Chayut Wongkamthong
netid: cw403

Sorting Algorithms

October 23, 2019

A. Executive Summary

In this project, we implemented five different sorting algorithms, evaluated the runtime and compared the result with the theoretical complexity. All of our implementations performed as per expectation for both sorted and unsorted arrays. We discussed advantages of using Quick Sort as the most appropriate sorting algorithm in most cases. We also discussed the measurement of runtime and how to properly record and compare it with theoretical complexity.

B. Implementation Runtime

The table below summarizes the theoretical runtime for different sorting algorithms that we implemented in this project. This information serves as the baseline for our comparison.

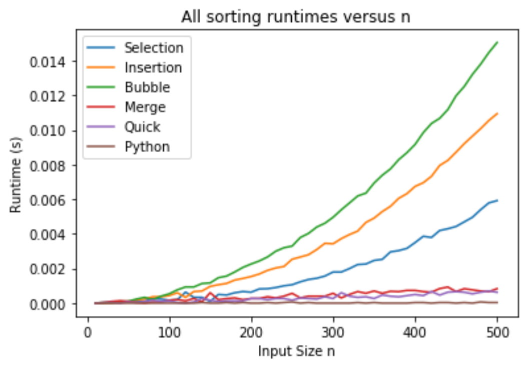
No.	Algorithm	Avg Complexity	Best Case Complexity	Worst Case Complexity
1	Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
2	Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$
3	Bubble Sort	$O(n^2)$	$O(n)$	$O(n^2)$
4	Merge Sort	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$
5	Quick Sort	$O(nlog(n))$	$O(nlog(n))$	$O(n^2)$

The table below shows the result of our implementation in term of the slope of log(n) vs log(T) for cases where we input unsorted arrays and sorted arrays.

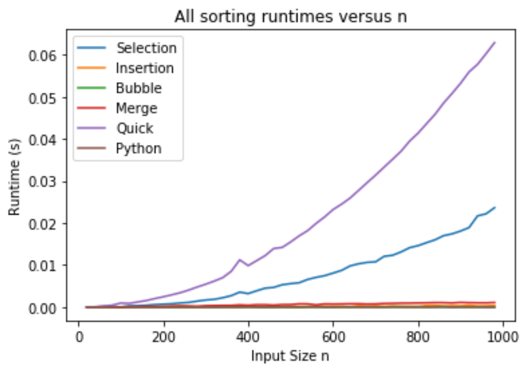
No.	Algorithm	Slope (Unsorted)	Slope (Sorted)
1	Selection Sort	2.04	2.05
2	Insertion Sort	2.06	1.16
3	Bubble Sort	2.01	0.80
4	Merge Sort	1.12	1.07
5	Quick Sort	0.96	2.11

Let's consider the normal case where our inputs are unsorted arrays. For the algorithms that are of $O(n^2)$ like Selection, Insertion, and Bubble Sorts, we expect to see the slope of approximately 2 when we plot array length and runtime in log-log scale. The results here showed what we would expect. For Merge sort and Quick sort, if we consider large value of n where $log(nlog(n)) \approx log(n)$, we can expect to see the slope on the log-log scale that is close to 1 as we obtained here.

Next for the sorted input array, this kind of input will serve as the best case for insertion sort (since it doesn't require any rearrangement) and bubble sort (since it doesn't require any swaps on the first loop). Since the best case complexity of both is of $O(n)$ complexity, we can see that the result showed the slope of approximately 1 for both cases.



(a) label 1



(b) label 2

Figure 1: Left: Unsorted Input, Right: Sorted Input

However, Selection and Merge sorts are invariant of the input and will show the same algorithmic complexity for all best, average, and worst case; therefore, we can see that the slopes here didn't change much.

Lastly, since in this implementation we use the last element in the array as the pivot for partitioning in Quick sort, the sorted input array will serve as the worst case. This is because we cannot have a balanced partition. (we use the highest-valued element as the pivot every time) Therefore, the slope in this case will approach 2 which is the slope of $O(n^2)$ in log-log scale.

These results are illustrated in Figure 1. In general case, we obtained better runtime using Merge sort and Quick sort (their runtimes actually approach that of python implementation). The other algorithms show quadratic trend. For the sorted input, it is the worst case for our implementation of Quick sort as suggested by the increase in runtime to the quadratic trend. While for Insertion and Bubble sort, this kind of input serves as the best case which can be seen by dramatic improve in the runtime.

In this analysis, we can clearly see that Quick sort and Merge sort will outperform the other three in general cases. While Quick sort shows worse algorithmic complexity than Merge sort in the worst case, it can be implemented in-place relatively easily. Moreover, we can always check whether the input array is already sorted before implementing helping us prevent the worst case. These are reasons why we believe that Quick sort serve as the best sorting algorithm in general.

In contrast, Selection sort is the only algorithm here that cannot perform better than $O(n^2)$ in any given situation. Therefore, in our opinion, it serves as the worst sorting algorithm being compared here.

C. Discussion of Runtime Measurement

In complexity theory of algorithms, we normally report the theoretical runtimes for asymptotically large values of n as we did in the first table. This is because we are usually interested in determining how slow our programs will be when we have large inputs.

This idea of reporting theoretical runtime is well adopted and usually is preferable over reporting the actual runtime on experiment. One main reason is that actual runtime highly depends on the system on which your program is runing. For example, a more complex algorithm might achieve better runtime in supercomputer when compared with the optimal algorithm on an out-of-date architecture. Moreover, it also depends on the other tasks being perform on the system at that time. For instance, the same algorithmic implementation may have worse runtime when performing its job while we are doing computationally expensive tasks like web scraping in the background (We did get worse outcome while openning several browsers while measuring runtime). The effect will be more pronounced when we have system with limited resources like one core for processing or limited RAM/CPU. Therefore, when we want to compare the complexity of different algorithms in general, being able to report complexity without depending on specific systems is preferable.

Regardless of these shortcomings, when we have a dedicated system for performing a specific task and we want to measure the real runtime to optimize other processes or compare two different programs, measuring the actual runtime will become valuable. For example, in a big company where we have a supercomputer to perform a simulation of real-time market volatility, measuring how long we typically need to wait for each of possible simulation algorithms can help us decide which choice we should go for to minimize downtime.

In this project, we didn't report slope for small value of n since the runtime is fluctuating (for small value of n, it is easy to have a random input that is close to either best or worst case for the algorithms). Therefore, we cannot generalize the result in this case. Another reason is that we cannot compare this outcome with theoretical behavior due to the fact that, with small n, other terms in the total cost like linear term will become relatively significant.

We also averaged our measured runtimes over 30 trials of random input to obtain the average value. If we use only 1 trial to evaluate a program, we found that some slopes were negative and the runtime was fluctuating. This is because it is prone to outliers from getting best or worst inputs from the random process (If we think of this statistically, having more trials should help us reduce the variance in our measures).