Kimberly Tom

CS 325

2/2/19

Homework 4

1) To schedule all the classes and place them in as few lecture halls as possible, we would start by looking at the start time and use as few lecture halls as possible with this order. Our input would be all the classes, and let's say all the classes are in a set C. From the set C, we will remove the class the earliest start time s_j. While there is a lecture hall available, place that class in that lecture hall, then remove c_j from the set of C as we have placed it in a lecture hall. Next, look at the remaining classes in C left, and pick the class with the next earliest start time. If there's a lecture hall available with classes overlapping with this class's start time, we can schedule that class on that machine. If there's no available lecture hall, then add a new lecture hall and schedule that class in the new lecture hall. Update the lecture hall count to reflect the addition of this new lecture hall. Continue removing a class from C and repeating the above steps, until there are no classes left in C. The output will be a non-conflicting class schedule with minimum number of lecture halls.
The running time of this algorithm is theta(nlogn). This includes the time it takes to sort to find the class with the next earliest start time.

2)

First, sort the penalties from of jobs from greatest to smallest. Then place the job with the greatest penalty from the remaining jobs we did not place closest to its deadline as possible. If there is no available space in the schedule, then place this job at the end and incur the penalty. The penalty will be added to the total penalty which will be a running total. Continue looping through the remaining jobs and selecting the next job to place that has the next highest penalty. The output will be an array that represents the schedule that has the least penalty and we will have kept track of the total penalty.

jobScheduleArray will be the schedule that has the least penalty
totalPenalty will be the least penalty we can incur after performing all jobs

// greedyJobs takes an array A of jobs that has attributes of penalties (p) and deadlines (d)

```
greedyJobs(A) {
// set maximum penalty to -1

maximumPenalty = -1

for (i = 1; i <= n; i++) {

        // keep looping until we find the job with greatest penalty
```

```
// after the loop terminates, job b is the job we will place next
for (b=1; b <=n; b++) {
        if (p_b > maximumPenalty) {
                maximumPenalty == p_b
        }
}
// place this job b so it finishes as close to its deadline as possible
// if there is no space for job b in time frame n, then place it at the end
// and incur the penalty
// set current position at the deadline of b
position= d_b
// while there's space in our schedule, place the job there and remove that position
while (jobScheduleArray[position] is available and position > 0) {
        position = position - 1
{
// if no positions left, incur penalty of job b and set our position to n (the end)
if (position == 0) {
        penaltyTotal = penaltyTotal + p_b
        position = n
        // move all the position left as we had to make room for the job b
        while (jobScheduleArray[position] != -1)
                position = position-1

}
// place the job b in the position in the array
jobScheduleArray[position] = j_b
// set the penalty of the current job to to -1 as we do not want to compare it again
// when finding the next greatest penalty
p_b = -1
```

}

The runtime for this algorithm is O(nlogn) because first we need to sort the algorithm, then we need to iterate n times for each job 1 through n.

3) This is similar to the greedy algorithm that looks for the activity that has the earliest finish time except we are starting from the backend of the time frame and looking for the latest start time first.
   Suppose A is a subset of S is an optimal solution, meaning it is a possible largest set of activities that do not conflict with each other. To get to this answer, order the activities in S by latest start time. That is placed in our subset A. Then, keep looking at the remaining activities for the next activity that has the latest start time and does not overlap with our previous activities in our solution. Add that next activity to subset A. Because looking for a solution in this manner finds the optimal solution in each stage (the next activity with the latest start time that does not conflict with the previous activities in subset A), it is a greedy algorithm. A greedy choice at each step yields a globally optimal solution.

4) The algorithm reads the input from act.txt then puts the element in a vector. Then it sorts the vector by latest start time, and puts the activity with the latest start time into the schedule vector first. Then it iterates through the remaining activities and if the finishing time of the next activity is less than or equal to the start time of the newest activity in our schedule vector, it adds that activity to the schedule vector.

```
// read in the values from act.txt and place elements in vector pairs
 activityVector.push_back(make_pair(activityStart, make_pair(activityEnd, activityID)));
// sort the activities by latest start time
 sort(activityVector.begin(), activityVector.end());
```

// set n to the number of activities (start times)
n <- length[s]
// place the activity with latest start time in vector
Vector <- {a_n}
i <- n
// loop starting from the end to the beginning
for m <- n-1 to 1
    // if the finish time of the next activity in sorted list is less than or equal
    // to the start of the activity in our optimal solution, add the
    // activity to the vector of optimal solution
    do if f_m <= s_i
    then Vector = Vector + a_m
    i <- m
return Vector and length[Vector]

The theoretical running time of my algorithm is O(n log n) because if the inputs are not sorted, it first has to sort the activities by latest start time which takes O(n log n) time, then the greedy part only takes O(n) time because it is just iterating through the whole list of size n to make comparisons and decide whether to add an activity to the schedule.