ASSIGNMENT 12.3

SHIVANI PAMBIDI

2303A52172

BATCH-41

QUESTION 1

TASK 1: Sorting Student Records for Placement Drive

PROMPT: Write a Python program to compare Quick Sort and Merge Sort for sorting student placement records.Each student should contain name, roll number, and CGPA.Sort the records by CGPA in descending order, then by name and roll number as tie-breakers.Generate random student data for testing.Measure and compare the execution time of both sorting algorithms.Display the sorted results and print the top 10 students

CODE:

Lab 12.py > ...

```python
#!/usr/bin/env python3
"""
placements_sort.py
Compare Quick Sort and Merge Sort implementations for sorting student placement data.
Sorts by CGPA (descending), with name and roll number as tie-breakers.
Provides runtime comparisons, speedup analysis, and displays the top 10 students.
Usage:
    python placements_sort.py --size 1000 --runs 5
"""
from __future__ import annotations
import argparse
import random
import time
from dataclasses import dataclass
from typing import List, Callable
@dataclass
class Student:
    """Dataclass representing a student."""
    name: str
    roll: int
    cgpa: float
def generate_students(size: int) -> List[Student]:
    """Generate a random dataset of students with familiar names."""
    sample_names = [
        "Raju", "Rahul", "Rani", "Priya", "Sita", "Gita", "Mohan",
        "Arjun", "Anjali", "Sunita", "Vijay", "Kiran", "Asha",
        "Deepak", "Neha", "Pooja", "Rajesh", "Meena", "Suresh", "Lakshmi"
    ]
    students = []
    for i in range(size):
        name = random.choice(sample_names)
        roll = i + 1
```

```python
def generate_students(size: int) -> List[Student]:
    students = []
    for i in range(size):
        name = random.choice(sample_names)
        roll = i + 1
        cgpa = round(random.uniform(5.0, 10.0), 2)
        students.append(Student(name, roll, cgpa))
    return students
def compare_students(a: Student, b: Student) -> int:
    """Comparison function for students: CGPA desc, then name asc, then roll asc."""
    if a.cgpa != b.cgpa:
        return -1 if a.cgpa > b.cgpa else 1
    if a.name != b.name:
        return -1 if a.name < b.name else 1
    return -1 if a.roll < b.roll else 1 if a.roll > b.roll else 0
def quick_sort(arr: List[Student]) -> List[Student]:
    """Quick Sort implementation."""
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if compare_students(x, pivot) < 0]
    middle = [x for x in arr if compare_students(x, pivot) == 0]
    right = [x for x in arr if compare_students(x, pivot) > 0]
    return quick_sort(left) + middle + quick_sort(right)
def merge_sort(arr: List[Student]) -> List[Student]:
    """Merge Sort implementation."""
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)
```

```python
def measure_runtime(sort_fn: Callable[[List[Student]], List[Student]], dataset: List[Student], runs:
        assert sorted_data == sorted(dataset, key=lambda s: (-s.cgpa, s.name, s.roll))
    return total_time / runs
def main() -> None:
    parser = argparse.ArgumentParser(description="Compare Quick Sort and Merge Sort for student place
    parser.add_argument("--size", type=int, default=1000, help="Number of students in dataset")
    parser.add_argument("--runs", type=int, default=5, help="Number of runs for averaging runtime")
    args = parser.parse_args()
    dataset = generate_students(args.size)
    quick_time = measure_runtime(quick_sort, dataset, args.runs)
    merge_time = measure_runtime(merge_sort, dataset, args.runs)
    speedup = merge_time / quick_time if quick_time > 0 else float("inf")
    print("\n==============================")
    print(" Correctly Sorted Student Records ")
    print("==============================")
    sorted_students = quick_sort(dataset)
    for student in sorted_students[:20]:  # show first 20 for verification
        print(f"{student.name:<8} Roll:{student.roll:<4} CGPA:{student.cgpa:.2f}")
    print("\n==============================")
    print(" Performance Comparison ")
    print("==============================")
    print(f"Dataset size: {args.size}, Runs: {args.runs}")
    print(f"Quick Sort: {quick_time:.6f} seconds (avg)")
    print(f"Merge Sort: {merge_time:.6f} seconds (avg)")
    print(f"Speedup (Merge/Quick): {speedup:.2f}x")
    print("\n==============================")
    print(" Top 10 Students ")
    print("==============================")
    for i, student in enumerate(sorted_students[:10], start=1):
        print(f"{i:2d}. Name: {student.name}, Roll: {student.roll}, CGPA: {student.cgpa:.2f}")
if __name__ == "__main__":
    main()
```

