**Merge Sort: Explanation, Algorithm, and Complexity**

Merge sort is a classic **divide-and-conquer** algorithm that breaks the list into smaller pieces, sorts those pieces, and then merges them back together into a single sorted list. This allows it to sort large lists efficiently.

**1. What is Merge Sort?**

Merge sort works by recursively splitting the list into two halves until each half has only one element or is empty. Since a list with one element is considered sorted, we can merge the sorted halves back together. The merging process is efficient and takes advantage of the fact that the two lists being merged are already sorted.

**2. Merge Sort Algorithm:**

The key idea behind merge sort is to **split** the list, **sort** each half recursively, and then **merge** the sorted halves. The merging step is crucial because it combines two sorted lists into one sorted list in linear time.

**Steps in the Merge Sort Algorithm:**

1. **Base Case**: If the list has one element or no elements, it's already sorted. Return the list.
2. **Recursive Case**:
   * Split the list into two halves.
   * Recursively apply merge sort to each half to sort them.
   * Merge the two sorted halves into a single sorted list.

**3. Merge Sort in Detail:**

**Merge Algorithm:**

The merge step is the core part of merge sort. Here’s how it works:

1. Start with two already sorted lists.
2. Use pointers to keep track of the current element in each list.
3. Compare the current elements from both lists.
4. Add the smaller of the two to the new merged list and move the pointer forward for the list from which the smaller element came.
5. Repeat until one list is exhausted, then copy the remaining elements from the other list into the merged list.

The merging step takes **O(n)** time, where **n** is the total number of elements in both lists.

**Merge Sort Algorithm Outline:**

1. **Start** with the entire list.
2. **Split** the list into two halves.
3. **Recursively** apply merge sort to each half, which sorts them.
4. **Merge** the two sorted halves back together into a single sorted list.

**4. Python Implementation:**

**merge\_sort() Function:**

This is the main function that starts the merge sort process. It creates an empty list to use during merging and calls the recursive function.

def merge\_sort(mylist):

empty\_list = [0] \* len(mylist) # Create an empty list for merging

recursive\_merge\_sort(mylist, 0, len(mylist) - 1, empty\_list) # Call recursive merge sort

**recursive\_merge\_sort() Function:**

This function recursively splits the list and calls itself for each half until it reaches the base case, where the list has 0 or 1 element.

python

Copy code

def recursive\_merge\_sort(mylist, first\_index, last\_index, empty\_list):

if first\_index < last\_index:

mid\_index = (first\_index + last\_index) // 2 # Find the middle of the list

recursive\_merge\_sort(mylist, first\_index, mid\_index, empty\_list) # Sort the first half

recursive\_merge\_sort(mylist, mid\_index + 1, last\_index, empty\_list) # Sort the second half

merge(mylist, first\_index, mid\_index + 1, last\_index, empty\_list) # Merge the two halves

**merge() Function:**

The merge function merges two sorted sections of the list into one. It takes the indexes of the two halves and merges them into the original list using the empty list as temporary storage.

python

Copy code

def merge(mylist, a\_first\_index, b\_first\_index, b\_last\_index, empty\_list):

a\_ptr = a\_first\_index # Pointer for the first half

b\_ptr = b\_first\_index # Pointer for the second half

empty\_list\_index = a\_ptr # Start from the first index

# Merge until one list is exhausted

while (a\_ptr < b\_first\_index) and (b\_ptr <= b\_last\_index):

if mylist[a\_ptr] <= mylist[b\_ptr]: # Compare and take smaller element

empty\_list[empty\_list\_index] = mylist[a\_ptr]

a\_ptr += 1

else:

empty\_list[empty\_list\_index] = mylist[b\_ptr]

b\_ptr += 1

empty\_list\_index += 1

# Copy remaining elements from the first half, if any

while a\_ptr < b\_first\_index:

empty\_list[empty\_list\_index] = mylist[a\_ptr]

empty\_list\_index += 1

a\_ptr += 1

# Copy remaining elements from the second half, if any

while b\_ptr <= b\_last\_index:

empty\_list[empty\_list\_index] = mylist[b\_ptr]

empty\_list\_index += 1

b\_ptr += 1

# Copy back from empty\_list to the original mylist

for i in range(a\_first\_index, b\_last\_index + 1):

mylist[i] = empty\_list[i]

**5. Time Complexity of Merge Sort:**

Merge sort has a time complexity of **O(n log n)** for all cases (best, worst, and average). Here's why:

* **Splitting** the list takes **O(log n)** time because each split divides the list into two halves, and it takes log n splits to reduce the list to individual elements.
* **Merging** takes **O(n)** time because each merge step combines the two halves into a sorted list, and we do this **n** times.

Thus, the overall time complexity is **O(n log n)**.

**6. Space Complexity of Merge Sort:**

Merge sort uses **extra space** because it requires additional storage for merging (the empty\_list in the implementation). This extra space is proportional to the size of the list. Therefore, the **space complexity** is **O(n)**.

**7. Example Walkthrough:**

Let’s walk through an example of how **merge sort** works on a small list:

**Input List:**

[5, 2, 9, 1, 6]

1. **Split** the list into two halves: [5, 2, 9] and [1, 6].
2. **Recursively split** the first half into [5] and [2, 9], and then split [2, 9] into [2] and [9].
3. **Merge** [2] and [9] to get [2, 9].
4. **Merge** [5] and [2, 9] to get [2, 5, 9].
5. **Recursively split** the second half [1, 6] into [1] and [6], then merge them to get [1, 6].
6. **Merge** [2, 5, 9] and [1, 6] to get the final sorted list [1, 2, 5, 6, 9].

**8. How to Explain Merge Sort in a Quiz:**

If you’re asked to explain the **time and space complexity** of merge sort in a quiz, you can structure your answer like this:

**Question:** Explain the time and space complexity of merge sort.

**Answer:**

Merge sort is a **divide-and-conquer** sorting algorithm that recursively splits the input list into two halves, sorts them, and then merges the sorted halves back together. The merge process is efficient and takes linear time because it merges two sorted lists.

1. **Time Complexity**:
   * Merge sort recursively splits the list into halves, which takes **O(log n)** time because the list is repeatedly halved until it contains individual elements.
   * Merging two sorted lists takes **O(n)** time because we must examine each element.
   * Overall, the time complexity is **O(n log n)** for all cases (best, worst, and average), making merge sort more efficient than quadratic algorithms like bubble sort or insertion sort.
2. **Space Complexity**:
   * Merge sort uses **O(n)** extra space to store the temporary lists used during the merging process.
   * Therefore, the space complexity is **O(n)**.

**Conclusion**: Merge sort is an efficient sorting algorithm with a time complexity of **O(n log n)**, but it requires additional space due to the temporary storage used during the merge step. It is preferred over other sorting algorithms when time efficiency is more important than space efficiency.

**9. Visualization of Merge Sort:**