CS 201 - HOMEWORK 2

Time Complexity Analysis of Find Median Algorithms

Section: 1

Name: Dilara Mandıracı

ID: 22101643

Description: Time complexity analysis of three different algorithms for finding the median of the given array.

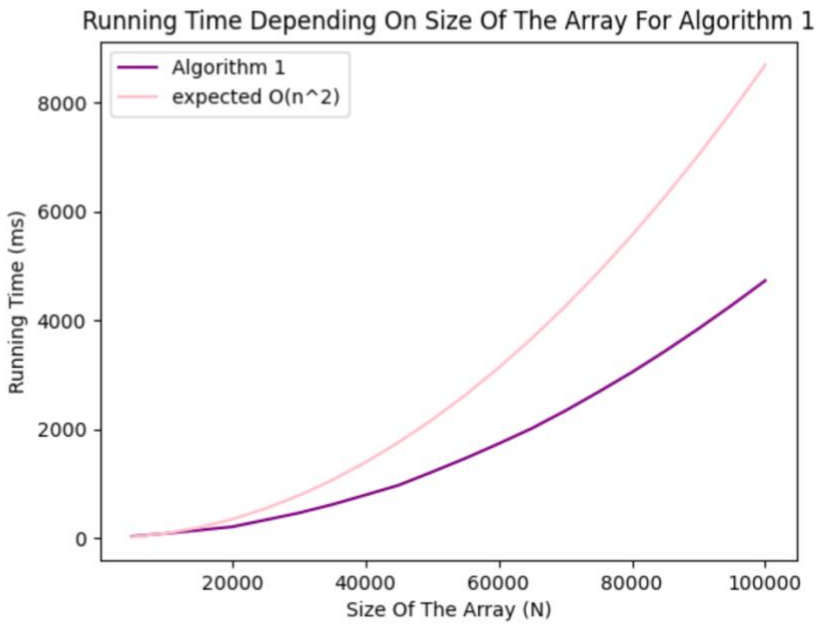
Algorithm 1

This algorithm works by first finding the element with the maximum value in the array starting from the index, which is the count variable, then continues with swapping the maximum valued element with the index that the count variable holds. This algorithm makes the first half of the array sorted in descending order because it first puts the largest element to the index of the count, then continues with the second largest, third largest, and so on. Traversing the array and finding the element with the largest value in the given interval (count – n) takes O(n) time. The algorithm finds the element with the biggest value for (n+1) / 2 times, and this can be shown with the big-Oh notation as O(n). It means that these time complexities need to be multiplied because they are in nested positions. To clarify what nested means here, we can say the algorithm says finding the maximum element of an array procedure needs to be done (n+1)/2 times. T(n) = O(n) * O(n) gives us O(n^2) time complexity for that algorithm.

Table shows execution time in milliseconds:

n (size of the array)	Find Median 1
10	0.006
20	0.009
50	0.012
70	0.015
100	0.035
500	0.604
1000	1.748
5000	32.782
10000	86.077
100000	4678.70
500000	117284

Graph for the algorithm 1:



Data in the table I preferred to show results from a wide range of array sizes, starting with size ten and ending with half a million. From time complexity analysis for algorithm 1, I found that it works in O(n^2) time complexity. The big-Oh notation indicates the closest upper bound of an algorithm; however in real time it may have a constant coefficient such as k next to N2. When drawing the graph, I preferred to collect data from a determined interval whose starting point is 20000 and ending bound is 100000 for array size. I increased the array size to 5000 in every step to get continuous data and a well-shaped graph to see if the time complexity of our algorithm looks like the complexity O(n^2). As can be seen from the graph, running time increases in a parabolic manner, just like O(n^2). To converge the lines, I needed to add a constant coefficient to our expected growth rate.

