

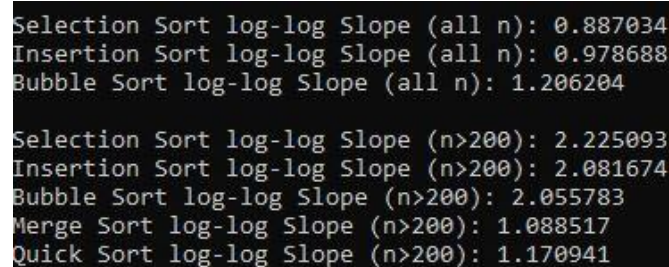
ECE 590 Project 1 Report

10/07/2020

Sang-Jyh Lin (sl605)
Roderick Whang (rjw34)

Sorting Algorithms

Our sorting algorithms can work as we expected to sort all input arrays no matter whether they are sorted or unsorted, and no matter how long the length of the input is. As shown in Fig. 1., when n is large (>200), the log-log slopes of the runtime for Selection, Insertion, and Bubble sort are close to 2, which means they are in $O(n^2)$.



```
Selection Sort log-log Slope (all n): 0.887034
Insertion Sort log-log Slope (all n): 0.978688
Bubble Sort log-log Slope (all n): 1.206204

Selection Sort log-log Slope (n>200): 2.225093
Insertion Sort log-log Slope (n>200): 2.081674
Bubble Sort log-log Slope (n>200): 2.055783
Merge Sort log-log Slope (n>200): 1.088517
Quick Sort log-log Slope (n>200): 1.170941
```

Fig. 1. log-log slope of the runtime for different sorting algorithms

In addition, Fig. 2. can provide us with more information for performance comparison. According to this runtime vs n plot, we can easily find that our best algorithm is Mergesort and Quicksort except Python, their runtimes are similar to each other and do not change a lot even though the size of n grows. Moreover, the trends for every sorting algorithm follows the theoretical complexity result. The theoretical average runtime for Mergesort and Quicksort are $O(n \log n)$, because these two algorithms always divide input arrays into two sub-arrays. On the other hand, Bubble sort is the worst in terms of runtime and its runtime follows $O(n^2)$ and increases fast as the size of n gets larger.

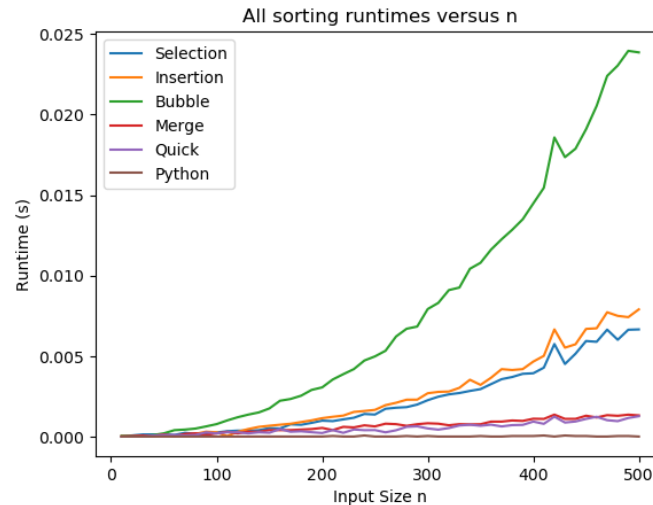


Fig. 2. All sorting runtimes versus n

This report compares algorithms mainly based on the asymptotically large values. This is because that as n increases, the dominant term ($n \log n$ or n^2) will be more influential to the total runtime. Therefore, we can get a better comparison result and estimate how slow the program will be on large inputs to improve our algorithm. On the other hand, we can find that it is hard to tell which algorithm outperforms others when n is small (<100) in the Fig. 2., because all runtimes are too close and any algorithm can become the worst if it happens to get its worst case input. Also any algorithm can become the best if it happens to get its best case input.

As our goal is to compare the performance of the algorithm itself only, in order to reduce the effect from other factors when comparing runtimes, we usually take average runtime across multiple trials rather than using only one trial. The main reason is that if we only use one trial, one “unlucky” algorithm can become the worst if it happens to get its worst case input. Similarly, one algorithm can become the best if it happens to get its best case input. Moreover, based on the central limit theorem, as we increase the number of trials of runtime, a distribution of average runtime would approach a normal distribution, so we can generalize the overall performance well. Therefore, if we want to evaluate a general performance for algorithms, doing average runtime across multiple trials would be a better choice.

Besides the input arrays, other factors that can affect the actual runtime are the hardware settings and software settings. For example, theoretically, we expected that the runtime would be longer if we time our code while performing a computationally expensive task in the background, and a laptop with i7-9750H and 16GB memory can run faster than a 486 computer. However, our experiments show contradictory results to our expectation in Fig. 3. The runtime in silent background settings looks similar runtime in gaming background. We think this might be because today’s cpu performance is very strong, and the game and websites we opened in background does not affect the runtime of our algorithms. (this laptop is with i7-9750H and 16GB memory)

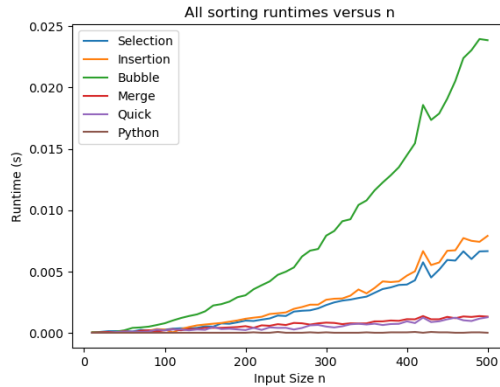


Fig. 3-a. running steam games

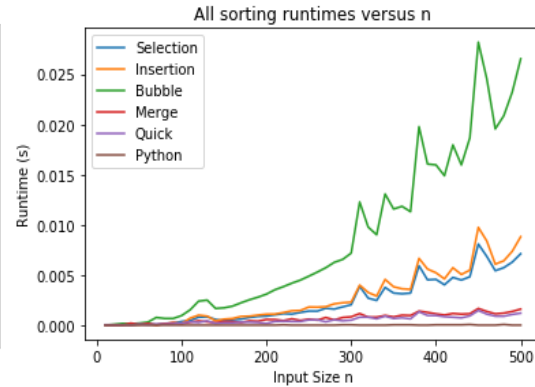


Fig. 3-b. silent background

In general, as discussed above, the actual runtime will be also affected by the hardware settings (CPU and RAM), the software settings (how many background programs we are running), and various input cases (best case or worst case). The actual run time will also contain the parameter for our n , which we do not want to take into our account when comparing the algorithm performance. In order to do a fair comparison, we should control other factors as constantly as we can. Therefore, we should mainly use the theoretical runtimes for algorithm comparisons. Also analysis of theoretical runtimes provides an insight into reasonable directions of search for efficient algorithms. However, if our goal is to find the best algorithm for solving a specific problem in a specific running environment. Since, in this scenario, we will have a better understanding about all of our possible input for our algorithm and the settings for running algorithms, we can use the actual runtime for comparison to find the best fit algorithm for our real-world application.