CPSC 319 ASSIGNMENT 1

John Ezekiel Juliano <john.juliano>

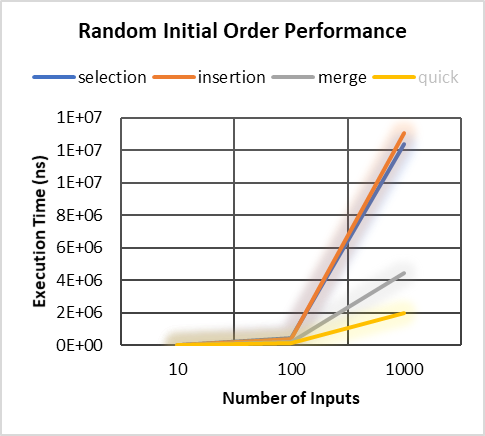
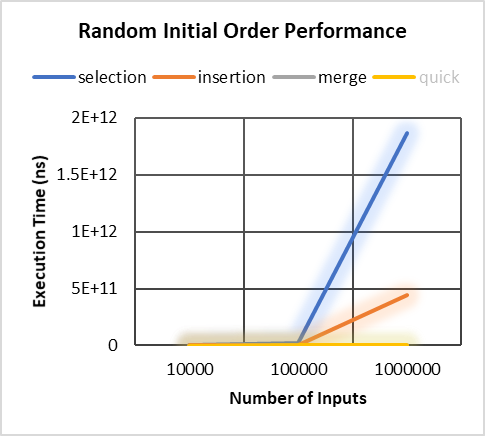
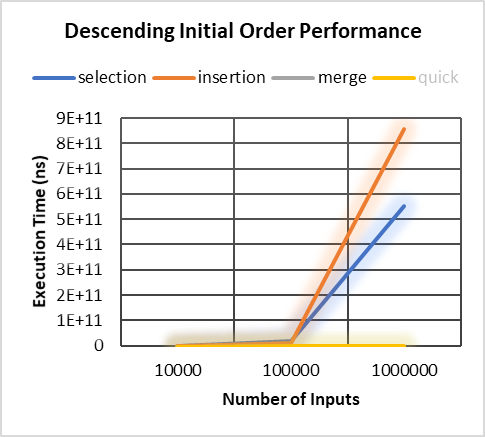
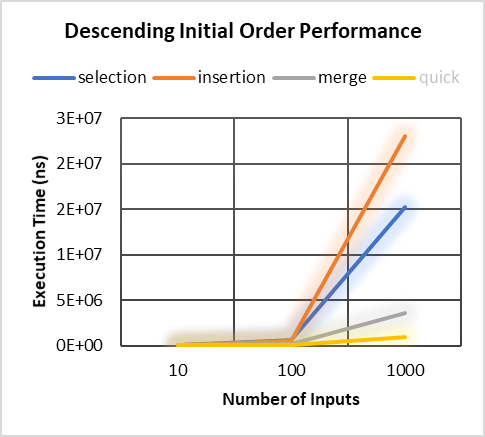
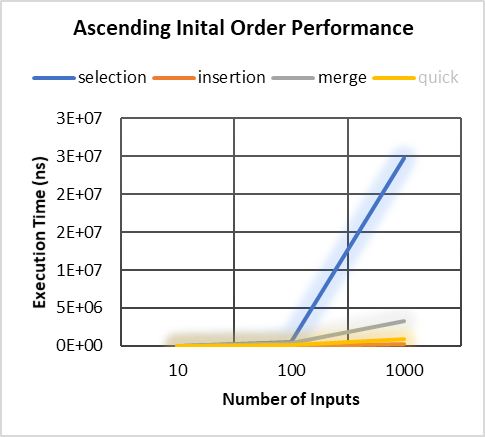
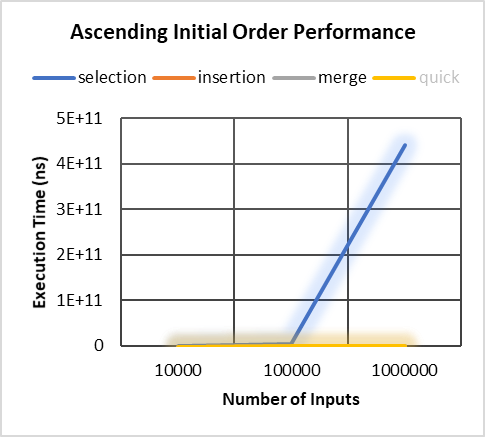
T05 <Rashid Mamunur>

**I. Experimental Methods**

There are various ways to sort an array or list with an arbitrary number of inputs. Some are faster and more efficient than others. There are also ones that need extra memory and ones that incorporates recursive method calls. The objective of this experiment is to compare the performance of four different sorting algorithms: selection, insertion, merge, and quick sort. Three different cases will be examined for each sorting algorithm an initial random, ascending, and descending array will be used to determine the average, best and worst cases (respectively) for each algorithm. A varying number of inputs will also be used to show the performance as inputs grow.

