Modification of CSC 364S Notes University of Toronto, Fall 2003

Dynamic Programming Algorithms

The setting is as follows. We wish to find a solution to a given problem which optimizes some quantity Q of interest; for example, we might wish to maximize profit or minimize cost. The algorithm works by *generalizing* the original problem. More specifically, it works by creating an array of related but simpler problems, and then finding the optimal *value* of Q for each of these problems; we calculate the values for the more complicated problems by using the values already calculated for the easier problems. When we are done, the optimal value of Q for the original problem can be easily computed from one or more values in the array. We then use the array of values computed in order to compute a *solution* for the original problem that attains this optimal value for Q. We will always present a dynamic programming algorithm in the following 4 steps.

Step 1:
Describe an array (or arrays) of values that you want to compute. (Do not say how to compute them, but rather describe what it is that you want to compute.) Say how to use certain elements of this array to compute the optimal value for the original problem.

Step 2:
Give a recurrence relating some values in the array to other values in the array; for the simplest entries, the recurrence should say how to compute their values from scratch. Then (unless the recurrence is obviously true) justify or prove that the recurrence is correct.

Step 3:
Give a high-level program for computing the values of the array, using the above recurrence. Note that one computes these values in a bottom-up fashion, using values that have already been computed in order to compute new values. (One does not compute the values recursively, since this would usually cause many values to be computed over and over again, yielding a very inefficient algorithm.) Usually this step is very easy to do, using the recurrence from Step 2. Sometimes one will also compute the values for an auxiliary array, in order to make the computation of a solution in Step 4 more efficient.

Step 4:
Show how to use the values in the array(s) (computed in Step 3) to compute an optimal solution to the original problem. Usually one will use the recurrence from Step 2 to do this.

Moving on a grid example

The following is a very simple, although somewhat artificial, example of a problem easily solvable by a dynamic programming algorithm.

Imagine a climber trying to climb on top of a wall. A wall is constructed out of square blocks of equal size, each of which provides one handhold. Some handholds are more dangerous/complicated than other. From each block the climber can reach three blocks of the row righ above: one right on top, one to the right and one to the left (unless right or left are no available because that is the end of the wall). The goal is to find the least dangerous path from the bottom of the wall to the top, where danger rating (cost) of a path is the sum of danger ratings (costs) of blocks used on that path.

We represent this problem as follows. The input is an $n \times m$ grid, in which each cell has a positive cost $C(i, j)$ associated with it. The bottom row is row 1, the top row is row n. From a cell (i, j) in one step you can reach cells $(i+1, j-1)$ (if $j > 1$), $(i+1, j)$ and $(i+1, j+1)$ (if $j < m$).

Here is an example of an input grid. The easiest path is highlighted. The total cost of the easiest path is 12. Note that a greedy approach – choosing the lowest cost cell at every step – would not yield an optimal solution: if we start from cell $(1, 2)$ with cost 2, and choose a cell with minimum cost at every step, we can at the very best get a path with total cost 13.

Grid example.

2	8	9	**5**	8
4	4	6	**2**	3
5	7	5	6	**1**
3	2	5	**4**	8

Step 1. The first step in designing a dynamic programming algorithm is defining an array to hold intermediate values. For $1 \leq i \leq n$ and $1 \leq j \leq m$, define $A(i, j)$ to be the cost of the cheapest (least dangerous) path from the bottom to the cell (i, j). To find the value of the best path to the top, we need to find the minimal value in the last row of the array, that is, $\min_{1 \leq j \leq m} A(n, j)$.

Step 2. This is the core of the solution. We start with the initialization. The simplest way is to set $A(1, j) = C(1, j)$ for $1 \leq j \leq m$. A somewhat more elegant way is to make an additional zero row, and set $A(0, j) = 0$ for $1 \leq j \leq m$.
There are three cases to the recurrence: a cell might be in the middle (horizontally), on the leftmost or on the rightmost sides of the grid. Therefore, we compute $A(i, j)$ for $1 \leq i \leq n$, $1 \leq j \leq m$ as follows:

$A(i, j)$ for the above grid.