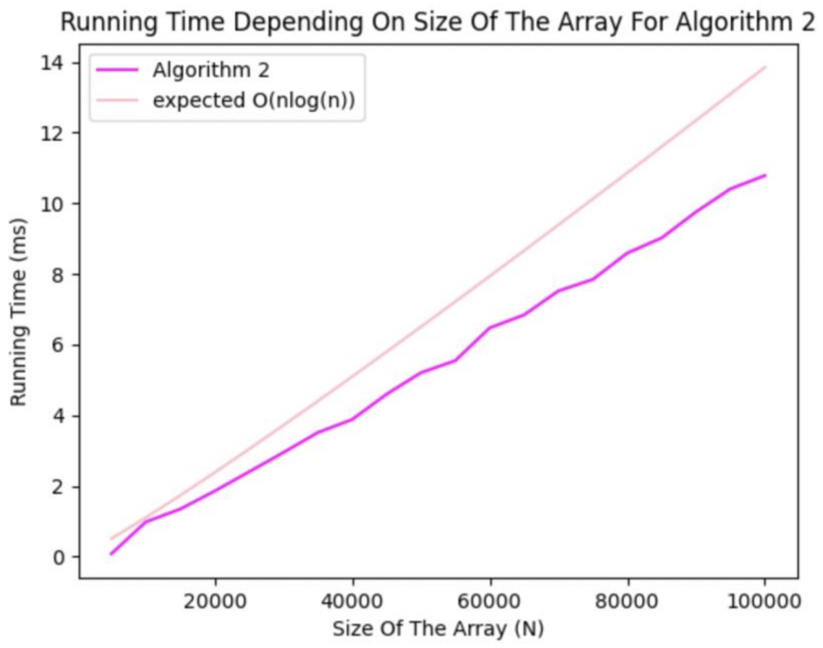
Algorithm 2

This algorithm first sorts the array elements in descending order by using the Quicksort method, a well-known sort algorithm with the average time complexity O (n log n). For the quick sort algorithm in the average case, if j is the index of a random pivot, time complexity will be T (n) = T (j) + T (n - j). If we solve this relation with the correct methods, we will get T (n) = O (n log n). After sorting, it picks the element at the index size/2 of the array, even if the size is even or odd in our case. It means the algorithm selects the median in constant time in all cases. If we look at the whole picture, the algorithm first sorts with complexity O (n log n) and then picks the median with O (1). The dominant complexity will be the time complexity of this algorithm which is O (n log n) since more extensive time determines the complexity.

Table shows execution time in milliseconds:

n (size of the array)	Find Median 2
10	0.002
20	0.004
50	0.007
70	0.010
100	0.019
500	0.110
1000	0.149
5000	0.862
10000	1.003
100000	10.795
500000	58.355

Graph for the algorithm 2:



Data in the table I preferred to show results from a wide range of array sizes, starting with size ten and ending with half a million. The graph array size range is the same as algorithm 1 to keep the same amount of constantly growing data. From time complexity analysis for algorithm 2, I found that it works in average and best case with O(n*log(n)) time complexity. In the worst case, the QuickSort algorithm works with $O(N^2)$, so our algorithm works like that because it uses QuickSort followed by constant O(1) time return statement too. In our graph, there are some up and down points; this situation may have resulted from the fact that there are different time complexities for different cases.

