a specified number of times.

2)Inputs / Outputs

- Inputs:
  - arr[]: The array of integers.
  - size: The number of elements in the array.
  - rotations: The number of positions to shift the array to the right.
- Outputs:
  - Modifies the input array by shifting its elements to the right by the specified number of rotations.

3)Logic and Steps

1. Check if the array size is less than or equal to 0. If true, print "Invalid size" and return.
2. Calculate the effective number of rotations needed by using `rotations % size`. This ensures that the number of rotations is within the array bounds (since shifting by the array's length results in the same array).
3. For each rotation:
   - Save the last element of the array (`arr[size-1]`).
   - Shift each element of the array to the right by one position, starting from the second-to-last element.
   - Place the saved last element at the beginning of the array (`arr[0]`).

4)Edge Cases

- If the array size is less than or equal to 0, the function will print "Invalid size" and not perform any operations.
- If the number of rotations is a multiple of the array size (e.g., 5 rotations for a size 5 array), the array will remain unchanged.

2. OPERATION ON ARRAYS

17 bubbleSort(int arr[], int size)

1)Purpose

The function sorts an array of integers in ascending order using the Bubble Sort algorithm, which repeatedly steps through the array, compares adjacent elements, and swaps them if they are in the wrong order.

2)Inputs / Outputs

- Inputs:
  - arr[]: The array of integers to be sorted.
  - size: The number of elements in the array.
- Outputs:
  - The input array is modified in-place to become sorted in ascending order.

3)Logic and Steps

1. If the array size is less than or equal to 1, return immediately since no sorting is needed.
2. For each pass through the array:
   - Iterate from the first element to the last unsorted element.
   - Compare each pair of adjacent elements.
   - If the current element is greater than the next, swap them using the `switchElements` function.
   - Set a flag (`changed`) to `true` if a swap occurs.
3. If no swaps occur during a pass, the array is already sorted, and the loop breaks early to optimize performance.

4)Edge Cases

- If the size of the array is less than or equal to 1, the function does nothing.
- If the array is already sorted, the function completes in O(n) time due to the early termination when no swaps are made.

2. OPERATION ON ARRAYS

18 selectionSort(int arr[], int size)

1)Purpose

The function sorts an array of integers in ascending order using the Selection Sort algorithm. The algorithm divides the array into a sorted and an unsorted portion, repeatedly finding the minimum element from the unsorted portion and swapping it into the sorted portion.

2)Inputs / Outputs

- Inputs:
  - arr[]: The array of integers to be sorted.
  - size: The number of elements in the array.
- Outputs:
  - The input array is modified in-place to become sorted in ascending order.

3)Logic and Steps

1. If the array size is less than or equal to 1, return immediately since no sorting is needed.
2. For each iteration `i` (starting from the first unsorted element to the second-to-last):
   - Find the smallest element in the remaining unsorted portion of the array.
   - Swap the current element `arr[i]` with the smallest element found using the `switchElements` function.

4)Edge Cases

- if the size of the array is less than or equal to 1, the function does nothing.
- If the array is already sorted, the function still goes through all iterations but does not make unnecessary swaps.

2. OPERATION ON ARRAYS

19 insertionSort(int arr[], int size)

1)Purpose

The function sorts an array of integers in ascending order using the Insertion Sort algorithm. It builds the final sorted array one item at a time by inserting each element into its correct position within the sorted portion of the array.

2)Inputs / Outputs

- Inputs:
  - arr[]: The array of integers to be sorted.
  - size: The number of elements in the array.
- Outputs:
  - The input array is modified in-place to become sorted in ascending order.

3)Logic and Steps

1. If the array size is less than or equal to 1, return immediately as no sorting is needed.
2. Starting from the second element (`i = 1`), compare each element with the element just before it (`arr[i]` with `arr[i-1]`).
   - If the current element is smaller than the previous element, swap them using the `switchElements` function.
   - Continue swapping until the current element is in its correct position relative to the sorted portion of the array.

4)Edge Cases

- If the size of the array is less than or equal to 1, the function does nothing.
- If the array is already sorted, the function still iterates through the entire array but makes no swaps.

2. OPERATION ON ARRAYS

20 mergeSort(int arr[], int left, int right)

1)Purpose

The function sorts an array of integers in ascending order using the Merge Sort algorithm. Merge Sort is a divide-and-conquer algorithm that splits the array into two halves, sorts each half recursively, and then merges the two sorted halves together.

2)Inputs / Outputs

- Inputs:
  - arr[]: The array of integers to be sorted.
  - left: The starting index of the portion of the array to be sorted.
  - right: The ending index of the portion of the array to be sorted.
- Outputs:
  - The input array is modified in-place to become sorted in ascending order.

3)Logic and Steps

1. The `mergeSort` function first checks if the portion of the array (from `left` to `right`) has more than one element. If not, it returns (the array is already sorted).
2. It calculates the middle index `mid` to divide the array into two halves.
3. The function then recursively calls `mergeSort` on the left and right halves of the array.
4. After sorting both halves, the `merge` function is called to merge the two sorted halves into a single sorted array.
5. The merging process compares elements from the two halves and places the smaller element in the correct position in the original array.

4)Edge Cases

- If the array contains only one element or is empty (`left >= right`), the function returns without any action.

2. OPERATION ON ARRAYS

21 quickSort(int arr[],, int left, int right)

1)Purpose

The function sorts an array of integers in ascending order using the Quick Sort algorithm. Quick Sort is an efficient divide-and-conquer algorithm that selects a "pivot" element and partitions the array into two subarrays, sorting each subarray recursively.

2)Inputs / Outputs

- Inputs:
  - arr[]: The array of integers to be sorted.
  - low: The starting index of the portion of the array to be sorted.
  - high: The ending index of the portion of the array to be sorted.
- Outputs:
  - The input array is modified in-place to become sorted in ascending order.

3)Logic and Steps

1. The `quickSort` function first checks if the portion of the array (from `low` to `high`) has more than one element. If not, it returns (the array is already sorted).
2. The `partition` function is called to choose a pivot element from the array and partition the array into two subarrays.
3. The array is rearranged such that all elements smaller than the pivot are on the left side, and all elements greater than the pivot are on the right side.
4. The `quickSort` function is then called recursively on the two subarrays to the left and right of the pivot.
5. This process continues until the base case is reached, where the subarrays are of size 1 or empty.

4)Edge Cases

- If the array contains only one element or is empty (`low >= high`), the function returns without any action.
- If the pivot is the smallest or largest element, the partition step will result in unbalanced partitions, but the algorithm will still work correctly.

2. OPERATION ON ARRAYS

22 findMissingNumber(int arr[], int size)

1)Purpose

The function calculates the missing number in a consecutive integer sequence starting from 1 to `n` (where `n` is the size of the given array) by using the sum of integers approach.

2)Inputs / Outputs

- Inputs:
  - arr[]: The array containing integers in a sequence from 1 to `n`, with one number missing.
  - size: The number of elements present in the array (which is `n-1` because one number is missing).
- Outputs:
  - The missing number in the sequence.

3)Logic and Steps

1. The total sum of numbers from 1 to `n+1` is calculated using the formula `(n+1)*(n+2)/2`.
2. The sum of the numbers in the given array is calculated using the `sumArray` function.
3. The difference between the total sum and the sum of the array elements gives the missing number.
4. This result is returned.

4)Edge Cases

-if the array size is 0 or negative, it will print an "Invalid size" message and return 0 (as handled by the `sumArray` function).
- If the array is already complete (no missing number), this function will return an incorrect result (not applicable in the context of this function).

2. OPERATION ON ARRAYS

23 findPairsWithSum(int arr[], int size, int sum)

1)Purpose

This function finds and prints all unique pairs of elements in the array that sum up to the specified value.

2)Inputs / Outputs

- Inputs:
  - arr[]: An array of integers.
  - size: The number of elements in the array.
  - sum: The target sum that we want to find pairs for.
- Outputs:
  - The function prints the pairs of elements from the array that sum to the specified value.

