Cryptography & Networking security

Assignment 2

Yoosuf Rifathu (S073094)

2023

Contents

[The suite of functions covered in this documentation includes: 2](#_Toc152880237)

[Greatest Common Divisor (GCD) Calculator: 2](#_Toc152880238)

[Purpose 2](#_Toc152880239)

[Algorithm: 2](#_Toc152880240)

[Flow: 2](#_Toc152880241)

[Extended Euclidean Algorithm: 2](#_Toc152880242)

[Purpose 3](#_Toc152880243)

[Algorithm: 3](#_Toc152880244)

[Flow: 3](#_Toc152880245)

[Hill Cipher Encryption/Decryption: 3](#_Toc152880246)

[Purpose: 3](#_Toc152880247)

[Algorithm: 3](#_Toc152880248)

[Flow: 3](#_Toc152880249)

[RSA Algorithm: 3](#_Toc152880250)

[Purpose: 3](#_Toc152880251)

[Algorithm: 3](#_Toc152880252)

[Flow: 4](#_Toc152880253)

[Source codes 4](#_Toc152880254)

[Greatest Common Divisor (GCD) 4](#_Toc152880255)

[Extended Euclidean Algorithm: 5](#_Toc152880256)

[Hill Cipher Encryption/Decryption 7](#_Toc152880257)

[RSA Algorithm 10](#_Toc152880258)

[Driver code 13](#_Toc152880259)

[User Interface 15](#_Toc152880260)

Introduction to Cryptographic Functions

This document provides a detailed overview of the cryptographic functions implemented in our application. These functions are essential building blocks in the field of cryptography, serving various purposes from basic security to complex data encryption. The purpose of this documentation is to offer a clear understanding of each function's mechanics, its application, and its significance in cryptographic operations.

# The suite of functions covered in this documentation includes:

Greatest Common Divisor (GCD) Calculator:

A fundamental function used to compute the greatest common divisor of two integers. The GCD is a key component in many cryptographic algorithms, playing a significant role in simplifying number-theoretic computations.

Purpose:

To find the largest number that divides both of the given integers without leaving a remainder.

### Algorithm:

Uses the Euclidean algorithm, a well-known method for computing the greatest common divisor.

### Flow:

* The user inputs two integer values.
* The function recursively calculates the remainder of the division of the larger number by the smaller one.
* This process is repeated, each time replacing the larger number with the smaller number and the smaller number with the calculated remainder.
* When the remainder becomes zero, the non-zero number at this stage is the GCD.

## Extended Euclidean Algorithm:

This algorithm extends the traditional Euclidean Algorithm to not only calculate the GCD of two numbers but also to find the coefficients of Bézout's identity. This is particularly useful in computations involving modular inverses, a concept central to many encryption and decryption algorithms.

Purpose:

To find the greatest common divisor of two numbers and the coefficients of Bézout's identity.

### Algorithm:

Extends the Euclidean algorithm, to find not only the GCD but also the integers x and y such that ax + by = gcd(a,b)

### Flow:

* The user enters two numbers.
* The algorithm recursively applies the Euclidean algorithm to these numbers.
* During each recursion, it maintains and updates the coefficients x and y.
* The recursion ends when one of the numbers becomes zero. At this point, the GCD is the other number, and the coefficients x and y are the Bézout coefficients.

## Hill Cipher Encryption/Decryption:

The Hill Cipher is a classical cipher based on linear algebra, using matrix multiplication for encryption and decryption. Our implementation allows encrypting and decrypting 2-character blocks of text, demonstrating the interplay between linear algebra and cryptography.

### Purpose:

To encrypt and decrypt text using linear algebra.

### Algorithm:

Based on matrix multiplication, it transforms the given text into a vector and then multiplies it by an invertible key matrix.

### Flow:

* For encryption, the user provides a 2-character string and a 2x2 key matrix.
* Each character of the string is converted into a numerical value (A=0, B=1, ..., Z=25).
* These numbers form a vector, which is then multiplied by the key matrix modulo 26.
* The resulting vector is converted back to characters, forming the encrypted text.
* For decryption, the process is reversed using the inverse of the key matrix.

## RSA Algorithm:

A cornerstone in modern cryptography, the RSA algorithm is a public-key cryptosystem used for secure data transmission. The implementation covers key generation, encryption, and decryption processes, showcasing one of the most widely-used methods for secure communication.

