Landon Crowther
CS 2420
02/16/2017

Assignment 05: Sorting Analysis:

This assignment required deep analysis of many different sort algorithms, because there were multiple factors being juggled throughout the analysis process. For example, some of the factors include: type of sort, number of elements that are being sorted, order of sort (N^2 vs NlogN), insertion cutoff, etc.

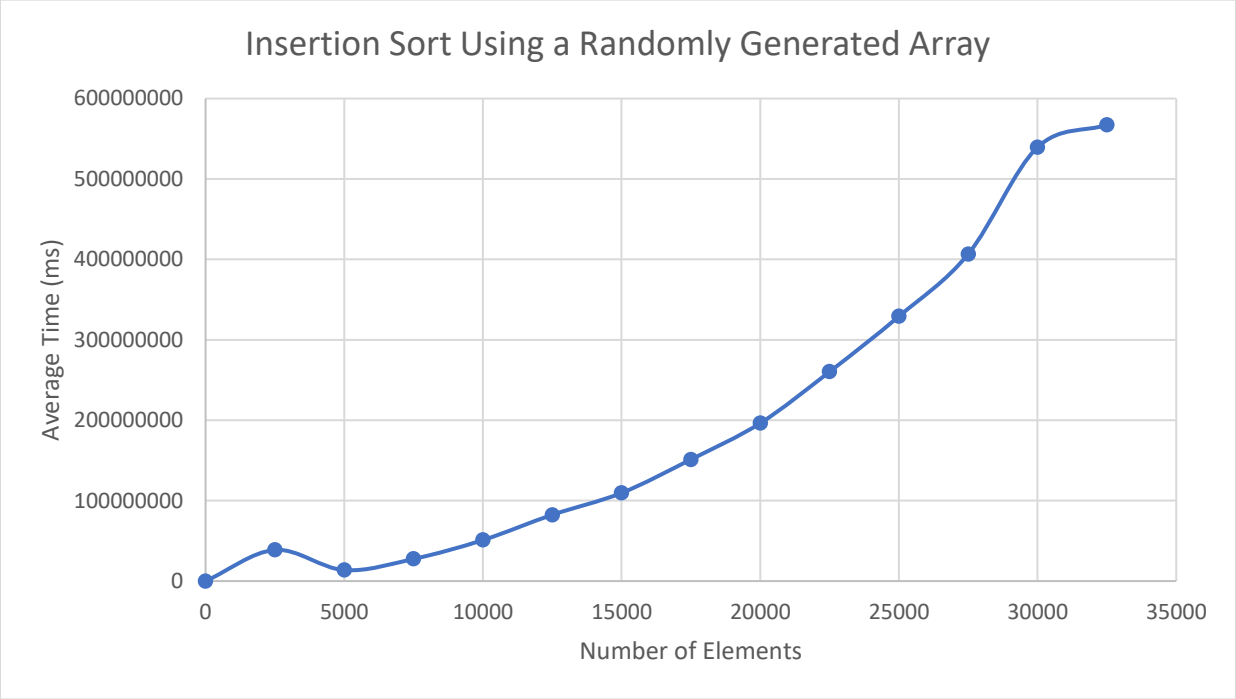
During our analysis, we tried to isolate each of these variables the best we could to get the best results. The following figures show our results:

Insertion Sort:

Insertion Sort is a N^2 algorithm. This is because there are two for loops within each other. In more general terms, the algorithm should "loop through N elements" and while this outer loop is occurring, another loop must "loop through (at most) N elements again while rearranging array." Hence, N^2 – two loops. Because this was the only sort that was N^2, it was graphed separately as to not make the other data look disproportionate.

Although Insertion Sort is terrible ineffective on random or non-sorted arrays, it is quite efficient on nearly sorted or sorted arrays. In fact, on array that are nearly sorted, the Big O behavior would be closer to N. This is because there is only one for loop that is cycling through each element. When an element needs to be altered, the nearly-sorted nature of the array ensures that the inner "shifting" loop doesn't have to travel very far.

In addition, Insertion Sort is super effective for smaller arrays. These effects will be discussed later.



