Nikita Borisov
CS3345.002
Professor Zhao
13 April 2023

Project 2

Mergesort and Quicksort with a median-of-3 partitioning and cutoff of 15 implementations

Runtime Comparisons:

Randomly ordered array comparisons:
Randomly ordered array of 10,000 elements ranging from 0 - 9999:
Mergesort: Average of around 2,153,880 nanoseconds from 10 runs
Quicksort: Average of around 894,370 nanoseconds from 10 runs

Randomly ordered array of 1,000 elements ranging from 0 - 999:
Mergesort: Average of around 239,790 nanoseconds from 10 runs
Quicksort: Average of around 141,260 nanoseconds from 10 runs

Randomly ordered array of 100 elements ranging from 0-99:
Mergesort: Average of around 61,110 nanoseconds from 10 runs
Quicksort: Average of around 9,800 nanoseconds from 10 runs

Randomly ordered array of 10 elements ranging from 0-9:
Mergesort: Average of around 6,090 nanoseconds from 10 runs
Quicksort: Average of around 890 nanoseconds from 10 runs

Almost sorted array comparisons:
Almost sorted array of 10,000 elements ranging from 0 - 9999:
Mergesort: Average of around 1,349,020 nanoseconds from 10 runs
Quicksort: Average of around 350,620 nanoseconds from 10 runs

Almost sorted array of 1,000 elements ranging from 0 - 999:
Mergesort: Average of around 163,240 nanoseconds from 10 runs
Quicksort: Average of around 115,460 nanoseconds from 10 runs

Almost sorted array of 100 elements ranging from 0-99:
Mergesort: Average of around 80,030 nanoseconds from 10 runs
Quicksort: Average of around 11,670 nanoseconds from 10 runs

Almost sorted array of 10 elements ranging from 0-9:
Mergesort: Average of around 11,080 nanoseconds from 10 runs
Quicksort: Average of around 1,910 nanoseconds from 10 runs

It's important to note that my implementation of the 'nearly sorted array' just involves varying each index by a random value ranging from 0-3, so results could significantly vary if by sheer luck the array comes out sorted.

Throughout all of these tests, Quicksort has been the clear 'victor' against Mergesort. This is actually surprising because my initial thoughts had been that merge sort would perform better with larger data sets, but it was the opposite that was true. It might just be that I implemented one of the algorithms wrong or it's Java being Java. It's hard to tell. Regardless, here are the results summarized from above:

Randomly ordered array comparisons: For large randomly ordered arrays (10,000 elements), Quicksort performed better than Mergesort, with an average runtime of around 894,370 nanoseconds compared to Mergesort's average runtime of around 2,153,880 nanoseconds. For smaller randomly ordered arrays (1,000 elements), Quicksort still performed better than Mergesort, with an average runtime of around 141,260 nanoseconds compared to Mergesort's average runtime of around 239,790 nanoseconds.

Almost sorted array comparisons: For almost sorted arrays, Quicksort performed better than Mergesort across all array sizes tested. For large almost sorted arrays (10,000 elements), Quicksort had an average runtime of around 350,620 nanoseconds, while Mergesort had an average runtime of around 1,349,020 nanoseconds. For smaller almost sorted arrays (1,000 elements), Quicksort had an average runtime of around 115,460 nanoseconds, while Mergesort had an average runtime of around 163,240 nanoseconds.

In summary, Quicksort with a median-of-3 partitioning and a cutoff value of 15 appears to be the faster sorting algorithm for both randomly ordered and almost sorted arrays across all array sizes tested.