# Discrete Mathematics

## Assignment 2: Number Theory

Name : Dahlia Chehata Mahmoud

28

# Question 1:

# Problem Statement:

Implement a subroutine that takes three positive integer arguments ($a$, $b$, $n$) and returns the value of ($a$^b mod $n$), where the arguments are represented by about 100 decimal digits. Com- pare the execution time of that subroutine implementation using the four different approaches discussed at the lecture. Draw a 2D line chart for the fours implementations, where the x-axis represents the integer size and the y-axis represents the execution time.

# Code Snippet/Algorithm used :

```
package numberTheory;

import java.math.BigInteger;
import java.util.Scanner;
 /**
  * calculate pow (a,b) mod n
  * @author select
  *
  */
public class AexpBmodN {
     /**Ů
      * first method
      * @param a
      * @param b
      * @param n
      * @return
      */
    static BigInteger modulo1(BigInteger a, BigInteger b, BigInteger n) {
        BigInteger sol=BigInteger.ONE;
        for (BigInteger i=BigInteger.ONE;i.compareTo(b)<=0
                ; i = i.add(BigInteger.ONE)) {
            sol=sol.multiply(a);
        }
        sol=sol.mod(n);
        return sol;

    }


```

```java
    /**
     * second method
     * @param a
     * @param b
     * @param n
     * @return
     */
    static BigInteger modulo2(BigInteger a, BigInteger b, BigInteger n) {
        BigInteger sol=BigInteger.ONE;
        for (BigInteger i=BigInteger.ONE;i.compareTo(b)<=0
                ; i = i.add(BigInteger.ONE)) {
            sol=sol.multiply(a);
            sol=sol.mod(n);
        }
        return sol;

    }
    static BigInteger mulmod (BigInteger a, BigInteger b, BigInteger n)
    {
        BigInteger x=BigInteger.ZERO;
        BigInteger y=a.mod(n);
        while(b.compareTo(BigInteger.ZERO) > 0){
            if(b.mod(BigInteger.valueOf(2)) .equals(BigInteger.ONE) )
                x = (x.add(y)).mod(n);
            y = (y.multiply(BigInteger.valueOf(2))).mod(n);
            b =b.divide(BigInteger.valueOf(2));
        }
        return x.mod(n);
    }
    /**
     * third method
     * @param a
     * @param b
     * @param n
     * @return
     */
    static BigInteger modulo3(BigInteger a, BigInteger b, BigInteger n) {
        BigInteger x=BigInteger.ONE,y=a;

            while(b.compareTo(BigInteger.ZERO) > 0)
            {
                if (b.mod(BigInteger.valueOf(2)) .equals(BigInteger.ONE))
                    x=mulmod(x,y,n);
                y=mulmod(y,y,n);
                b =b.divide(BigInteger.valueOf(2));         }
            return x.mod(n);

    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a,b,n respectively to calculate (a exponential b) mod n");
        BigInteger a = sc.nextBigInteger();
        BigInteger b = sc.nextBigInteger();
        BigInteger n = sc.nextBigInteger();
        /**
         *
         */
        long startTime1 = System.nanoTime();
        System.out.println( "The answer is :"+modulo1(a, b, n));
        long stopTime1 = System.nanoTime();
        System.out.println("Execution time for first method 1 in nanoseconds is : " +
        (stopTime1 - startTime1));
        /**
```

```
          *
          */
         long startTime2 = System.nanoTime();
         System.out.println("The answer is :"+modulo2(a, b, n));
         long stopTime2 = System.nanoTime();
         System.out.println("Execution time for first method 2 in nanoseconds
           is : " +
                        (stopTime2 - startTime2));
               /**
          *
          */
          long startTime3 = System.nanoTime();
          System.out.println("The answer is :"+modulo3(a, b, n));
          long stopTime3 = System.nanoTime();
         System.out.println("Execution time for first method 3 in nanoseconds
   is : " +
                        (stopTime3 - startTime3));
            }

        }
```

## Used data structures:

There is no data structure used, only for loop to calculate the a to the power b modulus n according to the 3 methods algorithms explained in the lecture

