CSC 1120A

4/16/2024

LAB 12 BIG O ANALYSIS

Wood, Michael

Does the performance of the algorithm change based on the starting order of the elements?

The Approach

I decided to approach this problem by creating a tester method that would time how long it takes to sort a list using my sorting method. First, I had to decide which kinds of lists I would sort. I decided that I would run benchmarks on a random list, sorted list, nearly sorted list, and reverse sorted list in order to determine whether the performance of the algorithm would change based on the starting order of the elements. I decided to use a list size of 100 elements for each of the tests.

Results of the Benchmarking

Table 1: SmallerBiggerSort.sort() benchmarks

Trials	Random Order	Sorted Order	Nearly Sorted Order	Reverse Sorted Order
Trial 1	2,096,100 ns	1,250,800 ns	675,700 ns	4,469,400 ns
Trial 2	1,837,900 ns	1,897,100 ns	768,800 ns	751,200 ns
Trial 3	2,473,000 ns	1,381,200 ns	1,027,700 ns	947,800 ns
Trial 4	2,108,900 ns	1,602,000 ns	1,021,300 ns	530,100 ns
Trial 5	2,130,200 ns	1,350,600 ns	709,300 ns	547,800 ns
Average	2,129,220 ns	1,496,340 ns	840,560 ns	1,449,260 ns

On average sorting a list in random order took approximately 2,129,220 ns. On average sorting a list that was already sorted took approximately 1,496,340 ns. On average sorting a list that was nearly sorted took approximately 840,560 ns. On average sorting a list that was in reverse sorted order took 1,449,260 ns.

Analysis

The performance of the algorithm does change based on the starting order of the elements. This algorithm has worst case O(n^2) and this happens when the pivot element is either the smallest or the largest element in the unsorted sublist. We can see this from our results in table 1 where the sorted list and reverse sorted list had nearly twice the runtime as the nearly sorted list. This happens because when the sort method is recursively called, the sublists are very uneven in size where the sorted sublist is typically only increasing by 1 element each time.

This algorithm has best case O(nlogn) and this happens when the pivot in each sublist is the median (middle value) of the sublist. This causes this logn behavior because as the sort method is recursively called, each sublist is n/2 in size compared to the previous sublist. We can see this in table 1 because the nearly sorted list had pivots which were typically near the medians every 10th element. This was exceptionally quicker than an already sorted list, reverse sorted list, and the random order list.