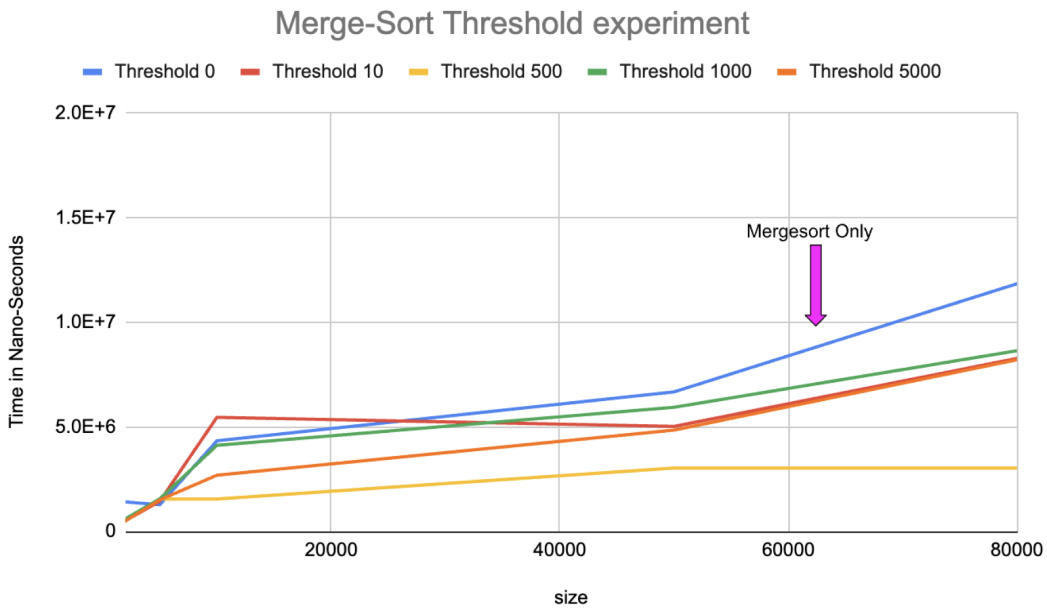
Analysis doc

1. **Who are your team members?**
   Aiden Pratt
2. **Mergesort Threshold Experiment**

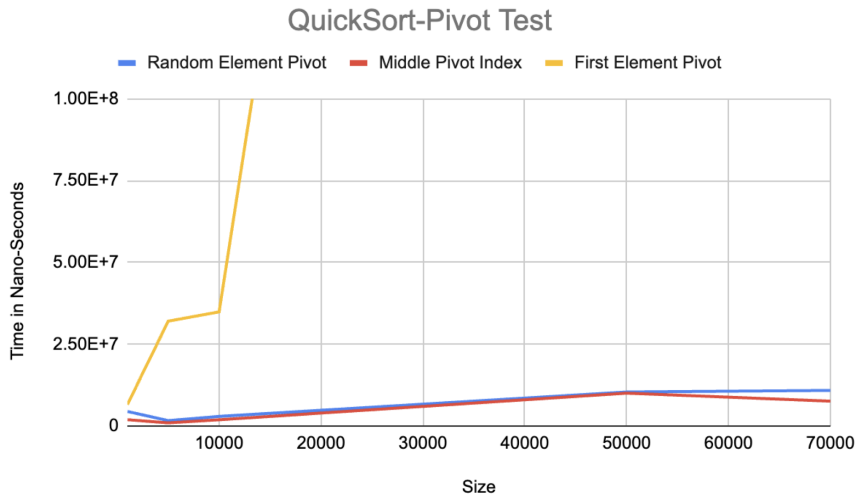
Determine the best threshold value for which mergesort switches over to insertion sort.



The threshold value of 500 appears to perform well in the provided dataset. This is because it strikes a balance between using merge sort for larger subarrays and insertion sort for smaller subarrays. This threshold value allows merge sort to handle most of the sorting process while avoiding the overhead of recursive calls for very small subarrays.

2. **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.)