Landon Crowther
CS 2420
02/16/2017

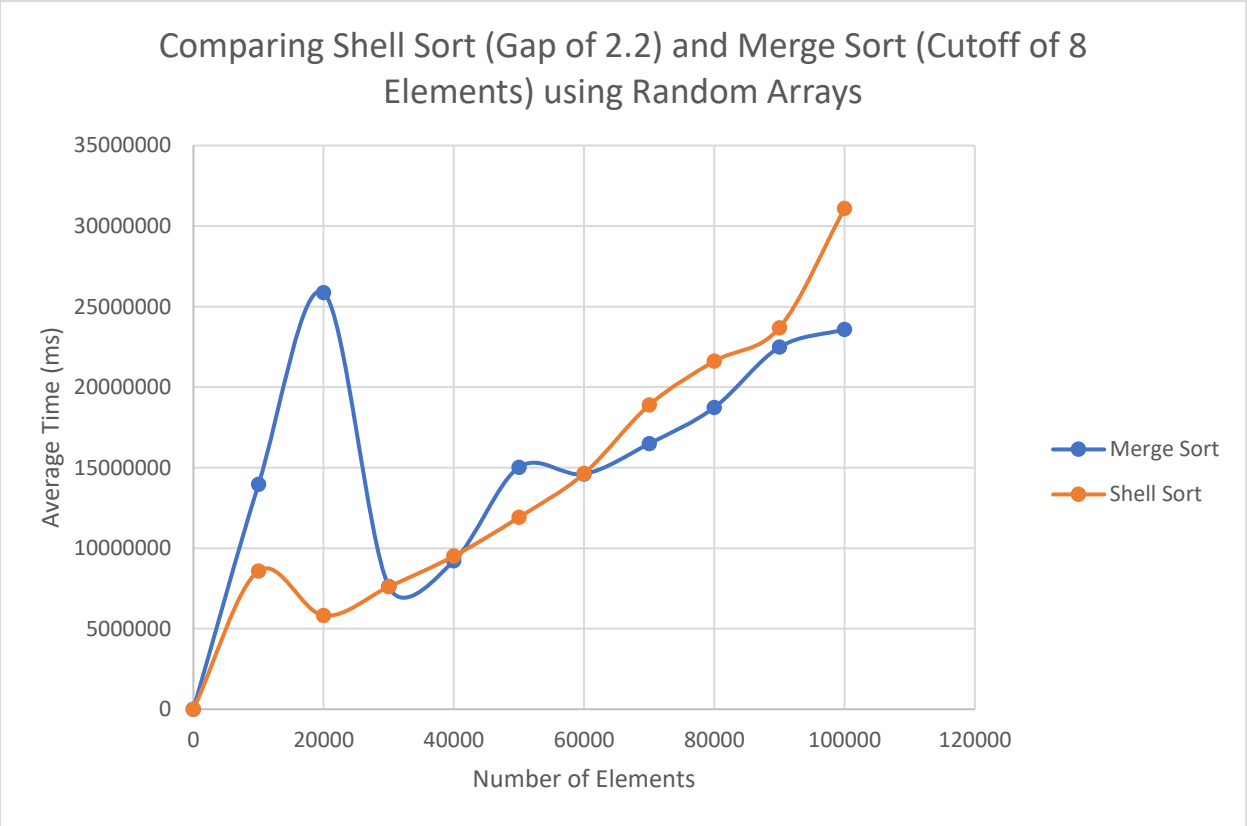
Shell Sort:

Shell Sort is the next closest sort to Insertion Sort. Shell sort is similar in nature to Insertion Sort, but rather than cycling through every single element, it skips around and sorts fewer elements in the array, leaving a gap in between. This same procedure is repeated, using slightly different gap values. The result is a nearly sorted array, which can be sorted with Insertion Sort. Because the algorithm for Shell Sort is like Insertion Sort, the Big O behavior is similar. Although Shell Sort may not quite be N^2, it is actually N^(4/3). 4/3 is between 1 and two.

Merge and Quick Sort Big O Behavior:

Both sorts are order NlogN; therefore: these methods rely on recursion to break the original array into smaller pieces. Each time the recursive method is called, the array is split in half. An array of length N can be split (at most) $\log_2 N$ times. (this is where the logN comes from). Once the array has been broken down into smaller values, it must be recomposed in the correct order. Although the is no longer one stationary array, the method that reconstructs the array must look through each element at most N times. For example, if two arrays (each of length two) are being composed, the method that combines them must look at a total of four elements, which is the size as the new combined array. This is where the N comes from. When combining the recursion as well as the reconstruction, the Big O behavior is NlogN.