OUTPUT:

```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/Local/Programs/Python/Py
thon313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding/Lab 12.py"

==============================
 Correctly Sorted Student Records
==============================
Anjali   Roll:97   CGPA:10.00
Gita     Roll:461  CGPA:10.00
Priya    Roll:832  CGPA:10.00
Priya    Roll:641  CGPA:9.99
Asha     Roll:335  CGPA:9.98
Deepak   Roll:962  CGPA:9.98
Pooja    Roll:104  CGPA:9.98
Sita     Roll:63   CGPA:9.98
Kiran    Roll:46   CGPA:9.97
Kiran    Roll:432  CGPA:9.97
Sunita   Roll:232  CGPA:9.96
Suresh   Roll:533  CGPA:9.96
Anjali   Roll:177  CGPA:9.95
Rahul    Roll:646  CGPA:9.95
Kiran    Roll:251  CGPA:9.94
Mohan    Roll:266  CGPA:9.94
Pooja    Roll:566  CGPA:9.94
Pooja    Roll:951  CGPA:9.94
Anjali   Roll:130  CGPA:9.93
Priya    Roll:887  CGPA:9.93

==============================
 Performance Comparison
==============================
Dataset size: 1000, Runs: 5
Quick Sort: 0.004719 seconds (avg)
Merge Sort: 0.001961 seconds (avg)
```

```
==============================
p 10 Students
==============================
 Name: Anjali, Roll: 97, CGPA: 10.00
 Name: Gita, Roll: 461, CGPA: 10.00
 Name: Priya, Roll: 832, CGPA: 10.00
 Name: Priya, Roll: 641, CGPA: 9.99
 Name: Asha, Roll: 335, CGPA: 9.98
 Name: Deepak, Roll: 962, CGPA: 9.98
 Name: Pooja, Roll: 104, CGPA: 9.98
 Name: Sita, Roll: 63, CGPA: 9.98
 Name: Kiran, Roll: 46, CGPA: 9.97
 Name: Anjali, Roll: 97, CGPA: 10.00
 Name: Gita, Roll: 461, CGPA: 10.00
 Name: Priya, Roll: 832, CGPA: 10.00
 Name: Priya, Roll: 641, CGPA: 9.99
 Name: Asha, Roll: 335, CGPA: 9.98
 Name: Deepak, Roll: 962, CGPA: 9.98
 Name: Pooja, Roll: 104, CGPA: 9.98
 Name: Sita, Roll: 63, CGPA: 9.98
 Name: Kiran, Roll: 46, CGPA: 9.97
 Name: Priya, Roll: 832, CGPA: 10.00
 Name: Priya, Roll: 641, CGPA: 9.99
 Name: Asha, Roll: 335, CGPA: 9.98
 Name: Deepak, Roll: 962, CGPA: 9.98
 Name: Pooja, Roll: 104, CGPA: 9.98
 Name: Sita, Roll: 63, CGPA: 9.98
 Name: Kiran, Roll: 46, CGPA: 9.97
 Name: Priya, Roll: 641, CGPA: 9.99
```

JUSTIFICATION:

This program helps us understand how different sorting algorithms work in real situations like student placements.Quick Sort and Merge Sort are both efficient sorting techniques used in computer science.By comparing their execution time, we can know which algorithm performs faster for a given dataset.Sorting by CGPA, name, and roll number ensures proper ranking without mistakes. Displaying the top 10 students makes the results easy to verify.Overall, this program improves understanding of sorting efficiency and performance comparison.

QUESTION 2

TASK 2: Implementing Bubble Sort with AI Comments

PROMPT:

Write a Python program to implement Bubble Sort to sort a list of integers in ascending order.The program should compare adjacent elements and swap them if they are in the wrong order.Repeat the process until the list is completely sorted.Print the original array and the sorted array.Also mention the time and space complexity of the algorithm.

CODE:

```
Lab 12.py > ...
from typing import List
def bubble_sort(arr: List[int]) -> List[int]:
    n = len(arr)
    # Outer loop: controls the number of passes
    for i in range(n):
        # Inner loop: compares adjacent elements
        for j in range(0, n - i - 1):
            # If the current element is greater than the next, swap them
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
        # After each pass, the largest element among the unsorted part
        # "bubbles up" to its correct position at the end
    # When no swaps are needed, the list is sorted and the algorithm terminates
    return arr
if __name__ == "__main__":
    data = [64, 34, 25, 12, 22, 11, 90]
    print("Original array:", data)
    sorted_data = bubble_sort(data)
    print("Sorted array:", sorted_data)
    # Complexity Analysis:
    # Bubble Sort has worst-case and average-case time complexity O(n^2),
    # because it requires nested loops over the list.
    # Best case is O(n) when the list is already sorted (with an optimized version).
    # Space complexity is O(1), since sorting is done in-place.
```

OUTPUT:

```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding/Lab 12.py"
Original array: [64, 34, 25, 12, 22, 11, 90]
Sorted array: [11, 12, 22, 25, 34, 64, 90]
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding>
```

JUSTIFICATION:

This program helps to understand how the Bubble Sort algorithm works step by step.It shows how adjacent elements are compared and swapped to arrange numbers in order.Bubble Sort is easy to learn and useful for basic understanding of sorting.By printing the original and sorted array, we can clearly see the result.The program also explains time and space complexity for better understanding of performance.

QUESTION 3

TASK 3: Quick Sort and Merge Sort Comparison

PROMPT:

Write a Python program to implement Quick Sort and Merge Sort algorithms using recursion.Generate different types of datasets such as random, sorted, and reverse-sorted lists.Measure the execution time of both sorting algorithms using multiple runs.Calculate the average runtime for better accuracy.Compare their performance for each test case.Display the execution time results clearly for analysis.

CODE:

Lab 12.py > ...

```python
import random
import time
from typing import List
def quick_sort(arr: List[int]) -> List[int]:
    """
    Recursive Quick Sort implementation.
    Splits the list around a pivot and recursively sorts sublists.
    """
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
def merge_sort(arr: List[int]) -> List[int]:
    """
    Recursive Merge Sort implementation.
    Divides the list into halves, recursively sorts them,
    and merges the sorted halves.
    """
    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: List[int], right: List[int]) -> List[int]:
    """Helper function to merge two sorted lists."""
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS

```python
def merge(left: List[int], right: List[int]) -> List[int]:
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
def measure_runtime(sort_fn, arr: List[int], runs: int = 3) -> float:
    """Measure average runtime of a sorting function."""
    total = 0.0
    for _ in range(runs):
        data_copy = arr[:]
        start = time.perf_counter()
        sort_fn(data_copy)
        end = time.perf_counter()
        total += (end - start)
    return total / runs
def main():
    sizes = [1000]
    test_cases = {
        "Random": lambda n: [random.randint(0, 10000) for _ in range(n)],
        "Sorted": lambda n: list(range(n)),
        "Reverse-Sorted": lambda n: list(range(n, 0, -1)),
    }
    for size in sizes:
        print(f"\n=== Dataset Size: {size} ===")
        for case_name, generator in test_cases.items():
            dataset = generator(size)
            quick_time = measure_runtime(quick_sort, dataset)
            merge_time = measure_runtime(merge_sort, dataset)
            print(f"\nCase: {case_name}")
            print(f"Quick Sort: {quick_time:.6f} seconds (avg)")
            print(f"Merge Sort: {merge_time:.6f} seconds (avg)")
if __name__ == "__main__":
    main()
```

OUTPUT:

```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding/Lab 12.py"

=== Dataset Size: 1000 ===

Case: Random
Quick Sort: 0.001297 seconds (avg)
Merge Sort: 0.001401 seconds (avg)

Case: Sorted
Quick Sort: 0.000880 seconds (avg)
Merge Sort: 0.001285 seconds (avg)

Case: Reverse-Sorted
Quick Sort: 0.000831 seconds (avg)
Merge Sort: 0.001578 seconds (avg)
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding>
```

JUSTIFICATION:

This program helps to compare the performance of Quick Sort and Merge Sort in different situations.It checks how the algorithms behave with random, sorted, and reverse-sorted data.By measuring the average runtime, we get more accurate performance results.It helps in understanding time complexity and practical efficiency.The comparison shows which algorithm works better under different conditions.

Overall, it improves knowledge of sorting techniques and performance analysis.

QUESTION 4

TASK 4: (Real-Time Application – Inventory Management System)

PROMPT:

Write a Python program to manage product inventory using a Product class. Implement Binary Search to search products by ID (on sorted data). Implement Linear Search to search products by name.Use Merge Sort to sort products by price, quantity, or product ID.Display sample search results and sorted outputs.Print a small table showing recommended algorithms with justification.

CODE:

```python
from dataclasses import dataclass
from typing import List, Optional
@dataclass
class Product:
    product_id: int
    name: str
    price: float
    quantity: int
# --- Search Algorithms ---
def binary_search(products: List[Product], target_id: int) -> Optional[Product]:
    """Binary search for product by ID (requires sorted by ID)."""
    low, high = 0, len(products) - 1
    while low <= high:
        mid = (low + high) // 2
        if products[mid].product_id == target_id:
            return products[mid]
        elif products[mid].product_id < target_id:
            low = mid + 1
        else:
            high = mid - 1
    return None
def linear_search(products: List[Product], target_name: str) -> Optional[Product]:
    """Linear search for product by name (works on unsorted data)."""
    for product in products:
        if product.name.lower() == target_name.lower():
            return product
    return None
# --- Sort Algorithms ---
def merge_sort(products: List[Product], key: str) -> List[Product]:
    """Merge Sort implementation to sort products by a given key."""
    if len(products) <= 1:
        return products
```

```python
def print_algorithm_table():
    print("\n=== Recommended Algorithms ===")
    print(f"{'Operation':<25}{'Algorithm':<20}{'Justification'}")
    print("-" * 70)
    print(f"{'Search by ID':<25}{'Binary Search':<20}"
          f"{'Efficient O(log n) lookup on sorted IDs'}")
    print(f"{'Search by Name':<25}{'Linear Search':<20}"
          f"{'Names unsorted; O(n) acceptable for occasional lookups'}")
    print(f"{'Sort by Price/Quantity':<25}{'Merge Sort':<20}"
          f"{'Stable O(n log n) sort, good for large datasets'}")
def main():
    # Sample inventory
    inventory = [
        Product(101, "Apple", 50.0, 120),
        Product(205, "Banana", 20.0, 200),
        Product(150, "Orange", 30.0, 80),
        Product(300, "Mango", 100.0, 50),
    ]
    # Show recommended algorithms
    print_algorithm_table()
    # Search examples
    print("\n=== Search Examples ===")
    sorted_by_id = merge_sort(inventory, "product_id")
    found = binary_search(sorted_by_id, 150)
    print("Search by ID 150:", found)
    found_name = linear_search(inventory, "Mango")
    print("Search by Name 'Mango':", found_name)
    # Sort examples
    print("\n=== Sort Examples ===")
    sorted_by_price = merge_sort(inventory, "price")
    print("Sorted by Price:", sorted_by_price)
    sorted_by_quantity = merge_sort(inventory, "quantity")
    print("Sorted by Quantity:", sorted_by_quantity)
if __name__ == "__main__":
    main()
```

OUTPUT:

```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/Local/Programs/Python/Py
thon313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding/Lab 12.py"

=== Search Examples ===
Search by ID 150: Product(product_id=150, name='Orange', price=30.0, quantity=80)
Search by Name 'Mango': Product(product_id=300, name='Mango', price=100.0, quantity=50)

=== Sort Examples ===
Sorted by Price: [Product(product_id=205, name='Banana', price=20.0, quantity=200), Product(product_id=150, name=
'Orange', price=30.0, quantity=80), Product(product_id=101, name='Apple', price=50.0, quantity=120), Product(prod
uct_id=300, name='Mango', price=100.0, quantity=50)]
Sorted by Quantity: [Product(product_id=300, name='Mango', price=100.0, quantity=50), Product(product_id=150, nam
e='Orange', price=30.0, quantity=80), Product(product_id=101, name='Apple', price=50.0, quantity=120), Product(pr
oduct_id=205, name='Banana', price=20.0, quantity=200)]
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding>
```

JUSTIFICATION:

This program helps to understand how searching and sorting work in real applications like inventory management.Binary Search is used for fast searching when data is sorted.Linear Search is useful when data is not sorted.Merge Sort is efficient and stable for sorting large datasets.The program compares different algorithms and explains why each one is chosen.It improves understanding of time complexity and practical algorithm selection.

QUESTION 5

TASK 5: Real-Time Stock Data Sorting & Searching

PROMPT:

Write a Python program to simulate real-time stock data with stock symbol, opening price, and closing price.Calculate percentage gain or loss for each stock.Implement Heap Sort to rank stocks based on percentage change.
Use a Hash Map (dictionary) to search stock details instantly using stock symbol.Compare performance with Python's built-in sorted() and dictionary lookup.Display sorting results and search outputs clearly.

CODE:

```
import random
import time
def generate_stocks(n):
    symbols=["TCS","INFY","HDFC","RELIANCE","WIPRO","SBIN","ITC","ADANI","AXIS","ICICI"]
    stocks=[]
    for i in range(n):
        symbol=random.choice(symbols)+str(i)
        open_price=random.randint(100,5000)
        close_price=open_price+random.randint(-200,200)
        percent_change=((close_price-open_price)/open_price)*100
        stocks.append({"symbol":symbol,"open":open_price,"close":close_price,"change":round(percent_c
    return stocks
def heapify(arr,n,i):
    largest=i
    left=2*i+1
    right=2*i+2
    if left<n and arr[left]["change"]>arr[largest]["change"]:
        largest=left
    if right<n and arr[right]["change"]>arr[largest]["change"]:
        largest=right
    if largest!=i:
        arr[i],arr[largest]=arr[largest],arr[i]
        heapify(arr,n,largest)
def heap_sort(arr):
    n=len(arr)
    for i in range(n//2-1,-1,-1):
        heapify(arr,n,i)
    for i in range(n-1,0,-1):
        arr[i],arr[0]=arr[0],arr[i]
        heapify(arr,i,0)
    return arr[::-1]
def build_hash_map(stocks):
    return {stock["symbol"]:stock for stock in stocks}
def search_stock(stock_map,symbol):
    return stock_map.get(symbol,"Stock not found")
# Generate data
```

```python
# Generate data
n=1000
stocks=generate_stocks(n)
stock_map=build_hash_map(stocks)
sorted_heap=heap_sort(stocks.copy())
while True:
    print("\n--- Stock Analysis Menu ---")
    print("1. Show Top 5 Stocks")
    print("2. Search Stock by Symbol")
    print("3. Performance Comparison")
    print("4. Exit")
    choice=input("Enter your choice: ")
    if choice=="1":
        print("\nTop 5 Stocks by % Gain/Loss:\n")
        for i in range(5):
            s=sorted_heap[i]
            print(f"{i+1}. {s['symbol']} | Open:{s['open']} | Close:{s['close']} | Change:{s['change'
    elif choice=="2":
        symbol=input("Enter Stock Symbol: ")
        result=search_stock(stock_map,symbol)
        print("Search Result:",result)
    elif choice=="3":
        start=time.perf_counter()
        heap_sort(stocks.copy())
        heap_time=time.perf_counter()-start
        start=time.perf_counter()
        sorted(stocks,key=lambda x:x["change"],reverse=True)
        builtin_time=time.perf_counter()-start
        print("\nPerformance Comparison:")
        print("Heap Sort Time:",round(heap_time,6),"seconds")
        print("Built-in sorted() Time:",round(builtin_time,6),"seconds")
    elif choice=="4":
        print("Exiting program...")
        break
    else:
        print("Invalid choice! Try again.")
```

OUTPUT:

```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding/Lab 12.py"

--- Stock Analysis Menu ---
1. Show Top 5 Stocks
2. Search Stock by Symbol
3. Performance Comparison
4. Exit
Enter your choice: 1

Top 5 Stocks by % Gain/Loss:

1. SBIN125 | Open:120 | Close:318 | Change:165.0%
2. AXIS658 | Open:110 | Close:281 | Change:155.45%
3. WIPRO609 | Open:120 | Close:254 | Change:111.67%
4. SBIN666 | Open:127 | Close:250 | Change:96.85%
5. INFY206 | Open:213 | Close:386 | Change:81.22%

--- Stock Analysis Menu ---
1. Show Top 5 Stocks
2. Search Stock by Symbol
3. Performance Comparison
4. Exit
Enter your choice: 2
Enter Stock Symbol: 3
Search Result: Stock not found

--- Stock Analysis Menu ---
1. Show Top 5 Stocks
2. Search Stock by Symbol
3. Performance Comparison
4. Exit
Enter your choice: 4
Exiting program...
```

JUSTIFICATION:

This program helps to understand how sorting and searching are used in real-time financial systems.Heap Sort is efficient for ranking large stock datasets. Hash Maps provide very fast searching using stock symbols.Comparing with built-in functions helps analyze performance differences.It shows practical use of data structures in FinTech applications.Overall, it improves knowledge of algorithm optimization and real-world problem solving.