NSCS

المدرسة الوطنية العليا في الأمن السيبراني

NATIONAL SCHOOL OF CYBERSECURITY

END-SEMESTER PROJECT

Prepared by :

Kassoul Mohammed Ali
Ghribi Abdennour

☎ 06-96-60-92-79

✉ m.kassoul@enscs.edu.dz
a.ghribi@enscs.edu.dz

NSCS

REPORT

of the aalgorithm's project for the end of the first semester

TO	Soualmi Abdallah
FROM	Kassoul Mohammed Ali Ghribi Abdennour
DATE	19/01/2025
SUBJECT	A detailed report

1. Project Objectives and Problem Statement:

The goal of this project is to design and implement a comprehensive C library that performs a wide variety of operations on numbers, strings, arrays, and matrices. This initiative aims to strengthen students' skills in algorithmic thinking and modular programming using the C language. By building these functionalities, students will gain practical experience in creating reusable code and solving complex data processing tasks.

Problem Statement:

Data processing tasks often require the use of specialized algorithms for numerical analysis, string manipulations, array handling, and matrix computations. To address these needs, the project proposes developing a structured library that consolidates these functionalities into a cohesive and efficient framework. This library will serve as a resource for learning and applying programming concepts in real-world scenarios.

2. Analysis and Algorithm Design for Each Function:

Operations on Numbers:

- **Sum of Digits (int sumOfDigits(int num)):**
    - Analysis: Extract each digit using modulus and division, then sum them.
    - **Algorithm:**
        - Initialize sum = 0.
        - While num > 0, extract the last digit (digit = num % 10) and add it to sum.
        - Update num = num / 10.
        - Return sum.
- **Reverse Number (int reverseNumber(int num)):**
    - **Analysis**: Reverse the digits by iteratively building a new number.
    - **Algorithm:**
        - Initialize reverse = 0.
        - While num > 0, extract the last digit (digit = num % 10) and append it to reverse.
        - Update num = num / 10.
        - Return reverse.
- **Palindrome (bool isPalindrome(int num)):**
    - **Analysis:** Check if a number reads the same forward and backward.
    - **Algorithm:**
        - Store the original number.
        - Reverse the number using reverseNumber.
        - Compare the reversed number with the original.
        - Return true if they match, false otherwise.
- **Prime (bool isPrime(int num)):**
    - **Analysis:** Check if a number has no divisors other than 1 and itself.
    - **Algorithm:**
        - Return false for numbers less than 2.
        - Loop from 2 to the square root of the number.
        - If num % i == 0, return false.
        - Otherwise, return true.
- **Greatest Common Divisor (int gcd(int a, int b)):**
    - **Analysis:** Use the Euclidean algorithm to find the GCD.
    - **Algorithm:**
        - While b != 0, set a = b and b = a % b.
        - Return a.
- **Least Common Multiple (int lcm(int a, int b)):**
    - **Analysis:** Compute LCM using the formula lcm = (a * b) / gcd(a, b).
    - **Algorithm:**
        - Calculate the GCD of a and b.
        - Compute and return (a * b) / gcd.

- **Factorial (long factorial(int num)):**
  - **Analysis:** Multiply all integers from 1 to num.
  - **Algorithm:**
    - Initialize result = 1.
    - Loop from 1 to num, multiplying result by the loop index.
    - Return result.
- **Even/Odd (bool isEven(int num)):**
  - **Analysis:** Check if a number is divisible by 2.
  - **Algorithm:**
    - Return true if num % 2 == 0, otherwise return false.
- **Prime Factorization (void primeFactors(int num)):**
  - **Analysis:** Find and print all prime factors of a number.
  - **Algorithm:**
    - Divide num by 2 while it's even and print 2.
    - For odd numbers, divide num by i while num % i == 0 and print i.
    - Increment i by 2 and repeat.
    - Stop when i * i > num and print num if it's greater than 1.
- **Armstrong Number (bool isArmstrong(int num)):**
  - **Analysis:** Check if a number equals the sum of its digits raised to the power of the number of digits.
  - **Algorithm:**
    - Count the digits of num.
    - Compute the sum of each digit raised to the power of the digit count.
    - Return true if the sum equals num.
- **Fibonacci Sequence (void fibonacciSeries(int n)):**
  - **Analysis:** Generate the Fibonacci sequence up to the nth term.
  - **Algorithm:**
    - Initialize a = 0 and b = 1.
    - Print a and b.
    - Loop n-2 times, updating c = a + b, a = b, and b = c.
- **Sum of Divisors (int sumDivisors(int num)):**
  - **Analysis:** Calculate the sum of all divisors of a number.
  - **Algorithm:**
    - Initialize sum = 0.
    - Loop from 1 to num / 2. If num % i == 0, add i to sum.
    - Add num to sum.
- **Perfect Number (bool isPerfect(int num)):**
  - **Analysis:** Check if the sum of divisors (excluding itself) equals the number.
  - **Algorithm:**
    - Use sumDivisors(num) - num.
    - Return true if the result equals num.

