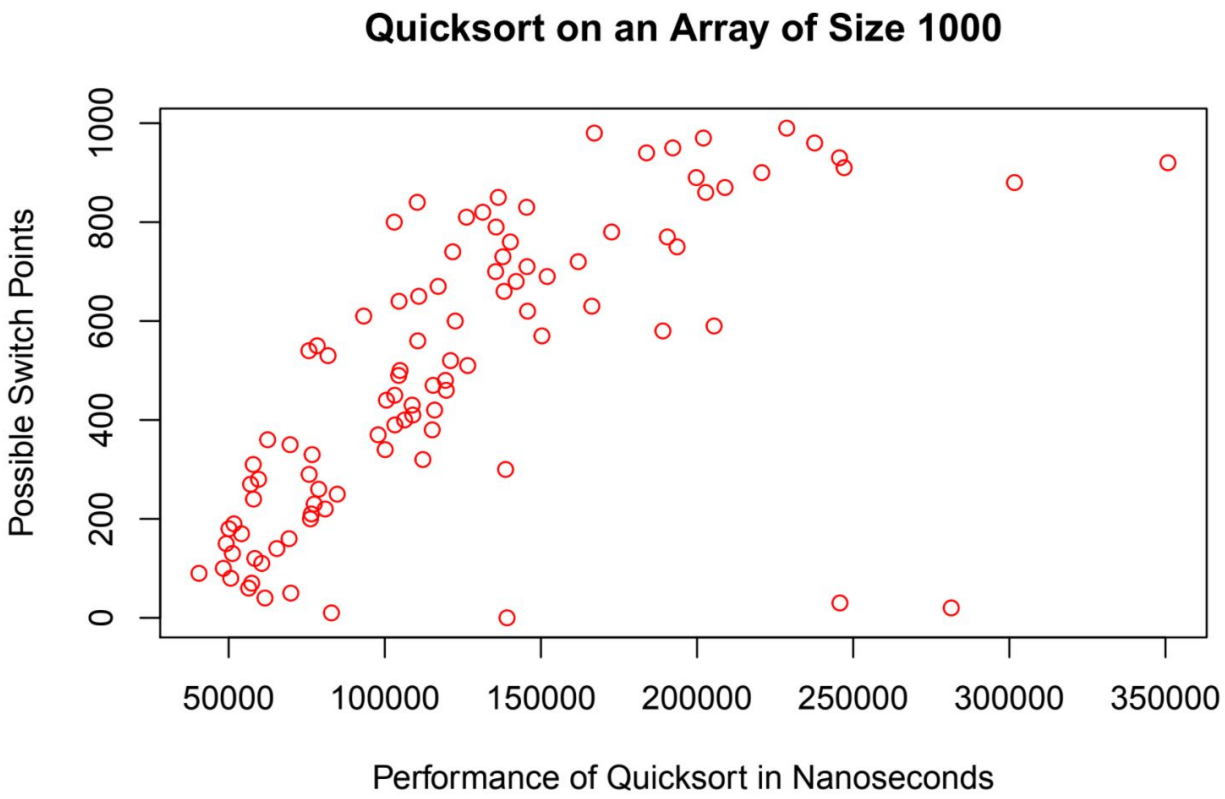
Dean Gladish
Ben Tordi
Monday, February 19

