Name: Ariyan Kumaraswamy

Mystery Sort Lab Report

The Problem:

This project involves determining the names of 5 sorting algorithms that have been arbitrarily named mystery01, mystery02, etc. We know that the algorithms are bubble sort, insertion sort, merge sort, quicksort, and selection sort. However, we don't know which of these algorithms corresponds to the mystery algorithms. We can identify these algorithms using runtime analysis.

Strategy:

The strategy I opted to use was to create a 'Sorter' class that would allow me to collect runtime data on the mystery algorithms. The goal was to vary the input size and run each algorithm with the same datasets, then compare the resulting runtimes. I chose to use 3 configurations for the data: sorted, reversed, and randomized. It is important to vary the dataset ordering because most of the listed algorithms will perform better with some configurations and worse with others. I selected these 3 because many of the algorithms have a best-case runtime when run on a sorted set, a worst-case runtime when run on a reversed set, and an average-case runtime when run on a randomized set. The data sizes I chose ranged from 10,000 inputs to 150,000 inputs. Varying input size is necessary because the efficiency of these algorithms changes based on how much data there is to sort.

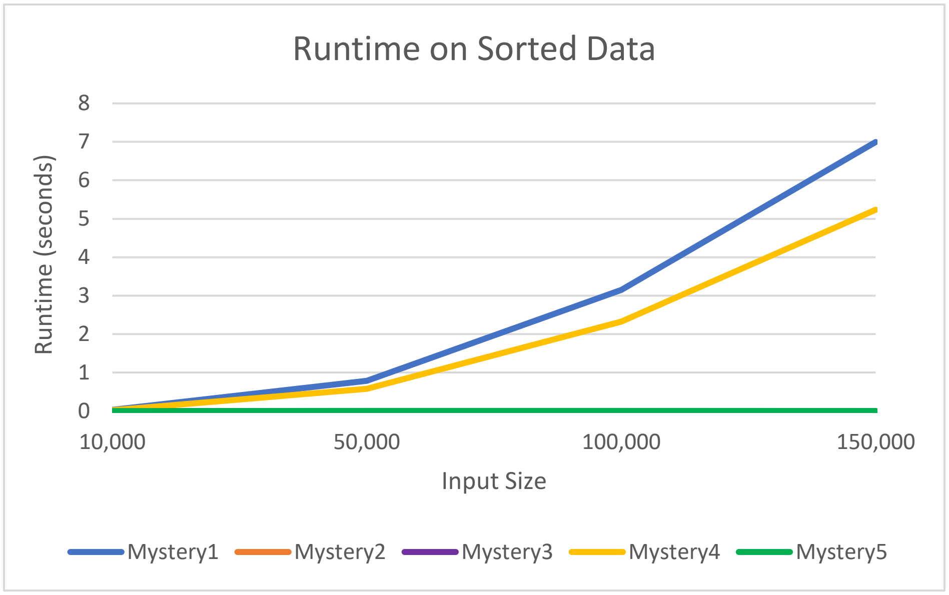
The program generates the datasets based on the specified input size, runs each mystery algorithm, then reports the time in seconds that it took for each.

Results:

Here is the runtime data I collected of each algorithm at each input size, organized by data configuration:

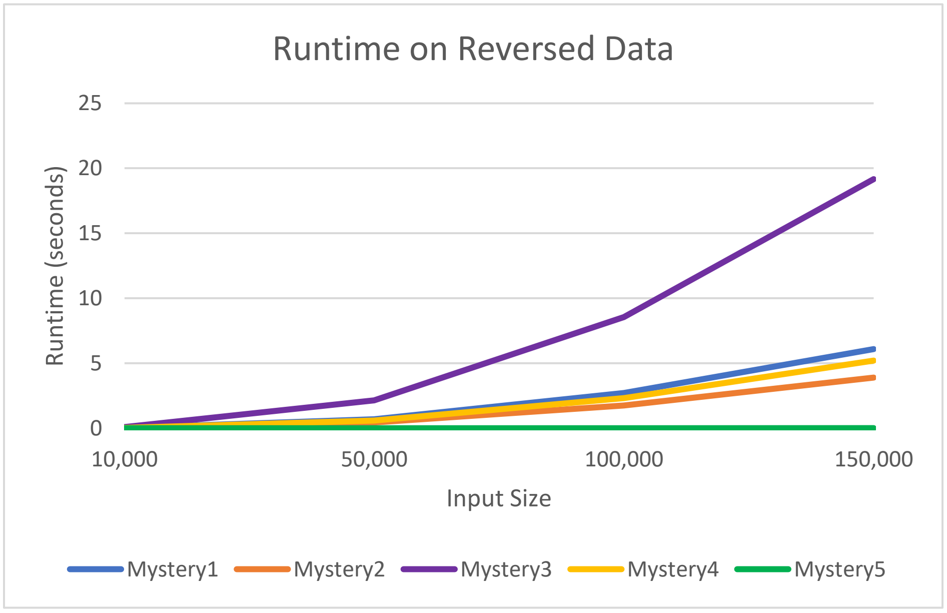
Sorted

Algorithm	10,000	50,000	100,000	150,000
Mystery1	0.0325886	0.7837318	3.1493653	6.9959426
Mystery2	0.0000108	0.0000494	0.0000965	0.0001585
Mystery3	0.0000116	0.0000309	0.0000637	0.0000871
Mystery4	0.0231069	0.5773368	2.3278277	5.2345848
Mystery5	0.0001908	0.0010169	0.0021041	0.0031721



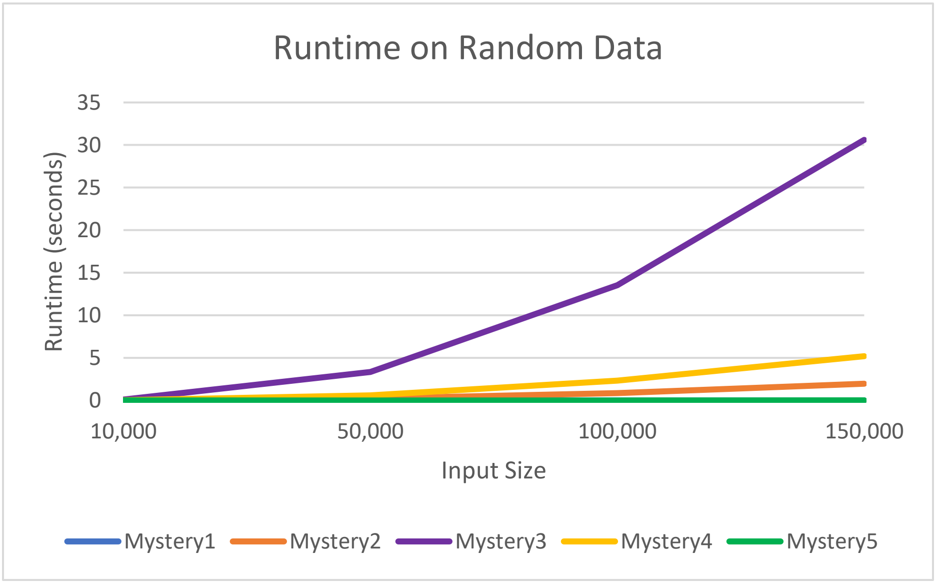
Reversed

Algorithm	10,000	50,000	100,000	150,000
Mystery1	0.0272193	0.6779258	2.7161209	6.0842693
Mystery2	0.0173042	0.4328797	1.7563118	3.891091
Mystery3	0.0859465	2.1310991	8.5362993	19.163495
Mystery4	0.0230591	0.5794076	2.3092476	5.1994866
Mystery5	0.0002339	0.0010202	0.0021097	0.0031796



Random

Algorithm	10,000	50,000	100,000	150,000
Mystery1	0.0003852	0.0021826	0.0046041	0.007204
Mystery2	0.00882	0.2212636	0.8679265	1.960401
Mystery3	0.1004025	3.3308358	13.5367907	30.6065748
Mystery4	0.0234525	0.5827591	2.3103918	5.1923992
Mystery5	0.0005565	0.0031984	0.0068131	0.0105697



Analysis and Discussion:

My best estimation of the mystery algorithms is as such:

- Mystery Algo 1 = Quicksort
- Mystery Algo 2 = Insertion Sort
- Mystery Algo 3 = Optimized Bubble Sort
- Mystery Algo 4 = Selection Sort
- Mystery Algo 5 = Merge Sort

My reasoning for quicksort and merge sort's classification is the simplest. Both are the only 'divide and conquer' algorithms in the list, which means they split up the data into smaller pieces that are much easier (and faster) to sort. Because this quicksort implementation uses the end value as the pivot, it's runtime is much longer when handling sorted and reversed data since it is most efficient when the pivot is the median of the list. As such, this algorithm has a worst-case runtime of O(n^2). As we would expect, it becomes much faster on a randomized list, where its endpoint pivot is much less likely to hinder its partitioning. Merge sort is likely the 5th algorithm because it is by far the most efficient when handling much larger input sizes of random and reversed data. Although it isn't always the fastest, the fact that its runtime never exceeds 1 second is proof of its O(n log n) complexity. It is also understandably slower when handling very small input sizes, where halving the data is efficient, which I confirmed by testing the algorithm with 10 inputs. Insertion sort is the 2nd algorithm because its best-case is when it's handling sorted data (at O(n)), which is exactly what we see in the tables. Its worst-case is when dealing with reversed data (at O(n^2), which is also shown by the tables. Optimized bubble sort is the 3rd algorithm because it is by far the fastest when dealing with sorted data. Since this is an optimized version of the algorithm, it stops running after discovering that the data is already sorted, which brings its best-case runtime to O(n). However, it is by far the slowest when handling larger input sizes since it has to make several passes over all the values before it can sort the list. Finally, selection sort is the 4th algorithm because its runtime is very similar regardless of the input size or configuration. This is because it passes the whole list once for each element, looking for a smaller minimum value. Since it's non-adaptive its best, worst, and average cases are all O(n^2).