**II. Data**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Time (ns) | Selection Sort | | | Time (ns) | Insertion Sort | | |
| Random | Ascending | Descending | Random | Ascending | Descending |
| 10 | 1.97E+04 | 1.92E+04 | 1.92E+04 | 10 | 1.71E+04 | 1.28E+04 | 1.80E+04 |
| 100 | 4.43E+05 | 4.39E+05 | 6.64E+05 | 100 | 3.88E+05 | 3.46E+04 | 5.23E+05 |
| 1000 | 1.24E+07 | 2.48E+07 | 1.53E+07 | 1000 | 1.31E+07 | 2.31E+05 | 2.30E+07 |
| 10000 | 1.85E+08 | 6.54E+07 | 1.94E+08 | 10000 | 6.40E+07 | 3.05E+06 | 1.25E+08 |
| 100000 | 1.73E+10 | 4.64E+09 | 1.88E+10 | 100000 | 4.03E+09 | 1.95E+07 | 9.34E+09 |
| 1000000 | 1.87E+12 | 4.42E+11 | 5.52E+11 | 1000000 | 4.46E+11 | 2.97E+07 | 8.56E+11 |
| Time (ns) | Merge Sort | | | Time (ns) | Quick Sort | | |
| Random | Ascending | Descending | Random | Ascending | Descending |
| 10 | 5.05E+04 | 2.91E+04 | 3.08E+04 | 10 | 3.29E+04 | 2.05E+04 | 2.14E+04 |
| 100 | 2.21E+05 | 3.62E+05 | 1.94E+05 | 100 | 1.63E+05 | 7.31E+04 | 1.17E+05 |
| 1000 | 4.48E+06 | 3.30E+06 | 3.63E+06 | 1000 | 1.97E+06 | 8.70E+05 | 9.78E+05 |
| 10000 | 1.55E+07 | 1.17E+07 | 1.05E+07 | 10000 | 7.06E+06 | 6.21E+06 | 6.84E+06 |
| 100000 | 6.81E+07 | 7.24E+07 | 1.20E+08 | 100000 | 5.85E+07 | 6.75E+07 | 7.07E+07 |
| 1000000 | 3.46E+08 | 1.20E+08 | 2.18E+08 | 1000000 | 2.36E+08 | 1.08E+08 | 2.40E+08 |

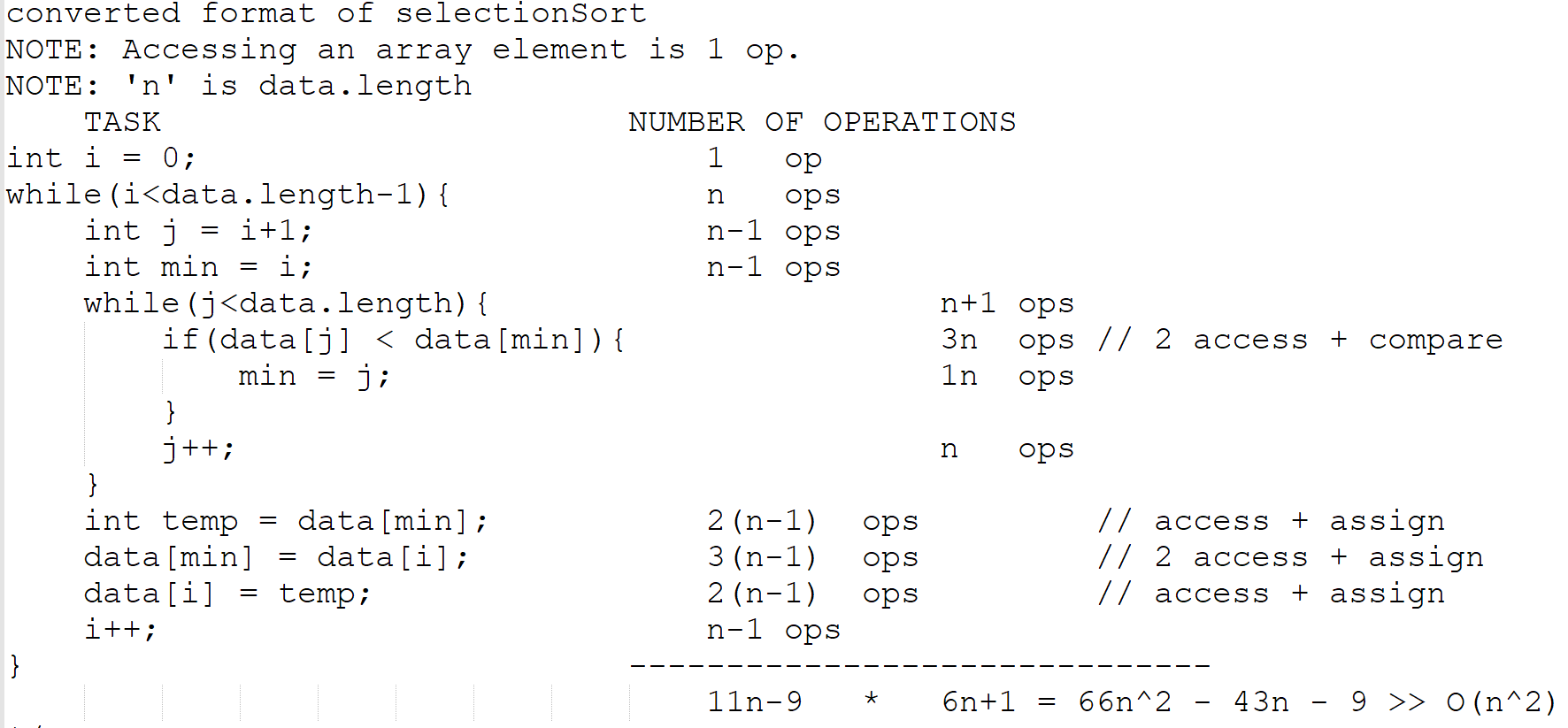


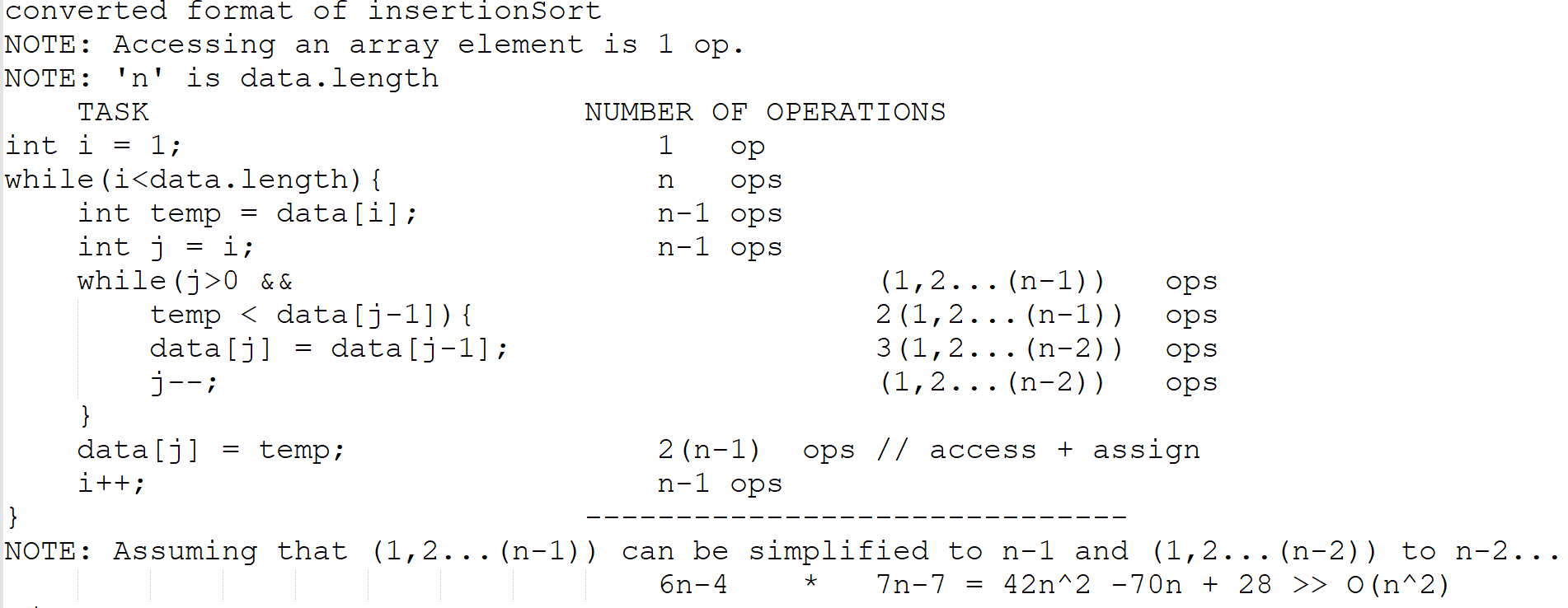
**III. Data Analysis**

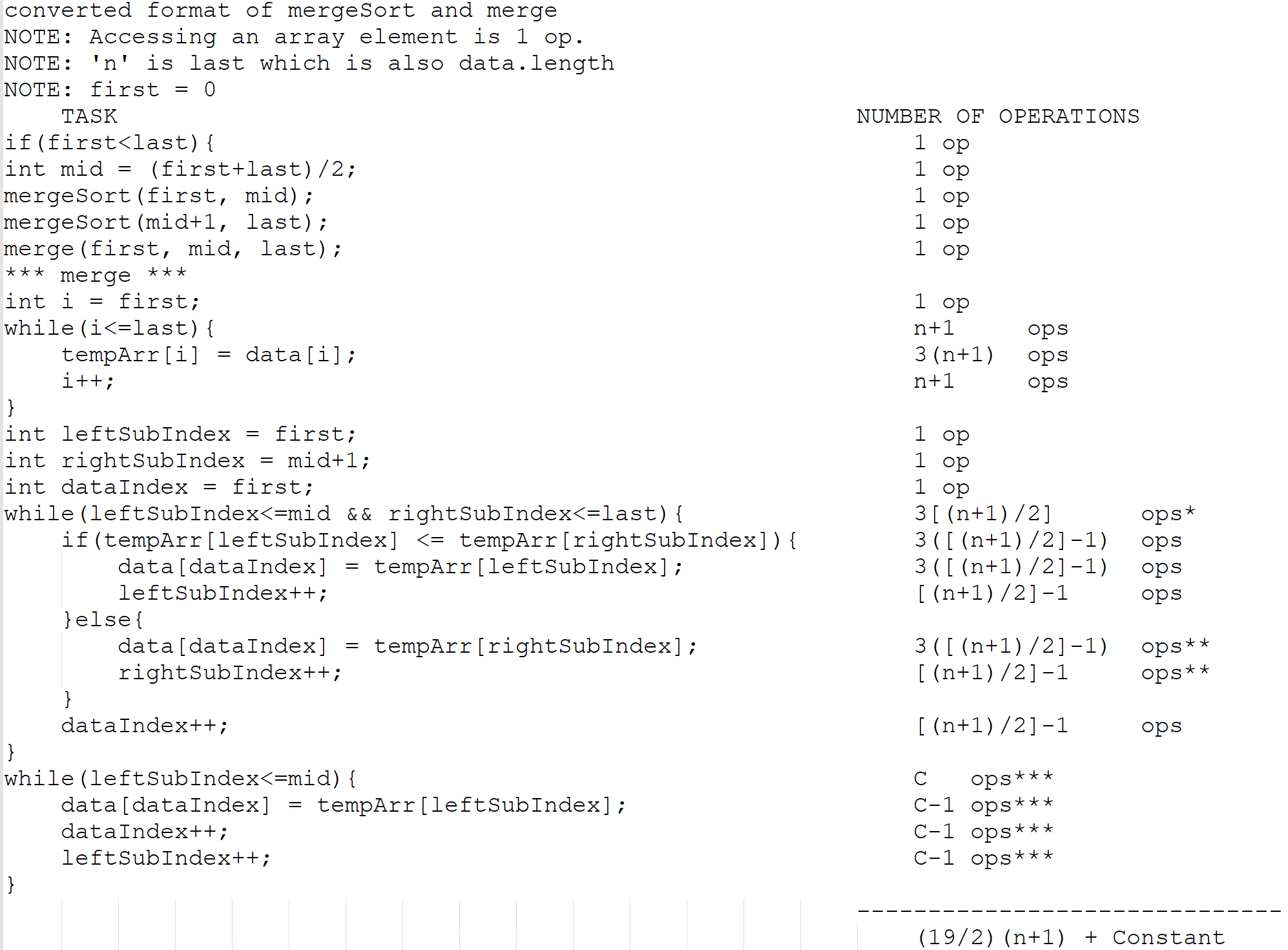
Examining the graphs of ascending initial order, it can be seen that selection sort does poorly in both small and big number of inputs compared to the other algorithms. It is not as visible in the graph but from the table, insertion sort does the best under all situations. For descending initial order, it is insertion sort that suffers as the number of inputs increase while merge and quick sort are relatively the quickest ones. Lastly, for a random initial order, for a small number of input insertion sort is the slowest but for larger ones – selection. Quick sort is the one that performs well in a randomly arranged array.

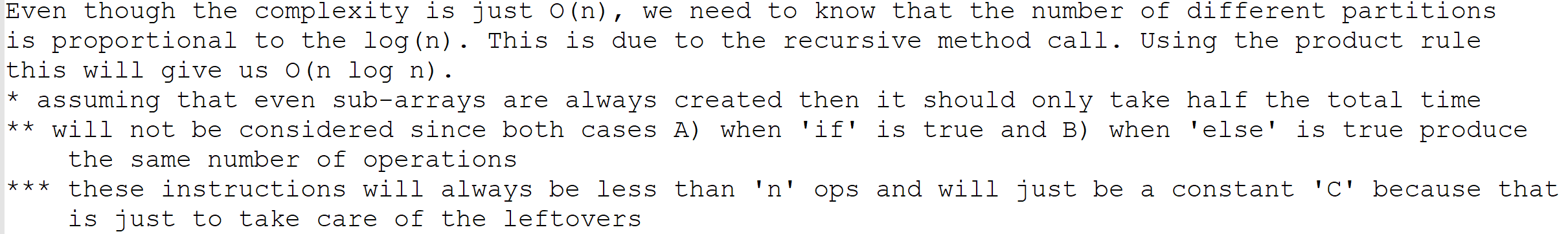
**IV. Complexity Analysis**

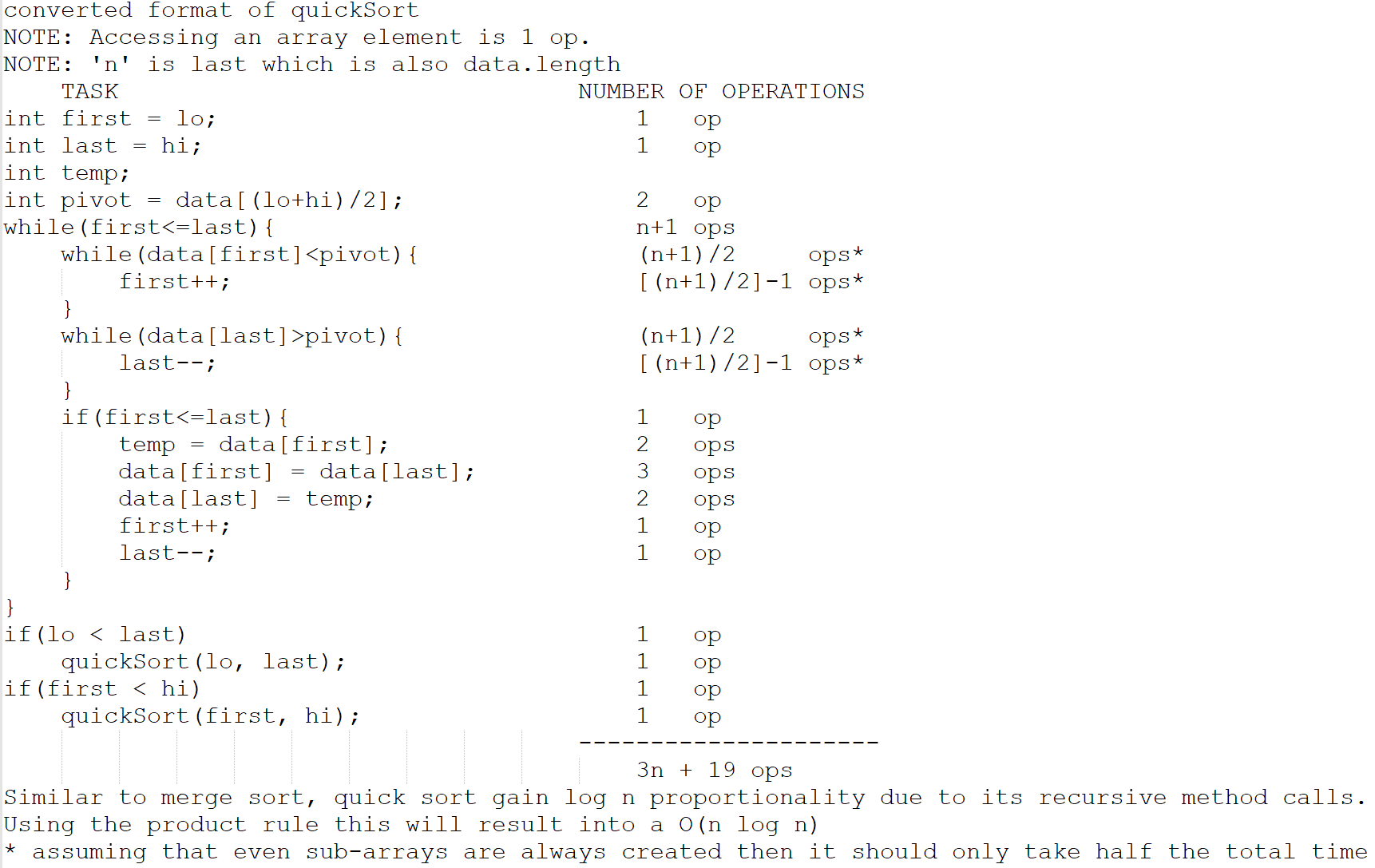
The complexity analysis of each sorting algorithm is conducted assuming the worst-case scenarios. This could mean that if-statements are always true and other conditional statements are always satisfied.











**V. Interpretation**

Based on the empirical data collected, for a fixed number of inputs all but one algorithm – insertion sort – performs at a relatively constant speed. This means that the sorting speed of selection, merge and quick sort are not dependent on the initial order of the array. In terms of which algorithm is best for each input size, we can infer to the data table and see that selection and insertion are the most fit for this task. Due to the simplicity of these algorithms, smaller sample sizes are very easily handled compared to the more complex ones such as merge and quick sort. Comparing the experimental with the theoretical performance of each algorithm based on the complexity analysis conducted, we get the following table:

|  |  |  |
| --- | --- | --- |
| Big O | Worst-Case Complexity Analysis | |
| Experimental | Theoretical |
| Selection | O(n2) | O(n2) |
| Insertion | O(n2) | O(n2) |
| Merge | O(n log n) | O(n log n) |
| Quick | O(n log n) | O(n log n) |

It is evident that the experimental results coincide with the theoretical values of Big-O for each algorithm. We can also see that selection and insertion sort have the same Big-O classification and so does merge and quick sort. Even though these have similar Big-O classifications they may have differences when it comes to their lower bounds since Big-O only specifies the upper bound. We can examine the graphs and table above to see the differences in performance. We can treat the ascending initial order of an array as a condition for lower bound. This clearly illustrates the difference in performance of insertion and selection sort. Insertion sort, even though it is thought to be a simpler algorithm does better than the more complex ones in this situation even more so than selection sort which has a similar Big-O classification.

**VI. Conclusion**

There’s plenty of sorting algorithms that can be implemented. Each having their own strong and weak points. From the four algorithms examined in this lab it is ideal to use either selection or insertion sorts for smaller samples. Merge and quick sorts are very useful when dealing with larger amounts of data. There are also specific sorts that might fit a certain task better than others. If it is known that the initial list of data is relatively sorted, regardless of data size, insertion sort is the best algorithm to use (refer to tables and graphs). For most cases (average case), it is ideal to use quick or merge sorts. The main dividing factor is whether using extra memory (merge sort) does not interfere with other tasks at hand. Even though the Big-O classification for some algorithms are the same as others, when further examined for specific cases, differences in performance can be observed. It must also be kept in mind that for some applications a specific algorithm will have limited capabilities such as merge sort in an embedded system. This will be very memory expensive for a system that already lacks the excess memory to perform other tasks even more so a sorting algorithm that demands memory space to create a temporary holder.

**VII. Appendix**