Merge Sort Performance Analysis

By Anders Jensen

November 19, 2024

Objective: The objective of this project is to analyze and compare the performance of different implementations of Merge sort.

Comparing Merge Sort Algorithms – Initial Analysis

4 merge sort algorithms were pulled from chapters 2 and 7 in the *Foundations of Algorithms Book – Fifth Edition* by Richard E. Neapolitan. Each merge sort algorithm has its own unique qualities that differ from the others. mergesort1 and mergesort2 both use divide and conquer recursive techniques, mergesort3 uses a dynamic programming approach, and mergesort4 leverages linked lists.

While mergesort1 and mergesort2 are quite similar, mergesort1 operates based on the length of the array and the array itself, while mergesort2 operates based on indices low, high, and the array itself. Mergesort1 calls for the explicit creation of more arrays during recursion, while mergesort2 creates a single temporary array only during the merge step, which in theory should increase memory efficiency. This also implies that for mergesort1, memory consumption is proportional to the size of the current partition, and will then increase for every level of recursion. The time complexity should be approximately the same at O(nlogn) and including temporary arrays, the space complexity O(nlogn) for mergesort1 and O(n) for mergesort2.

In contrast to mergesort1 and mergesort2, mergesort3 uses a dynamic, bottom-up implementation of the algorithm. It starts by sorting small subarrays (of size 1), then merging pairs of subarrays into progressively larger sorted sections (2, 4, 8, etc.), which in turn eliminates the need for explicit recursion. It uses index slicing to split the subarrays during merging. Mergesort3 still has a time complexity of O(nlogn) and O(n) space complexity, similar to mergesort2.

While the other merge sort algorithms were array-based, mergesort4 is designed for linked lists, although these can be converted to and from arrays before and after sorting, as we have done in the program. In mergesort4, recursion is once again used, while adjusting pointers low and high without creating new subarrays. The use of linked lists eliminates the need for additional storage, which in turn should drop the space complexity to O(1), which is less than all other implementations. Mergesort4 shares the time complexity of O(nlogn).

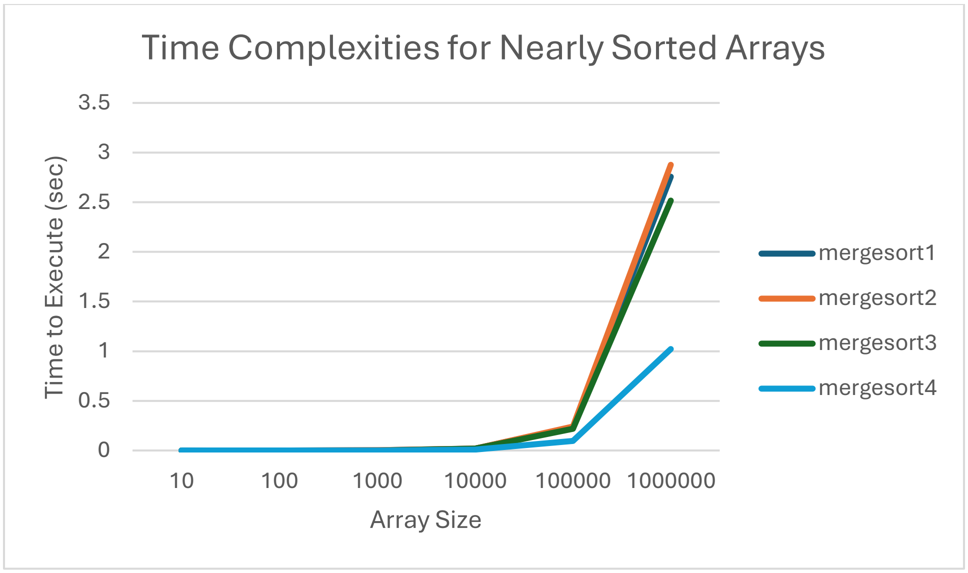
Feature	**mergesort1**	**mergesort2**	**mergesort3**	**mergesort4**
Data Structure	Array	Array	Array	Linked List
Splitting Method	Copy arrays	Indices (implicit)	Iterative, implicit	Pointer manipulation
Merge Technique	Temporary arrays	Temporary arrays	Slicing	Pointer-based
Recursion	Yes	Yes	No	Yes
Space Complexity	O(nlogn)	O(n)	O(n)	O(1)
Time Complexity	O(nlogn)	O(nlogn)	O(nlogn)	O(nlogn)

