Project #1
CS3410- Dr Gallagher
9/11/09
Kevin Bender, Stephen Halwes, Alex Laird

Speeds of Insertion and Quick Sort

Insertion Sort vs. Quick Sort

To write the tests which would compare the sorting time of Insertion Sort and Quick Sort, we used the Java programming language. The basic algorithms remained constant throughout all the tests; however we ran our tests on multiple computers and tested varying styles of coding (methods versus classes). Changing our environment frequently changed the overall time of sorting, but the generalizations of the comparisons remained relatively constant from setup to setup. We used the System.nanoTime() function to display results in nano seconds. This ensured more precise sort times for our tests.

The first test was Insertion Sort speeds versus Quick Sort speeds. As is shown in Figure 1, in a basic Insertion Sort versus Quick Sort, the sort times returned as expected: initially the Insertion Sort algorithm performed better than Quick Sort. Then, at a particular threshold somewhere between 10 to 90, Quick Sort became more efficient. We noted that this region varied quite a bit depending on different characteristics of the computer, most importantly the processor speed, so we ran these tests using several different processor speeds.

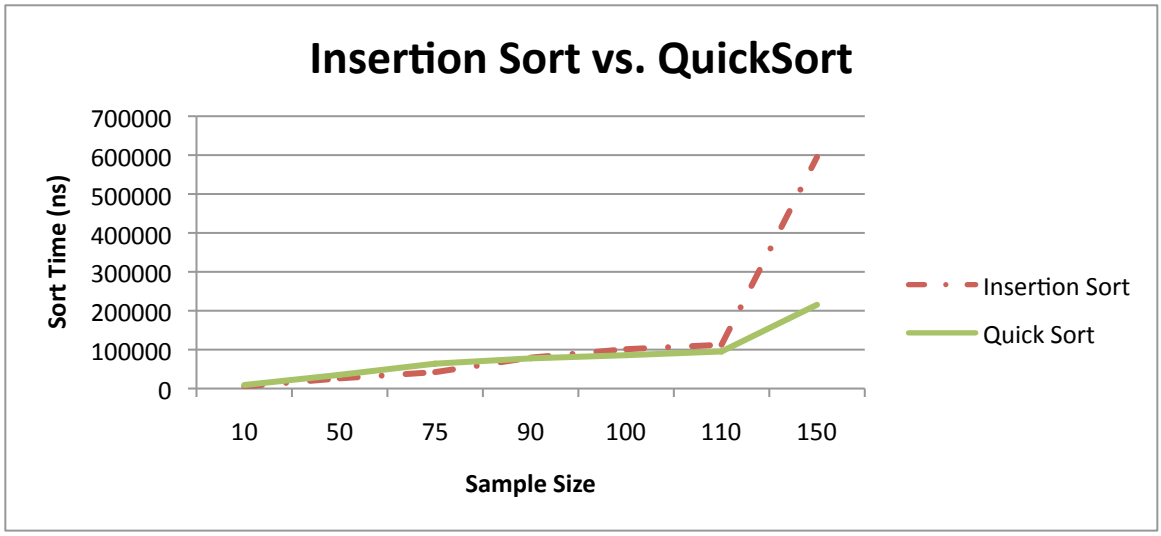

Figure 1 InsertionSort vs. QuickSort

In tests of greater than 100, we saw a trend where Quick Sort began to perform much better than an Insertion Sort. This was to be expected due to the asymptotic analysis of the sorts; Insertion Sort is $O(n^2)$ and Quick Sort being $O(n \lg n)$, so we expected that this trend would be indicated in our data. We also noted that both algorithms would not deal with sample sizes of 100,000 or more, specifically on pre-sorted data, because of stack overflow errors. This was, in many aspects, a hardware shortfall.

Sorts on Randomized Data

Figure 2 and Figure 3 go on to show the varying differences in times from search to search. Understand that Figures 2 and 3 are not meant to compare Insertion Sort to Quick Sort, rather they are meant to compare one random sort of size X to the next random sort of size X. They are meant to illustrate how much one sort time can vary from the next sort time, even on arrays of the same size. As is shown by the figure, the amount of time that can vary from sort to sort increases as the size of the array also increases.

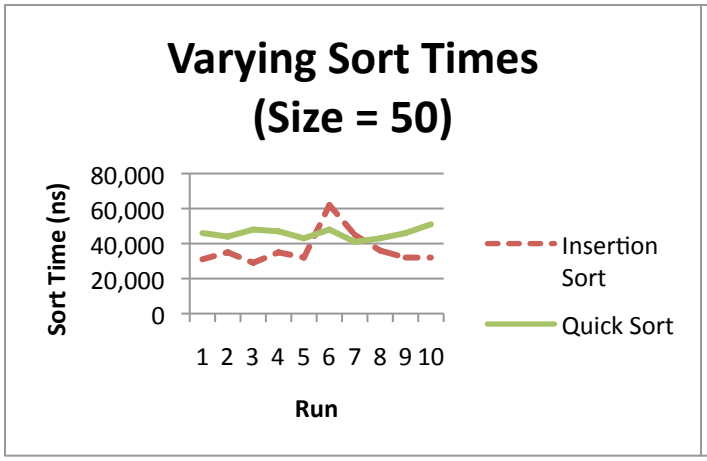

Figure 2 Varying Sort Times (Size = 50)

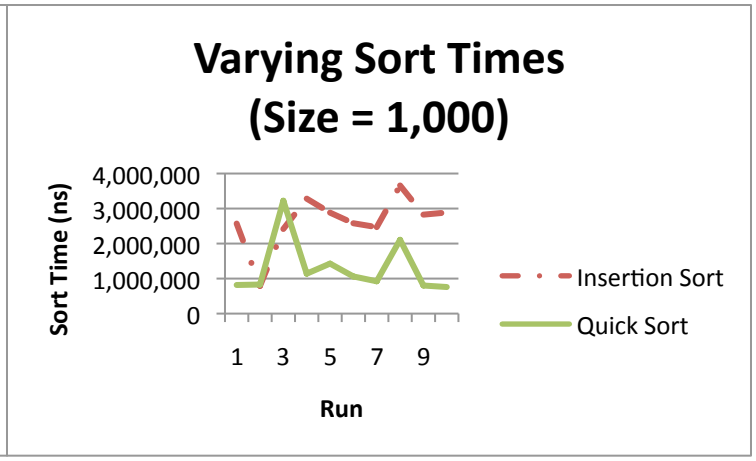

Figure 3 Varying Sort Times (Size = 1,000)

Analysis of Algorithms on Pre-Sorted Data

When running the sort algorithms on pre-sorted data, we found that Insertion Sort was the most effective, having the lowest runtime on any sample size. Since Quick Sort still has to partition the array recursively before attempting to sort the data, whether the data is sorted or unsorted is not a not a contributing factor to the efficiency of Quick Sort. Insertion Sort, however, only enters the nested loop, where the brunt of the performance time is lost, if an array element is found out of place. Therefore, Insertion Sort is the best algorithm to use if the data is most likely pre-sorted (or relatively sorted). As is shown in Figure 4, Even on 50,000 elements, Insertion Sort still only requires 568,000 nanoseconds while Quick Sort requires over 12 million nanoseconds.

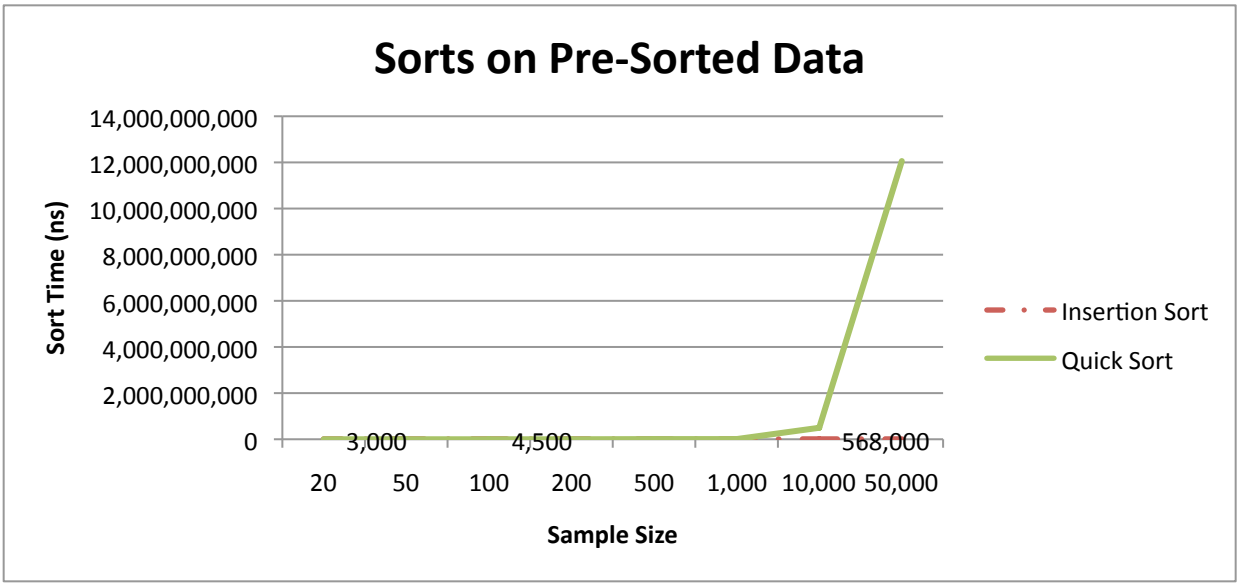

Figure 4 Sorts on Pre-Sorted Data

Differing Partition Methods

For differing partition methods, we found that, in general, randomized partitioning as opposed to standardized partition was slower on unsorted arrays. If the array was pre-sorted, randomized partitioning usually finished the sort assurances quicker. These findings are represented in Figure 5. The for loop choosing a random pointer, which used a single-pointer approach, was fastest for smaller sample sizes. However, once the sample sizes got into the thousands, it became evident that the while loop with standard partitioning, which used a double-pointer approach, performed the fastest sort. Large-scale projections of this are shown in Figure 6. The details regarding the differences between randomized and standardized partition as well as for loop and while loop partition can be found in the code comments.

