DATA STRUCTURE ALGORITHMS AND APPLICATIONS (CT- 159)

Lecture – 5
Merge Sort and Quick Sort
(Divide and Conquer Algorithms)

Instructor: Engr. Nasr Kamal

Department of Computer Science and Information Technology

What is Divide and Conquer

Divide: divide the input array into smaller sub-arrays.

Conquer:k solve the problem for these sub-arrays (either by recursive sorting in the case of merge sort or by partitioning in the case of quick sort).

Combine: The solutions of the sub-problems are combined to form the final sorted array.

Two of the most important and efficient comparison-based sorting algorithms are:

- **Merge Sort**
- **Quick Sort**

Both have an average time complexity of O(nlogn), but they work differently and have distinct performance characteristics

Merge Sort

- Merge Sort is a popular and efficient sorting algorithm that follows the Divide and Conquer strategy.
- It works by breaking down a large problem (sorting an array) into smaller, easier-to-solve sub-problems and then combining the solutions.

Step by Step Process

1- Divide: Split the array into halves

- The first thing Merge Sort does is split the array into two smaller halves.
- It keeps dividing the array until each part has only one element.
- A single element is always sorted by itself, so these small pieces are our starting point.

Example

- Suppose we have an array: [38, 27, 43, 3, 9, 82, 10].
- We first split it into two halves:
- Left half: [38, 27, 43]
- Right half: [3, 9, 82, 10]
- We keep splitting these halves until we get arrays of just one element:
- [38], [27], [43], [3], [9], [82], [10]

2- Conquer: Sort the individual pieces

- Once we have split the array into individual elements, we can now start combining them back together. But, as we combine them, we also sort them. This is where the magic of merge sort happens.
- For each pair of elements, we merge them by comparing and placing them in order.

Example

- We start by comparing the first two elements: [38] and [27].
- Since 27 is smaller, we merge them as [27, 38].
- Now, compare and merge [27, 38] with [43]. Since 43 is larger than 38, the merged array becomes [27, 38, 43].
- Do the same on the right half:
- Compare [3] and [9] to get [3, 9]. Then compare [82] and [10] to get [10, 82].
- Next, merge [3, 9] and [10, 82]. Compare the smallest values from each array and merge them in order to get [3, 9, 10, 82].

3- Combine: Merge the sorted halves

- Once the left and right halves are sorted, we merge them back together into one sorted array. We use the same process of comparing the smallest values in each half and combining them in sorted order.
- Example:
- Now merge [27, 38, 43] with [3, 9, 10, 82].
- We compare the smallest values from both sides:
- Compare 27 and 3 → 3 is smaller, so it goes first.
- Compare 27 and 9 → 9 is smaller.
- Compare 27 and 10 → 10 is smaller.
- Compare 27 and 27 → 27 goes next.
- Continue until all elements are merged in order.

Example

- Now merge [27, 38, 43] with [3, 9, 10, 82].
- We compare the smallest values from both sides:
- Compare 27 and 3 → 3 is smaller, so it goes first.
- Compare 27 and 9 → 9 is smaller.
- Compare 27 and 10 → 10 is smaller.
- Compare 27 and 27 → 27 goes next.
- Continue until all elements are merged in order.
- Finally, the sorted array becomes:
- [3, 9, 10, 27, 38, 43, 82].

Visual Representation

Original array: [38, 27, 43, 3, 9, 82, 10]

Split the array:

[38, 27, 43] [3, 9, 82, 10]

Further split:

[38] [27, 43] [3, 9] [82, 10]

[38] [27] [43] [3] [9] [82] [10]

Start merging:

[27, 38] [27, 38, 43] [3, 9] [10, 82]

Merge the halves:

[3, 9, 10, 27, 38, 43, 82]

Time Complexity

- Best, Average, and Worst Case: Merge Sort always takes
- O(nlogn), where n is the number of elements.
- This is because each time we split the array, it takes
- O(logn), and the merging process takes O(n).

Quick Sort

- Quick Sort is one of the most popular and efficient sorting algorithms that uses the Divide and Conquer approach.
- It’s often faster in practice than other sorting algorithms like Merge Sort, although it can have a worse case under certain conditions.

Step-by-Step Process:

1- Pick a Pivot:

- The first step in Quick Sort is to pick an element from the array, which we call the pivot.
- This pivot helps in rearranging the elements so that we can divide the array into smaller parts.

- A pivot can be any element of the array: the first, last, or even a random element.
- The performance of Quick Sort can depend on how well the pivot is chosen, but for simplicity, let’s assume we are picking the first element as the pivot.

Example

- Suppose we have an array: [10, 80, 30, 90, 40, 50, 70].
- We choose the last element, 10, as the pivot

2- Partition the Array:

- The next step is to partition the array. In this step, we rearrange the array so that:
- All elements smaller than the pivot go to the left of the pivot.
- All elements greater than the pivot go to the right.
- We do this by iterating over the array and comparing each element with the pivot.

Example

Example: Sorting the Array [10, 80, 30, 90, 40, 50, 70]

Step 1: Initial Call

- **Array**: [10, 80, 30, 90, 40, 50, 70]
- **Pivot**: The first element, which is 10.
- **Left Index (i)**: Starts at 1 (the element right after the pivot).
- **Right Index (j)**: Starts at 6 (the last element).

Example Contd.

step 2: Partitioning the Array

- We compare elements with the pivot (10):
  - **j = 6**: 70 is greater than 10, so no action, decrement j to 5.
  - **j = 5**: 50 is greater than 10, so no action, decrement j to 4.
  - **j = 4**: 40 is greater than 10, so no action, decrement j to 3.
  - **j = 3**: 90 is greater than 10, so no action, decrement j to 2.
  - **j = 2**: 30 is greater than 10, so no action, decrement j to 1.
  - **j = 1**: 80 is greater than 10, so no action, decrement j to 0.

At this point, j reaches 0, and i is still at 1.

3- Recursively Apply Quick Sort:

Step 3: Swapping the Pivot

- We swap the pivot 10 with the element at j:
  - No actual swap is needed since 10 is already in the correct position.
- **Array remains**: [10, 80, 30, 90, 40, 50, 70]
- **Partition Index**: 0 (the position of the pivot).

Now, we make recursive calls on the two sub-arrays:

1. Left sub-array: arr[0...-1] (invalid, no elements).
2. Right sub-array: arr[1...6] → [80, 30, 90, 40, 50, 70].

Time Complexity

- Best and Average Case: O(nlogn) – When the pivot divides the array into relatively equal parts, Quick Sort performs very efficiently.
- Worst Case:
- O(n 2) – If the pivot is always the smallest or largest element (for example, if the array is already sorted), the array will not be divided evenly, leading to poor performance.
- This can be mitigated by using a better pivot selection strategy (e.g., choosing the middle element or using a randomized pivot).

Activity

- Create two groups named Merge Sort and Quick Sort, with each group presenting the advantages of their respective sorting algorithm.
- In addition, each group will also point out the disadvantages of the other algorithm.
- Present at least 4 points
- Each point presented by a group will be worth 0.5 mark.