Problem 1:

a)

Because we want only the jobs that can't fit in the schedule to be the minimum penalties, we will want to construct our schedule so that we can complete jobs with the highest penalties first. To start, we would want to sort our jobs in descending order of penalties, then go through each job and fit it into a "free space" that is as close to the deadline as possible. If there are no free spaces left that would meet the deadline, we discard the job and move on. This will provide a schedule with the maximum possible penalty jobs being completed, leaving a minimum penalty leftover.

```
function schedule (arr, times):
   sort arr in descending order

   create arr of size times to hold free slots
   for each job:
      for each slot <= deadline:
         if slot is free:
            set slot to taken
            add job to result

   return result
```

b)

The time complexity would be $O(nlogn)$ time depending on the sorting algorithm used, since looping through each job would take n times, and checking slots would be logn. If we used optimal sorting, then our total complexity would be O(nlogn)

Problem 2:

This approach is a greedy algorithm because we are selecting the most optimal local solution at each step of the algorithm, which will result in the globally optimal solution. We can prove this with the following:

Let S be a set of activities $S = \{a_1, a_2, a_3, \ldots, a_n\}$, where $a_i = [s_i, f_i)$, and we want to find an optimal solution by selecting the last activity to start that is compatible with all previously selected activities. Let's now create $S_2 = \{a_{1b}, a_{2b}, \ldots, a_{nb}\}$, where $a_{ib} = [f_i, s_{ib})$. This means a_{ib} is the reverse of a_i. A subset of $\{a_1, a_2, \ldots, a_n\} \subset S$ is mutually compatible if and only if the corresponding subset $\{a_{1b}, a_{2b}, \ldots, a_{nb}\} \subset S_2$ is also mutually compatible. Thus an optimal solution for S maps directly to an optimal solution for S_2. So the approach of selecting the last activity to start that is compatible with all previous selected activities when run on S, gives the exact same greedy answer as S_2 from the text, selecting activities first to finish, which we know from the text to be optimal. So the last to start approach must also give an optimal solution.

Problem 3:
This is basically the same algorithm as presented in the book for selecting the first activity to finish, but instead we pick the last activity to start and work backwards. The following algorithm works as follows: First we sort by start times. Since we consider the activities in order of start times, the maximum finish time m_t will always be the maximum finish time for any activity in A. The loop considers every activity and adds activity to the result if it is compatible with all previous selected activities

```
function activity_selector(activities)
   sort array by start time
   m_t = max finish time //First one appended to solution
   results = []

   for each activity:
      if activity[finsish_time] <= m_t //Is it compatible?
         add activity to results

      m_t = activity[start_time] //move to next activity in sorted list as "last to start"

   return results
```

Running time: Since the for loop for each activity is O(n) time, and the sorting is $O(nlogn)$ when using merge sort, the running time of the whole function is O(nlogn)