Anthony Joo

CSS 342

Eyal Arian

Lab 3

Analyzing the Speed of Sorting Algorithms in Varying Scenarios

**Introduction:**

One of the most important tools the use of computers provides us with is the ability to sort a list of values automatically and relatively quickly given the proper computing power. The code that powers the ability to sort a list is termed a sorting algorithm and it is important to every programmer that they know how to use it and when to use certain sorting algorithm. The speed of a given sorting algorithm is very important because once you begin working with datasets upwards of a million the algorithm can be the difference between days of computation versus minutes.

**Purpose:**

The purpose of the study is to investigate the varying complexities and speeds of 6 different algorithms: Bubble Sort, Insertion Sort, Merge Sort, Iterative Merge Sort, Quick Sort, and Shell Sort. The time complexities of Bubble Sort and Insertion Sort are on average O(n^2). As for Quick Sort, Merge Sort, and Iterative Merge Sort, the average time complexity is O(n \* log(n)). Lastly, the time complexity of Shell Sort is on average O(n \* (log(n))^2). Something to keep in mind is that these are just average time complexities, and these differ in certain “best case” and “worst case” scenarios. These average time complexities are just rough estimate as to what type of data curve to look for in our datasets. I will implement different functions in the driver file that set up the original array as an unordered, ordered, reversed, partially-ordered set.

**Method:**

1. Implement each sorting algorithm.
2. In the driver file, create an unordered array of size 10 to pass into the sorting algorithm.
3. Record the time (in ms) it takes to run the sorting algorithm on this array.
4. Repeat two more times in order to take the average time and better ensure the accuracy of the measurement.
5. Repeat Steps 2-4 with array sizes 100, 1000, 10000, 25000, 50000, 100000, and 1000000.
6. Repeat Steps 2-5 with an ordered array, reverse-ordered array, and partially-sorted array.
7. Then graph the results of each different type of array passed in (ordered, unordered, reverse-ordered array, and partially-sorted array.
8. Analyze and compare the results.

**Results (Empty cells are where sorting could not be completed in appropriate amount of time)**:

Unordered



*Bubble:* This graph illustrates the effectiveness of bubble sort when it is at its average speed. The Bubble Sort curve is exponential and thus illustrates an O(n^2) time complexity.

*Insertion:* This graph shows the average time complexity of Insertion Sort. The Insertion Sort curve is exponential thus showing it is O(n^2) time complexity.

*Merge:* This graph shows that Merge Sort is a nearly linear line with a slight upward trend. This would also describe n \* log(n) and so the time complexity of Merge Sort here is average at O(n \* log(n)).

*Iter. Merge:* This graph also shows that Iterative Merge Sort is slightly linear with the upward trend. This describes the n \* log(n) shape which is the average complexity of this sort (O(n \* log(n)).

*Quick:* Quick Sort also appears to be its average complexity at O(n \* log(n)).

*Shell:* Among the non-O(n^2) time complexity algorithms Shell Sort appears to have the steepest curve compared to Quick, Iterative Merge, or Merge Sort. Thus, the shape of this curve resembles a time complexity of O(n \* (log(n))^2).

Ordered



*Bubble:* Bubble sort seems to be linear showing that with an ordered array the bubble sort has the best-case scenario. The time complexity is O(n). It is best case because the list is already ordered meaning

*Insertion:* Insertion sort seems to be linear showing that Insertion sort is the best case scenario with an ordered array. The Time Complexity becomes O(n).

*Merge:* Merge is again nearly linear with an upward trend and thus the Time Complexity is O(n \* log(n)).

*Iter. Merge:* The graph of this sorting algorithm is linear with a lean upward and so the time complexity is O(n \* log(n)).

*Quick:* The graph of quick sort is now exponential showing that its time complexity is O(n^2). This is the worst-case scenario for Quick Sort and it is this bad because the array is already sorted. A solution to this would be to pick a random pivot.

*Shell:* In this case the shape of the shell sort graph is similar to merge and iterative merge sort which indicates the Shell Sort has a time complexity of O(n \* log(n)). This is a best-case scenario and occurs because the list is ordered.

Reverse-Ordered



*Bubble:* Bubble Sort seems to be growing exponentially meaning that the time complexity is O(n^2).

*Insertion:* Insertion Sort seems to be growing exponentially as well. Thus the time complexity of this sort is O(n^2).

*Merge:* Merge sort seems to be growing linearly and has an upward curve which resembles the curve of a time complexity of O(n \* log(n)).

*Iter. Merge:* Iterative merge sort seems to grow linearly with a slight upward curve. This resembles a time complexity of O(n \* log(n)).

*Quick:* Quick sort seems to be growing exponentially which is unlike the usual for this sorting method. Thus, the time complexity if the worst case O(n^2).

*Shell:* Shell sort appears to be growing in tangent with the growth of merge sort or iterative merge sort. This would leave me to believe the time complexity of Shell Sort in this case is O(n \* log(n)).

Partially-Ordered



*Bubble:* This sort grows exponentially meaning its time complexity is O(n^2).

*Insertion:* This sort grows exponentially showing that its time complexity if O(n^2).

*Merge:* Merge sort displays a linear growth with an upward curve illustrating its O(n \* log(n)) time complexity.

*Iter. Merge:* This sort also displays linear growth with an upward curve showing that its time complexity is O(n \* log(n)).

*Quick:* Quick sort is exponential in this graph showing that it is in its worst-case scenario for partially-ordered lists. It is O(n^2).

*Shell:* Shell Sort seems to grow at a greater rate than Iterative Merge Sort or Merge Sort showing that its time complexity is likely O(n \* (log(n))^2).

Conclusion:



We can see from this table that the optimal sorting algorithm for an unordered list is either merge, iterative merge, quick, or shell sort. As for an ordered list the optimal sorting algorithm would be insertion sort. For a reverse-ordered list the optimal sorting method would be merge, iterative merge, or shell sort. For a partially sorted list the optimal sorting algorithm is merge and iterative merge sort.



On average the best sorting method Merge or Iterative Merge sort for two reasons. One is that their time complexity of O(n \* log(n)) is relatively fast and reliable. More importantly, these are very consistent algorithms as they do not have a best or worst case and are always at O(n \* log(n)) no matter the circumstance.