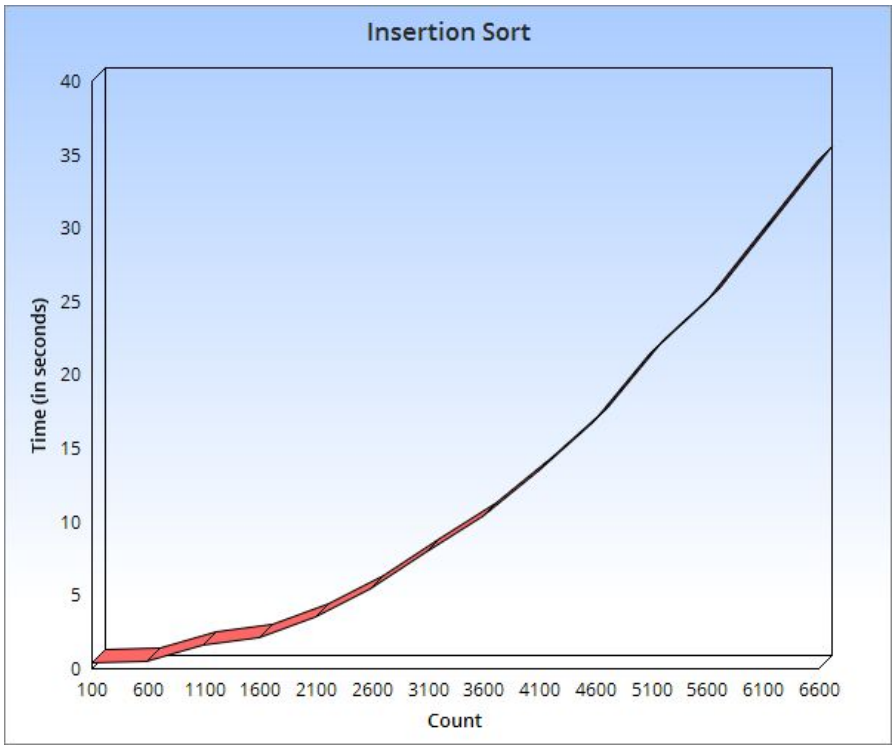
Tony Diep
u0934661
02/19/17
2420
Assignment 5: Sorting Analysis
Cindy Liao
I pledge that the work done here was my own and that I have learned how to write this program (*such that I could throw it out and restart and finish it in a timely manner*). I am not turning in any work that I cannot understand, describe, or recreate. Any sources (e.g., web sites) other than the lecture that I used to help write the code are cited in my work. When working with a partner, I have contributed an equal share and understand all the submitted work. Further, I have helped write all the code assigned as pair-programming and reviewed all code that was written separately.

(Tony Diep)

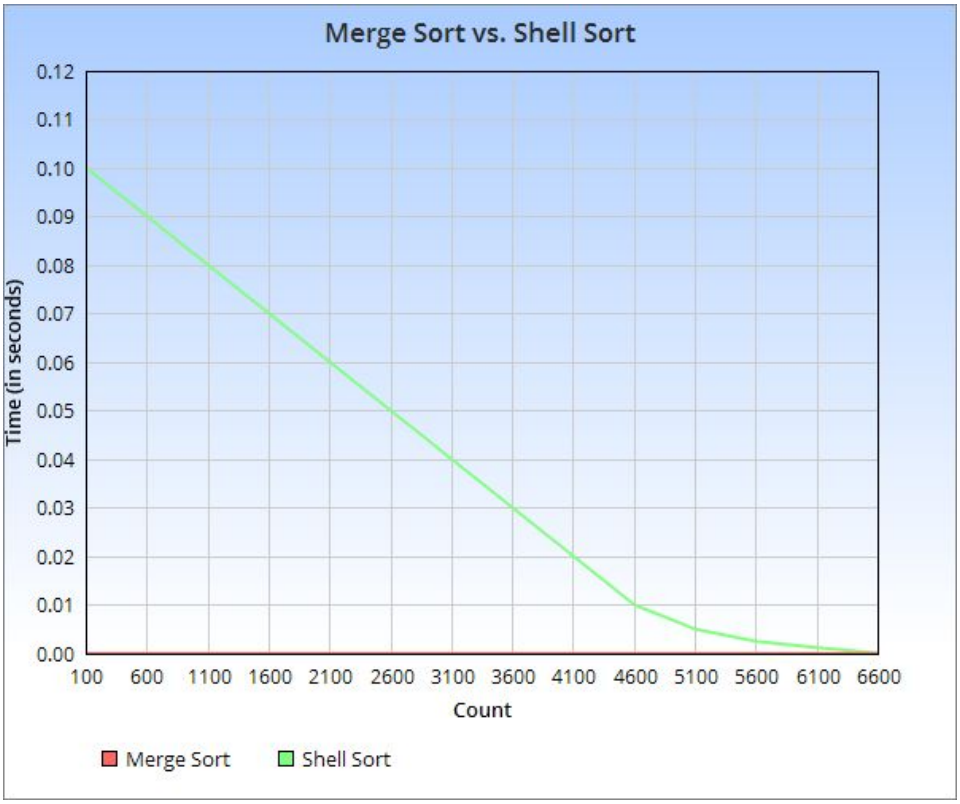
Sorting Analysis and The Graphs:

Insertion Sort



The graph above shows that insertion sort has a Big O Complexity of O(N^2). This is because of the implementation of insertion sort, in that there are two for loops. The first for loop iterates starting at index 0 which gets the single element used for comparing to the subsequent elements. The second for loop iterates backward starting at the array's length minus 1 and it's comparing each subsequent element to the current candidate element. Each loop has a complexity of O(N), and because there are two loops, the complexity multiplies to be O(N*N), which is indeed O(N^2). Insertion sort is only efficient for partly sorted arrays as well as small sized arrays and the worst case at a completely unsorted array.

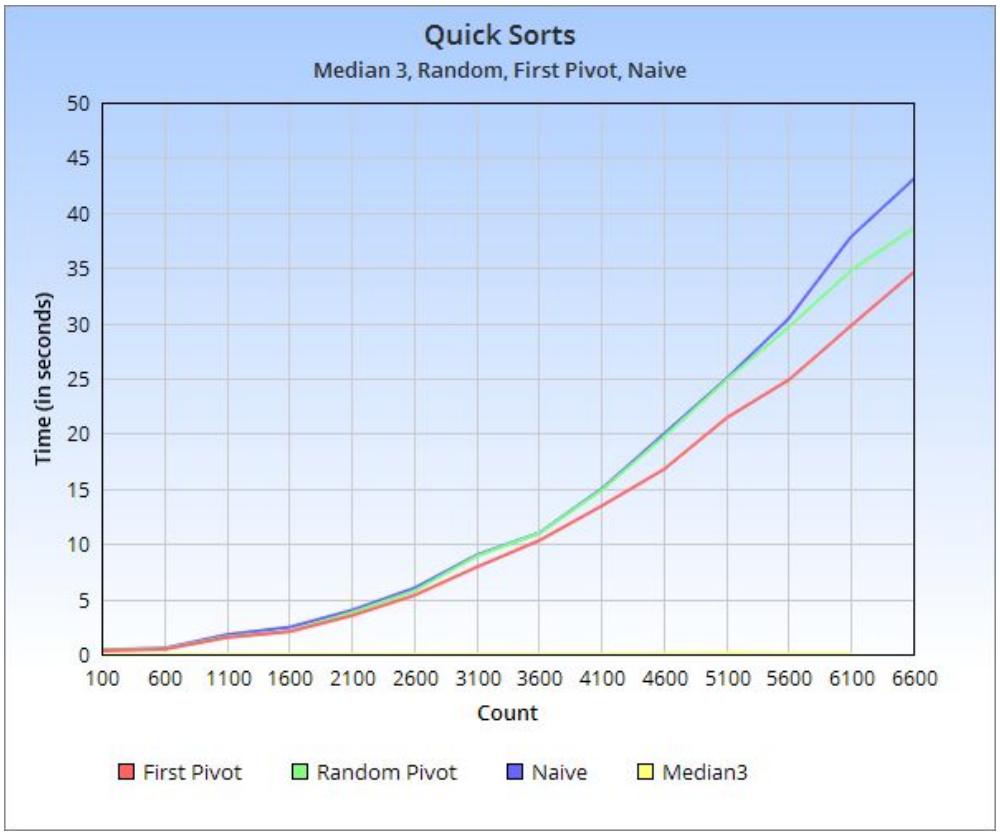
Merge Sort vs. Shell Sort



The graph above shows the comparison between merge sort and shell sort. Although both merge and shell sort both have a complexity of O(N Log N), the graph shows a less steeper curve for merge sort as opposed to shell sort. Merge sort involves recursive calls to itself because of dividing the given array into virtual arrays until a virtual array is at size 1. In other words, merge sort is a "divide and conquer" algorithm where the array is continuously divided into subarrays. Once all of the virtual arrays are divided until of size 1, then the conquering part comes in. The conquering is essentially the recombining of the smallest virtual arrays. Shell sort, on the other hand, is an insertion variant but it mitigates the issue of large gaps. Shell sort is not a "divide and conquer"

algorithm so the size of the array is still the same. A key difference in terms of implementation between the two algorithms is that merge sort does not implement any loops but rather uses recursion. Shell sort uses one for loop and a gap variable that decreases over time. However, for immense array sizes, merge sort may be very costly since recursion requires multiple calls of the method itself. Thus, shell sort may be the better implementation for immense array sizes.

