Assignment-12.5

Name:Thumiki Satya Sree

Hallticket:2303A51222

Batch:04

1)

Task Description #1 (Sorting – Merge Sort Implementation)

- Task: Use AI to generate a Python program that implements the Merge Sort algorithm.
- Instructions:
  - Prompt AI to create a function merge_sort(arr) that sorts a list in ascending order.
  - Ask AI to include time complexity and space complexity in the function docstring.
  - Verify the generated code with test cases.
- Expected Output:
  - A functional Python script implementing Merge Sort with proper documentation.

Code:

```python
ass_12_5.py > ...
from typing import List, Any
def merge_sort(arr: List[Any]) -> List[Any]:
    """Sort a list in ascending order using merge sort.
    Parameters
    ----------
    arr : List[Any]
        The list of comparable elements to sort.
    Returns
    -------
    List[Any]
        A new list containing the sorted elements.
    Time Complexity: O(n log n) on all cases.
    Space Complexity: O(n) due to auxiliary lists used during merging.
    """
    # Base case: lists with 0 or 1 element are already sorted
    if len(arr) <= 1:
        return arr[:]
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    # merge two sorted lists
    merged: List[Any] = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
    # append any remaining elements
    merged.extend(left[i:])
    merged.extend(right[j:])
    return merged
if __name__ == "__main__":
    # simple demo when run as a script
    sample = [38, 27, 43, 3, 9, 82, 10]
    print("Original:", sample)
    print("Sorted:", merge_sort(sample))
```

Output:

```
Original: [38, 27, 43, 3, 9, 82, 10]
Sorted: [3, 9, 10, 27, 38, 43, 82]
```

Justification:

The program correctly implements Merge Sort using the divide-and-conquer approach by recursively splitting the list and merging sorted halves. It always runs in **O(n log n)** time because the list is divided into log n levels and merged in linear time. The algorithm uses **O(n)** extra space and is **stable**, since equal elements keep their original order.

2)

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.
- Instructions:
  - Prompt AI to create a function binary_search(arr, target) returning the index of the target or -1 if not found.
  - Include docstrings explaining best, average, and worst-case complexities.
  - Test with various inputs.
- Expected Output:
  - Python code implementing binary search with AI-generated comments and docstrings.

Code:

```python
def binary_search(arr, target):
    """
    Searches for a target element in a sorted list using Binary Search.
    Binary Search works by repeatedly dividing the search interval
    in half. If the target value is less than the middle element,
    the search continues in the left half; otherwise, it continues
    in the right half.
    Time Complexity:
        Best Case    : O(1)      -> target found at middle immediately
        Average Case: O(log n)  -> repeatedly halves search space
        Worst Case   : O(log n)  -> element not present or found last
    Space Complexity:
        O(1) – iterative version uses constant extra space.
    Parameters:
        arr (list): A sorted list of elements.
        target: The value to search for.
    Returns:
        int: Index of target if found, otherwise -1.
    """
    left = 0
    right = len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        # Check if target is at mid
        if arr[mid] == target:
            return mid
        # If target is greater, ignore left half
        elif arr[mid] < target:
            left = mid + 1
        # If target is smaller, ignore right half
        else:
            right = mid - 1
    return -1
if __name__ == "__main__":
    tests = [
        ([1, 3, 5, 7, 9], 5),
        ([2, 4, 6, 8, 10], 2),
        ([10, 20, 30, 40], 35),
        ([], 1),
        ([5], 5),
        ([1, 2, 3, 4, 5, 6, 7], 7)
    ]

    for arr, target in tests:
        print("Array :", arr)
        print("Target:", target)
        print("Index :", binary_search(arr, target))
        print("-" * 40)
```

Output:

```
Array : [1, 3, 5, 7, 9]
Target: 5
Index : 2
----------------------------------------
Array : [2, 4, 6, 8, 10]
Target: 2
Index : 0
----------------------------------------
Array : [10, 20, 30, 40]
Target: 35
Index : -1
----------------------------------------
Array : []
Target: 1
Index : -1
----------------------------------------
Array : [5]
Target: 5
Index : 0
----------------------------------------
Array : [1, 2, 3, 4, 5, 6, 7]
Target: 7
Index : 6
----------------------------------------
```

Justification:

- Efficient search using divide-and-conquer.
- Requires sorted input.
- Runs in **O(log n)** time instead of **O(n)** like linear search.
- Uses constant memory.

