AI ASSISTANT CODING - ASSIGNMENT-12.3

Name: Valaboju Harshavardhanu

Roll No: 2303A52354 **Batch:** 41

Task 1: Sorting Student Records for Placement Drive

Prompt:

generate a programm that stores the student records (name, roll number, CGPA),and implement the quick sort and merge sort algorithms to sort the records based on CGPA in decreasing order.Measure and compare runtime performance for large datasets and write a function to display the top 10 students based on CGPA.

Code:

```
#generate a programm that stores the student records (name, roll number, CGPA),and implement tl
#remove docstrings
import time
class StudentRecord:
    def __init__(self, name: str, roll_number: int, cgpa: float):
        self.name = name
        self.roll_number = roll_number
        self.cgpa = cgpa
def quick_sort(records):
    if len(records) <= 1:
        return records
    else:
        pivot = records[0]
        less_than_pivot = [record for record in records[1:] if record.cgpa > pivot.cgpa]
        greater_than_pivot = [record for record in records[1:] if record.cgpa <= pivot.cgpa]
        return quick_sort(less_than_pivot) + [pivot] + quick_sort(greater_than_pivot)
def merge_sort(records):
    if len(records) <= 1:
        return records
    mid = len(records) // 2
    left_half = merge_sort(records[:mid])
    right_half = merge_sort(records[mid:])
    return merge(left_half, right_half)
def merge(left, right):
    merged = []
    while left and right:
        if left[0].cgpa > right[0].cgpa:
            merged.append(left.pop(0))
        else:
            merged.append(right.pop(0))
```

```
    return merged
def display_top_students(records, n=10):
    sorted_records = quick_sort(records)
    print(f"Top {n} students based on CGPA:")
    for i in range(min(n, len(sorted_records))):
        student = sorted_records[i]
        print(f"{i + 1}. Name: {student.name}, Roll Number: {student.roll_number}, CGPA: {student.cgpa}")
if __name__ == "__main__":
    students = [
        StudentRecord("Alice", 1, 3.8),
        StudentRecord("Bob", 2, 3.6),
        StudentRecord("Charlie", 3, 3.9),
        StudentRecord("David", 4, 3.7),
        StudentRecord("Eve", 5, 3.5),
        StudentRecord("Frank", 6, 3.4),
        StudentRecord("Grace", 7, 3.9),
        StudentRecord("Heidi", 8, 3.6),
        StudentRecord("Ivan", 9, 3.8),
        StudentRecord("Judy", 10, 3.7)
    ]
    start_time = time.time()
    sorted_students_quick = quick_sort(students)
    end_time = time.time()
    print(f"Quick Sort Runtime: {end_time - start_time:.6f} seconds")
    start_time = time.time()
    sorted_students_merge = merge_sort(students)
    end_time = time.time()
    print(f"Merge Sort Runtime: {end_time - start_time:.6f} seconds")
    display_top_students(students)
```

Output:

```
PS C:\Users\harsh\OneDrive\Desktop\AI Assist lab> & C:\U
/AI Assist lab/12.py"
Quick Sort Runtime: 0.000019 seconds
Merge Sort Runtime: 0.000023 seconds
Top 10 students based on CGPA:
1. Name: Charlie, Roll Number: 3, CGPA: 3.9
2. Name: Grace, Roll Number: 7, CGPA: 3.9
3. Name: Alice, Roll Number: 1, CGPA: 3.8
4. Name: Ivan, Roll Number: 9, CGPA: 3.8
5. Name: David, Roll Number: 4, CGPA: 3.7
6. Name: Judy, Roll Number: 10, CGPA: 3.7
7. Name: Bob, Roll Number: 2, CGPA: 3.6
8. Name: Heidi, Roll Number: 8, CGPA: 3.6
9. Name: Eve, Roll Number: 5, CGPA: 3.5
10. Name: Frank, Roll Number: 6, CGPA: 3.4
PS C:\Users\harsh\OneDrive\Desktop\AI Assist lab>
```

Justification:

This task helps in efficiently organizing student records based on CGPA for placement shortlisting. Quick Sort and Merge Sort are efficient algorithms suitable for large datasets. Comparing runtime helps understand which algorithm performs better. Displaying the top 10 students makes the system practical for placement use. This task improves understanding of sorting algorithms in real-world applications.

