Numbers

1. Function: `sumOfDigits`

Title and Overview

Function Name: `sumOfDigits`
Purpose: Calculates the sum of the digits of a given integer.

Features

- Takes an integer as input.
- Computes the sum of its digits using a loop.

Inputs and Outputs

Inputs:

- `num`: An integer whose digits are to be summed.

Outputs:

- Returns an integer representing the sum of the digits.

Workflow

1. Initialize a variable `S` to 0.
2. Use a `while` loop to extract each digit of `num` using the modulo operator (`%`).
3. Add the extracted digit to `S`.
4. Divide `num` by 10 to remove the last digit.
5. Repeat until `num` becomes 0.
6. Return the sum `S`.

Strengths and Limitations

Strengths:

- Simple and efficient for small numbers.

Limitations:

- Does not handle negative inputs (returns incorrect results).

Code Summary

- **Language:** C
- **Key Tools:** Loops, modulo operator, integer division.

---

2. Function: `reverseNumber`

Title and Overview

Function Name: `reverseNumber`
Purpose: Reverses the digits of a given integer.

Features

- Takes an integer as input.
- Reverses its digits using a loop.

Inputs and Outputs

Inputs:

- `num`: An integer to be reversed.

Outputs:

- Returns an integer representing the reversed number.

Workflow

1. Initialize a variable `R` to 0.
2. Use a `while` loop to extract each digit of `num` using the modulo operator (`%`).
3. Append the extracted digit to `R` by multiplying `R` by 10 and adding the digit.
4. Divide `num` by 10 to remove the last digit.
5. Repeat until `num` becomes 0.
6. Return the reversed number `R`.

Strengths and Limitations

Strengths:

- Efficient and straightforward.

Limitations:

- Does not handle negative inputs (returns incorrect results).

Code Summary

- **Language:** C

- **Key Tools:** Loops, modulo operator, integer division.

---

3. Function: `isPalindrome`

Title and Overview

Function Name: `isPalindrome`
Purpose: Checks if a given integer is a palindrome.

Features

- Uses the `reverseNumber` function to reverse the input number.
- Compares the reversed number with the original number.

Inputs and Outputs

Inputs:

- `num`: An integer to be checked.

Outputs:

- Returns a boolean value (`true` if the number is a palindrome, `false` otherwise).

Workflow

1. Call the `reverseNumber` function to reverse `num`.
2. Compare the reversed number with the original number.
3. Return `true` if they are equal, otherwise return `false`.

Strengths and Limitations

Strengths:

- Reuses the `reverseNumber` function, promoting code reuse.

Limitations:

- Does not handle negative inputs (returns incorrect results).

Code Summary

- **Language:** C
- **Key Tools:** Helper function (`reverseNumber`), comparison.

---

4. Function: `isPrime`

Title and Overview

Function Name: `isPrime`
Purpose: Checks if a given integer is a prime number.

Features

- Uses a loop to check divisibility of the number by integers up to its square root.

Inputs and Outputs

Inputs:

- `num`: An integer to be checked.

Outputs:

- Returns a boolean value (`true` if the number is prime, `false` otherwise).

Workflow

1. Handle edge cases (numbers less than 2 are not prime).
2. Use a `while` loop to check divisibility of `num` by integers starting from 2 up to the square root of `num`.
3. If `num` is divisible by any integer, return `false`.
4. If no divisors are found, return `true`.

Strengths and Limitations

Strengths:

- Efficient for small to medium-sized numbers.

Limitations:

- Performance degrades for very large numbers.

Code Summary

- **Language:** C
- **Key Tools:** Loops, modulo operator, square root optimization.

---

5. Function: `gcd`

Title and Overview

Function Name: `gcd`
Purpose: Computes the greatest common divisor (GCD) of two integers using the Euclidean algorithm.

Features

- Takes two integers as input.
- Uses the Euclidean algorithm to compute the GCD.

Inputs and Outputs

Inputs:

- `a`: First integer.
- `b`: Second integer.

Outputs:

- Returns an integer representing the GCD of `a` and `b`.

Workflow

1. Ensure `a` is greater than `b` (swap if necessary).
2. Use a `do-while` loop to repeatedly apply the Euclidean algorithm:
    - Compute the remainder of `a` divided by `b`.
    - Update `a` to `b` and `b` to the remainder.
3. Repeat until the remainder is 0.
4. Return `a` as the GCD.

Strengths and Limitations

Strengths:

- Efficient and widely used algorithm.

Limitations:

- Does not handle negative inputs (returns incorrect results).

Code Summary

- **Language:** C
- **Key Tools:** Euclidean algorithm, loops, modulo operator.

---

6. Function: `lcm`

Title and Overview

Function Name: `lcm`
Purpose: Computes the least common multiple (LCM) of two integers.

Features

- Uses the `gcd` function to compute the LCM.

Inputs and Outputs

Inputs:

- `a`: First integer.
- `b`: Second integer.

Outputs:

- Returns an integer representing the LCM of `a` and `b`.

Workflow

1. Compute the GCD of `a` and `b` using the `gcd` function.
2. Use the formula: `LCM(a, b) = abs(a * b) / GCD(a, b)`.
3. Return the computed LCM.

Strengths and Limitations

Strengths:

- Efficient due to reuse of the `gcd` function.

Limitations:

- Does not handle negative inputs (returns incorrect results).

Code Summary

- **Language:** C
- **Key Tools:** Helper function (`gcd`), mathematical formula.

---

7. Function: `factorial`

Title and Overview

Function Name: `factorial`
Purpose: Computes the factorial of a given integer.

Features

- Uses a loop to compute the factorial iteratively.

Inputs and Outputs

Inputs:

- `num`: An integer for which the factorial is to be computed.

Outputs:

- Returns a long integer representing the factorial of `num`.

Workflow

1. Initialize a variable `f` to 1.
2. Use a `while` loop to multiply `f` by `num` and decrement `num` until `num` becomes 0.
3. Return the computed factorial `f`.

Strengths and Limitations

Strengths:

- Simple and efficient for small numbers.

Limitations:

- May cause overflow for large numbers (e.g., `num > 20`).

Code Summary

- **Language:** C
- **Key Tools:** Loops, multiplication.