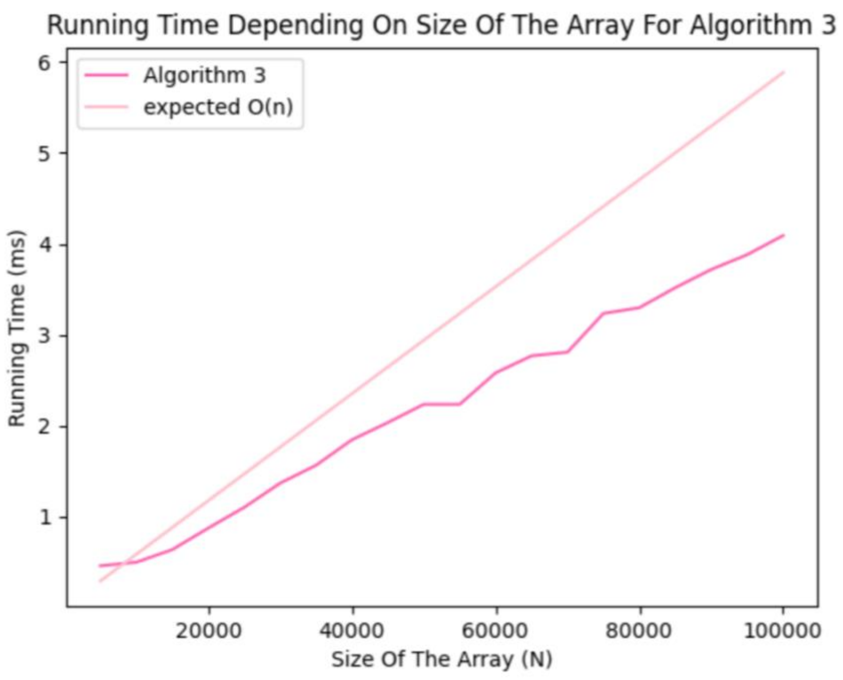
Algorithm 3

This algorithm works as finding the $n/2^{th}$ smallest element. If $n < 6$, it just sorts and returns the median in constant time, O(1). If $n>=6$, with O(n), it divides the given array into subarrays with length 5, the last one may have less than 5. It sorts each subarray and finds their medians with O(n) time. Then recursively do this again. In other words, it will do the same procedure as the medians array, which gives the algorithm the name. It needs to find the median of medians again until it finds the correct median. Hence, this algorithm finds the $n/2^{th}$ smallest element in linear time, O(n).

Table shows execution time in milliseconds:

n (size of the array)	Find Median 3
10	0.002
20	0.006
50	0.013
70	0.015
100	0.026
500	0.140
1000	0.128
5000	0.465
10000	0.502
100000	4.084
500000	20.432

Graph for the algorithm 1:

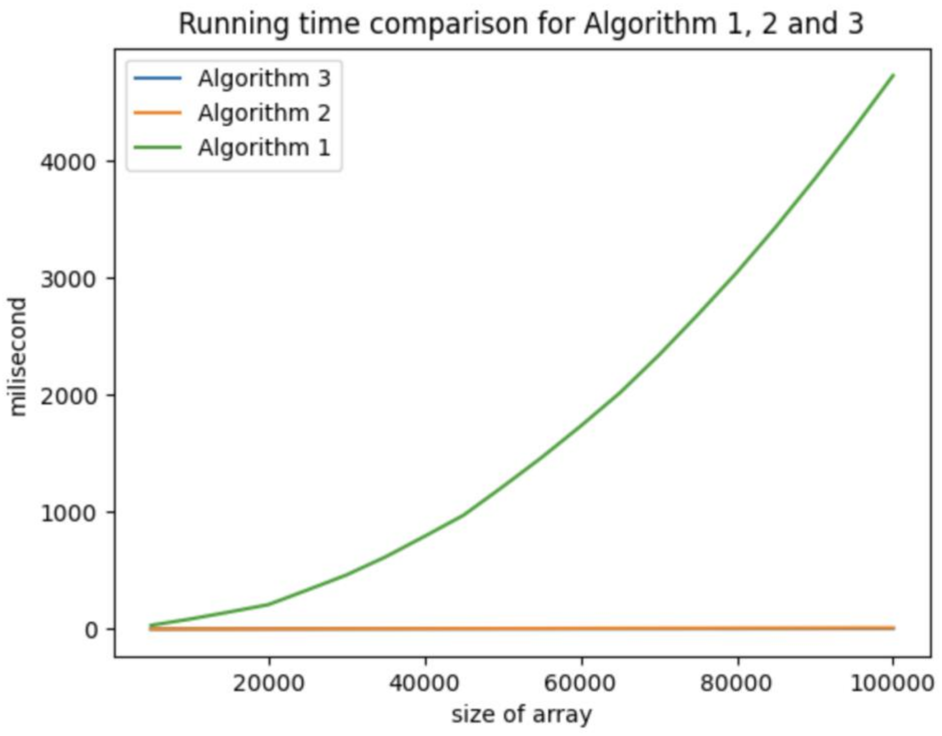


Graph and table ranges are precisely the same with algorithms 1 and 2 in order to compare their results in the same graph later. Algorithm 3 works in linear time complexity, shown as O(N), but again in real-time, it may have some constant k. Some up and down points may result from array features, repetition of numbers, etc. However, in the bigger picture, it is easily seen that it works in linear time. In addition, it can be seen from the data table. For example, in the data table running time of the median of the medians' algorithm is approximately 4 milliseconds for size 100000, and it is about 20 milliseconds for size 500000. It can be analyzed like that if the size is 5 times larger, then the running time should be 5 times larger according to O(N) time complexity.

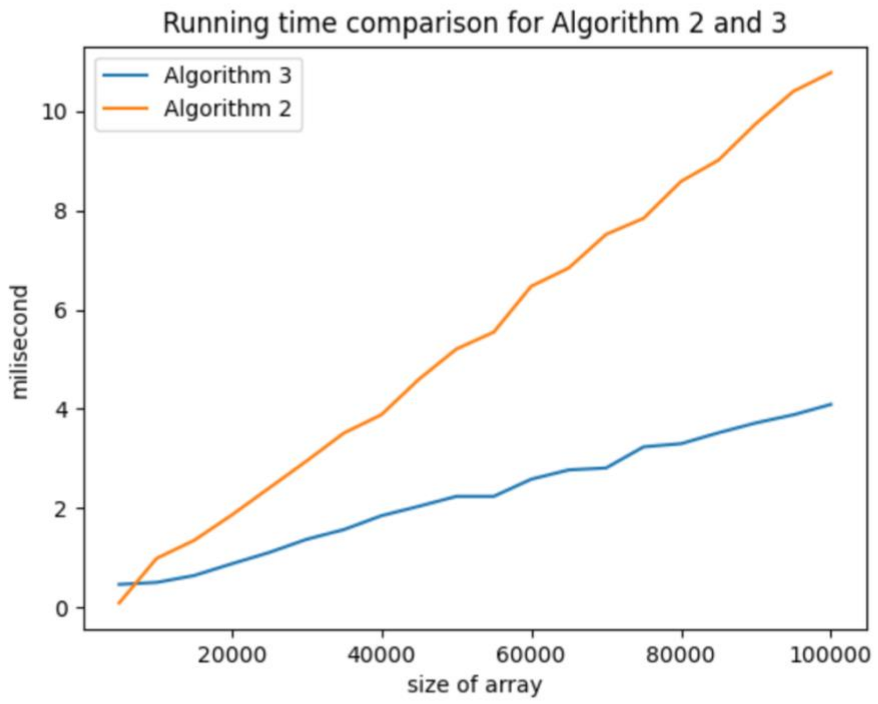
n (size of the array)	1	2	3
5000	32.109	0.0855	0.461
10000	85.434	0.986	500
15000	145.745	1.348	0.639
20000	208.387	1.857	0.875
25000	333.956	2.402	1.103
30000	462.115	2.948	1.369
35000	616.492	3.513	1.566
40000	792.275	3.882	1.846
45000	973.291	4.597	2.036
50000	1215.68	5.206	2.236
55000	1466.91	5.548	2.367
60000	1735.76	6.474	2.582
65000	2018.52	6.841	2.769
70000	2340.93	7.52	2.809
75000	2687.92	7.844	3.235
80000	3049.36	8.589	3.298
85000	3439.19	9.017	3.518
90000	3851.4	9.755	3.718
95000	4282.36	10.405	3.881
100000	4733.53	10.781	4.091

