Experimental Analysis of Mergesort and Insertion Sort Thresholds

Arlo Steyn
Student Number: 24713848

14 March 2025

Abstract

This report presents an experimental analysis that determines the optimal threshold values at which to switch from mergesort to insertion sort for small subarrays. We evaluated the performance of both algorithms through various input types (sorted, unsorted, and reverse sorted arrays) and array sizes to identify the most efficient threshold values. Our findings show that while implementing a hybrid sorting approach, combining mergesort and insertion sort for smaller subarrays, yields some improvement. The performance gain was minimal. The hybrid sort outperformed base mergesort only for array sizes smaller than or equal to 10^3 with threshold values of 32 for both sorting algorithms. However, this improvement was not substantial enough to justify the added complexity of the hybrid approach, especially for small array sizes where the time difference is negligible. Although the hybrid sort might be more effective under different testing conditions, such as using JMH for JVM performance tuning or performing a more thorough space complexity analysis, the minimal benefits do not warrant its implementation in practice.

Contents

1 Introduction

Mergesort is known for its consistent $O(n \log n)$ performance regardless of input data characteristics [1]. However, for small arrays, the overhead of recursive calls and additional memory allocation can outweigh its advantages. Insertion sort, that has a worst case time complexity of $O(n^2)$, performs efficiently on small or nearly sorted arrays due to its low constant factors and minimal overhead[2]. This report investigates the optimal threshold at which to switch from mergesort to insertion sort for small subarrays. We examine two threshold parameters:

- **SORT_THRESHOLD**: If an arrays length is less than this value, we use insertion sort instead of mergesort initially.
- **MERGESORT_THRESHOLD**: During the recursive mergesort process, if a subarrays length is less than this value, we use insertion sort for that subarray.

The goal is to determine threshold values that consistently deliver optimal performance across various input types and sizes.
In the section that follows, we'll go through the environment created to set up and create the tests needed to run our experiments.

2 Experimental Setup

Briefly, our main idea is to run a base case of tests for the insertion and mergesort. We do this to determine the ranges of the respective thresholds values to use for our main experiment and to also see if we've actually made an improvement.
In this subsection, we briefly outline the hardware and software used to run these experiments.

2.1 Hardware and Software Environment

It is important to note that we are running our environment on a virtual machine called Virtual Box. I include this if some of the tests results seem slower than what they should be. The computer specifications used in this experiment are as follows:

- CPU: AMD Ryzen 7 3750H (2.30 GHz) using only 5 of the 8 cores available due to the virtual machine.
- RAM: 10GB on the virtual machine
- Operating System: Ubuntu 22.04 LTS
- Java Version: 17

The next subsection goes over the way in which data was generated for our experiments.

2.2 Data Generation

2.2.1 Array Types Used

We tested the sorting algorithms using three types of input arrays:

- **Sorted arrays**: Arrays already in ascending order
- **Reverse sorted arrays**: Arrays in descending order
- **Unsorted arrays**: Arrays with randomly distributed elements

Testing our sorting algorithms on this diverse set of array types will ensure that we obtain a comprehensive understanding of their performance across different input scenarios, highlighting their strengths and weaknesses in various conditions. Real world scenarios work mostly with unsorted arrays of data. This implies that the unsorted (random) arrays carry (by far) the most weight with respect to their importance and results.

2.2.2 Array Sizes Used

For each array type, six orders of magnitude are tested. Given my struggling virtual machine (which would take too long for higher orders), the selected array sizes are 10, 10^2, 10^3, 10^4, 10^5, and 10^6. All arrays consist of integers. This broad range of array sizes allows us to observe how the algorithms scale and perform under varying loads.
The following section outlines the base test cases of our experiments.

2.3 The Base Cases

The respective sort thresholds pose some what of a cold start problem. What values should we choose? What range should we test? etc. Furthermore, even if we did have these ranges, how would we know that we've actually made an improvement when sorting?
A simple approach that solves both problems would be to simply run the same test setup with only insertion sort and then also only mergesort. This will allow us to narrow down the ranges for the respective sort thresholds, reducing the exploration space that we would've used to find them in the first place (if a more brute force like approach was used). This simplifies our experiments, makes them more efficient, and also allows an easy baseline in comparison.
Following this, the next subsection focuses on how we compare the sorting algorithms and their thresholds.