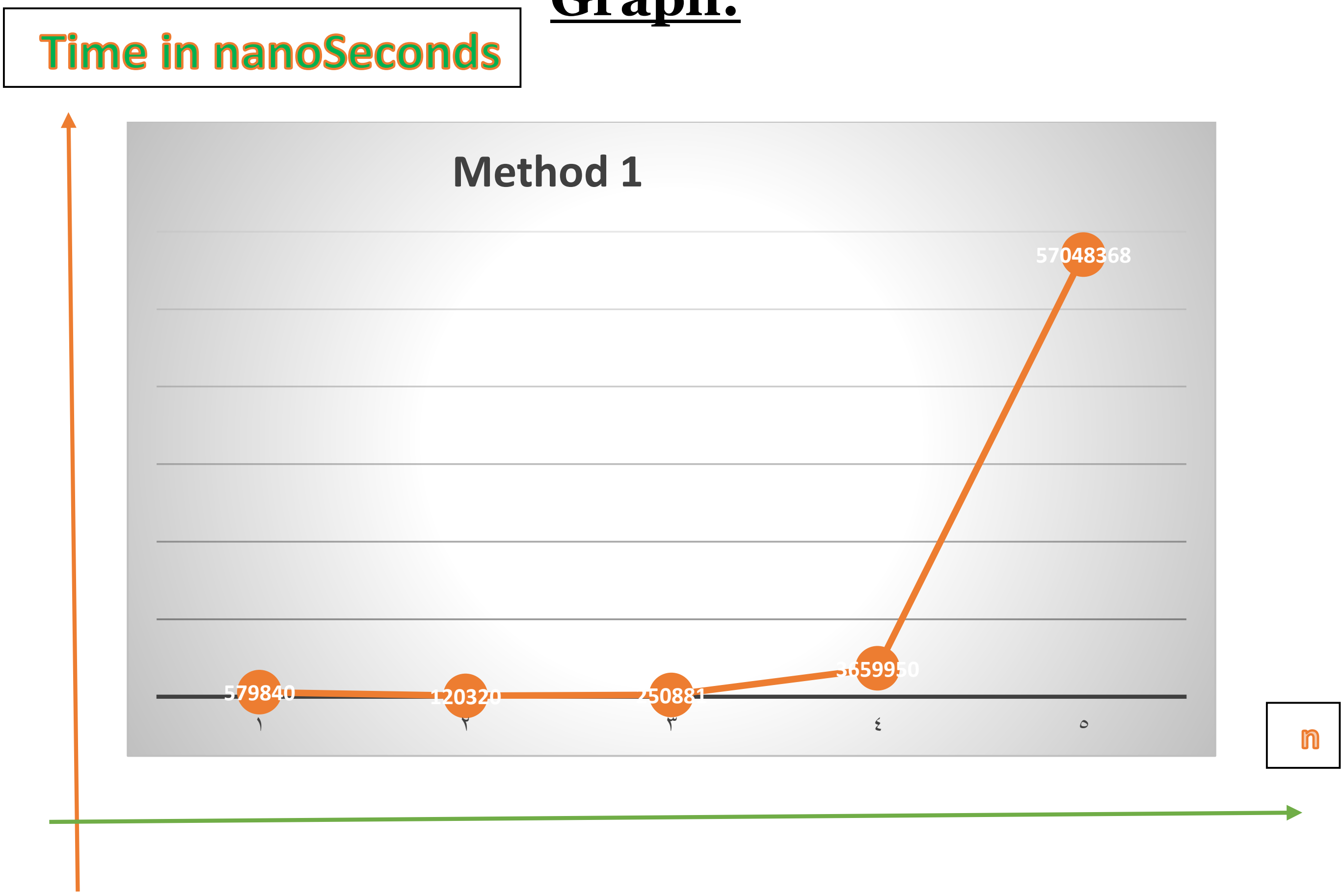
## Assumptions, details and design decisions:

