Discrete Structures Assignment 2

Fast Exponentiation

Name: Abdelrhman Abdelfattah Kassem

ID: 18010948

Problem Statement:

Implement fast exponentiation in 4 versions 2 naïve, an iterative and a recursive version, then compare the execution time of each with the increase of number of bits representing the integers, also report when the operation overflows.

Algorithms Used To calculate c=ab % m: -

1st naïve:

```
c=1
for i=1 to b
        c=c*a
c=c % m
return c
```

2nd naïve :

```
c=1
for i=1 to b
        c=c*a % m
return c
```

3rd Iterative:

```
a = a % m
while (b>0)
        //tests the least significant bit of b
        if(b & 1)
                c = c * a % m
        a = a * a % m
        //shifts the bits of b to work on the next bit
        b = b >> 1
return c
```

4th Recursive:

```
c = rec(a,b,m)
rec (a,b,m){
        if (b==0)
                return 1
        c = rec(a,b/2,m)
        c = c * c % m
        if(b&1==0)
                return c
        else
                return c * (a%m) % m
}
```

Code details and assumptions:

I implemented a java application that uses long to store user inputs, which will empower the use of up to 64 bits to store the numbers. The user enters the three numbers and the program computes the output and the execution time of each method with an overflow alert for the first two methods and input rejection in case of using more that the limit of 64-bit numbers (2^{63}- 1).

Sample runs:

```
Enter the three integers A,B,M to find A^B % M
3 644 645
OVERFLOW
Method 1 result: 220
Elapsed time: 16200 ns


Method 2 result: 36
Elapsed time: 44500 ns


Method 3 result: 36
Elapsed time: 1700 ns


Method 4 result: 36
Elapsed time: 7200 ns
```

```
Enter the three integers A,B,M to find A^B % M
3333 3333 100000
OVERFLOW
Method 1 result: -13067
Elapsed time: 68900 ns


Method 2 result: 75413
Elapsed time: 139800 ns


Method 3 result: 75413
Elapsed time: 1500 ns


Method 4 result: 75413
Elapsed time: 2100 ns
```

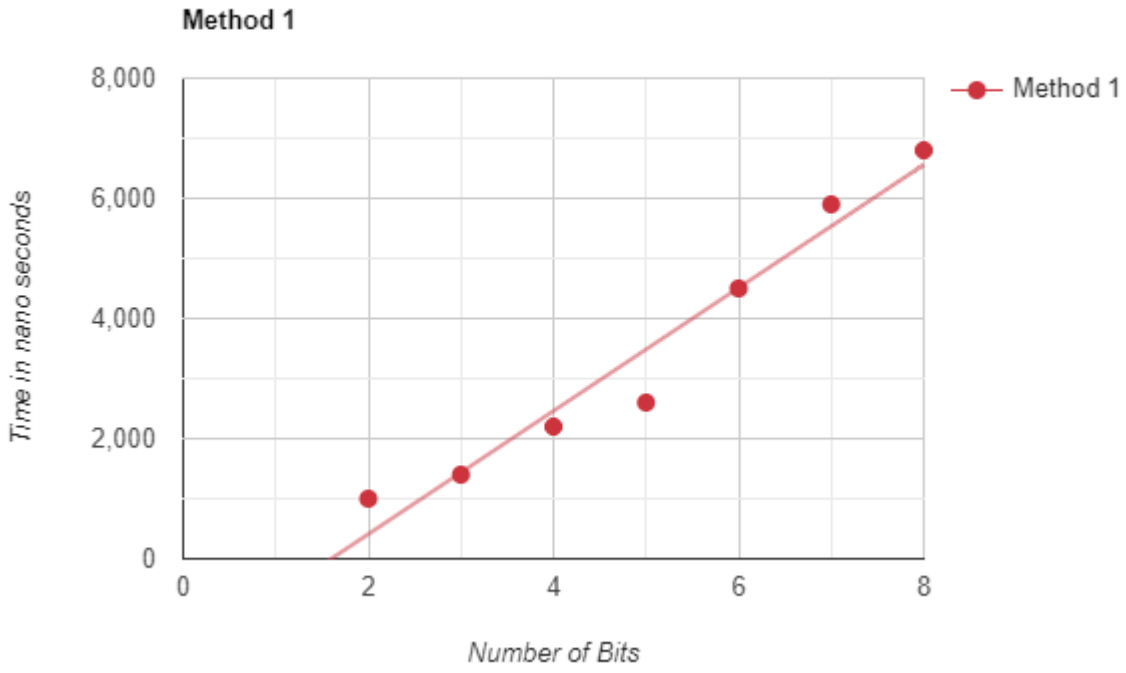
```
Enter the three integers A,B,M to find A^B % M
128 256 10000
OVERFLOW
Method 1 result: 0
Elapsed time: 13600 ns


Method 2 result: 8896
Elapsed time: 8900 ns


Method 3 result: 8896
Elapsed time: 1200 ns


Method 4 result: 8896
Elapsed time: 1700 ns
```