Advanced Operations on Numbers

- **Binary Conversion (void toBinary(int num)):**
  - **Analysis:** Convert a number to its binary representation.
  - **Algorithm:**
    - Use a loop to print remainders of num % 2.
- **Narcissistic Number (bool isNarcissistic(int num)**
  - **Analysis**: Checks if a number equals the sum of its digits raised to the power of the number of digits.
  - **Algorithm**:
    - The same as armstrong number
- **Square Root Calculation (double sqrtApprox(int num)**
  - **Analysis**: Approximates the square root of a number using the Babylonian method.
  - **Algorithm**:
    - Start with an initial guess.
    - Iteratively improve the guess using the formula: (guess+num/guess)/2(guess + num/guess) / 2.
    - Stop when the difference between successive guesses is below a set threshold.
- **Exponentiation (double power(int base, int exp))**
  - **Analysis**: Computes the result of raising a base to a given exponent.
  - **Algorithm**:
    - Use a loop or recursion to multiply the base by itself exp times.
    - Return the result.
- **Happy Number (bool isHappy(int num)**
  - **Analysis**: Determines if the sum of the squares of a number's digits eventually equals 1.
  - **Algorithm**:
    - Calculate the sum of the squares of the digits.
    - Repeat the process until the number becomes 1 or enters a loop.
    - Return true if the result is 1; otherwise, return false.
- **Abundant Number (bool isAbundant(int num)**
  - **Analysis**: Checks if the sum of a number's divisors (excluding itself) is greater than the number.
  - **Algorithm**:
    - Calculate sumDivisors(num) - num.
    - Return true if the result is greater than the number.
- **Deficient Number (bool isDeficient(int num)**
  - **Analysis**: Determines if the sum of a number's divisors (excluding itself) is less than the number.
  - **Algorithm**:
    - Calculate sumDivisors(num) - num.
    - Return true if the result is less than the number.

- **Sum of Fibonacci Even Numbers (int sumEvenFibonacci(int n)**
  - **Analysis**: Calculates the sum of even terms in the Fibonacci sequence up to the nth term.
  - **Algorithm**:
    - Generate Fibonacci terms iteratively.
    - Check if each term is even.
    - Add even terms to a cumulative sum.
- **Harshad Number (bool isHarshad(int num)**
  - **Analysis**: Checks if a number is divisible by the sum of its digits.
  - **Algorithm**:
    - Compute the sum of the digits.
    - Return true if num modulo this sum is 0.
- **Catalan Number Calculation (unsigned long catalanNumber(int n)**
  - **Analysis**: Computes the nth Catalan number, important in combinatorics.
  - **Algorithm**:
    - Use the formula: Cn=(2n)!(n+1)!⋅n!C_n = \frac{(2n)!}{(n+1)! \cdot n!}.
    - Perform factorial calculations iteratively or recursively.
- **Pascal Triangle (void pascalTriangle(int n)**
  - **Analysis**: Generates the first n rows of Pascal’s Triangle, representing binomial coefficients.
  - **Algorithm**:
    - using the binomial coefficient ***x+1 = x* (i - j) / (j + 1);***
    - Print elements row by row.
- **Bell Number (unsigned long bellNumber(int n)**
  - **Analysis**: Calculates the nth Bell number, representing the number of ways to partition a set.
  - **Algorithm**:
    - Use an array where each element is the sum of the previous row's elements.
    - when we finish with a row we do not need it next so we type above it the current row
    - Return the last element of the nth row.
- **Kaprekar Number (bool isKaprekar(int num)**
  - **Analysis**: Checks if the square of the number can be split into two parts that sum to the original number.
  - **Algorithm**:
    - Compute the square of the number.
    - Split the square into two parts and sum them.
    - Return true if the sum equals the original number.
- **Smith Number (bool isSmith(int num)**
  - **Analysis**: Checks if the sum of a number’s digits equals the sum of the digits of its prime factors.
  - **Algorithm**:
    - Factorize the number into primes.
    - Calculate the sum of the digits of the number and its factors.
    - Return true if the sums are equal.

