Sorting has become a subfield of computer science. The ability to take a set of data and arrange it into a specific sequence of ascending or descending values has created a multitude of algorithms and analyses. Our challenge was to create a class from which we could run at least 6 different sorting algorithms coded in Java and compare the performance of each on a given set of data.

Our team created a class that contained 6 different sorting routines. We also created a separate class to handle the acquisition of the data to be sorted, the invoking of the Sort class to access the individual sort methods, and the recording of the time that it would take to perform each task. We also wanted to be able to provide different sizes of data sets to be able to compare sort times for varying values of n.

Our task was not to code each sorting routine from scratch, although we did so with the Bubble sort and our own Force Push sort. Our task was, instead, to assemble the sorts from other documented sources such as the textbook (in the case of the Quick Sort) or the Internet as noted in each method. Once the sorting routines were gathered and assembled, a methodology had to be developed to provide fair and accurate testing and recording.

Each sorting algorithm was structured to receive a reference to an array of type Point. The class Point was defined in a separate point.java class file and consisted of two double values representing the (x,y) Cartesian graphing pair. The array would be sorted in ascending order based on the x coordinate. No return type was required since the reference passed as an actual parameter was a reference to the array from the invoking class. Thus, the original array was the one being sorted.

System time was recorded in nanoseconds before and after each sorting routine was called. The difference of the two times was logged as the time that it took that sort algorithm to perform a sort on the array with n elements.

Our methodology for testing the different sorting algorithms was to create a sort testing class that would have an array of integers representing the different sizes of arrays (different size n arrays) to be sorted. This gave us great flexibility in deciding the different sizes of the data sets. Given a file with 100,000 test elements, this array of integers was used to create sub arrays from the full list.

Each sort was invoked given an array of sizes varying from 10 elements to 100,000 elements. Each sort was timed and that time was recorded. Because run time of a program can be influenced by other tasks the computer may be performing at the same time, we ran each test 10 times and took the average time for each test case.

We noticed that there were anomalies with the results when the test cases used n < 10, regardless of the sort routine called. In fact, the first test of each routine resulted in times that were higher than the second run of the same routine even when the same data was being used. To counter this anomaly, we disregarded the results of the first few tests (i.e. n=2,n=3,n=4,...n=9).

We were working on a hypothesis that an O(n^2) sort might have better real time performance on lower n elements that an O(nlogn) sort. Thus, we structured our test cases to cover more tests of n<100. However, our tests did not show a significant benefit for using a nonrecursive or O(n^2) sort for lower values of n.

These were the averages of the test results for n<=100, based on the random data from the original file:

<graph n<=100, random data>

<data table used for this graph>

The overall graph of the 7 sort routines that we tested is as follows:

<graph of n ?> 100 or n?> 1000, whichever gives a well defined curve to distiguish between n^2 and nlogn >

<full data table for random data>

We also ran test cases for the same test sizes of n using data that was already sorted. This graph shows the results of this test for best-case senario:

<graph of times for data already sorted>

<data table>

In addition, we ran the same tests for data that was in perfect reverse order. This graph shows the results of a worst-case senario:

<graph of times for data in sorted in reverse order>

<data table>

For this sorting task analysis, we were required to include 5 well know sorts: Bubble, Merge, Insertion, Selection, and Quick. We were also required to select a sixth sorting algorithm of our choice. Our team chose the Cocktail shaker sort, which derives it's name from a visualization of the way the sort "bubbles" the largest item to the right, then the smallest item to the left, then the next largest item to its place on the right, etc... until all items are sorted. Our perception of the Cocktail Shaker sort was that it was an improvement over the Bubble sort because it correctly identified and placed the largest and smallest two items in their proper place on each pass. Further examination proved our initial perception to be incorrect. By definition, the Cocktail Shaker sort is indeed a Bubble sort that makes two passes: the first pass, from left to right, bubbles the largest item to the right until it is in it's proper place; the second pass, from right to left, bubbles the smallest item to the left. This alternation of directions can provide better performance in some cases.