3)

Task Description #3: Smart Healthcare Appointment Scheduling System

A healthcare platform maintains appointment records containing appointment ID, patient name, doctor name, appointment time, and consultation fee. The system needs to:

1. Search appointments using appointment ID.
2. Sort appointments based on time or consultation fee.

Student Task

- Use AI to recommend suitable searching and sorting algorithms.
- Justify the selected algorithms.
- Implement the algorithms in Python.

Code:

```python
class Appointment:
    def __init__(self, aid, patient, doctor, time, fee):
        self.aid = aid
        self.patient = patient
        self.doctor = doctor
        self.time = time
        self.fee = fee
    def __repr__(self):
        return f"{self.aid} | {self.patient} | {self.doctor} | {self.time} | {self.fee}"
# ---------- MERGE SORT ----------
def merge_sort(arr, key):
    if len(arr) <= 1:
        return arr
    mid = len(arr)//2
    left = merge_sort(arr[:mid], key)
    right = merge_sort(arr[mid:], key)
    return merge(left, right, key)
def merge(left, right, key):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if getattr(left[i], key) <= getattr(right[j], key):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# ---------- BINARY SEARCH ----------
def binary_search(arr, target):
    left, right = 0, len(arr)-1
    while left <= right:
        mid = (left+right)//2
        if arr[mid].aid == target:
            return arr[mid]
        elif arr[mid].aid < target:
            left = mid + 1
        else:
            right = mid - 1
    return None
# ---------- TEST DATA ----------
appointments = [
    Appointment(103,"Asha","Dr.Smith","10:30",500),
    Appointment(101,"Ravi","Dr.John","09:00",300),
    Appointment(105,"Meena","Dr.Khan","12:00",700),
    Appointment(102,"Arun","Dr.John","09:30",300),
]
# Sort by appointment ID for searching
appointments = merge_sort(appointments, "aid")
print("Sorted by ID:")
for a in appointments:
    print(a)
print("\nSearch ID 102:")
print(binary_search(appointments, 102))
# Sort by time
print("\nSorted by Time:")
for a in merge_sort(appointments, "time"):
    print(a)
# Sort by fee
print("\nSorted by Fee:")
for a in merge_sort(appointments, "fee"):
    print(a)
```

Output:

```
Sorted by ID:
101 | Ravi | Dr.John | 09:00 | 300
102 | Arun | Dr.John | 09:30 | 300
103 | Asha | Dr.Smith | 10:30 | 500
105 | Meena | Dr.Khan | 12:00 | 700

Search ID 102:
102 | Arun | Dr.John | 09:30 | 300

Sorted by Time:
101 | Ravi | Dr.John | 09:00 | 300
102 | Arun | Dr.John | 09:30 | 300
103 | Asha | Dr.Smith | 10:30 | 500
105 | Meena | Dr.Khan | 12:00 | 700

Sorted by Fee:
101 | Ravi | Dr.John | 09:00 | 300
102 | Arun | Dr.John | 09:30 | 300
103 | Asha | Dr.Smith | 10:30 | 500
Sorted by Fee:
101 | Ravi | Dr.John | 09:00 | 300
102 | Arun | Dr.John | 09:30 | 300
103 | Asha | Dr.Smith | 10:30 | 500
103 | Asha | Dr.Smith | 10:30 | 500
105 | Meena | Dr.Khan | 12:00 | 700
```

Justification:

- **Binary Search** is chosen over linear search because it drastically reduces search time for large datasets.
- **Merge Sort** is preferred over Quick Sort because it has consistent worst-case performance and maintains order of equal elements (important in scheduling systems).
- Both algorithms are reliable and scalable for real healthcare platforms.

Searching (Appointment ID):

- **Binary Search**
  - Efficient for searching in sorted data.
  - Time Complexity: **O(log n)**
  - Suitable because appointment IDs are unique and can be sorted once and searched many times.

Sorting (Time or Fee):

- **Merge Sort**
  - Stable and guarantees **O(n log n)** time.
  - Works well for structured records like appointments.
  - Stability is useful when multiple records have same time/fee.

4)

Task Description #4: Railway Ticket Reservation System

Scenario

A railway reservation system stores booking details such as ticket ID, passenger name, train number, seat number, and travel date. The system must:

1. Search tickets using ticket ID.
2. Sort bookings based on travel date or seat number.

Student Task

- Identify efficient algorithms using AI assistance.
- Justify the algorithm choices.
- Implement searching and sorting in Python.

