**CS201, Spring 2022-2023**

**Homework Assignment 2**

**Deniz Polat**

**22103590**

**Section 1**

I have studied the four different solutions in order to solve the maximum subsequence sum problem and the time complexity of each solution. Let's assume that the arrays that are used for these solutions have sizes N. The first algorithm, consisting of three nested for loops, has a time complexity of $O(N^3)$. The second algorithm is similar to the first algorithm but one of the for loops is avoided, resulting in a time complexity of $O(N^2)$. The third algorithm uses recursion for solution with the helper of "max3" and a driver function and has a time complexity of O(N*log(N). The fourth algorithm, which uses a single for loop has a linear time complexity O(N).

After implementing these solution algorithms in C++, I have created a "main.cpp" program to create different sized arrays with random values in them. I've called these four functions for each different array and used the standard library header "ctime" to compute the execution times for each algorithm. Figure 1 shows the terminal output of this main program. These values are included in Table 1.

```
[d.polat@dijkstra cs201-hw2]$ ./cs201-hw2.out
Algorithm 1: Execution took 0.021 milliseconds for n = 5
Algorithm 2: Execution took 0.002 milliseconds for n = 5.
Algorithm 3: Execution took 0.001 milliseconds for n = 5.
Algorithm 4: Execution took 0.002 milliseconds for n = 5.
Algorithm 1: Execution took 0.003 milliseconds for n = 10.
Algorithm 2: Execution took 0.002 milliseconds for n = 10.
Algorithm 3: Execution took 0.002 milliseconds for n = 10.
Algorithm 4: Execution took 0.001 milliseconds for n = 10.
Algorithm 1: Execution took 0.002 milliseconds for n = 100.
Algorithm 2: Execution took 0.002 milliseconds for n = 100.
Algorithm 3: Execution took 0.001 milliseconds for n = 100.
Algorithm 4: Execution took 0.002 milliseconds for n = 100.
Algorithm 1: Execution took 38.546 milliseconds for n = 1000.
Algorithm 2: Execution took 0.593 milliseconds for n = 1000.
Algorithm 3: Execution took 0.039 milliseconds for n = 1000.
Algorithm 4: Execution took 0.004 milliseconds for n = 1000.
Algorithm 1: Execution took 4044.32 milliseconds for n = 5000.
Algorithm 2: Execution took 14.796 milliseconds for n = 5000.
Algorithm 3: Execution took 0.208 milliseconds for n = 5000.
Algorithm 4: Execution took 0.011 milliseconds for n = 5000.
Algorithm 1: Execution took 31544.7 milliseconds for n = 10000.
Algorithm 2: Execution took 58.951 milliseconds for n = 10000.
Algorithm 3: Execution took 0.394 milliseconds for n = 10000.
Algorithm 3: Execution took 0.024 milliseconds for n = 10000.
```

**Figure 1: Terminal output for the main.cpp**

**Table 1: Execution time due to increasing array size (N).**

| N / Algorithm Number | Algorithm 1 $O(N^3)$ | Algorithm 2 $O(N^2)$ | Algorithm 3 O(N*log(N)) | Algorithm 4 O(N) |
|---|---|---|---|---|
| 5 | 0.021 ms | 0.002 ms | 0.001 ms | 0.002 ms |
| 10 | 0.003 ms | 0.002 ms | 0.002 ms | 0.001 ms |
| 100 | 0.002 ms | 0.002 ms | 0.001 ms | 0.002 ms |
| 1000 | 38.546 ms | 0.593 ms | 0.039 ms | 0.004 ms |
| 5000 | 4044.32 ms | 14.796 ms | 0.208 ms | 0.011 ms |
| 10000 | 31544.7 ms | 58.951 ms | 0.394 ms | 0.024 ms |

The reason why I chose N = 10000 as the upper boundary for my computations is that I got a timeout for the algorithm 1 for the values bigger than 10000.

I have plotted a graph with the values in Table 1 and observed the growth rate differences for the four different algorithms.

