Problem 4: Guards for the farmer [11 marks]

A farmer who grows watermelons wants to put guards on her warehouse in the time interval $[0, M]$. There are n guards and the i-th guard can work during time interval $[s_i, f_i]$. The farmer wants to hire a minimum number of guards, such that in any time point $t \in [0, M]$ there is at least one guard whose time interval covers t, i.e., $t \in [s_i, f_i]$. For example, for $M = 15$ and $n = 6$, the guards can work at time intervals $[0, 3], [5, 9], [9, 13], [0, 7], [10, 15]$, and $[7, 12]$, there is a solution of size 3 by hiring 3 guards who work at times: $[0, 7], [7, 12]$, and $[10, 15]$.

(a) (3 marks) under what conditions on the time intervals the farmer can not hire a minimum number of guards, such that in any time point $t \in [0, M]$ there is at least one guard whose time interval covers t, i.e., $t \in [s_i, f_i]$?

(b) (8 marks) Assume that it is possible for the farmer to hire such a minimum number of guards. Help the farmer design a greedy algorithm to hire, suggest an implementation of complexity $O(n \log n)$, and prove its correctness.

Solution. **Please write down your solution to Problem 4 here.**

(a) **part a** When the interval that the guards can work at are disjoint are no guards work at an interval that could cover another one's work.

(b) **part b**

```
optimalHire([[s1,f1],···,[sn,fn]], s, f)
{
    //sort work interval in ascending order of left ends,
    // for intervals with the same left ends,
    //sort descendingly with respect to right ends
    SORT([[s1,f1],···,[sn,fn]]);

    S← ∅, t← s;

    for i in 1 to n{

    \\if the point that we already covered have exceeded end point f
    \\end for loop, we are done
        if(t >= f){
            break;
        }

    \\if the skip indicator is not 0, we do not have to
    \\look at current element, it is ought to
    \\be skipped, since we have made the greedy choice
```

```
            if(skip > 0){
                skip -= 1;
                continue;
            }


            else{
                container = []; \\we prepare a contianer to store
                                \\all the element that have s(i)<=t
                skip = 0;

                \\traverse, while current element
                \\overlaps with the farthest right we have already reached

                while intervals[i][0] <= end{
                    add A[i] into container;
                    skip += 1; \\ skip one element in the future
                    i +=1;
                }

                \\we find the interval with maximum finishing time
                \\add it to the result set
                \\update our covered right bound t to be finish time
                \\ of our choice

                choice ← max(finish(i)) for i in container;
                S ← S ∪ {choice};
                end ← choice[1];

                \\we need to minus 1 on skip, since we in fact already

                passed current element in the for loop
                skip← skip-1;
            }
        }
        return S;
    }
```

we start from the left end, we always pick the interval with the largest right end while covering the current starting point. We first sort the input interval array in ascending order according to left end value, for identical left end value pair, we sort in descending order of right end value. The sorting operation take $O(nlogn)$ time, and the second while loop take $O(n)$ time.
further explanation: in the code there is a **skip** flag, this is to indicate if current interval have been looked at and was decided not to choose, since it give ways to our greedy choice – the interval with largest right end while still covering the starting point(0 when no interval have been picked or the right end of the previously picked interval).
If the skip flag is 0, that means we have to start looking at next possible solution. We introduce a **container** array to store all the succeeding interval, whose left end overlaps with our current starting point, we know that next greedy choice has to lie in there, or there would be no solution if a point on the interval is missing.
Then, we determine the interval with the maximum right end inside the container,add this to the result, and set our new starting point to be its finishing time.

Proof. If we take the interval I_{max} which contains the starting point s and has the largest end point, we can change the first interval in any optimal solution to be I_{max}, and it is still an optimal solution. This step could be repeated for the rest interval $I - I_{max}$ and so on, Therefore, our solution is optimal. □