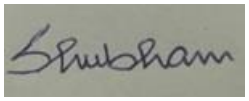
Department of MBA

Details of Lesson Plan

S.No.	Particulars	Details
1.	Course Name	Advanced Data Structures and Algorithms
2.	Course Code	MCSE103
3.	Academic Year	2024-2025
4.	Semester	1st
5.	Number of Lesson plans	34
6.	Faculty Assigned	Mr. Shubham Gupta



Faculty Signature

Lesson Plan No. 1.1	Course Name: Advanced Data Structures and Algorithms Topic: Role of Algorithms in Computing	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Define and explain the role of algorithms in solving computational problems. b. Analyze the importance of algorithm efficiency in real-world applications. c. Differentiate between algorithm design paradigms (e.g., divide-and-conquer, dynamic programming, greedy algorithms). d. Apply algorithmic thinking to a simple computational problem.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Ask questions. "What are some everyday tasks where you think algorithms play a role?" "How do you think search engines rank results or your social media feed is organized?" - Transition to explaining the role of algorithms in computing by highlighting their necessity in automating tasks and solving problems efficiently. - Introduce a formal definition: "An algorithm is a step-by-step procedure for solving a problem or accomplishing a task in finite time." 2. **Development** (30 minutes) a. Algorithms - Explain the significance of algorithms in determining software performance, scalability, and reliability. b. Algorithm Efficiency - Discuss Big-O notation and its importance in evaluating algorithm performance. - Illustrate with examples: Comparing insertion sort ($O(n^2)$) vs. merge sort ($O(n \log n)$). c. Algorithm Design Paradigms - Introduce key paradigms with examples - Divide-and-Conquer: Merge Sort. - Dynamic Programming: Shortest path problem using Dijkstra's algorithm. - Greedy Algorithms: Huffman encoding.

	3. Exercise (5 minutes) – a. "You are given a list of tasks with deadlines. How would you schedule them to maximize efficiency?" Use Nearpod to collect responses and discuss the answers.
	1. Summarize the Lesson Learning Outcomes - Importance and efficiency of algorithms. - Applications in solving real-world problems. - Introduction to various algorithm paradigms. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions - "Why is it important to consider efficiency when designing algorithms?" - "How does choosing the right algorithm affect computational resources?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 1.2	Course Name: Advanced Data Structures and Algorithms Topic: Algorithms as a Technology and Insertion Sort	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Explain the concept of algorithms as a technology and their role in solving computational problems. b. Understand the working of the Insertion Sort algorithm. c. Analyze the time complexity of Insertion Sort in best, average, and worst-case scenarios. d. Implement Insertion Sort in a programming language.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Ask questions. "What comes to your mind when you hear the term 'algorithm'?" "Can you name a scenario where algorithms are vital, such as in online shopping or banking? - Define algorithms as a systematic approach to solving problems using a sequence of well-defined steps. - Highlight the significance of algorithms in technology, from sorting and searching data to powering AI systems. 2. **Development** (30 minutes) a. Algorithms as a Technology - Explain how algorithms form the backbone of computational thinking. - Use examples to show how algorithms impact various fields - Emphasize efficiency: Discuss trade-offs between time and space complexity. b. Insertion Sort Algorithm - Define Insertion Sort as a comparison-based sorting algorithm. - Discuss how it mimics the way humans sort playing cards. - Time Complexity Analysis 3. Exercise (5 minutes) – a. Ask students to sort an array [8, 2, 5, 9, 1, 4] using Insertion Sort and calculate the total number of comparisons. b. Group activity: Modify Insertion Sort to count the number of swaps during execution.

	Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Importance of algorithms as a technology. - Detailed understanding of Insertion Sort and its efficiency. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions - "What makes Insertion Sort inefficient for large datasets?" - "How would you decide when to use Insertion Sort?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 1.3	Course Name: Advanced Data Structures and Algorithms Topic: Analyzing Algorithms	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the need for algorithm analysis and its role in computational problem-solving. b. Define and apply key metrics like time complexity and space complexity. c. Explain and compute asymptotic notations (Big-O, Big-Theta, Big-Omega). d. Compare and analyze the efficiency of different algorithms for the same problem.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Ask questions. "Why do we need to analyze algorithms?" "What makes one algorithm better than another for the same problem?" - Discuss practical scenarios where choosing an efficient algorithm makes a significant difference, such as sorting millions of records or optimizing a delivery route. - Define Algorithm Analysis: The process of determining the computational efficiency of an algorithm in terms of time and space. 2. **Development** (30 minutes) a. Importance of Analyzing Algorithms - Explain why analysis is critical for: - Resource optimization. - Scalability in large systems. - Practical feasibility in real-world applications. b. Metrics for Algorithm Analysis - Time Complexity: - Show examples: Linear search (O(n)), Binary search (O(log n)). - Space Complexity - Use examples to highlight memory trade-offs. c. Asymptotic Notations - Introduce key notations: - Big-O (Worst Case): Upper bound. - Big-Theta (Average Case): Tight bound. - Big-Omega (Best Case): Lower bound.

	- Illustrate with Examples: Compare bubble sort ($O(n^2)$) and merge sort ($O(n \log n)$). - Use graphs to visualize the growth rates of common functions. 3. Exercise (5 minutes) – a. Ask students to compute the time complexity of searching algorithms and discuss their answers. Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Importance of analyzing algorithms. - Key metrics (time and space complexity). - Understanding and application of asymptotic notations. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions - "Why is Big-O notation widely used in computer science?" - "How does space complexity influence algorithm choice in constrained environments?" 2. Compare two sorting algorithms (e.g., insertion sort and quicksort) and discuss their efficiency for large datasets. 3. Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 1.4	Course Name: Advanced Data Structures and Algorithms Topic: Designing Algorithms	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Define the process of algorithm design and its importance. b. Explain key steps in designing efficient algorithms. c. Understand and apply algorithm design paradigms (e.g., divide-and-conquer, greedy, dynamic programming). d. Develop a solution for a real-world problem using systematic algorithm design.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Ask questions. "How do you approach solving a complex problem in your daily life?" "Can we think of step-by-step instructions that guarantee success in solving computational problems?" - Transition to the importance of algorithm design in creating scalable, efficient solutions. - Define algorithm design as "the art of crafting step-by-step solutions to computational problems." 2. **Development** (30 minutes) a. Steps in Designing Algorithms - Understand the Problem: Define input, output, and constraints. - Choose a Paradigm: Select a suitable design strategy (e.g., divide-and-conquer). - Formulate the Algorithm: Create pseudocode or flowcharts. - Analyze Complexity: Evaluate time and space complexity. - Test and Debug: Use test cases to validate the algorithm b. Algorithm Design Paradigms - Divide-and-Conquer - Dynamic Programming - Greedy Approach - Emphasize the importance of selecting the right paradigm based on problem characteristics. c. Real-Life Applications of Algorithm Design - Pathfinding in navigation apps (Dijkstra's algorithm). - Optimizing delivery routes (Traveling Salesman Problem). - Data compression (Huffman encoding).

