An Analysis of Common Sorting Algorithms Using Random Data

Joshua Glass

The College of Charleston  
66 George St  
Charleston, SC

glassja@g.cofc.edu

# INTRODUCTION

Sorting algorithms have been studied and optimized since the dawn of computer science because of their importance and because of how frequently sorting is required. It has always been of great importance that sorting algorithms run efficiently but also that they are not so complicated as to be excessively difficult to write. There are numerous sorting algorithms, each tailored to a best use case but three sorts in particular have found extensive use as general case sorting algorithms, insertion sort, merge sort and quicksort. Each is suited to its own ideal use case but all three are useful in most situations and in fact can even be used to further optimize each other in some cases. In this paper we explore the general runtime complexity of these three sorts and explore the possibility of realizing a runtime speed up by implementing a hybrid search algorithm which takes advantage of the speed of insertion sort for small cases.

# METHODOLOGY

The three sorting algorithms were implemented using Java 8 and run on a Macbook Pro with 16 GB of RAM and a quad-core Intel i7 CPU. Being that the tests were run on a non-dedicated machine there was an expectation of anomalous data appearing in the results which will be discussed shortly. The sorting algorithms were testing using arrays of integers that ranged in size from 0 elements to 25,400 elements in a non-continuous fashion. A total of 8350 arrays were sorted by each algorithm where each distinct size had 10 random arrays which were then averaged before being graphed so as to normalize the data. The raw data was written to a CSV file where each entry consisted of the number of elements in the array that was sorted and the time it took to sort the array. The CSV files were then read into Microsoft Excel where the averaging was done and then graphed.

SAMPLE: Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*Conference’10*, Month 1–2, 2010, City, State, Country.

Copyright 2010 ACM 1-58113-000-0/00/0010 …$15.00.

DOI: http://dx.doi.org/10.1145/12345.67890

# RESULTS AND FIGURES

|  |  |  |
| --- | --- | --- |
| Runtimes in miliseconds [1]  N | O(n^2) | O(nlogn) |
| 1 | 0.0001909 | 0.0001243 |
| 10 | 0.0013155 | 0.0023877 |
| 100 | 0.0024522 | 0.0063739 |
| 1000 | 0.109373 | 0.0522794 |
| 10,000 | 7.4720752 | 0.5570327 |

**Figure 1. Graph of the entire data set obtained. It can be clearly seen that insertion sort is in fact O(n2) while merge sort and quicksort are both almost linear in runtime complexity.**

**Figure 2. A close up of the above graph in Figure 1 showing the behavior of insertion sort, merge sort, and quicksort when the input size is relatively small.** **Insertion sort behaves somewhat erratically in this data set and does not exceed the runtime of merge sort and quicksort until an input size of approximately 450 items.**

# DISCUSSION

## Runtime Complexity of Each Algorithm

The runtime complexity of each sorting algorithm is the piece of data of greatest interest. Upon graphing the data, it is clearly seen that insertion sort has a runtime complexity of O(n2) but does run faster in the smaller case generally than either merge sort or quicksort. Merge sort and quick sort are both seen to have a runtime complexity of O(n log(n)) which is partially known from the implementation of the algorithms and is then supported by the data graphs.

## Improvements to be Made to Merge Sort and Quicksort

Based upon the data collected here it does not appear that either merge sort or quicksort would benefit substantially from being optimized to run insertion sort instead of merge sort or quick sort when the list size is below a threshold value. The threshold value for the data collected here would be approximately 450 items and the speed up would be minimal.

It is certainly possible that the lack of significant difference is due to the CPU opting to perform other tasks simultaneously or that it is a flaw in the way testing was done and further investigation is the only way to determine which is the case.

# REFERENCES

1. Weiss, M. A. Data Structures and Algorithm Analysis in Java 3rd Ed. Pearson, 2012