Christopher Ragasa
CS 325 – HW 4
4/29/18

Problem 1: *(5 points)* **Class Scheduling**:
Suppose you have a set of classes to schedule among a large number of lecture halls, where any class can class place in any lecture hall. Each class cj has a start time sj and finish time fj. We wish to schedule all classes using as few lecture halls as possible. Verbally describe an efficient greedy algorithm to determine which class should use which lecture hall at any given time. What is the running time of your algorithm?

This problem can be solved using a greedy algorithm scheduling method that was discussed in this week's lecture. In the lecture, there is an example where we are given a set T of n tasks, with each task having a start (si) and finish (fi) time. The goal was to perform all tasks on the minimum number of machines. Problem 1 can be solved using the same exact approach.

The first thing we would need to do with the set of classes (let this set be T) is sort them by their start times (soonest start time at the beginning of the list, latest start time at the end of the list). This can be done in O(n log n) time.

After the set T is sorted, we can use it as the input to our greedy algorithm with the output being a non-conflicting class schedule with the minimum number of lecture halls. The first thing to do in the algorithm is to initialize the number of lecture halls to zero. You would next run a while loop while the set T is not empty. In this while loop, you would first remove class i (task at the beginning of the set with the smallest si). If there is a lecture hall j available for class i, schedule class i in lecture hall j. Otherwise, if there is no lecture hall available, create a new lecture hall by incrementing m by 1 and then schedule that class in lecture hall h.

Let's take a look at the pseudocode:
```
    Algorithm ClassScheduling(T):
        h <- 0
        while T is not empty
            remove class i with smallest si
            avail <- 0
            for j from 0 to h
                if fj of class j <= si of class i
                    schedule class i on machine j
                    avail = 1
            if avail = 0
                h = h + 1
                schedule class i on machine h
```

Let’s take a look at the running time of the ClassScheduling algorithm. There is a while loop that iterates while T is not empty. This runs at O(T). Inside this while loop, there is another for loop that checks the availability of each lecture hall. The amount of lecture halls (let this value be h) is dependent on the start and finish times of the classes in set T. In the worst possible case, where every class needs to be scheduled in a new lecture hall, h will grow to be size T. However, if we store these lecture halls in a heap data structure, we can find availability in O(log n) time. Therefore, the running time of this algorithm is O(n log n).

Problem 2: *(5 points)* **Road Trip**:
Suppose you are going on a road trip with friends. Unfortunately, your headlights are broken, so you can only drive in the daytime. Therefore, on any given day you can drive no more than d miles. You have a map with n different hotels and the distances from your start point to each hotel $x_1 < x_2 < ... < x_n$. Your final destination is the last hotel. Describe 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. What is the running time of your algorithm?

This problem can be done with a greedy algorithm where the selection criteria is the hotel with the largest distance x that does not exceed d (amount of miles possible to drive in a single day).

Let the algorithm take the following inputs: N (list of hotels with given distances from start point) and d (max drivable distance per day). Given these inputs, we can return an output A which is an optimized list of all the hotels visited before reaching the last hotel.

The first step of the algorithm is to initialize days, miles, the last hotel, and an empty list. Next, iterate through each of the hotels in the list and keep track of the total distance traveled. If the total distance traveled exceeds the max allowable, increment the day count by 1 and append the previous hotel to the list. After the loop is done, append the final hotel (since it is the final destination) and return the list. This is an optimized list with the minimum amount of travel days.

Let’s take a look at the pseudocode:

```
RoadTrip (N, d):
    days ← 0
    miles ← d
    h_last ← 0
    H ← [ ]
    for i in range(N):
        h = i – 1
        if h < 0:
            h = 0
            miles = miles – $N_i$
        else:
            miles = miles – (N_i – N_h)
        if miles <= 0:
            days++
            miles = d
            append N_h
    append N_last
```

In order to determine the running time, let's take a look at the algorithm line by line. The first four lines are assignments; these operations run in O(1) constant time. The for loop that starts on line 5 iterates through each of items in the list. This runs in O(n) time. All other operations within this loop run in constant time. Therefore, the total run time of this algorithm is O(n).

Problem 3: *(5 points)* **Scheduling jobs with penalties**:
For each $1 \leq i \leq n$ job j_i is given by two numbers d_i and p_i, where d_i is the deadline and p_i is the penalty. The length of each job is equal to 1 minute and once the job starts it cannot be stopped until completed. We want to schedule all jobs, but only one job can run at any given time. If job i does not complete on or before its deadline, we will pay its penalty p_i. Design a greedy algorithm to find a schedule such that all jobs are completed and the sum of all penalties is minimized. What is the running time of your algorithm?