3)Logic and Steps

1. It first initializes an array `seen` to store previously checked elements to avoid checking pairs multiple times.
2. For each element `arr[i]`, the function calculates the `target` value as `sum-arr[i]`.
3. It checks if the `target` value is already in the array (`arr[]`) and if `arr[i]` has already been checked (using the `seen[]` array).
4. If the target exists in the array and the pair hasn't been considered yet, it prints the pair `(arr[i], target)`.
5. After processing, it adds both `arr[i]` and `target` to the `seen[]` array to track the elements that have already been processed.

4)Edge Cases

- If the array size is 0 or negative, the function will do nothing as there's no array to process (handled by the check `if(size <= 0)`).
- The function doesn't consider the same pair multiple times. For example, `(2, 3)` and `(3, 2)` will not be counted separately.
- If no pairs exist that sum to the target value, the function won't output anything.

2. OPERATION ON ARRAYS

24 findSubArrayWithSum(int arr[], int size, int sum)

1)Purpose

This function finds and prints the indices of the first contiguous subarray in the given array whose elements sum up to the specified value.

2)Inputs / Outputs

- Inputs:
  - arr[]: An array of integers.
  - size: The number of elements in the array.
  - sum: The target sum for the subarray.
- Outputs:
  - The function prints the starting and ending indices of the subarray that sums to the specified value.
  - If no such subarray exists, the function does not output anything.

3)Logic and Steps

1. It iterates over all possible starting points in the array (`i`).
2. For each starting point, it initializes `current` (a variable to store the cumulative sum) and another index `j` to track the elements being added.
3. It adds elements one by one from the current starting index until `current` equals or exceeds the target sum (`sum`) or the end of the array is reached.
4. If `current == sum`, the function prints the indices of the subarray (`i` to `j-1`) and exits.
5. If the cumulative sum exceeds the target value during iteration, the loop breaks and moves to the next starting point.

4)Edge Cases

- If the array size is 0 or negative, the function exits without doing anything (handled by `if(size <=0)).
- If no subarray with the desired sum is found, the function does not print any output.
- If all elements in the array are negative and the sum is positive, the function will not find a valid subarray.

2. OPERATION ON ARRAYS

25 rearrangeAlternatePositiveNegative(int arr[], int size)

1)Purpose

This function rearranges the elements of the input array so that positive and negative integers alternate, maintaining their relative order as much as possible.

2)Inputs / Outputs

- Inputs:
  - arr[]: An array of integers.
  - size: The number of elements in the array.
- Outputs:
  - The input array is modified in-place, so the elements alternate between positive and negative numbers wherever possible.

3)Logic and Steps

1. It starts with three pointers:
   - `i_write`: Tracks the position in the array where the next alternating positive or negative element should be placed.
   - `iNegative`: Tracks the current index to find the next negative element.
   - `iPositive`: Tracks the current index to find the next positive element.
2. The function uses a `current` variable to alternate between selecting a negative (`current == 1`) or positive (`current == -1`) element for placement.
3. It iterates over the array to find the next positive or negative number:
   - For negatives (`current == 1`), it advances `iNegative` until it finds a negative number. If found, it swaps it with the element at `i_write`.
   - For positives (`current == -1`), it advances `iPositive` until it finds a positive number. If found, it swaps it with the element at `i_write`.
4. The process continues until either all positive or all negative numbers have been placed, or `i_write` reaches the end of the array.

4)Edge Cases

- If the size of the array is 0 or negative, the function immediately exits without doing anything.
- If the array contains only positive or only negative numbers, the function leaves the array unchanged.

2. OPERATION ON ARRAYS

26 findMajorityElement(int arr[], int size)

1)Purpose

This function identifies the majority element in an array. A majority element is defined as an element that appears more than `size / 2` times in the array.

2)Inputs / Outputs

- Inputs:
  - `arr[]`: An array of integers.
  - `size`: The number of elements in the array.
- Outputs:
  - Returns the majority element if it exists; otherwise, returns `INT_MIN`.

3)Logic and Steps

1. Candidate Selection(-MooreAlgorithm):
   - initialize `ele` (candidate) to `-1` and `count` to `0`.
   - Iterate through the array:
     - If `count` is `0`, set the current element as the candidate (`ele`) and reset `count` to `1`.
     - If the current element matches the candidate, increment `count`.
     - If the current element does not match the candidate, decrement `count`.
   - At the end of this loop, `ele` is the candidate that might be the majority element.

2. Validation:
   - Reset `count` to `0`.
   - Iterate through the array again to count the occurrences of the candidate (`ele`).
   - If the candidate appears more than `size / 2` times, return it. Otherwise, return `INT_MIN`.

4)Edge Cases

- If the array size is `0` or negative, the function returns `INT_MIN`.
- If no majority element exists, the function returns `INT_MIN`.

2. OPERATION ON ARRAYS

27 longestIncreasingSubsequence(int arr[], int size)

1)Purpose

This function calculates the length of the "longest increasing subsequence' in an array. Note: The elements of the subsequence are not required to be consecutive.

2)Inputs / Outputs

- Inputs:
  - `arr[]`: An array of integers.
  - `size`: The number of elements in the array.
- Outputs:
  - Returns the length of the longest increasing subsequence in the array. If the array is empty or invalid, it returns `0`.

3)Logic and Steps

1. initialization:
   - If the array size is less than or equal to 0, return `0` as there is no valid subsequence.
   - Initialize `max_length` to `1` to track the longest increasing subsequence.
   - Initialize `length` to `1` to track the current increasing subsequence length.

2. Iterate Through the Array
   - Starting from the second element (`i = 1`), compare each element with the previous element.
   - If the current element is greater than the previous one (`arr[i] > arr[i-1]`):
     - Increment the `length` of the current subsequence.
     - Update `max_length` if the current `length` is greater than `max_length`.
   - Otherwise, reset `length` to `1` because the sequence is no longer increasing.

3.return the Result:
   - After traversing the array, `max_length` holds the length of the longest increasing subsequence.

4)Edge Cases

- If the array size is `0` or negative, the function returns `0`.
- If the array has only one element, the function returns `1`.
- If the array is strictly decreasing, the function returns `1` because no increasing subsequence exists beyond individual elements.

2. OPERATION ON ARRAYS

28 findDuplicates(int arr[], int size)

1)Purpose

This function identifies and prints all duplicate elements in a given array.

2)Inputs / Outputs

- Inputs:
  - `arr[]`: An array of integers.
  - `size`: The number of elements in the array.
- Outputs:
  - Prints each duplicated element in the array.

3)Logic and Steps

1. *validation:
   - If the array size is less than or equal to 0, the function returns without performing any operations.

2.Sorting:
   - The function sorts the array using the `quickSort` function. Sorting ensures that duplicates are grouped together, making them easier to identify.

3. Iterate Through the Array:
   - Starting from the second element (`i = 1`), compare each element with the previous one (`arr[i] == arr[i-1]`).
   - If a duplicate is found, print the element and skip over all additional occurrences of the same value using a nested loop.

4. end the Function:
   - Continue until the end of the array, processing all elements.

4)Edge Cases

- If the array size is `0` or negative, the function does nothing.
- If the array has no duplicates, nothing is printed.
- If the array is already sorted, the function works without any issue.

2. OPERATION ON ARRAYS

29 findIntersection(int arr[], int size)

1)Purpose

This function finds the intersection of two arrays (i.e, common elements between the two arrays) and prints the shared elements.

2)Inputs / Outputs

- Inputs:
  - `arr1[]`: The first array of integers.
  - `size1`: The number of elements in the first array.
  - `arr2[]`: The second array of integers.
  - `size2`: The number of elements in the second array.
- Outputs:
  - Prints the elements that are present in both arrays.

3)Logic and Steps

1. iterate Through the First Array:
   - For each element in `arr1`, check if it exists in `arr2`.

2. Check for Inclusion:
   - The `includes()` function is used to determine whether the current element of `arr1` exists in `arr2`.