### Purpose:

To encrypt and decrypt messages using one of the earliest and widely used public-key cryptosystems.

### Algorithm:

Relies on a pair of keys—a public key for encryption and a private key for decryption.

### Flow:

* The user inputs two prime numbers, P and Q, and an integer E, which must be coprime to phi(n) where phi(n) is calculated as (P-1) times (Q-1).
* The product of P and Q gives n, which, along with E, forms the public key.
* The modular inverse of E modulo phi(n) is computed to derive the private key.
* To encrypt a message, each character in the plaintext is converted into a number and then raised to the power of E and taken modulo n.
* Decryption is performed by raising each number in the encrypted message to the power specified by the private key exponent and then taking the result modulo n, which retrieves the original plaintext message.

# Source codes

The functions are implemented using **JavaScript** language. The UI and Results are displayed in HTML.

You can also execute the functions directly from the JavaScript console and see the output.

Greatest Common Divisor (GCD)

/\*\*

 \* GCD Calculator

 \* Computes the greatest common divisor (GCD) of two numbers using the Euclidean algorithm.

 \* @param {number} a - First number

 \* @param {number} b - Second number

 \* @returns {number} - The GCD of a and b

 \*/

function gcd(a, b) {

    if (b === 0) return a;

    return gcd(b, a % b);

  }

## Extended Euclidean Algorithm:

/\*\*

 \* Extended Euclidean Algorithm

 \* This function computes the Greatest Common Divisor (GCD) of two numbers 'a' and 'b',

 \* along with the coefficients (s and t) of the Bézout's identity, i.e., as + bt = gcd(a, b).

 \* It stores each step of the algorithm including intermediate s and t values.

 \* @param {number} a - The first number.

 \* @param {number} b - The second number.

 \* @returns {Array} - An array of objects representing each step of the algorithm.

 \*/

function extendedEuclidean(a, b) {

    let steps = [];  // Array to store each step of the algorithm

    /\*\*

     \* Recursive helper function for the Extended Euclidean Algorithm.

     \* @param {number} a - The first number.

     \* @param {number} b - The second number.

     \* @returns {Array} - Array containing the coefficients x, y, gcd, s1, s2, s3, t1, t2, t3.

     \*/

    function \_extendedEuclidean(a, b) {

      if (b === 0) {

        // Base case: if b is 0, gcd is a and coefficients s = 1, t = 0

        return [1, 0, a, 1, 0, 0, 1];  // Initial values for s and t coefficients

      }

      let [x1, y1, gcd, s2, t2, s1, t1] = \_extendedEuclidean(b, a % b);

      let x = y1;

      let y = x1 - Math.floor(a / b) \* y1;

      let remainder = a % b;

      // Calculate s3 and t3 for the current step using previous step's values

      let s3 = s2 - Math.floor(a / b) \* s1;

      let t3 = t2 - Math.floor(a / b) \* t1;

      // Store the current step with all relevant values

      steps.push({ quotient: Math.floor(a / b), a, b, remainder, s1, s2, s3, t1, t2, t3, gcd });

      return [x, y, gcd, s3, t3, s2, t2];

    }

    \_extendedEuclidean(a, b); // Start the recursive process

    return steps; // Return the array of steps after completion

  }

  /\*\*

 \* Displays the steps of the Extended Euclidean Algorithm in a table format.

 \* This function takes two numbers 'a' and 'b', executes the Extended Euclidean Algorithm,

 \* and then creates an HTML table to display each step of the process.

 \* @param {number} a - The first number.

 \* @param {number} b - The second number.

 \*/

function displayExtendedEuclideanSteps(a, b) {

    let steps = extendedEuclidean(a, b); // Execute the algorithm to get all steps

    let tableHTML = `<table border="1">

    <tr>

    <th>Quotient</th>

    <th>a</th>

    <th>b</th>

    <th>Remainder</th>

    <th>s1</th>

    <th>s2</th>

    <th>s3</th>

    <th>t1</th>

    <th>t2</th>

    <th>t3</th>

  </tr>`;

    // Loop through each step and add a row to the table for that step

    for (let step of steps) {

      tableHTML += `<tr>

      <td>${step.quotient}</td>

      <td>${step.a}</td>

      <td>${step.b}</td>

      <td>${step.remainder}</td>

      <td>${step.s1}</td>

      <td>${step.s2}</td>

      <td>${step.s3}</td>

      <td>${step.t1}</td>

      <td>${step.t2}</td>

      <td>${step.t3}</td>

      </tr>`;

    }

    tableHTML += `</table>`;

    document.getElementById('result').innerHTML = tableHTML; // Display the table in the 'result' element

  }

## Hill Cipher Encryption/Decryption

/\*\*

 \* Multiplies a 2x2 matrix with a 2x1 vector.

