COT5405, Spring 2010

(Partial) **Answer Key to Assignment #5**

1. (10 pts.) Exercise 15-7 on p. 369 of Cormen et al.'s text. Specifically, be sure to, (1) define a recurrence with explanations including the boundary conditions for solving the problem; (2) describe a bottom-up implementation of the algorithm; and (3) analyze both the space- and time-complexity of the implementation.

Answer: define a function $V(S, j)$ for subset $S \subseteq \{a_1, a_1, \ldots, a_n\}$ and $0 \le j \le d$ where $d = \sum_{k=1}^{n} t_k$, t_k denotes job k's processing time, and $V(S, j)$ = the maximum profit obtained by scheduling jobs in S with a starting time set at j. Thus, $V(\emptyset, j) = 0$ for $0 \le j \le d$ since no jobs are available for scheduling. Note that we may assume every deadline $d_k \le d = \sum_{k=1}^{n} t_k$; this is because the most that can be scheduled are all n jobs and any deadline beyond the total processing time of these jobs can be reduced to d without changing the outcome. Thus, $V(S, d) = 0$ for any subset S of jobs since the deadlines for all jobs have passed if the current time is d.

In general, when $S \ne \emptyset$ and $0 \le j < d$, there are two cases in calculating $V(S, j)$ depending on whether or not all jobs in S have passed the deadlines at time j: (1) if the deadlines for all jobs in S have past at time j, then $V(S, j) = 0$; and (2) otherwise, the optimal scheduling for $V(S, j)$ selects a next job a_i from S, then, according to the principle of optimality, the remaining part of $V(S, j)$ schedules the jobs in $S - \{a_i\}$in an optimal fashion given the starting time set at $j + t_i$; that is, $V(S, j) = \max_{a_i \in S,\, j+t_i \le d_i} \{p_i + V(S - \{a_i\}, j + t_i)\}$.

A sketch of a bottom-up implementation for computing $V(S, j)$ is given below:

```
(S1) declare a table $V(S, j)$ where $S \subseteq \{a_1, a_2, \ldots, a_n\}$ and $0 \le j \le d$, and $d = \sum_{k=1}^{n} t_k$ ,
        initialize $V(\emptyset, j) = 0$ for $0 \le j \le d$ and $V(S, d) = 0$ for $S \subseteq \{a_1, a_1, \ldots, a_n\}$.
(S2) for $k = 1$ to $n$ do
(S3)     for $j = 0$ to $d$ do
(S4)         for each subset $S \subseteq \{a_1, a_2, \ldots, a_n\}$ with $|S| = k$
(S5)             $V(S, j) = 0$
(S6)             for each job $a_i \in S$ do
(S7)                 if $j + t_i \le d_i$ and $p_i + V(S - \{a_i\}, j + t_i) > V(S, j)$ then
(S8)                     $V(i, j) = p_i + V(S - \{a_i\}, j + t_i)$
```

Note that the time complexity is $O(\sum_{k=1}^{n} (d+1)\binom{n}{k}k)$ because there are $\binom{n}{k}$ subsets in Step S4 to consider for each k, $1 \le k \le n$, and for each selected subset S the inner loop in Step S7 runs k times. Note that $\sum_{k=1}^{n} (d+1)\binom{n}{k}k = (d+1)\sum_{k=1}^{n} \binom{n-1}{k-1}n = n(d+1)2^{n-1} = O(n^3 2^n)$ since $d = \sum_{k=1}^{n} t_k \le n^2$ (because every processing time is $\le n$ by assumption). The space complexity is

$O(d2^n) = O(n^2 2^n)$ but can be improved to $O\left(d\binom{n}{\lfloor \frac{n}{2} \rfloor}\right) = O\left(n^2\binom{n}{\lfloor \frac{n}{2} \rfloor}\right) = O(n^{1.5}2^n)$ if only two rows of the table $V(S, j)$ are maintained. Finally, $V(\{a_1, a_2, \ldots, a_n\}, 0)$ is the desired optimal solution.

2. (10 pts.) Exercise 24-3 on p. 615 of Cormen et al.'s text, by modifying Floyd's all-pairs-shortest-paths algorithm.

