Discussion 7: Advanced Sorting Algorithms

Discussion Topic:

The merge sort is a recursive algorithm that can be used to sort an array by first sorting smaller versions of an initial list of objects. In this discussion, provide a simple of code sample of Merge sort and describe the benefits of using the merge sort. What are the key steps that must be taken to ensure an efficient sorting of objects?
In your answer, specifically think of and give a real-life scenario where

- Merge sort may be used
- Merge sort may be a better fit than Quick and Radix sorts

My Post:

Hello Class,

In computer science, Merge-Sort is classified as a "divide-and-conquer" algorithm with a time complexity of *O(n log n)* that typically uses recursion to sort data. It is best suited for large datasets, especially when stability is important, and for scenarios where the data cannot fit into memory all at once, such as external sorting or sorting linked lists.

A divide-and-conquer algorithm can be defined as having three steps:

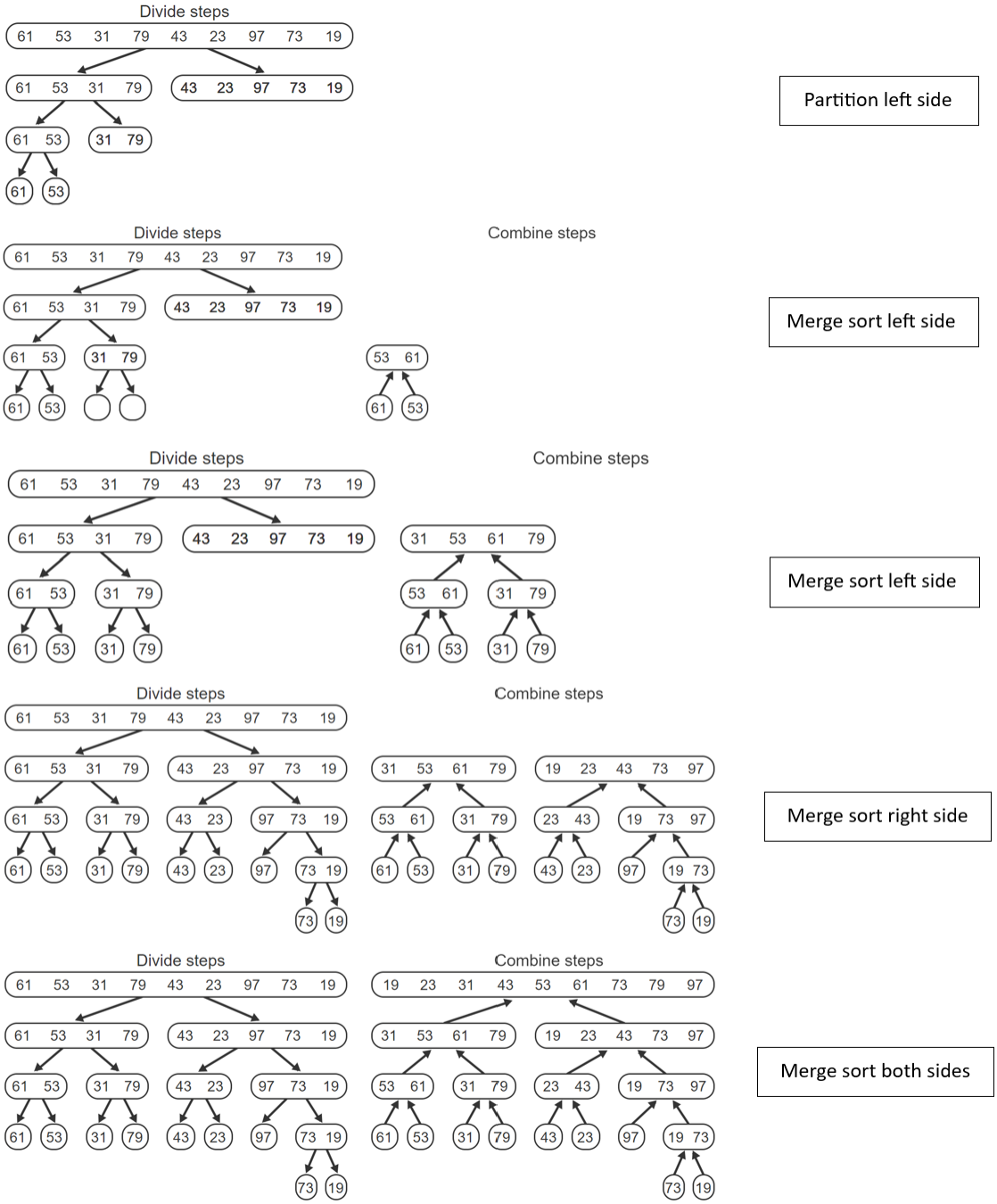
1. ***Divide:*** If the input size is smaller than a certain threshold (say, one or two elements), solve the problem directly using a straightforward method and return the solution so obtained. Otherwise, divide the input data into two or more disjoint subsets.
2. ***Conquer:*** Recursively solve the subproblems associated with the subsets.
3. ***Combine:*** Take the solutions to the subproblems and merge them into a solution to the original problem.