	3. Exercise (5 minutes) – a. "Design an algorithm to find the largest sum of a contiguous subarray." b. Encourage students to explore divide-and-conquer (Kadane's Algorithm) or brute force approaches. Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Key steps in algorithm design. - Overview of design paradigms and their relevance. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions - "Why is choosing the right design paradigm crucial for problem-solving?" - "How does analyzing complexity help in evaluating an algorithm?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 1.5	Course Name: Advanced Data Structures and Algorithms Topic: Growth of Functions: Asymptotic Notation	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Define and explain asymptotic notations (Big-O, Big-Ω, Big-Θ). b. Analyze and compare the growth of functions. c. Apply asymptotic notations to evaluate algorithm efficiency
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Ask questions. "How do we compare two algorithms solving the same problem?" "What do we mean by the 'efficiency' of an algorithm?" - Introduce the idea of **growth of functions** and its relevance in analyzing algorithm performance. 2. **Development** (30 minutes) a. Overview of Asymptotic Notation - Define and explain: - Big-O Notation: Upper bound (e.g., $O(n^2)$). - Big-Ω Notation: Lower bound (e.g., $\Omega(n)$). - Big-Θ Notation: Tight bound (e.g., $\Theta(n \log n)$). - Discuss common complexities: O(1), O(log n), O(n), O(n log n), $O(n^2)$. b. Graphical Representation - Use graphs to compare common growth rates. - Highlight differences between O(n) vs. $O(n^2)$, O(log n) vs. O(n log n). c. Examples and Applications - Provide examples of algorithms: - Linear Search: O(n). - Binary Search: O(log n). - Sorting Algorithms: O(n log n) or $O(n^2)$. - Demonstrate analyzing a simple algorithm for its time complexity. 3. Exercise (5 minutes) – a. Give problems to identify asymptotic notation: b. Example: Analyze the function $T(n) = 5n^2 + 3n + 2$. c. Group activity: Compare two functions and determine which grows faster.

	Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Importance of asymptotic notation. - Recognizing algorithm efficiency using growth rates. - Real-world relevance in choosing better algorithms. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions - "Why is Big-O used more commonly than Big-Θ or Big-Ω?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 1.6	Course Name: Advanced Data Structures and Algorithms Topic: Standard Notations and Common Functions	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Define standard notations used in algorithm analysis (e.g., Big-O, Ω, and Θ). b. Explain and identify common functions in computational analysis (e.g., polynomial, exponential, logarithmic). c. Apply standard notations to evaluate algorithm efficiency.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Ask questions. "What do you think is more efficient: an algorithm that runs in $O(n^2)$ or one that runs in $O(n \log n)$?" - Introduce the purpose of standard notations 2. **Development** (30 minutes) a. Standard Notations - Define key notations with simple examples: - Big-O (O): Upper bound (e.g., Bubble Sort is $O(n^2)$). - Big-Ω (Ω): Lower bound (e.g., Linear Search is Ω(1) in best-case). - Big-Θ (Θ): Tight bound (e.g., Merge Sort is $\Theta(n \log n)$). - Illustrate differences with a graph (compare $O(n)$, $O(n^2)$, and $O(\log n)$). b. Common Functions - Explain common growth-rate functions with examples: - Constant: $O(1)$ → Accessing an array element. - Logarithmic: $O(\log n)$ → Binary Search. - Linear: $O(n)$ → Traversing a list. - Quadratic: $O(n^2)$ → Nested loops (Matrix Multiplication). - Exponential: $O(2^n)$ → Recursive solutions like the Tower of Hanoi. c. Application of Notations - Show how to analyze simple algorithms using these notations (e.g., a loop running n times is $O(n)$) 3. Exercise (5 minutes) – a. Give problems to identify asymptotic notation: b. Group activity: Compare two functions and determine which grows faster.

	Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Importance of standard notations in analyzing algorithms. - Common growth-rate functions and their real-world relevance. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions - "Why is it crucial to understand the growth rates of algorithms in large-scale systems?" 2. Analyze the time complexity of an algorithm and graph its growth rate Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 1.7	Course Name: Advanced Data Structures and Algorithms Topic: Recurrences - The Substitution Method	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Define the substitution method for solving recurrences. b. Apply the substitution method to prove time complexity. c. Solve example recurrence relations using the substitution method.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Define a recurrence relation: - "A recurrence is an equation or inequality describing a function in terms of its value at smaller inputs." - Explain why solving recurrences is essential (e.g., analyzing recursive algorithms like Merge Sort or Quick Sort). - Briefly introduce the substitution method: - "A technique where we guess the solution form and prove it using mathematical induction." 2. **Development** (30 minutes) a. Steps of the Substitution Method - Guess the solution (form of T(n)). - Prove the guess by induction: - Base Case: Show that the relation holds for small values (e.g., n = 1). - Inductive Step: Assume the guess holds for n = k, and prove it for n = k + 1. 3. Exercise (5 minutes) – a. Students solve T(n)=T(n−1)+nT(n) = T(n-1) + nT(n)=T(n−1)+n using the substitution method. Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Substitution is about guessing and proving solutions via induction. - Practice ensures better intuition for guessing solutions. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Solve T(n)=3T(n/3)+nT(n) = 3T(n/3) + nT(n)=3T(n/3)+n.

	Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 1.8	Course Name: Advanced Data Structures and Algorithms Topic: The Recursion Tree Method	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the concept of the recursion tree method for analyzing recursive algorithms. b. Visualize the structure of a recursion tree to determine time complexity. c. Apply the recursion tree method to solve computational problems. d. Compare the recursion tree method with other techniques like the Master Theorem and Substitution Method.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Define a recurrence relation: - "What is a recursive algorithm? Can you give some examples from real life or programming?" - "How do you think we analyze the efficiency of recursive algorithms?" Transition: Explain that the recursion tree method provides a visual way to break down and analyze recursive algorithms' time complexity. 2. **Development** (30 minutes) a. Concept Overview - Define the recursion tree method: A technique to represent recurrence relations visually to analyze time complexity. - Highlight its utility in breaking down problems like T(n) = T(n/2) + O(n) into simpler parts. b. Building the Recursion Tree - Start from the root node representing the original problem. - Break down the problem into subproblems (children of the node). - Continue until reaching the base case. - Add costs at each level to find total cost. c. Comparison with Other Methods - The Master Theorem (for divide-and-conquer recurrences). - The Substitution Method (plugging guesses and proving by induction). d. Applications in Real-World Problems - Merge Sort (recursive divide-and-conquer). - Binary Search Tree operations.

	3. Exercise (5 minutes) – a. Solve the recurrence $T(n) = 3T(n/3) + O(n)$ using the recursion tree method. Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - The recursion tree method simplifies analyzing recursive algorithms. - Real-world relevance in understanding algorithms like divide-and-conquer. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Solve $T(n)=3T(n/3)+nT(n) = 3T(n/3) + nT(n)=3T(n/3)+n$. Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 2.1	Course Name: Advanced Data Structures and Algorithms Topic: Hierarchical Data Structures: Binary Search Trees	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Define Binary Search Trees and explain their hierarchical structure. b. Analyze the properties and operations of BSTs (e.g., insertion, deletion, and traversal). c. Evaluate the efficiency of BST operations using Big-O notation. d. Apply BST concepts to solve computational problems (e.g., searching and sorting).
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Define a recurrence relation: - "How do you think hierarchical structures are used in organizing data like a dictionary or phonebook?" - "What are some real-world examples of search problems?" - "A Binary Search Tree is a hierarchical data structure where each node has at most two children, and all nodes follow the property: left child < parent < right child." - "A Binary Search Tree is a hierarchical data structure where each node has at most two children, and all nodes follow the property: left child < parent < right child." 2. **Development** (30 minutes) a. Properties of BST - Structure and rules of BST. - Visualize BST with an example. b. Operations on BST - Insertion: Step-by-step explanation with a sample tree. - Deletion: Three cases (leaf node, one child, two children). - Traversal: Preorder, Inorder, Postorder, and their applications. c. Efficiency Analysis - Search, Insert: O(h), where h is the height of the tree - Worst case: O(n)O(n)O(n) (unbalanced tree), Best case: O(logn)O(\log n)O(logn) (balanced tree) d. Applications of BST - Searching and sorting applications. - Use in databases and indexing systems. 3. Exercise (5 minutes) – a. "Given a sequence of numbers [15, 10, 20, 8, 12, 18, 25], build

