Project 3: Sorting

An Analysis of the Insertion, Merge, and Quicksort algorithms

The University of Georgia
CSCI 2720
FALL 2015

Montana Wong
8103439480

Qiang Hao
810624643

December 5th, 2015

Experimentation

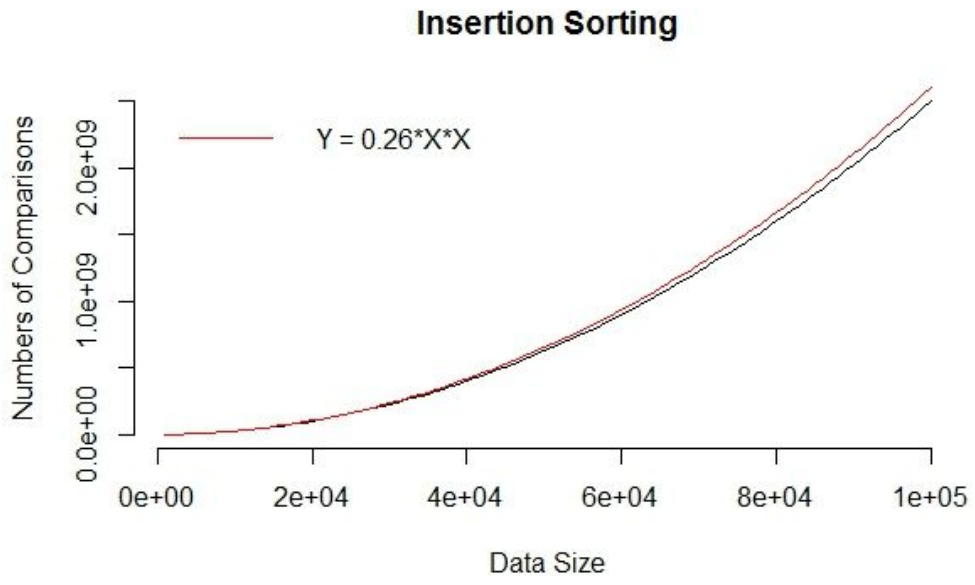
The experiments that were performed were par to the project instructions. 100 data sets were used, each set with 100 more elements than the previous. These data sets were generated using a c++ built-in pseudo random number generator, and the number of comparisons per set cardinality was graphed using an R script.