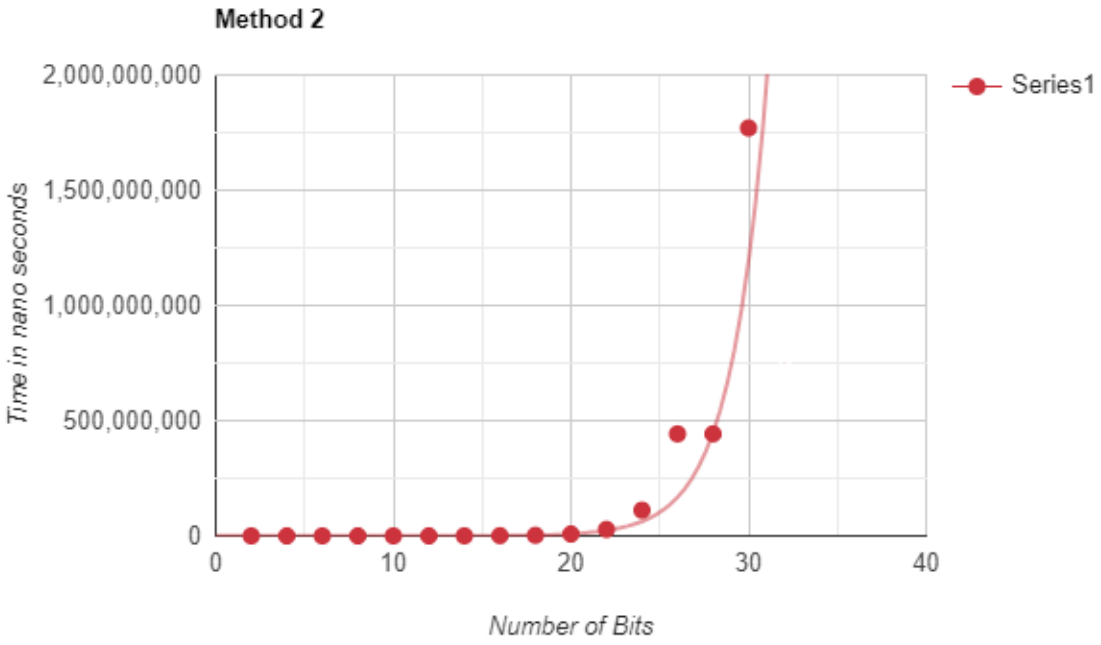
```
Enter the three integers A,B,M to find A^B % M
3 128 10000
OVERFLOW
Method 1 result: -6943
Elapsed time: 4199 ns


Method 2 result: 6961
Elapsed time: 7200 ns


Method 3 result: 6961
Elapsed time: 1299 ns


Method 4 result: 6961
Elapsed time: 1999 ns
```

