NATIONAL SCHOOL OF CYBER SECURITY

MODULE: Algorithms and static data structure

DATE: 19/01/2025

BY: LAIFAOUI MARYA TASNIM

NOURREDDINE MALAK

End-semester project REPORT:

Introduction:

Numerical computations are essential in programming, forming the basis for solving diverse computational problems, from simple arithmetic to advanced algorithms like prime factorization and Fibonacci sequences. This project aims to create a C library that implements a wide range of numerical functions, categorized into basic, intermediate, and advanced levels.

The library not only provides solutions to specific problems but also emphasizes modular design, code reuse, and algorithmic thinking. Functions such as palindrome checks, greatest common divisor (GCD), and binary conversion allow students to deepen their understanding of mathematical algorithms and programming techniques.

By combining theoretical concepts with hands-on implementation, this project fosters practical skills in problem-solving, algorithm design, and modular programming, offering a valuable resource for both academic learning and real-world applications.

Objective:

The objective of this project is to enhance our skills in algorithmic thinking and C programming by creating a well-structured library. Through this project, we aim to design modular and reusable code while solving computational problems such as numerical operations, sorting, searching, and operations on arrays. This practical implementation will provide valuable experience in applying key programming concepts effectively

---

Problem Statement:

In programming, solving computational problems efficiently often requires reusable and modular code. This project addresses the need for a comprehensive C library that implements essential operations, such as numerical computations, sorting, and searching. By developing this library, we aim to simplify the process of reusing code and ensure that these functions are implemented in a way that promotes modularity and clarity.

Operations on numbers:

1/basic:

0. int SumOfDigits(int num):

Definition:
This function calculates the sum of the digits of a given integer num.

Steps:

1. **Initialize s = 0 (this will store the sum of digits).**
2. **While num≠0**

- **Extract the last digit of num using num%10.**
- **Add the digit to s.**
- **Remove the last digit from num by dividing it by 10.**

3. **Return the sum s.**

Example:
If num=1234

- **Step 1: Initialize s=0.**
- **Step 2: Start iterations:**
  - **1234%10=4 , s=0+4=4 , num=123**
  - **123%10=3 , s = 4 + 3 = 7, num=12**
  - **12 \% 10 = 2, s=7+2=9, num=1**
  - **1%10=1 , s=9+1=10, num=0.**

Output: The function returns 10.

---

1. int reverseNumber(int num):

- **Definition**: This function reverses the digits of a given integer num.

Steps:

1. Initialize reverse = 0.
2. While num is not zero:
   - Get the last digit using num % 10.
   - Add this digit to reverse * 10.
   - Remove the last digit from num by dividing it by 10.
3. Return reverse.

Example:

If num = 1234:

- Step 1: reverse = 0
- Step 2: 1234 % 10 = 4, add 4 → reverse = 0 * 10 + 4 = 4, num = 123
- Step 3: 123 % 10 = 3, add 3 → reverse = 4 * 10 + 3 = 43, num = 12
- Step 4: 12 % 10 = 2, add 2 → reverse = 43 * 10 + 2 = 432, num = 1
- Step 5: 1 % 10 = 1, add 1 → reverse = 432 * 10 + 1 = 4321, num = 0
- The function returns 4321.

---

2. bool isPalindrome(int num):

- **Definition**: This function checks if the given integer num is a palindrome (reads the same forwards and backwards).

Steps:

1. Reverse the number using the reverseNumber() function.
2. Compare the reversed number with num.
3. Return true if they are equal, otherwise return false.

Example:

If num = 121:

- Step 1: Reverse 121 → reverse = 121.
- Step 2: Compare 121 with num. They are equal, so return true.

If num = 123:

- Step 1: Reverse 123 → reverse = 321.
- Step 2: Compare 321 with num. They are not equal, so return false.

---

3. bool isPrime(int num):

- **Definition**: This function checks if a given integer num is a prime number (greater than 1 and not divisible by any other number except 1 and itself).

Steps:

1. If num <= 1, return false (since numbers less than 2 are not prime).
2. Loop through numbers from 2 to sqrt(num):
    - If num % i == 0, return false (num is divisible by i).
3. If no divisors are found, return true (num is prime).

Example:

If num = 5:

- Step 1: num > 1, continue.
- Step 2: Check divisibility by 2. Since 5 % 2 != 0, continue.
- Step 3: No divisors are found, so return true.