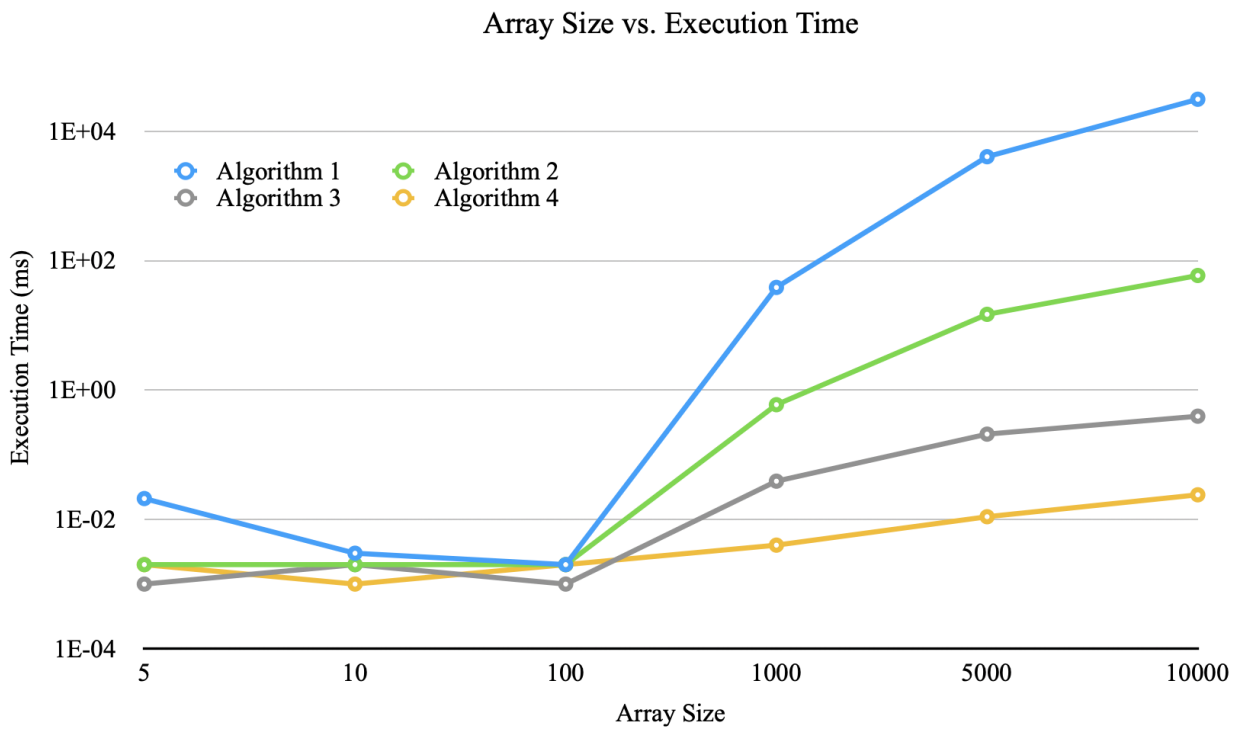


**Figure 2: Array size (N) versus Execution Time graph.**

I chose the values in the y axis to increase logarithmically in order to observe the behavior of the graph more clearly. As it can be seen from Figure 2, the line for the first algorithm increases faster than the other three algorithms, as expected. Algorithm 4 has a smaller growth rate, which is also expected since it has a time complexity of O(N).

After plotting the graph of the outcomes that I obtained from my computations, I plotted another graph (Figure 3) for the expected growth rates of each time complexity.

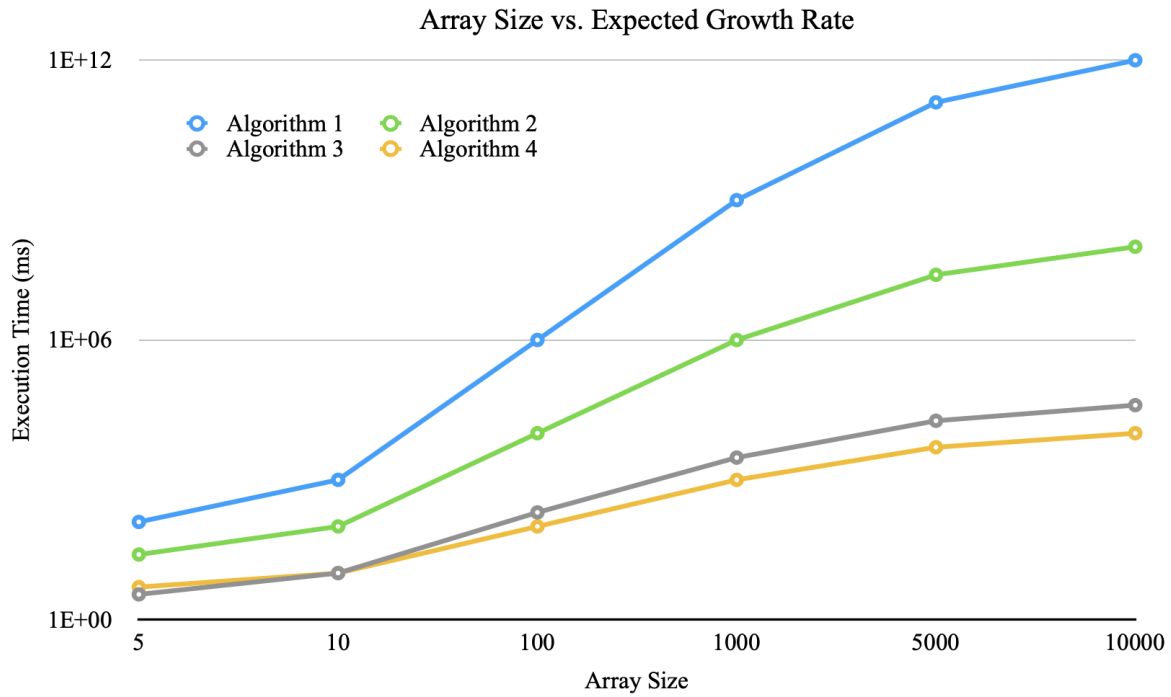


**Figure 3: Array size (N) versus Expected Growth Rate graph.**

Because an algorithm with an input number of n does not have to complete its execution in exactly 100s, the values of the execution times in this graph are more representative than realistic. This graph is also plotted with logarithmic y-axis.

When the two graphs are compared, the practical results are different from the theoretical results until N = 100. The reason for this difference could be because the compiler handles these solutions way faster for such small array sizes, and the differences between these values are hard to read in such a graph.

However, between N = 100 and N = 10000, the growth rates are so similar and gives the correct expected graph behaviors. The time complexity of $O(N^3)$ grows way faster than the time complexities of $O(N^2)$, O(N*log(N)) and O(N) for large numbers. In other words, as N grows, the growth rates become more observable.

In the expected results, O(N*log(N)) and O(N) have very similar behaviors since we are dealing with the powers of 10 and 10-based logarithmic numbers. However, algorithm 3 has a faster growth rate than algorithm 4 in practical results.

Differences like these can occur due to the operating system, RAM, or CPU usage of the computer that I used, because all of these execution time values will be different for different computers, and even for multiple runs on the same computer.

During this assignment, I have used a MacBook Pro with M1 chip and macOS operating system. It has a RAM of 16 GB and Apple M1 chip has 8-core CPU with 4 performance cores and 4 efficiency cores; and a 9-core GPU.