2.4 Measurement Methodology

Speed and memory are ways in which we compare sorting algorithms. We will be comparing speeds of the algorithms. Memory comparisons are outside the scope of this project (even though one of the main reasons for doing the hybrid sort method is the memory overhead for small sub-arrays within recursive merge calls - we leave this as an exersize for the reader).
To ensure accurate timing measurements, we:

- Used Java's `System.nanoTime()` for high precision timing.
- Used identical copies of input arrays across different algorithm configurations to ensure a fair comparison.
- Conducted JVM warm up before each test, with 5 warm up iterations on arrays of size 222, 222 to mitigate any initial performance overhead[3]. These warm up runs are not printed.
- Performed multiple runs for each configuration (50 iterations per run). This is done to attempt to get a general average per iteration and iron out most variation in run time. However, due to the JVM, this is not such a simple science. No matter the amount of iterations, we did still find some variation, but we'd like to think that the 50 iterations still gives us a good approximation of the true results.
- A new random seed was used for every iteration, each time the data was genetrated. This was done to ensure randomness (as much as randomness can actually be ensured) and fair testing[4].
- Calculated the average execution time across the remaining iterations.
- For the respective sorting thresholds, once we have found the respective ranges for each, we will iteratively run through them and collate the results.

The next subsection highlights the implementations of the sorting algorithms and how our sort thresholds are integrated into the sorting algorithms.

2.5 Implementation Details

The core sorting algorithms were implemented as follows as seen in the *Utilities.java* file:

2.5.1 Insertion Sort

A basic insertion sort based off [5].

```java
private static <T> void insertionSort(T[] data, Comparator<? super T> cmp,
        int low, int high) {
    for (int i = low + 1; i <= high; i++) {
        T currElem = data[i];
        int j = i - 1;

        while (j >= low && cmp.compare(data[j], currElem) > 0) {
            data[j + 1] = data[j];
            j--;
        }
        data[j + 1] = currElem;
    }
}
```

Listing 1: Insertion Sort Implementation

2.5.2 Hybrid Merge Approach

This sort method is used to integrate the MERGESORT_THRESHOLD. With reference to [6]

```java
private static <T> void mergeSort(T[] data, T[] aux, Comparator<? super T> cmp, int
    low, int high) {
    if (low < high) {
        // Insertion Sort Threshold
        if (high - low + 1 < MERGESORT_THRESHOLD) {
            insertionSort(data, cmp, low, high);
        } else {
            int mid = (low + high) / 2;
            // Sort 1st and 2nd halves
            mergeSort(data, aux, cmp, low, mid);
            mergeSort(data, aux, cmp, mid + 1, high);
            merge(data, aux, cmp, low, mid, high);
        }
    }
}

@SuppressWarnings("unchecked")
private static <T> void merge(T[] data, T[] aux, Comparator<? super T> cmp, int low,
    int mid, int high) {
    // Copy data to aux array
    for (int k = low; k <= high; k++) {
        aux[k] = data[k];
    }

    int a = low;
    int b = mid + 1;

    for (int i = low; i <= high; i++) {
        if (a > mid) {
            data[i] = aux[b++];
        } else if (b > high) {
            data[i] = aux[a++];
        } else if (cmp.compare(aux[a], aux[b]) <= 0) {
            data[i] = aux[a++];
        } else {
            data[i] = aux[b++];
        }
    }
}
```

Listing 2: Hybrid Mergesort Implementation

2.5.3 General Sort Method Used

This sort method is used to integrate the SORT_THRESHOLD and is used by all testing scripts to determine which sorts to use. Note that for this test setup, we temporarily set the private static final SORT_THRESHOLD & MERGESORT_THRESHOLD to public static, taking away the privitization and final keyword to allow the altering of the respective thresholds.

```java
public static <T> void sort(T[] data, Comparator<? super T> cmp) {
    // Sorts the data array, using the cmp comparator.
    if (cmp == null || data == null) {
        throw new NullPointerException();
    }

    // No sorting needed
    if (data.length <= 1) {
        return;
    }

    if (data.length < SORT_THRESHOLD) {
        insertionSort(data, cmp, 0, data.length - 1);

    } else {
        @SuppressWarnings("unchecked")
        T[] aux = (T[]) new Object[data.length];
        mergeSort(data, aux, cmp, 0, data.length - 1);
    }

}
```

Listing 3: General Sort Implementation

The subsection that follows dives into the procedure for data collection.

2.6 Data Collection

As you could of probably guessed based on the way we are measuring and running experiments, theres going to be quite a substantial amount of data. Broadly:

- **Base mergesort vs base insertion sort each with:**
  1. 3 different array types,
  2. 6 orders of magnitude,
  3. An average of 50 runs.
- **Hybrid sort with:**
  1. 6 different sort thresholds (spoiler),
  2. 5 different meregsort thresholds (spoiler),
  3. 3 different array types,
  4. 6 orders of magnitude,
  5. An average of 50 runs.

For the base merge and insertion sort we will be collecting the following data: *Size* (array size), *Type* (array type) & *Time(s)* (Time it takes to sort the array in seconds).
For the hybrid sort we will be collecting the following data: *Size* (array size), *SortThreshold*, *Mergesort Threshold* , *Type* (array type) & *Time(s)* (Time it takes to sort the array in seconds).
Roughly, this works out to about 28 000 induvidual sorts and about 600 rows of data. So what data are we actually going to use and why?:

2.6.1 The base cases:

Here it is quite simple, we will compare the base merge and insertion with each other and themselves. Again this is more of a proof of concept to showcase the the base results for fair comparison and to paint a clear picture of the differences of time complexity amongst the two sorting methods.