If num = 4:

- Step 1: num > 1, continue.
- Step 2: Check divisibility by 2. Since 4 % 2 == 0, return false.

---

4. int gcd(int a, int b):

- **Definition**: This function calculates the greatest common divisor (GCD) of two integers a and b using the Euclidean algorithm.

Steps:

1. While b != 0:
    - Set temp = b.
    - Set b = a % b.

- Set a = temp.

2. Return a as the GCD.

Example:

If a = 36 and b = 60:

- Step 1: b != 0, so temp = 60, b = 36 % 60 = 36, a = 60.
- Step 2: b != 0, so temp = 36, b = 60 % 36 = 24, a = 36.
- Step 3: b != 0, so temp = 24, b = 36 % 24 = 12, a = 24.
- Step 4: b != 0, so temp = 12, b = 24 % 12 = 0, a = 12.
- The function returns 12 (GCD of 36 and 60).

---

5. int lcm(int a, int b):

- **Definition**: This function calculates the least common multiple (LCM) of two integers a and b.

Steps:

1. Use the formula LCM = (a * b) / GCD(a, b).

Example:

If a = 36 and b = 60:

- Calculate GCD(36, 60) = 12.
- Use the formula LCM = (36 * 60) / 12 = 180.

---

6. long factorial(int num):

- **Definition**: This function calculates the factorial of a non-negative integer num (i.e., num! = num * (num - 1) * ... * 1).

Steps:

1. If num < 0, return -1 (error condition).
2. If num == 0 or num == 1, return 1.
3. Otherwise, initialize result = 1 and loop from 2 to num:
    - Multiply result by i in each iteration.
4. Return result.

Example:

If num = 5:

- Step 1: Initialize result = 1.
- Step 2: Multiply result = 1 * 2 = 2.
- Step 3: Multiply result = 2 * 3 = 6.
- Step 4: Multiply result = 6 * 4 = 24.
- Step 5: Multiply result = 24 * 5 = 120.

- The function returns 120 (5!).

---

7. bool isEven(int num):

- **Definition**: This function checks if a given integer num is even.

Steps:

1. If num % 2 == 0, return true.
2. Otherwise, return false.

Example:

If num = 4, return true (since 4 % 2 == 0).

If num = 7, return false (since 7 % 2 != 0).

2/intermediate:

8. void primeFactors(int num):

- **Definition**: This function prints all prime factors of a given number num.

Steps:

1. Loop from 2 to sqrt(num):
    - If i is prime and divides num, print i.
    - Divide num by i.
2. If num > 1, print num.

Example:

If num = 36:

- Step 1: Print 2 (since 36 % 2 == 0).
- Step 2: Print 2 again (since 36 / 2 = 18).
- Step 3: Print 2 again (since 18 / 2 = 9).
- Step 4: Print 3 (since 9 % 3 == 0).
- The function prints 2, 2, 2, 3.

---

9. bool isArmstrong(int num):

- **Definition**: This function checks if the given number num is an Armstrong number (a number that is equal to the sum of its digits raised to the power of 3).

Steps:

1. Initialize s = 0.
2. While num is not zero:
    - Add the cube of the last digit to s.
    - Remove the last digit from num.

3. If s == num, return true. Otherwise, return false.

Example:

If num = 153:

- Step 1: s = 0.
- Step 2: 153 % 10 = 3, add 3^3 = 27 to s, s = 27, num = 15.
- Step 3: 15 % 10 = 5, add 5^3 = 125 to s, s = 152, num = 1.
- Step 4: 1 % 10 = 1, add 1^3 = 1 to s, s = 153, num = 0.
- Since s == num, return true.

---

10. void fibonacciSeries(int n):

- **Definition**: This function prints the Fibonacci series up to the nth term.

Steps:

1. Initialize a = 0 and b = 1.
2. Print the first two terms (a and b).
3. Loop from 3 to n:
    - Calculate j = a + b.
    - Print j.
    - Update a = b and b = j.

Example:

If n = 5:

- Step 1: Print 0 and 1.
- Step 2: For i = 3: j = 0 + 1 = 1, print 1, update a = 1, b = 1.
- Step 3: For i = 4: j = 1 + 1 = 2, print 2, update a = 1, b = 2.
- Step 4: For i = 5: j = 1 + 2 = 3, print 3, update a = 2, b = 3.
- The function prints 0 1 1 2 3.

---

11. int sumDevisors(int num):

- **Definition**: This function calculates the sum of all divisors of a given integer num.

