DP Practice Problems

1. (True/False) The running time of a dynamic programming algorithm is always $\Theta(P)$ where P is the number of subproblems.

Solution: False. The running time of a dynamic program is the number of subproblems times the time per subproblem. This would only be true if the time per subproblem is $O(1)$.

2. You just started a consulting business where you collect a fee for completing various types of projects (the fee is different for each project). You can select in advance the projects you will work on during some finite time period. You work on only one project at a time and once you start a project it must be completed to receive your fee. There is a set of n projects $p_1, p_2, .. p_n$ each with a duration $d_1, d_2, .. d_n$ (in days) and you receive the fee $f_1, f_2, .., f_n$ (in dollars) associated with it. That is project p_i takes d_i days and you collect f_i dollars after it is completed.

Each of the n projects must be completed in the next D days or you lose its contract. Unfortunately, you do not have enough time to complete all the projects. Your goal is to select a subset S of the projects to complete that will maximize the total fees you earn in D days.

(a) What type of algorithm would you use to solve this problem? Divide and Conquer, Greedy or **Dynamic Programming**. Why? **It is similar to the 0-1 knapsack.**

(b) Describe the algorithm verbally. If you select a DP algorithm give the formula used to fill the table or array.

Let OPT(i,d) be the maximum fee collected for considering projects 1,…, i with d days available.

The base cases are OPT(i, 0) = 0 for i = 1, …, n and OPT(0, d) = 0 for d = 1, …, D.

```
For i = 1 to n {
  For d = 1 to D {
        If (di > d ) {
              OPT(i, d) = OPT( i-1, d)    // not enough time to complete project i }
        Else {
              OPT(i, d) = max ( OPT( i-1, d),    // Don't complete project i
                                OPT( i-1, d-di) + fi  ) //  Complete project I and earn fee fi
                              )
        }
  }
}
```

(c) What is the running time of your algorithm?

Θ(nD)

DP Practice Problems

3. Product Sum

Given a list of n integers, $v_1, \ldots, v_n$, the product-sum is the largest sum that can be formed by multiplying adjacent elements in the list. Each element can be matched with at most one of its neighbors.

For example, given the list 1, 2, 3, 1 the product sum is 8 = 1 + (2 × 3) + 1, and given the list 2, 2, 1, 3, 2, 1, 2, 2, 1, 2 the product sum is 19 = (2 × 2) + 1 + (3 × 2) + 1 + (2 × 2) + 1 + 2.

Solutions:

a) 1 + (4x3) + 2 + (3x4) + 2 = 29

b) OPT[j] = $\begin{cases} 0 & if\ j = 0 \\ v_1 & if\ j = 1 \\ \max\{\ \mathrm{OPT}[j-1] + v_j, \mathrm{OPT}[j-2] + v_j * v_{j-1}\} \end{cases}$

c) running time Θ(n)

DP Practice Problems

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, where $j \geq i$. You are given an array R[i, j] defining the costs of a canoe which is picked up at post i and dropped off at post j, for $1 \leq i \leq j \leq n$. Assume that R[i,i] = 0 and that you can't take a canoe upriver. Your problem is to determine a sequence of rentals which start at post 1 and end at post n, and that has the minimum total cost.*

(a) Describe verbally and give pseudo code for a DP algorithm to compute the cost of the cheapest sequence of canoe rentals from trading post 1 to n. Give the recursive formula you used to fill in the table or array.

Define a 1 dimensional table *C*[1…*n*] where $C[i]$ is the cost of an optimal sequence of canoe rentals that starts at post 1 and ends at post *i*, for $1 \leq i \leq n$. When this table is filled, we simply return the value $C[n]$. *Note could use a different variable name for C[n].*

Clearly $C[1] = 0$.

Define $C[i]$ in terms of earlier table entries. Indeed its clear that $C[i] = C[k] + R[k,i]$. Since we do not know the post *k* beforehand, we take the minimum of this expression over all *k* in the range $1 \leq k < i$. Define

$$C[i] = \begin{cases} 0 & i = 1 \\ \min\limits_{1 \leq k < i} \left(C[k] + R[k,i] \right) & 1 < i \leq n \end{cases}$$

With this formula, the algorithm for filling in the table is straightforward.

<u>Below is a strictly bottom-up approach. Give full credit for memorized DP.</u>

<u>CanoeCost(*R*)</u>

1. $n \leftarrow \#\text{rows}[R]$
2. $C[1] \leftarrow 0$
3. for $i \leftarrow 2$ to *n*
4. $\quad \min \leftarrow R[1,i]$
5. $\quad$ for $k \leftarrow 2$ to $i-1$
6. $\quad\quad$ if $C[k] + R[k,i] < \min$
7. $\quad\quad\quad \min \leftarrow C[k] + R[k,i]$
8. $\quad C[i] \leftarrow \min$
9. return $C[n]$

b) Print Sequence

Explanation
To determining the actual sequence of canoe rentals which minimizes cost alter the CanoeCost() algorithm so as to construct the optimal sequence while the table *C*[1…*n*] is being filled. In the following algorithm we maintain an array *P*[1…*n*] where $P[i]$ is defined to be the post *k* at which

the last canoe is rented in an optimal sequence from 1 to *i*. Note that the definition of $P[1]$ can be arbitrary since it is never used. Array *P* is then used to recursively print out the sequence.

CanoeSequence(*R*)

```
1.  n ← #rows[R]
2.  C[1] ← 0 , P[1] ← 0
3.  for i ← 2 to n
4.      min ← R[1,i]
5.      P[i] ← 1
6.      for k ← 2 to i − 1
7.          if C[k] + R[k,i] < min
8.              min ← C[k] + R[k,i]
9.              P[i] ← k
10.     C[i] ← min
11. return P
```

PrintSequence(*P, i*) (Pre: $1 \le i \le \text{length}[P]$)

```
1.  if i > 1
2.      PrintSequence(P, P[i])
3.      print "Rent a canoe at post " P[i] " and drop it off at post " i
```

c) What is the running time of your algorithm to find the minimum cost and to find the sequence?

CanoeCost() runs in time $\Theta(n^2)$, since the inner for loop performs $i-2$ comparisons in order to determine $C[i]$, and $\sum_{i=2}^{n}(i-2) = \frac{(n-1)(n-2)}{2} = \Theta(n^2)$.

The top level call to PrintSequence(*P*, *n*) has a cost that is the depth of the recursion, which is in turn, the number of canoes rented in the optimal sequence from post 1 to post *n*. Thus in worst case, PrintSequence() runs in time $\Theta(n)$.