2.6.2 The hybrid sorts:

Here, we are not as lucky. We've added two dimensions of comparison. so we are going to cheat slightly by mainly comparing our most useful and likely scenario, the unsorted arrays. We are going to determine the optimal values of our respective sort thresholds for each array size as well as a best choice pair values for this and compare these with the base case as well as briefly with the other respective thresholds. Furthermore, we'll average the time taken and produce results to show the respective sort thresholds pairs. Lastly, we will do a few brief comparisons with the sorted and reverse sorted arrays. The next section focuses on the results obtained from the base case tests and also includes the ranges for the respective sort thresholds.

3 Base Case Results and Analysis

Jumping straight in, figure 1) compares the results of the base merge and insertion sorts for each respective type of array. Here we compare logarithmic time in seconds versus the logarithmic array sizes for a simpler view.

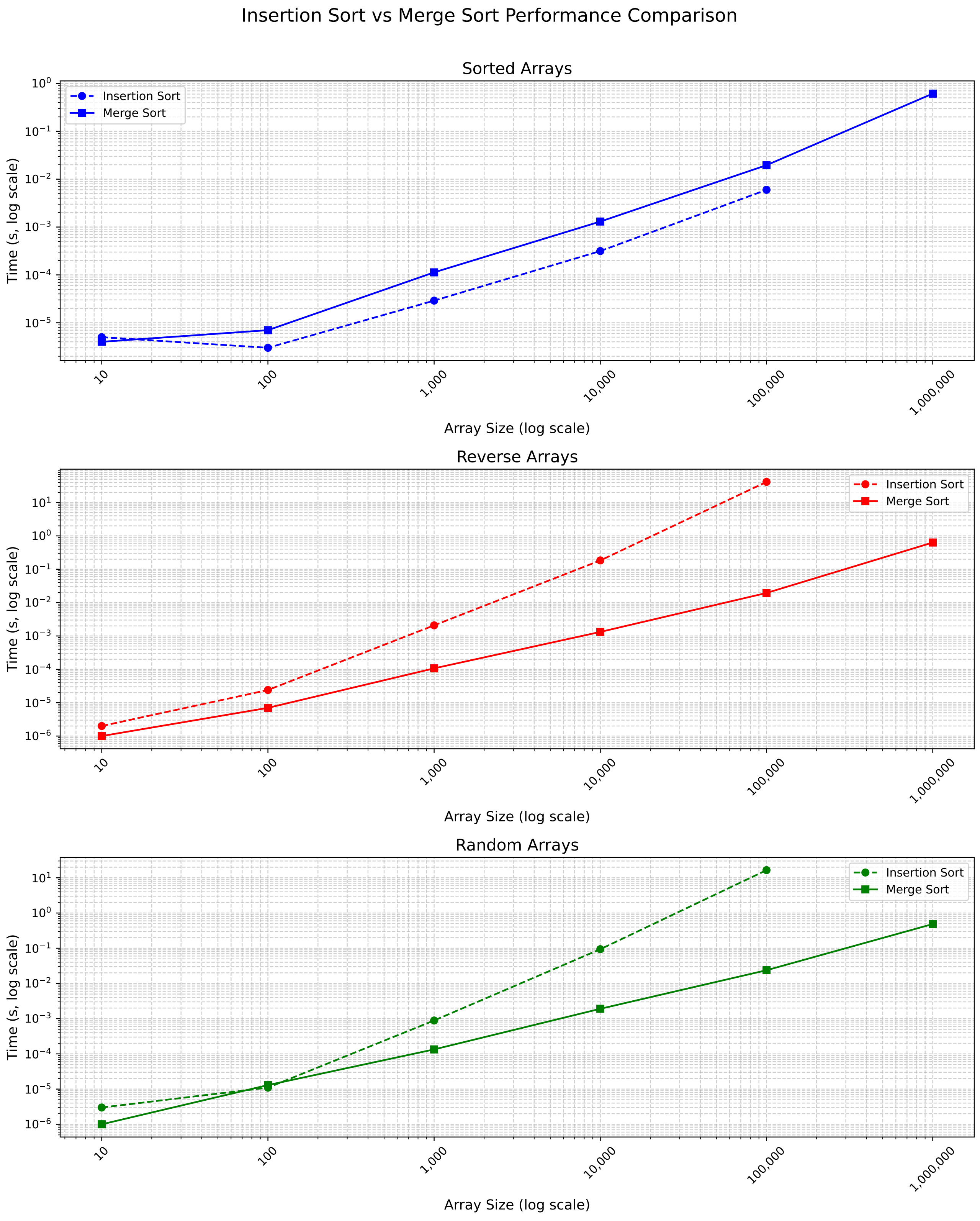

Figure 1: Comparison of base merge and insertion sorts.