Homework 7: Sorting Comparisons

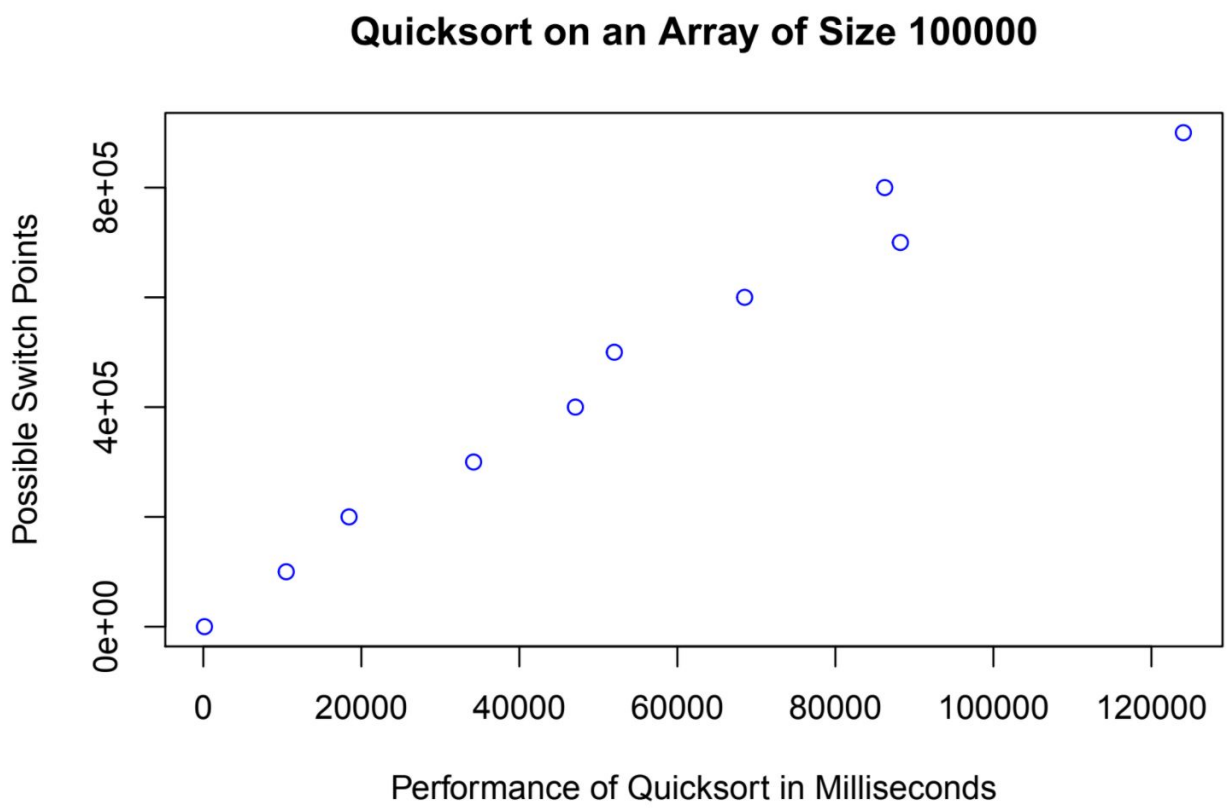
In both of our timing experiments, we use averages of the algorithm's runtime in order to obtain a more stable and accurate approximation of the sorting time using Quicksort and Shellsort. For the Quicksort analysis we condition on the Insertion sort threshold; that is, we modify the minimum subarray size below which the Quicksort algorithm switches to the use of Insertion sort. In our analysis of Shellsort we condition on the gap size multiplier (the value that changes the rate of change of the gap). Our result for a multiplier of 2.25 is the shortest sorting time out of all of our trials. Thus, we recommend the use of this multiplier when using Shellsort.

For Quicksort, we use two different array sizes - one of 1000 and one of 1 million. For the smaller array we calculate the sorting times in nanoseconds due to the briefness of sorting such a small array (and although the actual start and stop times are inaccurate, the difference is accurate). Using nested for-loops (the first loop generates different Insertion sort threshold parameters and the second takes the average of many trials at that threshold), we fill two Arraylists with threshold values and average Quicksort times respectively. We then generate a scatterplot of these values with the thresholds and their corresponding times as coordinates:



The Quicksort time appears to be an exponential function of the Insertion sort switch point. This makes sense because the higher the insertion point is the more Insertion sorts take place. However, for extremely low switch points (0, 10, 20, and 30) the Quicksort algorithm takes a relatively longer time than it does for the following points. That is why Insertion sort should be used at some point; Quicksort is extremely inefficient when there is no switch or when the switch occurs at array sizes that are very small. As a policy we recommend setting the threshold no more than 100 for small arrays such as this.

