Mergesort Threshold Experiment:

Threshold 5: Good for small input sizes but becomes inefficient for large lists due to frequent switching to Insertion Sort.

Threshold 10 to 30: Transition period where Insertion Sort is used for smaller sublists but not too often. This leads to a performance increase for medium-sized inputs.

Threshold 50: Best overall performance for larger inputs. MergeSort is used more extensively, and Insertion Sort is used sparingly, resulting in fewer overheads from the smaller sublist sorting.

Threshold "No Insertion Sort" (effectively, Integer.MAX_VALUE): The MergeSort only handles all sublists without switching to Insertion Sort. For large lists, this is very efficient because MergeSort is designed to handle large lists efficiently (O(n log n)), but for small lists, this can be slower than using Insertion Sort. As we can see, the times for size 100 and 500 are higher in this case because MergeSort alone is less efficient for small sublists compared to Insertion Sort.

Quicksort Pivot Experiment:

For Small List Sizes (100):

> RANDOM pivot performs the best with the lowest time, which is expected due to the simplicity and efficiency of random pivot selection.
>
> MIDDLE pivot is slower than RANDOM, which is unusual but can happen if there is a small amount of overhead or variability in the partitioning.
>
> MEDIAN_OF_QUARTILES performs well in this case and shows the smallest time, which is a bit surprising, as this pivot strategy usually adds overhead.

For Medium List Sizes (500):

> RANDOM pivot continues to outperform the other strategies.
>
> MIDDLE pivot is slower than RANDOM but still acceptable.
>
> MEDIAN_OF_QUARTILES shows an increase in time, highlighting the overhead of finding the pivot.

For Larger List Sizes (1000, 5000, 10000):

RANDOM pivot consistently performs well across all sizes, and its times increase linearly with size.

MIDDLE pivot performs reasonably well but shows a slight increase in time as list size increases.

MEDIAN_OF_QUARTILES shows very large increases in time as the list size grows, which is due to the additional complexity of selecting the pivot.

RANDOM pivot is the most efficient across all list sizes, with minimal overhead and good scaling performance, particularly in random permuted lists.

MIDDLE pivot performs well in many cases but may have slightly slower performance compared to RANDOM pivot, especially on larger lists.

MEDIAN_OF_QUARTILES shows significant overhead and doesn't provide enough performance improvement to justify its complexity for random permuted lists, particularly with larger lists.

MergeSort Vs QuickSort Experiment:

MergeSort is O(N log N) in all cases (best, average, and worst), so we expect it to show relatively consistent performance across all cases. It should scale with list size in a predictable manner.
QuickSort is O(N log N) in the best and average cases, but $O(N^2)$ in the worst case.

For Small List Sizes (100):

MergeSort performs significantly slower than QuickSort in all cases. This is expected because:

MergeSort involves more overhead due to the extra space required for merging and copying elements.

QuickSort can be faster for small lists with its divide-and-conquer partitioning approach.

QuickSort's best case is quite fast (13,370 ns), as expected, due to the efficient partitioning when the list is nearly sorted.

For Medium List Sizes (500):

MergeSort shows an increase in time as list size grows, but it still consistently performs worse than QuickSort.

QuickSort's best case is still faster than MergeSort, but the difference is less dramatic than for smaller sizes.

QuickSort's average and worst cases are still very efficient compared to MergeSort, but the worst-case performance (31,220 ns) is noticeably slower than the best case (30,525 ns).

For Larger List Sizes (1000, 5000, 10000):

MergeSort continues to show relatively consistent performance for best, average, and worst cases, since its time complexity is O(N log N). The performance increases steadily with the size of the list.

QuickSort, in contrast, shows more variation across different cases, especially for larger lists:

In the best case, QuickSort remains efficient but still slower than MergeSort for very large lists.

The average case shows a slightly slower performance for QuickSort compared to MergeSort at list sizes of 1000 and 5000.

The worst case shows significant slowdowns, particularly as the list size increases. QuickSort’s worst-case performance increases dramatically, especially for larger lists like 5000 and 10000.

QuickSort’s best case should perform the fastest (especially for small list sizes).
MergeSort will have a more consistent performance across all cases, but it will generally take longer for small list sizes because of the merging overhead.
QuickSort’s worst case performance will degrade significantly for larger list sizes, especially as the list becomes more ordered or reversed.