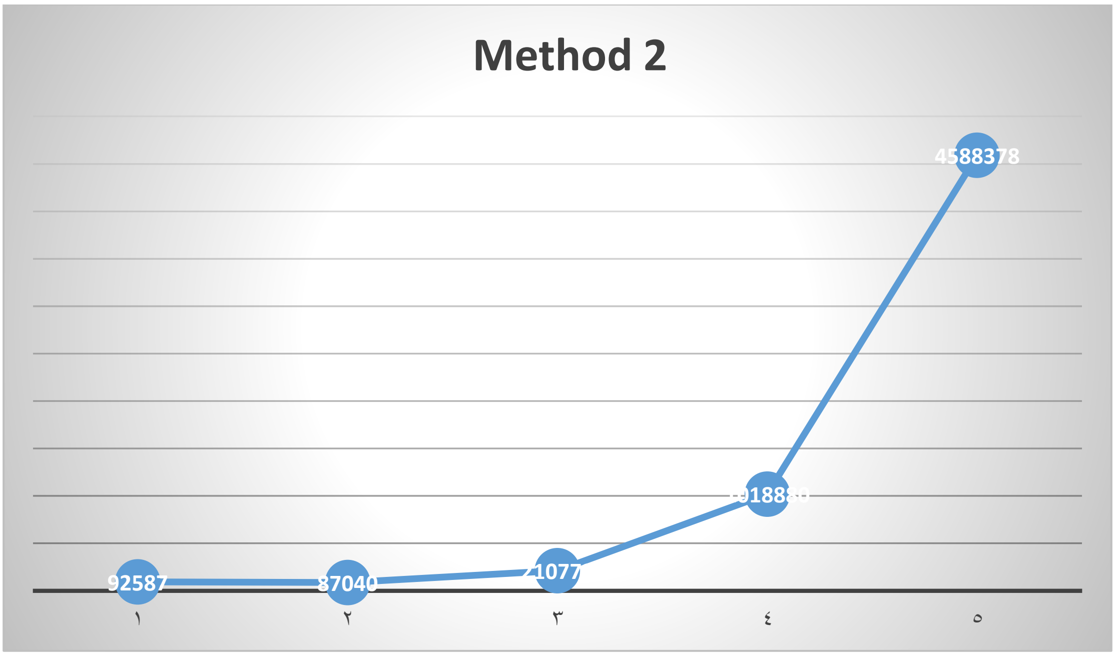
The code is written in java. The BigInteger class is used to provide 100 decimal digits length as required.
System.currentTime () method in Java was used to calculate the execution time for each method as well as System.nanoTime() method, as it is shown in the code and the sample runs

