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	
CourseCode	23CS002PC304	**Course Title**	AI Assisted Coding
Year/Sem	III/II	**Regulation**	R23
Date and Day of Assignment	**Week6 – Wednusday**	**Time(s)**	23CSBTB01 To 23CSBTB52
Duration	2 Hours	**Applicable to Batches**	All batches
Assignment Number:12.3(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 Algorithm Optimization Using AI Tools**	Week3 -

	Lab Objectives The objectives of this laboratory exercise are to: • Apply AI-assisted programming techniques to implement sorting and searching algorithms. • Analyze and compare algorithm efficiency using time and space complexity. • Understand how AI tools can suggest optimizations and alternative algorithmic approaches. • Strengthen problem-solving skills through real-world, data-driven scenarios. --- **Learning Outcomes** After completing this lab, students will be able to: • Implement and optimize classic algorithms using AI-assisted coding tools. • Compare multiple algorithms for the same problem and justify their selection. • Measure and analyze runtime performance using experimental data. • Critically review and refine AI-generated algorithmic solutions. --- **Task 1: Sorting Student Records for Placement Drive** **Scenario** SR University's Training and Placement Cell needs to shortlist candidates efficiently during campus placements. Student records must be sorted by **CGPA in descending order**. **Tasks** 1. Use **GitHub Copilot** to generate a program that stores student records (Name, Roll Number, CGPA). 2. Implement the following sorting algorithms using AI assistance: o Quick Sort o Merge Sort 3. Measure and compare runtime performance for large datasets. 4. Write a function to display the **top 10 students** based on CGPA. **Expected Outcome** • Correctly sorted student records. • Performance comparison between Quick Sort and Merge Sort. • Clear output of top-performing students. **Prompt:** #Generate a Python program to manage student placement records (name, roll number, CGPA), implement Quick Sort and Merge Sort to sort students by CGPA in descending order, compare their runtime performance on large datasets, and display the top 10 students with clear output and comments.	Wednesday

```python
#generate a Python program to manage student placement records (name, roll number, CGPA), impler
import random
import time
class Student:
    def __init__(self, name, roll_number, cgpa):
        self.name = name
        self.roll_number = roll_number
        self.cgpa = cgpa
def quick_sort(students):
    if len(students) <= 1:
        return students
    pivot = students[len(students) // 2].cgpa
    left = [x for x in students if x.cgpa > pivot]
    middle = [x for x in students if x.cgpa == pivot]
    right = [x for x in students if x.cgpa < pivot]
    return quick_sort(left) + middle + quick_sort(right)
def merge_sort(students):
    if len(students) <= 1:
        return students
    mid = len(students) // 2
    left = merge_sort(students[:mid])
    right = merge_sort(students[mid:])
    return merge(left, right)
def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i].cgpa > right[j].cgpa:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
```

```
    result.extend(right[j:])
    return result
def generate_students(num_students):
    students = []
    for i in range(num_students):
        name = f"Student_{i+1}"
        roll_number = f"RN{i+1:03d}"
        cgpa = round(random.uniform(0, 10), 2)
        students.append(Student(name, roll_number, cgpa))
    return students
def display_top_students(students, top_n=10):
    print(f"Top {top_n} Students:")
    print(f"{'Name':<15} {'Roll Number':<15} {'CGPA':<5}")
    for student in students[:top_n]:
        print(f"{student.name:<15} {student.roll_number:<15} {student.cgpa:<5}")
# Generate a large dataset of students
num_students = 1000
students = generate_students(num_students)
# Sort using Quick Sort and measure runtime
start_time = time.time()
sorted_students_quick = quick_sort(students)
end_time = time.time()
print(f"Quick Sort Runtime: {end_time - start_time:.4f} seconds")
# Sort using Merge Sort and measure runtime
start_time = time.time()
sorted_students_merge = merge_sort(students)
end_time = time.time()
print(f"Merge Sort Runtime: {end_time - start_time:.4f} seconds")
# Display the top 10 students sorted by CGPA
display_top_students(sorted_students_quick)
display_top_students(sorted_students_merge)
```

```
Student_314     RN314          9.99
Student_41      RN041          9.99
Student_496     RN496          9.94
Student_207     RN207          9.93
Student_767     RN767          9.91
Student_754     RN754          9.9
Student_635     RN635          9.89
Student_185     RN185          9.88
Student_132     RN132          9.88
```

