Performance Analysis of Merge Sort Variants

Conrad Voigt

CSCI 311: Algorithm Analysis

Instructor: Dr. Hala El Aarag

November 19, 2024

Introduction

Merge Sort is an important comparison-based sorting algorithm that works on the divide and conquer principle. It divides the input array into two halves, sorts each half recursively, and finally merges the sorted halves to produce a fully sorted array. With a time complexity of O(nlogn) Merge Sort is known for its efficiency which makes it significantly faster than other sorting algorithms like Bubble Sort or Insertion Sort, especially as the size of the dataset grows.

The main objective of this project is to perform a comparative performance analysis of the Merge Sort algorithm using four different implementations. By implementing these variants in Python and running them on datasets of various types and sizes, this study evaluates their performance under different conditions. Key performance metrics, including execution time and memory usage, are analyzed to determine how each implementation handles diverse data structures and sizes. The analysis ultimately provides insights into best practices for implementing Merge Sort in specific scenarios, such as when memory usage is a key constraint or when the data is already sorted.

Methodology

The four variants of Merge Sort were implemented in Python based on pseudocode provided in Neapolitan et al. (2015) The implementation process involved translating the different Merge Sort variations written in C++ into Python syntax and ensuring that each function correctly performed the core operations of dividing, sorting, and merging subarrays

To evaluate the performance of these implementations under various conditions, datasets of different structures and sizes were generated. Four types of data were used for testing which were random, sorted, almost sorted and reversely sorted data. These different configurations provided a robust foundation for analyzing the performance and adaptability of each Merge Sort variant.

Implementation of Merge Sort Variants

The four different implementations used in this analysis have distinct properties, which will be explained in greater detail below.

Mergesort 1 (Recursive)

Mergesort1 is a classic recursive implementation of the Merge Sort algorithm that follows the divide-and-conquer approach. The array is repeatedly divided into smaller halves until each subarray contains only one element. Then, a merge function combines these smaller arrays in sorted order. This implementation uses temporary storage during merging, which helps maintain stability, meaning equal elements retain their original order. The pseudocode for Mergesort1 is provided in *Appendix A/1.*

Mergesort 2 (Modified Recursive)

Mergesort2 is another recursive implementation of Merge Sort, similar to Mergesort1. The main difference lies in how it handles the boundaries of the array during recursion, using explicit index management to split and merge the subarrays. While the underlying logic is the same as Mergesort1, this approach introduces minor variations that could affect performance in some cases. Temporary storage is also used in the merge step, preserving the algorithm's stability. The pseudocode for Mergesort2 can be found in *Appendix A/2.*

Mergesort 3 (Iterative Version)

Mergesort3 is an iterative, bottom-up version of Merge Sort. Instead of using recursion, it starts with individual elements as sorted subarrays of size 1. The algorithm then repeatedly doubles the subarray size and merges them until the entire array is sorted. This method avoids the need for recursive calls,

making it more efficient in environments where recursion depth is limited. It also provides better control over memory usage. The pseudocode for Mergesort3 is available in *Appendix A/3.*

Mergesort 4 (Linked List Version)

Mergesort4 is specifically designed to work with linked lists instead of arrays. It splits the linked list into two halves, sorts each half, and merges them back together using pointer manipulation. This approach is especially useful for linked lists because it avoids the need to shift elements during the merging step. Mergesort4 is stable and performs well when working with data stored in linked structures. The pseudocode for Mergesort4 is included in *Appendix A/4.*

Correctness Testing

To ensure that each Merge Sort variant produced correctly sorted output, a correctness test was conducted on a small dataset of 10 randomly generated integers. Each algorithm was applied to the dataset, and the output *(Appendix B)* was manually verified by comparing the sorted result against Python's built-in sorted function. This verification step ensured that each implementation correctly sorted the data, establishing a foundation for subsequent performance testing on larger datasets. The verification test also served as a quick check to ensure no logical errors were present in the code before moving on to more extensive analysis.

Experimental Setup

The performance tests for this project were conducted on a computer with the following specifications: an Intel Core i7-9700K processor with 8 cores and a base clock speed of 3.6 GHz (boosting up to 4.9 GHz), 16 GB of DDR4 RAM, and the Windows 10 (64-bit) operating system. These specifications provided sufficient computational resources to handle datasets of varying sizes, ensuring accurate and reliable measurements of execution time and memory usage across the different Merge Sort implementations.

To evaluate the efficiency of each Merge Sort variant, two key performance metrics were recorded: execution time and memory usage. Execution time was measured in milliseconds by using Python's time.perf_counter_ns() function. The start and end times for each sorting operation were recorded, and the difference, initially calculated in nanoseconds, was converted to milliseconds for clarity. Each dataset configuration was tested 100 times, and the average execution time across these runs was computed to minimize the impact of random delays or fluctuations. Memory usage was estimated by calculating the memory footprint of the main data structures involved in sorting. Python's sys.getsizeof() function was used to measure the size of primary arrays, including input arrays and temporary arrays, to approximate memory usage in bytes. This provided a reliable method to assess the additional memory allocation required by each implementation, reflecting the overhead associated with the Merge Sort process.

