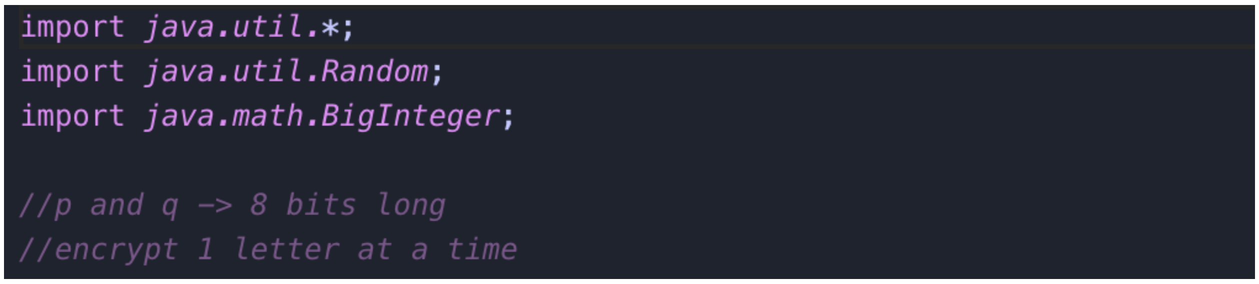
RSA Encryption

A. Introduction

RSA (Rivest–Shamir–Adleman) is an algorithm used by modern computers to encrypt and decrypt messages. It is an asymmetric cryptographic algorithm. In this implementation, Java is the language used. There are 8 sections in this report in order to accomplish the encryption and decryption: Modular, GCD, Phi Function, Square and Multiply, Miller-Rabin Test, Generating Primes, RSA Encryption and RSA Decryption.

Presupposition:

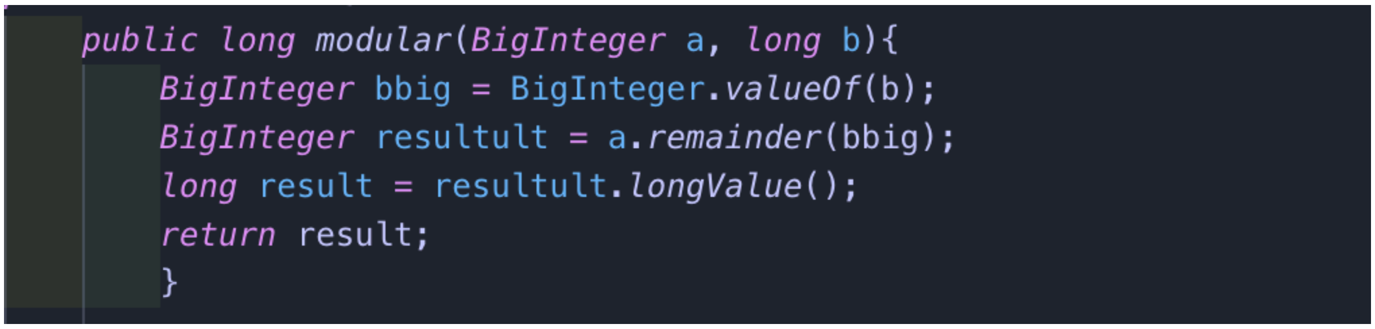


```java
import java.util.*;
import java.util.Random;
import java.math.BigInteger;

//p and q -> 8 bits long
//encrypt 1 letter at a time
```

B. Section I from Task 1

Modular Operation

The goal of modular Operation is to find the remaining using two integers. For the convenience of future operation, thw type of parameters are BigInteger and long. After this operation, *result* will be converted to *long* before being returned.

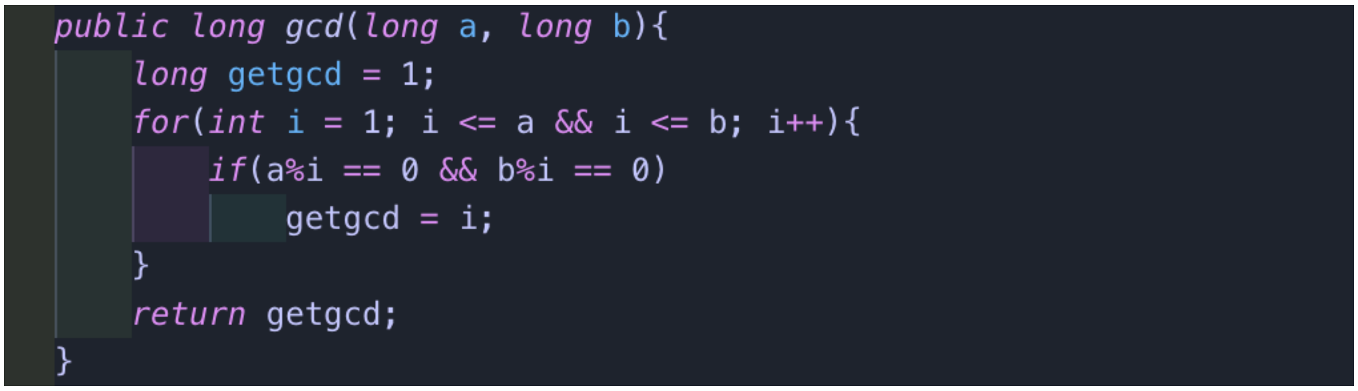


```java
public long modular(BigInteger a, long b){
    BigInteger bbig = BigInteger.valueOf(b);
    BigInteger resultult = a.remainder(bbig);
    long result = resultult.longValue();
    return result;
    }
```

C. Section II from Task 2

GCD Operation

GCD is used to find the greatest common divisor of two large integers. It's one of the most basic operation for RSA Encryption. The implementation uses brute-force to keep updating *getgcd* whenever there is a larger divisor found.

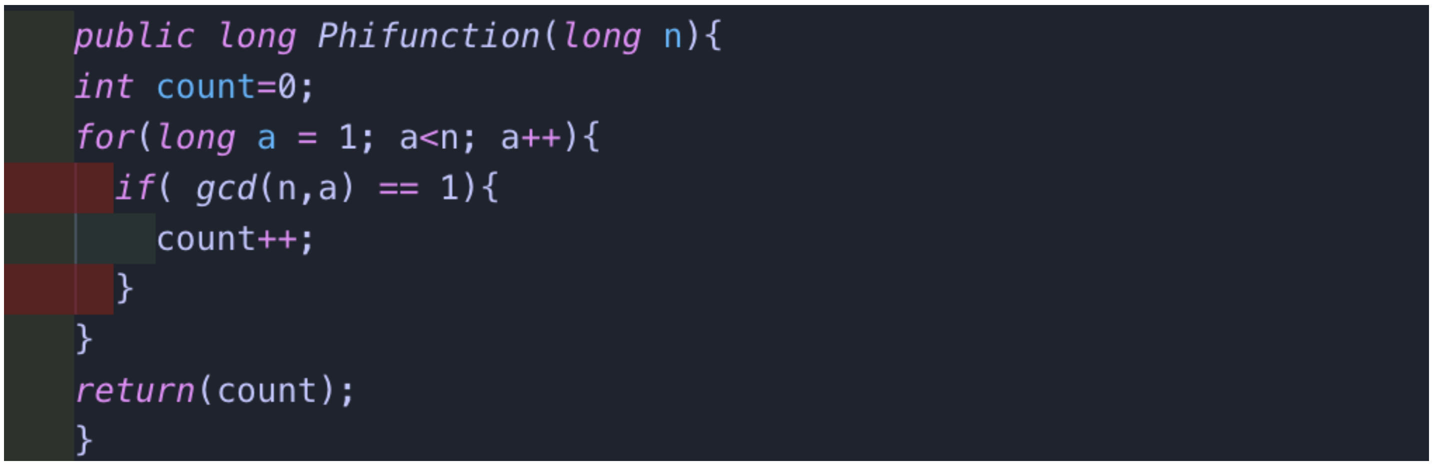


```java
public long gcd(long a, long b){
    long getgcd = 1;
    for(int i = 1; i <= a && i <= b; i++){
        if(a%i == 0 && b%i == 0)
            getgcd = i;
    }
    return getgcd;
}
```

D. Section III from Task 3

Phi Function

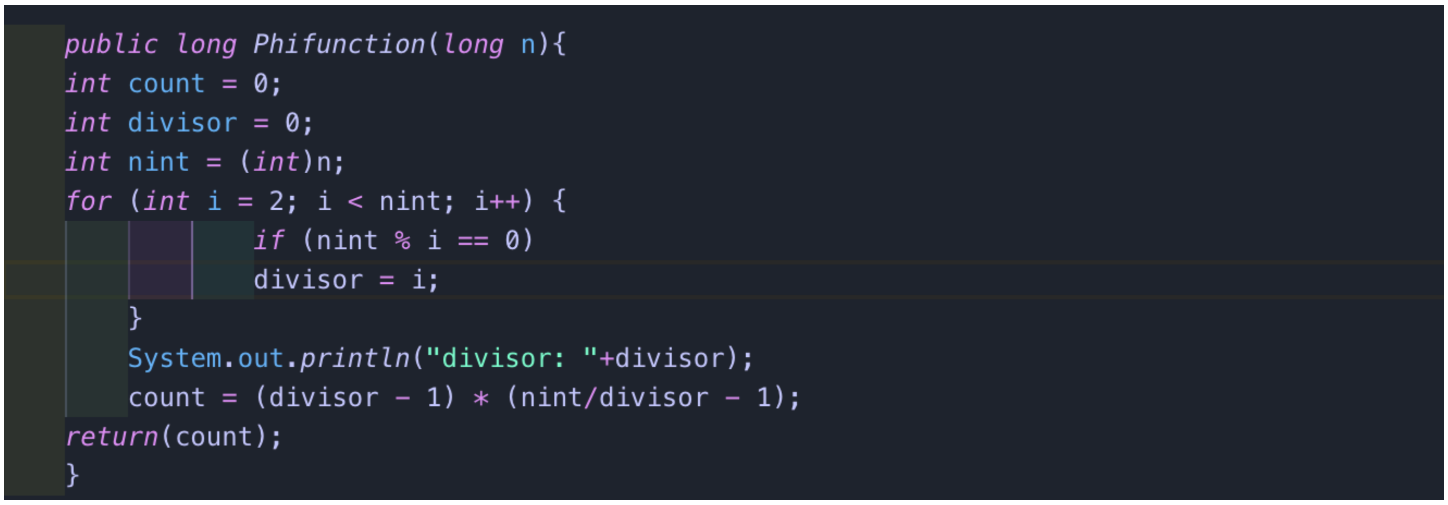
Here, we achieve the Phi Function by utilizing the GCD function previously defined. The condition *gcd(n,a) == 1* means a coprime is found, in this case we increase *count* by 1. The result is returned in *long*.



```
public long Phifunction(long n){
int count=0;
for(long a = 1; a<n; a++){
  if( gcd(n,a) == 1){
    count++;
  }
}
return(count);
}
```

Note that we know this operation is used to compute phi of *n*, which is equal to *p x q*. Consequently, since *p* and *q* are both primes, phi of n = (p-1) x (q -1).

As a result, the Phi function can be modified as below:

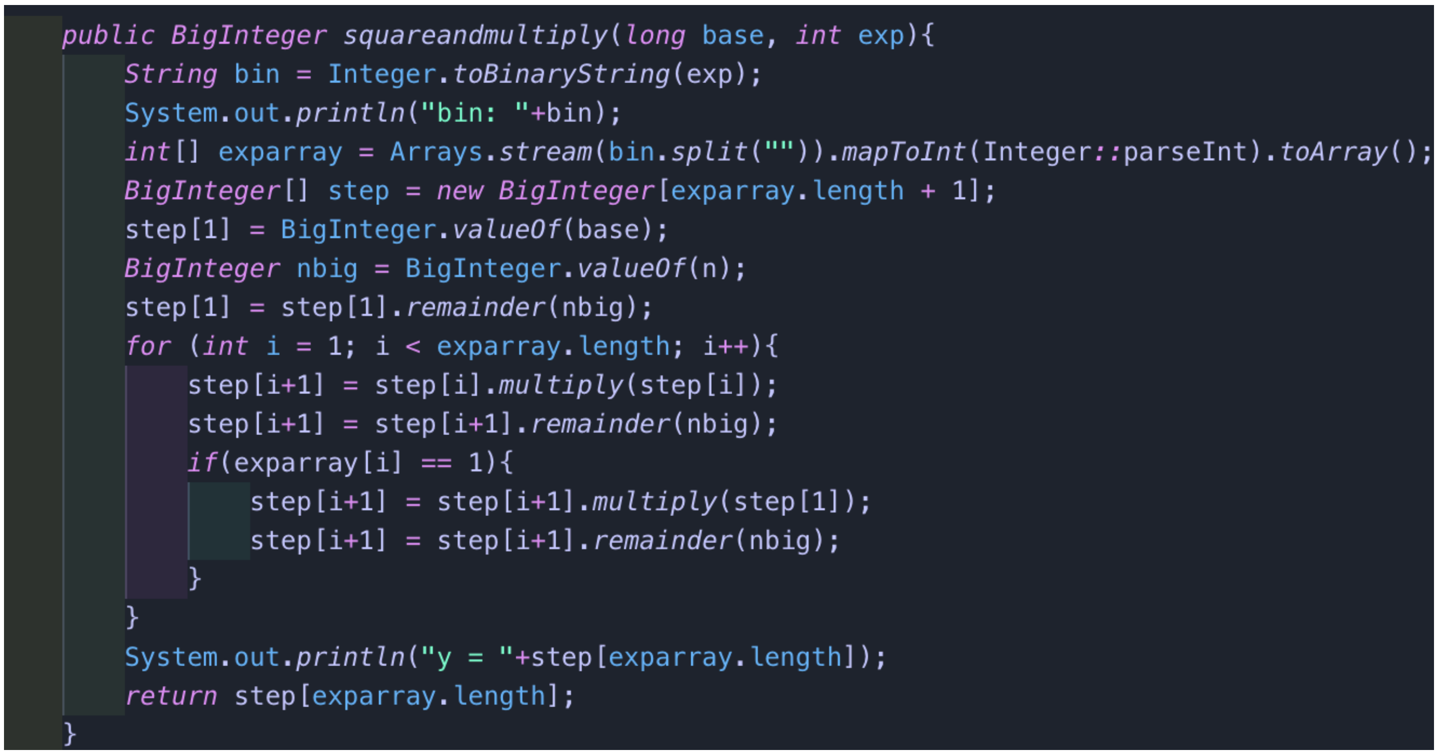


```
public long Phifunction(long n){
int count = 0;
int divisor = 0;
int nint = (int)n;
for (int i = 2; i < nint; i++) {
            if (nint % i == 0)
            divisor = i;
    }
    System.out.println("divisor: "+divisor);
    count = (divisor - 1) * (nint/divisor - 1);
return(count);
}
```

E. Section IV from Task 4

Square and Multiply

Square-and-Multiply is for quicker computation of exponent. Based on the process we learned from class, the exponent will firstly be converted to binary. Then we ran through each digit from left to right, and decide the operation based on the value of each digit. It returns a *BigInteger* as I prepared this function solely for computing *y*, which might be a larger number comparing to any other parameters. Also, this function directly catches *n* after the global variable *n* is generated, so I didn't set *n* as one of the parameters that the function takes in.