Task 2: Implementing Bubble Sort with AI Comments

- **Task**: Write a Python implementation of **Bubble Sort**.
- **Instructions**:
    - Students implement Bubble Sort normally.
    - Ask AI to generate **inline comments explaining key logic** (like

swapping, passes, and termination).

- Request AI to provide **time complexity analysis**.
- **Expected Output**:
  - A Bubble Sort implementation with AI-generated explanatory comments and complexity analysis.

PROMPT:

#Generate a Python implementation of Bubble Sort with clear inline comments explaining the swapping process, passes, and termination condition, and include a brief time complexity analysis

```python
#Generate a Python implementation of Bubble Sort with clear inline comments explaining the swapping process, passes,
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all elements in the array
    for i in range(n):
        # Initialize a flag to check if any swapping occurs
        swapped = False
        # Last i elements are already in place, no need to check them
        for j in range(0, n-i-1):
            # Swap if the element found is greater than the next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]  # Swapping process
                swapped = True  # Set the flag to True if a swap occurred
        # If no swapping occurred, the array is already sorted
        if not swapped:
            break  # Termination condition: exit the loop if the array is sorted
# Example usage
arr = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(arr)
print("Sorted array is:", arr)
```

Output:

```
Sorted array is: [11, 12, 22, 25, 34, 64, 90]
```

Task 3: Quick Sort and Merge Sort Comparison

- **Task:** Implement Quick Sort and Merge Sort using recursion**.**
- **Instructions:**
  - Provide AI with partially completed functions for recursion.
  - Ask AI to complete the missing logic and add docstrings.
  - Compare both algorithms on random, sorted, and reverse-sorted lists.
- **Expected Output:**
  - Working Quick Sort and Merge Sort implementations.
  - AI-generated explanation of average, best, and worst-case complexities.

PROMPT:

#Complete the recursive Python implementations of Quick Sort and Merge Sort by filling in the missing logic, add clear docstrings, compare their performance on random, sorted, and reverse-sorted lists, and explain their average, best, and worst-case time

	complexities.	

```python
#Complete the recursive Python implementations of Quick Sort and Merge Sort by filling in the missing logic, add c
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]  # Choosing the middle element as the pivot
    left = [x for x in arr if x < pivot]  # Elements less than the pivot
    middle = [x for x in arr if x == pivot]  # Elements equal to the pivot
    right = [x for x in arr if x > pivot]  # Elements greater than the pivot
    return quick_sort(left) + middle + quick_sort(right)  # Recursively sort and combine
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2  # Finding the mid of the array
    left = merge_sort(arr[:mid])  # Recursively sort the left half
    right = merge_sort(arr[mid:])  # Recursively sort the right half
    return merge(left, right)  # Merge the sorted halves
def merge(left, right):
    """Merges two sorted lists into a single sorted list.
    Parameters:
    left (list): The first sorted list.
    right (list): The second sorted list.
    Returns:
    list: A merged and sorted list.
    """
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])  # Append the smaller element
            i += 1
        else:
            result.append(right[j])  # Append the smaller element
            j += 1
```

```python
    result.extend(left[i:])  # Append any remaining elements from left
    result.extend(right[j:])  # Append any remaining elements from right
    return result
# Performance comparison
import random
# Generate random, sorted, and reverse-sorted lists
random_list = [random.randint(1, 1000) for _ in range(1000)]
sorted_list = sorted(random_list)
reverse_sorted_list = sorted(random_list, reverse=True)
# Compare Quick Sort performance
import time
start_time = time.time()
quick_sort(random_list)
print(f"Quick Sort on random list took {time.time() - start_time:.4f} seconds")
start_time = time.time()
quick_sort(sorted_list)
print(f"Quick Sort on sorted list took {time.time() - start_time:.4f} seconds")
start_time = time.time()
quick_sort(reverse_sorted_list)
print(f"Quick Sort on reverse-sorted list took {time.time() - start_time:.4f} seconds")
# Compare Merge Sort performance
start_time = time.time()
merge_sort(random_list)
print(f"Merge Sort on random list took {time.time() - start_time:.4f} seconds")
start_time = time.time()
merge_sort(sorted_list)
print(f"Merge Sort on sorted list took {time.time() - start_time:.4f} seconds")
start_time = time.time()
merge_sort(reverse_sorted_list)
print(f"Merge Sort on reverse-sorted list took {time.time() - start_time:.4f} seconds")
```

