Time Analysis of 3 Sorting Algorithms
Bubble Sort, Merge Sort, and Binary Tree Sort

Introduction:

Sorting is a fundamental part of Computing Science. You are given a list of \( N \) numbers and you are told “sort them.” How would you do that? Well, there are many different types of sorting algorithms; swap/exchange sorts, selection sorts, insertion sorts, and “divide-and-conquer” sorts. Each type has a different runtime. Runtime is an important concept when dealing with algorithms because the algorithm that runs the fastest and takes up the least amount of space possible is usually considered the most efficient algorithm. The problem is, there isn’t a best universal algorithm, each one works well for their own situation. This project looked to identify the strengths and weaknesses of 3 specific sorting algorithms; bubble, merge, and binary tree.

Methods:

To test these 3 algorithms, a data type needed to be chosen to sort. The simplest type to sort is the primitive type \( \text{int} \). Basically, the program would sort arrays of different sizes containing only whole numbers. This makes comparison easier because time complexities due to varying data types are avoided. On the other hand, a generic program would have been better to analyze because it should work for each comparable data type; Character, Long, Short, String, Float, Integer, Byte, Double and any other custom made objects that extend the Comparable interface. I avoided generics because the code became increasingly complex as more and more data types had to be accounted for. There came a point when I decided that generics were getting in the way of testing the actual algorithms and were therefore becoming a “third-variable” which would just skew the results.

To test, the program ran a for loop with loop variable \( s \) that starts at 100 and went up to 5000 in increments of 100. This represented the variable of size. Inside that loop an \( \text{int} \) array is created with size \( s \). It is then populated by putting numbers 0 to \( s-1 \), in sequential order. This represents a completely sorted list with a randomness of 0. Next, another for loop is run with a loop variable of \( j \) that starts at 0 and increases by 1 to 100. This is the randomness loop. A method called Random is called which randomly swaps numbers in the list. The amount of swaps is based on the desired randomness of the list, this is represented by \( j \). Now, each algorithm is tested individually. For each, the algorithm is run 5 times over the list to be sorted to get an average time. The average time is then recorded and the next algorithm is processed. This process is repeated for each randomness level and each size.

The results were recorded directly into a Microsoft Excel spreadsheet using a FileOutputSteam. To set this up, the amount of randomness (0 to 100) was printed along the first row of each spreadsheet starting from the second column. After each size iteration, the size
Paul Weiss

would be printed in the leftmost column and each result would be printed in each succeeding column under its randomness value. This made it easy to create a surface plot to see the differences in time for each algorithm. This also eliminated the need for me to manipulate the data in any way, shape, or form. Additionally, this way makes it easy to see the results just using the spreadsheet as a table. Also, it eliminated the need for a multitude of data structures to hold each time result which cut down on code complexity and work that the computer needed to do.

Results:

Bubble Sort:

The time complexity of a Bubble Sort in its worst case is \( O(n^2) \) and in its best case \( O(n) \) if the list is completely sorted. For Bubble Sort, an exponential surface plot emerged (see Figure 1). This is to be expected since the time complexity has \( n \) being squared as the size increases. For a list of size 4900 and a randomness of 100 the time was 125 which can be considered the “worst case” since it is the most complex list to sort. It is also interesting to note that when randomness was 0, the average time was 0 ms. This is because Bubble Sort’s best case is \( O(n) \) because it just has to run through the list once to see if it is sorted, and if it is it terminates.

Figure 1. Bubble Sort
Merge Sort:

The time complexity for Merge Sort is always $O(n \log n)$ because it uses the “divide-and-conquer” technique of sorting. So basically, the time complexity does not change so all the results should fall into a pattern. Also, the results should increase in what appears to be an exponential pattern but it actually does not. It increases in a logarithmic pattern. The results showed that the time increased gradually from 0 to about 36 ms as compared to the Bubble Sort which increases exponentially from 0 to around 125 ms. This shows that the Merge Sort is faster than the Bubble Sort with one exception. When the list is almost perfectly sorted (randomness approaches 0%) the Bubble Sort works better because if it doesn’t need to do any swaps the algorithm terminates, whereas Merge Sort is required to break itself into parts every time, regardless of the original state of the list. This is the one major flaw with Merge Sort, its best and worst case time complexities are exactly the same. Therefore, it will work the same no matter what. This provides stability but it can also become inefficient very quickly compared to other sorting algorithms with a best case of $O(n)$, such as Bubble Sort.

Figure 2. Merge Sort
Paul Weiss

Binary Search Tree Sort:

For the Binary Search Tree Sort (BST Sort), the methods were changed. After the original test an error came to light in the form of a StackOverflowError once the size hit around 4300. This happened when adding nodes to the tree as well as when the tree was traversed to get the sorted list out of it. This is because the recursive calls build up in a stack and when it reaches its maximum size, it will throw a StackOverflowError. To avoid this, the Binary Search Tree class had to be rewritten to accommodate for this. To accomplish this, the insertion method had to be modified so that it didn’t make any recursive calls. Likewise, the iterate method in the Driver class had to be changed to a non-recursive inorder iteration. Also, after the original test it was concluded that any random level above around 15% was negligible because of the structure of the tree and its time complexity. So the test was run again with the new BST insertion and iteration implementations. Since the likelihood of getting a StackOverflowError was greatly diminished, the size of the list could be pushed beyond 5,000 to 10,000 elements, also the randomness only went up to 15% to eliminate the unneeded data. Figure 3 is a graphical representation of the data. This shows that after about 1% randomness, the algorithm works with almost no time at all, 0-3ms. Before 1%, the time grows logarithmically from 0ms to about 250ms when the list approaches 10,000 elements.

Figure 3. Binary Search Tree Sort
Paul Weiss

**Discussion:**

The results of this study are quite interesting. It is clear from the way the Bubble Sort is implemented it would do well with a list at or close to 0% randomness, but after that point the time needed to sort grows exponentially. For the Merge Sort, the implementation forces it to run at $O(n \log n)$ always. Therefore, it runs better than Bubble Sort only after about 2% randomness which is when Bubble Sort starts to grow exponentially. The BST Sort works well only after about 1%. But, the BST Sort runs faster than the Merge Sort after 1% for any size.

The reason, I believe, that the BST Sort runs so fast is because the comparisons are quick and it doesn’t require that much computing power to do them. Also, there is no resizing or copying of arrays and the use of a temporary array is not needed. The time is so high at 0% and 1% because the tree basically turns into a linked list and the sorting is done over that. This becomes the worst case for the tree, $O(n^2)$, and is very inefficient.

The results of this study show that for a list that is mostly sorted with a randomness percentage of less than 2% a Bubble Sort is the most efficient and effective sorting algorithm of the three tested. After 2% a Binary Search Tree Sort is the most efficient and effective by far, again of the 3 presented in this study. This does not mean that a BST Sort is the most efficient sort in all instances, but in the case of sorting and array of int’s with a controlled degree of randomness, the BST Sort works the best out of the given algorithms.

**Questions:**

- Is there a best sort for any situation?
- Will the time change for different data types?
- Would the Binary Search Tree Sort work less efficiently if the data type was changed to a comparable data type? (Integer, Double, etc…)
- Will using generic programming change the runtime of the sorts?
- Will running these sorts on different platforms (LINUX, MAC OS, Windows, etc…) change the runtime?