	a BST and show the Inorder traversal." Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Definition and importance of BSTs. - Efficiency in searching and sorting. - Applications in real-world problems. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "Why is the balance of a BST crucial for efficiency?" - "What are the limitations of a BST, and how can they be addressed?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 2.2	Course Name: Advanced Data Structures and Algorithms Topic: Basics and Querying a Binary Search Tree	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Define a Binary Search Tree (BST) and its properties. b. Understand and explain the fundamental operations of a BST (insertion, deletion, and searching). c. Analyze the time complexity of BST operations. d. Apply BST concepts to solve real-world computational problems.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Start with real-life scenarios: - "Have you ever searched for a book in a library organized by title or author? How did you quickly locate the book?" - Relate this to the concept of ordered data and introduce Binary Search Trees. - Define BST: "A Binary Search Tree is a hierarchical data structure where each node has at most two children, and the left child contains values less than the parent, while the right child contains values greater than the parent." 2. **Development** (30 minutes) a. Basics of BST - Explain the structure and properties of BST using diagrams. - Illustrate examples of BSTs (both valid and invalid trees). b. Querying Operations in BST - Search Operation: - Explain the step-by-step process using pseudocode. - Discuss best-case, average-case, and worst-case scenarios. - Insertion Operation: - Demonstrate how to insert elements into a BST. - Use visuals to show the incremental building of a tree. - Deletion Operation: - Discuss three cases: - Node with no children. - Node with one child. - Node with two children. c. Efficiency of BST Operations - Introduce the time complexity: O(h), where h is the height of the tree. - Highlight the need for balanced trees to maintain efficiency.

	3. Exercise (5 minutes) – a. "What happens to the efficiency of BST operations when the tree becomes unbalanced?" b. "Can you think of scenarios where BSTs are not an ideal data structure?" Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Definition and structure of BST. - Importance of BST in querying and organizing data. - Efficiency and challenges of using BSTs in real-world applications. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "Why is the balance of a BST crucial for efficiency?" - "What are the limitations of a BST, and how can they be addressed?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 2.3	Course Name: Advanced Data Structures and Algorithms Topic: Properties of Red-Black Trees	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the structure and significance of Red-Black Trees in maintaining balanced binary search trees. b. Explain the properties that define a Red-Black Tree. c. Relate the properties to how Red-Black Trees ensure logarithmic height for efficient operations. d. Analyze the time complexity of operations in Red-Black Trees.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Ask Questions - "What happens to the efficiency of a BST when it becomes unbalanced? How can we ensure the tree stays balanced?" - Transition to Red-Black Trees as a solution for balancing. - Briefly introduce Red-Black Trees: - "A Red-Black Tree is a self-balancing binary search tree that maintains balanced height using color properties of its nodes." 2. **Development** (30 minutes) a. Definition of Red-Black Trees - Define Red-Black Trees as a BST with additional rules to ensure balancing. - Use a simple example tree to highlight its structure. b. Properties of Red-Black Trees - Use diagrams to show the tree's growth after each insertion c. How the Properties Maintain Balance - Explain how these properties prevent the tree from becoming unbalanced. - Use diagrams to show the impact of these properties on height balance. 3. Exercise (5 minutes) – a. "Draw a Red-Black Tree for the following sequence of insertions: 30, 20, 40, 10, 25, 35, 50. Label the nodes with appropriate colors to satisfy all Red-Black Tree properties." Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Node colors. - Black height uniformity.

	- Logarithmic height guarantee. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "How do these properties make Red-Black Trees different from AVL Trees?" - "Why is it critical for practical systems like databases to use balanced trees like Red-Black Trees?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 2.4	Course Name: Advanced Data Structures and Algorithms Topic: Rotations, Insertion, and Deletion in Red-Black Trees	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the role of rotations in maintaining Red-Black Tree properties. b. Perform insertion and deletion operations in a Red-Black Tree while maintaining its balance. c. Analyze the time complexity of insertion, deletion, and rotation operations. d. Apply Red-Black Tree concepts to solve computational problems involving balanced search trees.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Recap the Properties of Red-Black Trees and why balancing is crucial: - Logarithmic height. - Efficient search, insertion, and deletion. - Introduce the need for rotations during insertion and deletion to maintain these properties: - "Rotations in Red-Black Trees ensure that the tree remains balanced while preserving the Red-Black properties." 2. **Development** (30 minutes) a. Rotations in Red-Black Trees - Types of Rotations: - Left Rotation: - Rotates a subtree counterclockwise. - Right Rotation: - Rotates a subtree clockwise. - When Rotations Are Needed: - To resolve violations of Red-Black Tree properties during insertion and deletion. - Illustrations and Examples: b. Insertion in Red-Black Trees - Steps for Insertion: - Insert the node as in a standard BST. - Color the node red. - Fix Red-Black Tree violations (if any) using rotations and recoloring. c. Deletion in Red-Black Trees

	- Steps for Deletion - Common Cases - Fixing Violations After Deletion - Illustrations and Examples 3. Exercise (5 minutes) – a. "Construct a Red-Black Tree by inserting the sequence: 40, 20, 50, 10, 30, 60. Then delete the node with value 20 and show the tree after each operation." Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Rotations (left and right) as the foundation for balancing. - Steps for insertion and deletion and how Red-Black properties are maintained. - Efficiency of Red-Black Trees for dynamic datasets. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "Why is it important for rotations to preserve the BST structure during balancing?" - "How do Red-Black Trees compare with AVL Trees in terms of complexity and efficiency?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 2.5	Course Name: Advanced Data Structures and Algorithms Topic: B-Trees: Definition, Basic Operations, and Deletion	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Define B-Trees and understand their role in multi-level indexing. b. Perform basic operations on B-Trees (insertion, traversal). c. Explain and implement the deletion of a key in a B-Tree while maintaining its properties. d. Analyze the time complexity of B-Tree operations.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Begin with a question: - "How do databases efficiently store and retrieve data from a massive dataset? How do search operations remain efficient as data grows?" - Relate this to B-Trees as a solution for balanced multi-level indexing in databases and file systems. 2. **Development** (30 minutes) a. Definition and Properties of B-Trees - Structure of a B-Tree Node: - A node contains n keys where $t - 1 \leq n \leq 2t - 1$ (t is the minimum degree). - Each internal node has n + 1 children. - Properties of B-Trees: - All leaves are at the same level. - Keys in a node are sorted in ascending order. - B-Tree of order t ensures a balanced height, maintaining O(log n) operations. b. Basic Operations on B-Trees - Insertion into a B-Tree - Traversal of a B-Tree. - Deletion in a B-Tree 3. Exercise (5 minutes) – a. "Construct a B-Tree of order 3 using the sequence: 15, 25, 5, 10, 20, 30. Then delete the keys 25 and 15 while maintaining B-Tree properties." Use Nearpod to collect responses and discuss the answers.

Closure	1. Summarize the Lesson Learning Outcomes - Definition and structure of B-Trees. - Basic operations: Insertion, traversal, and deletion. - The importance of maintaining balance in B-Trees. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "How does splitting and merging during insertion and deletion ensure the B-Tree remains balanced?" - "Why are B-Trees preferred over Binary Search Trees for databases?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 2.6	Course Name: Advanced Data Structures and Algorithms Topic: B-Trees: Definition, Basic Operations, and Deletion	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Define B-Trees and understand their role in multi-level indexing. b. Perform basic operations on B-Trees (insertion, traversal). c. Explain and implement the deletion of a key in a B-Tree while maintaining its properties. d. Analyze the time complexity of B-Tree operations.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Begin with a question: - "How do databases efficiently store and retrieve data from a massive dataset? How do search operations remain efficient as data grows?" - Relate this to B-Trees as a solution for balanced multi-level indexing in databases and file systems. 2. **Development** (30 minutes) a. Definition and Properties of B-Trees - Structure of a B-Tree Node: - A node contains n keys where $t - 1 \leq n \leq 2t - 1$ (t is the minimum degree). - Each internal node has n + 1 children. - Properties of B-Trees: - All leaves are at the same level. - Keys in a node are sorted in ascending order. - B-Tree of order t ensures a balanced height, maintaining O(log n) operations. b. Basic Operations on B-Trees - Insertion into a B-Tree - Traversal of a B-Tree. - Deletion in a B-Tree 3. Exercise (5 minutes) – a. "Construct a B-Tree of order 3 using the sequence: 15, 25, 5, 10, 20, 30. Then delete the keys 25 and 15 while maintaining B-Tree properties." Use Nearpod to collect responses and discuss the answers.

Closure	1. Summarize the Lesson Learning Outcomes - Definition and structure of B-Trees. - Basic operations: Insertion, traversal, and deletion. - The importance of maintaining balance in B-Trees. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "How does splitting and merging during insertion and deletion ensure the B-Tree remains balanced?" - "Why are B-Trees preferred over Binary Search Trees for databases?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 2.7	Course Name: Advanced Data Structures and Algorithms Topic: Fibonacci Heaps: Structure, Merge-Heap Operations	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the structure of a Fibonacci Heap and its properties. b. Explain the merge operation in Fibonacci Heaps. c. Analyze the time complexity of Fibonacci Heap operations. d. Apply the Fibonacci Heap to optimize algorithms such as Dijkstra's shortest path.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Ask Question - "Imagine you are managing a priority queue for tasks in an operating system. How would you efficiently retrieve the task with the highest priority?" - Relate this to the concept of priority queues and introduce Fibonacci Heaps as a data structure used for efficient priority queues. - Define Fibonacci Heap: 2. **Development** (30 minutes) a. Structure of Fibonacci Heap - Explain the basic structure of a Fibonacci Heap using diagrams: - Illustrate how trees are represented and how nodes are linked. b. Merge-Heap Operation - Explain the merge (union) operation in Fibonacci Heaps: - Introduce the pseudocode for the merge operation. - Discuss the time complexity 3. Exercise (5 minutes) – a. "Given two Fibonacci Heaps, Heap1 with elements 50, 30, 70, and Heap2 with elements 40, 60, 80. Merge the two heaps. Then perform the delete-min operation on the merged heap and visualize the structure after each operation." Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - The structure of Fibonacci Heaps and its advantages for merging heaps and decreasing keys.

	- The time complexity of Fibonacci Heap operations and its application in optimizing algorithms like Dijkstra's. - The importance of merge-heap operations for efficiently managing dynamic data. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "Why does the Fibonacci Heap support efficient merge operations, and how does this impact its use in real-world applications?" - "Can you think of any scenarios where Fibonacci Heaps would not be suitable compared to other heap structures?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 3.1	Course Name: Advanced Data Structures and Algorithms Topic: Graphs: Minimum Spanning Trees	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the concept of a Minimum Spanning Tree (MST) and its properties. b. Apply algorithms like Kruskal's and Prim's to find the MST of a given graph. c. Analyze the time complexity of MST algorithms. d. Apply MST concepts to real-world problems such as network design and circuit layout.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Ask Question - "Imagine you are building a network of roads connecting multiple cities. How would you ensure that the roads connecting the cities are as cost-effective as possible, while keeping all cities connected?" - Relate this to the concept of Minimum Spanning Trees, which is used to find the least-cost connections between nodes in a connected graph. - Define MST 2. **Development** (30 minutes) a. Basics of Graphs and Spanning Trees - Briefly review graph terminology: nodes, edges, weighted edges, and spanning trees. - Define a spanning tree: "A spanning tree is a subgraph that connects all the vertices of the graph with the minimum number of edges, without any cycles." - Discuss the properties of a Minimum Spanning Tree b. Kruskal's Algorithm - Introduce Kruskal's Algorithm: - "Kruskal’s algorithm is a greedy algorithm that finds the MST by adding edges in increasing order of weight, ensuring no cycles are formed." - Step-by-step explanation - Discuss time complexity 3. Exercise (5 minutes) – a. "Given a connected, weighted graph, apply both Kruskal’s and

	Prim's algorithms to find the MST. Compare the steps and edge selections made by each algorithm." Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - The concept of a Minimum Spanning Tree and its significance in optimizing cost or distance in graph-based problems. - Two popular algorithms for finding the MST: Kruskal's (edge-based) and Prim's (vertex-based). - Time complexity and efficiency considerations for both algorithms. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "What are the main differences between Kruskal's and Prim's algorithms in terms of edge selection and overall approach?" - "Can you think of any applications where the choice between Kruskal's and Prim's might impact performance?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 3.2	Course Name: Advanced Data Structures and Algorithms Topic: Graphs: Kruskal and Prim's Algorithm for Minimum Spanning Tree	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the concepts of Minimum Spanning Tree (MST). b. Learn how Kruskal's and Prim's algorithms are used to find the MST of a graph. c. Compare and contrast Kruskal's and Prim's algorithms in terms of approach and efficiency. d. Apply these algorithms to solve real-world problems such as network design and cost minimization.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Ask Question - "Imagine you are tasked with laying out cables to connect several buildings in a city. How do you ensure you use the least amount of cable while connecting all the buildings?" - Relate this to the concept of Minimum Spanning Trees (MST), which is used in various network design problems. - Define MST 2. **Development** (30 minutes) a. Kruskal's Algorithm - Introduce Kruskal's Algorithm: - "Kruskal's algorithm is a greedy algorithm that finds the MST by adding edges in increasing order of weight, ensuring no cycles are formed." - Step-by-step explanation - Discuss time complexity b. Prim's Algorithm - Introduce Prim's Algorithm - Step-by-step explanation - Discuss time complexity 3. Exercise (5 minutes) – a. Ask students to find the MST using both Kruskal's and Prim's algorithms and compare the results. Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes

	- Kruskal's and Prim's algorithms are both greedy algorithms that find the Minimum Spanning Tree but approach the problem differently. - Kruskal's algorithm works well for sparse graphs, while Prim's is more efficient for dense graphs when using a priority queue. - Both algorithms guarantee an optimal solution for finding the MST. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "Why does Prim's algorithm require a priority queue, and how does this impact its time complexity?" - "In which scenarios would you prefer Kruskal's algorithm over Prim's and vice versa?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 3.3	Course Name: Advanced Data Structures and Algorithms Topic: Single-Source Shortest Paths: The Bellman-Ford Algorithm	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the concept of the Single-Source Shortest Path (SSSP) problem. b. Learn how the Bellman-Ford algorithm works to solve the SSSP problem, including handling graphs with negative edge weights. c. Identify when the Bellman-Ford algorithm is preferred over other algorithms like Dijkstra's. d. Analyze the time complexity and limitations of the Bellman-Ford algorithm.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Ask Question - "Imagine you're traveling through a city with different roads (edges) that have different travel times (weights). How do you find the quickest route from your starting point to every other point in the city, especially if some roads have negative travel times (like tolls or discounts)?" - Relate this scenario to the concept of the Single-Source Shortest Path (SSSP) problem in graph theory, where the goal is to find the shortest path from a single source node to all other nodes in the graph. - Introduce the Bellman-Ford algorithm: - "The Bellman-Ford algorithm is used to find the shortest paths from a single source node to all other nodes in a graph. Unlike other algorithms like Dijkstra's, it works with graphs that have negative edge weights." 2. **Development** (30 minutes) a. Problem Definition and Bellman-Ford Overview - Define the Single-Source Shortest Path (SSSP) problem: - "Given a weighted graph with edges that may have negative weights, the goal is to find the shortest path from a single source node to every other node in the graph." - Bellman-Ford Algorithm Overview b. Bellman-Ford Algorithm Steps - Step-by-step demonstrate how the algorithm relaxes edges and updates the shortest path estimates. - Show the changes in the distance array after each iteration.