(Goodrich et al., 2023, section 12.1 Merge Sort)

Merge sort applies the divide-and-conquer approach by sequentially dividing the data into two roughly equal partitions and recursively sorting the left partition into smaller sorted sub-partitions until it reaches a single element. It then combines and sorts all the sub-partitions to construct the final sorted left partition. The same process is repeated for the unsorted right partition. Finally, the left and right partitions are combined and sorted to produce the final sorted output. See Figure 1 for the merge sort divide-and-combine steps.

Figure 1

Merge-Sorts



Note: the figure does not show all the step details, but enough to understand the process. From "). Chapter 12: Algorithms: Sorting and Selection. *Data Structures and Algorithms*" by Goodrich et Al. (2023). Modify.

Merge-Sort Advantages

Merge-Sort has several advantages. First, it is a stable sort algorithm meaning that it maintains the relative order of unsorted data order of entry. For example, two data elements with the same value will retain their original order of entry. This is essential for applications where the order of entry must be kept, such as when sorting a student's course grades by letter grade, where a student may have multiple A's or B's in the same class.

Additionally, the Merge-Sort algorithm is particularly well-suited for linked lists due to its sequential access patterns (Khandelwal, 2023). Linked lists do not support indexed access like arrays do, which allows data elements to be accessed randomly. In a linked list, to access an element at a specific index, the list needs to be traversed usually from the head node, thus the data cannot be randomly accessed. However, with Merge-Sort, the list is split into two halves and recursively sorted without needing to index the data. The data is accessed sequentially, which naturally aligns with the access pattern of a linked list, making Merge-Sort well-suited for sorting data stored in linked lists.

Another advantage of the algorithm is that it guarantees *O(n log n)* time complexity, making it both reliable and efficient for sorting large datasets. Its time complexity is reliable because it remains consistent in all case scenarios.:

- **Worst Case: *O(n log n)* –** This occurs when the array is repeatedly divided into halves until individual elements are reached, and then the merging process takes place.
- **Average Case: *O(n log n)*** – Similar to the worst case, the algorithm consistently divides the array into halves and then merges them.
- **Best Case: *O(n log n)*** – Even when the array is partially sorted, merge sort divides it into halves and merges them, resulting in the same time complexity as the worst and average cases.

(Khandelwal, 2023, p.1)

Its reliable time complexity is a better choice than Quick-Sort for environments where predictable and stable time complexity is essential. Although Quick-Sort's time complexity is also *O(n log n)*, it can degrade to $O(n^2)$ in environments where the data may be partially sorted, making Marge-Sort a better choice for such situations.

Additionally, its space complexity is *O(n)* due to the additional space required for the temporary arrays used in the merging process; However, it is stable and predictable, making the algorithm ideal for sorting large datasets where resource availability needs to be predictable.

When compared to the Radix-Sort algorithm, although its time complexity is *O(n + k)* which is better than the Merge-Sort's time complexity, Merge-Sort is better suited for situations where the data is diverse and random. This is because radix sort is mostly useful for types of data, such as integers or fixed-length strings. In contrast, Merge-Sort works well with all types of data, and it is not limited by the need for a fixed or limited range of values. Whereas Radix-Sorts' performance is dependent on the range of input data and can degrade if the range is large

Furthermore, its recursive and partition nature (sorting left or right) can be implemented concurrently or

in parallel. This greatly improves the efficiency of the Merge-Sort algorithm when handling large-scale data sets in modern computing systems supporting parallel processing and multi-threading.

Merge-Sort Disadvantages

On the other hand, some of the disadvantages of Merge-Sort include the extra overhead caused by handling recursion calls and the potential risk of a stack overflow in cases of deep recursion.
The algorithms can be lower at sorting small data set datasets compared to simpler algorithms like Insertion-Sort.
Additionally, it is more complex to implement and has more space complexity than simpler algorithms such as Insertion-Sort or Selection-Sort, which require less memory and may be less complex to implement in some applications.

Java Code Examples

As pointed out earlier, Merge-Sort recursive partition sorting can be implemented in parallel or concurrently. The first code snippet is a classic Merge-Sort implementation, the second example is the implementation of a Merge-Sort using parallelism.

Basic Merge-Sort in Java:

```
import java.util.Arrays;
import java.util.Comparator;

public class MergeSort {

	public static <T> void mergeSort(T[] array, Comparator<? super T> comp) {
		int n = array.length;

		// --- Base case ---
		// If the array has 1 or 0 elements, it is already sorted, so return.
		if (n < 2) {
			return;
		}

		// --- Divide step ---
		// Find the midpoint to divide the array into two halves
		int mid = n / 2;

		// Create two subarrays: one from the left half and one from the right half
		// left subarray from index 0 to mid-1
		T[] left = Arrays.copyOfRange(array, 0, mid);
		// right subarray from index mid to n-1
		T[] right = Arrays.copyOfRange(array, mid, n);
		// --- Conquer step ---
		// Recursive call - sort the left (first) and right halves of the array.
		mergeSort(left, comp); // sort the left half
		mergeSort(right, comp); // sort the right half

		// --- Combine step ---
		// After both halves are sorted, merge them back into a single sorted array.
		merge(left, right, array, comp); // merge sorted halves into original array
	}

	// Method to merge two sorted subarrays (left and right) into the original
```

```
	// result array
	private static <T> void merge(T[] left, T[] right, T[] result, Comparator<? super T> comp)
{
		 // i, j track position in left and right arrays; k tracks result
		 int i = 0, j = 0, k = 0;

		// --- Merging process ---
		// Compare elements from the left and right arrays and place the smaller one
		// into result.
		while (i < left.length && j < right.length) {
			// Compare elements and merge them in sorted order
			if (comp.compare(left[i], right[j]) <= 0) {
				// copy from left array and move the index i
				result[k++] = left[i++];
			} else {
				// copy from right array and move the index j
				result[k++] = right[j++];
			}
		}
		// --- Copy remaining elements ---
		// If there are any remaining elements in the left array, copy them into result
		while (i < left.length) {
			result[k++] = left[i++];
		}
		// If there are any remaining elements in the right array, copy them into
		// result
		while (j < right.length) {
			result[k++] = right[j++];
		}
	}

	public static void main(String[] args) {
		Integer[] array = { 3, 5, 1, 6, 4, 7, 2 };
		mergeSort(array, Comparator.naturalOrder());
		System.out.println(Arrays.toString(array));
	}
}
```

Parallel Merge-Sort in Java:

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.concurrent.RecursiveAction;
import java.util.concurrent.ForkJoinPool;

// ParallelMergeSort class
class ParallelMergeSort<T> extends RecursiveAction {

    private final T[] array;
    private final Comparator<? super T> comp;

    // Constructor to initialize the array and comparator
    public ParallelMergeSort(T[] array, Comparator<? super T> comp) {
        this.array = array;
        this.comp = comp;
    }
```

```
    // The compute method defines the parallel sorting process
    @Override
    protected void compute() {
        int n = array.length;

        // --- Base case ---
        // If the array has 1 or 0 elements, it is already sorted, so return
        if (n < 2) {
            return;
        }

        // --- Divide step ---
        // Divide the array into two halves: left (first half) and right (second half)
        int mid = n / 2;

        // Create two subarrays: left subarray from index 0 to mid-1
        T[] left = Arrays.copyOfRange(array, 0, mid);

        // Right subarray from index mid to n-1
        T[] right = Arrays.copyOfRange(array, mid, n);

        // --- Conquer step ---
        // Create two tasks to sort the left and right halves in parallel
        ParallelMergeSort<T> leftTask = new ParallelMergeSort<>(left, comp);
        ParallelMergeSort<T> rightTask = new ParallelMergeSort<>(right, comp);

        // Invoke the tasks concurrently, allowing them to run in parallel
        invokeAll(leftTask, rightTask);

        // --- Combine step ---
        // After both halves are sorted, merge them back into the original array
        merge(left, right, array, comp);
    }

    // Method to merge two sorted subarrays (left and right) into the original result array
    private static <T> void merge(T[] left, T[] right, T[] result, Comparator<? super T> comp) {
        int i = 0, j = 0, k = 0; // i, j track position in left and right arrays; k tracks result

        // --- Merging process ---
        // Compare elements from the left and right arrays and place the smaller one into result
        while (i < left.length && j < right.length) {
            if (comp.compare(left[i], right[j]) <= 0) {
                result[k++] = left[i++]; // Take element from left array and move the index i
            } else {
                result[k++] = right[j++]; // Take element from right array and move the index j
            }
        }

        // --- Copy remaining elements ---
        // If there are any remaining elements in the left array, copy them into result
        while (i < left.length) {
            result[k++] = left[i++];
        }

        // If there are any remaining elements in the right array, copy them into result
        while (j < right.length) {
```

```
            result[k++] = right[j++];
        }
    }

    // Initializes parallel merge sort using ForkJoinPool
    public static <T> void parallelMergeSort(T[] array, Comparator<? super T> comp) {
        // Create a ForkJoinPool for parallel execution
        ForkJoinPool pool = new ForkJoinPool();

        // Start the parallel sorting task by invoking the main parallelMergeSort task
        pool.invoke(new ParallelMergeSort<>(array, comp));
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Integer[] array = {3, 5, 1, 6, 4, 7, 2};
        System.out.println("Unsorted array: " + Arrays.toString(array));
        ParallelMergeSort.parallelMergeSort(array, Comparator.naturalOrder());
        System.out.println("Sorted array: " + Arrays.toString(array));
    }
}
```

Real Life Use Example

Merge-Sort is commonly implemented in situations where large datasets need to be fetched and stored on disk, such as in data centers, this process is often referred to as external sorting. An online retailer is a good example of it such as Amazon or eBay, where millions of customer orders need to be sorted orders based on timestamps. Since the dataset is too large to fit in memory, Merge-Sort is well suited for this task. The data can be loaded in chunks, sorted in parallel in memory preserving the stability of the data, and merged on the disk.

To summarize, Merge-Sort is reliable, stable, and able to handle large datasets, its recursive partition can be implemented in parallel making specialty well suited for large-scale online store sorting applications. However, its extra space and recursion overhead should be considered when implementing the algorithm, especially in environments where resources might be limited.

-Alex

References:

Goodrich T, M., Tamassia, R., & Goldwasser H. M. (2023, June). Chapter 12: Algorithms: sorting and selection. *Data structures and algorithms*. zyBook ISBN: 979-8-203-40813-6.

Khandelwal, V. (2023, October 25). *What is merge sort algorithm: How does it work, and more*. SimpliLearn. https://www.simplilearn.com/tutorials/data-structure-tutorial/merge-sort-algorithm