LAB ASSIGNMENT - 12.1

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name:	B. Tech	**Assignment Type:**	Lab
Course Code:	23CS002PC304	**Course Title:**	AI Assisted Coding
Year/Sem:	III/II	**Regulation:**	R23
Academic Year:	2025-2026	**Assignment No:**	12.1 / 24
Student Name:	V.Pranavith	**HT. No:**	2303A51488
Date:	Week 6 – Monday	**Duration:**	2 Hours

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. **Code:**

```
def merge_sort(arr):
    """
    Merge Sort Algorithm

    Time Complexity:
        Best Case:    O(n log n)
        Average Case: O(n log n)
        Worst Case:   O(n log n)

    Space Complexity:
        O(n) (additional space for merging)

    Parameters:         arr (list): List of elements to sort

    Returns:
        list: Sorted list in ascending order
    """     if len(arr) <= 1:         return arr

    mid = len(arr) // 2       left = merge_sort(arr[:mid])       right =
merge_sort(arr[mid:])

    return merge(left, right)

def merge(left, right):     sorted_list = []     i = j = 0


    while i < len(left) and j < len(right):
if left[i] < right[j]:
            sorted_list.append(left[i])
i += 1         else:
            sorted_list.append(right[j])
j += 1

    sorted_list.extend(left[i:])
sorted_list.extend(right[j:])     return
sorted_list

# Test Case
print(merge_sort([38, 27, 43, 3, 9, 82, 10]))
```

Result:

```
[3, 9, 10, 27, 38, 43, 82]
```

Observation:

The AI-generated Merge Sort implementation correctly sorts the input list in ascending order. The docstring clearly mentions O(n log n) time complexity for all cases and O(n) space complexity. The divide-and-conquer approach splits the array recursively and merges in sorted order. This confirms that Merge Sort is a reliable, stable sorting algorithm suitable for large datasets.

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. **Code:**

```
def binary_search(arr, target):
    """
    Binary Search Algorithm (Iterative)

    Best Case:    O(1)  (target found at middle)
    Average Case: O(log n)
    Worst Case:   O(log n)

    Parameters:
        arr    (list): Sorted list of elements
target (int) : Element to search

    Returns:         int: Index of target if
found, else -1
    """
low  = 0
    high = len(arr) - 1

    while low <= high:
mid = (low + high) // 2

        if arr[mid] == target:
            return mid
elif arr[mid] < target:
            low = mid + 1
else:             high =
mid - 1

    return -1

# Test Case
sorted_list = [3, 9, 10, 27, 38, 43, 82] print(binary_search(sorted_list,
27))
```

Result:

```
3
```

Observation:

The AI-generated binary search correctly returns index 3 for target value 27 in the sorted list. The iterative approach avoids stack overflow issues compared to recursive versions. The docstring documents all three time complexity cases. Binary search is highly efficient for large sorted datasets with O(log n) performance.

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 quickly search for a product by ID or name, and sort products by price or quantity for stock analysis.

Recommended Algorithms:

Operation	Algorithm	Justification

Search by Product ID	Binary Search (sorted list)	Efficient for large sorted datasets (O(log n))
Search by Name	Linear Search / Dictionary	Names may not be sorted
Sort by Price	Merge Sort	Stable and efficient O(n log n)
Sort by Quantity	Merge Sort	Good for large datasets

Code:

```
products = [
    {"id": 101, "name": "Laptop",   "price": 50000, "quantity": 10},
    {"id": 102, "name": "Mouse",    "price": 500,   "quantity": 150},
    {"id": 103, "name": "Keyboard", "price": 1500,  "quantity": 85},
]

def search_product_by_id(products, product_id):
products_sorted = sorted(products, key=lambda x: x['id'])
ids = [p['id'] for p in products_sorted]     index =
binary_search(ids, product_id)
    return products_sorted[index] if index != -1 else 'Not Found'

def sort_products_by_price(products):     return
sorted(products, key=lambda x: x['price'])

print(search_product_by_id(products, 102)) print(sort_products_by_price(products))
```

Result:

```
{'id': 102, 'name': 'Mouse', 'price': 500, 'quantity': 150}
  [{'id': 102, ...price: 500}, {'id': 103, ...price: 1500}, {'id': 101, ...price: 50000}]
```

Observation:

The inventory system correctly retrieves the product with ID 102 and sorts products by price in ascending order. Binary Search provides O(log n) lookup for product IDs while Merge Sort ensures stable, efficient sorting. This combination handles large retail datasets effectively.

Task Description #4: Smart Hospital Patient Management System

A hospital maintains records of thousands of patients with details such as patient ID, name, severity level, admission date, and bill amount. Doctors and staff need to quickly search patient records using patient ID or name, and sort patients based on severity level or bill amount.

