Discrete Maths

Assignment 2

Name / Seif Mohamed Mahmoud Gneedy
ID / 18010834

➢ Problem statement :

Implement 4 procedures(naive1 ,naive2,fast exponentiation -iterative and recursive-) to compute : $b^n \bmod m$ and comparing the execution time of each algorithm with the increase of bits representing an integer and when the overflow happens.

➢ Used data structure :

Used long representation -64 bit- to represent the integers just to avoid overflow as much as possible.

➢ Algorithms used :

1. The first naive algorithm :

```java
// (b^n) mod m
/*
 * In this algorithm, we compute (b^n) first then mod m
 */
static long naive1(long b, long n, long m) {
    long c = 1;
    for (long i = 1; i <= n; i++) {
        c = Math.multiplyExact(c, b);
    }
    c %= m;
    return c;
}
```

2. The second naive algorithm :

```java
/*
 * In this algorithm, we compute the mod after every multiplication of b to
 * avoid overflow
 */
static long naive2(long b, long n, long m) {
    long c = 1;
    for (long i = 1; i <= n; i++) {
        c = (Math.multiplyExact(c, b)) % m;
    }
    return c;
}
```

3. Fast exponentiation iterative algorithm :

```java
/*
* We use the property of binary representation of n to reduce the time
* of the algorithm
*/
static long fastExponentIterative(long b, long n, long m) {
    long result = 1;
    long power = b % m;
    while (n > 0) {
        if (n % 2 == 1)
            result = (Math.multiplyExact(result, power)) % m;
        n = n >> 1;
        power = (Math.multiplyExact(power, power)) % m;
    }
    return result;
}
```

4. Fast exponentiation recursive algorithm :

```java
/*
 * We use divide and conquer technique in this recursive approach As if b is
 * even : (b^n)%m = ((b^n/2)*(b^n/2))%m and if b is odd : (b^n)%m =
 * (b*(b^(n-1)))%c
 */
static long fastExponentRecursive(long b, long n, long m) {
    if (b == 0)
        return 0;
    if (n == 0)
        return 1;
    long res;
    if (n % 2 == 0) {
        res = fastExponentRecursive(b, n / 2, m);
        res = (Math.multiplyExact(res, res)) % m;
    } else {
        res = ((b % m) * fastExponentRecursive(b, n - 1, m) % m) % m;
    }
    // res is negative and |res|<m so we add m to get the positive mod
    return (res + m) % m;
}
```

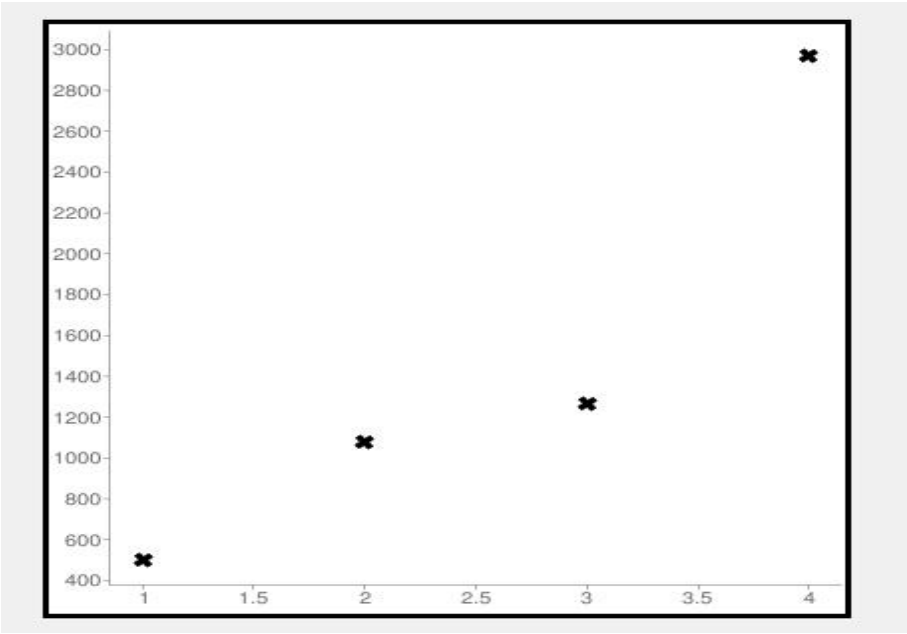
➢ <u>Assumptions and details :</u>

1. Assumptions :
   * In time calculation ,I'm increasing the number of bits for b,n,m with the same amount
   *I'm getting the time of every run on the same bits 10 times and get the average time to get more accurate result.