The following figure compares the effectiveness of Merge and Shell Sort. Although they follow different Big O behaviors, the runtime happens to be similar.

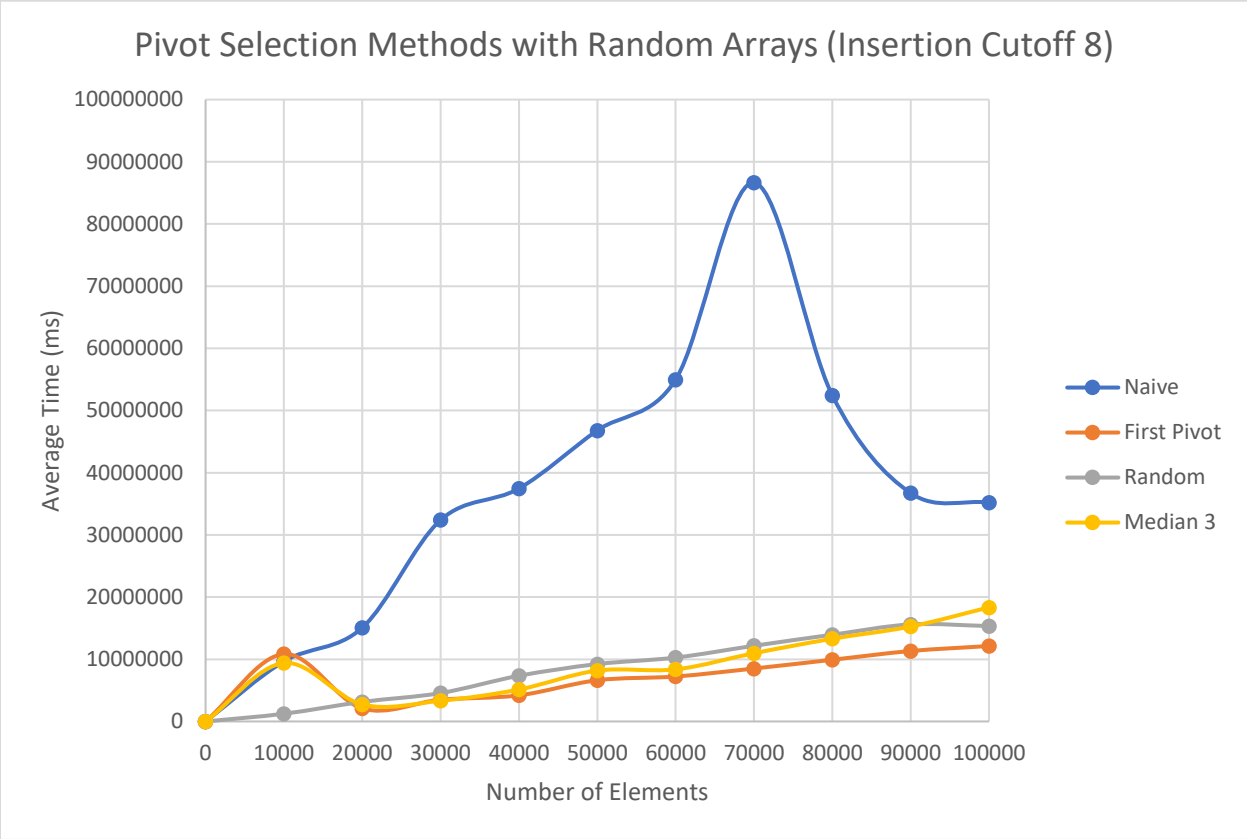


Landon Crowther
CS 2420
02/16/2017

As seen by the figure, the merge sort compares very similarly to the shell sort. I would hypothesize that if we could test more array elements (i.e. 500,000 +), the merge sort would prove to be more effective. The only reason that the number of elements stopped at 100,000 is that for some reason our code would always stop working after 100,000 elements due to timeout. If more time was available, I would take measures to insist that the array length could increase beyond 100,000.

