CS201: Fundamental Structures of Computer Science

Homework Assignment - 2

Time Complexity Analysis of Algorithms

Zeynep Begüm Kara
ID: 22003880
CS201 - 01

April 3, 2022

1)

I. Complexity of Naive Algorithm

```
/*
 * Algorithm I: Naive Algorithm
 * multiplies 'a' with itself in a for loop and takes modulus 'p' in each step
 *
 * @param long number a, long power n, long p denotes the congruence relation
 * @return a^n (mod p)
 */
unsigned long long int naiveAlgorithm(unsigned long long int a, unsigned long long int n, int p)
{
   if ( n == 0 ) { return 1; }
   else
   {
       unsigned long long int res = 1;
       for(unsigned long long i = 0; i < n; i++)
       {
           res *= a;
           res = res % p;
       }
       return res;
   }
}
```

In the *if block* of this function, function just returns a value in a constant time O(1).

In the *else block* of this function; first, there exists a variable initialization with constant complexity with O(1). After that, a single *for loop* executes *n* times. It means that, each of the two statements, which has constant complexity with O(1), in the *for block* are called *n* times. Thus, we have $n \times O(1) = O(n)$.

Therefore, the best case of the function occurs when $n = 0$. Its upper bound is a some constant c, and the complexity of function is O(1).

The worst case occurs otherwise i. e. $n \neq 0$. Its upper bound is n, and the complexity of function is O(n).

II. Complexity of Naive Algorithm with Cycle Shortcut

```
/*
 * Algorithm II: Naive algorithm with Cycle shortcut
 *-------------------------------------------------------------
 *
 * @param long number a, long power n, long p denotes the congruence relation
 * @return a^n (mod p)
 */
unsigned long long int cycleShortcutAlgorithm(unsigned long long int a, unsigned long long int n, int p)
{
  if ( n == 0 ) { return 1; }
  else
  {
      unsigned long long int res = 1;
      for(unsigned long long int i = 0; i < n; i++)
      {
          res *= a;
          res = res % p;
          if ( res % p == 1 ) {
              return naiveAlgorithm(a, (n % (i + 1)), p);
          }
      }
      return res;
  }
}
```

In the *if block* of this function, function just returns a value in a constant time O(1).

In the *else block* of this function; first, there exists a variable initialization with constant complexity with O(1). After that, a *for loop* executes until it satisfies the given condition. If the problem hasn't any cyclic situation in given values of a, n, and p, then the function performs in a same way as the naive algorithm and corollary has the same complexity O(n). However, if modular exponentiation is cyclic, it performs the first algorithm with *n mod(i)* which is considerably smaller than the beginning. Since the statement is returned in this situation, the first algorithm is performed only once, which has made the complexity of this state O(n).

Therefore, the best case of the function occurs when n = 0. Its upper bound is a some constant c, and the complexity of function is O(1).

The worst case occurs otherwise i. e. $n \neq 0$. Its upper bound is n, and the complexity of function is O(n). One should observe that since in the cyclic situation, by definition *n mod(i)* is smaller than *n* itself -especially in long n's- it is a very efficient shortcut in terms of time executed.

III. Complexity of Recursive Algorithm

```
/*
 * Algorithm III: Recursive algorithm
 * ----------------------------------
 *
 * @param long number a, long power n, long p denotes the congruence relation
 * @return a^n (mod p)
 */
unsigned long long int recursiveAlgorithm(unsigned long long int a, unsigned long long int n, unsigned long long int p)
{
    if (n == 0) { return 1; }
    else if (n == 1) { return a % p; }
    else if (n % 2 == 0)
    {
      return (recursiveAlgorithm(a, n / 2, p) * recursiveAlgorithm(a, n / 2, p)) % p;
    }
    else
    {
      return (a * recursiveAlgorithm(a, (n-1) / 2, p) * recursiveAlgorithm(a, (n-1) / 2, p)) % p;
    }
}
```

In the *if block* of this function, function just returns a value in a constant time O(1).

The first *else if block* is the base case of the algorithm and has a complexity O(1).

