Advanced Data Structures- Lab 1
External Sorting: Quick Sort vs Merge Sort Adaptations

200682T – Vihidun D.P.T.

1. Introduction

This report presents the implementation of external sorting algorithms using Quick Sort and Merge Sort, designed for the specific constraints of **16MB of internal memory**. The objective is to sort a **256MB file** containing unsorted integers ranging from 1 to 1 million. Given the memory constraints, external sorting techniques were employed to divide the data into manageable chunks for sorting and subsequent merging.

The algorithms were tested on three unsorted files, and their performance was analyzed in terms of runtime and memory usage. The comparison between Quick Sort and Merge Sort focuses on how each algorithm performs under the given constraints, particularly concerning buffer management and the efficiency of disk I/O operations.

2. Methodology

2.1 Quick Sort Implementation

Interval Heap Adaptation:

Quick Sort was adapted using an interval heap to find the minimum and maximum values in each partitioned chunk. The interval heap was crucial for partitioning data efficiently before performing the sorting operation. The internal Quick Sort algorithm was then applied within the bounds of the allocated buffers.

Buffer Allocation:

The 16MB memory was divided into different buffers, with special emphasis on the **middle buffer**, as it played a key role in partitioning the data for sorting:

- **Middle Buffer**: Largest allocation to optimize performance by reducing the number of recursion calls.
- **Small/Large Buffers**: Used for storing boundary elements during partitioning.

Challenges and Optimizations:

The recursive nature of Quick Sort caused high memory consumption. Therefore, manual garbage collection was introduced to prevent memory overflow. Optimizing the middle buffer size also helped in reducing the recursion depth, which in turn improved performance.

2.2 Merge Sort Implementation

K-Way Merging and Tournament Trees:

Merge Sort was adapted using tournament trees to manage the merging of runs efficiently. A **K-way merge algorithm** was used to merge sorted runs stored on disk. Different values of K (number of runs merged simultaneously) were tested, and a K-value of 10 provided optimal performance.

Buffer Allocation:

Memory was allocated to store sorted runs before merging, ensuring that the total memory usage remained within the 16MB limit.

Challenges and Optimizations:

Merge Sort's performance heavily relied on efficient disk I/O management. The use of tournament trees minimized the number of comparisons during merging, significantly reducing the overall runtime.

3. Performance Results

3.1 Quick Sort Performance

Quick Sort was tested on three different unsorted files, with the following results:

File	Runtime (seconds)	Peak Memory Usage (MB)
File 1 (256MB)	9982.27	14.01
File 2 (256MB)	9642.02	13.25
File 3 (256MB)	9764.14	13.56

Quick Sort exhibited longer runtimes due to the overhead of recursive partitioning and the high cost of disk I/O operations. Despite optimizations such as manual garbage collection and buffer size adjustments, Quick Sort faced challenges in reducing its overall runtime.

3.2 Merge Sort Performance

Merge Sort significantly outperformed Quick Sort, as shown by the results:

File	Runtime (seconds)	Peak Memory Usage (MB)
File 1 (256MB)	1381.31	12.91
File 2 (256MB)	1417.29	13.28
File 3 (256MB)	1397.08	13.09

Merge Sort's efficiency can be attributed to the reduced number of disk accesses and the K-way merging process. By merging multiple runs at once, Merge Sort minimized the need for repeated reads and writes, leading to a significant reduction in runtime.

4. Analysis and Discussion

4.1 Runtime Comparison

Merge Sort outperformed Quick Sort by a factor of five to six across all test cases. The primary reason for this was the reduction in disk I/O operations. Quick Sort, while effective in managing memory through partitioning, was hindered by the need for frequent recursive calls, leading to higher runtimes. On the other hand, Merge Sort efficiently managed the merging of sorted runs using tournament trees, which reduced the number of passes over the data.

4.2 Memory Usage and Disk I/O

Both algorithms maintained memory usage within the 16MB constraint, but Merge Sort was more efficient in handling disk I/O operations. The K-way merge algorithm, combined with tournament trees, reduced the need for excessive read/write operations, allowing Merge Sort to complete the sorting task with fewer disk accesses.

5. Conclusion

The results of this experiment demonstrate that **Merge Sort** is the superior algorithm for external sorting under constrained memory conditions. Its ability to minimize disk I/O operations through efficient run merging led to significantly faster runtimes compared to Quick Sort. Despite efforts to optimize Quick Sort, its performance was limited by the high cost of recursive partitioning and disk accesses.

Overall, Merge Sort is recommended for large-scale external sorting tasks where memory is limited, and disk I/O operations are a significant concern. Future work could explore further optimizations to buffer management or the use of parallel processing to improve the performance of both algorithms.

5. Code

The code for this lab is available at the following github repository.
https://github.com/tharoosha/ADS_Assigment_External_Sorting