Task 2: Implementing Bubble Sort with AI Comments

Prompt:

generate a python code for bubble sort, and generate the inline comments explaining key logic (like swapping, passes and termination) and provide the time complexity analysis.

Code:

```
#generate a python code for bubble sort, and generate the inline comments explaining key lo
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
                swapped = True  # Set the flag to True if swapping occurs
        # If no swapping occurred, the array is already sorted, so we can terminate early
        if not swapped:
            break
# Time complexity analysis:
# The worst-case and average-case time complexity of bubble sort is O(n^2) because
# it requires two nested loops to compare each element with every other element.
# The best-case time complexity is O(n) when the array is already sorted, as it
# only requires one pass to check that the array is sorted without any swaps.
# Example usage
if __name__ == "__main__":
    arr = [64, 34, 25, 12, 22, 11, 90]
    print("Original array:", arr)
    bubble_sort(arr)
    print("Sorted array:", arr)
```

Output:

```
PS C:\Users\harsh\OneDrive\Desktop\AI Assist lab> & C:\
/AI Assist lab/12.py"
Original array: [64, 34, 25, 12, 22, 11, 90]
Sorted array: [11, 12, 22, 25, 34, 64, 90]
PS C:\Users\harsh\OneDrive\Desktop\AI Assist lab>
```

Justification:

This task helps to understand the basic working of Bubble Sort. AI-generated comments explain swapping, passes, and stopping conditions clearly. It improves code readability and learning. Time complexity analysis helps understand algorithm efficiency. This task strengthens basic sorting concepts.

Task 3: Quick Sort and Merge Sort Comparison

Prompt:

generate a partially completed functions for recursion of quick sort and merge sort and complete the missing logics and add docstrings, compare both algorithms on random, sorted, and reverse-sorted lists.

Code:

```
import random
def quick_sort(arr):
    """Sorts an array using the quick sort algorithm."""
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]  # Choosing the first element as the pivot
        less_than_pivot = [x for x in arr[1:] if x < pivot]  # Elements less than the pivot
        greater_than_pivot = [x for x in arr[1:] if x >= pivot]  # Elements greater than or equal to the pivot
        return quick_sort(less_than_pivot) + [pivot] + quick_sort(greater_than_pivot)  # Recursively sort and combine
def merge_sort(arr):
    """Sorts an array using the merge sort algorithm."""
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2  # Finding the mid of the array
    left_half = merge_sort(arr[:mid])  # Recursively sorting the left half
    right_half = merge_sort(arr[mid:])  # Recursively sorting the right half
    return merge(left_half, right_half)  # Merging the sorted halves
def merge(left, right):
    """Merges two sorted arrays into a single sorted array."""
    merged = []
    while left and right:
        if left[0] < right[0]:  # Compare the first elements of both halves
            merged.append(left.pop(0))  # Append the smaller element and remove it from the list
        else:
            merged.append(right.pop(0))  # Append the smaller element and remove it from the list
    merged.extend(left)  # If there are remaining elements in left, add them to merged
    merged.extend(right)  # If there are remaining elements in right, add them to merged
    return merged
```

```python
if __name__ == "__main__":
    # Generate random, sorted, and reverse-sorted lists
    random_list = [random.randint(1, 100) for _ in range(10)]
    sorted_list = sorted(random_list)
    reverse_sorted_list = sorted(random_list, reverse=True)

    # Compare quick sort on different lists
    print("Random List:", random_list)
    print("Quick Sort on Random List:", quick_sort(random_list))

    print("Sorted List:", sorted_list)
    print("Quick Sort on Sorted List:", quick_sort(sorted_list))

    print("Reverse Sorted List:", reverse_sorted_list)
    print("Quick Sort on Reverse Sorted List:", quick_sort(reverse_sorted_list))

    # Compare merge sort on different lists
    print("Merge Sort on Random List:", merge_sort(random_list))
    print("Merge Sort on Sorted List:", merge_sort(sorted_list))
    print("Merge Sort on Reverse Sorted List:", merge_sort(reverse_sorted_list))
```

Output:

```
PS C:\Users\harsh\OneDrive\Desktop\AI Assist lab> & C:\Users\harsh\AppData\
/AI Assist lab/12.py"
Random List: [26, 5, 93, 77, 47, 23, 3, 98, 94, 47]
Quick Sort on Random List: [3, 5, 23, 26, 47, 47, 77, 93, 94, 98]
Sorted List: [3, 5, 23, 26, 47, 47, 77, 93, 94, 98]
Quick Sort on Sorted List: [3, 5, 23, 26, 47, 47, 77, 93, 94, 98]
Reverse Sorted List: [98, 94, 93, 77, 47, 47, 26, 23, 5, 3]
Quick Sort on Reverse Sorted List: [3, 5, 23, 26, 47, 47, 77, 93, 94, 98]
Merge Sort on Random List: [3, 5, 23, 26, 47, 47, 77, 93, 94, 98]
Merge Sort on Sorted List: [3, 5, 23, 26, 47, 47, 77, 93, 94, 98]
Merge Sort on Reverse Sorted List: [3, 5, 23, 26, 47, 47, 77, 93, 94, 98]
PS C:\Users\harsh\OneDrive\Desktop\AI Assist lab>
```

Justification:

This task helps compare two important recursive sorting algorithms. Testing on random, sorted, and reverse lists shows real performance differences. AI helps complete recursion logic and documentation. Complexity analysis improves theoretical understanding. This task builds knowledge of advanced sorting techniques.

Task 4: Real-Time Application – Inventory Management System

Prompt:

generate a python code for a Inventory management system with attributes like product id, name, price, stock quantity and for quick search of product by id and name and sort products by price or quantity suggest and implement the most efficient search and sorting algorithms for this use case. Justify the choice based on the dataset size, update frequency, and performance requirements.

Code:

```python
#generate a python code for a Inventory management system with attributes like product id, name, pri
class Product:
    def __init__(self, product_id: int, name: str, price: float, stock_quantity: int):
        self.product_id = product_id
        self.name = name
        self.price = price
        self.stock_quantity = stock_quantity
class InventoryManagementSystem:
    def __init__(self):
        self.products = []
        self.product_id_index = {}
        self.product_name_index = {}
    def add_product(self, product: Product):
        self.products.append(product)
        self.product_id_index[product.product_id] = product
        self.product_name_index[product.name] = product
    def search_by_id(self, product_id: int):
        return self.product_id_index.get(product_id, None)  # O(1) time complexity
    def search_by_name(self, name: str):
        return self.product_name_index.get(name, None)  # O(1) time complexity
    def sort_by_price(self):
        return sorted(self.products, key=lambda x: x.price)  # O(n log n) time complexity
    def sort_by_stock_quantity(self):
        return sorted(self.products, key=lambda x: x.stock_quantity)  # O(n log n) time complexity
```

```python
# Justification:
# For searching products by id and name, we use a hash map (dictionary) which provides O(1) average time complexity for lookups, making it
# For sorting products by price or stock quantity, we use Python's built-in sorted() function
# which implements Timsort (a hybrid sorting algorithm derived from merge sort and insertion sort) with O(n log n) time complexity, making
# Example usage
if __name__ == "__main__":
    inventory = InventoryManagementSystem()
    inventory.add_product(Product(1, "Laptop", 999.99, 10))
    inventory.add_product(Product(2, "Smartphone", 499.99, 20))
    inventory.add_product(Product(3, "Headphones", 199.99, 15))

    print("Search by ID (1):", inventory.search_by_id(1).__dict__)
    print("Search by Name (Smartphone):", inventory.search_by_name("Smartphone").__dict__)

    print("Products sorted by price:")
    for product in inventory.sort_by_price():
        print(product.__dict__)

    print("Products sorted by stock quantity:")
    for product in inventory.sort_by_stock_quantity():
        print(product.__dict__)
```

Output:

```
PS C:\Users\harsh\OneDrive\Desktop\AI Assist lab> & C:\Users\harsh\AppData\Local\Programs\Python\Python314\p
/AI Assist lab/12.py"
Search by ID (1): {'product_id': 1, 'name': 'Laptop', 'price': 999.99, 'stock_quantity': 10}
Search by Name (Smartphone): {'product_id': 2, 'name': 'Smartphone', 'price': 499.99, 'stock_quantity': 20}
Products sorted by price:
{'product_id': 3, 'name': 'Headphones', 'price': 199.99, 'stock_quantity': 15}
{'product_id': 2, 'name': 'Smartphone', 'price': 499.99, 'stock_quantity': 20}
{'product_id': 1, 'name': 'Laptop', 'price': 999.99, 'stock_quantity': 10}
Products sorted by stock quantity:
{'product_id': 1, 'name': 'Laptop', 'price': 999.99, 'stock_quantity': 10}
{'product_id': 3, 'name': 'Headphones', 'price': 199.99, 'stock_quantity': 15}
{'product_id': 2, 'name': 'Smartphone', 'price': 499.99, 'stock_quantity': 20}
PS C:\Users\harsh\OneDrive\Desktop\AI Assist lab>
```

Justification:

This task simulates a real retail inventory system. Efficient searching and sorting improve product management speed. AI helps select suitable algorithms based on dataset size. The implementation improves practical coding skills. This task shows real-time application of algorithms.

Task 5: Real -Time Stock Data Sorting & Searching

Prompt:

generate a python code for stock price movements to quickly sort stocks by daily gain or loss, simulate stock price data(stock symbol, opening price and closing price), implement the sorting algorithms to rank stocks by percentage change, implement a search function that retrieves stock data and optimize the sorting with heap sort and searching with hashmaps and compare performance with standard library functions(sorted(), dict lookups) and analyze trade-offs.

Code:

```python
#generate a python code for stock price movements to quickly sort stocks by daily gain or loss, simulate stock price data(stock s
import heapq
import random
class Stock:
    def __init__(self, symbol: str, opening_price: float, closing_price: float):
        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
class StockMarket:
    def __init__(self):
        self.stocks = []
        self.stock_index = {}
    def add_stock(self, stock: Stock):
        self.stocks.append(stock)
        self.stock_index[stock.symbol] = stock
    def search_stock(self, symbol: str):
        return self.stock_index.get(symbol, None)  # O(1) time complexity
    def sort_by_percentage_change(self):
        return sorted(self.stocks, key=lambda x: x.percentage_change(), reverse=True)  # O(n log n) time complexity
    def heap_sort_by_percentage_change(self):
        return heapq.nlargest(len(self.stocks), self.stocks, key=lambda x: x.percentage_change())  # O(n log k) time complexity

if __name__ == "__main__":
    stock_market = StockMarket()
    symbols = ["AAPL", "GOOGL", "MSFT", "AMZN", "TSLA"]
    for symbol in symbols:
        opening_price = random.uniform(100, 500)
        closing_price = opening_price + random.uniform(-20, 20)
        stock_market.add_stock(Stock(symbol, opening_price, closing_price))

    print("Stocks sorted by percentage change (using sorted()):")
    for stock in stock_market.sort_by_percentage_change():
        print(f"{stock.symbol}: {stock.percentage_change():.2f}%")

    print("\nStocks sorted by percentage change (using heap sort):")
    for stock in stock_market.heap_sort_by_percentage_change():
        print(f"{stock.symbol}: {stock.percentage_change():.2f}%")

    # Performance comparison
    import time
    start_time = time.time()
    stock_market.sort_by_percentage_change()
    end_time = time.time()
    print(f"\nStandard library sorted() runtime: {end_time - start_time:.6f} seconds")

    start_time = time.time()
    stock_market.heap_sort_by_percentage_change()
    end_time = time.time()
    print(f"Heap sort runtime: {end_time - start_time:.6f} seconds")

# Justification:
# The search function uses a hashmap (dictionary) for O(1) average time complexity,
# making it efficient for retrieving stock data by symbol.
# The sorting function uses Python's built-in sorted() which is efficient for large datasets with O
# (n log n) time complexity. The heap sort implementation using heapq.nlargest is efficient for finding the top k elements, with O(n log k)
```

Output:

```
/AI Assist lab/12.py"
Stocks sorted by percentage change (using sorted()):
AMZN: -0.59%
TSLA: -2.22%
AAPL: -2.25%
MSFT: -4.53%
GOOGL: -4.98%

Stocks sorted by percentage change (using heap sort):
AMZN: -0.59%
TSLA: -2.22%
AAPL: -2.25%
MSFT: -4.53%
GOOGL: -4.98%

Standard library sorted() runtime: 0.000004 seconds
Heap sort runtime: 0.000005 seconds
PS C:\Users\harsh\OneDrive\Desktop\AI Assist lab>
```

Justification:

This task simulates a real FinTech stock analysis system. Heap Sort helps rank stocks efficiently by percentage change. Hash Maps allow fast searching using stock symbols. Performance comparison shows advantages of optimized methods. This task demonstrates real-world AI-assisted data processing.