Empirical Analysis of Sorting Algorithms

Rachael Johnson

March 5th, 2016

Table Of Contents

1.0 Introduction

Sorting Algorithms are vital to everyday operations as a computer science. This project attempts to complete an overview and analysis of six different sorting algorithms. The following algorithms were selected for this project analysis:

1. Selection Sort
2. Insertion Sort
3. Merge Sort
4. Bucket Sort
5. Quicksort
6. HeapSort
7. Java Sort (Java proprietary)

2.0 Algorithm Descriptions

2.1 Selection Sort

Selection sort is one of the easiest sorts to write, and is often the very first sort that is learned. Selection sort has a fairly bad time efficiency of $\Theta(n^2)$. Selection sort works by starting at the first element in the array, going through the rest of the array to see the minimum value, and then swapping that with the first element. It then moves on to the second element in the array, then compares it to the rest of the array that is past the second element, finds the minimum value, and then swaps it with the second array element. Selection sort will always perform the same number of comparisons, and its best case complexity class is the same as its worse case complexity class.

2.2 Insertion Sort

Insertion is an easy to implement sort, often requiring only a few lines of code. Unfortunately, as the experimental results show, although this is a simple sort, it is not very efficient, and the larger the array size n gets, the worse the sorting efficiency is. The calculated Big Theta complexity class for insertion Sort is $\Theta(n^2)$. Insertion sort is efficient for small arrays of integers, and is also rather efficient for a *mostly* sorted array. In fact, the time complexity for a mostly sorted array is $\Theta(nk)$, where k is an element that is k places away from the sorted position. In addition, insertion sort is a stable sort, in that when there are two elements of equal value, the insertion sort sorts them in their original order. Insertion sort is similar to Selection Sort, however Insertion sort varies the amount of comparisons it has, depending on the order of the original array. Best case performance is O(n), when the array is already sorted, and average case is also O(n). Worst case performance is $O(n^2)$, and is when the array is in reverse sorted order.

This is because insertion sort works by starting at the beginning of an array. It then proceeds to the second element of the array, and then compares it to the first element. If the second element is less than the first element, it swaps the two and because it is at the beginning of the list, its job is done. Next, it goes to the third element in the array. It compares the third element to the second. If it is less than the second element, it swaps the two. It then compares the now second element to the first element. If it is less then it swaps the two, otherwise it stays put.

2.3 Merge Sort

Merge sort is a divide and conquer sorting algorithm that has several different incarnations. Merge Sort works by continuing to dividing the list into subsections, over and over again, until it reaches two elements, then it sorts those two, goes up a subdivision, sorts those, and so on and so forth. Merge sort is a comparison sort, and useful if you know nothing of your input. Merge sort has a constant run time of Θ(nlog(n)) for worse case, best case, and average case.

2.4 Bucket Sort

Bucket sort is usually considered a non-comparison sort. It relies on knowing something about the inputted array. If all the elements are the same, this would result in a terrible input for bucket sort, in fact it would be a worst case scenario. The buckets are created using the minimum and some maximum value. The bigger the bucket, the slower the sorting. If there is only one bucket, then the run time efficiency is the same as insertion sort. This means that the worst case efficiency for bucket sort is Θ(n^2). This also means that the average case performance is Θ(n+k). Best case is performance is Ω(n+k).

2.5 Quicksort

Quicksort, when written correctly, can be 3 to 5 times faster than Merge Sort. This project uses a basic version of quicksort, without most of the optimizations. Quicksort, like merge sort, is a comparison sort. Although it is quite often faster than Merge Sort (and heapsort), quick sort has a worst case complexity of O(n^2). This worst case can come up if the pivot chosen is the first or the last element in the array. This can occur when randomly choosing a pivot, and not optimizing the pivot selection process. But because this worst case is almost never encountered, quicksort is often used instead of merge sort or heapsort. The version that I am using avoids using random pivots and instead uses an average of the middle, left, and right elements. Quicksort works by choosing a pivot, and then comparing all the values in the array to the pivot, putting greater values to the left, and lesser values to the right. Next, quicksort is recursively called on the lesser and greater parts of the array, and continues the pattern until the entire array is sorted.