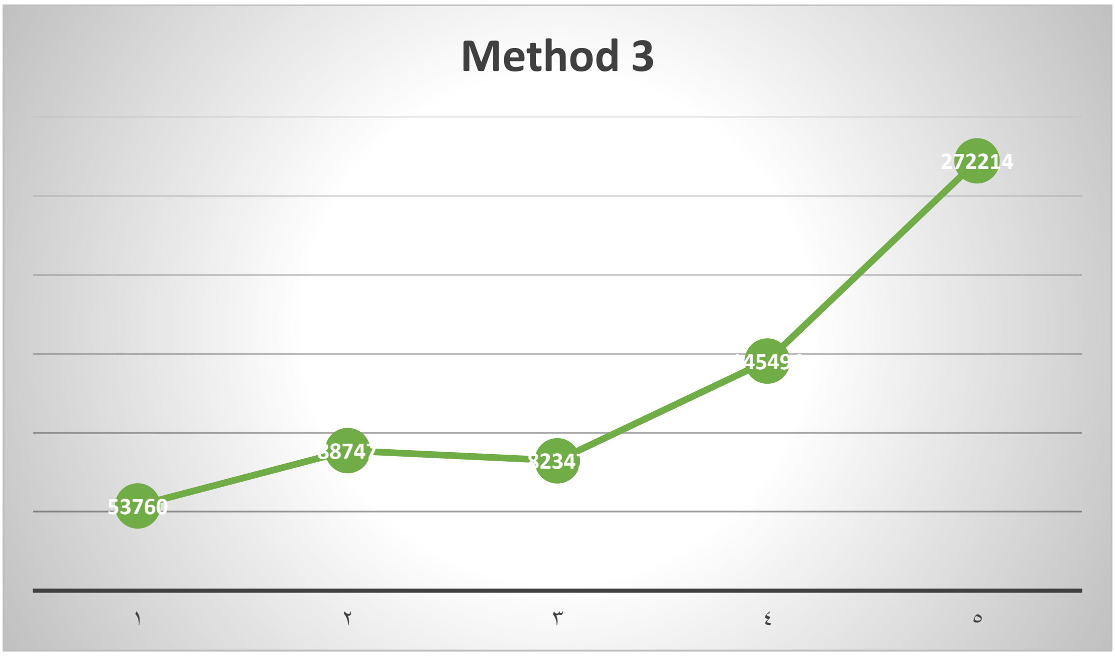
## Sample Runs:

```
AexpBmodN [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (Dec 9, 2016, 6:58:07 PM)
1000
1000  1000
The answer is :0
Execution time for first method 1 in nanoseconds is : 3629657
The answer is :0
Execution time for first method 2 in nanoseconds is : 747094
The answer is :0
Execution time for first method 3 in nanoseconds is : 121600

Enter a,b,n respectively to calculate (a exponential b) mod n
20 10 13
The answer is :4
Execution time for first method 1 in nanoseconds is : 229120
The answer is :4
Execution time for first method 2 in nanoseconds is : 58880
The answer is :4
Execution time for first method 3 in nanoseconds is : 172374

Enter a,b,n respectively to calculate (a exponential b) mod n
123456
169872
100000
The answer is :65536
Execution time for first method 1 in nanoseconds is : 12152651650
The answer is :65536
Execution time for first method 2 in nanoseconds is : 34761843
The answer is :65536
Execution time for first method 3 in nanoseconds is : 974934
```

## Graph:






*****Note:***
***all the readings are in the chart:***
***N was taken from range 1 to 5 on X axis***
***And the time readings in nanoseconds are on the Y axis***

Conclusion:
From the graph and by comparing the execution time, the third method is the most efficient one

# Question 2

## Problem Statement:

Implement the extended Euclids algorithm that finds the multiplicative inverse of $a$ mod $n$, where $a$ and $n$ are positive integers that are relatively prime (i.e. gcd($a$, $n$) = 1).
The multiplicative inverse $b$ is a positive integer that is uniquely determined such that ($ab$) mod $n$ = 1.

## Code Snippet/Algorithm used:

```
package numberTheory;

import java.util.Scanner;
/**
 * Extended Euclidean Algorithm to calculate modular multiplicative inverse

 * @author select
 *
 */
public class ExtendedEuclideanAlgorithm {

    /**
     * Function to find modulo inverse of a
     * @param a
     * @param m
     * @return
     */
    public static int modInverse(int a, int m)
    {
        int []sol= ExtendedEuclid(a,m);
        int g =sol[0];
        if (g == 1) {
            // m is added to handle negative x
             return (sol[2]%m + m) % m;
        }
        return -1;
    }
     /**
      * This function will perform the Extended Euclidean algorithm
        to find the GCD of a and b.  We assume here that a and b
        are non-negative (and not both zero).  This function also
        will return numbers j and k such that
               d = j*a + k*b
        where d is the GCD of a and b.
      * @param a
      * @param b
      * @return
      */
    public static int[] ExtendedEuclid(int a, int b)
    {
```

```
        int[] ans = new int[3];

        if (a == 0)  {  /*  If a = 0, then we're done...  */
            ans[0] = b; //b
            ans[1] = 1; //y
            ans[2] = 0; //x
        }
        else
            {     /*  Otherwise, make a recursive function call  */
               ans = ExtendedEuclid (b%a, a);
               int temp = ans[1] - ans[2]*(b/a);
               ans[1] = ans[2];
               ans[2] = temp;
            }

        return ans;
    }

    /**
     * Driver Program
     * @param args
     */
    public static  void main(String[]args)
    {
        int a ,m;
        Scanner sc=new Scanner(System.in);
        System.out.println(" Enter a and m respectively to calculate the modular "
                + "multiplicative inverse of a mod m" );
        a=sc.nextInt();
        m=sc.nextInt();
        if (modInverse(a, m)==-1)
        System.out.println("Inverse doesn't exist");
        else
            System.out.println( "Modular multiplicative inverse is " + modInverse(a, m));
    }
}
```