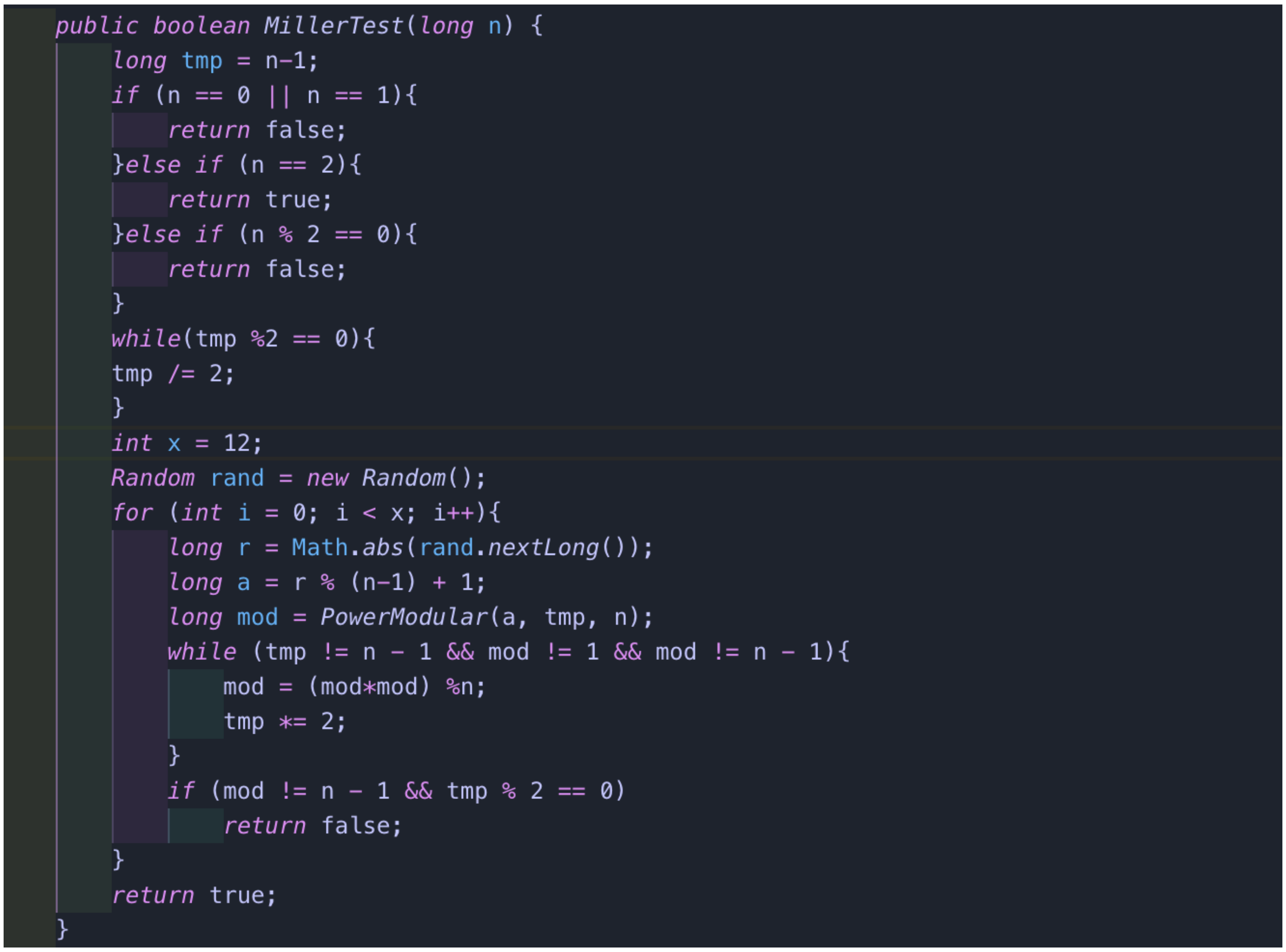


```java
public BigInteger squareandmultiply(long base, int exp){
    String bin = Integer.toBinaryString(exp);
    System.out.println("bin: "+bin);
    int[] exparray = Arrays.stream(bin.split("")).mapToInt(Integer::parseInt).toArray();
    BigInteger[] step = new BigInteger[exparray.length + 1];
    step[1] = BigInteger.valueOf(base);
    BigInteger nbig = BigInteger.valueOf(n);
    step[1] = step[1].remainder(nbig);
    for (int i = 1; i < exparray.length; i++){
        step[i+1] = step[i].multiply(step[i]);
        step[i+1] = step[i+1].remainder(nbig);
        if(exparray[i] == 1){
            step[i+1] = step[i+1].multiply(step[1]);
            step[i+1] = step[i+1].remainder(nbig);
        }
    }
    System.out.println("y = "+step[exparray.length]);
    return step[exparray.length];
}
```

