CS 102: SOLUTIONS TO DYNAMIC PROGRAMMING ALGORITHMS
FALL 2001: ASSIGNMENT 5)

Problem 1. Matrix Chain Multiplication Problem:
This problem and the solution has been discussed in the textbook and in the class. A succint description and the associated pseudo-code has been distributed in the class as well under the heading 8.1 (Iterated Matrix Product).

Please be aware that there are two problems. The first problem is to find the order in which to multiply the matrices to minimize the number of multiplications. The second problem is to find the least number of multiplications.

In this problem you are to design a dynamic programming algorithm for both the problems. Do the following for the first problem and then for the second problem. Describe the table and what does each entry in the table mean? How will the table be initialized? In which order the table will be filled? What is the recurrence? How will you use the table to find the order of multiplications (for the first problem) and the actual number of multiplications (for the second problem)? Compute the asymptotic complexity of the algorithms. It is very important that you practice writing your own solutions to this problem even though you may have perfect understanding of the solution.

Practice the solution to the above problem when there are 4 matrices with order 2×5, 5×4, 4×1, and 1×10. (In actual quiz, these numbers may vary).

Solution. The solution to this problem is discussed in the textbook. The solution to the specific example will be presented in the class. ■

Problem 2. 0-1 Knapsack Problem:
Consider the 0-1 knapsack problem: A thief robbing a store finds n items: the ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. For every item, the thief has to make a binary choice: whether to take the item or leave it. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integer W.

There are two problems. First, what is the value of the most valuable load he can steal? Second, what items he should take?

You are to design a dynamic programming algorithm for both the problems. Describe the table and what does each entry in the table mean? How will the table be initialized? In which order the table will be filled? What is the recurrence? How will you use the table to find what is the total value of the goods stolen (for the first problem) and which goods should be stolen (for the second problem)? Compute the asymptotic complexity of the algorithms. It is very important that you practice writing your own solutions to this problem even though you may have perfect understanding of the solution.

Practice the above problem when you are stealing 4 items worth 2, 4, 5, and 3 dollars and weighing 1, 2, 3 and 1 lbs. and the weight of the thief's bag is 4 lbs. (In actual quiz, these numbers may vary).

Solution. Solution to this problem was distributed in the class. Please note that the dimensions of the table used should be $(W+1) \times (n+1)$ rather than $W \times n$ with the first column initialized to 0. ■

Problem 3. Coin Changing Problem:
Given the k coin values $c_0 < c_1 < c_2 < \cdots < c_{k-1}$ (where $c_0 = 1$), and a value v, find a way to give the value v in change using as few coins as possible (sufficient coins of each denomination are available).

There are two problems. First, what is the minimum number of coins required for the change? Second, how many coins of each type will be given for this change?

You are to design a dynamic programming algorithm for both the problems. Describe the table and what does each entry in the table mean? How will the table be initialized? In which order the table will be filled? What is the recurrence? How will you use the table to find the minimum number of coins (for the first problem) and how many coins of each type (for the second problem)? Compute the asymptotic complexity of the algorithms. It is very important that you practice writing your own solutions to this problem even though you may have perfect understanding of the solution.

Practice the above problem with coin denominations 1, 3, and 4 seeking change for 6. (In actual quiz, these numbers may vary).

Solution. Solution to this problem was distributed in the class. Please note that the dimensions of the table used should be $1 \times (C+1)$ rather than $1 \times C$ with the first entry initialized to 0.

Also, the solution to the example has errors in the 4th column as discussed in the class. These numbers should be 1 and 3 rather than 2 and (4 or 1). ■

Problem 4. Canoe Rental Problem:
There are n trading posts numbered 1 to n as you travel downstream. At any trading post i you can rent a canoe to be returned at any of the downstream trading posts $j > i$. You are given a cost array $R(i,j)$ giving the cost of these rentals for all $1 \leq i < j \leq n$. We can assume that $R(i,i) = 0$, and that you can't go upriver (so perhaps $R(i,j) = \infty$ if $i > j$). For example, one cost array with $n = 4$ might be the following.

C		to j			
		1	2	3	4
	1	0	2	3	7
from	2	–	0	2	4
i	3	–	–	0	2
	4	–	–	–	0

The problem is to find a dynamic programming algorithm that computes the cheapest sequence of rentals taking you from post 1 all the way down to post n. In this example, the cheapest way is to rent canoes from post 1 to post 3, and then from post 3 to post 4 for a total cost of 5. The second problem is to find the least cost associated with this sequence.

You are to design a dynamic programming algorithm for both the problems. Describe the table and what does each entry in the table mean? How will the table be initialized? In which order the table will be filled? What is the recurrence? How will you use the table to find what is the cheapest sequence of canoe rentals (for

the first problem) and the least cost of the canoe rentals (for the second problem)? Compute the asymptotic complexity of the algorithms. It is very important that you practice writing your own solutions to this problem even though you may have perfect understanding of the solution.

Solution to this problem has been provided on the web.

Practice the above problem with numbers given in the table. (In actual quiz, these numbers may vary).

Solution. First we will define a function *Cheap* that gives the cost of cheapest sequences of canoe rentals, and then we will find a recurrence for *Cheap*.

We define, for $1 \leq k \leq n$, the function *Cheap*(k) to be the cheapest cost of canoe rentals from trading post 1 to k. $R(i, j)$ represents the cost of renting a canoe at node i and return it at j, with no intermediate rentals.

$$\text{Cheap}(k) = \begin{cases} 0 & \text{if } k = 1 \\ \min_{1 \leq i < k}\{\text{Cheap}(i) + R(i,k)\} & \text{otherwise} \end{cases} \tag{1}$$

This recurrence can be justified by considering an optimal (cheapest) sequence of canoe rentals ending at some post $k > 1$. This sequence must contain a last canoe rental from some other post i to post k. The cost of this optimal sequence is then the cost of an optimal rental sequence from post 1 to post i, plus the cost of the last rental form post i to post j. Since all possibilities for this last rental are considered, the recurrence correctly computes the costs of optimal rental sequences.

Note that it would be an error to define Cheap(k) using $\min_{1 \leq i \leq k}\{\text{Cheap}(i) + R(i,k)\}$ as this would be defining Cheap(k) in terms of itself.

Second, describe an appropriate table for remembering previously computed values in order to reduce the work done by the straightforward recursive algorithm for computing the values of *cheap*.

We will keep an one-dimensional array $C[1 \ldots n]$ where $C[k] =$*Cheap*(k).

Third, we now outline an iterative procedure for filling in the table.

$C[1 \ldots n]$ can be filled in in the following fashion:

```
procedure Cost(R[1...n][1...n])
begin
(1)  C[1]=0
(2)  for i=2 to n do            /*  compute C[i] */
(3)    C[i] = C[i-1] + R(i-1,i)
(4)    for j=i-2 down to 1 do
(5)      if C[j] + R(j,i) < C[i] then
(6)        C[i] = C[j] + R(j,i)
(7)      end do
(8)    end do
(9)  end do
end
```

lines (3)–(8) compute the *minimum* over last rentals in the recurrence.

Fourth, we now indicate how to find an optimal sequence of canoe rentals taking you from post 1 to post n using your table.

Create a second table $L[1 \ldots n]$ (for *L*ast rental), and record in this table the trading post which gives the minimum (as in the recurrence). This can be accomplished by adding the following lines to the above procedure:

```
(3.5)  L[i] = i-1
(6.5)  L[i] = j      (add inside the ''if'')
```

Therefore, we have that $L[k]$ is where the last rental is made on the path from 1 to k. From this table, the *path* from 1 to k can be recovered by traversing $L[1 \ldots n]$ in the following manner:

```
procedure Path(n)
begin
(1) if n != 1 then Path(L[n])
(2) Print L[n]
end
```

The general idea behind the above code is that once we find a last vertex $\ell = L[n]$ on the path from 1 to n we next need to find the last vertex on the path from 1 to ℓ.

Finally, we now give a Θ-expressions for the running times of solutions to the third and fourth parts as functions of n, the number of trading posts.

In order to fill in each location of $C[k]$, and subsequently $L[k]$, the procedure *Cost* must compute the maximum of $k-1$ things (lines 4-7). Summing up for each of the n table locations (and adding 1 unit of time for initializing $C[1]$ and $L[1]$) we have that filling in these tables takes time:

$$1 + \sum_{i=2}^{n} i \in \Theta(n^2).$$

In order to recover shortest path from $L[1 \ldots n]$, *Path* may have to report at most n nodes (each edge corresponding to a constant-time recursive call). Therefore, recovering the path information from $N[1 \ldots n]$ takes time in $O(n)$. (The worst case will be $\Theta(n)$ when each rental goes only one trading post, but it might take $o(n)$ time if the optimal sequence uses fewer rentals.)

■

Problem 5. Optimal Binary Search Trees:
The problem is described in a hard copy handout titled 8.3 (Optimal Binary Search Trees).

Solution. The solution is described in a hard copy handout titled 8.3 (Optimal Binary Search Trees). ■

Problem 6. Partition problem:

Given a sequence n positive integers, $k_1, k_2, \ldots, k_n$, that sum to s (you can assume that s is even), find a subset I of $\{1, \ldots, n\}$ such that

$$\sum_{i \in I} k_i = \sum_{i \notin I} k_i = s/2$$

or determine that there is no such subset.

You are to find a dynamic programming solution to this problem.

Solution. First, define and give a recurrence for a function that can be used to determine if such a set I exists.

Consider the boolean function *CanGet*(j, g) which is TRUE (1) if there is a subsequence of $k_1, \ldots, k_j$ whose elements sum to g (goal value), and FALSE (0) otherwise. One can interpret *CanGet*(j, g) as *CanGet*$(\langle k_1, \ldots k_j \rangle, g)$ since the j encodes the last k-value considered. Initially we are interested in *CanGet*$(n, s/2)$. Using this notation we derive the following recurrence for the Partition problem.

(2)

$$\text{CanGet}(j,g) = \begin{cases} 1 & \text{if } g = 0 \\ 0 & \text{if } g < 0 \\ 0 & \text{if } g > 0 \text{ and } k < 1 \\ \text{CanGet}(j-1, g-k_j) \text{ or } \text{CanGet}(j-1, g) & \text{otherwise} \end{cases}$$

The (most important) last line in the above recurrence can was derived using the following key fact: if there is a subsequence of $k_1, \ldots, k_j$ whose elements sum to g, then that subsequence either contains the k_j, or it does not. Furthermore, *if* no subsequence of $k_1, \ldots, k_{j-1}$ sums to g *and* no subsequence of $k_1, \ldots, k_{j-1}$ sums to $g - k_j$, *then* no subsequence of $k_1, \ldots, k_j$ sums to g.

Second, describe an appropriate table for adding memoization to the straightforward recursive algorithm for computing your function.

The table which can be used for *memoization* for the above recurrence is two-dimensional (one dimension for each of the arguments to *CanGet*). This table has rows which correspond to values of j (or subsequences $\langle k_1, \ldots, k_j \rangle$), and columns which correspond to goal values from 1 to $s/2$. In summary, we use the table $P[0 \ldots n][0 \ldots s/2]$ to keep track of the solutions to the recursive subproblems.

Third, outline an iterative procedure for filling in the table.

Using the above recurrence, this table can be filled in (bottom up) in the following manner:

```
procedure CanGet(m,s)
begin
(1)  P[j][0] = 1   // for all j>=0 (every column)
(2)  P[0][v] = 0   // for all v>=1 (every row, note P[0][0]=1)
(3)  for j=1 to m do
(4)    for v=1 to s/2 do
(5)      if (v-k_j < 0) then
(6)         P[j][v] = P[j-1][v]
(7)      else
(8)         P[j][v] = ( P[j-1][v] or P[j-1][v-k_j] )
(9)    end do
(10) end do
```

We could optimize the above procedure by noticing that we could stop filling in the table if ever we get a TRUE value in $P[i][s/2]$ for

any value of i, meaning that we have found a subset of S which sums to $s/2$. We might further notice that in order to fill in any row in the table only the immediately previous row is needed, and therefore if we are clever we only really need a one-dimensional table. (Note to readers – these optimizations are not required for full credit.)

Fourth, indicate how to find an actual subset I from your table (assuming one exists).

We can find the actual subset I from our table if we keep track of more information at each table location. For each location `P[j][g]` in the table which has the value 1 (TRUE) we would like to remember whether we got that 1 by adding in the k_j value or whether there was a subsequence not including k_j that summed to g. We can keep track of this by storing a second bit at each table location that is 1 when k_j was added in and 0 if it was not included. Alternatively, we might keep a separate table $K[1 \ldots n][1 \ldots s/2]$, in which these signposts are kept. With this new information we can recover the subset (should one exist) by tracing back from $K[n, s/2]$, taking k_n and continuing that trace at $K[n-1, s/2-k_n]$ if it is 1 and not taking k_n and continuing the trace at $K[n-1, s/2]$ if it is 0. By repeating this process we will end at $K[0,0]$ after finding a subsequence of the elements adding up to $s/2$. (Note: we need a two-dimensional table for $K[j,g]$ even if we use the storage optimization for the $P[j][g]$ table.

Finally, Give Θ-expressions for the running times of your solutions to the third and fourth parts as functions of s (the sum of the integers) and n.

In the worst case, we will have to fill in the entire $n * s/2$ dimension table. However, there is only a constant amount of work required to fill in each table location, as described in the algorithm above. Therefore the time required to fill in the table is $\Theta(ns)$. The time required to recover the actual subset, if one exists is in $\Theta(n)$, as there are at most n traceback steps each taking constant time.

■