**Merge Sort Explanation**

Merge Sort is a **divide-and-conquer** algorithm that divides the input array into two halves, sorts each half recursively, and then merges the two sorted halves. It is known for its stable sorting behavior and consistent time complexity.

**Steps Involved in Merge Sort:**

1. **Divide**: Split the array into two halves.
2. **Recursion**: Recursively sort the two halves.
3. **Merge**: Merge the two sorted halves back together in sorted order.

**Merge Sort Process Illustration:**

Given an array: [38, 27, 43, 3, 9, 82, 10]

**Step-by-Step Breakdown:**

1. **Initial Array**: [38, 27, 43, 3, 9, 82, 10]
2. **Divide the array into halves**:
   * Left: [38, 27, 43]
   * Right: [3, 9, 82, 10]
3. **Divide further**:
   * Left: [38, 27], Right: [43]
   * Left: [3, 9], Right: [82, 10]
4. **Continue splitting** until each sub-array has one element:
   * Left: [38], Right: [27]
   * Left: [3], Right: [9]
   * Left: [82], Right: [10]
5. **Merge the sub-arrays back together**:
   * [38] and [27] merged to form [27, 38]
   * [3] and [9] merged to form [3, 9]
   * [82] and [10] merged to form [10, 82]
6. **Final merging**:
   * [27, 38] and [43] merged to form [27, 38, 43]
   * [3, 9] and [10, 82] merged to form [3, 9, 10, 82]
7. **Final sorted array**: [3, 9, 10, 27, 38, 43, 82]

**Visualization of Merge Sort Process:**

* **Initial Array**:

[38, 27, 43, 3, 9, 82, 10]

* **First Split**:

[38, 27, 43] [3, 9, 82, 10]

* **Second Split**:

[38, 27] [43] [3, 9] [82, 10]

* **Third Split**:

[38] [27] [43] [3] [9] [82] [10]

* **Merge Back**:

[27, 38] [43] [3, 9] [10, 82]

* **Final Merge**:

[27, 38, 43] [3, 9, 10, 82]

* **Final Sorted Array**:

[3, 9, 10, 27, 38, 43, 82]

**Python Code for Merge Sort**

def merge\_sort(arr):

"""

Function to implement Merge Sort algorithm.

Args:

arr: List of elements to be sorted.

Returns:

Sorted list.

"""

# Base case: if the array has one or zero elements, it is already sorted

if len(arr) <= 1:

return arr

# Find the middle of the array

mid = len(arr) // 2

# Divide the array into two halves

left\_half = arr[:mid] # Elements from the start to the middle

right\_half = arr[mid:] # Elements from the middle to the end

# Recursively sort both halves

left\_sorted = merge\_sort(left\_half)

right\_sorted = merge\_sort(right\_half)

# Merge the two sorted halves

return merge(left\_sorted, right\_sorted)

def merge(left, right):

"""

Merge two sorted lists into a single sorted list.

Args:

left: A sorted list.

right: A sorted list.

Returns:

A merged and sorted list.

"""

merged = []

i = j = 0

# Merge the two lists by comparing elements

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

# If there are remaining elements in left or right, append them

merged.extend(left[i:])

merged.extend(right[j:])

return merged

# Example usage:

arr = [38, 27, 43, 3, 9, 82, 10]

sorted\_arr = merge\_sort(arr)

print("Sorted Array:", sorted\_arr)

**Explanation of the Code:**

1. **Base case**: If the length of the array is 1 or 0, it is already sorted, and we return it.
2. **Divide**: We find the middle index of the array using len(arr) // 2, and split the array into two halves, left\_half and right\_half.
3. **Recursion**: We recursively apply merge\_sort to both halves of the array.
4. **Merge**: After sorting the two halves, the merge function combines them into a single sorted array by comparing their elements and appending the smaller one to the merged list.

**Time Complexity:**

* **Best case**: O(nlogn)O(n \log n)O(nlogn)
* **Worst case**: O(nlogn)O(n \log n)O(nlogn)
* **Average case**: O(nlogn)O(n \log n)O(nlogn)

**Space Complexity:**

* O(n)O(n)O(n) because we create temporary arrays for left and right sub-arrays during the merge step.

**Key Points:**

* **Stable Sort**: Merge Sort preserves the relative order of equal elements.
* **Consistent performance**: Merge Sort guarantees O(nlog⁡n)O(n \log n)O(nlogn) time complexity, which is efficient for large datasets.

Here's a Python function to implement **Merge Sort** that accepts input from the user:

def merge\_sort(arr):

# Base case: If the list has one element, it's already sorted

if len(arr) <= 1:

return arr

# Find the middle point and divide the list into two halves

mid = len(arr) // 2

left\_half = arr[:mid]

right\_half = arr[mid:]

# Recursively split and merge the left and right halves

left\_half = merge\_sort(left\_half)

right\_half = merge\_sort(right\_half)

# Merge the sorted halves

return merge(left\_half, right\_half)

def merge(left, right):

# Merge two sorted lists into one sorted list

sorted\_arr = []

i = j = 0

# Compare elements from both halves and append the smaller one to the sorted array

while i < len(left) and j < len(right):

if left[i] < right[j]:

sorted\_arr.append(left[i])

i += 1

else:

sorted\_arr.append(right[j])

j += 1

# Append remaining elements from left or right (if any)

sorted\_arr.extend(left[i:])

sorted\_arr.extend(right[j:])

return sorted\_arr

# Input from the user

user\_input = input("Enter a list of numbers separated by spaces: ")

# Convert the input string into a list of integers

arr = list(map(int, user\_input.split()))

# Perform merge sort

sorted\_arr = merge\_sort(arr)

# Output the sorted list

print("Sorted list:", sorted\_arr)

**Explanation:**

1. **merge\_sort(arr)**: This is the main function that divides the input list into smaller sub-lists and sorts them recursively.
2. **merge(left, right)**: This helper function merges two sorted sub-lists into one sorted list by comparing their elements.
3. **User Input**: The program asks the user to input a list of numbers separated by spaces. It then splits this string and converts it to a list of integers for sorting.

**Example usage:**

Enter a list of numbers separated by spaces: 34 7 23 32 5 62

Sorted list: [5, 7, 23, 32, 34, 62]