- **Sum of Prime Numbers (int sumOfPrimes(int n)**
    - **Analysis**: Computes the sum of all prime numbers up to a given number.
    - **Algorithm**:
        - Iterate through numbers from 2 to n.
        - Check if each number is prime.
        - Add prime numbers to a cumulative sum

Analysis of String Operations

Basic String Functions

1. **Calculate String Length (int stringLength(char* str)**
    - **Analysis**: Returns the length of the string by counting characters until the null terminator is reached.
    - **Algorithm**:
        - Initialize a counter to 0.
        - Traverse the string character by character, incrementing the counter.
        - Stop when the null terminator (\0) is encountered.
        - Return the counter.
2. **Copy String (void stringCopy(char* dest, const char* src)**
    - **Analysis**: Copies the contents of the source string into the destination string.
    - **Algorithm**:
        - Traverse the source string.
        - Copy each character from src to dest.
        - Append a null terminator (\0) to dest after copying the last character.
3. **Concatenate Strings (void stringConcat(char* dest, const char* src)**
    - **Analysis**: Appends the source string to the end of the destination string.
    - **Algorithm**:
        - Find the null terminator in dest.
        - Traverse the source string and append its characters to the end of dest.
        - Append a null terminator (\0) to the final position in dest.
4. **Compare Strings (int stringCompare(const char* str1, const char* str2)**
    - **Analysis**: Compares two strings lexicographically.
    - **Algorithm**:
        - Compare characters at corresponding positions in both strings.
        - Return a negative value if str1 is less than str2, a positive value if greater, or 0 if equal.

- **Check if Empty (bool isEmpty(char* str)**
  - **Analysis**: Checks if the string contains any characters.
  - **Algorithm**:
    - Return true if the first character of the string is the null terminator (\0), otherwise return false.
- **Reverse a String (void reverseString(char* str)**
  - **Analysis**: Reverses the characters in the string in place.
  - **Algorithm**:
    - Swap the first and last characters, then the second and second-to-last, and so on.
    - Stop when the middle of the string is reached.
- **Convert to Uppercase (void toUpperCase(char* str)**
  - **Analysis**: Converts all lowercase characters in the string to uppercase.
  - **Algorithm**:
    - Traverse the string character by character.
    - For each character, if it is between 'a' and 'z', convert it to its uppercase equivalent.
- **Convert to Lowercase (void toLowerCase(char* str)**
  - **Analysis**: Converts all uppercase characters in the string to lowercase.
  - **Algorithm**:
    - Traverse the string character by character.
    - For each character, if it is between 'A' and 'Z', convert it to its lowercase equivalent.
- **Intermediate String Functions**
- **Palindrome (bool isPalindrome(char* str)**
  - **Analysis**: Checks if the string reads the same forwards and backwards.
  - **Algorithm**:
    - Compare the first and last characters, then the second and second-to-last, and so on.
    - Stop and return false if any mismatch is found; return true if the middle is reached.
- **Count Vowels and Consonants (void countVowelsConsonants(char* str, int* vowels, int* consonants)**
  - **Analysis**: Counts the number of vowels and consonants in the string.
  - **Algorithm**:
    - Traverse the string.
    - Check each character to determine if it is a vowel or consonant.
    - Increment the respective counter.

1. **Find Substring (int findSubstring(const char* str, const char* sub)**
   - **Analysis**: Locates the first occurrence of a substring in a string.
   - **Algorithm**:
     - Traverse the main string and check if the substring matches from the current position.
     - Return the starting index of the substring if found, or -1 otherwise.
2. **Remove Whitespaces (void removeWhitespaces(char* str)**
   - **Analysis**: Removes all spaces from the string.
   - **Algorithm**:
     - Traverse the string.
     - Copy non-whitespace characters to a new position in the string.
3. **Anagram (bool isAnagram(char* str1, char* str2)**
   - **Analysis**: Checks if two strings are anagrams of each other.
   - **Algorithm**:
     - Sort both strings.
     - Compare the sorted strings character by character.
4. **Character Frequency (void charFrequency(char* str, int* freq)**
   - **Analysis**: Calculates the frequency of each character in the string.
   - **Algorithm**:
     - Initialize a frequency array to 0.
     - Increment the frequency of each character’s index as it is encountered.

