Daniel Parker & Joel Beckmeyer

September 15, 2018

Experimenting with Sorting algorithms

**Insertion Sort Data**

|  |  |
| --- | --- |
| **Elements** | **Time Elapsed(sec)** |
| 100 | 0.000003319 |
| 500 | 0.000050391 |
| 1000 | 0.000202167 |
| 5000 | 0.003406371 |
| 10000 | 0.013393143 |
| 50000 | 0.346964652 |
| 100000 | 1.420933205 |
| 500000 | 35.46087305 |
| 1000000 | 142.1365249 |
| 5000000 | 3574.766212 |
| 10000000 | 14457.22497 |

**Merge Sort Data**

|  |  |
| --- | --- |
| **Elements** | **Time Elapsed(sec)** |
| 100 | 0.000032588 |
| 500 | 0.000115567 |
| 1000 | 0.000205789 |
| 5000 | 0.001093515 |
| 10000 | 0.002137242 |
| 50000 | 0.011045888 |
| 100000 | 0.022583012 |
| 500000 | 0.11542043 |
| 1000000 | 0.231529436 |
| 5000000 | 1.22100164 |
| 10000000 | 2.506953052 |

**Binary In-place Radix Sort Data**

|  |  |
| --- | --- |
| **Elements** | **Time Elapsed(sec)** |
| 100 | 0.000003621 |
| 500 | 0.000016596 |
| 1000 | 0.000039529 |
| 5000 | 0.000203978 |
| 10000 | 0.000331313 |
| 50000 | 0.001493022 |
| 100000 | 0.002368377 |
| 500000 | 0.015504134 |
| 1000000 | 0.030929814 |
| 5000000 | 0.168700613 |
| 10000000 | 0.347136041 |

**Interpretation of results:**

For insertion sort, we were able to very easily implement the algorithm that we found in the course materials. As we started to run the algorithm, it became apparent very quickly that the running time was indeed O(n2). We can see this from the data and graph, as a 2nd-degree polynomial trend line fits the data very well. We found it advantageous to make our timing more precise than just down to the second. Specifically, we made it precise down to nano-seconds. Insertion sort was the easiest to implement, but the hardest to run.

For merge sort, we had more of an interesting experience. We took a similar approach of implementation, using class materials as reference. However, once it was implemented and after fixing some memory allocation bugs, we became confused since the data seemed to exhibit linear behavior. However, after generating data that was generated to be an n lg n relationship, we discovered that plotting this data gave a similarly ambiguous graph in terms of linear vs. n lg n behavior. Also, if we look closely at our plot with a linear trendline, we can see where the trendline does not quite fit the data perfectly. With these conclusions, as well as our previous understanding of what the run time of the algorithm must be, we determined that the running time of merge sort must be n lg n.

For radix sort, we decided to implement binary in-place most-significant-digit radix sort. This makes the sorting algorithm used for each digit very simple and efficient (it is a simple if-else block). This algorithm sorts each element into either a zero or one array, similarly to in-place quick sort. It then recursively sorts each array on the next digit. The generated run time data for this algorithm exhibits an obvious linear relationship which can be confirmed by the fact that we know this sort must be linear. Interestingly, this implementation of radix sort is faster than our implementation of merge sort for all values of n that we tested.