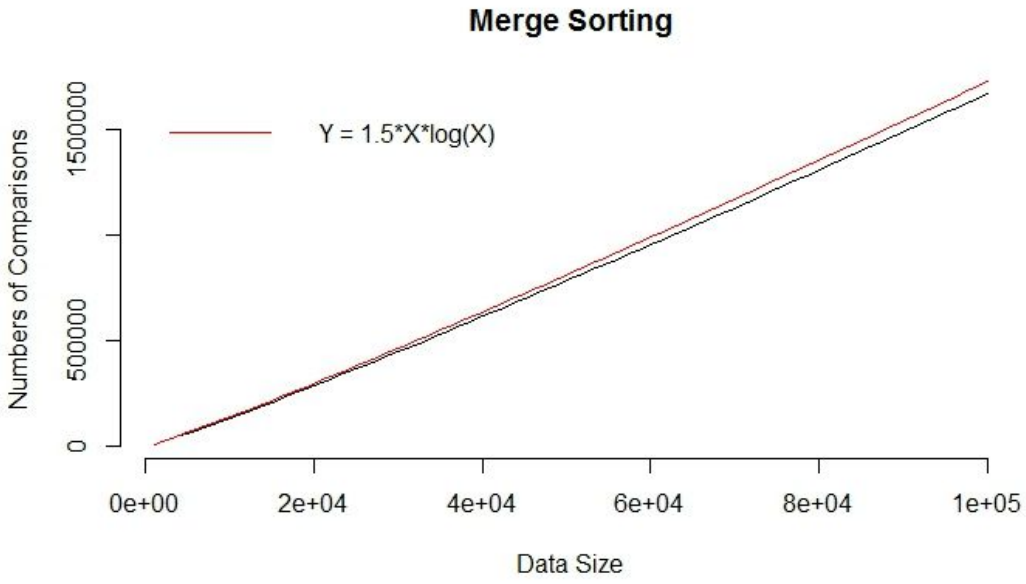
Quicksort Implementation

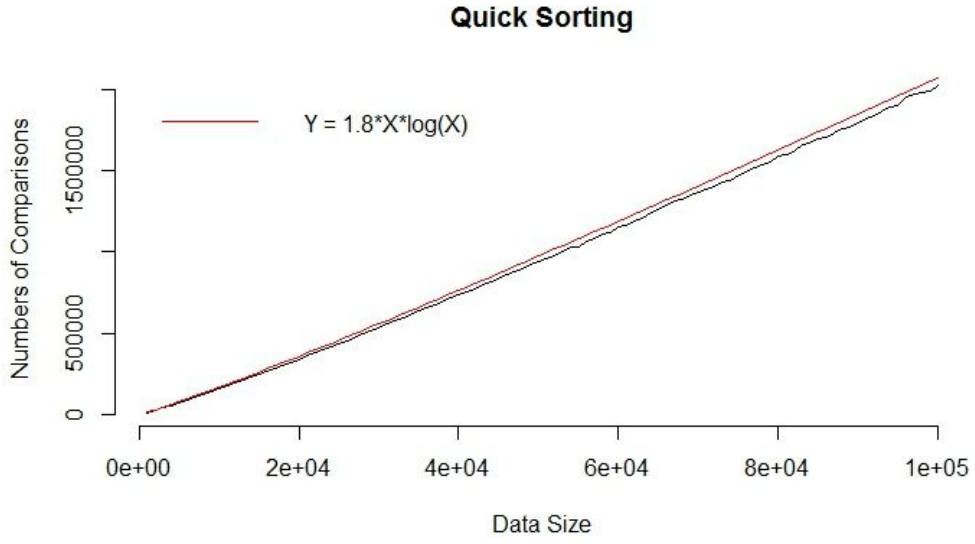
Quicksort is a divide and conquer sorting algorithm allows for efficient sorting of array like structures. Quicksort can be broken into two important steps. The **first** is the "Divide" step. The algorithm moves all elements in the subarray with a smaller value than the pivot before it, and all elements with a greater value after this. The last element in the subarray is always chosen to be the pivot point. The method of choosing a pivot point is very important in any quicksort implementation, this will be discussed in more detail later in the report. This is implemented in our C++ code as a method called Partition, which returns the index of the pivot. For example, assume that the array before partition is called is (1, 5, 2, 9, 7, 4). After Partition is called, the new array will be (1, 2, 4, 5, 9, 7), and the index (2) of the pivot 4 is returned. Note that the partitioned array is not necessarily sorted. **The second** is the "Conquer" step. The algorithm splits the array into two, not necessarily equal, halves based on the chosen "pivot" point. Then quicksort is recursively called on both halves until the base condition is met. At this point our array is sorted.

Choosing a pivot point to partition the two subarrays at each recursive call is a key component in any efficient implementation of quicksort. The sorting algorithm runs in average case O(n*log(n)) and worst case $O(n^2)$. Generally an algorithm's runtime is determined by its performance in the worst case. However, quicksort's worst case condition is a very rare case; if the pivot chosen is always either the greatest or smallest element in the array (and subsequent subarrays). This problem may be alleviated by using methods such as picking a random index as the pivot or using methods such as "the median of three". Despite these more sophisticated options, our pivot selection method works well in most cases.

Comparison of Algorithms







Analysis of Data

Based on the data above, it is fairly easy to see that mergesort and quicksort are in a different order of magnitude than insertion sort. The number of comparisons for insertion sort tend towards the tens of millions, while merge and quicksort tend towards the hundreds of thousands for data sets of size ~ 10,000. This is due to the nature of the algorithms. Divide and conquer algorithms generally run in n*log(n) time. Insertion sort is a quadratic algorithm and therefore runs in n^2 time. Just by basic mathematical calculations we can deduce the large disparity between the number of comparisons of the two groups as the data set tends to infinity. For example, for a data set of size 6000, insertion sort's cn^2 complexity should result in roughly $c* 6000^2$ (c * 36,000,000) comparisons. Merge and Quicksort's c*nlog(n) complexity should result in roughly c * 6000 * log(6000) (~50,000) comparisons. Both of these predictions are verified by our data plots, our experiments agree with the established theoretical Big-O values. Actual values of c were predicted during our analysis that satisfy the Big-O bound, they can be found on the plots. (c = 0.26, 1.5, 1.8 for insertion, merge, and quicksort respectively).

Examining the merge and quicksort data plots reveals another disparity, the former has a few tens of thousand less comparisons than the latter. This may be a result of any number of randomized things such as the values of the data sets used, whether or not the computer was scheduling tasks while the program was running, etc. However, it is known that quicksort has a worst case algorithmic time complexity of n^2 in certain situations (data set is sorted/reversed). It may have been the case that one or more of the data sets produced could have been (*nearly*) sorted/reversed,

which would have caused the increased number of comparisons compared to mergesort. Mergesort's pitfall is in it's O(n) space complexity, however it was not measured in this exercise.