ADSA Topic Covered

1. Complexity of an Algorithm

a. Time Complexity

Time complexity refers to the computational time an algorithm takes to run as a function of the size of the input data. It allows us to estimate the efficiency of an algorithm irrespective of hardware and software implementation details.

- **Types**:

- **Constant Time** $O(1)$: The algorithm's runtime is constant, irrespective of input size (e.g., accessing an array element).
- **Logarithmic Time** $O(\log n)$: Runtime grows logarithmically, common in divide-and-conquer algorithms like binary search.
- **Linear Time** $O(n)$: Runtime grows linearly with input size (e.g., iterating through an array).
- **Quadratic Time** $O(n^2)$: Common in nested loops (e.g., Bubble Sort).
- **Exponential Time** $O(2^n)$: Often seen in brute force solutions to complex problems.

b. Worst Case

This scenario represents the maximum time an algorithm will take for any input of size nnn. It helps in understanding the algorithm's efficiency under the least favorable conditions.

- **Example**: For Quick Sort, the worst case occurs when the pivot is always the smallest or largest element, leading to $O(n^2)$ complexity.

c. Omega (Ω) Notation

Omega notation represents the lower bound of an algorithm's runtime, i.e., the best-case scenario. This is important to evaluate how well an algorithm can perform under ideal conditions.

- **Example**: In searching, the best-case scenario for linear search occurs when the desired element is the first element, $\Omega(1)$.

---

2. Algorithms

i. Divide and Conquer

This technique involves solving a problem by breaking it into smaller, manageable sub-problems, solving them independently, and then combining their solutions to solve the original problem.

- **Steps**:
    1. **Divide**: Split the problem into smaller sub-problems.
    2. **Conquer**: Solve the sub-problems recursively.
    3. **Combine**: Merge the solutions of the sub-problems.
- **Examples**:
    - **Merge Sort**: Splits the array into halves, sorts them, and merges.

    - **Quick Sort**: Divides the array around a pivot element, recursively sorts sub-arrays.

ii. Greedy Algorithm

Greedy algorithms make locally optimal choices at each step, aiming for a globally optimal solution.

- **Steps**:
    1. Choose the best option at the current step.
    2. Repeat for the remaining sub-problems.
- **Examples**:
    - **Activity Selection Problem**: Select the maximum number of activities that don't overlap.
    - **Huffman Coding**: Builds the optimal prefix code using a priority queue.

iii. Local Optimization Approach

A variant of greedy algorithms that focuses on optimizing smaller parts of a problem in the hope of achieving the best global result.

- **Example**: Hill climbing in optimization problems.

---

3. Dynamic Programming (DP)

Dynamic programming solves problems by breaking them into overlapping sub-problems and storing the results to avoid redundant computations.

- **Characteristics**:
    - **Optimal Substructure**: The solution to a problem can be constructed from solutions to its sub-problems.
    - **Overlapping Sub-problems**: Sub-problems recur during computation.
- **Example**:
    - **Fibonacci Numbers**: Using recursion with memoization to reduce redundancy.
    - **Matrix Chain Multiplication**: Computes the minimum number of scalar multiplications.

---

4. Knapsack Problem

This problem involves selecting items to maximize profit without exceeding a given weight capacity.

i. Fractional Knapsack Problem

- Items can be divided into fractions.
- Uses a greedy approach, sorting items by profit-to-weight ratio.

ii. 0/1 Knapsack Problem

- Items cannot be divided; either take the entire item or skip it.
- Solved using dynamic programming.

5. Master Theorem

The Master Theorem provides a shortcut for analyzing the time complexity of divide-and-conquer algorithms. It applies to recurrence relations of the form:
$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$
Where a, b, and d are constants.

- **Case 1**: $a/b^d > 1$: $T(n) = O(n^{\log_b a})$.
- **Case 2**: $a/b^d = 1$: $T(n) = O(n^d \log n)$.
- **Case 3**: $a/b^d < 1$: $T(n) = O(n^d)$.

6. Travelling Salesman Problem (TSP)

Problem Statement:

Given a set of cities and distances between each pair, find the shortest possible route that visits each city exactly once and returns to the origin city.

Example:

- **Cities**: A, B, C, D.
- **Distances**:
  ```
  A → B: 10, A → C: 15, A → D: 20
  B → C: 35, B → D: 25
  C → D: 30
  ```
  Brute force evaluates all routes: A → B → C → D → A, A → B → D → C → A, etc., calculating distances for each.

Real-World Applications:

- Logistics and delivery systems.
- Circuit design in electronics.