To thoroughly evaluate the performance of each Merge Sort variant, test cases were generated with varying data structures and sizes. The datasets used for testing included arrays of sizes 10, 100, 1,000, 10,000, 100,000, and 1,000,000. This range of sizes allowed for an analysis of how well each implementation scaled from small to very large datasets. Furthermore, four distinct types of data configurations were tested for each size. Random data consisted of arrays filled with randomly generated integers, simulating typical unordered datasets. Sorted data comprised arrays already arranged in ascending order, allowing the evaluation of how efficiently each algorithm handled pre-sorted data. Reverse sorted data represented a worst-case scenario, with arrays arranged in descending order, requiring the algorithms to completely reorder the elements. Finally, nearly sorted data consisted of arrays that were mostly sorted but included a few random swaps, simulating real-world datasets that might require minimal adjustments.

Each Merge Sort variant was applied to these configurations to observe and analyze its performance under different conditions, providing comprehensive insights into their scalability, efficiency, and adaptability to various data types and sizes.

Analysis and Discussion

Random Data

From *Figure 1* (Execution Time Performance for Random Data), Mergesort1 and Mergesort4 stand out as the best-performing algorithms in terms of execution time, with Mergesort1 exhibiting a slight edge for larger datasets. Mergesort2 and Mergesort3 are comparatively slower, with Mergesort3 being the least efficient for random data. These differences can be attributed to the design of each algorithm: Mergesort1 employs a simple and efficient recursive approach, while Mergesort4 uses pointer-based operations in its linked list implementation to achieve enhanced performance. On the other hand, Mergesort3 incurs extra computational costs due to its iterative structure, and Mergesort2 introduces additional boundary-handling cost. All implementations conform to the expected theoretical O(nlogn) time complexity, as illustrated by the trendlines and the R^2 very close to 1.

In *Figure 2* (Memory Usage Performance for Random Data), Mergesort4 again proves to be the most memory-efficient implementation, thanks to its linked list structure that eliminates the need for temporary arrays. In contrast, Mergesort1, Mergesort2, and Mergesort3 all require additional memory for their temporary arrays, leading to higher memory consumption. As the array size increases, Mergesort4 consistently uses less memory, making it particularly appealing for large datasets where memory efficiency is critical. However, all implementations exhibit linear memory growth, complexity of O(n), which can be seen by the perfect fit of the trendline to the observed data with an R^2 of 1.0000.

Overall, Mergesort4 is the most balanced implementation for random data, offering both superior memory efficiency and fast execution time. Mergesort1 provides a strong alternative for array-based applications, while Mergesort2 and Mergesort3 are less suited for handling large-scale random datasets due to their higher overhead.

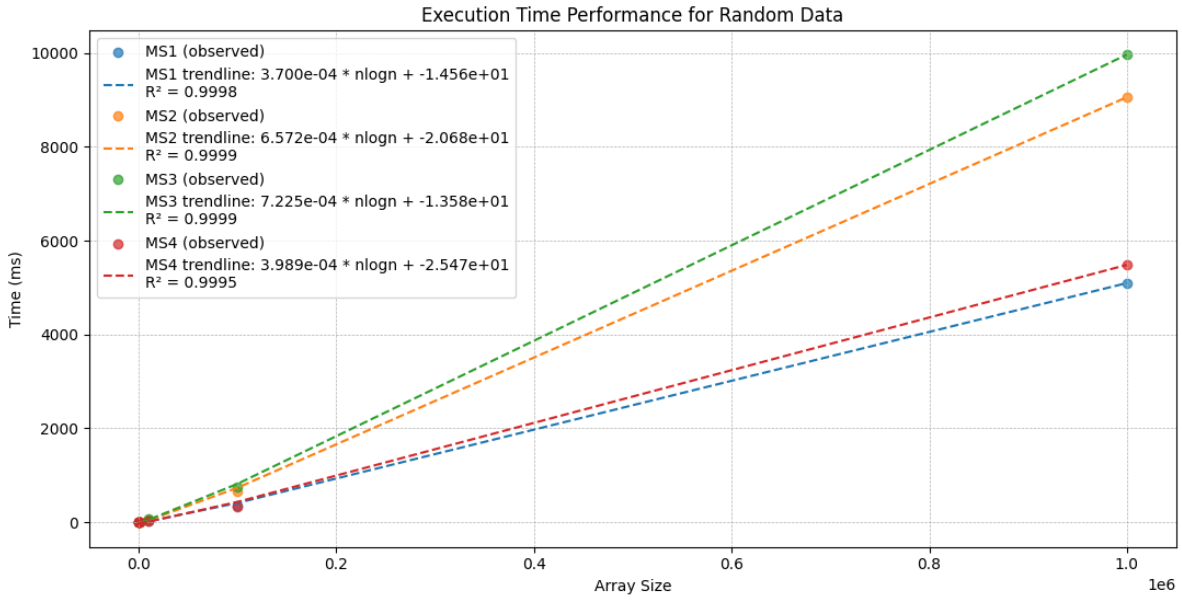

Figure 1: Execution Time Performance of Mergesort 1 through 4 for Random Data.

