1. Introduction to Data Structures

Basic Terminology

- **Data**: A collection of facts or information that can be processed or analyzed. Data can be numbers, characters, or any other type of information.
- **Data Structure**: A data structure is a way of organizing, storing, and managing data efficiently so that it can be accessed or modified in an optimal way. It allows for better performance in terms of memory usage and execution time.
- **Algorithm**: A step-by-step procedure or formula for solving a problem. Algorithms operate on data structures to solve a problem.
- **Operations on Data Structures**: Common operations include insertion, deletion, searching, updating, and traversal (iterating over the elements).

Types of Data Structures

Data structures can be broadly classified into the following categories:

1. Primitive Data Structures

- These are the basic data types that are directly operated on by the machine. Examples:
  - **Integer**
  - **Character**
  - **Float**
  - **Boolean**

2. Non-Primitive Data Structures

These are more complex structures that are derived from primitive data types. They are mainly classified as:

Linear Data Structures

- **Array**: A collection of elements of the same type stored in contiguous memory locations.
- **Linked List**: A linear data structure where elements (nodes) are connected via pointers.
- **Stack**: A collection of elements that follows the Last In First Out (LIFO) principle.
- **Queue**: A collection of elements that follows the First In First Out (FIFO) principle.

Non-Linear Data Structures

- **Tree**: A hierarchical structure with nodes connected by edges. Common types include binary trees, AVL trees, and B-trees.
- **Graph**: A set of nodes (vertices) connected by edges. Graphs can be directed or undirected, and can also be weighted or unweighted.

Hashing

- **Hash Table**: A data structure that maps keys to values using a hash function for fast access.

Applications of Data Structures

1. **Arrays**: Useful for storing data that needs to be accessed randomly, like in databases, image processing, and sorting algorithms.
2. **Linked Lists**: Used in situations where data is frequently inserted and deleted, such as in dynamic memory allocation, and managing free lists in operating systems.
3. **Stacks**: Essential for algorithms that involve backtracking, such as in expression evaluation (e.g., converting infix expressions to postfix) or undo operations in software.
4. **Queues**: Used in scheduling tasks, managing processes in operating systems, and in breadth-first search (BFS) algorithms.
5. **Trees**: Used in hierarchical data representation like file systems, and in search operations (e.g., binary search trees, AVL trees).
6. **Graphs**: Used in network routing algorithms, social networks, and pathfinding in maps (e.g., shortest path algorithms like Dijkstra's).

---

2. Algorithm and Efficiency

What is an Algorithm?

An **algorithm** is a set of well-defined instructions or steps that describe how to solve a problem or perform a task. It should be:

- **Finite**: An algorithm must always terminate after a finite number of steps.
- **Well-defined**: Each step in an algorithm must be clear and unambiguous.
- **Input**: The algorithm should accept input(s) to process.
- **Output**: It should provide the expected output.

Example: **Finding the largest number in an array**

Algorithm:

1. Start
2. Initialize a variable `max` to the first element of the array.
3. Traverse through each element of the array.
4. If an element is greater than `max`, update `max`.
5. After traversing the entire array, output `max` as the largest element.
6. End

Efficiency of an Algorithm

The **efficiency** of an algorithm is crucial because it determines how fast and resource-efficient it is when solving a problem. There are two main aspects to consider when analyzing the efficiency of an algorithm:

1. **Time Complexity**: Refers to the amount of time an algorithm takes to run as a function of the input size. It helps us understand how the running time of an algorithm grows as the input size increases.

2. **Space Complexity**: Refers to the amount of memory or storage required by an algorithm as a function of the input size.

Time Complexity

Time complexity can be classified into different classes, depending on how the execution time grows with the size of the input:

- **O(1)**: Constant time. The algorithm's performance is unaffected by the size of the input.
- **O(log n)**: Logarithmic time. The algorithm's performance grows logarithmically with the input size (e.g., binary search).
- **O(n)**: Linear time. The performance grows linearly with the input size (e.g., linear search).
- **O(n log n)**: Log-linear time. Common for algorithms like quicksort and mergesort.
- **O(n^2)**: Quadratic time. Often seen in algorithms with nested loops, like bubble sort.
- **O(2^n)**: Exponential time. The performance grows exponentially with the input size. Algorithms with this complexity are generally impractical for large inputs.

Example of Time Complexity:

Consider two algorithms for adding all elements of an array:

- **Algorithm 1** (Linear Time - O(n)):

```
sum = 0
for i from 0 to n-1:
    sum = sum + A[i]
return sum
```

Time complexity: O(n) because the loop runs `n` times.

- **Algorithm 2** (Quadratic Time - O(n^2)):

```
sum = 0
for i from 0 to n-1:
    for j from 0 to n-1:
        sum = sum + A[i] + A[j]
return sum
```

Time complexity: O(n^2) because of the nested loop.

Space Complexity

Space complexity refers to the amount of memory the algorithm uses relative to the input size. It includes:

- **Fixed part**: The space required for variables, constants, etc., which doesn't change with input size.
- **Variable part**: The space required for dynamic memory allocation, such as for recursion or large data structures.

Example of Space Complexity:

Consider the following recursive algorithm for computing the factorial of a number `n`:

```
Factorial(n):
   if n == 0:
       return 1
   else:
       return n * Factorial(n-1)
```

This algorithm has **O(n)** space complexity because each recursive call consumes space on the stack. For `n = 5`, the recursion depth would be 5, leading to 5 function calls in memory.

Efficiency Analysis

When analyzing an algorithm's efficiency, focus on:

1. **Best case**: The best performance scenario (e.g., the element is found at the first position in a search algorithm).
2. **Worst case**: The worst performance scenario (e.g., the element is found at the last position).
3. **Average case**: The expected performance on average, considering a random input distribution.