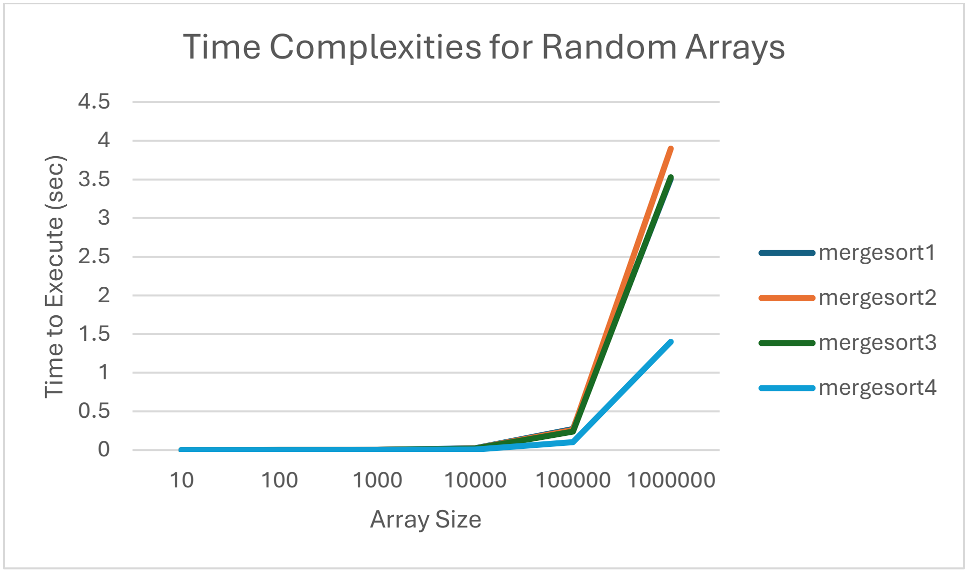
Comparing Merge Sort Algorithms – Results Analysis

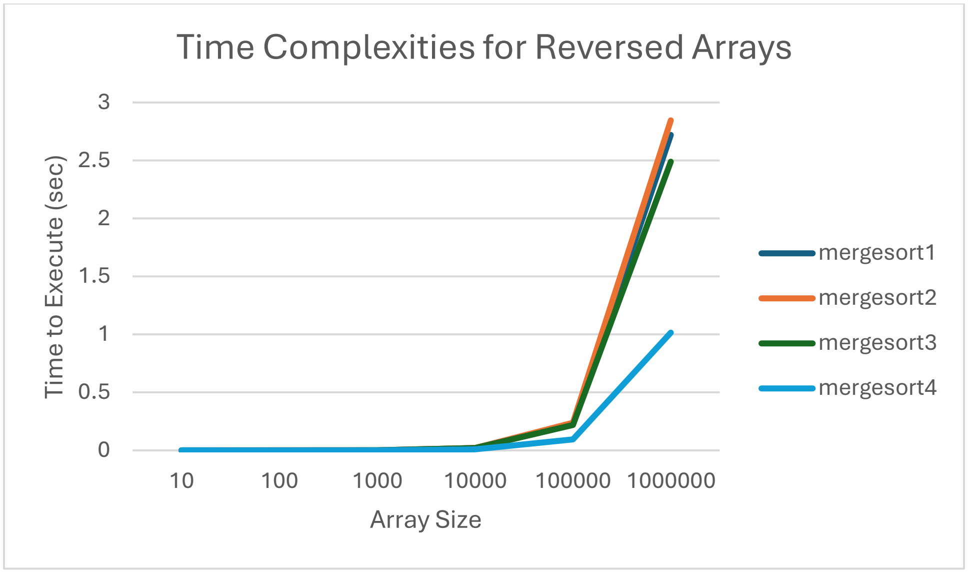
All code for this program was developed in Python. All merge sort algorithms were thoroughly tested to be accurate for arrays of sizes 10, 100, 1000, 10000, and 1000000. For the main program itself, these same sized arrays were fed 100 times and execution time was averaged, then 100 times again where memory allocation was measured using the memory_profiler module. To avoid undesired influence over execution time, the time and memory allocation were measured separately. This was repeated for random arrays, nearly sorted arrays, sorted arrays, and reversed data.

