Alejandro Zaragoza
Dr. Andrew Predoehl
CSC 345
Program 2 Report

Introduction

A common necessity many computer programmers need in code, especially code that keeps track of massive amounts of data, is a sorting algorithm. Sorting is any process of arranging items systematically. Sorting is useful in everyday life for things such as listing students in alphabetical order, numbering our bills in our wallets from $1 to $100, or standing from shortest to tallest. Computer programs need data to be sorted so that when applications need to refer to certain data entries, it would be much easier to find later. Since this is such a common task that computers need to do, especially for enormous amounts of data that sorting by hand would be ridiculous, algorithms have been developed and accepted into common computer programming practice.

Some of the most common sorting algorithms are **Bubble Sort, Insertion Sort, Heap Sort, Merge Sort,** and **Quicksort**. Thanks in part to large amount of funding the government as given our firm, we now have the means to test and analyze how each of these algorithms compare for various sizes of data. By the end of this scientific experiment, we should see how each algorithm stacks up and is the best!

*UPDATE: We have just received contact from our outer space reptilian crypto-overloads that they heard about our firm's experiment and would like us to test their sorting algorithm, **Robo Reach Sort**, against our algorithms to put our human developed algorithms to shame once and for all! If we do not compromise, we are all doomed!*

Perfect! So this experiment will analyze six different sorting algorithms. We will investigate which algorithm performs best for small amounts of data, as well as large amounts of data? How much does performance randomly vary? And subjectively present which algorithm performs best overall.

Procedures

To test each algorithm, a program was written up in Java by our unpaid intern.This program contained a master file: *SortTime.java*, which contained code obtained from D2L that was updated to meet our firm's specifications. Specifications being:

1. REPS (the repetitions variable) was changed to 1000 for small array sizes (under 1000), and changes to 100 for large array sizes.
2. The first argument was changed to take the max array size to test. A *for-loop* was implemented to run from array size '100' to the 'max' size specified by the user.
3. The program now takes a second argument which corresponds to which sorting algorithm to test. Each number, 0 through 5, corresponds to one of the six sorting algorithms.

This code generates arrays filled with random integers and each algorithm must go through numerous times and sort these randomly generated array of ints.

The code for **Bubble Sort** and **Insertion Sort**, being the two easiest to implement, was primarily written by the unpaid intern Alejandro, but the slides on D2L were referenced to check the work. These two sort methods are completely contained in the master *SortTime* java file. There were no issues when writing these functions up.

HeapSort was outsourced to its own java file. This java file contains three methods: *heap_sort_master*, *build_heap, and max_heap*. Since the unpaid intern Alejandro just learned about heapsort only recently, Shaffer's text was primarily referred to in writing these functions. The functions themselves were written differently, but were inspired from Shaffer's text. The main issues arose from writing the *max_heap* function since it was the most complex and dynamic involving recursion.

MergeSort was outsourced to its own java file. Shaffer's text was less relied on here, but was still used as a failsafe. *Merge_sort_master* is called from the main java file, then *sort* is called which splits the arrays. *Merge* takes two arrays and joins them together. The most writing was the *merge* function, but was not very complicated. The other two functions were somewhat simple as well.

QuickSort was outsourced to its own java file. It was written with majority of the help from Shaffer's text. Partition was rather difficult to write, since the partitioning itself isn't as most humans would think of sorting. Shaffer's text proved very useful here to explain how the partitioning and entire sort function worked.

Robo Reach Sort was completely supplied by the alien like beings.

The input sizes were determined by the first argument when running SortTime. This number was the maximum array size to test.

To switch between the six algorithms, the second argument passed in when running **SortJava** was an indicator on which algorithm to use. The sorting algorithm key is as follows:

0 - Robo Sort
1 - Bubble Sort
2 - Insertion Sort
3 - Heapsort
4 - Mergesort
5 - Quicksort

For example: *java SortTime 500 3* would run Heapsort as the algorithm, with the maximum array reaching a size of 500.