*I'm saving the results in (.txt) files in certain pattern to be compatible with the site I'm using to draw it and copying their content to the site below then, it produces the graph.
alcula.com/scatter-plot

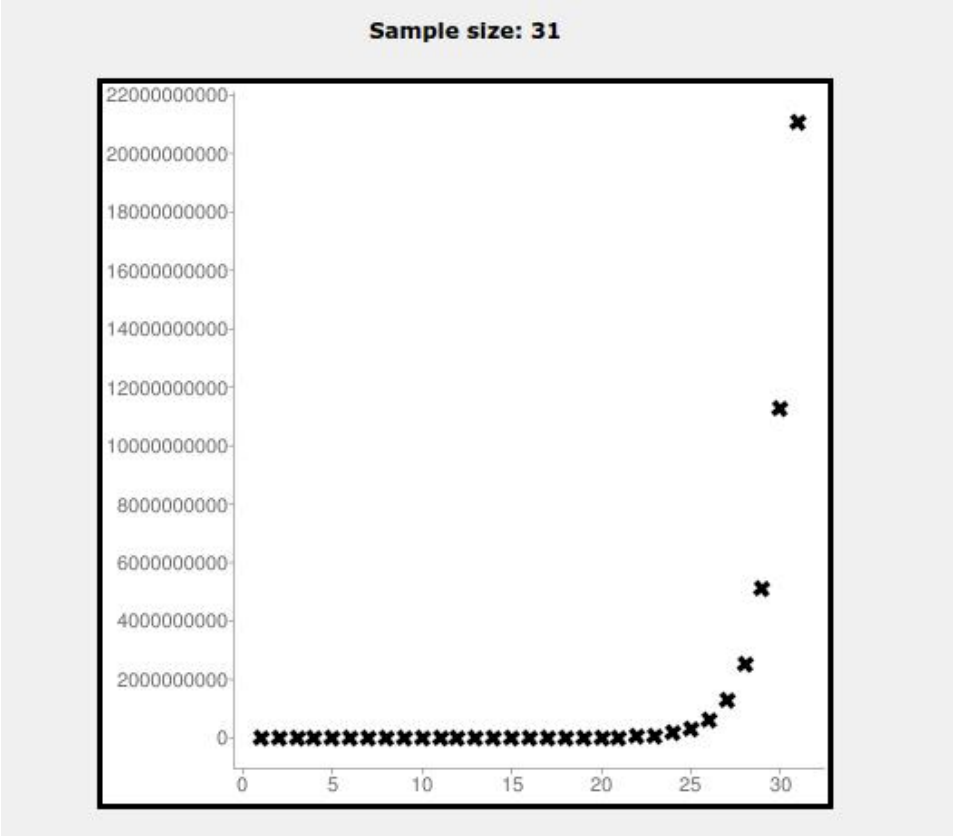
2. Details -overflow and execution time in each algorithm- :
    1. Naive1 :
       The execution time is O(n) so it takes long time and overflows very fast as the bits in (C) increases 64-bits

       The graph is between number of bits-till overflow occurs- and time :



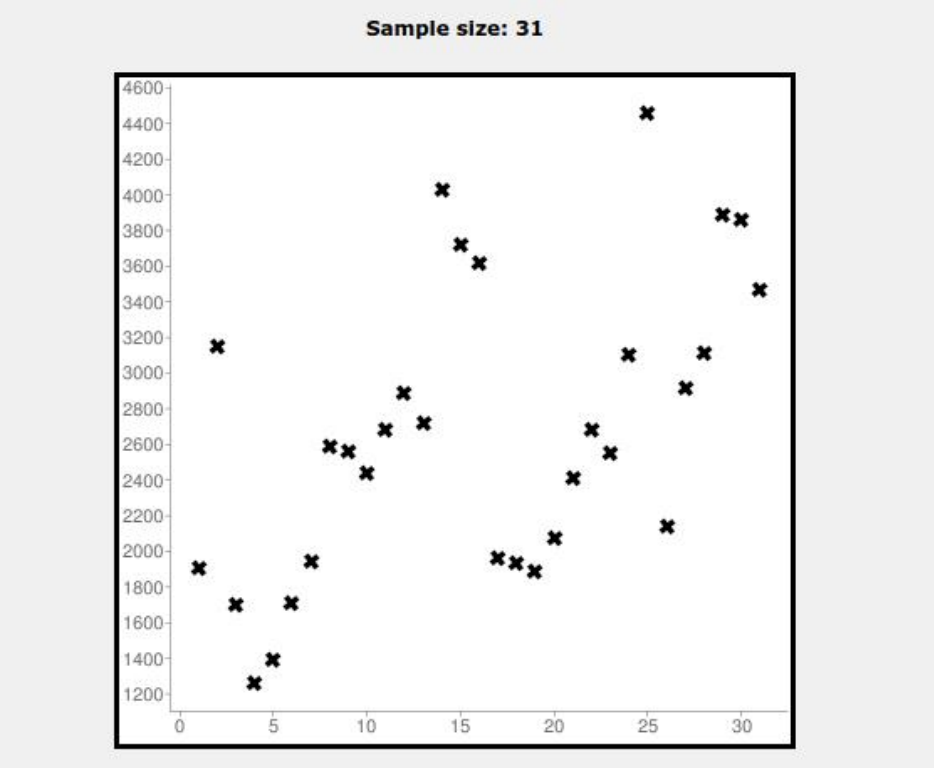
    2. Naive2 :
       The execution time is O(n) but it doesn't overflow fast as it overflows if (c*b) exceeds ((2^64)-1).

