

Name:I. Abhinay H.No:2303A51811 Batch:26

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: <b>12.4</b> (Present assignment number)/ <b>24</b> (Total number of assignments)			
Q.No.	Question		Expected Time to complete
1	Lab 12 – Algorithms with AI Assistance – Sorting, Searching, and Optimizing Algorithms		Week 6

	<p><b>Lab Objectives</b></p> <ul style="list-style-type: none"> <li>• Apply AI-assisted programming to implement and optimize sorting and searching algorithms.</li> <li>• Compare different algorithms in terms of efficiency and use cases.</li> <li>• Understand how AI tools can suggest optimized code and complexity improvements.</li> </ul> <p><b>Learning Outcome</b></p> <p>After completing this assignment, students will be able to:</p> <ul style="list-style-type: none"> <li>• Implement classical algorithms with AI assistance</li> <li>• Compare algorithm efficiency using real-world scenarios</li> <li>• Understand when optimization is necessary</li> <li>• Critically evaluate AI-generated suggestions instead of blindly accepting them</li> </ul>	
	<h2>Task 1: Bubble Sort for Ranking Exam Scores</h2> <p><b>Scenario</b></p> <p>You are working on a <b>college result processing system</b> where a small list of student scores needs to be sorted after every internal assessment.</p> <p><b>Task Description</b></p> <ul style="list-style-type: none"> <li>• Implement <b>Bubble Sort</b> in Python to sort a list of student scores.</li> <li>• Use an AI tool to: <ul style="list-style-type: none"> <li>○ Insert inline comments explaining key operations such as comparisons, swaps, and iteration passes</li> <li>○ Identify early-termination conditions when the list becomes sorted</li> <li>○ Provide a brief time complexity analysis</li> </ul> </li> </ul> <p><b>Expected Outcome</b></p> <ul style="list-style-type: none"> <li>• A Bubble Sort implementation with: <ul style="list-style-type: none"> <li>○ AI-generated comments explaining the logic</li> <li>○ Clear explanation of best, average, and worst-case complexity</li> <li>○ Sample input/output showing sorted scores</li> </ul> </li> </ul>	

```

1 def bubble_sort(scores):
2     n = len(scores)
3     for i in range(n - 1):
4         swapped = False
5         for j in range(n - 1 - i):
6             if scores[j] > scores[j + 1]:
7                 scores[j], scores[j + 1] = scores[j + 1], scores[j]
8             swapped = True
9     if not swapped:
10        break
11    return scores
12
13 scores = [85, 42, 97, 63, 51, 74, 88]
14 print("Before:", scores)
15 print("After:", bubble_sort(scores))

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Terminal output:

```

PS C:\Users\shash\Downloads & "c:\Users\shash\anaconda\envs\shashidhar\python.exe" "c:\Users\shash\vscode\extensions\ms-python.debug-2025.18.0-win32-x64\bundled\libs\debug\launcher" "53307" -- "c:\Users\shash\Downloads\AAC A 12.4.py"
Before: [85, 42, 97, 63, 51, 74, 88]
After: [42, 51, 63, 74, 85, 88, 97]

```

## 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

The screenshot shows a code editor interface with a Python file named `AAC A 12.4.py`. The code contains two sorting functions: `bubble_sort` and `insertion_sort`. The `bubble_sort` function uses nested loops to compare adjacent elements and swap them if they are in the wrong order. The `insertion_sort` function iterates through the array, selecting each element as a 'key' and shifting the previous elements to make space for it if it is smaller than its predecessor.

```

C:\> Users > shash > Downloads > AAC A 12.4.py > ...
1 import time
2 def bubble_sort(arr):
3     n = len(arr)
4     for i in range(n - 1):
5         swapped = False
6         for j in range(n - 1 - i):
7             if arr[j] > arr[j + 1]:
8                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
9             swapped = True
10        if not swapped:
11            break
12    return arr
13
14 def insertion_sort(arr):
15    for i in range(1, len(arr)):
16        key = arr[i]
17        j = i - 1
18        while j >= 0 and arr[j] > key:
19            arr[j + 1] = arr[j]
20            j -= 1
21        arr[j + 1] = key
22    return arr
23
24 nearly_sorted = [1, 2, 3, 4, 5, 6, 8, 7, 9, 10]

```

The terminal below shows the execution of the script with nearly sorted input. It prints the 'Before' state ([42, 51, 63, 74, 85, 88, 97]), the 'After' state ([42, 51, 63, 74, 85, 88, 97]), and the execution times for both sorts.