	- Detect if there is any negative weight cycle in the graph. 3. Exercise (5 minutes) – a. Ask students to apply the Bellman-Ford algorithm and find the shortest path from vertex A to all other vertices. Vertices: A, B, C, D, E Edges: (A, B, 3), (A, C, -1), (B, C, 1), (B, D, 7), (C, D, 4), (D, E, -2) Source: A Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - The Bellman-Ford algorithm finds the shortest paths from a single source node to all other nodes, even in graphs with negative edge weights. - The algorithm works by relaxing all edges up to V−1V-1V−1 times and can detect negative weight cycles. - Time complexity is O(V×E), making it less efficient for dense graphs compared to other algorithms like Dijkstra's (for graphs without negative edge weights). Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "What would happen if we didn't perform the extra iteration after V−1V-1V−1 passes in the Bellman-Ford algorithm?" - "How would you modify the Bellman-Ford algorithm if the graph is very large and sparse?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 3.4	Course Name: Advanced Data Structures and Algorithms Topic: Single-Source Shortest Paths in Directed Acyclic Graphs (DAGs)	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the concept of Single-Source Shortest Path (SSSP) in Directed Acyclic Graphs (DAGs). b. Learn how to compute the shortest path from a single source node to all other nodes in a DAG efficiently. c. Apply topological sorting to optimize the SSSP computation in a DAG. d. Understand the time complexity and advantages of using DAGs for shortest path problems.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Start by discussing Directed Acyclic Graphs (DAGs): - "A Directed Acyclic Graph (DAG) is a graph that has directed edges and no cycles. DAGs are useful in modeling systems where tasks need to be executed in a specific order, such as project scheduling or compiling dependencies." - Introduce the Single-Source Shortest Path (SSSP) problem in the context of a DAG: - "In many applications, we need to find the shortest path from a single source node to all other nodes in a DAG. Since DAGs don't have cycles, we can take advantage of this property to compute the shortest paths efficiently." 2. **Development** (30 minutes) a. Problem Definition and DAG Characteristics - DAG Properties: - A DAG contains vertices and directed edges with no cycles. - The graph can be acyclic but may have multiple paths between nodes. - Why Use Topological Sorting: - "Since a DAG does not have cycles, it can be linearly ordered using topological sorting. This property is key to solving the Single-Source Shortest Path problem efficiently." - Discuss how topological sorting is performed in a DAG, where the vertices are ordered such that for every directed edge (u, v), vertex u comes before v. b. Single-Source Shortest Path Algorithm for DAGs - Steps of the Algorithm

	- Topological Sort - Initialize Distances - Relax Edges - Output the Shortest Path 3. Exercise (5 minutes) – a. Present students with the following DAG and ask them to compute the shortest path from node A to all other nodes: Vertices: A, B, C, D, E, F Edges: (A, B, 2), (A, C, 4), (B, C, 1), (B, D, 7), (C, E, 3), (D, F, 1), (E, F, 5) Source: A Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - The Single-Source Shortest Path (SSSP) problem in Directed Acyclic Graphs (DAGs) can be solved efficiently by combining topological sorting and edge relaxation. - The time complexity of the algorithm is O(V+E), making it very efficient for sparse graphs. - DAGs are particularly useful in scenarios like project scheduling and dependency resolution, where tasks must be performed in a specific order. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "How would you modify the algorithm if the graph were not acyclic, and why wouldn't topological sorting work in such a case?" - "Can you think of real-world scenarios where using DAGs for shortest path computation would be more efficient than using other algorithms like Dijkstra's?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 3.5	Course Name: Advanced Data Structures and Algorithms Topic: All-Pairs Shortest Paths	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the concept of All-Pairs Shortest Paths (APSP) in graphs. b. Learn how to compute the shortest paths between all pairs of nodes in a graph. c. Explore two key algorithms for APSP: Floyd-Warshall and Johnson's Algorithm. d. Analyze the time complexity of these algorithms and their use cases.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Introduce the problem of finding the shortest paths between all pairs of vertices in a graph: - "In many real-world applications, we may need to know the shortest path between all pairs of nodes, not just from a single source. For example, in routing networks or city navigation systems, finding all-pairs shortest paths helps in efficiently directing traffic." - Define the All-Pairs Shortest Path (APSP) problem: - "Given a graph, we need to compute the shortest path between every pair of nodes." 2. **Development** (30 minutes) a. Overview of APSP Problem - Discuss scenarios where APSP is useful - Highlight that solving APSP can be done using various algorithms, with the two main ones being Floyd-Warshall Algorithm and Johnson's Algorithm. b. Floyd-Warshall Algorithm - Algorithm Overview: - The Floyd-Warshall algorithm computes the shortest paths between all pairs of vertices in a weighted graph with positive or negative edge weights, as long as there are no negative weight cycles. 3. Exercise (5 minutes) – a. Present a graph and ask students to compute the shortest paths between all pairs of vertices using Floyd-Warshall or Johnson's

	Algorithm. Vertices: A, B, C, D Edges: (A, B, 4), (A, C, 1), (B, D, 1), (C, B, 2), (C, D, 5) Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - The All-Pairs Shortest Path (APSP) problem computes the shortest path between all pairs of nodes in a graph. - The **Floyd-Warshall Algorithm** is a direct approach with a time complexity of O(n^3), suitable for dense graphs. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - How would the APSP algorithm change if the graph contains negative weight cycles?" - "Can you think of a scenario where Floyd-Warshall would be less efficient than Johnson's Algorithm?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 3.6	Course Name: Advanced Data Structures and Algorithms Topic: Matrix Multiplication	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the concept of matrix multiplication and its rules. b. Implement and optimize the standard matrix multiplication algorithm. c. Explore and analyze the Strassen's algorithm for matrix multiplication. d. Analyze the time complexities of the matrix multiplication algorithms.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Start with a real-life scenario to make the topic relatable: - "Matrix multiplication is used in various fields such as computer graphics, data science, machine learning, and even solving systems of linear equations. Imagine transforming a 3D model into a 2D screen; this process often involves matrix operations." - Define matrix multiplication: - "Matrix multiplication involves multiplying two matrices to produce a new matrix. This operation is not as straightforward as element-wise multiplication; it requires summing products of corresponding elements from rows and columns." 2. **Development** (30 minutes) a. Standard Matrix Multiplication b. Strassen's Matrix Multiplication Algorithm - Algorithm Overview: - Strassen's algorithm is an optimized approach that reduces the number of multiplications required, improving the time complexity. - Strassen's algorithm divides each matrix into four submatrices and recursively computes products using a divide-and-conquer approach. It performs matrix multiplication with fewer recursive steps compared to the standard algorithm. 3. Exercise (5 minutes) – a. Encourage students to compare the computational efficiency of both methods observing runtime for larger matrices. Use Nearpod to collect responses and discuss the answers.

