ECE590/MATH590 PROJECT 1 REPORT

Haohong Zhao, Yifan Li

In this project, we implemented five different sorting algorithms in Python and tested the runtime of the implemented sorting algorithms under different conditions.

Before we tested the performance of all the implemented sorting algorithms, we deemed that quicksort was the best sorting algorithm. QuickSort's overall time complexity is O(n log(n)), which is much better than simple quadratic sorting algorithms such as bubble sort, insertion sort, and selection sort. Furthermore, quicksort is an in-place algorithm that exhibits good cache locality and this makes it faster than merge sort in many cases like in virtual memory environment. Merge sort is not an in-place algorithm and its speed was affected by low memory access speed. In our opinion, bubble sort was the worst algorithm due to the fact that small elements at the end of the array move slowly each iteration.

The algorithms implemented behave as expected for both unsorted and sorted input arrays. As we can see from figure 1, Python's built-in sorting algorithm is the fastest, and merge sort and quicksort also have decent performance. Simple quadratic sorting algorithms like insertion sort and selection sort are much slower, and bubble sort is the slowest. When the input arrays are sorted, we are dealing with the best-case scenarios of sorting algorithms. In theory, among the five sorting algorithms implemented, insertion sort and bubble sort are expected to have O(n) time complexity in the best-case scenario, while other sorting algorithms' time complexity remains the same as the average-case scenario. Figure 2 shows the runtime versus input size when input arrays are sorted, and the test results are consistent with the theory and our expectation. Both insertion sort and bubble sort have huge runtime improvements compared to average-case scenarios.

Fig.1 All sorting runtimes versus input size tested on sorted input arrays averaged over 30 trials

Fig.2 All sorting runtimes versus input size tested on unsorted input arrays averaged over 30 trials

Sorting algorithms are designed to perform well for large dataset. We notice that when input size is small, tested algorithms do not show a significant difference in runtime. This is an expected phenomenon because constant factors play an important role in complexity functions when input size is small. We cannot omit the effect of constant factors when the input size is small.

According to the law of large numbers, the average of the results obtained from a large number of trials should be close to the expected value and will tend to become closer as more trials are performed. Thus, we could get more accurate average-case time complexity by averaging the runtime across multiple trials. In contrast, the results obtained from one single trial would not be representative enough. For instance, on the one hand, It is possible that we get lucky on one trial and the input array is almost sorted. And the results we obtain from this trial could be very similar to the results shown in figure 2. However, this result is not the actual average-case result we expect. On the other hand, figure 3 shows a test result obtained from 1 trial and we can see that the curves are very rugged compared to the curves shown in figure 1. By averaging over multiple trials, we should be able to get a smooth and stable plot to compare the runtimes of different algorithms.

Fig.3 All sorting runtimes versus input size tested on unsorted input arrays of 1 trial

Fig.4 All sorting runtimes versus input size on unsorted arrays averaged over 30 trials when running computationally expensive tasks in the background

Fig. 4 shows the runtime results obtained when running a computationally expensive task in the background. Running sorting algorithms while running a computationally expensive task simultaneously can result in longer and more unstable runtime. As we can see from figure 4 and figure 1, most algorithms' runtime almost doubles and the curves obtained in figure 4 are not as smooth as the curves shown in figure 1 because the CPU occupancy rate of other computationally expensive tasks in the background varies.

The analysis of the theoretical runtimes provides a uniform standard and starting point to compare and choose different sorting algorithms. For example, programmers would prefer sorting algorithms of time complexity O(n log(n)) to the sorting algorithms of time complexity O(n^2). After choosing a certain type of algorithms, programmers would like to pick the best one. And due to the difference of platforms(hardware, compiler, CPU occupation rate, etc.), actual runtimes of some sorting algorithms may vary. Programmers then need to analyze the actual runtimes obtained from the experimental results to main the best decision. For example, the runtime results we obtained in the project show that quicksort is generally better than merge sort, although they both have the time complexity of O(n log(n)). In a word, the analysis of both the theoretical runtimes of algorithms and the actual experimental runtimes of algorithms are necessary for programmers to make the best decisions.