Insertion sort was not able to run an array size of 10^6, even after several hours. But we get the general idea. Insertion sorts $O(n^2)$ cannot keep up with mergesorts $O(n \log n)$ time. However, as expected, for already sorted arrays, insertion sort is quicker than mergesort, but they both run on the same scale/time complexity. Furthermore, for smaller array sizes such as sizes 10^1 & 10^2, insertion sort seems to keep up and even in most cases, outperforms mergesort. This is good news as it informs us that our hybrid sort may just have a case of succeeding.

Now, based on these results, we can determine optimal thresholds for selecting between insertion sort

and mergesort. Currently, the range for choosing insertion sort over mergesort lies between 0 and 100. By adopting a step size of powers of 2, we create a range consisting of steps of 2 within the interval of 0 to 100.
Using step sizes that are powers of 2 is important because these values align well with the underlying binary nature of sorting algorithms and data splits. Binary divisions naturally suit algorithms like mergesort because it repeatedly divides data in half. Using powers of 2 for the step size ensures consistency and efficiency in both sorting and splitting which leads to more predictable performances and reduces potential biases caused by unoptimal range incrementations [7].
Hence why our thresholds are as follows:

- SORT_THRESHOLD = {0, 4, 8, 16, 32, 64}
- MERGE_THRESHOLD = {0, 4, 8, 16, 32}

Lastly (for this section), we plot the times for the base mergesort algorithm for all respective array types and array sizes (in logarithmic form) in figure 2). We use this as a base to compare our results with the hybrid approach in the next section.

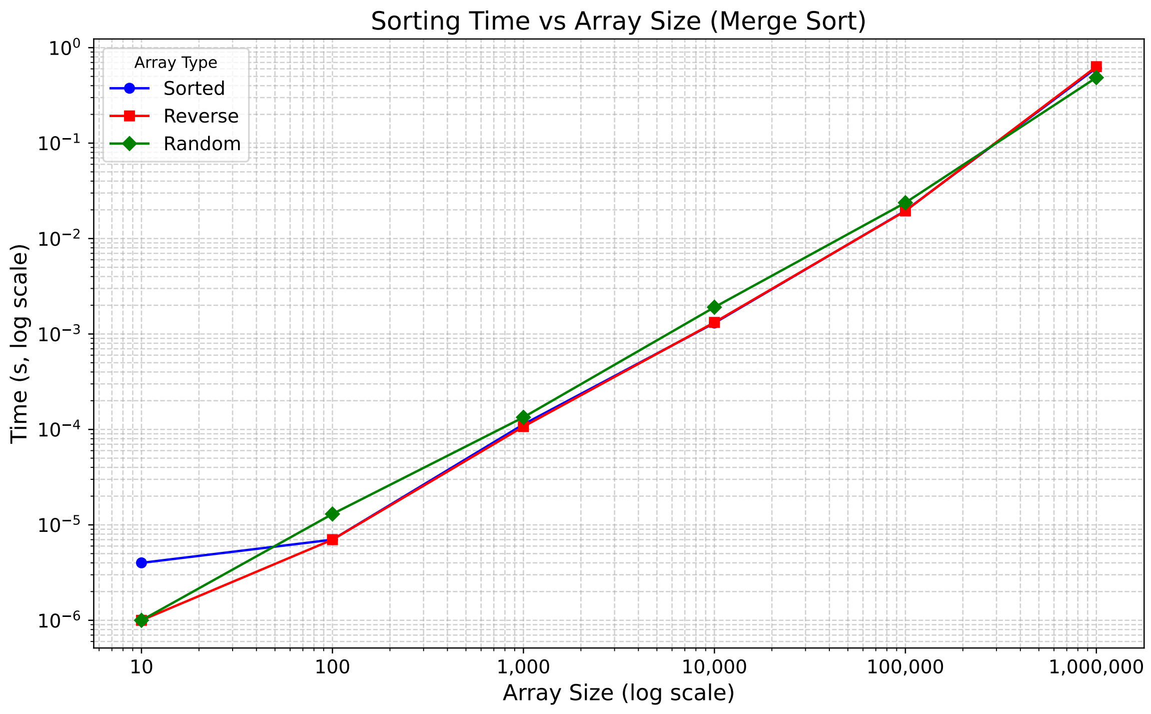

Figure 2: Comparison of base merge for array sizes and types.

As seen, mergesort holds true to its consistent $O(n \log n)$ time as expected. There is little variation amongst the array types and respective sizes, again as expected. The little variation that is seen is likely due to the inconsistencies of the JVM.
The next section explores the results obtained, coupled with an in depth analysis of the hybrid merge and insertion approach.

