Homework Assignment II

CS 201-01

2022-23 Fall semester

Yassin Younis

22101310

CS

Algorithm I:

The first algorithm performs some constant time initializations that take time c_1, then runs through a while loop $\frac{(n+1)}{2}$ times and in each iteration after initializing some variables and performing some operations that take constant time c_2, it runs through another for loop n-k times, where k is the number of while loop iterations performed so far, to find the maximum number to eliminate. Inside the nested loop more operations are performed that take constant time c_3. Yielding the following function:

$$c_1 + \frac{c_2 c_3}{2}(n^2 - nk + n - k)$$

Dropping lower order terms and eliminating constants we get n^2 or in other words O(n^2).

Algorithm II:

The second algorithm simply quick-sorts the array then returns the middle element which takes constant time c_1 to return the middle element and whatever time it takes to perform quicksort: To perform quicksort we first divide array into 2 around a randomized pivot (for this example), where the array to the left of the pivot contains values strictly less than the pivot and the array to the pivot's right contains values bigger than the pivot. The process of recursively dividing the array can be represented by $log(n)$ and the process of running through the array and making sure values to the right are bigger than the pivot and values to the left less takes n time, multiplied by some initializations and constants time operations c_2, we get:

$$c_1 + c_2 nlog(n)$$

Dropping lower order terms and eliminating constants we get $nlog(n)$ or in other words O($nlog(n)$). However note that quicksort can sometimes run in $O(n^2)$ when the array passed to the algorithm is already sorted.

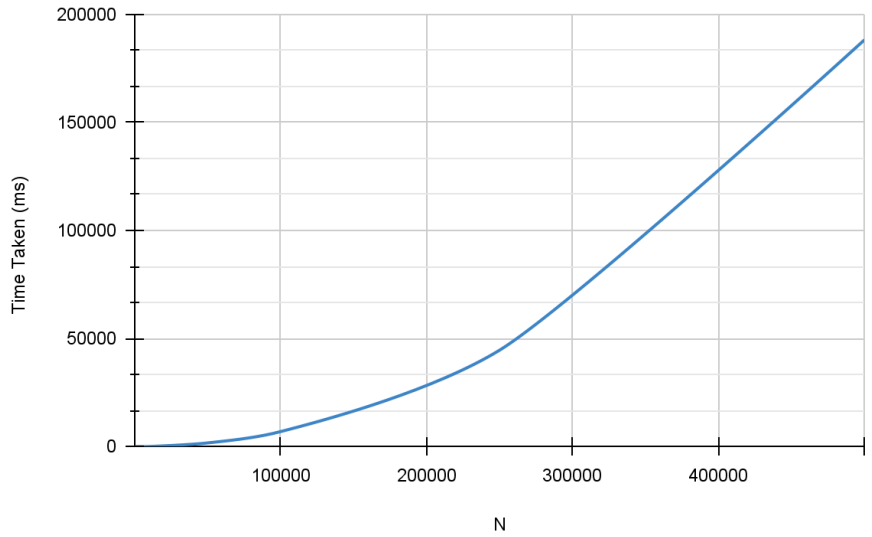
Algorithm III:

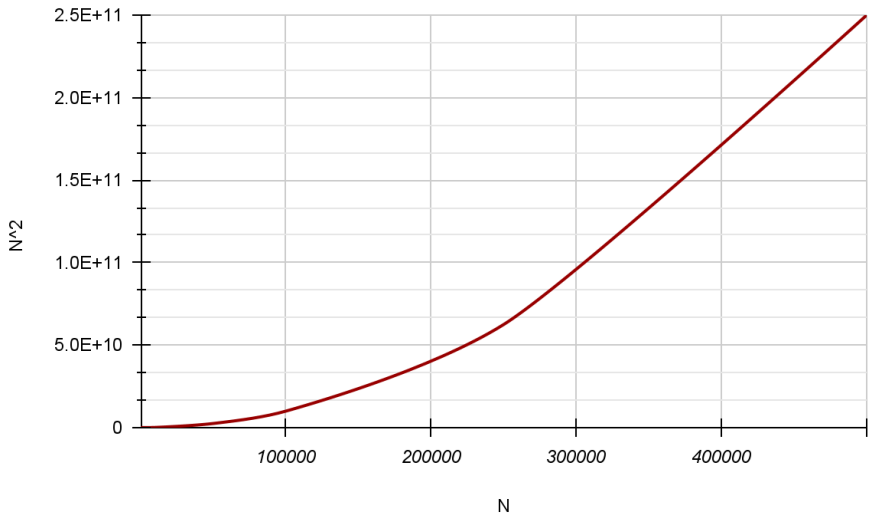
The previous algorithm is much more efficient than the first one but is it really necessary to sort the whole array to find the middle element? And can we take advantage of the pivot picking process instead of picking a random pivot? These are all challenges and questions the third algorithm answers. The third algorithm firstly divides the array into arrays of 5 integers and recursively gets the median of each of these arrays, then gets the median of these medians and selects that value as the pivot. The algorithm now acts like quicksort, in the fact that it divides the array in two halves and makes it so that the elements to the left of the pivot are less than the pivot and the elements to the right of the pivot are bigger than the pivot. The algorithm, unlike the previous, doesn't sort or even look at the array which doesn't have the index of the median. The algorithm keeps doing so until the index of the pivot equals the index of the median. It would seem that the algorithm would be running with $nlog(n)$ complexity, however that implies that the algorithm looks at all n items $log(n)$ times, which is not the case since we ignore half the array during pivoting and dividing. According to multiple sources and the tests conducted below, the algorithm runs at $O(n)$ complexity.

Time in milliseconds (ms) to find median for different n's:

n	Algorithm I	Algorithm II	Algorithm III
1	0.001ms	0ms	0.001ms
5	0.001ms	0.001ms	0.001ms
10	0.001ms	0.001ms	0.04ms
50	0.004ms	0.004ms	0.012ms
100	0.013ms	0.01ms	0.02ms
500	0.189ms	0.044ms	0.04ms
750	0.432ms	0.069ms	0.065ms
1000	0.788ms	0.095ms	0.085ms
5000	20.027ms	0.637ms	0.331ms
10,000	74.843ms	1.086ms	0.52ms
25,000	446.078ms	2.807ms	1.279ms
50,000	1720.28ms	5.888ms	2.566ms
60,000	2440.15ms	7.18ms	3.24ms
75,000	3828.03ms	9.266ms	3.918ms
90,000	5434.95ms	10.93ms	4.709ms
100,000	6944.29 ms	12.605ms	5.137ms
150,000	15554.3ms	19.276ms	7.991ms
200,000	28121.3ms	25.677ms	10.094ms
250,000	44584.3ms	34.251ms	13.066ms
300,000	63156.9ms	43.377ms	16.669ms
400,000	112163ms	61.764ms	23.071ms
500,000	188180ms	80.255ms	29.505ms

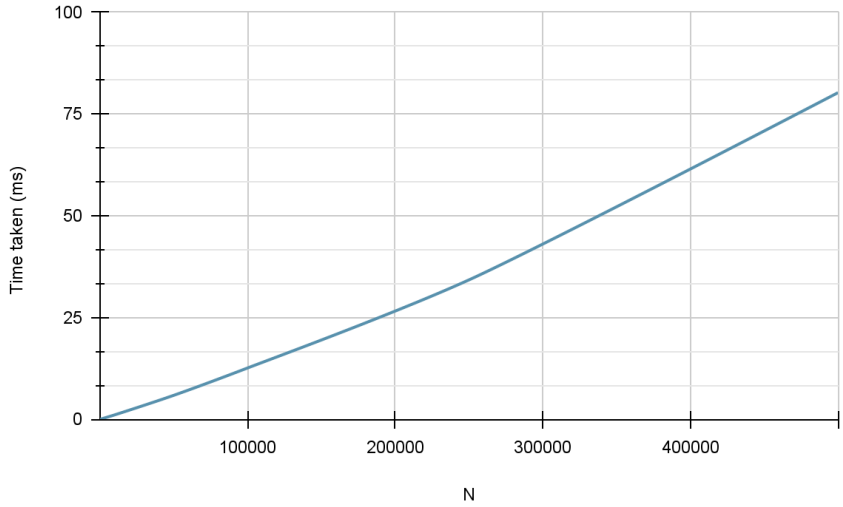
Algorithm I's Graph:

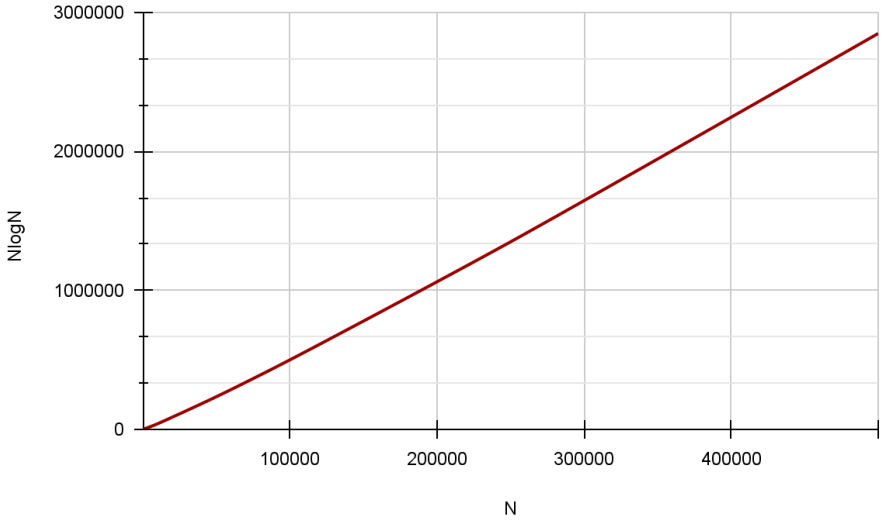




The graph clearly exhibits the O(n^2) behavior we were expecting. Both theoretical and real result graphs look similar. However, the algorithm proved itself to be a very efficient algorithm for big data. This algorithm should only be used for small data; because for values of N less than a 100 the algorithm performs better than algorithm III, the best algorithm. So when one is expecting small inputs it is better to use this algorithm or in general this idea of nested loops.

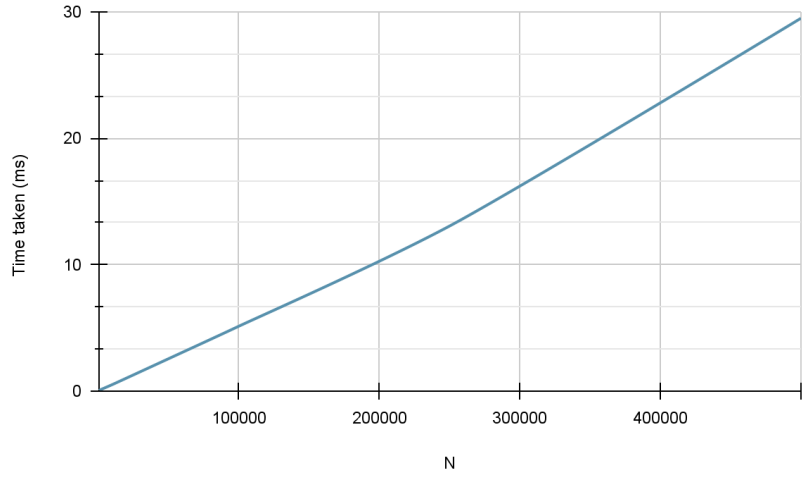
Algorithm II's Graph:

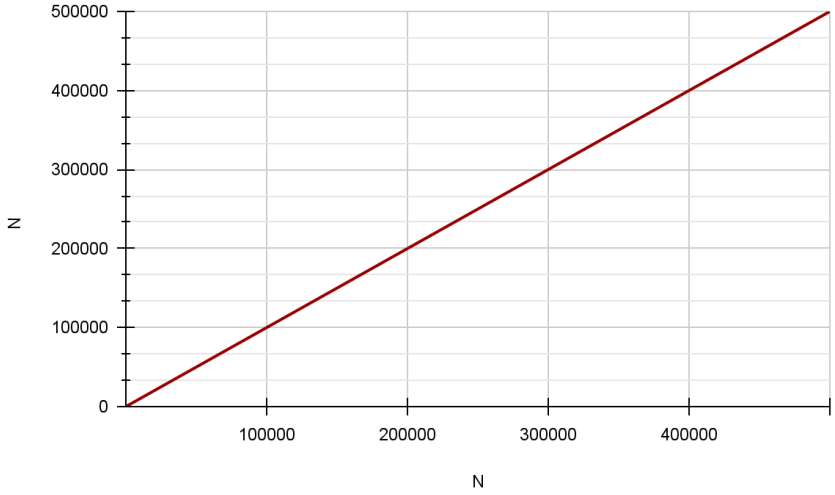




Both theoretical graph and actual result graph look the same although they both look linear, however when we compare it to the next algorithm's graph it won't look as linear. I think this algorithm is the algorithm to use if you don't know how many N's your program will receive, it runs the best out of both algorithms for small N and it doesn't do very bad compared to the third algorithm for big values, which runs at constant time.

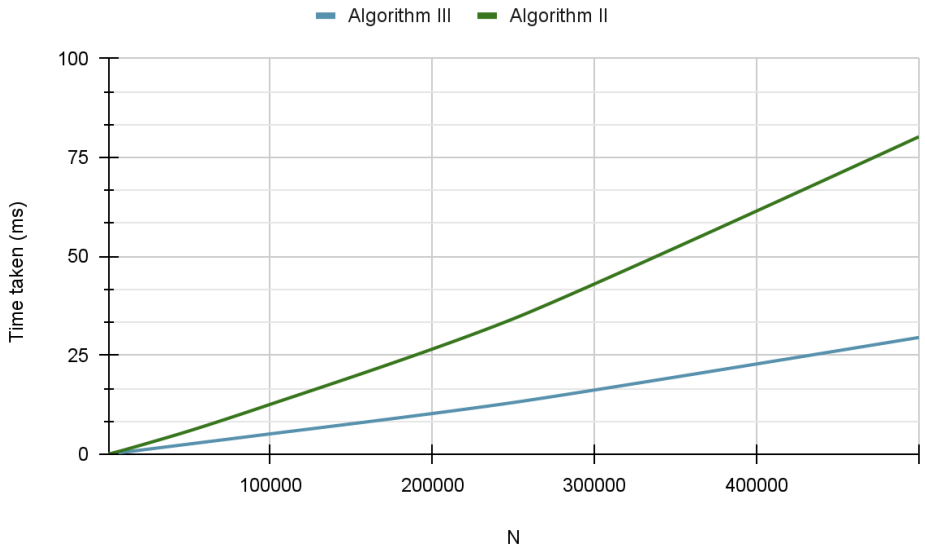
Algorithm III's Graph:





The graph clearly exhibits the linear behavior shown in the theoretical graph, proving that this algorithm is the best algorithm out of all three for big data.

Algorithm III vs Algorithm II Graphs:



In this graph differences between algorithm II, and III become a little more clear. Algorithm II looks to have slow exponential growth when put next to the very linear algorithm III.

Concluding remarks:

Overall, I would call this a successful result, the graphs look very consistent and similar to their theoretical counterparts. Slight Inconsistencies in the table might be caused by factors like thermal throttling, since the laptop used has a notoriously hot CPU, and maybe some random arrays generated were harder for some algorithms to process than for others. I learned a lot from this experiment/assignment: for example, when one needs to loop twice through an array of big data to perform a certain task or action, he or she might want to avoid doing so and stick to a "divide and conquer" approach like the one used in the last two algorithms.

Relevant specifications for device used to conduct these tests:

Processor: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz

Installed RAM: 16.0 GB (15.8 GB usable)

Operating System: Windows 11 Pro 21H2 Build 22000.1219

System Type: 64-bit operating system, x64-based processor