CSSE230: Sorting Races

# Name(s): Kaelyn Bock and Olivia Penry

# Part 1: Data

Table 1 shows the runtimes of 6 sorts for at least 4 different types of arrays:

Runtime in milliseconds

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Java’s Merge Sort | Java Quick Sort | Binary Heap Sort | Tree Sort | Quick Sort |
| Random Array | 23 | 5 | 15 | 21 | 4 |
| Unique Shuffled Array | 6 | 1 | 15 | 15 | 2 |
| Almost Sorted Array | 0 | 1 | 4 | 1735 | 39 |
| Almost Reverse Sorted | 1 | 1 | 11 | 1866 | 66 |

# Part 2: Discussion

Include your discussion of the runtimes in Table 1, as described in the specification.

We think that we might have messed up our Tree Sort, seeing as it jumps from 15 & 21 milliseconds to over 1700 milliseconds. However, we think this could also be due to the fact that when you add things to a binary search tree that are almost either in reverse order or sorted order, it creates basically like a linked list, which could also explain the crazy runtimes. When making the array around 100000, we were having memory issues in tree sort, so we kept the array size less than that. Other than that, all of them ran fine at this smaller size. Java’s Merge Sort unsurprisingly worked super efficiently for the almost sorted and almost reverse arrays. Java’s Quick Sort was surprisingly insanely efficient, which we thought it’s implementation of two pivots was very cool. Binary Heap Sort was pretty unsurprising, and kept pretty steady runtimes for everything except the almost sorted array, which was expected. Our implementation of Quicksort was not super surprising, considering the unique shuffled array should be the fastest, because the pivot would be likely to not be in a worst case place, like it would be in the almost sorted or almost reverse sorted.