COMP 360 Homework 1: Algorithm Tuning

Matt Adelman

September 30, 2012

Let me start off by providing the code for my different versions of quicksort. Since the partition function and the insert function for small sizes are always the same, I will provide those first. Note that all programming and testing was done in Java. First, we have partition. The way partition functions is that it takes an array, a starting index, and an ending index and re orders the array such that every element that is less than or equal to the first element comes before it, everything greater will come after it. It then returns the new position of our original first element. The code is a modified version from Sedgewick.

```
1  public static int partition(int[] array, int left, int right) {
2      int pivot = array[left];
3      // Counters to work up and down the array
4      int i = left;
5      int j = right;
6      // Temporary variable for swapping data
7      int temp = 0;
8      while(j > i) {
9          // Walk up the array until you find something bigger
10         while(array[i] <= pivot && j > i) { i++; }
11         // Walk down until you find something smaller
12         while(array[j] > pivot && j >= i) { j--; }
13         if (j > i) {
14             temp = array[i];
15             array[i] = array[j];
16             array[j] = temp;
17         }
18     }
19     temp = array[p];
20     array[p] = array[j];
21     array[j] = temp;
22     return j;
23 }
```

Now we have our code for insertsort. Insertsort walks through the array one element at a time, starting at left and puts it in it's correct place by putting it before the first element that's greater than it. We do this until we end at the position right.

```
1  public static void insertSort(int[] array, int left, int right) {
2      // Our counters and temporary storage
3      int i, j, temp;
4      // Walking through the array
5      for (i = left; i <= right; i++) {
6          temp = array[i];
7          j = i;
8          // Finding the right place to put our current element
9          while (j > 0 && array[j - 1] > temp) {
10             array[j] = array[j - 1];
11             j--;
12         }
13         array[j] = temp;
14     }
15 }
```

Let us now examine the multiple versions of quicksort that are implemented. First off, we take the original quicksort. The way this works is that it partitions the array and then recurses on each half so we end up with a completely sorted array. This code if modified from Sedgewick.

```
1  public static void quickSort(int[] array, int left, int right) {
2      if (left < right) {
3          int pivotNewIndex = partition(array, left, right);
4          quickSort(array, left, pivotNewIndex - 1);
5          quickSort(array, pivotNewIndex + 1, right);
6      }
7  }
```

Our next version of quicksort takes a parameter m, where if the size of the array we are working on is less than m, we call insertsort on that portion of the array. Through experimentation we can find the optimal value for m.

```
1  public static void quickSortIn(int[] array, int left, int right, int m) {
2      // Small arrays
3      if (right - left < m) {
4          insertionSort(array, left, right);
5      }
6      else {
7          if (left < right) {
8              int pivotNewIndex = partition(array, left, right);
9              quickSortIn(array, left, pivotNewIndex - 1, m);
10             quickSortIn(array, pivotNewIndex + 1, right, m);
11         }
12     }
13 }
```

The final version of quicksort we have also take a parameter m, but in this case when our

array is smaller than m, we do nothing, and then sort the whole array using insertsort at the end. That code is as follows:

```
1  public static void quickSortOut(int[] array, int left, int right, int n) {
2      \\ Only execute the sort if array is larger than our threshold
3      if (right - left >= n) {
4          if (left < right) {
5              int pivotNewIndex = partition(array, left, right);
6              quickSortOut(array, left, pivotNewIndex - 1, n);
7              quickSortOut(array, pivotNewIndex + 1, right, n);
8          }
9      }
10     // Finish sorting the array
11     insertionSort(array, 0, array.length - 1);
12 }
```

It must be stated that for all of the tests I was using the following parameters. Mac OS X Lion 10.7.4, a 3.06 GHz Intel Core 2 Duo processor, and 4 GB 1067 MHz DDR3 memory with a Darwin x86 architecture on a 64 bit system. In addition, I was using Java version "1.6.0_33" from Apple Inc. (found from `$ java -version`). The javac compiler also is the same version. The one modification I made in the experiments was that I extended the virtual memory by 2GB to allow for larger array sizes. Without this extension, java only allowed array creation of up to 27 million. So all of my calls were of the form `$ java -Xmx2000m QuickSort`.

For this tuning, I performed a series of experiments. The first was to verify that our first implementation of quicksort did in fact run in $O(n \lg(n))$ time. This was accomplished in the following way. I had a few variables set up. The first one was a value n that represented how large of and array I wanted to work with. The second variable k represented how many times I would like to test each array size. To get a good handle on if my first quicksort ran in the time frame expected I went from $n = 10000000$ to $n = 100000000$ and incrementing n by 1000000 each time. Then for each array size, we ran our quicksort on ten different arrays and took the average of the running times. MY original plan was to do upwards of 1000 trials for each array size, but with the average time value for an array of size 100000000 being around 15 seconds, it was not feasible in this setting. The values for each array were chosen by the java `Random.nextInt()` method which returns a pseudo-random integer in the range of all word sized integers. A new array was made for each trial. The data can be seen in Figure 1.

