Brandon Graybeal

IT 279

Paper 1

**Analysis of Different Sorting Methods**

**Introduction**

In order for a program to perform and execute as quickly as possible, we must first make sure we have optimized our code by using the most efficient methods and algorithms. However, determining which algorithm is the most efficient can sometimes be hard to measure. Because there are variables that can affect the performance of an algorithm across different machines, simply timing how fast an algorithm executes is not sufficient. In this experiment, we take a different approach to measuring four different sorting methods. In each sorting algorithm, the number of comparisons and the number of assignments are tracked so that the user can see the total number of each for varying sizes of an array of double data type numbers. The resulting numbers can then be compared across algorithms to see which one makes the least amount of comparisons or assignments as it sorts the array. For this specific example, the input data to be sorted comes from the file "data-random.txt" provided. After we have the results, we can then apply a "cost" to each algorithm based on which programming language they are being implemented in, and the language with the least resulting cost can be determined as the most efficient.

Included in this report is the analysis of measuring how efficient a heap sort, insertion sort, quick sort, and merge sort algorithm are. The data is displayed by graphing both the number of comparisons and assignments for each method of sorting. In addition , the cost of using each sort is also included to see how the implementations are affected by the language .

**Analysis**

After performing this experiment, we get the resulting graph shown above for the total number of comparisons each algorithm performs. As you can see on this scale, Insertion sort performs significantly more comparisons than any of the other sorts. This is because it must compare to every item in the array, and is the reason it performs in O(n2) time.

We can zoom in on our data points for the other sorting algorithms by removing Insertion sort from the graph. After doing so, we can see that Merge sort makes the fewest number of comparisons, followed by Quick sort, and then finally Heap sort. As we can see above, when an array is very small, all three algorithms make relatively the same amount of comparisons. However, as the size increases, it becomes evident that Merge sort is the most efficient.

Shown above is the resulting graph of the total number of assignments each algorithm performs. Again we can see that Insertion sort makes significantly more assignments as well.

As you can see, we have again removed the insertion sort from the graph so a clearer picture can be seen about the total number of comparisons. In this experiment, I found that for a very large array size, Merge sort again makes the fewest assignments. This is followed by Quick sort and then by Heap sort. However, whenever the array size is small in the data set, Quick sort actually performed better than Merge sort, although it is not by that much.

After determining the number of comparisons and assignments, we can then use these values to simulate the cost of how both Java and C++ implement these. Again, as expected, Insertion sort has a very high cost of implementation.

After removing Insertion sort, we again get another graph similar to the others pictured. For a very large array, Merge sort is the most efficient in a Java implementation. However, for smaller sized arrays, Quick sort is nearly identical in cost to Merge sort.

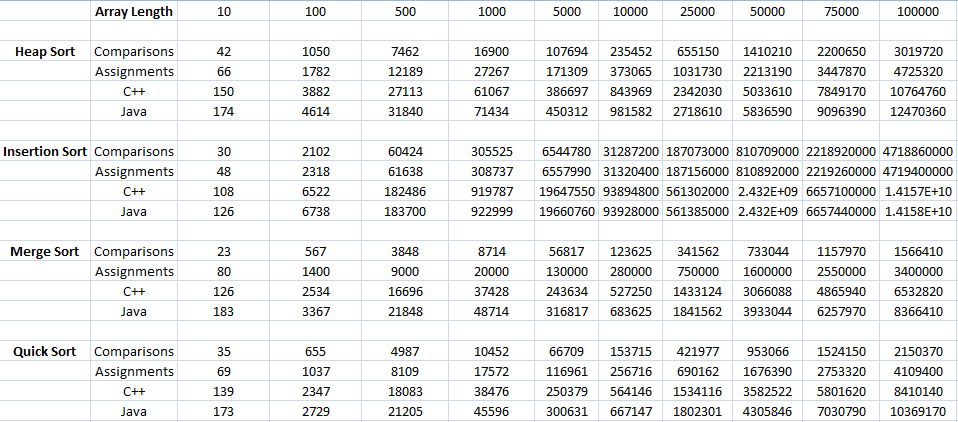
After simulating a similar cost graph for a C++ implementation, we see that Insertion sort has a very high cost. But as stated, this is expected because it makes many more comparisons and assignments than the rest of the algorithms do.

After zooming in, we get the same results as in the Java implementation. For large arrays, Merge sort is the superior algorithm. For smaller array sizes, Quick sort and Merge sort are nearly identical in cost.

**Conclusion**

* We observe that the Merge sort makes the fewest number of comparisons, followed by Quick sort, Heap sort, and finally Insertion sort.
* We observe that Merge sort also makes the fewest number of assignments, followed by Quick sort, Heap sort, and Insertion sort.
* Because of this, the resulting cost for a Java implementation has Merge sort as the superior algorithm for large arrays, however, for smalls arrays Quick sort has a nearly identical cost.
* Merge sort is also the most efficient in terms of cost in a C++ implementation, and again, Quick sort has a cost very close to Merge sort.
* Merge sort appears to be the most efficient algorithm, however, it also has a space requirement of O(n) that Insertion sort, Quick sort, or Heap sort do not require.
* This extra space requirement allows Merge sort to make fewer comparisons and assignments, and be the most efficient sorting algorithm for this data set.

**Appendix**

****