Erin Alltop
10-19-17
CS325 – Fall 2017
HW 4

1. A greedy algorithm is one that picks the most optimal local choice at any given time. If we have a set of classes to schedule among a large number of lectures halls where any call can place in any lecture hall, we can use a greedy algorithm to efficiently place classes in the least amount of lecture halls possible. An iterative approach to solving this problem with would be to iterate first sort our set of classes into their starting times. Then we can easily go through our set and assign them to compatible classes. For each class we can check if the finishing time of the class overlaps with the previous class. If so, then we move the class to the next lecture hall and check again. If it is not compatible with any lecture halls, then we can put in a new lecture hall, confident that there is no other place for it. This should guarantee that the least number of lecture halls are used for our classes. Pseudocode for this algorithm is as follows:

```
scheduleClasses(Set of classes)
{
//1. sort classes by start time
//iterate through classes by start time
for (x = 0; x < classes; x++)
{
        y = 0 //set y to first lecture hall
        while(x is not scheduled)
        {
        if(x finish time compatible with lecture hall y)
        {
        Schedule class x in lecture hall y
        }
        else if(x finish time not compatible with lecture hall y)
        y = y +1 //allocates new lecture hall if necessary
        }
}
}
```

   This algorithm has a for loop that contains a while loop. The for loop executes n times so it is O(n). In a worst case, our while loop will need to iterate through all of the lecture halls, so that would also be O(n). And since the while loop is nested within the for loop, then this should have a **running time of $O(n^2)$.**

2. If we are going on a road trip and can only travel d miles, then with a map of n hotels and the distances between them (in increasing order), then the greedy choice would be to find the furthest hotel we can within d miles each day until we reach the last hotel which is our destination. Thus, our most efficient algorithm would be an iterative one. For each new day

while we have not yet reached our destination, we would loop through the hotels that are beyond where we have gone on the map and check if they are beyond our driving distance. Once we find the hotel that is just beyond our driving distance then we pick the preceding hotel to be where we will stop for the night. Some pseudocode for our algorithm:

n = number of hotels
d = distance we can drive in a day
set of x = hotels
h = distance of hotel from previous
while ($x != x_n$) //while we have not reached our destination hotel
//check if x is greater than distance d away
//if yes, then x = x - i
//add x to solution set

This would algorithm would be $\theta(n)$ because we are iteratively going through each hotel to determine its distance. We only need to check each hotel once.

3. If we have a problem where we wish to schedule jobs such that the penalties for missing deadlines is minimal, we can use an efficient greedy algorithm to solve this problem. The first thing we would do is sort the jobs by their penalties in descending order. For example, if we have a set of jobs
j = {1, 2, 3, 4, 5, 6, 7, 8} with deadlines
d = {5, 7, 6, 2, 3, 1, 4, 9} and penalties
p = {50, 40, 10, 80, 90, 20, 30, 60}, then we want to sort our jobs like so:
j = {5, 4, 6, 1, 2, 8, 7, 3}
d = {3, 2, 1, 5, 7, 9, 4, 6}
p = {90, 80, 60, 50, 40, 30, 20, 10}

   By sorting our data into descending penalty order, then we can minimize the amount of penalties incurred. In our example, if each job takes 1 minute to complete and if we are given 5 minutes to schedule our jobs, then we will be unable to complete jobs j = {8, 7, 3} which will give us a total penalty of 60. Compared to our original set, if we had not sorted our data our penalty would have been 110. And if we had sorted to maximize the number of jobs completed, our data set would be:
j = {6, 4, 5, 7, 1, 3, 2, 8}
d = {1, 2, 3, 4, 5, 6, 7, 9}
p = {20, 80, 90, 50, 10, 40, 60}
In this scenario our penalties would also be 110, the same as our original unsorted set.

   If we use an efficient sorting method such as merge sort, our sorting time would be $O(n \log n)$. Next we want to find the set of jobs that can be completed at or before their deadline. We will iterate through each job and check to see if its deadline has passed. If it has then we do nothing with it, if not then we can add it to our set.

Some simple pseudocode for this algorithm:

```
//sort in decreasing order of penalties
for j_i to j_n
{
        while the deadline is greater than current time
        {
                select j_i for our set
        }
}
```

Since this algorithm has a nested while loop within the for loop, then the **time complexity can be said to be $O(n^2)$.**

4. Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. This approach is both a greedy algorithm and yields an optimal solution. The approach is the same algorithm starting from the end instead of the beginning (as would be expected).
   For example, if we are given the same set of S activities that have start and finish times denoted by s_i and f_i respectively. If we instead switch the start and end times in the algorithm it will work the same way in reverse as it does in the original algorithm. That is, we will select the last activity to start that is compatible with all of the previously selected activities. The algorithm will discard any activities that are incompatible with the already chosen activities which is exactly how the normal algorithm works as well. It will yield the same set of activities.

5. For the last to start algorithm implementation, we are examining each activity beginning with the activity that is, obviously, the last to start. The first thing we need to do is sort our data set in descending order of start times. This way we can begin with selecting our first activity that is the last to start and work our way "backwards" through the activities to see if they are compatible with each other. Using the data set example for the act.txt file, we can examine the second set of data points as an example. Namely S = {1, 8}, {1, 2}, {3, 9} with each pair of numbers represented as s_k and f_k which are the start and finish times of each activity.

   We want to first sort our data set with an efficient algorithm, so we will choose the mergesort algorithm which has an O(n log n) running time. We are sorting by start times in descending order, so our newly sorted set is S = {3, 9}, {1, 8}, {1, 2}. Note that the two latter activities have the same start time so might be sorted differently based on the algorithm used. We automatically select the first activity for our activities selection:

Activity 2:
Start: 3 Finish: 9

   Note here that we are selecting the last activity of the day. Next we examine the next activity which has a start and finishing time of {1, 8}. Instead of checking to see if the start time is less

than the finishing time as we would do in the first to finish algorithm, we instead need to check to see if the next or better said "previous" activity has a finish time that is less than or equal to our start time. We can see in this example that it does not, so they are incompatible and we throw that activity out.
Lastly we look at the final activity which has start and finishing times {1, 2}. We examine it against our last selected activity and see that the finish time is less than or equal to the start time of our selected activity so it is compatible with our activities. We can then select it for our set:

Activity 1: Selected 2nd.	Activity 2: Selected 1st.
Start: 1 Finish: 2	Start: 3 Finish: 9

Pseudocode for the last to start algorithm of the activity selection problem:

```
//first sort data set by descending start times
lastStart(A[1…0]) //array or vector of objects
//select first activity at beginning on algorithm
//for(j = 1; j < sizeof(A); j++)
If(finish time of next activity <= start time of current activity)
//select activity
else discard activity
```

For my algorithm I will be using the mergesort algorithm to efficiently sort the data set. This has a running time of O(n log n). Since our activity selection algorithm is looping through activities and checking them against each other, then the complexity will be O(n). So with this algorithm, the **theoretical running time will be O(n log n).**

Code has been submitted to TEACH.