   Answer: One way is to change the edge weight $w(u, v)$ to $-\lg w(u, v)$ then run Floyd's all-pairs-shortest-paths algorithm. Note that this transformation changes a path (i.e., sequence of edges) with a maximum product value of edge weights to a path of minimum sum of edge weights. When Floyd's algorithm terminates, for each vertex u, compute the minimum value of cycle-weight(u) = $d(u, v) + w(v, u)$, where vertex v ranges over all vertices of the digraph, $v \neq u$, and $d(u, v)$ represents the final shortest distance from u to v when the program terminates. If any of the values of cycle-weight(u) for a vertex u is negative, we have found a profitable sequence of currency exchanges (i.e., an arbitrage). The running time of the procedure is $O(n^3)$ (Floyd's algorithm) + $O(n^2)$ (computing a negative cycle) = $O(n^3)$.
3. (10 pts.) Read pages 623 – 625 of Cormen et al.'s text to understand another dynamic programming approach to solving the all-pairs-shortest-paths problem. Do Exercise 25-1-1 on p. 627 of the text. Relevant pages are included at the end of this assignment.
4. (10 pts.) Modify the solution to the Shuffle problem (see below) so that when Z is a shuffle of arrays X and Y, the output includes two arrays index[1..m+n] and source[1..m+n] in which index[i] records the index of symbol $Z[i]$ in its source array (either X or Y), and source[i] indicates which array (X or Y) where the symbol $Z[i]$ originated.

(The Shuffle problem and a dynamic programming solution) Suppose you are given three strings of characters:

X = x1 x2 ... xm;
Y = y1 y2 ... yn; and
Z = z1 z2 ... z(m+n).

String Z is said to be a shuffle of X and Y if Z can be formed by interspersing the characters from X and Y in a way that maintains the left-to-right ordering of the characters from each string.
For example, "cchocohilaptes" is a shuffle of "chocolate" and "chips", but "chocochilatspe" is not. Design a dynamic programming algorithm that takes input X, Y, Z, m, and n, and determines whether or not Z is a shuffle of X and Y. Analyze the worst-case time and space complexities of your algorithm.

Solution: We fist define the notation

Shuffle(i, j) = true or false, depending on if string Z[1..i+j]
is a shuffle of string X[1..i] and string Y[1..j].

where the strings X, Y, and Z are denoted by the array notations X[1..m], Y[1..n], and Z[1..m+n].

A recurrence for Shuffle(i. j) is as follows, where AND and OR are the standard logical operators:

```
If 1 <= i <= m and 1 <= j <= n, then
Shuffle(i, j) = (X[i] = Z[i+j] AND Shuffle(i-1, j)) OR
                (Y[j] = Z[i+j] AND Shuffle(i, j-1))
```

The explanation of the recurrence is as follows. When deciding if Z[1..i+j] is a shuffle of X[1..i] and Y[1..j], there are two possibilities to consider: if X[i] matches Z[i+j] and string Z[1..(i-1)+j] is a shuffle of strings X[1..i-1] and Y[1..j]; or, alternatively, if Y[j] matches Z[i+j] and string Z[1..i+(j-1)] is a shuffle of strings X[1..i] and Y[1..j-1]. The boundary conditions are:

```
Shuffle(0, j) = True if strings Y[1..j] and Z[1..j] match
                identically; False otherwise;
Shuffle(i, 0) = True if strings X[1..i] and Z[1..i] match
                identically; False otherwise.
```

A bottom-up implementation of the above recurrence uses a 2-dimensional table T[0..m, 0..n] to save the corresponding truth values for Shuffle(i, j), 0 <= i <= m and 0 <= j <= n. A pseudocode description of the algorithm is as follows:

```
1. set Shuffle[0, 0] = True

2. for j = 1 to n do
3.   if string Y[1..j] matches string Z[1..j] identically, then
       set Shuffle[0, j] to True
4.   else set Shuffle[0, j] to False

5. for i = 1 to m do
6.   if string X[1..i] matches string Z[1..i] identically, then
       set Shuffle[i, 0] to True
7.   else set Shuffle[i, 0] to False

8. for i = 1 to m do
9.   for j = 1 to n do
10.    set T[i, j] = (X[i] = Z[i+j] AND T(i-1, j)) OR
                     (Y[j] = Z[i+j] AND T(i, j-1))
```

The space complexity is O(m) if the most recent row of the table is kept when computing the next row. The time complexity of Steps 8-10 is O(mn) since there are O(mn) entries of the table to compute in order to find the final answer T[m, n], and each entry requires O(1) time. Also, Step 1 takes O(1) time; the initialization steps (1 through 4) can be done in O(n) time if implemented efficiently; similarly, steps 5-7 can be done in O(m) time. Thus, the total time complexity is O(mn).