1. **Count Words (int countWords(char* str)**
   - **Analysis**: Counts the number of words in a string.
   - **Algorithm**:
     - Traverse the string.
     - Increment the word count when transitioning from a whitespace to a non-whitespace character.
2. **Remove Duplicate Characters (void removeDuplicates(char* str)**
   - **Analysis**: Removes duplicate characters from the string.
   - **Algorithm**:
     - Traverse the string.
     - Copy unique characters to a new position in the string.

Advanced String Functions

1. **String Compression (void compressString(char* str, char* result)**
   - **Analysis**: Compresses the string using Run-Length Encoding (RLE).
   - **Algorithm**:
     - Traverse the string.
     - Count consecutive occurrences of each character and store the count alongside the character in the result.
2. **Find Longest Word (void longestWord(char* str, char* result)**
   - **Analysis**: Finds the longest word in a sentence.
   - **Algorithm**:
     - Traverse the string, using spaces or punctuation as delimiters.
     - Track the length of each word and update the result if a longer word is found.
3. **String Rotation Check (bool isRotation(char* str1, char* str2)**
   - **Analysis**: Checks if one string is a rotation of another.
   - **Algorithm**:
     - Concatenate str1 with itself.
     - Check if str2 is a substring of the concatenated result.
4. **Count Specific Character (int countChar(char* str, char ch)**
   - **Analysis**: Counts occurrences of a specific character in the string.
   - **Algorithm**:
     - Traverse the string and increment a counter whenever ch is encountered.
5. **Find and Replace (void findAndReplace(char* str, char* find, char* replace)**
   - **Analysis**: Replaces all occurrences of a substring with another substring.
   - **Algorithm**:
     - Locate occurrences of find in the string.
     - Replace each occurrence with replace while maintaining the original order.
6. **Longest Palindromic Substring (void longestPalindrome(char* str, char* result)**
   - **Analysis**: Identifies the longest palindromic substring in the string.
   - **Algorithm**:
     - Use dynamic programming or expand around each character to find the longest palindrome.

1. **String Permutations (void printPermutations(char* str)**
    - **Analysis**: Generates and prints all permutations of the string.
    - **Algorithm**:
        - Use recursion to swap characters and generate permutations
2. **Split String (void splitString(char* str, char delimiter, char tokens[][100], int* tokenCount)**
    - **Analysis**: Splits a string into multiple tokens based on a specified delimiter, storing each token in an array.
    - **Algorithm**:
        - Traverse the string character by character.
        - Copy characters to a token buffer until the delimiter is encountered.
        - Store the completed token in the tokens array.
        - Reset the buffer and continue until the end of the string.
        - Update tokenCount with the number of tokens.

Analysis of Array Operations

Basic Array Functions

1. **Initialize Array (void initializeArray(int arr[], int size, int value)**
    - **Analysis**: Sets all elements of an array to a specified value.
    - **Algorithm**:
        - Iterate through the array from index 0 to size - 1.
        - Assign value to each element.
2. **Print Array (void printArray(int arr[], int size)**
    - **Analysis**: Outputs all elements of an array in order.
    - **Algorithm**:
        - Traverse the array from index 0 to size - 1.
        - Print each element.
3. **Find Maximum (int findMax(int arr[], int size)**
    - **Analysis**: Identifies the largest element in an array.
    - **Algorithm**:
        - Initialize max with the first element.
        - Traverse the array, comparing each element to max.
        - Update max if a larger element is found.
        - Return max.
4. **Find Minimum (int findMin(int arr[], int size)**
    - **Analysis**: Identifies the smallest element in an array.
    - **Algorithm**:
        - Initialize min with the first element.
        - Traverse the array, comparing each element to min.
        - Update min if a smaller element is found.
        - Return min.

1. **Calculate Sum (int sumArray(int arr[], int size)**
    - **Analysis**: Computes the total sum of elements in an array.
    - **Algorithm**:
        - Initialize sum to 0.
        - Traverse the array and add each element to sum.
        - Return sum.
2. **Calculate Average (double averageArray(int arr[], int size)**
    - **Analysis**: Calculates the average value of array elements.
    - **Algorithm**:
        - Compute the total sum of elements using sumArray.
        - Divide the sum by size and return the result.
3. **Check if Sorted (bool isSorted(int arr[], int size)**
    - **Analysis**: Checks if the elements in the array are in ascending order.
    - **Algorithm**:
        - Traverse the array and compare each pair of adjacent elements.
        - Return false if any pair is out of order; otherwise, return true.

