Day 25 / 100 :

Topic - Greedy, sorting

1 Problem statement: Non overlapping intervals (Medium)

Given an array of intervals intervals where intervals[i] = [starti, endi], return the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

Example 1:
Input: intervals = [[1,2],[2,3],[3,4],[1,3]]
Output: 1
Explanation: [1,3] can be removed and the rest of the intervals are non-overlapping.

Example 2:
Input: intervals = [[1,2],[1,2],[1,2]]
Output: 2
Explanation: You need to remove two [1,2] to make the rest of the intervals non-overlapping.

Solutions :
Approach 1 - Greedy Approach

Intuition:

- Minimum number of intervals to remove .
- Which is nothing but maximum number of intervals we can should keep.
- Then it comes under Maximum Meeting we can attend.

Explanation:
Imagine we have a set of meetings, where each meeting is represented by an interval [start_time, end_time]. The goal is to find the maximum number of non-overlapping meetings we can attend.

1. **Sorting by end times (cmp function):**

The function first sorts the intervals based on their end times in ascending order using the custom comparator cmp. This sorting is crucial because it allows us to prioritize intervals that finish early, giving us more opportunities to accommodate additional meetings later on.

2. **Initializing variables:**

The function initializes two variables, prev and count. The prev variable is used to keep track of the index of the last processed interval, and count is used to store the number of

non-overlapping meetings found so far. We start count with 1 because the first interval is considered non-overlapping with itself.

3. **Greedy approach:**

The function uses a greedy approach to find the maximum number of non-overlapping meetings. It iterates through the sorted intervals starting from the second interval (index 1) because we've already counted the first interval as non-overlapping. For each interval at index i, it checks if the start time of the current interval (intervals[i][0]) is greater than or equal to the end time of the previous interval (intervals[prev][1]). If this condition is true, it means the current interval does not overlap with the previous one, and we can safely attend this meeting. In that case, we update prev to the current index i and increment count to reflect that we have attended one more meeting.

4. **Return result:**

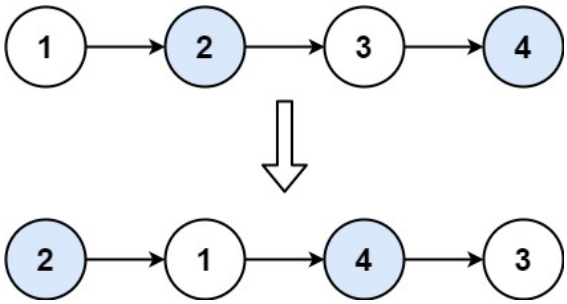
Finally, the function returns the number of intervals that need to be removed to make the remaining intervals non-overlapping. Since we want to maximize the number of meetings we can attend, this value is calculated as n - count, where n is the total number of intervals.

```cpp
class Solution {
public:
    static bool cmp(vector<int>& a, vector<int>& b){
        return a[1] < b[1];
    }

    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        int n = intervals.size();
        sort(intervals.begin(), intervals.end(), cmp);

        int prev = 0;
        int count = 1;

        for(int i = 1; i < n; i++){
            if(intervals[i][0] >= intervals[prev][1]){
                prev = i;
                count++;
            }
        }
        return n - count;
    }
};
```

2 Problem statement: Swap nodes in pair (Medium)

Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed.)

Example 1:



Input: head = [1,2,3,4]
Output: [2,1,4,3]

Example 2:
Input: head = []
Output: []

Example 3:
Input: head = [1]
Output: [1]

Solutions :
Approach 1 - Traverse and Swap

Intuition:
Traverse the list and swap pairs of nodes one by one.

Approach:
The node "ans" is to point to the head of the original list. It then uses a "curr" node to traverse the list and swap pairs of nodes. The loop continues as long as there are at least two more nodes to swap.

Inside the loop, the solution uses two temporary nodes, "t1" and "t2", to hold the first and second nodes of the pair. Then, it updates the pointers to swap the nodes, and moves "curr" two nodes ahead. At the end, it returns the modified list starting from the next node of the "ans" node.

Complexity:
Time complexity:O(n)
Space complexity:O(1)

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
         if (head == NULL || head->next == NULL) {
        return head;
    }
    struct ListNode* ans = (struct ListNode*) malloc(sizeof(struct
ListNode));
    ans->next = head;
    struct ListNode* curr = ans;
    while (curr->next != NULL && curr->next->next != NULL) {
        struct ListNode* t1 = curr->next;
        struct ListNode* t2 = curr->next->next;
        curr->next = t2;
        t1->next = t2->next;
        t2->next = t1;
        curr = curr->next->next;
    }
    return ans->next;
    }
};
```