Steps:

1. Initialize s = 0.
2. Loop from 1 to num - 1:
    - If num % i == 0, add i to s.
3. Return s.

Example:

If num = 6:

- Step 1: Initialize s = 0.
- Step 2: Check divisors 1, 2, 3. Sum = 1 + 2 + 3 = 6.
- The function returns 6.

---

12. bool isPerfect(int num):

- **Definition**: This function checks if the number num is a perfect number (i.e., the sum of its divisors equals the number itself).

Steps:

1. Call sumDevisors(num) and check if the result equals num.
2. If sumDevisors(num) == num, return true. Otherwise, return false.

Example:

If num = 6:

- Step 1: sumDevisors(6) = 6.
- Step 2: Since sumDevisors(6) == 6, return true.

If num = 8:

- Step 1: sumDevisors(8) = 1 + 2 + 4 = 7.
- Step 2: Since sumDevisors(8) != 8, return false.

---

13. bool isMagic(int num):

- **Definition**: This function checks if the number num is a magic number (where the sum of the digits of the number, and then the sum of the digits of that sum, equals 1).

Steps:

1. Call SumOfDigits(SumOfDigits(num)).
2. If the result equals 1, return true. Otherwise, return false.

Example:

If num = 19:

- Step 1: SumOfDigits(19) = 1 + 9 = 10.
- Step 2: SumOfDigits(10) = 1 + 0 = 1.
- Since SumOfDigits(10) == 1, return true.

---

14. bool isAutomorphic(int num):

- **Definition**: This function checks if a number num is an automorphic number (the number's square ends with the number itself).

Steps:

1. Calculate the number of digits in num.
2. Compute the square of num.
3. Extract the last digits of the square equal to the number of digits in num.
4. Compare the last digits of the square with num.

Example:

If num = 5:

- Step 1: Number of digits in 5 is 1.
- Step 2: Square of 5 is 25.
- Step 3: The last 1 digit of 25 is 5, which matches num.
- The function returns true.

If num = 6:

- Step 1: Number of digits in 6 is 1.
- Step 2: Square of 6 is 36.
- Step 3: The last 1 digit of 36 is 6, which matches num.
- The function returns true.

3/advanced:

15. void toBinary(int num):

- **Definition**: This function recursively prints the binary representation of the integer num.

Steps:

1. If num == 0, stop the recursion.
2. Recursively call toBinary(num / 2).
3. Print the remainder when num is divided by 2 (num % 2).

Example:

If num = 6:

- Step 1: Call toBinary(6 / 2) = toBinary(3).
- Step 2: Call toBinary(3 / 2) = toBinary(1).
- Step 3: Call toBinary(1 / 2) = toBinary(0).
- Step 4: Print 1 (from 1 % 2), then 0 (from 3 % 2), and then 0 (from 6 % 2).
- The function prints 110.

---

16. bool isNarcissitic(int num):

- **Definition**: This function checks if the number num is a narcissistic number (i.e., the sum of each digit raised to the power of the number of digits in the number equals the number itself).

Steps:

1. Count the number of digits in num.
2. For each digit in num, raise it to the power of the number of digits and sum them.
3. If the sum equals num, return true. Otherwise, return false.

Example:

If num = 153:

- Step 1: 153 has 3 digits.
- Step 2: Sum of 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153.
- Since the sum equals num, return true.

---

17. double sqrtApprox(int num):

- **Definition**: This function approximates the square root of a number num using the Babylonian method.

Steps:

1. Start with an initial guess x = num.
2. Iteratively update x using the formula x = 0.5 * (x + num / x), until the difference between x^2 and num is less than a small threshold epsilon.
3. Return x as the approximated square root.

Example:

If num = 9:

- Step 1: Start with x = 9.
- Step 2: Update x = 0.5 * (9 + 9 / 9) = 0.5 * (9 + 1) = 5.
- Step 3: Update x = 0.5 * (5 + 9 / 5) = 0.5 * (5 + 1.8) = 3.4.
- Repeat until the desired precision is reached.
- The function approximates the square root to 3.

---

18. double power(int base, int exp):

- **Definition**: This function calculates base raised to the power exp.

Steps:

1. Initialize result = 1.
2. Loop from 1 to exp:
    - Multiply result by base in each iteration.

3. Return result.

Example:

If base = 2 and exp = 3:

- Step 1: result = 1.
- Step 2: Multiply result = 1 * 2 = 2.
- Step 3: Multiply result = 2 * 2 = 4.
- Step 4: Multiply result = 4 * 2 = 8.
- The function returns 8.

