المدرسة الوطنية العليا في الأمن السيبراني
NATIONAL SCHOOL OF CYBERSECURITY

END SEMESTER PROJECT REPORT (ALSDS)

2025

Presented for :

SOUALMI Abdellah
HENTOUT Abdelfetah
BERGHOUT Yesser Moussa

Presented by :

Hassani Fateh
Haddadi Rabeh

Introduction

As part of the **end-of-semester project for the ALSD** module, I have developed a comprehensive **programming library** designed to assist programmers in their work with **numbers, strings, arrays, and matrices**.

This library is structured with a collection of well-documented functions and procedures, each tailored to simplify complex operations and enhance efficiency.

The library aims to serve as a versatile toolkit for developers by providing solutions to common computational challenges. Every function and procedure is designed with clear objectives, well-defined inputs and outputs, and logical approaches to ensure clarity and ease of use.

This report presents a detailed breakdown of the components in our library, including the **objective**, **input-output** specifications, and the **thinking methods** used to develop each feature.

This documentation not only highlights the practical use of each code but also reflects the systematic approach taken to ensure its functionality and reliability. The project will be presented on **GitHub** to professors for review and feedback.

Following the introduction, you will find the full report that outlines the development process and structure of the project.

Operations on numbers

Basic operations on numbers

1. Sum of Digits

- **Objective :** Calculate the sum of all digits of a given number.
- **Inputs** : A single integer (num).
- **Outputs** : An integer representing the sum of the digits of the input number.
- **Thinking Method :** The function uses a loop to repeatedly extract the last digit of the number using modulo (num % 10) and adds it to a running sum. The number is divided by 10 (num /= 10) in each iteration until it becomes 0.

2. Reverse the Digits of a Number

- **Objective**: Reverse the order of digits of a given number.
- **Inputs:** A single integer (num).
- **Outputs**: The reversed integer.
- **Thinking Method:** The function iteratively appends the last digit of the number to a result variable and removes the digit from the original number.

3. Palindrome

- **Objective:** Check if a number is a palindrome.
- **Inputs:** A single integer (num).
- **Outputs**: A boolean (true or false).
- **Thinking Method**: A number is checked by reversing it using the reverseNumber function and comparing it to the original.

4. Prime Number

- **Objective:** Determine if a number is prime.
- **Inputs**: A single integer (num).
- **Outputs:** A boolean indicating primality.
- **Thinking Method:** The function checks divisibility from 2 to the square root of the number. If any divisor is found, the number is not prime.

5. Greatest Common Divisor (GCD)

- **Objective:** Find the greatest common divisor of two numbers.
- **Inputs:** Two integers (a and b).
- **Outputs**: An integer representing the GCD.
- **Thinking Method:** The function iterates through all possible divisors up to the smaller of the two numbers, storing the highest common divisor.

6. Least Common Multiple (LCM)

- **Objective:** Calculate the least common multiple of two numbers.
- **Inputs**: Two integers (a and b).
- **Outputs:** An integer representing the LCM.
- **Thinking Method:** The LCM is computed using the formula: lcm(a, b) = (a * b) / gcd(a, b).

7. Factorial

- **Objective:** Compute the factorial of a number.
- **Inputs:** A single integer (num).
- **Outputs:** A long integer representing the factorial.
- **Thinking Method:** A loop multiplies numbers from 1 to the input number, accumulating the product.

8. Combination

- **Objective:** Calculate the combination of n items taken p at a time.
- **Inputs**: Two integers (n, p).
- **Outputs**: An integer representing the combination.
- **Thinking Method:** The formula : $C(n,k)=\frac{n!}{k!(n-k)!}$ (k=p)
- is implemented using the factorial function.

9. Even or Odd

- **Objective**: Check if a number is even or odd.
- **Inputs:** A single integer (num).
- **Outputs:** A boolean (true for even, false for odd).
- **Thinking Method:** The modulo operator (num % 2) determines if a number is divisible by 2.

10. Number of Digits

- **Objective:** Count the number of digits in a number.
- **Inputs:** A single integer (num).
- **Outputs:** The count of digits.
- **Thinking Method:** The function repeatedly divides the number by 10, incrementing a counter until the number becomes 0.

11. Prime Factorization

- **Objective:** List all prime factors of a number.
- **Inputs:** A single integer (num).
- **Outputs**: Prime factors printed to the console.
- **Thinking Method:** Divisors are checked for primality using isPrime, and valid factors are printed.

Intermediate operations on numbers

1. Armstrong Number

- **Objective:** Determine if a number is an Armstrong number.
- **Inputs:** A single integer (num).
- **Outputs:** A boolean indicating if the number is Armstrong.
- **Thinking Method:** Each digit of the number is raised to the power of the total number of digits and summed. The sum is compared to the original number.

2. Fibonacci Sequence

- **Objective:** Generate the first n Fibonacci numbers.
- **Inputs:** A single integer (n).
- **Outputs:** The sequence printed to the console.
- **Thinking Method:** The function iteratively computes Fibonacci numbers by summing the last two values.

3. Sum of Divisors

- **Objective:** Calculate the sum of all divisors of a number.
- **Inputs:** A single integer (num).
- **Outputs**: An integer representing the sum of divisors.
- **Thinking Method:** Divisors are identified by iterating up to the square root of the number.

4. Perfect Number

- **Objective:** Check if a number is a perfect number.
- **Inputs:** A single integer (num).
- **Outputs**: A boolean (true if perfect, false otherwise).
- **Thinking Method:** The sum of divisors (excluding the number itself) is compared to the number.

5. Magic Number

- **Objective**: Determine if a number is a magic number.
- **Inputs:** A single integer (num).
- **Outputs**: A boolean indicating if the number is magic.
- **Thinking Method:** The sum of digits is repeatedly calculated until a single digit is obtained. If the result is 1, the number is magic.

