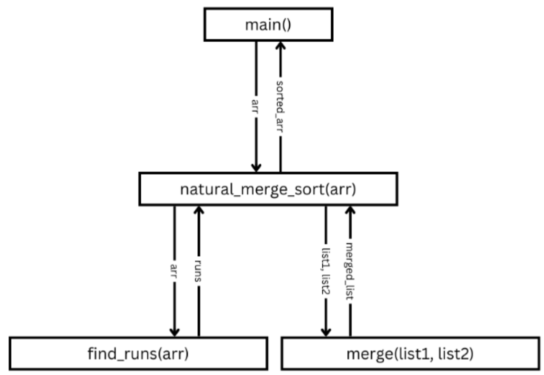
1. Modularization Metrics



natural_merge_sort()

Cohesion: High - This function focuses solely on sorting the list by repeatedly merging naturally ordered runs.

Coupling: Moderate - This function depends on find_runs() and merge() to perform its operations. Although there is dependency, it does not modify or interfere with the internal workings of the other functions.

find_runs()

Cohesion: High - This function's sole responsibility is to identify and separate the list into runs, so it has high cohesion.

Coupling: Low - This function is self-contained and does not rely on any other function to achieve its task.

merge()

Cohesion: High - The function only merges two sorted lists, so its purpose is clear and cohesion is high.

Coupling: Low - This function is self-contained and does not rely on any other function to achieve its task.

2. Algorithmic Metrics

PSEUDOCODE:

```
FUNCTION natural_merge_sort(arr):
    sorted <- False
    WHILE !sorted
        runs <- []
        start <- 0
        WHILE start < len(arr)
            end <- start + 1
            WHILE end < len(arr) AND arr[end] >= arr[end - 1]
                end <- end + 1
            runs.append(arr[start:end])
            start <- end
        IF len(runs) == 1
            RETURN runs[0]
        ELSE
            arr <- find_runs(runs)

FUNCTION find_runs(runs):
    WHILE len(runs) > 1
        merged_runs <- []
        FOR i IN range(0, len(runs), 2)
            IF i + 1 < len(runs)
                merged_runs.append(merge(runs[i], runs[i + 1]))
            ELSE
                merged_runs.append(runs[i])
        runs <- merged_runs
    RETURN runs[0]

FUNCTION merge(list1, list2):
```

```
merged <- []
i, j <- 0, 0
WHILE i < len(list1) AND j < len(list2)
    IF list1[i] <= list2[j]:
        merged.append(list1[i])
        i <- i + 1
    ELSE:
        merged.append(list2[j])
        j <- j + 1
merged.extend(list1[i:])
merged.extend(list2[j:])
RETURN merged
```

ALGORITHMIC EFFICIENCY:

natural_merge_sort()

- In the worst case, the entire list may be treated as individual elements requiring a typical O(n log n) merge complexity. However, with naturally ordered runs, efficiency improves when the input is partially sorted.

find_runs()

- Runs in O(n)O(n)O(n) because it only scans through the list once, grouping elements into natural runs.

merge()

- Each merge operation between two sorted sublists of sizes a and b takes O(a + b) time, as it only makes a single pass through each sublist. In total, the merging step has O(n log n) complexity.

OVERALL, THE WORST CASE COMPLEXITY IS O (N LOG N)!!!

3. Test
    a. Already Sorted List
        i. Input: [1, 2, 3, 4, 5, 6]
        ii. Expected Output: [1, 2, 3, 4, 5, 6]
    b. Reversed Sorted List
        i. Input: [6, 5, 4, 3, 2, 1]
        ii. Expected Output: [1, 2, 3, 4, 5, 6]

c. Random Order List
    i. Input: [3, 1, 4, 1, 5, 9]
    ii. Expected Output: [1, 1, 3, 4, 5, 9]
d. List with Duplicate Elements
    i. Input: [2, 3, 3, 1, 5, 2]
    ii. Expected Output: [1, 2, 2, 3, 3, 5]
e. Single Element List
    i. Input: [7]
    ii. Expected Output: [7]
f. Empty List
    i. Input: []
    ii. Expected Output: []

AUTOMATION DRIVER:

```
def test_natural_merge_sort():

    test_cases = [

        ([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6]),

        ([6, 5, 4, 3, 2, 1], [1, 2, 3, 4, 5, 6]),

        ([3, 1, 4, 1, 5, 9], [1, 1, 3, 4, 5, 9]),

        ([2, 3, 3, 1, 5, 2], [1, 2, 2, 3, 3, 5]),

        ([7], [7]),

        ([], [])

    ]


    for i, (input_data, expected) in enumerate(test_cases):

        result = natural_merge_sort(input_data)

        assert result == expected, f"Test case {i+1} failed: expected
{expected}, got {result}"


    print("All test cases passed!")
```