```

PS C:\Users\shash\Downloads> & 'c:\Users\shash\anaconda3\envs\ShashiDhar\python.exe' 'c:\Users\shash\vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '53907' -- 'c:\Users\shash\Downloads\AAC A 12.4.py'
Before: [42, 51, 63, 74, 85, 88, 97]
After: [42, 51, 63, 74, 85, 88, 97]
Bubble Sort Time: 0.0001785
Insertion Sort Time: 0.0001245

```

An AI-assisted explanation panel on the right provides context for the sorting methods:

- Used 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
- give simple code in python without comments

The status bar at the bottom indicates the file has been reviewed and updated, and the AI has restored the code.

The screenshot shows a code editor with a dark theme. The top bar has tabs for "Welcome" and "AAC A 12.4.py". The code in "AAC A 12.4.py" is as follows:

```
C: > Users > shash > Downloads > AAC A 12.4.py > ...
14     def insertion_sort(arr):
15         j = i - 1
16         while j >= 0 and arr[j] > key:
17             arr[j + 1] = arr[j]
18             j -= 1
19         arr[j + 1] = key
20
21     return arr
22
23
24 nearly_sorted = [1, 2, 4, 3, 5, 6, 8, 7, 9, 10]
25
26 print("Nearly Sorted Input:", nearly_sorted)
27
28 data1 = nearly_sorted.copy()
29 start = time.perf_counter()
30 print("Bubble Sort Result:", bubble_sort(data1))
31 print(f"Bubble Sort Time: {time.perf_counter() - start:.6f}s")
32
33 data2 = nearly_sorted.copy()
34 start = time.perf_counter()
35 print("Insertion Sort Result:", insertion_sort(data2))
36 print(f"Insertion Sort Time: {time.perf_counter() - start:.6f}s")
```

Below the code editor is a terminal window showing the execution of the script:

- PS C:\Users\shash\Downloads> & 'c:\Users\shash\anaconda3\envs\Shashidhar\python .18.0-win32-x64\bundled\libs\debugpy\launcher' '53307' '--' 'c:\Users\shash\Downloads\AAC A 12.4.py'
- Before: [85, 42, 97, 63, 51, 74, 88]
- After: [42, 51, 63, 74, 85, 88, 97]
- PS C:\Users\shash\Downloads> c:; cd 'c:\Users\shash\Downloads'; & 'c:\Users\shash\anaconda3\envs\Shashidhar\python .18.0-win32-x64\bundled\libs\debugpy\launcher' '53307' '--' 'c:\Users\shash\Downloads\AAC A 12.4.py'
- Nearly Sorted Input: [1, 2, 4, 3, 5, 6, 8, 7, 9, 10]
- Bubble Sort Result: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
- Bubble Sort Time: 0.000178s
- Insertion Sort Result: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
- Insertion Sort Time: 0.000124s

## 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

The screenshot shows a Jupyter Notebook interface with the following components:

- Code Cell:** Contains Python code for implementing Linear Search and Binary Search, and demonstrates their use on an unsorted list.
- Output Cell:** Shows the execution results, including the sorted list, the target value (115), and the search results for both methods.
- Terminal:** Shows the command-line execution of the code, resulting in the same output as the notebook cells.
- Performance Comparison Table:** A table comparing the time complexities of Linear and Binary Search under different conditions.
- When to use which:** A section listing the pros and cons of each search method.
- Describe what to build next:** A text input field for future tasks.