---

8. Function: `isEven`

Title and Overview

Function Name: `isEven`
Purpose: Checks if a given integer is even.

Features

- Uses the modulo operator to check divisibility by 2.

Inputs and Outputs

Inputs:

- `num`: An integer to be checked.

Outputs:

- Returns a boolean value (`true` if the number is even, `false` otherwise).

Workflow

1. Check if `num % 2 == 0`.
2. Return `true` if the condition is met, otherwise return `false`.

Strengths and Limitations

Strengths:

- Extremely simple and efficient.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Modulo operator.

---

9. Function: `primeFactors`

Title and Overview

Function Name: `primeFactors`
Purpose: Prints the prime factors of a given integer.

Features

- Uses a loop to find and print prime factors.

Inputs and Outputs

Inputs:

- `num`: An integer whose prime factors are to be printed.

Outputs:

- Prints the prime factors to the console.

Workflow

1. Initialize a variable `n` to `num`.
2. Use a `for` loop to iterate through integers from 2 to `num / 2`.
3. Use a `while` loop to divide `n` by the current integer if it is a factor.
4. Print the factor if found.
5. Repeat until all factors are printed.

Strengths and Limitations

Strengths:

- Simple and effective for small numbers.

Limitations:

- Inefficient for very large numbers.

Code Summary

- **Language:** C
- **Key Tools:** Loops, modulo operator, integer division.

---

10. Function: `isArmstrong`

Title and Overview

Function Name: `isArmstrong`
Purpose: Checks if a given integer is an Armstrong number.

Features

- Computes the sum of the digits raised to the power of the number of digits.
- Compares the sum to the original number.

Inputs and Outputs

Inputs:

- `num`: An integer to be checked.

Outputs:

- Returns a boolean value (`true` if the number is an Armstrong number, `false` otherwise).

Workflow

1. Count the number of digits in `num`.
2. Compute the sum of each digit raised to the power of the number of digits.
3. Compare the sum to the original number.
4. Return `true` if they are equal, otherwise return `false`.

Strengths and Limitations

Strengths:

- Accurate and straightforward.

Limitations:

- Inefficient for very large numbers.

Code Summary

- **Language:** C
- **Key Tools:** Loops, power calculation, modulo operator.

---

11. Function: `fibonacciSeries`

Title and Overview

Function Name: `fibonacciSeries`
Purpose: Prints the Fibonacci series up to a given number of terms.

Features

- Uses a loop to generate and print the Fibonacci series.

Inputs and Outputs

Inputs:

- `n`: An integer representing the number of terms in the series.

Outputs:

- Prints the Fibonacci series to the console.

Workflow

1. Initialize variables `F1` and `F2` to 0 and 1, respectively.
2. Print the first two terms if `n` is at least 1 or 2.
3. Use a `for` loop to compute and print subsequent terms by summing the previous two terms.
4. Repeat until `n` terms are printed.

Strengths and Limitations

Strengths:

- Simple and efficient for small values of `n`.

Limitations:

- May cause overflow for large values of `n`.

Code Summary

- **Language:** C
- **Key Tools:** Loops, addition.

---

12. Function: `sumDivisors`

Title and Overview

Function Name: `sumDivisors`
Purpose: Computes the sum of the divisors of a given integer (excluding the number itself).

Features

- Uses a loop to find and sum the divisors.

Inputs and Outputs

Inputs:

- `num`: An integer whose divisors are to be summed.

Outputs:

- Returns an integer representing the sum of the divisors.

Workflow

1. Initialize a variable `S` to 1.
2. Use a `for` loop to iterate through integers from 2 to `num / 2`.
3. Add the current integer to `S` if it is a divisor of `num`.
4. Return the sum `S`.

Strengths and Limitations

Strengths:

- Simple and effective for small numbers.

Limitations:

- Inefficient for very large numbers.

Code Summary

- **Language:** C
- **Key Tools:** Loops, modulo operator.

---

13. Function: `isPerfect`

Title and Overview

Function Name: `isPerfect`
Purpose: Checks if a given integer is a perfect number.

Features

- Uses the `sumDivisors` function to check if the sum of divisors equals the number.

Inputs and Outputs

Inputs:

- `num`: An integer to be checked.

Outputs:

- Returns a boolean value (`true` if the number is perfect, `false` otherwise).

Workflow

1. Call the `sumDivisors` function to compute the sum of divisors of `num`.
2. Compare the sum to `num`.
3. Return `true` if they are equal, otherwise return `false`.

Strengths and Limitations

Strengths:

- Reuses the `sumDivisors` function, promoting code reuse.

Limitations:

- Inefficient for very large numbers.

Code Summary

- **Language:** C
- **Key Tools:** Helper function (`sumDivisors`), comparison.

---

14. Function: `isMagic`

Title and Overview

Function Name: `isMagic`
Purpose: Checks if a given integer is a magic number (a number whose recursive sum of digits equals 1).

Features

- Uses a loop to compute the recursive sum of digits.

Inputs and Outputs

Inputs:

- `num`: An integer to be checked.

Outputs:

- Returns a boolean value (`true` if the number is a magic number, `false` otherwise).

Workflow

1. Use a `do-while` loop to compute the sum of digits of `num`.
2. Repeat the process until the sum is a single digit.
3. Return `true` if the final sum is 1, otherwise return `false`.

Strengths and Limitations

Strengths:

- Simple and effective.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Loops, modulo operator.

---

15. Function: `isAutomorphic`

Title and Overview

Function Name: `isAutomorphic`
Purpose: Checks if a given integer is an automorphic number (a number whose square ends with the number itself).

Features

- Computes the square of the number and checks the last digits.

Inputs and Outputs

Inputs:

- `num`: An integer to be checked.

Outputs:

- Returns a boolean value (`true` if the number is automorphic, `false` otherwise).

Workflow

1. Count the number of digits in `num`.
2. Compute the square of `num`.
3. Check if the last digits of the square match `num`.
4. Return `true` if they match, otherwise return `false`.

Strengths and Limitations

Strengths:

- Accurate and straightforward.

Limitations:

- Inefficient for very large numbers.

