WILLIAM DUNN

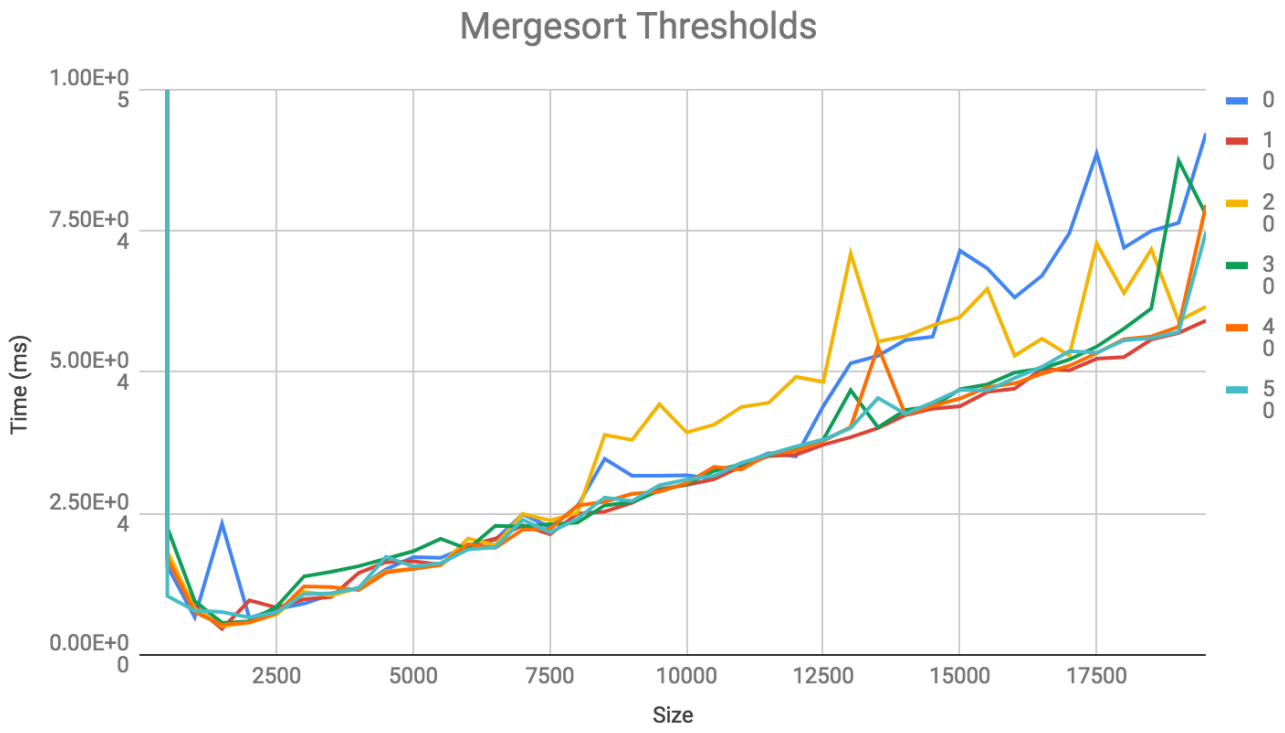
Assignment04 - Quicksort and Mergesort

SUMMARY

1. Who are your team members?

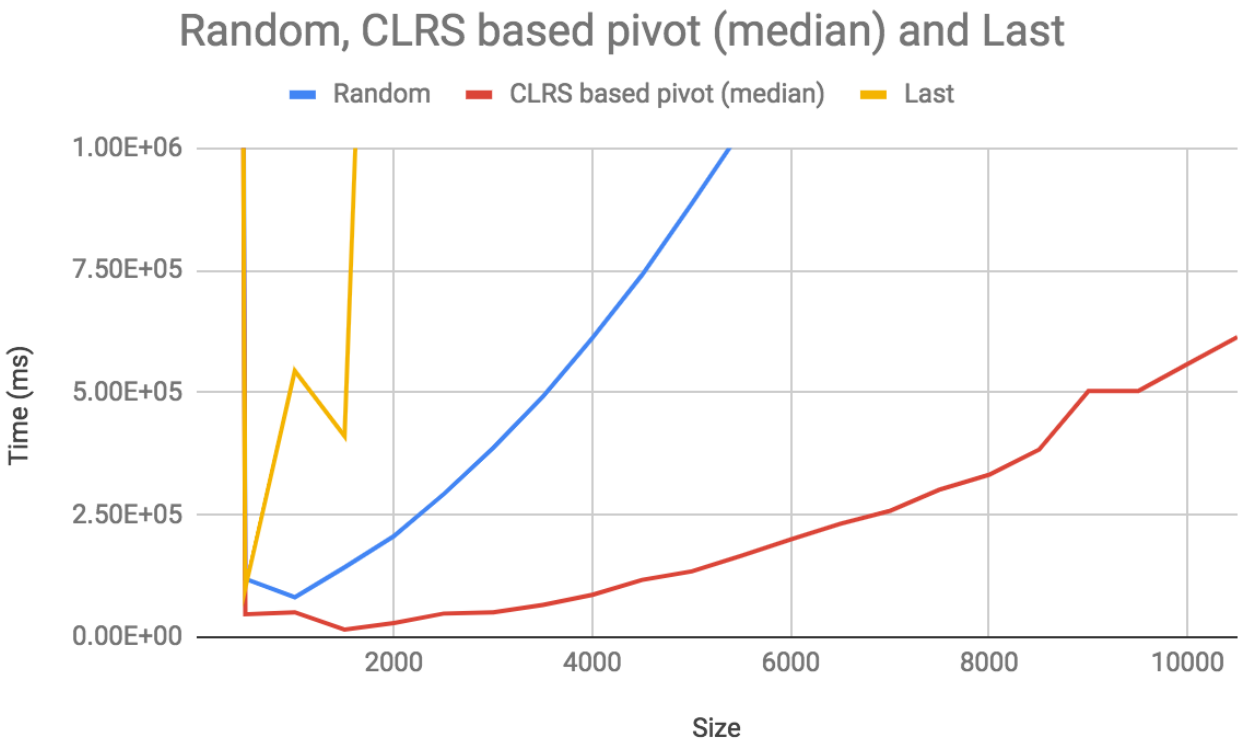
Qi Liu

2. Mergesort Threshold Experiment: Determine the best threshold value for which mergesort switches over to insertion sort. Your list sizes should cover a range of input sizes to make meaningful plots, and should be large enough to capture accurate running times. To ensure a fair comparison, use the same set of permuted-order lists for each threshold value. Keep in mind that you can't resort the same ArrayList over and over, as the second time the order will have changed. Create an initial input and copy it to a temporary ArrayList for each test (but make sure you subtract the copy time from your timing results!). Use the timing techniques we already demonstrated, and be sure to choose a large enough value of timesToLoop to get a reasonable average of running times. Note that the best threshold value may be a constant value or a fraction of the list size. Plot the running times of your threshold mergesort for five different threshold values on permuted-order lists (one line for each threshold value). In the five different threshold values, be sure to include the threshold value that simulates a full mergesort, i.e., never switching to insertion sort (and identify that line as such in your plot).



Having the merge sort switch to insertion sort at a threshold of 10 performed the fastest. The constant value did better than percentages when 30 and various percentages were used. Due to this, I tested multiple constant values and no switch. It should be noted that the lack of a switch actually performed the worst in many conditions. Because 10 appeared to performed best in most conditions, and without any spikes after a small N, it will likely be used with the implementation in the future, if I need to use it. Regardless of the tested thresholds, the sorts followed the complexity of O(nlogn).

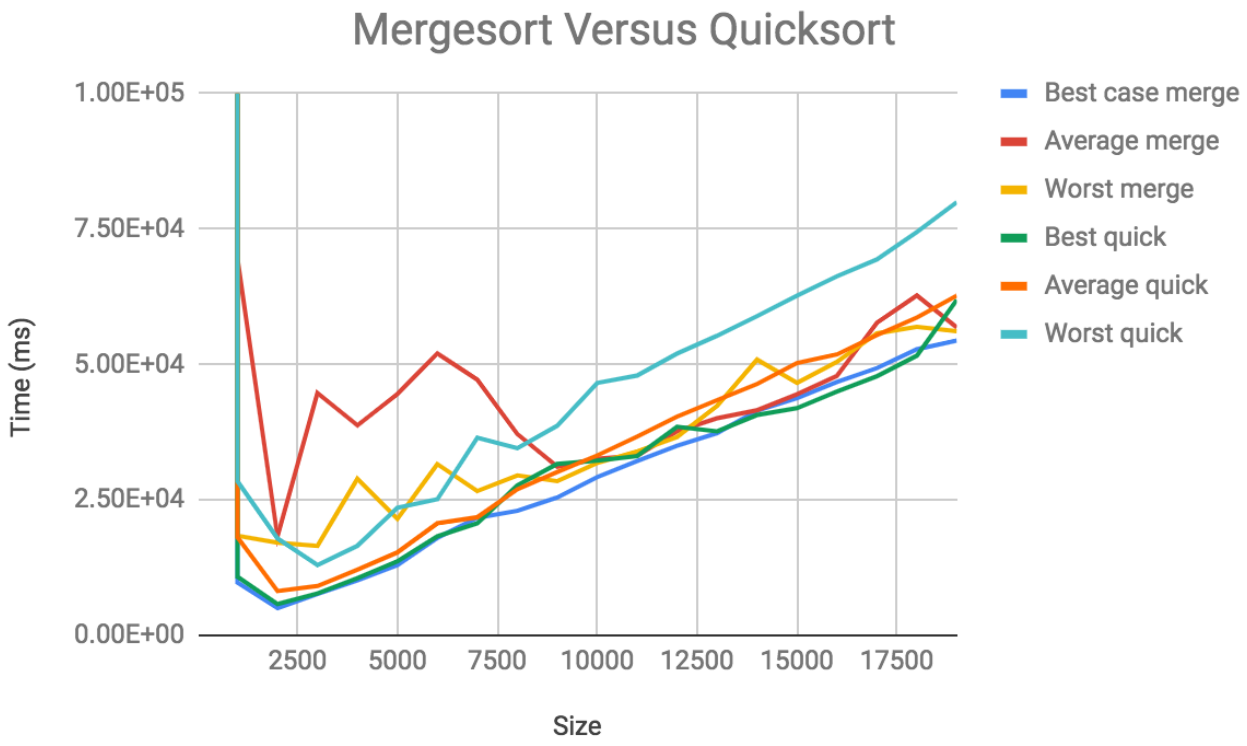
3. Quicksort Pivot Experiment: Determine the best pivot-choosing strategy for quicksort. (As in #2, use large list sizes, the same set of permuted-order lists for each strategy, and the timing techniques demonstrated before.) Plot the running times of your quicksort for three different pivot-choosing strategies on permuted-order lists (one line for each strategy).



The best pivot strategy ended up being selecting the median. This is listed as the CLRS method, after the authors of the Introductions to Algorithms 3rd edition book, because it mirrors much of what they recommended in the pseudocode of their textbook. It followed a O(nlogn) behavior for average case conditions, and ended up being easier to implement than the other two pivot selections. The next best was the random selection. While this performed at a $O(n^2)$ complexity in average conditions, this might have been due to it selecting a poor pivot during many occasions or even that the algorithm had been built for the median option, and it had to be altered (and not optimized) for a random pivoting strategy. The worst performer, compared here, was the using the last element. It's terrible performance mirror cubic complexity. While choosing the last index at the pivot is a poor choice of pivot, the lack of optimization too could have led to this performance.

4. Mergesort vs. Quicksort Experiment: Determine the best sorting algorithm for each of the three categories of lists (best-, average-, and worst-case). For the mergesort, use the threshold value that you determined to be the best. For the quicksort, use the pivot-choosing strategy that you determined to be the best. Note that the best pivot strategy on permuted lists may lead to O(N^2) performance on best/worst case lists. If this is the case, use a different pivot for this part. As in #2, use large list sizes, the same list sizes for

each category and sort, and the timing techniques demonstrated before. Plot the running times of your sorts for the three categories of lists. You may plot all six lines at once or create three plots (one for each category of lists).



It should be noted prior to commenting on each case that if space is a concern, quick sort would be chosen in each condition because it does not require a temp array like merge sort does.

Best: This is difficult to say whether or not the quick- or merge sort algorithms performed better than one another as they seemed to switch depending on the sample size. It should be noted that the merge sort handled a larger sample size, while the quick sort reached stack overflow. For this reason, I'd choose the merge sort for its scalability. Both exhibited O(nlogn) complexity.

Average: the average case merge sort didn't racked a competitive performance until ~8000 samples had been reached. It did perform slightly better until ~15500 samples before being outperformed by the quick sort again, but this isn't enough to change my mind - I thought quick sort performed better. Both exhibited O(nlogn) complexity, but with varying coefficients.

Worst: the worst case quick sort only performed better than merge until ~7000 samples (with one small dip prior). Because of this and the lack of scalability of the quick sort, in the worst case I'd stick with the merge sort. Both of which performed at O(nlogn), but it should be noted that that the best pivot was chosen here because the worst

pivot could not get an ample enough sample size before crashing and the random pivot exhibited quadratic complexity, which would also lead to merge sort being chosen over it.

5. Do the actual running times of your sorting methods exhibit the growth rates you expected to see? Why or why not? Please be thorough in this explanation.

All but the worst case, worst pivot for quick sort performed how expected. That one met cubic complexity, where I expected it to do quadratic. While it wouldn't be chosen for any real-world applications with either of those complexities and difficulty in implementing, it will exceeded my expectations in how bad it would perform. For all the others, they worked on divide and conquer principles which led to logarithmic complexities. Because this was required with an n complexity a top the nested recursion in both cases due to final sorting that took place (like the merging in merge sort), a total of (nlogn) resulted from all other cases. What did surprise me was the stack overflow that occurred with the quick sort implementations which did not occur with the merge sort, even though the merge sort required more space.