```

Welcome   AAC A 124.py X
C:\Users\shash\Downloads> AAC A 124.py > linear_search > roll_numbers
1 def linear_search(roll_numbers, target):
2     for i in range(len(roll_numbers)):
3         if roll_numbers[i] == target:
4             return i
5     return -1
6
7 def binary_search(roll_numbers, target):
8     low, high = 0, len(roll_numbers) - 1
9     while low <= high:
10         mid = (low + high) // 2
11         if roll_numbers[mid] == target:
12             return mid
13         elif roll_numbers[mid] < target:
14             low = mid + 1
15         else:
16             high = mid - 1
17     return -1
18
19 unsorted = [112, 105, 121, 108, 115, 103, 119]
20 sorted_list = [103, 105, 108, 112, 115, 119, 121]
21 target = 115
22
23 print("Unsorted:", unsorted)
24 print("Sorted:", sorted_list)
25
26 print("Searching for roll number: 115")
27 print("Linear Search: Found at index 4")
28 print("Binary Search: Found at index 4")
29
30 print("Searching for roll number: 999")
31 print("Linear Search: Not Found")
32 print("Binary Search: Not Found")

```

	Linear Search	Binary Search
Best	$O(1)$	$O(1)$
Average	$O(n)$	$O(\log n)$
Worst	$O(n)$	$O(\log n)$
Requires sorted data	No	Yes

When to use which:

- **Linear Search** — works on unsorted data; suitable when data changes frequently and sorting cost is not justified.

The screenshot shows a VS Code interface with the following components:

- Code Editor:** Displays a Python script comparing Linear Search and Binary Search. The script defines two lists: `unsorted` and `sorted_list`, and a target value `target`. It then prints the lists, performs linear search, and binary search for the target value.
- Terminal:** Shows the output of running the script. The terminal window title is "Python Debug". The output includes the lists, the target value, and the results of the search operations.
- Performance Comparison:** A table comparing the characteristics of Linear Search and Binary Search. The table has columns for "Linear Search" and "Binary Search".
- When to use which:** A list of bullet points explaining the appropriate use cases for each search algorithm.

	Linear Search	Binary Search
Best	$O(1)$	$O(1)$
Average	$O(n)$	$O(\log n)$
Worst	$O(n)$	$O(\log n)$
Requires sorted data	No	Yes

**When to use which:**

- **Linear Search** — works on unsorted data; suitable when data changes frequently and sorting cost is not justified.
- **Binary Search** — requires sorted data; far more efficient for large, static datasets like a pre-sorted student database.

## 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

C:\> Users > shash > Downloads > AAC A 12.4.py > ...

```

1 import time, random
2 def quick_sort(arr):
3     if len(arr) <= 1:
4         return arr
5     pivot = arr[len(arr) // 2]
6     left = [x for x in arr if x < pivot]
7     middle = [x for x in arr if x == pivot]
8     right = [x for x in arr if x > pivot]
9     return quick_sort(left) + middle + quick_sort(right)
10
11 def merge_sort(arr):
12     if len(arr) <= 1:
13         return arr
14     mid = len(arr) // 2
15     left = merge_sort(arr[:mid])
16     right = merge_sort(arr[mid:])
17     return merge(left, right)
18
19 def merge(left, right):
20     result = []
21     i = j = 0
22     while i < len(left) and j < len(right):
23         if left[i] <= right[j]:
24             result.append(left[i])
25         else:
26             result.append(right[j])
27         i += 1
28         j += 1
29     result.extend(left[i:])
30     result.extend(right[j:])
31     return result
32
33 def test_sort(name, sort_func, data):
34     start = time.perf_counter()
35     result = sort_func(data.copy())
36     elapsed = time.perf_counter() - start
37     print(f" {name}: ({elapsed:.6f}s) {result[:5]}...{result[-5:]}" )
38     random_data = random.sample(range(1, 1001), 500)
39     sorted_data = list(range(1, 501))
40     reverse_data = list(range(500, 0, -1))
41     for label, data in [("Random", random_data), ("Sorted", sorted_data), ("Reverse", reverse_data)]:
42         print(f"\n{label}:")
43         test.sort("Quick Sort", quick_sort, data)
44         test.sort("Merge Sort", merge_sort, data)

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\shash\Downloads> cd ..> c:\Users\shash\Downloads ; & "c:\Users\shash\anaconda3\envs\Shashidhar\python.exe" "c:\Users\shash\vscode\extensions\ms-python.debugger-2025.18.0-win32-x64\bundled\libs\debug\launcher" "49685" -- "c:\Users\shash\Downloads\AAC A 12.4.py"

Random Data:

Quick Sort: 0.000821s -> [5, 10, 11, 15, 16]...[996, 997, 998]

Merge Sort: 0.001611s -> [5, 10, 11, 15, 16]...[996, 997, 998]

Sorted Data:

Quick Sort: 0.000537s -> [1, 2, 3, 4, 5]...[498, 499, 500]

Merge Sort: 0.001283s -> [1, 2, 3, 4, 5]...[498, 499, 500]

Reverse Data:

Quick Sort: 0.000877s -> [1, 2, 3, 4, 5]...[498, 499, 500]

C:\> Users > shash > Downloads > AAC A 12.4.py > merge

```

19 def merge(left, right):
20     while i < len(left) and j < len(right):
21         if left[i] <= right[j]:
22             result.append(left[i])
23             i += 1
24         else:
25             result.append(right[j])
26             j += 1
27     result.extend(left[i:])
28     result.extend(right[j:])
29     return result
30
31 def test_sort(name, sort_func, data):
32     start = time.perf_counter()
33     result = sort_func(data.copy())
34     elapsed = time.perf_counter() - start
35     print(f" {name}: ({elapsed:.6f}s) {result[:5]}...{result[-5:]}" )
36     random_data = random.sample(range(1, 1001), 500)
37     sorted_data = list(range(1, 501))
38     reverse_data = list(range(500, 0, -1))
39     for label, data in [("Random", random_data), ("Sorted", sorted_data), ("Reverse", reverse_data)]:
40         print(f"\n{label}:")
41         test.sort("Quick Sort", quick_sort, data)
42         test.sort("Merge Sort", merge_sort, data)
43

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\shash\Downloads> cd ..> c:\Users\shash\Downloads ; & "c:\Users\shash\anaconda3\envs\Shashidhar\python.exe" "c:\Users\shash\vscode\extensions\ms-python.debugger-2025.18.0-win32-x64\bundled\libs\debug\launcher" "49685" -- "c:\Users\shash\Downloads\AAC A 12.4.py"

Sorted Data:

Quick Sort: 0.000537s -> [1, 2, 3, 4, 5]...[498, 499, 500]

Merge Sort: 0.001611s -> [1, 2, 3, 4, 5]...[498, 499, 500]

Reverse Data:

Quick Sort: 0.000877s -> [1, 2, 3, 4, 5]...[498, 499, 500]

Merge Sort: 0.001465s -> [1, 2, 3, 4, 5]...[498, 499, 500]

# 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 ( $O(n^2)$ )
    - Optimized ( $O(n)$ )
  - AI-assisted explanation showing how and why performance improved

VS Code Editor showing code for finding duplicates in user IDs.

```

19
20     user_ids = [101, 203, 305, 101, 407, 203, 509, 305, 611, 407]
21     print("User IDs:", user_ids)
22
23     start = time.perf_counter()
24     result1 = find_duplicates_brute(user_ids)
25     t1 = time.perf_counter() - start
26     print(f"\nBrute Force O(n2): {result1} Time: {t1:.6f}s")
27
28     start = time.perf_counter()
29     result2 = find_duplicates_optimized(user_ids)
30     t2 = time.perf_counter() - start
31     print(f"Optimized O(n): {result2} Time: {t2:.6f}s")
32     large = random.choices(range(1, 5001), k=10000)
33     start = time.perf_counter()
34     find_duplicates_brute(large)
35     t1 = time.perf_counter() - start
36     start = time.perf_counter()
37     find_duplicates_optimized(large)
38     t2 = time.perf_counter() - start
39     print(f"\nLarge dataset (10000 IDs):")
40     print(f"Brute Force: {t1:.4f}s")
41     print(f"Optimized: {t2:.4f}s")

```

TERMINAL OUTPUT:

- PS C:\Users\shash\Downloads> c:, cu C:\Users\shash\Downloads , & C:\Users\shash\anaconda3\envs\py37\python.exe\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '56364
- PS C:\Users\shash\Downloads> c:; cd 'c:\Users\shash\Downloads'; & 'c:\Users\shash\anaconda3\envs\py37\python.exe\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '64204
- User IDs: [101, 203, 305, 101, 407, 203, 509, 305, 611, 407]
- Brute Force O(n<sup>2</sup>): [101, 203, 305, 407] Time: 0.000036s
- Optimized O(n): [305, 203, 101, 407] Time: 0.000022s
- Large dataset (10000 IDs):
- Brute Force: 3.2178s
- Optimized: 0.0009s

PS C:\Users\shash\Downloads>

**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**