Table 4

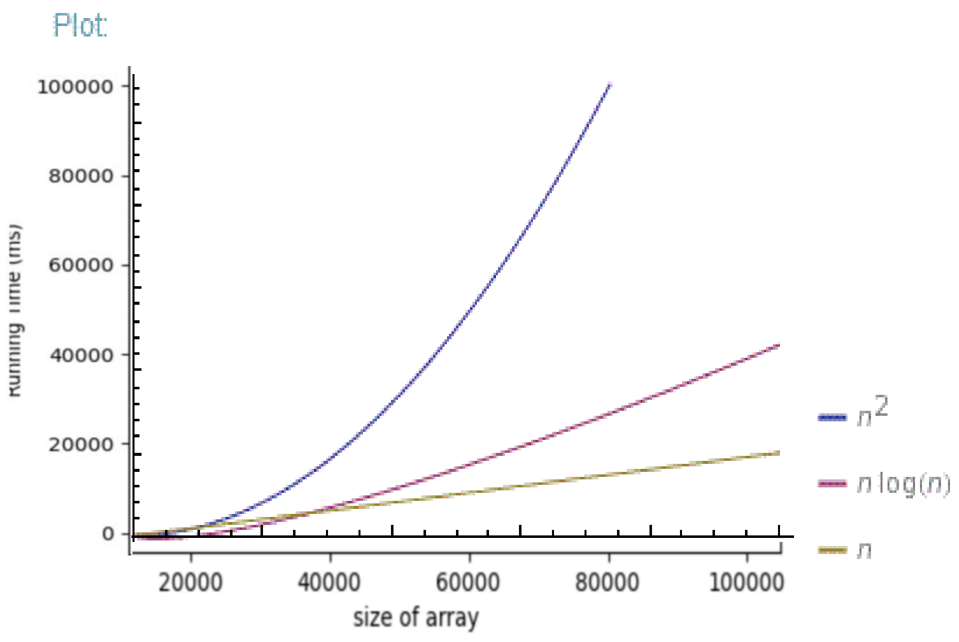
Table 4 shows the data collected from both three algorithms for arithmetically increased sizes which are used to draw all the graphs above. As I mentioned before, I checked them with various sizes in the other tables. In this table, there are also different results, but I increased them equally in every step to see the resulting graph continuously. The data above shows that an algorithm that has the time complexity O(n2) is not a good option if we have solutions with better complexities. Results for the maximum size of that table shows that the time of algorithm 2 and 3 is less than the result of the minimum size of algorithm 1. These three functions use different algorithms, but in the end, they give the same solution; in other words, each of them finds the median. I saw how important time complexity analysis is while finding a solution to a problem.



The above table shows the comparison of the results of these three algorithms. Since the growth rate O(N^2) is so much bigger than the others, it is hard to compare all three in the same graph. The results of algorithm 1 are so bigger that we cannot see the others' results. For example, the largest time for algorithm 3, is 4 millisecond and 10 milliseconds for algorithm 2, whereas the largest time for algorithm 1 is approximately 5000 milliseconds. We saw the comparison of those three and realized that Algorithm 1 dominates the graph. So, to compare algorithms 2 and 3, I removed the line of Algorithm 1 from the graph. In the below table growth rate of running time for Algorithm 2 is faster than Algorithm 1 because while algorithm 2 runs in O(n*log(n)) time complexity, algorithm 3 works in O(n). We can also see these results from data tables. It means if the array size is equal for all three, when it is duplicated first algorithm's running time increases 4 times, the second algorithm rises 2 times, third one will between these two, but the result will be nearer to the result from algorithm 2. Since the quantity of log n is much smaller than n, n*n will be much bigger than n * log(n).



In general, the comparison of O(N2), O(n*log(n)), and O(N) look like in the below graph. Linear time has the lowest growth rate, and quadratic time has the biggest. There are huge differences between the expected results with our obtained data; as I mentioned above, this may result from computer specifications and coefficients of the growth rate. But it didn't change their comparison. I mean, we can see algorithm 1, which has the higher growth rate O(N2), and it has a parabolic shape as it is expected. O(N2) grows faster than O (N log(N)), and it grows faster than O(N). I saw the difference in this homework with collecting data from three algorithms that work in different complexities and understanding how I should analyze such cases.



Specifications of the Computer:

Model Identifier: MacBookPro17,1

Chip: Apple M1

Total Number of Cores: 8 (4 performance and 4 efficiency)

Memory: 16 GB

System Firmware Version: 7459.141.1

OS Loader Version: 7459.141.1