CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Merge Sort

Department of Computer Science
University of Maryland, College Park

Introduction to Merge Sort

- **Merge Sort** is a classic **divide-and-conquer** algorithm.
- Developed by **John von Neumann** in 1945.
- It's a **recursive algorithm** that:
  1. Divides the array
  2. Sorts each half
  3. Merges the halves
- Always runs in **O(n log n)** time.
- It is a **stable sort**, meaning equal elements keep their original order.

Divide and Conquer – The Three Steps

- **1. Divide**
  - Split the array into two halves.
- **2. Conquer**
  - Recursively sort each half.
- **3. Combine**
  - Merge the two sorted halves into one.

Example

```
Original: [8, 4, 5, 2]
→ Divide: [8, 4] and [5, 2]
→ Sort:   [4, 8] and [2, 5]
→ Merge:  [2, 4, 5, 8]
```

Easy Split, Hard Merge

- **Splitting** is easy:
  - Just compute the midpoint: mid = (left + right) / 2
  - No comparisons are done.
  - Just index math.
- **Merging** is the work-intensive part:
  - Requires combining two sorted arrays.
  - Takes linear time, proportional to the total number of elements being merged.
  - Extra space is needed to hold merged results.

Note:

- Merge Sort = **Easy Split, Hard Merge**
- QuickSort = **Hard Split, Easy Merge**

Example of Merge Sort

- Let's sort this array: [8, 3, 1, 7, 0, 10, 2]

Divide Phase

Split 1: [8, 3, 1, 7] and [0, 10, 2]

Split 2: [8, 3] [1, 7] [0, 10] [2]

Split 3: [8] [3] [1] [7] [0] [10] [2]

Conquer and Merge Phase

Merge [8] and [3] → [3, 8]

Merge [1] and [7] → [1, 7]

Merge [0] and [10] → [0, 10]

[2] remains alone for now

Merge [3, 8] and [1, 7] → [1, 3, 7, 8]

Merge [0, 10] and [2] → [0, 2, 10]

Final Merge → [1, 3, 7, 8] and [0, 2, 10] → [0, 1, 2, 3, 7, 8, 10]

Why O(n log n)?

- Each level **splits the array in half** → takes **$\log_2(n)$** levels.
- At **each level**, you merge all elements → takes **O(n)** work.

For n = 8:

- $\log_2(8) = 3$ levels
- Work per level: O(8)
- Total work: **3 × 8 = 24 steps** = **O(n log n)**

Recursion Tree (Simplified):

Level 0: [8 elements]

Level 1: [4] [4]

Level 2: [2] [2] [2] [2]

Level 3: [1][1] [1][1] [1][1] [1][1] → base case

Each level processes all elements once → total work = **O(n log n)**

The proof idea is the same as the 50-50 split for Quick Sort, with the key difference being that in Merge Sort, we don't need to assume a 50-50 split — we are guaranteed one.

Why O(n) Space?

Merging Uses Extra Memory:

- To merge two sorted halves, we need a **temporary array**.
- It holds up to n elements.

So:

- Merge Sort uses **O(n)** space for:
  - Temporary arrays during merge
  - Recursion stack (O(log n))
- Note 1: In-place merge sort is possible but much harder and slower in practice.
- Note 2: **Linked list merge sort** can be done with O(1) space since we just rearrange pointers.

Merge Step – Visual Walkthrough

- Let's merge [3, 8] and [1, 7] into a sorted array.

```
[3, 8]         [1, 7]
 ^              ^
Compare 3 vs 1 → 1 goes into merged array
[3, 8]         [1, 7]
 ^                 ^
Compare 3 vs 7 → 3 goes in
[3, 8]         [1, 7]
    ^              ^
Compare 8 vs 7 → 7 goes in
[3, 8]         [1, 7]
    ^
Only 8 remains → append
[1, 3, 7, 8]
```

Merging takes **O(n)** time and requires **temporary space**.

Merge Sort Recursive Structure

```
// Pseudocode
mergeSort(arr):
    if size <= 1:
        return
    split into left and right
    mergeSort(left)
    mergeSort(right)
    merge(left, right)
```

Merge Logic:

- Compare smallest elements of left and right.
- Copy the smaller one into result array.
- Repeat until all elements are merged.

See: MergeSort Example. If time allows, make it generic so that it works with other types, such as Strings.

Merge Sort on Linked Lists

- **Why Merge Sort Works Well on Linked Lists**
- **No index access needed**: Unlike arrays, linked lists can't support random access efficiently. Merge Sort only needs *sequential traversal*.
- **No swapping**: Elements are rearranged by *changing pointers*, not values.
- **Efficient splitting**: Use the "slow/fast pointer" technique to find the middle node and divide the list into two halves.
- **Merging**: Two sorted linked lists can be merged in O(n) time by walking through them and relinking nodes.
- **Space-efficient**: No need for extra arrays — merging is done via pointers.
- **Time Complexity**: O(n log n) (due to recursive halving and merging)
- **Space Complexity**: O(log n) recursive stack; **no extra memory for merging**.
- **Note**: Merge Sort is preferred over QuickSort for linked lists because QuickSort requires random access and complex node rearrangements.

Merge Sort for External Data

- **External Merge Sort: Sorting Data Too Big for RAM**
- **Problem**: Datasets may be too large to fit in memory (e.g., multi-GB/terabyte log files).
- **Goal**: Sort data using **limited RAM** with minimal disk reads/writes.
- **Phase 1 – Create Sorted Runs**
  - Read manageable-sized chunks of data into memory.
  - Sort each chunk with in-memory Merge Sort or QuickSort.
  - Write each sorted chunk ("run") back to disk.
- **Phase 2 – Multi-Way Merge**
  - Open multiple sorted runs.
  - Merge them together in a **sequential** fashion.
  - Use a priority queue/min-heap to keep track of the smallest elements across files.
  - Write the merged result back to disk.
- **Time Complexity**: O(n log n)
  Disk Efficiency: Sequential I/O is used — avoids random disk seeks
  Applications: Large-scale log processing, database systems, MapReduce/Hadoop jobs.
- **Merge Sort is the industry-standard approach for external sorting** due to its sequential disk access and scalability.