Under the assumption that the input to the algorithm is a list of job objects with properties deadline (d_i) and penalty (p_i), we can design an algorithm that returns a list of jobs in an order than minimizes the penalty.

The first thing to do in this algorithm is create a copy of the input list of jobs that is sorted decreasing penalty. Let this list be A_sorted. You would then create an empty list B, which will be the optimized schedule of jobs, and another empty list C, which will contain all of the jobs that we were not able to schedule the first time around.

Next step would be to have a for loop that iterates through each of the jobs in A_sorted, starting with the highest penalty job (first index). For each of these jobs, create an index for it's deadline and compare it to B[index – 1] to check for availability. If there is availability in that slot, schedule the job and move onto the next one. Otherwise, check for earlier spots in B to see if that job can be scheduled. If it can't, append that job to C, so that we can store it and eventually append it to B after we do the first iteration of scheduling jobs. Let's take a look at the pseudocode:

```
JobScheduling(A):
    A_sorted ← sort A by decreasing penalty
    B ← [ ]
```

```
C ← [ ]
for each in A_sorted:
    index = A_sorted[i].deadline
    if B[index – 1] == 0
        B[index – 1] = A_sorted[i]
    else
        while(B[index – 1] != 0 and index >= 0)
            index = index – 1
        if index == -1
            push A_sorted[i] to C
for each in C:
    B.append(C[i])
return A_sorted
```

To determine the running time of this algorithm, let's take a look at the code line by line. Starting with the first line, we create a new sorted list. This is done in O(n log n) time. The next two steps are creating empty lists. These 2 operations can be done in constant time, O(1). On line 4, we create a for loop that goes through each element in A_sorted. This is done in O(n) time. Inside the loop, we have a while loop that iterates through the B list. In the worst case, this operation will also be O(n). After the first for loop, we have a second for loop that appends all of the missing jobs to the B list. However, since they are not nested, we don't need to multiply their time complexities together. In conclusion, final running time of this algorithm is O(n^2), which is caused by the loop in lines 4-12.

Problem 4: *(5 points)* **CLRS 16-1-2 Activity Selection Last-to-Start**
Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible will all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

A greedy algorithm is defined in the book as one that always makes the choice that looks best at the moment. This is exactly the behavior that the algorithm above exhibits. Instead of choosing the task with the earliest finish time, it chooses the task with the latest start time. It is the same exact greedy algorithm in reverse.

Let's take a look at an example.

In the first scenario, we are given a set S = {a1, a2, …, an} of activities where a_i = [s_i , f_i). We are asked to find an optimal solution by selecting the last activity to start that is compatible with all previously selected activities. For the sake of example, let's create another set S' = {a1', a2', … , an'} where a_i' = [f_i', s_i'). It is clear that a_i' is a_i in reverse. Therefore, an optimal solution for S' maps directly to an optimal solution for S.

Problem 5: *(10 points)* **Activity Selection Last-to-Start Implementation**

You may use any language you choose to implement the activity selection last-to-start algorithm described in problem 4. Include a verbal description of your algorithm, pseudocode and analysis of the theoretical running time. *You do not need to collected experimental running times.*
The program should read input from a file named "act.txt". The file contains lists of activity sets with number of activities in the set in the first line followed by lines containing the activity number, start time & finish time.

See attached python file for code.

My algorithm first initializes an empty list to hold a sorted list of jobs by decreasing start time. For each sorted list, insert the first activity. Since the list is sorted by latest start time and we are creating the schedule starting from the end, the first activity in the sorted list will always be selected. Next, iterate through the rest of the activities. If the activity has a finish time less than or equal to the start time of the previously selected activity, then select it. Print the schedule to the console.

Let's take a look at the pseudocode:

```
for i from 0 to length of all_lists
    list_of_jobs ← [ ]
    n = length of total_schedule[i]
    m = 0
    insert first activity into list_of_jobs

    for j from 0 to length of the current_list
        if stop time of job[i][j] is less than the start time last added job
        insert job into list_of_jobs
        m = j

print the list
```

In order to determine the theoretical running time, let's take a look at the pseudocode line by line. The first line starts with a for loop that iterates through all of the lists within the list that is passed as input. This could be size n. Therefore, this operates in O(n) time. The next 4 lines after this operate in constant O(1) time. The next for loop iterates through each of the sublists within the list passed as input. This could also be size n, which operates in O(n) time. The work within this loop is constant time. Therefore, since these lists are nested, the total running time of the algorithm, is O(n^2).