4 Threshold Results and Analysis

We first look at the most common, best performing frequencies of each respective threshold values amongst all the data. The best frequencies were chosen based of the best time for each grouped array size and time value pair for unsorted arrays only, as seen in figure 3) Here we see that a SORT_THRESHOLD value of 32 is preffered with a MERGESORT_THRESHOLD value of 16 or 32. Next we take a look at the average time vs the respective mergesort and sort thresholds for unsorted and sorted arrays of size 10^6. Figures 4) & 5) showcase these results.

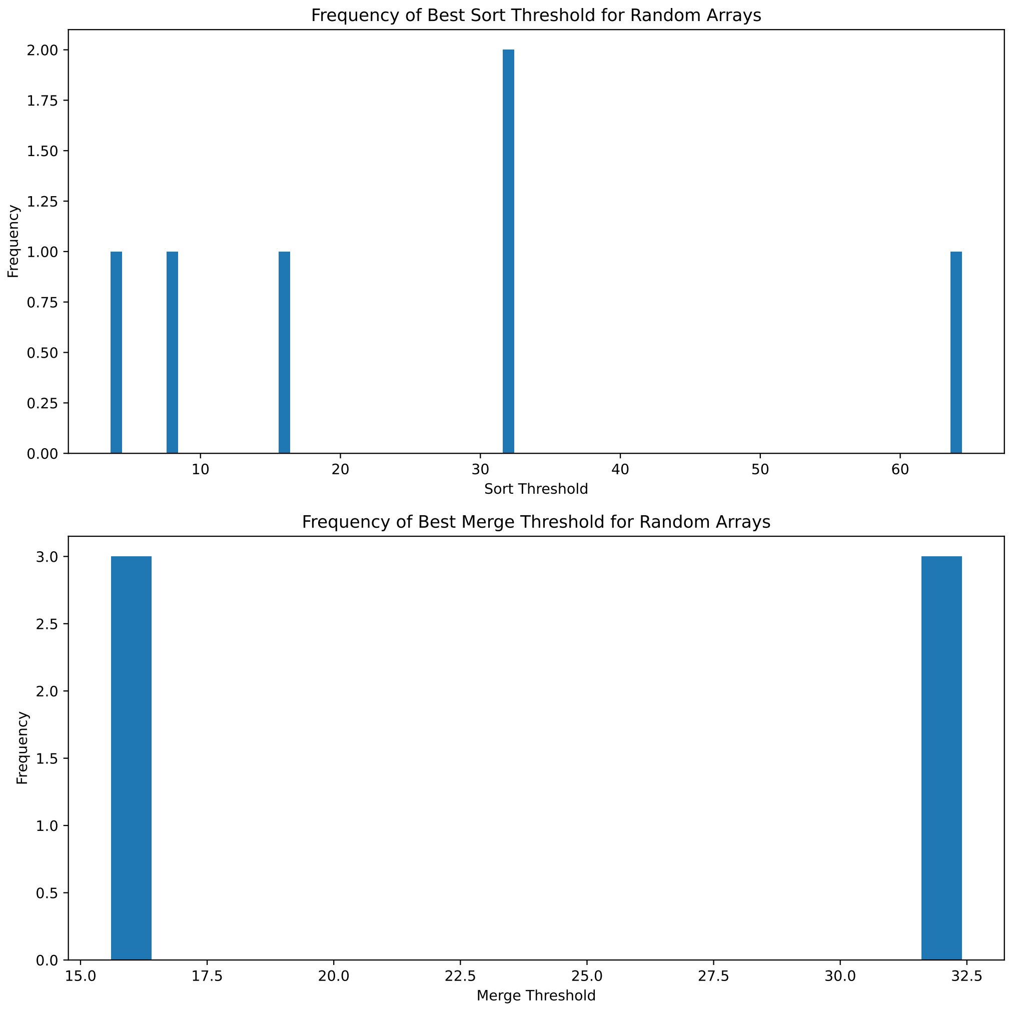

Figure 3: Frequencies of the best merge and insertion sort thresholds.