2.6 Heapsort

Heapsort is a comparison sort that is a similar but better version of selection sort. It varies from selection sort because it makes use of a binary heap. In the first step of the sort, a heap is constructed using the inputted array. Next, the largest element is removed from the root of the heap and placed in the the array. Then the heap is sifted, moving the largest element to the root of the heap, and the process is repeated until the whole array is sorted. Heapsort has a consistent performance of O(nlog(n)) for worst case, best case, and average case. Heapsort can compete with quicksort because quicksort has a worst case performance of O(n^2), although that rare case most likely won’t occur, depending on the implementation and array input. Although Heapsort is not a stable sort, it is can be advantageous to use instead of merge sort for systems where memory is an issue, because Heapsort uses O(1) space complexity.

2.7 Java Sort (Java Proprietary)

The Java array sort is the language proprietary sort that I used. The Java sort uses a modified TimSort from Python. It is a stable, adaptive, and iterative merge sort [1]. It is generally much faster than O(nlog(n)), although that is its complexity class. The best case complexity class is O(n). The best case is when the input array is mostly sorted.

3.0 Experimental Procedure

3.1 Input

The input was a Java arrayList of integers, randomly created. The range of values of the array were held constant during each increment of array size. In addition, the range of values was increased with subsequent tests. All data was recorded in tables. The results seen in the graph are the averages of 3 subsequent runs per setting.

3.2 Running Time

Running time was carefully monitored using the nanoTime function included in Java. The time was recorded in nanoseconds. In each run each sort was timed and then the time was stored and displayed. The time was recorded from the start of the algorithm running to the end of the algorithm running. No other processes were timed.
Results were run several times, and then an average was used of the recorded times to produce the graphs. All data is available to view upon request.

3.3 Programming

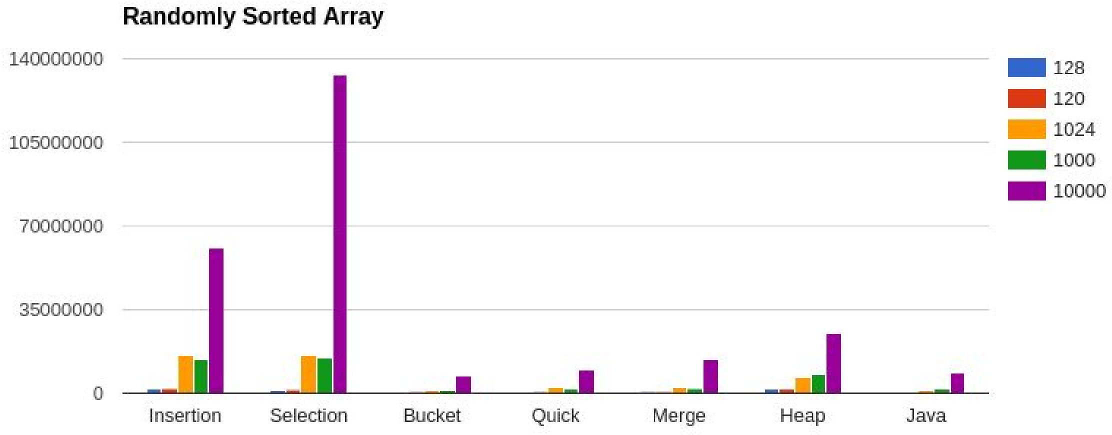
All code was written in Java, and the code was compiled and run using IntelliJ software. Code was created using references from the internet and adopted to use arrayLists.