       The graph is between number of bits-till overflow occurs- and time :



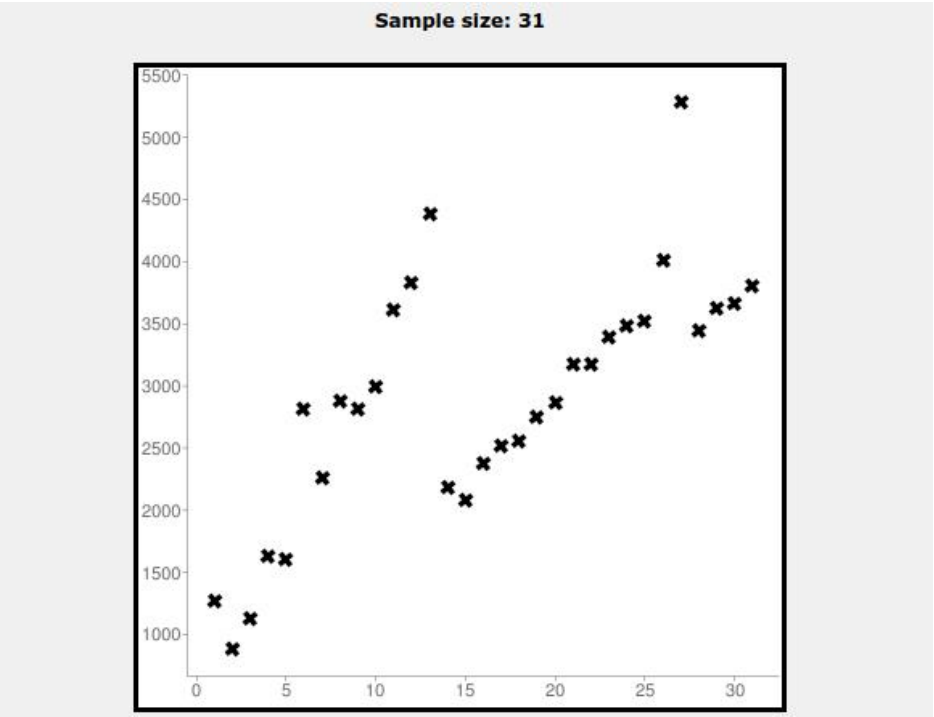
3. Fast exponentiation iterative :
   The execution time is O(log(n)) as its iterations depends on the number of bits representing the number (n) and it overflows if m is very large as when computing power after taking mod m -it doesn't effect as it's bigger than power- it exceeds ((2^64)-1).

   The graph is between number of bits-till overflow occurs- and time :



4. Fast exponentiation recursive :
   The time execution is O(log(n)) as it follows divide and conquer technique and it overflows if the product of (b^n/2)*(b^n/2) and (a*(b^(n-1)) exceeds ((2^64)-1).

   The graph is between number of bits-till overflow occurs- and time :



➢ Sample runs :

Without overflow :

```
The program is used to compute ((b^n)mod m) with 4 different methods
Time data is saved as .txt files
Enter b : 8
Enter n : 10
Enter m : 7
1       (naive1)
1       (naive2)
1       (Fast Exponentiation Iterative)
1       (Fast Exponentiation Recursive)
```

With overflow in naive1 :

```
The program is used to compute ((b^n)mod m) with 4 different methods
Time data is saved as .txt files
Enter b : 3
Enter n : 644
Enter m : 645
Overflow occured with naive1 method
36      (naive2)
36      (Fast Exponentiation Iterative)
36      (Fast Exponentiation Recursive)
```

With overflow in all methods :

```
The program is used to compute ((b^n)mod m) with 4 different methods
Time data is saved as .txt files
Enter b : 876826346332864
Enter n : 3463276478326
Enter m : 2346236482736
Overflow occured with naive1 method
Overflow occured with naive2 method
Overflow occured with Fast Exponentiation Iterative method
Overflow occured with Fast Exponentiation Recursive method
```