Output:

```
Quick Sort on random list took 0.0017 seconds
Quick Sort on sorted list took 0.0011 seconds
Quick Sort on reverse-sorted list took 0.0010 seconds
Merge Sort on random list took 0.0019 seconds
Merge Sort on sorted list took 0.0017 seconds
Merge Sort on reverse-sorted list took 0.0020 seconds
```

Task 4 (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:

#For a retail inventory management system, recommend efficient search and sorting algorithms for each operation, present them in a table with justifications based on dataset size and performance needs, and generate working Python functions to implement the chosen algorithms with comments.

```
#For a retail inventory management system, recommend efficient search and sorting algorithms for each operation, pres
| Operation               | Recommended Algorithm | Justification
|-------------------------|---------------------|------------------------------------------------------------------
| Search for a product    | Binary Search        | Efficient for large sorted datasets, O(log n) time complexity.
| Sort products by price  | Merge Sort           | Stable sorting algorithm with O(n log n) time complexity, suitab
def binary_search(arr, target):
    """Performs binary search on a sorted list.
    Parameters:
    arr (list): A sorted list of products.
    target: The product to search for.
    Returns:
    int: The index of the target product if found, otherwise -1.
    """
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2  # Calculate the middle index
        if arr[mid] == target:
            return mid  # Target found at index mid
        elif arr[mid] < target:
            left = mid + 1  # Search in the right half
        else:
            right = mid - 1  # Search in the left half
    return -1  # Target not found
def merge_sort(arr):
    """Sorts a list of products by price using merge sort.
    Parameters:
    arr (list): A list of products, where each product is a dictionary with a 'price' key.
    Returns:
    list: A sorted list of products by price in ascending order.
    """
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2  # Finding the mid of the array
```

```python
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2  # Finding the mid of the array
    left = merge_sort(arr[:mid])  # Recursively sort the left half
    right = merge_sort(arr[mid:])  # Recursively sort the right half
    return merge(left, right)  # Merge the sorted halves
def merge(left, right):
    """Merges two sorted lists of products into a single sorted list by price.
    Parameters:
    left (list): The first sorted list of products.
    right (list): The second sorted list of products.
    Returns:
    list: A merged and sorted list of products by price in ascending order.
    """
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i]['price'] < right[j]['price']:
            result.append(left[i])  # Append the product with the lower price
            i += 1
        else:
            result.append(right[j])  # Append the product with the lower price
            j += 1
    result.extend(left[i:])  # Append any remaining products from left
    result.extend(right[j:])  # Append any remaining products from right
    return result
```

```python
# Example usage
products = [
    {'name': 'Product A', 'price': 10.99},
    {'name': 'Product B', 'price': 5.49},
    {'name': 'Product C', 'price': 7.99},
    {'name': 'Product D', 'price': 3.99}
]
sorted_products = merge_sort(products)
print("Products sorted by price:")
for product in sorted_products:
    print(f"{product['name']}: ${product['price']}")
# Searching for a product
target_product = 'Product B'
index = binary_search(sorted_products, {'name': target_product, 'price': 5.49})
if index != -1:
    print(f"{target_product} found at index {index}.")
else:
    print(f"{target_product} not found in the product list.")
```

Output:

```
der/AI ASSISTANT/lab 12.3.py
Products sorted by price:
Product D: $3.99
Product B: $5.49
Product C: $7.99
Product A: $10.99
Product B found at index 1.
```

Task 5: Real-Time Stock Data Sorting & Searching

Scenario:

An AI-powered **FinTech Lab** at SR University is building a tool for analyzing **stock price movements**. The requirement is to quickly **sort stocks by daily gain/loss** and search for specific stock symbols efficiently.

