Group 9 Sorting Competition Presentation

Ben G. and Johannes M.

Timings

Small Dataset Runtimes:

- 427 ms
- 458 ms
- 452 ms
- Median: 452 ms

Large Dataset Runtimes:

- 538 ms
- 678 ms
- 855 ms
- Median: 678 ms

Algorithmic Improvements

- Still using Java's built in arrays.sort functionality, with custom comparator
- Built-in Java functions for many simpler things
  - The Integer class has a bit-counting function (constant time)
- Algorithm for finding substrings in $O(n^2)$ time.
  - Based on Rajput-JI's implementation (published in geeksforgeeks.org)
  - Uses a two-dimensional array to store the lengths of each non-overlapping shared substring (i.e. the value at [x][y] is the length of the longest common substring starting both at position x and position y)
  - We used the Integer numberOfLeadingZeros method to find the bit position at which the 0 padding ends. We checked the status of each bit beyond that using a bitwise mask.

Comparators and Memoization

- Comparator has a “fallback” functionality, to only perform substring length comparison if the number of binary 1s is different.
- Since substring length finding is awful, we memoize it; all lengths found are stored in a static* map, which we check before computing.
    - Each element is checked at most once.
    - *map is reset after warmup runs

Theoretical time

- Based off of Arrays.sort, so $O(n^2)$ in the worst case and $O(n \log n)$ in the average case.
- Constant time is greatly reduced, by significantly reducing the time per element. Also minimizes the impact of bit length m, since instead of being called for each of the potentially n^2 comparisons, it's called at most n times (i.e. $n^2 + nm$ instead of n^2m).

Potential/unimplemented improvements

- Original plan was “bucketing”: divide the information into “buckets” based on numbers of 1s in binary representation, sort each individually, and then recombine.
  - Divides into buckets of 10 or less, 11, 12, 13, 14, or 15 or more, and then sorts each individually.
  - Separate comparators were because only the “outlier” buckets would need to call the comparator for checking bits.
- Wouldn’t have reduced worst case time (everything COULD end up in the same outlier bucket, though it’s vanishingly improbable), BUT reduces max total comparisons and prevents bitcount from being computed as many times.
  - Repeated bitcounting isn’t the problem, but prevents cross-bucket comparison
  - Takes partial advantage of linear nature of bucket sort.

Questions?