The second *else if block* executes itself twice with n / 2. It is clear that the number of execution of this recursive algorithm is depended on the n. Also the *else* block approximately halves n and calls it twice. Since the number *n* is divided in half, then the half divided into half and so on until it hits the base case. Thus, the algorithm performs logN calculations for each half to reach the base case and it is performed twice for each call because the function is called twice and multiplied by itself in related blocks. Therefore, $2^{\log n} = n$ and corollary has a complexity O(n).

The best case of the function occurs when n = 0 or when it is the base case which is n = 1. Its upper bound is a some constant c, and the complexity of function is O(1).

The worst case occurs otherwise meaning that when function starts to call itself recursively. The upper bound is n, and the complexity of function is O(n).

2) Device Specifications

Processor	AMD Ryzen 7 3700U with Radeon Vega Mobile, Gfx 2.30 GHz
Installed RAM	8.00 GB
System Type	64-bit operating system, x64-based processor

2) This table demonstrates the elapsed time for three different algorithms for the same problem and examines how quickly the algorithm's time requirement grows as a function of problem size. Each algorithm calculates the following expression

$$\alpha^n \ (mod\ p)$$

Even though the number of operations that each of the three algorithms required is independent of the value of α, due to calculation differences, and for the sake of simplicity and consistency it is chosen the same number for all executions arbitrarily, which is α = 3.

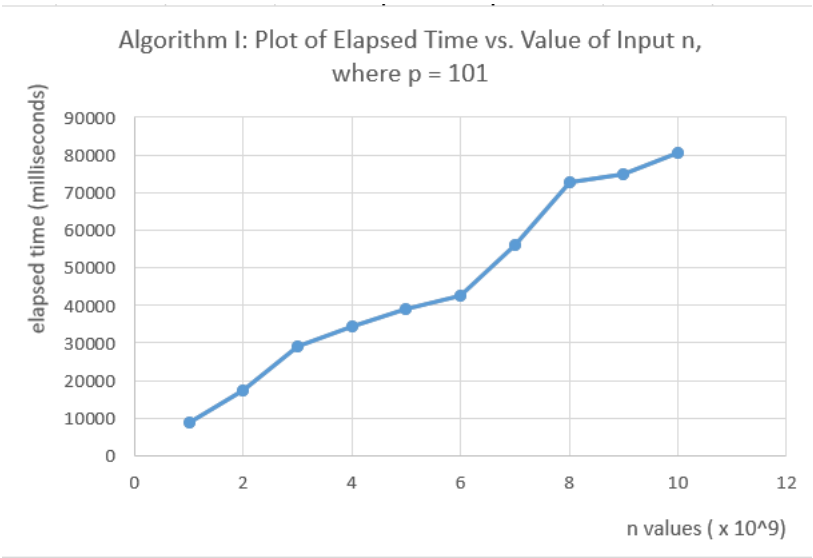
Elapsed Time (in milliseconds) as a Function of the Problem Size n and p

n	Algorithm I			Algorithm II			Algorithm III		
	p = 101	p = 1009	p = 10007	p = 101	p = 1009	p = 10007	p = 101	p = 1009	p = 10007
$1 * 10^9$	8744	8792	8796	0.01078	0.02214	0.69326	4720	4911	4648
$2 * 10^9$	17420	17764	17713	0.01008	0.02147	0.68904	8739	9698	9507
$3 * 10^9$	29177	24256	27931	0.01271	0.02261	0.68688	18422	19113	18222
$4 * 10^9$	34509	32930	32105	0.00993	0.02382	0.67888	18662	19080	18699
$5 * 10^9$	39008	41423	38995	0.01014	0.02497	0.65834	37811	37978	35681
$6 * 10^9$	42415	49587	46768	0.00979	0.02203	0.67048	36912	37960	34830
$7 * 10^9$	55906	57053	55096	0.01020	0.02387	0.68830	39119	37086	34817
$8 * 10^9$	72867	65306	61804	0.01100	0.02514	0.65949	39496	37442	40190
$9 * 10^9$	75035	77574	68728	0.01077	0.02801	0.69306	74238	72825	71985
$1 * 10^{10}$	80427	86054	85744	0.01102	0.02916	0.69936	75113	72930	72652

