Data Structure & Algorithm

Data Structure

- **Definition**: A data structure is a specific way of organizing, managing, and storing data in a computer so that it can be accessed and modified efficiently. Data structures provide a means to manage large amounts of data for various operations like storage, retrieval, and modification.
- **Examples**:
  - **Linear Data Structures**: Arrays, Linked Lists, Stacks, Queues.
  - **Non-linear Data Structures**: Trees, Graphs, Heaps.
  - **Hash-based Structures**: Hash Tables.
- **Importance**:
  - **Efficiency**: Proper use of data structures allows for efficient data processing, which is critical in software development.
  - **Scalability**: Helps in handling large datasets and complex operations.
  - **Optimized Memory Usage**: Efficient data structures help in optimizing the memory usage of a program.

Algorithm

- **Definition**: An algorithm is a step-by-step procedure or formula for solving a problem or performing a task. It is a sequence of well-defined instructions that take input, process the data, and produce output.
- **Examples**:
  - **Sorting Algorithms**: Bubble Sort, Quick Sort, Merge Sort.
  - **Searching Algorithms**: Binary Search, Linear Search.

- **Graph Algorithms**: Dijkstra's Algorithm, Bellman-Ford Algorithm.
- **Dynamic Programming**: Knapsack Problem, Fibonacci Sequence.

Asymptotic Notation

- ✓ **Big O (O)**: Upper bound, worst-case scenario.
- ✓ **Omega (Ω)**: Lower bound, best-case scenario.
- ✓ **Theta (Θ)**: Exact bound, both upper and lower bounds.
- ✓ **Little o (o)**: Upper bound, not tight.
- ✓ **Little omega (ω)**: Lower bound, not tight.

Data Structures

1. Array

- **Definition**: A collection of elements identified by index or key. Elements are stored in contiguous memory locations, and each element can be accessed by its index.
- **Time Complexity**:
  - **Access**: O(1)
  - **Search**: O(n)
  - **Insertion**: O(n)(for inserting at the beginning or middle)
  - **Deletion**: O(n)(for deleting from the beginning or middle)

2. Linked List

- **Definition**: A linear data structure where each element (node) contains a value and a pointer (or reference) to the next node in the sequence. There are various types of linked lists, such as singly linked lists, doubly linked lists, and circular linked lists.
- **Time Complexity**:

- **Access**: O(n)
- **Search**: O(n)
- **Insertion**: O(1)(assuming insertion at the beginning or end with a tail pointer)
- **Deletion**: O(1)(assuming deletion of the first node or known node)

3. Stack

- **Definition**: A linear data structure that follows the Last In, First Out (LIFO) principle. The element that is inserted last is the first one to be removed. Operations are typically performed at one end, called the "top" of the stack.
- **Time Complexity**:
  - **Push (Insertion)**: O(1)
  - **Pop (Deletion)**: O(1)
  - **Peek (Access Top Element)**: O(1)
  - **Search**: O(n)

4. Queue

- **Definition**: A linear data structure that follows the First In, First Out (FIFO) principle. The element that is inserted first is the first one to be removed. Operations are typically performed at both ends—enqueuing at the rear and dequeuing at the front.
- **Time Complexity**:
  - **Enqueue (Insertion)**: O(1)
  - **Dequeue (Deletion)**: O(1)
  - **Peek (Access Front Element)**: O(1)
  - **Search**: O(n)

5. Deque (Double-Ended Queue)

- **Definition**: A linear data structure that allows insertion and deletion of elements from both ends, i.e., from the front and the rear.
- **Time Complexity**:
  - **Insertion at both ends (Enqueue/Push Front/Rear)**: O(1)
  - **Deletion from both ends (Dequeue/Pop Front/Rear)**: O(1)
  - **Peek (Access Front/Rear Element)**: O(1)
  - **Search**: O(n)

6. Circular Queue

- **Definition**: A linear data structure in which the last position is connected back to the first position to make a circle. It operates like a queue but in a circular manner.
- **Time Complexity**:
  - **Enqueue (Insertion)**: O(1)
  - **Dequeue (Deletion)**: O(1)
  - **Peek (Access Front Element)**: O(1)
  - **Search**: O(n)