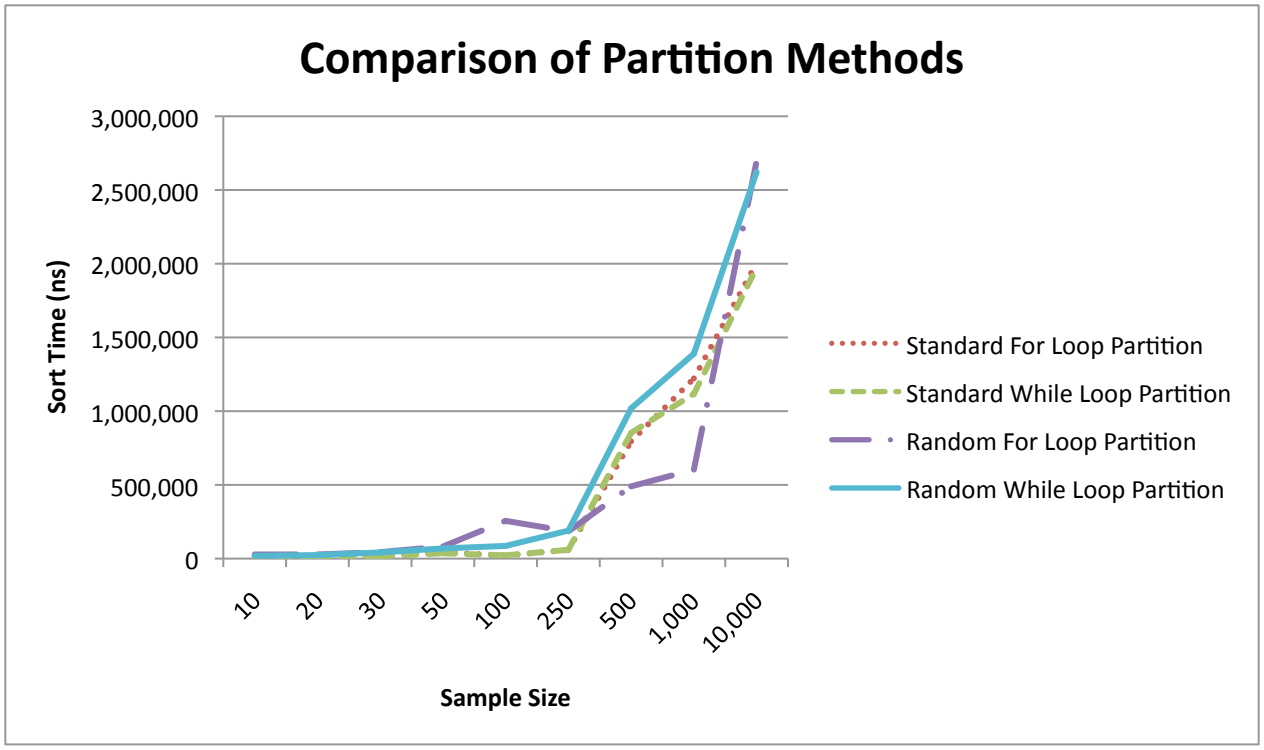

Figure 5 Comparison of Partition Methods

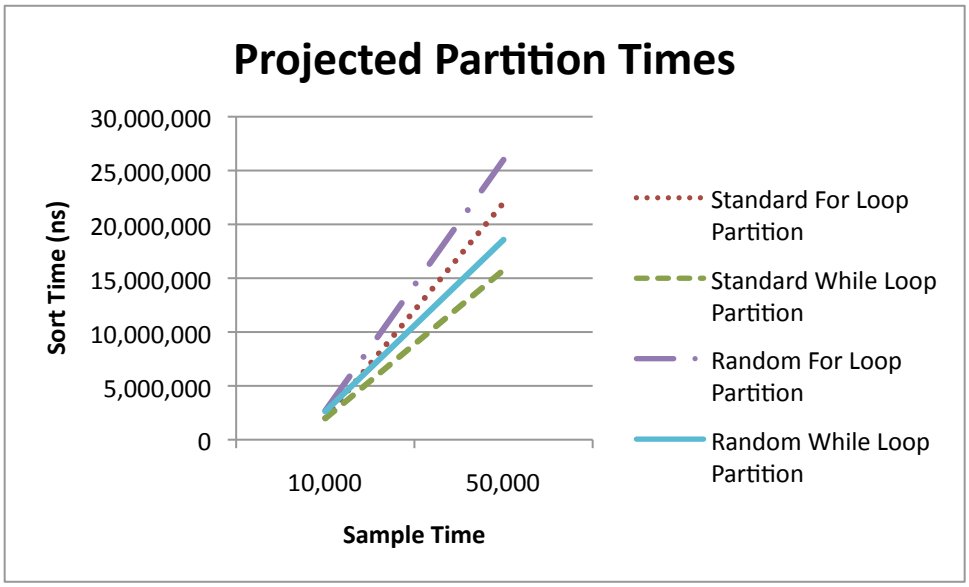

Figure 6 Projected Partition Times

Our code compiled correctly and ran the tests with expected results.