Figure 2: Memory Usage Performance of Mergesort 1 through 4 for Random Data

Sorted Data

From *Figure 3* (Execution Time Performance for Sorted Data), Mergesort4 demonstrates the fastest execution time across all array sizes, followed by Mergesort1. Both algorithms effectively capitalize on their designs to handle already sorted data efficiently. Mergesort2 and Mergesort3, on the other hand, exhibit slower performance, with Mergesort3 slightly outperforming Mergesort2. The superior performance of Mergesort4 is attributable to its pointer-based operations, which minimize redundant operations when the data is pre-sorted. Mergesort1 also benefits from its straightforward recursive structure, while Mergesort2's boundary handling and Mergesort3's iterative merging process add extra cost. Despite these differences, all implementations adhere to the O(nlogn) time complexity, shown by the R^2 very close to 1.

In *Figure 4* (Memory Usage Performance for Sorted Data), Mergesort4 remains the most memory-efficient implementation due to its linked list design, which avoids auxiliary storage. Conversely, Mergesort1, Mergesort2, and Mergesort3 exhibit similar memory usage, as they all rely on temporary arrays during the merge phase. While Mergesort4 consistently requires less memory, confirming its status as the most efficient algorithm for sorted data, all implementations exhibit linear O(n) memory growth, shown once again with the perfect R^2.

Overall, for sorted data, Mergesort4 is the most efficient implementation, offering unmatched performance in terms of both execution time and memory usage. Mergesort1 also performs well, making it a viable choice for environments where array-based operations are preferred. Meanwhile, Mergesort2 and Mergesort3 are less suitable for handling large, sorted datasets due to their relatively higher execution times and memory demands.

