COMP4500/7500 Advanced Algorithms & Data Structures
Sample Solution to Tutorial Exercise 7 (2014/2)*

School of Information Technology and Electrical Engineering, University of Queensland

September 8, 2014

1. (CLRS Exercise 25.2-4, p699 [3rd] ; CLR Exercise 26.2-2)

   The Floyd-Warshall algorithm (CLRS p695 [3rd]) requires $\Theta(n^3)$ space because we compute $D^{(k)}$ for $k = 0, 1, \ldots, n$ and each matrix has n^2 elements.

   FLOYD-WARSHALL(W)

   ```
   1  // W is the weight matrix
   2  n = W.rows
   3  D^(0) = W
   4  for k = 1 to n
   5      let D^(k) = (d_ij^(k)) be a new n × n matrix
   6      for i = 1 to n
   7          for j = 1 to n
   8              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
   9  return D^(n)
   ```

   Show that the following procedure, which simply drops all the superscripts, is correct, and thus only $\Theta(n^2)$ space is required. You may assume that W has no negative-weight cycles.

   FLOYD-WARSHALL$'(W)$

   ```
   1  n = W.rows
   2  D = W
   3  for k = 1 to n
   4      for i = 1 to n
   5          for j = 1 to n
   6              d_ij = min(d_ij, d_ik + d_kj)
   7  return D
   ```

   Sample solution. The result of the algorithm is unchanged. What happens is that in the k^{th} iteration of the main loop of FLOYD-WARSHALL$'$, $d_{ij}^{(k-1)}$ is overwritten by $d_{ij}^{(k)}$ in location d_{ij}. You need to argue that the overwriting does not damage the algorithm.

   We'll show that after the k^{th} iteration of the loop in the revised algorithm, D is the same as $D^{(k)}$ in the basic algorithm. Before the outer loop starts we have $D = D^{(0)} = W$. Consider iteration k of the outer loop of the revised algorithm. We'll assume that at the start of the iteration $D = D^{(k-1)}$ and show that $D = D^{(k)}$ at the end of the iteration. Element d_{ij} is updated if $d_{ik} + d_{kj} < d_{ij}$. Because of the order in which elements d_{ij} are updated, the value of d_{ik} may have been already updated on the k^{th} iteration for $k < j$ and similarly, the value of d_{kj} may have been already been updated $k < i$. (For $k \geq j$, $d_{ik} = d_{ik}^{(k-1)}$ and for $k \geq i$, $d_{kj} = d_{kj}^{(k-1)}$.) In the k^{th} iteration d_{ik} is updated if $d_{ik} + d_{kk} < d_{ik}$, however, assuming there are no negative-weight cycles (see question 2), we have $d_{kk} = 0$ and hence d_{ik} is unchanged. Similarly, d_{kj} is updated if $d_{kk} + d_{kj} < d_{kj}$ but $d_{kk} = 0$ and hence d_{kj} is not updated. Hence when d_{ij} is updated, we have $d_{ij} = d_{ij}^{(k-1)}$, $d_{ik} = d_{ik}^{(k-1)}$ and $d_{kj} = d_{kj}^{(k-1)}$, and hence the new value of d_{ij} equals $d_{ij}^{(k)}$. Each element d_{ij} is updated at most once on iteration k and hence at the end of the iteration $D = D^{(k)}$.

   Note that the above arguments do not apply if W has negative-weight cycles because d_{kk} may be negative if there is a negative-weight cycle that includes vertex k. In this case when d_{ij} is updated we may have have $d_{ik} < d_{ik}^{(k-1)}$ or $d_{kj} < d_{kj}^{(k-1)}$, or both, and hence we may have the updated value of $d_{ij} < d_{ij}^{(k)}$.

2. (CLRS Exercise 25.2-6, p700 [3rd]; CLR Exercise 26.2-5)

   How can the output of the Floyd-Warshall algorithm be used to detect the presence of a negative-weight cycle?

   Sample solution. There is a negative weight cycle if and only if $d_{ii}^{(n)} < 0$ for some vertex i.

   IF: $d_{ii}^{(n)}$ is a path weight from i to itself, so, if it is negative, there is a path from i to itself (i.e., a cycle) with negative weight.

   ONLY IF: Assume there is a negative weight cycle. Consider one with fewest vertices. If it has one vertex (self-loops allowed) then $w_{ii} < 0$, so d_{ii} starts out negative. Since d values are never increased it is still negative at the end.

   Otherwise, let k be the highest numbered vertex on the cycle and let i be any other vertex on the cycle. Then $d_{ik}^{(k-1)} + d_{ki}^{(k-1)}$ must be negative, since the negative cycle will be implicitly considered in the evaluation of these d values because $d_{ik}^{(k-1)}$ and $d_{ki}^{(k-1)}$ both consider intermediate paths through vertices 1 through $k-1$ and k is the highest numbered vertex in the cycle. It follows that $d_{ii}^{(k)}$ will be set to $d_{ik}^{(k-1)} + d_{ki}^{(k-1)}$, which is negative. Again, since d values are never increased it is still negative at the end.