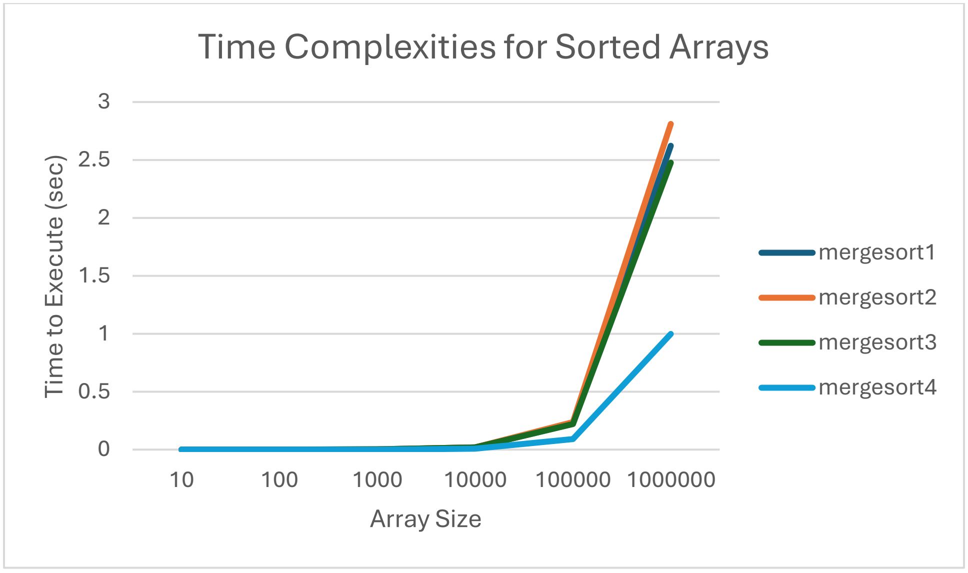
Time to execute each merge sort algorithm was short, with mergesort4 clearly leading with the fastest execution times as the arrays increased in size. For example, mergesort1 sorted an initially random array of size 10 in 0.00004 seconds and an array of size 1000000 in 3.50878 seconds. Mergesort4 performed its job on an array of size 10 in 0.00002 seconds and an array of size 1000000 in 1.39954 seconds.

Graphical representation of the data illustrates a logarithmic-shaped curve for every merge sort algorithm.