Code Summary

- **Language:** C
- **Key Tools:** Loops, power calculation, modulo operator.

---

16. Function: `toBinary`

Title and Overview

Function Name: `toBinary`
Purpose: Converts a given integer into its **binary representation** and prints it to the console.

Features

- Uses a loop to extract binary digits from the input number.
- Reverses the binary digits to display the correct binary representation.

Inputs and Outputs

Inputs:

- `num`: An integer to be converted to binary.

Outputs:

- Prints the binary representation of `num` to the console.

Workflow

1. Initialize a variable `S` to 0.
2. Use a `while` loop to extract each binary digit of `num` using the modulo operator (`%`).
3. Append the extracted digit to `S` by multiplying `S` by 10 and adding the digit.
4. Divide `num` by 2 to remove the last binary digit.
5. Repeat until `num` becomes 0.
6. Reverse `S` to get the correct binary representation and print it.

Strengths and Limitations

Strengths:

- Simple and effective for small numbers.

Limitations:

- Inefficient for very large numbers due to the limitations of integer size in C.
- Does not handle negative inputs.

Code Summary

- **Language:** C
- **Key Tools:** Loops, modulo operator (`%`), integer division (`/`).

---

17. Function: `isNarcissistic`

Title and Overview

Function Name: `isNarcissistic`
Purpose: Checks if a given integer is a **narcissistic number** (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 narcissistic because $1^3 + 5^3 + 3^3 = 153$.

Features

- Counts the number of digits in the input number.
- Computes the sum of each digit raised to the power of the number of digits.
- Compares the sum to the original number.

Inputs and Outputs

Inputs:

- `num`: An integer to be checked.

Outputs:

- Returns a boolean value (`true` if the number is narcissistic, `false` otherwise).

Workflow

1. Count the number of digits in `num`.
2. Compute the sum of each digit raised to the power of the number of digits.
3. Compare the sum to the original number.
4. Return `true` if they are equal, otherwise return `false`.

Strengths and Limitations

Strengths:

- Accurate and straightforward.

Limitations:

- Inefficient for very large numbers due to the power calculations.

Code Summary

- **Language:** C
- **Key Tools:** Loops, power calculation, modulo operator (`%`).

---

18. Function: `sqrtApprox`

Title and Overview

Function Name: `sqrtApprox`
Purpose: Approximates the **square root** of a given integer using the **Newton-Raphson method**.

Features

- Uses an iterative approach to approximate the square root.

- Stops iterating when the difference between successive approximations is very small.

Inputs and Outputs

Inputs:

- `num`: An integer whose square root is to be approximated.

Outputs:

- Returns a double representing the approximated square root.

Workflow

1. Initialize a variable `x` to `num / 2`.
2. Use a `do-while` loop to iteratively apply the Newton-Raphson formula:
    - Update `x` to `0.5 * (x + num / x)`.
3. Repeat until the difference between successive approximations is very small (less than `0.000000001`).
4. Return the final value of `x` as the approximated square root.

Strengths and Limitations

Strengths:

- Highly accurate and efficient for approximating square roots.

Limitations:

- Requires floating-point arithmetic, which may introduce precision errors for very large numbers.

Code Summary

- **Language:** C
- **Key Tools:** Newton-Raphson method, loops, floating-point arithmetic.

---

19. Function: `power`

Title and Overview

Function Name: `power`
Purpose: Computes the **power** of a given base raised to a given exponent.

Features

- Handles both positive and negative exponents.
- Uses a loop to compute the power iteratively.

Inputs and Outputs

Inputs:

- `base`: The base number.
- `exp`: The exponent.

Outputs:

- Returns a double representing the result of `base^exp`.

Workflow

1. Handle the case where `exp` is 0 (return 1).
2. Initialize a variable `result` to 1.
3. If `exp` is negative, invert the base and make `exp` positive.
4. Use a `for` loop to multiply `result` by `base` for `exp` iterations.
5. Return the computed result.

Strengths and Limitations

Strengths:

- Handles both positive and negative exponents.
- Simple and efficient for small exponents.

Limitations:

- May cause overflow or precision issues for very large exponents.

Code Summary

- **Language:** C
- **Key Tools:** Loops, multiplication, conditional logic.

---

20. Function: `getnext`

Title and Overview

Function Name: `getnext`
Purpose: Helper function for `isHappy` that computes the **sum of the squares of the digits** of a number.

Features

- Computes the sum of the squares of the digits of a given number.

Inputs and Outputs

Inputs:

- `n`: An integer whose digits are to be squared and summed.

Outputs:

- Returns an integer representing the sum of the squares of the digits.

Workflow

1. Initialize a variable `S` to 0.
2. Use a `while` loop to extract each digit of `n` using the modulo operator (`%`).
3. Square the extracted digit and add it to `S`.
4. Divide `n` by 10 to remove the last digit.

5. Repeat until `n` becomes 0.
6. Return the sum `S`.

Strengths and Limitations

Strengths:

- Simple and efficient.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Loops, modulo operator (`%`), integer division (`/`).

---

21. Function: `isHappy`

Title and Overview

Function Name: `isHappy`
Purpose: Checks if a given integer is a **happy number** (a number that eventually reaches 1 when replaced by the sum of the squares of its digits).

Features

- Uses the `getnext` function to compute the sum of the squares of the digits.
- Uses **Floyd's cycle-finding algorithm** to detect loops.

Inputs and Outputs

Inputs:

- `num`: An integer to be checked.

Outputs:

- Returns a boolean value (`true` if the number is happy, `false` otherwise).

Workflow

1. Initialize two variables, `slow` and `fast`, to `num`.
2. Use a `do-while` loop to repeatedly apply the `getnext` function:
    - Update `slow` to `getnext(slow)`.
    - Update `fast` to `getnext(getnext(fast))`.
3. Repeat until `slow` equals `fast`.
4. Return `true` if `slow` equals 1, otherwise return `false`.

Strengths and Limitations

Strengths:

- Efficient and avoids infinite loops using Floyd's algorithm.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Helper function (`getnext`), Floyd's cycle-finding algorithm.

---

22. Function: `isAbundant`

Title and Overview

Function Name: `isAbundant`
Purpose: Checks if a given integer is an **abundant number** (a number whose sum of proper divisors is greater than the number itself).

Features

- Uses the `sumDivisors` function to compute the sum of proper divisors.

Inputs and Outputs

Inputs:

- `num`: An integer to be checked.

Outputs:

- Returns a boolean value (`true` if the number is abundant, `false` otherwise).

Workflow

1. Call the `sumDivisors` function to compute the sum of proper divisors of `num`.
2. Compare the sum to `num`.
3. Return `true` if the sum is greater than `num`, otherwise return `false`.

Strengths and Limitations

Strengths:

- Reuses the `sumDivisors` function, promoting code reuse.

Limitations:

- Inefficient for very large numbers.

Code Summary

- **Language:** C
- **Key Tools:** Helper function (`sumDivisors`), comparison.

---

23. Function: `isDeficient`

Title and Overview

Function Name: `isDeficient`
Purpose: Checks if a given integer is a **deficient number** (a number whose sum of proper divisors is less than the number itself).

Features

- Uses the `sumDivisors` function to compute the sum of proper divisors.

Inputs and Outputs

Inputs:

- `num`: An integer to be checked.

Outputs:

- Returns a boolean value (`true` if the number is deficient, `false` otherwise).

Workflow

1. Call the `sumDivisors` function to compute the sum of proper divisors of `num`.
2. Compare the sum to `num`.
3. Return `true` if the sum is less than `num`, otherwise return `false`.

Strengths and Limitations

Strengths:

- Reuses the `sumDivisors` function, promoting code reuse.

Limitations:

- Inefficient for very large numbers.

Code Summary

- **Language:** C
- **Key Tools:** Helper function (`sumDivisors`), comparison.

---

24. Function: `sumEvenFibonacci`

Title and Overview

Function Name: `sumEvenFibonacci`
Purpose: Computes the **sum of even-valued terms** in the Fibonacci sequence up to a given number of terms.

Features

- Uses a loop to generate Fibonacci numbers and sums the even-valued ones.

Inputs and Outputs

Inputs:

- `n`: An integer representing the number of terms in the Fibonacci sequence.

Outputs:

- Returns an integer representing the sum of even-valued Fibonacci terms.

Workflow

1. Initialize a variable `S` to 0.
2. Use a `for` loop to generate Fibonacci numbers up to `n` terms.
3. Check if the current Fibonacci number is even.
4. If even, add it to `S`.
5. Return the sum `S`.

Strengths and Limitations

Strengths:

- Simple and effective for small values of `n`.

Limitations:

- May cause overflow for large values of `n`.

Code Summary

- **Language:** C
- **Key Tools:** Loops, addition, modulo operator (`%`).

---

25. Function: `isHarshad`

Title and Overview

Function Name: `isHarshad`
Purpose: Checks if a given integer is a **Harshad number** (a number divisible by the sum of its digits).

Features

- Uses the `sumOfDigits` function to compute the sum of digits.

Inputs and Outputs

Inputs:

- `num`: An integer to be checked.

Outputs:

- Returns a boolean value (`true` if the number is a Harshad number, `false` otherwise).

Workflow

1. Call the `sumOfDigits` function to compute the sum of digits of `num`.
2. Check if `num` is divisible by the sum of its digits.
3. Return `true` if divisible, otherwise return `false`.

Strengths and Limitations

Strengths:

- Reuses the `sumOfDigits` function, promoting code reuse.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Helper function (`sumOfDigits`), modulo operator (`%`).

---

26. Function: `catalanNumber`

Title and Overview

Function Name: `catalanNumber`
Purpose: Computes the **nth Catalan number**.

Features

- Uses a loop to compute the Catalan number iteratively.

Inputs and Outputs

Inputs:

- `n`: An integer representing the index of the Catalan number.

Outputs:

- Returns an unsigned long representing the nth Catalan number.

Workflow

1. Handle base cases (n = 0 or n = 1, return 1).
2. Initialize a variable `catalan` to 1.
3. Use a `for` loop to compute the Catalan number using the formula:
    - `catalan = catalan * (2 * n - i) / (i + 1)`.
4. Return the computed Catalan number divided by `(n + 1)`.

Strengths and Limitations

Strengths:

- Efficient for small values of `n`.

Limitations:

- May cause overflow for large values of `n`.

Code Summary

- **Language:** C
- **Key Tools:** Loops, multiplication, division.

---

27. Function: `pascalTriangle`

Title and Overview

Function Name: `pascalTriangle`
Purpose: Prints **Pascal's triangle** up to a given number of rows.

Features

- Uses nested loops to compute and print the triangle.

Inputs and Outputs

Inputs:

- `n`: An integer representing the number of rows in Pascal's triangle.

Outputs:

- Prints Pascal's triangle to the console.

Workflow

1. Use a `for` loop to iterate through each row.
2. Use another `for` loop to print spaces for alignment.
3. Use a third `for` loop to compute and print the binomial coefficients for the current row.
4. Repeat until all rows are printed.

Strengths and Limitations

Strengths:

- Simple and effective for small values of `n`.

Limitations:

- May cause overflow for large values of `n`.

Code Summary

- **Language:** C
- **Key Tools:** Nested loops, binomial coefficient calculation.

---

28. Function: `binomialCofficient`

Title and Overview

Function Name: `binomialCofficient`
Purpose: Computes the **binomial coefficient** C(n, p).

Features

- Uses the `factorial` function to compute the binomial coefficient.

Inputs and Outputs

Inputs:

- `n`: Total number of items.
- `p`: Number of items to choose.

Outputs:

- Returns an unsigned long representing the binomial coefficient.

Workflow

1. Compute the factorial of `n`, `p`, and `(n - p)` using the `factorial` function.
2. Use the formula: `C(n, p) = factorial(n) / (factorial(p) * factorial(n - p))`.
3. Return the computed binomial coefficient.

Strengths and Limitations

Strengths:

- Reuses the `factorial` function, promoting code reuse.

Limitations:

- May cause overflow for large values of `n` or `p`.

Code Summary

- **Language:** C
- **Key Tools:** Helper function (`factorial`), mathematical formula.

---

29. Function: `bellNumber`

Title and Overview

Function Name: `bellNumber`
Purpose: Computes the **nth Bell number**.

Features

- Uses recursion and the `binomialCofficient` function to compute the Bell number.

Inputs and Outputs

Inputs:

- `n`: An integer representing the index of the Bell number.

Outputs:

- Returns an unsigned long long representing the nth Bell number.

Workflow

1. Handle the base case (n = 0, return 1).
2. Use a `for` loop to compute the Bell number using the formula:
    - `bell += binomialCofficient(n - 1, i) * bellNumber(i)`.

3. Return the computed Bell number.

Strengths and Limitations

Strengths:

- Accurate and follows the mathematical definition.

Limitations:

- Inefficient due to recursion and repeated calculations.

Code Summary

- **Language:** C
- **Key Tools:** Recursion, helper function (`binomialCofficient`).

---

30. Function: `isKaprekar`

Title and Overview

Function Name: `isKaprekar`
Purpose: Checks if a given integer is a **Kaprekar number** (a number whose square can be split into two parts that add up to the original number).

Features

- Computes the square of the number and checks the splitting condition.

Inputs and Outputs

Inputs:

- `num`: An integer to be checked.

Outputs:

- Returns a boolean value (`true` if the number is a Kaprekar number, `false` otherwise).

Workflow

1. Compute the square of `num`.
2. Count the number of digits in `num`.
3. Split the square into two parts based on the number of digits.
4. Check if the sum of the two parts equals `num`.
5. Return `true` if they are equal, otherwise return `false`.

Strengths and Limitations

Strengths:

- Accurate and straightforward.

Limitations:

- Inefficient for very large numbers.

Code Summary

- **Language:** C
- **Key Tools:** Loops, power calculation, modulo operator (`%`).

---

31. Function: `isSmith`

Title and Overview

Function Name: `isSmith`
Purpose: Checks if a given integer is a **Smith number** (a composite number whose sum of digits equals the sum of the digits of its prime factors).

Features

- Uses the `sumOfDigits` function to compute the sum of digits.
- Computes the sum of the digits of the prime factors.

Inputs and Outputs

Inputs:

- `num`: An integer to be checked.

Outputs:

- Returns a boolean value (`true` if the number is a Smith number, `false` otherwise).

Workflow

1. Check if `num` is prime (return `false` if prime).
2. Compute the sum of the digits of `num` using the `sumOfDigits` function.
3. Compute the sum of the digits of the prime factors of `num`.
4. Compare the two sums.
5. Return `true` if they are equal, otherwise return `false`.

Strengths and Limitations

Strengths:

- Accurate and follows the mathematical definition.

Limitations:

- Inefficient for very large numbers.

Code Summary

- **Language:** C
- **Key Tools:** Helper function (`sumOfDigits`), prime factorization.

---

32. Function: `sumOfPrimes`

Title and Overview

Function Name: `sumOfPrimes`
Purpose: Computes the **sum of all prime numbers** up to a given integer.

Features

- Uses the `isPrime` function to check for prime numbers.

Inputs and Outputs

Inputs:

- `n`: An integer representing the upper limit.

Outputs:

- Returns an integer representing the sum of prime numbers up to `n`.

Workflow

1. Initialize a variable `S` to 0.
2. Use a `for` loop to iterate through integers from 1 to `n`.
3. Check if the current integer is prime using the `isPrime` function.
4. If prime, add it to `S`.
5. Return the sum `S`.

Strengths and Limitations

Strengths:

- Reuses the `isPrime` function, promoting code reuse.

Limitations:

- Inefficient for very large values of `n`.

Code Summary

- **Language:** C
- **Key Tools:** Helper function (`isPrime`), loops.

Arrays

1. Function: `initializeArray`

Title and Overview

Function Name: `initializeArray`
Purpose: Initializes all elements of an array with a specified value.

Features

- Iterates through the array and assigns the specified value to each element.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be initialized.
- `size`: The size of the array.
- `value`: The value to initialize the array with.

Outputs:

- Modifies the array to contain the specified value in all elements.

Workflow

1. Use a `for` loop to iterate through each element of the array.
2. Assign the specified value to each element.
3. Continue until all elements are initialized.

Strengths and Limitations

Strengths:

- Simple and efficient for initializing arrays.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Loops, array indexing.

---

2. Function: `printArray`

Title and Overview

Function Name: `printArray`
Purpose: Prints all elements of an array.

Features

- Iterates through the array and prints each element.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be printed.
- `size`: The size of the array.

Outputs:

- Prints the elements of the array to the console.

Workflow

1. Use a `for` loop to iterate through each element of the array.
2. Print each element followed by a space.
3. Print a newline character after all elements are printed.

Strengths and Limitations

Strengths:

- Simple and effective for printing arrays.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Loops, array indexing, `printf`.

---

3. Function: `findMax`

Title and Overview

Function Name: `findMax`
Purpose: Finds the maximum element in an array.

Features

- Iterates through the array and tracks the maximum value.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be searched.
- `size`: The size of the array.

Outputs:

- Returns the maximum element in the array.

Workflow

1. Initialize a variable `max` with the first element of the array.
2. Use a `for` loop to iterate through the remaining elements.
3. Update `max` if a larger element is found.

4. Return the value of `max`.

Strengths and Limitations

Strengths:

- Simple and efficient for finding the maximum element.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Loops, conditional checks.

---

4. Function: `findMin`

Title and Overview

Function Name: `findMin`
Purpose: Finds the minimum element in an array.

Features

- Iterates through the array and tracks the minimum value.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be searched.
- `size`: The size of the array.

Outputs:

- Returns the minimum element in the array.

Workflow

1. Initialize a variable `min` with the first element of the array.
2. Use a `for` loop to iterate through the remaining elements.
3. Update `min` if a smaller element is found.
4. Return the value of `min`.

Strengths and Limitations

Strengths:

- Simple and efficient for finding the minimum element.

Limitations:

- None.

Code Summary

- **Language:** C

- **Key Tools:** Loops, conditional checks.

---

5. Function: `sumArray`

Title and Overview

Function Name: `sumArray`
Purpose: Calculates the sum of all elements in an array.

Features

- Iterates through the array and accumulates the sum of its elements.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be summed.
- `size`: The size of the array.

Outputs:

- Returns the sum of all elements in the array.

Workflow

1. Initialize a variable `sum` to 0.
2. Use a `for` loop to iterate through each element of the array.
3. Add each element to `sum`.
4. Return the value of `sum`.

Strengths and Limitations

Strengths:

- Simple and efficient for calculating the sum of array elements.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Loops, arithmetic operations.

---

6. Function: `averageArray`

Title and Overview

Function Name: `averageArray`
Purpose: Calculates the average of all elements in an array.

Features

- Uses the `sumArray` function to calculate the sum and divides it by the size of the array.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be averaged.
- `size`: The size of the array.

Outputs:

- Returns the average of all elements in the array as a double.

Workflow

1. Call the `sumArray` function to calculate the sum of the array.
2. Divide the sum by the size of the array.
3. Return the result as a double.

Strengths and Limitations

Strengths:

- Reuses the `sumArray` function, promoting code reuse.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Helper function (`sumArray`), arithmetic operations.

---

7. Function: `isSorted`

Title and Overview

Function Name: `isSorted`
Purpose: Checks if an array is sorted in ascending order.

Features

- Iterates through the array and checks if each element is less than or equal to the next.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be checked.
- `size`: The size of the array.

Outputs:

- Returns a boolean value (`true` if the array is sorted, `false` otherwise).

Workflow

1. Use a `for` loop to iterate through the array.
2. Check if each element is less than or equal to the next element.
3. Return `false` if any element is greater than the next.
4. Return `true` if the loop completes without finding any out-of-order elements.

Strengths and Limitations

Strengths:

- Simple and efficient for checking if an array is sorted.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Loops, conditional checks.

---

8. Function: `reverseArray`

Title and Overview

Function Name: `reverseArray`
Purpose: Reverses the elements of an array in place.

Features

- Swaps elements from the beginning and end of the array until the middle is reached.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be reversed.
- `size`: The size of the array.

Outputs:

- Modifies the array to contain the reversed elements.

Workflow

1. Use a `for` loop to iterate through the first half of the array.
2. Swap each element with its corresponding element from the end of the array.
3. Continue until the middle of the array is reached.

Strengths and Limitations

Strengths:

- Simple and efficient for reversing arrays.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Loops, swapping.

---

9. Function: `countEvenOdd`

Title and Overview

Function Name: `countEvenOdd`
Purpose: Counts the number of even and odd elements in an array.

Features

- Iterates through the array and counts even and odd elements.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be analyzed.
- `size`: The size of the array.
- `evenCount`: A pointer to store the count of even elements.
- `oddCount`: A pointer to store the count of odd elements.

Outputs:

- Modifies `evenCount` and `oddCount` to contain the respective counts.

Workflow

1. Initialize `evenCount` and `oddCount` to 0.
2. Use a `for` loop to iterate through each element of the array.
3. Increment `evenCount` if the element is even, otherwise increment `oddCount`.
4. Continue until all elements are processed.

Strengths and Limitations

Strengths:

- Simple and effective for counting even and odd elements.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Loops, modulo operator (`%`).

---

10. Function: `secondLargest`

Title and Overview

Function Name: `secondLargest`
Purpose: Finds the second largest element in an array.

Features

- Iterates through the array and tracks the two largest elements.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be searched.
- `size`: The size of the array.

Outputs:

- Returns the second largest element in the array.

Workflow

1. Initialize two variables, `first` and `second`, to the smallest possible integer value.
2. Use a `for` loop to iterate through each element of the array.
3. Update `first` and `second` based on the current element.
4. Return the value of `second`.

Strengths and Limitations

Strengths:

- Efficient for finding the second largest element.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Loops, conditional checks.

---

11. Function: `elementFrequency`

Title and Overview

Function Name: `elementFrequency`
Purpose: Finds the frequency of each unique element in an array.

Features

- Uses a frequency array to count occurrences of each element.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be analyzed.
- `size`: The size of the array.

Outputs:

- Prints the frequency of each unique element in the array.

Workflow

1. Initialize a frequency array to track counts of each element.
2. Use nested loops to count occurrences of each element.
3. Print the frequency of each unique element.

Strengths and Limitations

Strengths:

- Simple and effective for calculating element frequency.

Limitations:

- Inefficient for large arrays due to nested loops.

Code Summary

- **Language:** C
- **Key Tools:** Nested loops, frequency array.

---

12. Function: `removeDuplicates`

Title and Overview

Function Name: `removeDuplicates`
Purpose: Removes duplicate elements from a sorted array and returns the new size.

Features

- Iterates through the array and removes duplicates in place.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be processed.
- `size`: The size of the array.

Outputs:

- Returns the new size of the array after removing duplicates.

Workflow

1. Initialize a variable `uniqueIndex` to 0.
2. Use a `for` loop to iterate through the array.
3. If the current element is not equal to the next element, copy it to the `uniqueIndex` position.
4. Increment `uniqueIndex`.

5. Return the value of `uniqueIndex`.

Strengths and Limitations

Strengths:

- Efficient for removing duplicates in a sorted array.

Limitations:

- Requires the array to be sorted.

Code Summary

- **Language:** C
- **Key Tools:** Loops, conditional checks.

---

13. Function: `binarySearch`

Title and Overview

Function Name: `binarySearch`
Purpose: Performs binary search on a sorted array to find a target element.

Features

- Uses a divide-and-conquer approach to search for the target element.

Inputs and Outputs

Inputs:

- `arr[]`: The sorted array to be searched.
- `size`: The size of the array.
- `target`: The element to be searched for.

Outputs:

- Returns the index of the target element if found, otherwise returns `-1`.

Workflow

1. Initialize `left` to 0 and `right` to `size - 1`.
2. Use a `while` loop to divide the array into halves.
3. Compare the middle element with the target.
4. Adjust the search range based on the comparison.
5. Return the index if the target is found, otherwise return `-1`.

Strengths and Limitations

Strengths:

- Efficient for searching in sorted arrays.

Limitations:

- Requires the array to be sorted.

Code Summary

- **Language:** C
- **Key Tools:** Divide-and-conquer, loops, conditional checks.