3. (CLRS Exercise 16.1-4, p422 [3rd], 16.1-3 [2nd]; CLR Exercise 17.1-2 [1st])

   Suppose that we have a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. Each activity has a given start and finish time. We wish to schedule all the activities using as few lecture halls as possible.

   (a) Give an efficient greedy algorithm to determine which activity should use which lecture hall.

   Sample solution. Let S be the set of activities. One solution is to make use of the algorithm given in Chapter 16 of CLRS (Chapter 17 of CLR) to find a maximal size set of compatible activities. We apply this algorithm to S to find the activities which are to be scheduled for the first lecture hall, remove these activities from S, and apply the algorithm again to determine the activities for the second hall, and so on, until all activities have been allocated to a lecture hall. This takes $O(|S|^2)$ time in the worst case, where $|S|$ is the size (cardinality) of the set S.

   A better approach is to consider the start and finish events of all the activities. We process these in time order. If there is a finish event of one activity at the same time as a start event for another, we process the finish event first. We maintain a set of lecture halls that have been used but are currently *free*, and a set of *never* allocated halls. We process the events as follows,

   start event if there is a *free* hall allocate it to the activity, otherwise allocate a *never* used hall.

   finish event return the hall used by the activity to the set of *free* halls.

   (b) Justify that you algorithm has the greedy-choice property.

   Sample solution. The strategy uses as few halls as possible. To see this consider the schedule created by the strategy and let h be the number of lecture halls used by the schedule. Consider the point in the algorithm where the last of the h lecture halls is allocated to an activity for the first time. A lecture hall is only allocated to an activity for its duration and on completion the hall is always returned to the set *free*. When the hth hall is allocated for the first time *free* must be empty (otherwise an element of *free* would be allocated) and hence all $h-1$ other halls must be in use by other activities. That implies that there are h activities concurrently active and requiring lecture halls at the same time. Hence any strategy for allocating halls will use at least h halls.

   (c) Give the worst-case time complexity of your algorithm.

   Sample solution. There are $2|S|$ start and finish events. Sorting these into time order can be done in $O(|S| \lg |S|)$ time. The events are then processed in order. Each of these operations can be accomplished in $O(1)$ time, giving a total of $O(|S|)$ for the event processing. The worst case time complexity is therefore $O(|S| \lg |S|)$.

   The $O(1)$ complexity for the individual event processing can be achieved by representing the set *free* as a list. Adding to the set is implemented by inserting at the head of the list, and choosing from the set is

implemented by selecting its first element. If we assume that the lecture halls are numbered consecutively starting at one, then the set $never$ can be represented by a single number: the least lecture hall number of all those lecture halls that have never been allocated.

4. (CLRS Exercise 16.1-3, p422 [3rd]; 16.1-4 [2nd]; CLR Exercise 17.1-3 [1st])

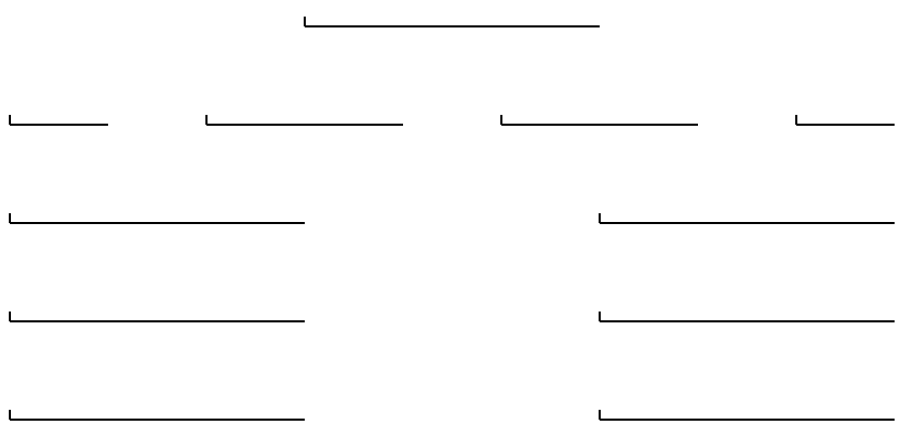
   Not just any greedy approach to the activity selection problem produces a maximum-size set of mutually compatible activities.

   (a) Give an example to show that the approach of selecting the activity of least duration from those that are compatible with previously selected activities does not work.

   Sample solution. The activity of least duration from those that are compatible with previously selected activities may overlap with two activities of longer duration, which themselves do not overlap. For example, assume the activities are [3,6), [6,9) and [5,7). The activity of least duration is [5,7), but it overlaps with and hence excludes both the other activities. To maximise the number of activities we should choose both [3,6) and [6,9).

   (b) Do the same for the approach of always selecting the activity that overlaps the fewest other remaining activities.