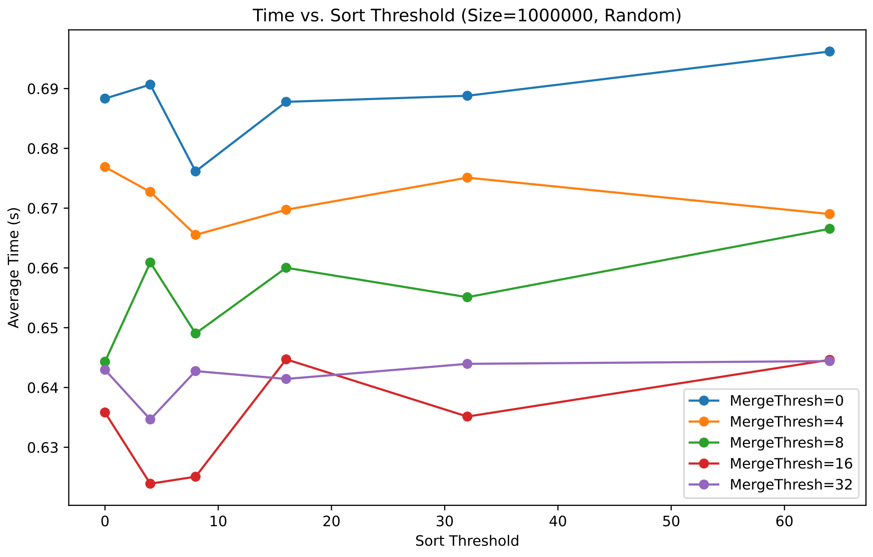

Figure 4: Respective sort thresholds for an averaged time of an array size of 10^6 for unsorted arrays.

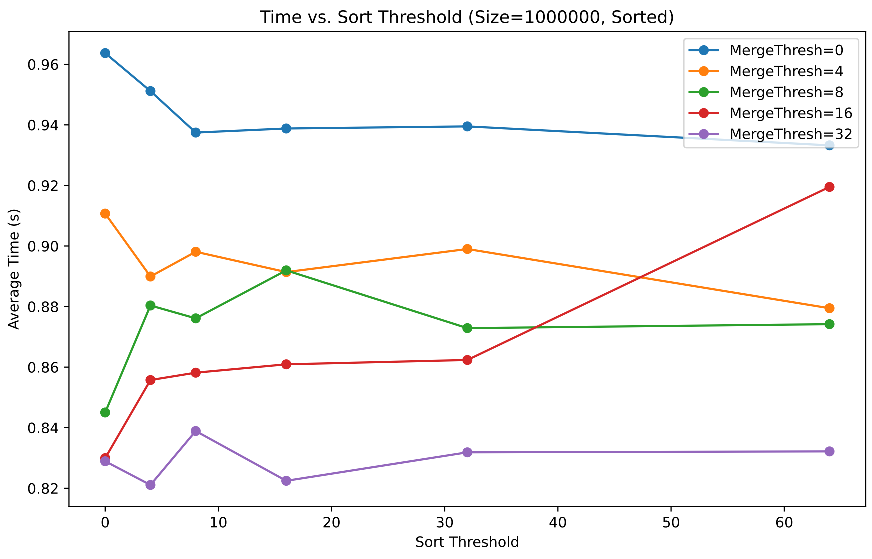

Figure 5: Respective sort thresholds for an averaged time of an array size of 10^6 for sorted arrays.

Figure 4 & 5 showcases that mergesort performs better for unsorted arrays than sorted arrays with a higher array size. Here, we see that in every case, that there is about a 20ms time difference for each respective threshold pair when comapring the sorted and unsorted arrays. This is to be expected as the amount of recursive merge calls dominate the amount of insertion calls at such large array sizes. And as we know, mergeosrt performs better on unsorted arrays than sorted arrays [8].
We now do the same comparison but for smaller array sizes of 10^4 in figures 6 & 7.

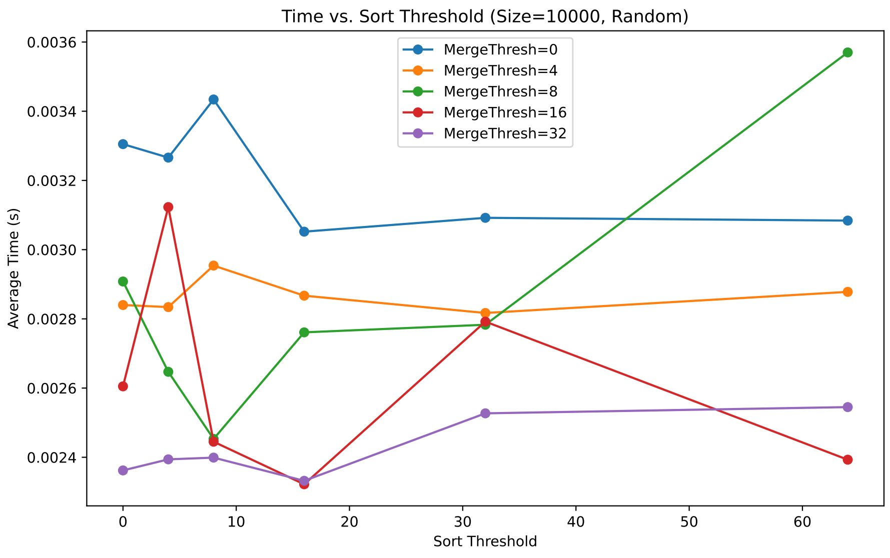

Figure 6: Respective sort thresholds for an averaged time of an array size of 10^4 for unsorted arrays.

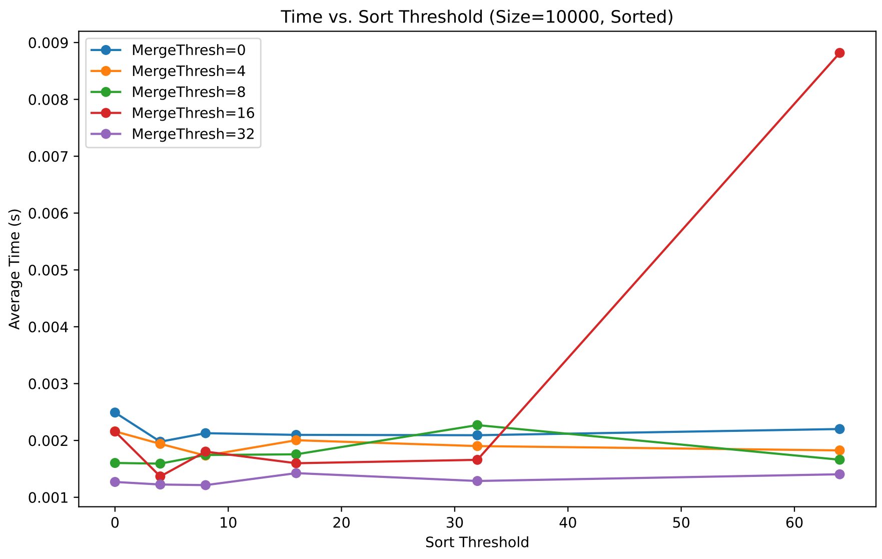

Figure 7: Respective sort thresholds for an averaged time of an array size of 10^4 for sorted arrays.

We can see a clear and immediate difference. For the sake of simplicity (and the amount of pages), this is a common trend. As array size decreases, the less the recusive merge calls dominate and the more the insertion calls (percentage wise) has an effect. Here we see that the percentage difference is less, and both figures' times are similar. Again, note that the variances seen is likely due to JVM consistencies. The point here is that the times are effectively really similar.
The next figure, 8 shows the average times for all respective sort thresholds for random arrays (averaged across all array sizes).

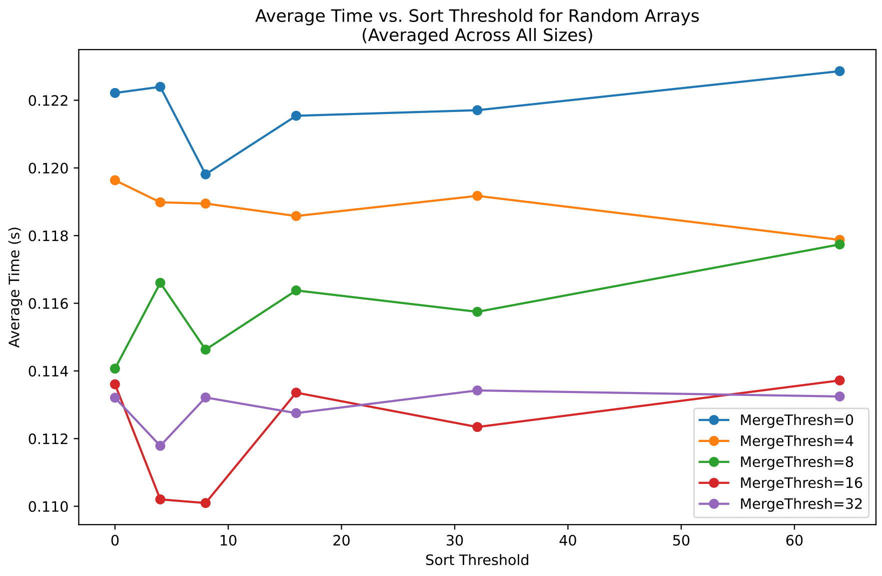

Figure 8: Average times for all respective sort thresholds for random arrays (averaged across all array sizes).

It is evident that low values of SORT_THRESHOLD such as ranges (0, 4, 8, 16, 32) work well with MERGESORT_THRESHOLD values of (16 and 32). This is reitterated by figure 3. Clearly a SORT_THRESHOLD value of 32 and a MERGESORT_THRESHOLD value of 16 and 32 is optimal. To determine the best pair, we take the minimum value of the two (averaged accross array sizes and types). The optimal value turns out to be a SORT_THRESHOLD value of 32 and a MERGE-

SORT_THRESHOLD of 32. The difference and determining factor of the two values (that we tested the minimum for) are again rough estimates due to the inconsitencies of the JVM.
And finally, now that we have our 'penultimate pair' of respective threshold values for the hybrid sort as well as a base case for mergesort, we can compare the two, as seen in figure 9.

