Adithep Narula
Section 4

Environment
Processor: Intel Core i5-4300U CPU 1.90Ghz 2.50Ghz
RAM: 8.00 GB
System type: 64 -bit Operating System, x64-based processor

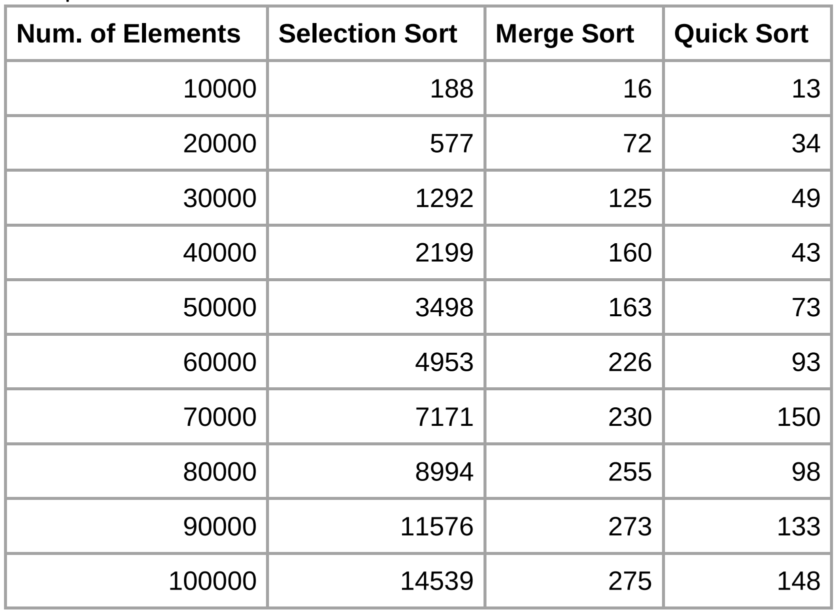
Controlled variables
Did not set random seed. Integer[] as array type. Programs currently running on my laptop: Onenote, Ecllipse, Chrome (with 10 tabs).

Methods
For a fixed number of elements, I run each sorting method 3 times and average the result. This is because execution time can be affected by what the computer is doing in the background, such as when Java run-time engine is performing garbage collection (cite Book, pg 43). Therefore, averaging the result help reduce anomalies, which makes my data more accurate and reliable.

No random seed was set for Random() , and array type used is Integer[]. The results in nanoseconds are converted to milliseconds by dividing 1,000,000. The outcomes of the three runs for each sorting method including the average time is recorded in the file "threeruns". The average result is attached as an image in this document.

Result Table

Table1. Average result for specified number of elements in milliseconds for selection sort, merge sort and quick sort.



Num. of Elements	Selection Sort	Merge Sort	Quick Sort
10000	188	16	13
20000	577	72	34
30000	1292	125	49
40000	2199	160	43
50000	3498	163	73
60000	4953	226	93
70000	7171	230	150
80000	8994	255	98
90000	11576	273	133
100000	14539	275	148

Table2. Average result for specified number of elements in milliseconds for selection sort, merge sort and quick sort.

Num. of Elements	Merge Sort	Quick Sort
50000	250	96
100000	327	239
150000	384	318
200000	425	303
250000	470	311
300000	574	308
350000	596	350
400000	612	321
450000	692	400
500000	699	437
550000	793	405
600000	775	485
650000	835	483
700000	867	490
750000	902	535
800000	1009	530
850000	1027	564
900000	1088	618
950000	1188	625
1000000	1098	637