3. Print the Element:
   - If the element is found in both arrays, it is printed.

4)Edge Cases

- If one or both arrays are empty, no elements are printed.
- If there are no common elements between the two arrays, no output is printed.
-if an element in `arr1` appears multiple times, it will be printed each time it is found in `arr2`.(not mentioned in the problem that they should be unique)

2. OPERATION ON ARRAYS

30 findUnion(int arr1[], int size1, int arr2[], int size2)

1)Purpose

This function finds the union of two arrays (i.e., all unique elements from both arrays) and prints the result.

2)Inputs / Outputs

- Inputs:
  - `arr1[]`: The first array of integers.
  - `size1`: The number of elements in the first array.
  - `arr2[]`: The second array of integers.
  - `size2`: The number of elements in the second array.
- Outputs:
  - Prints all unique elements present in either `arr1` or `arr2`.

3)Logic and Steps

1. Iterate Through the First Array:
   - For each element in `arr1`, check if it is already included in the `nums` array (used to store unique elements).
   - If it is not already included, add it to `nums`.

2. Iterate Through the Second Array:
   - For each element in `arr2`, check if it is already included in the `nums` array.
   - If it is not already included, add it to `nums`.

3. Print the Union:
   - Iterate through the `nums` array and print all the elements

4)Edge Cases

- If both arrays are empty, no output is printed.
- If all elements in `arr2` are already present in `arr1`, the function will only add unique elements from `arr2`.

3. OPERATION ON STRINGS

01 stringLength(char* str)

1)Purpose

This function calculates and returns the length of a string by counting the number of characters until the null terminator (`'\0'`), which signifies the end of the string.

2)Inputs / Outputs

- Input:
  - `char* str`: A pointer to the first character of a null-terminated string.
- Output:
  - Returns the length of the string as an integer.

3)Logic and Steps

1. Iterate Through the First Array:
   - For each element in `arr1`, check if it is already included in the `nums` array (used to store unique elements).
   - If it is not already included, add it to `nums`.

2. Iterate Through the Second Array:
   - For each element in `arr2`, check if it is already included in the `nums` array.
   - If it is not already included, add it to `nums`.

3. Print the Union:
   - Iterate through the `nums` array and print all the elements

4)Edge Cases

- If the string is empty (`""`), the function will return `0` because the first character itself is `'\0'`.

3. OPERATION ON STRINGS

02 stringCopy(char* dest, const char* src)

1)Purpose

This function copies a null-terminated string from the source (`src`) to the destination (`dest`).

2)Inputs / Outputs

- Input:
  - `char* dest`: A pointer to the destination buffer where the string will be copied.
  - `const char* src`: A pointer to the source string that will be copied.
- Output:
  - The source string is copied into the destination buffer, including the null terminator (`'\0'`)

3)Logic and Steps

1. Initialize an index variable `i` to 0.
2. Use a `while` loop to traverse each character of the source string (`src`).
   - Copy the current character (`src[i]`) into the destination buffer (`dest[i]`).
   - Increment the index `i`.
3. After the loop (once `'\0'` is reached), add the null terminator to the destination (`dest[i] = '\0'`).
4. The string in `src` is now fully copied into `dest`.

4)Edge Cases

- If `src` is an empty string (`""`), the destination (`dest`) will also be set to an empty string (`dest[0] = '\0'`).
-the function assumes that the destination(`dest`) has enough space to accommodate the source string, including the null terminator.if not,it may lead to errors.

3. OPERATION ON STRINGS

03 stringConcat(char* dest, const char* src)

1)Purpose

This function concatenates the `src` string to the end of the `dest` string.

2)Inputs / Outputs

- Inputs:
  - `dest`: A pointer to the destination string, which will be modified to include `src`.
  - `src`: A constant pointer to the source string that will be appended to `dest`.
- Outputs:
  - The `dest` string is updated in place with the concatenated result.
  - No return value.

3)Logic and Steps

1. Find the End of the Destination String:
   - Start with `i = 0`.
   - Traverse the `dest` string to locate the null terminator (`\0`), which marks the end of the string.

2. Append the Source String:
   - Start with `j = 0`.
   - Traverse the `src` string until the null terminator (`\0`).
   - Copy each character from `src` to the end of `dest` (starting from `dest[i]`).
   - Increment both `i` and `j` to keep track of the current positions in `dest` and `src`.

3. Null-Terminate the Updated Destination String:
   - After all characters from `src` are appended, set the next character in `dest` to `\0` to terminate the string properly.

4)Edge Cases

- If `src` is an empty string (`""`), `dest` remains unchanged.
- if `dest` does not have sufficient memory allocated to accommodate the concatenated result, this function may cause undefined behavior

3. OPERATION ON STRINGS

04 stringCompare(const char* str1, const char* str2)

1)Purpose

This function compares two strings `str1` and `str2` lexicographically.

2)Inputs / Outputs

- Inputs:
  - `str1`: A pointer to the first string.
  - `str2`: A pointer to the second string.
- Outputs:
  - Returns `0` if the strings are equal.
  - Returns `-1` if `str1` is lexicographically smaller than `str2`.
  - Returns `1` if `str1` is lexicographically greater than `str2`.

3)Logic and Steps

1. Compare Corresponding Characters:
   - Start at the beginning of both strings (`i = 0`).
   - Traverse the strings character by character until either:
     - A mismatch is found, or
     - One of the strings reaches its null terminator (`\0`).

   - If characters at the same position differ:
     - Return `-1` if `str1[i] < str2[i]`.
     - Return `1` if `str1[i] > str2[i]`.

2. Handle Equal Length Strings:
   - If the loop ends and both strings reach their null terminators, return `0` (strings are equal).

3. Handle Unequal Length Strings:
   - If one string ends before the other:
     - Return `-1` if `str1` ends first.
     - Return `1` if `str2` ends first.

4)Edge Cases

- Both strings are empty (`""`): Return `0`.
- `str1` is a prefix of `str2`: Return `-1` (e.g., `str1 = "abc"` and `str2 = "abcd"`).
- `str2` is a prefix of `str1`: Return `1` (e.g., `str1 = "abcd"` and `str2 = "abc"`).

3. OPERATION ON STRINGS

05 bool isEmpty(char* str)

1)Purpose

This function checks whether a given string is empty.

2)Inputs / Outputs

- Input:
  - `str`: A pointer to the string to be checked.
- Output:
  - Returns `false` if the string is `NULL` (not pointing to a valid memory location).
  - Returns `true` if the string is empty (`""` or its first character is the null terminator `\0`).
  - Returns `false` otherwise

.

3)Logic and Steps

1. Check for `NULL`:
   - If the input string pointer (`str`) is `NULL`, the function returns `false`. This prevents accessing invalid memory.

2.check for Empty String:
   - If `str[0] == '\0'`, the string is considered empty because it has no characters before the null terminator.
   - The function then returns `true`.

3.wDefault Case:
   - If the string pointer is valid and the first character is not `'\0'`, it is not empty, and the function returns `false`

4)Edge Cases

- If `str` is `NULL`, the function returns `false`.
- If `str` points to an empty string (`""`), the function returns `true`.

3. OPERATION ON STRINGS

06 reverseString(char* str)

1)Purpose

This function reverses a given string in place.

2)Inputs / Outputs

- Input:
  - `str`: A pointer to a null-terminated string.
- Output:
  - The input string is modified to be its reversed form,no value is returned.

.

3)Logic and Steps

1. Null Check:
   - If the input `str` is `NULL`, the function returns immediately to prevent undefined behavior.
2. Find String Length:
   - The `stringLength` function is called to determine the length of the input string.
   - This ensures the loop knows how far to iterate.
3. Reverse Characters:
   - Iterate from the start of the string to the middle (`length / 2`).
   - Swap the character at the current position with its corresponding character from the end (`length-1-i`).
   - Use a temporary variable (`temp`) for the swap to avoid overwriting characters.

4. End of Function:
   - After the loop, the string is reversed in place.

4)Edge Cases

- If `str` is `NULl`,the function does nothing.
- If the string is empty (`""`), the function does nothing.
- If the string contains only one character, it remains unchanged.

3. OPERATION ON STRINGS

07 toUpperCase(char* str)

1)Purpose

This function converts all lowercase letters in a string to uppercase.

