Design and Analysis of Algorithm

MOST IMPORTANT

UNIT 1

🔹 1. Algorithm and Order Notation and Time Complexity

✅ What is an Algorithm?

An algorithm is a **finite set of instructions** designed to solve a particular problem.
It should be:

- **Clear** and **unambiguous**
- **Finite** (must terminate)
- **Efficient**

⏳ Time Complexity

It tells us **how the runtime of an algorithm increases** with the size of input.

Types:

- **Best Case:** Minimum time (rarely used for analysis)
- **Worst Case:** Maximum time (safest for analysis)
- **Average Case:** Expected time (mathematically intensive)

🧠 Order Notations (Asymptotic Notations)

Used to express time complexity:

- **Big O (O)** → Worst case
  E.g., `O(n^2)` means time grows with square of input size.
- **Omega (Ω)** → Best case
- **Theta (Θ)** → Average/Exact case

Example:
For Linear Search:

- Best Case: Ω(1)
- Worst Case: O(n)

🔹 2. Divide and Conquer and How It Is Different from Greedy Method

⚔️ Divide and Conquer (D&C)

Idea: Break the problem → Solve recursively → Combine the results

Steps:

1. **Divide**: Break into subproblems
2. **Conquer**: Solve subproblems recursively
3. **Combine**: Merge the results

✅ Examples:

- Binary Search
- Merge Sort
- Quick Sort
- Matrix Multiplication (Strassen's Algorithm)

🍃 Greedy Method

Idea: Make the **locally optimal** choice at each step **hoping** for a global optimum.

✅ Examples:

- Kruskal's and Prim's MST
- Fractional Knapsack
- Huffman Coding

🔁 Difference Table:

Feature	Divide and Conquer	Greedy Method
Strategy	Divide → Solve → Combine	Choose best option at each step
Recursion	Usually recursive	Iterative or simple recursion
Solution Guarantee	Always optimal (if correct)	May or may not be optimal
Examples	Merge Sort, Binary Search	Kruskal, Prim, Dijkstra

🔹 3. Binary Search & Substitution Method

🔍 Binary Search

Used to find an element in a **sorted array**
Time Complexity: `O(log n)`

How it works:

- Compare mid element
- If target == mid → done
- If target < mid → search left half
- If target > mid → search right half

📈 Substitution Method (for solving recurrences)

Used in D&C to find time complexity

Example Recurrence for Binary Search:

r
CopyEdit
```
T(n) = T(n/2) + c
```

Using substitution:

- T(n) = T(n/2) + c
- = T(n/4) + c + c
- = T(n/8) + c + c + c

- ...
- = T(1) + c * log n → **O(log n)**

🔹 4. Kruskal and Prim's Algorithm + Comparison

Both are used for **Minimum Spanning Tree (MST)**

🌿 Kruskal's Algorithm

- Works on **edges**
- Sort edges → pick smallest edge that doesn't form cycle
- Uses **Disjoint Set (Union-Find)**

Time Complexity: `O(E log E)` (E = number of edges)

🌲 Prim's Algorithm

- Works on **vertices**
- Start with one vertex → grow MST by adding cheapest edge from visited to unvisited
- Uses **Priority Queue (Min Heap)**

Time Complexity: `O(E + log V)`

📊 Comparison Table:

Feature	Kruskal	Prim
Approach	Edge-based	Vertex-based
Data Structure	Disjoint Sets	Min Heap
Graph Type	Works well with sparse graphs	Works better for dense graphs
Cycle Handling	Avoids cycle using Union-Find	Doesn't form cycles naturally

---

🔹 5. Knapsack Problem (0/1 Knapsack)

🎒 Problem Statement:

Given weights w[ ] and profits p[ ] of n items, select items to **maximize profit** such that total weight ≤ W
0/1 version: You either take **whole item** or leave it (no fractions)

📋 **Recursive Formula:**

css
CopyEdit
```
if w[i] > W:
    dp[i][W] = dp[i-1][W]
else:
    dp[i][W] = max(dp[i-1][W], p[i] + dp[i-1][W - w[i]])
```

⌛ **Time Complexity:**

- Recursive: Exponential → O(2^n)
- Dynamic Programming: O(nW)

UNIT 2

🔹 1. Longest Common Subsequence (LCS) Algorithm

✅ **What is LCS?**

LCS is the **longest sequence that appears in the same relative order** in both strings, but not necessarily contiguously.

