UNIT–1: Introduction & Analysis of Algorithms

Algorithm

An **algorithm** is a finite sequence of well-defined steps used to solve a problem.

Characteristics of an Algorithm

1. **Input** – Takes zero or more inputs.
2. **Output** – Produces at least one output.
3. **Definiteness** – Each step is clear and unambiguous.
4. **Finiteness** – Terminates after a finite number of steps.
5. **Effectiveness** – Each step is basic and feasible.
6. **Generality** – Applicable to a class of problems.

Analysis of Algorithm

Algorithm analysis determines **efficiency** in terms of **time and space**.

Asymptotic Analysis

Used to analyze algorithm behavior for **large input sizes (n)**.

Complexity Bounds

1. **Big-O (O)** – Upper bound (Worst case)
2. **Big-Ω (Ω)** – Lower bound (Best case)
3. **Big-Θ (Θ)** – Tight bound (Average case)

Example:
Linear Search

- Best case: $\Omega(1)$
- Worst case: $O(n)$
- Average case: $\Theta(n)$

Best, Average, and Worst Case Analysis

- **Best Case**: Minimum time taken
- **Average Case**: Expected time for random input
- **Worst Case**: Maximum time taken

Performance Measurement

1. **Time Complexity** – Number of operations
2. **Space Complexity** – Memory used

Time–Space Trade-off

An algorithm may use:

- More memory to reduce time (e.g., hashing)
- More time to save memory (e.g., brute force)

Analysis of Recursive Algorithms

Uses **recurrence relations**.

Substitution Method

- Guess the solution
- Prove by induction

Recursion Tree Method

- Represent recursive calls as a tree
- Sum cost at each level

Master's Theorem

Used for recurrences of form:

```
T(n) = aT(n/b) + f(n)
```

Three cases:

1.
2.
3.

UNIT–2: Divide and Conquer & Heaps

Divide and Conquer Paradigm

Steps:

1. Divide problem into sub-problems
2. Conquer recursively
3. Combine results

Binary Search

- Search in sorted array
- Time: O(log n)

Merge Sort

- Divide array, sort recursively, merge
- Time: O(n log n)
- Stable sorting

Quick Sort

- Choose pivot
- Partition array
- Average: O(n log n)
- Worst: O(n^2)

Linear Time Selection

Find k-th smallest element using **Median of Medians**

- Time: O(n)

Strassen's Matrix Multiplication

- Reduces multiplication operations
- Time: O(n^2.81)

Karatsuba Algorithm

- Fast multiplication of large numbers
- Time: O(n^1.585)

Heap

A **complete binary tree** satisfying heap property.

Min Heap

- Parent ≤ children

Max Heap

- Parent ≥ children

Build Heap

- Convert array into heap
- Time: O(n)

Heap Sort

1. Build heap
2. Repeatedly remove root

- Time: O(n log n)
- In-place sorting

UNIT–3: Algorithm Design Techniques

Brute Force

- Try all possibilities
- Simple but inefficient
- Example: Linear search

Greedy Algorithm

- Make locally optimal choice

Greedy Examples

1. **Minimum Cost Spanning Tree**
    - Prim's Algorithm
    - Kruskal's Algorithm
2. **Knapsack (Fractional)**
    - Choose highest profit/weight ratio
3. **Job Sequencing**
    - Maximize profit with deadlines
4. **Huffman Coding**
    - Optimal prefix code for compression
5. **Single Source Shortest Path**
    - Dijkstra's algorithm

Backtracking

- Try possible solutions
- Undo when constraint violated
- Example: N-Queens

Branch and Bound

- Optimization version of backtracking
- Uses bounds to prune search space

UNIT–4: Dynamic Programming & Heuristics

Dynamic Programming (DP)

Used when:

- Overlapping sub-problems
- Optimal substructure

DP vs Divide and Conquer

DP	Divide & Conquer
Stores results	No storage
Avoids recomputation	Recomputes
Bottom-up	Top-down

DP Applications

1. **Fibonacci Series**
2. **Matrix Chain Multiplication**
3. **0-1 Knapsack**

4. **Longest Common Subsequence (LCS)**
5. **Travelling Salesman Problem**
6. **Rod Cutting**
7. **Bin Packing**

Heuristics

- Approximate solutions
- Faster than exact algorithms
- Used in NP-hard problems

Characteristics

- Not always optimal
- Problem-specific
- Efficient

Application Domains

- Scheduling
- Routing
- AI problems

UNIT–5: Graph & Tree Algorithms

Graph Representation

1. **Adjacency Matrix**
2. **Adjacency List**

Traversal Algorithms

DFS

- Uses stack/recursion
- Depth-wise traversal

BFS

- Uses queue
- Level-wise traversal

Shortest Path Algorithms

Bellman-Ford

- Handles negative weights
- Time: O(VE)

Dijkstra’s Algorithm

- Greedy approach
- Uses priority queue
- Time: O(E log V)

Floyd-Warshall

- All-pairs shortest path
- Time: $O(n^3)$

Other Graph Algorithms

- **Transitive Closure**
- **Topological Sorting**
- **Network Flow (Ford-Fulkerson)**
- **Connected Components**

UNIT–6: NP Problems & Advanced Algorithms

Tractable vs Intractable Problems

- **Tractable**: Polynomial time

- **Intractable**: Exponential time

Computability

Determines whether a problem is solvable by algorithm.

Complexity Classes

- **P**: Solvable in polynomial time
- **NP**: Verifiable in polynomial time
- **NP-Complete**: Hardest problems in NP
- **NP-Hard**: At least as hard as NP-Complete

Cook's Theorem

- SAT is NP-Complete
- Foundation of NP-Completeness

Standard NP-Complete Problems

- Travelling Salesman
- Knapsack
- Vertex Cover
- Clique
- Hamiltonian Cycle

Reduction Techniques

Transform one problem to another in polynomial time.

Approximation Algorithms

- Near-optimal solutions
- Polynomial time

Randomized Algorithms

- Use random numbers
- Faster on average
- Example: Randomized Quick Sort