---

14. Function: `linearSearch`

Title and Overview

Function Name: `linearSearch`
Purpose: Performs linear search to find a target element in an array.

Features

- Iterates through the array and checks each element for a match.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be searched.
- `size`: The size of the array.
- `target`: The element to be searched for.

Outputs:

- Returns the index of the target element if found, otherwise returns `-1`.

Workflow

1. Use a `for` loop to iterate through each element of the array.
2. Check if the current element matches the target.
3. Return the index if a match is found.
4. Return `-1` if the loop completes without finding a match.

Strengths and Limitations

Strengths:

- Simple and effective for searching in unsorted arrays.

Limitations:

- Inefficient for large arrays.

Code Summary

- **Language:** C
- **Key Tools:** Loops, conditional checks.

---

15. Function: `leftShift`

Title and Overview

Function Name: `leftShift`
Purpose: Shifts the elements of an array to the left by a specified number of positions.

Features

- Rotates the array elements to the left.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be shifted.
- `size`: The size of the array.
- `rotations`: The number of positions to shift.

Outputs:

- Modifies the array to contain the shifted elements.

Workflow

1. Use a `for` loop to perform the specified number of rotations.
2. In each rotation, store the first element in a temporary variable.
3. Shift all elements to the left by one position.
4. Place the temporary variable in the last position.
5. Repeat until all rotations are completed.

Strengths and Limitations

Strengths:

- Simple and effective for left-shifting arrays.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Loops, swapping.

---

16. Function: `rightShift`

Title and Overview

Function Name: `rightShift`
Purpose: Shifts the elements of an array to the right by a specified number of positions.

Features

- Rotates the array elements to the right.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be shifted.
- `size`: The size of the array.
- `rotations`: The number of positions to shift.

Outputs:

- Modifies the array to contain the shifted elements.

Workflow

1. Use a `for` loop to perform the specified number of rotations.
2. In each rotation, store the last element in a temporary variable.
3. Shift all elements to the right by one position.
4. Place the temporary variable in the first position.
5. Repeat until all rotations are completed.

Strengths and Limitations

Strengths:

- Simple and effective for right-shifting arrays.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Loops, swapping.

---

17. Function: `bubbleSort`

Title and Overview

Function Name: `bubbleSort`
Purpose: Sorts an array using the bubble sort algorithm.

Features

- Compares adjacent elements and swaps them if they are in the wrong order.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be sorted.
- `size`: The size of the array.

Outputs:

- Modifies the array to contain the sorted elements.

Workflow

1. Use nested `for` loops to iterate through the array.
2. Compare adjacent elements and swap them if they are in the wrong order.
3. Repeat until the array is sorted.

Strengths and Limitations

Strengths:

- Simple and easy to implement.

Limitations:

- Inefficient for large arrays.

Code Summary

- **Language:** C
- **Key Tools:** Nested loops, swapping.

---

18. Function: `selectionSort`

Title and Overview

Function Name: `selectionSort`
Purpose: Sorts an array using the selection sort algorithm.

Features

- Selects the smallest element and swaps it with the first unsorted element.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be sorted.
- `size`: The size of the array.

Outputs:

- Modifies the array to contain the sorted elements.

Workflow

1. Use nested `for` loops to iterate through the array.
2. Find the smallest element in the unsorted portion of the array.
3. Swap it with the first unsorted element.
4. Repeat until the array is sorted.

Strengths and Limitations

Strengths:

- Simple and easy to implement.

Limitations:

- Inefficient for large arrays.

Code Summary

- **Language:** C
- **Key Tools:** Nested loops, swapping.

---

19. Function: `insertionSort`

Title and Overview

Function Name: `insertionSort`
Purpose: Sorts an array using the insertion sort algorithm.

Features

- Builds the sorted array one element at a time by inserting each element into its correct position.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be sorted.
- `size`: The size of the array.

Outputs:

- Modifies the array to contain the sorted elements.

Workflow

1. Use a `for` loop to iterate through the array.
2. For each element, insert it into its correct position in the sorted portion of the array.
3. Repeat until the array is sorted.

Strengths and Limitations

Strengths:

- Efficient for small arrays or nearly sorted arrays.

Limitations:

- Inefficient for large arrays.

Code Summary

- **Language:** C
- **Key Tools:** Loops, conditional checks.

---

20. Function: `mergeSort`

Title and Overview

Function Name: `mergeSort`
Purpose: Sorts an array using the merge sort algorithm.

Features

- Divides the array into two halves, sorts them recursively, and merges them.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be sorted.
- `left`: The starting index of the array.
- `right`: The ending index of the array.

Outputs:

- Modifies the array to contain the sorted elements.

Workflow

1. Divide the array into two halves.
2. Recursively sort each half.
3. Merge the two sorted halves.

Strengths and Limitations

Strengths:

- Efficient for large arrays.

Limitations:

- Requires additional memory for merging.

Code Summary

- **Language:** C
- **Key Tools:** Recursion, merging.

---

21. Function: `quickSort`

Title and Overview

Function Name: `quickSort`
Purpose: Sorts an array using the quick sort algorithm.

Features

- Partitions the array around a pivot and recursively sorts the partitions.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be sorted.
- `low`: The starting index of the array.

- `high`: The ending index of the array.

Outputs:

- Modifies the array to contain the sorted elements.

Workflow

1. Choose a pivot element.
2. Partition the array such that elements less than the pivot are on the left and elements greater than the pivot are on the right.
3. Recursively sort the partitions.

Strengths and Limitations

Strengths:

- Efficient for large arrays.

Limitations:

- Performance depends on the choice of pivot.

Code Summary

- **Language:** C
- **Key Tools:** Recursion, partitioning.

---

22. Function: `findMissingNumber`

Title and Overview

Function Name: `findMissingNumber`
Purpose: Finds the missing number in an array of size `n-1` containing numbers from 1 to `n`.

Features

- Uses the sum of the first `n` natural numbers to find the missing number.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be analyzed.
- `size`: The size of the array.

Outputs:

- Returns the missing number.

Workflow

1. Calculate the sum of the first `n` natural numbers.
2. Subtract the sum of the array elements from this value.
3. Return the result as the missing number.