6. Automorphic Number

- **Objective:** Check if a number is automorphic.
- **Inputs:** A single integer (num).
- **Outputs:** A boolean (true if automorphic, false otherwise).
- **Thinking Method:** A number is automorphic if its square ends in the number itself.

Advanced operations on numbers

1. Binary Conversion

- **Objective:** Convert a decimal number to its binary representation.
- **Inputs:** A single integer (num).
- **Outputs:** The binary equivalent printed to the console.
- **Thinking Method:** The function repeatedly computes the remainder of division by 2
- (num % 2) and builds the binary number by appending digits.

2. Narcissistic Number(same as Armastrong number)

- **Objective:** Determine if a number is a narcissistic number.
- **Inputs:** A single integer (num).
- **Outputs:** A boolean (true if narcissistic, false otherwise).
- **Thinking Method**: Each digit of the number is raised to the power of the total number of digits, summed, and compared to the original number.

3. Square Root Calculation

- **Objective:** Approximate the square root of a number.
- **Inputs:** A single integer (num).
- **Outputs:** A double representing the approximate square root.
- **Thinking Method:** The Babylonian method is used, iteratively improving the approximation until the change is negligible.

4. Exponentiation

- **Objective:** Compute the result of raising a base to an exponent.
- **Inputs:** Two integers (base and exp).
- **Outputs:** A double representing the result.
- **Thinking Method:** For positive exponents, a loop accumulates the product. For negative exponents, the reciprocal of the base is used.

5. Happy Number

- **Objective:** Determine if a number is a happy number.
- **Inputs:** A single integer (num).
- **Outputs:** A boolean (true if happy, false otherwise).
- **Thinking Method:** The function calculates the sum of the squares of the digits repeatedly until a single digit is reached. A happy number will eventually equal 1.

6. Abundant Number

- **Objective:** Check if a number is abundant.
- **Inputs:** A single integer (num).
- **Outputs:** A boolean (true if abundant, false otherwise).
- **Thinking Method:** A number is abundant if the sum of its proper divisors exceeds the number itself.

7. Deficient Number

- **Objective:** Check if a number is deficient.
- **Inputs:** A single integer (num).
- **Outputs:** A boolean (true if deficient, false otherwise).
- **Thinking Method:** A number is deficient if it is not abundant.

8. Sum of Fibonacci Even Numbers

- **Objective:** Compute the sum of even Fibonacci numbers up to the nth term.
- **Inputs:** A single integer (n).
- **Outputs:** The sum of even Fibonacci numbers.
- **Thinking Method:** The function generates Fibonacci numbers and adds only the even numbers to the sum.

9. Harshad Number

- **Objective:** Determine if a number is a Harshad number.
- **Inputs:** A single integer (num).
- **Outputs:** A boolean (true if Harshad, false otherwise).
- **Thinking Method:** A number is Harshad if it is divisible by the sum of its digits.

10. Catalan Number Calculation

- **Objective:** Compute the nth Catalan number.
- **Inputs:** A single integer (n).
- **Outputs:** The nth Catalan number printed to the console.
- **Thinking Method:** The Catalan number is calculated using the formula:
- Cn= $\frac{(2n)!}{(n+1)!\,n!}$

11. Pascal's Triangle

- **Objective**: Generate Pascal's Triangle up to the nth row.
- **Inputs:** A single integer (n).
- **Outputs:** The triangle printed to the console.
- **Thinking Method:** Each element is computed using the combination formula C(n,k)

12. Bell Number

- **Objective:** Calculate the nth Bell number.
- **Inputs:** A single integer (n).
- **Outputs:** The nth Bell number.
- **Thinking Method:** The Bell number is computed iteratively by summing the products of combinations and previous Bell numbers.

13. Kaprekar Number

- **Objective:** Check if a number is a Kaprekar number.
- **Inputs:** A single integer (num).
- **Outputs:** A boolean (true if Kaprekar, false otherwise).
- **Thinking Method:** A number is Kaprekar if its square can be split into two parts that sum to the original number.

14. Smith Number

- **Objective:** Check if a number is a Smith number.
- **Inputs**: A single integer (num).
- **Outputs**: A boolean (true if Smith, false otherwise).
- **Thinking Method:** A composite number is Smith if the sum of its digits equals the sum of the digits of its prime factors.

15. Sum of Prime Numbers

- **Objective:** Calculate the sum of all prime numbers up to a given number.
- **Inputs:** A single integer (n).
- Outputs: The sum of prime numbers.
- **Thinking Method:** The function iterates through numbers, checks for primality, and adds primes to the sum.

Operations on strings

Basic String Functions

1. stringLength

- **Objective:** To calculate the length of a string.
- **Inputs:** A string str (character array).
- **Outputs**: The length of the string as an integer.
- **Thinking Method:** The function iterates through each character in the string, counting each until the null-terminator ('\0') is encountered.

2. stringCopy

- Objective: To copy one string to another.
- **Inputs**: A destination string dest and a source string src.
- **Outputs:** Copies the content of src into dest.
- **Thinking Method:** The function loops through each character of the source string and copies it to the destination string, stopping when it reaches the null-terminator.

3. stringConcat

- **Objective:** To concatenate one string to the end of another string.
- **Inputs:** A destination string dest and a source string src.
- **Outputs**: The source string is concatenated to the destination string.
- **Thinking Method:** The function first locates the end of the destination string and then copies the characters of the source string to this position.

4. stringCompare

- **Objective:** To compare two strings for equality.
- **Inputs:** Two strings str1 and str2.
- **Outputs:** A positive integer if str1 is greater, a negative integer if str1 is smaller, or 0 if the strings are equal.
- **Thinking Method:** The function compares characters of both strings one by one, returning the difference when a mismatch is found or 0 if the strings are identical.

