Project 1: Sorting

ECE 590-22: Theory and Practice of Algorithms

Varun Prasad (vp60) and Nathan Warren (naw32)

October 9, 2020

Project Overview

The goal of this project was to develop efficient algorithms for the following five popular sorting methods: Selection, Insertion, Bubble, Merge, and Quick. Although each algorithm will succeed in sorting a list in ascending order, the algorithms differ in their runtimes and efficiency. The different complexities are more easily seen on large inputs and on different types of lists, such as lists sorted in reverse order or containing repeated elements. In this project, we analyze our own implementations of these algorithms to better understand their runtimes when given a variety of different types of lists to sort.

Results and Discussion

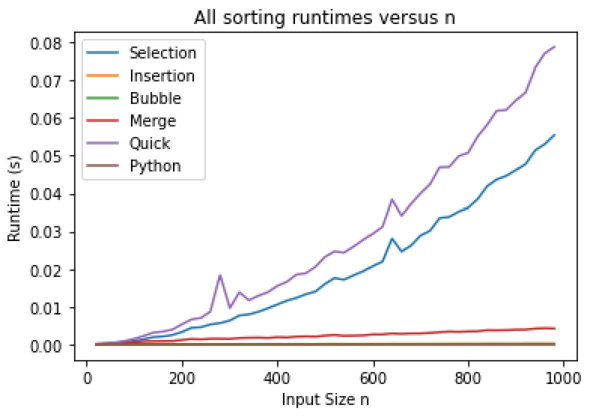
After designing and running our algorithms against our own test cases, we evaluated them using the provided test code. The graphs in Figure 1 show the runtimes of each algorithm for both sorted and unsorted lists of size n ranging from 1 to 1000 and 1 to 500, respectively. For each n, the average runtime over 30 trials is conducted in order to reduce the effect of random noise and other processes that may affect the compute time. Using only one or few trials would not work well for multiple reasons. First, different machines will record different raw runtimes due to their hardware differences. We want to understand how runtime increases with an increase in input size for all machines. Secondly, even the same machine will record different times depending on available computation power. Finally, sometimes algorithms are too fast, especially for smaller inputs, and the built-in timers are not fast enough. To have more confidence in our answer, we can increase the number of trials to lower the standard deviation of our estimate for the runtime.

The algorithms we implemented did indeed run as we expected on both sorted and unsorted lists. When running our algorithms while opening an internet browser we did not see any difference in the runtime but had we opened up something significantly more computationally expensive we predict that the runtime of the algorithms would increase. The results of our runtime for average and worst cases (which depends on both the algorithm and if the list is already sorted), are shown below in Table 1. In our implementation of Selection sort we saw a runtime of O(n^2) for both sorted and unsorted lists. This result is expected since the algorithm performs the same regardless of the input; the list size decreases by 1 with each iteration. For Insertion and Bubble sort we saw a best case runtime of O(n) when using a sorted list and an average run time of O(n^2) when using an unsorted list. This result is expected as there will be no reverse bubbling in Insertion sort and no bubbling for Bubble sort if the numbers are already arranged least to greatest. Our Merge sort algorithm had an average and worst case runtime of O(nlog(n)). This was expected because regardless of what the input is, Merge sort must iteratively break down the list into distinct element lists and re-combine these lists by comparing elements of adjacent lists. Our Quicksort algorithm had an average runtime of O(nlog(n)) and a worst case runtime of O(n^2). This worst case runtime is due to the fact that our algorithm picks the last index in the list to be our pivot value. Since we picked the last index to be our pivot, our pivot is essentially the greatest value in the sorted list. When the largest or smallest element is chosen as the pivot for Quicksort, this leads to a worst case runtime which is O(n^2). These algorithms are visually compared to Python's built-in sorting algorithm called Timsort, which is a hybrid of Merge sort and Insertion sort. Timsort has a best runtime of O(n) and a worst and average runtime of O(nlog(n)). Based on both Table 1 and Figure 1, we believe that Merge sort is the best algorithm since it has consistent high performance on both sorted and unsorted lists. The other algorithms have varying performance between the two types of lists.