 \* This function is specifically used in the Hill Cipher for encryption.

 \* @param {array} matrix - A 2x2 matrix (array of arrays).

 \* @param {array} vector - A 2x1 vector (array).

 \* @returns {array} - The resulting 2x1 vector after multiplication.

 \*/

function multiplyMatrices(matrix, vector) {

  const result = [0, 0];

  for (let i = 0; i < 2; i++) {

    for (let j = 0; j < 2; j++) {

      // Multiply the matrix element by the corresponding vector element and add to the result.

      result[i] += matrix[i][j] \* vector[j];

    }

  }

  return result;

}

/\*\*

 \* Hill Cipher Encryption (2x2 Matrix Key)

 \* Encrypts a 2-character string using a 2x2 matrix key.

 \* Note: This is a simple version and does not handle all edge cases.

 \* @param {string} text - The text to encrypt (2 characters)

 \* @param {array} key - The 2x2 matrix key for encryption

 \* @returns {string} - The encrypted text

 \*/

function hillCipherEncrypt(text, key) {

  // Convert text to numerical values (A=0, B=1, ..., Z=25)

  text = text.toUpperCase(); //to ignore case

  const messageVector = [

    text.charCodeAt(0) - "A".charCodeAt(0),

    text.charCodeAt(1) - "A".charCodeAt(0),

  ];

  const encryptedVector = multiplyMatrices(key, messageVector);

  // Convert back to letters and return encrypted text

  return String.fromCharCode(

    (encryptedVector[0] % 26) + "A".charCodeAt(0),

    (encryptedVector[1] % 26) + "A".charCodeAt(0)

  );

}

/\*\*

 \* Function to decrypt a message using the Hill Cipher.

 \* The decryption process involves several linear algebra steps:

 \* 1. Calculate the determinant of the key matrix.

 \* 2. Find the multiplicative inverse of the determinant in modulo 26.

 \* 3. Calculate the adjugate matrix of the key matrix.

 \* 4. Multiply the adjugate matrix by the determinant's inverse, then apply modulo 26.

 \* @param {string} encryptedText - The encrypted text to decrypt (2 characters).

 \* @param {array} key - The 2x2 matrix key used for decryption.

 \* @returns {string} - The decrypted text.

 \*/

function hillCipherDecrypt(encryptedText, key) {

  /\*\*

   \* Function to find the modular inverse of a number.

   \* This is used to find the inverse of the determinant in modulo 26 for the Hill Cipher.

   \* @param {number} n - The number to find the inverse for.

   \* @param {number} mod - The modulo (26 for Hill Cipher).

   \* @returns {number} - The modular inverse of n.

   \*/

  function \_modInverse(n, mod) {

    for (let x = 1; x < mod; x++) {

      if (((n % mod) \* (x % mod)) % mod == 1) {

        return x;

      }

    }

    return -1; // Return -1 if no inverse exists

  }

  encryptedText = encryptedText.toUpperCase(); //to ignore case

  // Calculate the determinant of the key matrix and ensure it's positive and mod 26

  let det = key[0][0] \* key[1][1] - key[0][1] \* key[1][0];

  det = ((det % 26) + 26) % 26;

  // Find the multiplicative inverse of the determinant

  let detInverse = \_modInverse(det, 26);

  if (detInverse < 0) {

    throw new Error("Inverse does not exist. Key matrix is not invertible.");

  }

  // Calculate the adjugate matrix: swap diagonal elements and negate others, then mod 26

  let adj = [

    [key[1][1], -key[0][1]],

    [-key[1][0], key[0][0]],

  ];

  adj = adj.map((row) => row.map((el) => (((el \* detInverse) % 26) + 26) % 26));

  // Decrypt the message by multiplying the adjugate matrix with the encrypted vector

  const encryptedVector = [

    encryptedText.charCodeAt(0) - "A".charCodeAt(0),

    encryptedText.charCodeAt(1) - "A".charCodeAt(0),

  ];

  const decryptedVector = multiplyMatrices(adj, encryptedVector);

  // Convert numeric values back to letters

  return String.fromCharCode(

    (decryptedVector[0] % 26) + "A".charCodeAt(0),

    (decryptedVector[1] % 26) + "A".charCodeAt(0)

  );

}

## RSA Algorithm

/\*\*

 \* Modular Inverse Function

 \* Computes the modular multiplicative inverse of 'e' modulo 'phi'.

 \* It's used in RSA to find the private key 'd'.

 \* The modular inverse is the number 'd' such that (e \* d) % phi = 1.

 \* This function implements the Extended Euclidean Algorithm to find such 'd'.

 \* @param {number} e - The exponent used in the public key.

 \* @param {number} phi - Euler's totient function of n (n = p\*q).

 \* @returns {number} - The modular inverse of 'e' modulo 'phi'.

 \*/

function modInverse(e, phi) {

    function \_extendedEuclidean(a, b) {

        function \_\_extendedEuclidean(a, b) {

          if (b === 0) {

            return [1, 0, a]; // Coefficients x, y, and gcd

          }

          let [x1, y1, gcd] = \_\_extendedEuclidean(b, a % b);

          let x = y1;

          let y = x1 - Math.floor(a / b) \* y1;

          return [x, y, gcd];

        }

        return \_\_extendedEuclidean(a, b); // Only return the required values

      }

    let [x, y, gcd] = \_extendedEuclidean(e, phi);

    if (gcd !== 1) {

      console.error("Inverse doesn't exist");

      return -1;

    } else {

      return (x % phi + phi) % phi; // Ensure result is positive

    }

  }

  /\*\*

   \* RSA Key Generation

   \* Generates the public and private keys used in RSA.

   \* RSA keys are based on large prime numbers. The security of RSA comes from

   \* the difficulty of factoring large numbers that are a product of two primes.

   \* @param {number} p - A large prime number.

   \* @param {number} q - Another large prime number, different from 'p'.

   \* @param {number} e - Choose 'e' such that 1 < e < phi and e is co-prime to phi.

   \* @returns {object} - An object containing the public key (e, n) and private key (d, n).

   \*/

  function rsaKeyGeneration(p, q, e) {

    // Choose 'e' such that 1 < e < phi and e is co-prime to phi.

    const n = p \* q; // n is the modulus for both the public and private keys. Its length, usually expressed in bits, is the key length.

    const phi = (p - 1) \* (q - 1); // Euler's totient function of n.

    // Find 'd', the modular inverse of 'e' modulo 'phi'.

    let d = modInverse(e, phi);

    if (d === -1) {

        throw new Error("Invalid 'e' value. Unable to compute the modular inverse.");

    }

    // Public key is (e, n) and private key is (d, n).

    return { publicKey: { e, n }, privateKey: { d, n } };

  }

  /\*\*

   \* RSA Encryption

   \* Encrypts a plaintext message using the RSA public key.

   \* The RSA encryption is based on modular exponentiation.

   \* @param {string} text - The plaintext message to encrypt.

   \* @param {object} publicKey - The RSA public key {e, n}.

   \* @returns {string} - The encrypted message, represented as a space-separated string of numbers.

   \*/

  function rsaEncrypt(text, publicKey) {

    const { e, n } = publicKey;

    // Convert each character to its ASCII value, then encrypt it using modular exponentiation.

    return text.split('').map(char => {

      let m = char.charCodeAt(0);

      return BigInt(m) \*\* BigInt(e) % BigInt(n);

    }).join(' ');

  }

  /\*\*

   \* RSA Decryption

   \* Decrypts an encrypted message using the RSA private key.

   \* The decryption is the reverse process of the encryption, also based on modular exponentiation.

   \* @param {string} encryptedText - The encrypted message, represented as a space-separated string of numbers.

   \* @param {object} privateKey - The RSA private key {d, n}.

   \* @returns {string} - The decrypted plaintext message.