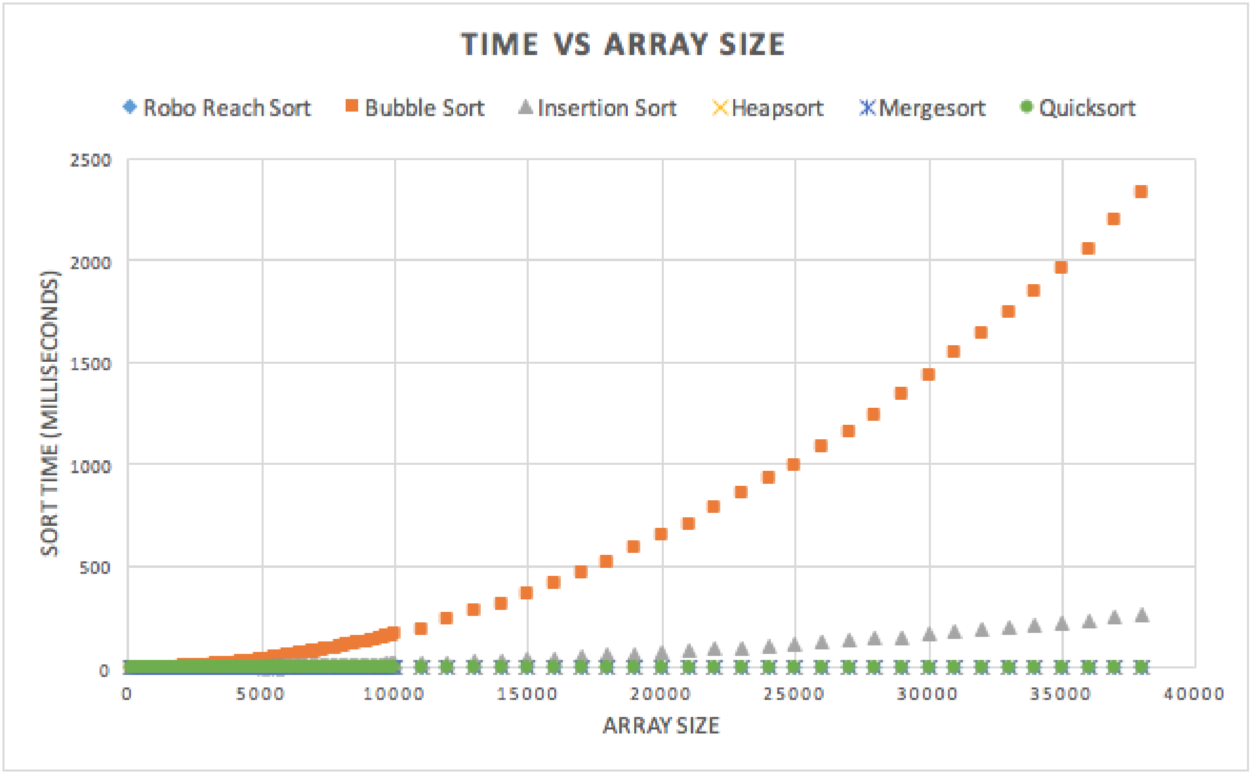
The amount of repetitions done for each array were 100, unless the array size was smaller than 1000. In which case, 1000 repetitions were done.

The sample data collected in this experiment was collected by running my java files on lectura. All except for bubble and insertion sort were ran with maximum array sizes of 100,000. The outputs containing the run times in milliseconds were stored into text files using IO redirection. The large amounts of data on these text files were then imported into Microsoft Excel.