5. isEmpty

- **Objective**: To check if a string is empty.
- **Inputs**: A string str.
- **Outputs**: true if the string is empty, false otherwise.
- **Thinking Method:** The function checks if the length of the string is zero using the stringLength function.

6. reverseString

- **Objective:** To reverse the content of a string.
- **Inputs**: A string str.
- **Outputs:** The string with its characters in reverse order.
- Thinking Method: The function swaps the characters from the beginning and end of the string until it reaches the middle.

7. toUpperCase

- **Objective:** To convert a string to uppercase.
- **Inputs:** A string str.
- **Outputs**: The string with all lowercase letters converted to uppercase.
- **Thinking Method:** The function checks each character of the string and converts it to uppercase if it is a lowercase letter.

8. toLowerCase

- **Objective:** To convert a string to lowercase.
- **Inputs:** A string str.
- **Outputs:** The string with all uppercase letters converted to lowercase.
- **Thinking Method:** The function checks each character of the string and converts it to lowercase if it is an uppercase letter.

Intermediate String Functions

1. isPalindrome

- **Objective:** To check if a string is a palindrome (reads the same forwards and backwards).
- **Inputs:** A string str.
- **Outputs**: true if the string is a palindrome, false otherwise.
- **Thinking Method:** The function first creates a copy of the original string, reverses the copy, and then compares the original string with the reversed one.

2. countVowelsConsonants

- **Objective:** To count the number of vowels and consonants in a string.
- **Inputs:** A string str, two integer pointers vowels and consonants to store the results.
- **Outputs:** Updates the pointers with the counts of vowels and consonants.
- **Thinking Method:** The function iterates over the string, checking if each character is a vowel or consonant. It counts and updates the respective variable.

3. findSubstring

- **Objective:** To find the position of a substring within a string.
- **Inputs**: A string str and a substring sub.
- **Outputs:** The index of the first occurrence of sub in str, or -1 if not found.
- **Thinking Method:** The function searches for the substring by iterating through the main string and comparing each segment with the substring.

4. removeWhitespaces

- **Objective:** To remove all whitespace characters from a string.
- **Inputs:** A string str.
- **Outputs:** The string with all spaces removed.
- **Thinking Method:** The function loops through the string, shifting non-space characters towards the beginning of the string and null-terminating the string at the new end.

5. countWords

- **Objective:** To count the number of words in a string.
- **Inputs:** A string str.
- **Outputs:** The number of words in the string.
- **Thinking Method:** The function checks each character in the string and counts spaces as word boundaries.

6. removeDuplicates

- **Objective:** To remove duplicate characters from a string.
- **Inputs:** A string str.
- **Outputs:** The string with duplicate characters removed.
- **Thinking Method:** The function checks each character and compares it with subsequent characters, removing any duplicate characters by shifting the remaining characters left.

Advanced String Functions Report

1. String Compression (Run-Length Encoding)

- **Objective:** Compress a string using Run-Length Encoding (RLE), where consecutive occurrences of the same character are replaced by the character followed by its frequency.
- **Inputs**: str (char*): The input string to be compressed.
- **Outputs:** result (char*): The compressed string in RLE format.
- **Thinking Method:** Iterate over the string, count consecutive occurrences of characters, and append each character followed by its count to the result string.

2. Find Longest Word

- **Objective**: Identify the longest word in a given sentence.
- **Inputs:** str (char*): The input sentence string.
- **Outputs**: result (char*): The longest word in the sentence.
- **Thinking Method:** Process the sentence by iterating through each word, comparing their lengths, and storing the longest word found .

3. String Rotation Check

- **Objective:** Check if one string is a rotation of another string.
- **Inputs**: str1 (char*): The first string. str2 (char*): The second string to check if it is a rotation of the first.
- **Outputs:** bool: Returns true if str2 is a rotation of str1, otherwise returns false.
- **Thinking Method**: Check if both strings have the same length, then concatenate str1 with itself and check if str2 is a substring of this concatenated string.

4. Count Specific Character

- Objective: Count how many times a specific character appears in a string.
- Inputs: str (char*): The input string to search. ch (char): The character to count.
- Outputs: int: The number of occurrences of the specified character in the string.
- Thinking Method: Iterate through the string and increment a counter each time the specified character is found.

5. Find and Replace

- **Objective**: Replace all occurrences of a specified substring with another substring.
- **Inputs:** str (char*): The input string. find (char*): The substring to find. replace (char*): The substring to replace with.
- **Outputs:** void: The modified string with all occurrences of find replaced by replace.
- **Thinking Method:** Iterate through the string, search for the find substring, and replace it with replace until all occurrences are replaced.

6. Longest Palindromic Substring

- **Objective:** Find the longest palindromic substring in a given string.
- **Inputs**: str (char*): The input string to check for palindromic substrings.
- **Outputs**: result (char*): The longest palindromic substring.
- **Thinking Method**: Iterate through the string, checking for palindromes centered at each character, and store the longest palindrome found.

7. String Permutations

- **Objective**: Generate and print all possible permutations of a given string.
- **Inputs:** str (char*): The input string to generate permutations for.
- **Outputs:** void: Prints all possible permutations of the string.
- **Thinking Method:** Use recursion and backtracking to generate permutations by swapping each character in the string and printing the permutations as they are generated.

8. Split String

- **Objective:** Split a string into tokens based on a specified delimiter.
- **Inputs:** str (char*): The input string to split. delimiter (char): The character that will be used as the delimiter.
- **Outputs:** tokens (char[ ][100]): An array of strings where each string is a token from the input string. tokenCount (int*): The number of tokens found.
- **Thinking Method:** Iterate through the string, and whenever the delimiter is encountered, extract the substring and store it in the tokens array. Count the number of tokens as you go.

