Merge Sort

bluesky: @leonlonsdale.dev
discord: https://discord.gg/dhrdFh98UA

How it works

- A divide and conquer recursive sorting algorithm.
- It recursively divides the list in half until the length is 1
- All lists of length 1 are considered sorted.
- Two sorted lists are passed to a merging function which combines the lists into a single sorted list and returns it.
- This happens recursively until the array is reconstructed in order.

Pseudocode

`merge()` Function Pseudocode

```
1. Create an empty list called results to store the merged elements.
2. Set two pointers i and j to 0 (these will be used to traverse the left and right sub-lists).
3. While both pointers i and j are within bounds of their respective sub-lists:
    • If the element at left[i] is smaller or equal to right[j]:
        • Add left[i] to results.
        • Increment i by 1.
    • Otherwise:
        • Add right[j] to results.
        • Increment j by 1.
4. Once one sub-list is exhausted (i.e., either i or j goes out of bounds):
    • Add the remaining elements of the left sub-list (from i onwards) to results.
    • Add the remaining elements of the right sub-list (from j onwards) to results.
5. Return the results list, which contains the merged and sorted elements.
```

`merge_sort()` Function Pseudocode

```
1. If the list has fewer than two elements (i.e., it is either empty or contains one element):
    • Return the list (it is already sorted).
2. Otherwise:
    • Find the middle index of the list.
    • Divide the list into two halves: left and right.
3. Recursively apply merge_sort() to both the left and right sub-lists.
4. Once both sub-lists are sorted:
    • Call the merge() function to merge the two sorted sub-lists.
5. Return the merged and sorted list.
```

Example Code

```python
def merge(left,right):
    results = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            results.append(left[i])
            i += 1
        else:
            results.append(right[j])
            j += 1

    results.extend(left[i:])
    results.extend(right[j:])

    return results

def merge_sort(arr):

    if len(arr) < 2:
        return arr

    mid = len(arr) // 2

    return merge(merge_sort(arr[:mid]), merge_sort[arr[mid:]])
```

Complexity

Time Complexity

The time complexity of merge sort is $O(n\log(n))$.

- **Divide Step:** The list is recursively split in half. This happens at each level, creating a binary tree structure with a depth of $\log(n)$.
- **Merge Step:** At each level of the recursion tree, merging the sorted subarrays requires processing every element in the array. When you sum the work done by all merge operations at a given level, the total merging work is, $O(n)$. This happens $O(n\log(n))$ times.

Space Complexity

The space complexity of merge sort is $O(n)$, and here's why:

- **Auxiliary Space for Merging:**
  During the merge step, merge sort requires extra space to store the left and right subarrays. These subarrays are created as temporary lists to store the split parts of the original array. While merging, new lists are formed by combining the elements from these temporary subarrays.
  Since the merge step combines all elements in the array at each level, the total space used at any point is proportional to the size of the original array.
- **Recursive Call Stack:**
  Merge sort is a recursive algorithm, meaning it uses a call stack. In the worst case, the depth of the recursion tree is $\log(n)$ (since the array is split in half each time). However, this doesn't increase the space complexity because the recursion only uses space proportional to the number of subarrays being merged at each level, which is still $O(n)$ overall.

Therefore, even though the recursion depth is $\log(n)$, the main factor contributing to space complexity is the additional space used for temporary subarrays during the merging process. This results in a total space complexity of $O(n)$.