Berfin Kavşut
21602459
CS201 – Section 1
April 22, 2020

# REPORT FOR HOMEWORK ASSIGNMENT 2

## 1. Introduction

When implementing our algorithms, we want to theoretically know about the usage of our sources, which are mainly time and memory. In this homework, we will focus on time complexity of sorting algorithms. Firstly, we will study rules of time complexity calculations and Big-Oh notation. Then, we will analyse our three algorithms given to use for assignment. We will compare our theoretical results with experimental results. We will observe algorithms with growing arrays to see the behaviour of algorithms, record operation time for each array size and then plot our N vs. time graphs.

## 2. Analysis of Time Complexities of Algorithms

*Big-Oh Notation*: Big-Oh is used to define an upper bound for algorithms. Big-oh function for functions should be such that after a certain point, the function will be always smaller than its big-oh function.

*Definition:* $T(N) = O(f(N))$ if there are positive constants c and $n_0$ such that: $T(N) \leq c(f(N))$ when $\geq n_0$.

Rules for Time Complexity:

1. For loops, time complexity is calculated by multiplying time complexity of statement inside loop with by number of iterations.
2. For nested loops, number of iterations are multiplied and then time complexity is calculated by multiplying this iteration number with time complexity of statement inside loops. This is actually the same rule as the first rule, just a special case.
3. For if/else statements, time complexity is maximum of time complexities of statements. This can be seen as time complexity for worst case. Also, the conditioning statement is added to analysis.
4. For consecutive functions, their time complexities are summed up.
5. Simple instructions are assumed to take one unit time. By simple instructions, we mean arithmetic operations, comparison, assignment and so forth.

Actually, the time required for the programme is more than our theoretical analysis in this homework. The reason is that we do not take the time required for creating the array into account. Also, the programme theoretically should be able to go for all array sizes infinitely assuming we have infinite memory, but it cannot in reality. The reason is memory complexity of algorithms. The algorithms having recursive functions cannot go to larger N values and the programme crashes. Nevertheless, we will not focus on memory complexity for our algorithms and the data we obtained is enough to see relative growth rates in terms of time.

### Selection Algorithms

Table 1. Time Complexities of Algorithms (k=1000, constant)

| | Best Case | Average Case | Worst Case |
|---|---|---|---|
| Algorithm 1 | $O(N)$ | $O(N)$ | $O(N)$ |
| Algorithm 2 | $O(NlogN)$ | $O(NlogN)$ | $O(N^2)$ |
| Algorithm 3 | $O(Nlog^2N)$ | $O(Nlog^2N)$ | $O(N^2logN)$ |

*Algorithm 1:* There is one nested for-loop in solution 1. The loop inside is $O(N)$ as the number of iterations is N. The loop outside is $O(k)$ since the number of iterations is k. This is a nested loop, so by rule 2, time complexity of algorithm 1 is $O(Nk)$. k=1000, which is constant in our code. Therefore, time complexity becomes $O(N)$.

*Algorithm 2:* Quick sort cannot select largest k elements inside array of N directly. First, it sorts all array in descending order, then returns the largest k elements. After the array is sorted, it takes $O(k)$ times. Total time complexity for average case becomes $O(k)$ + $O(NlogN)$. k = 1000, and constant in our code. Therefore, time complexity becomes $O(NlogN)$ for average case.

Worst case:

For worst case, the array should be already sorted in descending or ascending order. Let us analyse the descending order array. The solution2 function enters into partition function, whose time complexity is $O(N)$. After partition one, the function enters to first quick sort function. With this new quicksort function, the function enters to partition again. The for-loop inside partition cannot be operated this time. The next quick sort is operated for the rest of array, which is in descending order again, and array size is n-1.

$\mathrm{T}(N) = T(0) + T(N-1) + O(N)$

$\mathrm{T}(N) = T(N-2) + O(N) + O(N)$

$\mathrm{T}(N) = T(N-k) + kO(N)$

$\mathrm{T}(N) = O(N^2)$

Best case:

For best case, the pivot should be chosen in middle of array each time.

$T(N) = T\left(\frac{N}{2}\right) + O(N)$

$T(N) = T\left(\frac{N}{2^k}\right) + kO(N)$

$T(N) = T(1) + \log(N)O(N)$

$T(N) = O(NlogN)$

*Algorithm 3:* The analysis is similar to analysis for algorithm 2. This time, we will have $O(NlogN)$ instead of $O(N)$ at the beginning, because quick sort is called inside select function recursively.

Worst case:

$T(N) = T(N-k) + kO(NlogN)$

$T(N) = O(N^2logN)$

Best case:

$$T(N) = T\left(\frac{N}{2^k}\right) + kO(NlogN)$$

$$T(N) = O(Nlog^2N)$$

## 3. Results

Results are obtained by the computer whose specifications are given here.

Computer Specifications

Computer: ASUS, UX410UQK

OS: Microsoft Windows 10 Home Single Language

CPU: Intel(R) Core(TM) i7-7500U CPU @2.70 GHz, 2.90 MHz

Memory: 8.00 GB RAM

Figure 1 is plotted with the data in Table 2. There are two tables, because running three algorithms at once was giving less data than desired. Hence, I ran the code for each algorithm. Still, the limitations can be seen in Table 3. Separate plots for algorithms are shown in Figure 2-4 with data taken from Table 3.