Intermediate Array Functions

1. **Reverse Array (void reverseArray(int arr[], int size)**
    - **Analysis**: Reverses the order of elements in the array.
    - **Algorithm**:
        - Swap the first and last elements, then the second and second-to-last, and so on.
        - Stop when the middle of the array is reached.
2. **Count Even and Odd Numbers (void countEvenOdd(int arr[], int size, int* evenCount, int* oddCount)**
    - **Analysis**: Counts the number of even and odd elements in an array.
    - **Algorithm**:
        - Initialize evenCount and oddCount to 0.
        - Traverse the array and check each element's divisibility by 2.
        - Increment evenCount for even numbers and oddCount for odd numbers.
3. **Find Second Largest (int secondLargest(int arr[], int size)**
    - **Analysis**: Finds the second-largest element in the array.
    - **Algorithm**:
        - Initialize largest and secondLargest to minimum possible values.
        - Traverse the array, updating largest and secondLargest as needed.

1. **Find Frequency of Elements (void elementFrequency(int arr[], int size)**
    - **Analysis**: Calculates the frequency of each unique element in the array.
    - **Algorithm**:
        - Use a hash map or a secondary array to count occurrences of each element.
        - Traverse the array, updating the frequency map for each element.

1. **Remove Duplicates (int removeDuplicates(int arr[], int size)**
    - **Analysis**: Removes duplicate elements and returns the new size of the array.
    - **Algorithm**:
        - Traverse the array, copying unique elements to a new array or modifying the original array in place.
        - Return the size of the new array.
2. **Binary Search (int binarySearch(int arr[], int size, int target)**
    - **Analysis**: Efficiently locates a target element in a sorted array.
    - **Algorithm**:
        - Set low to 0 and high to `size -
        - While low < = high, calculate the middle index mid.
        - If the target is at mid, return the index.
        - If the target is smaller than arr[mid], adjust high to mid - 1 to search the left half.
        - If the target is larger than arr[mid], adjust low to mid + 1 to search the right half.
        - If the target is not found, return -1.

Array Operations

1. **Subarray with Given Sum (void findSubArrayWithSum(int arr[], int size, int sum)**
    - **Analysis**: Finds a continuous subarray that adds up to a given sum.
    - **Algorithm**:
        - Use two nested loops or a sliding window approach to iterate through all subarrays.
        - Check if the sum of the current subarray matches the target.
        - Print the indices if found.
2. **Rearrange Positive and Negative Numbers (void rearrangeAlternatePositiveNegative(int arr[], int size)**
    - **Analysis**: Rearranges the array to alternate positive and negative numbers.
    - **Algorithm**:
        - Partition the array into positive and negative numbers.
        - Merge the partitions alternately to form the final arrangement.
3. **Find Majority Element (int findMajorityElement(int arr[], int size)**
    - **Analysis**: Finds the majority element in the array, which appears more than n/2 times.
    - **Algorithm**:
        - Use the Boyer-Moore Voting Algorithm to identify a candidate.
        - Verify the candidate's count in a second pass.
4. **Longest Increasing Subsequence (int longestIncreasingSubsequence(int arr[], int size)**
    - **Analysis**: Finds the length of the longest increasing subsequence in the array.
    - **Algorithm**:
        - Use dynamic programming to track the maximum subsequence length ending at each index.
        - Return the maximum value from the DP table.

1. **Find Duplicates (void findDuplicates(int arr[], int size)**
   - **Analysis**: Identifies duplicate elements in the array.
   - **Algorithm**:
     - Use a hash map or sort the array to find repeated elements.
     - Traverse the array and store elements with frequency greater than 1.
2. **Find Intersection of Two Arrays (void findIntersection(int arr1[], int size1, int arr2[], int size2)**
   - **Analysis**: Finds common elements between two arrays.
   - **Algorithm**:
     - Sort both arrays.
     - Use two pointers to traverse and find matching elements.
3. **Find Union of Two Arrays (void findUnion(int arr1[], int size1, int arr2[], int size2)**
   - **Analysis**: Finds the union of two arrays, containing all unique elements.
   - **Algorithm**:
     - Sort both arrays.
     - Merge them, skipping duplicates.

Analysis of Matrix Operations

Basic Matrix Functions

1. **Initialize Matrix (void initializeMatrix(int rows, int cols, int matrix[rows][cols], int value)**
   - **Analysis**: Sets all elements of a matrix to a given value.
   - **Algorithm**:
     - Use nested loops to assign value to each element.
2. **Print Matrix (void printMatrix(int rows, int cols, int matrix[rows][cols])**
   - **Analysis**: Prints the matrix in a formatted way.
   - **Algorithm**:
     - Use nested loops to traverse and print elements row by row.
3. **Input Matrix (void inputMatrix(int rows, int cols, int matrix[rows][cols])**
   - **Analysis**: Allows the user to input elements for a matrix.
   - **Algorithm**:
     - Use nested loops to scan input for each element.

