Tc and Sc Understanding

Great question! Let me break down the time and space complexity analysis step by step.

Time Complexity Analysis

Let's analyze each part of the code:

1. Building the array (Lines 4-7)

```
arr = []
for i in range(N):
    val = [S[i], F[i], i+1]
    arr.append(val)
```

- **Operation**: Loop N times, each iteration does O(1) work
- **Time**: O(N)

2. Sorting the array (Line 12)

```
arr.sort(key = lambda x: x[1])
```

- **Operation**: Sorting N elements
- **Time**: O(N log N)

3. Finding maximum meetings (Lines 14-26)

```
for i in range(1, N):
    start_time, end_time, idx = arr[i]

    if prev_meeting_end_time >= start_time:
        continue

```

```
    prev_meeting_end_time = end_time
    count += 1
    indexes.append(idx)
```

- **Operation**: Single loop through N elements, each iteration is O(1)
- **Time**: O(N)

4. Sorting the result (Line 28)

```
return sorted(indexes)
```

- **Operation**: Sorting the selected meeting indices
- **Key insight**: `indexes` can have at most N elements (worst case: all meetings are selected)
- **Time**: O(N log N) in worst case

Total Time Complexity

```
O(N) + O(N log N) + O(N) + O(N log N) = O(N log N)
```

The sorting operations dominate, so **Time Complexity = O(N log N)**

Space Complexity Analysis

1. The `arr` array

```
arr = []
for i in range(N):
    val = [S[i], F[i], i+1]
    arr.append(val)
```

- **Space**: Stores N meetings, each with 3 elements
- **Memory**: O(N)

2. The `indexes` array

```
indexes = [arr[0][2]]
# ... later additions in the loop
```

- **Space**: Stores selected meeting indices
- **Worst case**: All meetings are selected → O(N)
- **Best case**: Only 1 meeting selected → O(1)
- **Average/Worst case**: O(N)

3. Other variables

```
prev_meeting_end_time = arr[0][1]
count = 1
start_time, end_time, idx = arr[i]  # loop variables
```

- **Space**: O(1) for all auxiliary variables

4. Sorting space overhead

```
arr.sort(key = lambda x: x[1])
sorted(indexes)
```

- **Space**: Most sorting algorithms use O(log N) extra space for recursion stack
- **Memory**: O(log N)

Total Space Complexity

```
O(N) + O(N) + O(1) + O(log N) = O(N)
```

Space Complexity = O(N)

Summary

Aspect	Complexity	Explanation
Time	**O(N log N)**	Dominated by sorting operations
Space	**O(N)**	Dominated by storing meeting data and results

Visual Breakdown

```
def maxMeetings(self, N, S, F):
    # O(N) time, O(N) space
    arr = []
    for i in range(N):
        arr.append([S[i], F[i], i+1])

    # O(N log N) time, O(log N) space
    arr.sort(key = lambda x: x[1])

    # O(N) time, O(N) space (worst case)
    prev_meeting_end_time = arr[0][1]
    count = 1
    indexes = [arr[0][2]]

    for i in range(1, N):
        start_time, end_time, idx = arr[i]
        if prev_meeting_end_time >= start_time:
            continue
        prev_meeting_end_time = end_time
        count += 1
        indexes.append(idx)

    # O(N log N) time, O(log N) space
    return sorted(indexes)
```

Key Insights

1. **Why O(N log N) time?** The algorithm must sort meetings by end time to apply the greedy strategy effectively.
2. **Why O(N) space?** We need to store all meeting information and potentially all meeting indices in the result.
3. **Can we optimize?** The time complexity is optimal for this problem since we need sorting. Space could theoretically be optimized by sorting indices in-place, but it would complicate the code significantly.
4. **Greedy vs DP**: This greedy approach is much more efficient than a DP solution, which would typically be O(2^N) time for this problem.