4.0 Experimental Results

4.1 Overview

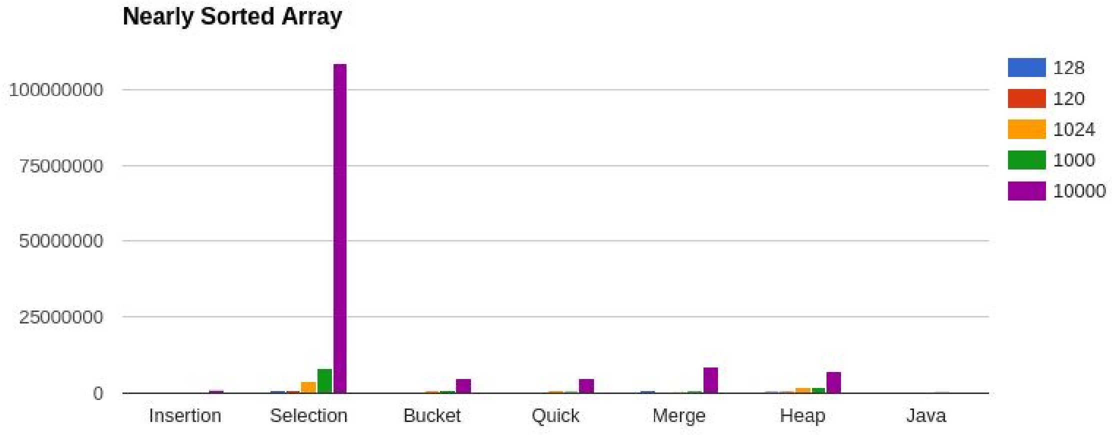
In these charts, three different types of array lists were sorted. One was a standard randomly sorted array, one was an nearly sorted array in which every 5th element was swapped, and the third was a reverse sorted array.

[1] https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#sort(java.lang.Object[])



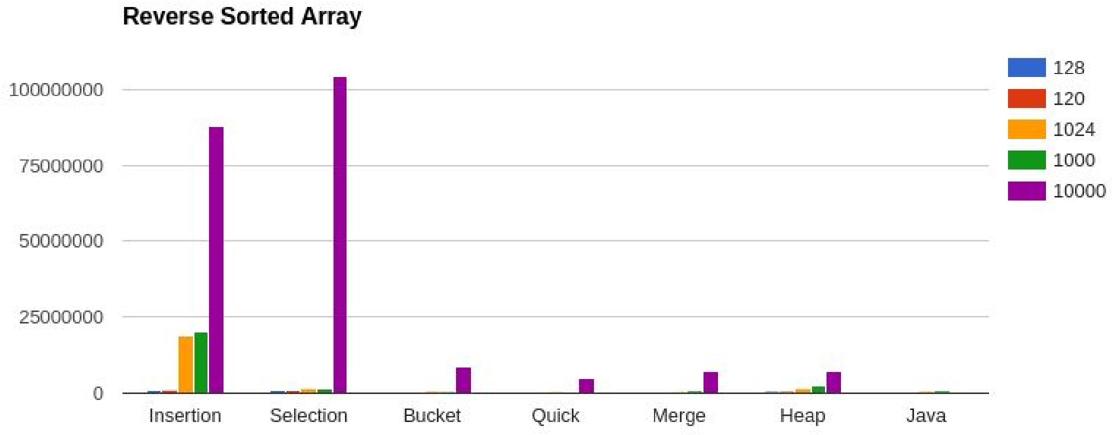
Randomly Sorted Array

Sort Runtime in Nano Seconds, for Integers from 0 to 100



Nearly Sorted Array

Sort Runtime in Nano Seconds, for Integers from 0 to 100



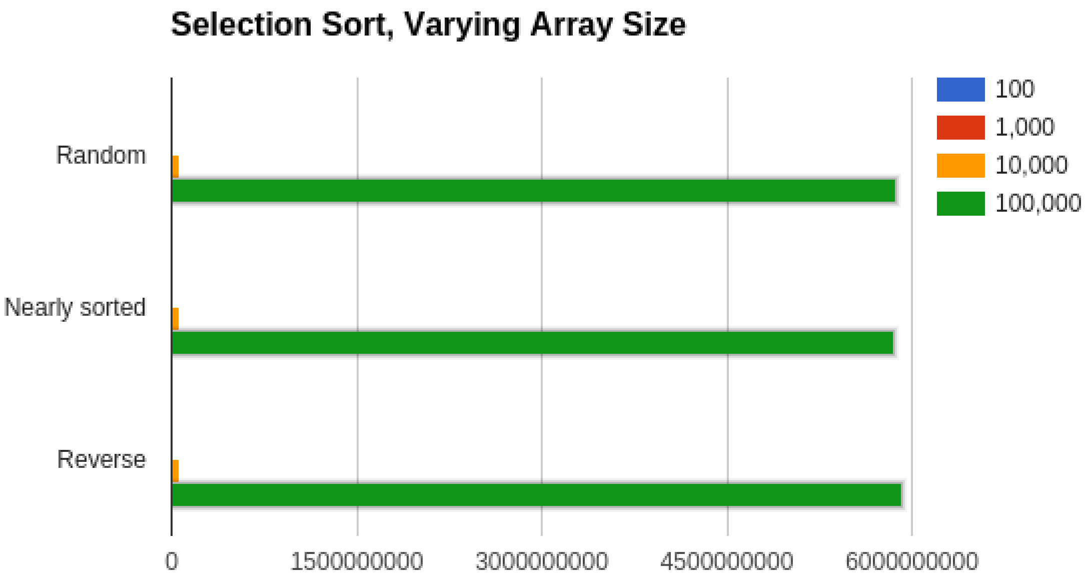
Reverse Sorted Array

Sort Runtime in Nano Seconds, for Integers from 0 to 100

4.2 Experimental Results per Sorting Algorithm

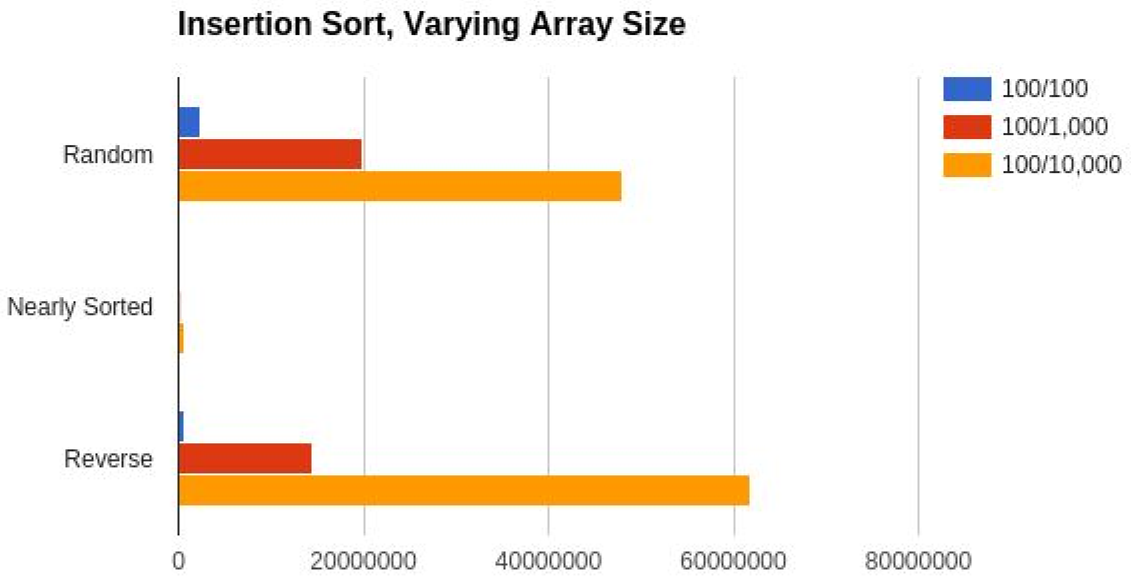
4.2.1 Selection Sort

After testing, selection sort consistently performed the worst compared to all the sorts tested. The testing proved that selection sort took the same amount of time, no matter the sorting of the array. This is good and bad. It means that the sort won't take longer than expected for a given input, but it also means it won't perform better than expected for a given input. This sort is clearly n^2 running time. Unfortunately, that means that because of testing time, tests were only run up to array size 100,000.