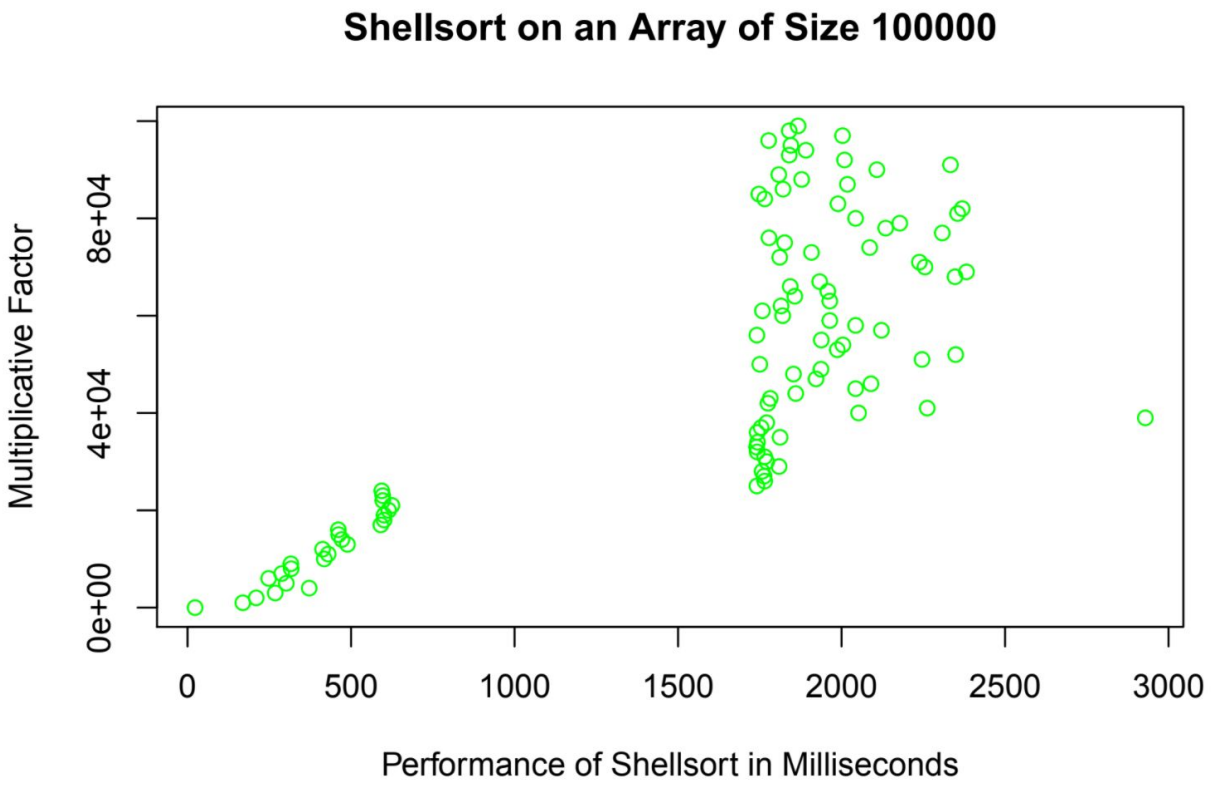
We do the same process for a larger array:



The result mostly corroborates our findings that there is a positive relationship between sorting time and insertion switch points. It also suggests a slight exponential relationship between the dependent variable of performance time and the independent variable of Insertion sort threshold. The only discrepancy is that using a threshold of 0 does not seem to result in the same increase in sorting time as it did for the smaller array. This could be due to the fact that we only take an average of three sorting times for each trial. It could also be due to the large size of the array. The result suggests that the difference between algorithms may not matter as much when we are dealing with larger sorting times in milliseconds. As such and because Insertion sort is inefficient on large arrays, one should use a similarly low switch point for large arrays as for small arrays.

For the Shellsort, we use the gap size multiplier as a parameter within the method so that we can modify it as we do with the Quicksort class. Since there is no real need to get the

average for this experiment we only need to use one for-loop with which we increment the multiplier by 1000 and test it on an array of size 100,000. We generate two Arraylists of times and multipliers respectively in order to generate the following scatterplot:



As shown on the plot there is a large increase in sorting time past a multiplicative factor of approximately (25% of the array size) - 1. This gap is due to the fact that we first divide the array by 2 to give the first gap size. After that, we find a new gap size using the multiplicative factor. Before this factor the new gap size is three or greater due to the ceiling function. After 25% of the array size, the new gap size is two or less which goes down to one automatically as specified in the assignment. The result of this is that once we are at a gap size of 1, everything that has a multiplier of under 25% gets more chances to achieve more sorted subarrays before resorting to incremental Insertion sort. The incremental Insertion sort takes more time due to the fact that its efficiency is lower on less-sorted arrays.

In the first 25% of multipliers we get more gap sizes which results in faster sorting times. Since we go to a gap size of 1 after this, the only reason for variation seems to be the difference in the time taken to perform Insertion sort. For these arrays, we get the initial gap size first which goes to 1 subsequently. The sorting time for these arrays is essentially the same (approximately 1800 milliseconds).

Finally, we also briefly examine the differences in variance among different Insertion sort thresholds for our Quicksort algorithm.

```
var(listTimes0k) / mean(listTimes0k)
```

```
## [1] 4.128718
```

```
var(listTimes25k) / mean(listTimes25k)
```

```
## [1] 22.00852
```

```
var(listTimes50k) / mean(listTimes50k)
```

```
## [1] 14.67683
```

```
var(listTimes75k) / mean(listTimes75k)
```

```
## [1] 206.9345
```

Using 30 sorting times for each threshold (0, 25,000, 50,000, and 75,000), we attempt to calculate and normalize our variances to determine if there is any relative increase in variance as the Insertion sort threshold increases. The sorting times depart from the normal distribution as the threshold increases. All distributions are heavily right-skewed except for the one for which the minimum array size is set to 50,000 (one-half of the total array size). We suggest that the sortedness of the arrays has a increasing impact on sorting time as the Insertion sort threshold increases. The natural variance of the arrays' sortedness has a large impact on Insertion sort because it determines the difference between $O(n)$ or $O(n^2)$ time.