Brendan Jang
CS 325
Homework 4

1. The greedy algorithm would be as follows. For each day, we would need to loop through the hotels that come after the hotel in which we stayed the previous night. If a hotel “h” is found which is more than “d” distance away from the previous hotel we stayed at, then the hotel previous of “h” is chosen to stay for that night. This is the greedy step and we stay in this hotel. We run these steps repeatedly until we have reached the last hotel x_n.

The worst case scenario occurs if each hotel is at a distance of successive multiples of “d”. In this case, we calculate the distance to each hotel two times in the whole computation. Therefore, the worst case running time would be $\Theta(2n) = \Theta(n)$. This would be linear time.

2. Using greedy algorithm, first we need to sort the penalties in descending order. Suppose the deadline for the first job is j. If j = i, then we have to schedule the this job first. Otherwise, if j > 1, then j - 1 jobs can be scheduled before this job without receiving a penalty. Therefore, we have to choose j - 1 optimal jobs from the jobs after the first job. Continuing with this logic, if the deadline for the second job is k > 2, then k - 2 jobs can be scheduled before this job without receiving a penalty. This would be the most optimal schedule of jobs without incurring penalties. To sum up, the algorithm below: First we would sort jobs in decreasing order of penalties. We denote the jobs in this sorted array as {j_1, j_2,...,j_n}. If deadline(j_i) == i or i ==n we schedule j_i at this slot i. Otherwise, we call the algorithm on the jobs from {j_i, j_i+1,..., j_n}.

The sorting in this algorithm will take $\Theta(n \log n)$ time. One linear search of the sorted array will be be enough if implemented correctly. Therefore, the running time is $\Theta(n \log n)$.

3. Here we suppose that s = {a_1, a_2,...a_n) is a set of activities and each a_i = [s_i, f_i). The activities are sorted by finish time. We need to find the largest possible set of non-overlapping of activities by selecting the activities from the ending instead of the beginning. The algorithm, if we were to start by selecting the last activity to start, is as follows.

```
n = s.length
a = {a_n)
k = n
For m = n -1 to 1
    If f[m] <= s[k]
        A = A U {a_m}
        k = m
return A
```

This algorithm takes the starting times and finish times of the activities as input. They are sorted by finish times. As described above, the last activity to start is selected last. Then the algorithm goes through the activities in descending order and finds an activity to add it to set A. Because the algorithm attempts to search for an optimal solution in each stage, it is a greedy algorithm.

We know that the original activity selection problem always selects the first activity to finish first and the original problem is optimal. Furthermore, activities are chosen by finishing times in ascending order if the activity is compatible with already selected activities. Similarly, if the approach where we select the last activity to start, activities are selected by finishing times in descending order. We see that the modified approach addressed in this problem is a reverse of the original problem. Therefore, we can see that the new approach would also give an optimal solution.

4. Implementation lastToStart.cpp

This algorithm would be done in 4 steps. First we would sort the activities in increasing order of finish times. The second step would be to create a vector to store the chosen activities and initialize it with the activity that has the first finish time. Step 3 would be to declare a variable that keeps track of the key of the last selected activity. In the last step, we iterate from the second element of the array up to the last element. We choose an activity if the start time is greater than the activity's finish time. This would give us the final result which would be the optimal solution

The theoretical running time would be $\Theta(n \log n)$.