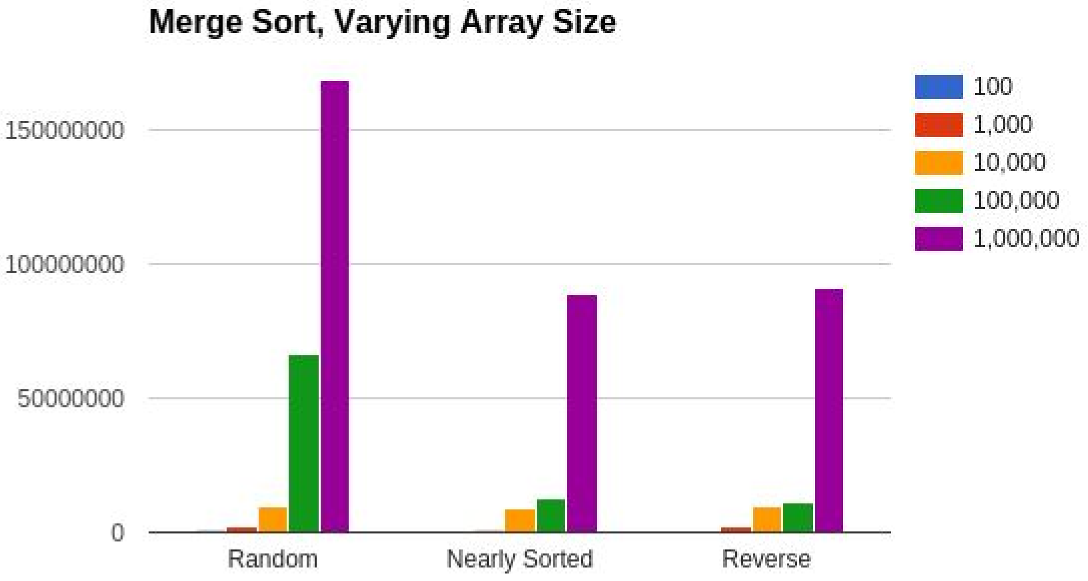
4.2.2 Insertion Sort

Consistently the slowest on average for the randomly sorted array and the reverse sorted array, the Insertion Sort performance was pretty abysmal, increasing exponentially as the size of array n increased. The only area where insertion sort was better than most of the competition is when the array was mostly sorted. According to my tests, there was no significant difference in the run time of the sort when the integer range was increased from 100 to 1000. Because of the slow times, tests were only run to array size 100,000. Because of trouble displaying all the values, only array sizes up to 10,000 are included in the graph.



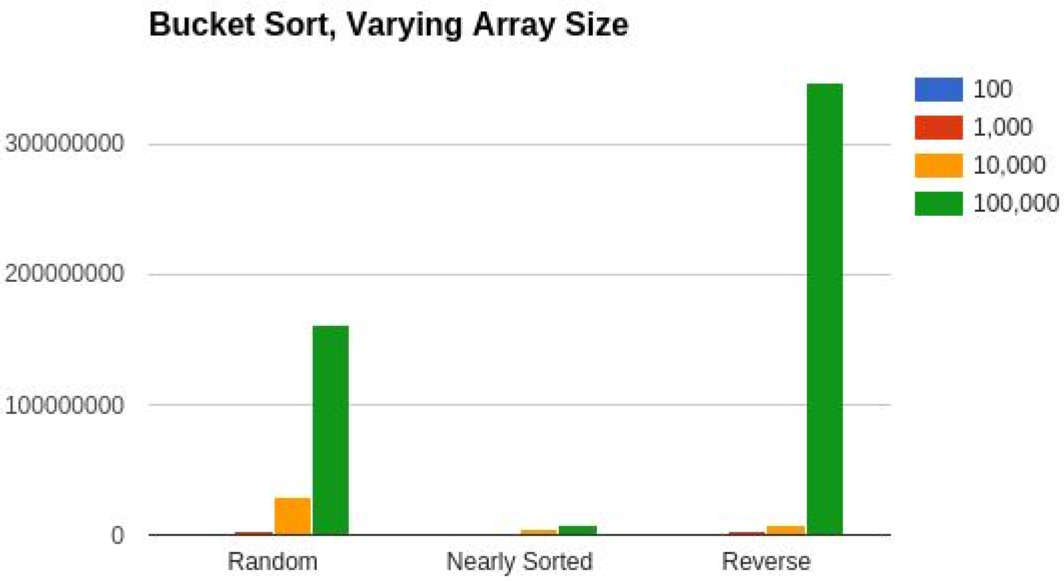
4.2.3 Merge Sort

Merge sort was a good sort, consistently performing well. Although not quite as fast as quicksort, it was a close competitor. Although Merge sort is supposed to have a constant run time of O(nlog(n)), variations in the testing machine may have contributed to varying final results.



