Problem 1

First use merge sort to sort the set of intervals by s_i $(1 \leq i \leq n)$ in increment order of s_i. The merge sort works like the following.

1. Divide the set into halves at each level until the size of each subset is one.
2. Begin from the last level and move up by ordering all the intervals in each subset by ascending order of their s_i values.

This sort algorithm is $O(n \log n)$ because at each level, all the n intervals are checked, and there are $\log_2 n$ number of levels. After we obtain a sorted set S', we loop through every interval in S' starting from the first one, I'_1. Each interval is recorded by (s'_i, f'_i), $1 \leq i \leq n$. We find the intercept between I'_1 and I'_2. If $f'_1 < s'_2$, then I'_1 and I'_2 are not intercepted, and the intervals beyond I'_2 cannot intercept with I'_1. So in this case, we will assign a point to cover I'_1 by increasing the point counter by one (the counter begins with zero). If $f'_1 \geq s'_2$, then we record the interception between the two by $(s'_2, min(f'_1, f'_2))$ and name it prevIn. Meanwhile, we record the currently intercepted intervals (name it inceptIn). We proceed to the next interval I'_3 and check if I'_3 and prevIn are intercepted by following the same procedure above. If the two are intercepted, the prevIn will be updated according to their intersection and the current interval will be added to the inceptIn. We check the rest of the intervals by using the same process. If the size of prevIn becomes 1, assign a point to cover all the intervals in inceptIn, start from the next interval and assign prevIn to be its coverage range.

Overall, my algorithm is $O(n \log n)$ because the sorting takes $O(n \log n)$ and after the sorting, I have a for-loop that takes $O(n)$. The operations inside the loop are all value checks that are constant time. So the overall run-time is $n \log n + n \in O(n \log n)$.

Finally, I want to justify that my algorithm finds the minimum size cover. Given two adjacent intervals in the sorted list, I_i and I_{i+1}, there are several possible cases.

1. If prevIn $= I_i$ (which means I_i is a new start) and I_i intercepts with I_{i+1} and prevIn is updated accordingly. Then my algorithm will precede to find if the next interval intercepts with prevIn. If not, then there will be two points used to cover I_i I_{i+1}, and the next interval. Two points will be the minimum possible points used because the three intervals don't have a common interception. One point cannot cover all three intervals. If the next interval does intercept with prevIn, then prevIn will be updated again and one points will be used to cover the three intervals, which is also the minimum possible points used.
2. If prevIn $= I_i$ and I_i doesn not intercept with I_{i+1}. Then then algorithm will use two points to cover the two intervals. Two

is a minimum number because the set is in sorted order, if I_{i+1} doesn't intercept with I_i, then intervals beyond I_{i+1} can't possibly intercept with I_i. So one point must be used to cover I_i, and another one to cover I_{i+1}.

3. If I_i intercepts with prevIn and I_{i+1} doesn't intercept with the updated prevIn. Then two points are used to cover the two intervals. If I_{i+1} intercepts with I_i but doesn't intercept with prevIn, then two points are needed to cover the intervals in inceptIn and I_{i+1}, no matter where we put I_i (so it doesn't matter if I_i is grouped with inceptIn or with I_{i+1}). If I_{i+1} doesn't intercept with I_i, then at least two points are needed to cover I_i and I_{i+1}.
4. If I_i intercepts with prevIn and I_{i+1} also intercepts with the updated prevIn. Then one point is used to cover the two intervals, which is the minimum possible points used.

In all, my algorithm uses the minimum possible points to cover the set of intervals.