CS 201, Fall 2022
Homework Assignment 2
Due: November 23, 2022, 23.59

Zeynep Su Uçan
22103044
Section 3

Summary

The aim of this assignment is to study the solution to find the "median" of an array of n integers. The median of an array of integers is the middle element of the array if the numbers in the array are sorted. We are given three different algorithms that study the problem of finding the median. All of these algorithms find the median of a randomly generated, unsorted array of integers in a different way. And most importantly we analyze the difference between these algorithms' time-complexities. Firstly, we create different sized arrays which consist of numbers between [-size, size]. This way we can observe the change in running time for both same sized arrays with different algorithms and different sized arrays with the same algorithm. To see a high range in the running time I preferred using arrays of 5, 10, 20, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000 integers. Additionally, my computer could not handle any array with more integers than 600000. Now I will describe how the complexity of each algorithm is calculated and analyze the results I get after implementing them.

FIND_MEDIAN_1:

The first algorithm finds the median of a randomly generated array by searching through the integers to find the maximum number and swaps it with the first element of the array, then continues doing this with the first half of the array because after sorting the first half of the array, the number we find is the median and there is no longer need to sort the rest of it. Looking more into the algorithm, there are two loops: a while and a for loop. The while loop iterates through the first half of the array whereas the for loop starts from the (n+1)th number and goes through the rest of the array. The while loop equates the maxIndex to the index it is on and the maxValue to this maxIndex's value for the first half and has a time complexity of O(n) even though it does the same thing n/2 times because in Big O notation the constants are not taken into account. Additionally, the for loop is nested in the while loop and when going through the numbers in the array it tries to find the maximum number to swap. It starts this search from the index count and continues until it reaches n (size of the array). This part also has a time complexity of O(n) even though it doesn't repeat this action n times each time because the number of times the action is done depends on the array size which is n. To sum up, this algorithm has a time complexity of $O(n^2)$ because the second loop (O(n)) is nested in the first one (O(n)).

FIND_MEDIAN_2:

The second algorithm finds the median of the randomly generated array by sorting the whole array and simply taking the middle element. The crucial part of this algorithm is that it uses a specific sorting algorithm which is called QuickSort. This algorithm uses divide and conquer technique and has an average time complexity of O(n*log(n)). Finding the middle of the sorted array only takes O(1)-time. Therefore, the overall time complexity is also O(n*log(n)).

FIND_MEDIAN_3:

The third algorithm uses the QuickSelect algorithm which is similar to but not quite the same as the QuickSort algorithm. This algorithm originally has O(n*log(n)) average time-complexity and O(n) as worst case time complexity however it can give an optimal worst-case performance with a little modification. The QuickSelect algorithm's aim is to find the kth largest element and to do this it doesn't recursively search both sides of the partition like QuickSort but only one. It uses the median of medians algorithm which is a selection algorithm that divides the array into pieces for example into smaller arrays of 5 integers and finds each of their medians. Then, it finds the median of these medians and uses this as a pivot in the quickSelect algorithm. This way both the best and the worst time complexities become O(n).

The output of the code:

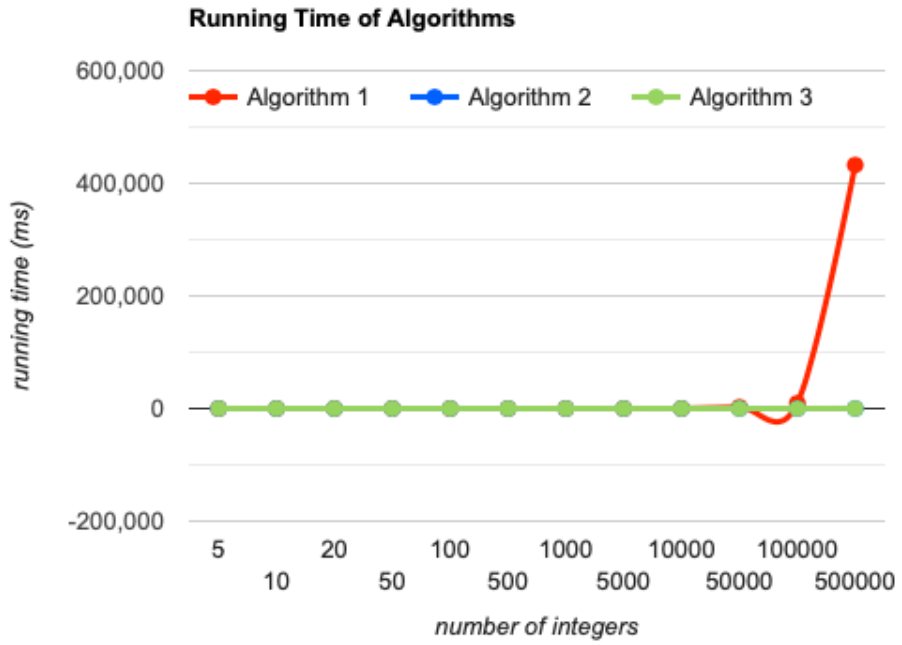
```
Enter the size of array: 5
Execution took 0.007 milliseconds.
Execution 2 took 0.003 milliseconds.
Execution 3 took 0.003 milliseconds.
Would you like to continue? (write yes or no)
yes

Enter the size of array: 10
Execution took 0.006 milliseconds.
Execution 2 took 0.003 milliseconds.
Execution 3 took 0.171 milliseconds.
Would you like to continue? (write yes or no)
yes

Enter the size of array: 20
Execution took 0.008 milliseconds.
Execution 2 took 0.005 milliseconds.
Execution 3 took 0.014 milliseconds.
Would you like to continue? (write yes or no)
yes

Enter the size of array: 50
Execution took 0.043 milliseconds.
Execution 2 took 0.016 milliseconds.
Execution 3 took 0.028 milliseconds.
Would you like to continue? (write yes or no)
yes

Enter the size of array: 100
Execution took 0.063 milliseconds.
Execution 2 took 0.034 milliseconds.
Execution 3 took 0.057 milliseconds.
Would you like to continue? (write yes or no)
yes

Enter the size of array: 500
Execution took 0.742 milliseconds.
Execution 2 took 0.131 milliseconds.
Execution 3 took 0.127 milliseconds.
Would you like to continue? (write yes or no)
yes
```

```
Enter the size of array: 1000
Execution took 2.804 milliseconds.
Execution 2 took 0.266 milliseconds.
Execution 3 took 0.192 milliseconds.
Would you like to continue? (write yes or no)
yes

Enter the size of array: 5000
Execution took 48.243 milliseconds.
Execution 2 took 1.046 milliseconds.
Execution 3 took 0.659 milliseconds.
Would you like to continue? (write yes or no)
yes

Enter the size of array: 10000
Execution took 136.704 milliseconds.
Execution 2 took 1.587 milliseconds.
Execution 3 took 0.882 milliseconds.
Would you like to continue? (write yes or no)
yes

Enter the size of array: 50000
Execution took 2434.47 milliseconds.
Execution 2 took 8.231 milliseconds.
Execution 3 took 3.744 milliseconds.
Would you like to continue? (write yes or no)
yes

Enter the size of array: 100000
Execution took 9557.56 milliseconds.
Execution 2 took 17.144 milliseconds.
Execution 3 took 6.694 milliseconds.
Would you like to continue? (write yes or no)
yes

Enter the size of array: 500000
Execution took 432525 milliseconds.
Execution 2 took 94.374 milliseconds.
Execution 3 took 35.639 milliseconds.
Would you like to continue? (write yes or no)
no
```