   \*/

  function rsaDecrypt(encryptedText, privateKey) {

    const { d, n } = privateKey;

    // Convert each number back to its original character using modular exponentiation.

    return encryptedText.split(' ').map(num => {

      let c = BigInt(num);

      return String.fromCharCode(Number(c \*\* BigInt(d) % BigInt(n)));

    }).join('');

  }

## Driver code

The main driver code used to get input from user interface and execute the functions.

// Function to update input fields based on selected operation

function updateInputFields() {

  const operation = document.getElementById("operation").value;

  const inputFields = document.getElementById("inputFields");

  inputFields.innerHTML = "";

  switch (operation) {

    case "gcd":

      inputFields.innerHTML =

        '<input type="number" id="inputA" placeholder="Enter number A">' +

        '<input type="number" id="inputB" placeholder="Enter number B">';

      break;

    case "extendedEuclidean":

      inputFields.innerHTML =

        '<input type="number" id="inputA" placeholder="Enter number A">' +

        '<input type="number" id="inputB" placeholder="Enter number B">';

      break;

    case "hillCipherEnc":

      inputFields.innerHTML =

        '<input type="text" id="inputText" placeholder="Enter text (2 characters)">' +

        '<input type="text" id="inputKey" placeholder="Enter key (4 numbers separated by space 2X2 matrix)">';

      break;

      case "hillCipherDec":

        inputFields.innerHTML =

          '<input type="text" id="inputText" placeholder="Enter text (2 characters)">' +

          '<input type="text" id="inputKey" placeholder="Enter key (4 numbers separated by space 2X2 matrix)">';

        break;

    case "rsa":

      inputFields.innerHTML =

        '<input type="number" id="inputP" placeholder="Enter prime number P">' +

        '<input type="number" id="inputQ" placeholder="Enter prime number Q">' +

        '<input type="number" id="inputE" placeholder="Enter \'e\' value">' +

        '<input type="text" id="inputText" placeholder="Enter text to encrypt">';

      break;

  }

}

// Function to perform the selected operation

function performOperation() {

  const operation = document.getElementById("operation").value;

  let result = "";

  let updateResult = true;

  try {

  switch (operation) {

    case "gcd":

      const a = parseInt(document.getElementById("inputA").value);

      const b = parseInt(document.getElementById("inputB").value);

      result = "GCD: " + gcd(a, b);

      break;

    case "extendedEuclidean":

      const x = parseInt(document.getElementById("inputA").value);

      const y = parseInt(document.getElementById("inputB").value);

      displayExtendedEuclideanSteps(x, y);

      updateResult = false;

      break;

    case "hillCipherEnc":

      const hillCipherEnc\_text = document.getElementById("inputText").value;

      const hillCipherEnc\_keyString = document.getElementById("inputKey").value;

      const hillCipherEnc\_key = hillCipherEnc\_keyString.split(" ").map(Number);

      result =

        "Hill Cipher (Encrypted): " +

        hillCipherEncrypt(hillCipherEnc\_text, [

          [hillCipherEnc\_key[0], hillCipherEnc\_key[1]],

          [hillCipherEnc\_key[2], hillCipherEnc\_key[3]],

        ]);

      break;

    case "hillCipherDec":

      const hillCipherDec\_text = document.getElementById("inputText").value;

      const hillCipherDec\_keyString = document.getElementById("inputKey").value;

      const hillCipherDec\_key = hillCipherDec\_keyString.split(" ").map(Number);

      result =

        "Hill Cipher: (Decrypted)" +

        hillCipherDecrypt(hillCipherDec\_text, [

          [hillCipherDec\_key[0], hillCipherDec\_key[1]],

          [hillCipherDec\_key[2], hillCipherDec\_key[3]],

        ]);

      break;

    case "rsa":

      const p = parseInt(document.getElementById("inputP").value);

      const q = parseInt(document.getElementById("inputQ").value);

      const e = parseInt(document.getElementById("inputE").value);

      const rsaText = document.getElementById("inputText").value;

      const { publicKey, privateKey } = rsaKeyGeneration(p, q, e);

      const encrypted = rsaEncrypt(rsaText, publicKey);

      const decrypted = rsaDecrypt(encrypted, privateKey);

      result = `RSA Encryption: ${encrypted}, Decryption: ${decrypted}`;

      break;

  }

  if (updateResult) {

    document.getElementById("result").innerText = `Result: ${result}`;