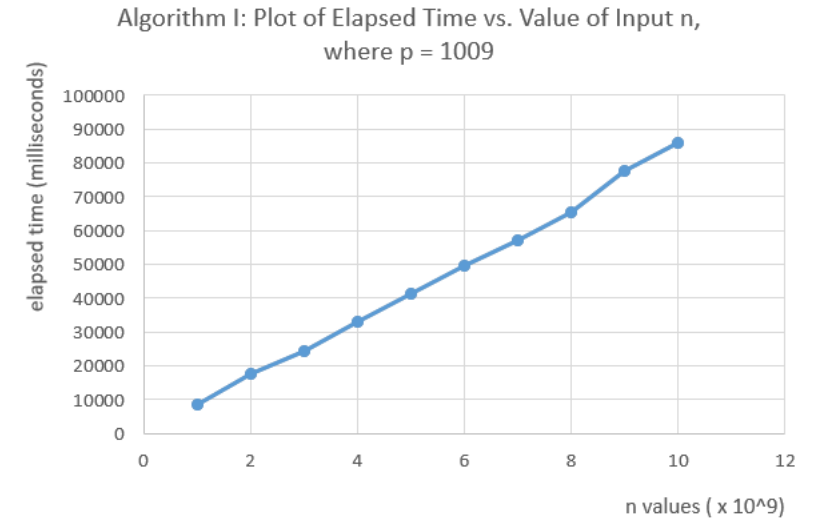
4) Plots of Algorithms

I. Algorithm I: Naive Algorithm

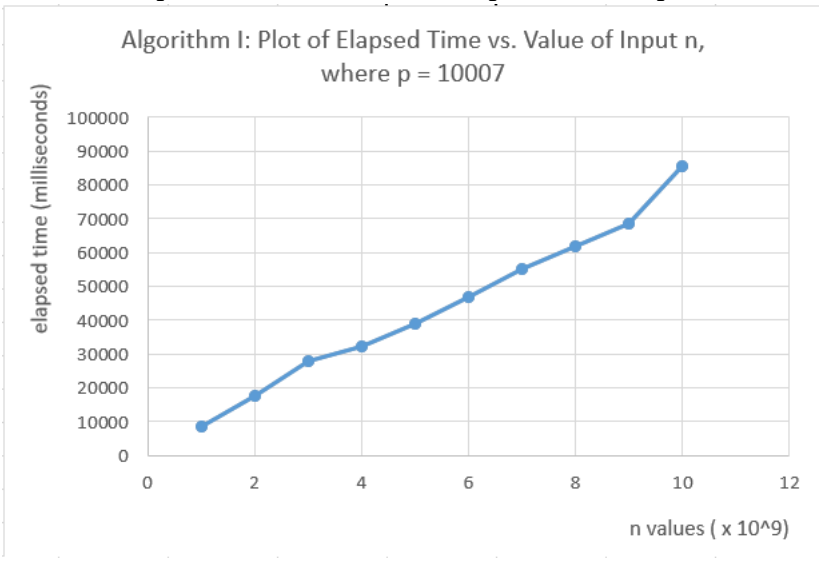
a. Plot I-a



b. Plot I-b

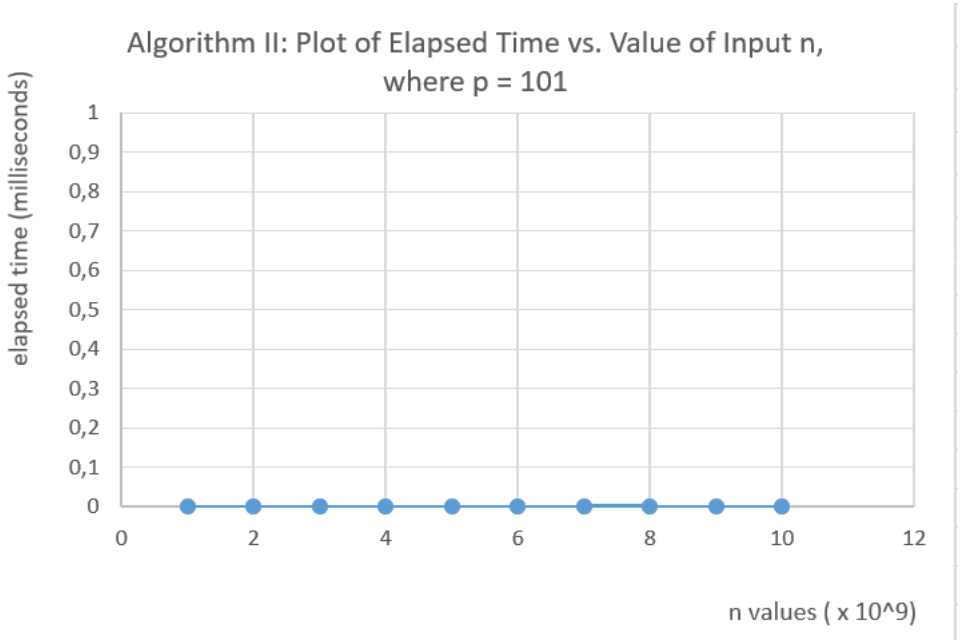