7. Priority Queue

- **Definition**: A special type of queue where each element is associated with a priority, and elements are served based on their priority. Elements with higher priority are dequeued before elements with lower priority.
- **Time Complexity**:
  - **Insertion (Enqueue)**: O(logn) (using a heap)
  - **Deletion (Dequeue)**: O(logn)

- **Peek (Access Highest Priority Element)**: O(1)(if using a max-heap)

Summary of Time Complexities:

Linear Data Structure	Access	Search	Insertion	Deletion
Array	O(1)	O(n)	O(n)	O(n)
Linked List	O(n)	O(n)	O(1)	O(1)
Stack	O(1)	O(n)	O(1)	O(1)
Queue	O(1)	O(n)	O(1)	O(1)
Deque	O(1)	O(n)	O(1)	O(1)
Circular Queue	O(1)	O(n)	O(1)	O(1)
Priority Queue	O(logn)	O(n)	O(logn)	O(logn)

8.Hash Table

Definition: A hash table (or hash map) is a data structure that stores key-value pairs. It uses a hash function to compute an index (also called a hash code) into an array of buckets or slots, from which the desired value can be found. Hash tables are designed for efficient insertion, deletion, and lookup operations.

Components:

1. **Keys**: Unique identifiers used to store and retrieve data.
2. **Values**: Data associated with keys.
3. **Hash Function**: A function that takes a key as input and produces an index in the array (hash code). The quality of the hash function affects the efficiency of the hash table.

4. **Buckets (Slots)**: The array where the data is stored. Each index in this array is called a bucket or slot.

Operations:

- **Insertion**: Given a key and value, compute the hash code for the key using the hash function, and store the key-value pair in the corresponding bucket.
- **Lookup**: Given a key, compute its hash code, and retrieve the associated value from the corresponding bucket.
- **Deletion**: Given a key, compute its hash code, locate the key-value pair in the bucket, and remove it.

Collision Handling: Collisions occur when two different keys hash to the same index. There are several methods to handle collisions:

1. **Chaining**: Each bucket points to a linked list (or another data structure) of key-value pairs that share the same hash code. In the case of a collision, the new key-value pair is simply added to the list.
2. **Open Addressing**: If a collision occurs, the algorithm searches for the next empty slot in the array (using techniques like linear probing, quadratic probing, or double hashing) and places the new key-value pair there.

Time Complexity:

- **Average Case**:
  - **Insertion**: O(1)
  - **Lookup**: O(1)
  - **Deletion**: O(1)
- **Worst Case** (when collisions are poorly handled or the hash function is inefficient):
  - **Insertion**: O(n)

- **Lookup**: O(n)
- **Deletion**: O(n)The worst case can occur when all keys hash to the same index, leading to a chain of length n.

Tree

1. Binary Tree

- **Definition**: A tree data structure in which each node has at most two children, referred to as the left child and the right child.
- **Example**: A simple structure to represent hierarchical data, such as file systems.

2. Binary Search Tree (BST)

- **Definition**: A binary tree where each node has a value, and the left child’s value is less than the parent node's value, while the right child’s value is greater. This property makes searching efficient.
- **Example**: Useful in implementing efficient search algorithms, like in databases.

3. Balanced Tree

- **Definition**: A binary tree in which the height of the left and right subtrees of any node differs by at most one. Examples include AVL trees and Red-Black trees.
- **Example**: Ensures that operations like search, insert, and delete can be performed in O(log⁡n)O(\log n)O(logn) time.

4. AVL Tree

- **Definition**: A self-balancing binary search tree where the difference in heights between the left and right subtrees (balance factor) is at most one for every node.

- **Example**: Provides O(log⁡n)O(\log n)O(logn) time complexity for insertion, deletion, and search operations.

5. Red-Black Tree

- **Definition**: A balanced binary search tree where each node contains an extra bit for storing color (either red or black). The tree maintains balance by ensuring that no red node has a red child and that every path from a node to its descendant leaves has the same number of black nodes.
- **Example**: Commonly used in the implementation of associative containers like maps and sets in the C++ Standard Library.

6. B-Tree

- **Definition**: A self-balancing search tree in which nodes can have more than two children. It is typically used in databases and file systems. B-trees maintain sorted data and allow for searches, sequential access, insertions, and deletions in logarithmic time.
- **Example**: Used in databases and file systems for efficient data retrieval and storage.