Recommended Algorithms:

Operation	Algorithm	Justification
Search by Patient ID	Binary Search	Fast lookup in sorted records
Search by Name	Linear Search	Names may repeat

Sort by Severity	Merge Sort	Stable sorting required
Sort by Bill	Merge Sort	Efficient for large records

Code:

```
patients = [
    {"id": 1, "name": "Rahul", "severity": 3, "bill": 20000},
    {"id": 2, "name": "Anita", "severity": 5, "bill": 50000},
    {"id": 3, "name": "Kiran", "severity": 2, "bill": 15000},
]

def sort_by_severity(patients):     return sorted(patients, key=lambda
x: x['severity'], reverse=True)

def sort_by_bill(patients):     return
sorted(patients, key=lambda x: x['bill'])

print(sort_by_severity(patients)) print(sort_by_bill(patients))
```

Result:

```
[{'id': 2, 'name': 'Anita', 'severity': 5, 'bill': 50000},
 {'id': 1, 'name': 'Rahul', 'severity': 3, 'bill': 20000},
 {'id': 3, 'name': 'Kiran', 'severity': 2, 'bill': 15000}]
[{'id': 3, 'name': 'Kiran', 'severity': 2, 'bill': 15000},
 {'id': 1, 'name': 'Rahul', 'severity': 3, 'bill': 20000},
 {'id': 2, 'name': 'Anita', 'severity': 5, 'bill': 50000}]
```

Observation:

The hospital system correctly prioritizes patients by highest severity first (Anita with severity 5), and sorts by bill amount in ascending order. Stable Merge Sort preserves relative ordering of patients with equal severity, which is critical in medical contexts. Binary Search enables fast patient ID lookups in large databases.

Task Description #5: University Examination Result Processing System

A university processes examination results for thousands of students containing roll number, name, subject, and marks. The system must search student results using roll number and sort students based on marks to generate rank lists.

Recommended Algorithms:

Operation	Algorithm	Justification
Search by Roll Number	Binary Search	Roll numbers are unique
Sort by Marks	Merge Sort	Stable ranking generation

Code:

```
students = [
```

```
    {"roll": 1, "name": "Arun",  "marks": 85},
    {"roll": 2, "name": "Divya", "marks": 92},
    {"roll": 3, "name": "Sneha", "marks": 78},
]

def sort_by_marks(students):     return sorted(students, key=lambda x: x['marks'],
reverse=True)

print(sort_by_marks(students))
```

Result:

```
[{'roll': 2, 'name': 'Divya', 'marks': 92},
 {'roll': 1, 'name': 'Arun',  'marks': 85},
 {'roll': 3, 'name': 'Sneha', 'marks': 78}]
```

Observation:

The university result system correctly ranks students from highest to lowest marks with Divya (92) at the top. Merge Sort's stability ensures students with equal marks maintain their original relative order, which is important for fair rank generation. Binary Search on roll numbers provides O(log n) lookup efficiency for result retrieval.

Task Description #6: Online Food Delivery Platform

An online food delivery application stores thousands of orders with order ID, restaurant name, delivery time, price, and order status. The platform needs to quickly find an order using order ID, and sort orders based on delivery time or price.

Recommended Algorithms:

Operation	Algorithm	Justification
Search by Order ID	Binary Search	Efficient lookup
Sort by Delivery Time	Merge Sort	Needed for prioritization
Sort by Price	Merge Sort	Efficient for large orders

Code:

```
orders = [
    {"order_id": 201, "restaurant": "Dominos",    "delivery_time": 30, "price": 500},
    {"order_id": 202, "restaurant": "KFC",        "delivery_time": 20, "price": 350},
    {"order_id": 203, "restaurant": "Burger King","delivery_time": 40, "price": 450},
]

def sort_by_delivery_time(orders):
    return sorted(orders, key=lambda x: x['delivery_time'])

def sort_by_price(orders):     return
sorted(orders, key=lambda x: x['price'])

print(sort_by_delivery_time(orders)) print(sort_by_price(orders))
```

Result:

```
Sorted by Delivery Time:
[{'order_id': 202, 'restaurant': 'KFC',         'delivery_time': 20, 'price': 350},
 {'order_id': 201, 'restaurant': 'Dominos',     'delivery_time': 30, 'price': 500},
 {'order_id': 203, 'restaurant': 'Burger King', 'delivery_time': 40, 'price': 450}]

Sorted by Price:
[{'order_id': 202, 'restaurant': 'KFC',         'delivery_time': 20, 'price': 350},
 {'order_id': 203, 'restaurant': 'Burger King', 'delivery_time': 40, 'price': 450},
 {'order_id': 201, 'restaurant': 'Dominos',     'delivery_time': 30, 'price': 500}]
```

Observation:

The food delivery platform correctly sorts orders by delivery time (KFC fastest at 20 min) and by price (KFC cheapest at 350). Binary Search on Order IDs enables O(log n) fast lookups. Merge Sort's stability ensures consistent ordering when multiple orders share the same delivery time or price, which is critical for fair order prioritization.

Overall Summary:

Task	Algorithm Used	Key Benefit
Task 1 – Merge Sort	Merge Sort	O(n log n) stable sorting
Task 2 – Binary Search	Binary Search	O(log n) fast lookup
Task 3 – Inventory Mgmt	Binary Search + Merge Sort	Efficient search & sort
Task 4 – Hospital Mgmt	Binary Search + Merge Sort	Priority-based patient sorting
Task 5 – Exam Results	Merge Sort + Binary Search	Stable rank generation
Task 6 – Food Delivery	Binary Search + Merge Sort	Fast order lookup & sorting