## Assumptions, details and design decisions:

The code uses the Euclidean Algorithm to compute the GCD of the 2 numbers first ,if their GCD is equal to 1 ,it computes its multiplicative inverse else the number has no multiplicative inverse

## Sample Runs:

```
ExtendedEuclideanAlgorithm [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (Dec 9, 2016, 7:27:43 PM)
 Enter a and m respectively to calculate the modular multiplicative inverse of a mod m
15 20
Inverse doesn't exist

 Enter a and m respectively to calculate the modular multiplicative inverse of a mod m
200 10
Inverse doesn't exist

 Enter a and m respectively to calculate the modular multiplicative inverse of a mod m
11 4
Modular multiplicative inverse is 3

 Enter a and m respectively to calculate the modular multiplicative inverse of a mod m
121 12
Modular multiplicative inverse is 1

 Enter a and m respectively to calculate the modular multiplicative inverse of a mod m
100000 30
Inverse doesn't exist

 Enter a and m respectively to calculate the modular multiplicative inverse of a mod m
18 7
Modular multiplicative inverse is 2

 Enter a and m respectively to calculate the modular multiplicative inverse of a mod m
123564 236
Inverse doesn't exist

 Enter a and m respectively to calculate the modular multiplicative inverse of a mod m
2 6
Inverse doesn't exist
```

# 3 Question 3:

## Problem Statement:

Implement the unique mapping of Chinese Remainder Theorem that is stated as follows:

Let $M = m1 \times m2 \times \cdots \times mk$, such that for every $i \neq j$, gcd($mi$,$mj$ )=1 (i.e. relatively prime) There is a bijection $A \leftrightarrow (a1, a2, \ldots, ak)$

where $A \in ZM$ and the k-tuple $(a1, a2, \ldots, ak) \in Zm1 \times Zm2 \times \cdots \times Zmk$

The mapping from $A$ to $(a1, a2, \ldots, ak)$ is such that $ai = A \bmod mi$

For the reverse mapping from $(a1, a2, \ldots, ak)$ to $A$

$A$ = Σki ($aiMiMi$−1 mod $M$ ), where

$Mi = mi \times m2 \times \cdots \times mi{-}1 \times mi{+}1 \times \cdots \times mk$

*Mi -1 = multiplicative inverse of Mi mod mi (use implementation of Question 2)*

Compare the execution time of addition and multiplication in the two domains ($ZM$ , $Zm1 \times$

$Zm2 \times \cdots \times Zmk$).

Similarly, draw a 2D line chart of the integer size versus the execution time in the two cases. Assume $M$ factoring to $m1 \times m2 \times \cdots \times mk$ is given.

