P1: Experimental Analysis of Sorting Algorithms

This project centered on the experimentation and breakdown of four main sorting algorithms: insertion sort, merge sort, quicksort, and randomized quicksort. Each algorithm was given 16 data sets of varying n values and orders of the integers within them. There were four different orders of the numbers: randomized, sorted, reversed, and identical. Randomized meant every number was completely randomly generated. Sorted meant that the random numbers were all in the order from smallest to largest. Reversed meant all the numbers were in order from largest to smallest. Identical meant every number was the same. Each one of these four types also had varying sizes of n: 10,000, 20,000, 40,000, and 80,000. Below I will breakdown each algorithm individually on the expected performances, actual performances, and a comparison between the two.

**Insertion Sort**

First up was insertion sort. Insertion sort has a best case of , an average/expected case of , and a worst case of . The best case for insertion sort would be that the data is already in order and does not need sorted. This is present in our data sets of sorted data and identical data. Both should theoretically achieve . The worst case possible would be that the data is in reverse order. This is present in the data set of reverse order data which should theoretically achieve . Below is the data of the actual achieved results.



Figure 1.1: Actual results of implementation of insertion sort.

You will see that as described above the best cases of the sorted data and identical data were perfect, both achieving or about 80,000 for the highest n on their comparisons which was the theoretical best case. The worst case possible was theoretically when the data was in reverse order. Above in Figure 1.1 we see that we didn’t quite achieve as expected and instead got about half of that. The average case listed would theoretically apply to the randomized data set since it being completely randomized means it on average would achieve the most average case scenario. Theoretically this would be but above we see we had an actual value of about ¼ of that for our comparisons made. We can see that the results didn’t completely match up with our theoretical values with the biggest anomalies being the average and worst cases not reaching their expected values. This surprises me since we made the actual worst case possible for this algorithm a reality and it didn’t manage to achieve its worst case or even average case.

**Merge sort**

Next up is merge sort. Merge sort has a best case of , an average/expected case of , and a worst case of . As shown, every case of merge sort will theoretically return the same number of comparisons since its worst, average, and best cases all have the same value of . That means regardless of which of our four different randomized data sets it is run on, it should return the same number of comparisons in relation to n.



Figure 1.2: Actual results of implementation of merge sort.

In Figure 1.2 you see that every single type of data set achieved the same number of comparisons if they had a matching number n. This matches with the fact that we decided theoretically they should all share the same performance at every case since its worst, average, and best cases are all the same. For all cases where n = 80,000 I got a number of comparisons of 1,228,929. This is slightly below the expected number of = 1,303,016. This is overall slightly lower across the board than every expected performance but the consistency across each data set shows that we achieved a good actual implementation of the merge sort algorithm. Overall, I am pleased with these numbers and but was a bit surprised that I was unable to get closer to the expected values by tweaking my algorithm and its counter.

**Quicksort**

Next is quicksort. Quicksort has a best case of , an average/expected case of , and a worst case of . This algorithm holds a higher worst case scenario performance than that of the randomized quicksort which brings down its worst case by grabbing a random pivot each time rather than selecting the same destination of the pivot each iteration which doesn’t help balance performance at the higher end.



Figure 1.3: Actual results of implementation of quicksort.

As seen in Figure 1.3 quicksort returned the overall highest comparison numbers of all the implemented algorithms in this project. The worst case was thought to be but the highest we achieved was half of that with the identical and sorted data sets. For the average, which would correspond to the randomized data set, we expected to get about = 1,303,016 but we got 1,557,700 which is slightly above what we thought but since it is somewhat close this appears to be a good implementation. Overall, the numbers appear good and somewhat consistent with its values in comparison to the theoretical values, as well as compared to the other algorithms and their comparisons to their theoretical values.

Finally, for this project is randomized quicksort. Randomized quicksort has a best case of , an average/expected case of , and a worst case of . The algorithm differs from the regular quicksort algorithm by bringing down the expected worst-case scenario from to  by randomizing the pivot rather than grabbing the same element each time which helps to average out on the upper end of the possible performance.



Figure 1.4: Actual results of implementation of randomized quicksort.

In Figure 1.4 we see that randomized quicksort managed to outperform quick sort which was expected. The worst-case performance was theoretically for this algorithm but we see here we far surpassed this in both the identical and reverse data sets which achieved closer to the worst-case performance of standard quicksort. As for the average case performance we landed close with an expected performance of = 1,303,016 but an actual value of 1,558,480 in the randomized data set. This is close to the average which can vary since different data sets return different performance and there is room for variance. Overall, this algorithm performed well but the worst-case performance seemed to have been broken which appears to be strange given what we know.

This project went well and gave great insight on four of the most common sorting algorithms used. We achieved great performance in almost every algorithm we implemented and tested and created points to raise questions in the future about different algorithms and their behavior. This project has great detail on all four implementations and the implementations provide a great tool for people looking to learn more about their performance.