Lab Goals

• Alter the Insertion Sort, Mergesort, and Quicksort algorithms with some new behaviors in an attempt to improve their run times.

• Evaluate just how much of an improvement was achieved with these changes.

Assignment Details

In several classes, we discussed making a few improvements to the algorithm implementations detailed in the book. For example, we noted that stopping Mergesort and Quicksort after reaching a set threshold, then turning to Insertion Sort to sort the smaller subarray, could improve the runtime by 10-15%. In this lab, we will modify those book implementations, to judge just how much of an improvement can be reached.

Part One: Altering Insertion Sort (15 points)

When we examined the algorithm for Insertion Sort, we evaluated its performance with respect to both the less() function and the exch() function. The less() function handled compares, and the exch() function handled exchanges. In this part of the lab, we will modify the exch() function to make it more efficient.

Note that each time the exch() function is called, three operations are performed:

1. The left item a[i] is saved to a temporary variable.
2. The right item a[j] overwrites the left a[i].
3. The temporary variable overwrites a[j].

This is the most efficient way to move an item one position to the left, but what about moving the item two positions, or three, or ten? Consider the two position case: we run the exch() function twice, yielding six operations. Instead, what if we back up a[j] to the temporary variable, shift both a[i] and a[i+1] to the right one position, and then replace the value formerly in a[i] with the temporary variable. We eliminated two of the writes, dropping us down to four operations.
Here is the generic process for an unknown movement distance:

1. We will call the item we are currently inserting \(a[j]\) as before. Scan left in \(a[]\) to find the appropriate location to position this item. Call the new location \(a[i]\), as before.

2. Save \(a[j]\) to a temporary variable.

3. For each item between \(a[j-1]\) and \(a[i]\) (counting to the left):
   
   (a) Shift the item to the right one position.

4. Overwrite \(a[i]\) with the temporary variable.

Obviously, this implementation of Insertion Sort will reduce the number of \texttt{exch()} calls with the expense of some additional \texttt{less()} calls. The question is, will this tradeoff improve the running time of the algorithm?

Once you have implemented the change in the code (let’s call this new version \texttt{NewInsertionSort} for lack of a better name), it’s time to test it with \texttt{SortCompare}! Compare the original Insertion Sort with \texttt{NewInsertionSort} on arrays of 1,000, 10,000, and 100,000 doubles as in Lab 3, with 5 trials for each array size, discarding any results beyond 25% of the mean. Don’t forget to check to be sure that your updated algorithm is running and sorting the data correctly before running the tests!

As with the last lab, you will also submit a table summarizing your data runs (with mean and standard deviation) and a log-linear graph showing the ratio between runtimes. Did you notice an improvement with \texttt{NewInsertionSort}? Was the improvement significant? If you did notice an improvement, was it constant, or did it increase/decrease as you changed the size of the array being sorted?

**Part Two: Altering Mergesort (15 points)**

During the Mergesort lecture, I quickly noted a few improvements that could be made to this algorithm to improve runtime. The one that we spent the most time discussing was transitioning to a different sorting algorithm, such as Insertion Sort, when the subarray under consideration in the recursive call reached a set threshold size. In this section of the lab, we will attempt to pin down the best subarray size for this threshold.

You should first modify the book’s Top-Down Mergesort code, so that if the subarray size \((hi-lo+1)\) is small enough, you run Insertion Sort on that subarray. This will effectively replace the \texttt{if (hi <= lo) return;} line in the code. Let’s call this new algorithm \texttt{MergeInsertion}.

Now, let’s experiment with a few threshold sizes. You will run \texttt{SortCompare} on Mergesort and \texttt{MergeInsertion}. Because we are interested in determining the best threshold, we will make that our independent variable and stick to a set array size of 100,000. Again, you will run 5 trials for each threshold value, discarding any outlying results, and generating tables and graphs for your final submission. Don’t forget to check to be sure that your updated algorithm is running and sorting the data correctly before running the tests!
For your threshold values, try runs with 5, 10, 25, and 100. Which of the thresholds gave the best performance boost over the book’s Mergesort?

**Part Three: Altering Quicksort (15 points)**

During the Quicksort lecture, I also quickly noted a few improvements that could be made to the Quicksort algorithm to improve runtime. In this section of the lab, we will try “Median-of-Three” partitioning. Remember that this partitioning style involved choosing three potential pivots, and eventually going with the median of the three options, guaranteeing that the default left-most pivot will not be selected if it is the worst possible choice from a partitioning perspective. (Naturally, you do not have to do Median-of-Three when the size of the subarray drops below three.) The tradeoff in this case is to evaluate the extra computation required to find the median pivot versus the extra computation time for choosing a bad pivot.

Again, you will modify the book’s Quicksort code to handle the Median-of-Three partitioning. Because the first step of Quicksort is to shuffle the array values, you can safely use the first three values in the subarray as your three pivots to analyze, though you can certainly choose them randomly from elsewhere in the subarray if you would like. To simplify things, after you have located the median of the three potential pivots, you can simply exchange it with the first item in the subarray and then run the \texttt{partition()} function already provided.

Once you have implemented the change in the code (let’s call this new version \texttt{ThreeQuicksort}), we can start to test it with \texttt{SortCompare}. Don’t forget to check to be sure that your updated algorithm is running and sorting the data correctly before running the tests! Compare the original Quicksort with ThreeQuicksort on arrays of size 10,000, 100,000, and 1,000,000, again creating the tables and graphs as before. Did you notice an improvement with ThreeQuicksort? Was the improvement significant? If you did notice an improvement, was it constant, or did it increase/decrease as you changed the size of the array being sorted?

**Part Four: While You Have Some Downtime (5 points)**

While you wait for the above program executions to complete, you should have a bit of downtime to answer some additional questions:

1. Trace how Merge Sort will sort the string ALGORITHMS.
2. Trace how Quick Sort will sort the string ALGORITHMS. For consistent answers across the class, let’s say that the \texttt{StdRandom.shuffle()} call returns “HOTMARSGIL.”
Submission Details

For this lab, please submit the following, either to BitBucket, on paper, or both as specified:

1. (Upload) Your source code for your new algorithms `NewInsertionSort.java`, `MergeInsertion.java`, and `ThreeQuicksort.java`, plus the modified `SortCompare.java` that you used for your tests.

2. (Upload and Print) The tables, graphs, and questions noted in Parts 1-3, and your responses to the questions in Part 4.

3. (Upload and Print) An Assignment Information Sheet filled out for your source files.

Please remember that all files that you submit should be your own work, though you are welcome to discuss high-level topics and algorithms with classmates.