Code:

```python
class Ticket:
    def __init__(self, tid, name, train, seat, date):
        self.tid = tid
        self.name = name
        self.train = train
        self.seat = seat
        self.date = date
    def __repr__(self):
        return f"{self.tid} | {self.name} | Train:{self.train} | Seat:{self.seat} | {self.date}"
# -------- MERGE SORT --------
def merge_sort(arr, key):
    if len(arr) <= 1:
        return arr
    mid = len(arr)//2
    left = merge_sort(arr[:mid], key)
    right = merge_sort(arr[mid:], key)
    return merge(left, right, key)
def merge(left, right, key):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if getattr(left[i], key) <= getattr(right[j], key):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# -------- BINARY SEARCH --------
def binary_search(arr, target):
    left, right = 0, len(arr)-1
    while left <= right:
        mid = (left+right)//2
        if arr[mid].tid == target:
            return arr[mid]
        elif arr[mid].tid < target:
            left = mid + 1
        else:
            right = mid - 1
    return None
# -------- TEST DATA --------
tickets = [
    Ticket(203,"Ravi",12627,45,"2026-03-10"),
    Ticket(201,"Asha",12015,12,"2026-02-28"),
    Ticket(205,"Meena",12627,22,"2026-03-12"),
    Ticket(202,"Arun",12015,30,"2026-02-28"),
]
# Sort by Ticket ID for searching
tickets = merge_sort(tickets, "tid")
print("Sorted by Ticket ID:")
for t in tickets:
    print(t)
print("\nSearch Ticket 202:")
print(binary_search(tickets, 202))
# Sort by travel date
print("\nSorted by Date:")
for t in merge_sort(tickets, "date"):
    print(t)
# Sort by seat number
print("\nSorted by Seat:")
for t in merge_sort(tickets, "seat"):
    print(t)
```

Output:

```
Sorted by Ticket ID:
201 | Asha | Train:12015 | Seat:12 | 2026-02-28
202 | Arun | Train:12015 | Seat:30 | 2026-02-28
203 | Ravi | Train:12627 | Seat:45 | 2026-03-10
205 | Meena | Train:12627 | Seat:22 | 2026-03-12

Search Ticket 202:
202 | Arun | Train:12015 | Seat:30 | 2026-02-28

Sorted by Date:
201 | Asha | Train:12015 | Seat:12 | 2026-02-28
202 | Arun | Train:12015 | Seat:30 | 2026-02-28
203 | Ravi | Train:12627 | Seat:45 | 2026-03-10
205 | Meena | Train:12627 | Seat:22 | 2026-03-12

Sorted by Seat:
201 | Asha | Train:12015 | Seat:12 | 2026-02-28
205 | Meena | Train:12627 | Seat:22 | 2026-03-12
202 | Arun | Train:12015 | Seat:30 | 2026-02-28
203 | Ravi | Train:12627 | Seat:45 | 2026-03-10
```

Justification:

Binary Search is chosen because ticket IDs are unique and searching is frequent; it is much faster than linear search.

Merge Sort is chosen because it guarantees good performance even for worst cases and preserves record order when keys match.

Together, they ensure the system is scalable, efficient, and reliable for real-world reservation systems.

Efficient Algorithms Selection

Searching (Ticket ID):

- **Binary Search**
  - Works best for searching unique IDs in sorted records.
  - Time Complexity: **O(log n)** — very fast for large booking databases.

Sorting (Travel Date or Seat Number):

- **Merge Sort**
  - Stable and consistent **O(n log n)** performance.
  - Ideal for structured records.
  - Maintains order when values are equal (important for same date bookings).

5)

Task Description #5: Smart Hostel Room Allocation System

A hostel management system stores student room allocation details including student ID, room number, floor, and allocation date. The system needs to:

1. Search allocation details using student ID.
2. Sort records based on room number or allocation date.

Student Task

- Use AI to suggest optimized algorithms.
- Justify the selections.
- Implement the solution in Python.

Code:

```python
class Allocation:
    def __init__(self, sid, room, floor, date):
        self.sid = sid
        self.room = room
        self.floor = floor
        self.date = date
    def __repr__(self):
        return f"{self.sid} | Room:{self.room} | Floor:{self.floor} | {self.date}"
# -------- MERGE SORT --------
def merge_sort(arr, key):
    if len(arr) <= 1:
        return arr
    mid = len(arr)//2
    left = merge_sort(arr[:mid], key)
    right = merge_sort(arr[mid:], key)
    return merge(left, right, key)
def merge(left, right, key):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if getattr(left[i], key) <= getattr(right[j], key):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# -------- BINARY SEARCH --------
def binary_search(arr, target):
    left, right = 0, len(arr)-1
    while left <= right:
        mid = (left+right)//2
        if arr[mid].sid == target:
            return arr[mid]
        elif arr[mid].sid < target:
            left = mid + 1
        else:
            right = mid - 1
    return None
# -------- TEST DATA --------
records = [
    Allocation(302,"A12",1,"2026-02-01"),
    Allocation(301,"B05",2,"2026-01-28"),
    Allocation(305,"A02",1,"2026-02-05"),
    Allocation(303,"C10",3,"2026-02-03"),
]
# Sort by Student ID for searching
records = merge_sort(records, "sid")
print("Sorted by Student ID:")
for r in records:
    print(r)
print("\nSearch Student 303:")
print(binary_search(records, 303))
# Sort by Room Number
print("\nSorted by Room:")
for r in merge_sort(records, "room"):
    print(r)
# Sort by Allocation Date
print("\nSorted by Date:")
for r in merge_sort(records, "date"):
    print(r)
```

Output:

```
Sorted by Student ID:
301 | Room:B05 | Floor:2 | 2026-01-28
302 | Room:A12 | Floor:1 | 2026-02-01
303 | Room:C10 | Floor:3 | 2026-02-03
305 | Room:A02 | Floor:1 | 2026-02-05

Search Student 303:
303 | Room:C10 | Floor:3 | 2026-02-03

Sorted by Room:
305 | Room:A02 | Floor:1 | 2026-02-05
302 | Room:A12 | Floor:1 | 2026-02-01
303 | Room:C10 | Floor:3 | 2026-02-03
305 | Room:A02 | Floor:1 | 2026-02-05

Search Student 303:
303 | Room:C10 | Floor:3 | 2026-02-03

Sorted by Room:
305 | Room:A02 | Floor:1 | 2026-02-05
302 | Room:A12 | Floor:1 | 2026-02-01
Sorted by Room:
305 | Room:A02 | Floor:1 | 2026-02-05
302 | Room:A12 | Floor:1 | 2026-02-01
302 | Room:A12 | Floor:1 | 2026-02-01
301 | Room:B05 | Floor:2 | 2026-01-28
303 | Room:C10 | Floor:3 | 2026-02-03

Sorted by Date:
301 | Room:B05 | Floor:2 | 2026-01-28
302 | Room:A12 | Floor:1 | 2026-02-01
303 | Room:C10 | Floor:3 | 2026-02-03
305 | Room:A02 | Floor:1 | 2026-02-05
```

Justification:

Binary Search is faster than linear search for large datasets and is perfect for student IDs because they are unique.
Merge Sort is chosen because it guarantees consistent performance and is stable, ensuring records with the same room/date maintain their original order.
These algorithms together make the hostel system scalable and efficient.

1. Searching (Student ID):

- **Binary Search**
  - Efficient for unique IDs stored in sorted order.
  - Time Complexity: **O(log n)**
  - Ideal when searches are frequent.

2. Sorting (Room Number or Allocation Date):

- **Merge Sort**
  - Stable sorting algorithm.
  - Time Complexity: **O(n log n)** in all cases.
  - Works well for structured records and preserves order of equal values.

6)

Task Description #6: Online Movie Streaming Platform
A streaming service maintains movie records with movie ID, title, genre, rating, and release year. The platform needs to:

1. Search movies by movie ID.
2. Sort movies based on rating or release year.

Student Task

- Recommend searching and sorting algorithms using AI.
- Justify the chosen algorithms.
- Implement Python functions.

