Programming Assignment #1 Report

Aaron Van De Brook

CS 315

10/07/2019

1. Introduction

The purpose of this assignment was to implement two sorting algorithms, 1 elementary algorithm and 1 advanced algorithm. The wall-clock runtimes were then calculated for each algorithm on sets of data of varying lengths in varying orders. Specifically, the initial data was in random order, ascending order, and descending order to test the average, worst, and best-case runtimes for an algorithm.

The algorithms used and presented in this report are Insertion sort, for the elementary algorithm, and Quick sort, for the advanced algorithm. The Big-O time complexity of Insertion sort is $O(n^2)$ for the average and worst-case scenarios, and $O(n)$ for the best-case, aka an array of data that has already been sorted. The time complexity for Quick sort is $O(n \log(n))$ for the average and best-case time complexity, and $O(n^2)$ for the worst-case time complexity.

2. Complexity Analysis

When calculating the time average time complexity for the Insertion sort algorithm, a runtime of $4n + 2$ was found for the inner-loop of the algorithm. This lead to a runtime of $O(4n (4n + 2) + 2)$, which expands to $O(18n^2 + 8n + 2)$, when including the outer-loop. The time complexity of the entire function was determined by the equation $O((4n (4n + 2) + 2) + 3)$, which expands to $O(16n^2 + 8n + 5)$. This yielded a Big-O time complexity of $O(16n^2 + 8n + 5)$ which simplifies to $O(n^2)$. This is the time complexity for both the average and worst-case input scenarios. When an array of already sorted data is passed to the function, however, the time complexity of the algorithm becomes $O(n)$ because the inner-loop of the Insertion Sort algorithm will never execute.

	Average	Best	Worst
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$
Quick Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$

Figure 1: Average, best, and worst-case time complexities of the Insertion and Quick sort algorithms.

The average and best-case time complexities for the Quick sort algorithm is $O(n \log(n))$, this is due to the way the partition algorithm splits the array into powers of two in order to sort it. The worst-case time complexity of the Quick sort algorithm is $O(n^2)$, this occurs when the array passed to the algorithm is sorted in reverse order (in other words, in descending order). This worst-case time complexity is due to the way the partition algorithm selects a boundary element to recurse on, it basically selects a boundary element that is at one extreme of the array thereby losing the efficiency that the "divide and conquer" algorithm provides. This forces the time complexity to degrade to $O(n^2)$ in the worst case.

3. Experimental Design

To effectively test and evaluate how the Insertion and Quick sort algorithms scale with the size of a data set, five tests were created and run for both small and large data sets. The average, best, and worst-case time-complexities of these algorithms occur when sorted and reverse sorted arrays are passed to the algorithms, so to test the extremes of the algorithm's arrays in random, ascending, and descending order are used. By using these arrays, the average and extreme behavior of these algorithms can be effectively tested.

The following data sizes were used to evaluate and plot the time complexities of the algorithms used:

- Small Data Set ($n \leq 1000$):
    1. n = 10
    2. n = 50
    3. n = 100
    4. n = 500
    5. n = 1000
- Large Data Set ($n > 1000$):
    1. n = 10,000
    2. n = 20,000
    3. n = 30,000
    4. n = 40,000
    5. n = 50,000

4. Results

At the beginning of the small data set experiments the Insertion sort algorithm scaled better than the Quick sort algorithm in all categories until the third set of tests, where Quick sort began to perform faster than Insertion sort. Quick sort continued to outperform Insertion sort in all categories except in the descending data arrays. This was because Quick sort's worst-case time complexity occurs in data sets sorted in reverse order, where the time-complexity degrades to $O(n^2)$.

Insertion Sort					
Small Data			Large Data		
Random	Ascending	Descendin	Random	Ascending	Descendin
3900	1300	2101	13091900	46600	21204600
16900	2301	41499	40083400	99801	76894600
103200	3600	100501	83138100	162800	1.8E+08
1403601	13001	592501	1.45E+08	83700	3.5E+08
2452100	5199	6017200	2.62E+08	88601	5.45E+08

Table 1: Insertion sort; small and large data set times to execute. Times in nanoseconds.

It is worth noting that when testing Quick sort on large data sets, the stack size of the Java Virtual Machine (JVM) needs to be increased to avoid stack overflows due to the recursive nature of the algorithm.

Quick Sort					
Small Data			Large Data		
Random	Ascending	Descendin	Random	Ascending	Descendin
13700	2600	3100	4642600	23445500	23144001
55400	38300	123301	11573600	77127100	85679901
73399	139300	47200	20341400	1.85E+08	1.76E+08
479899	638300	564200	20544400	3.19E+08	3.2E+08
670600	1689100	1461400	11068500	4.98E+08	4.11E+08

Table 2: Quick sort; times to sort small and large data sets. Times in nanoseconds.

5. Conclusion

Quick sort outperforms Insertion sort in randomly sorted data, because Quick sort has an average time complexity of O(n log(n)) and Insertion sort has an average time complexity of $O(n^2)$ meaning that Quick sort’s time to sort a set of data will scale better and more consistently with time than that of Insertion sort’s. However, Insertion sort will outperform Quick sort when an array is already sorted, this is Insertion sort’s best-case scenario and leads to a time complexity of O(n) whereas in a sorted data set Quick sort will still have an average time complexity and will perform with O(n log(n)), in this case Insertion sort will scale linearly and Quick sort will scale log-linearly, meaning that Insertion sort’s runtime will scale slightly better than Quick sort’s. In the array of descending data Quick sort and Insertion sort will have similar times because they will both scale with $O(n^2)$, this is a worst-case scenario for Quick sort, and an average case for Insertion sort.

6. Appendix

See ZIP file for source code.