Alexander Swanson

April 4, 2018

Professor Lisa Dion

CS 124 – Data Structures and Algorithms

Project 4

*Project 4 Report*

Dear CS 124 Grader,

For over seven months now, I have been working on a project fundamentally aimed at evaluating and influencing online public opinion. There are many platforms to work on this problem – *Facebook*, *Twitter*, *Instagram*, etc. However, attempting to access all of these platforms immediately is not a practical approach. Thus, working with the social, information aggregation forum *Reddit*, a large amount of data has been collected to begin evaluating public opinion. A portion of this data has been selected for use with this project in order to complete fundamental analysis.

In this third portion of work for *CS 124 – Data Structures and Algorithms*,

a class was designed to represent a row of data from the original dataset. The data is composed of general metadata of Reddit comments posted to submissions discussing world news events. An instance of this class, “*RedditElement*,” includes the following fields:

*ID*

The unique ID of the Reddit *comment* object.

*Subreddit Name: Prefixed*

The common identifier of a categorical forum on Reddit.

*Ups*

The comment’s total resulting *upvotes*.

*Downs*

The comment’s total resulting *downvotes*.

*Score*

The overall score, a function of the comment’s upvotes and downvotes.

*Category*

The general topic category of the comment’s text body.

*Sentiment Score*

The score of the sentiment, rational to the sentiment magnitude. That is, the type of opinion expressed in the comment text body.

*Sentiment Magnitude*

The magnitude of the sentiment, rational to the sentiment score. That is, the weight of the opinion expressed in the comment text body.

*Created*

The UTC unit measure of the date the comment was created.

This newer project has been completed with the dataset used for the *Project 1* – however, some data fields determined to be irrelevant were removed.

Beside functionality for setting and getting data field values for an instance of RedditElement, the class also allows for the formatted output of all the statistics of the object with the *info*() method. Moreover, using the *sentiment\_score* data field, the *measure\_happiness*() function performs a sum of every instance of *sentiment\_score* for each RedditElement object and calculates the arithmetic mean, yielding a naïve measurement of the “happiness.” These calculations are not, however, the most complex for the program. Such is, rather, the data input function, which exhibits a complexity of *O*(*N*2), as it contains a nested *while* loop.

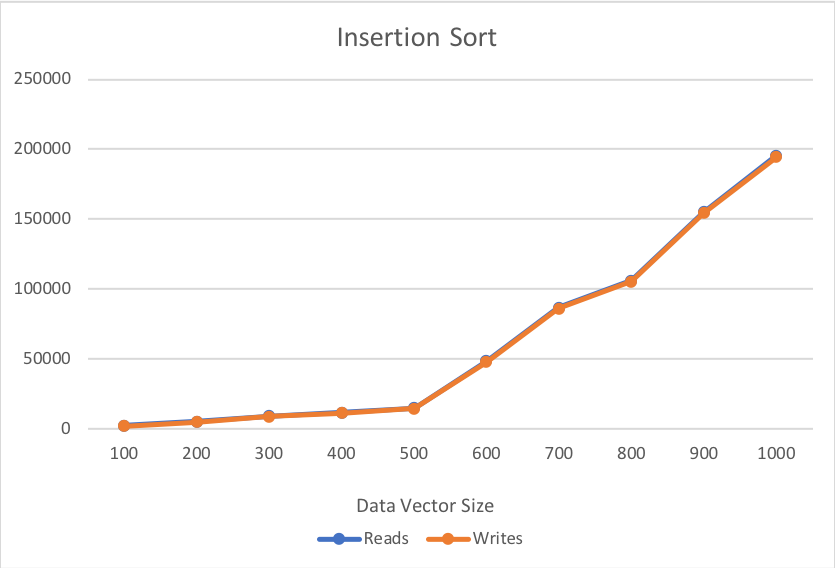
In this continuation of the venture, we work to sort the data with different commonplace sorting algorithms:

* *Bubble Sort ­*– *O*(*N*2)
* *Insertion Sort – O*(*N*)
* *Quick Sort – O*(*N* log *N*)
* *Heap Sort – O*(*N* log *N*)
* *Shell Sort – O*(*N* log *N*)