Table 2. Comparison of Operation Times of Algorithms

| N | Time for Algorithm 1 (ms) | Time for Algorithm 2 (ms) | Time for Algorithm 3 (ms) |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 |
| 16 | 0 | 0 | 0 |
| 32 | 0 | 0 | 0 |
| 64 | 0 | 0 | 0 |
| 128 | 0 | 0 | 0 |
| 256 | 0 | 0 | 0 |
| 512 | 1 | 0 | 1 |
| 1024 | 1 | 0 | 1 |
| 2048 | 4 | 1 | 0 |
| 4096 | 8 | 2 | 2 |
| 8192 | 18 | 7 | 3 |
| 16384 | 37 | 19 | 9 |
| 32768 | 75 | 50 | 12 |
| 65536 | 161 | 172 | 57 |
| 131072 | 344 | 602 | 136 |
| 262144 | 626 | 2405 | 776 |
| 524288 | 1280 | 9226 | 3380 |
| 1048576 | 2524 | 36728 | 13077 |

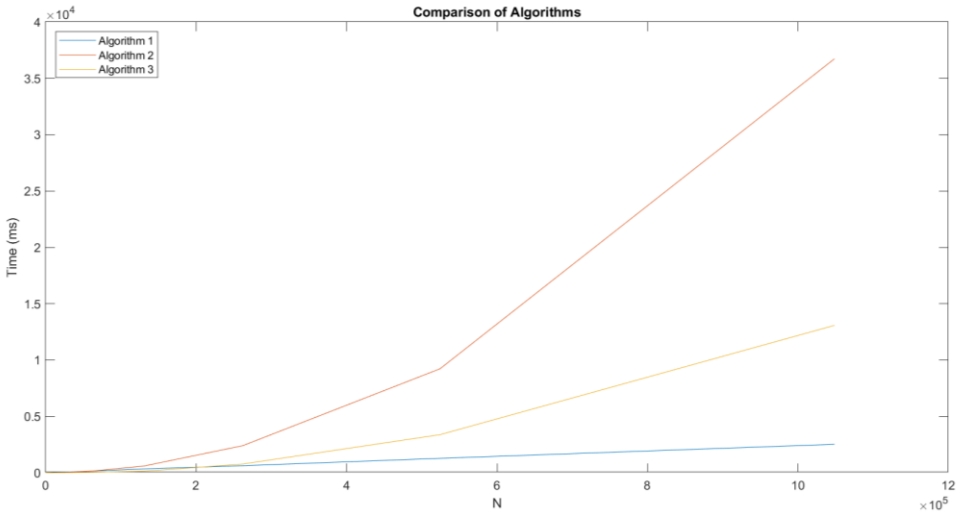


Figure 1. Comparison of Algorithms

Table 2. Comparison of Operation Times of Algorithms

| N | Time for Algorithm 1 (ms) | Time for Algorithm 2 (ms) | Time for Algorithm 3 (ms) |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 |
| 8 | 0 | 1 | 0 |
| 16 | 0 | 0 | 0 |
| 32 | 0 | 0 | 0 |
| 64 | 0 | 0 | 0 |
| 128 | 0 | 0 | 0 |
| 256 | 0 | 0 | 0 |
| 512 | 1 | 0 | 0 |
| 1024 | 2 | 1 | 0 |
| 2048 | 3 | 0 | 1 |
| 4096 | 12 | 2 | 3 |
| 8192 | 21 | 5 | 4 |
| 16384 | 44 | 16 | 10 |
| 32768 | 81 | 48 | 13 |
| 65536 | 173 | 166 | 55 |
| 131072 | 320 | 774 | 245 |
| 262144 | 662 | 2353 | 875 |
| 524288 | 1379 | 9214 | 3649 |
| 1048576 | 2716 | 36601 | 13536 |
| 2097152 | 5336 | 150575 | N/A |
| 4194304 | 9800 | N/A | N/A |
| 8388608 | 19559 | N/A | N/A |
| 16777216 | 39080 | N/A | N/A |
| 33554432 | 78247 | N/A | N/A |
| 67108864 | 157353 | N/A | N/A |
| 134217728 | 321354 | N/A | N/A |
| 268435456 | 628167 | N/A | N/A |
| 536870912 | 1299690 | N/A | N/A |
| 1,074E+09 | 2593150 | N/A | N/A |