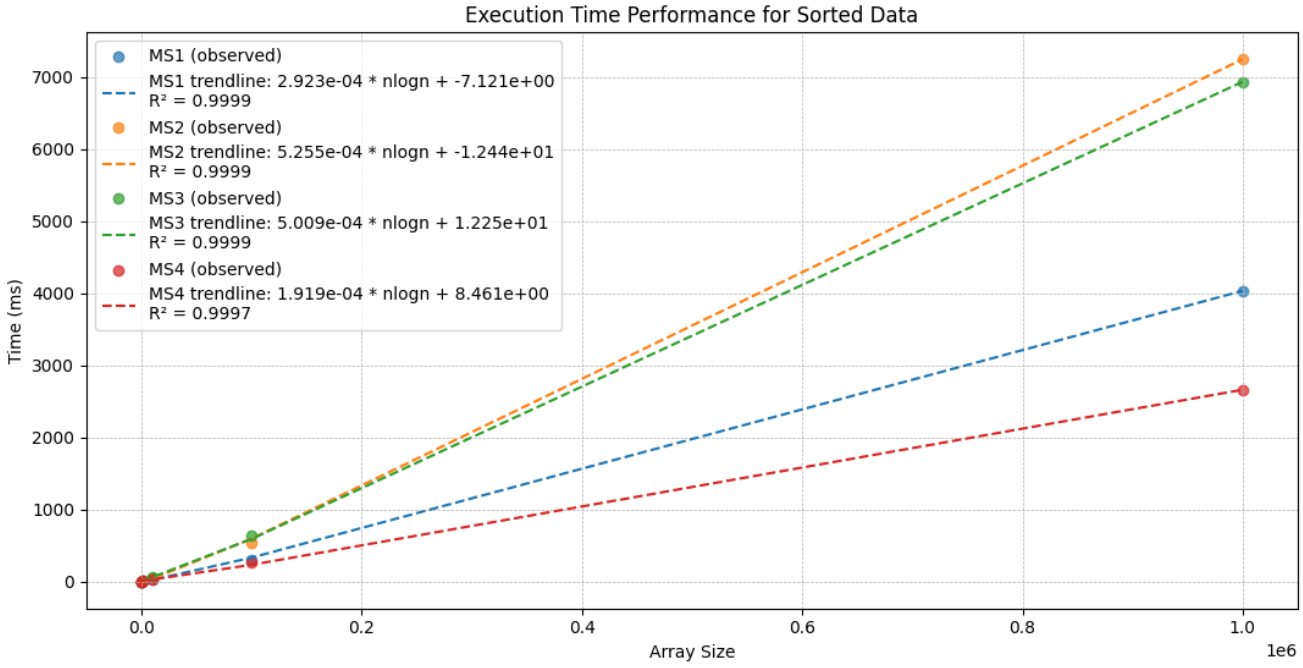

Figure 3: Execution Time Performance of Mergesort 1 through 4 for Sorted Data

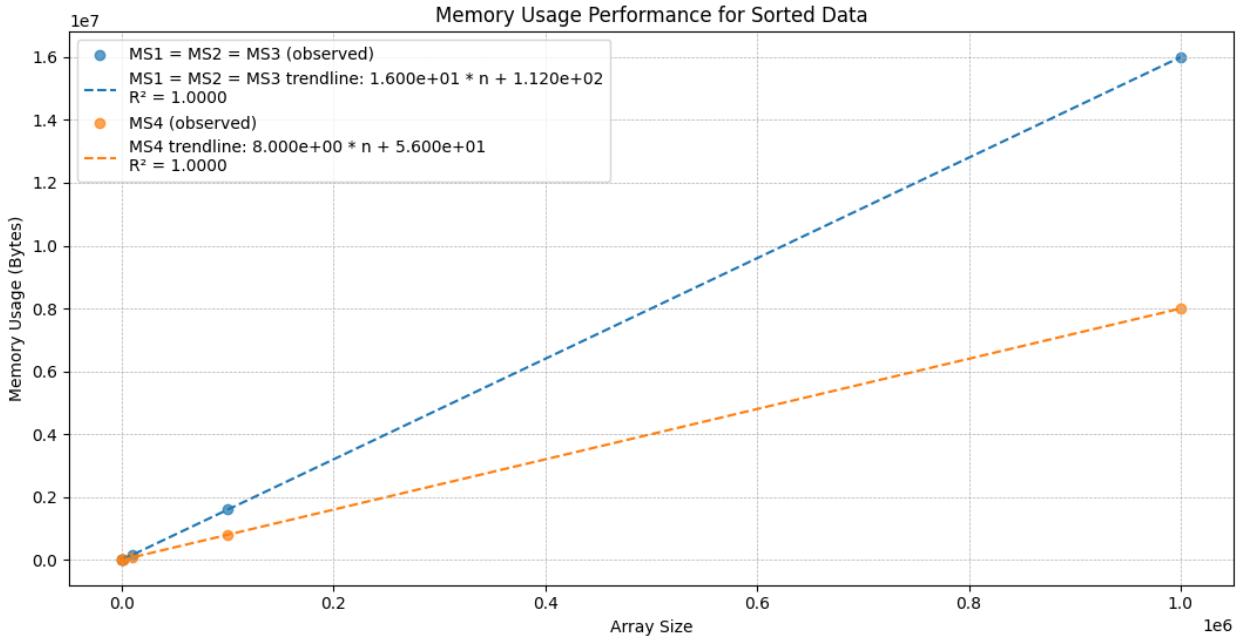

Figure 4: Memory Usage Performance of Mergesort 1 through 4 for Sorted Data

Nearly Sorted Data

From *Figure 5* (Execution Time Performance for Nearly Sorted Data), Mergesort4 again leads in terms of execution time across all array sizes, followed by Mergesort3. Both implementations excel at efficiently sorting data that is almost in order, using their structural advantages to minimize unnecessary operations. Mergesort2 and Mergesort1, however, demonstrate slower performance, with Mergesort1 slightly outperforming Mergesort2 for larger sizes. The exceptional performance of Mergesort4 stems from its linked list-based design, which effectively handles slight disarray in the data while the other implementations introduce additional overhead leading to slower execution times. Despite these differences, all implementations adhere to the theoretical O(nlogn) time complexity.

In *Figure 6* (Memory Usage Performance for Nearly Sorted Data), Mergesort4 proves to be the most memory-efficient algorithm, requiring minimal auxiliary storage due to its pointer-based operations. In contrast, Mergesort1, Mergesort2, and Mergesort3 all rely on temporary arrays, resulting in higher memory consumption. These algorithms all show linear O(n) memory growth, but Mergesort4 consistently uses less memory, making it ideal for scenarios where efficiency is important.

For nearly sorted data, Mergesort4 stands out as the most efficient implementation, providing faster execution times and lower memory usage. Mergesort3 is a reliable alternative for array-based applications, while Mergesort2 and Mergesort1 are less suited for such datasets due to their higher overhead.