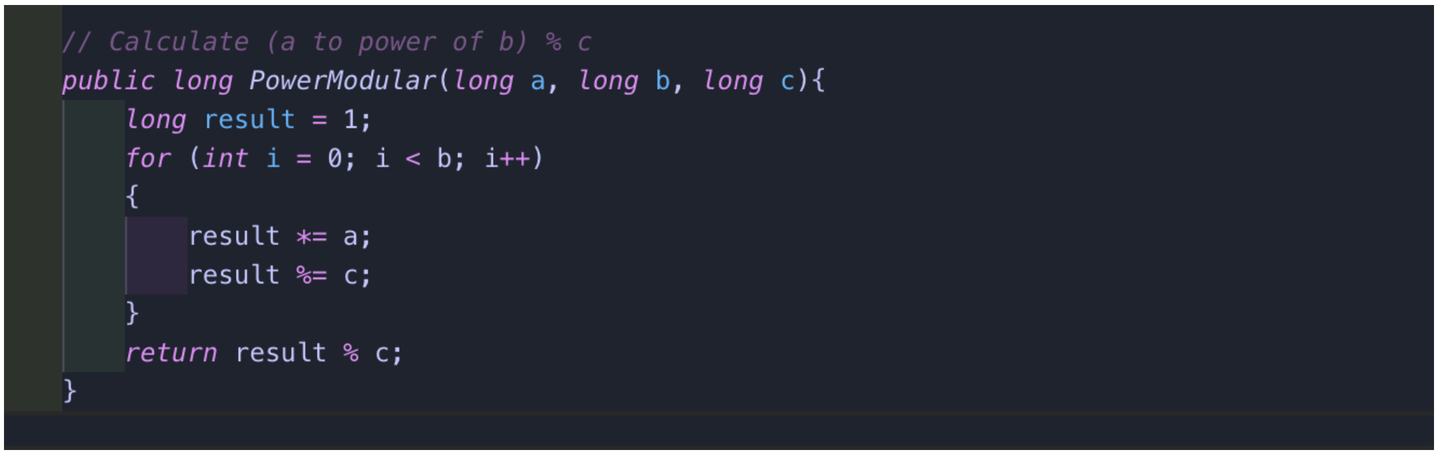
F. Section V from Task 5

Miller-Rabin Primality Test

In this implementation, Miller-Rabin Primality Test is an algorithm used to check whether the number is a prime. Obviously, the returned data type is *Boolean*. To start, if the number is 0 or 1 or even, we skip all operations and simply return false; if it's 2 then we skip all operations and return false. A sub-operation was defined in order to calculate $a^b \% c$ where *a, b* and *c* are all *long*.



```java
public boolean MillerTest(long n) {
    long tmp = n-1;
    if (n == 0 || n == 1){
        return false;
    }else if (n == 2){
        return true;
    }else if (n % 2 == 0){
        return false;
    }
    while(tmp %2 == 0){
    tmp /= 2;
    }
    int x = 12;
    Random rand = new Random();
    for (int i = 0; i < x; i++){
        long r = Math.abs(rand.nextLong());
        long a = r % (n-1) + 1;
        long mod = PowerModular(a, tmp, n);
        while (tmp != n - 1 && mod != 1 && mod != n - 1){
            mod = (mod*mod) %n;
            tmp *= 2;
        }
        if (mod != n - 1 && tmp % 2 == 0)
            return false;
    }
    return true;
}
```

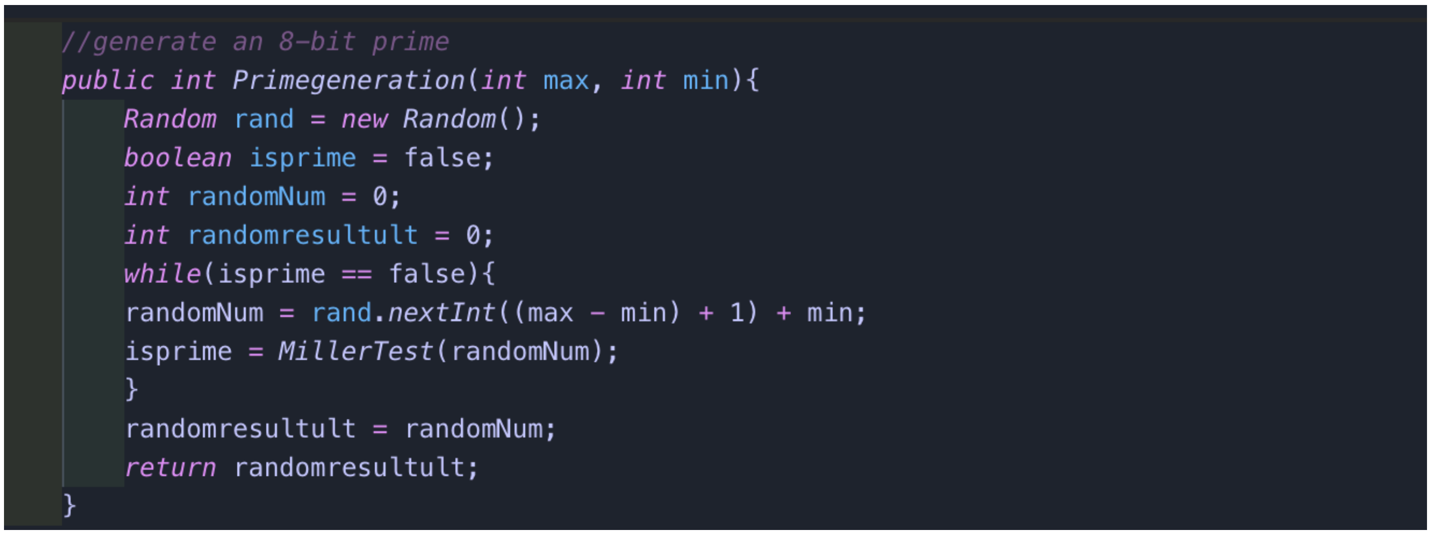


```java
// Calculate (a to power of b) % c
public long PowerModular(long a, long b, long c){
    long result = 1;
    for (int i = 0; i < b; i++)
    {
        result *= a;
        result %= c;
    }
    return result % c;
}
```