The next experiment was to find the critical value for m for which it was better to use insertsort on a smaller data set. Not only do we want to find a value for this, but we want to find the best value. What I did was I fixed an array size (10000). The reason I did this this, is that our value for m should be optimal no matter what our input size is, so I fixed a relatively small input size. What I then did was I let m vary from 0 to 190. This was just an original guess size. I knew that it would be larger than 7, because that was the best over two decades ago. I did not know how large it would be, but this seemed like a good guess. I then let m vary in increments of 10 and did 100 trials for each size. For each trial,

I created 3 arrays, each full of the same contents and ran one on each version of quicksort. I then recorded the time differentials between our tuned quicksorts and our original quicksort. This data can be seen in Figure 2. Now, we observe that there is a peak between 40 and 100, so I ran the test again, this time only incrementing by 1 and using 1000000 trials for each value of m. This led to the data produced in Figure 3.

From this, we can see that for using insertsort inside the quicksort, our optimal value is when $n = 60$. However, when insertsort is performed all at the end, we see an optimal value of $n = 59$. Therefore, for my last test, I used the same parameters for array and trial sizes that we used in our first test. This data produced Figure 4.

Looking at the data, we see that both versions of quicksort outperform the original version of quicksort, but neither really outperform each other. Once we're at array size 100000000 we get a time differential of about 1.7 seconds. This, while small is not insignificant. One of the most interesting conclusions that I drew from this experiment is how large an array you need to make before quicksort is actually faster. I was curious what the critical point was, and not just the optimal point for quicksort versus insert sort, and we can see that in Figure 5. This means that until we get to input sizes of 115 or greater, insert sort is definitively faster. In all, it is fascinating how such a simple modification can make such a big difference for large datasets.

Our figures:

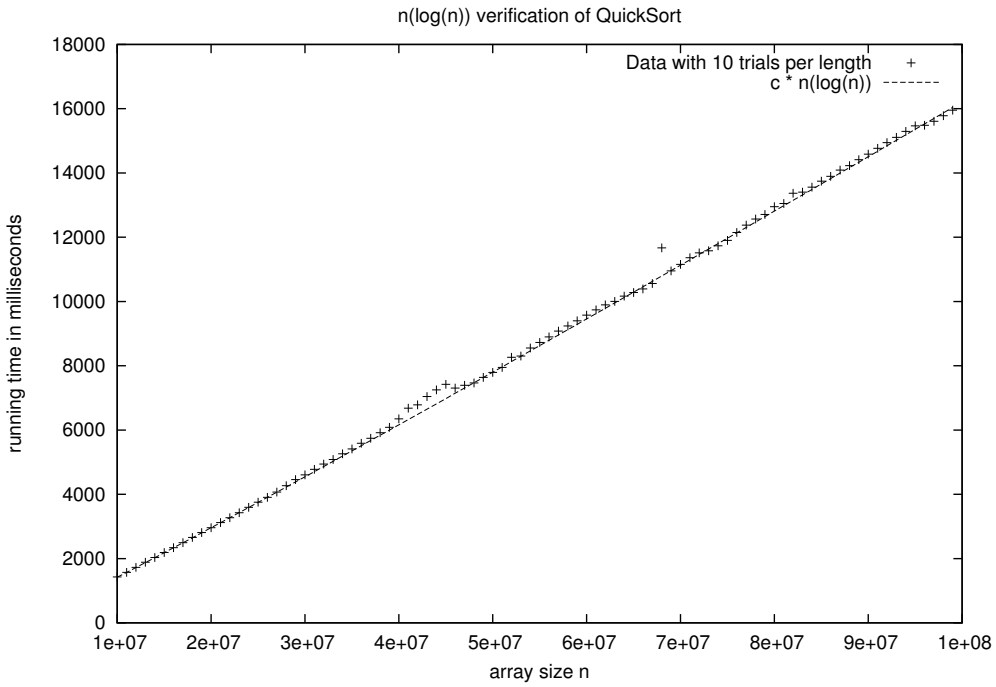

Figure 1: Verification of $O(nlg(n))$ time.