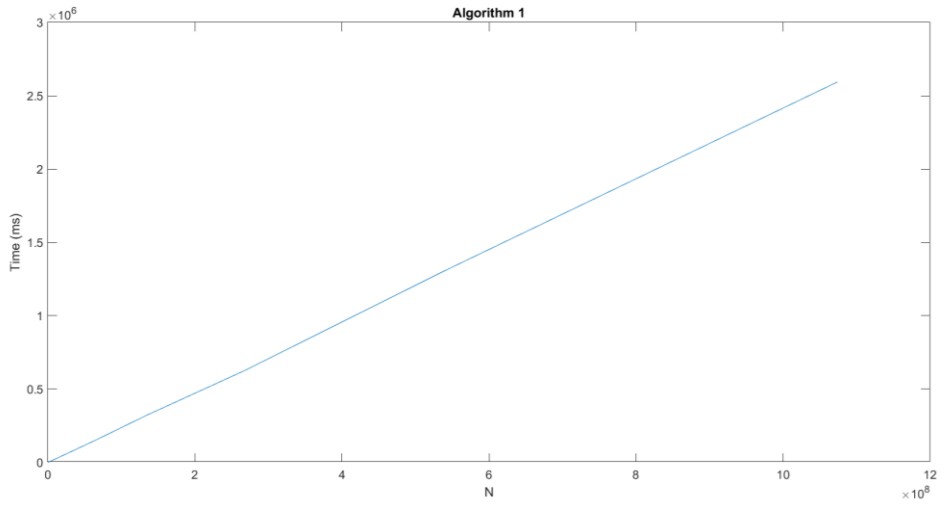


Figure 2. Operation Times for Algorithm 1

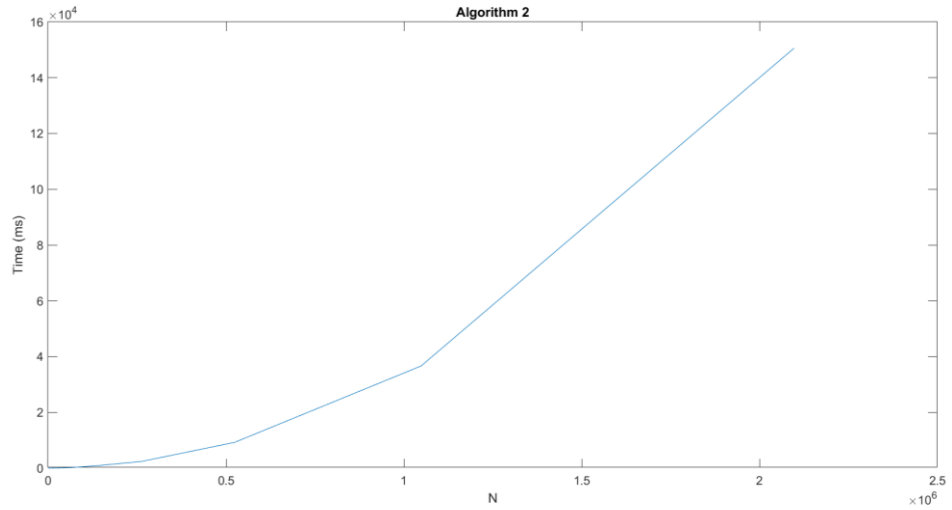


Figure 3. Operation Times for Algorithm 2

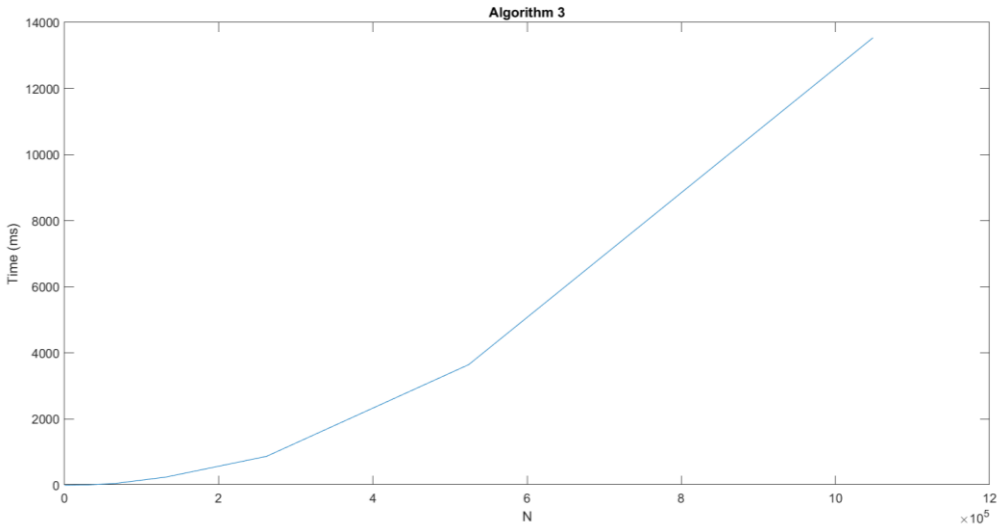


Figure 4. Operation Times for Algorithm 3

From Figure 1, we can see that our analysis is consistent with our obtained data for time complexities. Operation time for algorithm 1 increases linearly and therefore growth rate is constant. Growth rates of algorithm 2 and algorithm 3 increase as arrays becomes larger.

In terms of time consumption, algorithm 2 is better than algorithm 1 until N = 65536 and algorithm 3 is better than algorithm 1 until N = 131072. This means algorithm 1 is the worst for small sized arrays. However, for large sized arrays, algorithm 1 is the best algorithm, which can be seen from Figure 1. In terms of time consumption, algorithms 3 is always better than algorithm 2.