Medina Lamkin

CS 350

02/12/2019

Implementing and Testing Sorting Functions

For this assignment, I decided to implement insertion sort, in addition to merge sort and selection sort. I initially considered two different programming languages to complete this assignment – C++ or Python. In the end, I opted to implement these algorithms using Python. I have had previous experience of implementing a couple of simple sorting algorithms in C++ as well as a lot of other experiences programming in the language, so I was aware of how difficult it can be to use at times. Python on the other hand is much simpler to use. I have only used Python for machine learning related exercises, so I have been wanting to try using it for some class assignments. This was a great opportunity for that, so I went ahead and used Python.

To implement the sorting algorithms, I referred to class notes. Due to extensive similarities between how pseudocode is written and how Python code is implemented, much of my code is very similar to the pseudocode from class, especially for insertion sort and selection sort. To test the functionality of my code, I first created 10 element numpy arrays of random, sorted, reverse sorted, identical, and half-random-half-sorted input. Although it was only specified that the algorithms should be timed with random and sorted input, I decided to expand the test data sets to include reverse sorted data, which would be the worst case, as well as the other two just to see how they affected the run time.

Once the algorithms performed correctly on this set of inputs, I created arrays of three more sets of inputs, each with the five types of data mentioned previously, of sizes 100, 1 000, 10 000, and 100 000 to test the run time of the algorithms. The timing was done on the hardware specifications listed in Image 1, using the Python ‘time’ module, specifically the ‘process_time’ function, which gives the total CPU and user time for the process in seconds.

Image 1: Hardware information

For the most part, the development process proceeded smoothly, with the exception of the implementation the merge algorithm for merge sort. This difficulty was due to the use of array slicing, which only creates a new reference to the initial array, rather than making a separate copy. If successfully implemented using array slicing, this would result in an in-place merge sort. After many trials and errors, I overcame this difficulty by sorting the arrays as I merged them using insertion sort.

Table 2 shows the times selection sort took to sort a variety of inputs of different sizes. In class, the basic operation chosen to analyze sorting algorithms by was the total number of comparisons done. Using this, theoretically, insertion sort should be upper bound by 'n^2' in the best case and worst case.

	Type of Input				
Number of Inputs	Random	Sorted	Reverse Sorted	Identical	Half-random-half-sorted
10	2.086×10^{-5}	1.775×10^{-5}	1.732×10^{-5}	1.702×10^{-5}	1.703×10^{-5}
100	1.045×10^{-3}	1.018×10^{-3}	1.036×10^{-3}	9.786×10^{-4}	9.837×10^{-4}
1 000	1.001×10^{-1}	9.346×10^{-2}	9.336×10^{-2}	9.331×10^{-2}	9.220×10^{-2}
10 000	12.14	12.97	12.87	12.65	12.65
100 000	1259	1283	1278	1289	1240

Table 1: Time in seconds taken to sort arrays using selection sort

While I don't know how long a single comparison of two values would take, the timings for all types of input data of the same size takes approximately the same amount of time, which matches the expectation that the best and worst case would be the same. Based on the results of the analysis of the next two algorithms, it becomes obvious that the time taken is not within the expected time.

Table 2 shows the times merge sort took to sort a variety of inputs of different sizes. Theoretically, by the master's theorem, insertion sort should be upper bound by 'n' in the best case and worst case.

	Type of Input				
Number of Inputs	Random	Sorted	Reverse Sorted	Identical	Half-random-half-sorted
10	7.158×10^{-5}	4.179×10^{-5}	5.030×10^{-5}	3.986×10^{-5}	4.499×10^{-5}
100	1.161×10^{-3}	4.498×10^{-4}	1.827×10^{-3}	4.489×10^{-4}	1.092×10^{-3}
1 000	1.001×10^{-1}	9.346×10^{-2}	9.336×10^{-2}	9.331×10^{-2}	9.220×10^{-2}
10 000	8.826	6.590×10^{-2}	17.12	6.362×10^{-2}	7.763
100 000	881.1	7.266×10^{-1}	1740	7.262×10^{-1}	770.4

Table 2: Time in seconds taken to sort arrays using merge sort

Merge sort and insertion sort, which will be discussed next, highlight the inconsistencies between the theoretical results and empirical results. For merge sort, the best case (when the inputs are already sorted or all the data is identical) performs significantly better than the worst case (where the data is reverse sorted), despite the theoretical analysis indicating that they would perform the same.

Table 3 shows the times insertion sort took to sort a variety of inputs of different sizes. Theoretically, insertion sort should be upper bound by ‘n’ in the best case and ‘n^2’ in the worst case.

	Type of Input				
Number of Inputs	Random	Sorted	Reverse Sorted	Identical	Half-random-half-sorted
10	1.758×10^{-5}	6.205×10^{-6}	1.856×10^{-5}	5.430×10^{-6}	1.403×10^{-5}
100	7.692×10^{-4}	3.434×10^{-5}	1.508×10^{-3}	3.411×10^{-5}	9.551×10^{-4}
1 000	7.576×10^{-2}	3.093×10^{-4}	1.358×10^{-1}	3.007×10^{-4}	8.634×10^{-2}
10 000	9.493	4.748×10^{-3}	20.49	4.558×10^{-3}	12.69
100 000	1012	4.567×10^{-2}	2025	4.586×10^{-2}	1252

Table 3: Time in seconds taken to sort arrays using insertion sort

As with merge sort, the best case performs significantly better than the worst case, despite the theoretical analysis indicating that they would perform the same. For both merge sort and insertion sort, it is likely that these results varied so drastically due to the need to make many

memory accesses, comparisons, and memory writes, as opposed to just memory accesses and comparisons that are required for the best cases.

Memory writes clearly create a large overhead in the performance of the algorithm. For this reason, to get a more realistic understanding of how an algorithm will perform, it might be better to analyze it based on the number of memory writes it would incur, even though a memory write might not occur at all in the execution of the algorithm.

Of the three sorting algorithms implemented and executed, merge sort performed the best for most cases on all the data sizes it was tested with, except for the worst case when the input data was in reverse sorted order. For this reason, I would prefer merge sort in the cases where I didn't have much information on the general layout of the data in question.

Insertion sort performed even better than merge sort for the best case, where the data is all sorted already – both of which outperformed selection sort significantly. Since it is advantageous in this case, I would prefer to rely on insertion sort for managing the order of a data set. In cases where new data is added sparsely, say in a small dental clinic where they receive a couple new patients a day and input all their information into the computer system at the end of the day, insertion sort would likely be able to sort this data the quickest of the three algorithms.

The only case in which selection sort performs better than the other two cases is when the data is in reverse order. This would be useful in scenarios, such as if we had a list organized in reverse alphabetical order or had the grades of students arranged from highest to lowest, and this order needed to be reversed. It might also be better to use selection sort if I had to guarantee that the sorting process would be completed within a certain time, since this algorithm performs very consistently across different types of inputs, making it easier to predict with greater certainty when it would finish.