🧠 **Example:**

text
CopyEdit
```
X = "ACDBE"
Y = "ABCDE"
LCS = "ACDE"
```

📋 **Recurrence Relation:**

Let L[i][j] be the LCS length of first i characters of X and first j characters of Y.

lua
CopyEdit
```
if X[i] == Y[j]:
    L[i][j] = 1 + L[i-1][j-1]
```

```
else:
    L[i][j] = max(L[i-1][j], L[i][j-1])
```

⌛ **Time Complexity:**

- Recursive: Exponential
- Dynamic Programming: `O(m * n)` for strings of lengths m and n

🔹 2. Dynamic Programming (DP)

✅ What is DP?

Dynamic Programming is an algorithmic technique to solve problems by **breaking them into overlapping subproblems** and solving each **only once**, storing the result (memoization/tabulation).

✨ Features:

- Overlapping Subproblems
- Optimal Substructure

💡 Famous Examples:

- 0/1 Knapsack
- LCS
- Matrix Chain Multiplication
- Fibonacci Numbers
- Shortest Path (Floyd-Warshall)

Types of DP:

- **Top-down (Memoization)**
- **Bottom-up (Tabulation)**

🔹 3. Knapsack Problem

Already explained in Unit 1, but here it's treated more from a **DP approach**.

Type	Explanation
0/1 Knapsack	Either take item or leave it
Fractional Knapsack	Can take part of item (Greedy based)
DP Formula	`dp[i][w] = max(dp[i-1][w], value[i] + dp[i-1][w-weight[i]])`

UNIT 3

🔹 1. KMP (Knuth-Morris-Pratt) String Matching Algorithm

✅ **Goal:**

Find if a pattern exists in a text, efficiently.

❌ **Problem with Naive:**

Re-checks previous characters → Inefficient

💡 **KMP Idea:**

Uses a **prefix table (LPS array)** to skip rechecking characters.

👣 **Steps:**

1. Preprocess pattern → LPS (Longest Prefix Suffix)
2. Use LPS to shift pattern without unnecessary comparisons

📋 **Time Complexity:**

- LPS construction: `O(m)`
- Search: `O(n)`
- Total: `O(n + m)` where n = text length, m = pattern length

🔹 2. Naive String Matching

🧠 **Idea:**

Check pattern at every position in the text.

⌛ **Time Complexity:**

- Worst case: `O(m*n)`
- Best case: `O(n)`

Example:

Text: “ABABABC”
Pattern: “ABAB”
Naive tries from every position.

🔹 3. Randomized Algorithm

✅ **What is it?**

An algorithm that uses **random numbers** during computation.

⌛ **Why useful?**

Sometimes gives faster or simpler solutions on average.

🧠 **Types:**

- **Las Vegas**: Always correct, time may vary (e.g. QuickSort with random pivot)
- **Monte Carlo**: Time is fixed, answer may be incorrect (e.g. Primality tests)

🎯 **Applications:**

- 2-SAT Problem
- Primality Testing (Miller-Rabin)
- Pollard’s Rho Algorithm

🔹 4. P, NP, NP-Hard, NP-Complete

Class	Meaning
P	Solvable in polynomial time
NP	Verifiable in polynomial time
NP-Hard	As hard as the hardest problems in NP
NP-Complete	Both NP and NP-Hard

📍 **Key Examples:**

- P: Binary Search, Merge Sort
- NP: Sudoku verification
- NP-Complete: 0/1 Knapsack, Traveling Salesman
- NP-Hard: Halting Problem

🔹 5. Set Cover Problem

✅ **Problem:**

Given a **universe** of elements and a **set of subsets**, choose **minimum number of subsets** such that their union covers the universe.

❌ **Hardness:**

- NP-Complete
- Approximation algorithms exist (Greedy approach)

UNIT 4

🔹 1. Las Vegas vs. Monte Carlo Algorithm

🧠 **Difference:**

Feature	Las Vegas Algorithm	Monte Carlo Algorithm
Output	Always correct	Might be incorrect

Time	Varies (random)	Fixed
Example	Randomized Quick Sort	Miller-Rabin Primality Test
Use Case	When accuracy is critical	When speed is more important than accuracy

📝 **Summary:**

- **Las Vegas** = Random Time, Correct Answer
- **Monte Carlo** = Fixed Time, Possible Error

🔹 2. Randomized Algorithm for 2-SAT Problem

✅ **2-SAT Problem:**

Given a Boolean expression in CNF where each clause has **at most 2 literals**, determine if it's satisfiable.

💡 **Randomized Algorithm:**

- Papadimitriou's 2-SAT Algorithm is **randomized**.
- It selects a random assignment and flips a variable in any unsatisfied clause.
- Runs in `O(n^2 log n)` expected time.

✨ **Key Idea:**

- Repeats a limited number of times; if it finds a solution, it's valid.
- Uses randomness for quick resolution but may need retrying.

🔹 3. Vertex Cover Problem

✅ **Problem:**

Find a **minimum set of vertices** such that every edge in the graph is incident to at least one of the selected vertices.

📌 **Example:**

Graph with edges: (A-B), (B-C), (C-D)
One solution: Vertex cover = {B, C}

❌ **Complexity:**

- NP-Complete problem
- No polynomial-time exact solution known

✅ **Approximation Algorithm:**

- Pick any edge, add both vertices to cover, remove all incident edges
- Repeats until all edges are covered
- Time: `O(E)`
- Approximation ratio: 2

🔹 4. Set Cover Algorithm

Already explained in Unit 3 – remember:

- NP-Complete
- Approximation algorithm using **greedy** approach

UNIT 5

🔹 1. Proving a Problem is NP-Complete

✅ **Steps to Prove:**

1. Show problem is in **NP** → solution can be verified in poly time
2. Choose a known **NP-Complete** problem (like SAT, 3SAT, Clique)
3. Do a **Polynomial Time Reduction** from known NP-Complete problem to target problem

✍️ **Example:**

- Prove Vertex Cover is NP-Complete by reducing from 3-SAT or Clique

---

🔹 2. Vertex Cover Problem (detailed)

Already covered in Unit 4.

✔️ **Reminder:**

- Approximation ratio = 2
- Greedy algorithm gives near-optimal solution quickly

🔹 3. Set Cover Problem (detailed)

Already covered earlier – NP-Complete, Greedy Approximation

✅ Summary of Unit 5:

Topic	Type	Key Idea
NP-Completeness Proof	Core Theory	Reduction steps + logical flow
Vertex Cover	Applied NP-Comp.	With approximation
Set Cover	Applied NP-Comp.	Greedy, Approximation Ratio

STILL… IMPORTANT

UNIT 1

🔹 1. Algorithm Complexity

✅ What is it?

It measures **how efficiently an algorithm performs**, typically in terms of:

- **Time Complexity** (How fast)

- **Space Complexity** (How much memory)

📊 Order Notations:

Notation	Meaning	Example
O(n)	Worst-case	Linear Search
Ω(n)	Best-case	Best case of Bubble Sort
Θ(n)	Average case (tight bound)	Average case of Merge Sort

🧠 Why it matters:

Used to compare algorithms without running them. Helps in picking the most efficient one for large input sizes.

🔹 2. Divide and Conquer

🧩 Strategy:

Break the problem into **smaller sub-problems**, solve recursively, and **combine the results**.

🛠️ Steps:

1. **Divide** → Input into subproblems
2. **Conquer** → Solve subproblems recursively
3. **Combine** → Merge sub-results to form the solution

🔁 Examples:

- **Binary Search**: Divide the array by half each time → O(log n)
- **Merge Sort**: Split → Sort → Merge → O(n log n)
- **Quick Sort**: Choose pivot → Partition → Sort → O(n log n) on average

---

🔹 3. Greedy Approach

✅ Strategy:

At each step, **pick the locally best/optimal solution**, hoping it leads to the global optimum.

🔁 General Structure:

- Start with an empty solution
- At each step, choose the best available option
- Repeat until solution is complete

🧠 Characteristics:

- **No recursion**, unlike Divide & Conquer
- May not always lead to optimal global solution
- Used when **greedy-choice property** & **optimal substructure** exists

🧪 Example Problems:

- **Job Sequencing with Deadline**
- **Minimum Spanning Tree (Prim's, Kruskal's)**
- **Optimal Merge Pattern**
- **Fractional Knapsack (not 0/1)**

🔹 4. Binary, Quick, and Merge Sort

🔁 Binary Search:

- Search an element in **sorted array**
- Divide array in half, search left/right recursively
- Time Complexity: O(log n)

🔁 Merge Sort (Divide & Conquer):

- Divide array in half → sort both → merge
- Time: O(n log n)
- Stable sort

🔁 Quick Sort:

- Choose pivot → partition → recursively sort parts
- Average Time: O(n log n), Worst: $O(n^2)$
- Faster in practice than merge sort

🔹 5. Prim's Algorithm (for MST)

✅ Purpose:

Build a **Minimum Spanning Tree** from a graph.

🌳 How It Works:

- Start with any vertex
- Grow tree by **adding smallest edge** connecting a vertex **inside** the tree to a vertex **outside**

⏱️ Time Complexity:

- With Min-Heap & Adjacency List → O(E log V)

🔹 6. Knapsack Problem

🎒 Problem:

Given a set of items with **weights and values**, choose items to maximize value **without exceeding weight capacity**.

🧠 Types:

1. **0/1 Knapsack** – take or leave the item
   → Solved using **Dynamic Programming**

2. **Fractional Knapsack** – can take fractions
   → Solved using **Greedy Approach**

⏱️ Time:

- 0/1 Knapsack: $O(n \times W)$
- Fractional Knapsack: $O(n \log n)$

🔚 Final Recap Table – Final List 2, Unit 1

Topic	Method	Time Complexity	Type
Algorithm Complexity	–	–	Theory
Divide and Conquer	Recursive Strategy	Depends on algo	D&C
Greedy Approach	Step-by-step Best	Depends on algo	Greedy
Binary Search	D&C	$O(\log n)$	Search
Merge Sort	D&C	$O(n \log n)$	Sort
Quick Sort	D&C	Avg $O(n \log n)$	Sort
Prim's Algorithm	Greedy	$O(E \log V)$	Graph
Knapsack Problem	Greedy/DP	$O(n \log n)/O(n \times W)$	Optimization

---

UNIT 2

🔹 1. 0/1 Knapsack Problem (Dynamic Programming Approach)

Problem Statement:
Given weights and values of `n` items, put these items in a knapsack of capacity `W` to get the maximum total value.

> You can either **include the item or exclude it** (can't break it → 0/1 Knapsack)

🔢 **Example Input:**

- Items: `weights = [2, 3, 4, 5]`, `values = [3, 4, 5, 6]`, `capacity = 5`

📐 **Approach (DP Table):** Let `dp[i][w]` be the max value for first `i` items and weight `w`

plaintext
CopyEdit
```
if weight[i-1] > w
   dp[i][w] = dp[i-1][w]
else
   dp[i][w] = max(dp[i-1][w], value[i-1] + dp[i-1][w - weight[i-1]])
```

⏱️ **Time Complexity**: O(n × W)

📍 **Tip**: Memorize table approach and trace at least one numerical example!

🔹 2. Quadratic Assignment Problem (QAP)

📘 **What is it?**
It's a combinatorial optimization problem where:

- You assign **facilities to locations**
- Each assignment has a **flow cost × distance**

🧠 **Goal**:
Minimize the **total cost = Σ (flow × distance)**

🔢 **Example Applications**:

- Assigning jobs to machines
- Assigning departments in buildings

🧮 **Solving**: It's **NP-hard**, usually solved by **heuristics, genetic algorithms, or branch and bound**

🔹 3. Dynamic Programming Numerical Problems

These include:

- **Matrix Chain Multiplication**
- **LCS (Longest Common Subsequence)**
- **0/1 Knapsack**
- **All-Pairs Shortest Path (Floyd-Warshall)**

You should:

- Understand **how the table is built**
- Know the **recurrence relation**
- Be able to **trace a dry-run**

UNIT 3

🔹 1. Naïve String Matching Algorithm

💡 **Idea**: Slide pattern `P` over text `T` one by one and check character by character.

⏱️ **Time Complexity**:

- Best case: O(n)
- Worst case: O(mn), where `m` is pattern length, `n` is text length

---

🔁 **Example:**

text
CopyEdit
```
Text:     A B C D A B C D
Pattern:      A B C
```

You compare at every shift position.

📍 **Tip**: Easy to understand and write in exams.

🔹 2. KMP Algorithm (Knuth-Morris-Pratt)

💡 **Key Concept**: Avoid rechecking already matched characters using a **prefix table (LPS array)**.

📐 **LPS Array:** Stores the length of the **longest prefix that is also a suffix**.

Example: Pattern = `"ABABCABAB"`
LPS = `[0,0,1,2,0,1,2,3,4]`

Steps:

1. Build LPS array (O(m))
2. Use LPS to shift pattern intelligently (O(n))

⏱️ **Total Time**: O(m + n)

📍 **Benefit over Naïve**: Much faster for repeated patterns!

UNIT 4

🔹 1. Randomized Algorithm – Overview

A **randomized algorithm** uses a **random number** at least once during the computation to make decisions.
These are helpful for problems where **deterministic solutions are slow or unknown**.

There are **2 major types**:

Type	Characteristics	Result Accuracy	Time
Las Vegas	Always correct result, time may vary	✅ Always correct	⌛ Varies
Monte Carlo	May give incorrect result, time fixed	❌ Sometimes incorrect	⌛ Fixed

---

🔹 2. Las Vegas Algorithm

🔹 **Key Points:**

- Randomness used to decide how the algorithm proceeds
- **Output is always correct**
- Only **time taken** may vary

📘 **Example:**

- Randomized version of **Quicksort**:
    - Pivot is chosen randomly
    - Expected time: $O(n \log n)$, worst-case: $O(n^2)$
    - Still produces correct sorted array!

🧠 **Real-world analogy**:
Trying random keys on a door, you keep trying until the correct one is found.

🔹 3. Randomized Algorithm for 2-SAT Problem

2-SAT = Boolean satisfiability problem where each clause has **2 literals**
✅ Solvable in polynomial time

Randomized 2-SAT Algorithm:

- Start with a random assignment
- If any clause is unsatisfied, randomly flip one literal in it
- Repeat for $O(n^2)$ times

- With high probability → will find a satisfying assignment if one exists

Key Idea: Repeated random adjustments = better performance than brute force

---

UNIT 5

🔹 1. P, NP, NP-Hard, NP-Complete – Definitions

Class	Definition
P	Problems solvable in polynomial time (deterministically)
NP	Problems where **solution can be verified** in polynomial time
NP-Hard	At least as hard as the hardest problems in NP (may not be in NP)
NP-Complete	Problems in NP that are also NP-Hard

💡 **Key Relationships:**

- All problems in **P** are also in **NP**
- If one **NP-Complete** problem is solved in **P**, then **P = NP**
- Common NP-Complete problems:
  - **SAT**
  - **Hamiltonian Cycle**
  - **Vertex Cover**
  - **Subset Sum**

🔹 2. Vertex Cover Problem (VCP)

Definition:
Given a graph `G = (V, E)`, find the smallest set of vertices such that every edge has at least one endpoint in this set.

✅ Decision Version: "Is there a vertex cover of size ≤ k?"

Why Important?

- It's one of the **first problems proven NP-Complete**
- Common in **network security, social networks**

🔬 **Approximation Algorithm for VCP:**

- While edges remain:
  - Pick any edge (u, v)
  - Add both u and v to the cover
  - Remove all edges incident to u or v
- Guarantees a solution **≤ 2 × optimal**

⏱️ Time Complexity: O(E)

🔹 3. Set Cover Problem

Definition:
Given a **universe U** and a collection of subsets S1, S2, ..., Sn
→ Find the **smallest number of subsets** whose union equals U.

NP-Complete Problem

🔧 **Approximation Algorithm:**

- Greedy: Pick the subset that covers the most uncovered elements
- Repeat until all elements are covered

Performance Guarantee:

- Approximation ratio is **ln(n)**

📍 **Applications**: Wireless networks, sensor placement, data compression

STILL... IMPORTANT
UNIT 1

🔹 1. Algorithm Complexity and Order Notation

🔸 What is Algorithm Complexity?

It refers to the **resources (time & space)** used by an algorithm as a function of input size n.

Two Types:

- **Time Complexity**: How long the algorithm takes to run
- **Space Complexity**: How much memory it consumes

🔸 Order Notation (Asymptotic Notation):

Used to describe the **upper, lower, and average bounds** of algorithm complexity:

Notation	Meaning	Example (Binary Search)
$O(f(n))$	Upper Bound (Worst Case)	$O(\log n)$
$\Omega(f(n))$	Lower Bound (Best Case)	$\Omega(1)$
$\Theta(f(n))$	Tight Bound (Average Case)	$\Theta(\log n)$

🔸 Common Time Complexities:

Time Complexity	Description	Example
$O(1)$	Constant	Accessing an array element
$O(\log n)$	Logarithmic	Binary Search
$O(n)$	Linear	Linear Search
$O(n \log n)$	Linearithmic	Merge Sort, Quick Sort avg.
$O(n^2)$	Quadratic	Bubble Sort, Selection Sort

🔹 2. Divide and Conquer (D&C)

🔸 Idea:

Break the problem into **sub-problems**, solve them **recursively**, then **combine** the results.

Steps:

1. Divide – Break into subproblems
2. Conquer – Solve subproblems recursively
3. Combine – Merge results

🔸 Examples:

Problem	Time Complexity
Merge Sort	O(n log n)
Quick Sort	O(n log n) avg.
Binary Search	O(log n)
Strassen's Matrix Multiplication	O(n^2.81)

🔹 3. Comparison: Kruskal vs Prim Algorithm

Both are **Greedy Algorithms** used to find a **Minimum Spanning Tree (MST)**.

Feature	Kruskal's Algorithm	Prim's Algorithm
Approach	Edge-based	Vertex-based
Data Structure Used	Disjoint Set (Union-Find)	Priority Queue (Min-Heap)
Graph Type	Works well with **sparse** graphs	Works well with **dense** graphs
Sorting Required	Yes, edges are sorted	No sorting required
Time Complexity	O(E log E) or O(E log V)	$O(V^2)$ or O(E + log V) with heap
Cycle Check	Yes (using Union-Find)	Not needed (only adds safe vertices)

🔹 4. Binary Search and Substitution Method

🔸 Binary Search:

Searches for an element in a **sorted array** by dividing the search space in half repeatedly.

Working:

1. Find middle: `mid = (low + high) / 2`
2. If `key == a[mid]` → found
3. If `key < a[mid]` → search left half
4. Else → search right half

🔸 Time Complexity:

- **Worst Case**: O(log n)
- **Best Case**: O(1)
- **Space**: O(1) (iterative), O(log n) (recursive)

🔸 Substitution Method (for solving recurrences)

Used to prove time complexities of recursive algorithms.

Binary Search Recurrence:

r
CopyEdit

```
T(n) = T(n/2) + c
```

Substitution (guess and prove):

Assume: `T(n) = O(log n)`

1. Plug in the guess: `T(n) = T(n/2) + c` → `= T(n/4) + 2c` → `= T(n/8) + 3c` → … → `= T(n/2^k) + kc`

2. When $n/2^k = 1$ → $k = \log n$ → $T(n) = T(1) + c \log n = O(\log n)$ ✅

🔁 Summary Chart

Topic	Key Idea	Complexity
Algorithm Complexity	Time & space usage analysis	O(1) to O(n²+)
Order Notation	Describes bounds	O, Ω, Θ
Divide and Conquer	Break → Solve → Combine	O(log n) to O(n log n)
Kruskal vs Prim (MST)	Edge vs Vertex based greedy method	Kruskal: O(E log V)
Binary Search	Logarithmic search in sorted array	O(log n)
Substitution Method	Solve recurrence relations	Prove O(log n)

UNIT 2

🔹 1. Longest Common Subsequence (LCS)

🔸 What is it?

LCS is the longest sequence that appears **in the same order** in both strings but **not necessarily contiguous**.

