Data Security and Encryption

Assignment

Diffie Hellman Algorithm

Index

```
console.clear();
console.log('================================================================');
console.log('\t\t\t Diffie Hellman');
console.log('================================================================\n');

const randomPrimeNumber = PrimeNumber.getRandomPrimeNumber(1000);
const randomPrimitiveRoot = PrimitiveRoot.getRandomPrimitiveRoot(randomPrimeNumber);

const aliceDiffieHellman = new DiffieHellman(randomPrimeNumber, randomPrimitiveRoot);
const bobDiffieHellman = new DiffieHellman(randomPrimeNumber, randomPrimitiveRoot);

const alicePublicKey = aliceDiffieHellman.getPublicKey();
const bobPublicKey = bobDiffieHellman.getPublicKey();

const aliceSecretKey = aliceDiffieHellman.generateSharedKey(bobPublicKey);
const bobSecretKey = bobDiffieHellman.generateSharedKey(alicePublicKey);

console.log(
    'Both Alice and Bob have the same private/shared key:\t',
    aliceDiffieHellman.doesPrivateKeysMatch(bobDiffieHellman.getPrivateKey())
);

console.log('\n');
console.table({ aliceDiffieHellman, bobDiffieHellman });

console.log('\n\n================================================================\n');
```

DiffieHellman

```ts
export class DiffieHellman {
    primeNumber: number;
    primitiveRoot: number;
    publicKey: number;
    private privateKey: number;

    constructor(primeNumber: number, primitiveRoot: number) {
        this.primeNumber = primeNumber;
        this.primitiveRoot = primitiveRoot;
        this.publicKey = -1;
        this.privateKey = Math.floor(Math.random() * 10);
    }

    generatePublicKey() {
        let publicKey = Math.pow(this.primitiveRoot, this.privateKey) % this.primeNumber;
        this.publicKey = publicKey;
        return publicKey;
    }

    generateSharedKey(OtherPublicKey: number) {
        let sharedKey = Math.pow(OtherPublicKey, this.privateKey) % this.primeNumber;
        this.privateKey = sharedKey;
        return sharedKey;
    }

    getPublicKey() {
        if (this.publicKey === -1) {
            this.generatePublicKey();
        }
        return this.publicKey;
    }

    getPrivateKey() {
        return this.privateKey;
    }

    doesPrivateKeysMatch(OtherPrivateKey: number) {
        return this.privateKey === OtherPrivateKey;
    }
}
```

PrimeNumber

```ts
export class PrimeNumber {
    static isPrime(number: number) {
        for (let i = 2; i < number; i++) {
            if (number % i === 0) return false;
        }
        return number > 1;
    }

    static getPrimeNumbers(maxLimit: number) {
        let primeNumbers: number[] = [];
        for (let i = 2; i < maxLimit; i++) {
            if (this.isPrime(i)) {
                primeNumbers.push(i);
            }
        }
        return primeNumbers;
    }

    static getRandomPrimeNumber(maxLimit: number) {
        const primeNumbers = this.getPrimeNumbers(maxLimit);
        const randomIndex = Math.floor(Math.random() * (primeNumbers.length - 0) + 0);
        return this.getPrimeNumbers(maxLimit)[randomIndex];
    }
}
```

PrimitiveRoot

```
import bigInt from 'big-integer';

export class PrimitiveRoot {
    static getPrimitiveRoots(primeNumber: number) {
        let values: number[] = [];
        let duplicates: boolean = false;
        let primitiveRoots: number[] = [];

        for (let i = 1; i < primeNumber; i++) {
            for (let j = 0; j < primeNumber - 1; j++) {
                duplicates = false;

                let x = bigInt(i).pow(j);
                let y = x.mod(primeNumber);
                values.push(parseInt(y.toString()));

                if (this.hasDuplicates(values) === true) {
                    duplicates = true;
                    break;
                }
            }

            if (duplicates === false) {
                if (!this.isCorrect(values, primeNumber)) {
                    continue;
                } else {
                    primitiveRoots.push(i);
                }
            }
            values = [];
        }
        return primitiveRoots;
    }

    static getRandomPrimitiveRoot(primeNumber: number) {
        let values: number[] = this.getPrimitiveRoots(primeNumber);
        const randomIndex = Math.floor(Math.random() * (values.length - 0) + 0);
        return values[randomIndex];
    }

    private static hasDuplicates(numbersArray: number[]): boolean {
        return new Set(numbersArray).size !== numbersArray.length;
    }

    private static numericSorter(num1: number, num2: number) {
        return num1 - num2;
    }
```

```
    private static isCorrect(arrayToCheck: number[], primeNumber: number) {
        if (arrayToCheck.length >= primeNumber) {
            return false;
        }
        arrayToCheck = arrayToCheck.sort(this.numericSorter);

        for (let i = 1; i < primeNumber - 1; i++) {
            if (arrayToCheck[i - 1] !== i) {
                return false;
            }
        }
        return true;
    }
}
```

Output

```
================================================================================
                          Diffie Hellman
================================================================================

Both Alice and Bob have the same private/shared key:     true
```

(index)	primeNumber	primitiveRoot	publicKey	privateKey
aliceDiffieHellman	359	354	188	250
bobDiffieHellman	359	354	188	250

```
================================================================================
```