Code:

```python
class Movie:
    def __init__(self, mid, title, genre, rating, year):
        self.mid = mid
        self.title = title
        self.genre = genre
        self.rating = rating
        self.year = year
    def __repr__(self):
        return f"{self.mid} | {self.title} | {self.genre} | ⭐{self.rating} | {self.year}"
# ---------- MERGE SORT ----------
def merge_sort(arr, key):
    if len(arr) <= 1:
        return arr
    mid = len(arr)//2
    left = merge_sort(arr[:mid], key)
    right = merge_sort(arr[mid:], key)
    return merge(left, right, key)
def merge(left, right, key):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if getattr(left[i], key) <= getattr(right[j], key):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# ---------- BINARY SEARCH ----------
def binary_search(arr, target):
    left, right = 0, len(arr)-1
    while left <= right:
        mid = (left+right)//2
        if arr[mid].mid == target:
            return arr[mid]
        elif arr[mid].mid < target:
            left = mid + 1
        else:
            right = mid - 1
    return None
# ---------- TEST DATA ----------
movies = [
    Movie(103,"Inception","Sci-Fi",8.8,2010),
    Movie(101,"Avatar","Fantasy",7.9,2009),
    Movie(105,"Interstellar","Sci-Fi",8.6,2014),
    Movie(102,"Titanic","Romance",7.8,1997),
]
# Sort by ID for searching
movies = merge_sort(movies, "mid")
print("Sorted by Movie ID:")
for m in movies:
    print(m)
print("\nSearch Movie ID 102:")
print(binary_search(movies, 102))
# Sort by Rating
print("\nSorted by Rating:")
for m in merge_sort(movies, "rating"):
    print(m)
# Sort by Release Year
print("\nSorted by Year:")
for m in merge_sort(movies, "year"):
    print(m)
```

Output:

```
Sorted by Movie ID:
101 | Avatar | Fantasy | ⭐7.9 | 2009
102 | Titanic | Romance | ⭐7.8 | 1997
103 | Inception | Sci-Fi | ⭐8.8 | 2010
105 | Interstellar | Sci-Fi | ⭐8.6 | 2014

Search Movie ID 102:
102 | Titanic | Romance | ⭐7.8 | 1997

Sorted by Rating:
102 | Titanic | Romance | ⭐7.8 | 1997
101 | Avatar | Fantasy | ⭐7.9 | 2009
105 | Interstellar | Sci-Fi | ⭐8.6 | 2014
103 | Inception | Sci-Fi | ⭐8.8 | 2010

Sorted by Year:
102 | Titanic | Romance | ⭐7.8 | 1997
101 | Avatar | Fantasy | ⭐7.9 | 2009
103 | Inception | Sci-Fi | ⭐8.8 | 2010
105 | Interstellar | Sci-Fi | ⭐8.6 | 2014
```

Justification:

Binary Search is optimal because movie IDs are unique and searching is frequent. It is much faster than linear search for large movie databases.
Merge Sort is reliable and stable, meaning if two movies have the same rating or year, their original order is preserved.
Both algorithms scale well as the platform grows.

Searching (Movie ID):

- **Binary Search**
    - Efficient for searching unique IDs.
    - Time Complexity: **O(log n)** when data is sorted by movie ID.

Sorting (Rating or Release Year):

- **Merge Sort**
    - Stable and guarantees **O(n log n)** time in all cases.
    - Suitable for structured records like movie objects.

7)

Task Description #7: Smart Agriculture Crop Monitoring System

An agriculture monitoring system stores crop data with crop ID, crop name, soil moisture level, temperature, and yield estimate. Farmers need to:

1. Search crop details using crop ID.
2. Sort crops based on moisture level or yield estimate.

Student Task

- Use AI-assisted reasoning to select algorithms.
- Justify algorithm suitability.
- Implement searching and sorting in Python.

Code:

```python
class Crop:
    def __init__(self, cid, name, moisture, temp, yield_est):
        self.cid = cid
        self.name = name
        self.moisture = moisture
        self.temp = temp
        self.yield_est = yield_est
    def __repr__(self):
        return f"{self.cid} | {self.name} | Moisture:{self.moisture} | Temp:{self.temp} | Yield:{self.yield_est}"
# -------- MERGE SORT --------
def merge_sort(arr, key):
    if len(arr) <= 1:
        return arr
    mid = len(arr)//2
    left = merge_sort(arr[:mid], key)
    right = merge_sort(arr[mid:], key)
    return merge(left, right, key)
def merge(left, right, key):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if getattr(left[i], key) <= getattr(right[j], key):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# -------- BINARY SEARCH --------
def binary_search(arr, target):
    left, right = 0, len(arr)-1
    while left <= right:
        mid = (left+right)//2
        if arr[mid].cid == target:
            return arr[mid]
        elif arr[mid].cid < target:
            left = mid + 1
        else:
            right = mid - 1
    return None
# -------- TEST DATA --------
crops = [
    Crop(503,"Wheat",45,30,80),
    Crop(501,"Rice",60,28,95),
    Crop(505,"Corn",50,32,85),
    Crop(502,"Barley",40,29,70),
]
# Sort by Crop ID for searching
crops = merge_sort(crops, "cid")
print("Sorted by Crop ID:")
for c in crops:
    print(c)
print("\nSearch Crop ID 502:")
print(binary_search(crops, 502))
# Sort by Moisture Level
print("\nSorted by Moisture:")
for c in merge_sort(crops, "moisture"):
    print(c)
# Sort by Yield Estimate
print("\nSorted by Yield:")
for c in merge_sort(crops, "yield_est"):
    print(c)
```

