2XB3 Assignment 2

Justin Staples (staplejw)

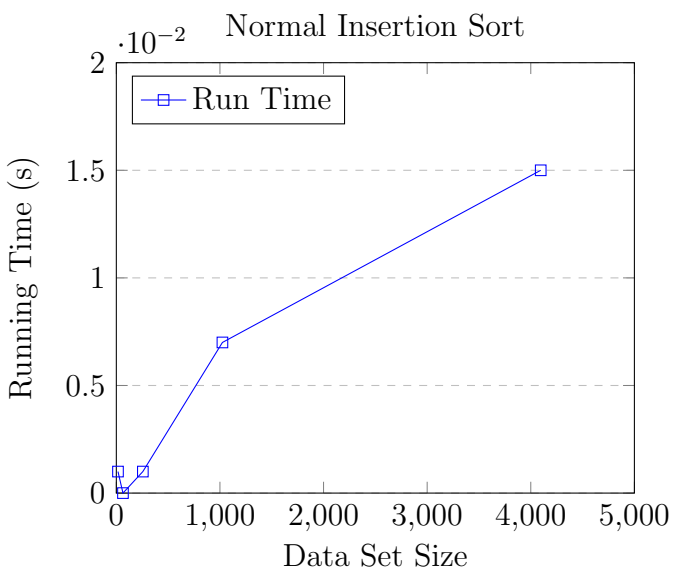
March 5, 2017

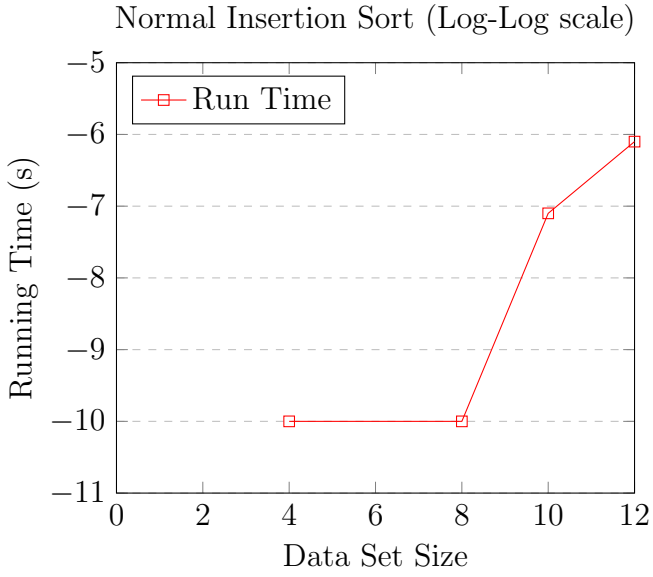
Contents

This report contains an analysis of a software engineering experiment that studies the behaviour of different sorting algorithms. An ADT (abstract data type) was implemented that stores information about the arrival time and processing time of certain CPU tasks or jobs. The idea was to sort these jobs based on their arrival and processing times to create a schedule or queue for the CPU to complete its tasks.