## Code Snippet/Algorithm used:

```
package numberTheory;

import java.util.Scanner;
/**
 * Number theory :Chinese remainder theorem
 * @author select
 *
 */
public class ChineseRemainderTheorem {
    /**
     * convert A to a1,a2,a3,....ak
     * @param num
     * @param A
     * @param k
     * @return
     */
    public static int []CRTMapping (int num[],int A,int k) {
        int [] res= new int [k];
         for (int i=0;i<k;i++) {
             res[i]=A % num[i];
         }
        return res;
    }
    /**
     * convert a1,a2,a3,....ak to A
     * @param num
     * @param rem
     * @param k
```

```java
     * @return
     */
    public static int ReverseCRTMapping(int num[], int rem[], int k)
    {
        // Compute product of all numbers
        int prod = 1;
        for (int i = 0; i < k; i++)
            prod *= num[i]; //M

        // Initialize result
        int result = 0;

        // Apply above formula
        for (int i = 0; i < k; i++)
        {
            int pp = prod / num[i];
            if (ExtendedEuclideanAlgorithm.modInverse (pp, num[i])!=-1)
            result += rem[i] * ExtendedEuclideanAlgorithm.modInverse (pp, num[i]) * pp;
        }

        return result % prod;
    }
    /**
     * Drive Program
     * @param args
     */
    public static  void main(String[]args)
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter the number k");
        int k=sc.nextInt();
        System.out.println("Enter m1,m2,m3....mk");
        int num[] =new int [k];
        for (int i=0;i<k;i++) {
            num[i]=sc.nextInt();
        }
               int A,B,M = 1;

                System.out.println("Enter a1,a2,a3....ak to calculate A:");

                int remA[] =new int[k];
                for (int i=0;i<k;i++)
                    remA[i]=sc.nextInt();
                System.out.println("Enter b1,b2,b3....bk to calculate B:");

                int remB[] =new int[k];
                for (int i=0;i<k;i++)
                    remB[i]=sc.nextInt();

                long s1=System.nanoTime();

                A= ReverseCRTMapping(num,remA,k);
                System.out.println("A =" + A );

                B=ReverseCRTMapping(num,remB,k);
                System.out.println("B =" + B);


                 for (int i = 0; i < k; i++)
                    M*= num[i];

                System.out.println("A+B mod n =");

                System.out.println("first method :"+(( A+B )% M));
                long t1=System.nanoTime();
```

```
                System.out.println("Execution time :"+(t1-
s1)+" nanoseconds");

                long s2=System.nanoTime();
                int []remSol=new int[k];
                for (int i=0;i<k;i++) {
                    remSol[i]=((remA[i]+remB[i])% num[i]);
                }
                System.out.println("Second method :"+ReverseCRTMapping(num,r
emSol,k));
                        long t2=System.nanoTime();
                        System.out.println("Execution time :"+(t2-
s2)+" nanoseconds");


            }
        }
```

## Assumptions, details and design decisions:

- The CRT mapping function and the reverse mapping functions, both are implemented as shown in the code snippet

- In the example in the below screenshot,

  The calculation using the first method:

  A+B mod (n) → referred as (method 1 )in the chart,

  Took 1041921 nanoseconds.

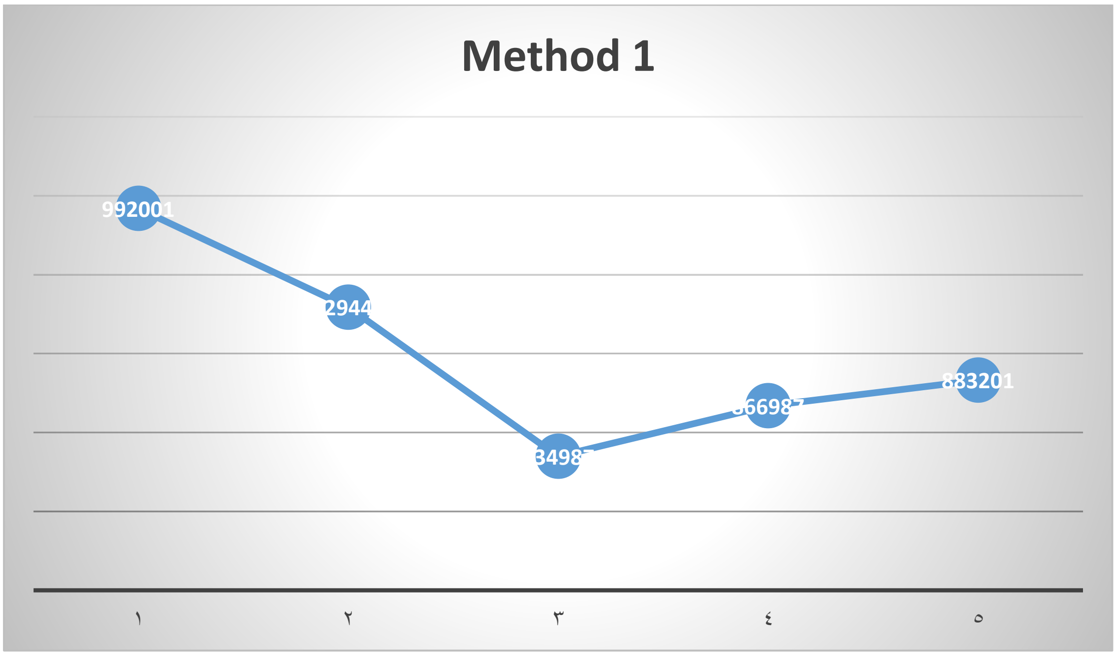
  While using the second method: → (Method 2)
  A+B mod n=
  [(a1+b1) mod m1 +.....+ (ak+bk) mod mk]
  The execution time was only 55893 nanoseconds

