Name: Alexa Tang__________

(Use this page as the cover sheet when you turn in your project report)

Project #2
CS3310 Design and Analysis of Algorithms

1. (40 points)	Date Sets, Test Strategies & Results	
2. (10 points)	Theoretical Complexity Comparisons	
3. (10 points)	Select 2 Versus Select 3	
4. (10 points)	Select 4 Versus Select 1	
5. (15 points)	Strength and Constraints of Your Work (at least 150 words)	
6. (15 points)	Program Correctness	
(100 points)	**Total**	

1:	40 points
2:	10 points
3:	10 points
4:	10 points
5:	15 points
6:	15 points

Total: 100 points

$$\frac{\quad}{100}$$

Alexa Tang

1 December 2020

CS 3310 – 03

Project 2

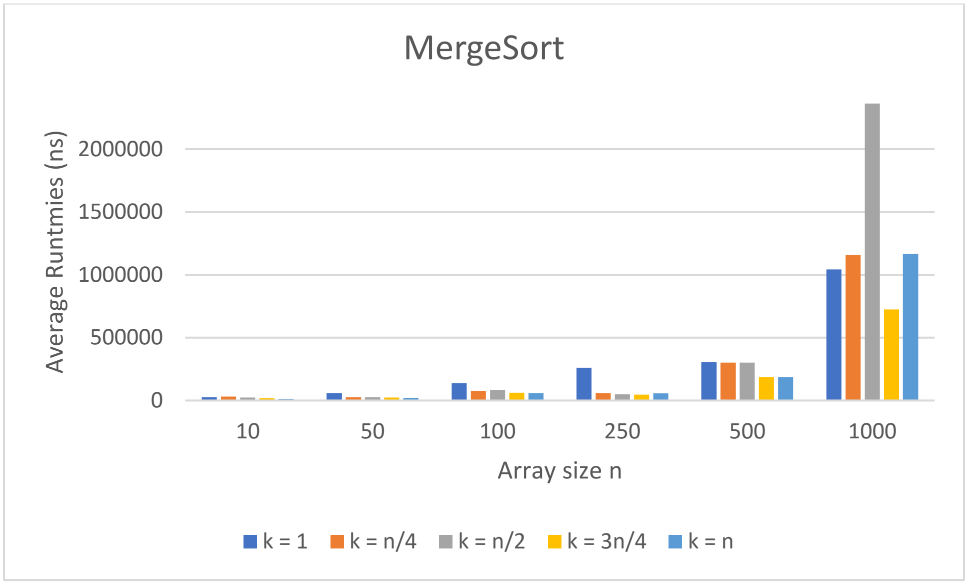
Selection Problem analysis

In order to compare the sorting algorithms discussed in class, I first generated arrays of size 10, 50, 100, 250, 500, and 1000, filling them in with random numbers ranging from 0 to 100. I arbitrarily chose this range of numbers since it would not affect the complexity of the algorithms. I only collected data for 6 arrays, since the way I collected my data took more than an hour for each algorithm. Each of the arrays were sorted by the algorithms 200 times for each k when k = 1, n/4, n/2, 3n/4, and n. The total runtimes for each algorithm per k value were then divided by 200 to get the time taken to find the kth smallest value. These were run without any other programs running in the background to allow for the greatest amount of resources available for its execution.

Select-kth 1 was implemented using the MergeSort algorithm which takes in a list of size n, splits the list in half, then does 2 recursive calls to sort the first or left half of the array then sort the second or right side of the array. The two sorted halves are then merged. The execution of this implementation produced the following runtimes (in nanoseconds) for each kth value per array size n:

	Select-kth 1, MergeSort					
	n = 10	***n = 50***	***n = 100***	***n = 250***	***n = 500***	***n = 1000***
k = 1	25467 ns	59090 ns	139511 ns	260077 ns	307668 ns	1043996.5 ns
k = n/4	32453 ns	26084 ns	77015 ns	58661 ns	301437.5 ns	1157359 ns
k = n/2	24205.5 ns	27509.5 ns	83709 ns	50353.5 ns	301437.5 ns	2362507.5 ns
k = 3n/4	17709.5 ns	24133 ns	63216 ns	47252.5 ns	187585.5 ns	723658 ns
k = n	13038.5 ns	22252.5 ns	60476.5 ns	55728.5 ns	187585.5 ns	1166599.5 ns

