End-Semester Project

Level: 1st Year **Material:** Algorithms and Static Data Structures

1. Project description

The aim of this project is for students to create a comprehensive C library covering a variety of data processing functions. The library will include modules for handling numbers, strings, arrays, and matrices. These modules are designed to enhance students algorithmic skills and programming expertise. While specific functions have been proposed, students are encouraged to add additional functions based on their own research and creativity. The project is divided into four main categories:

- **Operations on Numbers**: Various mathematical functions and procedures for number analysis.
- **Operations on Strings**: Functions for strings manipulation and processing.
- **Operations on Arrays**: Basic and advanced algorithms for handling 1D arrays.
- **Operations on Matrices**: Functions to perform various operations on 2D matrices.

2. Objectives

- To strengthen students skills in algorithmic thinking and C programming.
- To provide a practical application of functions, modularity, and code reuse.
- To help students understand complex data manipulations, including numerical analysis, string handling, array sorting/searching, and matrices computations.

3. Expected deliverables

- A complete C library (.c and .h files) containing all implemented functions.
- A test suite with sample programs to demonstrate the use of each function.
- A detailed report including:
  - Project objectives and problem statement.
  - Analysis and algorithm design for each function.
  - Implementation details and code snippets.

4. Evaluation criteria

- **Correctness**: The library should work correctly for all specified operations.
- **Modularity**: The use of functions and procedures should demonstrate good programming practices.
- **Documentation**: Code comments and a well-structured report are essential.

5. Project schedule

All the students are requested to submit their complete C library, test suite, and project report by January, 19th 2025.

6. Examples of modules

Develop the following modules:

6.1. Operations on numbers

- **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.
  - **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.

End-Semester Project

Level: 1st Year **Material:** Algorithms and Static Data Structures

- **Greatest Common Divisor (GCD) (int gcd(int a, int b))**: Computes the GCD of two numbers using the Euclidean algorithm.
- **Least Common Multiple (LCM) (int lcm(int a, int b))**: Calculates the LCM of two numbers.
- **Factorial (long factorial(int num))**: Computes the factorial of a number.
- **Even/Odd (bool isEven(int num))**: Returns TRUE if the number is even, FALSE otherwise.

- **Intermediate operations on numbers**
  - **Prime Factorization (void primeFactors(int num))**: Prints all prime factors of a number.
  - **Armstrong Number (bool isArmstrong(int num))**: Checks if a number is an Armstrong number (e.g., 153 because $1^3+5^3+3^3=153$).
  - **Fibonacci Sequence (void fibonacciSeries(int n))**: Generates the Fibonacci sequence up to the n^{th} term.
  - **Sum of Divisors (int sumDivisors(int num))**: Calculates the sum of all divisors of a number.
  - **Perfect Number (bool isPerfect(int num))**: Checks if a number is perfect (e.g., 6 because 1+2+3=6).
  - **Magic Number (bool isMagic(int num))**: Checks if the sum of the digits recursively equals 1 (e.g., 19 is a magic number because 1+9=10 and 1+0=1).
  - **Automorphic Number (bool isAutomorphic(int num))**: Checks if a number square ends with the number itself (e.g., 25 is automorphic because $25^2=625$).

- **Advanced operations on numbers**
  - **Binary Conversion (void toBinary(int num))**: Converts a number to its binary representation.
  - **Narcissistic Number (bool isNarcissistic(int num))**: Checks if a number is equal to the sum of its digits raised to the power of the number of digits (e.g., 370).
  - **Square Root Calculation (double sqrtApprox(int num))**: Calculates the square root using the Babylonian method.
  - **Exponentiation (double power(int base, int exp))**
  - **Happy Number (bool isHappy(int num))**: Checks if a number is happy (repeatedly summing the squares of its digits eventually leads to 1).
  - **Abundant Number (bool isAbundant(int num))**: Checks if the sum of the divisors of a number is greater than the number itself.
  - **Deficient Number (bool isDeficient(int num))**: Checks if the sum of the divisors of a number is less than the number itself.
  - **Sum of Fibonacci Even Numbers (int sumEvenFibonacci(int n))**: Calculates the sum of even Fibonacci numbers up to n^{th} term.
  - **Harshad Number (bool isHarshad(int num))**: Checks if a number is divisible by the sum of its digits.
  - **Catalan Number Calculation (unsigned long catalanNumber(int n))**: Computes the n^{th} Catalan number.
  - **Pascal Triangle (void pascalTriangle(int n))**: Generates the first n rows of Pascal Triangle.
  - **Bell Number (unsigned long bellNumber(int n))**: Computes the n^{th} 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 (e.g., 45 because $45^2=2025$ and 20+25=45).
  - **Smith Number (bool isSmith(int num))**: Checks if a number is a Smith number (non-prime numbers whose sum of digits equals the sum of 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.

