|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **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** | | | | Dr. Rishabh Mittal | | | | | |
| **Instructor(s)Name** | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  | | --- | | Mr. S Naresh Kumar | | Ms. B. Swathi | | Dr. Sasanko Shekhar Gantayat | | Mr. Md Sallauddin | | Dr. Mathivanan | | Mr. Y Srikanth | | Ms. N Shilpa | | Dr. Rishabh Mittal (Coordinator) | | Dr. R. Prashant Kumar | | Mr. Ankushavali MD | | Mr. B Viswanath | | Ms. Sujitha Reddy | | Ms. A. Anitha | | Ms. M.Madhuri | | Ms. Katherashala Swetha | | Ms. Velpula sumalatha | | Mr. Bingi Raju | | Mr. G. Kranthi | | | | | | | |
| **Course Code** | | | 23CS002PC304 | **Course Title** | | AI Assisted Coding | | | |
| **Year/Sem** | | | III/I | **Regulation** | | R23 | | | |
| **Date and Day**  **of Assignment** | | | Week 6 - Thursday | **Time(s)** | | 23CSBTB01 To 23CSBTB52 | | | |
| **Duration** | | | 2 Hours | **Applicable to**  **Batches** | | All Batches | | | |
| **AssignmentNumber:12.4** (Present assignment number)/**24**(Total number of assignments) | | | | | | | | | |
|  | | | | | | | | | |
|  | **Q.No.** | **Question** | | | | | | ***ExpectedTime***  ***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.   **Learning Outcome**  After completing this assignment, students will be able to:   * Implement classical algorithms with AI assistance * Compare algorithm efficiency using real-world scenarios * Understand when optimization is necessary * Critically evaluate AI-generated suggestions instead of blindly accepting them | | | | | | Week 6 |  |
|  |  | **Task 1: Bubble Sort for Ranking Exam Scores**  **Scenario**  You are working on a **college result processing system** where a small list of student scores needs to be sorted after every internal assessment.  **Task Description**   * Implement **Bubble Sort** in Python to sort a list of student scores. * Use an AI tool to:   + Insert inline comments explaining key operations such as comparisons, swaps, and iteration passes   + Identify early-termination conditions when the list becomes sorted   + Provide a brief time complexity analysis   **Expected Outcome**   * A Bubble Sort implementation with:   + AI-generated comments explaining the logic   + Clear explanation of best, average, and worst-case complexity   + Sample input/output showing sorted scores   **EXPLAINATION:**  Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. It gets its name because smaller elements "bubble" to the top of the list.    **Task 2: Improving Sorting for Nearly Sorted Attendance Records**  **Scenario**  You are maintaining an **attendance system** where student roll numbers are already *almost sorted*, with only a few late updates.  **Task Description**   * Start with a Bubble Sort implementation. * Ask AI to:   + Review the problem and suggest a more suitable sorting algorithm   + Generate an **Insertion Sort** implementation   + Explain why Insertion Sort performs better on nearly sorted data * Compare execution behavior on nearly sorted input   **Expected Outcome**   * Two sorting implementations:   + Bubble Sort   + Insertion Sort * AI-assisted explanation highlighting efficiency differences for partially sorted datasets   **EXPLAINATION:** **Insertion Sort for Nearly Sorted Data:** Insertion Sort performs very well on nearly sorted data because it only shifts elements that are larger than the current element (the key) to the right. If the array is almost sorted, few elements need to be moved, resulting in fewer comparisons and swaps. Its best-case time complexity is O(n), making it efficient for such scenarios.    **Task 3: Searching Student Records in a Database**  **Scenario**  You are developing a **student information portal** where users search for student records by roll number.  **Task Description**   * Implement:   + **Linear Search** for unsorted student data   + **Binary Search** for sorted student data * Use AI to:   + Add docstrings explaining parameters and return values   + Explain when Binary Search is applicable   + Highlight performance differences between the two searches   **Expected Outcome**   * Two working search implementations with docstrings * AI-generated explanation of:   + Time complexity   + Use cases for Linear vs Binary Search * A short student observation comparing results on sorted vs unsorted lists   **EXPLAINATION:** **Linear Search** checks each element sequentially until the target is found or the list ends. It works on both sorted and unsorted data but can be slow for large lists. **Binary Search** is much faster for large lists, but it requires the list to be sorted. It works by repeatedly dividing the search interval in half until the target is found or the interval becomes empty.        **Task 4: Choosing Between Quick Sort and Merge Sort for Data Processing**  **Scenario**  You are part of a **data analytics team** that needs to sort large datasets received from different sources (random order, already sorted, and reverse sorted).  **Task Description**   * Provide AI with partially written recursive functions for:   + **Quick Sort**   + **Merge Sort** * Ask AI to:   + Complete the recursive logic   + Add meaningful docstrings   + Explain how recursion works in each algorithm * Test both algorithms on:   + Random data   + Sorted data   + Reverse-sorted data   **Expected Outcome**   * Fully functional Quick Sort and Merge Sort implementations * AI-generated comparison covering:   + Best, average, and worst-case complexities   + Practical scenarios where one algorithm is preferred over the other * EXPLAINATION: **Quick Sort:**   + **In-place (mostly):** While the recursive calls do use stack space, the partitioning step can often be done in-place, reducing auxiliary space requirements compared to Merge Sort.   + **Performance:** Generally faster in practice due to better constant factors and cache performance, especially on average.   + **Worst Case:** Its worst-case O(n^2) time complexity occurs when the pivot selection consistently leads to highly unbalanced partitions (e.g., already sorted array with first/last element as pivot). * **Merge Sort:**   + **Stable:** Preserves the relative order of equal elements.   + **Guaranteed Performance:** Consistently performs at O(n log n) in all cases (best, average, worst) because it divides the list into halves, ensuring balanced partitions.   + **Space Complexity:** Requires O(n) auxiliary space due to the need for temporary arrays during the merging step, which can be a disadvantage for very large datasets.       **Task 5: Optimizing a Duplicate Detection Algorithm**  **Scenario**  You are building a **data validation module** that must detect duplicate user IDs in a large dataset before importing it into a system.  **Task Description**   * Write a **naive duplicate detection algorithm** using nested loops. * Use AI to:   + Analyze the time complexity   + Suggest an optimized approach using sets or dictionaries   + Rewrite the algorithm with improved efficiency * Compare execution behavior conceptually for large input sizes   **Expected Outcome**   * Two versions of the algorithm:   + Brute-force (O(n²))   + Optimized (O(n)) * AI-assisted explanation showing how and why performance improved   **EXPLAINATION:**  **Brute-force** duplicate detection involves checking every element against every other element, leading to slow performance for large lists. The **optimized version** uses a hash set (Python's set) to keep track of elements encountered. This allows for constant-time average-case lookups, dramatically speeding up the detection process by avoiding redundant comparisons.    **Note: Report should be submitted a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots** | | | | | |  |  |