CS352 (Winter 2018) - Homework 4

Marc Tibbs (tibbsm@oregonstate.edu)

Due Date: February 4, 2018

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 take place in any lecture hall. Each class c_j has a start time s_j and finish time f_j. 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?

Suppose we have a set T of j classes each having a start time s_j and a finish time f_j where $s_j < f_j$. Using a greedy algorithm, we are able schedule all the classes without conflict, while optimizing the schedule so that the minimum number of lecture halls are used. The first step would be to sort all the classes by their start times. The sorting process, if we use merge sort, would have a running time of $\Theta(n \, log \, n)$.

After sorting, the greedy algorithm would continue by placing the class with the smallest starting time (c_1) into the an available lecture hall. The algorithm would then move onto placing the class with the second smallest starting time(c_2). If the starting time of (c_2) is greater than the previously placed class's finish time, then the algorithm would add the class into the same classroom after the first class. If it the starting time is less than that of the first class, then the algorithm would add the class to another lecture hall.

The algorithm would continue on in this manner until all classes are scheduled. For each class the algorithm would first check to see if any of the previously used class rooms are available for the class currently being scheduled. If none of the previous lecture halls are available then the algorithm would add the class to a new lecture hall. An example of this type of algorithm is shown below in pseudocode. The running time for such an algorithm would be $\Theta(n \, log \, n)$ as it is bound by the sorting process of the classes.

classSchedule(T)

```
1  $m \leftarrow 0$
2  while T is not empty
3      remove class $j$ with the smallest $s_j$
4      if there's a lecture hall $k$ for $j$ then
5          schedule $j$ in lecture hall $k$
6      else
7          $m \leftarrow m + 1$
8          schedule $j$ in lecture hall $m$
```

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?

The greedy strategy for finding the minimum days to get to the final desination is to drive as far as possible each day while keeping the total driving distance for the day $< d$. On the first day, the greedy algorithm would check each distance marker x_i against d until it found a value which was greater than d. The optimal solution for the first day would be the staying the night at the hotel at marker x_i that was less than d and was directly before the marker x_{i+1} which was greater than d. For the second day, we would use the same strategy making sure to subtract the total distance traveled in the first day from the current day's running distance calculations. This pattern would continue until the final destination x_n was reached. The running time for this algorithm would be $\Theta(n)$ as each value would only need to be tested once.

```
roadTrip(T)
1  distanceTraveled ← 0
2  hotelStops[] ← 0
3  for each T
4     if $x_i - distanceTraveled$ is $> d$ and $x_{i-1} - distanceTraveled \leq d$ then
5          $distanceTraveled \leftarrow x_{i-1}$
6          $hotelStops \leftarrow x_{i-1}$
```

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?

To complete all the task while minimizing the sum of penalties, a greedy algorithm should start by sorting the array of tasks by their penalties in decreasing order from left to right. Once the array is sorted, the greedy algorithm will place the task with the highest penalty in an array with the index of its deadline. The greedy algorithm would then continue to the next task and place it either at the index of the deadline or, if that space is taken by a previous task, it will look for a lower index to place the task at. If there is no space for the task before its deadline then it would be placed in the last time slot available and its penalty would be added to a running sum of all the penalties incurred. The running time for such an algorithm would be $\Theta(n \, log \, n)$ as it is bound by the sorting process of the classes.

taskSchedule(T)

```
1   while T is not empty
2     remove task $i$ with the largest penalty $p_i$
3     if the time slot where $i$'s deadline $d_i$ is available then
4         schedule $i$ in that time slot $k$
5     else if slot $k$ is not available then
6         while --$k \geq 0$
7             if an earlier time slot is available then
8                 schedule $i$ in the earlier time slot $k - x$
9     else if $i$ cannot be scheduled before its deadline then
10         schedule $i$ into the last available time
11         add its penalty to the running sum of penalties.
```

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.

This variation of the Activity Selection problem would essentially be the same thing as the original Activity Selection problem and could there for be solved with a greedy algorithm similar to the one that provides optimal results for the original problem. The differences would be selecting the last start times rather than the first finish time and after you would be looking for a finish time that was less than the previous activities start time.
A greedy algorithm to solve this would look almost identical to the First-to-Finish algorithm.
Suppose $A \subseteq S$ is an optimal solution.

- Order the activites in A by start time in descending order. The first activity in A is k.
  - if $k = 1$, the schedule A begins with a greedy choice.
  - if $k \neq 1$, we must show that there is an optimal solution B to S that begins with a greedy choice, activity 1.
- Let $B = A - \{k\} \cup \{1\}$
  - $f_1 \leq f_k \rightarrow$ activities in B are disjoint (compatible).
  - B has the same numer of activities as A.
  - Thus, B is optimal.

Once the greedy choice of activity 1 is made, the problem reduces to finding an optimal solution for the activity selection problem over those activities in S that are compatible with activity 1.

- if A is optimal to S then $A' = A - \{1\}$ is optimal to $S' = \{i \in S : f_i \leq s_1\}$
- if we could find a solution B' to S' with more activities than A', adding activity 1 to B' would yield a solution B to S with more activities than A, which contradicts the optimality of A.

After each greedy choice is made, we are left with an optimization problem of the same form as the original problem.

- By indcution on the number of choices made, making the greedy choice at every step produces an optimal solution.

Now we must ask if the greedy choice of choosing the last activity to start is always part of one of the optimal solutions to the problem. This is shown in the following theorem.

Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the latest start time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the latest start time. If $a_j = a_m$, we are done, since we have shown that a_m is in some maximum-size subset of mutually compatible activities of S_k. If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but substituting a_m for a_j. The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, a_j is the last activity in A_k to start, and $f_m \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities os S_k, and it includes a_m.

Therefore, we can repeatedly choose the activity that starts last, keep only the activities compatible with this activity, and repeat until no activities remain. Because we always choose the activity with the last start time, the start times of the activities must strictly decrease and we only need to consider each activity once.

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

A copy of all files, including the act.txt files, and a README file have been submitted to TEACH.

Include a verbal description of your algorithm, pseudocode and analysis of the theoretical running time. You do not need to collected experimental running times.

lastToStart(s, f)

```
1  $n = s.length$
2  $A = \{a_1\}$
3  $k = 1$
4  for $m = 2$ to $n$
5     if $f[m] \leq s[k]$
6           $A = A \cup \{a_m\}$
7           $k = m$
8     return A
```

This algorithm works very similarly to the Greedy-Activity-Selector algorithm that is provided in the textbook on p. 421. The variable k indexes the most recent addition to A. Since we consider the activities in order of monotonically decreasing start time, s_k is always the minimum start time of any activity in A. That is,

$$s_k = min\{s_i : a_i \in A\}.$$

Lines 2-3 select activity a_1, initialize A to contain just this activity, and initialize k to index this activity. Then, the **for** loop of lines 4-7 finds the latest activity in S_k to start. The loop considers each activity a_m in turn and adds a_m to A if it is compatible with all previously selected activities; such an activity is the earliest is S_k to finish. Once all the activities are consider and added, or not, the algorithm returns an array of all the selected, compatible activities.

The theoretical running time of the algorithm is $\Theta(n)$ if it receives a sorted list or $\Theta(n\,lg\,n)$ is it has to sort the list using merge sort.