- Use **GitHub Copilot** to fetch or simulate stock price data (Stock Symbol, Opening Price, Closing Price).
- Implement sorting algorithms to rank stocks by **percentage change**.
- Implement a **search function** that retrieves stock data instantly when a stock symbol is entered.
- Optimize sorting with **Heap Sort** and searching with **Hash Maps**.
- Compare performance with standard library functions (sorted(), dict lookups) and analyze trade-offs.

PROMPT:

#generate a Python program that simulates stock data (symbol, opening and closing price), sorts stocks by percentage change using Heap Sort, searches stocks efficiently using a hash map, compares performance with built-in sorted() and dictionary lookups, and includes clear comments and analysis of trade-offs."

```
#Generate a Python program that simulates stock data (symbol, opening and closing price), sorts stocks by percentage
import random
import time
class Stock:
    def __init__(self, symbol, opening_price, closing_price):
        self.symbol = symbol
        self.opening_price = opening_price
        self.closing_price = closing_price

    def percentage_change(self):
        return ((self.closing_price - self.opening_price) / self.opening_price) * 100
def heapify(arr, n, i):
    largest = i  # Initialize largest as root
    left = 2 * i + 1  # left = 2*i + 1
    right = 2 * i + 2  # right = 2*i + 2
    # If left child is larger than root
    if left < n and arr[left].percentage_change() > arr[largest].percentage_change():
        largest = left
    # If right child is larger than largest so far
    if right < n and arr[right].percentage_change() > arr[largest].percentage_change():
        largest = right
    # If largest is not root
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # Swap
        heapify(arr, n, largest)  # Recursively heapify the affected sub-tree
def heap_sort(arr):
    n = len(arr)
    # Build a maxheap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    # One by one extract elements from heap
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # Swap
        heapify(arr, i, 0)  # Heapify the root element
```

```python
def generate_stocks(num_stocks):
    stocks = []
    for i in range(num_stocks):
        symbol = f"STK{i+1}"
        opening_price = round(random.uniform(100, 500), 2)
        closing_price = round(random.uniform(100, 500), 2)
        stocks.append(Stock(symbol, opening_price, closing_price))
    return stocks
def create_hash_map(stocks):
    hash_map = {}
    for stock in stocks:
        hash_map[stock.symbol] = stock
    return hash_map
# Generate stock data
num_stocks = 1000
stocks = generate_stocks(num_stocks)
# Sort stocks by percentage change using Heap Sort and measure runtime
start_time = time.time()
heap_sort(stocks)
end_time = time.time()
print(f"Heap Sort Runtime: {end_time - start_time:.4f} seconds")
# Sort stocks using built-in sorted() and measure runtime
start_time = time.time()
sorted_stocks = sorted(stocks, key=lambda x: x.percentage_change(), reverse=True)
end_time = time.time()
print(f"Built-in sorted() Runtime: {end_time - start_time:.4f} seconds")
# Create a hash map for efficient stock lookup and measure runtime
hash_map = create_hash_map(stocks)
target_symbol = "STK500"
start_time = time.time()
stock = hash_map.get(target_symbol, None)
end_time = time.time()
if stock:
    print(f"{target_symbol} found: Opening Price: {stock.opening_price}, Closing Price: {stock.closing_price}")
```

Output:

```
Heap Sort Runtime: 0.0060 seconds
Built-in sorted() Runtime: 0.0005 seconds
STK500 found: Opening Price: 393.05, Closing Price: 478.04
Hash Map Lookup Runtime: 0.000004 seconds
```

Note: Report should be submitted as a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots.