# MODULE 2: ALGORITHMIC THINKING

# Session 1: Algorithmic Thinking 2

## Problem Solving Approaches and Heuristics

# Problem Solving Approaches and Heuristics

Problem Solving Approaches and Heuristics are strategies and techniques used to find solutions to complex problems efficiently. In algorithmic thinking, these methods help in devising algorithms that solve problems effectively

# Problem Solving Approaches

## 1. Divide and Conquer

Divide and Conquer is a strategy where a problem is divided into smaller, more manageable sub-problems, each of which is solved independently. The solutions to these sub-problems are then combined to solve the original problem.

# Divide and Conquer Cont…..

## Steps:

- Divide: Break the problem into smaller subproblems that are easier to handle.
- Conquer: Solve each sub-problem recursively.
- Combine: Merge the solutions of the sub-problems to get the solution to the original problem.

## Example: Merge Sort

- Problem: Sort an array of numbers.
- Solution:
  - Divide: Split the array into two halves.
  - Conquer: Recursively sort each half.
  - Combine: Merge the sorted halves into a single sorted array.

# 2. Greedy Algorithms

Greedy Algorithms make the locally optimal choice at each step with the hope of finding a global optimum. They work by making decisions that seem best at the moment, without considering the overall problem.

# Greedy Algorithms Cont….

## Steps:

- Select: Choose the best option available at each step.
- Evaluate: Check if this choice leads to a valid solution.
- Repeat: Continue until a complete solution is achieved.

**Example: Kruskal's Algorithm**

- Problem: Find the Minimum Spanning Tree (MST) of a graph.
- Solution:
  - Select: Sort all edges by weight.
  - Evaluate: Add the smallest edge to the MST if it doesn't form a cycle.
  - Repeat: Continue until all vertices are connected.

# 3. Dynamic programming

Dynamic Programming (DP): solves problems by breaking them into simpler overlapping subproblems and storing the results of these sub-problems to avoid redundant calculations

# Dynamic programming Cont….

Steps:

- Define Sub-Problems: Break down the problem into simpler, overlapping subproblems.
- Store Results: Save solutions to sub-problems in a table (or array).
- Build Solution: Use stored results to build up the solution to the original problem.

# Dynamic programming Cont….

Example: Fibonacci Sequence

- Problem: Compute the nth Fibonacci number.
- Solution:
  - Define Sub-Problems: Calculate Fibonacci numbers for smaller values.
  - Store Results: Use a table to store Fibonacci numbers as they are computed.
  - Build Solution: Use the stored results to compute higher Fibonacci numbers.

# 4. Back tracking

Backtracking involves incrementally building a solution and abandoning it as soon as it is determined that the current path cannot be extended to a valid solution. It explores all potential solutions and reverts when a solution fails.

# Back tracking Cont….

## Steps:

- Build Solution: Add elements incrementally to build a partial solution.
- Check Validity: Verify if the current solution is valid.
- Backtrack: If the current solution is invalid, remove the last added element and try a different option.

**Example: Sudoku Solver**

- Problem: Solve a Sudoku puzzle by filling in the grid.
- Solution:
  - Build Solution: Place numbers in empty cells.
  - Check Validity: Ensure no numbers violate Sudoku rules.
  - Backtrack: If a number placement leads to a conflict, remove it and try another number.

# Problem Solving Heuristics

## 1. Problem Simplification

Simplify the problem to a more manageable version, solve that, and then extend the solution to the original problem.

# 2. Analogies and Pattern Matching

Apply solutions from similar problems or recognize recurring patterns in the current problem.

# 3. Working Backward

Start from the goal and work in reverse to figure out the steps needed to reach it.

**Example**: In mathematical proofs or maze solving, work from the solution back to the start.

# 4. Decomposition

Divide a complex problem into smaller, more manageable parts, solve each part, and then integrate the solutions.

**Example**: Breaking a software project into modules or components.

# 5. Trial and Error

Experiment with possible solutions, learn from failures, and iterate until you find a working solution.

# Example: Scheduling Tasks with Deadlines and Penalties

Imagine you are given a set of tasks, each with a specific deadline and a penalty associated if it is not completed by that deadline. The goal is to determine the optimal order in which to complete the tasks to minimize the total penalty.

# For Example:

| Task | Deadline | Penalty |
|---|---|---|
| A | 2 days | $ 200 |
| B | 1 day | $ 500 |
| C | 3 days | $ 100 |

# Approach: Greedy Heuristic

Prioritize tasks based on a cost-benefit ratio, such as minimizing penalties. In this scenario, the heuristic would focus on completing tasks with the highest penalties first if their deadlines allow.

# Why Greedy ?

The greedy approach works well here because by always choosing the task with the highest penalty first (if possible), we minimize the overall penalty.

This heuristic works because the most expensive tasks have a higher cost of delay, so completing them earlier reduces potential penalties.

# Activity

You are tasked with designing and implementing sorting algorithms using a set of numbered index cards, focusing on understanding different algorithmic techniques and how they work in practice.

PAUSE 5 mins

- This marks the end of this lesson.
- In the next lesson we will learn about Mathematical Foundations of Algorithms

**THANK YOU**