Merge Sort Implementation - Explanation

Overview

This is an **efficient recursive merge sort** algorithm that sorts an array in O(n log n) time.

Function Breakdown

1. `merge(array, start, middle, end, temp)`

Merges two sorted halves of an array into one sorted section.

```
def merge(array, start, middle, end, temp):
    left, right, pos = start, middle + 1, start
```

- `left`: pointer for the left half (starts at `start`, goes to `middle`)
- `right`: pointer for the right half (starts at `middle + 1`, goes to `end`)
- `pos`: position in temp array where we'll place the next element

```
    while left <= middle and right <= end:
        if array[left] <= array[right]:
            temp[pos] = array[left]
            left += 1
        else:
            temp[pos] = array[right]
            right += 1
        pos += 1
```

- Compare elements from both halves
- Put the smaller element into `temp`
- Move the appropriate pointer forward
- This continues until one half is exhausted

```
    while left <= middle:
        temp[pos] = array[left]
        left, pos = left + 1, pos + 1
```

- Copy any remaining elements from the left half

```
    while right <= end:
        temp[pos] = array[right]
        right, pos = right + 1, pos + 1
```

- Copy any remaining elements from the right half

```
    array[start:end + 1] = temp[start:end + 1]
```

- Copy the sorted elements from `temp` back to the original `array`

---

2. `merge_sort_recursive(array, start, end, temp)`

The recursive function that divides the array and calls merge.

```
def merge_sort_recursive(array, start, end, temp):
    if start < end:
```

- **Base case**: if `start >= end`, there's only 1 element (or none), so it's already sorted

```
        middle = (start + end) // 2
```

- Find the middle point to divide the array in half

```
        merge_sort_recursive(array, start, middle, temp)
        merge_sort_recursive(array, middle + 1, end, temp)
```

- **Recursively sort** the left half (from `start` to `middle`)
- **Recursively sort** the right half (from `middle + 1` to `end`)

```
        merge(array, start, middle, end, temp)
```

- **Merge** the two sorted halves together

---

3. `merge_sort(array)`

Main entry point that sets up the sort.

```
def merge_sort(array):
    if len(array) <= 1:
        return array
```

- If array has 0 or 1 element, it's already sorted

```
    temp = [0] * len(array)
```

- Create a temporary array of the same size (allocated once for efficiency)

```
    merge_sort_recursive(array, 0, len(array) - 1, temp)
    return array
```

- Call the recursive function starting with the full array (indices 0 to n-1)
- Return the now-sorted array

---

4. `main()`

Handles user input and output.

```
def main():
    nums = list(map(int, input("Enter Input : ").split()))
```

- Reads a line of space-separated numbers
- Converts them to integers
- Stores in a list

```
    print(merge_sort(nums))
```

- Sorts the list and prints the result

How It Works (Example)

Input: `[3, 1, 4, 2]`

```
Step 1: Split into [3, 1] and [4, 2]
Step 2: Split [3, 1] into [3] and [1]
Step 3: Merge [3] and [1] → [1, 3]
Step 4: Split [4, 2] into [4] and [2]
Step 5: Merge [4] and [2] → [2, 4]
Step 6: Merge [1, 3] and [2, 4] → [1, 2, 3, 4]
```

Visual Representation

```
                      [3, 1, 4, 2]
                     /            \
                [3, 1]            [4, 2]
               /      \          /      \
            [3]      [1]      [4]      [2]
               \      /          \      /
                [1, 3]            [2, 4]
                     \            /
                      [1, 2, 3, 4]
```

Complexity Analysis

Aspect	Complexity	Explanation
Time (Best)	O(n log n)	Always divides array in half
Time (Average)	O(n log n)	Consistent performance
Time (Worst)	O(n log n)	Even on reversed arrays
Space	O(n)	One temporary array
Stable	Yes	Equal elements maintain order
In-place	No	Requires extra space

Key Optimizations in This Implementation

1. **Single temp array**: Allocated once instead of creating many arrays during recursion
2. **Index-based operations**: Uses indices instead of array slicing to avoid copying overhead
3. **In-place updates**: Sorts the array in place after using temp for merging
4. **Efficient memory**: O(n) space instead of O(n log n)

When to Use Merge Sort

☑ Good for:

- Large datasets
- When you need stable sorting
- Linked lists (works great with O(1) space)
- When consistent O(n log n) performance is required

✕ Not ideal for:

- Small arrays (insertion sort is faster)
- Memory-constrained systems (requires O(n) extra space)
- When in-place sorting is required (use quicksort or heapsort)

Code

```python
# ex01.py for Q3(Sort,Search)

def merge(array, start, middle, end, temp):
    left, right, pos = start, middle + 1, start
    while left <= middle and right <= end:
        if array[left] <= array[right]:
            temp[pos] = array[left]
            left += 1
        else:
            temp[pos] = array[right]
            right += 1
        pos += 1

    while left <= middle:
        temp[pos] = array[left]
        left, pos = left + 1, pos + 1

    while right <= end:
        temp[pos] = array[right]
        right, pos = right + 1, pos + 1

    array[start:end + 1] = temp[start:end + 1]

def merge_sort_recursive(array, start, end, temp):
    if start < end:
        middle = (start + end) // 2
        merge_sort_recursive(array, start, middle, temp)
        merge_sort_recursive(array, middle + 1, end, temp)
        merge(array, start, middle, end, temp)

def merge_sort(array):
    if len(array) <= 1:
        return array
    temp = [0] * len(array)
    merge_sort_recursive(array, 0, len(array) - 1, temp)
    return array

def main():
    nums = list(map(int, input("Enter Input : ").split()))
    print(merge_sort(nums))

if __name__ == "__main__":
    main()
```