   Sample solution. The activity that overlaps the fewest other remaining activities may overlap with two other activities that do not themselves overlap. For example, consider the following activities:



   The top-most activity only overlaps with the two activities immediately below it, but all other activities overlap with at least three other activities. If we select the top activity we exclude the two activities it overlaps with, and we can have at most three activities. However, the four activities on the second line (including the two that overlap the top activity) are compatible, so choosing the activity that overlaps with the least number of other activities does not guarantee an optimal solution.

5. (CLRS Exercise 16.2-1, p427 [3rd]; CLR Exercise 17.2-1)

   The **0-1 knapsack problem** is as follows. A thief finds N items; the ith item is worth v_i dollars and weighs w_i kilograms, where v_i and w_i are integers. He wants to take as valuable load as possible, but can carry at most W kilograms in his knapsack for some integer W. What items should he take? (It is called **0-1** because each item is either taken or left behind; a fraction of an item cannot be taken. This is a **hard** problem to solve optimally.)

   In the **fractional knapsack problem** the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item. A sample item for the 0-1 problem might be a gold bar, while for the fractional problem it might be gold dust.

   Prove that the fractional knapsack problem has the greedy-choice property. In the fractional knapsack problem we may take fractions of an item to add to the knapsack. The greedy strategy is to process the items in non-increasing order of their value per unit weight, adding them to the knapsack until it is full. For the last item we may only add a fraction of its available weight before filling the knapsack.

Sample solution. We want to show that this strategy is optimal: it maximises the value of the items in the knapsack. Consider any optimal solution S, we need to show that there is a solution T with the same value attained using the greedy strategy.

Let w_j be the available (maximum) weight of item j, and S_j be the weight of item j used in the optimal solution S. Assume that the items are numbered from 1 through to n in the order they are processed by the greedy strategy. The items are in non-increasing order of value per unit weight. Let item i be the first item in S which does not use all the available weight of that item. If i is the last item in S, then S corresponds to the solution obtained by the greedy strategy and we are done.

If i is not the last item in S then consider item $i + 1$. Its value per unit weight is no greater than that of item i, so we may replace a weight x of item $i + 1$ by an equal weight of item i without decreasing the value of the overall solution. We choose the weight x to be the minimum of the unused portion of item i and the used weight of item $i + 1$: $x = \min(w_i - S_i, S_{i+1})$. If $x \neq w_i - S_i$ (we have not used all of item i yet) then we can repeat the above process, this time replacing item $i + 2$ by item i. We can continue doing this until either we have no further items we can replace — i is the last item and we have the greedy solution — or we have used all the available weight of item i. In the latter case we repeat the whole of the above process starting with item $i + 1$ (which we know does not use all the available weight) instead of i.

6. (CLRS Exercise 16.2-4, p427 [3rd]; CLR Exercise 17.2-4)

   Professor Midas drives an automobile from Brisbane to Sydney along the New England Highway. Her car's petrol tank, when full, holds enough petrol to cover k kilometres, and her map gives the distances between petrol stations on the route. The professor wishes to make as few stops for petrol as possible along the way. You may assume her petrol tank is initially full.

   (a) Give an efficient method by which Professor Midas can determine at which petrol stations she should stop. (This is the easy part.)

   Sample solution. The strategy is simple: she should go as far as possible before stopping to fill up with petrol at each stage.

   (b) Prove that your strategy yields an optimal solution.

   Sample solution. To prove this gives an optimal solution, we prove that any optimal solution, S, to the problem can be converted to a solution, T, with the same number of stops, that is also the solution given by our strategy of going as far as possible before stopping. Our proof is by induction on the number of stops.

   Base case. If no stops are required, then the distance must be less than k, and our strategy of going as far as possible requires no stops as well. (We assume that the car begins with a full tank of petrol.)

   Inductive step. Assume that, for all problems for which an optimal solution with n steps exists, our strategy also gives an (optimal) solution with n stops. Now consider an optimal solution, S, to a problem requiring $n + 1$ stops. The first stop in S must occur before or at the first stop in T because the solution, T, is given by going as far as possible before stopping. Replacing the first stop in S with the first stop in T still gives an optimal solution, S', because

   - S' has the same number of stops as S,
   - the car could make it to the first stop in T and hence can make it to the first stop in S', and
   - the car could get from the first stop in S to the second stop in S, hence the car can get from the first stop in S', which is no earlier than the first stop in S, to the second stop in S', which is the same as the second stop in S.

   Now consider the problem of getting from the first stop in S' to the ultimate destination. If we omit the first stop from S', we must have a optimal solution, S'', to this problem, otherwise, there would be a solution to the whole problem with fewer stops. (Note that this corresponds to what we call optimal substructure: optimal solutions to the problem contain optimal solutions to subproblems). However, S'' is of length n so, by our induction hypothesis, our strategy also gives an (optimal) solution, T'', with n stops. If we add the first stop of S' (and also T) to T'' we get a solution, T' to the whole problem that is the same length as the optimal solution S' and hence is T' is also optimal. However, by construction, T' is the same as T. Hence T is optimal.