605.202: Data Structures

Ziyu(Yvonne) Lin

Lab 4 Analysis

Lab 4 Analysis

1. Description of data structures

In this lab, the primary data structures used include arrays, singly linked lists, etc. For all four quicksort variants, the main data structure is an array of integers. Arrays seem efficient for partitioning and element swapping during the sorting process. The natural merge sort is implemented using singly linked lists. Each node in the list contains an integer value and a reference to the next node. This choice is good for efficient memory usage, particularly during the merge steps. Merging can be achieved by simply relinking nodes rather than copying data.

In this lab, I used recursion for the quicksort variants and merge sort. Another data structure in this context is the recursive call stack. This recursive layering replaces the need for manually managed stacks or queues. And the system's call stack functions as an implicit data structure to manage the control flow.

2. Justification of the data structure choices and implementation

The recursive implementation of both sorting algorithms is clean and effective. Recursion elegantly captures the divide-and-conquer philosophy, allowing the problem to be broken down into smaller subproblems that are easier to solve. In quicksort, recursion handles the repeated partitioning of the array. In merge sort, recursion handles sorting two sub-lists and merging them. The use of recursion also enhances code clarity and modularity.

3. Discussion of appropriateness to the application

While recursion does have drawbacks, such as stack overflow risk with large and unbalanced inputs, in this assignment, recursion offers a clean and structured way to implement the sorting methods. The recursive design aligns with the divide-and-conquer nature of both Quicksort and Merge Sort. For quick sort, recursion is a natural fit due to its hierarchical partitioning. Each variant (basic, hybrid with insertion sort, median-of-three) leverages recursion to divide problems until base conditions are met. For natural merge sort, recursion mirrors the divide and conquer concept of it. Recursive calls simplify list splitting and merging operations without auxiliary arrays.

4. Design decisions and justifications

In the quick sort class, there is first a SortStats class that keeps track of comparisons and exchanges during quick sort. One helper function is the partition function. It compares the left pointer with the pivot value and moves the left index until a value greater than or equal to the pivot is found. It also compares the right pointer with the pivot value and moves the right index until a value less than or equal to the pivot is found. When both such values are found and the pointers have not yet crossed, they are swapped to ensure elements less than the pivot are moved

to the left side of the partition and those greater are moved to the right. This process continues until the pointers cross, at which point the partition index is returned.

There are four QuickSort variant methods in the QuickSort class. Each method uses a recursive approach, starting with the partition function to determine the pivot index. After partitioning, the method recursively calls itself to sort the left subarray (from the left index to pivotIndex - 1) and the right subarray (from pivotIndex to the right index). The first variant selects the first element of the partition as the pivot and treats partitions of size one or two as base cases. For partitions of size two, if the right pointer is greater than the left pointer and the value at the left index is greater than the value at the right index, the two elements are swapped to ensure correct ordering.

The second variant uses the same pivot selection strategy but introduces a hybrid approach: if the size of the current partition is 100 elements or fewer, the algorithm switches from Quicksort to insertion sort. This optimization leverages the fact that insertion sort is more efficient for small datasets due to its lower overhead. The third variant follows the same structure but lowers the threshold to 50 elements. The fourth variant also treats partitions of size one and two as base cases, similar to the first version. However, it differs in pivot selection—it chooses the median of the first, middle, and last elements (the median-of-three) as the pivot instead of simply using the first element. Another helper function is the insertion sort. In the insertion sort, the algorithm iterates through the array from left to right. For each element, it compares it with the elements before it and shifts larger elements one position to the right until the correct position for the current element is found. This process continues until the entire array is sorted. This is used as a fallback method when the size of a partition falls below a certain threshold, as it performs efficiently on small datasets.

There is also a natural merge sort class. In the merge sort, there is also a SortStats class that keeps track of comparisons and exchanges during quick sort. A key helper function in this implementation is the merge method, which is responsible for combining two sorted linked lists into one. The method begins by creating a dummy node that serves as a placeholder for building the merged list. It then uses two pointers, each pointing to the head of one of the input lists. At each step, it compares the values at the current nodes of the two lists. The dummy node is linked to the node with the smaller value, and the pointer in the corresponding list is advanced. This process continues until one of the lists is exhausted, after which the remaining nodes of the other list are appended to the merged list. In the sort function of natural merge sort, a base case is that the function returns the head if it is null or if there is only one node. To sort longer lists, it finds the middle using the slow and fast pointer method, then splits the list in two. Each half is recursively sorted, and the merge function is used to combine the sorted halves into one sorted list.

5. Traditional merge sort VS Natural merge sort

Implementing the natural merge sort using a linked list addresses a key limitation of traditional merge sort - its requirement for double the memory space. In the traditional merge sort, temporary arrays are needed during each merge step, which leads to space overhead, especially for large datasets. In contrast, natural merge sort with a linked implementation avoids this issue

by directly linking existing nodes during merges. It makes the memory usage more efficient. While the core logic of merging sorted sublists remains the same, the use of linked structures in Natural Merge Sort offers a practical advantage in terms of space usage. This trade-off becomes more important when the data size grows.

6. Iteration vs recursion way

In this project, I implemented both Merge Sort and Quicksort using recursion. If implemented iteratively, the algorithms would rely on explicit data structures like stacks or queues to simulate the behavior of call stack. For Quicksort, an iterative version would involve manually managing a stack to track the ranges of subarrays that still need to be sorted. For Mergesort, instead of recursively splitting the array into smaller pieces (top-down recursion), the iterative way starts with very small sorted pieces (single elements) and iteratively merge them into bigger and bigger sorted pieces.

In the recursion way, the algorithms use implicit call stacks. In recursive quicksort, after selecting a pivot, the algorithm recursively sorts the two partitions created. In recursive merge sort, the list is recursively divided into halves until base cases are reached, and then merged back together in sorted order.

7. Analysis of quick sort and merge sort

Quicksort tends to be faster for smaller to medium-sized in-memory datasets because it is low-memory usage and partitioning is efficient. It sorts the data in place without requiring significant additional storage. The four variations of quick sort make it more reliable. Quicksort's performance can degrade to $O(n^2)$ in the worst case if poor pivot choices lead to highly unbalanced partitions. Like median-of-three pivot selection and hybrid insertion sort for small partitions, quick sort could become more reliable when handling nearly sorted or randomly ordered datasets. Natural Merge Sort offered better memory scalability and consistency for larger datasets. By using linked lists instead of arrays, Natural Merge Sort avoids the doubling of memory space typically required by traditional merge sort. And this makes it more ideal for memory efficiency. While Merge Sort has a guaranteed O(n log n) time complexity regardless of input order, it usually has more pointer operations and overhead compared to the in-place nature of Quicksort.

Overall, Quicksort is generally preferred for in-memory, moderately sized datasets where speed is prioritized, while Natural Merge Sort is better suited for handling very large datasets.

8. Efficiency with respect to both time and space

Quick sort has an average case of O(nlogn) and performs very well on small to medium-sized datasets due to its in-place partitioning method and low memory overhead. But, in the worst

case, when the pivot consistently results in unbalanced partitions, it could be $O(n^2)$ in time complexity. The space complexity of Quicksort's recursive implementation is related to the depth of the recursion. In the best and average cases, the recursion tree has a depth of O(log n), resulting in O(log n) space complexity. However, in the worst case, the recursion tree can have a depth of O(n), leading to O(n) space complexity.

Natural Merge Sort has a O(n log n) time complexity for all cases, regardless of input ordering, making it more predictable in performance. When Natural Merge Sort is implemented recursively with a linked list, the space complexity mainly comes from the recursion stack. The recursion tree has a depth of O(log n), resulting in O(log n) space complexity.

There is also an insertion sort used in this lab. In the best case, when it is a sorted array, insertion sort has a time complexity of O(n) — only one comparison per element, no shifting needed. In the average case, when it is a random array, the time complexity is $O(n^2)$ — each element is compared with nearly half of the sorted part on average. In the worst case, when it is a reverse sorted array, the time complexity is $O(n^2)$ - each new element must be compared and shifted. Since insertion sort is an in-place sorting algorithm, its space complexity is always O(1). It sorts the elements in place.

In this lab, four different Quicksort variants were implemented, each introducing slight modifications to the pivot selection strategy and base case handling. In the first one, the first element of the partition is selected as the pivot, and partitions of size one or two are treated as stopping cases. This structure follows the traditional quick sort. The average time complexity is O(nlogn)**,** but it can degrade to $O(n^2)$ in the worst case if the pivot choices consistently lead to unbalanced partitions. The space complexity is O(log n) due to the depth of the recursion tree.

The second variant uses the same pivot selection but switches to insertion sort when the size of the partition is 100 or fewer elements. In the third variant, the threshold is lowered to 50 elements. In both cases, the overall average-case time complexity remains O(nlogn) because Quicksort still governs the majority of the sorting process on larger partitions. The use of insertion sort on small partitions can lead to practical performance improvements, as insertion sort performs very well on small or nearly sorted datasets with best-case time complexity of O(n) within those subarrays. The worst case is still $O(n^2)$ if the pivot selections create highly unbalanced partitions. The space complexity is O(log n) on average due to the depth of the recursion stack.

The fourth variant modifies the pivot selection by choosing the median of the left, middle, and right elements as the pivot. This median-of-three strategy significantly reduces the chances of encountering the worst case. Therefore, time complexity is O(nlogn). And space complexity is O(logn) on average due to the depth of the recursion stack. But the worst case still exists. The worst case is still $O(n^2)$ in time complexity if the pivot selections create highly unbalanced partitions.

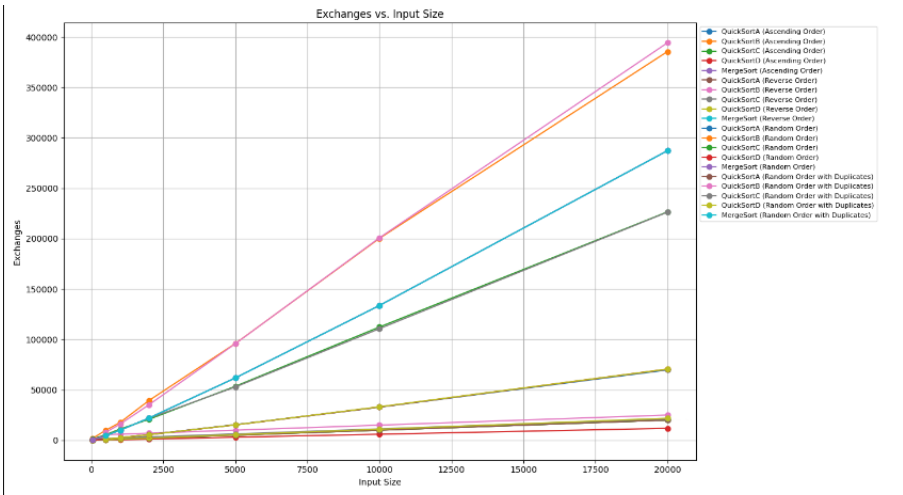
9. Analysis of comparisons and exchanges

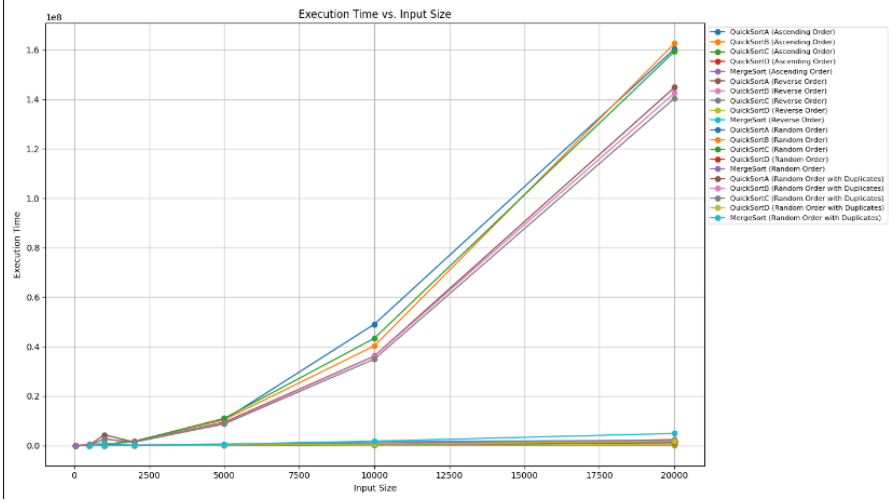
Sort	File	Comparisons	Exchanges	Execution Time	Input Type	Size
QuickSortA	asc50.dat	1321	48	6750	Ascending Order	50
QuickSortA	rev50.dat	1297	49	2542	Reverse Order	50
QuickSortA	ran50.dat	934	169	5833	Random Order	50
QuickSortB	asc50.dat	49	49	1166	Ascending Order	50
QuickSortB	rev50.dat	1225	1274	2417	Reverse Order	50
QuickSortB	ran50.dat	1951	1956	3500	Random Order	50
QuickSortC	asc50.dat	49	49	792	Ascending Order	50
QuickSortC	rev50.dat	1225	1274	2291	Reverse Order	50
QuickSortC	ran50.dat	1258	1094	3625	Random Order	50
QuickSortD	rev50.dat	391	55	2083	Reverse Order	50
QuickSortD	asc50.dat	392	31	8375	Ascending Order	50
QuickSortD	ran50.dat	858	166	4875	Random Order	50
MergeSort	ran50.dat	477	672	29750	Random Order	50
MergeSort	rev50.dat	133	286	18459	Reverse Order	50
MergeSort	asc50.dat	153	286	27833	Ascending Order	50
QuickSortA	dup500.dat	6106	1132	23791	Random Order with Duplicates	500
QuickSortA	ran500.dat	6284	1138	32000	Random Order	500
QuickSortA	rev500-1.dat	125497	499	96292	Reverse Order	500
QuickSortA	asc500.dat	125746	498	114084	Ascending Order	500
QuickSortB	dup500.dat	10031	7842	18875	Random Order with Duplicates	500
QuickSortB	ran500.dat	11655	9846	27458	Random Order	500
QuickSortB	asc500.dat	120699	499	105875	Ascending Order	500
QuickSortB	rev500-1.dat	125350	5449	92833	Reverse Order	500
QuickSortC	ran500.dat	7869	5399	24500	Random Order	500
QuickSortC	dup500.dat	7432	4943	16667	Random Order with Duplicates	500
QuickSortC	asc500.dat	124474	499	113084	Ascending Order	500
QuickSortC	rev500-1.dat	125425	1724	94959	Reverse Order	500
QuickSortD	dup500.dat	6093	1141	27750	Random Order with Duplicates	500
QuickSortD	ran500.dat	6129	1130	35083	Random Order	500
QuickSortD	asc500.dat	5264	255	470542	Ascending Order	500
QuickSortD	rev500-1.dat	5263	504	5750	Reverse Order	500
MergeSort	ran500.dat	3854	4488	113291	Random Order	500
MergeSort	rev500-1.dat	2216	4488	25625	Reverse Order	500

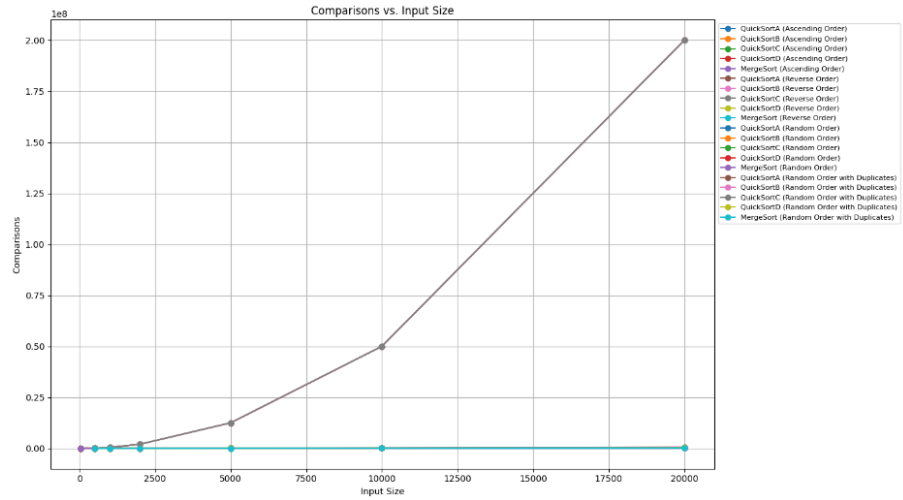
MergeSort	dup500.dat	3842	4488	39500	Random Order with Duplicates	500
MergeSort	asc500.dat	2272	4488	60333	Ascending Order	500
QuickSortA	rev1K.dat	500997	999	4320917	Reverse Order	1000
QuickSortA	asc1K.dat	501496	998	494458	Ascending Order	1000
QuickSortA	ran1K.dat	14055	2505	46709	Random Order	1000
QuickSortA	dup1K.dat	13519	2532	369791	Random Order with Duplicates	1000
QuickSortB	rev1K.dat	500850	5949	2528792	Reverse Order	1000
QuickSortB	asc1K.dat	496449	999	519375	Ascending Order	1000
QuickSortB	ran1K.dat	22859	17647	39958	Random Order	1000
QuickSortB	dup1K.dat	20759	15979	141375	Random Order with Duplicates	1000
QuickSortC	rev1K.dat	500925	2224	2697750	Reverse Order	1000
QuickSortC	asc1K.dat	500224	999	445625	Ascending Order	1000
QuickSortC	ran1K.dat	17516	11141	36916	Random Order	1000
QuickSortC	dup1K.dat	15959	10097	72125	Random Order with Duplicates	1000
QuickSortD	asc1K.dat	11532	511	11250	Ascending Order	1000
QuickSortD	rev1K.dat	11531	1010	145083	Reverse Order	1000
QuickSortD	dup1K.dat	12778	2562	62000	Random Order with Duplicates	1000
QuickSortD	ran1K.dat	13669	2486	117083	Random Order	1000
MergeSort	ran1K.dat	8705	9976	199708	Random Order	1000
MergeSort	dup1K.dat	8670	9976	196542	Random Order with Duplicates	1000
MergeSort	asc1K.dat	5044	9976	180000	Ascending Order	1000
MergeSort	rev1K.dat	4932	9976	1055625	Reverse Order	1000
QuickSortA	dup2K.dat	33400	5572	97042	Random Order with Duplicates	2000
QuickSortA	ran2K.dat	30540	5481	96792	Random Order	2000
QuickSortA	asc2K.dat	2002996	1998	1657625	Ascending Order	2000
QuickSortA	rev2K.dat	2001997	1999	1427125	Reverse Order	2000
QuickSortB	dup2K.dat	50480	35160	81125	Random Order with Duplicates	2000
QuickSortB	ran2K.dat	51459	39367	105458	Random Order	2000
QuickSortB	asc2K.dat	1997949	1999	1785541	Ascending Order	2000
QuickSortB	rev2K.dat	2001850	6949	1361292	Reverse Order	2000
QuickSortC	dup2K.dat	39174	21753	74834	Random Order with Duplicates	2000
QuickSortC	ran2K.dat	35345	20702	76292	Random Order	2000
QuickSortC	asc2K.dat	2001724	1999	1680500	Ascending Order	2000
QuickSortC	rev2K.dat	2001925	3224	1381250	Reverse Order	2000
QuickSortD	ran2K.dat	29285	5470	105916	Random Order	2000

QuickSortD	dup2K.dat	30757	5422	93000	Random Order with Duplicates	2000
QuickSortD	rev2K.dat	25067	2022	21583	Reverse Order	2000
QuickSortD	asc2K.dat	25068	1023	22625	Ascending Order	2000
MergeSort	dup2K.dat	19417	21952	148417	Random Order with Duplicates	2000
MergeSort	ran2K.dat	19436	21952	255083	Random Order	2000
MergeSort	rev2K.dat	10864	21952	84958	Reverse Order	2000
MergeSort	asc2K.dat	11088	21952	173542	Ascending Order	2000
MergeSort	rev2K.dat	10864	21952	84958	Reverse Order	2000
QuickSortA	ran5K.dat	88236	15404	271833	Random Order	5000
QuickSortA	dup5K.dat	90858	15227	267667	Random Order with Duplicates	5000
QuickSortA	rev5K.dat	12504997	4999	9433125	Reverse Order	5000
QuickSortA	asc5K.dat	12507496	4998	10426750	Ascending Order	5000
QuickSortB	ran5K.dat	137098	95939	235416	Random Order	5000
QuickSortB	dup5K.dat	138753	95776	240458	Random Order with Duplicates	5000
QuickSortB	rev5K.dat	12504850	9949	9043250	Reverse Order	5000
QuickSortB	asc5K.dat	12502449	4999	10449334	Ascending Order	5000
QuickSortC	ran5K.dat	101214	53499	221209	Random Order	5000
QuickSortC	dup5K.dat	102756	52817	221458	Random Order with Duplicates	5000
QuickSortC	rev5K.dat	12504925	6224	8714916	Reverse Order	5000
QuickSortC	asc5K.dat	12506224	4999	10982958	Ascending Order	5000
QuickSortD	dup5K.dat	82248	15237	290750	Random Order with Duplicates	5000
QuickSortD	ran5K.dat	85296	15200	282709	Random Order	5000
QuickSortD	asc5K.dat	71564	2951	53833	Ascending Order	5000
QuickSortD	rev5K.dat	71563	5450	60125	Reverse Order	5000
MergeSort	dup5K.dat	55324	61808	559333	Random Order with Duplicates	5000
MergeSort	ran5K.dat	55160	61808	530125	Random Order	5000
MergeSort	rev5K.dat	29804	61808	324792	Reverse Order	5000
MergeSort	asc5K.dat	32004	61808	218625	Ascending Order	5000
QuickSortA	rev10K.dat	50009997	9999	36209750	Reverse Order	10000
QuickSortA	asc10K.dat	50014996	9998	49057208	Ascending Order	10000
QuickSortA	ran10K.dat	188787	32800	533833	Random Order	10000
QuickSortA	dup10K.dat	185969	32816	560875	Random Order with Duplicates	10000
QuickSortB	rev10K.dat	50009850	14949	36020459	Reverse Order	10000
QuickSortB	asc10K.dat	50009949	9999	40290792	Ascending Order	10000
QuickSortB	ran10K.dat	290092	200224	604583	Random Order	10000

QuickSortB	dup10K.dat	287647	200783	505584	Random Order with Duplicates	10000
QuickSortC	rev10K.dat	50009925	11224	34936875	Reverse Order	10000
QuickSortC	asc10K.dat	50013724	9999	43382083	Ascending Order	10000
QuickSortC	dup10K.dat	211592	110637	489250	Random Order with Duplicates	10000
QuickSortC	ran10K.dat	215552	112261	466958	Random Order	10000
QuickSortD	rev10K.dat	153131	10902	193375	Reverse Order	10000
QuickSortD	asc10K.dat	153132	5903	110000	Ascending Order	10000
QuickSortD	ran10K.dat	171531	32950	595916	Random Order	10000
QuickSortD	dup10K.dat	171629	33152	585750	Random Order with Duplicates	10000
MergeSort	asc10K.dat	69008	133616	623458	Ascending Order	10000
MergeSort	rev10K.dat	64608	133616	1590208	Reverse Order	10000
MergeSort	dup10K.dat	120413	133616	1781917	Random Order with Duplicates	10000
MergeSort	ran10K.dat	120483	133616	1029875	Random Order	10000
QuickSortA	ran20K.dat	418467	69748	1200375	Random Order	20000
QuickSortA	dup20K.dat	427441	70292	1183583	Random Order with Duplicates	20000
QuickSortA	asc20K.dat	200029996	19998	160373750	Ascending Order	20000
QuickSortA	rev20K.dat	200019997	19999	144858792	Reverse Order	20000
QuickSortB	ran20K.dat	604735	385568	2384417	Random Order	20000
QuickSortB	dup20K.dat	619768	394291	1721625	Random Order with Duplicates	20000
QuickSortB	asc20K.dat	200024949	19999	162655000	Ascending Order	20000
QuickSortB	rev20K.dat	200019850	24949	142586666	Reverse Order	20000
QuickSortC	dup20K.dat	479419	226237	1674833	Random Order with Duplicates	20000
QuickSortC	ran20K.dat	470325	226538	1163208	Random Order	20000
QuickSortC	asc20K.dat	200028724	19999	159385125	Ascending Order	20000
QuickSortC	rev20K.dat	200019925	21224	140460709	Reverse Order	20000
QuickSortD	ran20K.dat	366857	70533	1283916	Random Order	20000
QuickSortD	dup20K.dat	376646	70572	1697584	Random Order with Duplicates	20000
QuickSortD	asc20K.dat	326268	11807	243250	Ascending Order	20000
QuickSortD	rev20K.dat	326267	21806	321667	Reverse Order	20000
MergeSort	ran20K.dat	260994	287232	2320459	Random Order	20000
MergeSort	dup20K.dat	260891	287232	4880333	Random Order with Duplicates	20000
MergeSort	rev20K.dat	139216	287232	2019292	Reverse Order	20000
MergeSort	asc20K.dat	148016	287232	1151458	Ascending Order	20000







In this study, we implemented and compared four variants of Quicksort and one Natural Merge Sort algorithm. We recorded the number of exchanges and comparisons each sort performed across various datasets and sizes to understand their efficiency.

Quicksort Variant A: (First element as pivot, stop for size 1 or 2)

- Comparisons: High. Always choosing the first element can lead to poor pivot selections, especially for nearly sorted or reversed data. It suffers horribly for sorted and reversed data — super high comparisons, which matches what we expect when Quicksort degrades to $O(n^2)$**:** tons of bad comparisons but relatively few swaps.
- Exchanges: High, especially if the input is nearly sorted or in reverse order. The partitioning is inefficient, causing unnecessary exchanges.

Quicksort Variant B: (First element as pivot, insertion sort when size ≤ 100)

- Comparisons: Reduced compared to Variant A because insertion sort is efficient for small partitions.
- Exchanges: Also reduced because insertion sort minimizes movement for almost sorted subarrays. It performs better especially when the dataset is large.

Quicksort Variant C: (First element as pivot, insertion sort when size ≤ 50)

- Comparisons: Slightly more than Variant B because insertion sort is only used for smaller partitions, so Quicksort still does more work on partitions between 50 and 100.
- Exchanges: Also reduced because insertion sort minimizes movement for almost sorted subarrays. It performs better especially when the dataset is large.

Quicksort Variant D: (Median-of-three pivot, stop for size 1 or 2)

- Comparisons: Much lower than Variants A, B, and C. The median-of-three technique significantly improves partitioning quality.
- Exchanges: Fewer exchanges because partitions are more balanced.

Natural Merge Sort (Linked List Based)

- Comparisons: Low to moderate. Natural runs reduce the number of merges needed. For already partially ordered data, the comparisons drop significantly.
- Exchanges: Low to moderate. It has a stable O(nlogn) time complexity for all cases, which makes it very efficient.

The order of the data had the greatest impact on sorting efficiency. Quicksort is particularly vulnerable to bad pivot choices in sorted or reverse-sorted data, causing it to degrade to O(n^2). Natural Merge Sort, on the other hand, benefits heavily from ordered data. Pivot selection (e.g., median-of-three) is another factor that dramatically improved Quicksort's performance, making it less sensitive to input order. Partition size thresholds helped, but not as much as the former factors. File size is also a big factor that affects the efficiency and run time, as we can see from the table.

10. What I learned

From this lab, I learned how to use recursion to implement both Merge Sort and Quicksort. Thinking through these problems helped me develop a much better understanding of sorting algorithms and how they work at a deeper level. Initially, I had trouble implementing the Quicksort algorithm correctly. I thought my approach was right, but when I compared my results to Java's built-in sort, I realized there were mistakes. Through careful debugging and gradual improvements, I was able to fix the issues. Working with the different Quicksort variations was also a great learning experience. I learned that in recursive algorithms, the base case doesn't always have to simply return; you can introduce a threshold and switch to another sorting method or just general another method, such as insertion sort, which was very interesting and new to me.

Working on Natural Merge Sort also helped me realize how useful linked lists can be. In my previous experience, especially using Python, I often relied on arrays because they felt more intuitive and straightforward. However, through this project, I saw firsthand that linked lists can actually save a significant amount of space due to their linked node structure, especially when merging large datasets. This lab showed me how important it is to choose the right data structure based on the problem, as it can make a real difference not only in space complexity but also in time complexity. Seeing the impact of data structure choices during an actual implementation made the concepts much more concrete and meaningful.

Since this was our last lab, I can say that I really enjoyed the lab assignments throughout the course. They helped me build a stronger understanding of different problems and gave me valuable hands-on experience. The structure and setup of the labs were also very helpful and contributed to a good learning experience.

11. What I might do differently next time

Since I implemented the sorting algorithms using the recursive approach this time, I would like to try the iterative approach in the future if given the opportunity. Exploring both recursive and iterative implementations would give me a deeper understanding of how different sorting algorithms work and how control flow and memory usage differ between the two. It would also help reinforce the concepts we've learned and allow me to think more critically about algorithm design and efficiency from multiple perspectives.

12. Enhancements

One enhancement I implemented is a JUnit test class named SortsTest, which I used to verify the correctness of my sorting algorithms. It includes tests on a small randomly generated array that contains a few duplicates. This helps me to ensure that both Quicksort and Natural Merge Sort handle typical input scenarios accurately.

Another enhancement I implemented is error checking to ensure the sorting algorithms work correctly, especially on larger input datasets. To validate the output, I compared my sorted arrays

against Java's built-in Arrays.sort() result. If a mismatch is detected, the program prints a warning message along with the file name and the specific sorting method that failed, helping me quickly identify and debug issues. In the args input, the program checks whether exactly one command-line argument is provided; if not, it prompts the user with the correct usage format. It also verifies that the specified folder contains .dat files, and if none are found, it prints a message indicating that no valid input files were detected. As always, I include standard I/O exception handling to catch any errors that may occur while reading from or writing to files.

One enhancement I added is timing the execution of each sorting algorithm. The time taken for each sort is measured in nanoseconds and recorded in the output file. I structured the output to include the original array, the sorted array, the file name, the name of the sorting method (e.g., QuickSortD), the number of comparisons and exchanges, and the execution time in nanoseconds. For example, an entry in the output file might look like: *Sort name: QuickSortD, File name: rev500-1.dat, Comparisons: ..., Exchanges: ..., Execution Time: .. ns*. To further analyze and visualize the results, I used Python and the matplotlib package to generate tables and comparison plots since matplotlib package is very convenient. I also incorporated the timing data into the earlier performance analysis section to support a more comprehensive evaluation of algorithm efficiency.

One enhancement I made was creating a random number generator to produce input files of four different sizes: 50, 1000, 2000, 5000, and 10,000 integers. For each size, I generated three versions: one with randomly ordered data, one with the integers in reverse order, and one with the integers in ascending order. I created a shuffle function to randomize the input files.

One enhancement I attempted was adding larger datasets — specifically, files with 100,000 integers in all three orderings (random, ascending, and reverse). However, when I tried running the program on these larger datasets, I encountered a stack overflow error, especially when executing the Quicksort A variant. I realized that Quicksort A, which selects the first element as the pivot without any optimizations, can easily lead to highly unbalanced partitions, particularly on sorted or reverse-sorted inputs. This causes the recursion depth to grow too large, exceeding the maximum stack size and resulting in a stack overflow. Through this experience, I learned how critical pivot selection strategies and recursion management are for scaling sorting algorithms to larger inputs, and why enhancements like median-of-three pivot selection are important to maintain both performance and program stability. For simplicity, I did not include all my tryouts in the codes.

One enhancement I implemented in this lab was introducing datasets with varying levels of duplication to evaluate how duplicates affect the performance of different sorting algorithms. Through experimentation, I discovered that the presence of duplicates has a significant impact, especially on Quicksort variants. Specifically, when the duplicate rate exceeded approximately 25%, the recursive Quicksort implementations became more prone to stack overflow errors, particularly those using the first element as the pivot. This is because duplicate-heavy datasets lead to highly unbalanced partitions — many elements equal to the pivot are repeatedly pushed into the same side of the recursive call tree. As a result, the recursion depth increases rapidly, consuming stack space inefficiently and causing stack overflows for relatively moderate input sizes. This issue was less pronounced in the median-of-three Quicksort, which more often

produces balanced partitions even in the presence of duplicates. Natural Merge Sort, on the other hand, was much more resilient to duplication.

One enhancement I implemented was adding a configurable cutoff value for QuickSort, allowing the algorithm to switch to insertion sort when partitions fall below a certain size. To keep the main function simple and clean, I chose not to include all of my experimental tryouts there. However, during my experiments, I tested a variety of cutoff values and observed some interesting patterns. In general, I found that using a larger cutoff threshold—such as 20, 50, 100, 150, 200, or even higher—could improve performance for moderately sized datasets. This is because insertion sort, despite its $O(n^2)$ worst-case complexity, is very efficient on small arrays due to its minimal overhead and cache-friendly behavior. A higher cutoff reduces the recursive depth and overhead of QuickSort, allowing insertion sort to handle small partitions more effectively. However, there was a trade-off. If the cutoff was set too high, QuickSort ended up delegating too much work to insertion sort, causing performance to degrade on larger partitions. Therefore, finding an optimal cutoff value became a balance between minimizing recursion and maximizing sorting efficiency. Overall, introducing a generalized cutoff parameter made the QuickSort implementation more flexible and provided valuable insights into how hybrid sorting algorithms can be tuned for better real-world performance.