Quick Sort:

Quick sort is also a recursive sort. Quick sort works by choosing a pivot, and then cycling through the array and re-arranging other elements based on the pivot. As stated before, quick sort is an n log n algorithm. The following figures shows the effects of choosing a pivot for quick sort:

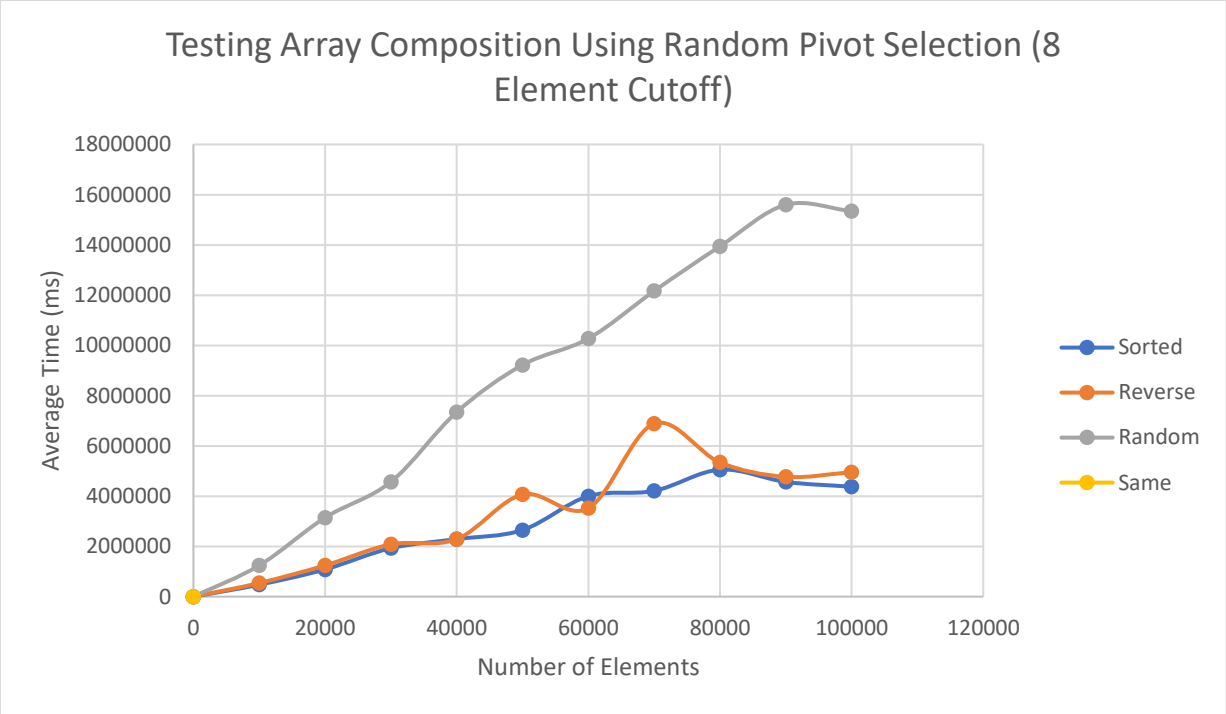


As seen by the figure, the Naïve method for selecting the pivot is incredibly inefficient. The median of three, random, and first methods are quite similar. If I could produce more data, I would infer that the median of three would be the most effective. However, because we were only able to test for arrays up to 100,000 elements, we stated that the random pivot would be most effective in our case. Even though first seems like it would be lower, it would be incredibly inefficient for arrays that are already sorted. Perhaps another reason why the median of three wasn't faster was because our implementation of deciding which values were the median out of first/middle/last could use improvement.

Landon Crowther
CS 2420
02/16/2017

Testing Best Implementation with different sorted arrays:

The following figure shows the effects of different types of arrays (i.e. pre-sorted, reverse-sorted, random, and all-same) using the random pivot selection:

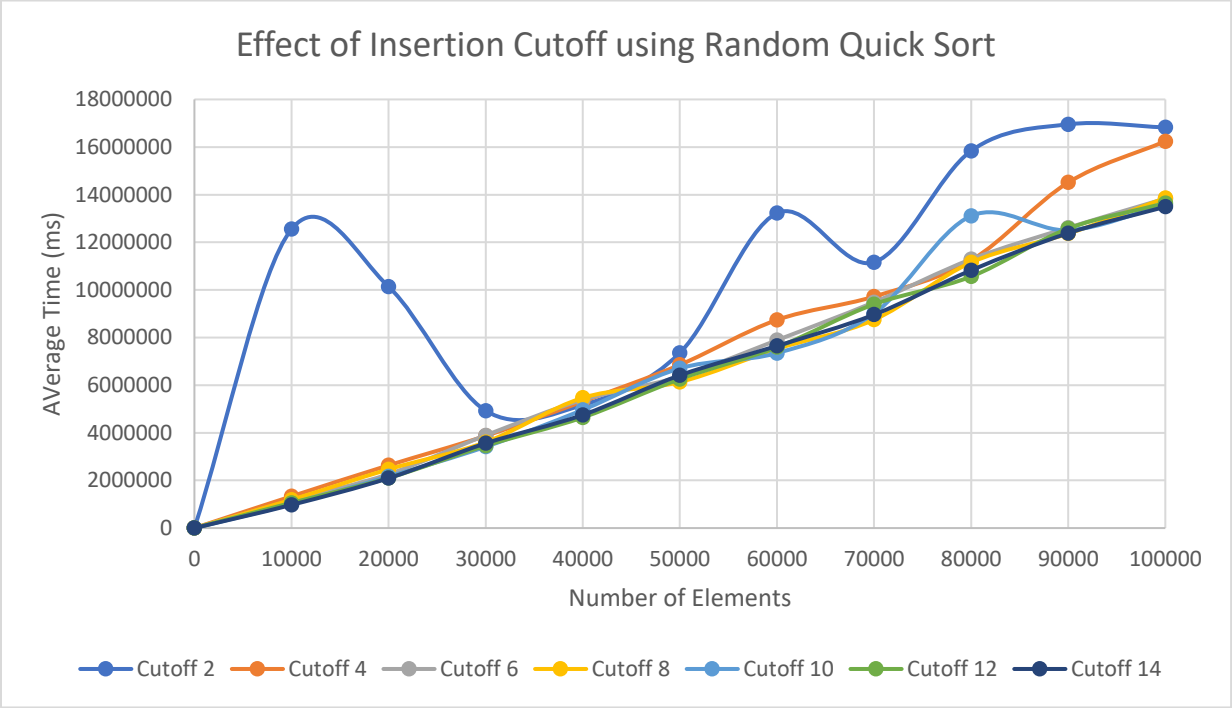


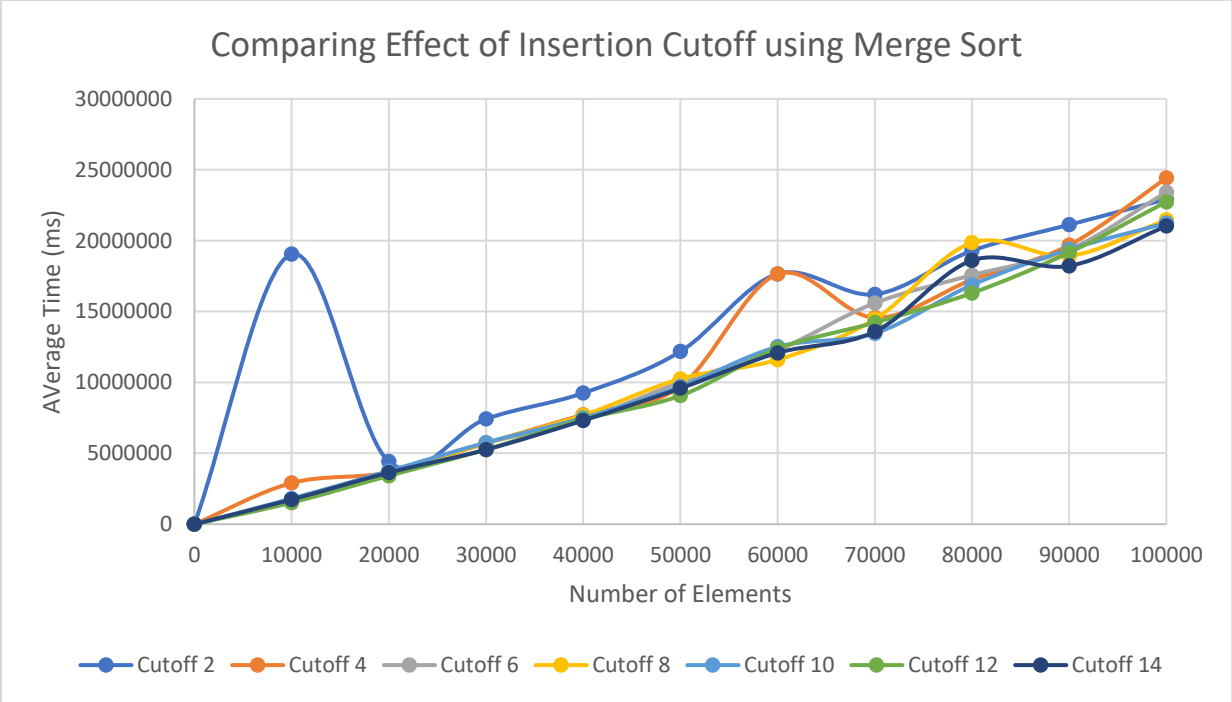
As seen by the data, sorting random arrays took the longest with quick sort. This would make sense, as the structure of the array has no “pattern” to it. All elements are random, so there is a lot of swapping going on, whereas in a sorted array, they ae already sorted and the algorithm implements much more quickly. The reverse-sorted array works efficiently because there is a lot of consistency in the algorithm. Every time a new pivot is selected, all the elements to the left of the pivot must be shifted. Therefore, the pivot index is not reselected as frequently as it would in the random array.