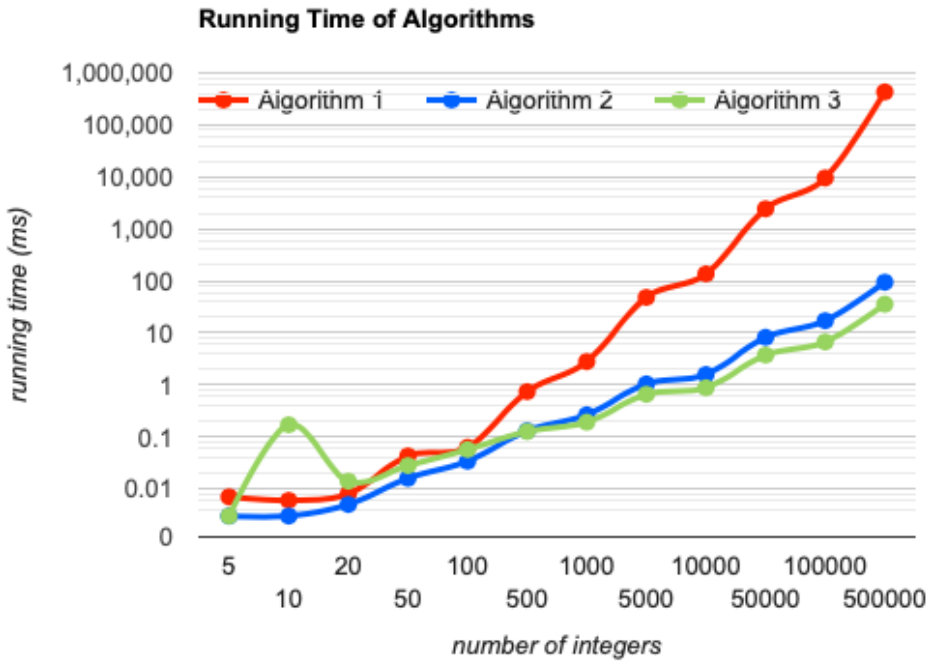
The table containing the results of the code:

	FIND_MEDIAN_1	FIND_MEDIAN_2	FIND_MEDIAN_3
5 integers	0.007 ms	0.003 ms	0.003 ms
10 integers	0.006ms	0.003 ms	0.171 ms
20 integers	0.008 ms	0.005 ms	0.014 ms
50 integers	0.043 ms	0.016 ms	0.028 ms
100 integers	0.063 ms	0.034 ms	0.057 ms
500 integers	0.742 ms	0.131 ms	0.127 ms
1000 integers	2.804 ms	0.266 ms	0.192 ms
5000 integers	48.243 ms	1.046 ms	0.659 ms
10000 integers	136.704 ms	1.587 ms	0.882 ms
50000 integers	2434.47 ms	8.231 ms	3.744 ms
100000 integers	9557.56 ms	17.144 ms	6.694 ms
500000 integers	432525.0 ms	94.374 ms	35.639 ms

The plot of the running time of the algorithms:



Plot 1



Plot 2

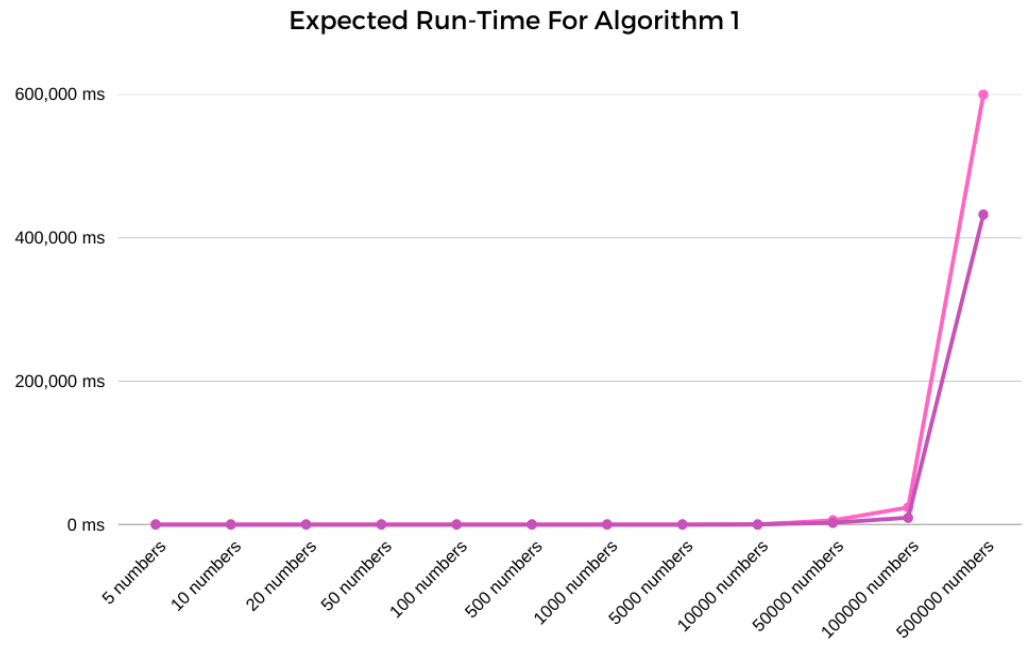
Above, you can see 2 plots: both are graphs showing the running time of the three algorithms. In the first one both the x and the y axis increase linearly, in the second while x-axis increases linearly, the y-axis (running time) is increasing logarithmically. I decided to put both of these graphs to better understand both the relation between all algorithms and each one within itself. Because the range of number of integers (from 5 to 500000) and the run time (0.007 to 432525) were very high this was one of the ways to clearly see the results. Now considering this:

Firstly, as it can be seen very clearly from both the table and the plots, regardless of the algorithm the time it takes to find the median of a random array increases as the size increases. Secondly, if all cases are considered, between the three algorithms, the first one usually takes the most time and the third one is the most efficient.

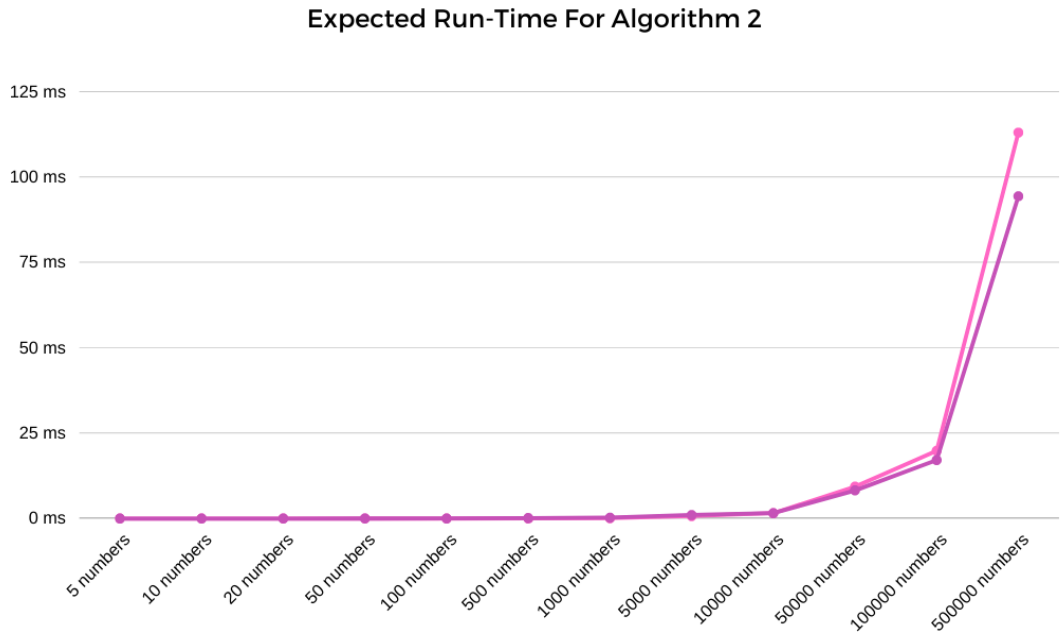
At the beginning, when the size of the array is smaller, the running time of all three algorithms are very close. For example, for the array with 5 integers the first algorithm takes 0.007 ms and the other two take 0.003 ms to find the median. Moreover, the possibility of error is much larger. If you look at plot 2, it is clearly seen that the third algorithm takes way more to function than it normally should. However, an error this large happening with an array with 1.000.000 integers is very unlikely.

