Merge Sort

Merge Sort

- **Overview**: A classic, recursive sorting algorithm.
- **Core Strategy: Divide and Conquer**.
- **Key Features**:
  - **Efficient**: Runs in a reliable *Theta(nlogn)* time.
  - **Stable**: Preserves the relative order of equal elements.
  - **Not In-place**: Requires *Theta(n)* additional memory.

The Core Idea: Divide and Conquer

Merge Sort follows a simple, recursive mantra:

1. **Divide**: If the list has more than one element, split it into two roughly equal halves.
2. **Conquer**: Recursively call Merge Sort on each half until you have lists of size 1 (which are inherently sorted).
3. **Combine**: Merge the two now-sorted halves back into a single, sorted list.

The real work happens in the "Combine" step.

The Heart of the Algorithm: The merge Function

The merge function is the engine of Merge Sort. It takes two already-sorted sub-arrays and combines them into one larger sorted array.

How it works:

1. Create a temporary array to hold the merged result.
2. Use two pointers, one for each sub-array, starting at the beginning.
3. Compare the elements at the pointers, copy the smaller one to the temporary array, and advance that pointer.
4. Repeat until one sub-array is empty.
5. Copy the remaining elements from the non-empty sub-array.
6. Copy the entire sorted temporary array back to the original array.

Visualizing the merge Function

The merge function's job is to combine two **pre-sorted** subarrays into one larger sorted array. Imagine you have two sorted decks of cards and you want to merge them into a single sorted pile.

The Process: A Two-Pointer Comparison

1. **Setup**: Create a temporary array to hold the result. Place a "pointer" at the beginning of each of the two sorted subarrays.
2. **Compare & Copy**: Compare the elements at both pointers. Copy the smaller of the two into the temporary array.
3. **Advance**: Move the pointer of the subarray from which you just copied the element.
4. **Repeat**: Continue the "Compare, Copy, Advance" cycle until one of the subarrays is completely empty.
5. **Cleanup**: Copy all remaining elements from the non-empty subarray into the temporary array. The result is a single, sorted array.

Visualizing the Full Algorithm

The entire algorithm follows a "**Divide and Conquer**" strategy. Visualize this as a tree structure with two main phases: splitting the array down the tree to its leaves, and then merging the results back up to the root.

Phase 1: The "Split Down" (Divide)

First, the algorithm recursively splits the main array in half. This continues until you are left with subarrays containing just one element. Think of this as traveling down the branches of the tree to the leaves. An array with a single element is, by definition, already sorted.

Phase 2: The "Merge Up" (Conquer)

Now, the merge function begins its work at the leaves. It takes pairs of sorted subarrays and merges them into a new, larger sorted subarray. This process continues up the tree, level by level, until all the pieces have been merged back into a single, completely sorted array at the root.

Implementation: The merge Helper Function

The main function orchestrates the two phases.

```
def merge(arr, left, mid, right):
    # temporary subarrays
    L = arr[left:mid + 1]
    R = arr[mid + 1:right + 1]

    i = j = 0         # indices for L and R
    k = left          # index for merged segment in
arr

    # merge while both have elements
    while i < len(L) and j < len(R):
        if L[i] <= R[j]:
            arr[k] = L[i]; i += 1
        else:
            arr[k] = R[j]; j += 1
        k += 1
```

```
    # copy any leftovers
    while i < len(L):
        arr[k] = L[i]; i += 1; k += 1
    while j < len(R):
        arr[k] = R[j]; j += 1; k += 1
```

Implementation: The Main merge_sort Function

The main function orchestrates the two phases.

```
def merge_sort(arr, left, right):
    if left >= right:
        return  # base case: 0 or 1 element

    mid = left + (right - left) // 2

    # sort halves
    merge_sort(arr, left, mid)
    merge_sort(arr, mid + 1, right)

    # merge halves (requires `merge` defined as before)
    merge(arr, left, mid, right)
```

Practical Tip: Hybrid Approach

For very small sub-arrays (e.g., *size < 16*), the overhead of recursion makes Merge Sort less efficient than a simpler sort. A common optimization is to switch to Insertion Sort for these small sub-arrays.

```
if right - left < 16:
    insertion_sort(arr, left, right)
else:
    # ... continue with merge sort logic
    pass
```

This hybrid approach is often faster in practice.

Performance Analysis

Time Complexity:

- The work at each level of the recursion tree involves merging a total of n elements, which takes *Theta(n)* time.
- The tree has *log_n* levels.
- **Total Time**: *Theta(n)*logn=Theta(nlogn)*.

Space Complexity:

- The merge function requires a temporary array whose size is proportional to the number of elements being merged.
- **Total Space**: *Theta(n)*. This is the main drawback of Merge Sort; it is not in-place.

Best, Worst, and Average Cases

Merge Sort's performance is remarkably consistent. The divide-and-conquer approach always results in the same number of comparisons and copies, regardless of the input data's initial order.

Best Case: *Theta(nlogn)*

Average Case: *Theta(nlogn)*

Worst Case: *Theta(nlogn)*

This makes it an excellent choice when a predictable, guaranteed runtime is needed.

Key Property: Stability

Merge Sort is a stable sorting algorithm.

- **Stability**: If two elements have equal keys, their relative order in the input array will be preserved in the sorted output array.

For example, if you sort a list of (name, city) pairs by city, and two people are from "Tokyo", a stable sort guarantees they will appear in the output in the same order they appeared in the input. This is a very important property for many applications.

When is Merge Sort Useful?

- **When stability** is required: It's one of the most efficient stable sorts.

- **External Sorting**: When the data is too large to fit in memory, Merge Sort is ideal because it can work on data in sequential chunks from disk.

- **Sorting Linked Lists**: It's very effective for linked lists because it doesn't rely on random access. Merging is an efficient operation on this data structure.

Summary: Merge Sort

- **Algorithm**: A recursive "Divide and Conquer" strategy that sorts sub-arrays and merges them.
- **Space**: *Theta(n)* (Not in-place).
- **Time**: *Theta(nlogn)* across all cases (Best, Average, and Worst).
- **Key Features**:
  - **Stable** sort.
  - Highly **predictable** and **reliable** performance.
  - Excellent for external sorting and linked lists.