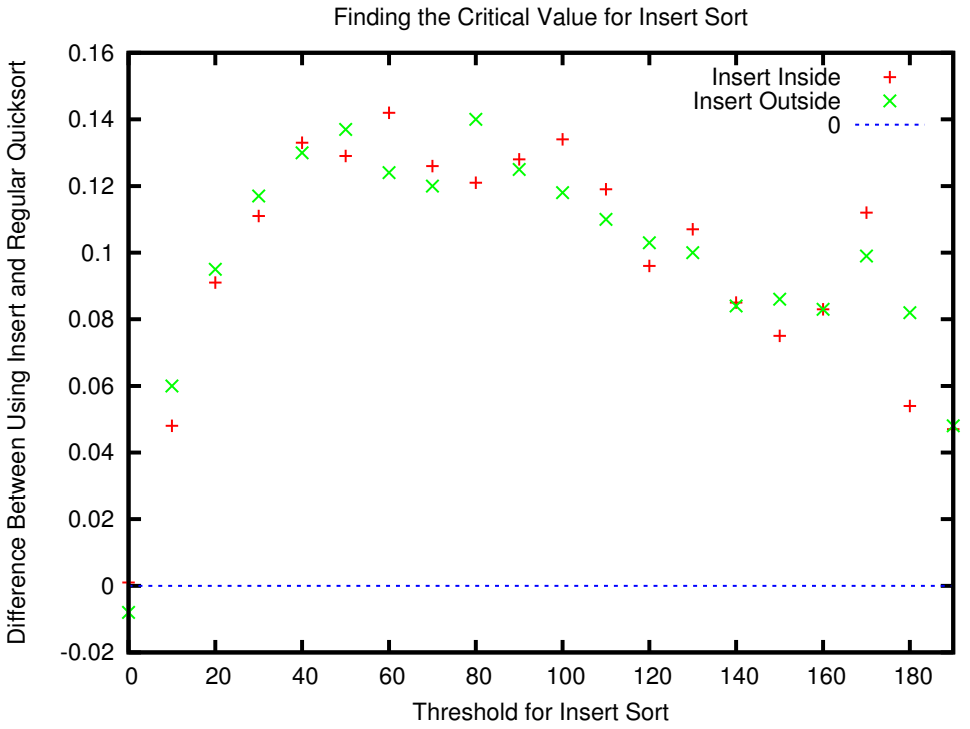

Figure 2: First attempt at finding critical m value.

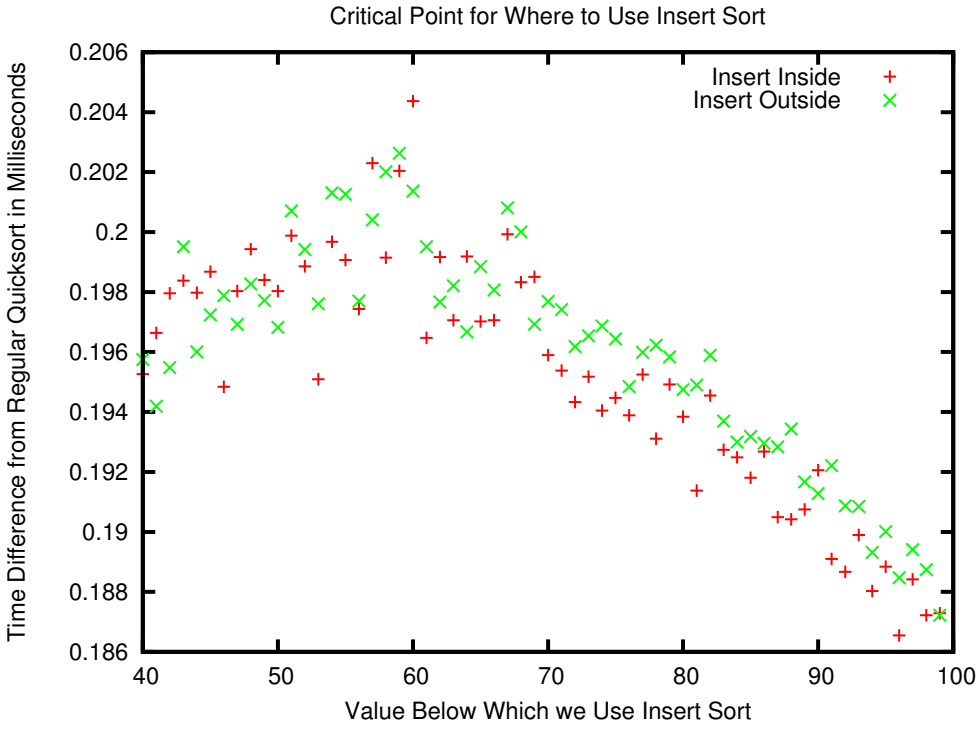

Figure 3: Finding critical m value.

