CS201 - HOMEWORK 2

REPORT

HOMEWORK 2 REPORT

Course Code:	CS 201
Section:	2
Name:	Arda
Surname:	İynem
Student ID:	22002717
Date:	06.04.2022

Content

Naive Algorithm

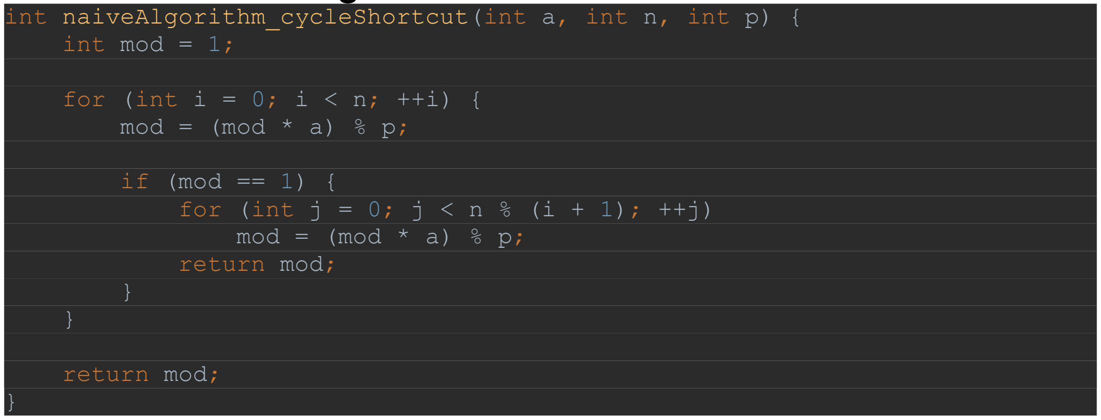
```
int naiveAlgorithm(int a, int n, int p) {
    int mod = 1;

    for (int i = 0; i < n; ++i)
        mod = (mod * a) % p;

    return mod;
}
```

Both the first assignment statement and the return statement at the end of the function will run just for one time no matter what the inputs are. Therefore, their time complexity equals to $\boldsymbol{T}(\mathbf{1}) + \boldsymbol{T}(\mathbf{1}) = \boldsymbol{O}(\mathbf{1})$ as consecutive statements' time complexities are summed**.** Whereas the for loop runs n times and it depends on the input n. So, its time complexity is $\boldsymbol{T}(\boldsymbol{n})$. Since the consecutive statements' time complexities should be summed to find total time complexity of the function.

$$\boldsymbol{T}(\boldsymbol{n}) + \boldsymbol{O}(\mathbf{1}) = \boldsymbol{O}(\boldsymbol{n} + \mathbf{1}) = \boldsymbol{O}(\boldsymbol{n})$$

Ultimately, this function's time complexity upper bound is $\boldsymbol{O}(\boldsymbol{n})$ in any case. So, it is expected to have a linear time complexity plot.

Naive Cycle with Shortcut Algorithm



```
int naiveAlgorithm_cycleShortcut(int a, int n, int p) {
    int mod = 1;

    for (int i = 0; i < n; ++i) {
        mod = (mod * a) % p;

        if (mod == 1) {
            for (int j = 0; j < n % (i + 1); ++j)
                mod = (mod * a) % p;
            return mod;
        }
    }

    return mod;
}
```

Fermat's Little theorem shows $\boldsymbol{a^{p-1}} \equiv \mathbf{1}\ (\boldsymbol{mod\ p})$ if p is prime. Considering all the **p** input values (101, 1009, 10007) used in the function are prime numbers, the outer for loop will execute for **i** times where **i** is equal to **p - 1**. Therefore, there are two cases for analyzing time complexity of this function

Case 1 (n < p - 1): If **n** is less than **p – 1** (i). Then **mod == 1** will never be true, hence for loop inside the if block will never run. So, the function will execute just like Naive Algorithm function which has $O(n)$ time complexity upper bound. Ultimately, in this case the function's time complexity upper bound is $\boldsymbol{O(n)}$. So, it is expected to have a linear time complexity plot.

Case 2 (n ≥ p - 1): If **n** is equal or greater than **p – 1** (i). Outer loop will execute **p – 1** time where in each step statements with $T(1)$ complexity will be executed. In the step p – 1, **mod == 1** will be true. Hence the for loop inside the if block will be executed for once then function will return. Statements with $T(1)$ time complexity will be executed for n (mod p - 1) time in the inner for loop, clearly n (mod p - 1) is less than p.