Landon Crowther
CS 2420
02/16/2017

Testing Effect of Insertion Cutoff:

In both merge and quick sorts, the original arrays are broken down until they reach a certain length (insertion cutoff). At this point, the sub arrays (virtual arrays) are sorted using insertion sort. Like previously mentioned, this is because insertion sort is extremely efficient on small arrays. The following figures show the effects of the insertion cutoff on the effectiveness of these sorts:





As seen by the figures above, an insertion cutoff of 2 was quite ineffective. Any cutoff between 4-14 seemed to be very similar in terms of computation time. If I had previously predicted this, a superior analysis would include cutoffs that are greater than 14 (i.e. 20, 40, 60, 100, etc.) I would hypothesize that at a certain value for insertion cutoff, the average time would start to skyrocket upwards. This is because insertion sort is extremely inefficient for large arrays, so using large values for insertion cutoffs would result in a huge increase in computation time.

Flaws of average value for quick sort:

It is obvious that the best pivot for quick sort would be the median of the array. However, to compute the median, the array must be sorted. If the user were to sort the array to find the median, the use of the median would no longer exist, as the median is already sorted. At this point, the user might want to compute the average instead of the median. Although this may be a better approach, the average also has its own problems. The first would be that sometimes the elements are not consistent. Take the following array for example: [0, 1, 2, 3, 4, 5, 100,000,000]. Although the first few elements are relatively close to one another, the final element is so large that the average would not accurately represent the data. The second flaw is that the average might not be contained within the data. Using the example from before, if one were to compute the average of the data, they would get a value that would not even exist in the array. Using a pivot that doesn't exist would break the algorithm, because the algorithm wouldn't be able to sort correctly.

Fisher Yates:

The fisher yates method is an unbiased way of shuffling elements in a list. They pick a number in the list and shuffle everything about that element. This was useful before computers, however, with large lists, the possibility of error or inconsistency is much greater. In a computer algorithm, the range of numbers that can be selected from is much greater.

Implementation Tweaks and Algorithmic Choices:

This assignment showed me that how a method is implemented means everything in terms of efficiency. This is good to know, because when applied to the real world, the programmer will most likely have some knowledge about the data they are receiving, and can make smart design decisions from that knowledge. As far as algorithmic choices are concerned, it is very critical to reduce steps if possible. One obvious example of an algorithmic decision is in insertion sort. Most people used some type of while loop to shift elements around in insertion sort. If the user tries to check every element to the left of the "bad" element to make the order right, a lot of time and performance will be wasted., especially if the array is very large. Instead, the user should check until a certain condition is met (i.e. already properly sorted), and then break out of the loop.

One way I could improve efficiency would be to simplify how the computer checks for the median of three algorithms. Our implementation does a "brute force" check, where we check a chain of possibilities until the right order (lowest, median, highest) is determined. Other alternatives could include a clever set of logic to reduce the number of steps required, or perhaps sorting the three values and computing the median that way (although I suspect that calling a sort on a 3-element array would take longer than brute force calling compareTo functions).