1. Who are your team members?

Aiden Pratt
Melanie Prettyman

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).

(GRAPH A, TABLE A)

Based on my experiments, the data helped determine that the best threshold value for merge sort to switch over to insertion sort is ~500. I ran the experiment on 5 differently sized arrays: 1000, 5000, 10000, 50000, 100000. and tested 7 different thresholds: 0 (which should always resort to merge sort), 10, 500, 1000, 5000, 50000, 100000(which will always insertion sort). In previous tests that I did not graph, I tested a range of other thresholds not displayed here. The 500 threshold was clearly faster in these cases and became a fairly constant time to sort the arrays as they grew in size. The other thresholds I tested were mostly linear. Interestingly, every threshold changed trajectory at my array of size 5000. This is where the 500 threshold began to go faster until the next size array at 10000 compared to the other thresholds.

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 strategies on permuted-order lists (one line for each strategy).

(GRAPH B, TABLE B)

For this experiment, I tested three different pivot points being the first index element in the array, the middle index element in the array, and the last index element in the array. I did this with an arraylist of generating the average case of arrays, being randomly organized. This data suggested that the first and end indexes being the pivot point act grow relatively linearly when growing the size of the array. The middle index being the pivot point took longer initially at lower size arrays but tapered off and began sorting the arrays quicker by the end of my experiment.

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).

(GRAPH C, TABLE C)

My experiments determined quick sort to be the best algorithm over merge sort in all three categories being best, average, and worst. With quick best and quick worst to be the fastest overall. This experiment found merge sort on the average case to be the slowest.

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.

The outcome of these experiments are roughly what I expected. Overall, I found that quick sort was faster than merge sort in my experiments. The experiment's growth rates are what I expected as every sorting method that I tested grew linearly when sorting arrays of larger sizes. However, the rate at which these grew was different which helped determine which sorting method works better in different conditions. I did predict that the quick sort would be faster overall than the merge sort because of the partition and merge helper methods themselves. It takes more time and data for the merge sort because it must create basic arrays and copy the data into them. while in quick sort, we do not need to allocate new data structures to complete the efficient sort. Quick sort may also have an advantage due to the threshold we set that could cause it to default to insertion sort when the array gets small enough.

Introducing the different scenarios for best, worst, and average case did not necessarily align with the growth rates and comparisons I would have expected. For example, the quick sort worst case sort performed fairly similarly to the quick sort best case sort. Also, the average case and worst case for the merge sort performed quite similarly.

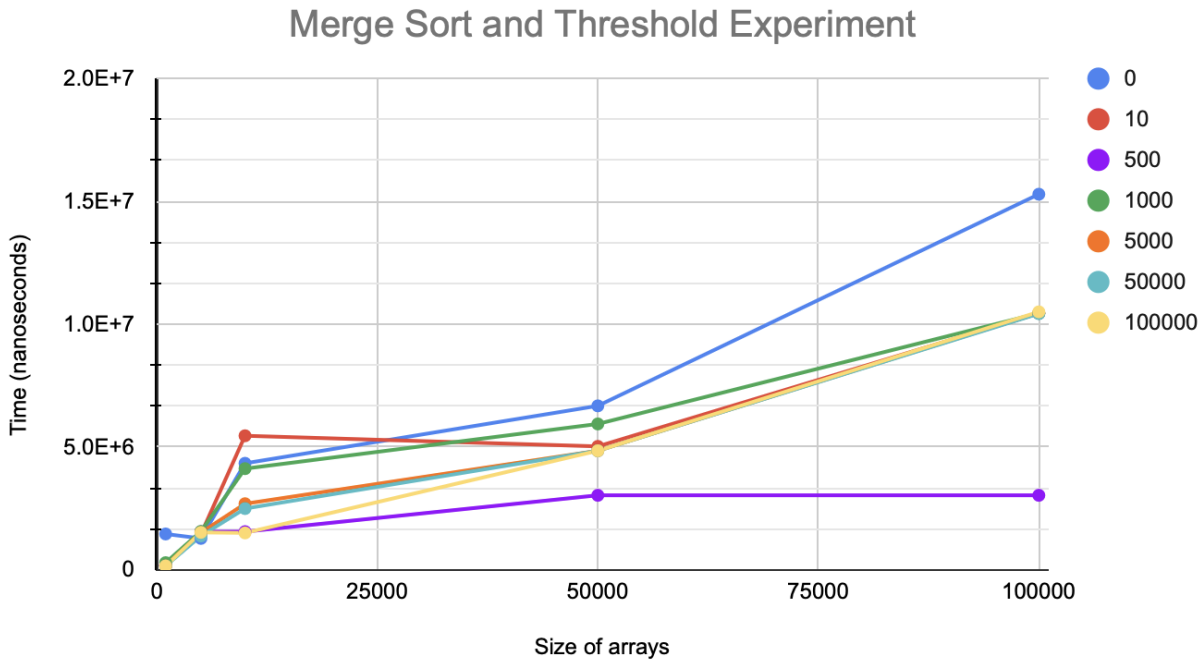
The Big O analysis can align with this data. In an average and best case scenario, both sort methods are expected to be O(N). In the worst case scenario these algorithms could be O(N^2) due to the potential that they will need to sort a completely backwards list.