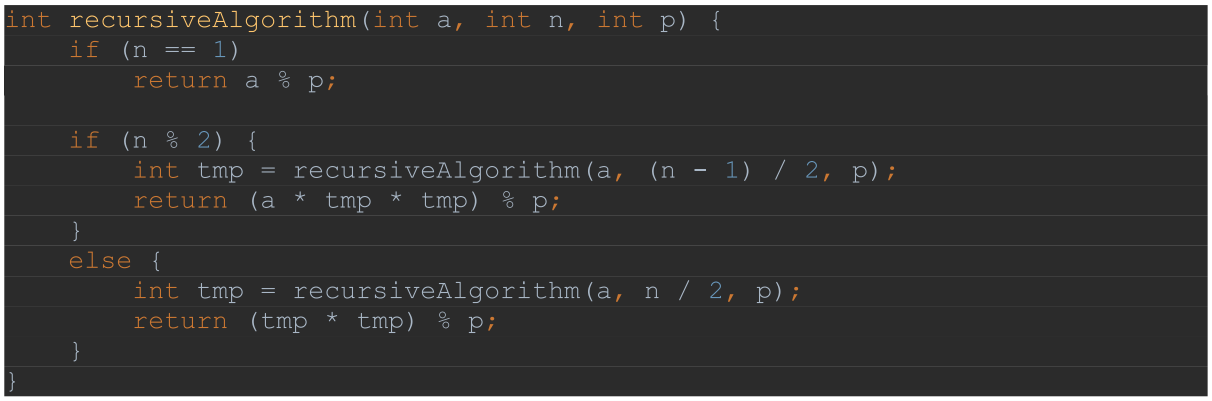
$T(p-1) * T(1) = T(p) = O(p)$ $\Longrightarrow$ Time complexity upper bound of outer for loop.

As n (mod p - 1) is less than p, its upper bound can be represented with $O(p)$ even if it is not the least upper bound.

$T(n\ (mod\ p-1)) * T(1) = O(p)$ $\Longrightarrow$ Time complexity of inner for loop.

Since inner loop will be executed only in last step of outer loop, total time complexity is $O(p) + O(p) = O(2p) = O(p)$ Ultimately, in this case the function's time complexity upper bound is $\boldsymbol{O(p)}$. So, it is expected to have a linear time complexity plot.

Recursive Algorithm



```
int recursiveAlgorithm(int a, int n, int p) {
    if (n == 1)
        return a % p;

    if (n % 2) {
        int tmp = recursiveAlgorithm(a, (n - 1) / 2, p);
        return (a * tmp * tmp) % p;
    }
    else {
        int tmp = recursiveAlgorithm(a, n / 2, p);
        return (tmp * tmp) % p;
    }
}
```

The function will go through a recursive call chain in which it calls itself each time either dividing input n by 2 or decrementing by 1 then dividing by 2 until it reaches the base case n equals 1. Since $\boldsymbol{T\left(\frac{n-1}{2}\right) = T\left(\frac{n}{2}\right)}$ both statements of if-else block can be represented by $\boldsymbol{T\left(\frac{n}{2}\right)}$ also $\boldsymbol{T(1) = O(1)}$ in the analysis.

$$\left.\begin{array}{r}
T(n) = T\left(\frac{n}{2}\right) + O(1) \\
T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + O(1) \\
T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + O(1) \\
\\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots
\end{array}\right\} \text{Executes for } \log_2 n \text{ times}$$

$$T(n) = \log_2 n * O(1) = O(\log_2 n) = O(\log n)$$

Ultimately, this function's time complexity upper bound is $\boldsymbol{O(\log n)}$. So, it is expected to have a logarithmic time complexity plot.

COMPUTER
SPECIFICATIONS

Hardware

Processor: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz (12 CPUs)

Memory: 16384MB RAM (16GB (2x8GB) DDR4 1.2V 2933MHz SODIMM)

Graphics Card Name: nVIDIA® GeForce® RTX2060 6GB GDDR6 192-Bit DX12 Refresh

OS

Operating System: Windows 11 Pro 64-bit (10.0, Build 22000, Version 21H2)

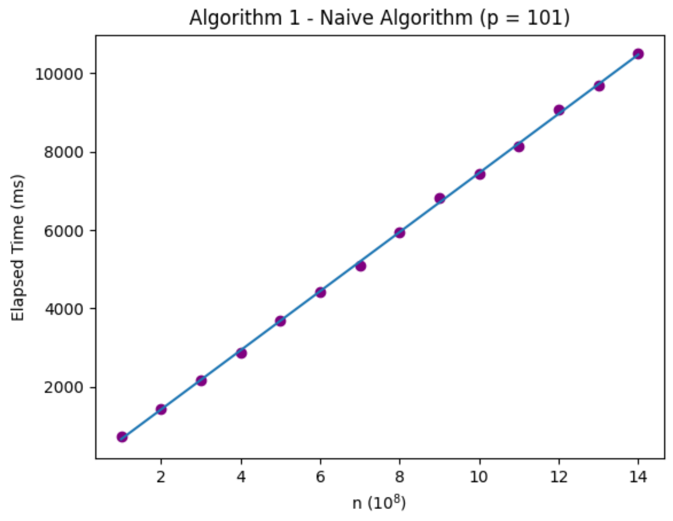
TABLE

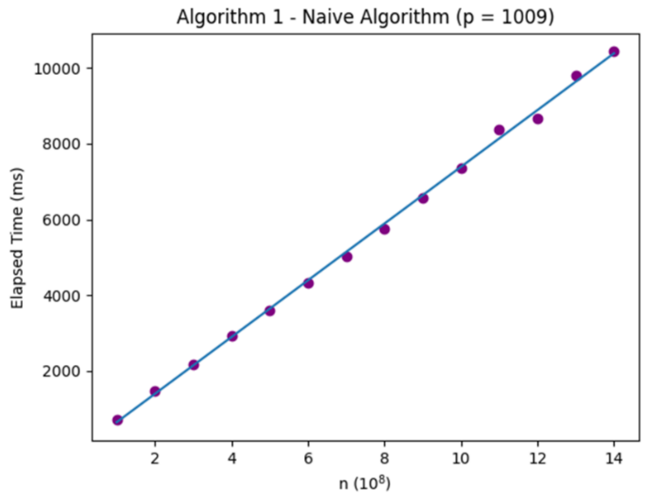
EXECUTION TIME OF EACH ALGORITHM WITH VARYING INPUTS

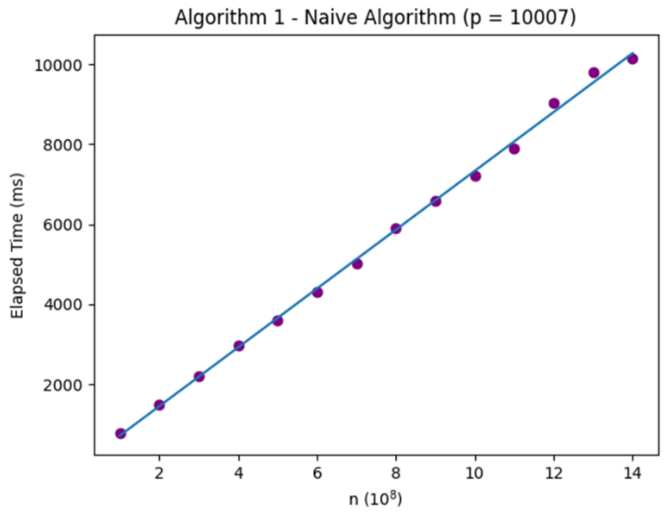
Execution Time (Milliseconds) of The Algorithms with Varying *n* and *p* Inputs

n	Algorithm 1			Algorithm 2			Algorithm 3		
	p=101	p=1009	p=10007	p=101	p=1009	p=10007	p=101	p=1009	p=10007
10^8	728.902	717.368	784.466	0.0009852	0.00186	0.0859361	0.00025006	0.000248137	0.000241681
$2*10^8$	1440.74	1470.39	1481.73	0.0009202	0.0016336	0.081918	0.000261037	0.00025161	0.000250519
$3*10^8$	2155.76	2165.27	2199.23	0.0008283	0.0014219	0.0824923	0.00026924	0.000259291	0.000258533
$4*10^8$	2881.73	2941.97	2957.35	0.0008288	0.001228	0.0854769	0.000267365	0.000259801	0.000259474
$5*10^8$	3683.49	3609.14	3597.81	0.0008495	0.0020249	0.0882921	0.000270502	0.000264401	0.000260925
$6*10^8$	4433.48	4342.38	4298.26	0.000941	0.001779	0.0854034	0.000275253	0.000271052	0.000265829
$7*10^8$	5085.37	5028.19	5023.09	0.0008287	0.0018539	0.0869055	0.000275819	0.000267521	0.000267829
$8*10^8$	5929.39	5746.27	5909.04	0.0010612	0.0016657	0.0849035	0.000280921	0.000269998	0.000267404
$9*10^8$	6815.52	6562.59	6575.09	0.0009798	0.0014456	0.0879175	0.000278335	0.000267843	0.000268892
$10*10^8$	7431.85	7368.48	7202.57	0.0008281	0.0025559	0.0850403	0.000276804	0.000272487	0.00027046
$11*10^8$	8138.72	8392.22	7900.91	0.0008298	0.001944	0.0869771	0.000282897	0.000275251	0.000273411
$12*10^8$	9065.71	8673.8	9044.59	0.0008283	0.0015719	0.084285	0.000285606	0.000276051	0.000276454
$13*10^8$	9705.6	9807.39	9795.98	0.0010809	0.0013608	0.0852037	0.000293497	0.000282436	0.000279366
$14*10^8$	10500	10434.2	10137.6	0.0011216	0.0011511	0.086608	0.000284145	0.000277937	0.000281481

PLOTS NAIVE ALGORITHM

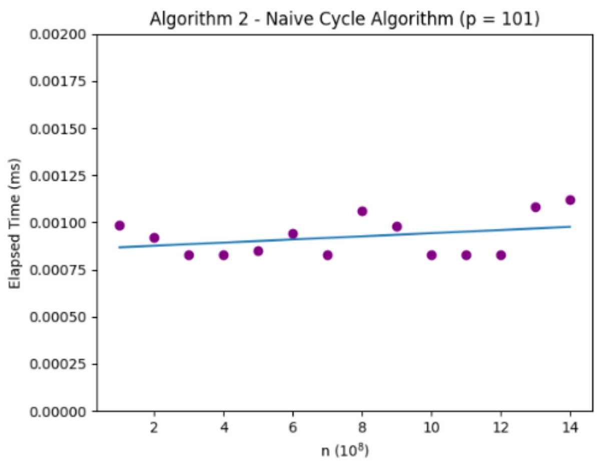


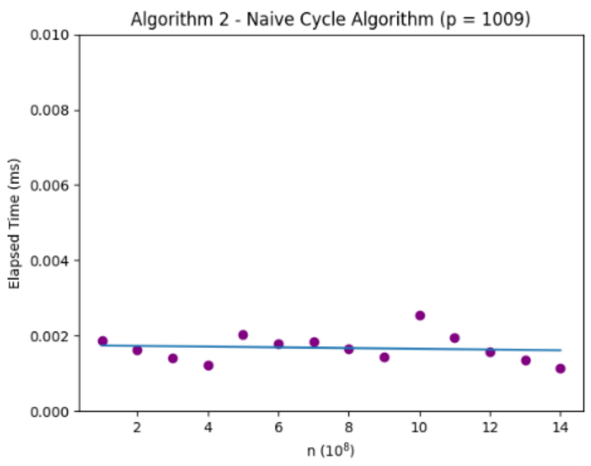


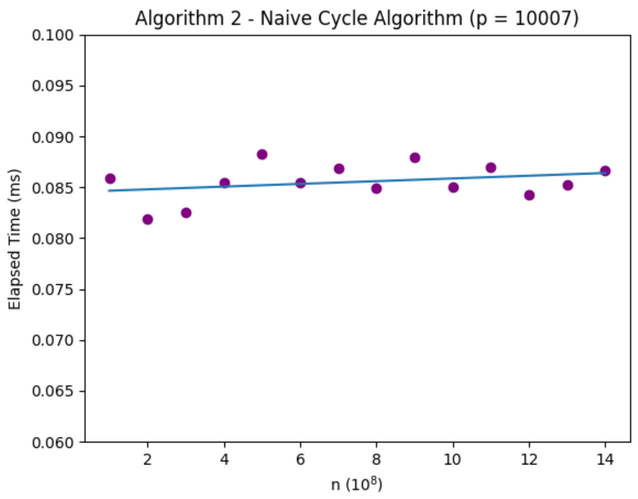


As Naive Algorithm function's time complexity upper bound was suggested $\boldsymbol{O(n)}$ earlier, algorithm's all 3 plots seem to be linear graphs as expected. Clearly, the points on the graph show that execution time of the algorithm function is linearly increased as the input n is increased. Moreover, execution time of the function isn't affected by the other input p as previously found time complexity upper bound $\boldsymbol{O(n)}$ is not depending on the input p. Conclusively, the graphs proved the suggested time complexity upper bound of this algorithm to be true.

PLOTS NAIVE ALGORITHM WITH CYCLE SHORTCUT

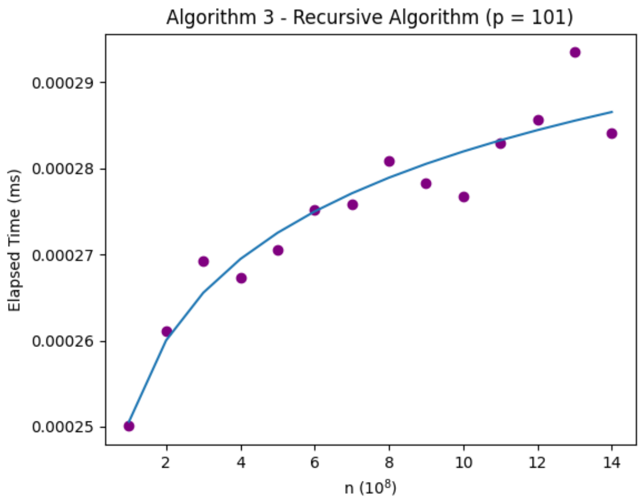


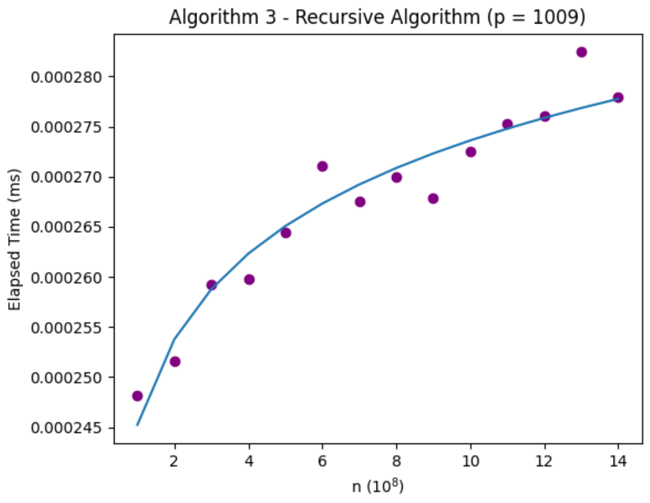


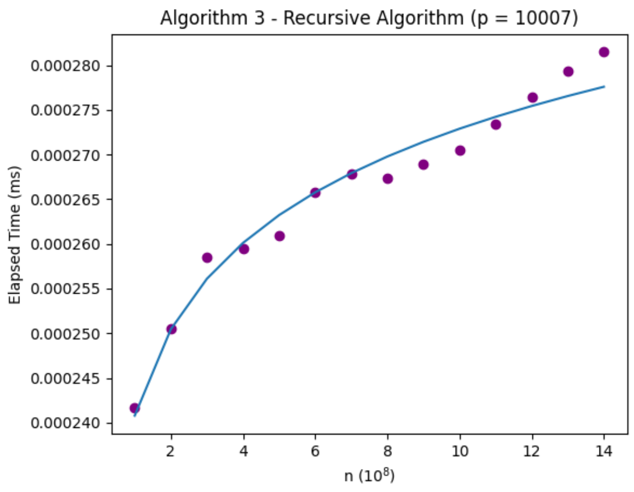


As Naive Cycle with Shortcut Algorithm function's time complexity upper bound was suggested $\boldsymbol{O}(\boldsymbol{n})$ or $\boldsymbol{O}(\boldsymbol{p})$ depending on whether n is less or greater than p - 1 when p is a prime number earlier, algorithm's plots seem to be linear as expected. However, the graphs drawn seem to resemble the graph of $\boldsymbol{O}(\mathbf{1})$ because n was increased instead of p while n is greater than p - 1. As previously mentioned, there were 2 cases and for all three plots, case 2 ($\boldsymbol{O}(\boldsymbol{p})$) applies since all n values used are greater than p - 1. Therefore, as p has been kept constant while n is increasing, plots resemble the graph of $\boldsymbol{O}(\mathbf{1})$ as it was given a constant value of p. However, when the graphs are compared with each other, it can be seen that execution time is increased as value of p is increased, for the same n values of each graph. Which shows that this algorithm's function execution time has a linear relation with input p, meaning its time complexity upper bound is $\boldsymbol{O}(\boldsymbol{p})$. Conclusively, the graphs proved the suggested time complexity upper bound of this algorithm to be true.

PLOTS RECURSIVE ALGORITHM







As Recursive Algorithm function's time complexity upper bound was suggested $\boldsymbol{O}(\log n)$ earlier, algorithm's plot seems to have a logarithmic increase as expected. Clearly, the points on the graph show that execution time of the algorithm function is increased logarithmically as the input n increased. Moreover, it can be seen that execution time of the function isn't affected by the other input p as previously found time complexity upper bound $\boldsymbol{O}(\log n)$ is not depending the input p. Conclusively, the graphs proved the suggested time complexity upper bound of this algorithm to be true.