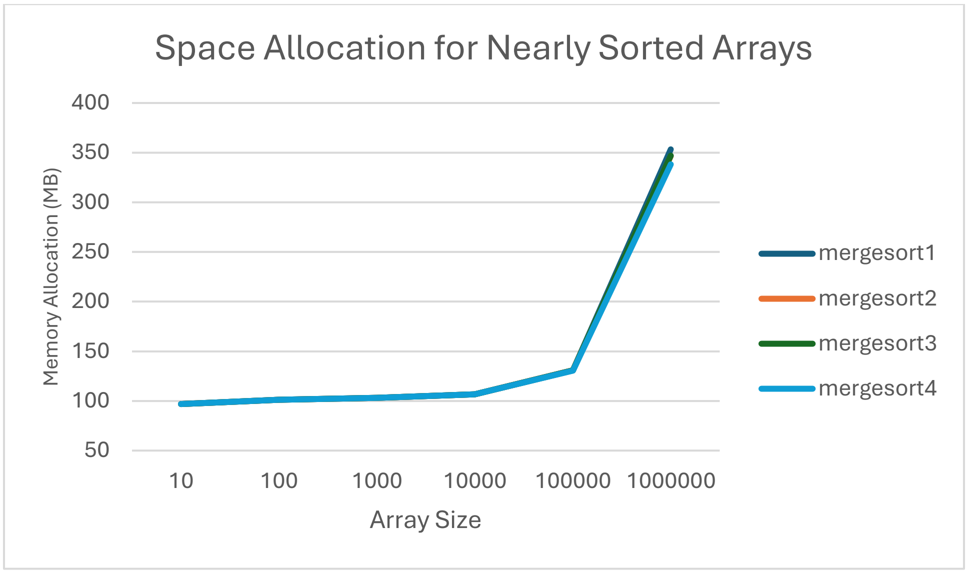


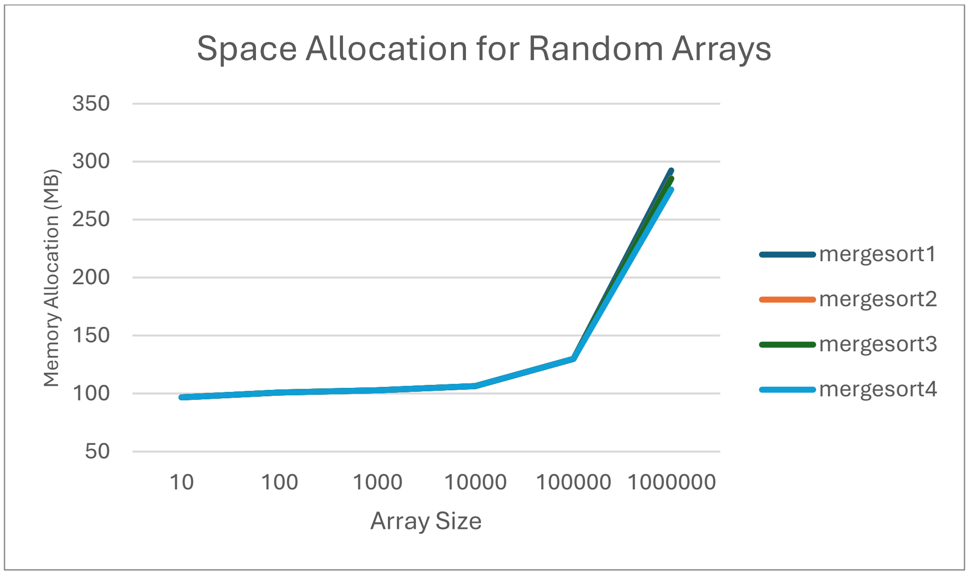


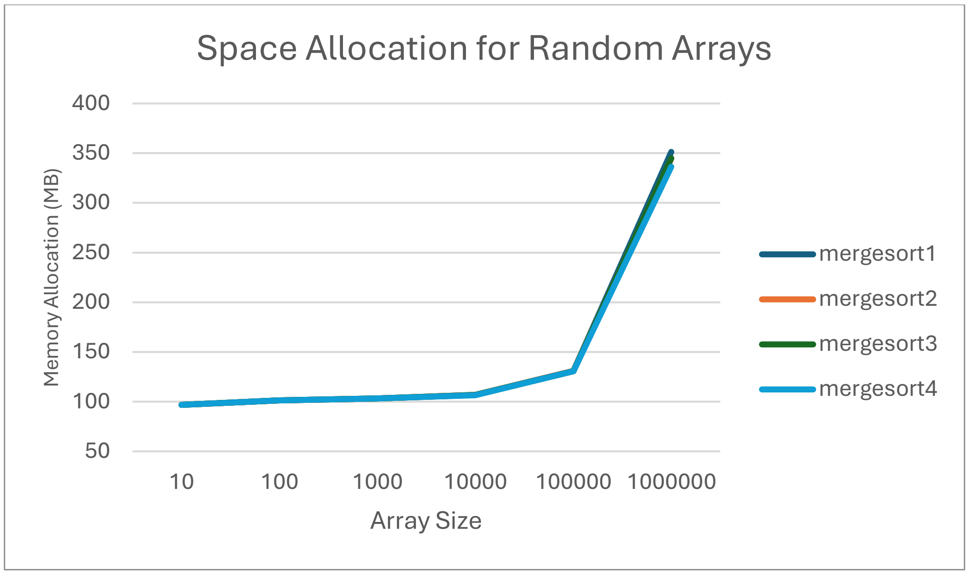

Time Complexities for Reversed Arrays
3
2.5
2
1.5
1
0.5
0
Time to Execute (sec)
10
100
1000
10000
100000
1000000
Array Size
mergesort1
mergesort2
mergesort3
mergesort4

