Yiduo Ke yk467

1) (have this here to have the automatic
2) numbering format in Word)
3) Before we design the algorithm, some terminology:

s(i) means the time shift i starts

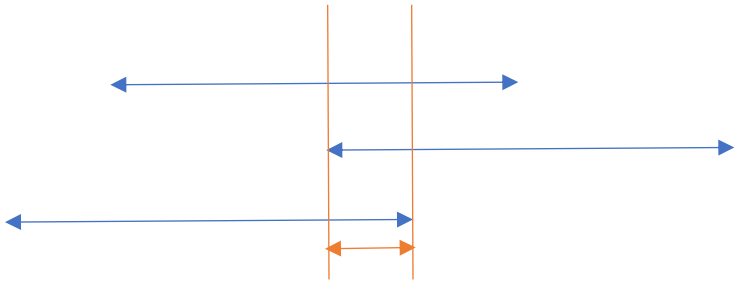
f(i) means the time shift i ends

Now the algorithm:

```
1  initialize R to be the set of all shifts;
2  initialize T to be empty;
3  while (R is not yet empty):
4      choose a shift i∈R that ends the earliest;
5      add f(i) to T;
6      delete all shifts from R that overlap with shift i;
7  endwhile
8  return set T as the boss's visit times
```

Proof that this returns an optimal solution

First, realize the obvious fact that the visit interval that works for a group of overlapping shifts starts at the latest start time of the group and ends at the earliest time of the group, like this:



Range of valid visit times

Second, realize that this algorithm is almost identical to the greedy algorithm for finding the biggest set of non-overlapping intervals we learned in class (interval scheduling). The only difference between these two is line 5 in the pseudocode shown above, where the interval scheduling problem has

add i to T;

instead.

Since R is empty by the end of the algorithm, that means the interval scheduling algorithm got rid of all intervals that overlap with the optimal set along the way. Since we established the working visit interval for a group of overlapping shifts, choosing a visit time at the earliest end time for each shift returned by the interval scheduling algorithm is an optimal visit time because it falls within the interval (the interval is between the latest start time and earliest end time, inclusive).

Also realize that one visit time cannot see 2 (or more, for that matter) non-overlapping shifts, so the cardinality of T cannot be any smaller than the cardinality of the set returned by the interval scheduling problem. **This algorithm thus returns the optimal set of the boss's visit times.**

Runtime Analysis

We can make our algorithm run in time O(n log n) as follows. We begin by sorting the n shifts in order of finishing time and labeling them in this order; that is, we will assume that $f(i) \leq f(j)$ when $i < j$. This takes time O(n log n). In an additional O(n) time, we construct an array S[1 … n] with the property that S[i] contains the value s(i).

We now select visit times by processing shifts in order of increasing f(i). We always select the first shift; we then iterate through the intervals in order until reaching the first interval j for which $s(j) \geq f(1)$; we then select this one's end time as well. More generally, if the most recent shift end time we've selected is f(i), we continue iterating through subsequent intervals until we reach the first j for which $s(j) \geq f(i)$. In this way, we implement the greedy algorithm analyzed above in one pass through the intervals, spending constant time per interval. Thus, this part of the algorithm takes time O(n). Taking it all together, it's O(n log n + 2n), which is O(n log n).