Activity selection :-

Activity Selection

Difficulty: **Medium** Accuracy: **36.21%** Submissions: **150K+** Points: **4** Average Time: **20m**

You are given a set of activities, each with a **start time** and a **finish time**, represented by the arrays **start[]** and **finish[]**, respectively. A single person can perform only **one activity** at a time, meaning **no two activities can overlap**. Your task is to determine the **maximum number of activities** that a person can complete in a day.

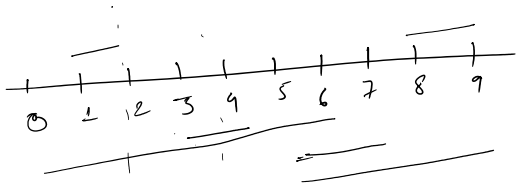
Examples:

Input: start[] = [1, 3, 0, 5, 8, 5], finish[] = [2, 4, 6, 7, 9, 9]
Output: 4
Explanation: A person can perform at most four activities. The maximum set of activities that can be executed is {0, 1, 3, 4}

Input: start[] = [10, 12, 20], finish[] = [20, 25, 30]
Output: 1
Explanation: A person can perform at most one activity.

Input: start[] = [1, 3, 2, 5], finish[] = [2, 4, 3, 6]
Output: 3
Explanation: A person can perform activities 0, 1 and 3.

Constraints:
$1 \leq start.size() = finish.size() \leq 2*10^5$
$1 \leq start[i] \leq finish[i] \leq 10^9$



Intuition :-

1. sort by finish time of activities.
2. choose the activity with earliest finish time & that is non-overlapping.

Using Priority Queue :-

```cpp
class Solution {
  public:
    int activitySelection(vector<int> &start, vector<int> &finish) {
        // code here
        int n=start.size();
        priority_queue<pair<int, int>, vector<pair<int, int>>, function<bool(pair<int, int>, pair<int, int>)>>
        q([](const pair<int, int> &a, const pair<int, int> &b){
            if(a.second==b.second)return a.first>b.first;
            return a.second>b.second;
        });
        for(int i=0;i<n;i++){
            q.push(make_pair(start[i], finish[i]));
        }
        int ans=0, last=0;
        while(!q.empty()){
            pair<int, int> p=q.top();
            q.pop();
            if(p.first<=last)continue;
            ans++;
            last=p.second;
        }
        return ans;
    }
};
```

T.C. O(n log n)
S.C. O(n)

By Sorting the array :-

```
class Solution {
public:
    int activitySelection(vector<int>& start, vector<int>& finish) {
        int n = start.size();
        vector<pair<int, int>> activities(n);
```

```
for(int i=0; i<n; ++i){
    activities[i] = {start[i], finish[i]};
}
sort(activities.begin(), activities.end(), [](const pair<int,int>& a,
const pair<int,int>& b){ return a.second < b.second; });
int ans = 0;
int last = -1;
```

//Greedy approach: select activities whose start time is greater than
// or equal to last finish time.

```
for(int i=0; i<n; ++i){
    if(activities[i].first > last){
        ++ans;
        last = activities[i].second; } }
return ans;
}
};
```

T.C: O(n log n)

S.C: O(n)

[Expected Approach -1] - Using Sorting - O(n*log n) Time & O(n) Space: -

```
int activitySelection(vector<int>& start, vector<int>& finish){
    int ans = 0;
    vector<vector<int>> arr;
    for(int i=0; i<start.size(); i++){
        arr.push_back({finish[i], start[i]}); }
    sort(arr.begin(), arr.end());
    int finish time = -1;
    for(int i=0; i<arr.size(); i++){
        vector<int> activity = arr[i];
        if(activity[1] > finishTime){
            finishTime = activity[0];
```

```
            ans++;
        }
    }
    return ans;
}
```

[Expected Approach 2] - Using Priority Queue - O(n log n) Time & O(n) Space :-

```
int activitySelection(vector<int>& start, vector<int>& finish){
    int ans = 0;
    priority_queue<pair<int,int>, vector<pair<int,int>>,
                   greater<pair<int,int>>> p;

    for(int i=0; i<start.size(); i++){
        p.push(make_pair(finish[i], start[i]));
    }

    while(!p.empty()){
        pair<int,int> activity = p.top();
        p.pop();
        if(activity.second > finishTime){
            finishTime = activity.first;
            ans++;
        }
    }

    return ans;
}
```