15-7 Scheduling to maximize profit (P.369)

Suppose you have one machine and a set of n jobs $a_1, a_2, \ldots, a_n$ to process on that machine. Each job a_j has a processing time t_j, a profit p_j, and a deadline d_j. The machine can process only one job at a time, and job a_j must run uninterruptedly for t_j consecutive time units. If job a_j is completed by its deadline d_j, you receive a profit p_j, but if it is completed after its deadline, you receive a profit of 0. Give an algorithm to find the schedule that obtains the maximum amount of profit, assuming that all processing times are integers between 1 and n. What is the running time of your algorithm?

24-3 Arbitrage (P. 615)

Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 46.4 Indian rupees, 1 Indian rupee buys 2.5 Japanese yen, and 1 Japanese yen buys 0.0091 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $46.4 \times 2.5 \times 0.0091 = 1.0556$ U.S. dollars, thus turning a profit of 5.56 percent.

Suppose that we are given n currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j.

a. Give an efficient algorithm to determine whether or not there exists a sequence of currencies $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ such that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1 .$$

Analyze the running time of your algorithm.

b. Give an efficient algorithm to print out such a sequence if one exists. Analyze the running time of your algorithm.

25.1 Shortest paths and matrix multiplication

P. 622

This section presents a dynamic-programming algorithm for the all-pairs shortest-paths problem on a directed graph $G = (V, E)$. Each major loop of the dynamic program will invoke an operation that is very similar to matrix multiplication, so that the algorithm will look like repeated matrix multiplication. We shall start by developing a $\Theta(V^4)$-time algorithm for the all-pairs shortest-paths problem and then improve its running time to $\Theta(V^3 \lg V)$.

Before proceeding, let us briefly recap the steps given in Chapter 15 for developing a dynamic-programming algorithm.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.

(The fourth step, constructing an optimal solution from computed information, is dealt with in the exercises.)

The structure of a shortest path

p. 623

We start by characterizing the structure of an optimal solution. For the all-pairs shortest-paths problem on a graph $G = (V, E)$, we have proven (Lemma 24.1) that all subpaths of a shortest path are shortest paths. Suppose that the graph is represented by an adjacency matrix $W = (w_{ij})$. Consider a shortest path p from vertex i to vertex j, and suppose that p contains at most m edges. Assuming that there are no negative-weight cycles, m is finite. If $i = j$, then p has weight 0 and no edges. If vertices i and j are distinct, then we decompose path p into $i \overset{p'}{\leadsto} k \to j$, where path p' now contains at most $m - 1$ edges. By Lemma 24.1, p' is a shortest path from i to k, and so $\delta(i, j) = \delta(i, k) + w_{kj}$.

A recursive solution to the all-pairs shortest-paths problem

Now, let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges. When $m = 0$, there is a shortest path from i to j with no edges if and only if $i = j$. Thus,

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \,, \\ \infty & \text{if } i \neq j \,. \end{cases}$$

For $m \geq 1$, we compute $l_{ij}^{(m)}$ as the minimum of $l_{ij}^{(m-1)}$ (the weight of the shortest path from i to j consisting of at most $m-1$ edges) and the minimum weight of any path from i to j consisting of at most m edges, obtained by looking at all possible predecessors k of j. Thus, we recursively define

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n} \left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right) \\ &= \min_{1 \leq k \leq n} \left\{l_{ik}^{(m-1)} + w_{kj}\right\} \,. \end{aligned} \tag{25.2}$$

The latter equality follows since $w_{jj} = 0$ for all j.

What are the actual shortest-path weights $\delta(i, j)$? If the graph contains no negative-weight cycles, then for every pair of vertices i and j for which $\delta(i, j) < \infty$, there is a shortest path from i to j that is simple and thus contains at most $n - 1$ edges. A path from vertex i to vertex j with more than $n - 1$ edges cannot have lower weight than a shortest path from i to j. The actual shortest-path weights are therefore given by

$$\delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots \,. \tag{25.3}$$

Computing the shortest-path weights bottom up

Taking as our input the matrix $W = (w_{ij})$, we now compute a series of matrices $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$, where for $m = 1, 2, \ldots, n - 1$, we have $L^{(m)} = \left(l_{ij}^{(m)}\right)$.