Closure	1. Summarize the Lesson Learning Outcomes - Matrix multiplication is fundamental to many algorithms in computer science, especially in fields like machine learning and computer graphics. - The **standard matrix multiplication** algorithm has a time complexity of O(m×n×p). Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "How does the number of operations in Strassen's algorithm compare to the standard matrix multiplication for large matrices?" - "Can you think of any real-world applications where Strassen's algorithm would be preferred over standard matrix multiplication?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 3.7	Course Name: Advanced Data Structures and Algorithms Topic: Application of Cryptography to Blockchain: Using Hash Functions to Chain Blocks	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the fundamental concept of blockchain and its applications. b. Learn how cryptography, particularly hash functions, is used in blockchain technology. c. Demonstrate how hash functions are used to chain blocks in a blockchain. d. Analyze the role of hash functions in ensuring the security and integrity of data in blockchain systems.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Start with a real-world analogy: - "Imagine a digital ledger where every transaction is recorded. Once recorded, the transaction cannot be changed or deleted. This ledger is shared across a decentralized network of computers, ensuring trust and transparency. This is essentially how blockchain works." - Define Blockchain: - "A blockchain is a distributed, decentralized, and immutable database that consists of blocks linked in a chain. Each block contains a list of transactions and a cryptographic hash of the previous block, ensuring security and continuity." - Introduce Cryptography: - "Cryptography is used in blockchain to ensure data security, privacy, and integrity. One important cryptographic technique used in blockchain is the hash function." 2. **Development** (30 minutes) a. Introduction to Hash Functions - Define Hash Functions - Properties of Hash Functions - Common Hash Functions b. Blockchain Structure and Chaining Blocks Using Hash Functions - Block Structure - Hashing to Link Blocks. - Demonstrating Blockchain with Hash Functions

	3. Exercise (5 minutes) – a. Provide a simple set of transactions and ask students to create a small blockchain with 3 blocks. Example: Block 1: "Alice sends 5 BTC to Bob" Block 2: "Bob sends 2 BTC to Charlie" Block 3: "Charlie sends 1 BTC to Dave" b. Ask students to compute the hash for each block and demonstrate how changing the data in one block alters the entire blockchain. Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Hash functions are a crucial component of blockchain technology, ensuring data integrity and security. - Blockchain uses hash functions to link blocks together, creating an immutable and tamper-proof chain. - The cryptographic nature of hash functions helps maintain trust in decentralized systems. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "What happens to the blockchain if the hash of one block is altered? How does this affect the security of the entire chain?" - "Why is decentralization important in ensuring the integrity of blockchain data?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 4.1	Course Name: Advanced Data Structures and Algorithms Topic: Algorithm Design Techniques	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the fundamental algorithm design techniques. b. Apply divide-and-conquer, greedy, dynamic programming, and backtracking methods to solve problems. c. Analyze the efficiency of algorithms designed using different techniques. d. Compare and contrast different algorithm design techniques and their use cases.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Real-world analogy: - "When you're trying to solve a complex problem, you may approach it in multiple ways, just like when you're looking for the quickest route on a map. You can either break it down into smaller steps, try an immediate best option, or consider all possibilities to find the optimal solution." - Introduce the concept of algorithm design techniques: - "In computer science, there are several techniques to design efficient algorithms, each of which is suited to different kinds of problems." - Briefly list the main techniques: - Divide and Conquer - Greedy Algorithms - Dynamic Programming - Backtracking 2. **Development** (30 minutes) a. Divide and Conquer - Definition: - "Divide and Conquer is a technique where the problem is divided into smaller subproblems, solved recursively, and then the solutions to the subproblems are combined to get the final solution." - Key Steps: - Divide the problem into smaller subproblems. - Conquer each subproblem recursively. - Combine the results to solve the original problem. - Example: "Merge Sort is a classic example. The array is split into smaller

	subarrays, sorted individually, and then merged together to form the final sorted array." b. Greedy Algorithms - Definition: - "A Greedy algorithm makes a series of choices by selecting the locally optimal option at each step with the hope that these local solutions lead to a global optimum." - Key Steps: - At each step, pick the best possible option that seems best at that moment. - Greedy algorithms do not reconsider choices once made. - Example: "The Fractional Knapsack Problem is a classic example, where you select the items with the highest value-to-weight ratio to maximize the total value of items that can be carried." 3. Exercise (5 minutes) – a. Problem: "Given a list of numbers, find the largest sum of a non-adjacent subsequence." b. Students should decide which algorithm design technique (dynamic programming or greedy) is best suited for this problem and justify their choice. Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Algorithm design techniques are fundamental strategies for solving complex problems efficiently. - Divide and Conquer, Greedy, Dynamic Programming, and Backtracking are four important techniques, each suited to different types of problems. - The choice of technique depends on the problem constraints and the goal of finding an optimal or approximate solution. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "In which cases would you prefer Dynamic Programming over Greedy algorithms?" - "Can you think of a problem where Backtracking would not be efficient? Why?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 4.2	Course Name: Advanced Data Structures and Algorithms Topic: Dynamic Programming: Matrix-Chain Multiplication	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the concept of matrix-chain multiplication. b. Apply dynamic programming to solve matrix-chain multiplication problems. c. Analyze the time complexity of the matrix-chain multiplication problem. d. Implement the matrix-chain multiplication algorithm using dynamic programming.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Real-world analogy: - "Imagine you are trying to multiply a series of matrices, but the order in which you multiply them matters, as it can affect the computational cost. How would you decide the best order to minimize the total cost?" - Introduce Matrix-Chain Multiplication: - "Matrix-Chain Multiplication is an optimization problem where we want to find the most efficient way to multiply a chain of matrices. The goal is to minimize the number of scalar multiplications." - Briefly outline the dynamic programming approach: - "We will use dynamic programming to break the problem into smaller subproblems and solve each subproblem once, storing results to avoid redundant calculations." 2. **Development** (30 minutes) a. Problem Definition and Key Concept - Matrix-Chain Multiplication Problem: - "Given a sequence of matrices, the task is to determine the most efficient way to multiply them together. The number of scalar multiplications required depends on the order of multiplication." - Example: - "Suppose we have three matrices A, B, and C, with dimensions 10×20, 20×30, and 30×40 respectively. The cost of multiplying two matrices is the product of their row and column dimensions." - Matrix Multiplication Cost: - "Multiplying a matrix of dimensions A×B with a matrix of

	dimensions B×C requires A * B * C scalar multiplications." Subproblem: - "The goal is to determine the optimal order to multiply the matrices such that the total number of scalar multiplications is minimized." b. Dynamic Programming Approach - Concept of Substructure: - "We can break the problem into smaller subproblems. If we know the optimal way to multiply the first i matrices and the optimal way to multiply the remaining matrices, we can combine them efficiently." c. Matrix-Chain Multiplication Algorithm 3. Exercise (5 minutes) – a. Problem: "Given the matrix dimensions 10×100, 100×5, 5×50, 50×1, find the minimum number of scalar multiplications required to multiply these matrices using dynamic programming." b. Students should calculate the matrix-chain order and discuss the steps involved in solving the problem. Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Matrix-Chain Multiplication is an optimization problem that can be efficiently solved using dynamic programming. - The key idea is to break the problem into smaller subproblems, solving each one and storing the results to avoid redundant calculations. - The dynamic programming solution minimizes the number of scalar multiplications required to multiply a chain of matrices. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "How does the matrix-chain multiplication problem relate to real-world computational tasks, such as graphics rendering or large-scale matrix operations?" - "Can you think of other problems where dynamic programming can be applied to optimize performance?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 4.3	Course Name: Advanced Data Structures and Algorithms Topic: Elements of Dynamic Programming	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the core elements and principles of dynamic programming (DP). b. Identify the characteristics of problems that can be solved using DP. c. Implement dynamic programming solutions to solve optimization problems. d. Analyze time and space complexity in DP algorithms.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Real-world analogy: - "Imagine you are planning a road trip with multiple stops, and you want to find the most efficient route. Instead of recalculating the best route every time you reach a stop, you keep track of previously calculated routes to avoid redundant computations. This is similar to the approach used in dynamic programming." - Introduce Dynamic Programming: - "Dynamic programming (DP) is a method for solving problems by breaking them down into smaller subproblems and solving each subproblem just once, storing the solutions for reuse. It's often used to solve optimization problems where we aim to find the best solution among many possible ones." 2. **Development** (30 minutes) a. Elements of Dynamic Programming - Optimal Substructure: - "A problem exhibits optimal substructure if the solution to the problem can be constructed from solutions to its subproblems." Example: "In the Fibonacci sequence, the nth number is the sum of the (n-1)th and (n-2)th numbers." - Overlapping Subproblems: - "A problem exhibits overlapping subproblems if the problem can be broken down into smaller subproblems that are solved repeatedly." Example: "In the Fibonacci sequence, we compute the same Fibonacci numbers multiple times if we solve the problem recursively without storing intermediate results." b. Classic Examples of DP Problems