2)Inputs / Outputs

Input:
- str: A pointer to a null-terminated string.

Output:
- The input string is modified in place, converting all lowercase letters to uppercase.

.

3)Logic and Steps

1. Null Check:
   - If the input string is NULL, the function returns immediately.

2. Conversion to Uppercase:
   - Iterate through each character of the string.
   - Check if the character is a lowercase letter (ASCII 97-122).
   - If true, subtract 32 from its ASCII value to convert it to uppercase.

3. End of String:
   - Stop processing when the null terminator ('\0') is reached.

4)Edge Cases

- If the input string is NULL, the function does nothing.
- If the string is empty (""), the function does nothing.
- Non-alphabetical characters remain unchanged.

3. OPERATION ON STRINGS

08 toLowerCase(char* str)

1)Purpose

This function converts all uppercase letters in a string to lowercase.

2)Inputs / Outputs

Input:
- str: A pointer to a null-terminated string.

Output:
- The input string is modified in place, converting all uppercase letters to lowercase.

.

3)Logic and Steps

1. Null Check:
   - If the input string is NULL, the function returns immediately.

2. Conversion to Lowercase:
   - Iterate through each character of the string.
   - Check if the character is an uppercase letter (ASCII 65-90).
   - If true, add 32 to its ASCII value to convert it to lowercase.

3. End of String:
   - Stop processing when the null terminator ('\0') is reached.

4)Edge Cases

- If the input string is NULL, the function does nothing.
- If the string is empty (""), the function does nothing.
- Non-alphabetical characters remain unchanged.

3. OPERATION ON STRINGS

09 isPalindrome(char* str)

1)Purpose

This function checks if a given string is a palindrome (case insensitive).

2)Inputs / Outputs

Input:
- str: A pointer to a null-terminated string.

Output:
- Returns true if the string is a palindrome, false otherwise.

.

3)Logic and Steps

1. Null or Empty String Check:
   - If the string is NULL or empty, return true as it is trivially a palindrome.
2. Case Conversion:
   - Convert the string to lowercase using the toLowerCase function to ensure case insensitivity.
3. Palindrome Check:
   - Compare characters from the start and end of the string, moving toward the middle.
   - If any mismatch is found, return false.
4. End of Function:
   - If all characters match, return true.

4)Edge Cases

- If the input string is NULL, the function returns true.
- If the string is empty (""), the function returns true.
- Single-character strings are always palindromes.
- The function ignores case differences.

3. OPERATION ON STRINGS

10 countVowelsConsonants(char* str, int* vowels, int* consonants)

1)Purpose

This function counts the number of vowels and consonants in a given string (case insensitive).

2)Inputs / Outputs

Input:
- str: A pointer to a null-terminated string.
- vowels: A pointer to an integer to store the count of vowels.
- consonants: A pointer to an integer to store the count of consonants.

Output:
- Updates the integers pointed to by vowels and consonants with the respective counts.

.

3)Logic and Steps

11. Initialization:
   - Initialize vowels and consonants to 0.
   - If the input string is NULL or empty, return immediately.
2. Case Conversion:
   - Convert the string to lowercase using the toLowerCase function.
3. Counting Characters:
   - Iterate through each character of the string.
   - If the character is a vowel (a, e, i, o, u), increment the vowel count.
   - If the character is an alphabetic consonant, increment the consonant count.
4. End of Function:
   - Stop processing when the null terminator ('\0') is reached.

4)Edge Cases

- If the input string is NULL, the function does nothing.
- If the string is empty (""), the vowel and consonant counts remain 0.
- Non-alphabetical characters are ignored.

3. OPERATION ON STRINGS

11 findSubstring(const char* str, const char* sub)

1)Purpose

This function finds the starting index of the first occurrence of a substring within a string.

2)Inputs / Outputs

Input:
- str: A pointer to a null-terminated string.
- sub: A pointer to a null-terminated substring to search for.

Output:
- Returns the starting index of the first occurrence of 'sub' in 'str'.
- Returns -1 if the substring is not found or if inputs are invalid.

.

3)Logic and Steps

1. Input Validation:
   - If 'str' or 'sub' is NULL or empty, return -1.

2. Iterative Search:
   - Use a nested loop to check each character in 'str' for a match with 'sub'.
   - If all characters in 'sub' match consecutively in 'str', return the index.

3. No Match Found:
   - If the end of 'str' is reached without a match, return -1.

4)Edge Cases

- 'str' or 'sub' is NULL or empty.
- 'sub' is longer than 'str'.
- Multiple occurrences of 'sub' in 'str'.
- Overlapping substrings are ignored.

3. OPERATION ON STRINGS

12 removeWhitespaces(char* str)

1)Purpose

This function removes all whitespace characters from a string in place.

2)Inputs / Outputs

Input:
- str: A pointer to a null-terminated string.

Output:
- The input string is modified in place with all whitespaces removed.

.

3)Logic and Steps

1. Input Validation:
   - If 'str' is NULL, return immediately.

2. Character Shifting:
   - Iterate through each character in 'str'.
   - Copy non-whitespace characters to the current 'write' position.
   - Adjust the null terminator at the end.

4)Edge Cases

- If 'str' is NULL, the function does nothing.
- If 'str' is empty, no changes are made.
- Strings with only whitespace are converted to an empty string.

3. OPERATION ON STRINGS

13 isAnagram(char* str1, char* str2)

1)Purpose

This function checks if two strings are anagrams of each other.

2)Inputs / Outputs

Input:
- str1: A pointer to a null-terminated string.
- str2: A pointer to a null-terminated string.

Output:
- Returns true if the strings are anagrams, false otherwise.

.

3)Logic and Steps

1. Input Validation:
   - If either string is NULL or their lengths differ, return false.

2. Frequency Count:
   - Count the occurrences of each character in both strings.
   - Compare the frequency counts of both strings.

3. Result:
   - If all frequency counts match, return true; otherwise, return false.

4)Edge Cases

- NULL or empty strings.
- Strings with different lengths.
- Case-sensitive comparison.

3. OPERATION ON STRINGS

14 charFrequency(char* str, int* freq)

1)Purpose

This function calculates the frequency of each character in a string.

2)Inputs / Outputs

Input:
- str: A pointer to a null-terminated string.
- freq: A pointer to an integer array of size 128, initialized to 0.

Output:
- Updates the 'freq' array with character counts.

.

3)Logic and Steps

. Input Validation:
  - If 'str' or 'freq' is NULL, return immediately.

2. Frequency Calculation:
  - For each character in 'str', increment the corresponding index in 'freq'.

3. End of Function:
  - Stop processing when the null terminator ('\0') is reached.

4)Edge Cases

- If 'str' or 'freq' is NULL, the function does nothing.
- Non-ASCII characters are ignored.
- Empty strings result in no frequency changes.

3. OPERATION ON STRINGS

15 countWords(char* str))

1)Purpose

This function counts the number of words in a given string.

2)Inputs / Outputs

Input:
- str: A pointer to a null-terminated string.

Output:
- Returns the number of words in the string.
- Returns 0 if the string is NULL or empty.

.

3)Logic and Steps

1. Input Validation:
   - If 'str' is NULL or empty, return 0.

2. Initial Spaces Skipping:
   - Skip leading spaces, tabs, or newline characters.

3. Word Counting:
   - Iterate through the string and increment the word count whenever a non-whitespace character is followed by whitespace.

4. Return Result:
   - Return the total number of words found.

4)Edge Cases

- Leading whitespace: Handled by skipping initial spaces.
- Trailing whitespace: Handled by ensuring no extra words are counted.
- Consecutive spaces or tabs: Handled by skipping consecutive whitespace characters.
- Empty or NULL string: Returns 0 as expected.

3. OPERATION ON STRINGS

16 removeDuplicates(char* str))

1)Purpose

This function removes duplicate characters from a string, preserving the first occurrence of each character.

2)Inputs / Outputs

Input:
- str: A pointer to a null-terminated string.

Output:
- The input string is modified in place with duplicates removed.

.