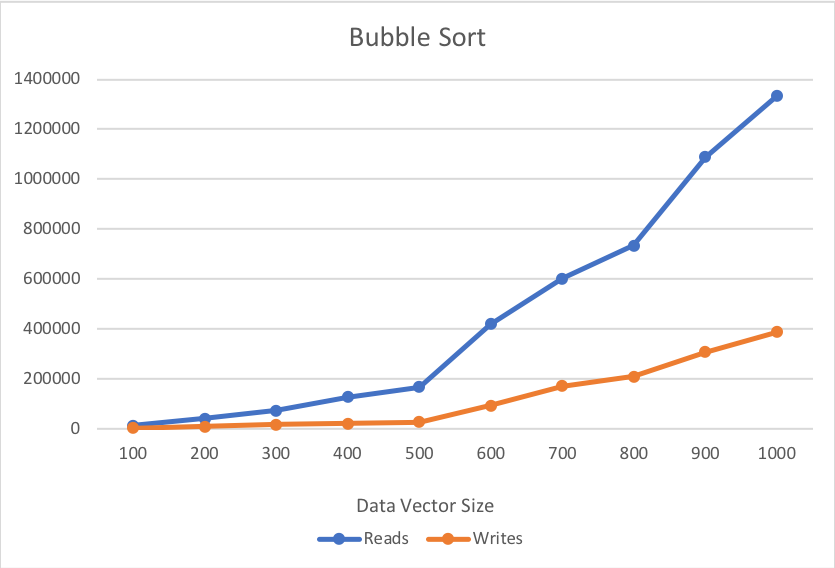
We sort the data using these algorithms and record the amount of *Reads* and *Writes* each algorithm generates. For this project, a Read is defined by the retrieval of data from the vector of RedditElements for use and a Write is defined by the assignment of data into the vector. These two measures provide a naïve but robust analysis of the efficiency of these sorting algorithms. For RedditElements, each object’s *ID* field is compared for alphanumeric equality – that is, the sum of the ASCII and numeric values of an *ID*. Moreover, we include the application of a modified Bubble Sort, which works with the *Score* data field of RedditElements.

Furthermore, in order to observe how each algorithm operates when given increasingly large amounts of data, we run the sorting operations for vectors of the same data, truncated to sizes up to indexes 100, 200, 300… 1000.

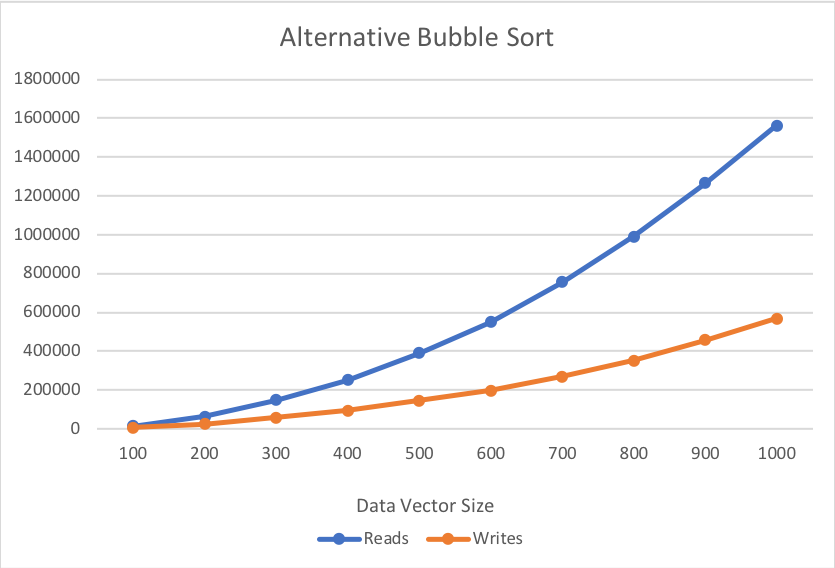
We observe these Read and Write statistics:



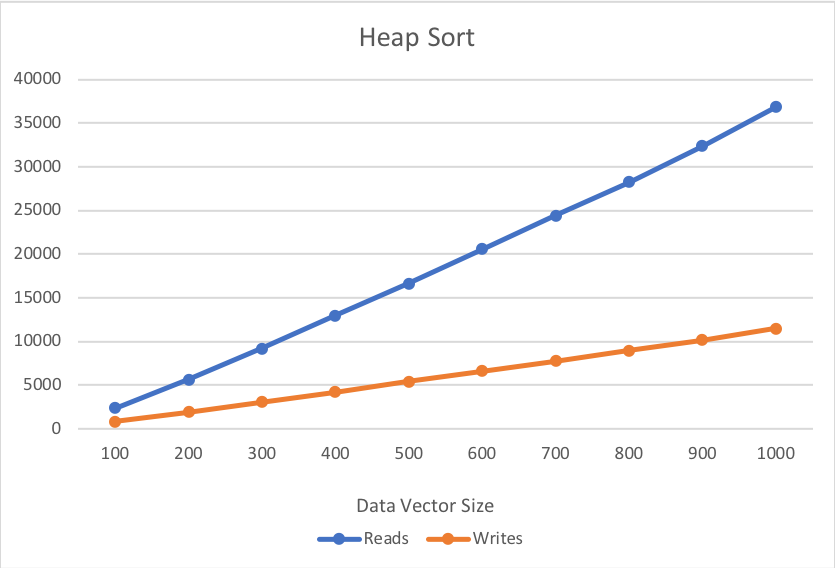
Beginning with Insertion Sort, one may be surprised to observe that the curves overlap. In comparison to the other sorting algorithms, Insertion Sort’s Read and Write numbers are not exceedingly high. However, they are nowhere near the most efficient.