	- Fibonacci Sequence - 0/1 Knapsack Problem: c. Time and Space Complexity of DP Algorithms 3. Exercise (5 minutes) – a. Problem: "Given the following set of items with weights and values, and a knapsack with a capacity of 50, determine the maximum value that can be obtained using dynamic programming." Items: Item 1: Weight = 10, Value = 60 Item 2: Weight = 20, Value = 100 Item 3: Weight = 30, Value = 120 b. Use a dynamic programming approach to solve this problem, and visualize the table of solutions. Discuss the steps involved in filling out the table. Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Dynamic programming is an optimization technique that involves solving problems by breaking them into overlapping subproblems and solving them efficiently. - The two main elements of dynamic programming are optimal substructure and overlapping subproblems. - DP is typically solved using either memoization (top-down approach) or tabulation (bottom-up approach), both of which improve the efficiency of naive recursive solutions. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "How can dynamic programming be applied to solve problems in areas like operations research, game theory, or machine learning?" - "What happens if the subproblems of a given problem are not overlapping—can dynamic programming still be applied?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 4.4	Course Name: Advanced Data Structures and Algorithms Topic: Longest Common Subsequence (LCS)	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the problem of finding the Longest Common Subsequence (LCS) between two sequences. b. Implement the dynamic programming approach to solve the LCS problem. c. Analyze the time and space complexity of the LCS algorithm. d. Apply LCS to real-world problems, such as text comparison and DNA sequence analysis.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Real-world analogy: - "Imagine you're comparing two documents to see how much of their content overlaps. You want to find the longest series of words or phrases that appear in both documents in the same order. This is similar to the Longest Common Subsequence problem." - Define LCS: - "The Longest Common Subsequence (LCS) of two sequences is the longest subsequence that appears in both sequences in the same order, but not necessarily contiguously." 2. **Development** (30 minutes) a. Explanation of LCS - Subsequence: - "A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements." - Examples: - For sequences A = [A, B, C, D, G, H] and B = [A, E, D, F, H, R], the LCS is [A, D, H]. - For sequences A = [A, B, C, D] and B = [A, C, D], the LCS is [A, C, D]. b. Problem Statement - "Given two sequences, X and Y, find the longest subsequence that is common to both. We are not required to find all common subsequences, just the longest one." c. Dynamic Programming Approach 3. Exercise (5 minutes) –

	a. Problem: "Given two sequences X = [A, B, C, D, G, H] and Y = [A, E, D, F, H, R], compute the LCS using dynamic programming. Visualize the DP table and trace the steps of the algorithm." b. Discuss the formation of the DP table and how the values are filled in according to the recurrence relation. Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - The LCS problem finds the longest subsequence common to two sequences, not necessarily contiguous. - The dynamic programming approach solves the problem efficiently by breaking it down into smaller subproblems. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "How does the LCS algorithm compare to brute-force approaches in terms of efficiency?" - "Can you think of real-world problems where finding the LCS can help in optimizing solutions?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 4.5	Course Name: Advanced Data Structures and Algorithms Topic: Greedy Algorithms: An Activity Selection Problem	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the concept of greedy algorithms and how they work. b. Solve the Activity Selection Problem using the greedy approach. c. Analyze the time complexity of the greedy algorithm for the activity selection problem. d. Apply greedy algorithms to solve other optimization problems in real-world scenarios.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Real-world analogy: - "Imagine you're attending several meetings in a day, and you want to maximize the number of meetings you can attend without any overlap. This is similar to the Activity Selection Problem." - Define Greedy Algorithm: - "A greedy algorithm makes the locally optimal choice at each step with the hope of finding the global optimum. In the case of activity selection, it aims to select the maximum number of activities that don't overlap in time." 2. **Development** (30 minutes) a. Problem Definition - "Given a set of activities with start and finish times, select the maximum number of activities that don't overlap. You can only be in one activity at a time." b. Greedy Approach for Activity Selection - "The greedy approach for the Activity Selection Problem is to always select the activity that finishes the earliest among the available options. This leaves the maximum remaining time for future activities." c. Time Complexity of the Greedy Approach - "The time complexity of the greedy algorithm for activity selection is O(n log n) because the primary operation is sorting the activities by finish time. After sorting, the selection step is O(n), which is linear." - The space complexity is O(n) due to the storage of activities and the selected set 3. Exercise (5 minutes) –

	a. "Given the following set of activities with their start and finish times, find the maximum number of non-overlapping activities: Activity 1: (2, 4) Activity 2: (3, 6) Activity 3: (5, 8) Activity 4: (7, 9) Activity 5: (6, 10) Activity 6: (8, 11) What is the maximum number of activities you can select using the greedy approach?" b. Allow students to work in pairs or individually to solve the problem and discuss the steps. Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - The Activity Selection Problem is solved using a greedy algorithm that selects the activity that finishes the earliest and leaves room for more activities. - Greedy algorithms are efficient and easy to implement for certain optimization problems like this one. - The time complexity of the algorithm is O(n log n) due to the sorting step. - The greedy approach may not always work for problems that require considering future consequences of current choices. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "What would happen if we selected the activity that starts the earliest instead of the one that finishes the earliest?" - "Can you think of any other optimization problems that might be solved using greedy algorithms?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 4.6	Course Name: Advanced Data Structures and Algorithms Topic: Huffman Codes	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the concept of Huffman coding and its applications in data compression. b. Implement the Huffman coding algorithm. c. Analyze the time complexity of the Huffman coding algorithm. d. Apply Huffman coding to encode and decode data.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - Real-life analogy: - "Imagine you're sending a message over the internet, and you want to send it as efficiently as possible. One way to do this is by reducing the size of the message through compression, so it takes up less space and can be sent more quickly." - Define Huffman Coding: - "Huffman coding is an algorithm used for data compression. It assigns variable-length codes to input characters, with shorter codes assigned to more frequent characters and longer codes to less frequent ones." - Applications: - "Huffman coding is used in file formats such as JPEG, MP3, and in data transmission protocols like ZIP compression." 2. **Development** (30 minutes) a. Problem Definition - "Given a set of characters and their frequencies, the goal is to assign variable-length codes to the characters such that the overall encoding is as efficient as possible, meaning the total number of bits used to represent the message is minimized." b. Huffman Coding Algorithm - Create a priority queue (min-heap) where each node represents a character and its frequency. - Build the Huffman tree: - Extract the two nodes with the lowest frequencies from the queue. - Create a new node with a frequency equal to the sum of the two nodes' frequencies. - Add this new node back into the priority queue. - Repeat until only one node remains in the queue (this is the root of the Huffman tree). - Assign codes: Traverse the Huffman tree, assigning binary

