Thanhbinh Truong
CS325
Homework #4
April 22, 2018

Problem 1:

An efficient greedy algorithm that determines which class should use which lecture hall at any given time will look at their start times to figure it out. The algorithm should start by creating a lecturehall variable that starts at 0 and sorts the set of classes in ascending order based on the time the class. From there, the algorithm will select the class with the earliest start time, which should be the first element in the list, and remove that element from the set of classes. After, the algorithm will make a lecture hall for that class.

Now the algorithm will go back to the next element on the list, first checking if this class start time conflicts with the previous classes end time. If the times conflict, then the algorithm will just make a new lecturehall for the current class. Otherwise, if the classes do not conflict, then it will be added to the old lecture hall and removed from the set of classes. This loop will continue until there are no more classes in the set of classes.

The running time of this algorithm should be $\Theta(nlgn)$.

Problem 2:

An efficient greedy algorithm that determines which hotels you should stay in if you want to minimize the number of days it takes you to get to your destination is one that chooses a hotel that allows for the maximum traveling distance in a day.

For each day, this algorithm will look through the sets of hotels to find the furthest hotel that the time will allow. It will look for a hotel that is after the hotel in which we stayed in the previous night.

If a hotel **a** is found that is further than distance **d** from the previous hotel, then the hotel before **a** is the one chosen to be stayed at for the night. This step will be repeatedly until the last hotel x_n is found.

The running time for this algorithm would be $O(2n) = O(n)$.

Problem 3:

Within the greedy algorithm, it would begin by sorting the jobs in descending order based on the amount of the penalty. The job with the highest penalty would then be scheduled so that it will run and complete so that no penalty occurs. If a job needs to be scheduled, but it is in a slot that is filled, it will be forced to wait for the next available slot, sometimes resulting into a penalty.

The running time for this algorithm is $O(n^2)$

Problem 4:
The proposed approach of selecting the last activity to be the first to start in addition to being compatible with all previous selected activity is just the same as the greedy algorithm. The only difference is that we are starting from the end rather than starting from the beginning.

- Suppose we are given a set A', A' = {a'_1, a'_2, …a'_n} where $a'_i = [f_i, s_i)$.
- a'_l corresponds to the reverse of a_i.
- A subset of {a_1, a_2…, a_n} $\subset$ A is mutually compatible IFF {a'_1, a'_2, …a'_n} $\subset$A'.
    - This means an optimal A would directly map to a optimal S'.
- The approach of choosing the last activity to start when ran on A will return the same answer as the greedy algorithm counterpart when run on A'. The proposed approach is optimal.

Problem 5:

Description: The algorithm will sort the activities in increasing order of finished times. Then from there, it will create a vector to store the chosen activities and initialize it with the activity with the earliest finish time. After, it will create a variable and index the last chosen activity. Then on we will iterate from the second element to the last, selecting an activity w/ a greater start time then the last activity's finished time.

Pseudocode:

```
Main()
        Open act.txt file
        Initialize variables
        While(some condition)
        {
                create vector
                for loop()
                {
                        input act.txt into variables
                        input act.xt into vector
                }
                sort(vector)
                create vector2
                create int = 0
                for loop()
                {
                        if(vector[i].sceond.first >= vector[last].first)
                        vector2.push_back(vector[].second.second);
                        int = I;
                }
        }
```

theoretical time = O(n)