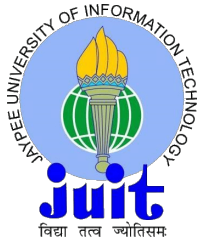

Data Structure

(Course Code: 25B11CI318)

Sorting

Mr. Sandeep Kumar Patel

Assistant Professor

Jaypee University of Information Technology

Waknaghat, Solan, HP-173234

Sorting Algorithms

Definition

Sorting is the process of arranging data elements in a particular order, usually ascending or descending.

Common sorting algorithms:

- Merge Sort
- Quick Sort
- Counting Sort

Goal

Arrange the elements efficiently while minimizing time and space requirements.

Recurrence Relation

Definition

A **recurrence relation** expresses the running time of a recursive algorithm in terms of the running time of smaller input sizes.

General form:

$$\boxed{T(n) = aT\left(\frac{n}{b}\right) + f(n)}$$

where:

- a = number of recursive subproblems
- n/b = size of each subproblem
- $f(n)$ = work performed outside recursion

Example of Recurrence Relation

Suppose an algorithm divides a problem into two equal subproblems and performs $\Theta(n)$ additional work.

Then:

$$\boxed{T(n) = 2T(n/2) + \Theta(n)}$$

For the base case:

$$\boxed{T(1) = \Theta(1)}$$

Using the Master Theorem:

$$a = 2, \qquad b = 2, \qquad f(n) = \Theta(n)$$

and:

$$n^{\log_b a} = n^{\log_2 2} = n$$

Merge Sort

Definition

Merge Sort is a divide-and-conquer sorting algorithm that divides the array into smaller subarrays, recursively sorts them, and then merges the sorted subarrays.

Three main steps:

1. **Divide** – Split the array into two halves.
2. **Conquer** – Recursively sort both halves.
3. **Combine** – Merge the two sorted halves.

$$\boxed{\text{Divide} \rightarrow \text{Sort} \rightarrow \text{Merge}}$$

Merge Sort: Recurrence Relation

For an array of size n:

$$\boxed{T(n) = 2T(n/2) + \Theta(n)}$$

Explanation:

- $2T(n/2)$ – Sort two halves.
- $\Theta(n)$ – Merge the two sorted halves.

Using the Master Theorem:

$$a = 2, \qquad b = 2, \qquad f(n) = \Theta(n)$$

Therefore:

$$\boxed{T(n) = \Theta(n \log n)}$$

Important

Merge Sort has the same asymptotic time complexity in the best, average, and worst cases.

Merge Sort: Example

Consider:

$$A = [38, 27, 43, 3, 9, 82, 10]$$

Step 1: Divide

$$[38, 27, 43, 3] \qquad [9, 82, 10]$$

Further divide:

$$[38]\ [27]\ [43]\ [3]$$

$$[9]\ [82]\ [10]$$

Step 2: Merge sorted elements

$$[27, 38] \qquad [3, 43]$$

$$[9, 82]$$

Then:

$$[3, 27, 38, 43] \qquad [9, 10, 82]$$

Merge Sort: Final Merge

The two sorted subarrays are:

$$[3, 27, 38, 43]$$

and

$$[9, 10, 82]$$

Merge them by repeatedly comparing the first remaining elements.

$$3 < 9$$

$$9 < 27$$

$$10 < 27$$

$$27 < 82$$

Finally:

$$\boxed{[3, 9, 10, 27, 38, 43, 82]}$$

Complexity

$$\boxed{O(n \log n)}$$

Merge Sort: Complexity

Case	Time Complexity
Best Case	$O(n \log n)$
Average Case	$O(n \log n)$
Worst Case	$O(n \log n)$
Space	$O(n)$

Key Point

Merge Sort guarantees $O(n \log n)$ time, but requires additional memory for merging.

Quick Sort

Definition

Quick Sort is a divide-and-conquer sorting algorithm that selects a **pivot** and partitions the array around the pivot.

Main steps:

1. Select a pivot.
2. Partition the array.
3. Elements smaller than the pivot go to the left.
4. Elements greater than the pivot go to the right.
5. Recursively sort both parts.

$$\boxed{\text{Pivot} \rightarrow \text{Partition} \rightarrow \text{Recursion}}$$

Quick Sort: Example

Consider:

$$A = [10, 7, 8, 9, 1, 5]$$

Choose the last element as pivot:

$$\boxed{pivot = 5}$$

After partitioning:

$$[1] \quad 5 \quad [10, 7, 8, 9]$$

The pivot 5 is now in its correct position. Recursively sort:

$$[10, 7, 8, 9]$$

Choose 9 as pivot:

$$[7, 8] \quad 9 \quad [10]$$

Therefore:

$$\boxed{[1, 5, 7, 8, 9, 10]}$$

Quick Sort: Recurrence Relation

If the pivot divides the array approximately equally:

$$\boxed{T(n) = 2T(n/2) + \Theta(n)}$$

Therefore:

$$\boxed{T(n) = \Theta(n \log n)}$$

However, if the pivot is always the smallest or largest element:

$$\boxed{T(n) = T(n-1) + \Theta(n)}$$

Expanding:

$$T(n) = T(n-2) + \Theta(n-1) + \Theta(n)$$

Therefore:

$$\boxed{T(n) = \Theta(n^2)}$$

Quick Sort: Complexity

Case	Time Complexity
Best Case	$O(n \log n)$
Average Case	$O(n \log n)$
Worst Case	$O(n^2)$
Average Space	$O(\log n)$
Worst Space	$O(n)$

Important

The choice of pivot strongly affects the performance of Quick Sort.

Counting Sort

Definition

Counting Sort is a non-comparison sorting algorithm that sorts elements by counting the number of occurrences of each value.

Basic idea:

$$\boxed{\text{Input} \rightarrow \text{Count Frequencies} \rightarrow \text{Construct Sorted Output}}$$

Counting Sort is especially useful when the range of input values is small compared with the number of elements.

Counting Sort: Example

Consider:

$$A = [4, 2, 2, 8, 3, 3, 1]$$

The maximum value is:

$$k = 8$$

Create a count array:

$$C[0 \ldots 8]$$

Value	0	1	2	3	4	5	6	7	8
Count	0	1	2	2	1	0	0	0	1

The count array tells us how many times each value occurs.

Counting Sort: Construct Output

From the count array:

$$C[1] = 1$$

so write:

$$1$$

Since:

$$C[2] = 2$$

write:

$$2, 2$$

Similarly:

$$3, 3$$

$$4$$

and:

$$8$$

Therefore:

$$\boxed{[1, 2, 2, 3, 3, 4, 8]}$$

Counting Sort: Complexity

Let:

- n = number of elements
- k = range of input values

The time complexity is:

$$\boxed{T(n) = O(n + k)}$$

Space complexity:

$$\boxed{O(n + k)}$$

When is Counting Sort Efficient?

Counting Sort is efficient when k is relatively small compared with n.

Merge Sort vs Quick Sort vs Counting Sort

Feature	**Merge Sort**	**Quick Sort**	**Counting Sort**
Technique	Divide & Conquer	Divide & Conquer	Counting
Comparison Based	Yes	Yes	No
Best	$O(n \log n)$	$O(n \log n)$	$O(n + k)$
Average	$O(n \log n)$	$O(n \log n)$	$O(n + k)$
Worst	$O(n \log n)$	$O(n^2)$	$O(n + k)$
Extra Space	$O(n)$	$O(\log n)$	$O(n + k)$
Stable	Yes	Depends	Yes

Key Takeaways

1. **Merge Sort**
   - Divide the array into two halves.
   - Recursively sort the halves.
   - Merge the sorted halves.
   - Time: $O(n \log n)$.

Key Takeaways

1. **Merge Sort**
   - Divide the array into two halves.
   - Recursively sort the halves.
   - Merge the sorted halves.
   - Time: $O(n \log n)$.
2. **Quick Sort**
   - Select a pivot.
   - Partition the array.
   - Recursively sort both sides.
   - Average: $O(n \log n)$.
   - Worst: $O(n^2)$.

Key Takeaways

1. **Merge Sort**
   - Divide the array into two halves.
   - Recursively sort the halves.
   - Merge the sorted halves.
   - Time: $O(n \log n)$.
2. **Quick Sort**
   - Select a pivot.
   - Partition the array.
   - Recursively sort both sides.
   - Average: $O(n \log n)$.
   - Worst: $O(n^2)$.
3. **Counting Sort**
   - Count occurrences of values.
   - Construct sorted output.
   - Time: $O(n + k)$.

Thank You

Questions?