	codes (0 for left, 1 for right) to each character based on its position in the tree. 3. Exercise (5 minutes) – a. "Given the following characters and their frequencies, construct the Huffman tree and assign codes: b. G: 30, H: 7, I: 8, J: 6, K: 12." c. Allow students to work in pairs or individually and then discuss the step-by-step process and the final encoding. Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Huffman coding is an optimal compression technique that assigns variable-length codes based on the frequency of characters. - The algorithm uses a greedy approach, always choosing the least frequent characters to merge first. - Huffman coding is widely used in compression formats like ZIP, JPEG, and MP3 due to its efficiency. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "Why is Huffman coding considered a greedy algorithm?" - "Can you think of any other areas where data compression might be useful outside of files and media?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 5.1	Course Name: Advanced Data Structures and Algorithms Topic: NP Complete And NP Hard:	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Differentiate between P, NP, NP-Complete, and NP-Hard classes. b. Understand the significance of NP-Complete and NP-Hard problems in computational theory. c. Analyze examples of NP-Complete problems such as the Traveling Salesman Problem and the Boolean Satisfiability Problem (SAT). d. Identify strategies for tackling NP-Hard problems, including approximation and heuristic methods.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - "Imagine you're solving a puzzle with a million pieces. Some puzzles have straightforward strategies, while others seem unsolvable within a lifetime. This is the essence of NP-Complete problems." - Key Definitions: - P: Problems that can be solved efficiently (in polynomial time). - NP: Problems where solutions can be verified efficiently. - NP-Complete: Problems in NP for which every other NP problem can be reduced to them in polynomial time. - NP-Hard: Problems as hard as NP-Complete but may not be in NP. - Importance in Computing: 2. **Development** (30 minutes) a. Problem Understanding - NP-Complete Problems: - Examples: SAT, Traveling Salesman Problem (TSP), 3-Coloring Problem. - Discuss the significance of polynomial-time reductions. - NP-Hard Problems: - Include problems that are not necessarily decision problems (e.g., TSP optimization version). b. Theoretical Background - Reduction technique: Explain how a problem is proven to be NP-Complete by reducing another NP-Complete problem to it. - SAT as the first NP-Complete problem: Briefly cover the Cook-Levin theorem. 3. Exercise (5 minutes) – a. "Given a problem like TSP or Vertex Cover, discuss whether it

	is NP-Complete or NP-Hard and why." Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - P vs NP: The challenge of finding efficient solutions vs verifying them. - NP-Complete problems are critical for understanding computational limits. - Approximation and heuristics are practical approaches for NP-Hard problems. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - Why is it significant to know whether a problem is NP-Complete or NP-Hard? - How might advancements in solving NP-Complete problems impact technology? Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 5.2	Course Name: Advanced Data Structures and Algorithms Topic: NP-Completeness: Polynomial Time	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Define and understand the concept of polynomial time in computational theory. b. Explain the relationship between P, NP, and NP-Complete classes in the context of polynomial time. c. Understand the significance of polynomial-time reductions in proving NP-Completeness. d. Analyze examples of problems to classify them as P, NP, or NP-Complete.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - "Imagine you're trying to solve a sudoku puzzle. For smaller grids, it is easy to find a solution quickly, but as the grid size grows, finding a solution becomes exponentially harder. This reflects the essence of polynomial time in computational complexity." - Key Definitions: - **Polynomial Time:** Problems solvable in O(nk)O(n^k)O(nk), where kkk is a constant. - P Class: Problems solvable in polynomial time. - NP Class: Problems where solutions can be verified in polynomial time. - NP-Complete: Problems in NP where every NP problem can be reduced to them in polynomial time. 2. **Development** (30 minutes) a. Polynomial Time and Complexity Classes - Polynomial Time - NP Problems: Explain problems like Sudoku or SAT where finding a solution is hard, but verifying one is easy. b. Understanding NP-Completeness - Cook-Levin Theorem - Reduction Example: - Show an example of reducing 3-SAT to another NP-Complete problem like Vertex Cover. 3. Exercise (5 minutes) – a. Divide the class into groups to analyze a new problem, decide

	if it's P, NP, or NP-Complete, and justify their reasoning. Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Polynomial time defines computational feasibility. - P, NP, and NP-Complete classes help categorize problems by their solvability and verification. - Polynomial-time reductions are essential for proving NP-Completeness. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - Why is it easier to verify solutions in NP problems than to find them? - How might proving P = NP (or not) change modern computing? Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 5.3	Course Name: Advanced Data Structures and Algorithms Topic: Polynomial-Time Verification	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Understand the concept of polynomial-time verification and its role in defining the NP class of problems. b. Differentiate between solving a problem and verifying a solution in polynomial time. c. Analyze examples of problems in NP and explain how their solutions can be verified efficiently. d. Apply the concept of polynomial-time verification to recognize NP problems.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - "Imagine you're grading a multiple-choice test. Checking a completed answer sheet is quick and straightforward, but if you were asked to solve the test from scratch, it could take much longer. This illustrates the concept of verification vs solving." - Key Definitions: - Verification: The process of confirming a solution is correct. - NP Class: The class of problems for which solutions can be verified in polynomial time. 2. **Development** (30 minutes) a. Key Concepts - Solving vs Verifying: - Solving: Finding a solution from scratch. - Verifying: Checking the correctness of a given solution. - Example: Sudoku puzzles (solving is hard, checking correctness is easy). - Verification in Polynomial Time: - Demonstrate with an example: - For SAT, given a boolean formula and an assignment of variables, it is easy to check whether the assignment satisfies the formula in polynomial time. b. Relationship Between P and NP - P Problems: Problems solvable in polynomial time. - NP Problems: Problems verifiable in polynomial time. - Traveling Salesman Problem (TSP): 3. Exercise (5 minutes) – a. Given a problem, outline the steps for verifying a solution in

	polynomial time." Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - Polynomial time defines computational feasibility. - P, NP, and NP-Complete classes help categorize problems by their solvability and verification. - Polynomial-time reductions are essential for proving NP-Completeness. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "Why is verification often easier than solving the problem itself?" Spend 5 minutes to evaluate student assimilation of the lesson contents

Lesson Plan No. 5.4	Course Name: Advanced Data Structures and Algorithms Topic: NP-Completeness and Reducibility	Course No.: MCSE103

Objectives	At the end of the lesson the student shall be able to: a. Define NP-Completeness and understand its significance in computational theory. b. Explain the concept of polynomial-time reducibility and its role in proving NP-Completeness. c. Analyze examples of NP-Complete problems and the process of reductions. d. Apply reducibility to classify new problems as NP-Complete.
Teaching Aids (if any)	a. Visual representations of algorithms (flowcharts, pseudocode). b. Code snippets for illustrating key algorithmic concepts. c. Whiteboard or digital whiteboard for step-by-step explanation.
Teaching Development	1. **Introduction** (5 minutes) - "Imagine you're trying to solve a series of puzzles. If you can solve one particularly challenging puzzle, you can use its solution to solve all the others. This is the essence of NP-Completeness and reducibility." - NP-Complete Problems: Problems in NP where every other problem in NP can be reduced to them in polynomial time. - Reducibility: A process of transforming one problem into another to show equivalence in complexity. 2. **Development** (30 minutes) a. Understanding NP-Completeness - Characteristics of NP-Complete Problems - Examples of NP-Complete Problems. - Significance of NP-Complete Problems b. Reducibility and Its Role - What is Polynomial-Time Reducibility? - Steps to Prove NP-Completeness: - Illustration 3. Exercise (5 minutes) – a. Provide a simple problem (e.g., Vertex Cover) and ask students to outline how it might be reduced from another NP-Complete problem like 3-SAT. Use Nearpod to collect responses and discuss the answers.
Closure	1. Summarize the Lesson Learning Outcomes - NP-Completeness defines the hardest problems in NP. - Reducibility is a powerful tool for proving NP-Completeness.

	- Understanding these concepts is crucial for tackling complex computational problems. Spend 5 minutes to wrap up and consolidate the learnings
Evaluation	1. Reflective Questions: - "Why is reducibility central to proving NP-Completeness?" - "How does knowing a problem is NP-Complete influence the approach to solving it?" Spend 5 minutes to evaluate student assimilation of the lesson contents