3)Logic and Steps

1. Input Validation:
   - If 'str' is NULL, the function does nothing.

2. Duplicate Removal:
   - Use a boolean array to track whether a character has been encountered before.
   - Copy non-duplicate characters to a new position in the string.

3. End of Function:
   - Add a null terminator at the end.

4)Edge Cases

- NULL string: Handled by returning immediately.
- Empty string: No changes made.
- All unique characters: Original string remains unchanged.
- All identical characters: Only the first occurrence is preserved.

3. OPERATION ON STRINGS

17 compressString(char* str, char* result)

1)Purpose

This function compresses a string by replacing consecutive duplicate characters with the character followed by its count.

2)Inputs / Outputs

Input:
- str: A pointer to a null-terminated string to compress.
- result: A pointer to a null-terminated string where the compressed string will be stored.

Output:
- The 'result' string contains the compressed version of 'str'.

.

3)Logic and Steps

1. Input Validation:
   - Iterate through 'str', counting consecutive duplicate characters.

2. Compression:
   - For each group of consecutive duplicates, append the character and its count to 'result'.
   - Skip duplicate characters during the count.

3. Finalize:
   - Add a null terminator at the end of 'result'.

4)Edge Cases

- Empty 'str': The 'result' string remains empty.
- No duplicate characters: 'result' is identical to 'str'.
- Single character string: Handled correctly by directly appending the character.
- String with all identical characters: Properly compresses to the character followed by its count.

3. OPERATION ON STRINGS

18 longestWord(char* str, char* result)

1)Purpose

This function finds and extracts the longest word in a given string.

2)Inputs / Outputs

- str: A pointer to a null-terminated string.
- result: A pointer to a null-terminated string where the longest word will be stored.

Output:

- The 'result' string contains the longest word found in 'str'.

.

3)Logic and Steps

1. Input Validation:
   - Iterate through 'str', identifying words and their lengths.

2. Identify Longest Word:
   - Track the start index and length of the current word.
   - Update the longest word's position and length if a longer word is found.

3. Finalize:
   - Copy the longest word into 'result' and add a null terminator.

4)Edge Cases

- Empty or NULL 'str': 'result' remains empty.
- Multiple words with the same maximum length: The first occurrence is returned.
- Single word in the string: Handled correctly as the longest word.
- No alphabetic characters: 'result' remains empty.

3. OPERATION ON STRINGS

19 isRotation(char* str1, char* str2)

1)Purpose

This function checks if one string is a rotation of another string.

2)Inputs / Outputs

- str1: A pointer to a null-terminated string.
- str2: A pointer to a null-terminated string.

Output:

- Returns true if str2 is a rotation of str1, otherwise returns false.
- Returns false if the strings have different lengths.

3)Logic and Steps

1. Input Validation:
   - If the lengths of 'str1' and 'str2' are not equal, return false.

2. Double str1:
   - Create a new string 'doubledStr' by concatenating 'str1' to itself.

3. Substring Search:
   - Check if 'str2' is a substring of 'doubledStr' by comparing characters.

4. Return Result:
   - Return true if a match is found, otherwise return false.

4)Edge Cases

- Different string lengths: Handled by immediately returning false.
- Empty strings: Handled as rotations of each other (both are empty).
- Single character strings: Handled by considering rotations as valid if characters match.

3. OPERATION ON STRINGS

20 countChar(char* str, char ch)

1)Purpose

This function counts the occurrences of a specified character in a given string.

2)Inputs / Outputs

Input:
- str: A pointer to a null-terminated string.
- ch: A character to count in the string.

Output:
- Returns the number of times 'ch' appears in 'str'.
- Returns 0 if the string is NULL.

3)Logic and Steps

1. Input Validation:
   - If 'str' is NULL, return 0.

2. Character Counting:
   - Iterate through 'str' and count how many times 'ch' appears.

3. Return Result:
   - Return the count of 'ch'.

4)Edge Cases

- NULL string: Handled by returning 0.
- Empty string: Returns 0 as no character can be found.
- Character not present: Returns 0.
- Character appears multiple times: Correctly counts all occurrences

3. OPERATION ON STRINGS

21 findAndReplace(char* str, char* find, char* replace)

1)Purpose

This function finds all occurrences of a substring in a string and replaces them with another substring.

2)Inputs / Outputs

Input:
- str: A pointer to a null-terminated string.
- find: A pointer to the substring to search for.
- replace: A pointer to the substring to replace 'find' with.

Output:
- The input string 'str' is modified in place with all occurrences of 'find' replaced by 'replace'.

3)Logic and Steps

1. Input Validation:
   - If any input string is NULL, return immediately.
2. Find and Replace Loop:
   - Loop through 'str', searching for occurrences of 'find'.
   - Once found, shift the string to remove 'find' and make space for 'replace'.
3. Result:
   - Modify 'str' to include the 'replace' string and continue replacing until the end.

4)Edge Cases

- NULL or empty input strings: Handled by returning early without changes.
- No occurrences of 'find': The string remains unchanged.
- 'find' is longer than 'replace': Handled by shifting the string and reducing its length accordingly.
- 'find' is the same as 'replace': The string remains unchanged.

3. OPERATION ON STRINGS

22 longestPalindrome(char* str, char* result)

1)Purpose

This function finds the longest palindrome substring in a string.

2)Inputs / Outputs

Input:
- str: A pointer to a null-terminated string.
- result: A pointer to a null-terminated string where the longest palindrome will be stored.

Output:
- The 'result' string contains the longest palindrome found in 'str'.

3)Logic and Steps

1. Palindrome Search:
   - Iterate through all possible substrings and check if each is a palindrome using 'isPalindrome2'(it jsut checks if a string equals its reversed string).

2. Track Longest Palindrome:
   - Keep track of the longest palindrome's start index and length.

3. Finalize:
   - Copy the longest palindrome substring into 'result'.

4)Edge Cases

- Empty string: The result remains empty.
- No palindromes: An empty result is returned.
- Single character string: The result is the same character.
- Multiple palindromes of equal length: The first one found is returned.

3. OPERATION ON STRINGS

23 printPermutations(char* str)

1)Purpose

This function is a function that calls the 'perute' function to print all permutations of a string.

2)Inputs / Outputs

Input:
- str: A pointer to a null-terminated string whose permutations are to be printed.

Output:
- Prints all permutations of the string.

3)Logic and Steps

1. Calculate the length of the string.
2. Call 'perute' with the string, start index 0, and the calculated length to generate and print permutations.

-How does perute function works :

Input:
- str: A pointer to a null-terminated string whose permutations are to be generated.
- start: The starting index for the recursion.
- end: The ending index of the string.

Output:
- Prints all permutations of the string.

Logic:
1. Base Case:
   - If 'start' equals 'end', print the current permutation of the string.
2. Recursive Case:
   - Loop through the characters from 'start' to 'end', swapping each character with the character at 'start'.
   - Recursively call the function with 'start+1' as the new starting index.
   - Swap the characters back to restore the original string.

4)Edge Cases

- Empty string: No permutations to print.
- Single character string: The string is printed as the only permutation..

3. OPERATION ON STRINGS

24 splitString(char* str, char delimiter, char tokens[][100], int* tokenCount)

1)Purpose

This function splits a string into tokens based on a specified delimiter.

2)Inputs / Outputs

Input:
- str: A pointer to the null-terminated string to split.
- delimiter: The character used to split the string.
- tokens: A 2D array of characters to store the resulting tokens.
- tokenCount: A pointer to an integer that will hold the number of tokens generated.

Output:
- The 'tokens' array contains the split substrings, and 'tokenCount' holds the number of tokens.

3)Logic and Steps

1. Initialize an index 'i' to traverse through the string, and 'k' to store token indices.
2. Loop through 'str' and extract tokens between occurrences of the 'delimiter'.
3. Store each token in 'tokens' and update the token count.
4. If the 'delimiter' is found, increment 'i' to continue searching for the next token.

4)Edge Cases

- Empty string: No tokens, 'tokenCount' will be 0.
- Delimiter not found: The entire string will be stored as a single token.
- Multiple consecutive delimiters: Empty tokens will be created between them.