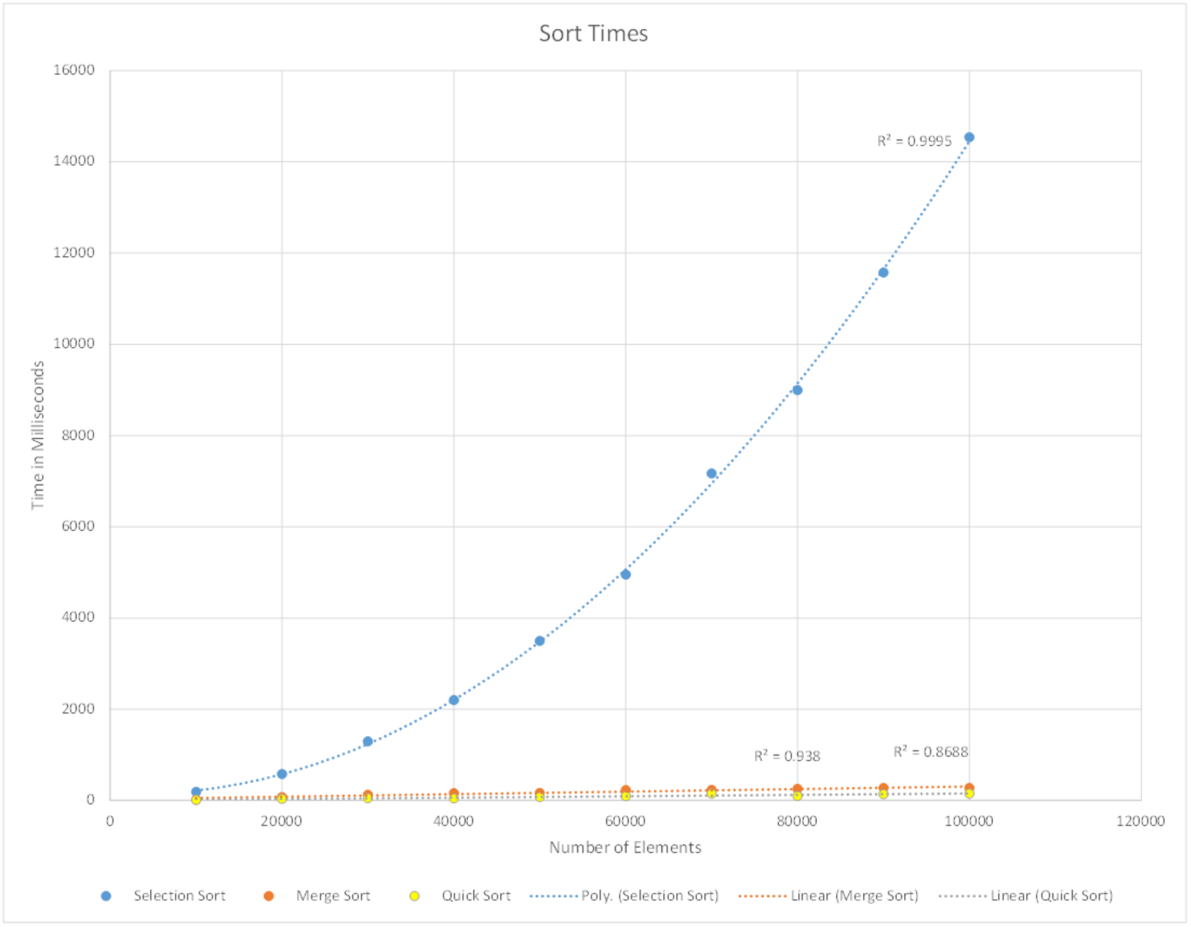
Version 1: SelectionSort_V1, MergeSort_V1, QuickSort_V1
Version 1 for the three sorting algorithms follow the general outline of the pseudocode written my professor Joanna. For all three sorting algorithms, the change in time as a function of number of elements were as expected. When the number of elements are small, in the order of 10000 - 20000 elements, the difference in the sorting time for all three algorithms, especially between selection sort and the other two, were relatively small. However, as the number of elements increase, the sorting time for selection sort grew dramatically relative to mergesort and quicksort, which can be seen in Table3. The large difference is due to the rapid increase in time for selection sort. This is because the time complexity for selection sort is $O(n^2)$, so the time required to sort grows quadratically. However, merge sort and quicksort (average case) are $O(n\log(n))$, and thus grows much slower since they grow almost linearly, because log(n) is nearly constant for **large** n. Their linear-like growth can be seen from the linear trend line plotted and R^2 value in graph1 and graph2. The R^2 values for merge sort are 0.938 and 0.985, and quicksort are 0.868 and 0.904, in graph1 and graph2, respectively. As the number of

elements grow larger, the R^2 values get closer to 1, roughly indicating that the linear trend line - ie. linear growth - is a good approximation.

Table3. Time in milliseconds for each sorting algorithms for 10,000 and 20,000 elements.

Num. of Elements	Selection Sort	Merge Sort	Quick Sort
10000	188	16	13
20000	577	72	34
90000	11576	275	133
100000	14539	273	148

Graph1. The sorting time of selection sort, merge sort and quick sort. The numbers plotted are from table 1.

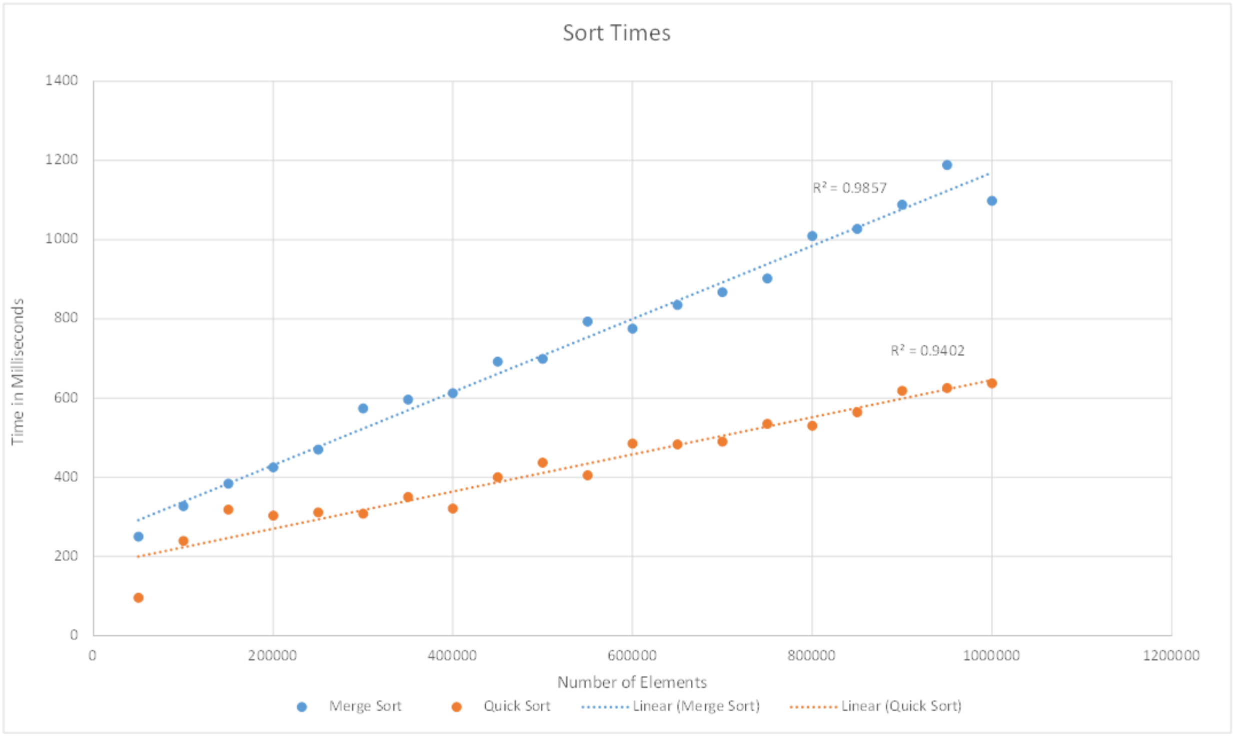


The sorting for 10,000 to 100,000 elements highlight how much slower selection sort is relative to merge sort and quicksort because of its quadratic growth rate in time as a function of number of elements. On the other hand, the second section (50,000 to 1,000,000 elements) is separated to underline the differences and similarities between merge sort and quicksort. Graph 2 clearly shows that, on average, merge sort is slower than quicksort. Out of the 60 runs conducted for in section 2, quicksort came out slower only once, when the number of elements was 15,000 as illustrated in table 4. This may be because we got a "bad" data set for quicksort like lots of repeated numbers, or it could also be due to some other factors, like java garbage collection or other triggering factors running in the background of my computer. Nonetheless, I considered this an anomaly and it barely raised the average, so therefore mergesort still came out slower. For other runs, merge sort was always slower than quicksort and this was probably due to the temporary array that is created (which has a cost of O(n) - creating and setting each position to null) and accessed numerous times during the merge stage. Quicksort does not create a temporary array, and thus does have any cost related to this. In addition to higher sorting time for merge sort, the rate of growth of the sorting time also increases faster than that of quicksort, which can be seen on graph2 (the orange line is less slanted than the blue line).

Table4. Sorting time for merge sort and quick sort from the three runs and their average for 15,000 elements. Use to show how quicksort came out slower once for run#1.

Run #	**Merge Sort**	**Quick Sort**
1	398	453
2	418	270
3	336	230
Average	384	318

Graph2. The sorting time of merge sort and quick sort. The numbers plotted are from table 2.



Version 2: QuickSort_V1 and QuickSort_V2.1

QuickSort_V1 always picks its pivot as the mid element and swaps the mid element to last position before sorting. In QuickSort_V2.1, I always picked the last element as the pivot and discarded the initial swapping statements from V1 (in V1, I picked the pivot as mid element and swapped to the last element). This reduces the overall cost of method, but since these swapping statements are O(1), I hypothesize that erasing them will not really affect the execution time. I ran QuickSort_V1 and QuickSort_V2.1 ten times each with 50000, 500000, and 1000000 elements. For 50,000 elements, the sorting time for QuickSort_V1 and QuickSort_V2.1 was very similar -- sometimes V1 was faster by a couple of seconds and other times V2.1 was faster. However, I found it very surprising that for 500,000 elements, approximately 8 out of 10 times, V2.1 was faster than V1. At first, I thought V1 was probably slower because it was called first and this may have something to do with java bringing the code or array into memory, consuming up a little bit of extra time. However, even when I called V2 first, the result was similar - V2.1 was 7/10 times faster than V1. For 1 million elements, V2.1 was also usually faster than V1. Even though V2.1 was generally faster than V1, it is only faster by a couple of milliseconds. Therefore, discarding the mid calculation and swapping statements in the beginning may after all increase the performance, but by very few milliseconds at best.

Version 3: QuickSort_V1, QuickSort V2.2, and SelectionSort V1

Unlike version 2 where I tried to optimize the method, I created version 3 to investigate the worse-case scenario of QuickSort. In class we discussed how the quick sort's worse-case produces O(n^2), which is the same big-O as selection sort. However, we never discussed whether the worse-case of quick sort or selection sort is slower, which is something I hope to find out in this version. Furthermore, I also want to find out by how much faster QuickSort_V1 is when compared to its worse-case. I used QuickSortV2.1 to test for worse-case, where the first element is chosen as the pivot and the QuickSort method is called on a sorted array. For the sake that we are investing in Version 3, we will call QuickSort V2.1, QuickSortV2.2. The results of various runs are noted in table 5, table 6, and table 7, respectively. All three tables below show that when QuickSort is in its worse-case scenario, it is slower than SelectionSort. Furthermore, when the number of elements are around 20,000, QuickSort_V2.2 causes stack to overflow most of the time. When the number of elements are 50,000, QuickSort_V2.2 always causes always causes stack over flow. I never thought this would occur, but when analyzing the method properly, over 20,000 and 50,000 recursive calls were made for 20,000 and 50,000 elements, respectively, and this could easily cause the stack to overflow. To answer the second question I have, QuickSort_V1 is around 10-15 times faster for 10,000 elements, and 30-40 times faster for 20,000 elements -- that is, if the stack is not overflowed!

Table5. Run time for the three methods with 10000 elements.

Run #	QuickSort_V1	QuickSort_V2.2	SelectionSort
1	10	156	149
2	9	168	149
3	9	160	152

Table6. Run time for the three methods with 20000 elements.

Run #	QuickSort_V1	QuickSort_V2.2	SelectionSort

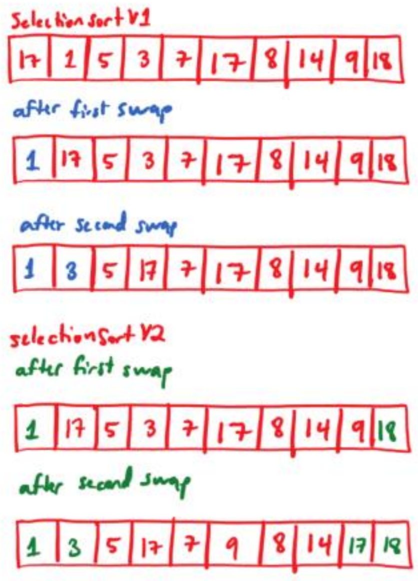
1	13	465	415
2	17	STACKOVERFLOW	588
3	15	STACKOVERFLOW	389

Table7. Run time for the three methods with 50000 elements.

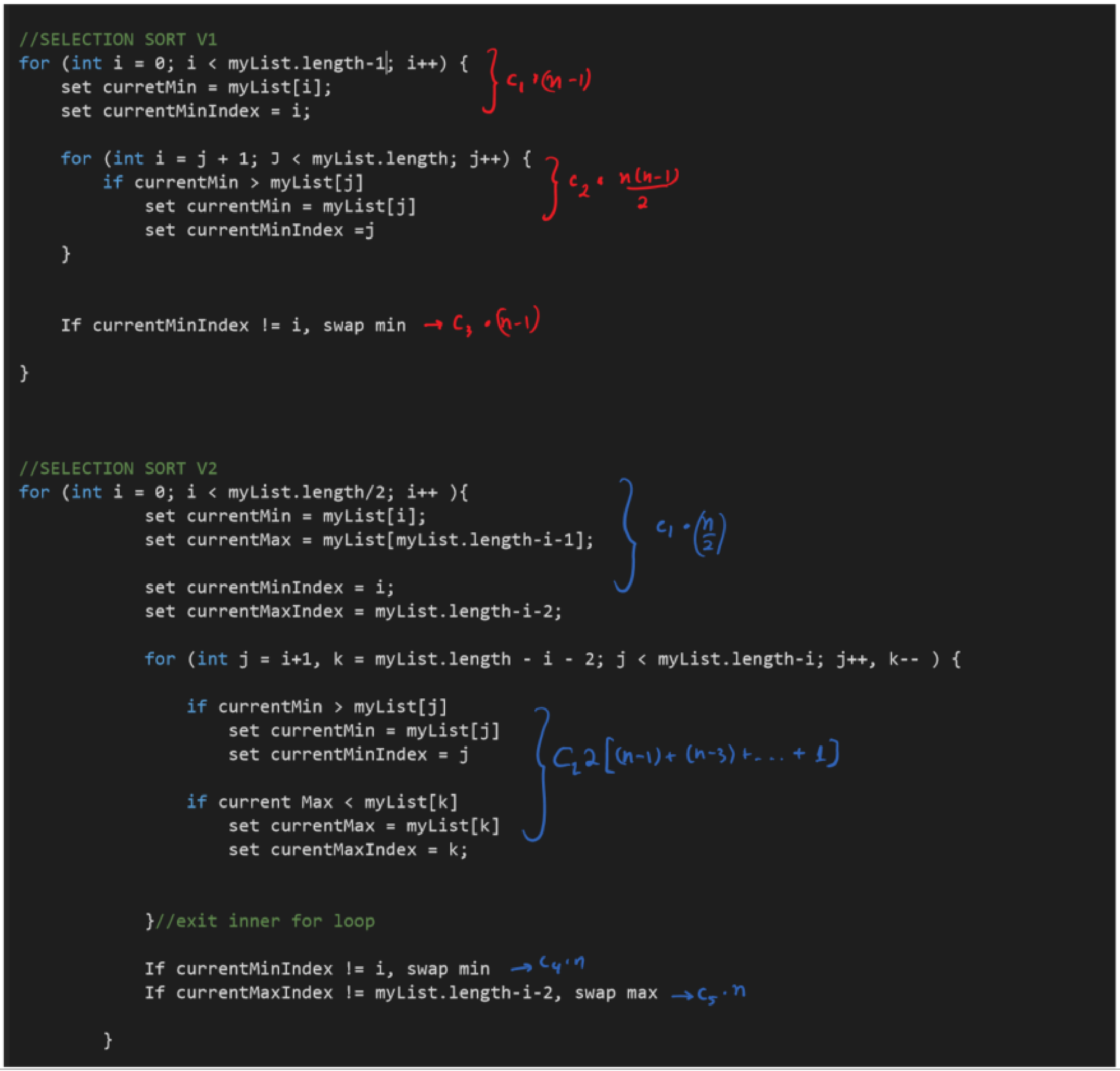
Run #	QuickSort_V1	QuickSort_V2.2	SelectionSort
1	58	STACKOVERFLOW	2701
2	109	STACKOVERFLOW	2926
3	90	STACKOVERFLOW	2770

Version 4: SelectionSort_V1, SelectionSort_V2

In SelectionSort_V2, I essentially combined descending selection sort and ascending selection sort together into one selection sort. Thus instead of searching for only current min each time outer loop iterates, I also searched for current max. Therefore, after the inner loop finish executing, I would find both min and max, and do swaps for both of them. The picture below shows the first and second swaps of SelectionSort_V1 and SelectionSort_V2.



At first glance, I thought that this would be more efficient than regular selection sort (ie. SelectionSort_V1) because instead of having the outer loop iterates n-1 times, the outer loop now only iterates n/2 times because I am sorting two positions each time. However, after timing it, V2 is always slower than V1 by around 100-200 milliseconds for 50,000 elements and by 300-600 million seconds for 100,000 elements. So I decided to do a time complexity analysis based on my pseudocode below:



```
//SELECTION SORT V1
for (int i = 0; i < myList.length-1; i++) {          } c1·(n-1)
    set curretMin = myList[i];
    set currentMinIndex = i;

    for (int i = j + 1; J < myList.length; j++) {      } c2· n(n-1)/2
        if currentMin > myList[j]
            set currentMin = myList[j]
            set currentMinIndex =j
    }


    If currentMinIndex != i, swap min  → c3·(n-1)

}



//SELECTION SORT V2
for (int i = 0; i < myList.length/2; i++ ){                  } c1·(n/2)
            set currentMin = myList[i];
            set currentMax = myList[myList.length-i-1];

            set currentMinIndex = i;
            set currentMaxIndex = myList.length-i-2;

            for (int j = i+1, k = myList.length - i - 2; j < myList.length-i; j++, k-- ) {

                if currentMin > myList[j]
                    set currentMin = myList[j]                } C2 2[(n-1)+(n-3)+...+1]
                    set currentMinIndex = j

                if current Max < myList[k]
                    set currentMax = myList[k]
                    set curentMaxIndex = k;


            }//exit inner for loop

            If currentMinIndex != i, swap min  → c4·n
            If currentMaxIndex != myList.length-i-2, swap max  → c5·n

        }
```

From this analysis I realized that even though the outer loop for selection sort V2 executes half the number of times selection sort V1 does, selection sort V2 doubles the number of comparisons. Therefore, relatively speaking, the "half" and the "double" should cancel out to give the same sorting time as V1, so these two selection sort should take approximately the same time to sort. The reason that selection sort V2 is a little slower may be because it declares many more variables and tests for many more conditions (there is also another else statement for currentMaxIndex that I did not include in the pseudocode).

Version 5: MergeSort_V1, MergeSort_V2

In MergeSort_V2 I made a minor change in the in mergeSort method. I added the conditional statement to only execute the merge method only if the last element of first half of the array is bigger than first element of the second half of the array. If the condition evaluates false it means that both "portions" are already in the correction positions and merge method does not have to execute. The following is the result I got for running both merge methods on 300,000 elements.

Table8. Time in milliseconds that it took for V1 and V2 to sort for 30,000 elements.

MergeSort_V1	MergeSort_V2
1205	1028
1194	989
1133	925
967	886
1208	810
1101	821

MergeSort_V2 was always faster than MergeSort_V1. The merge method is the fundamental part of the algorithm and eliminating some calls to it have helped reduce the sorting time. Although, the calls that were eliminated were probably for the first few merge calls when there were few number of elements. It is highly unlikely that the merge method will be eliminated when there are lots of elements. The more sorted the elements we get from the random array generator, the higher the chance that our merge method will not be called in V2. Therefore, if given a list sorted list, the merge method in MergeSort_V2 will not be called at all. However, if given an array in a reverse order, our MergeSort_V2 will probably take a tiny nanosecond more time than V1 because the if statement will always have to be tested in addition to the merge method being called every time.

Conclusion
The fastest sorting algorithm in this project is QuickSort_V2.1, followed by QuickSort_V1, MergeSort_V2, MergeSort_V1, SelectionSortV1, and then SelectionSort_V2. Overall, the results seems fairly accurate since three or more runs were tested for specified number of elements. One way to improve time accuracy is to manually run the garbage collection, this will help decrease the likelihood of it running during my sorting algorithms. However, due to time constraint, I did not have to add on garbage collection and redo my results, therefore I did not include it in my program.