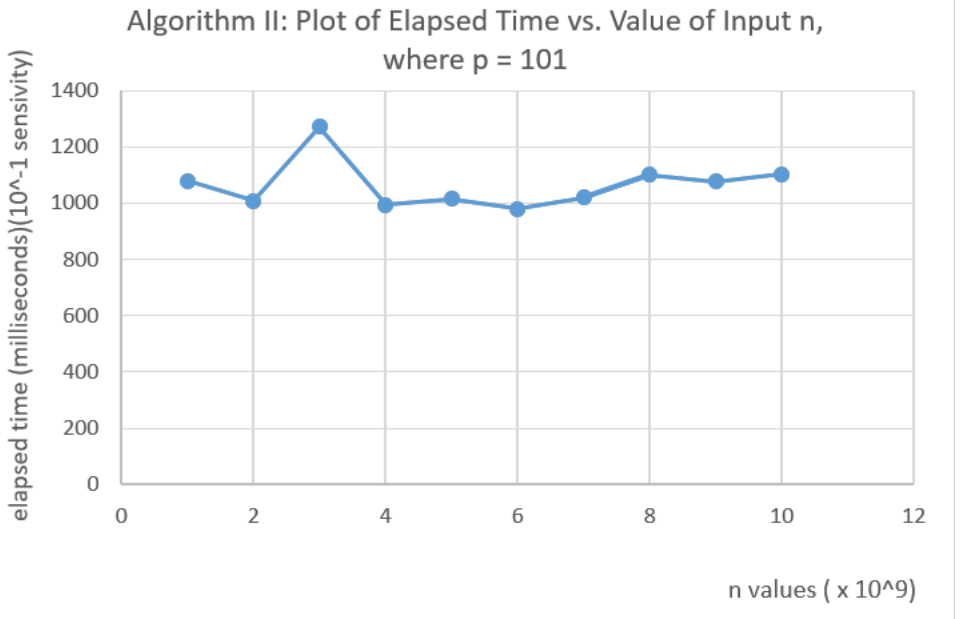
c. Plot I-c: Elapsed Time vs. Value of Input *n*, where p = 10007



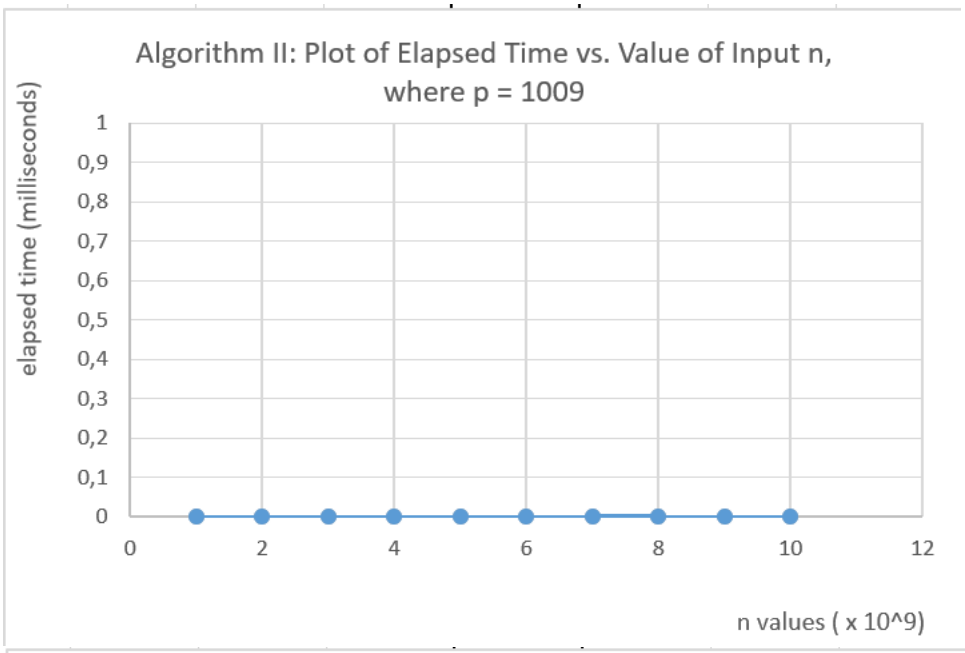
II. Algorithm II: Naive Algorithm with Cycle Shortcut