4. OPERATION ON Matrices

01 initializeMatrix(int rows, int cols, int matrix[rows][cols], int value)

1)Purpose

This function initializes a 2D matrix with a specified value. It sets all the elements of the matrix to the provided value, which could be useful for creating matrices with a specific starting state, such as setting all elements to zero or to any constant value.

2)Inputs / Outputs

Input:
- rows: An integer representing the number of rows in the matrix.
- cols: An integer representing the number of columns in the matrix.
- matrix: A 2D array that represents the matrix to be initialized. It will be modified in place.
- value: The value to set each element of the matrix to. It could be an integer or any data type appropriate for the matrix elements.

Output:
- Modifies the 'matrix' in place, setting every element of the matrix to the given 'value'. The function doesn't return anything, it operates on the matrix directly.

3)Logic and Steps

1. The function begins by looping through each row of the matrix.
2. Within each row, it loops through each column and sets the element at that position to the provided value.
3. This process ensures that every element in the matrix gets initialized to the given value.
4. Since the matrix is passed by reference, no new matrix is returned; the changes are reflected in the original matrix.

4)Edge Cases

- Empty matrix (rows or cols is 0): The matrix remains unmodified as there are no elements to initialize.

4. OPERATION ON Matrices

02 printMatrix(int rows, int cols, int matrix[rows][cols])

1)Purpose

This function prints a 2D matrix to the console in a human-readable format, where each row of the matrix is printed on a new line. This is useful for debugging or displaying matrices visually in the terminal.

2)Inputs / Outputs

Input:
- rows: An integer representing the number of rows in the matrix.
- cols: An integer representing the number of columns in the matrix.
- matrix: A 2D array representing the matrix whose elements will be printed.

Output:
- Prints each element of the matrix to the console. The elements in each row are separated by a space, and each row appears on a new line.

3)Logic and Steps

1. The function starts by printing a header message 'Matrix:' to indicate that the following output is a matrix.
2. It then loops through each row of the matrix. For each row, it loops through each element in the row.
3. Each element is printed, followed by a space. After printing all elements of a row, a newline character is printed to move to the next row.
4. The matrix is displayed in a readable form on the console, with each element and row clearly separated.

4)Edge Cases

- Empty matrix (rows or cols is 0): No matrix is printed, as there are no elements to display.

4. OPERATION ON Matrices

02 printMatrix(int rows, int cols, int matrix[rows][cols])

1)Purpose

This function prints a 2D matrix to the console in a human-readable format, where each row of the matrix is printed on a new line. This is useful for debugging or displaying matrices visually in the terminal.

2)Inputs / Outputs

Input:
- rows: An integer representing the number of rows in the matrix.
- cols: An integer representing the number of columns in the matrix.
- matrix: A 2D array representing the matrix whose elements will be printed.

Output:
- Prints each element of the matrix to the console. The elements in each row are separated by a space, and each row appears on a new line.

3)Logic and Steps

1. The function starts by printing a header message 'Matrix:' to indicate that the following output is a matrix.
2. It then loops through each row of the matrix. For each row, it loops through each element in the row.
3. Each element is printed, followed by a space. After printing all elements of a row, a newline character is printed to move to the next row.
4. The matrix is displayed in a readable form on the console, with each element and row clearly separated.

4)Edge Cases

- Empty matrix (rows or cols is 0): No matrix is printed, as there are no elements to display.

4. OPERATION ON Matrices

03 inputMatrix(int rows, int cols, int matrix[rows][cols])

1)Purpose

This function allows the user to manually input the elements of a 2D matrix from the console. It is typically used when the matrix is not predefined and needs to be filled with user-provided values.

2)Inputs / Outputs

Input:
- rows: An integer representing the number of rows in the matrix.
- cols: An integer representing the number of columns in the matrix.
- matrix: A 2D array where the entered elements will be stored. The array is modified in place.

Output:
- Modifies the 'matrix' in place by filling it with values input by the user, element by element.

3)Logic and Steps

1. The function begins by prompting the user to enter the number of rows and columns of the matrix.
2. It then iterates through the matrix elements, asking the user to enter the value for each element.
3. For each element, the user is prompted with its row and column position (e.g., 'Element [i][j]:').
4. After all values are entered, the matrix is fully populated and ready for further operations.

4)Edge Cases

- Empty matrix (rows or cols is 0): No input is requested, and the matrix remains unmodified.

4. OPERATION ON Matrices

04 addMatrices(int rows, int cols, int mat1[rows][cols], int mat2[rows][cols], int result[rows][cols])

1)Purpose

This function adds two matrices of the same size and stores the result in a third matrix. It performs element-wise addition, where each corresponding element from the two matrices is added together.

2)Inputs / Outputs

Input:
- rows: An integer representing the number of rows in both matrices.
- cols: An integer representing the number of columns in both matrices.
- mat1: The first matrix to add.
- mat2: The second matrix to add.
- result: A 2D array where the result of the addition will be stored.

Output:
- Modifies the 'result' matrix, storing the element-wise sum of 'mat1' and 'mat2'.

3)Logic and Steps

1. The function begins by iterating through each element in the input matrices ('mat1' and 'mat2').
2. For each pair of corresponding elements, it adds them together and stores the sum in the 'result' matrix at the same position.
3. Once all elements have been processed, the 'result' matrix contains the sum of the two matrices and is ready to be used.

4)Edge Cases

- Empty matrix (rows or cols is 0): the sum is 0

4. OPERATION ON Matrices

05 subtractMatrices(int rows, int cols, int mat1[rows][cols], int mat2[rows][cols], int result[rows][cols])

1)Purpose

This function subtracts the second matrix from the first matrix and stores the result in a third matrix. It performs element-wise subtraction, where each corresponding element from the two matrices is subtracted.

2)Inputs / Outputs

Input:
- rows: An integer representing the number of rows in both matrices.
- cols: An integer representing the number of columns in both matrices.
- mat1: The matrix to subtract from (minuend).
- mat2: The matrix to subtract (subtrahend).
- result: A 2D array where the result of the subtraction will be stored.

Output:
- Modifies the 'result' matrix, storing the element-wise difference of 'mat1' and 'mat2'.

3)Logic and Steps

1. The function begins by iterating through each element in the input matrices ('mat1' and 'mat2').
2. For each pair of corresponding elements, it subtracts the element of 'mat2' from the element of 'mat1' and stores the result in the 'result' matrix.
3. Once all elements have been processed, the 'result' matrix contains the difference of the two matrices.

4)Edge Cases

- Empty matrix (rows or cols is 0): the reslut is 0

4. OPERATION ON Matrices

06 void multiplyMatrices(int rows1, int cols1, int mat1[rows1][cols1], int rows2, int cols2, int mat2[rows2][cols2], int result[rows1][cols2])

1)Purpose

This function performs matrix multiplication, where two matrices (mat1 and mat2) are multiplied and the result is stored in the 'result' matrix. Matrix multiplication involves summing the product of corresponding elements from rows of the first matrix and columns of the second matrix.

2)Inputs / Outputs

Input:
- rows1 - cols1: Intergers representing dimensions of matrix 1
- rows2 - cols2: Intergers representing dimensions of matrix 2
- mat1: The matrix to subtract from (minuend).
- mat2: The matrix to subtract (subtrahend).
- result: A 2D array where the result of the subtraction will be stored.

Output:

- The 'result' matrix is populated with the product of 'mat1' and 'mat2'. Output:
- Modifies the 'result' matrix, storing the element-wise difference of 'mat1' and 'mat2'.

3)Logic and Steps

1. The function first checks if the number of columns in mat1 matches the number of rows in mat2 (cols1 != rows2).
2. If the condition is not met, an error message 'Invalid inputs!' is printed and the function returns without performing the multiplication.
3. If the condition is met, the function proceeds with matrix multiplication by iterating through each row of mat1 and each column of mat2.
4. For each element in the result matrix, the sum of products of corresponding elements from the row of mat1 and the column of mat2 is calculated and stored in the 'result' matrix.