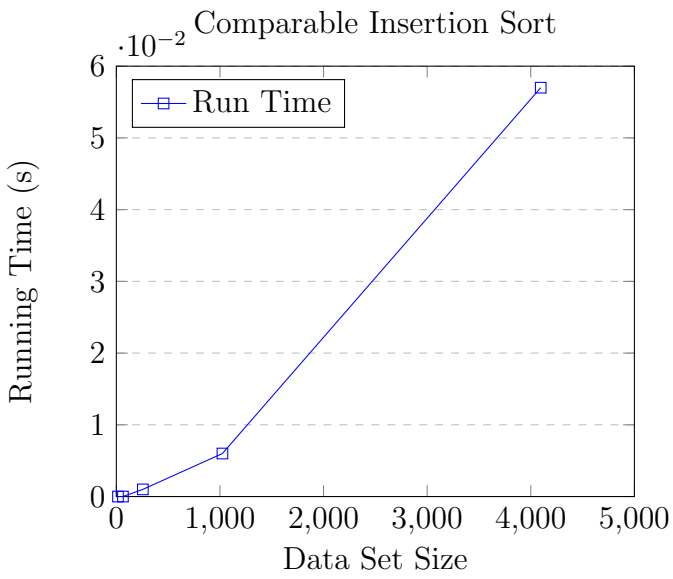
1 Graphical Results

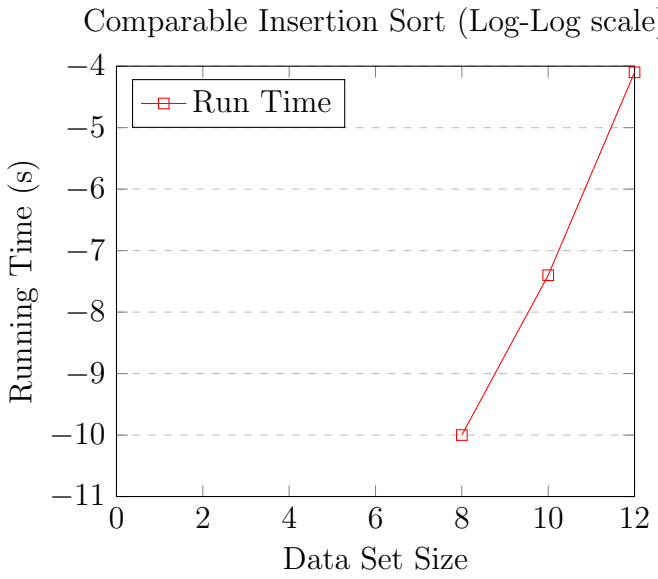
1.1 Normal Insertion Sort



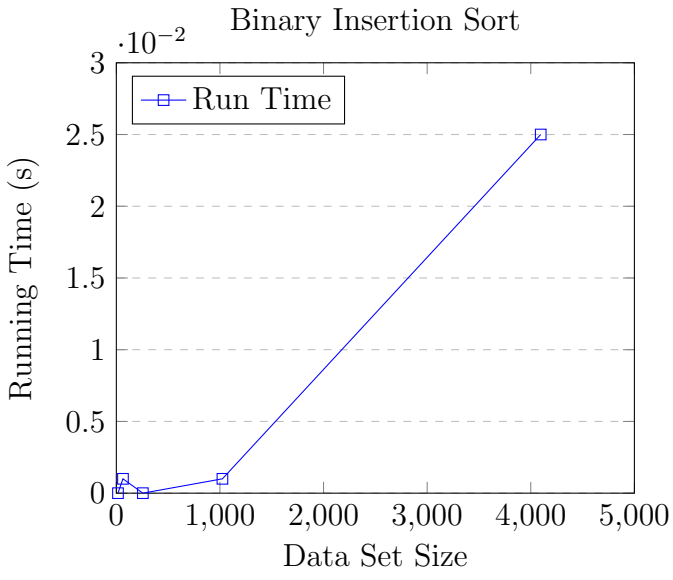


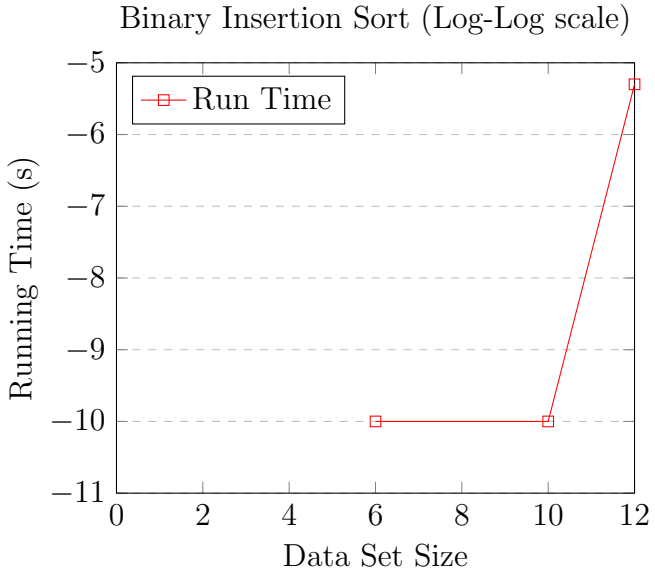
1.2 Comparable Insertion Sort



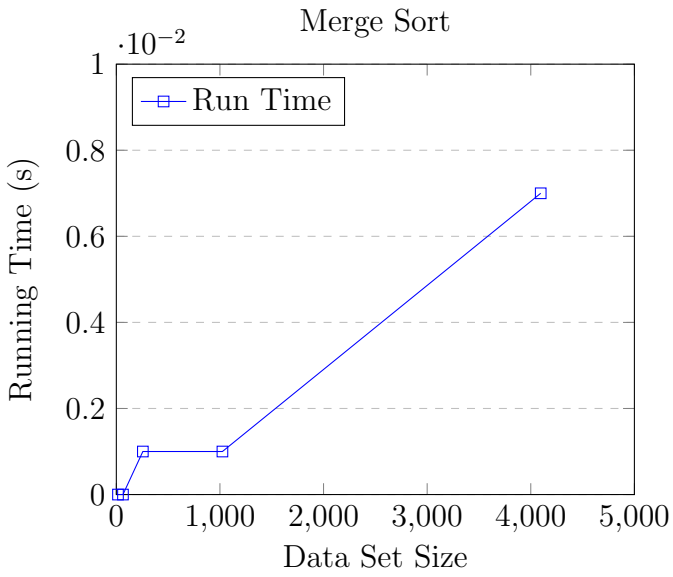


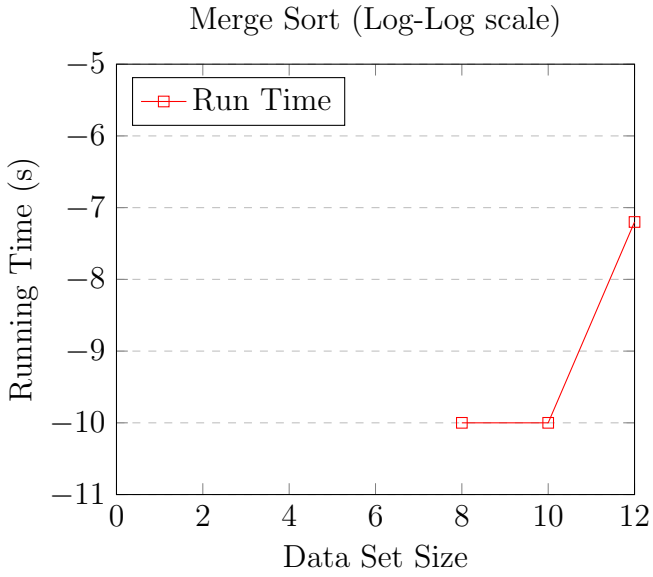
1.3 Binary Insertion Sort



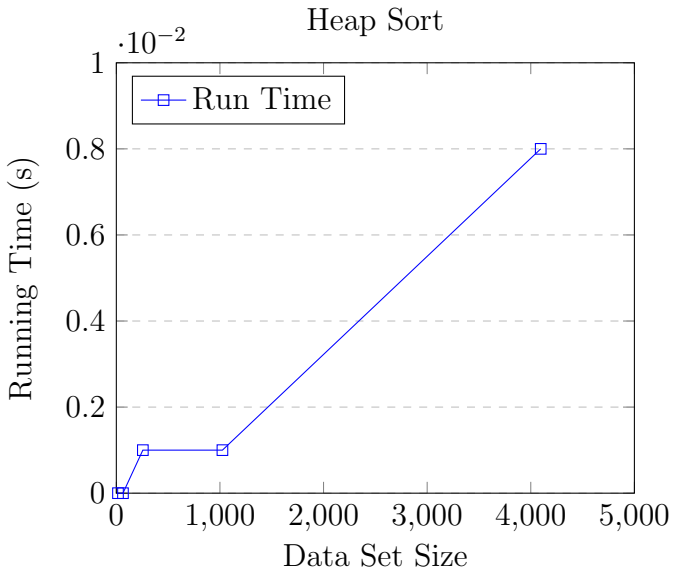


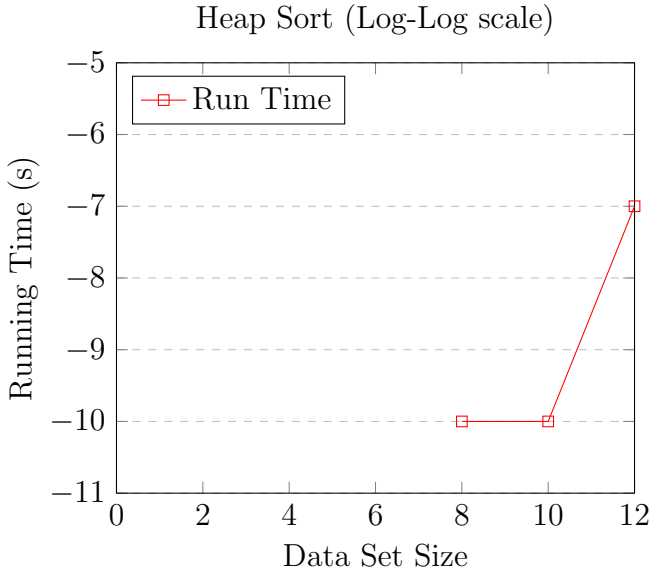
1.4 Merge Sort





1.5 Heap Sort





2 Hypotheses

2.1 Insertion Based Sorts

The insertion based sorting algorithms have implementations that used nested for loops in the code. Because of the nature of these algorithms, it can be suspected that the run time complexity will be in polynomial time. That is to say, the run time complexity will be of the form:

$$T(N) = aN^m$$

In the equation above, 'a' and 'm' are constants that can be solved for using the data. Taking the log (base 2) of both sides:

$$\log_2 T(N) = \log_2 aN^m$$

Applying logarithm rules:

$$\log_2 T(N) = m \log_2 N + \log_2 a$$

This reveals that the log-log plot of the data can be useful for calculating the exponent 'm'. This equation says that the slope of the log-log plot will determine the exponent.

In this experiment, the data points that represent the small data sets are not very helpful, as they are inaccurate and inconsistent. The rightmost data points will be used to determine the slope of the log-log plot. Once the exponent is determined, one of these sample data points will be substituted into the equation to find the value of 'a'. Here are the slopes of the log-log plots for the three insertion sorts, as well as the calcuation for the constant 'a':

- Normal Insertion Sort:

$$m = \frac{-6.1 - (-7.1)}{12 - 10} = 0.5$$
$$T(N) = aN^{0.5}$$
$$0.015 = 4096^{0.5}a$$
$$a = 0.00023$$
$$T(N) = 0.00023N^{0.5}$$

- Comparable Insertion Sort:

$$m = \frac{-4.1 - (-7.4)}{12 - 10} = 1.65$$
$$T(N) = aN^{1.65}$$
$$0.057 = 4096^{1.65}a$$
$$a = 0.000000062$$
$$T(N) = 0.000000062N^{1.65}$$

- Binary Insertion Sort:

$$m = \frac{-5.3 - (-10)}{12 - 10} = 2.35$$
$$T(N) = aN^{2.35}$$
$$0.025 = 4096^{2.35}a$$
$$a = 0.000000000081$$
$$T(N) = 0.000000000081N^{2.35}$$