The final matrix $L^{(n-1)}$ contains the actual shortest-path weights. Observe that $l_{ij}^{(1)} = w_{ij}$ for all vertices $i, j \in V$, and so $L^{(1)} = W$.

The heart of the algorithm is the following procedure, which, given matrices $L^{(m-1)}$ and W, returns the matrix $L^{(m)}$. That is, it extends the shortest paths computed so far by one more edge.

p. 624

```
EXTEND-SHORTEST-PATHS$(L, W)$
1  $n \leftarrow rows[L]$
2  let $L' = (l'_{ij})$ be an $n \times n$ matrix
3  for $i \leftarrow 1$ to $n$
4       do for $j \leftarrow 1$ to $n$
5              do $l'_{ij} \leftarrow \infty$
6                 for $k \leftarrow 1$ to $n$
7                     do $l'_{ij} \leftarrow \min(l'_{ij}, l_{ik} + w_{kj})$
8  return $L'$
```

The procedure computes a matrix $L' = (l'_{ij})$, which it returns at the end. It does so by computing equation (25.2) for all i and j, using L for $L^{(m-1)}$ and L' for $L^{(m)}$. (It is written without the superscripts to make its input and output matrices independent of m.) Its running time is $\Theta(n^3)$ due to the three nested **for** loops.

Now we can see the relation to matrix multiplication. Suppose we wish to compute the matrix product $C = A \cdot B$ of two $n \times n$ matrices A and B. Then, for $i, j = 1, 2, \ldots, n$, we compute

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} . \tag{25.4}$$

Observe that if we make the substitutions

$$\begin{aligned} l^{(m-1)} &\to a , \\ w &\to b , \\ l^{(m)} &\to c , \\ \min &\to + , \\ + &\to \cdot \end{aligned}$$

in equation (25.2), we obtain equation (25.4). Thus, if we make these changes to EXTEND-SHORTEST-PATHS and also replace ∞ (the identity for min) by 0 (the identity for $+$), we obtain the straightforward $\Theta(n^3)$-time procedure for matrix multiplication:

P. 625

```
MATRIX-MULTIPLY(A, B)
1  $n \leftarrow rows[A]$
2  let $C$ be an $n \times n$ matrix
3  for $i \leftarrow 1$ to $n$
4      do for $j \leftarrow 1$ to $n$
5             do $c_{ij} \leftarrow 0$
6                for $k \leftarrow 1$ to $n$
7                    do $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$
8  return $C$
```

Returning to the all-pairs shortest-paths problem, we compute the shortest-path weights by extending shortest paths edge by edge. Letting $A \cdot B$ denote the matrix "product" returned by EXTEND-SHORTEST-PATHS(A, B), we compute the sequence of $n-1$ matrices

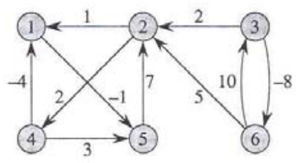
$$\begin{aligned} L^{(1)} &= L^{(0)} \cdot W &= W , \\ L^{(2)} &= L^{(1)} \cdot W &= W^2 , \\ L^{(3)} &= L^{(2)} \cdot W &= W^3 , \\ &\vdots \\ L^{(n-1)} &= L^{(n-2)} \cdot W &= W^{n-1} . \end{aligned}$$

As we argued above, the matrix $L^{(n-1)} = W^{n-1}$ contains the shortest-path weights. The following procedure computes this sequence in $\Theta(n^4)$ time.

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
1  $n \leftarrow rows[W]$
2  $L^{(1)} \leftarrow W$
3  for $m \leftarrow 2$ to $n-1$
4      do $L^{(m)} \leftarrow$ EXTEND-SHORTEST-PATHS$(L^{(m-1)}, W)$
5  return $L^{(n-1)}$
```

Figure 25.1 shows a graph and the matrices $L^{(m)}$ computed by the procedure SLOW-ALL-PAIRS-SHORTEST-PATHS.

25.1-1 (P. 627)
Run SLOW-ALL-PAIRS-SHORTEST-PATHS on the weighted, directed graph of Figure 25.2, showing the matrices that result for each iteration of the loop. ~~Then do the same for FASTER-ALL-PAIRS-SHORTEST-PATHS~~.



(P. 627)

Figure 25.2 A weighted, directed graph for use in Exercises 25.1-1, 25.2-1, and 25.3-1.