Christopher Will

CS 315 Bonus 2 Report

For the 20 randomized sequences and the average # of comparisons made:

These results for the average number of comparisons made for each of the 4 sorting algorithms are in agreement with what was discussed in class. For an average case merge sort should run in O(n logn) time. And the same is true for quicksort. Looking at the attached .txt file we see that both these algorithms made about the same number of comparisons for all the lists of varying size. We also find that when calculating n*logn that this number is usually more than the comparisons made by these 2 algorithms. Which it should be as big Oh notation gives us a worst case upper bound. The results for selection sort also agree with what was discussed. Average case time for selection sort is θ(n^2) as we have a for loop, and then a nested for loop, so there is no way to terminate from the algorithm until the loops have exited. We established that selection sort should make (n(n-1))/2 comparisons, when sorting n elements. And this is what we have. For instance, when n = 100, we get (100(99))/2 = 4950 comparisons. And finally, the number of comparisons made for insertion sort is bounded by O(n^2). We could be unlucky and have the sequence sorted in descending order, which gives us the worst case running time, for insertion sort, hence the need for an upper bound of n^2. This, however, is very unlikely and we can see that insertion sort was always below this n^2 upper bound.

For the sequences in descending order

These results also corroborate what was discussed in class. Insertion sort performs poorly when the data is given in descending order as it must go all the way through the list to extract the minimum element from the end and insert it in the proper place. The worst case number of comparisons for insertion sort is given by f(n) = (n(n-1))/2 for n integers. This is what we have as when n=100 we need 4950 comparisons for insertion sort. Selection sort performs that same regardless of how the integers are arranged. We must always search the list for the current minimum and put it to the end, and there is no way to improve on this. So we also have f(n) = (n(n-1))/2 comparisons for selection sort. Merge sort should run in O(n logn) time in any case, and this is what we have here. For n=100 we have 415 comparisons which is bounded by O(n logn) as that value evaluates to be 665. Quicksort will also give a O(n logn) estimate for the number of comparisons, which we have for this descending sequence.

For the sequence in ascending order

	Expected time	Actual comparisons for n=100	Agree with what was discussed in class?
Insertion Sort	O(n)	100	Yes
Selection Sort	θ(n^2)	4950	Yes
Quicksort	O(n logn)	667	Yes

Mergesort	O(n logn)	455	Yes

Looking at this table we see that each of the 4 algorithms behave as expected when the integers are given in ascending order. Insertion sort has the fewest number of comparisons as it performs best when the data is already in ascending order. Comparisons as a function of n would then be f(n) = n for insertion sort in this case. Selection sort always has the same number of comparisons regardless of the data is input, so these results also confirm what we discussed. That is, we have f(n) =(n(n-1))/2 comparisons. Quicksort will have its potential worst case if the data isn't randomized, and here we see that for n=100 we have 667 comparisons, which is bounded by its average case O(n * logn) number of comparisons. And merge sort should always have O(n logn) performance, which is what we see for all the values of n.