Output:

```
Sorted by Crop ID:
501 | Rice | Moisture:60 | Temp:28 | Yield:95
502 | Barley | Moisture:40 | Temp:29 | Yield:70
503 | Wheat | Moisture:45 | Temp:30 | Yield:80
505 | Corn | Moisture:50 | Temp:32 | Yield:85

Search Crop ID 502:
502 | Barley | Moisture:40 | Temp:29 | Yield:70

Sorted by Moisture:
502 | Barley | Moisture:40 | Temp:29 | Yield:70
503 | Wheat | Moisture:45 | Temp:30 | Yield:80
505 | Corn | Moisture:50 | Temp:32 | Yield:85
501 | Rice | Moisture:60 | Temp:28 | Yield:95

Sorted by Yield:
502 | Barley | Moisture:40 | Temp:29 | Yield:70
502 | Barley | Moisture:40 | Temp:29 | Yield:70
503 | Wheat | Moisture:45 | Temp:30 | Yield:80
505 | Corn | Moisture:50 | Temp:32 | Yield:85

Search Crop ID 502:
502 | Barley | Moisture:40 | Temp:29 | Yield:70

Sorted by Moisture:
502 | Barley | Moisture:40 | Temp:29 | Yield:70
503 | Wheat | Moisture:45 | Temp:30 | Yield:80
505 | Corn | Moisture:50 | Temp:32 | Yield:85
501 | Rice | Moisture:60 | Temp:28 | Yield:95

Sorted by Yield:
502 | Barley | Moisture:40 | Temp:29 | Yield:70
503 | Wheat | Moisture:45 | Temp:30 | Yield:80
501 | Rice | Moisture:60 | Temp:28 | Yield:95

Sorted by Yield:
502 | Barley | Moisture:40 | Temp:29 | Yield:70
503 | Wheat | Moisture:45 | Temp:30 | Yield:80
505 | Corn | Moisture:50 | Temp:32 | Yield:85
503 | Wheat | Moisture:45 | Temp:30 | Yield:80
505 | Corn | Moisture:50 | Temp:32 | Yield:85
505 | Corn | Moisture:50 | Temp:32 | Yield:85
501 | Rice | Moisture:60 | Temp:28 | Yield:95
```

Justification:

Binary Search is faster than linear search for large agricultural datasets and ensures quick retrieval of crop information.
Merge Sort is reliable and stable, which is important when multiple crops share the same moisture or yield values.
Both algorithms scale efficiently as data grows in smart farming systems.

Searching (Crop ID):

- **Binary Search**
  - Efficient for searching unique crop IDs.
  - Time Complexity: **O(log n)** when data is sorted by ID.
  - Suitable because IDs are unique and searches are frequent.

Sorting (Moisture Level or Yield Estimate):

- **Merge Sort**
  - Stable algorithm with guaranteed **O(n log n)** performance.
  - Works well for structured datasets.
  - Maintains original order if values are equal.

8)

Task Description #8: Airport Flight Management System

An airport system stores flight information including flight ID, airline name, departure time, arrival time, and status. The system must:

1. Search flight details using flight ID.
2. Sort flights based on departure time or arrival time.

Student Task

- Use AI to recommend algorithms.
- Justify the algorithm selection.
- Implement searching and sorting logic in Python.

Code:

```python
class Flight:
    def __init__(self, fid, airline, dep, arr, status):
        self.fid = fid
        self.airline = airline
        self.dep = dep
        self.arr = arr
        self.status = status
    def __repr__(self):
        return f"{self.fid} | {self.airline} | Dep:{self.dep} | Arr:{self.arr} | {self.status}"
# -------- MERGE SORT --------
def merge_sort(arr, key):
    if len(arr) <= 1:
        return arr
    mid = len(arr)//2
    left = merge_sort(arr[:mid], key)
    right = merge_sort(arr[mid:], key)
    return merge(left, right, key)
def merge(left, right, key):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if getattr(left[i], key) <= getattr(right[j], key):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# -------- BINARY SEARCH --------
def binary_search(arr, target):
    left, right = 0, len(arr)-1
    while left <= right:
        mid = (left+right)//2
        if arr[mid].fid == target:
            return arr[mid]
        elif arr[mid].fid < target:
            left = mid + 1
        else:
            right = mid - 1
    return None
# -------- TEST DATA --------
flights = [
    Flight(403,"Air India","10:30","12:45","On Time"),
    Flight(401,"IndiGo","09:15","11:00","Delayed"),
    Flight(405,"SpiceJet","14:00","16:10","On Time"),
    Flight(402,"Vistara","10:00","12:20","Boarding"),
]
# Sort by Flight ID for searching
flights = merge_sort(flights, "fid")
print("Sorted by Flight ID:")
for f in flights:
    print(f)
print("\nSearch Flight 402:")
print(binary_search(flights, 402))
# Sort by Departure Time
print("\nSorted by Departure:")
for f in merge_sort(flights, "dep"):
    print(f)
# Sort by Arrival Time
print("\nSorted by Arrival:")
for f in merge_sort(flights, "arr"):
    print(f)
```

Output:

```
Sorted by Flight ID:
401 | IndiGo | Dep:09:15 | Arr:11:00 | Delayed
402 | Vistara | Dep:10:00 | Arr:12:20 | Boarding
403 | Air India | Dep:10:30 | Arr:12:45 | On Time
405 | SpiceJet | Dep:14:00 | Arr:16:10 | On Time

Search Flight 402:
402 | Vistara | Dep:10:00 | Arr:12:20 | Boarding

Sorted by Departure:
401 | IndiGo | Dep:09:15 | Arr:11:00 | Delayed
402 | Vistara | Dep:10:00 | Arr:12:20 | Boarding
403 | Air India | Dep:10:30 | Arr:12:45 | On Time
405 | SpiceJet | Dep:14:00 | Arr:16:10 | On Time

Search Flight 402:
402 | Vistara | Dep:10:00 | Arr:12:20 | Boarding

Sorted by Departure:
401 | IndiGo | Dep:09:15 | Arr:11:00 | Delayed
402 | Vistara | Dep:10:00 | Arr:12:20 | Boarding
Search Flight 402:
402 | Vistara | Dep:10:00 | Arr:12:20 | Boarding

Sorted by Departure:
401 | IndiGo | Dep:09:15 | Arr:11:00 | Delayed
402 | Vistara | Dep:10:00 | Arr:12:20 | Boarding
Sorted by Departure:
401 | IndiGo | Dep:09:15 | Arr:11:00 | Delayed
402 | Vistara | Dep:10:00 | Arr:12:20 | Boarding
403 | Air India | Dep:10:30 | Arr:12:45 | On Time
405 | SpiceJet | Dep:14:00 | Arr:16:10 | On Time

Sorted by Arrival:
401 | IndiGo | Dep:09:15 | Arr:11:00 | Delayed
402 | Vistara | Dep:10:00 | Arr:12:20 | Boarding
403 | Air India | Dep:10:30 | Arr:12:45 | On Time
405 | SpiceJet | Dep:14:00 | Arr:16:10 | On Time
```

Justification:

Binary Search is optimal because flight IDs are unique and searching is frequent in airport systems. It is much faster than linear search for large flight databases.
Merge Sort is chosen because it provides consistent performance and stability, which is important when multiple flights share the same time.
These algorithms ensure scalability, efficiency, and reliability for real-time airport management.

1. Searching (Flight ID):

- **Binary Search**
  - Efficient for searching unique flight IDs.
  - Time Complexity: **O(log n)**
  - Best suited when flight records are sorted by ID.

2. Sorting (Departure or Arrival Time):

- **Merge Sort**
  - Stable and guarantees **O(n log n)** time.
  - Ideal for structured records.
  - Preserves order for flights with same time.