merge_sort

May 10, 2025

1 Merge Sort

1.0.1 Merge Sort: Logic Behind the Name

Merge Sort gets its name from the two main steps involved in the algorithm:

1. **Divide**: The list is recursively split into two halves.
2. **Merge**: The two halves are merged back together in a sorted order.

1.0.2 Key Ideas Behind Merge Sort:

1. **Divide and Conquer**: Merge Sort is a **divide-and-conquer** algorithm. It works by recursively splitting the list into two halves until each sublist contains a single element (or is empty). At this point, each sublist is trivially sorted (since a single element is naturally sorted).
2. **Merging**: After dividing the list into sublists, the algorithm **merges** the sublists back together in sorted order. The merging step ensures that at each level of recursion, smaller sublists are combined into larger sorted sublists.

1.0.3 Step-by-Step Process:

1. **Divide**:
   - If the list has more than one element, it is divided into two halves.
   - This division continues recursively until each sublist has one or no elements (base case).
2. **Merge**:
   - Once the list has been divided into individual elements, the algorithm starts **merging** them back together. During the merging step, two sorted sublists are combined by comparing the elements and arranging them in order.
   - This merging process is repeated until all sublists are merged into a single sorted list.

1.0.4 Example:

Let's sort the list `[38, 27, 43, 3, 9, 82, 10]` using **Merge Sort**:

Step 1: Divide the List

- **First Split**: `[38, 27, 43, 3, 9, 82, 10]` → `[38, 27, 43]` and `[3, 9, 82, 10]`
- **Second Split**:
  - `[38, 27, 43]` → `[38]` and `[27, 43]`

  - `[3, 9, 82, 10]` → `[3, 9]` and `[82, 10]`

- **Third Split**:
  - `[27, 43]` → `[27]` and `[43]`
  - `[3, 9]` → `[3]` and `[9]`
  - `[82, 10]` → `[82]` and `[10]`

Step 2: Merge the Sublists Now, we begin to merge the individual elements back into sorted sublists:

- Merge `[27]` and `[43]` → `[27, 43]`
- Merge `[3]` and `[9]` → `[3, 9]`
- Merge `[82]` and `[10]` → `[10, 82]`

Next, merge larger sublists:

- Merge `[38]` and `[27, 43]` → `[27, 38, 43]`
- Merge `[3, 9]` and `[10, 82]` → `[3, 9, 10, 82]`

Finally, merge the two large sorted sublists:

- Merge `[27, 38, 43]` and `[3, 9, 10, 82]` → `[3, 9, 10, 27, 38, 43, 82]`

Final Sorted List: `[3, 9, 10, 27, 38, 43, 82]`

1.0.5 Logic Behind the Name "Merge Sort":

- **Merge**: The core idea is the **merging** of sorted sublists back into one larger sorted list.
- **Sort**: The list is sorted by recursively breaking it down (dividing) and then merging it in the correct order.

1.0.6 Key Features of Merge Sort:

- **Efficiency**: Merge Sort has a time complexity of **O(n log n)**, which is better than algorithms like Bubble Sort and Selection Sort, which have a time complexity of $O(n^2)$.
- **Stable Sort**: Merge Sort is a **stable sort**, meaning that if two elements have the same value, they will retain their original relative order after sorting.
- **External Sorting**: Merge Sort is particularly useful for sorting large data sets that do not fit entirely into memory (external sorting), since it works well with data stored in external storage like hard drives.

1.1 Helper Function for merge sort

[4]:
```
# takes two sorted list;

# Then merge it; the placed in order
```

```
def merge(list1, list2):
    combined = []
    # you can't use for loops because you  dont konw how many are there in each␣
 ↪list

    i = 0
    j = 0

    # to point into first element of both list

    # we will run list as long as there is items in the list;

    # either list is empty, we will break out of loop
    while i<len(list1) and j<len(list2):
        if list1[i]<list2[j]:
            combined.append(list1[i])
            i +=1
        else:
            combined.append(list2[j])
            j+=1

    # if item still remaining in the list1
    while i<len(list1):
        combined.append(list1[i])
        i +=1


    while j<len(list2):
        combined.append(list2[j])
        j +=1

    return combined

# %%%%%%%%%%%



list1 = [1,2,7,8]
list2 = [3,4,5,6]

merge(list1, list2)
```

[4]: [1, 2, 3, 4, 5, 6, 7, 8]

1.1.1 Merge Process Breakdown

The lists list1 and list2 aren't actually getting reduced by the merge function. They're only being traversed, and elements are compared by their indices, not physically removed from the lists. So, to better reflect what's happening:

Step	Action	Explanation	Combined List	List1	List2
Step 1: Main Loop	Compare `list1[0]` (1) with `list2[0]` (3)	Since 1 < 3, append 1 to `combined`.	`[1]`	`[2, 7, 8]`	`[3, 4, 5, 6]`
	Compare `list1[1]` (2) with `list2[0]` (3)	Since 2 < 3, append 2 to `combined`.	`[1, 2]`	`[7, 8]`	`[3, 4, 5, 6]`
	Compare `list1[2]` (7) with `list2[0]` (3)	Since 3 < 7, append 3 to `combined`.	`[1, 2, 3]`	`[7, 8]`	`[4, 5, 6]`
	Compare `list1[2]` (7) with `list2[1]` (4)	Since 4 < 7, append 4 to `combined`.	`[1, 2, 3, 4]`	`[7, 8]`	`[5, 6]`
	Compare `list1[2]` (7) with `list2[2]` (5)	Since 5 < 7, append 5 to `combined`.	`[1, 2, 3, 4, 5]`	`[7, 8]`	`[6]`
	Compare `list1[2]` (7) with `list2[3]` (6)	Since 6 < 7, append 6 to `combined`.	`[1, 2, 3, 4, 5, 6]`	`[7, 8]`	`[]`
Step 2: Handling Remaining Elements	`list1` still has elements 7 and 8, so they are appended to `combined`.	The remaining elements of `list1` (7 and 8) are already sorted, so they are appended directly.	`[1, 2, 3, 4, 5, 6, 7, 8]`	`[]`	`[]`
Final Merged List	`[1, 2, 3, 4, 5, 6, 7, 8]`	The final merged and sorted list after all comparisons and append operations are complete.	`[1, 2, 3, 4, 5, 6, 7, 8]`	`[]`	`[]`

1.1.2 Merge Process Breakdown

Step	Action	Combined List	List1	List2	Pointer in List1 (i)	Pointer in List2 (j)
Initial State	-	`[1, 2, 7, 8]`	1, 2, 7, 8	3, 4, 5, 6| | | | **Step 1: Compare 1 & 3** | Comparelist1[0](1) withlist2[0](3). Since 1 < 3, append 1 tocombined. |[1]|1,2, 7, 8	3, 4, 5, 6	0
Step 2: Compare 2 & 3	Compare `list1[1]` (2) with `list2[0]` (3). Since 2 < 3, append 2 to `combined`.	`[1, 2]`	1,2, 7, 8| 3,4, 5, 6	1	0	

Step	Action	Combined List	List1	List2	Pointer in List1 (i)	Pointer in List2 (j)
Step 3: Compare 7 & 3	Compare `list1[2]` (7) with `list2[0]` (3). Since 3 < 7, append 3 to `combined`.	`[1, 2, 3]`	1,2,7, 8	3 4, 5, 6|2|0| | **Step 4: Compare 7 & 4** | Compare list1[2] (7) with list2[1] (4). Since 4 < 7, append 4 to combined. |[1, 2, 3, 4]| 1,2,7, 8| 3,4, 5, 6	2	1

Step	Action	Combined List	List1	List2	Pointer in List1 (i)	Pointer in List2 (j)
Step 5: Compare 7 & 5	Compare `list1[2]` (7) with `list2[2]` (5). Since 5 < 7, append 5 to `combined`.	`[1, 2, 3, 4, 5]`	1,2,7, 8	3,4,5,6 6|2|2| | **Step 6: Compare 7 & 6** | Compare list1[2](7) with list2[3](6). Since 6 < 7, append 6 to combined. |[1, 2, 3, 4, 5, 6]| 1,2,7, 8 | 3,4,5,6 |2|3| | **Step 7: Handle Remaining in List1** |list2 is empty, so append remaining elements from list1. |[1, 2, 3, 4, 5, 6, 7, 8]| 1,2,7, 8	`[ Empty ; Means all are appended ]`	2

Step	Action	Combined List	List1	List2	Pointer in List1 (i)	Pointer in List2 (j)
-	-	-	-	-	-	
Meaning now the pointer at list one is at 7: 7, 8	———- 7 and 8 are needed to append into the combined list		While loop runs here	While loop fails here		
Final Merged List	-	`[1, 2, 3, 4, 5, 6, 7, 8]`	`[]`	`[]`	`4`	`4`

1.1.3 Explanation of the Comment:

Comment: `# you can't use for loops because you don't know how many are there in each list`

- **Why it's necessary**: This comment highlights the reason why we can't use a `for` loop to iterate over the lists. In the `merge` function, we don't know in advance how many elements each list contains. A `for` loop usually iterates over a fixed range (e.g., `for i in range(len(list))`), but in this case, we need to dynamically handle lists of different lengths. Since we don't know the number of iterations required, a `while` loop is more appropriate because it continues until certain conditions (list length) are met.

  A `while` loop gives us the flexibility to process the lists until either list is exhausted.

Comment: `# to point into first element of both list`

- **Why it's necessary**: The variables `i = 0` and `j = 0` are used to **point to the first elements** of `list1` and `list2`, respectively. In merge sort, we start comparing the elements from the beginning of both lists, so initializing `i` and `j` to 0 allows us to keep track of our position as we iterate through each list. These variables act as **indexes** that track the current element of each list being processed.

Comment: `# we will run the list as long as there are items in the list`

- **Why it's necessary**: This comment refers to the **main while loop**:

  "'python while i < len(list1) and j < len(list2):