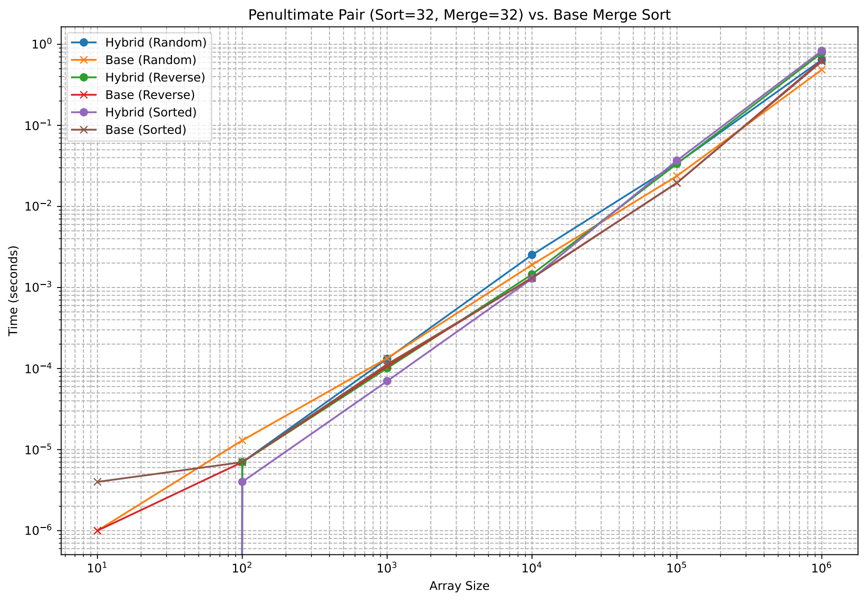

Figure 9: Penultimate threshold value pair vs the base mergesort case.

We can see clear variation amongst each of the data points. Is this difference significant enough to be due to our thresholds or due to the dreaded inconsistencies of the JVM? (more on this next section). For smaller array sizes such as 10^1, 10^2 & 10^3 we can see that the hybrid sort does outperform the base case, even if it is only a slight improvement. But for array sizes higher, the difference is minimal. Accross all different array types we still see almost no clear difference. The only difference we see is, again, at smaller array sizes.
The next section provides a conclusion for our findings and our report in general.

5 Conclusion

Implementing a variation of a well known sorting algorithm needs to be worth the time and effort that it takes to implement it in the first place. As seen in our case, the base mergesort was outperformed by the hybrid sort with a SORT_THRESHOLD value of 32 and a MERGESORT_THRESHOLD value of 32 for array sizes less than and equal to 10^3 +- some variation for all cases of array types. This improvement was minimal. It was not significant enough to be worth the time it took to even think about using and implementing the hybrid sort in the first place. For array sizes this small, it might be optimal, but it is not realistic as waiting an extra milisecond or 2 wont make a difference at such a small order of magnitude in practice. For higher array sizes and array types, the difference was also not notable. For our experimental environment as well as our results, I conclude that the hybrid sort, even with optimal respective sort threshold values, is not significant enough to be used or implemented for the effort that it is worth. However, this is not to say that the hybrid sort is not better than the base mergesort or vice-versa. Given different testing circumstances such as using JMH to attempt in trying to iron out the inconsistencies of the JVM (which was a big factor as seen in our results), and/or perfoming space complaxity analysis alongside time complexity analysis, we could make a more informed decision on which sort method is actually better. As it stands, I do not believe that the miniscule improvement of the hybrid sort (if it even is better) grants it to be worthwhile implementing.

References

[1] GeeksforGeeks, “Time and space complexity analysis of merge sort,” https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-merge-sort/, 2024, accessed: 2025-03-10.

[2] ——, “Time and space complexity of insertion sort algorithm,” https://www.geeksforgeeks.org/time-and-space-complexity-of-insertion-sort-algorithm/, 2024, accessed: 2025-03-10.

[3] A. Mytnik, “Warming up the jvm,” https://medium.com/@anton_mytnik/warming-up-the-jvm-89101f8d9217, 2019, accessed: 2025-03-10.

[4] T. D. Science, “Random seeds and reproducibility,” https://medium.com/towards-data-science/random-seeds-and-reproducibility-933da79446e3, 2020, accessed: 2025-03-10.

[5] R. Sedgewick and K. Wayne, “Insertion sort,” in *Algorithms*, 4th ed. Princeton University: Addison-Wesley, 2011, p. 250.

[6] ——, “Merge sort,” in *Algorithms*, 4th ed. Princeton University: Addison-Wesley, 2011, p. 270.

[7] D. Gottlieb, “Merge sort,” 2002. [Online]. Available: https://cs.nyu.edu/~gottlieb/courses/2000s/2002-03-fall/alg/lectures/lecture-21.html

[8] E. T. Oladipupo and O. C. Abikoye, “Comparison of quicksort and mergesort,” *Third International Conference on Computing and Network Communications (CoCoNet 2019)*, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9052201