RSA Algorithm

Index

```
console.clear();
console.log('================================================================');
console.log('\t\t\t RSA (Rivest–Shamir–Adleman)');
console.log('================================================================\n');

const aliceRSA = new RSA(100);
const bobRSA = new RSA(100);

const aliceRSAPublicKey = aliceRSA.getPublicKey();
const bobRSAPublicKey = bobRSA.getPublicKey();

const plaintText = 5;
console.log('Plain Text:\t', plaintText);

const encryptedData = bobRSA.encrypt(plaintText, aliceRSAPublicKey.key, aliceRSAPublicKey.n);
console.log('Encrypted Text:\t', encryptedData);

let decryptedData = aliceRSA.decrypt(encryptedData);
console.log('Decrypted Text:\t', decryptedData);

console.log('\n');
console.table({ aliceRSA, bobRSA });

console.log('\n\n================================================================\n');
```

RSA

```ts
import bigInt from 'big-integer';
import { PrimeNumber } from './PrimeNumber';
import { EulerToient } from './EulerToient';

interface RSAKey {
    key: number;
    n: number;
}

export class RSA {
    primeNum1: number = 0;
    primeNum2: number = 0;
    productOfPrime: number = 0;
    publicKey: number = 0;
    privateKey: number = 0;

    constructor(maxLimit: number) {
        this.primeNum1 = PrimeNumber.getRandomPrimeNumber(maxLimit);
        this.primeNum2 = PrimeNumber.getRandomPrimeNumber(maxLimit);
    }

    private computePublicKey() {
        this.productOfPrime = (this.primeNum1 - 1) * (this.primeNum2 - 1);
        this.publicKey = EulerToient.getCoPrime(this.productOfPrime)[0];
    }

    getPublicKey(): RSAKey {
        this.computePublicKey();
        const key = this.publicKey;
        const n = this.primeNum1 * this.primeNum2;
        return { key, n };
    }

    private computePrivateKey() {
        let i = 1;
        while (true) {
            this.privateKey = (this.productOfPrime * i + 1) / this.publicKey;
            i++;
            if (Number.isInteger(this.privateKey)) {
                break;
            }
        }
    }
```

```
    encrypt(plainText: number, publicKey: number, n: number) {
        let cipherText;
        if (plainText > n) {
            return -1;
        }
        cipherText = bigInt(plainText).pow(publicKey).mod(n);
        return parseFloat(cipherText.toString());
    }

    decrypt(cipherText: number) {
        this.computePrivateKey();
        let plainText;
        plainText = bigInt(cipherText)
            .pow(this.privateKey)
            .mod(this.primeNum1 * this.primeNum2);
        return parseFloat(plainText.toString());
    }
}
```

EulerToient

```
export class EulerToient {
    static eulersToient(number: number) {
        if (primeNumbers.includes(number)) {
            return number - 1;
        }
        let values: number[] = [];
        for (let i = 1; i < number; i++) {
            if (this.gcd(i, number) === 1) {
                values.push(i);
            }
        }
        return values.length;
    }

    static eulersToientAndMultiply(num1: number, num2: number) {
        let p: number;
        let q: number;

        if (primeNumbers.includes(num1)) {
            p = num1 - 1;
        } else {
            p = this.eulersToient(num1);
        }

        if (primeNumbers.includes(num2)) {
            q = num2 - 1;
        } else {
            q = this.eulersToient(num2);
        }

        return p * q;
    }

    static gcd(a: number, b: number): number {
        if (!b) {
            return a;
        }
        return this.gcd(b, a % b);
    }

    static gcd_rec(a: number, b: number): number {
        if (b) {
            return this.gcd_rec(b, a % b);
        } else {
            return Math.abs(a);
        }
    }
```

```
    static getCoPrime(num: number) {
        let values: number[] = [];
        for (let i = 2; i < num; i++) {
            if (this.gcd_rec(num, i) === 1) {
                values.push(i);
            }
        }
        return values;
    }
}

const primeNumbers = [/* long list of prime numbers */]
```

Output

```
================================================================================
                        RSA (Rivest–Shamir–Adleman)
================================================================================

Plain Text:      5
Encrypted Text:  3125
Decrypted Text:  5
```

(index)	primeNum1	primeNum2	productOfPrime	publicKey	privateKey
aliceRSA	79	73	5616	5	4493
bobRSA	97	29	2688	5	0

```
================================================================================
```

SHA-1 Algorithm

Index

```
console.clear();
console.log('========================================================================');
console.log('\t\t SHA-1 (Secure Hashing Algorithm 1)');
console.log('========================================================================\n');

const stringToHash = 'Omar Qazi';
console.log('Orignal Text:\t\t', stringToHash);

const sha1Hash = Hasher.textToSha1(stringToHash);
console.log('Hashed with SHA-1:\t', sha1Hash);

console.log('\n========================================================================\n');
```

MD5 Algorithm

Index

```
console.clear();
console.log('========================================================================');
console.log('\t\t MD5 (Message Digest 5)');
console.log('========================================================================\n');

const stringToHash = 'Omar Qazi';
console.log('Orignal Text:\t\t', stringToHash);

const sha1Hash = Hasher.textToMd5(stringToHash);
console.log('Hashed with MD5:\t', sha1Hash);

console.log('\n========================================================================\n');
```

Hasher

```ts
import crypto from 'crypto';

export class Hasher {

    static textToSha1(plainText: string) {
        return crypto.createHash('sha1').update(plainText).digest('hex');
    }

    static textToMd5(plainText: string) {
        return crypto.createHash('md5').update(plainText).digest('hex');
    }

}
```

Output

```
========================================================================
                SHA-1 (Secure Hashing Algorithm 1)
========================================================================

Orignal Text:           Omar Qazi
Hashed with SHA-1:      854e5ca47ca66a827e50157e4790da3c555f8c58

========================================================================
```

```
========================================================================
                MD5 (Message Digest 5)
========================================================================

Orignal Text:           Omar Qazi
Hashed with MD5:        1336f4da709638514b3c12af69252623

========================================================================
```