7. B+ Tree

- **Definition**: A type of B-tree in which all values are stored in the leaf nodes and internal nodes store only keys. Leaf nodes are linked, which allows for efficient range queries.
- **Example**: Often used in databases and file systems for efficient query operations.

8. Heap (Binary Heap)

- **Definition**: A complete binary tree that satisfies the heap property: in a max-heap, each parent node is greater than or equal to its children, and in a min-heap, each parent node is less than or equal to its children.

- **Example**: Commonly used in priority queues and the heap sort algorithm.

9. Trie (Prefix Tree)

- **Definition**: A tree-like data structure that stores a dynamic set of strings, typically used for quick lookup by keys that are prefixes of the stored elements.
- **Example**: Used in implementing dictionaries, autocomplete features, and spell checkers.

10. Segment Tree

- **Definition**: A binary tree used for storing intervals or segments. It allows querying which segments cover a point and updating segment values efficiently.
- **Example**: Used in range query problems, such as finding the sum, minimum, or maximum over an interval.

11. Fenwick Tree (Binary Indexed Tree)

- **Definition**: A tree-like data structure that provides efficient methods for calculation and manipulation of the prefix sums of a table of values.
- **Example**: Used in scenarios where there are frequent updates and queries on prefix sums.

12. Splay Tree

- **Definition**: A self-adjusting binary search tree where recently accessed elements are moved to the root. This property ensures that frequently accessed elements are quick to reach.
- **Example**: Used in implementing caches and other scenarios where access patterns exhibit locality.

13. Suffix Tree

- **Definition**: A compressed trie that contains all the suffixes of a given text. It allows for fast substring searches, among other operations.
- **Example**: Used in string matching algorithms and bioinformatics.

14. K-D Tree (k-dimensional tree)

- **Definition**: A space-partitioning data structure used to organize points in a k-dimensional space. It is a binary tree where each node represents a k-dimensional point.
- **Example**: Used in multidimensional search problems, like range searches and nearest neighbor searches.

15. Merkle Tree

- **Definition**: A tree in which every leaf node is labeled with the cryptographic hash of a data block, and every non-leaf node is labeled with the cryptographic hash of its child nodes. It is used to verify the integrity of data.
- **Example**: Commonly used in blockchain technology to ensure data consistency and integrity.

16. Suffix Array

- **Definition**: An array of integers providing the starting positions of suffixes of a string, sorted in lexicographical order. It’s not a tree per se but is closely related to the suffix tree and often discussed together.
- **Example**: Used in text processing, like searching for patterns and data compression algorithms.

17. Expression Tree

- **Definition**: A binary tree where the internal nodes represent operators and the leaf nodes represent operands. It is used to represent arithmetic expressions.
- **Example**: Used in compilers and calculators to evaluate expressions.

18. Decision Tree

- **Definition**: A tree used for decision-making purposes, where each internal node represents a test on an attribute, each branch represents the outcome of the test, and each leaf node represents a class label or decision.
- **Example**: Widely used in machine learning for classification and regression tasks.

19. Interval Tree

- **Definition**: A tree data structure that holds intervals, or ranges, and allows for efficient querying of all intervals that overlap with any given interval or point.
- **Example**: Used in computational geometry, particularly for scheduling problems.

20. Cartesian Tree

- **Definition**: A binary tree derived from a sequence of numbers, where an in-order traversal of the tree yields the original sequence, and the tree satisfies the heap property.
- **Example**: Used in applications like finding the lowest common ancestor or range minimum queries.

Graph

In data structures, a graph is a collection of nodes (also called vertices) and edges that connect pairs of nodes. Graphs are used to represent relationships between objects. Here's a detailed overview of the graph concept:

1. Basic Terminology

- Vertex (Node): A fundamental unit of a graph, representing an entity or object.
- Edge: A connection between two vertices in a graph.
- Directed Graph (Digraph): A graph where the edges have a direction, meaning they go from one vertex to another.
- Undirected Graph: A graph where the edges have no direction, meaning they simply connect two vertices.
- Weighted Graph: A graph where each edge has a numerical value or weight associated with it.
- Unweighted Graph: A graph where the edges do not have any weights.
- Adjacent Vertices: Two vertices are adjacent if they are connected by an edge.
- Degree of a Vertex: The number of edges connected to a vertex. In directed graphs, the in-degree is the number of incoming edges, and the out-degree is the number of outgoing edges.

2. Types of Graphs

- Complete Graph: A graph where every pair of distinct vertices is connected by a unique edge.
- Connected Graph: A graph where there is a path between every pair of vertices.
- Disconnected Graph: A graph where at least one pair of vertices is not connected by a path.
- Cyclic Graph: A graph that contains at least one cycle, which is a path that starts and ends at the same vertex.
- Acyclic Graph: A graph with no cycles. A special kind of acyclic graph is a Tree, where there is exactly one path between any two vertices.

3. Graph Representation

- Adjacency Matrix: A 2D array used to represent a graph, where the cell at row iii and column jjj represents the edge between vertices iii and jjj. This representation is space-inefficient for sparse graphs.
- Adjacency List: An array of lists, where each list corresponds to a vertex and contains a list of adjacent vertices. This is more space-efficient, especially for sparse graphs.
- Edge List: A list of all edges in the graph, where each edge is represented as a pair (or tuple) of vertices.

Recursion

Definition: Recursion is a programming technique where a function calls itself in order to solve a problem. The problem is typically broken down into smaller, more manageable subproblems of the same type. The recursive function continues to call itself with these subproblems until it reaches a base case, which stops the recursion.

Searching Algorithm

Linear Search

Definition: Linear search is a simple search algorithm that checks every element in a list sequentially until the desired element is found or the end of the list is reached. It is also known as a sequential search.

How it Works:

- Start at the beginning of the list.
- Compare the target value with each element of the list.
- If a match is found, return the index of the element.
- If the end of the list is reached without finding the target, return an indication that the target is not present (e.g., -1 or false).

Time Complexity:

- **Best Case**: O(1)(if the target is the first element).
- **Average Case**: O(n)(if the target is in the middle).
- **Worst Case**: O(n)(if the target is the last element or not present).

Binary Search

Definition: Binary search is an efficient search algorithm that works on sorted lists. It repeatedly divides the search interval in half and compares the target value with the middle element of the current interval.

How it Works:

- Start with the entire list and identify the middle element.
- Compare the target value with the middle element:
  - If they are equal, the search is successful.
  - If the target is less than the middle element, narrow the search to the left half.
  - If the target is greater than the middle element, narrow the search to the right half.
- Repeat the process on the new half until the target is found or the interval is empty.

Time Complexity:

- **Best Case**: O(1)(if the target is the middle element of the list).
- **Average Case**: O(logn) (due to the halving of the search space).
- **Worst Case**: O(logn) (when the list is fully searched without finding the target).

Summary:

Search Algorithm	Best Case	Average Case	Worst Case
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(logn)	O(logn)

Key Differences:

- **Linear Search** can be applied to unsorted lists but is less efficient for large datasets.
- **Binary Search** requires a sorted list but is much more efficient, particularly for large datasets, due to its logarithmic time complexity.

Sorting Algorithm

1. Insertion Sort

Definition: A simple sorting algorithm that builds the final sorted array one item at a time. It picks each element from the input data and inserts it into its correct position in a sorted part of the array.

2. Bubble Sort

Definition: A straightforward sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until no swaps are needed, indicating that the list is sorted.

3. Shell Sort

Definition: An improvement of insertion sort that sorts elements at a specific gap apart. It progressively reduces the gap between elements to be compared, eventually performing a standard insertion sort when the gap is reduced to one.

4. Heap Sort

Definition: A comparison-based sorting algorithm that uses a binary heap data structure. It first builds a heap from the input data and then repeatedly extracts the maximum (or minimum) element from the heap and rebuilds the heap until the list is sorted.

5. Counting Sort

Definition: A non-comparison-based sorting algorithm that counts the number of occurrences of each distinct element in the input. It then uses this count to place elements directly into their correct position in the output array.

6. Radix Sort

Definition: A non-comparison-based sorting algorithm that sorts numbers by processing individual digits. It processes digits from the least significant digit (LSD) to the most significant digit (MSD) or vice versa, using a stable sorting algorithm like counting sort as a subroutine.

7. Merge Sort

Definition: A divide-and-conquer sorting algorithm that divides the input array into two halves, recursively sorts each half, and then merges the two sorted halves to produce the final sorted array.

8. Quick Sort

Definition: A divide-and-conquer sorting algorithm that selects a 'pivot' element from the array, partitions the other elements into two sub-arrays according to whether they are less than or greater than the pivot, and recursively sorts the sub-arrays.

9. Selection Sort

Definition: A simple sorting algorithm that repeatedly selects the smallest (or largest) element from the unsorted portion of the array and swaps it with the first unsorted element, thereby growing the sorted portion of the array.

Sorting Algorithm	Best Case Time Complexity	Average Case Time Complexity	Worst Case Time Complexity	Space Complexity	Stable
Insertion Sort	O(n)	O(n2)O(n^2)O(n2)	O(n2)O(n^2)O(n2)	O(1)	Yes
Bubble Sort	O(n)	O(n2)O(n^2)O(n2)	O(n2)O(n^2)O(n2)	O(1)	Yes
Selection Sort	O(n2)O(n^2)O(n2)	O(n2)O(n^2)O(n^2)	O(n2)O(n^2)O(n2)	O(1)	No
Shell Sort	O(nlogn)	O(nlog2n)O(n \log^2 n)O(nlog2n)	O(nlog2n)O(n \log^2 n)O(nlog2n)	O(1)	No
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	No
Counting Sort	O(n+k)O(n + k)O(n+k)	O(n+k)O(n + k)O(n+k)	O(n+k)O(n + k)O(n+k)	O(n+k)O(n + k)O(n+k)	Yes
Radix Sort	O(d×(n+k))O(d * (n + k))O(d×(n+k))	O(d×(n+k))O(d * (n + k))O(d×(n+k))	O(d×(n+k))O(d * (n + k))O(d×(n+k))	O(n+k)O(n + k)O(n+k)	Yes
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	Yes
Quick Sort	O(nlogn)	O(nlogn)	O(n2)O(n^2)O(n2)	O(logn)O(\log n)O(logn)	No

Graph Traversal Algorithm:

1.Breadth-First Search (BFS)

Definition: A graph traversal algorithm that explores all the vertices at the present depth level before moving on to vertices at the next depth level. It uses a queue to keep track of the vertices to be explored next.

2.Depth-First Search (DFS)

Definition: A graph traversal algorithm that explores as far as possible along each branch before backtracking. It uses a stack (either implicitly through recursion or explicitly) to keep track of the vertices to be explored.

Greedy Algorithm

Definition: A greedy algorithm is a problem-solving technique that makes a series of choices, each of which looks best at the moment (i.e., it chooses the locally optimal solution). The algorithm aims to find a global optimum by selecting the best option available at each step with the hope that these local solutions will lead to a global optimal solution.

Examples:

1. Greedy Algorithms for Optimization Problems:
    - Activity Selection Problem: Select the maximum number of non-overlapping activities.
    - Kruskal's Algorithm: Finds the minimum spanning tree of a graph by selecting edges with the smallest weights.
    - Prim's Algorithm: Another algorithm for finding the minimum spanning tree, growing the tree by adding the smallest edge that connects a vertex in the tree to a vertex outside the tree.
    - Huffman Coding: Creates an optimal prefix code for data compression.
2. Fractional Knapsack Problem: Unlike the 0/1 knapsack problem, the fractional knapsack problem allows breaking items into fractions. The

greedy approach selects items based on their value-to-weight ratio and fills the knapsack with the most valuable items first.

Shortest Path Algorithm

1. Dijkstra's Algorithm

Definition: Dijkstra's Algorithm finds the shortest path from a single source vertex to all other vertices in a weighted graph with non-negative edge weights. It uses a priority queue to repeatedly select the vertex with the smallest known distance and updates the distances to its neighboring vertices.

2. Bellman-Ford Algorithm

Definition: The Bellman-Ford Algorithm computes the shortest paths from a single source vertex to all other vertices in a weighted graph, capable of handling negative edge weights. It works by relaxing all edges up to V−1V-1V−1 times, where VVV is the number of vertices, and can detect negative weight cycles by performing one additional relaxation step.

3. Floyd-Warshall Algorithm

Definition: The Floyd-Warshall Algorithm finds the shortest paths between all pairs of vertices in a weighted graph. It uses a dynamic programming approach to iteratively update the shortest paths considering each vertex as an intermediate point, accommodating both positive and negative edge weights (but no negative weight cycles).