G. Section VI from Task 6

Generating Primes

For all the process to begin, we need a mechanism generating primes. As stated at the top of this report, we will generate *p* and *q* in 8 bits. The algorithm works like this: First there will be a random number chosen between the range *min* to *max*; then *MillerTest()* will tell us whether it's a prime or not; if not, another random number will be choose again until *MillerTest()* returns *True.*

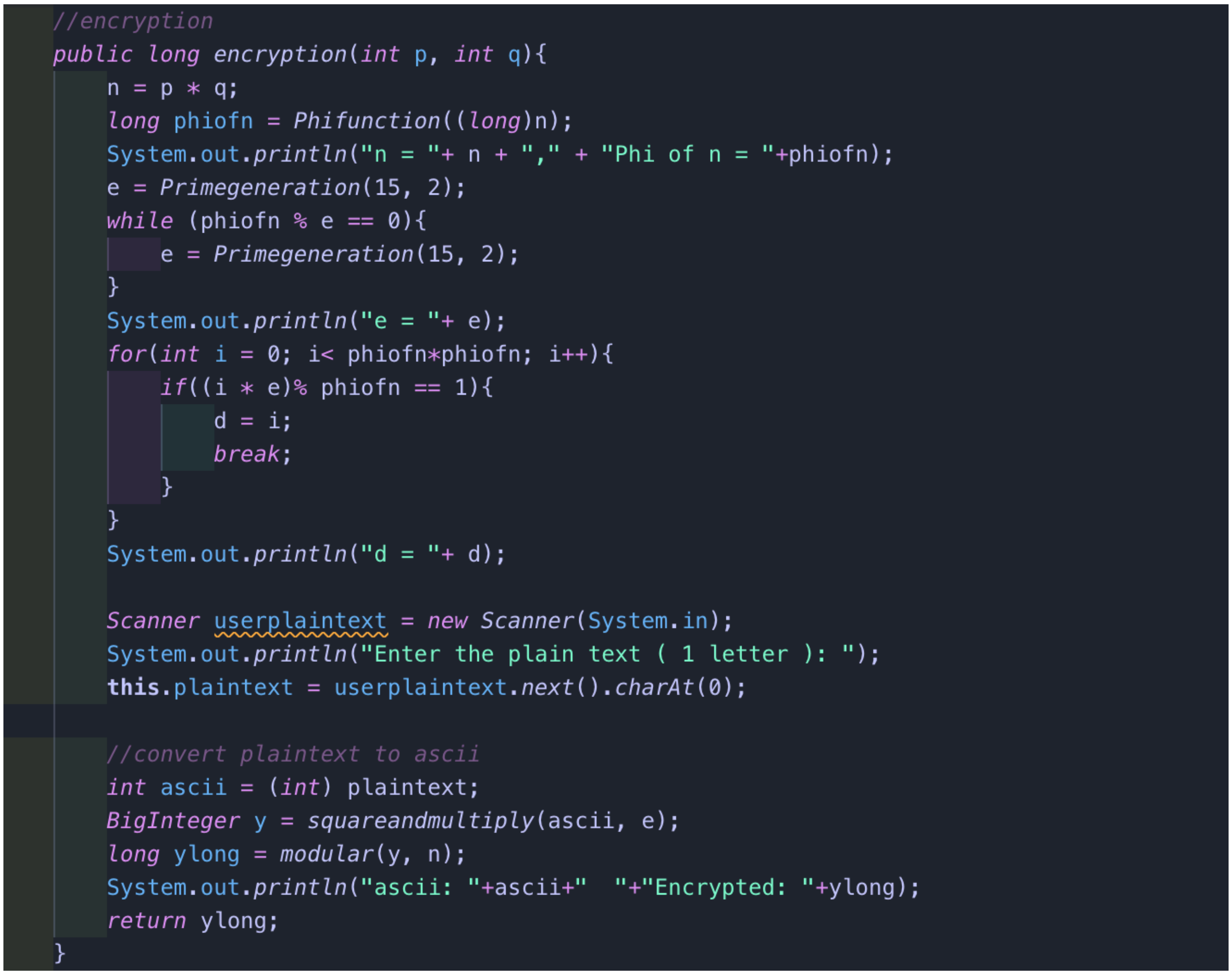


```
//generate an 8-bit prime
public int Primegeneration(int max, int min){
    Random rand = new Random();
    boolean isprime = false;
    int randomNum = 0;
    int randomresultult = 0;
    while(isprime == false){
    randomNum = rand.nextInt((max - min) + 1) + min;
    isprime = MillerTest(randomNum);
    }
    randomresultult = randomNum;
    return randomresultult;
}
```

H. Section VII from Task 7

Encryption

Now we have all our tools ready for encryption. The operation will take the *p* and *q*, generate *n* based on them, then select the *e* (to make it's a small one, the range is set between 15 and 2) before computing *d.* After these parameters are all decided, it then asks for user to input a letter. The letter will be then converted to ascii number and printed out.