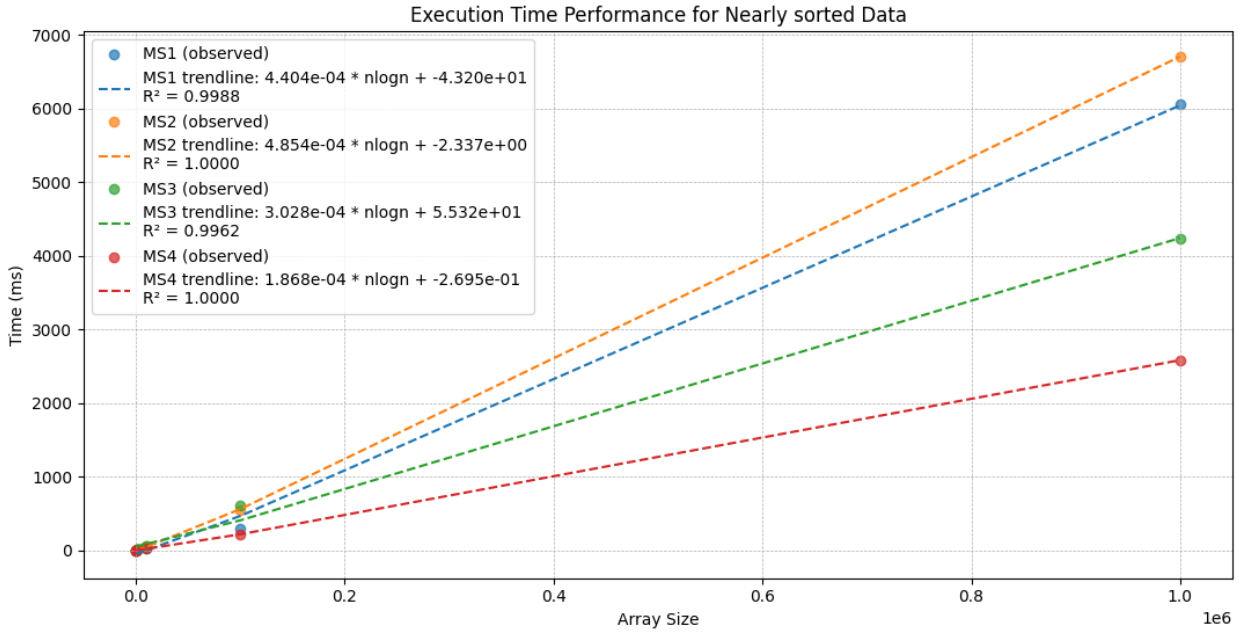

Figure 5: Execution Time Performance of Mergesort 1 through 4 for Nearly Sorted Data

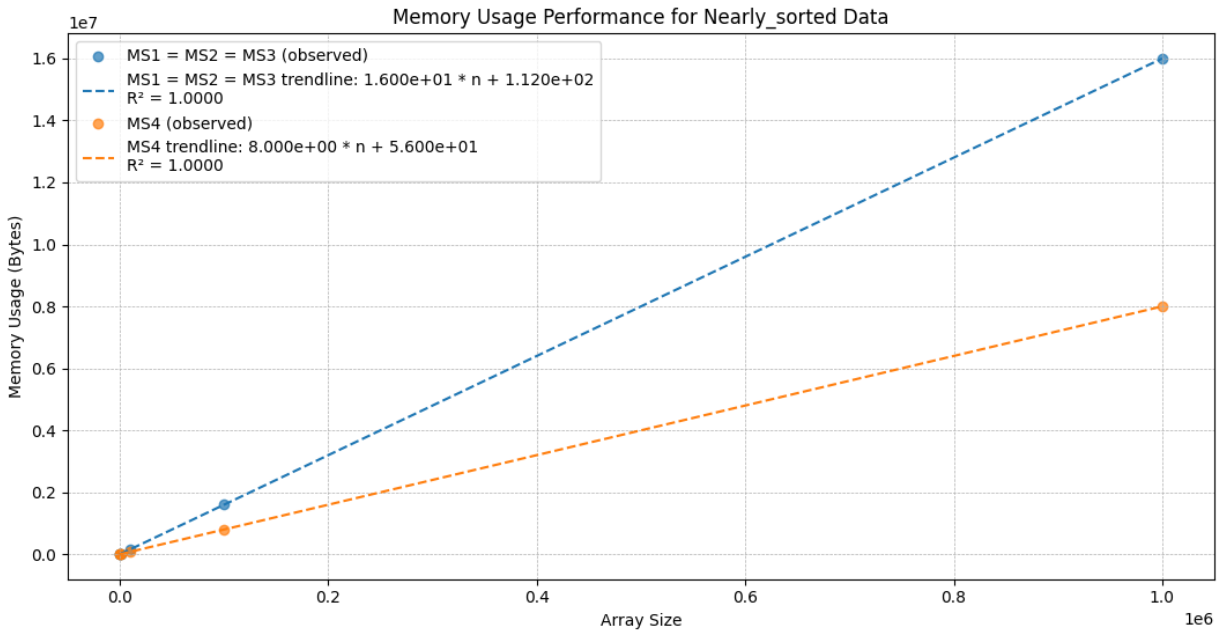

Figure 6: Memory Usage Performance of Mergesort 1 through 4 for Nearly Sorted Data

Reverse Sorted Data

From *Figure 7* (Execution Time Performance for Reverse Sorted Data), Mergesort4 retains its position as the fastest implementation, The other implementations again exhibit slower performance, with Mergesort1 slightly outperforming Mergesort2 and Mergesort3. The superior performance of Mergesort4 is due to its pointer-based linked list structure, which reduces unnecessary operations during the merging process. All other variants introduce additional overhead, resulting in slower execution times. All implementations conform to the expected O(nlogn) time complexity, as reflected in the trendlines and the high R^2.

