The Comparison of Different Sorting Algorithms

Joseph Hsieh (jch122) and Sutianyi Wen (sw490)

October 9th, 2020

There is a saying, “All roads may lead to Rome”, but some routes to the city may take longer than others. In this paper, we explore five algorithms performing the same task: sorting a list of numerical values in order from least to greatest. What we will see is that although all algorithms were able to complete the challenges set before them, some algorithms were able to perform the tasks more efficiently than others. Why? The answer boils down to “complexity”, and understanding the effects of algorithmic complexity will enlighten us as we build future algorithms.

The algorithms explored are common methods and are known by the following names below, with a brief description of how we implemented each algorithm:

Algorithm	Description
Selection Sort	This algorithm repeatedly loops over the entire unsorted array from left to right, looking for the minimum element per loop and stores the each value in an ordered list.
Insertion Sort	The way we implemented this algorithm, we swapped the *k+1* element with the preceding element until it was in the correct location, bubbling through the list until it was sorted.
Bubble Sort	This algorithm compares every two adjacent elements and swaps those elements if they are out of order. This process iterates until no swaps can be made. In our implementation, we reduce the size of the array iterated by 1 for each loop because the largest element will be bubbled to the end of the array with each iteration.
Merge Sort	A Divide & Conquer approach. This algorithm recursively divides the array into an atomic state, then iteratively compares elements, adding the smaller of the two elements to a sorted list. This is performed on arrays of size *n/2* at a time, reducing the run time.
Quick Sort	Another Divide & Conquer approach. This array starts with a sorted base case then chooses an element as a pivot to partition around, sorting the entire list around the pivot, then recursively resorting around a new pivot in the partitioned segments. In our implementation, we used a random element of the array as the pivot.

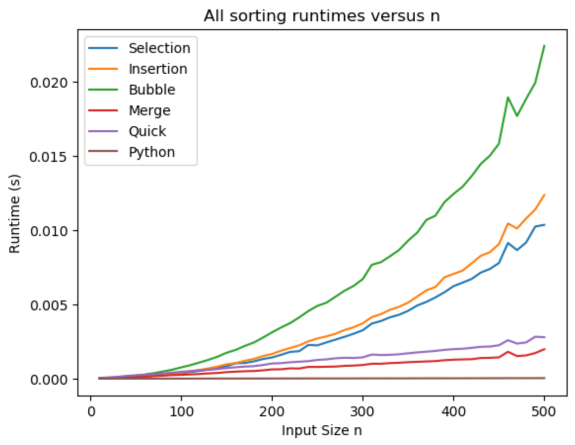
Table 1: Algorithm and Implementation

The average runtimes of the algorithms can be seen in Table 2, evaluated by the slope of Input Size (n) / Runtime (in seconds). In terms of our implementation, our algorithms performed as expected for both sorted and unsorted lists. The slopes calculated have been averaged over 30 trials to mitigate the interference of computer resources being consumed by other programs. The spikes in Graph 1 may have been a result of momentary bursts of other programs consuming processing power. With smaller values of n it was too difficult to differentiate between the different runtimes of the algorithms and no definitive assumptions could be made.Because the processing speeds of modern computing make evaluating algorithms differentiate, the increased trials and larger values of n operations were used to help the algorithms approach their asymptotic run times.

Slope of log(n)/log(runtime)	Sorted Array (n>400)	Random Array (n>200)
Selection Sort log-log Slope	1.90	2.14
Insertion Sort log-log Slope	0.98	2.12
Bubble Sort log-log Slope	0.80	2.07
Merge Sort log-log Slope	0.73	1.13
Quick Sort log-log Slope	0.87	1.06

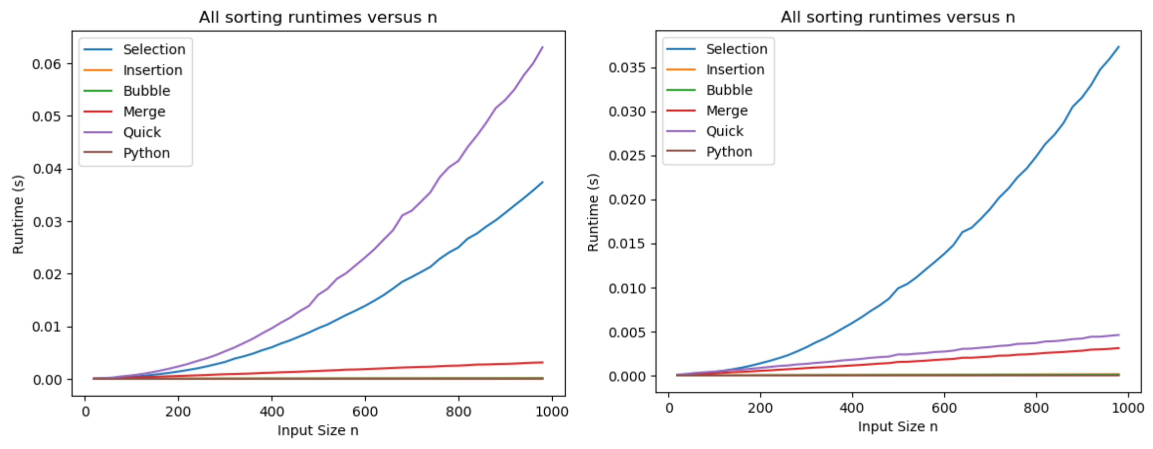
Table 2: Evaluation of Algorithms by Comparing Array Size and Run Time

In doing so, we were able to see the algorithms approach the theoretically average outcome for unsorted arrays and their theoretically best case outcomes for the sorted arrays. A benefit in considering each algorithm theoretically is that each action can be boiled down to a quantifiable Floating Point Operation (FLOP), which allows for easier comparison of performances. Graph 1 illustrates the slope of the input size of the arrays by the average run time of the algorithm, showing each algorithm approaching its theoretical average case outcome. Graph 2(b) illustrates each algorithm approaching it's theoretical best case outcome.



Graph 1: Sorting Speeds of Algorithms by Input Size

Although it is good to consider the theoretical run times, actually performing the experimental run times helped reveal nuances that are not always apparent. Particular evidence of this can be seen in Graphs 2(a) and (b), which highlight the effects of pivot point selection of the QuickSort. In Graph 2(a), the pivot point was originally selected to start with the last element. Although it did not have much bearing on sorting random arrays, seen in Graph 1, pre-sorted arrays effectively making our QuickSort algorithm operate in the worst case scenario. Thus, the closer an array was to being initially sorted, the worse the QuickSort algorithm would operate in O(n^2). This issue was resolved by selecting a random starting pivot point shown in Graph 2(b), allowing QuickSort to have an efficiency of O(n*log(n)).



(a) Last Value as Pivot (b) Random Value as Initial Pivot

Graph 2: Difference in Pivot Point Index of QuickSort: Runtimes on pre-sorted Lists

Overall, based on our implementation, our MergeSort algorithm performed consistently the best. There are some cases where the list is already sorted and MergeSort is outperformed by InsertionSort and BubbleSort, but in real world applications the average case is more often expected, and MergeSort consistently performed the quickest due to using a Divide and Conquer method to have a better average outcome compared to SelectionSort, InsertionSort and BubbleSort. In contrast, SelectionSort performed consistently the worst, as it was expected to have an efficiency of O(n^2) in all cases, regardless of whether the list was unsorted or sorted. SelectionSort was the worst because no matter how the list was organized, the algorithm has to look at every single element for n iterations, resulting in it’s best case and worst case scenario being the same.