Table 1: Log(n)/log(runtime) Slopes for Sorting Algorithms on Unsorted and Sorted Lists

	Log-Log Slope	
Algorithm	Unsorted (n>200)	Sorted (n>400)
Selection Sort	2.05	1.86
Insertion Sort	2.00	0.88
Bubble Sort	2.10	0.88
Merge Sort	1.08	0.94
Quicksort	1.17	1.83

Sorted



Unsorted

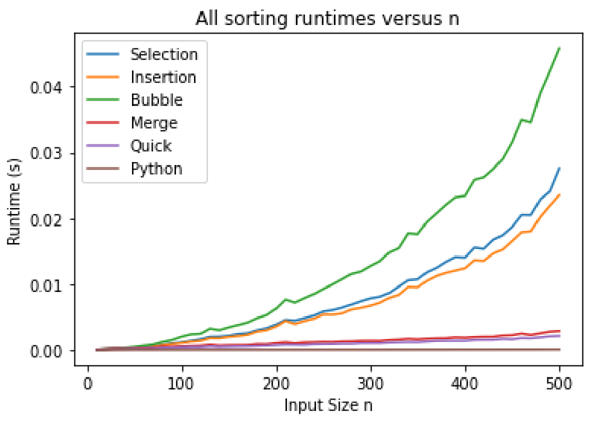


Figure 1: Sorting Algorithm Runtime Comparison for Sorted and Unsorted Lists

The graphs in Figure 1 illustrate the complexities of these different algorithms on sorted and unsorted lists as the size of the input n increases. The Merge sort algorithm performs very well on both sorted and unsorted lists, and is comparable to the Python algorithm. Selection sort shows the expected O(n^2) behavior on both types of lists, while Insertion and Bubble sort perform similarly to Merge Sort on sorted data but have longer runtimes on unsorted lists. The most interesting behavior is shown by Quicksort, which is very fast on unsorted lists but shows O(n^2) runtime on sorted lists. Since our Quicksort algorithm used the end of the list as the pivot, when the list is already ordered, the pivot is the largest value in the list. One of the partitions becomes of size n - 1 with each iteration, effectively making the algorithm similar to Selection sort, leading to an O(n^2) runtime. Selecting the pivot at random or by using the median-of-three rule would likely lead to a better runtime on a sorted list.

To accurately assess the runtimes of these different algorithms, we want to test the algorithms on a large range of inputs. At smaller values of n, it is less likely that the runtime for different algorithms will significantly vary since the input that they need to process is small. The runtime will also be extremely small for smaller units. This can be seen in Figure 1, where for values of n less than approximately 50, the runtime among all algorithms is similar. Only when n becomes large do we see the true differences in complexity for these algorithms. As a result, when considering theoretical runtimes of algorithms, we want to report these runtimes for asymptotically large values of n. For this report, we are considering values of n between 500 and 1000 as the end of our range, and this range is enough to see how the runtimes increase as n increases.

Theoretical runtimes present a more clear picture of how efficient the algorithm is in its best, average, and worst case. In addition, the theoretical runtimes allow us to more effectively compare the efficiency of different algorithms to accomplish a task. Experimental runtimes do not allow us to truly understand an algorithm's complexity because the runtime can be affected by how the computer's resources and processing power are being used. Furthemore, using the Big-O notation to represent theoretical runtime allows us to exclude constants and other factors that ultimately do not affect the runtime at large values of n. However, depending on the task, we may prefer to look at experimental runtimes. For example, if the problem we are trying to solve only concerns smaller lists, we may not be as interested in the theoretical runtime of the algorithms since the runtimes at smaller values of n will be similar. In addition, at smaller values of n, we can incorporate the constants and other factors that are otherwise excluded in Big-O notation. Experimental runtimes are more useful when we know specifically the limits of the task we want to accomplish, whereas theoretical runtimes are more useful in understanding the true long-term behavior of the algorithm.