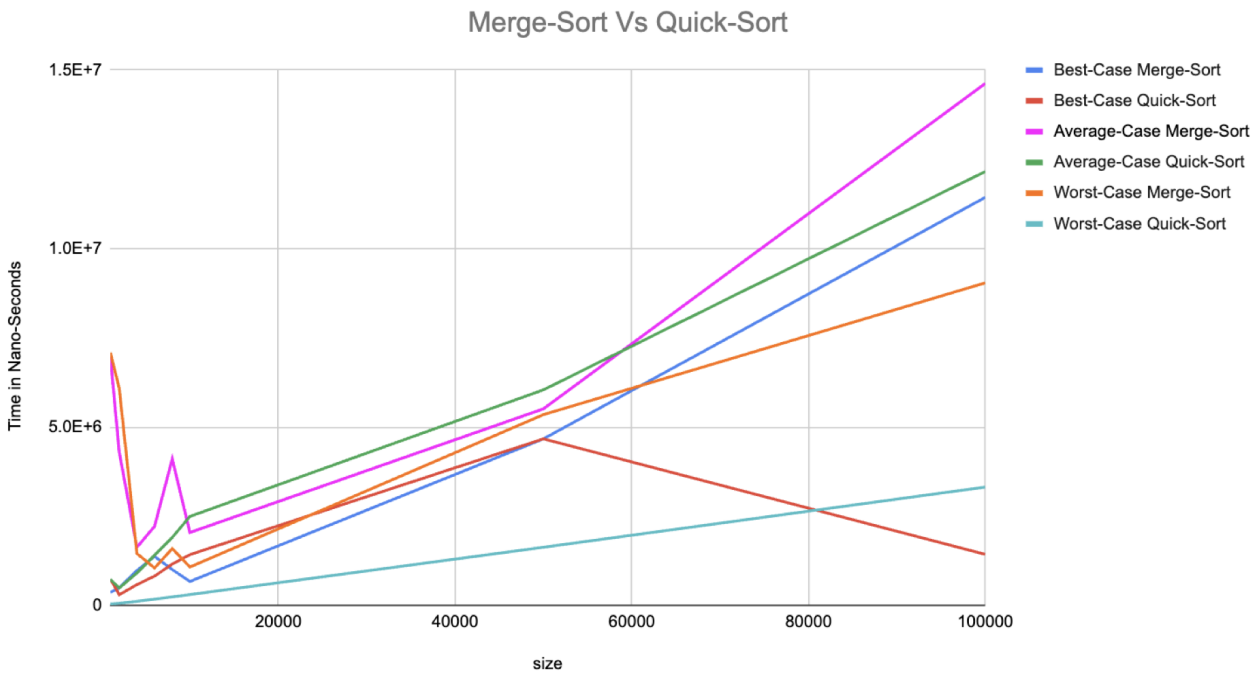


The best pivot strategy is choosing the middle index. The reason the middle index pivot is marginally better than the random index pivot is because it’s more likely to provide a pivot value closer to the median. By using the middle index, it's more likely to have a pivot that results in more balanced partitions, reducing the chances of heavily imbalanced splits.

Although the middle and random index methods might still suffer in specific cases, such as already sorted or nearly sorted arrays, they tend to handle these cases better than the first element method.

The first element method results in poor performance due to predictable partitioning. The random and middle index methods offer better average performance, with the middle index slightly edging out due to its tendency to provide more balanced partitions.

3. Mergesort vs. Quicksort Experiment



For best-case scenario, the quick sorting algorithm consistently outperforms the merge sorting algorithm. The running times of the quick algorithm are significantly lower than those of the MERGE algorithm for all list sizes.

In the average-case scenario, the merge sorting algorithm performs better than the quick sorting algorithm for smaller list sizes. However, as the list size increases, the quick sorting algorithm surpasses the merge sorting algorithm in terms of running time. This can be attributed to the efficient partitioning technique used in the quick sorting algorithm.

In the worst-case scenario, the quick sorting algorithm significantly outperforms the merge sorting algorithm. The running times of the quick algorithm are much lower than those of the merge algorithm for all list sizes.

The runtime analyses of these sorting methods do align with the projected runtime. In the best-case scenario, the quick sort algorithm out performed merge sort algorithm. In the best-case scenario, quick sort exhibits an O(n log n) performance, but merge sort also operates at O(n log n), yet merge sorting has a slightly slower runtime due to it having a slightly higher constant factor due to its additional overhead in merging steps.

In the average-case scenario, the results do follow their anticipated growth rates. Both algorithms maintain an O(n log n) average-case runtime; however, quick sort's partitioning

technique allows it to adapt better to larger datasets, optimizing its performance as the input size increases.

In the worst-case scenario, quick sort consistently outperforms merge sort, in line with expectations. While both algorithms maintain an O(n log n) worst-case complexity, the nature of quicks sort's partitioning method, coupled with its pivot selection strategy, mitigates potential inefficiencies encountered by MERGE sort in certain data distributions.