In **Figure 8** (Memory Usage Performance for Reverse Sorted Data), Mergesort4 maintains its position as the most memory-efficient implementation. Its linked list design minimizes auxiliary storage requirements, while Mergesort1, Mergesort2, and Mergesort3 exhibit higher memory consumption due to their reliance on temporary arrays. All algorithms scale linearly with O(n) memory growth (R^2=1) but Mergesort4 consistently requires less memory, reinforcing its efficiency.

For reverse sorted data, Mergesort4 is the most suitable choice, offering the best combination of execution time and memory efficiency. All other implementations are less effective due to their higher resource demands.

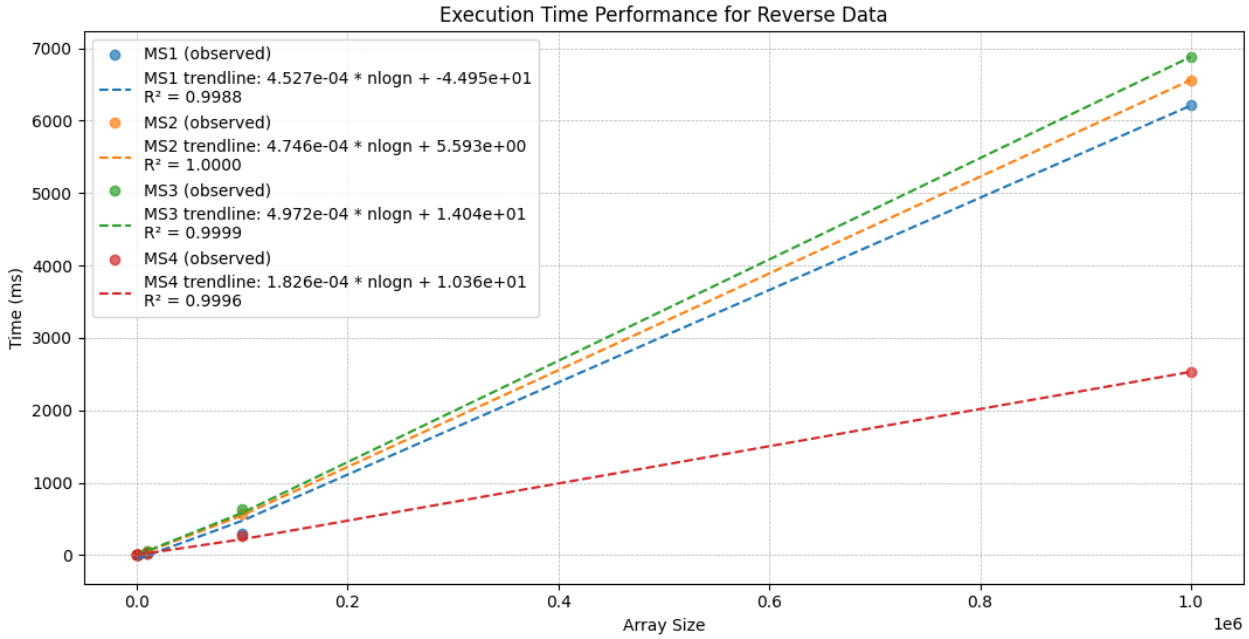

Figure 7: Execution Time Performance of Mergesort 1 through 4 for Reversely Sorted Data

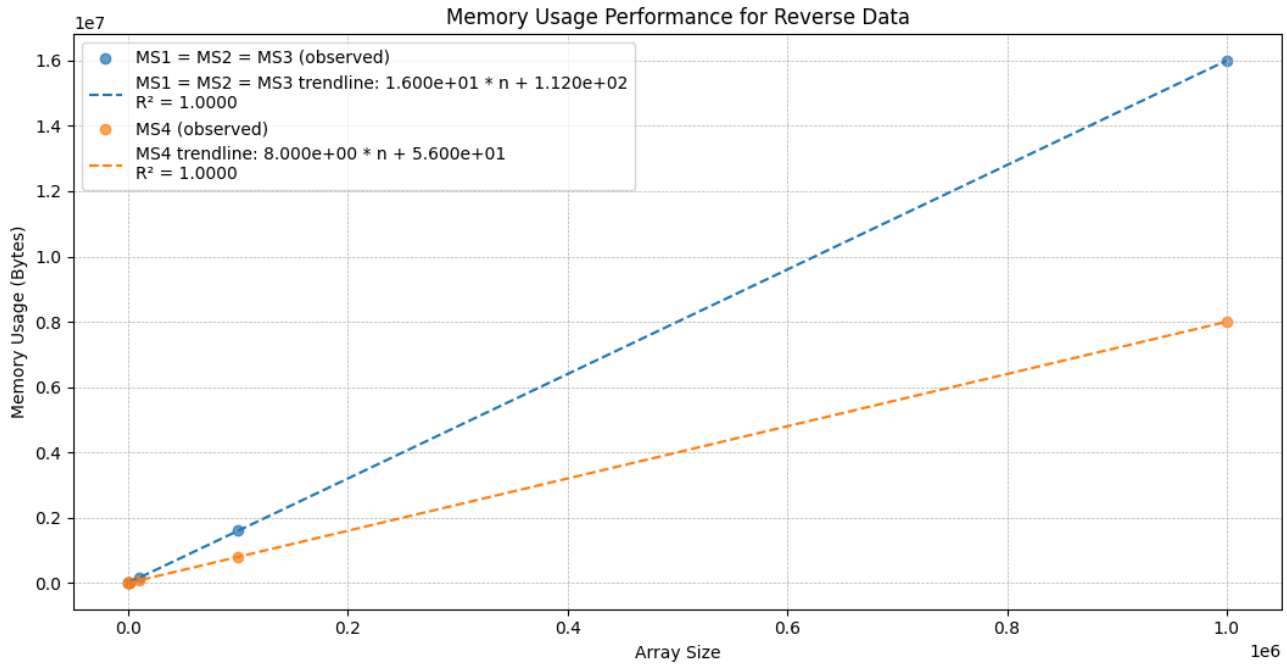

Figure 8: Memory Usage Performance of Mergesort 1 through 4 for Reversely Sorted Data

Conclusion

This project analyzed the performance of four Merge Sort implementations—Mergesort1, Mergesort2, Mergesort3, and Mergesort4—on datasets of different sizes and configurations. The findings highlight the strengths and weaknesses of each implementation and provide guidance on their best use cases.

Mergesort4 consistently outperformed the other variants in both execution time and memory usage. Its linked list-based design eliminated the need for temporary arrays, making it highly memory-efficient. This advantage, combined with its fast execution times, made Mergesort4 the best choice for all dataset types, particularly for large or memory-intensive tasks. Its ability to handle random, sorted, nearly sorted, and reverse sorted datasets with minimal overhead demonstrates its versatility and efficiency.

Mergesort1 also performed well, especially in terms of execution time. Its straightforward recursive approach made it a strong option for array-based applications. However, its reliance on temporary arrays led to higher memory usage compared to Mergesort4. Even so, Mergesort1 provided consistent and reliable performance, making it a solid alternative when linked lists are not used.

Mergesort3, the iterative version, was slower overall due to the additional overhead of its iterative merging process. While it may not be the fastest implementation, it can be useful in environments where recursion depth is a concern. Mergesort2, on the other hand, was the least efficient in both execution time and memory usage. Its extra boundary-handling overhead during recursive calls made it less suitable for large or complex datasets.

Dataset type also played a role in performance. Mergesort4 excelled in all cases, showing particularly strong results for nearly sorted and reverse sorted data. Mergesort1 also handled these datasets well, though its performance lagged slightly behind Mergesort4 for larger sizes. Both Mergesort2 and Mergesort3 struggled with more complex datasets, further emphasizing the efficiency of Mergesort4 and Mergesort1.

In terms of time complexity, all implementations adhered to the theoretical O(nlogn), as shown by the trendlines with R^2 values close to 1. Memory usage grew linearly, complexity O(n) with a perfect R^2 of 1, and Mergesort4 consistently used less memory than the other variants.

In conclusion, Mergesort4 is the best overall implementation, combining fast execution times with low memory usage. Mergesort1 is a strong alternative for array-based applications, while Mergesort2 and Mergesort3 are less suited for large or demanding datasets. These findings demonstrate the importance of choosing the right Merge Sort variant based on the specific requirements of the dataset and computational environment.

References

Neapolitan, R., & Naimipour, K. (2015). *Foundations of algorithms* (5th ed.). Jones and Bartlett Learning. ISBN-13: 9781284049190

McKinney, W. (2022). *Python for data analysis: Data wrangling with pandas, NumPy, and Jupyter* (2nd ed.). O'Reilly Media. ISBN: 9781491957660.

Appendices

Appendix A: ython Code for Each Merge Sort Variant

Mergesort1

```python
# Mergesort1 implementation with helper functions
def merge1(arr, temp_arr, left, mid, right):
    i = left    # Starting index for left subarray
    j = mid + 1 # Starting index for right subarray
    k = left    # Starting index to be sorted
    while i <= mid and j <= right:
        if arr[i] <= arr[j]:
            temp_arr[k] = arr[i]
            i += 1
        else:
            temp_arr[k] = arr[j]
            j += 1
        k += 1

    # Copy remaining elements of left subarray, if any
    while i <= mid:
        temp_arr[k] = arr[i]
        i += 1
        k += 1

    # Copy remaining elements of right subarray, if any
    while j <= right:
        temp_arr[k] = arr[j]
        j += 1
        k += 1

    # Copy the sorted subarray into original array
    for i in range(left, right + 1):
        arr[i] = temp_arr[i]

def mergesort1(arr, temp_arr, left, right):
    if left < right:
        mid = (left + right) // 2
        mergesort1(arr, temp_arr, left, mid)
        mergesort1(arr, temp_arr, mid + 1, right)
        merge1(arr, temp_arr, left, mid, right)

def sort_merge1(arr):
    temp_arr = [0] * len(arr)
    mergesort1(arr, temp_arr, 0, len(arr) - 1)
```

Mergesort2

```python
# Mergesort2 implementation with helper functions
def merge2(arr, temp_arr, left, mid, right):
    i = left    # Starting index for left subarray
    j = mid + 1 # Starting index for right subarray
    k = left    # Starting index to be sorted
    while i <= mid and j <= right:
        if arr[i] <= arr[j]:
            temp_arr[k] = arr[i]
            i += 1
        else:
            temp_arr[k] = arr[j]
            j += 1
        k += 1

    # Copy remaining elements of left subarray, if any
    while i <= mid:
        temp_arr[k] = arr[i]
        i += 1
        k += 1

    # Copy remaining elements of right subarray, if any
    while j <= right:
        temp_arr[k] = arr[j]
        j += 1
        k += 1

    # Copy the sorted subarray into original array
    for i in range(left, right + 1):
        arr[i] = temp_arr[i]

def mergesort2(arr, temp_arr, low, high):
    if low < high:
        mid = (low + high) // 2
        mergesort2(arr, temp_arr, low, mid)
        mergesort2(arr, temp_arr, mid + 1, high)
        merge2(arr, temp_arr, low, mid, high)

def sort_merge2(arr):
    temp_arr = [0] * len(arr)
    mergesort2(arr, temp_arr, 0, len(arr) - 1)
```

Mergesort3

```python
# Mergesort3 implementation (Iterative Version)
def merge3(arr, temp_arr, left, mid, right):
    i = left    # Starting index for left subarray
    j = mid + 1 # Starting index for right subarray
    k = left    # Starting index to be sorted
    while i <= mid and j <= right:
        if arr[i] <= arr[j]:
            temp_arr[k] = arr[i]
            i += 1
        else:
            temp_arr[k] = arr[j]
            j += 1
        k += 1

    # Copy remaining elements of left subarray, if any
    while i <= mid:
        temp_arr[k] = arr[i]
        i += 1
        k += 1

    # Copy remaining elements of right subarray, if any
    while j <= right:
        temp_arr[k] = arr[j]
        j += 1
        k += 1

    # Copy the sorted subarray into original array
    for i in range(left, right + 1):
        arr[i] = temp_arr[i]

def mergesort3(arr):
    n = len(arr)
    temp_arr = [0] * n
    size = 1

    # Iterative approach - Merge subarrays in bottom-up manner
    while size < n:
        for left in range(0, n - size, 2 * size):
            mid = min(left + size - 1, n - 1)
            right = min(left + 2 * size - 1, n - 1)
            merge3(arr, temp_arr, left, mid, right)
        size *= 2
```

Mergesort4

```
# Define a ListNode for Mergesort4 (Linked List Version)
class ListNode:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next

# Helper function to merge two sorted linked lists
def merge4(left, right):
    dummy = ListNode()
    tail = dummy
    while left and right:
        if left.value < right.value:
            tail.next = left
            left = left.next
        else:
            tail.next = right
            right = right.next
        tail = tail.next
    tail.next = left if left else right
    return dummy.next

# Recursive function for linked list mergesort
def linked_mergesort(head):
    if not head or not head.next:
        return head
    mid = get_middle(head)
    left = head
    right = mid.next
    mid.next = None
    left = linked_mergesort(left)
    right = linked_mergesort(right)
    return merge4(left, right)

# Function to find the middle node of a linked list
def get_middle(head):
    slow = head
    fast = head.next
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow

# Convert array to linked list, apply linked_mergesort, then convert back
def mergesort4(arr):
    if not arr:
```

```
        return []

    # Convert array to linked list
    head = ListNode(arr[0])
    current = head
    for value in arr[1:]:
        current.next = ListNode(value)
        current = current.next

    # Sort the linked list
    sorted_head = linked_mergesort(head)

    # Convert back to array
    sorted_arr = []
    current = sorted_head
    while current:
        sorted_arr.append(current.value)
        current = current.next

    return sorted_arr
```

Appendix B: Correctness Testing: Sample Outputs

```
Mergesort1:
Original Data: [21, 7, 9, 55, 25, 51, 30, 80, 91, 12]
Sorted Data: [7, 9, 12, 21, 25, 30, 51, 55, 80, 91]

Mergesort2:
Original Data: [17, 28, 47, 39, 35, 55, 38, 22, 44, 12]
Sorted Data: [12, 17, 22, 28, 35, 38, 39, 44, 47, 55]

Mergesort3:
Original Data: [77, 37, 80, 35, 41, 7, 65, 75, 84, 20]
Sorted Data: [7, 20, 35, 37, 41, 65, 75, 77, 80, 84]

Mergesort4:
Original Data: [29, 88, 63, 28, 14, 26, 93, 1, 20, 78]
Sorted Data: [1, 14, 20, 26, 28, 29, 63, 78, 88, 93]
```