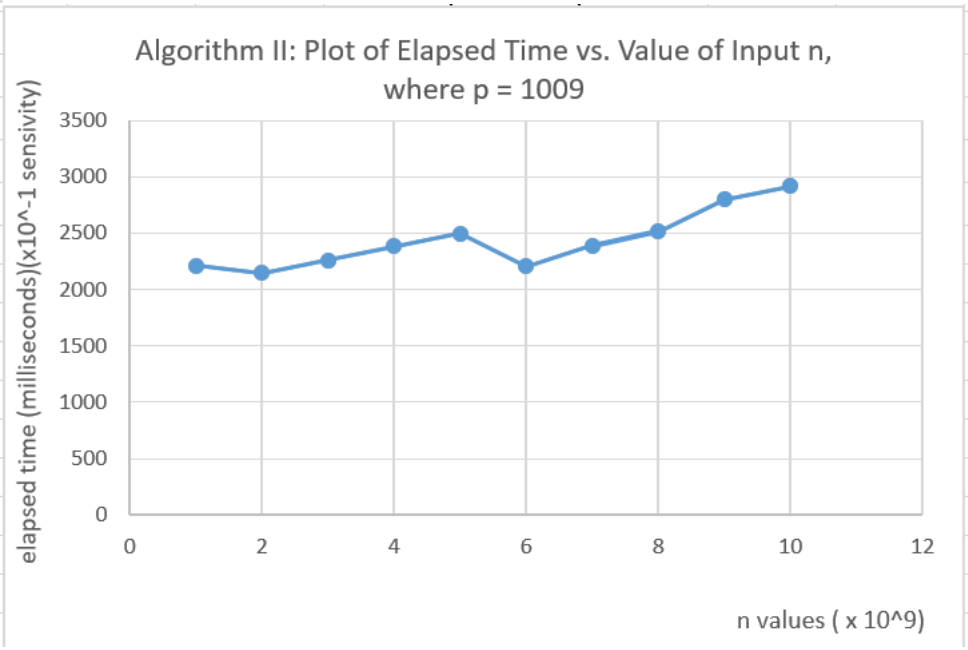
a. Plot II-a



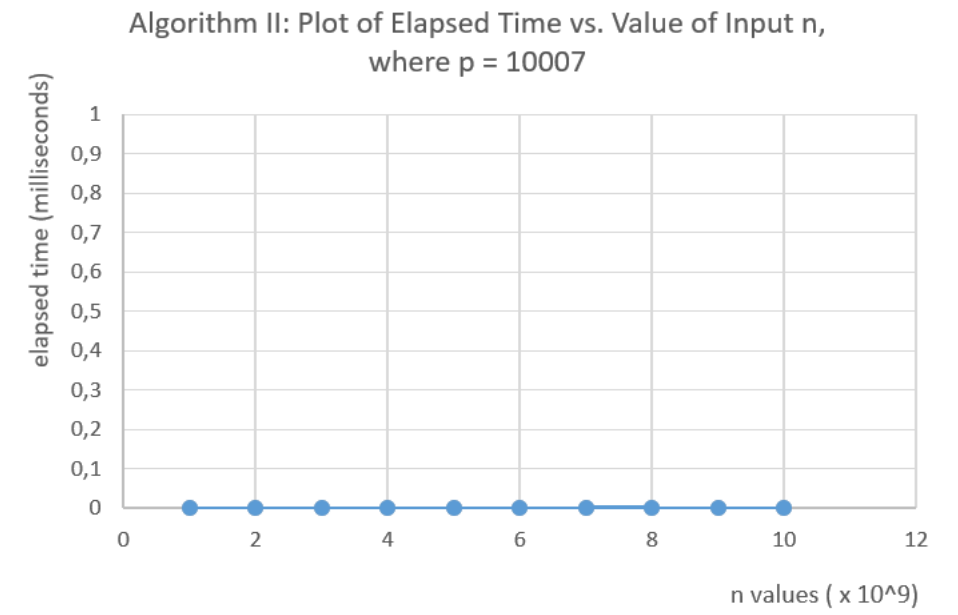


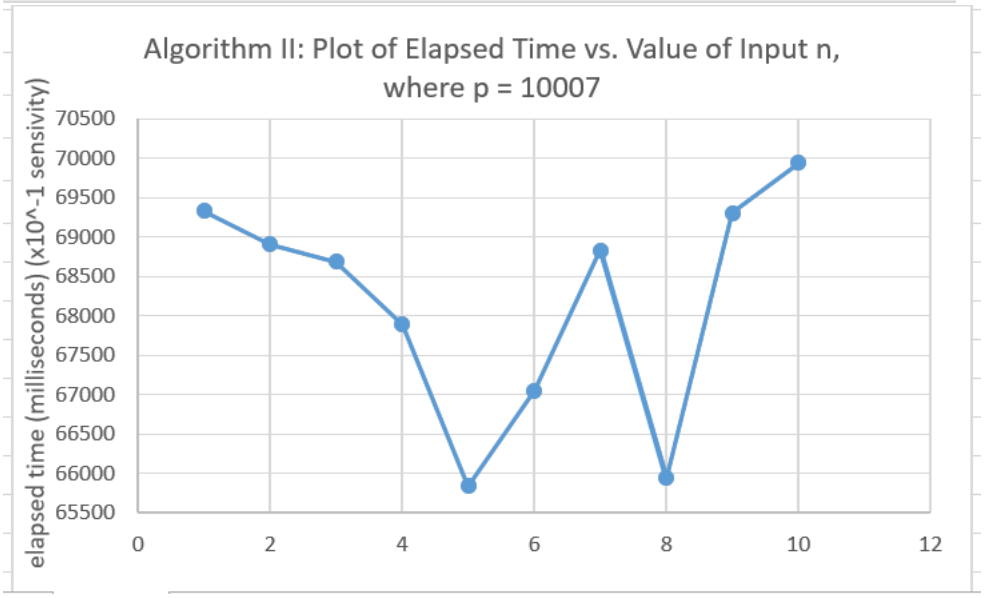
b. Plot II-b


Algorithm II: Plot of Elapsed Time vs. Value of Input n, where p = 1009
elapsed time (milliseconds)
1
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
0
2
4
6
8
10
12
n values (x 10^9)


Algorithm II: Plot of Elapsed Time vs. Value of Input n, where p = 1009
elapsed time (milliseconds)(x10^-1 sensivity)
3500
3000
2500
2000
1500
1000
500
0
0
2
4
6
8
10
12
n values (x 10^9)