After the size of the array surpasses 1000 integers, the results of different algorithms start to really differ. Up until this point, the results were similar however, after this as the size increases the difference between the running time of the first algorithm and the other two rise. For example, to find the median of an unsorted array of 500000 integers the first algorithm takes 432525 ms and the third one only takes 35.639 ms. This means it takes more than 1000 times the third one.

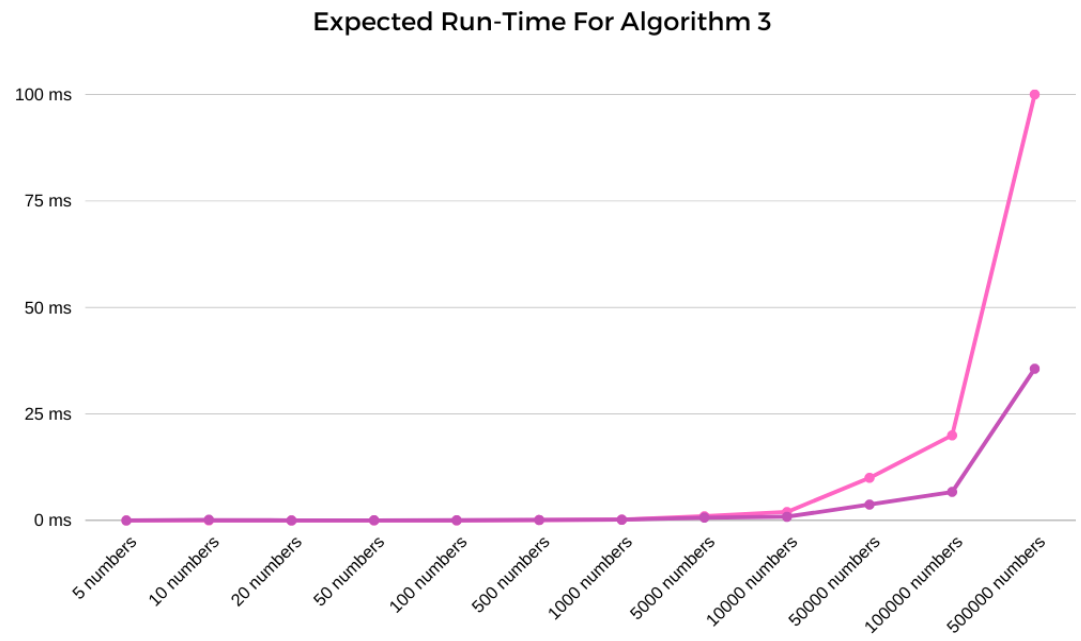
To sum up, from these two plots and the table we can understand that between all three algorithms, the first one which has a time-complexity of $O(n^2)$ takes the most time and the third one which has a time-complexity of $O(n)$ takes the least time. This difference becomes very extreme as the size of the array gets close to 500000 that when putting all of the data on the same plot, the running times of algorithm 2 and 3 are not even clearly seen. This means that having a time-complexity of $O(n^2)$ may not cause many problems for smaller operations but when the process becomes more complex, it becomes harder to operate. Thus, using algorithms that have a time-complexity of $O(n)$ or even $O(n*\log(n))$ is more efficient and intelligent.