Matrix Arithmetic

- **Matrix Addition (void addMatrices(int rows, int cols, int mat1[rows][cols], int mat2[rows][cols], int result[rows][cols])**
  - **Analysis**: Adds two matrices element-wise.
  - **Algorithm**:
    - Use nested loops to sum corresponding elements of mat1 and mat2.
- **Matrix Subtraction (void subtractMatrices(int rows, int cols, int mat1[rows][cols], int mat2[rows][cols], int result[rows][cols])**
  - **Analysis**: Subtracts the second matrix from the first matrix element-wise.
  - **Algorithm**:
    - Use nested loops to subtract elements of mat2 from mat1.
- **Matrix Multiplication (void multiplyMatrices(int rows1, int cols1, int mat1[rows1][cols1], int rows2, int cols2, int mat2[rows2][cols2], int result[rows1][cols2])**
  - **Analysis**: Multiplies two matrices.
  - **Algorithm**:
    - Use three nested loops to calculate the dot product for each cell in the result matrix.
- **Scalar Multiplication (void scalarMultiplyMatrix(int rows, int cols, int matrix[rows][cols], int scalar)**
  - **Analysis**: Multiplies each element of the matrix by a scalar value.
  - **Algorithm**:
    - Use nested loops to multiply each element by scalar.

Matrix Properties and Checks

- **Check if Square Matrix (bool isSquareMatrix(int rows, int cols)**
  - **Analysis**: Determines if the matrix has an equal number of rows and columns.
  - **Algorithm**:
    - Return true if rows == cols; otherwise, return false.
- **Check if Identity Matrix (bool isIdentityMatrix(int size, int matrix[size][size])**
  - **Analysis**: Checks if the matrix is an identity matrix (1s on the diagonal, 0s elsewhere).
  - **Algorithm**:
    - Traverse the matrix using nested loops.
    - Check if diagonal elements are 1 and all others are 0.

- **Check if Diagonal Matrix (bool isDiagonalMatrix(int size, int matrix[size][size])**
  - **Analysis**: Determines if all non-diagonal elements are zero.
  - **Algorithm**:
    - Traverse the matrix and verify that non-diagonal elements are zero.

- **Check if Symmetric Matrix (bool isSymmetricMatrix(int size, int matrix[size][size])**
  - **Analysis**: Checks if the matrix is equal to its transpose.
  - **Algorithm**:
    - Compare matrix[i][j] with matrix[j][i] for all indices. Return false if any mismatch is found.

Matrix Operations

1. **Check if Upper Triangular Matrix**
2. **Analysis**: Determines if the matrix is upper triangular, meaning all elements below the main diagonal are zero.

3**Algorithm**:

- Traverse each element of the matrix below the diagonal (i.e., where the row index is greater than the column index).
- If any element is non-zero, return false.
- If all elements below the diagonal are zero, return true.

1. **Transpose Matrix**
2. **Analysis**: Computes the transpose of a matrix, where rows are swapped with columns.
3. **Algorithm**:

- Create a new matrix of dimensions cols x rows.
- For each element in the original matrix, place it in the corresponding position in the new matrix by swapping its row and column indices.

4. **Determinant of a Matrix**
5. **Analysis**: Calculates the determinant of a square matrix.
6. **Algorithm**:

- For a 2x2 matrix, directly compute the determinant as ad - bc where the matrix is [[a, b], [c, d]].
- For larger matrices, perform cofactor expansion by selecting a row or column, multiplying each element by its cofactor, and summing the results.
- Recursively calculate the determinant of smaller submatrices.

7. **Inverse of a Matrix**
8. **Analysis**: Computes the inverse of a matrix using Gaussian elimination or other methods.
9. **Algorithm**:

- Augment the matrix with an identity matrix of the same size.
- Perform row operations to transform the original matrix into an identity matrix.
- The transformed identity matrix will become the inverse of the original matrix.

10. **Matrix Power**
11. **Analysis**: Raises a square matrix to a given power.
12. **Algorithm**:

- If the power is 1, return the matrix itself.
- For higher powers, multiply the matrix by itself repeatedly.
- Use a loop to multiply the matrix by itself until the desired power is reached.