Project 1: Sorting

Suqian Wang

sw443@duke.edu

1 Abstract

Sorting algorithms are crucial for any kind of data processing. Popular sorting algorithms such as Selection Sort, Insertion Sort, Bubble Sort, Merge Sort, and Quick Sort create a foundation for other more complex sorting algorithms. This project is intended to investigate the performance of sorting algorithms against various inputs.

2 Overview

The purpose of this project is to implement five sorting algorithms: Selection Sort, Insertion Sort, Bubble Sort, Merge Sort, and Quick Sort, as well as, investigate how the actual performance of each algorithm differs depending on its inputs. To do this, the project will take these algorithms and test them against a series of lists. The lists that will be tested represent different real-world conditions that these algorithms must handle to be an effective algorithm, from large lists to lists with repeated element. The experiment is also intended to illustrate a comparison between each algorithm as they have different theoretical time complexities, as well as, compare against the theoretical runtimes discussed in lecture. The plots may show which algorithm fares better or worse depending on the time taken to complete each trial of the experiment.

3 Theoretical Analysis

Each sorting algorithm has its theoretical runtime.

1. Selection Sort
   Selection Sort iteratively searches for the minimum element from the unsorted portion of an array and swaps that element with the first unsorted element, so that each iteration, there will be a minimum element sorted into the sorted portion of the array. This algorithm performs the same regardless of the input, so the runtime of this algorithm is $O(n^2)$.

2. Insertion Sort
   Insertion Sort takes the first element of the unsorted portion of the array and swap it with the preceding element until it is in the correct location. This algorithm is input dependent, if the input is an already sorted array, each iteration there will be no swapping performed, and the performance would be the same as traversing the array once. So the best case runtime of this algorithm is $O(n)$. The worst and average case runtime require each element to be moved to the very beginning or half-way of the sorted portion which will lead to a runtime of $O(n^2)$.

3. Bubble Sort
   Bubble Sort iterate through the array, compare every two adjacent element, and swap them if they are in reversed order. The best case would be dealing with a sorted array where we just need to traverse the entire array once, so the runtime would be $O(n)$. The worst and the averaged case would require $O(n^2)$ runtime.

4. Merge Sort
   Merge Sort recursively breaks the array in two halves and sort each half, and it merge the sorted halves back. The merging takes $O(n)$ time so the recursive definition of the runtime is $T(n) = 2T(n/2) + O(n)$ which solves for a complexity of $O(nlogn)$

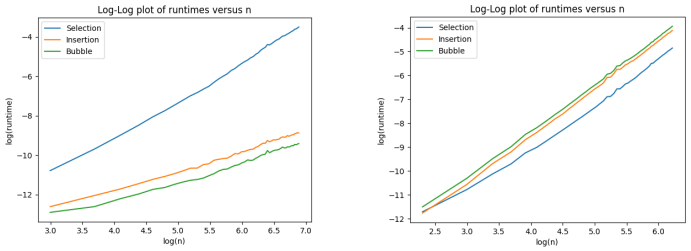
5. Quick Sort
   Quick Sort recursively partitions the array based on a pivot value so the left half of the array has elements no more than the pivot and the right half of the array has elements no less than the pivot. The worse case would be when the pivot value we choose each recursion were the smallest or the largest element of that array. In this case, one of the partition would have a size of original size - 1, and Quick Sort behaves like selection sort and leads of a runtime of $O(n^2)$ This is very unlikely to happen if the array is randomized and our selection of pivot is random. In most cases (best and average case), Quicksort have a time complexity of $O(nlogn)$.

4 Experimental Results

To verify an algorithm with polynomial runtime, we can plot the runtime as a function of input size n in a log-log scale and check the plot(which should be a straight line) and the slope value. For example, for an algorithm with a runtime of $O(n)$, if we plot the runtime vs. size n in a log-log scale, we should see a straight line with a slope of 1.

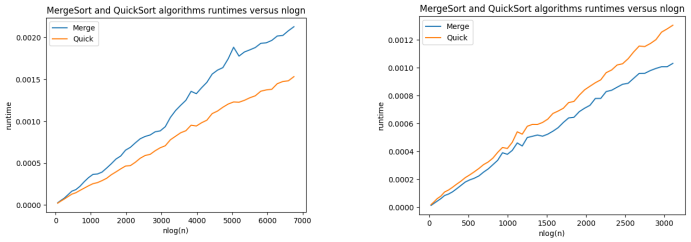
Similarly, we would see a straight line with a slope of 2 if the runtime is of $O(n^2)$.

To verify an algorithm with runtime $O(nlogn)$, we can plot the runtime as a function of $(nlogn)$ and see if the plot is a straight line. If we still do the log-log plot of this algorithm. When taking the log of both sides gives us $logT \sim logn + log(logn) + O(n)$. If we plot $y = logT$ vs. $x = logn$, we should see it approaches a straight line with a slope slightly greater than 1 when n is getting larger.



(a) Sorted input (b) Unsorted input

Figure 1: Selection, Insertion, Bubble Sort runtimes in log-log scale



(a) Sorted input (b) Unsorted input

Figure 2: Merge and Quick Sort runtimes vs. nlogn

Algorithms	sorted Input		unsorted input	
	All n	n > 400	All n	n > 200
Selection Sort	1.962817	2.067557	1.853286	2.084438
Insertion Sort	1.019483	1.104165	1.988107	1.977233
Bubble Sort	1.002985	1.094197	1.966474	1.995665
Merge Sort	1.163212	1.195573	1.160045	1.050109
Quick Sort	1.110472	1.139319	1.111344	1.100691

Figure 3: All sort algorithm log-log slope

- When the input array is already sorted(best case scenario):
  From Figure 1(a) and Figure 3:
  - Selection Sort have a slope ~ 2, which means its runtime is $\sim O(n^2)$.
  - Insertion Sort and Bubble Sort have a slope ~ 1, which means their runtime is $\sim O(n)$.

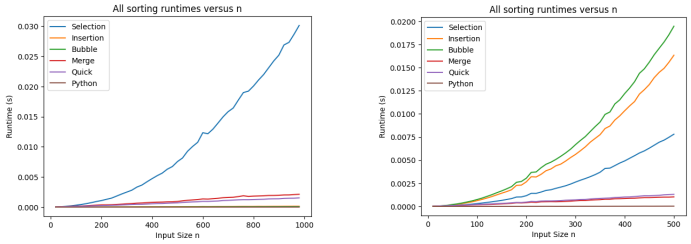
  From Figure 2(a) and Figure 3:
  - Merge Sort and Quick Sort both are linear when ploting their runtime against $nlogn$, and their log-log slope is slightly bigger than 1, this prove their runtime is $\sim O(nlogn)$
- When the input array is unsorted:
  From Figure 1(b) and Figure 3:
  - Selection Sort, Insertion Sort, and Bubble Sort all have a slope ~ 2, which means their runtime is $\sim O(n^2)$.

  From Figure 2(b) and Figure 3:
  - Merge Sort and Quick Sort both are linear when ploting their runtime against $nlogn$, and their log-log slope is slightly bigger than 1, this prove their runtime is $\sim O(nlogn)$

Thus, we can conclude that the algorithms behave as expected for both unsorted and sorted input arrays.

We can also notice there are some differences between the slopes when we counted all n values versus when we counted only the larger n values, the fitted line is more accurate when using only the larger n. For example, the Selection Sort is more close to 2(2.08 vs. 1.85) when only using the larger n. To explain this discrepancy, we need to look into the Big-O notation's definition

The Big-O notation denotes the growth of the algorithm instead of the execute time of the algorithm. In the Big-O notation, we are ignoring the lower-order terms. When n is small, these lower-order terms actually contributes to the function's execution time. When n is large, the highest-order term dominates the total time, and the overhead can be ignored. Thus we only report theoretical runtimes for large values of n.



(a) Sorted input (b) Unsorted input

Figure 4: All algorithms runtimes vs. n

From Figure 4, when the input arrays are sorted, Insertion Sort and Bubble Sort performs the best. This is because the input is already sorted and Insertion and Bubble Sort will only need to traverse the array once and result in a $O(n)$ complexity.
Selection Sort is performing very bad even when the size of the array is not super large. This matched with our theoretical analysis because it is input dependent and will

always perform at $O(n^2)$.

When the input arrays are unsorted, Merge Sort perform the best and Quick Sort are the 2nd best and very close to the Merge Sort performance. This is because of their divide-and-conquer technique that reduced their time complexity to $O(nlogn)$.
Meanwhile, Bubble Sort performs the worst. That is because in average cases, Bubble sort has a complexity of $O(n^2)$. Intuitively, Bubble Sort requires quite a lot of comparisons and swaps in its algorithm (the "bubbling" process). So when dealing with unsorted large dataset, Bubble Sort's runtime increased drastically.
The Insertion Sort algorithm is not much better because it also involving the "bubbling" process which increased its runtime.

Overall, Merge Sort and Quick Sort have good enough performances, and they are stable and consistent regardless the characteristic of the input data. Even in the cases of sorted input data, they still come very close to the $O(n)$ performance of Bubble Sort and Insertion Sort.

We can also see the runtime between different algorithms are very similar when n is small. When n is small, the algorithms finishes very quick, so the complexity difference doesn't show. Also as I mentioned before, the overhead and constant factors that was ignored in the Big-O notation can not be ignored and actually matters when n is small. For example, an algorithm that has $O(n)$ complexity with more overhead (more times per operation) might even be slower than an algorithm that has $O(n^2)$ complexity with less overhead.

5 Discussion

From this project, we can see that even though the actual runtime behaves like we expected in our theoretical analysis, there are differences between them. There are many factors such as processor, background processes, RAM limitations that can affect the actual runtime of your program. Thus we need to average the runtime across multiple trials for our result to be accurate. If we only use one trial you are only taking the data for that particular moment of time where your machine may be allocating memory or resources into other operations. Therefore, having multiple trials and averaging out those times ensures a overall result of how the algorithm performs. Related to this idea, if a machine is performing a computationally expensive task in the background it may affect the timing of the code. Computationally expensive processes could take away resources or may stall your program. Therefore, slowing down the algorithm and resulting in a different time versus if all resources were free.

In an experimental runtime analysis, we have to actually implement the algorithm. However, there are many ways to implement the same algorithm that could produce different runtimes, and implementing the algorithm takes more time than mathematical analysis of a pseudocode. We have to also consider all possible inputs and cases when we actually implementing the algorithm and calculate the runtime, and we might miss a few scenarios which might cause error in measuring the runtime.
When comparing algorithms experimentally, different hardware and software environment can result in different runtimes across machines. Also, in theoretical runtime analysis, we use Big-O notation to represent the runtime complexity. Big-O notation does well to capture the growth of an algorithm, the caveat being for large values of n. To reproduce a result similar to the theoretical runtime, the actual implementation of algorithm may take excess time to complete. Thus, it is arguably better to use theoretical analysis rather than experimental analysis. Although theoretical runtimes were used more often for comparing the growth of algorithms when n is large. There are cases where experimental runtimes provide more useful comparisons. For example, we might want to compare the speed of two algorithms that has the same theoretical runtime. Or we might want to see the performance of an algorithm when dealing with a small input size, in which case the Big-O notation doesn't matter when the constant factor and overhead dwarf the performance.