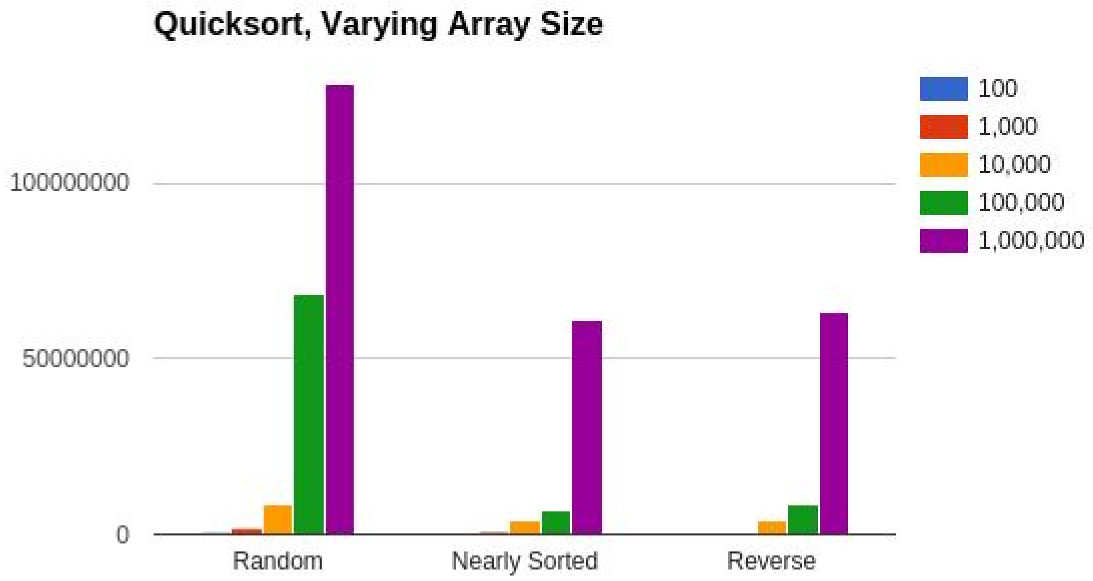
4.2.4 Bucket Sort

Bucket sort performed very well for a mostly sorted list, very similar to insertion sort. This is reasonable to expect, because bucket sort uses insertion sort once the array has been subdivided into “buckets”. Reverse array performed badly, which is also to be expected, because that is also the worst case for insertion sort. Bucket sort is definitely an improvement to insertion sort, but there are cases where it performs the same.



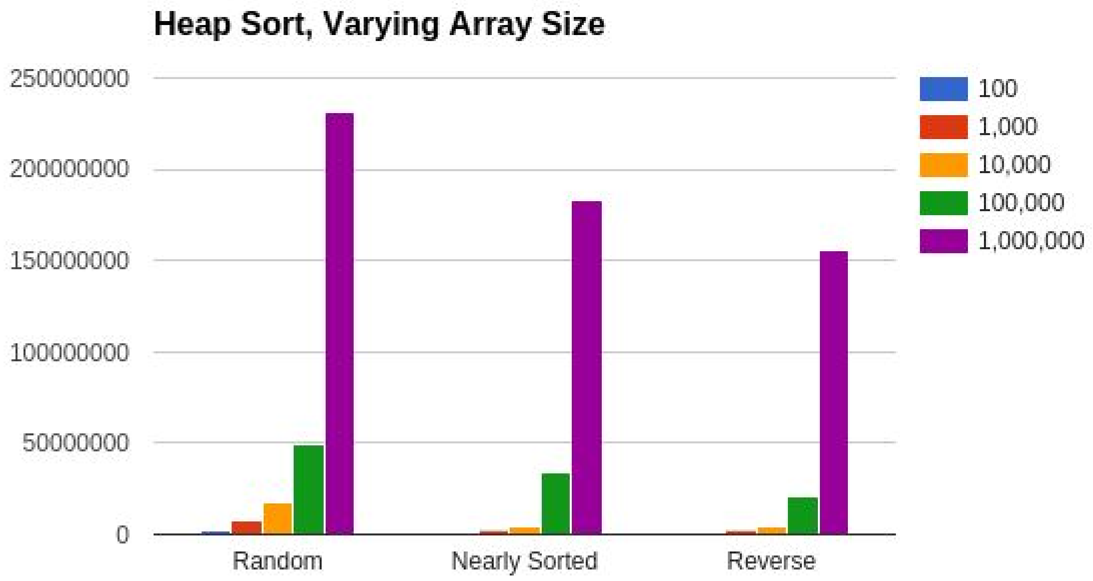
4.2.5 Quicksort

Quicksort turned out to be a great sort. It was fast and efficient, and performed well on nearly sorted and reverse sorted arrays. It performed well on randomly sorted data as well. Because of the implementation used, there wasn’t a way to produce a worst case scenario of O(n^2).