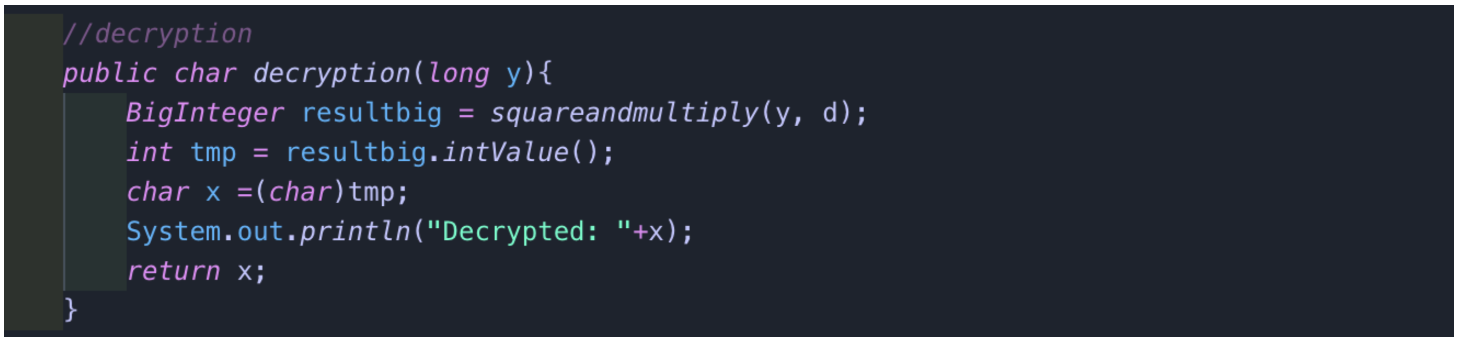


```
//encryption
public long encryption(int p, int q){
    n = p * q;
    long phiofn = Phifunction((long)n);
    System.out.println("n = "+ n + "," + "Phi of n = "+phiofn);
    e = Primegeneration(15, 2);
    while (phiofn % e == 0){
        e = Primegeneration(15, 2);
    }
    System.out.println("e = "+ e);
    for(int i = 0; i< phiofn*phiofn; i++){
        if((i * e)% phiofn == 1){
            d = i;
            break;
        }
    }
    System.out.println("d = "+ d);

    Scanner userplaintext = new Scanner(System.in);
    System.out.println("Enter the plain text ( 1 letter ): ");
    this.plaintext = userplaintext.next().charAt(0);

    //convert plaintext to ascii
    int ascii = (int) plaintext;
    BigInteger y = squareandmultiply(ascii, e);
    long ylong = modular(y, n);
    System.out.println("ascii: "+ascii+"  "+"Encrypted: "+ylong);
    return ylong;
}
```

I. Section VIII from Task 8

Decryption

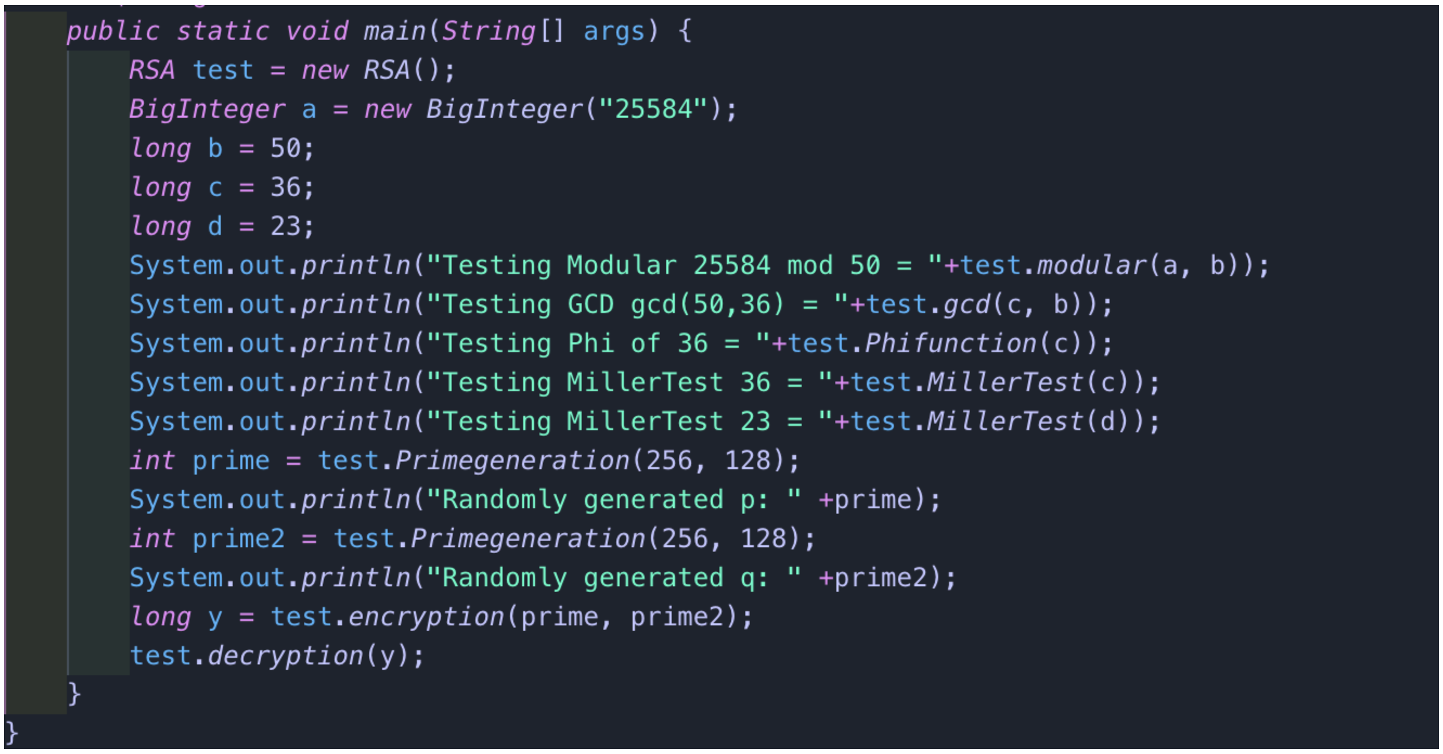
To decrypt the ciphertext, simply use *squareandmultiply()* to derive the *x*. Note that the number we got is not the answer yet – it needs to be converted back to *char*. The process is shown below.



```
//decryption
public char decryption(long y){
    BigInteger resultbig = squareandmultiply(y, d);
    int tmp = resultbig.intValue();
    char x =(char)tmp;
    System.out.println("Decrypted: "+x);
    return x;
}
```