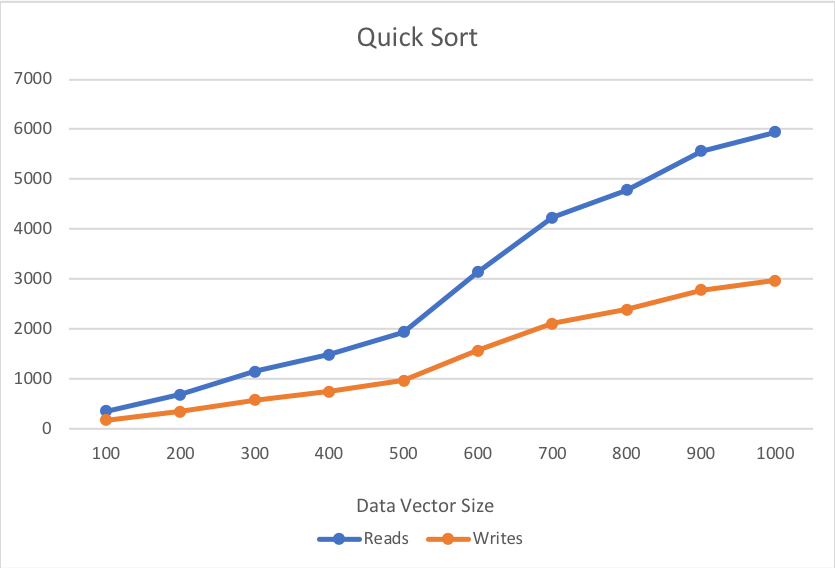
Bubble Sort, a very well-known algorithm, is incredibly inefficient – as can be seen. Not only are the rates of change of Reads and Writes inconsistent throughout the curve, they are also the second greatest.



Observing the alternative Bubble Sort, that which has made use of a different comparable data field of RedditElements, shows a remarkably smooth curve, as its amounts of Reads and Writes have significantly increased.

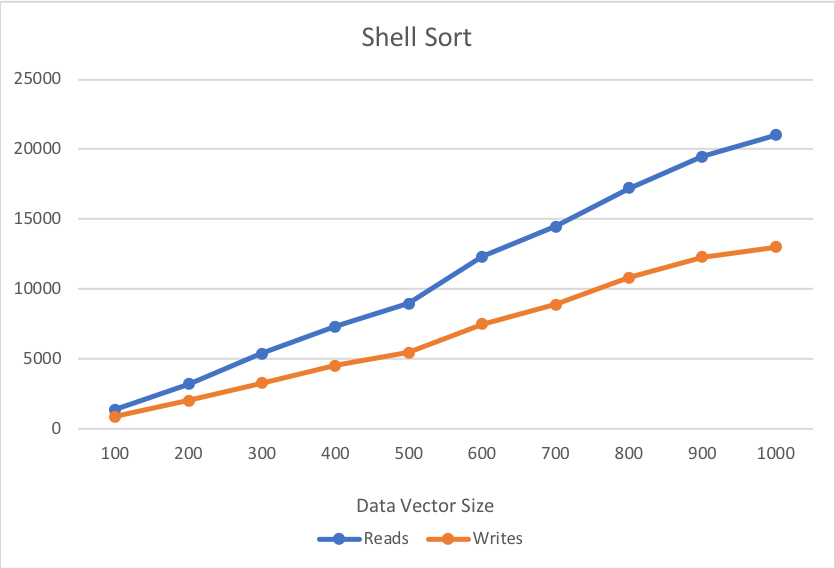


Heap Sort, a very common algorithm for small collections of data, clearly begins to lack reliability as the amount of data is increased. We do indeed observe a remarkable consistency of change in Reads and Writes, but it is clear that this sorting algorithm – along with Insertion and Bubble Sort, would not be ideal for any operations conducted on quantities of data exceeding the hundreds.



Quick Sort, as its name implies, is the most efficient of our sorting algorithms. It’s

fantastic implementation of “divide and conquer” calculations yields a remarkably computationally efficient operation.



Finally, we see that Shell Sort, another stable algorithm, operates within the average level of Reads and Writes. Moreover, for small amounts of data, this is certainly a comparable sorting algorithm.

We observe that Quick Sort is remarkably more efficient than any other sorting algorithm and find proof of its preference.

Questions

1. *If you need to sort a contacts list on a mobile app, which sorting algorithm(s) would you use and why?*
   1. In this case, it would probably be good to choose Shell Sort, as its levels of Reads and Writes are not incredibly high, and its implementation is very simple. This algorithm can be programmed within a function of only several lines, making it an excellent candidate for operation on a limited computational power device.
2. *What about if you need to sort a database of 20 million client files that are stored in a datacenter in the cloud?*
   1. With such a task, it would be best to choose Quick Sort – without a doubt. This is the best performing algorithm for large amounts of data. Moreover, the possession of such a quantity of data indicates that this database’s computational power is reliable enough to support such large divide-and-conquer algorithms that Quick Sort would yield.

References

*C++* Code for Insertion, Quick, Shell and Heap Sort:

Weiss, Mark Allen. *Data Structures and Algorithms in C++* (*Fourth Edition*). Florida

International University. 2014. <http://users.cs.fiu.edu/~weiss/dsaa_c++4/code/>

C++ Code for Bubble Sort:

Dion, Lisa. University of Vermont. 2018.

<https://bb.uvm.edu/bbcswebdav/pid-2501085-dt-content-rid-14203848_1/xid->14203848\_1