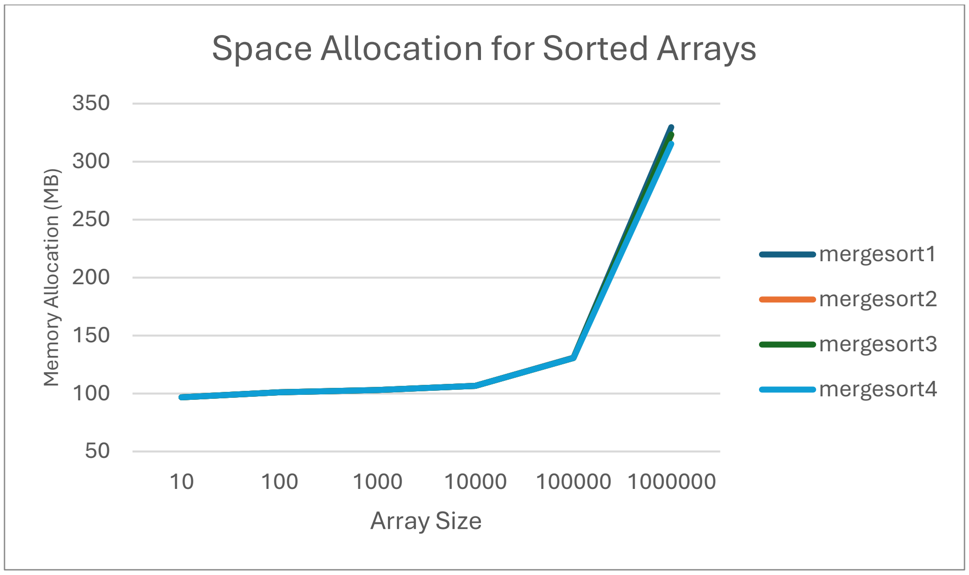

Time Complexities for Sorted Arrays
3
2.5
2
1.5
1
0.5
0
Time to Execute (sec)
10
100
1000
10000
100000
1000000
Array Size
mergesort1
mergesort2
mergesort3
mergesort4

Likewise, the comparison of memory allocation between different merge sort algorithms can be observed graphically. It can be noted that although mergesort4 had slightly less memory usage, all algorithms followed approximately the same pattern as the array size increased.






Space Allocation for Random Arrays
Memory Allocation (MB)
400
350
300
250
200
150
100
50
10
100
1000
10000
100000
1000000
Array Size
mergesort1
mergesort2
mergesort3
mergesort4


Space Allocation for Sorted Arrays
Memory Allocation (MB)
350
300
250
200
150
100
50
10
100
1000
10000
100000
1000000
Array Size
mergesort1
mergesort2
mergesort3
mergesort4

Discussion

Over the course of this project, we learned how to implement 4 different variations of the merge sort algorithm and analyze their similarities and differences. It was quite interesting to observe how each algorithm splits and manages the subarrays differently to achieve the same product. It was also interesting to see how much faster mergesort4 was with its use of linked lists, outpacing its competitors with less than ½ of their execution times. The first challenge in this project was extracting the program from the book. Spread across multiple chapters, the programs provided were not in python, and had pseudocode woven into them, making it more difficult to translate into a true, functional algorithm. Another hurdle arose while working on space allocation measurements. The memory_profiler module took significantly more time to execute a sort than without, being seconds longer, so we decided to run the time measurements and memory allocation measurements in separate function calls. This allowed for significantly more accurate results. Another note is that due to running the sorting algorithms being double the number of times and the memory_profiler taking so long, the time spent for the program to complete exceeded 8 hours. In the future, the programming techniques in these algorithms can be leveraged to create more effective and efficient code.