2.2 Merge and Heap Sort

These algorithms have a different nature than the insertion based ones. They are more efficient. Merge sort works by breaking the problem up into smaller and smaller sub-problems (cutting the problem in half). This is called the divide and conquer technique and it usually leads to a perfomance increase based on $\log N$. Heap sort is like this as well. It works by constructing a binary heap, and sorting the elements by removing the maximum every iteration and then correcting the heap. Because the heap is a binary tree, traversing it will cost at most $\log N$. Because of this quality of these algorithms, instead of predicting these to have a polynomial time, a different approximation can be made of the form:

$$T(N) = aN \log N$$

A data point for each algorithm can be selected to find an approximation for the constant 'a':

- Merge Sort:

$$0.007 = 4096 \log_2 4096 * a$$

$$a = 0.00000014$$

$$T(N) = 0.00000014 N \log N$$

- Heap Sort:

$$0.008 = 4096 \log_2 4096 * a$$

$$a = 0.00000016$$

$$T(N) = 0.00000016 N \log N$$

3 Testing the Hypotheses

By plugging in values of $N = 16382, N = 65536, N = 262144$ into the hypotheses that were formulated, predictions can be made about the run time with larger data sets. The comparison of these predictions with the actual meaured times for the larger data sets are summarized in the tables below.

A comprehensive print-out of all of the times for all data set sizes and algorithms can be found in the file named a2_out.txt.

- Normal Insertion Sort:

Data Set Size	Predicted Time (s)	Actual Time (s)
16384	0.029	0.172
65536	0.059	4.561
262144	0.118	182.130

- Comparable Insertion Sort:

Data Set Size	Predicted Time (s)	Actual Time (s)
16384	0.557	0.529
65536	5.490	18.997
262144	54.073	488.568

- Binary Insertion Sort:

Data Set Size	Predicted Time (s)	Actual Time (s)
16384	0.649	0.529
65536	16.874	18.997
262144	438.585	488.568

- Merge Sort:

Data Set Size	Predicted Time (s)	Actual Time (s)
16384	0.032	0.014
65536	0.147	0.030
262144	0.661	0.117

- Heap Sort:

Data Set Size	Predicted Time (s)	Actual Time (s)
16384	0.037	0.017
65536	0.168	0.030
262144	0.755	0.170

4 Discussion

Overall, it can be said that the predictions for the experiments were wildly off. This is due to a few factors. Firstly, the data sets that were used to do the sorting for the first few tests were extremely small. This creates a higher chance of stopwatch error coming into play. The fact that it took most of the tests less than one millisecond to run and the granularity of the stopwatch is only one millisecond made it very hard to get accurate results. These results were used to define the hypothesis and so that is why the models were innacurate. For a future experiment, it would be beneficial to create the models with tests that took more time so that the stopwatch error was not so huge. The stopwatch measurements were extremely inconsistent with small values, sometimes reporting 0.000

times and others 0.001. It was seemingly ranodm for the small tests. It is also tough to make a model off of just particular set of tests. The data is randomly assigned, so perhaps it would be better to run many tests, create an average of results, and then use the average to model.

Upon further investigation of the run time results, it is determined that the algorithms studied for these implementations fall into two major categories in terms of run time complexity: quadratic and linearithmic. All of the implementations based on insertion sort are have quadratic run time complexity proportional to N^2. This can be seen very clearly for normal insert and comparable insert. The core of their implementations are nested for loops, which means their effects will be multiplicative. As the size of the data grows, so does the size of each of the inner loops. For binary insertion sort, it might appear on the surface that this algorithm would be linearithmic as well because it finds the insertion point using binary search which can happen in time proportional to $\log N$. However, each iteration of the loop still requires most of the array to be shifted over, which is very costly. This means that this implementation is also quadratic in run time.

The tilde approximations for these implementations of merge sort and heap would be linearithmic (proportional to $N \log N$). For merge sort, this is because it uses the divide and conquer technique, which splits the problem into smaller and smaller sub-problems. Then it applies the merge step, which can happen in linear time. Heapsort is like this because on each iteration, it performs a sink operation, which can happen in no longer than logarithmic time.

For sorting applications that have large data sets (over 10000 - 50000), it is most practical to use the linearithmic algorithms presented in this report. They offer much greater performance.