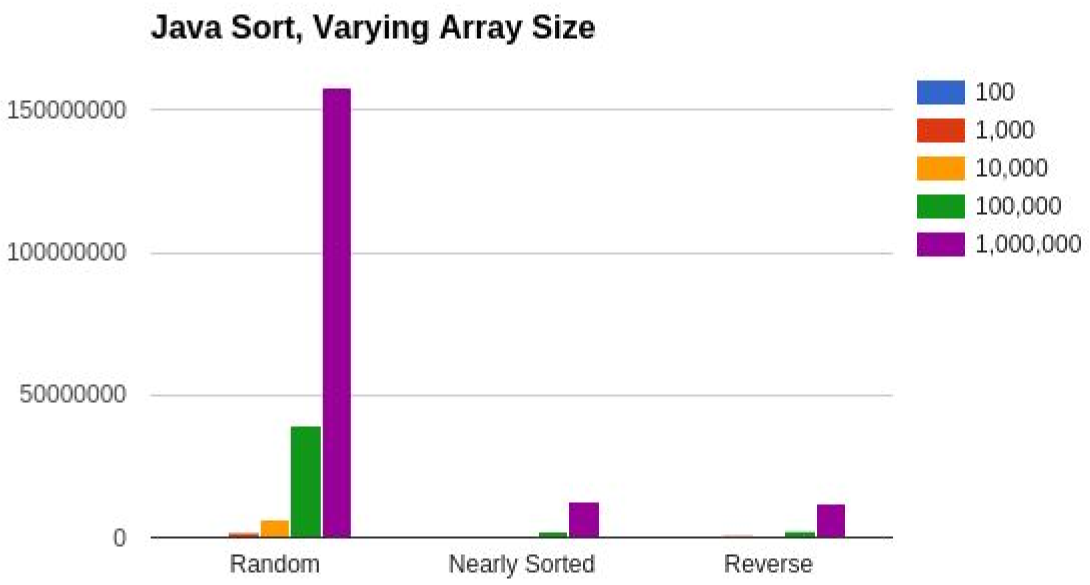
4.2.6 Heapsort

Heapsort performed fairly well, and certainly better than selection sort, which it is based on. It performed about the same for Random, Nearly Sorted, and Reverse arrays. It performed worse than Merge Sort, which was expected. Because of the near constant run time, this was a fairly reliable sort with no unexpected results.



4.2.7 Java Sort

The Java sort performed phenomenally. It performed similar to merge sort in some cases, but in most cases it performed much better. It performed extremely well with a reverse sorted list and with a nearly sorted list. Interestingly, Java Sort was slower than quicksort on the individual tests when it came to sorting a Random Array. Quicksort was much slower for nearly sorted and reverse sorted arrays, however. In the group averages, Java Sort and Quicksort are almost the same for a randomly sorted array. Java Sort performed almost the same as Merge Sort for sorting a random array. This is interesting considering the Java Sort is based on the Timsort, which in turn is a modified Merge Sort. Although the sort times were similar for a random array, the run time for a nearly sorted array and for a reverse array were much faster for Java Sort compared to Merge Sort.



5.0 Conclusions

In summary, best sort out of the seven tested is the Java Sort. If it isn't possible or feasible to use Java sort, then the next best recommendation is to use either bucket sort or quick sort. If the data is known, and it is not in reverse order, then use a bucket sort. Otherwise, use a quick sort. For sorting extremely small arrays where recursion overhead is unwanted, a simple insertion sort is recommended. For sorting something with small memory availability, a heap sort is useful. The different sorts tested had varying strengths and weaknesses, and the one to use depends on the situation.