---

19. bool isHappy(int num):

- **Definition**: This function checks if the number num is a happy number (i.e., repeatedly replacing the number by the sum of the squares of its digits eventually leads to 1).

Steps:

1. Repeat the process of summing the squares of the digits of num until num becomes 1 or 4 (to detect cycles).
2. If num == 1, return true. If num == 4, return false.

Example:

If num = 19:

- Step 1: Sum of squares of digits 1^2 + 9^2 = 1 + 81 = 82.
- Step 2: Sum of squares of digits 8^2 + 2^2 = 64 + 4 = 68.
- Step 3: Sum of squares of digits 6^2 + 8^2 = 36 + 64 = 100.
- Step 4: Sum of squares of digits 1^2 + 0^2 + 0^2 = 1 + 0 + 0 = 1.
- Since we reached 1, return true.

---

20. bool isAbundant(int num):

- **Definition**: This function checks if a number num is an abundant number (i.e., the sum of its divisors is greater than the number itself).

Steps:

1. Call sumDevisors(num) to calculate the sum of divisors of num.
2. If the sum is greater than num, return true. Otherwise, return false.

Example:

If num = 12:

- Step 1: sumDevisors(12) = 1 + 2 + 3 + 4 + 6 = 16.
- Step 2: Since 16 > 12, return true.

If num = 8:

- Step 1: sumDevisors(8) = 1 + 2 + 4 = 7.

- Step 2: Since 7 < 8, return false.

---

21. bool isDeficient(int num):

- **Definition**: This function checks if a number num is a deficient number (i.e., the sum of its divisors is less than the number itself).

Steps:

1. Call sumDevisors(num) to calculate the sum of divisors of num.
2. If the sum is less than num, return true. Otherwise, return false.

Example:

If num = 8:

- Step 1: sumDevisors(8) = 1 + 2 + 4 = 7.
- Step 2: Since 7 < 8, return true.

If num = 12:

- Step 1: sumDevisors(12) = 1 + 2 + 3 + 4 + 6 = 16.
- Step 2: Since 16 > 12, return false.

---

22. int sumEvenFibonacci(int n):

- **Definition**: This function calculates the sum of the even Fibonacci numbers up to the nth term.

Steps:

1. Initialize a = 0, b = 1, and s = 0 for the sum.
2. Loop through the Fibonacci sequence until n terms:
    - If b is even, add it to s.
    - Update a = b and b = a + b.
3. Return the sum s.

Example:

If n = 10:

- Step 1: Initialize a = 0, b = 1, s = 0.
- Step 2: Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.
- Step 3: Even Fibonacci numbers: 0, 2, 8, 34.
- Sum of even numbers: 0 + 2 + 8 + 34 = 44.
- The function returns 44.

---

23. bool isHarshad(int num):

- **Definition**: This function checks if a number num is a Harshad number (i.e., the number is divisible by the sum of its digits).

Steps:

1. Call SumOfDigits(num) to get the sum of the digits of num.
2. If num % SumOfDigits(num) == 0, return true. Otherwise, return false.

Example:

If num = 18:

- Step 1: SumOfDigits(18) = 1 + 8 = 9.
- Step 2: 18 % 9 == 0, so return true.

If num = 19:

- Step 1: SumOfDigits(19) = 1 + 9 = 10.
- Step 2: 19 % 10 != 0, so return false.

---

24. unsigned long catanNumber(int n):

- **Definition**: This function calculates the nth Catalan number using the formula (2n)! / (n! * (n+1)!).

Steps:

1. Call factorial(2 * n) to compute the factorial of 2 * n.
2. Call factorial(n) and factorial(n + 1) to compute the factorial of n and n + 1.
3. Return the result of (factorial(2 * n) / (factorial(n) * factorial(n + 1))).

Example:

If n = 3:

- Step 1: Calculate factorial(6) = 720.
- Step 2: Calculate factorial(3) = 6 and factorial(4) = 24.
- Step 3: Return 720 / (6 * 24) = 720 / 144 = 5.
- The function returns 5.

---

25. void pascalTriangle(int n):

- **Definition**: This function prints Pascal's triangle up to n rows.

Steps:

1. Loop through each line from 0 to n - 1.
    - For each line, initialize value = 1 (the first element in each row).
    - Loop to print spaces to align the triangle.
    - Loop through the elements in the row and calculate each element using the formula: value = value * (line - i) / (i + 1).

- Print the value and move to the next row.

2. Print a new line after each row.

Example:

If n = 4:

- The triangle will be:

1

1 1

1 2 1

1 3 3 1

---

26. unsigned long bellNumber(int n):

Definition:
This function computes the n-th Bell number, which represents the number of ways to partition a set of n elements.

Steps:

1. Initialize current = 1 (Base case: B(0)=1).
2. For each i from 1 to n:
    - Store the value of current in a temporary variable.
    - For each j from 1 to i:
        - Update temp as the sum of temp and the previous value of current.
        - Update current to hold the last value of temp.
3. After all iterations, return current, which now holds the n-th Bell number.

Example:
If n=3,

- Step 1: Initialize current = 1 (Base case: B(0)=1).
- Step 2: Start iterations:
    - Iteration 1 (i=1):
        - current=1 (Carry forward).
    - Iteration 2 (i=2)
        - temp=1+1=2.
        - current=2.
    - Iteration 3 (i=3):
        - temp=2+2=4, temp=4+1=5.
        - current=5.

Output: The function returns B(3)=5.

Intermediate Bell Numbers:

- B(0)=1, B(1)=1, B(2)=2, B(3)=5

---

27. bool isKaprekar(int num):

Definition:
This function checks if a given integer num is a Kaprekar number. A number is Kaprekar if, when squared, its digits can be split into two parts that add up to the original number.

Steps:

1. **Special case**: If num=1, return true.
2. Compute the square of num.
3. Determine the total number of digits in the square.
4. **For each possible split**:
    - Divide the square into two parts: left and right.
    - Check if left+right=num (ignoring cases where right = 0).
5. If any split satisfies the condition, return true. Otherwise, return false.

Example:
For num=45num = 45num=45:

- Step 1: Compute the square: square = 45^2 = 2025
- Step 2: Determine the number of digits in square=4.
- Step 3: Split square at each position:
    - Split at 1 digit: left=2,right=025 :

      left+right=2+25=27(not equal to 45)
    - Split at 2 digitsleft=20,right=25:

      left+right=20+25=45(equal to 45)

Output: The function returns true for num=45.

---

28. bool isSmith(int num):

- **Definition**: This function checks if a number num is a Smith number (i.e., a composite number where the sum of its digits equals the sum of the digits of its prime factors).

Steps:

1. If num is prime, return false (because Smith numbers are composite).
2. Find the prime factors of num.
3. Calculate the sum of the digits of num and the sum of the digits of the prime factors.
4. If the sums are equal, return true. Otherwise, return false.

Example:

If num = 22:

- Step 1: 22 is not prime.

- Step 2: Prime factors of 22 are 2 and 11.
- Step 3: Sum of digits of 22 is 2 + 2 = 4, sum of digits of prime factors (2 + 1 + 1 = 4).
- Step 4: Since the sums are equal, return true.

---

29. int sumOfPrimes(int n):

- **Definition**: This function calculates the sum of all prime numbers less than or equal to n.

Steps:

1. Loop through each number from 2 to n.
2. For each number, check if it's prime using the isPrime function.
3. If the number is prime, add it to the sum.
4. Return the sum.

Example:

If n = 10:

- Step 1: Loop through numbers 2 to 10.
- Step 2: Prime numbers are 2, 3, 5, 7.
- Step 3: Sum of primes is 2 + 3 + 5 + 7 = 17.
- The function returns 17.

Operations on arrays:

1/basic array functions:

1. void initializeArray(int arr[], int size, int value):

Definition: This function initializes all elements of an array with a specified value.

Steps:

1. Loop through each element of the array from index 0 to size - 1.
2. Assign the specified value to each element of the array.

Example:
Input: size = 5, value = 3

- Step 1: Loop through indices 0 to 4.
- Step 2: Set all elements of the array to 3.
  Output: [3, 3, 3, 3, 3]

---

2. void printArray(int arr[], int size):

Definition: This function prints all elements of the array.

Steps:

1. Loop through each element of the array from index 0 to size - 1.

2. Print the value of each element, separated by spaces or in the desired format.

Example:
Input: arr = [1, 2, 3, 4, 5], size = 5

- Step 1: Loop through indices 0 to 4.
- Step 2: Print 1 2 3 4 5.

---

3. int findMax(int arr[], int size):

Definition: This function finds and returns the maximum element in the array.