Solutions:

- **Brute Force**: Explores all permutations ($O(n!)$).
- **Dynamic Programming (Held-Karp)**: Tracks subproblems using a bitmask, reducing complexity to $O(n^2 \cdot 2^n)$.
- **Approximation**: Nearest Neighbor Algorithm finds a suboptimal solution quickly.

7. NP-Problem Algorithm

Definition:

An NP problem is one where verifying a solution can be done in polynomial time, but finding the solution itself might take exponential time.

Example:

- **Graph Coloring**: Assign colors to nodes of a graph such that no two adjacent nodes share the same color.
- **SAT (Boolean Satisfiability Problem)**: Determine if a boolean formula can be satisfied by some assignment of true/false values.

Key Distinctions:

- **P Problems**: Solvable in polynomial time (e.g., sorting).
- **NP Problems**: Solution verification in polynomial time (e.g., Subset Sum).
- **NP-Complete**: A subset of NP that is as hard as any other NP problem (e.g., TSP).

Importance:

If any NP-complete problem is solved in polynomial time, all NP problems can be solved in polynomial time.

---

8. Subset Sum Problem

Problem:

Given a set $\{3, 34, 4, 12, 5, 2\}$ and a target sum $S = 9$, determine if there is a subset whose elements sum to S.

Approach:

- **Dynamic Programming**:
  - Create a table $dp[i][j]$, where $dp[i][j]$ is true if a subset of the first i elements has a sum j.
  - **Recurrence Relation**:

    $$dp[i][j] = dp[i-1][j] \text{ (excluding the element) OR } dp[i-1][j - arr[i-1]] \text{ (including it)}.$$
- **Output**:
  For $S = 9$, the subset $\{4, 5\}$ satisfies the condition.

---

9. Vertex Cover Problem

Problem:

Find the minimum set of vertices that covers all edges in a graph.

Example:

- **Graph**:

```
A--B
|  |
C--D
```

  Vertex Cover: {A, C} or {B, D}.

Applications:

- Network security (monitoring critical points).
- Resource allocation in systems.

Algorithm:

1. Start with all vertices and remove unnecessary ones greedily.
2. Approximation: Ensures a solution within 2×OPTIMAL SIZE2 \times \text{OPTIMAL SIZE}2×OPTIMAL SIZE.

---

10. N-Queen Problem

Example for 4-Queens:

Place 4 queens on a 4×44 \times 44×4 chessboard so that no two queens threaten each other.

Steps:

1. Place a queen in the first row.
2. Use backtracking to explore possible placements in subsequent rows.
3. If a placement violates constraints, backtrack and try a different column.

Solution:

A valid configuration:

```
. Q . .
. . . Q
Q . . .
. . Q .
```

Applications:

- Constraint satisfaction problems.
- AI game-solving techniques.

---

11. Huffman Coding

Example:

Given characters and their frequencies:

- A: 5, B: 9, C: 12, D: 13, E: 16, F: 45.

Steps:

1. Build a priority queue with all characters as nodes.
2. Merge the two least frequent nodes into a parent node.
3. Repeat until only one node (the root) remains.

Codes:

- A: 1100, B: 1101, C: 100, D: 101, E: 111, F: 0.

Output:

Encoded string requires fewer bits compared to fixed-length encoding.

---

12. Longest Common Subsequence (LCS)

Example:

Find LCS of "ABCBDAB" and "BDCAB".

Steps:

1. Use a DP table:
   - Rows represent one string, columns represent the other.
   - $dp[i][j] = dp[i-1][j-1] + 1 \text{ (if match) OR } \max(dp[i-1][j], dp[i][j-1]) \text{ (if not)}$.
2. Trace back to construct LCS.

LCS: "BDAB".

Applications:

- File comparison.
- Bioinformatics for sequence alignment.

---

13. Trees (Expanded)

Examples of Traversals:

- **Inorder (Left-Root-Right)**:

  mathematica

  ```
  Input Tree:
      1
     / \
    2   3
  Inorder: 2 → 1 → 3.
  ```

- **Postorder (Left-Right-Root)**:

  mathematica

  ```
  Input Tree:
      1
     / \
    2   3
  Postorder: 2 → 3 → 1.
  ```

---

15. Strassen's Matrix Multiplication

Example:

Multiply two 2×2 matrices:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}.$$

Steps:

1. Compute 7 products: M_1, M_2, ..., M_7.
2. Combine to form the resultant matrix.

---

19. Ant Colony Optimization (ACO)

Example:

Solve TSP for cities A, B, C, and D:

- Ants randomly choose paths.
- Pheromones on shorter paths intensify, guiding subsequent ants.
- Paths are refined iteratively.

Applications:

- Network routing.
- Job scheduling.