Sam Brendel

5/5/2017

TCSS342, Data Structures

Assignment 2, sam2b-pr2

SVN revision # 29

**Summary Report on Big-O analysis**

**ABSTRACT**: The analysis of four sorting algorithms run on two different data structures is as follows. The corresponding graphs depict the Big-O growth rate for the respective sort algorithm according to the data structure with both best-case scenario (sorted ascending order) and worst-case scenario (sorted descending order). The data gathered is the average run times of 3 separate runs on the same machine with no other programs running (i.e. antivirus, browser, eclipse, unnecessary Windows services, etc), compiled with no debugging information, and run in a dedicated command prompt.

**NOTE**: the maximum quantity of elements tested is only 500,000.

**Graph 1**: ArrayList with random order Integer data. (average-case scenario)

Insertion Sort: O(n2)

Selection Sort: O(n2)

Quick Sort: O(nlog(n))

Merge Sort: O(nlog(n))

**Conclusion**: An ArrayList data structure sorting random Integer data experiences the longest sortation run times for selection sort and insertion sort respectively. Not only does selection sort take longer than insertion sort, but with each increase of the quantity of elements to sort, the steepness worsens with the slowness rate with selection sort. However, the Big-O analysis of these two algorithms are both O(n2) because Big-O notation omits constants and coefficients (reflecting operations performed). Similarly, the quick sort algorithm is O(n2), but the graph 1 shows extremely low run times, which is due to the Big-O notation omitting constants and coefficients (reflecting operations performed). The clear choice is to use either quick sort or merge sort on an array-based data structure when we need to sort data in a random order due to their nearly-linear run time.

**Graph 2**: ArrayList with sorted (ascending order) Integer data. (best-case scenario)

Insertion Sort: Ω(n)

Selection Sort: Ω(n2)

Quick Sort: Ω(nlog(n))

Merge Sort: Ω(nlog(n))

**Conclusion**: An ArrayList data structure attempting to sort Integer data that is already in ascending order has extremely lower run times because the quantity of operations is drastically reduced. The only operations necessary is to compare elements and advance to the next element. Because of that fact, all except the selection sort algorithm are nearly linear in run time. It is no surprise that selection sort takes exponentially longer even in this best-case scenario because of its inevitable n2 operations regardless of the order of the data. The clear choice for sorting an array-based list that is mostly sorted is NOT selection sort, and instead chose insertion sort, or quick sort, or merge sort respectively.

**Graph 3**: ArrayList with opposite (descending order) Integer data.

Insertion Sort: O(n2)

Selection Sort: O(n2)

Quick Sort: O(n2)

Merge Sort: O(nlog(n))

**Conclusion**: An ArrayList data structure sorting Integer data in sorted descending order is the worse-case scenario, and is a true test for speed. This graph 3 clearly shows n2 operations for both insertion sort and selection sort, looking quite parabolic, with insertion sort a bit slower than selection sort. However, even though quick sort is claimed to be O(n2) in a worst-case, the graph 3 results show it to be more linear than exponential. Merge sort on graph 3 shows more consistent with its O(nlog(n)) rating, and appears mostly linear in run time. The clear choice is to use quick sort or merge sort for a worst-case scenario on an array-based list.

**Graph 4**: LinkedList and ArrayList with random order Integer data.

LinkedList Insertion Sort: O(nm)

LinkedList Selection Sort: O(n2+m)

ArrayList Insertion Sort: O(n2)

ArrayList Selection Sort: O(n2)

**Conclusion**: In comparison, my own implementation of insertion sort and selection sort on a LinkedList data structure are naturally worse than on an ArrayList because of more instructions in the algorithm for LinkedList necessary to do the same operations as implemented in ArrayList. However, my implementation of selection sort on a linked list is nearly O(n3) in run time. I’m sure this is due to when I used nodeAt( ) twice in each loop iteration to “reset” the temporary variables’ position. I’m sure this is avoidable with a better implementation, and perhaps could be closer to O(n2) in run time without the use of nodeAt( ). My selection sort algorithm for LinkedList took 4.3 hours to finish sorting 500,000 random Integer data. I would be happier to see it run within 30 minutes. My selection sort algorithm clearly takes flight after about 100,000 elements, and then is clearly exponential in growth after that.