# LAB ASSIGNMENT

**NAME:-DONTHI MEGHANA**

**ROLL NO:-2403A510D9**

**BATCH:-05**

**DATE:-22-09-2025**

**Task Description #1 (Sorting – Merge Sort Implementation)**
**• Task: Use AI to generate a Python program that implements the Merge Sort algorithm**

**PROMPT:-**

Generate a Python function merge_sort(arr) that sorts a list in ascending order using the Merge Sort algorithm. Include time and space complexity in the docstring. Verify the code with test cases.

**CODE:-**

```python
def merge_sort(arr):


    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)
```

Zencoder

```python
def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

```python
# Test cases and output
if __name__ == "__main__":
    test_cases = [
        [],
        [1],
        [5, 2, 9, 1, 5, 6],
        [3, 2, 1, 0, -1, -2],
        [10, 20, 30, 40],
        [4, 3, 2, 1],
        [1, 2, 3, 4],
        [7, 7, 7, 7],
    ]
    for i, arr in enumerate(test_cases):
        sorted_arr = merge_sort(arr)
        print(f"Test case {i+1}: {arr} -> {sorted_arr}")
```

**OUTPUT:-**

```
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/12.1.py
Test case 1: [] -> []
Test case 2: [1] -> [1]
Test case 3: [5, 2, 9, 1, 5, 6] -> [1, 2, 5, 5, 6, 9]
Test case 4: [3, 2, 1, 0, -1, -2] -> [-2, -1, 0, 1, 2, 3]
Test case 5: [10, 20, 30, 40] -> [10, 20, 30, 40]
Test case 6: [4, 3, 2, 1] -> [1, 2, 3, 4]
Test case 7: [1, 2, 3, 4] -> [1, 2, 3, 4]
Test case 8: [7, 7, 7, 7] -> [7, 7, 7, 7]
PS C:\Users\Administrator\OneDrive\ai>
```

**OBSERVATIONS:-**

 Correctness:

- The algorithm works for all tested cases, including empty lists, already sorted lists, reverse-sorted lists, and lists with duplicates or negative numbers.

 Stability:

- Merge Sort is stable, meaning equal elements retain their original relative order.

 Efficiency:

- Time complexity remains O(n log n) regardless of input distribution (unlike Quick Sort).
- However, it requires extra space (O(n)), unlike in-place algorithms such as Heap Sort.

 Best Use Cases:

- Works well for linked lists (because splitting and merging are efficient).
- Useful when stable sorting is required.

**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.

**PROMPT:-**

"Create a Python function binary_search(arr, target) that finds the index of a target element in a sorted list using Binary Search. Include time and space complexity in the function docstring. Also add test cases to verify correctness."

**CODE:-**

```python
def binary_search(arr, target):
    """
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

# Test cases and output
if __name__ == "__main__":
    print("\nBinary Search Test Cases:")
    sorted_lists = [
        [],
        [1],
        [1, 2, 3, 4, 5],
        [10, 20, 30, 40, 50],
        [-5, 0, 3, 8, 12],
    ]
    targets = [1, 1, 3, 40, 8]
    for i, (arr, target) in enumerate(zip(sorted_lists, targets)):
        idx = binary_search(arr, target)
        print(f"Test case {i+1}: array={arr}, target={target} -> index={idx}")
    # Edge case: not found
    print(f"Not found case: array={[1, 2, 3]}, target=5 -> index={binary_search([1, 2, 3], 5)}")
```

**OUTPUT:-**

```
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/12.1.3.py

Binary Search Test Cases:
Test case 1: array=[], target=1 -> index=-1
Test case 2: array=[1], target=1 -> index=0
Test case 3: array=[1, 2, 3, 4, 5], target=3 -> index=2
Test case 4: array=[10, 20, 30, 40, 50], target=40 -> index=3
Test case 5: array=[-5, 0, 3, 8, 12], target=8 -> index=3
Not found case: array=[1, 2, 3], target=5 -> index=-1
PS C:\Users\Administrator\OneDrive\ai>
```

**OBSERVATION:-**

**✅ Correctness**

- Works for all tested cases including:
  - Target present in the middle, beginning, or end.
  - Target not present.
  - Empty array.

**✅ Efficiency**

- Much faster than linear search for large sorted datasets.
- Time complexity: O(log n) is optimal for search in sorted lists.

 **Space Usage**

- Iterative version uses **O(1)** space.
- A recursive version would use **O(log n)** stack space.

 **AI Optimization Idea**

- AI could detect whether to use **linear search or binary search** dynamically:
    - For very small lists (e.g., n < 20), **linear search may be faster** due to lower overhead.
    - For large sorted lists, **binary search dominates**.

**Task Description #3 (Real-Time Application – Inventory Management System)**

**o** Use AI to suggest the most efficient search and sort

algorithms for this use case.

o Implement the recommended algorithms in Python.

o Justify the choice based on dataset size, update frequency, and performance requirements.

**PROMPT:-**

"Design a Python-based Inventory Management System where products need to be searched and sorted efficiently. Recommend the best search and sort algorithms for this use case based on dataset size, update frequency, and performance requirements. Implement the recommended algorithms in Python with justification."

**CODE:-**

```python
class Inventory:
    Zencoder
    def __init__(self):
        self.items = {}  # Hash table for fast search

    Zencoder
    def add_item(self, item_id, name, quantity):
        self.items[item_id] = {'name': name, 'quantity': quantity}

    Zencoder
    def update_quantity(self, item_id, quantity):
        if item_id in self.items:
            self.items[item_id]['quantity'] = quantity

    Zencoder
    def search_item(self, item_id):
        return self.items.get(item_id, None)

    Zencoder
    def sorted_inventory(self):
        # Sort by item_id using Timsort (built-in sorted)
        return sorted(self.items.items(), key=lambda x: x[0])

# Test cases and output
if __name__ == "__main__":
    inv = Inventory()
    inv.add_item(102, "Apples", 50)
```

```python
# Test cases and output
if __name__ == "__main__":
    inv = Inventory()
    inv.add_item(102, "Apples", 50)
    inv.add_item(101, "Bananas", 30)
    inv.add_item(103, "Oranges", 20)
    print("Search for item 101:", inv.search_item(101))
    print("Search for item 105 (not found):", inv.search_item(105))
    print("\nSorted Inventory:")
    for item_id, info in inv.sorted_inventory():
        print(f"ID: {item_id}, Name: {info['name']}, Quantity: {info['quantity']}")
    inv.update_quantity(102, 60)
    print("\nAfter updating quantity for Apples:")
    print(inv.search_item(102))
```

**OUTPUT:-**

```
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/12.1.2.PY
Search for item 101: {'name': 'Bananas', 'quantity': 30}
Search for item 105 (not found): None

Sorted Inventory:
ID: 101, Name: Bananas, Quantity: 30
ID: 102, Name: Apples, Quantity: 50
ID: 103, Name: Oranges, Quantity: 20

After updating quantity for Apples:
{'name': 'Apples', 'quantity': 60}
PS C:\Users\Administrator\OneDrive\ai>
```

**OBSERVATIONS:-**

1. **Search Efficiency**
    - Using a dictionary (dict) ensures **O(1)** average lookup for product IDs.
    - Much faster than binary search on sorted lists when updates are frequent.
2. **Sorting Efficiency**
    - sorted() uses **Timsort**, which is adaptive and faster for real-world partially sorted datasets.
    - Sorting is **O(n log n)** and only performed when needed (e.g., generating reports).
3. **Real-Time Suitability**
    - Frequent updates are handled efficiently since dictionaries allow constant-time insertion and update.
    - Sorting is done on-demand, keeping the system responsive.
4. **Scalability**
    - Can scale up to millions of records (subject to memory).
    - Search remains **constant time** regardless of dataset size.