Grant McGovern

Dr. Pauca

CSC 222

March 2, 2015

Lab 2 Write-Up

Introduction:

Perhaps some of the most historical sorting algorithms in the history of computer science, quick Sort, merge Sort, and selection Sort have served as benchmark algorithms for understanding run time complexities within the backdrop of large data sets. While basic sorting algorithms like bubble sort do exist, their complexity is inferior to the above sorting algorithms that leverage powerful techniques such as divide and conquer steps. In this lab, I will implement and examine the above sorting algorithms and test them against different data sets. The first algorithm is selection sort whose worst case time complexity is O(n^2). The second algorithm I will implement and examine is merge sort whose worst case time complexity is O(nlogn). Lastly, the final sorting algorithm I will implement and examine is Quick Sort, whose worst-case time complexity is also O(n^2). For the purpose of this lab, all of the above algorithms will be implemented in the C++ programming language. The dataset I will be using is a 235,886 thousand-line file containing random words. Each entry in the file is delimited with a new line character ('\n'). Each sorting algorithm is constructed via a class and a public *sort* method, which can be called from the main program. All of the files are linked against a *makefile*, which upon execution, runs the program and outputs the results to a text file. This text file contains information regarding the size of the input data, as well as the time taken by each algorithm to sort it. In this lab, we will do analysis on each algorithm by examining these results and ultimately plotting them side-by-side on a graph to visually examine their complexities.

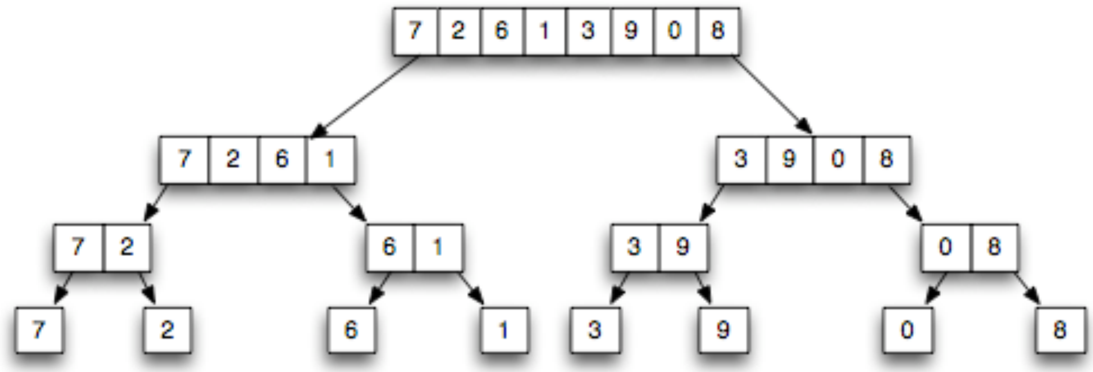
Algorithm Explanation:

Selection Sort:

Selection sort is an algorithm that works by finding minimums. It first scans the list, searching for the minimum, and uses that minimum as the basis (the first element) of our newly sorted list. We repeat the process, continually finding the next smallest minimum, and swap that element with the head of the unsorted list. This routine is recursively performed against (n-1) elements, until the newly formed (sorted) list is completed.

Merge Sort:

Merge sort is a quintessential divide and conquer algorithm that leverages the power of recursion to sort lists. It is also the perhaps the best performing sorting algorithm with a complexity of O(nlogn). As you will see in its implementation, merge sort is divided into two methods, the first one being the *mergeSort* driver method and the latter being the *merge* helper method. The *mergeSort* method first takes the list and splits it on the middle element (midpoint). This process is repeated until all the elements are broken up into individual elements such as below:



Then, the *merge* method merges together the elements by recursively comparing each element and selecting the smaller. This process is repeated until the list has been completely sorted. Because merge sort is recursive, we can develop the following recurrence relation:

$$T(n) = 2T(n/2) + O(n)$$

This is because the *merge* routine is called twice in the *mergeSort* method and the list size is halved each time through the algorithm (n/2). The extra "O(n)" on the end of the recurrence relation is the cost of merging the two lists, which is simply linear. Using the Master theorem to prove the time complexity, a = 2, b = 2, and d = 1, thereby giving us the equation $\log_2 2$ which is equal to 1. Therefore, the time complexity of merge sort is in fact O(nlogn).

Quick Sort:

Like merge sort, quick sort is also a recursive, divide and conquer algorithm. However, it differs from merge sort in that it uses a pivot around which to partition the list. This pivot, a return value of the *partition* method, can be calculated in several ways (even randomly). However, I chose the rightmost value in the array as my pivot because I thought it would be interesting to see how this would affect the complexity of the algorithm. I learned in 221 that sometimes this can be an advantageous decision, and because I knew the list was not sorted, I decided to run the experiment using the rightmost array value as my pivot.

Quick sort works by making sure all elements **less** than the pivot are to its left and all the elements **greater** than the pivot are to its right. To enforce this rule, quick sort employs a swapping method between **i** and **j,** comparing them each time so as to enforce the above rule. The quick sort method is then recursively called until both lists on each side of the pivot are sorted. Lastly, we swap the pivot with the last element in the array to return a fully sorted list.

Perhaps one of the greatest anomalies with quick sort is the discrepancy in complexities between the average and worst case. On average, quick sort has a complexity of O(nlogn). However, in the worst case, it has a time complexity of O(n^2). The worst case occurs if the chosen pivot happens to be the largest or smallest element in the list. This would yield a list containing the smallest or largest element, and another list containing the remaining elements. The same would be repeated after each recursive call, obviously resulting in a terrible run time complexity.

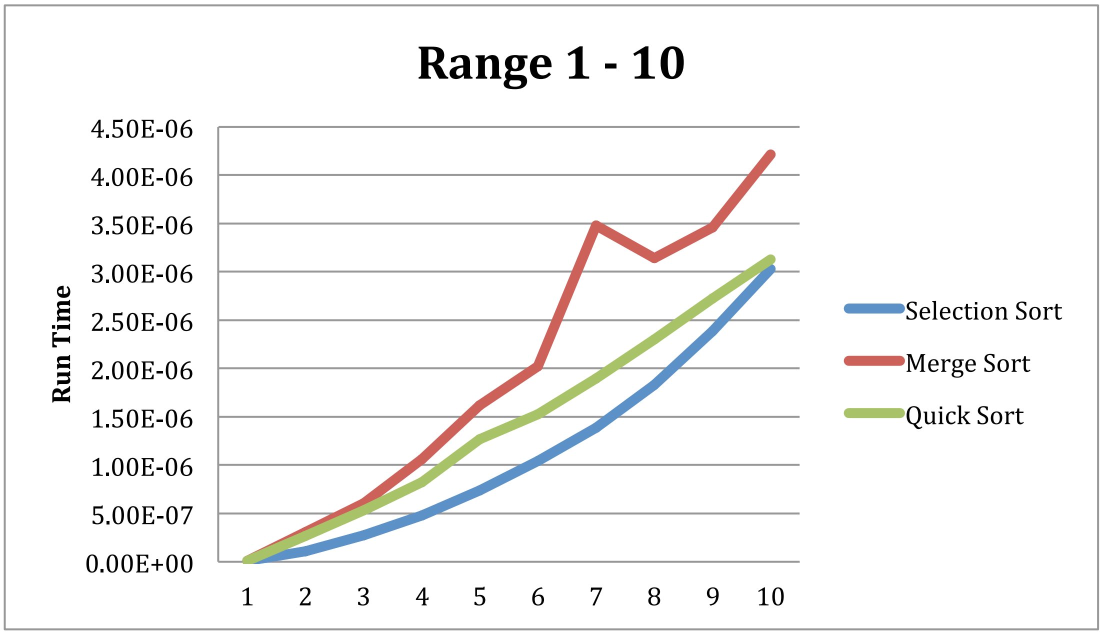
Method Explanation:

In order to best see the complexities of each sorting algorithm, I elected to test each algorithm against four different size ranges (where size is the number of elements being fed into each sorting algorithm). The ranges were 1-10, 10-100, 100-1,000, and 1,000-10,000. The reason why I decided to break down my experiment into multiple ranges was because I wanted to be able to visually see the differences between algorithms at small list sizes and at large list sizes. Initially, I ran the experiment one time between the values of 20 and 10,000. While I was able to get a good sense of the different complexities for each algorithm, there was such an immense range of data that it was hard to clearly see the differences between algorithms at small input sizes. Thus, I elected to break the lab into different ranges, so I could truly witness each algorithm's complexity at different scales. I also set the trial number equal to 500 because I wanted to ensure a high level of accuracy and a suitable number that would yield a steady decay. For the first two ranges, 1-10 and 10-100, I made the problem size equal to 10, and for large input sizes, I made the problem size equal to 50 because I wanted to see how a larger range of problems would affect the graphed complexity of each algorithm.

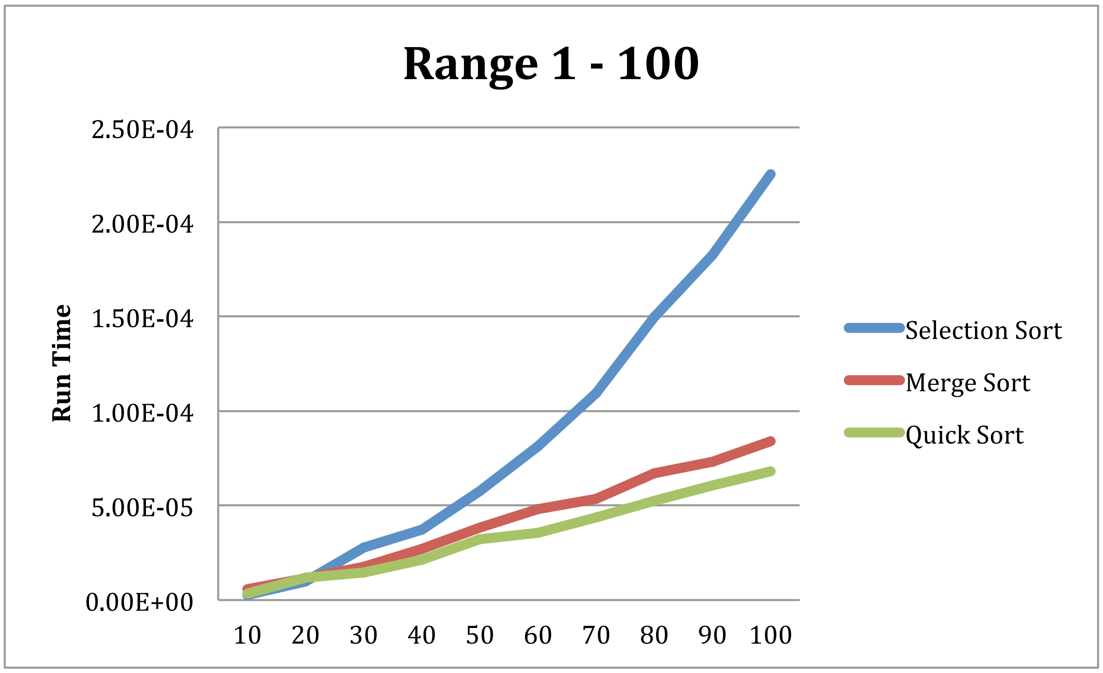
Having learned about the implementation and complexity of selection sort, merge sort, and quick sort, I developed several hypotheses prior to running the experiments. First, I assumed that selection sort would be faster for smaller input size because of its simplistic implementation. Secondly, having done a fair amount of research regarding sorting algorithms as a computer science undergraduate, I learned that quick sort is *generally* the most efficient algorithm in the average case and I expected that it would outperform merge sort with larger lists. I also assumed that selection sort would quickly become quadratic O(n^2) as it sorted against larger inputs.

Results:

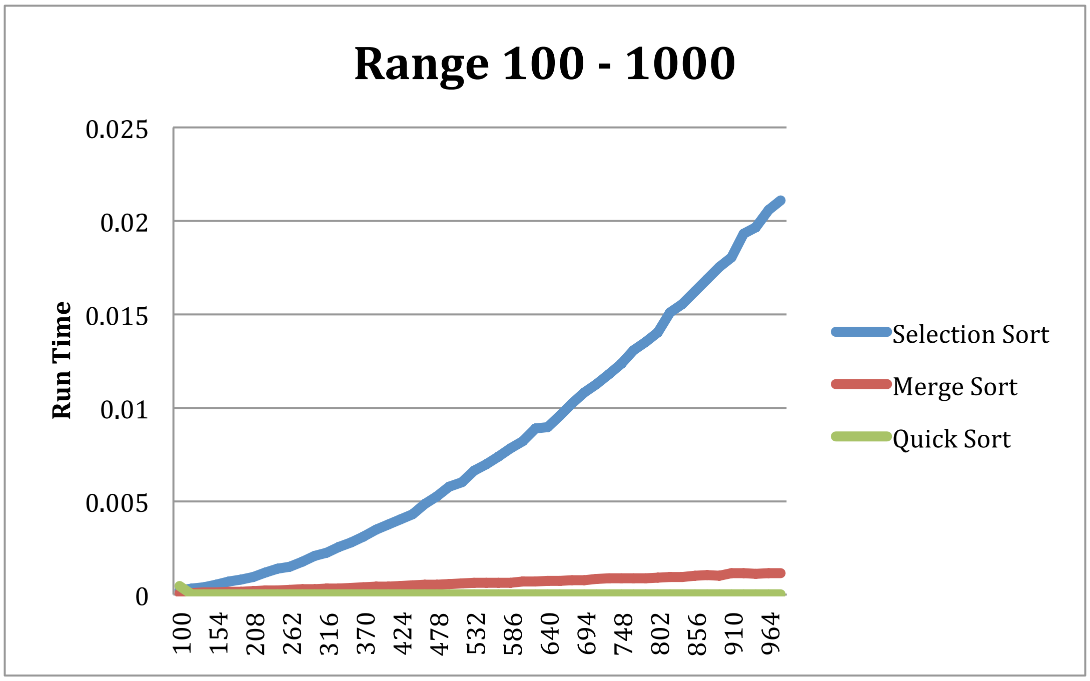
For the first range of 1-10, we can see that selection sort is the best algorithm. It outperformed merge sort and quick sort for small input sizes, which is what I expected given the algorithm's implementation.



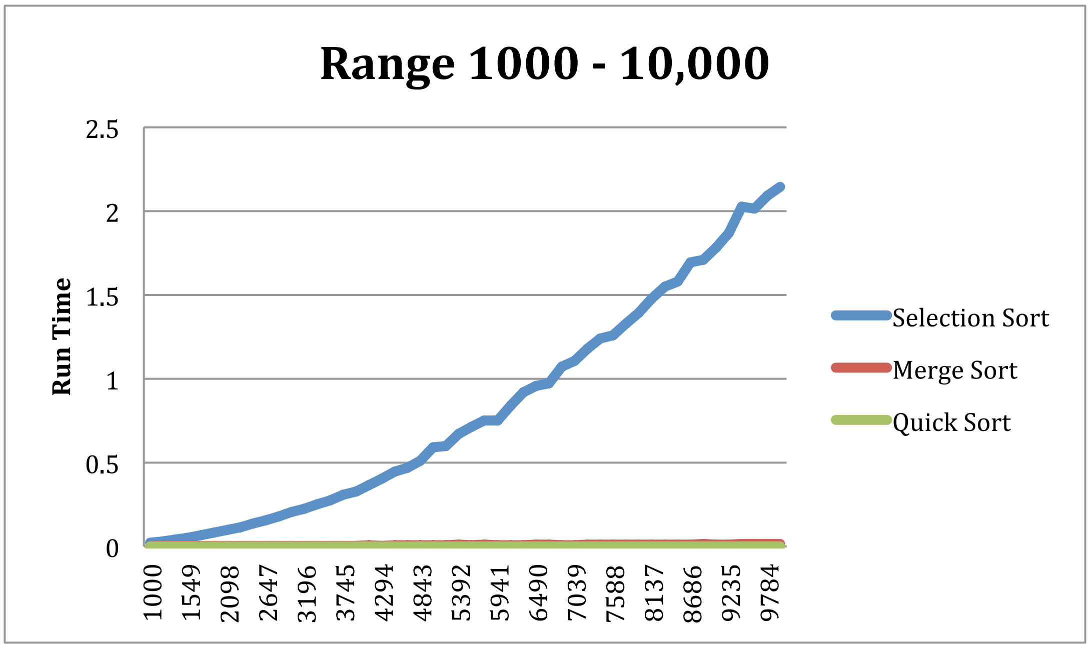
In the second range of 1-100, we quickly see the performance of selection sort decrease. The graph immediately becomes indicative of a quadratic function, and quick sort is still outperforming merge sort.



For larger inputs, such as 100-1000, we still see selection sort's complexity grow exponentially, however, quick sort and merge sort truly take the shape of O(nlogn). As proven above, this makes sense because merge sort has an O(nlogn) worst case complexity and quick sort has an O(nlogn) average case complexity. In this lab, we have not exploited the worst case of quick sort.



Finally, for a large range of input, 1000-10000, we see that merge sort and quick sort almost become indistinguishable unless examined very closely.



Conclusion:

Although the worst-case complexity of quick sort is much larger than the worst complexity of merge sort, after performing this lab experiment, I found that quick sort *generally* performs as well or slightly better than merge sort, especially for larger inputs. As expected, selection sort performed well for small input lists, but quick fell behind merge sort and quick sort as the input size grew.

After analyzing my lab data, it is hard for me to say whether quick or merge sort performed best in this lab. The data shows that their complexities were very close if not equivalent for some data points.