Steps:

1. Initialize max to the first element of the array (arr[0]).
2. Loop through the remaining elements of the array (from index 1 to size - 1).
3. If the current element is greater than max, update max to the current element.
4. Return the value of max.

Example:
Input: arr = [2, 8, 1, 6, 3], size = 5

- Step 1: Initialize max = 2.
- Step 2: Loop through 8, 1, 6, 3.
- Step 3: Update max to 8.
  Output: 8

---

4. int findMin(int arr[], int size):

Definition: This function finds and returns the minimum element in the array.

Steps:

1. Initialize min to the first element of the array (arr[0]).
2. Loop through the remaining elements of the array (from index 1 to size - 1).
3. If the current element is smaller than min, update min to the current element.
4. Return the value of min.

Example:
Input: arr = [4, 7, 1, 9, 5], size = 5

- Step 1: Initialize min = 4.
- Step 2: Loop through 7, 1, 9, 5.
- Step 3: Update min to 1.
  Output: 1

---

5. int sumArray(int arr[], int size):

Definition: This function calculates and returns the sum of all elements in the array.

Steps:

1. Initialize sum to 0.
2. Loop through each element of the array from index 0 to size - 1.
3. Add each element to sum.
4. Return the value of sum.

Example:
Input: arr = [3, 6, 2, 5], size = 4

- Step 1: Initialize sum = 0.
- Step 2: Add 3, 6, 2, 5 to sum.
- Step 3: sum = 16.
  Output: 16

---

6. double averageArray(int arr[], int size):

Definition: This function calculates and returns the average of all elements in the array.

Steps:

1. Use the sumArray function to calculate the sum of all elements in the array.
2. Divide the sum by the size of the array.
3. Return the resulting value as the average.

Example:
Input: arr = [4, 8, 2, 6], size = 4

- Step 1: Calculate sum = 20 using sumArray.
- Step 2: Divide 20 / 4.
  Output: 5.0

---

7. bool isSorted(int arr[], int size):

Definition: This function checks if the array is sorted in ascending order.

Steps:

1. If the array has zero or one element, return true (it's trivially sorted).
2. Loop through each element of the array from index 0 to size - 2.
3. If any element is greater than the next element, return false.
4. If the loop completes without finding unsorted elements, return true.

Example:
Input: arr = [1, 3, 5, 7], size = 4

- Step 1: Check 1 < 3, 3 < 5, 5 < 7.
- Step 2: No unsorted elements found.
  Output: true

Input: arr = [5, 3, 8, 7], size = 4

- Step 1: Check 5 > 3.
  Output: false

2/Intermediate arrays functions:

8.void reverseArray(int arr[], int size):

Definition: Reverses the order of elements in the array.

Steps:

1. Use two indices: one starting at the beginning (i = 0) and the other at the end (size - 1).
2. Swap the elements at these two indices.
3. Move the indices closer to each other and repeat until they meet.
4. The array is now reversed.

Example:
Input: arr = {1, 2, 3, 4, 5}, size = 5

- Step 1: Swap arr[0] and arr[4] → {5, 2, 3, 4, 1}
- Step 2: Swap arr[1] and arr[3] → {5, 4, 3, 2, 1}
- Step 3: Indices meet; stop swapping.

Output: arr = {5, 4, 3, 2, 1}

9.void countEvenOdd(int arr[], int size, int* evenCount, int* oddCount):

Definition: Counts the number of even and odd elements in the array and stores the results in the provided pointers.

Steps:

1. Initialize *evenCount and *oddCount to 0.
2. Loop through each element in the array.
3. If the current element is divisible by 2, increment *evenCount.
4. Otherwise, increment *oddCount.
5. The final values of *evenCount and *oddCount represent the counts of even and odd elements in the array.

Example:
Input: arr = {1, 2, 3, 4, 5}, size = 5

- Step 1: Check 1 → Odd → oddCount = 1, evenCount = 0
- Step 2: Check 2 → Even → oddCount = 1, evenCount = 1
- Step 3: Check 3 → Odd → oddCount = 2, evenCount = 1
- Step 4: Check 4 → Even → oddCount = 2, evenCount = 2
- Step 5: Check 5 → Odd → oddCount = 3, evenCount = 2

Output: evenCount = 2, oddCount = 3

10.int secondLargest(int arr[], int size):

Definition: Finds the second largest element in the array.

