Roya Salei

December 1, 2020

CS 3310

Project 2

Data:

In this project we aimed to compare the time complexity of finding the kth smallest value in an array using 4 different partitioning algorithms. Different array sizes n = { 10, 50, 100, 250, 500, 1000} and 5 unique pivot points p ={ 1, n/4, n/2, 3n/4, n} were used:

1. Select-Kth1 Merge sort
2. Select-Kth2 Quick sort --iterative mode
3. Select-Kth3 Quick sort --recursive mode
4. Select-Kth4 Median of medians

To create arrays with distinct values, I used two different loops one starting from 0 and filling the odd indices (incrementing by two) and the other loop started from twice the size of array and filled the even indices (decrementing by two). This method allowed me to make sure that all values are unsorted and are distinct.

Then, I ran the program for 1000 times for every single algorithm / per size / per pivot points.

Test Strategies:

Below is the average result for each algorithm per array size. While performing this experiment I noticed a wired behavior in my computer which I could not figure it out. At the beginning I set to save all the times in a csv file, however I noticed that some number do not get recorded, even though all of them were printed on the console but the fact that some time records were randomly missing on the csv file makes me think that there must an underlying condition that

creates such an strange behavior. Also, noted that as the array size grew in some iterations I got what felt like a random number. Regardless, the following is the result of up to1000 iteration per algorithm and it shows that the first algorithm (select-kth1) is slower than the other 3 algorithms.

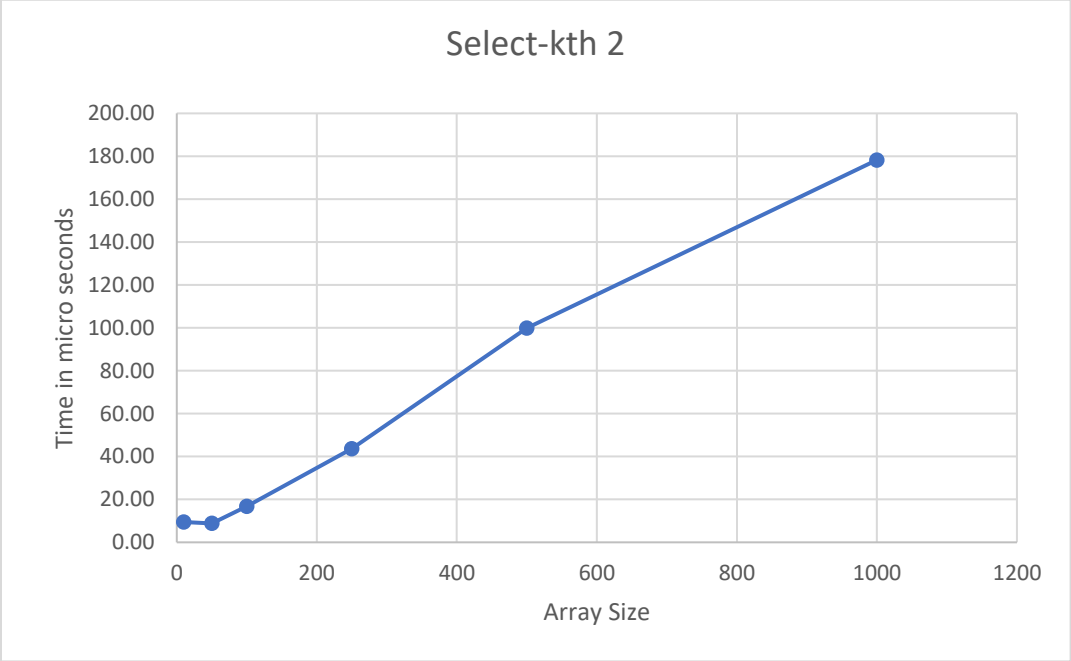
Table 1 array size versus performance time (micro second) for 4 algorithms

Array Size	Select-kth 1	Select-kth 2	Select-kth 3	Select-kth 4
10	15.93	9.48	9.74	25.12
50	20.29	8.82	9.70	45.38
100	25.03	16.77	26.47	90.27
250	59.28	43.63	88.86	111.20
500	119.60	99.90	155.52	140.74
1000	437.98	178.27	189.83	133.72