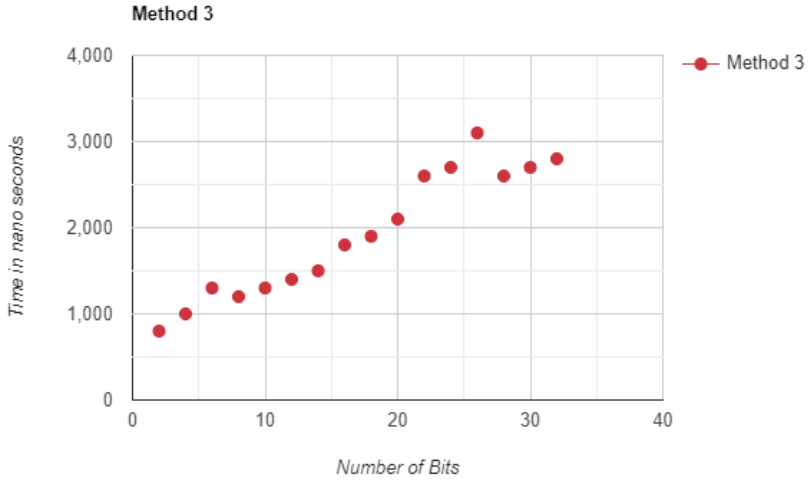
```
Enter the three integers A,B,M to find A^B % M
33333333 33333333 10000000
OVERFLOW
Method 1 result: -4753403
Elapsed time: 49813200 ns


Method 2 result: 4865413
Elapsed time: 698233899 ns


Method 3 result: 4865413
Elapsed time: 8801 ns


Method 4 result: 4865413
Elapsed time: 24400 ns
```