Quick Sorts



The graph above shows the four quick sort algorithms: Quick Sort Median 3, Quick Sort First Pivot, Quick Sort Random Pivot, and Quick Sort Naive. The graph shows that the quick sort median 3 has the most constant curve compared to the other three quick sort algorithms. This is because quick sort median 3 has a complexity of O(N Log N), while the other three have a complexity of O(N^2). Median 3 takes the three elements in a given array (the beginning, the middle, and the end element), performs insertion sort on it, and utilizes the swapping mechanism twice. Then we set those new first, second, and third elements back into the beginning, middle, and end index respectively. The implementation for median 3 does not utilize any loops which voids the cost of N^2 complexity. Of course, the graph also shows that the Median 3 Naive performed the worst out of the other three quick sort algorithms. Quick sort naive involves a lot of complexity since it involves

two recursive calls per call and a for loop. With quick sort naive in retrosepct to large sized arrays, quick sort naive will likely take the longest to perform.

Average Value as the Pivot: Two Flaws

One flaw when using the average value of the array is that the average value can take a long time to calculate depending on the size of the array. For small sized arrays (i.e. 100), the average would be 50 which is not too far apart from 0 and 100. However, if we considered bigger sized arrays (i.e. 100,000, then the average would be bigger and the distance from the average to the beginning element (in other words, the gap) would be drastically bigger. The same can be said for the gap distance between the average element and the largest element.

Another flaw is that if the data is not normally distributed, the average may not be accurate due to outliers in the data. If the data is particularly skewed, the big O time can come close to worst case of N^2 time. It would intuitively make sense to say the average would best represent a middle value but we must keep in mind that any pivot can be chosen as the "average" value of the array. We could get a poor pivot choice and this would yield the complexity to be O(N^2).

Fisherman-Yates Shuffle

Let us start by explaining the differences between using "i" and "n". "i" represents the index of the given array and it has a starting value of 0. The ending value goes til the array's length - 1. Contrastly, the "n" has a different ending value because "n" represents the number of elements in the array. "N" has a starting value of zero similarly to "i" but the number of elements is "n" and not "n-1".

Because of this, if we were to interchangeably use "n" as the index for accessing elements in the array, then we are likely to get "IndexOutOfBounds" exceptions. We certainly would avoid this by choosing the index "i" instead of "N".

Additionally, because "N" represents the number of elements in the array, the number of elements does not really change in place and is thus fixed. When an array is declared to be size "N", then it is set and stone to be that "N" and can only hold "N" elements at that given time. This means the array cannot dynamically grow and shrink in terms of N. For example, when an array is declared to be size "N", it cannot be changed to "N-3" as the new length. Another array would have to be constructed in order for it to hold "N-3" elements. On the other hand, indexing is not fixed because it is the iterator for looping. "I" will start at 0 but then will change to 1, then 2, then all the way the last index, which is at the array's length -1. Because of this, choosing which numbers to shuffle is not as biased as opposed to using the number of elements as the indexing which will have some bias in shuffling.

Implementation and Algorithm Tweaks

There are a couple of implementation tweaks we made in order to improve performance. First, we focused on creating less arraylists and less loops than necessary because they can be costly. We took advantage of utilizing index variables since they have a constant complexity compared to nested loops and instantiating arrays than necessary. Merge Sort was the only sort in this assignment in that we created an extra supplementary array space (called the auxiliary array). We are aware that merge sort requires recursive calls and because of this, this requires more stack frames to be called. This runs the risk of StackOverFlowExceptions because there may be more recursive calls than our computer can handle in terms of memory. New reservations of memory must be constructed in our machine every time the program makes a recursive call.

In terms of algorithms, we think about of using the fewest sources possible because for every object (i.e. ArrayList) created, there already becomes an increase of time for an algorithm to perform the task in a timely manner. Thus, we correlate the relationship between memory usage and the time it takes to complete the algorithm.