Shrey Agrawal – CSC 301 – Project #1 report

Sorting algorithms

In this report, I will be discussing several sorting algorithms like Selection Sort, Insertion Sort, Bubble Sort, Merge Sort, and Quick Sort that are used to sort elements in ascending or descending order. Each of the algorithms have their own implementation differences which makes their efficiency unique. The report will analyze the behavior of these algorithms, their strengths, weaknesses, theoretical runtimes, and experimental runtimes.

In the case for unsorted inputs, Insertion Sort, Selection Sort, and Bubble Sort are all expected to perform at O(n^2) time complexity in the worst case. The worst case for these algorithms is when the input arrays are unsorted. When calculating the algorithms log-log slope on unsorted inputs, and averaging over 30 trials, insertion sort got a slope of 1.89, selection sort got a slope of 1.51, and bubble sort got a slop of 1.85. Since these slopes are greater than 1, but less than 2, it suggests that the algorithms follow a O(n^2) time complexity. Let's take insertion sort's log-log slope of 1.89 as an example to discuss this. The log-log slope of 1.89 indicates that the time complexity of insertion sort can be expressed as a power law function, where the size of the input array (n) is raised to the power of approximately 1.89. This means that the time complexity grows faster than n^1 (which would be a linear relationship), but slower than n^2 (which would be a quadratic relationship). However, 1.89 when rounded up is 2, so we can say that insertion sort behaves as expected on unsorted inputs, O(n^2). Similar argument can be made for selection sort and bubble sort given the slopes of 1.51 and 1.85 respectively. Out of these three algorithms, according to the loglog run time analysis done in the implementation I did, selection sort seems to be performing the best on sorted inputs. This matches with the general notion that selection sort performs better than bubble sort and insertion sort on unsorted arrays, as it has the smallest number of comparisons, but all three algorithms are inefficient for large, unsorted arrays. Algorithms that are more efficient for large, unsorted arrays are merge sort and quick sort. In terms of big O notation, both algorithms have a time complexity of O (n log n). As seen in figure 1 below, the run time for merge sort and quick sort do not grow as quickly as they do for insertion sort, bubble sort, and selection sort, as the size of the unsorted input array increases. Most importantly, the run time does not look quadratic.

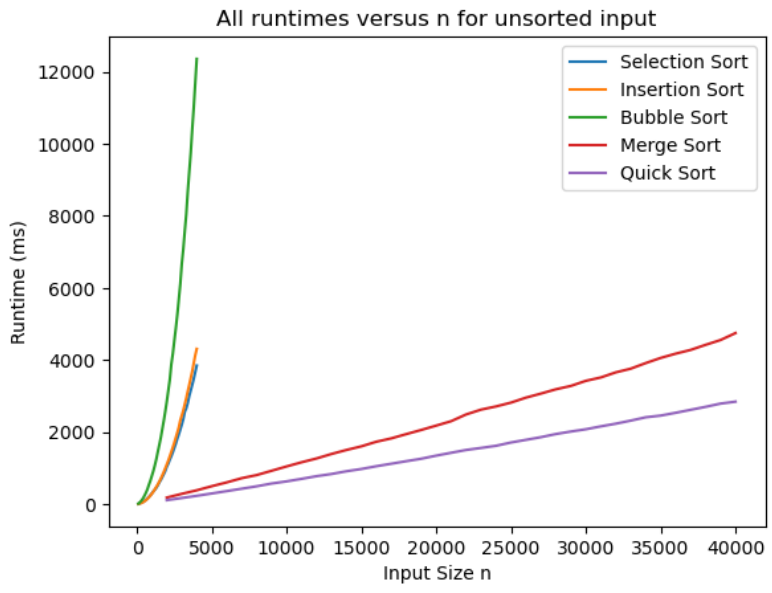

Figure 1

Sorting algorithms

On the other hand, on already sorted input arrays, bubble sort and insertion sort can achieve their best-case time complexity of O(n), where n is the size of the input array. This is because the algorithms can detect that the input array is already sorted and terminate early, without having to perform any swaps or comparisons. However, even with a sorted array (best case), the time complexity for selection sort is O(n^2). When calculating the algorithms log-log slope on sorted inputs, and averaging over 30 trials, bubble sort got a slope of 1.13, insertion sort got a slope of 2.03, and selection sort got a slop of 1.89. Since the slope of bubble sort is quite close to 1, it suggests that the algorithm follows a O(n) time complexity, which is expected for sorted inputs in bubble sort. To my surprise, the loglog slope of insertion sort was close to 2, rather than 1, suggesting that my implementation of insertion sort algorithm follows a O(n^2) time complexity, which is unexpected for sorted inputs in insertion sort. I only realized this now after having turned in the code. Selection sort's best case (sorted input) tends to follow a time complexity of O(n^2), which is indicative by the loglog slope of 1.89, and is expected. However, all three of these algorithms are inefficient for larger sorted arrays. In contrast, merge sort and quick sort have a time complexity of O (n log n), which means their efficiency scales better with larger input array sizes. For quicksort, the worst case is O(n^2) and it happens if the pivot is the smallest or largest. The way the pivot was chosen in my implementation was that it was the first element in the array. In a sorted array input, this would be the smallest element, leading to the worst case. When calculating the algorithms log-log slope on sorted inputs, and averaging over 30 trials, quick sort got a slope of 1.86, which is expected and indicative of a O(n^2) time complexity for the worst case of quick sort.

In my opinion, and according to figure 1, the best sorting algorithm out of these is quick sort. The run time seems to be the lowest at larger inputs of unsorted array. Merge sort is also very similar in terms of run time achieved, especially for smaller inputs but quick sort becomes more efficient as the input size grows. However, both merge sort and quick sort are better than insertion, bubble, and selection sort due to their O(n log n) time complexity as compared to O(n^2). On the other hand, bubble sort is the worst sorting algorithm because of its O(n^2) time complexity. Bubble sort is inefficient when sorting large arrays because it takes a long time to complete. From figure 1, we can see that it has the steepest and fastest growing curve.

Theoretical runtimes refer to the expected runtime of an algorithm for asymptotically large values of n. We report theoretical runtimes for large values of n to understand how the algorithms scale as the input array size increases, and potentially reaches infinity. However, the theoretical runtimes may not accurately represent the algorithm's behavior for small input sizes. For smaller values of n, the runtime may be faster or slower than the theoretical runtime because of factors such as the CPU power and architecture, background tasks, and cache performance. The same algorithm will probably run faster on a newer more powerful machine than an old machine with outdated hardware. Experimental runtimes refer to the actual runtime of an algorithm under specific conditions, such as specific input array and computer hardware. This is why it is important to average the runtime across multiple trials to account for variability in the system's performance. Different trials may produce different results due to factors such as CPU utilization, memory allocation, and thread management. Taking 30 trials helps us to get a better picture of how our implement algorithm performs.

Sorting algorithms

In addition, experimental runtimes will not only vary across different machines, but change based on background tasks being performed by a machine when the algorithm is executed. This was evident when I opened 6 Firefox tabs during the run of the algorithm. Since the resources on a machine are limited, they are shared across all processes that occur, and hence, adding a process in the background increased the run time of the sorting algorithms. This is another reason why actual or experimental run times are not reported when talking about time complexity of any algorithm.

Hence, we analyze theoretical runtimes for algorithms to compare their time complexity and scalability. Theoretical runtimes provide useful comparisons because they help us understand how the algorithm behaves as the input size increases, since they are reported in terms of the size of the input (n in most cases). However, experimental runtimes provide more useful comparisons for specific use cases. For instance, if we are sorting a small input array, experimental runtimes may provide a more accurate comparison of the algorithms' performance. Also, if we are interested in seeing how CPU architecture or memory allocation techniques influence the run time of our algorithm and are interested in comparing it across different machine architectures, experimental runtimes are helpful since these are not considered in theoretical runtimes.

To conclude, we can say that the efficiency of sorting algorithms is highly dependent on the size and order of the input array. Insertion sort, selection sort, and bubble sort are all inefficient for large, unsorted arrays, with a worst-case time complexity of O(n^2). Merge sort and quick sort are more efficient, with a time complexity of O(n log n), making them suitable for larger input sizes. However, it's important to note that the theoretical runtimes may not always be reflective of the actual runtimes, as there can be variations due to implementation details and hardware limitations.