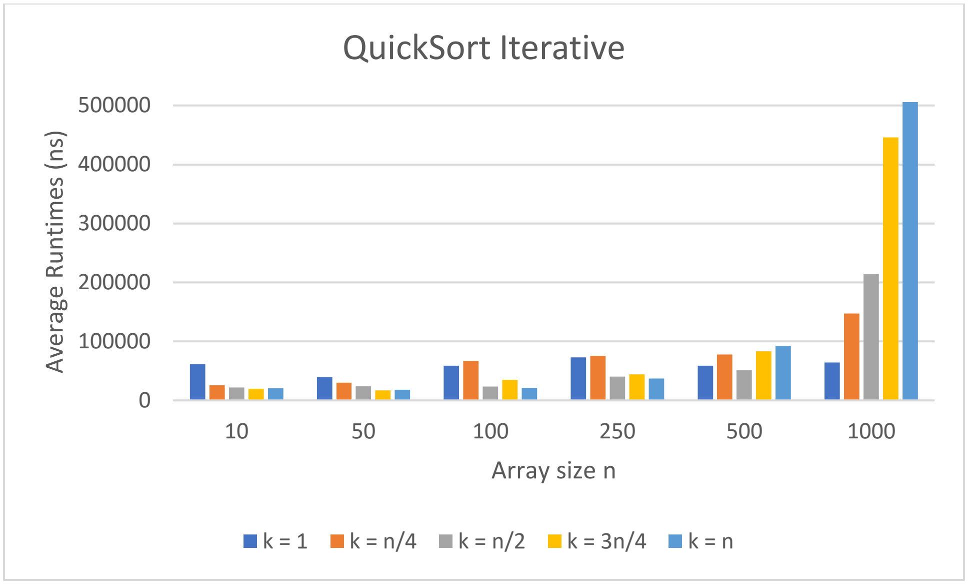
Comparing the times to sort each array of length n:



For MergeSort, in theory the runtimes for each array of size n, should not vary since, regardless of the kth value we are looking for, the entire array will be sorted before this value is found. In the data I was able to collect I was able to notice this trend except for a few outliers. For the most part, the times to sort each array of length n did not vary drastically.

Select-kth 2 was implemented using the partition procedure of the QuickSort algorithm iteratively. It takes a list, its length and the kth value being searched for. While the pivot position does not correspond to the kth smallest value in the list, the partition procedure is called. The execution of this implementation produced the following runtimes (in nanoseconds) for each kth value per array size n:

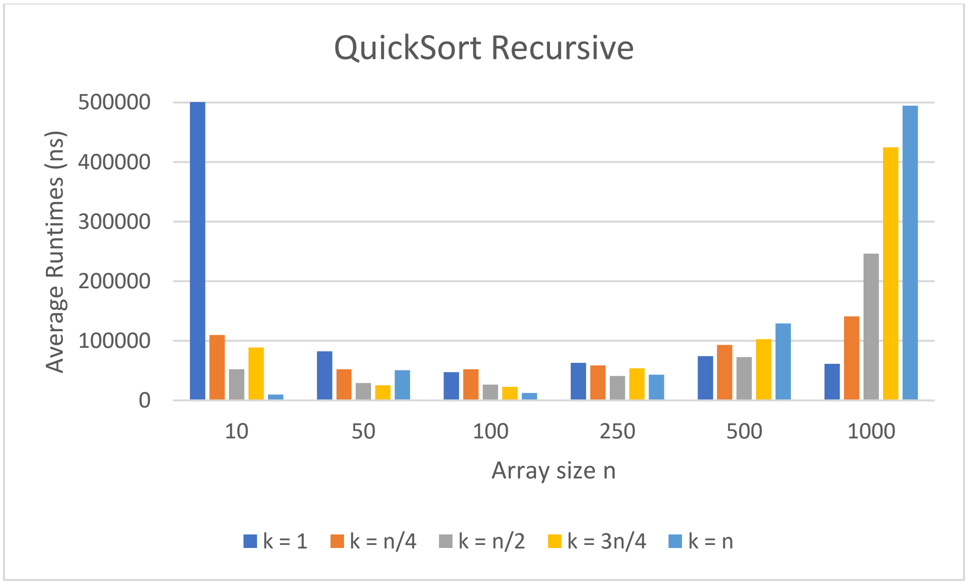
	Select-kth 2, Iterative QuickSort					
	n = 10	***n = 50***	***n = 100***	***n = 250***	***n = 500***	***n = 1000***
k = 1	61652 ns	39745.5 ns	58792.5 ns	72838 ns	58805.995 ns	64264 ns
k = n/4	25923.5 ns	30082.5 ns	67010.5 ns	75634.995 ns	77652.5 ns	147563 ns
k = n/2	21686.005 ns	24033.995 ns	23635 ns	40438.5 ns	51295.505 ns	214480.5 ns
k = 3n/4	19548.5 ns	17059.995 ns	34787.5 ns	44355 ns	83054.5 ns	445856.495 ns
k = n	20776 ns	18279 ns	21108 ns	37150.5 ns	92392 ns	520767.5 ns



In the case of the iterative QuickSort implementation, the partition procedure is called until the pivot position corresponds to the kth value, and the array will not always be completely sorted like in the MergeSort algorithm, hence there is a more significant variance in runtimes across all kth values. For example, in the way the partition procedure is implemented, the pivot is the first element. Since the partition procedure will sort the pivot in the correct position, if this happens to be the kth value, the sorting stops and the value at the pivot position is returned.

Select-kth 3 also uses the partition procedure of the QuickSort algorithm recursively. It takes the list, its left and right indexes, and kth value being searched for. If there is more than one element in the array, and if the pivot returned by the partition procedure does not correspond to the kth value being searched for, the quickSortRecursive function is called to sort both sides of the list. The execution of this implementation produced the following runtimes (in nanoseconds) for each kth value per array size n:

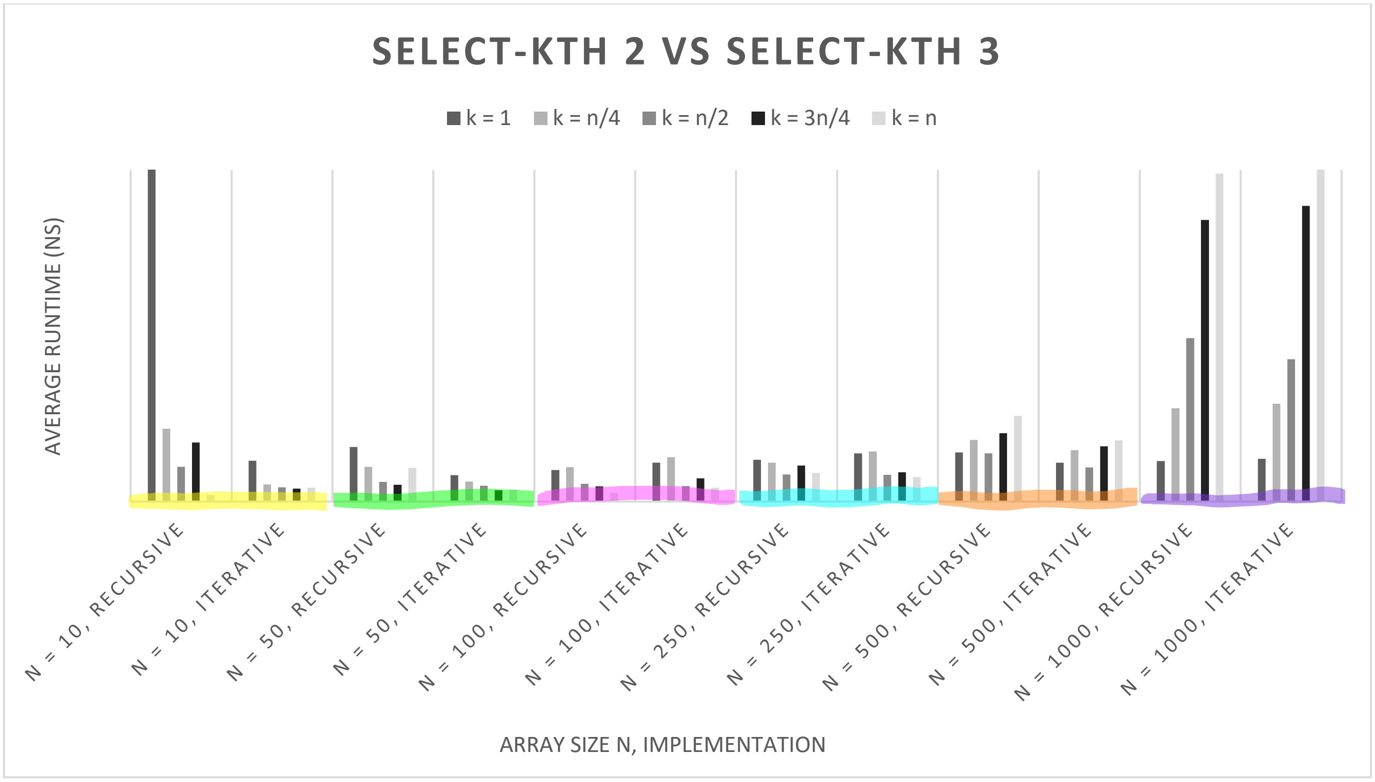
	Select-kth 3, Recursive QuickSort					
	n = 10	***n = 50***	***n = 100***	***n = 250***	***n = 500***	***n = 1000***
k = 1	4698351 ns	4698351 ns	47643.5 ns	63094 ns	74169 ns	61278.5 ns
k = n/4	109912.5 ns	109912.5 ns	52110 ns	58808.5 ns	92997.5 ns	140750 ns
k = n/2	52193 ns	52193 ns	26735 ns	40886 ns	72844.5 ns	246417.5 ns
k = 3n/4	88909 ns	88909 ns	22807.5 ns	54125.5 ns	102953 ns	424633 ns
k = n	10102 ns	10102 ns	12679.5 ns	43141 ns	129269 ns	494864.5 ns



Similar to the iterative implementation using the partition procedure, the recursive procedure keeps partitioning the array until the pivot is at the kth slot.

Select-kth 4 was supposed to be implemented using the partition procedure from QuickSort, but before we first need to divide the list into subsets of size r and find the median values of each subset. The medians from each subset are then put into a separate set and the median of the medians is found and used at the pivot to then sort the array.

Comparing the runtimes for all k values per array size between Select-kth 2 and 3:



According to the data I was able to collect, when n = 100, Select-kth 3 or the recursive implementation was consistently faster than Select-kth 2 or the iterative implementation for all values of k. Select-kth 3 was also faster than for most k values but not all when n = 100 and 250.

The time complexity of the MergeSort algorithm is T(n) = $O(n \log n)$. Since MergeSort does not pay attention to the original order of the list when sorting, it will continue to divide the list in two halves until the sub-lists contain 1 element and then will begin to merge the list which takes $O(n)$ time.

For the QuickSort algorithm, its time complexity is the same as MergeSort in average cases. However, its best case is $O(n)$ if the pivot is the kth smallest since in this case the partition procedure will only happen once. Its worse case is $O(n^2)$, where the partition procedure is called $O(n)$ times and each time it executes in $O(n)$ time. This would occur if the pivot is always the greatest or smallest element.

The time complexity of QuickSort implemented to use the Median of Medians as the pivot is T(n) = $O(n)$. In using the median of medians as our pivot, in worst case we are left with 3/4th of the list to find the kth value.

In the data for the Select-kth 1 or the MergeSort algorithm, there was slight variation in the runtimes in finding each k despite the implementation implies that the runtimes for each list size should be the same since regardless of the kth value we are looking for, the list will always be completely sorted by the time the kth value is returned. Additionally, although the instructions stated that to obtain the time taken to solve a given instance the total runtime was to be divided by the number of times the algorithms were tested. However, I found that the runtimes for each time an algorithm was executed were different. In collecting the runtimes for each time each of the algorithms were executed and averaging these times would likely give me a more accurate overall average runtime value. In addition, the trends between the Select-kth 2 and 3 would have likely been more prominent if I collected runtimes for a greater range of array sizes.