4)Edge Cases

- Invalid matrix dimensions: If the number of columns in mat1 does not match the number of rows in mat2, the function prints an error message and returns without performing any computation.
- Empty matrices (either rows1 or cols1 is 0, or rows2 or cols2 is 0): No multiplication is performed, and the result matrix remains unchanged.

4. OPERATION ON Matrices

07_scalarMultiplyMatrix(int rows, int cols, int matrix[rows][cols], int scalar)

1)Purpose

This function multiplies each element of a matrix by a scalar value. It is useful for scaling a matrix, such as increasing or decreasing all elements by a constant factor.

2)Inputs / Outputs

Input:
- rows: An integer representing the number of rows in the matrix.
- cols: An integer representing the number of columns in the matrix.
- matrix: A 2D array representing the matrix to be scaled.
- scalar: The value by which each element of the matrix will be multiplied.

Output:
- The elements of the 'matrix' are modified in place, with each element multiplied by the scalar.

3)Logic and Steps

1. The function iterates through each element in the matrix.
2. For each element, it multiplies the current element by the provided scalar.
3. The matrix is updated in place, and no new matrix is returned.

4)Edge Cases

- Empty matrix (rows or cols is 0): No multiplication is performed, and the matrix remains unchanged.
- Scalar value of 0: All elements of the matrix are set to 0.

4. OPERATION ON Matrices

08 isSquareMatrix(int rows, int cols)

1)Purpose

This function checks if a given matrix is square, i.e., if the number of rows is equal to the number of columns. A square matrix has special properties that are useful in many mathematical operations.

2)Inputs / Outputs

Input:

- rows: An integer representing the number of rows in the matrix.
- cols: An integer representing the number of columns in the matrix.

Output:

- Returns true if the matrix is square (rows == cols), otherwise returns false.

3)Logic and Steps

1. The function checks if the number of rows is equal to the number of columns.
2. If the condition is met, it returns true, indicating that the matrix is square.
3. If the condition is not met, it returns false.

4)Edge Cases

- Matrix with 0 rows or 0 columns: The matrix is not square, and the function returns false

4. OPERATION ON Matrices

09 isIdentityMatrix(int size, int matrix[size][size])

1)Purpose

This function checks if a matrix is an identity matrix. An identity matrix is a square matrix with ones on the diagonal and zeros elsewhere.

2)Inputs / Outputs

Input:
- size: The size of the square matrix (size x size).
- matrix: A 2D array representing the matrix to check.

Output:
- Returns true if the matrix is an identity matrix, otherwise returns false.

3)Logic and Steps

1. The function iterates through each element in the matrix.
2. It checks that each diagonal element (where i == j) is 1, and all non-diagonal elements are 0.
3. If any element does not meet the condition, the function returns false.
4. If all conditions are met, the function returns true, indicating that the matrix is an identity matrix

4)Edge Cases

- Matrix is not square: The function assumes the matrix is square. If it's not, the result may not be valid.

4. OPERATION ON Matrices

10 isDiagonalMatrix(int size, int matrix[size][size])

1)Purpose

This function checks if a matrix is a diagonal matrix. A diagonal matrix is a square matrix where all elements off the main diagonal are zero.

2)Inputs / Outputs

Input:
- size: The size of the square matrix (size x size).
- matrix: A 2D array representing the matrix to check.

Output:
- Returns true if the matrix is diagonal, otherwise returns false.

3)Logic and Steps

1. The function iterates through each element in the matrix.
2. It checks that all elements off the diagonal (i != j) are zero.
3. If any non-zero off-diagonal element is found, the function returns false.
4. If all off-diagonal elements are zero, the function returns true, indicating that the matrix is diagonal.

4)Edge Cases

-- Matrix is not square: The function assumes the matrix is square. If it's not, the result may not be valid.

4. OPERATION ON Matrices

11_isSymmetricMatrix(int size, int matrix[size][size])

1)Purpose

This function checks if a matrix is symmetric, i.e., if the matrix is equal to its transpose. A symmetric matrix has the property that the element at position [i][j] is equal to the element at position [j][i].

2)Inputs / Outputs

Input:

- size: The size of the square matrix (size x size).
- matrix: A 2D array representing the matrix to check.

Output:

- Returns true if the matrix is symmetric, otherwise returns false.

3)Logic and Steps

1. The function iterates through the upper triangle of the matrix (i < j) and checks that each element is equal to its mirror element across the diagonal.
2. If any pair of elements does not satisfy this condition, the function returns false.
3. If all elements satisfy the symmetry condition, the function returns true.

4)Edge Cases

- Matrix is not square: The function assumes the matrix is square. If it's not, the result may not be valid.

4. OPERATION ON Matrices

12 isUpperTriangular(int size, int matrix[size][size])

1)Purpose

This function checks if a matrix is upper triangular, i.e., if all elements below the main diagonal are zero.

2)Inputs / Outputs

Input:

- size: The size of the square matrix (size x size).
- matrix: A 2D array representing the matrix to check.

Output:

- Returns true if the matrix is upper triangular, otherwise returns false.

3)Logic and Steps

1. The function iterates through the elements below the main diagonal (i > j) and checks if they are all zero.
2. If any non-zero element below the diagonal is found, the function returns false.
3. If all elements below the diagonal are zero, the function returns true.

4)Edge Cases

- Matrix is not square: The function assumes the matrix is square. If it's not, the result may not be valid.

4. OPERATION ON Matrices

13 transposeMatrix(int rows, int cols, int matrix[rows][cols], int result[cols][rows])

1)Purpose

This function calculates the transpose of a matrix, where rows become columns and columns become rows. The transpose of a matrix is useful in many linear algebra operations.

2)Inputs / Outputs

Input:
- rows: The number of rows in the matrix.
- cols: The number of columns in the matrix.
- matrix: A 2D array representing the matrix to transpose.
- result: A 2D array where the transposed matrix will be stored.

Output:
- The 'result' matrix is populated with the transposed version of 'matrix'.

3)Logic and Steps

1. The function iterates through the matrix, swapping the row and column indices to generate the transposed matrix.
2. Each element from position [i][j] in the original matrix is moved to position [j][i] in the result matrix.

4)Edge Cases

- Empty matrix (rows or cols is 0): No transposition is performed, and the result matrix remains unchanged.

4. OPERATION ON Matrices

14 determinantMatrix(int size, int matrix[size][size])

1)Purpose

This function calculates the determinant of a square matrix of any size using recursion. It reduces larger matrices to 2x2 matrices to apply the formula for the determinant.

2)Inputs / Outputs

Input:
- size: The size of the square matrix (size x size).
- matrix: A 2D array representing the matrix for which the determinant is to be calculated.

Output:
- Returns the determinant of the matrix, an integer.

3)Logic and Steps

1. If the matrix is 2x2, the determinant is calculated directly using the formula: det = (mat[0][0] * mat[1][1]) - (mat[0][1] * mat[1][0]).
2. For larger matrices, the function performs cofactor expansion along the first row.
3. For each element in the first row, the minor (submatrix after removing the row and column of the element) is extracted, and its determinant is recursively calculated.
4. The determinant is the sum of these minors, each multiplied by the corresponding element from the first row and a factor (+1 or -1).
5. The final result is returned after all recursion calls are completed

4)Edge Cases

- Zero matrix: The determinant will be zero.
- Identity matrix: The determinant will be one.

4. OPERATION ON Matrices

15 inverseMatrix(int size, double matrix[size][size], double result[size][size])

1)Purpose

This function calculates the inverse of a square matrix using Gaussian elimination. It simultaneously transforms the matrix into the identity matrix while applying operations to the result matrix.

2)Inputs / Outputs

Input:
- size: The size of the square matrix (size x size).
- matrix: A 2D array representing the matrix to be inverted.
- result: A 2D array to store the resulting inverted matrix.

Output:
- Modifies the 'result' matrix to hold the inverse of the input matrix

3)Logic and Steps

1. The function first creates an identity matrix as the initial 'result'.
2. It then constructs an augmented matrix by concatenating the input matrix with the identity matrix.
3. The function performs row operations to reduce the augmented matrix into row echelon form.
4. If the pivot element is zero, it indicates the matrix cannot be inverted, and an error message is displayed.
5. After row operations, the left part of the augmented matrix becomes the identity matrix, and the right part becomes the inverse.
6. The function copies the inverse from the augmented matrix into the 'result' matrix.