👉 Example:
String X = `ABCBDAB`
String Y = `BDCABA`
LCS = BCBA (length = 4)

🔸 Approach: Dynamic Programming

Let `X[1..m]` and `Y[1..n]` be two strings.

We define:

perl
CopyEdit
```
L[i][j] = length of LCS of X[1..i] and Y[1..j]
```

🔸 Recurrence Relation:

sql
CopyEdit
```
If X[i] == Y[j]:
    L[i][j] = 1 + L[i-1][j-1]
Else:
    L[i][j] = max(L[i-1][j], L[i][j-1])
```

🔸 Base Case:

CopyEdit
```
L[0][j] = 0 and L[i][0] = 0
```

🔸 Time Complexity:

- **Time**: O(m * n)
- **Space**: O(m * n)

🔹 2. Dynamic Programming Numerical (Knapsack Example)

🔸 0/1 Knapsack Problem

You are given:

- `n` items
- `w[i]` = weight of item i
- `v[i]` = value of item i
- A bag of capacity `W`

🔸 Goal:

Maximize total value such that total weight ≤ `W`

🔸 **Recurrence Relation:**

Let `dp[i][w]` = max value using first `i` items and weight `w`.

css
CopyEdit
```
If w[i] > w:
    dp[i][w] = dp[i-1][w]
Else:
    dp[i][w] = max(dp[i-1][w], v[i] + dp[i-1][w - w[i]])
```

🔸 **Time Complexity:**

- **Time**: O(n * W)
- **Space**: O(n * W)

🔹 3. Quantitative Assignment Problem

This is a **special type of optimization problem**: assign `n` agents to `n` tasks so that the total cost is minimized.

🔸 **Given:**

A cost matrix `C[i][j]` where:

- `i` = agent
- `j` = task
- `C[i][j]` = cost for agent `i` to do task `j`

🔸 **Method:**

Hungarian Algorithm (most commonly used)

🔸 Steps (Simplified):

1. Row reduction
2. Column reduction

3. Cover all 0s with minimum number of lines
4. Modify matrix if not enough lines
5. Repeat until assignment possible

UNIT 3

🔹 1. KMP (Knuth-Morris-Pratt) String Matching

Used to search a pattern in a text in **linear time**, unlike the naive approach.

🔸 **Naive:**

- Compare every index: O(n * m)

🔸 **KMP:**

- Uses a **prefix table (LPS array)** to avoid rechecking characters
- Preprocessing LPS takes O(m)
- Searching takes O(n)

🔸 Total Time:

O(n + m)

🔸 **Example:**

Pattern: `ABABCABAB`
LPS: `0 0 1 2 0 1 2 3 4`

🔹 2. KMP vs Naive Approach:

Feature	Naive	KMP
Time Complexity	O(n*m)	O(n + m)
Extra Space	None	LPS array: O(m)
Backtracking	Yes	No
Efficient on	Short patterns	Long patterns

🔹 3. Las Vegas vs Monte Carlo Algorithm

Feature	Las Vegas	Monte Carlo
Output Correctness	Always correct	May be incorrect
Time Complexity	Varies (random)	Fixed time (bounded)
Example	Randomized Quick Sort	Miller-Rabin Primality Test

🔹 4. Randomized Algorithm for 2-SAT Problem

🔸 2-SAT Problem:

Given boolean formula in CNF with **2 literals per clause**, find if there's a satisfying assignment.

🔸 Randomized Algorithm (Papadimitriou's):

- Randomly assign values to variables
- Repeat:
  - If satisfied: return result
  - If not: flip one variable from an unsatisfied clause
- Repeat up to **O(n^2)** steps

✅ Works with high probability in **polynomial time**

🧠 Summary Table:

Topic	Technique	Time Complexity
LCS	DP	O(m * n)
Knapsack	DP	O(n * W)
Assignment Problem	Hungarian Algo	O(n^3)
KMP Algorithm	String Matching	O(n + m)

Naive vs KMP	Efficiency Check	O(nm) vs O(n+m)
Las Vegas vs Monte Carlo	Randomized Algo	Probabilistic
2-SAT Randomized Algo	Randomized	$O(n^2)$ expected