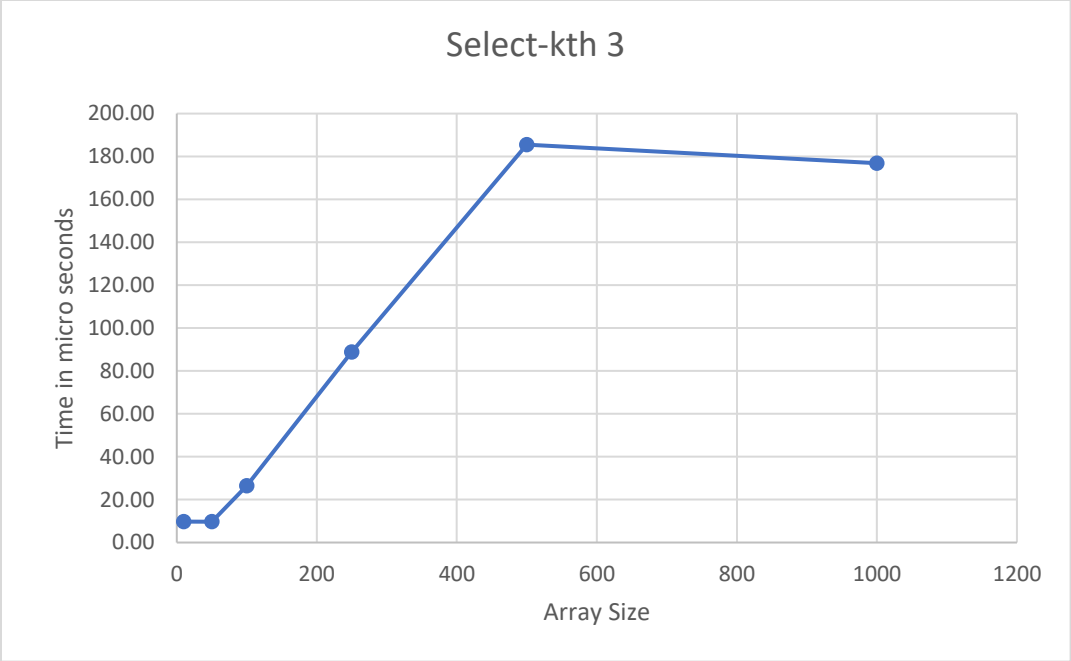
2nd algorithm vs the 3rd algorithm

Both of these algorithms use a similar approach to partition and return the value of the kth smallest element in the array with not much noticeable differences, even as the size increases the performance was not that different. Perhaps, in much larger array sizes we could see that the current minimal gap increases, especially when the memory and space is an issue (which is the case of my computer. I do have a windows laptop and My hard drive is almost 80% full and even though I do have 8G memory so many startup processes consume the memory (some times up to 70%) before even starting visual studio for my programming.) Having said that, still within the acquired date, the relationship difference is noticeable as the size of array increases. However, in small array sizes, the difference is negligible. Another issue which might have hampered the performance of the 3rd algorithm compare to the 2nd one could be related the recursion and use of stack in the memory. From the theoretical stand point both algorithms have time complexity of O(n) when the kth smallest element is on the pivot position and the worst complexity O(n^2) happens when the kth smallest element is on the either one of the far ends of the array. Below are

individual graphs showing the relationship between the array size and the algorithm performance (measured in microseconds).



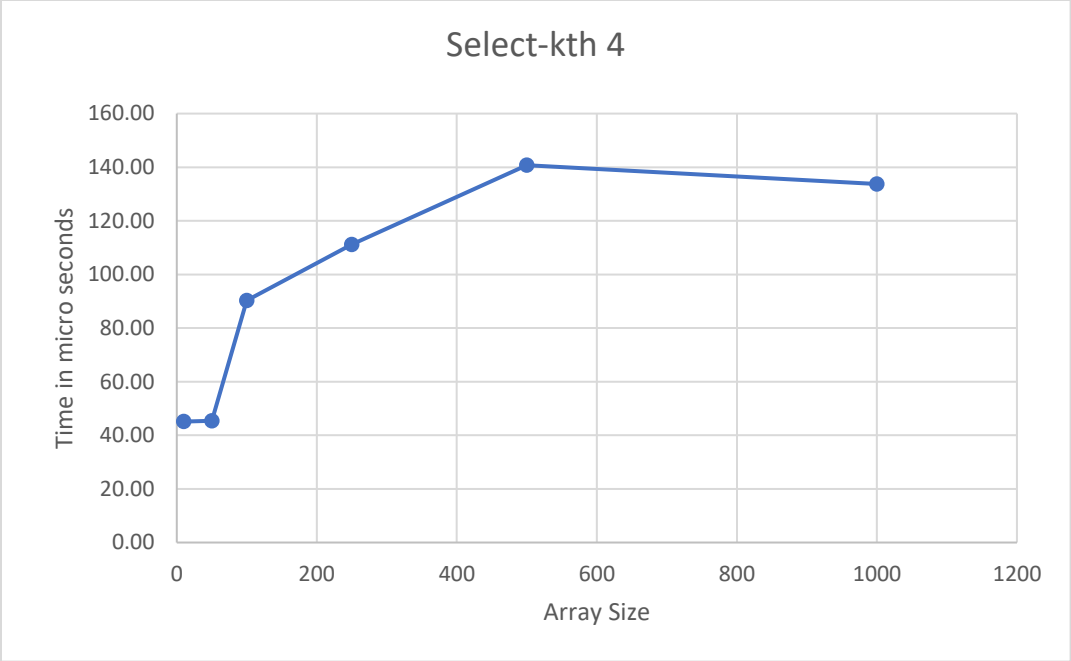
Graph 1 shows relationship between size of array and performance (in microsecond)



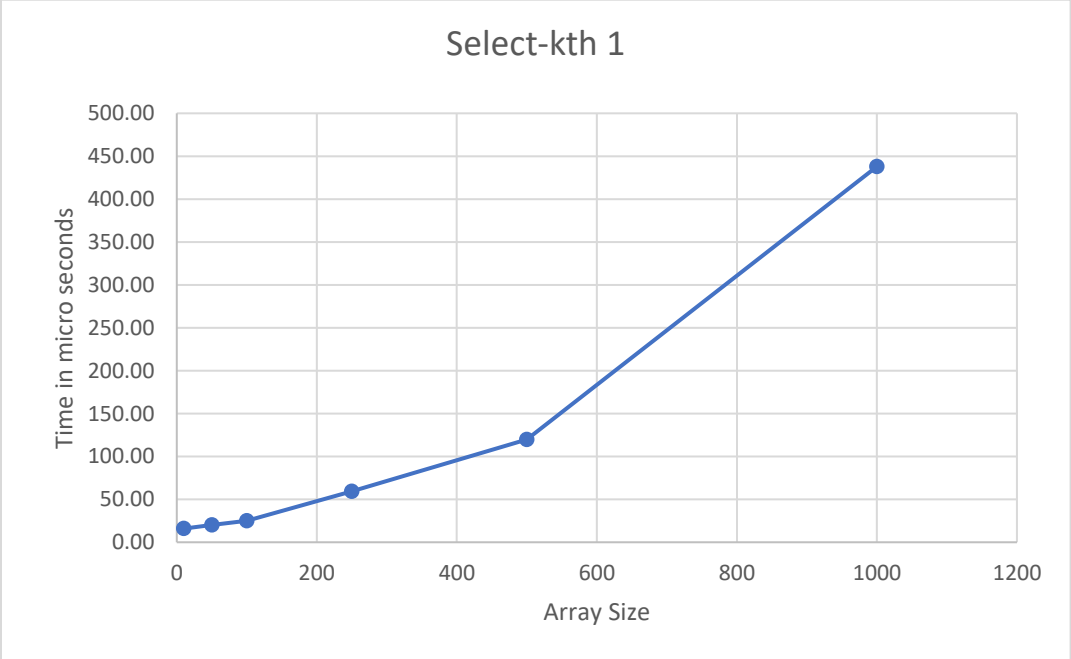
Graph 2shows relationship between size of array and performance (in microsecond)

1st vs 4th Algorithm

Overall, we can say that the 4th algorithm (using median of medians to find the pivot) is faster especially when the n was between 50 and 250 and then there was a bump in the data which shows the mm algorithm (4th algorithm) was slower than the merge sort algorithm (which happened at 500 array size) but again at 1000 array size we notice that 4th algorithm is faster than 1st algorithm which uses merge sort. Again, this could be related to position of pivot. However, merge sort, regardless of the dividing positions, its performance time is consistent with its theoretical time complexity of O(nlogn). Below are two graphs showing the result of this experiment.



Graph 3shows relationship between size of array and performance (in microsecond)



Graph 4shows relationship between size of array and performance (in microsecond)

Strength and Constraints:

This experiment shows that it is possible to quantify our algorithm performance and efficiency using real time data which also consolidates our theoretical knowledge about program efficiency and speed.

My major constraint in this project is my computer memory, and CPU which most of the time the background processes consume majority of those resources. For that matter, I tried to end some processes that may affect the correctness of the program also, closing memory-intensive applications such as browser, Microsoft application and even startup application such a mail and discord, and ran several tests to hoping to get a more accurate result.

Another, constraint was number of array sizes. At the beginning I noticed a wide range of deviations in the results and as I got closed to 1000 trials the result looked more cohesive, however I believe increasing number of array sizes and also increasing number of trials could further explain the differences and reduce the output deviations. Even though, I performed 1000 trials per algorithm, but it seems due to closeness of result especially for algorithms 2, 3 and 4 perhaps a much larger number of trials could demonstrate a better picture.)