|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE** | | | | | **DEPARTMENT OF COMPUTER SCIENCE ENGINEERING** | | | | |
| **Program Name:** B. Tech | | | | **Assignment Type: Lab** | | | **Academic Year:**2025-2026 | | |
| **Course Coordinator Name** | | | | Venkataramana Veeramsetty | | | | | |
| **Instructor(s) Name** | | | | |  | | --- | | Dr. V. Venkataramana (Co-ordinator) | | Dr. T. Sampath Kumar | | Dr. Pramoda Patro | | Dr. Brij Kishor Tiwari | | Dr.J.Ravichander | | Dr. Mohammand Ali Shaik | | Dr. Anirodh Kumar | | Mr. S.Naresh Kumar | | Dr. RAJESH VELPULA | | Mr. Kundhan Kumar | | Ms. Ch.Rajitha | | Mr. M Prakash | | Mr. B.Raju | | Intern 1 (Dharma teja) | | Intern 2 (Sai Prasad) | | Intern 3 (Sowmya) | | NS\_2 ( Mounika) | | | | | | |
| **Course Code** | | | 24CS002PC215 | **Course Title** | | AI Assisted Coding | | | |
| **Year/Sem** | | | II/I | **Regulation** | | R24 | | | |
| **Date and Day**  **of Assignment** | | | Week6 - Monday | **Time(s)** | |  | | | |
| **Duration** | | | 2 Hours | **Applicable to**  **Batches** | |  | | | |
| **AssignmentNumber:12.1**(Present assignment number)/**24**(Total number of assignments) | | | | | | | | | |
|  | | | | | | | | | |
|  | **Q.No.** | **Question** | | | | | | ***Expected Time***  ***to complete*** |  |
|  | 1 | **Lab 12: Algorithms with AI Assistance – Sorting, Searching, and Optimizing Algorithms**  **Lab Objectives:**   * Apply AI-assisted programming to implement and optimize sorting and searching algorithms. * Compare different algorithms in terms of efficiency and use cases. * Understand how AI tools can suggest optimized code and complexity improvements.   **Task Description #1 (Sorting – Merge Sort Implementation)**   * Task: Use AI to generate a Python program that implements the Merge Sort algorithm. * Instructions:   + Prompt AI to create a function merge\_sort(arr) that sorts a list in ascending order.   + Ask AI to include time complexity and space complexity in the function docstring.   + Verify the generated code with test cases. * Expected Output:   + A functional Python script implementing Merge Sort with proper documentation.   **PROMPT:**  Write a Python function `merge\_sort(arr)` that returns a sorted list (ascending). Include a docstring describing the algorithm, and the best/average/worst-case time complexities and space complexity. Make it stable and easy to read. Provide at least 3 assert test cases afterwards.  **CODE:**      **OUTPUT:**    **EXPLANATION:**   * Why: Merge sort is a stable O(n log n) algorithm with predictable performance and good worst-case guarantees. It is an excellent teaching algorithm and suitable for sorting large datasets where worst-case guarantees matter. * Complexity: Best/Average/Worst = O(n log n). Extra space: O(n) (for merged arrays). * Stable: Yes — equal elements preserve original order.   **Task Description #2 (Searching – Binary Search with AI Optimization)**   * Task: Use AI to create a binary search function that finds a target element in a sorted list. * Instructions:   + Prompt AI to create a function binary\_search(arr, target) returning the index of the target or -1 if not found.   + Include docstrings explaining best, average, and worst-case complexities.   + Test with various inputs. * Expected Output:   + Python code implementing binary search with AI-generated comments and docstrings.   **PROMPT:**  Write a Python function `binary\_search(arr, target)` that returns the index of target in sorted list arr, or -1 if not present. Add a docstring with best/average/worst-case time complexities and space complexity. Provide at least 3 assert test cases covering hits, misses, and edge cases.  **CODE:**      **OUTPUT:**    **EXPLANATION:**   * Why: For membership/index search on a sorted array, binary search is optimal in comparison model, O(log n) time, O(1) space for iterative version. Use it when data is already sorted and random access index is O(1). * Caveat: If data is frequently updated (insertions/deletions), consider other structures (balanced BST, hash table) because maintaining sorted array costs O(n) per insert.   **Task Description #3 (Real-Time Application – Inventory Management System)**   * Scenario: A retail store’s inventory system contains thousands of products, each with attributes like product ID, name, price, and stock quantity. Store staff need to:   1. Quickly search for a product by ID or name.   2. Sort products by price or quantity for stock analysis. * Task:   + Use AI to suggest the most efficient search and sort algorithms for this use case.   + Implement the recommended algorithms in Python.   + Justify the choice based on dataset size, update frequency, and performance requirements. * Expected Output:   + A table mapping operation → recommended algorithm → justification.   + Working Python functions for searching and sorting the inventory.   **PROMPT:**    Implement an Inventory system in Python to handle thousands of products. Each product has: product\_id, name, price, quantity.  Requirements:   * Provide O(1) lookup by product\_id. * Provide case-insensitive lookup by name (returning all matching products). * Provide sorting functions to sort products by price and by quantity (both ascending and descending). * For sorting, provide an implementation using a stable merge\_sort (and allow using Python's built-in sort as an alternative). * Include 3+ assert tests for searching and 3+ for sorting. * Explain and justify the recommended algorithms (mapping operation -> algorithm -> justification).   **CODE:**            **OUTPUT:**    **EXPLANATION:**  Operation → Recommended algorithm → Justification   * Lookup by product ID → Hash table (Python dict) → O(1) average lookup, ideal for exact-id queries and frequent lookups. * Lookup by name → Secondary index: map lower(name) → list of products → Case-insensitive, O(1) to find list; returns multiple matches. If you need approximate or prefix search, use trie or inverted index. * Sort by price/quantity for reports → Timsort (Python built-in sorted() / list.sort()) or Merge Sort → Python sorted() is highly optimized (O(n log n), stable) and often faster in practice; custom merge\_sort is useful if you need to control behavior or demonstrate algorithm. * Large dataset considerations: If dataset is very large and sorting frequently, maintain data in a database with indices and ask DB to sort. For in-memory analytics, use efficient sort + streaming approach.   ✅ Deliverables (For All Tasks)   1. AI-generated prompts for code and test case generation. 2. At least 3 assert test cases for each task. 3. AI-generated initial code and execution screenshots. 4. Analysis of whether code passes all tests. 5. Improved final version with inline comments and explanation. 6. Compiled report (Word/PDF) with prompts, test cases, assertions, code, and output. | | | | | | Week6 - Monday |  |