Name : Enas Morsy Mohamed Shehata

ID : 20

Micro : Assignment 2

Code :

https://drive.google.com/drive/folders/1KpUcmQ6zx6vvuzRWXDNyOYjgNSWSKXmF?usp=sharing

The assignment consists of 3 classes :

- Montgomery_M_M
- Part1
- Part2

★ Montgomery_M_M :

- **private boolean enter_to_reduction :**
  Variable to know if it enters the addition reduction step or not .

- **boolean check () :**
  To return the enter_to_reduction variable when called .

- **Calculate_unchanged_values(BigInteger N) :**
  It calculates the unchanging values which depend on the modulus only .
  R = 2^(bit length of N) .
  N’ = -1 * N.modInverse(R).
  R_Inverse = R.modInverse(N).
  It returns them as an array of BigInteger of size =3.
- **mog_mod_mul(BigInteger a, BigInteger b, BigInteger N, BigInteger R, BigInteger n_bar, BigInteger R_Inverse, int part)**

```
public BigInteger mog_mod_mul(BigInteger a, BigInteger b, BigInteger N, BigInteger R, BigInteger n_bar, BigInteger R_Inverse, int part) {
    // set enter_to_reduction to false
    enter_to_reduction = false;
    // calculate a'= a R mod N        b'= b R mod N
    BigInteger a_bar = (a.multiply(R)).mod(N);
    BigInteger b_bar = (b.multiply(R)).mod(N);
    // t= a' b'    m= t N' mod R        t= (t + mN) /R
    BigInteger t = a_bar.multiply(b_bar);
    BigInteger m = (t.multiply(n_bar)).mod(R);
    t = (t.add(m.multiply(N))).divide(R);
    BigInteger c_bar = BigInteger.valueOf(0);
    // if t <N  c'=t    else    c'= t-N
    if (t.compareTo(N) < 0)
        c_bar = t;
    else {
        if (part == 1)
            c_bar = (t.subtract(N));
        else if (part == 2) {
            enter_to_reduction = true ;
            for (int i = 0; i < 10; i++)
                c_bar = (t.subtract(N));
        }
    }
    // c = c' R^(-1) mod N
    return ((c_bar.mod(N)).multiply(R_Inverse)).mod(N);
}
```

- First it sets the enter_to_reduction variable equals false.
- Calculate
  - ➔ a’ = a*R mod N
  - ➔ b’ = b*R mod N

➔ T = a’ * b’

➔ M = t *N’ mod R

➔ T = (t+m*N)/R

- Initialize c’ equals zero .
- Check if t < N then C’ = t
- Other wise check if part 1 is calculated just put C’ = t-N
- If part 2 then set enter_to_reduction equals true and make the subtraction for 10 times without changing the result .
- Finally calculate c = c’ * R^(-1) mod N and returns it .

★ Part1:

The Part1 class contains 4 functions “ the past implementation “ with modification :

- Function to calculate the time in msec of decryption in the **main** function .
- Instead of multiplication in the **modExp** function , an object from the Montgomery_M_M class is taken to call the function **mog_mod_mul** which calculates the result instead of multiplication .

Challenges i faced was to decrease the time as much as possible so i observed that R , N’ and (R_inverse_ mod N) depend only on the N so they can be calculated once using **Calculate_unchanged_values** function and sent as a parameter to function **mog_mod_mul** .

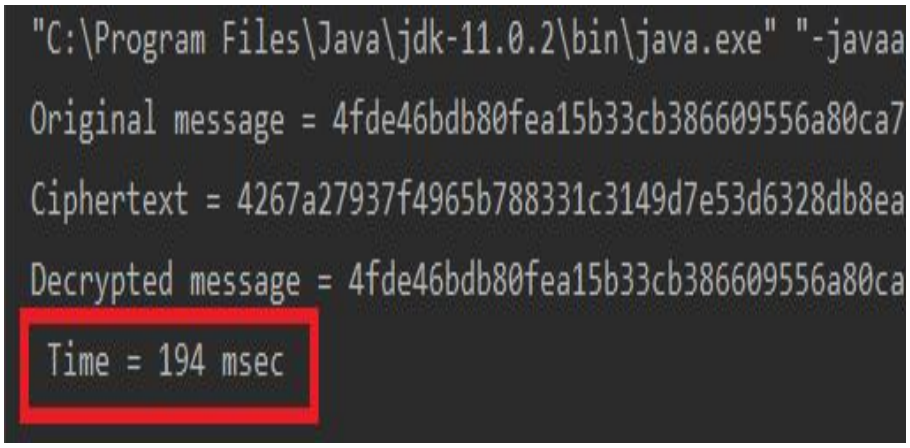
Important functions :

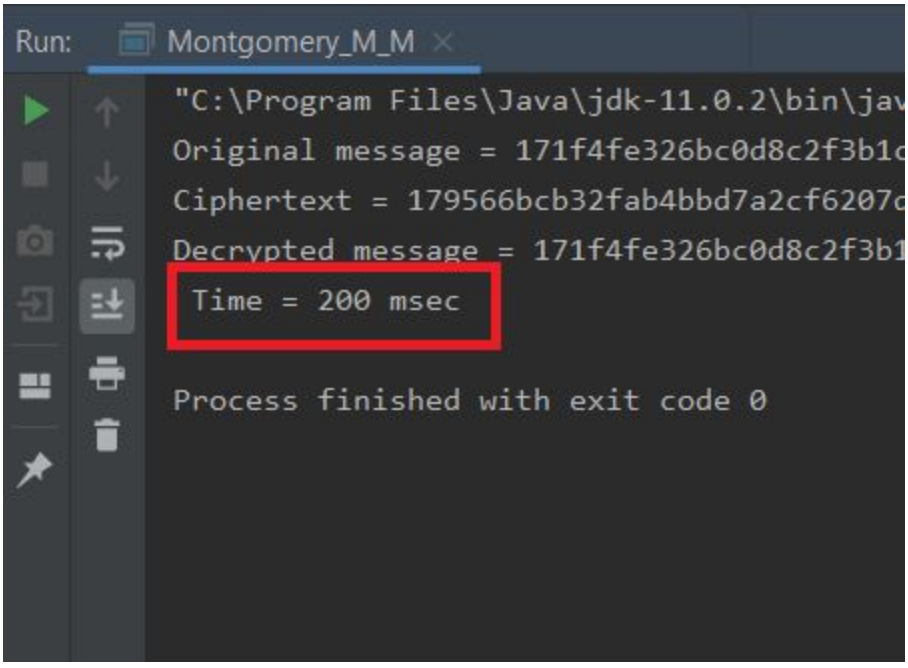
```
private static  BigInteger modExp(BigInteger a, BigInteger exponent, BigInteger N) {
    Montgomery_M_M obj =new Montgomery_M_M();
    BigInteger [] temp =obj.Calculate_unchanged_values(N);
    // Note: This code assumes the most significant bit of the exponent is 1, i.e., the exponent is not zero.
    BigInteger result = a;
    int expBitlength = exponent.bitLength();
    for (int i = expBitlength - 2; i >= 0; i--) {
        result = obj.mog_mod_mul(result,result,N,temp[0],temp[1],temp[2], part: 1);
        if (exponent.testBit(i)) {
            result = obj.mog_mod_mul(result,a,N,temp[0],temp[1],temp[2], part: 1);
        }
    }
    return result;
}
```

- First object from Montgomery is taken .
- “Calculate _unchanged_values “ is called to calculate R,N’ and R_inverse_ mod N and store them in array temp to be calculated once .
- Initialize the result equals to a.
- Loop on the bit length of the exponent from **n-2** to **0** .
- Call the “ mog_mod_mul “ to calculate result* result mod N.
- If the bit equals one , calculate result*a mod N using mog .
- Finally returns the result .

This function is called twice , one from the encrypt function to calculate m^(e) mod N and the other time from the decrypt method to calculate c^(d) mod N.

Sample Runs :


"C:\Program Files\Java\jdk-11.0.2\bin\java.exe" "-javaa
Original message = 4fde46bdb80fea15b33cb386609556a80ca7
Ciphertext = 4267a27937f4965b788331c3149d7e53d6328db8ea
Decrypted message = 4fde46bdb80fea15b33cb386609556a80ca
Time = 194 msec


Run:
Montgomery_M_M
"C:\Program Files\Java\jdk-11.0.2\bin\jav
Original message = 171f4fe326bc0d8c2f3b1c
Ciphertext = 179566bcb32fab4bbd7a2cf6207c
Decrypted message = 171f4fe326bc0d8c2f3b1
Time = 200 msec
Process finished with exit code 0

★ Part 2 :

The part2 contains 4 functions :

- **The main function :**
  - ➢ Declare the lengths in an array of size = 6 .
  - ➢ Loop on this array and for each length call the Calculate_for_each_length function .
  - ➢ Get the result and calculate how many ones are over 20 times it succeeded and print it .

- **Calculate_for_each_length(BigInteger d) :**
  - ➢ Initialize result array of size = 20 .
  - ➢ Loop on this array and each time call Calculate_experiment and store the result either 0 or 1 in the array .

- **boolean modExp(BigInteger a, BigInteger exponent, BigInteger N , boolean bit) :**

  Looks like the one in part 1 but with little difference , it takes a **boolean bit** as a parameter which equals true if the target bit is assumed to be one and false otherwise .
  - ➢ An object from Montgomery _M_M class is taken .
  - ➢ Calculate the unchanged values .
  - ➢ Initialize the result equals a .
  - ➢ Either the target bit is assumed the square step should calculated so set result = result* result mod N using mog .

- ➢ If the target bit is assumed to be one set
  result =result *m mod N .
- ➢ Start looping on the exponent bit length from **n-3** to **0** .
- ➢ Only for the 3rd most bit check if it enters the reduction save the boolean res to be returned when finished .
- ➢ Continue looping as part 1 .
- ➢ Return res .

- **BigInteger average (LinkedList<BigInteger> list) :**
  - ➢ Loop on the list to get the sum of its element .
  - ➢ Calculate average = (sum)/ size of list .
  - ➢ Return average .

- **int Calculate_experiment(BigInteger d) :**
  - ➢ Initialize 4 linked lists to store the time .
    If the target bit assumed to equal 1
    T 11 -> it enters the reduction.
    T 10 -> it doesn't enter .
    If the target bit assumed to equal 0
    T 01 -> it enters the reduction.
    T 00 -> it doesn't enter .
  - ➢ Call the mod Exp with bit = true and if it returns true, store the time in T11. otherwise , store it in T 10 .
  - ➢ Call the mod Exp with bit = false and if it returns true, store the time in T01. otherwise , store it in T 00 .
  - ➢ Calculate the average of each linked list .

★ By printing the average values i noticed that the function of time isn't so accurate so i tried to fix this with this check .

- ➢ Check if avg 11 > avg 10 and the abs(avg 00 - avg 01) is less than the abs (avg 11 - avg 10) , return 1 as succeeded
- ➢ Otherwise return 0 .

Sample Runs :

Length	No of succeeded
3	8
5	9
10	11
20	9
50	13
100	14

I observed that the output is not accurate but i think this because of my laptop as i hear his sound cannot handle all these samples .