Steps:

1. If the array has fewer than 2 elements, return an error message and -1.
2. Initialize two variables: largest and secondLargest.
3. Loop through the array and compare each element:
    - If an element is greater than largest, update secondLargest to largest, then update largest.
    - If an element is greater than secondLargest and not equal to largest, update secondLargest.
4. If no second largest element is found (i.e., all elements are the same), return -1.
5. Otherwise, return secondLargest.

Example:
Input: arr = {1, 3, 5, 7}, size = 4

- Step 1: largest = 1, secondLargest = -1
- Step 2: Compare 3 → largest = 3, secondLargest = 1
- Step 3: Compare 5 → largest = 5, secondLargest = 3
- Step 4: Compare 7 → largest = 7, secondLargest = 5

Output: secondLargest = 5

11.void elementFrequency(int arr[], int size):

Definition: Prints the frequency of each unique element in the array.

Steps:

1. Initialize an array visited[] of the same size as the input array to keep track of visited elements (initialized to 0).
2. Loop through the array to check each element:
    - If the element has not been visited (i.e., visited[i] == 0), count how many times it appears in the rest of the array.
    - Mark the visited elements to avoid counting them again.
3. Print the frequency of each element as it's counted.
4. The process repeats until all elements have been processed.

Example:
Input: arr = {1, 2, 2, 3, 1, 4, 4}, size = 7

- Step 1: 1 occurs 2 times, mark both occurrences as visited.
- Step 2: 2 occurs 2 times, mark both occurrences as visited.
- Step 3: 3 occurs 1 time, mark as visited.
- Step 4: 4 occurs 2 times, mark both occurrences as visited.

12.int removeDuplicates(int arr[], int size):

Definition: Removes duplicate elements from the array and returns the new size.

Steps:

1. If the array has zero or one element, return the size as it is (no duplicates possible).
2. Initialize a variable j to keep track of the index of the next unique element.
3. Loop through the array starting from the second element.

4. For each element, if it is not equal to the last unique element (i.e., arr[i] != arr[j]), place it at the next position (arr[j + 1]), and increment j.
5. The array will now have only unique elements up to index j.
6. Return j + 1 as the new size of the array (the count of unique elements).

Example:
Input: arr = {1, 1, 2, 3, 3, 4}, size = 6

- Step 1: Compare 1 and 1 → Skip duplicate.
- Step 2: Compare 2 → Place at index 1.
- Step 3: Compare 3 → Place at index 2.
- Step 4: Compare 4 → Place at index 3.

Output:
arr = {1, 2, 3, 4}, New size = 4

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

Definition: Performs binary search on a sorted array to find the target element.

Steps:

1. Initialize two pointers: left at 0 and right at size - 1.
2. Loop until left is less than or equal to right.
3. Find the middle index: mid = left + (right - left) / 2.
4. If arr[mid] equals target, return mid (the index of the target).
5. If arr[mid] is less than target, search in the right half by setting left = mid + 1.
6. If arr[mid] is greater than target, search in the left half by setting right = mid - 1.
7. If the target is not found, return -1.

Example:
Input: arr = {1, 3, 5, 7, 9}, size = 5, target = 5

- Step 1: left = 0, right = 4, mid = 2 → arr[mid] = 5
- Step 2: Target found at index 2.

Output: 2

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

Definition: Performs a linear search to find the target element in the array.

Steps:

1. Loop through each element of the array from index 0 to size - 1.
2. For each element, check if it equals the target.
3. If a match is found, return the index of the element.
4. If no match is found after looping through the entire array, return -1.

Example:
Input: arr = {5, 3, 8, 1, 2}, size = 5, target = 8

- Step 1: Compare 5 with 8 → No match.
- Step 2: Compare 3 with 8 → No match.
- Step 3: Compare 8 with 8 → Match found at index 2.

Output: 2

15.void leftShift(int arr[], int size, int rotations):

Definition: Shifts the array to the left by a specified number of positions.

Steps:

1. If rotations is greater than or equal to the array size, reduce rotations by taking the modulus with the array size (rotations = rotations % size).
2. Create a temporary array temp[] to store the first rotations elements of the array.
3. Loop through the array from the index rotations to size - 1, shifting each element to the left by rotations positions.
4. Copy the elements stored in temp[] to the end of the array.
5. The array will now be shifted to the left by the specified number of rotations.

Example:
Input: arr = {1, 2, 3, 4, 5}, size = 5, rotations = 2

- Step 1: Store first 2 elements {1, 2} in temp[].
- Step 2: Shift remaining elements {3, 4, 5} to the left.
- Step 3: Place {1, 2} at the end.