∞	0	0	0	0	0	∞
∞	3	2	5	**4**	8	∞
∞	7	9	7	10	**5**	∞
∞	11	11	13	**7**	8	∞
∞	13	19	16	**12**	15	∞

$$A(i,j) = \begin{cases} C(i,j) + \min\{A(i-1,j-1), A(i-1,j)\} & \text{if } j = m \\ C(i,j) + \min\{A(i-1,j), A(i-1,j+1)\} & \text{if } j = 1 \\ C(i,j) + \min\{A(i-1,j-1), A(i-1,j), A(i-1,j+1)\} & \text{if } j \neq 1 \text{ and } j \neq m \end{cases}$$

We can eliminate the cases if we use some extra storage. Add two columns 0 and $m+1$ and initialize them to some very large number ∞; that is, for all $0 \leq i \leq n$ set $A(i,0) = A(i,m+1) = \infty$. Then the recurrence becomes, for $1 \leq i \leq n$, $1 \leq j \leq m$,

$$A(i,j) = C(i,j) + \min\{A(i-1,j-1), A(i-1,j), A(i-1,j+1)\}$$

Step 3 . Now we need to write a program to compute the array; call the array B. Let INF denote some very large number, so that $INF > c$ for any c occurring in the program (for example, make INF the sum of all costs +1).

```
// initialization
for j = 1 to m do
    B(0, j) ← 0
for i = 0 to n do
    B(i, 0) ← INF
    B(i, m + 1) ← INF
// recurrence
for i = 1 to n do
    for j = 1 to m do
        B(i, j) ← C(i, j) + min{B(i − 1, j − 1), B(i − 1, j), B(i − 1, j + 1)}
// finding the cost of the least dangerous path
cost ← INF
for j = 1 to m do
    if (B(n, j) < cost) then
        cost ← B(n, j)
return cost
```

Step 4. The last step is to compute the actual path with the smallest cost. The idea is to retrace the decisions made when computing the array. To print the cells in the correct order, we make the program recursive. Skipping finding j such that $A(n,j) = cost$, the first call to the program will be $PrintOpt(n,j)$.

```
procedure PrintOpt(i,j)
    if (i = 0) then return
    else if (B(i, j) = C(i, j) + B(i − 1, j − 1)) then PrintOpt(i-1,j-1)
```

else if $(B(i,j) = C(i,j) + B(i-1,j))$ **then** PrintOpt(i-1,j)
else if $(B(i,j) = C(i,j) + B(i-1,j+1))$ **then** PrintOpt(i-1,j+1)
end if
put "Cell " (i,j)
end PrintOpt

Longest Common Subsequence

The input consists of two sequences $\vec{x} = x_1, \ldots, x_n$ and $\vec{y} = y_1, \ldots, y_m$. The goal is to find a longest common subsequence of $\vec{x}$ and $\vec{y}$, that is a sequence $z_1, \ldots, z_k$ that is a subsequence both of $\vec{x}$ and of $\vec{y}$. Note that a sub*sequence* is not always sub*string*: if $\vec{z}$ is a subsequence of $\vec{x}$, and $z_i = x_j$ and $z_{i+1} = x_{j'}$, then the only requirement is that $j' > j$, whereas for a sub*string* it would have to be $j' = j + 1$.

For example, let $\vec{x}$ and $\vec{y}$ be two DNA strings $\vec{x} = TGACTA$ and $\vec{y} = GTGCATG$; $n = 6$ and $m = 7$. Then one common subsequence would be GTA. However, it is not the longest possible common subsequence: there are common subsequences $TGCA$, $TGAT$ and $TGCT$ of length 4.

To solve the problem, we notice that if $x_1 \ldots x_i$ and $y_1 \ldots y_j$ are prefixes of $\vec{x}$ and $\vec{y}$ respectively, and $x_i = y_j$, then the length of the longest common subsequence of $x_1 \ldots x_i$ and $y_1 \ldots y_j$ is one plus the length of the longest common subsequence of $x_1 \ldots x_{i-1}$ and $y_1 \ldots y_{j-1}$.

Step 1. We define an array to hold partial solution to the problem. For $0 \leq i \leq n$ and $0 \leq j \leq m$, $A(i,j)$ is the length of the longest common subsequence of $x_1 \ldots x_i$ and $y_1 \ldots y_j$. After the array is computed, $A(n,m)$ will hold the length of the longest common subsequence of $\vec{x}$ and $\vec{y}$.

Step 2. At this step we initialize the array and give the recurrence to compute it.

For the initialization part, we say that if one of the two (prefixes of) sequences is empty, then the length of the longest common subsequence is 0. That is, for $0 \leq i \leq n$ and $0 \leq j \leq m$, $A(i,0) = A(0,j) = 0$.
The recurrence has two cases. The first is when the last element in both subsequences is the same; then we count that element as part of the subsequence. The second case is when they are different; then we pick the largest common sequence so far, which would not have either x_i or y_j in it. So, for $1 \leq i \leq n$ and $1 \leq j \leq m$,

$A(i,j)$ for the above example.

	∅	G	T	G	C	A	T	G
∅	0	0	0	0	0	0	0	0
T	0	0	1	1	1	1	1	1
G	0	1	1	2	2	2	2	2
A	0	1	1	2	2	3	3	3
C	0	1	1	2	3	3	3	3
T	0	1	2	2	3	3	4	4
A	0	1	2	2	3	4	4	4

$$A(i,j) = \begin{cases} A(i-1,j-1)+1 & \text{if } x_i = y_j \\ \max\{A(i-1,j), A(i,j-1)\} & \text{if } x_i \neq y_j \end{cases}$$

Step 3. Skipped.

Step 4. As before, just retrace the decisions.

Longest Increasing Subsequence

Now let us consider a simpler version of the LCS problem. This time, our input is only one sequence of distinct integers $\vec{a} = a_1, a_2, \ldots, a_n.$, and we want to find the longest *increasing* subsequence in it. For example, if $\vec{a} = 7, 3, 8, 4, 2, 6$, the longest increasing subsequence of $\vec{a}$ is $3, 4, 6$.

The easiest approach is to sort elements of $\vec{a}$ in increasing order, and apply the LCS algorithm to the original and sorted sequences. However, if you look at the resulting array you would notice that many values are the same, and the array looks very repetitive. This suggest that the LIS (longest increasing subsequence) problem can be done with dynamic programming algorithm using only one-dimensional array.

Step 1: Describe an array of values we want to compute.
For $1 \leq i \leq n$, let $A(i)$ be the length of a longest increasing sequence of $\vec{a}$ that end with a_i. Note that the length we are ultimately interested in is $\max\{A(i) \mid 1 \leq i \leq n\}$.

Step 2: Give a recurrence.
For $1 \leq i \leq n$,
$A(i) = 1 + \max\{A(j) \mid 1 \leq j < i \text{ and } a_j < a_i\}$.
(We assume $\max \emptyset = 0$.)
We leave it as an exercise to explain why, or to prove that, this recurrence is true.
Step 3: Give a high-level program to compute the values of A.
This is left as an exercise. It is not hard to design this program so that it runs in time $O(n^2)$. (In fact, using a more fancy data structure, it is possible to do this in time $O(n \log n)$.)

LCS and LIS arrays for the example

A(i,j)	∅	7	3	8	4	2	6
∅	0	0	0	0	0	0	0
2	0	0	0	0	0	**1**	1
3	0	0	**1**	1	1	1	1
4	0	0	1	1	**2**	2	2
6	0	0	1	1	2	2	**3**
7	0	**1**	1	1	2	2	3
8	0	1	1	**2**	2	2	3

$A(i)$		**1**	**1**	**2**	**2**	**1**	**3**

Step 4: Compute an optimal solution.
The following program uses A to compute an optimal solution. The first part computes a value m such that $A(m)$ is the length of an optimal increasing subsequence of $\vec{a}$. The second part computes an optimal increasing subsequence, but for convenience we print it out in reverse order. This program runs in time $O(n)$, so the entire algorithm runs in time $O(n^2)$.

```
m ← 1
for i : 2..n
    if A(i) > A(m) then
        m ← i
    end if
end for
```

```
put a_m
while A(m) > 1 do
    i ← m − 1
    while not(a_i < a_m and A(i) = A(m) − 1) do
        i ← i − 1
    end while
    m ← i
    put a_m
end while
```