Expected Run-Time For Algorithm 1
600,000 ms
400,000 ms
200,000 ms
0 ms
5 numbers
10 numbers
20 numbers
50 numbers
100 numbers
500 numbers
1000 numbers
5000 numbers
10000 numbers
50000 numbers
100000 numbers
500000 numbers

Plot 3

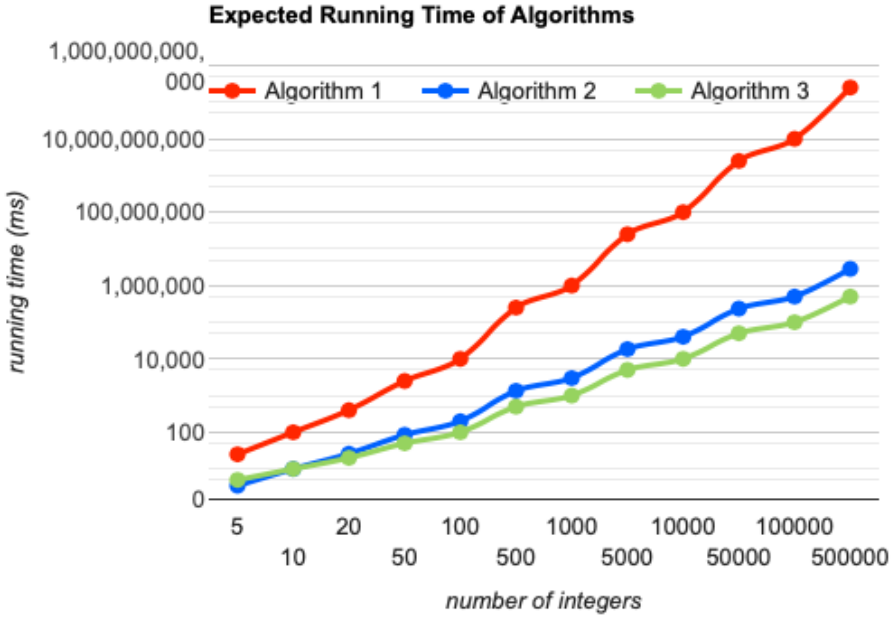

Expected Run-Time For Algorithm 2
125 ms
100 ms
75 ms
50 ms
25 ms
0 ms
5 numbers
10 numbers
20 numbers
50 numbers
100 numbers
500 numbers
1000 numbers
5000 numbers
10000 numbers
50000 numbers
100000 numbers
500000 numbers

Plot 4



Plot 5

These three plots show the expected and the real run times. The light pink lines show the expected run time and the dark pink-purple lines show the real run times. These plots help us observe the differences between what was expected and the actual results. In each case the difference is very very little, almost unnoticeable until 10000 numbers. After this point for all of the algorithms, the real run times are smaller than the expected ones. For the plot 3, the biggest difference is for the array of 500.000 integers and other than this, the graph is very similar to the expected. In plot 4, again the expected and the real results are not very far apart. However, in plot 5 the difference is increased and the gap, especially after 5000 integers, is higher than the ones in plots 3 and 4. Overall, all of these three plots show that with minor differences the results weren't very far away from the expected and the running time of all 3 algorithms fit with the mathematical and theoretical values.



Plot 6

In this plot, the x-axis shows the number of integers of the array and the y-axis shows the time it takes to actualize the action. Again like plot 2, the values in the y-axis increase logarithmically. The aim of this plot is to observe the time expectancy of all three algorithms together, as it can be seen from the plot, the most efficient, that means the algorithm that takes less time to do the same operation is algorithm 3.

The specification of the computer I used:

The code was run on MacBook Air (Retina, 13-inch, 2020) with 1.1GHz dual-core Intel Core i3-based MacBook Air systems with 8GB of RAM and 256GB SSD. Operating system is Macos.