Depth-First Search (DFS) Traversals

In DFS, the algorithm explores as far down a branch as possible before backtracking. DFS traversal can be implemented in three primary ways:

1. Inorder Traversal (Left, Root, Right)

- Description: In an Inorder traversal, the left subtree is visited first, followed by the root, and then the right subtree. This traversal is

particularly important for binary search trees (BST) because it visits the nodes in non-decreasing order.

- Algorithm:
    1. Recursively traverse the left subtree.
    2. Visit the root node.
    3. Recursively traverse the right subtree.

2. Preorder Traversal (Root, Left, Right)

- Description: In a Preorder traversal, the root node is visited first, followed by the left subtree and then the right subtree. This traversal is used to create a copy of the tree or to evaluate expressions.
- Algorithm:
    1. Visit the root node.
    2. Recursively traverse the left subtree.
    3. Recursively traverse the right subtree.

3. Postorder Traversal (Left, Right, Root)

- Description: In a Postorder traversal, the left subtree is visited first, followed by the right subtree, and finally the root node. This traversal is used for deleting the tree or for postfix expression evaluation.
- Algorithm:
    1. Recursively traverse the left subtree.
    2. Recursively traverse the right subtree.
    3. Visit the root node.

Breadth-First Search (BFS) Traversal

In BFS, also known as Level Order Traversal, the algorithm visits all the nodes at the present depth level before moving on to nodes at the next depth level.

4. Level Order Traversal

- Description: Level Order traversal visits nodes level by level from top to bottom and left to right. It uses a queue to keep track of nodes at the current level.
- Algorithm:
    1. Enqueue the root node.
    2. While the queue is not empty:
        - Dequeue a node from the front.
        - Visit the dequeued node.
        - Enqueue its left and right children (if they exist).

Dynamic Programming

Dynamic programming (DP) is a powerful technique for solving complex problems by breaking them down into smaller, simpler subproblems. It stores the solutions to these subproblems to avoid recomputing them repeatedly, leading to efficient solutions for problems that exhibit certain characteristics.

Here's a comprehensive breakdown of dynamic programming:

Key Concepts:

- **Overlapping Subproblems:** A problem can be broken down into subproblems, and some of these subproblems might be solved multiple times during the overall solution process.
- **Optimal Substructure:** The optimal solution to the entire problem can be constructed from optimal solutions to its subproblems. This means

that if you have the best solutions for smaller pieces, you can combine them to find the best solution for the bigger problem.

- **Memoization:** A technique to store the solutions to subproblems to avoid redundant calculations. Dynamic programming often relies on memoization to store previously computed results and retrieve them efficiently when needed.

How Dynamic Programming Works:

1. **Identify Overlapping Subproblems:** The first step is to analyze the problem and recognize if it can be broken down into subproblems that are solved repeatedly.
2. **Define the Subproblems:** Clearly define the subproblems and the optimal solution you want for each one.
3. **Build Bottom-Up:** Start by solving the smallest and most basic subproblems. Then, gradually build up solutions to larger subproblems using the solutions to smaller ones stored through memoization.
4. **Solve the Original Problem:** Once you have solutions to all necessary subproblems, use them to construct the optimal solution for the original problem.

Benefits of Dynamic Programming:

- **Efficiency:** By storing subproblem solutions, DP avoids redundant calculations, leading to significant performance improvements for problems with overlapping subproblems.
- **Clarity and Modularity:** Breaking down complex problems into smaller, solvable subproblems can improve code readability and maintainability.

Drawbacks of Dynamic Programming:

- **Space Complexity:** Storing solutions to subproblems can require additional memory space.

- **Applicability:** Not all problems are suitable for dynamic programming. It's most effective for problems with overlapping subproblems and optimal substructure properties.

Examples of Dynamic Programming Problems:

- **Fibonacci Numbers:** Calculating the nth Fibonacci number can be done recursively, but it involves redundant calculations. DP stores previously computed Fibonacci numbers to solve for larger n efficiently.
- **Knapsack Problem:** You have a knapsack with a weight limit and items with varying weights and values. DP helps determine the optimal combination of items to maximize the total value within the weight limit.
- **Longest Common Subsequence (LCS):** Given two sequences, DP can find the longest subsequence that appears in both sequences in the same order.
- **Edit Distance:** DP can be used to find the minimum number of edits (insertions, deletions, substitutions) required to transform one string into another.