Academic Year: 2024/2025

End-Semester Project

Level: 1st Year **Material:** Algorithms and Static Data Structures

6.2. Operations on strings

- **Basic String Functions**
  - **Calculate String Length (int stringLength(char* str))**: Returns the length of the string.
  - **Copy String (void stringCopy(char* dest, const char* src))**: Copies the source string to the destination string.
  - **Concatenate Strings (void stringConcat(char* dest, const char* src))**: Concatenates the source string to the end of the destination string.
  - **Compare Strings (int stringCompare(const char* str1, const char* str2))**: Compares two strings lexicographically.
  - **Check if Empty (bool isEmpty(char* str))**: Checks if a string is empty.
  - **Reverse a String (void reverseString(char* str))**: Reverses the characters in a string.
  - **Convert to Uppercase (void toUpperCase(char* str))**: Converts all characters in a string to uppercase.
  - **Convert to Lowercase (void toLowerCase(char* str))**: Converts all characters in a string to lowercase.

- **Intermediate String Functions**
  - **Palindrome (bool isPalindrome(char* str))**: Checks if a string reads the same forwards and backwards.
  - **Count Vowels and Consonants (void countVowelsConsonants(char* str, int* vowels, int* consonants))**: Counts the number of vowels and consonants in a string.
  - **Find Substring (int findSubstring(const char* str, const char* sub))**: Finds the first occurrence of a substring in a string.
  - **Remove Whitespaces (void removeWhitespaces(char* str))**: Removes all spaces from a string.
  - **Anagram (bool isAnagram(char* str1, char* str2))**: Checks if two strings are anagrams of each other.
  - **Character Frequency (void charFrequency(char* str, int* freq))**: Calculates the frequency of each character in a string.
  - **Count Words (int countWords(char* str))**: Counts the number of words in a string.
  - **Remove Duplicate Characters (void removeDuplicates(char* str))**: Removes duplicate characters from a string.

- **Advanced String Functions**
  - **String Compression (void compressString(char* str, char* result))**: Compresses a string using basic Run-Length Encoding (RLE).
  - **Find Longest Word (void longestWord(char* str, char* result))**: Finds the longest word in a sentence.
  - **String Rotation Check (bool isRotation(char* str1, char* str2))**: Checks if one string is a rotation of another (e.g., "abc" is a rotation of "cab").
  - **Count Specific Character (int countChar(char* str, char ch))**: Counts the occurrences of a specific character in a string.
  - **Find and Replace (void findAndReplace(char* str, char* find, char* replace))**: Replaces all occurrences of a substring with another substring.
  - **Longest Palindromic Substring (void longestPalindrome(char* str, char* result))**: Finds the longest palindromic substring in a given string.
  - **String Permutations (void printPermutations(char* str))**: Generates and prints all permutations of a string.

End-Semester Project

Level: 1st Year **Material:** Algorithms and Static Data Structures

- **Split String (void splitString(char* str, char delimiter, char tokens[][100], int* tokenCount))**: Splits a string into tokens based on a specified delimiter.

6.3. Operations on arrays

- **Basic Array Functions**
  - **Initialize Array (void initializeArray(int arr[], int size, int value))**: Initializes an array with a specified value.
  - **Print Array (void printArray(int arr[], int size))**: Prints the elements of the array.
  - **Find Maximum (int findMax(int arr[], int size))**: Finds the maximum element in the array.
  - **Find Minimum (int findMin(int arr[], int size))**: Finds the minimum element in the array.
  - **Calculate Sum (int sumArray(int arr[], int size))**: Returns the sum of all elements in the array.
  - **Calculate Average (double averageArray(int arr[], int size))**: Calculates the average of the elements of the array.
  - **Check if Sorted (bool isSorted(int arr[], int size))**: Checks if the array is sorted in ascending order.

- **Intermediate Array Functions**
  - **Reverse Array (void reverseArray(int arr[], int size))**: Reverses the elements of the array.
  - **Count Even and Odd Numbers (void countEvenOdd(int arr[], int size, int* evenCount, int* oddCount))**: Counts the number of even and odd elements in the array.
  - **Find Second Largest (int secondLargest(int arr[], int size))**: Finds the second largest element in the array.
  - **Find Frequency of Elements (void elementFrequency(int arr[], int size))**: Finds the frequency of each unique element in the array.
  - **Remove Duplicates (int removeDuplicates(int arr[], int size))**: Removes duplicate elements from the array and returns the new size.
  - **Binary Search (int binarySearch(int arr[], int size, int target))**: Performs binary search on a sorted array to find a target element.
  - **Linear Search (int linearSearch(int arr[], int size, int target))**: Performs linear search to find a target element in the array.
  - **Left Shift Array (void leftShift(int arr[], int size, int rotations))**: Shifts the array to the left by a specified number of positions.
  - **Right Shift Array (void rightShift(int arr[], int size, int rotations))**: Shifts the array to the right by a specified number of positions.

- **Sorting Algorithms**
  - **Bubble Sort (void bubbleSort(int arr[], int size))**: Sorts the array using the bubble sort algorithm.
  - **Selection Sort (void selectionSort(int arr[], int size))**: Sorts the array using the selection sort algorithm.
  - **Insertion Sort (void insertionSort(int arr[], int size))**: Sorts the array using the insertion sort algorithm.
  - **Merge Sort (void mergeSort(int arr[], int left, int right))**: Sorts the array using the merge sort algorithm.
  - **Quick Sort (void quickSort(int arr[], int low, int high))**: Sorts the array using the quick sort algorithm.

- **Advanced Array Functions**
  - **Find Missing Number (int findMissingNumber(int arr[], int size))**: Finds the missing number in an array of size n-1 containing numbers from 1 to n.
  - **Find Pairs with Given Sum (void findPairsWithSum(int arr[], int size, int sum))**: Finds all pairs of elements whose sum is equal to a given value.

End-Semester Project

Level: 1st Year **Material:** Algorithms and Static Data Structures

- **Subarray with Given Sum (void findSubArrayWithSum(int arr[], int size, int sum))**: Finds a continuous subarray that adds up to a given sum.
- **Rearrange Positive and Negative Numbers (void rearrangeAlternatePositiveNegative(int arr[], int size))**: Rearranges the array such that positive and negative numbers alternate.
- **Find Majority Element (int findMajorityElement(int arr[], int size))**: Finds the majority element in the array (an element that appears more than n/2 times).
- **Longest Increasing Subsequence (int longestIncreasingSubsequence(int arr[], int size))**: Finds the length of the longest increasing subsequence in the array.
- **Find Duplicates (void findDuplicates(int arr[], int size))**: Identifies duplicate elements in the array.
- **Find Intersection of Two Arrays (void findIntersection(int arr1[], int size1, int arr2[], int size2))**: Finds the common elements between two arrays.
- **Find Union of Two Arrays (void findUnion(int arr1[], int size1, int arr2[], int size2))**: Finds the union of two arrays.

6.4. Operations on matrices

- **Basic Matrix Functions**
  - **Initialize Matrix (void initializeMatrix(int rows, int cols, int matrix[rows][cols], int value))**: Initializes all elements of a matrix to a given value.
  - **Print Matrix (void printMatrix(int rows, int cols, int matrix[rows][cols]))**: Prints the elements of the matrix in a formatted way.
  - **Input Matrix (void inputMatrix(int rows, int cols, int matrix[rows][cols]))**: Allows the user to input elements of the matrix.

- **Matrix Arithmetic**
  - **Matrix Addition (void addMatrices(int rows, int cols, int mat1[rows][cols], int mat2[rows][cols], int result[rows][cols]))**: Adds two matrices element-wise.
  - **Matrix Subtraction (void subtractMatrices(int rows, int cols, int mat1[rows][cols], int mat2[rows][cols], int result[rows][cols]))**: Subtracts the second matrix from the first matrix element-wise.
  - **Matrix Multiplication (void multiplyMatrices(int rows1, int cols1, int mat1[rows1][cols1], int rows2, int cols2, int mat2[rows2][cols2], int result[rows1][cols2]))**: Multiplies two matrices and stores the result in a third matrix.
  - **Scalar Multiplication (void scalarMultiplyMatrix(int rows, int cols, int matrix[rows][cols], int scalar))**: Multiplies each element of the matrix by a given scalar.

- **Matrix Properties and Checks**
  - **Check if Square Matrix (bool isSquareMatrix(int rows, int cols))**: Checks if the matrix is square (i.e., rows ==cols).
  - **Check if Identity Matrix (bool isIdentityMatrix(int size, int matrix[size][size]))**: Checks if the matrix is an identity matrix.
  - **Check if Diagonal Matrix (bool isDiagonalMatrix(int size, int matrix[size][size]))**: Checks if all non-diagonal elements are zero.
  - **Check if Symmetric Matrix (bool isSymmetricMatrix(int size, int matrix[size][size]))**: Checks if the matrix is symmetric (i.e., matrix[i][j]==matrix[j][i]).

End-Semester Project

Level: 1st Year **Material:** Algorithms and Static Data Structures

- **Check if Upper Triangular Matrix (bool isUpperTriangular(int size, int matrix[size][size]))**: Checks if the matrix is upper triangular (i.e., all elements below the main diagonal are zero).

- **Matrix Operations**
  - **Transpose Matrix (void transposeMatrix(int rows, int cols, int matrix[rows][cols], int result[cols][rows]))**: Computes the transpose of a matrix.
  - **Determinant of a Matrix (int determinantMatrix(int size, int matrix[size][size]))**: Calculates the determinant of a square matrix.
  - **Inverse of a Matrix (void inverseMatrix(int size, double matrix[size][size], double result[size][size]))**: Calculates the inverse of a matrix using Gaussian elimination.
  - **Matrix Power (void matrixPower(int size, int matrix[size][size], int power, int result[size][size]))**: Raises a square matrix to a given power.

- **Advanced Matrix Functions**
  - **Cofactor Matrix (void cofactorMatrix(int size, int matrix[size][size], int cofactor[size][size]))**: Computes the cofactor matrix of a given square matrix.
  - **Adjoint Matrix (void adjointMatrix(int size, int matrix[size][size], int adjoint[size][size]))**: Computes the adjoint of a given square matrix.
  - **LU Decomposition (void luDecomposition(int size, double matrix[size][size], double lower[size][size], double upper[size][size]))**: Performs LU decomposition of a matrix.
  - **Matrix Rank (int matrixRank(int rows, int cols, int matrix[rows][cols]))**: Determines the rank of a matrix.

- **Special Matrix Operations**
  - **Find Trace of a Matrix (int traceMatrix(int size, int matrix[size][size]))**: Calculates the trace of a square matrix (sum of diagonal elements).
  - **Rotate Matrix 90 Degrees (void rotateMatrix90(int size, int matrix[size][size]))**: Rotates the matrix by 90 degrees clockwise.
  - **Find Eigenvalues (using Numerical Methods) (void findEigenvalues(int size, double matrix[size][size], double eigenvalues[size]))**: Estimates the eigenvalues of a square matrix.

7. **Optional parts**
   - **Caeser cipher (void caesarCipher(char *text, int shift))**: shifts each letter in the text by a fixed number of positions in the alphabet.
   - **Substitution cipher (void substitutionCipher(char *text, const char *key))**: replaces each letter with a corresponding letter from a substitution alphabet.
   - **XOR cipher (void xorCipher(char *text, char key))**: applies a bitwise XOR operation between the message and a key.
   - **Vigenere cipher (void vigenereCipher(char *text, const char *key, int encrypt))**: uses a keyword to shift each letter of the plaintext.
   - **Atbash cipher (void atbashCipher(char *text))**: substitutes each letter with its reverse counterpart in the alphabet (A, a with Z, z; B, b with Y, y; etc.).
   - **Rail fence cipher (void railFenceCipher(const char *text, char *result, int depth))**: arranges text in a zigzag pattern and reads it row by row.