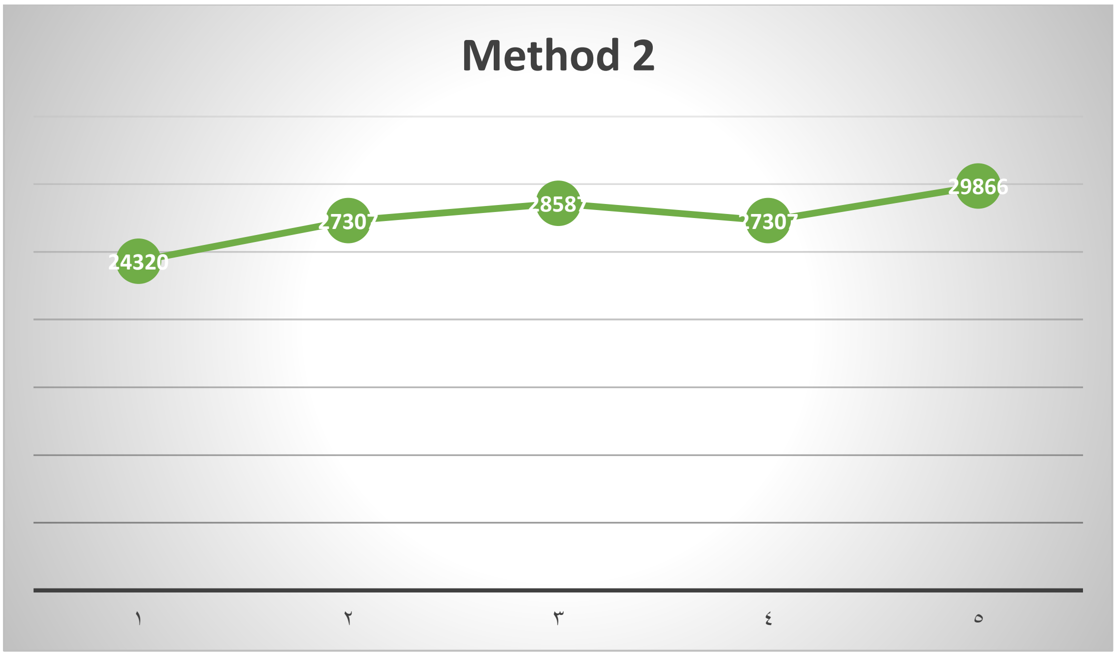
  Denoting that the CRT algorithm is faster

# Sample Runs:

```
<terminated> ChineseRemainderTheorem [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (Dec 9, 2016, 7:32:46 PM)
Enter the number k
3
Enter m1,m2,m3....mk
3
5
8
Enter a1,a2,a3....ak to calculate A:
1
0
4
Enter b1,b2,b3....bk to calculate B:
0
2
7
A =100
B =87
A+B mod n =
first method :67
Execution time :1041921 nanoseconds
Second method :67
Execution time :55893 nanoseconds
```

## Graph:






# Refrences:

*Geeksforgeeks site

*Wikipedia

*Stackoverflow

------------------------------------------------------------------------