Strengths and Limitations

Strengths:

- Simple and efficient for finding the missing number.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Arithmetic operations.

---

23. Function: `findPairsWithSum`

Title and Overview

Function Name: `findPairsWithSum`
Purpose: Finds all pairs of elements in an array whose sum is equal to a given value.

Features

- Uses nested loops to find pairs with the specified sum.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be analyzed.
- `size`: The size of the array.
- `sum`: The target sum.

Outputs:

- Prints all pairs of elements whose sum equals the target value.

Workflow

1. Use nested `for` loops to iterate through the array.
2. Check if the sum of the current pair equals the target sum.
3. Print the pair if a match is found.

Strengths and Limitations

Strengths:

- Simple and effective for finding pairs with a given sum.

Limitations:

- Inefficient for large arrays due to nested loops.

Code Summary

- **Language:** C
- **Key Tools:** Nested loops, conditional checks.

---

24. Function: `findSubArrayWithSum`

Title and Overview

Function Name: `findSubArrayWithSum`
Purpose: Finds a continuous subarray whose elements add up to a given sum.

Features

- Uses nested loops to find subarrays with the specified sum.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be analyzed.
- `size`: The size of the array.
- `sum`: The target sum.

Outputs:

- Prints the indices of the subarray if found.

Workflow

1. Use nested `for` loops to iterate through the array.
2. Calculate the sum of the current subarray.
3. If the sum matches the target, print the indices and return.
4. If no subarray is found, print a message.

Strengths and Limitations

Strengths:

- Simple and effective for finding subarrays with a given sum.

Limitations:

- Inefficient for large arrays due to nested loops.

Code Summary

- **Language:** C
- **Key Tools:** Nested loops, arithmetic operations.

---

25. Function: `rearrangeAlternatePositiveNegative`

Title and Overview

Function Name: `rearrangeAlternatePositiveNegative`
Purpose: Rearranges the array such that positive and negative numbers alternate.

Features

- Separates positive and negative numbers and rearranges them alternately.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be rearranged.
- `size`: The size of the array.

Outputs:

- Modifies the array to contain alternating positive and negative numbers.

Workflow

1. Separate positive and negative numbers.
2. Rearrange the array to alternate positive and negative numbers.
3. Handle cases where there are more positive or negative numbers.

Strengths and Limitations

Strengths:

- Simple and effective for rearranging arrays.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Loops, swapping.

---

26. Function: `findMajorityElement`

Title and Overview

Function Name: `findMajorityElement`
Purpose: Finds the majority element in an array (an element that appears more than `n/2` times).

Features

- Uses the Boyer-Moore voting algorithm to find the majority element.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be analyzed.
- `size`: The size of the array.

Outputs:

- Returns the majority element if found, otherwise returns `-1`.

Workflow

1. Initialize a candidate and a count.
2. Iterate through the array and update the candidate and count.

3. Verify if the candidate is the majority element.
4. Return the candidate if it is the majority element, otherwise return `-1`.

Strengths and Limitations

Strengths:

- Efficient for finding the majority element.

Limitations:

- None.

Code Summary

- **Language:** C
- **Key Tools:** Loops, conditional checks.

---

27. Function: `longestIncreasingSubsequence`

Title and Overview

Function Name: `longestIncreasingSubsequence`
Purpose: Finds the length of the longest increasing subsequence in an array.

Features

- Uses dynamic programming to track the length of the longest increasing subsequence.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be analyzed.
- `size`: The size of the array.

Outputs:

- Returns the length of the longest increasing subsequence.

Workflow

1. Initialize a dynamic programming array to track the length of the longest increasing subsequence.
2. Use nested loops to update the dynamic programming array.
3. Return the maximum value in the dynamic programming array.

Strengths and Limitations

Strengths:

- Efficient for finding the longest increasing subsequence.

Limitations:

- Requires additional memory for the dynamic programming array.

Code Summary

- **Language:** C
- **Key Tools:** Dynamic programming, nested loops.

---

28. Function: `findDuplicates`

Title and Overview

Function Name: `findDuplicates`
Purpose: Identifies duplicate elements in an array.

Features

- Uses nested loops to find duplicate elements.

Inputs and Outputs

Inputs:

- `arr[]`: The array to be analyzed.
- `size`: The size of the array.

Outputs:

- Prints duplicate elements in the array.

Workflow

1. Use nested `for` loops to iterate through the array.
2. Check if the current element matches any other element.
3. Print the duplicate element if found.

Strengths and Limitations

Strengths:

- Simple and effective for finding duplicates.

Limitations:

- Inefficient for large arrays due to nested loops.

Code Summary

- **Language:** C
- **Key Tools:** Nested loops, conditional checks.

---

29. Function: `findIntersection`

Title and Overview

Function Name: `findIntersection`
Purpose: Finds the common elements between two arrays.

Features

- Uses nested loops to find common elements.

Inputs and Outputs

Inputs:

- `arr1[]`: The first array.
- `size1`: The size of the first array.
- `arr2[]`: The second array.
- `size2`: The size of the second array.

Outputs:

- Prints the common elements between the two arrays.

Workflow

1. Use nested `for` loops to iterate through both arrays.
2. Check if the current element of the first array matches any element in the second array.
3. Print the common element if found.

Strengths and Limitations

Strengths:

- Simple and effective for finding common elements.

Limitations:

- Inefficient for large arrays due to nested loops.

Code Summary

- **Language:** C
- **Key Tools:** Nested loops, conditional checks.

---

30. Function: `findUnion`

Title and Overview

Function Name: `findUnion`
Purpose: Finds the union of two arrays.

Features

- Combines elements from both arrays and removes duplicates.

Inputs and Outputs

Inputs:

- `arr1[]`: The first array.
- `size1`: The size of the first array.
- `arr2[]`: The second array.

- `size2`: The size of the second array.

Outputs:

- Prints the union of the two arrays.

Workflow

1. Combine elements from both arrays into a new array.
2. Remove duplicates from the new array.
3. Print the elements of the new array.

Strengths and Limitations

Strengths:

- Simple and effective for finding the union of two arrays.

Limitations:

- Requires additional memory for the new array.

Code Summary

- **Language:** C
- **Key Tools:** Loops, conditional checks.