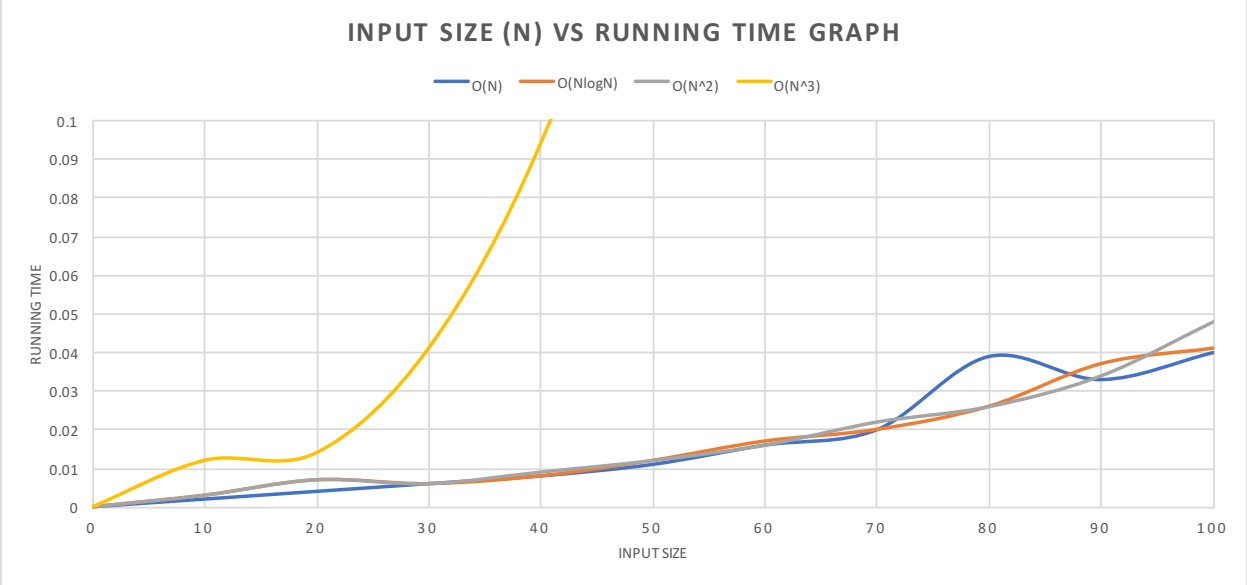
Bartu Atabek, 21602229
CS 201 - Sec. 03 / Ercüment Çiçek
HW 02 - November 30, 2017