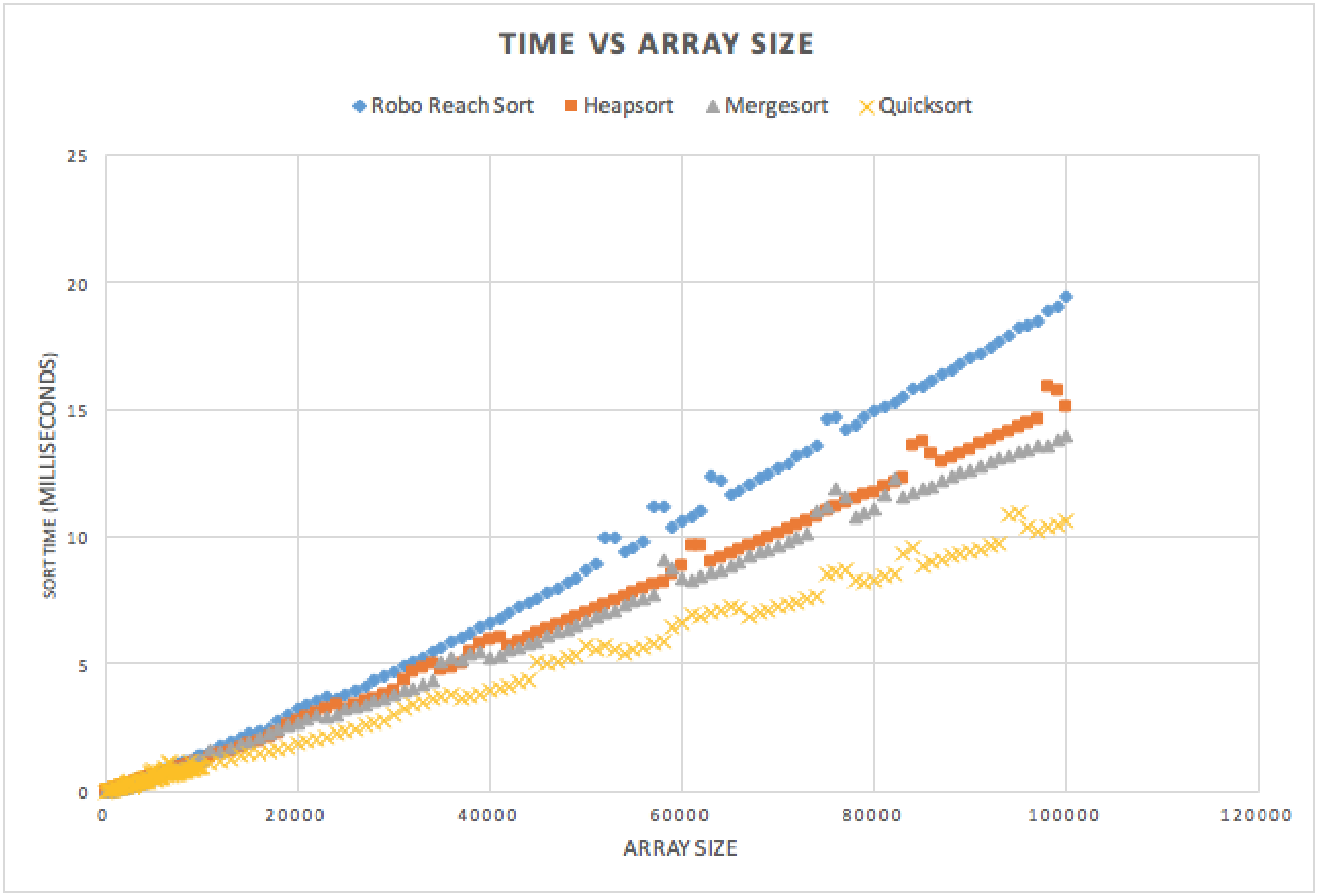
The data in Excel was then graphed to see how each algorithm performed in comparison to each other. The results are displayed below.

Results

The graphs below are Time vs Array Size for the different search algorithms. Time is measured in milliseconds. The graphs are scattered plots and give a good indication on what the trendline looks like for arbitrary large sets of data.



This graph only contains data for array sizes up to about 38,000 on all six sorting algorithms. The reason being that Bubble Sort would take so long that lectura would give a broken pipe. It is apparent that at this scale Heapsort, Mergesort, Quicksort, and even Robo Reach sort all perform relatively the same as each of their trend lines are not easily identifiable. Based on this data, Bubble Sort performs the worse with, Insertion Sort in second, on a large scale when compared to the other algorithms.



This graph does not include Bubble Sort and Insertion sort so the other four algorithms can be identified and analyzed much closer. In this graph array sizes are as large as 100,000. Other than a few rare instances, each algorithm pretty much grew linearly as the size of the array got bigger.

Conclusion

Based on the results of this experiment, it is apparent that Quicksort is the best performing sorting algorithm for massive amounts of data. When analyzing smaller amounts of data Quicksort, Heapsort, Robo Reach Sort, and Mergesort all perform relatively the same. Bubble Sort is by far the worse sorting algorithm. Insertion Sort does not perform as poorly as Bubble Sort, but still does not compare well to the other four algorithms.

The results indicate that the algorithms that perform best for large amounts of data could also be used for smaller amounts of data. Based on the findings, all algorithms work relatively the same for small amounts of data so using the best algorithm for large amounts of data would not be of expense.

Performance stays pretty constant for Heapsort, Mergesort, Quicksort, and Robo Reach Sort. There were a few odd indications, but overall there is consistency. Bubble Sort’s performance gets much worse as array sizes grow. Insertion Sort does grow faster than the four fast algorithms, but no where near as drastic as Bubble Sort.

Surprisingly Robo Reach Sort did not stack up as well as the other alien beings had hoped.