Operations on arrays

Basic Arrays Functions Report

1.Initialize Array

- **Objective:** Initialize an array with a specified value.
- **Inputs:** arr[] (int[]): The array to initialize. size (int): The size of the array. value (int): The value to fill the array with.
- **Outputs**: void: The array is modified to have the specified value in each of its elements.
- **Thinking Method:** Iterate through the array and set each element to the given value.

2. Print Array

- **Objective**: Print the elements of an array.
- **Inputs**: arr[] (int[]): The array to print. size (int): The size of the array.
- **Outputs:** void: Prints the array elements to the console.
- **Thinking Method**: Loop through the array and print each element.

3. Find Maximum

- **Objective**: Find the maximum value in an array.
- **Inputs:** arr[] (int[]): The array to search. size (int): The size of the array.
- **Outputs:** int: The maximum value in the array.
- **Thinking Method:** Iterate through the array and keep track of the largest element found.

4. Find Minimum

- **Objective:** Find the minimum value in an array.
- **Inputs**: arr[] (int[]): The array to search. size (int): The size of the array.
- **Outputs:** int: The minimum value in the array.
- **Thinking Method:** Iterate through the array and keep track of the smallest element found.

5. Calculate Sum

- **Objective:** Calculate the sum of all elements in an array.
- **Inputs**: arr[] (int[]): The array to sum. size (int): The size of the array.
- **Outputs:** int: The sum of the elements in the array.
- **Thinking Method:** Iterate through the array, adding each element to a sum variable.

6. Calculate Average

- **Objective:** Calculate the average of the elements in an array.
- **Inputs**: arr[] (int[]): The array to calculate the average. size (int): The size of the array.
- **Outputs:** double: The average of the elements in the array.
- **Thinking Method:** Use the sum of the array elements (calculated using sumArray) and divide it by the size of the array.

7. Check if Sorted

- **Objective:** Check if an array is sorted in ascending order.
- **Inputs:** arr[] (int[]): The array to check. size (int): The size of the array.
- **Outputs:** bool: Returns true if the array is sorted, otherwise returns false.
- **Thinking Method:** Iterate through the array and check if each element is less than or equal to the next element.

Intermediate Array Functions

1. Reverse Array

- **Objective:** Reverse the elements of an array.
- **Inputs:** arr[] (int[]): The array to reverse. size (int): The size of the array.
- **Outputs:** void: The array is modified to be in reversed order.
- **Thinking Method:** Swap the first and last elements, then move inward, swapping pairs of elements until reaching the center of the array.

2. Count Even and Odd Numbers

- **Objective**: Count the number of even and odd numbers in an array.
- **Inputs**: arr[] (int[]): The array to check. size (int): The size of the array.
- **Outputs:** void: The counts of even and odd numbers are returned through pointers.
- Thinking Method: Iterate through the array and check if each element is even or odd, updating the respective counters.

3. Find Second Largest

- **Objective:** Find the second largest element in an array.
- **Inputs**: arr[] (int[]): The array to search. size (int): The size of the array.
- **Outputs**: int: The second largest value in the array.
- **Thinking Method**: Remove the minimum element from the array and then find the maximum value in the remaining elements.

4. Find Frequency of Elements

- **Objective:** Find the frequency of each element in an array.
- **Inputs:** arr[] (int[]): The array to check. size (int): The size of the array.
- **Outputs:** void: Prints the frequency of each element in the array.
- **Thinking Method:** Use a nested loop to compare each element to every other element and count how many times each element occurs.

5. Remove Duplicates

- **Objective**: Remove duplicate elements from an array.
- **Inputs:** arr[ ] (int[ ]): The array to modify. size (int): The size of the array.
- **Outputs:** int: The new size of the array after duplicates are removed.
- **Thinking Method:** Use nested loops to find duplicates, shift the elements to remove duplicates, and decrease the size of the array.

6. Binary Search

- **Objective:** Perform a binary search to find a target value in a sorted array.
- **Inputs:** arr[ ] (int[ ]): The sorted array. size (int): The size of the array. target (int): The value to search for.
- **Outputs:** int: The index of the target value, or -1 if not found.
- **Thinking Method:** Repeatedly divide the array into halves, comparing the middle element to the target and narrowing the search range.

7. Linear Search

- **Objective:** Perform a linear search to find a target value in an array.
- **Inputs:** arr[ ] (int[ ]): The array to search. size (int): The size of the array. target (int): The value to search for.
- **Outputs:** int: The index of the target value, or -1 if not found.
- **Thinking Method:** Iterate through the array and compare each element to the target until a match is found.

8. Left Shift Array

- **Objective:** Shift the elements of an array to the left by a specified number of rotations.
- **Inputs:** arr[ ] (int[ ]): The array to shift. size (int): The size of the array. rotations (int): The number of left shifts to perform.
- **Outputs:** void: The array is modified after the shifts.
- **Thinking Method:** Shift all elements to the left by one position at a time and wrap around the first element to the end.

9. Right Shift Array

- **Objective:** Shift the elements of an array to the right by a specified number of rotations.
- **Inputs:** arr[ ] (int[ ]): The array to shift. size (int): The size of the array. rotations (int): The number of right shifts to perform.
- **Outputs:** void: The array is modified after the shifts.
- **Thinking Method:** Shift all elements to the right by one position at a time and wrap around the last element to the front.

Sorting Algorithms

1. Bubble Sort

- **Objective:** Sort an array in ascending order using the bubble sort algorithm.
- **Inputs:** arr[] (int[]): The array to sort. size (int): The size of the array.
- **Outputs:** void: The array is sorted in ascending order.
- **Thinking Method:** Repeatedly compare adjacent elements and swap them if they are in the wrong order. This process continues until no more swaps are needed, indicating that the array is sorted.

2. Selection Sort

- **Objective:** Sort an array in ascending order using the selection sort algorithm.
- **Inputs:** arr[] (int[]): The array to sort. size (int): The size of the array.
- **Outputs:** void: The array is sorted in ascending order.
- **Thinking Method:** For each element in the array, find the smallest element from the unsorted portion and swap it with the current element. This reduces the unsorted portion by one each time.

3. Insertion Sort

- **Objective:** Sort an array in ascending order using the insertion sort algorithm.
- **Inputs:** arr[] (int[]): The array to sort. size (int): The size of the array.
- **Outputs:** void: The array is sorted in ascending order.
- **Thinking Method:** Iteratively build a sorted portion of the array by inserting each element from the unsorted portion into its correct position within the sorted portion.

4. Merge Sort

- **Objective:** Sort an array in ascending order using the merge sort algorithm.
- **Inputs:** arr[] (int[]): The array to sort. left (int): The starting index of the array. right (int): The ending index of the array.
- **Outputs:** void: The array is sorted in ascending order.
- **Thinking Method:** Recursively divide the array into two halves until each half has one element, then merge the halves back together in sorted order. The merge operation ensures that the elements are sorted during the merging process.

5. Quick Sort

- **Objective:** Sort an array in ascending order using the quick sort algorithm.
- **Inputs:** arr[] (int[]): The array to sort. low (int): The starting index of the array. high (int): The ending index of the array.
- **Outputs:** void: The array is sorted in ascending order.
- **Thinking Method:** Select a pivot element from the array and partition the array into two sub-arrays: elements less than the pivot and elements greater than the pivot. Recursively apply the same process to each sub-array.

Advanced Array Functions

1. Find Missing Number

- **Objective:** Find the missing number in an array of size n-1, where the array contains numbers from 1 to n.
- **Inputs:** arr[] (int[]): The array of numbers. size (int): The size of the array (should be n-1).
- **Outputs:** int: The missing number in the sequence.
- **Thinking Method:** The sum of the first n natural numbers is given by the formula n(n+1)/2. The sum of the elements in the given array is subtracted from this sum to find the missing number.

2. Find Pairs with Given Sum

- **Objective:** Find all pairs of elements in an array whose sum is equal to a given value.
- **Inputs:** arr[] (int[]): The array of numbers. size (int): The size of the array. sum (int): The target sum for the pairs.
- **Outputs:** void: Prints pairs of numbers whose sum equals the target sum.
- **Thinking Method:** Iterate over the array and, for each element, search for another element that, when added, equals the given sum. This can be done using a nested loop or using a hash table for better performance.

3. Subarray with Given Sum

- **Objective:** Find a continuous subarray within an array that adds up to a given sum.
- **Inputs:** arr[] (int[]): The array of numbers. size (int): The size of the array. sum (int): The target sum for the subarray.
- **Outputs:** void: Prints the subarray whose elements sum to the target sum.
- **Thinking Method:** Use the sliding window technique. Start with an empty window, expand the window to include elements, and adjust the window size when the sum exceeds the target.

4. Rearrange Positive and Negative Numbers

- **Objective**: Rearrange an array such that positive and negative numbers alternate.
- **Inputs**: arr[] (int[]): The array of numbers. size (int): The size of the array.
- **Outputs:** void: The array is rearranged with alternating positive and negative numbers.
- **Thinking Method:** Partition the array into positive and negative numbers. Then, rearrange the elements so that positive and negative numbers are alternated, ensuring the ordering of the original elements is maintained.

5. Find Majority Element

- **Objective:** Find the majority element in an array, an element that appears more than n/2 times.
- **Inputs:** arr[] (int[]): The array of numbers. size (int): The size of the array.
- **Outputs**: int: The majority element in the array.
- **Thinking Method:** Use the Boyer-Moore Voting Algorithm, which efficiently finds the majority element by iterating through the array once and maintaining a candidate element.

6. Longest Increasing Subsequence

- **Objective:** Find the length of the longest increasing subsequence in an array.
- **Inputs:** arr[] (int[]): The array of numbers. size (int): The size of the array.
- **Outputs:** int: The length of the longest increasing subsequence.
- **Thinking Method:** Use dynamic programming to store the length of the longest increasing subsequence that ends at each index. Update the subsequence length based on previously computed values.

7. Find Duplicates

- **Objective:** Identify duplicate elements in an array.
- **Inputs:** arr[] (int[]): The array of numbers. size (int): The size of the array.
- **Outputs:** void: Prints the duplicate elements in the array.
- **Thinking Method:** Use a hash table to track the occurrences of each element. If an element appears more than once, it is a duplicate.

8. Find Intersection of Two Arrays

- **Objective:** Find the common elements between two arrays.
- **Inputs:** arr1[] (int[]): The first array of numbers. size1 (int): The size of the first array. arr2[] (int[]): The second array of numbers. size2 (int): The size of the second array.
- **Outputs:** void: Prints the common elements between the two arrays.
- **Thinking Method:** Use a hash table to store elements of one array and check if elements from the other array exist in the hash table. This provides an efficient way to find the intersection.

9. Find Union of Two Arrays

- **Objective:** Find the union of two arrays, which contains all the distinct elements from both arrays.
- **Inputs:** arr1[] (int[]): The first array of numbers. size1 (int): The size of the first array. arr2[] (int[]): The second array of numbers. size2 (int): The size of the second array.
- **Outputs:** void: Prints the union of the two arrays.
- **Thinking Method:** Use a hash table to store elements from both arrays, ensuring that duplicates are eliminated. The hash table can then be used to print the distinct union of the two arrays.

Operations on matrices

Basic Matrix Functions

1. Initialize Matrix:

- **Objective:** Sets all elements of a matrix to a specified value.
- **Inputs:**
  - rows: Number of rows in the matrix.
  - cols: Number of columns in the matrix.
  - matrix[rows][cols]: The matrix to initialize.
  - value: The value to assign to each matrix element.
- **Outputs:** The matrix is updated with the specified value in all positions.
- **Thinking Method:** Use nested loops to iterate through all matrix elements and assign the value.

2. Print Matrix:

- **Objective:** Displays the matrix in a structured and formatted way.
- **Inputs:**
  - rows: Number of rows in the matrix.
  - cols: Number of columns in the matrix.
  - matrix[rows][cols]: The matrix to print.
- **Outputs:** Matrix elements printed row by row.
- **Thinking Method**: Use nested loops to traverse rows and columns, printing elements in a structured grid.

3. Input Matrix:

- **Objective:** Allows the user to input the elements of the matrix.
- **Inputs:**
  - rows: Number of rows in the matrix.
  - cols: Number of columns in the matrix.
  - matrix[rows][cols]: The matrix to populate.
- **Outputs:** Matrix is populated with user-provided values.
- **Thinking Method:** Use nested loops to prompt and accept input for each matrix element.

Matrix Arithmetic

1. Matrix Addition:

- **Objective**: Performs element-wise addition of two matrices.
- **Inputs:**
  - rows, cols: Dimensions of the matrices.
  - mat1[rows][cols], mat2[rows][cols]: The two matrices to add.
  - result[rows][cols]: The matrix to store the addition result.
- **Outputs:** The result matrix contains the element-wise sum of mat1 and mat2.
- **Thinking Method:** Use nested loops to iterate over corresponding elements and compute the sum.

2. Matrix Subtraction:

- **Objective**: Subtracts the second matrix from the first matrix element-wise.
- **Inputs:**
  - rows, cols: Dimensions of the matrices.
  - mat1[rows][cols], mat2[rows][cols]: The matrices to subtract.
  - result[rows][cols]: The matrix to store the subtraction result.
- **Outputs:** The result matrix contains the element-wise difference of mat1 and mat2.
- Thinking Method: Use nested loops to subtract corresponding elements.

3. Matrix Multiplication:

- **Objective**: Multiplies two matrices and stores the result in a third matrix.
- **Inputs:**
  - rows1, cols1: Dimensions of the first matrix.
  - rows2, cols2: Dimensions of the second matrix.
  - mat1[rows1][cols1], mat2[rows2][cols2]: The matrices to multiply.
  - result[rows1][cols2]: The matrix to store the product.
- **Outputs:** result matrix containing the product of mat1 and mat2.
- **Thinking Method:** Verify cols1 == rows2. For each element in result, calculate the dot product of the corresponding row and column.

4. Scalar Multiplication:

- **Objective**: Multiplies every matrix element by a given scalar.
- **Inputs:**
  - rows, cols: Dimensions of the matrix.
  - matrix[rows][cols]: The matrix to multiply.
  - scalar: The scalar value.
- **Outputs**: matrix is updated with elements multiplied by scalar.
- **Thinking Method:** Use nested loops to traverse the matrix and multiply each element by the scalar.

Matrix Properties and Checks

1. Check if Square Matrix:

- **Objective:** Determines whether the matrix is square.
- **Inputs:**
    - rows, cols: Dimensions of the matrix.
- **Outputs:** true if the matrix is square (rows == cols); otherwise, false.
- **Thinking Method:** Compare rows and cols.

2. Check if Identity Matrix:

- **Objective**: Checks if the matrix is an identity matrix.
- **Inputs:**
    - size: Dimensions of the square matrix.
    - matrix[size][size]: The matrix to check.
- **Outputs:** true if all diagonal elements are 1 and all others are 0; otherwise, false.
- **Thinking Method:** Use nested loops to verify diagonal and off-diagonal elements.

3. Check if Diagonal Matrix:

- **Objective**: Determines if all non-diagonal elements of the matrix are zero.
- **Inputs:**
    - size: Dimensions of the square matrix.
    - matrix[size][size]: The matrix to check.
- **Outputs:** true if all non-diagonal elements are zero; otherwise, false.
- **Thinking Method:** Use nested loops to check every element, ensuring off-diagonal elements are 0.

4. Check if Symmetric Matrix

- **Objective**: Check if the matrix is symmetric (i.e., matrix[i][j]=matrix[j][i]matrix[i][j] = matrix[j][i]matrix[i][j]=matrix[j][i]).
- **Inputs:**
    - int size: The size of the matrix (number of rows and columns).
    - int matrix[size][size]: The matrix to check.
- **Outputs:** bool: Returns true if the matrix is symmetric, otherwise false.
- **Thinking Method:** Compare each element matrix[i][j]matrix[i][j]matrix[i][j] with matrix[j][i]matrix[j][i]matrix[j][i], and return false if any pair is unequal, otherwise return true.

5. Check if Upper Triangular Matrix:

- **Objective**: Determines if all elements below the main diagonal are zero.
- **Inputs:**
    - size: Dimensions of the square matrix.
    - matrix[size][size]: The matrix to evaluate.
- **Outputs**: Returns true if the matrix is upper triangular; otherwise, false.
- **Thinking Method:** Use nested loops to verify all elements below the diagonal (i > j) are zero.

Matrix Operations

1. Transpose Matrix:

- **Objective:** Computes the transpose of a matrix.
- **Inputs:**
  - rows, cols: Dimensions of the matrix.
  - matrix[rows][cols]: The matrix to transpose.
  - result[cols][rows]: The matrix to store the transposed values.
- **Outputs:** The result matrix contains the transpose of matrix.
- **Thinking Method:** Swap rows and columns while assigning values.

2. Determinant of a Matrix:

- **Objective**: Calculates the determinant of a square matrix.
- **Inputs:**
  - size: Dimensions of the square matrix.
  - matrix[size][size]: The matrix whose determinant to compute.
- **Outputs**: The determinant value.
- **Thinking Method:** Use recursive expansion or Gaussian elimination.

3. Inverse of a Matrix:

- **Objective**: Computes the inverse of a square matrix using Gaussian elimination.
- **Inputs:**
  - size: Dimensions of the square matrix.
  - matrix[size][size]: The matrix to invert.
  - result[size][size]: The matrix to store the inverse.
- **Outputs:** The result matrix contains the inverse of matrix.
- **Thinking Method:** Form an augmented matrix and reduce it to row-echelon form.

4. Matrix Power:

- **Objective**: Raises a square matrix to a given power.
- **Inputs:**
  - size: Dimensions of the square matrix.
  - matrix[size][size]: The matrix to raise to a power.
  - power: The exponent to apply.
  - result[size][size]: The matrix to store the result.
- **Outputs:** The result matrix contains matrix raised to power.
- **Thinking Method:** Use repeated multiplication and identity matrix initialization for power 0.

Advanced Matrix Functions

1. Cofactor Matrix:

- **Objective:** Computes the cofactor matrix of a square matrix.
- **Inputs:**
  - size: Dimensions of the square matrix.
  - matrix[size][size]: The matrix to compute the cofactor for.
  - cofactor[size][size]: The matrix to store the cofactor values.
- **Outputs:** The cofactor matrix.
- **Thinking Method:** For each element, compute the determinant of the minor matrix, applying alternating signs.

2. Adjoint Matrix:

- **Objective:** Computes the adjoint (transpose of the cofactor matrix) of a square matrix.
- **Inputs:**
  - size: Dimensions of the square matrix.
  - matrix[size][size]: The matrix to compute the adjoint for.
  - adjoint[size][size]: The matrix to store the adjoint.
- **Outputs:** The adjoint matrix.
- **Thinking Method:** Compute the cofactor matrix and then transpose it.

3. LU Decomposition:

- **Objective**: Decomposes a square matrix into lower (L) and upper (U) triangular matrices.
- **Inputs:**
  - size: Dimensions of the square matrix.
  - matrix[size][size]: The matrix to decompose.
  - lower[size][size]: The matrix to store lower triangular values.
  - upper[size][size]: The matrix to store upper triangular values.
- **Outputs:** The lower and upper matrices.
- **Thinking Method:** Use Doolittle's or Crout's algorithm for decomposition.

Special Matrix Operations

1. Find Trace of a Matrix:

- **Objective:** Computes the sum of diagonal elements in a square matrix.
- **Inputs:**
    - size: Dimensions of the square matrix.
    - matrix[size][size]: The matrix to compute the trace for.
- **Outputs**: The trace value.
- **Thinking Method:** Sum the elements where row and column indices are equal.

2. Rotate Matrix 90 Degrees:

- **Objective**: Rotates a square matrix by 90 degrees clockwise.
- **Inputs:**
    - size: Dimensions of the square matrix.
    - matrix[size][size]: The matrix to rotate.
- **Outputs:** The rotated matrix.
- **Thinking Method:** Transpose the matrix and reverse rows.

Optional parts

1. Caesar Cipher

- **Objective:** Shifts each letter by a fixed number of positions in the alphabet.
- **Inputs:**
  - char *text: The string to be encrypted or decrypted.
  - int shift: The number of positions to shift each letter.
- **Outputs:** The modified text.
- **Thinking Method:**
  a. Check each character in the text and leave non-alphabetic characters unchanged.
  b. Determine whether it's uppercase or lowercase.
  c. Calculate the new position using the shift and handle negative values.
  d. Adjust for ASCII values to get the encrypted/decrypted letter.

2. Substitution Cipher

- **Objective**: Replaces each letter with a corresponding letter from a substitution alphabet.
- **Inputs:**
  - char *text: The string to be encrypted or decrypted.
  - const char *key: The substitution alphabet.
- **Outputs:** The modified text.
- **Thinking Method:**
  a. Validate the substitution key.
  b. Create mappings for encryption and decryption.
  c. Replace each letter in the text using the mappings.
  d. Leave non-alphabetic characters unchanged.

3. XOR Cipher

- **Objective**: Applies a bitwise XOR operation between the message and a key.
- **Inputs:**
  - char *text: The string to be encrypted or decrypted.
  - char key: The XOR key.
- **Outputs**: The modified text.
- **Thinking Method:**
  a. For each character in text, XOR it with the key.
  b. Convert the result back to a character.
  c. Handle cyclic repetition if the key is longer than one character.

4. Vigenere Cipher

- **Objective:** Uses a keyword to variably shift each letter.
- **Inputs:**
  - char *text: The string to be encrypted or decrypted.
  - const char *key: The keyword used for encryption or decryption.
  - int encrypt: A flag (1 for encryption, 0 for decryption).
- **Outputs**: The modified text.
- **Thinking Method:**
  - Repeat the key cyclically to match the length of text.
  - For encryption, add the keyword shift; for decryption, subtract it.
  - Calculate the new position using modulo arithmetic.
  - Skip non-alphabetic characters and adjust for case sensitivity.

5. Atbash Cipher

- **Objective:** Substitutes each letter with its reverse counterpart in the alphabet.
- **Inputs:**
  - char *text: The string to be encrypted or decrypted.
- **Outputs:** The modified text.
- **Thinking Method:**
  - For uppercase letters, calculate the reverse counterpart.
  - For lowercase letters, calculate the reverse counterpart.
  - Leave non-alphabetic characters unchanged.

6. Rail Fence Cipher

- **Objective:** Arranges text in a zigzag pattern and reads it row by row.
- **Inputs:**
  - const char *text: The string to be encrypted or decrypted.
  - char *result: The resulting string after encryption or decryption.
  - int depth: The number of rows in the zigzag pattern.
- **Outputs:** The modified text.
- **Thinking Method:**
  - Create a 2D grid to represent the zigzag pattern.
  - Populate the grid in a zigzag manner.
  - For encryption, read the grid row by row.
  - For decryption, mark and fill the zigzag pattern and reconstruct the original text.

Conclusion

In conclusion , this project , which will be presented on **January 19, 2025**, marks a significant milestone in my programming journey.

The development of the library, written in C, involved a deep dive into a variety of computational problems, including operations on numbers, strings, arrays, and matrices.

From **the second week of December 2024**, I began with number operations, which laid the groundwork for the library . **The third week** presented a personal challenge with string operations. Since this topic wasn't covered in lectures, I had to independently dive into learning and applying them. The skills acquired during this time were critical for the project's success. **By the final week of December 2024**, I had mastered array operations and successfully incorporated them into the library.The matrices operations, along with the optional parts, were completed in **Junuary 2025**, adding depth and complexity to the library with advanced functions like matrix inversion, determinant calculation, and LU decomposition.

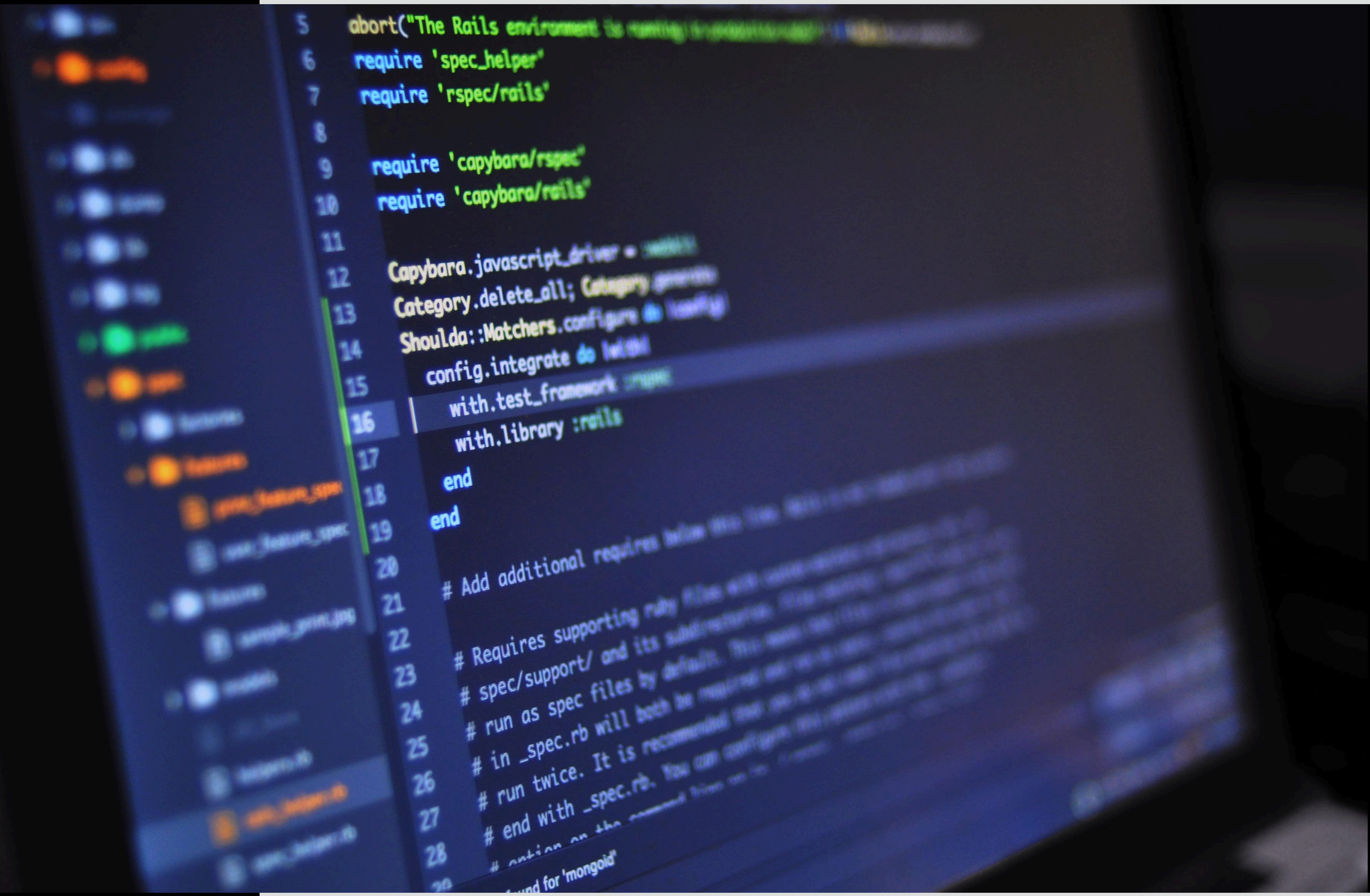
The entire report was written piece by piece after completing each section, documenting the library's development in detail. This project, which will be shared with professors on **GitHub,** is not only a reflection of my growth as a programmer but also a showcase of problem-solving, self-learning, and the application of theoretical knowledge to practical programming challenges.

I hope to share the knowledge and experience gained, while also receiving valuable feedback from professors and peers. I am confident that the work done will be beneficial not only for my academic growth but also for future endeavors in the field of programming.

Thank you for taking the time to explore this project.

END
SEMESTER
PROJECT

NSCS
المدرسة الوطنية العليا في الأمن السيبراني
NATIONAL SCHOOL OF CYBERSECURITY



Thank you !

For further information regarding this project, please contact me :

My Contact

f.hassani@enscs.edu.dz
r.haddadi@enscs.edu.dz