UNIT 4

🔹 1. P, NP, NP-Hard, NP-Complete

✅ **Definitions:**

Class	Meaning
P	Problems solvable in **polynomial time**. Eg: Binary Search, Merge Sort
NP	Problems **verifiable** in polynomial time. Eg: Sudoku, Hamiltonian Path
NP-Hard	At least as hard as NP problems. **Not necessarily in NP**
NP-Complete	Problems in **NP** and as hard as any NP problem (NP-Hard). Eg: SAT, Vertex Cover

🔸 **Hierarchy Diagram:**

markdown
CopyEdit

```
P ⊆ NP ⊆ NP-Hard
       ↳ NP-Complete = NP ∩ NP-Hard
```

🔹 2. Proving Vertex Cover is NP-Complete

✅ **Vertex Cover Problem:**

> Given a graph `G=(V,E)` and integer `k`, is there a set of `k` vertices such that every edge has at least one endpoint in this set?

🧠 **To prove it's NP-Complete:**

1. **In NP**:
   Given a set of vertices, we can check in polynomial time if it covers all edges.

2. **NP-Hard**:
   Reduce a known NP-Complete problem to Vertex Cover.

🔸 **Common Reduction: From 3-SAT or Clique problem**

Example:

- Reduce CLIQUE to VERTEX COVER using complement graph

🔹 3. Set Cover Problem & Approximation Algorithm

✅ **Problem:**

Given:

- A universe `U = {1,2,…,n}`
- A collection of subsets `S1, S2,…, Sm` Find the **minimum number of subsets** that cover all elements in `U`.

🔸 **NP-Hard**

🔸 **Greedy Approximation Algorithm:**

- At each step, pick the subset that covers the **most uncovered elements**.
- Repeat until all elements are covered.

🔸 **Approximation Ratio:**

- **ln(n)**, where n = number of elements in the universe

UNIT 5

🔹 1. Approximation Algorithms

Used for solving **NP-Hard** problems quickly by giving **"good enough"** solutions.

🔸 **Commonly used for:**

- Vertex Cover

- Set Cover
- TSP
- Knapsack

🔸 **Example: Approximate Vertex Cover**

text
CopyEdit

```
Greedy Algorithm:
1. Pick an edge (u, v)
2. Add both u and v to the cover
3. Remove all edges incident to u or v
4. Repeat
```

🔸 **Approximation Ratio:**

≤ 2 × Optimal

🔹 2. Las Vegas vs Monte Carlo Algorithms (with example)

Feature	Las Vegas	Monte Carlo
Correctness	Always correct	May be incorrect
Runtime	Variable	Fixed
Example	Randomized Quicksort	Miller-Rabin Primality Test

🔸 **Las Vegas Example: Randomized Quicksort**

- Chooses a pivot randomly
- Worst case: $O(n^2)$, but expected time: $O(n \log n)$

🔸 **Monte Carlo Example:**

- Miller-Rabin: Says if a number is “probably prime” with a small error probability

---

🔹 3. Randomized Algorithm for 2-SAT Problem

✅ **Problem:**

Boolean formula with **2 literals per clause**
Find an assignment of variables that satisfies all clauses.

🔸 **Randomized Algorithm (Papadimitriou's 1991):**

1. Start with a **random assignment**
2. While clause is unsatisfied:
   - Pick a random clause that is false
   - Flip one variable in that clause
3. Repeat for up to `O(n²)` steps

🔸 **Success Probability:**

- With enough repetitions, high chance of finding a satisfying assignment (if one exists)

🧠 Summary Table:

Topic	Type	Strategy/Algorithm	Complexity
P, NP, NP-Hard, NP-Complete	Theory	Classification	—
Vertex Cover NP-Complete Proof	Theoretical	Reduction from Clique or 3-SAT	—
Set Cover Approximation	Greedy	Covers max uncovered at each step	$O(n \log n)$
Approximation for Vertex Cover	Greedy	Pick both ends of uncovered edge	$O(E)$
Las Vegas Algorithm	Randomized	Always correct (e.g., RQS)	Varies
Monte Carlo Algorithm	Randomized	May be wrong (e.g., Miller-Rabin)	Fast
2-SAT Randomized Algorithm	Randomized	Variable flipping	$O(n^2)$