Output:
arr = {3, 4, 5, 1, 2}

16.void rightShift(int arr[], int size, int rotations):

Definition: Shifts the array to the right by a specified number of positions.

Steps:

1. If rotations is greater than or equal to the array size, reduce rotations by taking the modulus with the array size (rotations = rotations % size).
2. Create a temporary array temp[] to store the last rotations elements of the array.
3. Loop through the array from the index size - rotations to size - 1, storing each element in temp[].
4. Shift the remaining elements from the start of the array towards the right.
5. Copy the elements stored in temp[] to the beginning of the array.
6. The array will now be shifted to the right by the specified number of rotations.

Example:
Input: arr = {1, 2, 3, 4, 5}, size = 5, rotations = 2

- Step 1: Store last 2 elements {4, 5} in temp[].

- Step 2: Shift remaining elements {1, 2, 3} to the right.
- Step 3: Place {4, 5} at the beginning.

Output:
arr = {4, 5, 1, 2, 3}

3/sorting algorithms:

17.void bubbleSort(int arr[], int size):

Definition: Sorts the array in ascending order using the bubble sort algorithm.

Steps:

1. Loop through the entire array multiple times, where each pass pushes the largest unsorted element to the end of the array.
2. On each pass, compare adjacent elements (arr[j] and arr[j + 1]).
3. If arr[j] is greater than arr[j + 1], swap the elements.
4. Repeat this process for size - 1 passes, as the array will be sorted after this number of iterations.
5. The array will be sorted in ascending order after the loop completes.

Example:
Input: arr = {5, 2, 9, 1, 5, 6}, size = 6

- Step 1: Compare 5 and 2, swap them → {2, 5, 9, 1, 5, 6}.
- Step 2: Continue comparing and swapping.
- Step 3: After all passes, array will be sorted: {1, 2, 5, 5, 6, 9}.

Output:
arr = {1, 2, 5, 5, 6, 9}

18.void selectionSort(int arr[], int size):

Definition: Sorts the array in ascending order using the selection sort algorithm.

Steps:

1. Start with the entire array.
2. Find the smallest element in the unsorted portion of the array.
3. Swap this smallest element with the first element of the unsorted portion.
4. Move the boundary of the sorted portion one element to the right.
5. Repeat steps 2-4 for the remaining unsorted portion of the array.
6. After size - 1 passes, the array will be sorted.

Example:
Input: arr = {64, 34, 25, 12, 22, 11, 90}, size = 7

- Step 1: Find the minimum value 11 and swap it with the first element → {11, 34, 25, 12, 22, 64, 90}.
- Step 2: Find the next minimum value 12 and swap it with the second element → {11, 12, 25, 34, 22, 64, 90}.
- Step 3: Continue sorting until the array is fully sorted.

- Final sorted array: {11, 12, 22, 25, 34, 64, 90}.

Output:
arr = {11, 12, 22, 25, 34, 64, 90}

19.void insertionSort(int arr[], int size):

Definition: Sorts the array in ascending order using the insertion sort algorithm.

Steps:

1. Start from the second element of the array (arr[1]), considering the first element as already sorted.
2. Compare the current element (arr[i]) with the elements in the sorted portion (from right to left).
3. Shift the larger elements one position to the right to make space for the current element.
4. Insert the current element in its correct position in the sorted portion.
5. Repeat this process for every element from the second to the last element.
6. The array will be sorted in ascending order after all passes.

Example:
Input: arr = {12, 11, 13, 5, 6}, size = 5

- Step 1: Compare 11 with 12 and insert it before → {11, 12, 13, 5, 6}.
- Step 2: Compare 13 with 12, no change → {11, 12, 13, 5, 6}.
- Step 3: Compare 5 with 13, 12, and 11, then insert it → {5, 11, 12, 13, 6}.
- Step 4: Compare 6 with 13, 12, and 11, then insert it → {5, 6, 11, 12, 13}.

Output:
arr = {5, 6, 11, 12, 13}

_____

Conclusion:

This project strengthens programming skills by emphasizing algorithms, modularity, and numerical computations. Through the implementation of these functions, students not only improve their theoretical knowledge but also gain practical experience in C programming, reinforcing both the conceptual and hands-on aspects of the language.

_____

Attached:

mylib.c: **C source file**. It contains the **implementation** of the functions defined in the library.

mylib.h: **header file**. It contains **declarations** for the functions and variables that are defined in mylib.c