  }

  } catch (error) {

    document.getElementById('result').innerText = `Error: ${error.message}`;

  }

}

// Event listener for operation change

document

  .getElementById("operation")

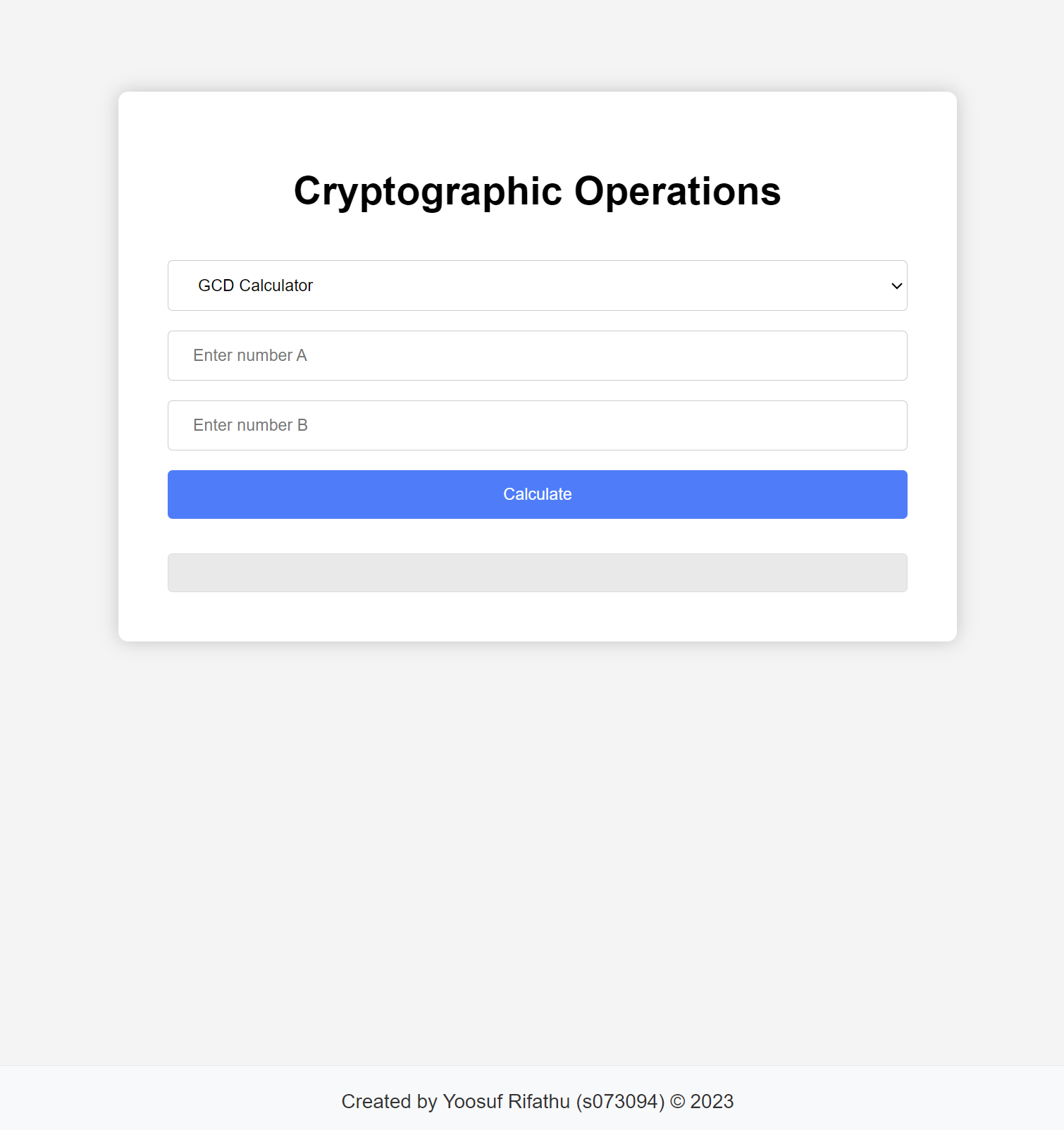
  .addEventListener("change", updateInputFields);

// Initial input fields setup

updateInputFields();

# User Interface

**User interface can be accessed by opening the ‘index.html’ in the source code.**



A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated