CSSE230: Sorting Races

# Name(s): Joshua Eckels and Simon Snider

# Part 1: Data

Table 1 shows the runtimes of all 5 sorts for at least 4 different types of arrays:

Input size: 1,000,000

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Sorting Algorithm** | | | | |
| **Array Type** | **Default (Integer) (ms)** | **Default (int) (ms)** | **Treesort (ms)** | **Heapsort (ms)** | **Quicksort (ms)** |
| **Random** | 596 | 209 | 1118 | 889 | 122 |
| **Shuffled** | 362 | 70 | 1046 | 767 | 87 |
| **Almost sorted** | 40 | 23 | 443 | 418 | 53 |
| **Almost sorted reversed** | 70 | 18 | 469 | 412 | 50 |

# Part 2: Discussion

The treesort algorithm had the longest runtime across all arrays, while the default (int) sorting algorithm was the fastest on most of the arrays. We would expect the default sorts to be faster than our sorts, because they use hybrid sorting algorithms that are optimized for speed. The quicksort was faster than treesort because, although both are O(NlogN) time best case, the treesort requires an extra O(N) step to copy everything out of a tree structure into an array, while the quicksort handles sorting in-place. The heapsort algorithm runs slower than quicksort as expected, because it must run buildHeap on the array before beginning the sort, adding a O(logN) step. There is also an extra step of selection sort at the beginning that adds an O(N) step. Quicksort runs faster than these algorithms because the only extra necessary step is picking a pivot value, which is O(1); everything else is sorted in place in O(NlogN) best case time.

In general, sorting a random array was the slowest for all of the sorting algorithms, as expected. Because many of the algorithms can skip over several array elements when they are already in order, their runtimes are faster for almost-sorted arrays. We also know for a fact there are no duplicates in the almost sorted arrays and the shuffled array, simply by the way we wrote the test cases; these arrays will be sorted faster.

It was interesting to note that the default sort algorithm for Integer objects took longer than the default sort for primitive ints. We believe this is due to the fact that Objects are sorted fundamentally different than primitives, and require more data and different methods to interact with.

\*Note: The default sort was faster than our quicksort algorithm consistently for all tests except the random array. We would expect the default Java sort to be faster and more optimized than our (rough) code pretty consistently for all general-use arrays, although this was not the case for the random array. We think this is most likely due to some built-in assumptions made in our code that allowed it to skip over special-use cases. There was inconsistent test data collected on this case.