Things that could have impacted this experiment are the way timing is conducted. Timing in nanoseconds is a very fast test and could lead to a large amount of data variability. If one part of a test, such as copying an array, takes more time , it could greatly change the results. Repeating these tests thousands of times with many more array lengths would provide more data and allow me to draw more patterns from the data. It would also be interesting to test these algorithms with different data types such as Strings or Students.

FIGURES

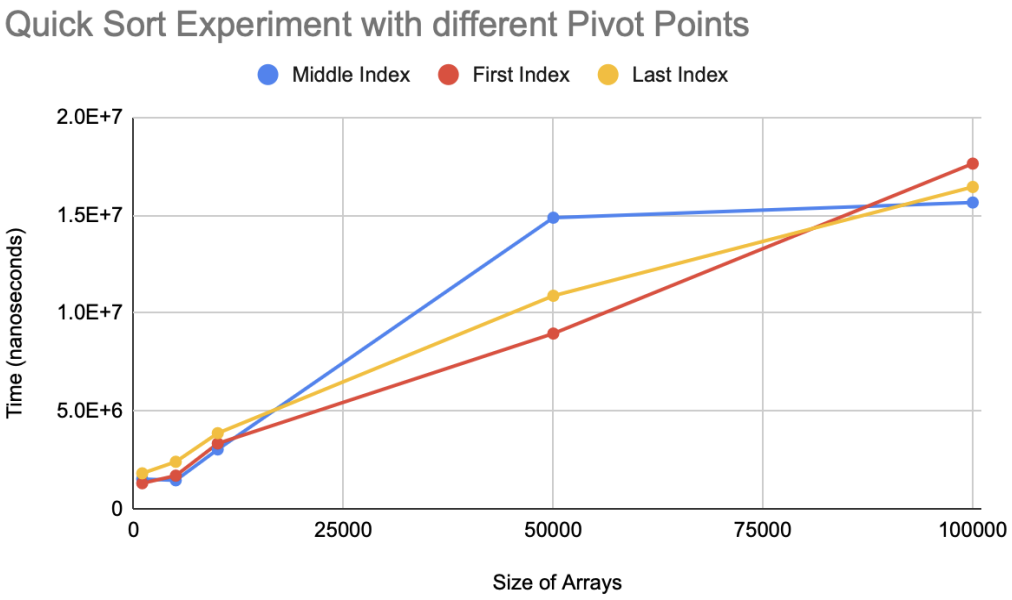
A) MERGE SORT AND THRESHOLD EXPERIMENT

size/threshold	0	10	500	1000	5000	50000	100000
1000	1461458	201375	170833	293000	168875	162291	157041
5000	1281250	1450292	1555709	1565000	1513791	1382500	1525166
10000	4338541	5459750	1555709	4121333	2692708	2496625	1505041
50000	6677667	5027625	3035959	5940875	4850125	4850125	4849750
100000	15293375	10444500	3035959	10446292	10453375	10432417	10503875



B) QUICK SORT EXPERIMINT WITH DIFFERENT PIVOT POINTS

size	middle	start	end
1000	1520333	1304334	1809875
5000	1453542	1692125	2400917
10000	3031250	3337041	3861083
50000	14871333	8951375	10882083
100000	15648083	17634292	16442625



C) MERGE SORT WITH THRESHOLD 500 VS QUICK SORT WITH MIDDLE PIVOT POINT

size	merge average	quick average	merge best	quick best	merge worst	quick worst
1000	1370333	1738500	1901917	1680334	1203666	1076375
5000	1413584	4576667	1474958	818166	962917	972000
10000	2434542	3008875	2465458	1386917	2254708	1959708
50000	16458334	7245750	12641125	7437416	10928750	8825375
100000	30037875	17398000	12360750	2420208	27346625	3827459