

# Assignment-12.5

Hallticket:2303A51185

Batch:03

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:

```
ass_12_5.py > ...
1 from typing import List, Any
2 def merge_sort(arr: List[Any]) -> List[Any]:
3     """Sort a list in ascending order using merge sort.
4     Parameters
5     -----
6     arr : List[Any]
7         The list of comparable elements to sort.
8     Returns
9     -----
10    List[Any]
11        A new list containing the sorted elements.
12    Time Complexity: O(n log n) on all cases.
13    Space Complexity: O(n) due to auxiliary lists used during merging.
14    """
15    # Base case: lists with 0 or 1 element are already sorted
16    if len(arr) <= 1:
17        return arr[:]
18    mid = len(arr) // 2
19    left = merge_sort(arr[:mid])
20    right = merge_sort(arr[mid:])
21    # merge two sorted lists
22    merged: List[Any] = []
23    i = j = 0
24    while i < len(left) and j < len(right):
25        if left[i] <= right[j]:
26            merged.append(left[i])
27            i += 1
28        else:
29            merged.append(right[j])
30            j += 1
31    # append any remaining elements
32    merged.extend(left[i:])
33    merged.extend(right[j:])
34    return merged
35 if __name__ == "__main__":
36     # simple demo when run as a script
37     sample = [38, 27, 43, 3, 9, 82, 10]
38     print("Original:", sample)
39     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:

```
1 def binary_search(arr, target):
2     """
3     Searches for a target element in a sorted list using Binary Search.
4     Binary Search works by repeatedly dividing the search interval
5     in half. If the target value is less than the middle element,
6     the search continues in the left half; otherwise, it continues
7     in the right half.
8     Time Complexity:
9         Best Case : O(1)      -> target found at middle immediately
10        Average Case: O(log n) -> repeatedly halves search space
11        Worst Case : O(log n) -> element not present or found last
12    Space Complexity:
13        O(1) - iterative version uses constant extra space.
14    Parameters:
15        arr (list): A sorted list of elements.
16        target: The value to search for.
17    Returns:
18        int: Index of target if found, otherwise -1.
19    """
20    left = 0
21    right = len(arr) - 1
22    while left <= right:
23        mid = (left + right) // 2
24        # Check if target is at mid
25        if arr[mid] == target:
26            return mid
27        # If target is greater, ignore left half
28        elif arr[mid] < target:
29            left = mid + 1
30        # If target is smaller, ignore right half
31        else:
32            right = mid - 1
33    return -1
34 if __name__ == "__main__":
35     tests = [
36         ([1, 3, 5, 7, 9], 5),
37         ([2, 4, 6, 8, 10], 2),
38         ([10, 20, 30, 40], 35),
39         ([], 1),
40         ([5], 5),
41         ([1, 2, 3, 4, 5, 6, 7], 7)
42     ]
43
44     for arr, target in tests:
45         print("Array :", arr)
46         print("Target:", target)
47         print("Index :", binary_search(arr, target))
48         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:

```

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
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:

```

1 class Ticket:
2     def __init__(self, tid, name, train, seat, date):
3         self.tid = tid
4         self.name = name
5         self.train = train
6         self.seat = seat
7         self.date = date
8     def __repr__(self):
9         return f"({self.tid} | {self.name} | Train:{self.train} | Seat:{self.seat} | {self.date})"
10
11 # ----- MERGE SORT -----
12 def merge_sort(arr, key):
13     if len(arr) <= 1:
14         return arr
15     mid = len(arr)//2
16     left = merge_sort(arr[:mid], key)
17     right = merge_sort(arr[mid:], key)
18     return merge(left, right, key)
19 def merge(left, right, key):
20     result = []
21     i = j = 0
22     while i < len(left) and j < len(right):
23         if getattr(left[i], key) <= getattr(right[j], key):
24             result.append(left[i])
25             i += 1
26         else:
27             result.append(right[j])
28             j += 1
29     result.extend(left[i:])
30     result.extend(right[j:])
31     return result
32
33 # ----- BINARY SEARCH -----
34 def binary_search(arr, target):
35     left, right = 0, len(arr)-1
36     while left <= right:
37         mid = (left+right)//2
38         if arr[mid].tid == target:
39             return arr[mid]
40         elif arr[mid].tid < target:
41             left = mid + 1
42         else:
43             right = mid - 1
44     return None
45
46 # ----- TEST DATA -----
47 tickets = [
48     Ticket(203, "Ravi", 12627, 45, "2026-03-10"),
49     Ticket(201, "Asha", 12015, 12, "2026-02-28"),
50     Ticket(205, "Meena", 12627, 22, "2026-03-12"),
51     Ticket(202, "Arun", 12015, 30, "2026-02-28"),
52 ]
53
54 # Sort by Ticket ID for searching
55 tickets = merge_sort(tickets, "tid")
56 print("Sorted by Ticket ID:")
57 for t in tickets:
58     print(t)
59
60 print("\nSearch Ticket 202:")
61 print(binary_search(tickets, 202))
62
63 # Sort by travel date
64 print("\nSorted by Date:")
65 for t in merge_sort(tickets, "date"):
66     print(t)
67
68 # Sort by seat number
69 print("\nSorted by Seat:")
70 for t in merge_sort(tickets, "seat"):
71     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:



```

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
301 | Room:B05 | Floor:2 | 2026-01-28

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:

```

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:

```

1 class Crop:
2     def __init__(self, cid, name, moisture, temp, yield_est):
3         self.cid = cid
4         self.name = name
5         self.moisture = moisture
6         self.temp = temp
7         self.yield_est = yield_est
8     def __repr__(self):
9         return f"{self.cid} | {self.name} | Moisture:{self.moisture} | Temp:{self.temp} | Yield:{self.yield_est}"
10
11 # ----- MERGE SORT -----
12 def merge_sort(arr, key):
13     if len(arr) <= 1:
14         return arr
15     mid = len(arr)//2
16     left = merge_sort(arr[:mid], key)
17     right = merge_sort(arr[mid:], key)
18     return merge(left, right, key)
19 def merge(left, right, key):
20     result = []
21     i = j = 0
22     while i < len(left) and j < len(right):
23         if getattr(left[i], key) <= getattr(right[j], key):
24             result.append(left[i])
25             i += 1
26         else:
27             result.append(right[j])
28             j += 1
29     result.extend(left[i:])
30     result.extend(right[j:])
31     return result
32
33 # ----- BINARY SEARCH -----
34 def binary_search(arr, target):
35     left, right = 0, len(arr)-1
36     while left <= right:
37         mid = (left+right)//2
38         if arr[mid].cid == target:
39             return arr[mid]
40         elif arr[mid].cid < target:
41             left = mid + 1
42         else:
43             right = mid - 1
44     return None
45
46 # ----- TEST DATA -----
47 crops = [
48     Crop(503, "Wheat", 45, 30, 80),
49     Crop(501, "Rice", 60, 28, 95),
50     Crop(505, "Corn", 50, 32, 85),
51     Crop(502, "Barley", 40, 29, 70),
52 ]
53
54 # Sort by Crop ID for searching
55 crops = merge_sort(crops, "cid")
56 print("Sorted by Crop ID:")
57 for c in crops:
58     print(c)
59
60 print("\nSearch Crop ID 502:")
61 print(binary_search(crops, 502))
62
63 # Sort by Moisture Level
64 print("\nSorted by Moisture:")
65 for c in merge_sort(crops, "moisture"):
66     print(c)
67
68 # Sort by Yield Estimate
69 print("\nSorted by Yield:")
70 for c in merge_sort(crops, "yield_est"):
71     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
503 | Wheat | Moisture:45 | Temp:30 | Yield:80
505 | Corn | Moisture:50 | Temp:32 | Yield:85
501 | Rice | Moisture:60 | Temp:28 | Yield:95

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
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:

```

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.