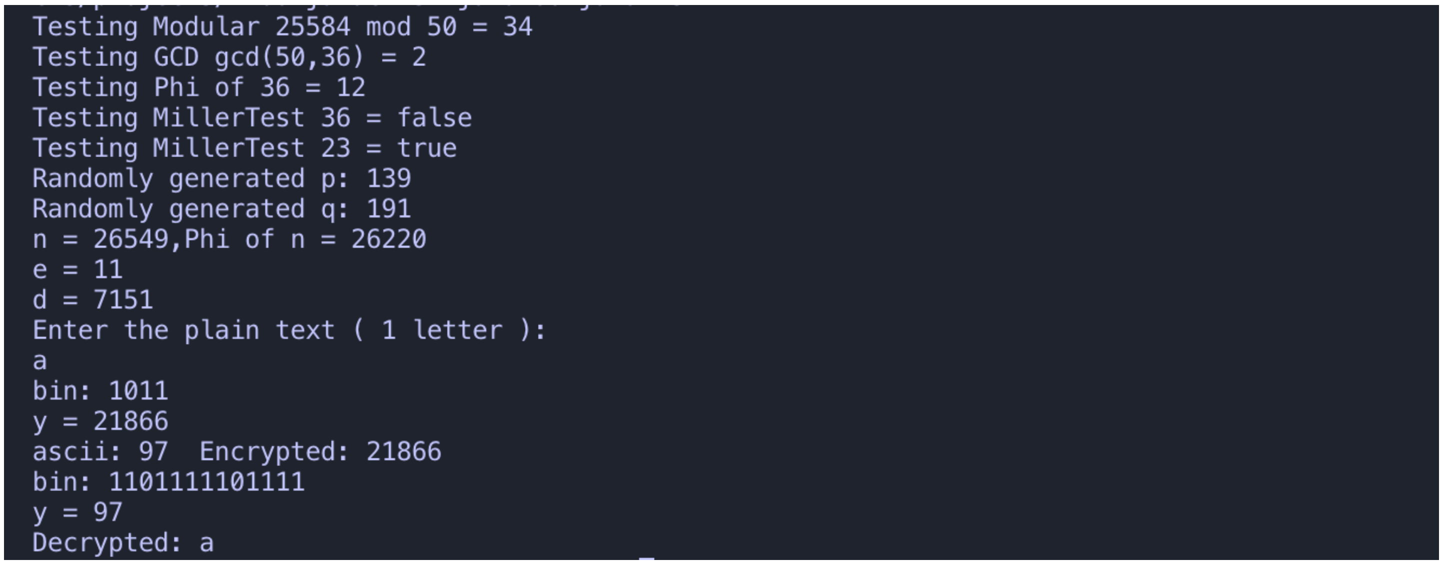
J. Test Cases

This section shows the code testing the operations above.



```
    public static void main(String[] args) {
        RSA test = new RSA();
        BigInteger a = new BigInteger("25584");
        long b = 50;
        long c = 36;
        long d = 23;
        System.out.println("Testing Modular 25584 mod 50 = "+test.modular(a, b));
        System.out.println("Testing GCD gcd(50,36) = "+test.gcd(c, b));
        System.out.println("Testing Phi of 36 = "+test.Phifunction(c));
        System.out.println("Testing MillerTest 36 = "+test.MillerTest(c));
        System.out.println("Testing MillerTest 23 = "+test.MillerTest(d));
        int prime = test.Primegeneration(256, 128);
        System.out.println("Randomly generated p: " +prime);
        int prime2 = test.Primegeneration(256, 128);
        System.out.println("Randomly generated q: " +prime2);
        long y = test.encryption(prime, prime2);
        test.decryption(y);
    }
}
```

Terminal:



```
Testing Modular 25584 mod 50 = 34
Testing GCD gcd(50,36) = 2
Testing Phi of 36 = 12
Testing MillerTest 36 = false
Testing MillerTest 23 = true
Randomly generated p: 139
Randomly generated q: 191
n = 26549,Phi of n = 26220
e = 11
d = 7151
Enter the plain text ( 1 letter ):
a
bin: 1011
y = 21866
ascii: 97  Encrypted: 21866
bin: 1101111101111
y = 97
Decrypted: a
```

K. Summary

It is the easiest one among three projects in this semester. Once getting familiar with the mathematical theorems, all operation can be done in a short time. Again, the troublesome part is still the data type – whether to make a function takes *long* or *BigInteger*, and the conversion between two types took me some thought with hesitation. I would make better decisions next time if asked to improve the implementation.