4)Edge Cases

- Singular matrix (determinant = 0): The matrix cannot be inverted.
- Identity matrix: The inverse of the identity matrix is the identity matrix itself.

4. OPERATION ON Matrices

16 matrixPower(int size, int matrix[size][size], int power, int result[size][size])

1)Purpose

This function calculates the matrix raised to a specific power by multiplying the matrix with itself repeatedly.

2)Inputs / Outputs

Input:
- size: The size of the square matrix (size x size).
- matrix: A 2D array representing the matrix to be raised to a power.
- power: The exponent to which the matrix should be raised (must be a positive integer).
- result: A 2D array to store the result of the matrix raised to the specified power.

Output:
- Modifies the 'result' matrix to store the matrix raised to the specified power.

3)Logic and Steps

1. The function first checks if the 'power' is a valid positive integer.
2. It initializes the result matrix as a copy of the input matrix.
3. Then, it multiplies the matrix by itself repeatedly until the specified power is reached.
4. Each multiplication is stored in a temporary matrix, which is copied into the result matrix after each step.
5. The result matrix is returned after all multiplications are completed.

4)Edge Cases

- Power = 1: The result is the matrix itself.
- Power = 0: Typically, any matrix raised to the zero power results in the identity matrix, but this is not explicitly handled by the function.
- Negative or zero powers: The function returns an error.

4. OPERATION ON Matrices

17 cofactorMatrix(int size, int matrix[size][size], int cofactor[size][size])

1)Purpose

This function calculates the cofactor matrix for a given square matrix. The cofactor matrix is used in calculating the determinant and adjoint of a matrix.

2)Inputs / Outputs

Input:
- size: The size of the square matrix (size x size).
- matrix: The 2D array representing the matrix whose cofactor matrix is to be calculated.
- cofactor: A 2D array to store the resulting cofactor matrix.

Output:
- Populates the 'cofactor' array with the cofactor matrix.

3)Logic and Steps

1. If the size of the matrix is less than 2, the function prints an error and exits.
2. If the size is 2, it directly calculates the cofactor matrix using the 2x2 matrix formula.
3. For larger matrices, the function iterates through each element of the matrix, creating minors (submatrices) and calculating their determinants recursively.
4. Each element is assigned a sign (+ or -) based on its position (even or odd index sum).
5. The cofactor matrix is built by calculating the determinant of the minor for each element.

4)Edge Cases

- Size 1 matrix: The function assumes the matrix size is greater than 1.

4. OPERATION ON Matrices

18 adjointMatrix(int size, int matrix[size][size], int adjoint[size][size])

1)Purpose

This function calculates the adjoint (or adjugate) matrix, which is the transpose of the cofactor matrix.

2)Inputs / Outputs

Input:
- size: The size of the square matrix (size x size).
- matrix: The 2D array representing the matrix whose adjoint is to be calculated.
- adjoint: A 2D array to store the resulting adjoint matrix.

Output:
- Populates the 'adjoint' array with the adjoint matrix.

3)Logic and Steps

1. The function first calculates the cofactor matrix by calling the 'cofactorMatrix' function.
2. It then transposes the cofactor matrix to obtain the adjoint matrix.
3. The adjoint matrix is then stored in the 'adjoint' array.

4)Edge Cases

- Singular matrix: The adjoint matrix can still be calculated for singular matrices, but it may not be useful for finding the inverse.

4. OPERATION ON Matrices

19 luDecomposition(int size, double matrix[size][size], double lower[size][size], double upper[size][size])

1)Purpose

This function performs LU decomposition, which factors a square matrix into the product of a lower triangular matrix (L) and an upper triangular matrix (U).

2)Inputs / Outputs

Input:
- size: The size of the square matrix (size x size).
- matrix: The 2D array representing the matrix to be decomposed.
- lower: A 2D array to store the resulting lower triangular matrix.
- upper: A 2D array to store the resulting upper triangular matrix.

Output:
- Populates the 'lower' and 'upper' arrays with the LU decomposition of the input matrix.

3)Logic and Steps

1. The function initializes the 'lower' matrix as the identity matrix and sets the 'upper' matrix as a copy of the input matrix.
2. It iterates through each element of the matrix to perform Gaussian elimination.
3. If a zero pivot is encountered, it attempts to swap rows to ensure numerical stability.
4. For each element, the function calculates the factor used to eliminate elements below the pivot, and updates both the 'lower' and 'upper' matrices accordingly.
5. After decomposition, the resulting 'lower' and 'upper' matrices are returned.

4)Edge Cases

- Singular matrix: If the matrix is singular (non-invertible), LU decomposition may fail or produce incorrect results.
- Zero pivot: If a zero pivot is encountered and no row swaps can be made, the function will fail and print an error message.

4. OPERATION ON Matrices

20 matrixRank(int rows, int cols, int matrix[rows][cols])

1)Purpose

This function computes the rank of a matrix using Gaussian elimination, determining the maximum number of linearly independent rows or columns.

2)Inputs / Outputs

Input:
- rows: The number of rows in the matrix.
- cols: The number of columns in the matrix.
- matrix: A 2D array representing the matrix to compute the rank for.

Output:
- Returns the rank of the matrix, an integer.

3)Logic and Steps

1. The function first converts the input matrix into a temporary matrix of doubles to allow precise calculations.
2. It initializes the rank to 0, which will be incremented as independent rows are found.
3. The function performs Gaussian elimination to bring the matrix to row echelon form.
4. For each column, it finds the row with the largest absolute value in the current column to use as a pivot.
5. If a non-zero pivot is found, the row is swapped with the current rank row, and the rank is incremented.
6. The pivot row is scaled to make the pivot element equal to 1.
7. Other rows are reduced using the pivot row by subtracting appropriate multiples.
8. The rank is returned after all rows are processed.

4)Edge Cases

- Zero matrix: The rank will be 0, as there are no independent rows.
- Identity matrix: The rank will be equal to the size of the matrix, as all rows are independent.
- Non-square matrix: The function works for both square and rectangular matrices.

4. OPERATION ON Matrices

21_traceMatrix(int size, int matrix[size][size])

1)Purpose

This function calculates the trace of a square matrix, which is the sum of the diagonal elements.

2)Inputs / Outputs

Input:
- size: The size of the square matrix (size x size).
- matrix: A 2D array representing the matrix for which the trace is to be calculated.

Output:
- Returns the trace of the matrix, an integer, which is the sum of the diagonal elements.

3)Logic and Steps

1. The function initializes a variable 'sum' to 0.
2. It loops over the diagonal elements of the matrix (elements where row index equals column index).
3. Each diagonal element is added to 'sum'.
4. After looping through all diagonal elements, the 'sum' is returned as the trace of the matrix.

4)Edge Cases

- Zero matrix: The trace will be 0, as all diagonal elements are zero.
- Identity matrix: The trace will be equal to the size of the matrix, as all diagonal elements are 1.

4. OPERATION ON Matrices

22 traceMatrix(int size, int matrix[size][size])

1)Purpose

This function rotates a square matrix by 90 degrees in the clockwise direction.

2)Inputs / Outputs

Input:
- size: The size of the square matrix (size x size).
- matrix: A 2D array representing the matrix to be rotated.

Output:
- The matrix is modified in-place to its rotated form (90 degrees clockwise).

3)Logic and Steps

1. The function creates a temporary matrix to hold the rotated values.
2. It loops over the rows and columns of the original matrix.
3. For each element at position (i, j) in the original matrix, the element is placed at position (j, size-1-i) in the temporary matrix.
4. After the rotation, the original matrix is updated with the values from the temporary matrix.

4)Edge Cases

- Identity matrix: The rotated matrix will have its rows and columns swapped.
- Zero matrix: The rotated matrix will be the same as the original matrix (still a zero matrix).
- 1x1 matrix: A 1x1 matrix remains unchanged after rotation.