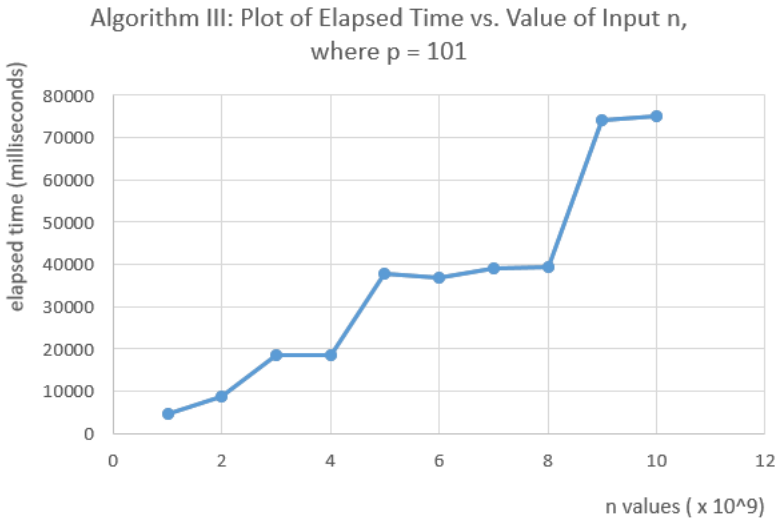
c. Plot II-c


Algorithm II: Plot of Elapsed Time vs. Value of Input n, where p = 10007
elapsed time (milliseconds)
1
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
0
2
4
6
8
10
12
n values (x 10^9)

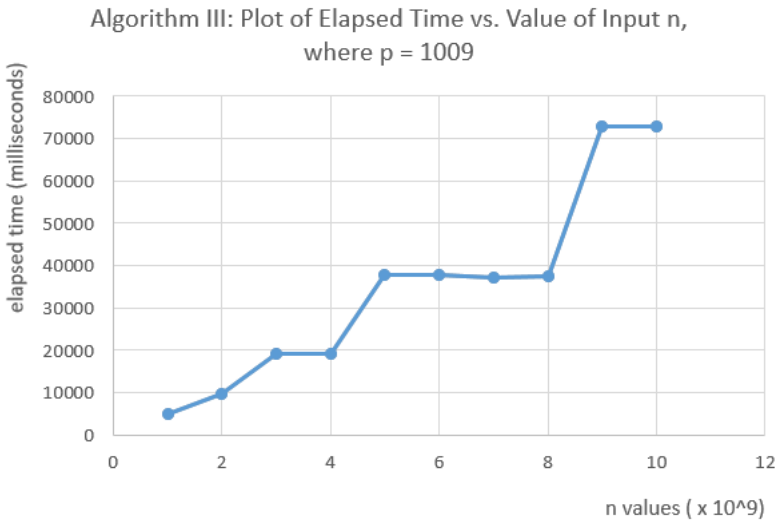

Algorithm II: Plot of Elapsed Time vs. Value of Input n, where p = 10007
elapsed time (milliseconds) (x10^-1 sensivity)
70500
70000
69500
69000
68500
68000
67500
67000
66500
66000
65500
0
2
4
6
8
10
12

III. Algorithm III: Recursive Algorithm

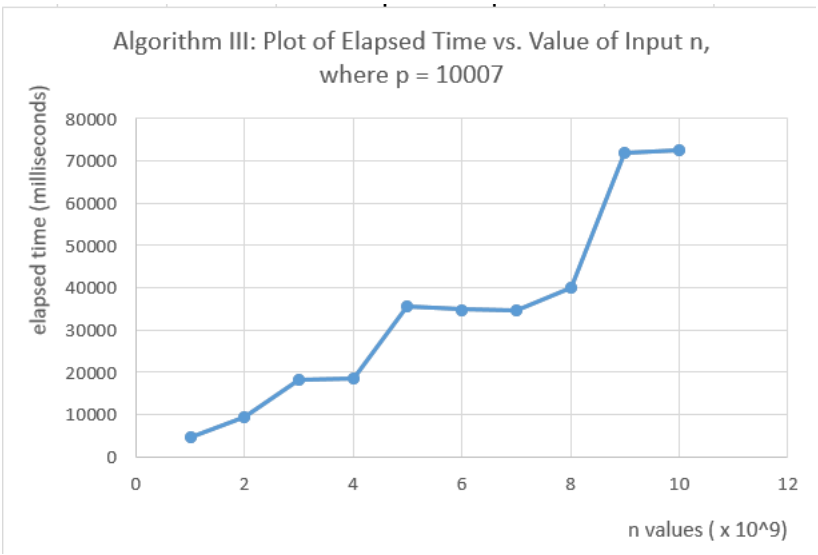
a. Plot III-a



b. Plot III-b



c. Plot III-c



Overall, for this chosen inputs, algorithm II has the best results due to cyclic shortcuts it has at the very beginning of the modular exponential. The fluctuations of these algorithm is due to cyclic mod it has at very different values. The rest of the algorithms behave similar to a linear function as expected.