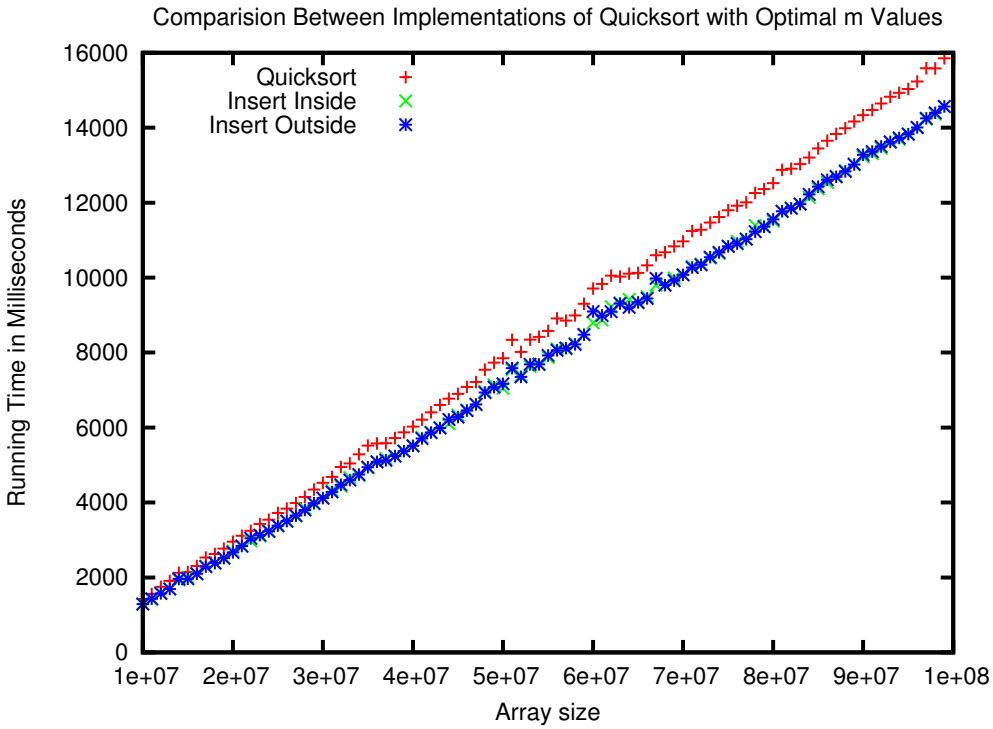

Figure 4: Using Tuned quicksorts on large data.

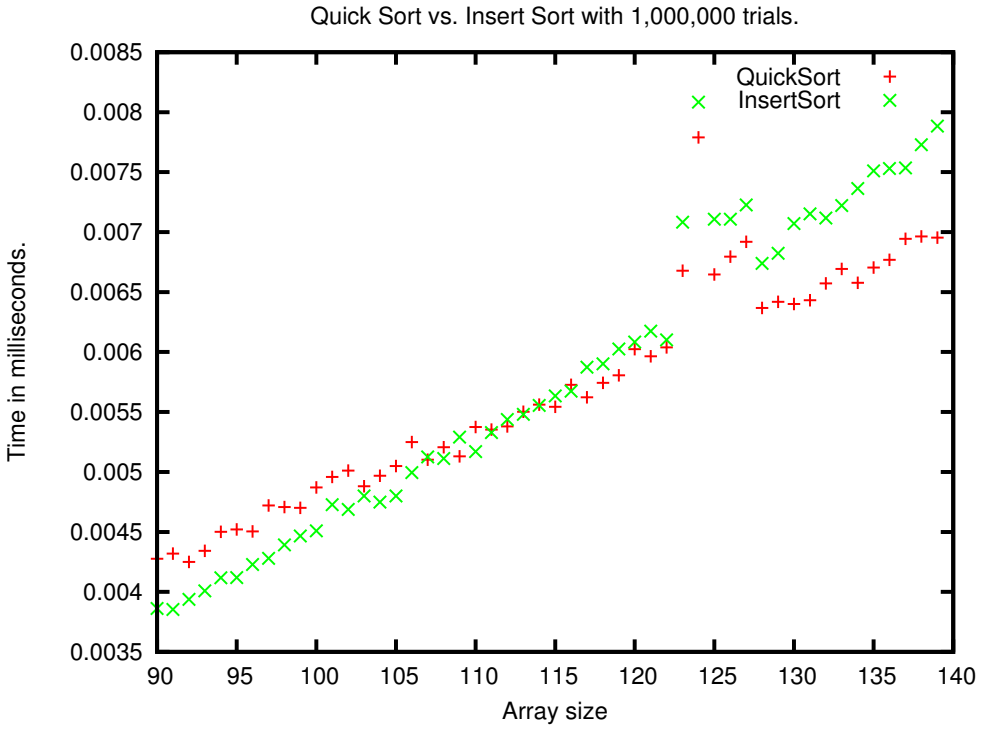

Figure 5: Finding crossing of quick and insert sort.