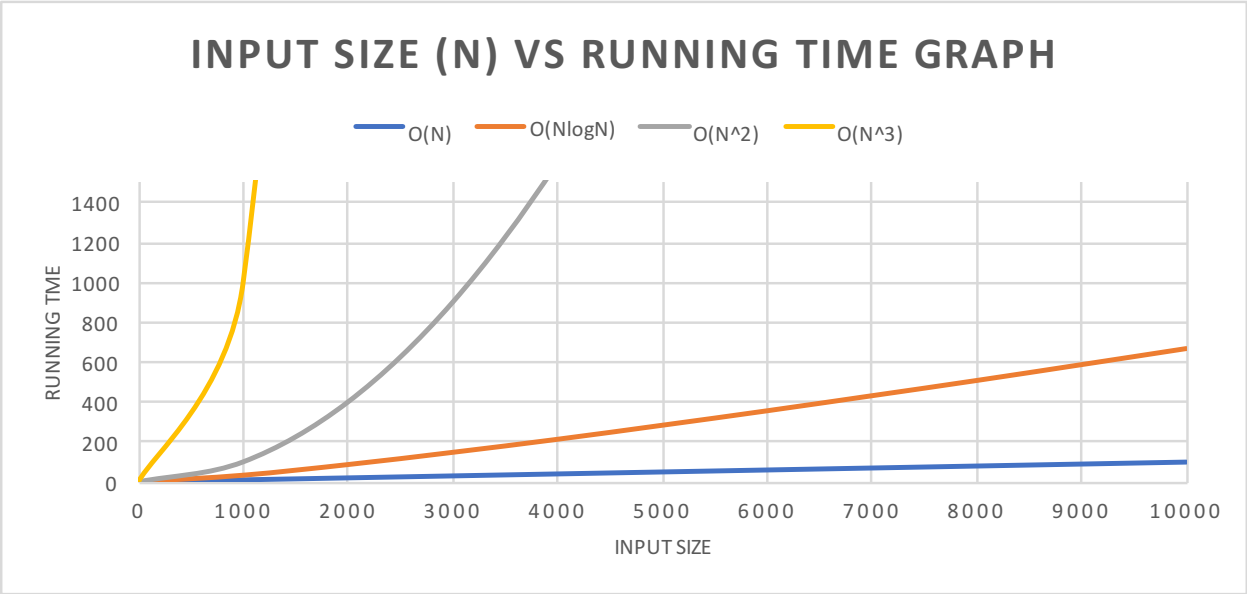
Homework 02

	Algorithm Time			
	1	2	3	4
Input Size	$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
N = 10	0.011	0.006	0.003	0.002
N = 100	1.435	0.069	0.044	0.039
N = 1,000	1296.41	3.667	3.675	3.605
N = 10,000	NA	338.461	370.122	381.08
N = 100,000	NA	36145.1	35563.4	36180.7

Table 1: Running times of 4 different algorithms for maximum subsequence sum in milliseconds



Graph 1: Input size (N) vs running time in milliseconds plot



Graph 2: Theoretical values of Input size (N) vs running time in milliseconds plot

As it is calculated and specified in the explanation of the first algorithm, it has the time complexity of $O(N^3)$ by using the Big-O notation. Apart from the other three algorithms, the first algorithm is by far the longest time taking one so that when an array with size n = 100,000 is inputted, it will do 1,000,000,000,000,000 overall operations that are necessary to complete the task. Because of this growth rate will be increased exponentially with the arrays of large sizes, so it will take a long time to run. From the values obtained from Table 1 it could be observed that for small amount of data this algorithm could be sufficient in order to compute the maximum subsequence sum, however, as the size of the data increases the algorithm becomes inefficient and even can't compute the result as it's shown for the input size of 10,000 and 100,000. When compared with the expected growth rate value of $O(N^3)$ the plot on the graph 1 follows a similar pattern to its theoretical value, however, it is evident that there are some unexpected fluctuations in the graph 1 for the first algorithm.

For the second algorithm which has the time complexity of $O(N^2)$, it is one of the more efficient ones when we compare it to the first algorithm such that the overall required number of operations are 10x times lesser. Although it is 10x faster than the first one it is still not efficient for a large number of data sets for instance for input size n = 1,000 it takes approximately 3 ms whereas for n = 10,000 it takes over 340 ms. When compared with its theoretical value it is obvious that there's a drastic difference between them and the curve of $O(N^2)$ starts to imitate it's theoretical value not until the input size reaches over 10,000.

Third algorithm has O(NlogN) complexity which is quite efficient and maybe the most efficient in cases with large data inputs such as 10,000 and 100,000 compared to every other algorithm. It is quite efficient although as a downside it takes a long time to complete the process when working with large input sizes. In comparison with its theoretical value form the Graph 2, it succeeds to replicate the linear line close enough

with some visible ripples yet it is closer to its theoretical value than the first two algorithms.

Finally, for the fourth algorithm, the time complexity is O(N) which indicates that its constant. Like algorithm three it is quite efficient in many cases and goes head to head with it in cases with large input sizes. Similar to the previous algorithms when the input size increases the time for the operation to complete increases for this algorithm too, however, the growth rate is immensely smaller than the other algorithms since it's a constant function. When compared with its theoretical counterpart from Graph 2 the curve in Graph 1 seems identical to the previous two algorithms and even in some cases exceeds them which is unexpected and in addition, it should have a much smaller growth curve than the others like it's theoretical counterpart in Graph 2.

Because the memory and processors are not just running for these four algorithms the first graph differs from the expected theoretical one as shown in Graph 2. Moreover, when working with inputs that have a considerable amount of data such as n = 10.000 or n = 100.000, the overall running time takes an unacceptably long time. Besides that, during the process, the machine is also running other programs, using memory and CPU. Because of the factors listed above and the differences in the operating systems, multi-cores and turbo boosts of different computers the time complexity is having larger errors when we are working with these larger input sizes shown in Table 1.

Conclusively, usually for an algorithm to be fast and efficiently handle a large amount of data they should have O(NlogN) time since it is the most efficient one as shown in the table and graphs.

Computer Specifications

Hardware Overview:

Model Name:	MacBook Pro
Model Identifier:	MacBookPro10,1
Processor Name:	Intel Core i7
Processor Speed:	2.4 GHz
Number of Processors:	1
Total Number of Cores:	4
L2 Cache (per Core):	256 KB
L3 Cache:	6 MB
Memory:	8 GB
Boot ROM Version:	MBP101.00F2.B00
SMC Version (system):	2.3f36