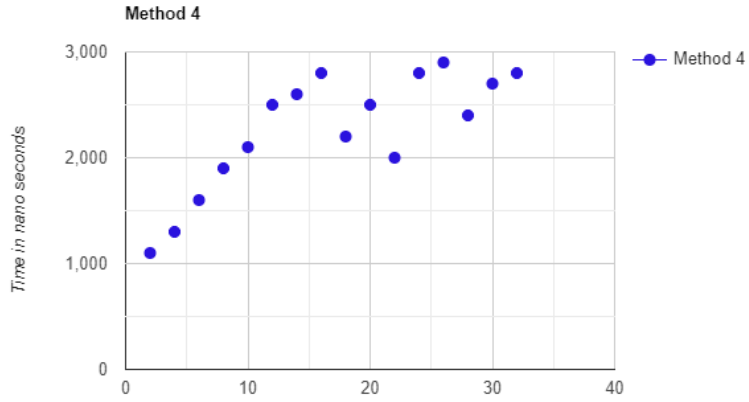
Comparisons and Conclusion: -

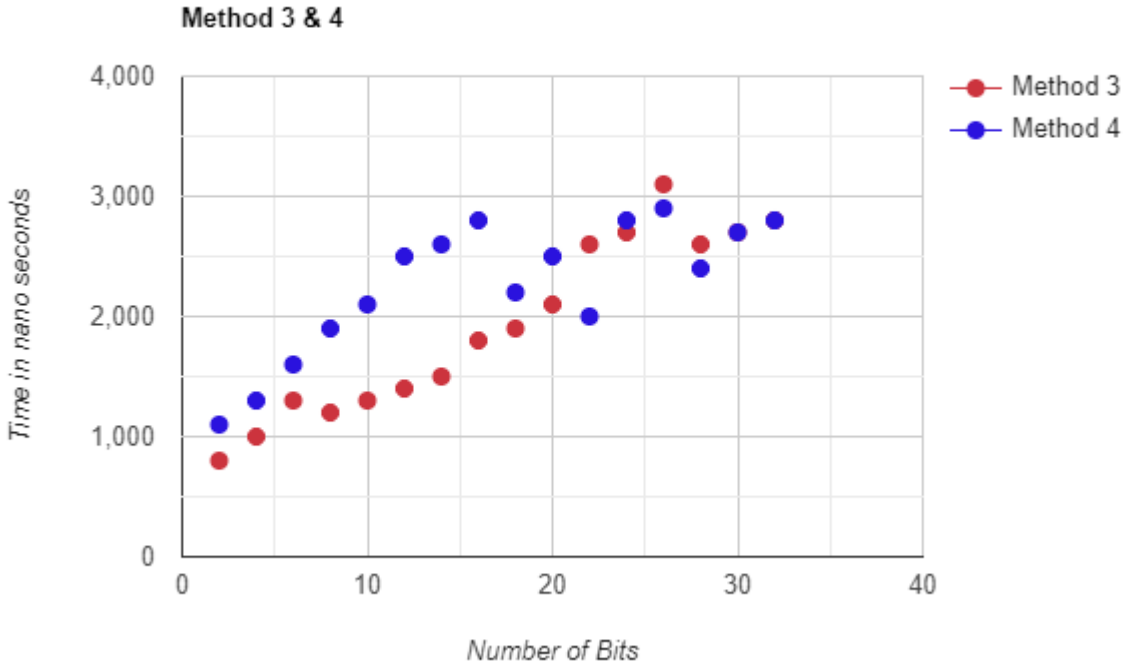
Graphed below is number of bits used to represent all numbers and calculations in a method vs the execution time in nano seconds.






Method 3
4,000
3,000
2,000
1,000
0
Time in nano seconds
0
10
20
30
40
Number of Bits
Method 3


Method 4
3,000
2,000
1,000
0
Time in nano seconds
0
10
20
30
40
Method 4


Method 3 & 4
4,000
3,000
2,000
1,000
0
Time in nano seconds
0
10
20
30
40
Number of Bits
Method 3
Method 4

Side note:

Using first two methods with numbers represented in more that 32-bits becomes very costly as the number of iterations become huge. More over the first method overflow when using more that 8 bit numbers.

The binary methods are very efficient though with execution time not increasing a lot by increasing the number of bits.

Conclusion:

Overflow:

First of all, let's discuss overflow, the first naïve method overflow whenever a^b exceeds the limit of the number that can be represented in the chosen number of bits, so it overflow quite often and is a very unreliable method.

Second method you can't know for sure when it overflows but you can be certain it didn't overflow if a*(m-1) doesn't exceed the limit of the bits used, if the aforementioned product does exceed the limit overflow may happen but we can't be sure the program will try to calculate that product as a*a%m can range from a*0 to a*(m-1), this method is very reliable for small values of m.

Third and fourth methods are very fast and very reliable but they aren't flawless as they overflow when the input number are bigger that the limits of the bits used.

Execution time:

First two methods are naïve and shouldn't be used, second method is faster than the first and a little more reliable though, third and fourth method are very reliable but fourth method (recursive) is usually slower than the iterative version maybe that's due to the overhead of the recursive call.

The relationship between the execution time of the first method and the number of bits used is somewhat linear for the first 8 bits, using more than 8 makes the operation overflow even when storing the results in 64 bits and the run time increases exponentially after that, the second method's time and bits curve is clearly exponentially but as mentioned before they tend to be too costly for large numbers.

Since third and fourth are quite close only that the fourth method usually takes a bit more time. I will discuss the third method and that can be generalized to include the fourth.

It is very fast and reliable as mentioned before but there is a very important concept that need to be discussed.

The execution time of this method doesn't really co-relate with the number of bits used to represent the number (i.e., its value) but it co-relates with the number of the bits set to 1 in the exponent.

So, increasing the exponent doesn't necessarily increase the execution time by a big margin but operating on a value with a bit number of bits set to 1 does increase the execution time, so in my calculations I've opted for using numbers that have more bits set to 1 than not.