Our Bubble sort had a calculated runtime of \[8n^2 + 5n + 2 \] and the Cocktail Shaker sort had a runtime of <???????>. While both the Bubble and Cocktail Shaker sorts have a Big-O of n^2, the data show that the Cocktail Shaker out-performs the Bubble sort<??????>.

Given the misconception of the algorithm for the Cocktail Shaker sort, our team decided to try to implement what we thought was a common sense improvement on the Bubble sort. We decided to call our sort the "Force Push" sort, given the way that each pass forces an item left and an item right. If a Jedi were standing in the middle of the area to be sorted, he would "Push" both items to either end and have them land in their appropriate location. So we created, from scratch, an algorithm to make a pass through the data and locate, but not swap, the largest and smallest items at the same time. At the end of the first iteration, we would swap the smallest element to the leftmost side, and swap the largest element to the rightmost side. The next iteration would then not need to include the first or last element when searching for the remaining minimum and maximum value and repeating the process.

In our Force Push sort, the first iteration would take n passes, the second iteration would require (n-2) passes. Each iteration would decrease n by 2. Thus, the number of iterations would be n + (n-2) + (n-4) + (n-6)... + (n - (n-2) ) with only two element swaps per iteration. A Bubble sort would do n passes through the data on the first pass and n passes through the data on each successive iteration. Bubble sort can be improved to skip the area already sorted, thus decreasing the number of passes through the list of elements by 1 on each iteration. This means that a better coded Bubble sort would take n + (n-1) + (n-2) + (n-3) + ... + (n - (n-1) ) passes, with an average of n/2 swaps on the first iteration and (n-1)/2 swaps on the second iteration, and so on.

The Force Push sort is also in the Big-O of n^2. Our Force Push sort has a run-time formula of <????????>

(c)  Complexity: Provide the O(n), Θ(n), and the Ω(n)

Conclusion:

We found many different ideas and methods when analyzing different sort methods. Most of them are fascinating and many are ingenious. Data will always need to be sorted. We found that coding a simple sort routing can be quick and easy, as in the case of the Bubble sort. We did Bubble sort from memory and typed it straight into our class. We copied the Quick sort from the textbook and easily adapted it to our data type. Other sorts were copied straight from the Internet and adapted. But coding our Force Push sort required more testing and troubleshooting than anticipated.

Our first attempt at coding the Force Push sort used recursion. This proved to be memory intensive and caused overflow errors. After the testing was done on the method's algorithm, we easily converted the implementation to an iterative approach. As can be seen from the data, we bested the Bubble sort, even though we were still in the family of n^2.

Java has a built-in sort, and we wanted to implement a TimSort from the internet. But the ability to sort non-primitive data types using generics was beyond the scope of our knowledge. Implementing different sorts and having to adjust the coding to handle our Point data type for sorting whets the appetite to learn generics. The library of sorting methods that we have tested is already a valuable reference

Sixth Algorithm Selection. Provide a description about the sixth al- gorithm you selected to test. For this algorithm, provide:

(a)  Description about the algorithm: What is the input, output, and main features of the algorithm

(b)  Historic details, i.e., who invented, what year was invented, when was created, under what circumstances was invented (make sure you cite your references properly.

(c)  Complexity: Provide the O(n), Θ(n), and the Ω(n)

Performance of our approach. Here goes the theoretical study of your algorithm, i.e., run time analysis and description of the algorithm complexity (i.e., Big–O)

Testing strategy. Describe the test cases that you include for testing your five different algorithms. Enumerate your test cases, e.g., unordered data, sorted data, duplicates, reverse order data.

Experiments and Results. Provide the experiments and the results of each experiment. You need to be as detail as possible. Provide tables, graphs, or any kind of ideas to demonstrate the work you did. For this specific lab, you can include the theoretical results (i.e., the run time analysis) and the experimental results (the actual execution time).

Conclusion. Summarize your work done in the report, recap everything you learned and justify the results of your experiments.

Implementation Annexes In this section you must include your code for your algorithms and your programs including the test cases.