Pranav Prabhu

Problem 1:

a) We want to minimize the number of days. So, after sorting the end times, perform a start to last greedy algorithm, that picks the first time after each stop. For each day, we loop through the hotels which come after the hotel you stayed at last night. If a hotel is found which is more than the distance away from the last stayed hotel, then the previous hotel is chosen to stay for that night. We repeat this process until we have reached the last hotel.

Pseudocode:

```
n = number_of_stops
for (i = n; i >= 0; i--):
     if start[i] >= end[n]:
          print(stop[i])
          i = n
```

b) Running time: O(n)

Problem 2:

This is a greedy algorithm because whether we are going start to finish, or finish to start, the same optimal pathway will always be selected. The same start and end values are selected, either way, so no change in the outcome should occur.

Proof:

Let $i, k \in \mathbb{N}$ be arbitrary. Let $A = \{ a_i, \ldots, a_i\}$ be an arbitrary set of activities. Each activity a_i consist of a start integer s_i , and an end integer e_i , were $e_i > s_i$. If $A = 0$, no optimal solutions exists. If $A = 1$, then the optimal subset is the sole activity of A, with the optimal start s_i and end e_i. If $A > 1$, then a possibly optimal subset of A exists that provide the fewest activities a_i . Let k be the number of activities in A. For each activity of A. If $e_i \leq s_k$ then set i as k, and store a_i . Therefore, this will increase the value of the solution and give the optimal solution of A.

` Suppose s = $\{ a_1, a_2, \ldots, a_n\}$ is a set of activities and each $a_i = [s_i, f_i)$. Consider that the activities are sorted by finish time. Now, the goal is to find the largest possible set of

non-overlapping of activities by selecting the activities from the ending instead selecting from the beginning. The following is the pseudocode of the algorithm that finds the optimal solution by selecting the last activity to start:

```
select_activity(s, f):
    n = s.length
    A = {a_n}
    k = n
    for m = n-1 down to 1
        if f[m] ≤ s[k]
        A=A ∪ {a_m}
        k = m
    return A
```

The above algorithm takes starting times and finishing times of activities as input. They are sorted by finish time. The algorithm initially selects the last activity to start last. Then the algorithm scans through the activities in descending order and finds a compatible activity to add to set *A*. Since the above algorithm tries to find an optimal solution in each stage. Thus, this approach is obviously a greedy algorithm.

Problem 3:

A last to start greedy algorithm will start at the activity with the highest end time and use that activity's start time as the new baseline. It will then find the next activity whose end time is smaller than or equal to the baseline start time, and then use that as the new baseline. This process repeats until it reaches the end of the arrays.

Pseudocode:

```
num_activities(activity, start, finish, size):
    max = 1
    i = size - 1
    for j in range(size-2, -1, -1):
```

```
            if finish[j] <= start[i]:
                max += 1
                i = j

    print("Maximum number of activities = ", max)
    return max

activity(activity, start, finish, max):
    array = []
    selected_array = []
    i = size - 1
    selected_array.append(activity[i])

    for j in range(size-2, -1, -1):
        if finish[j] <= start[i]:
            selected_array.append(activity[j])
            i = j
    merge_sort(selected_array, max_activities)

main():
    open and read data_file
    array = []
    store integers in array

    while array_length != len(array):
        num_of_activities = array[0]
        activity_set = array[1 : (num_of_activities * 3) + 1]
        activity = activity_set[0 : len(activity_set) : 3]
        start = activity_set[1 : len(activity_set) : 3]
        finish = activity_set[2 : len(activity_set) : 3]
```

```
        insertion_sort(activity, start, finish,
num_of_activities)
        max_activities = num_activities(activity, start,
finish, num_of_activities)
        activity_selection(activity, start, finish,
num_of_activities, max_activities)

        del array[0: (num_of_activities * 3) + 1]
```

Runtime:

I used both Merge Sort and Insertion Sort to print the maximum number of activities as well as each selected activity. Thus, there time complexities are θ(nlgn) & $O(n^2)$. The greedy algorithm itself has a time complexity of O(n). Therefore, summing up all three complexities leaves me with the fastest growing runtime, which is $O(n^2)$.

- ➢ Merge Sort: θ(nlgn)
- ➢ Insertion Sort: $O(n^2)$
- ➢ Greedy Program: O(n)

Running time: $O(nlgn + n + n^2) = O(n^2)$