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

The (General) Knapsack Problem

First, recall the Simple Knapsack Problem: We are given a sequence of nonnegative integer weights $w_1, \cdots, w_n$ and a nonnegative integer capacity C, and we wish to find a subset of the weights that adds up to as large a number as possible without exceeding C.

The general knapsack problem is as follows. Let $w_1, \cdots, w_n \in \mathbb{N}$ be weights, let $g_1, \cdots, g_n \in \mathbb{R}^{\geq 0}$ be profits, and let $C \in \mathbb{N}$ be a weight. For each $S \subseteq \{1, \cdots, n\}$ let $K(S) = \sum_{i \in S} w_i$ and let $P(S) = \sum_{i \in S} g_i$. (Note that $K(\emptyset) = P(\emptyset) = 0$.) We call $S \subseteq \{1, \cdots, n\}$ *feasible* if $K(S) \leq C$.
The goal is to find a feasible S so that $P(S)$ is as large as possible.

We can view a Simple Knapsack Problem as a special case of a General Knapsack Problem, where for every i, $g_i = w_i$.

We will use a dynamic programming algorithm.
Step 1: Describe an array of values we want to compute.
For $0 \leq i \leq n$ and $0 \leq w \leq C$, define $A(i, w) = \max\{P(S) \mid S \subseteq \{1, \cdots, i\} \text{ and } K(S) \leq w\}$.

Steps 2,3,4: Exercise.

All pairs Shortest Path Problem

We define a *directed graph* to be a pair $G = (V, E)$ where V is a set of vertices and $E \subseteq V \times V$ is a set of (directed) edges. Sometimes we will consider *weighted* graphs where associated with each edge (i, j) is a weight (or cost) $c(i, j)$. A (directed) path in G is a sequence of one or more vertices $v_1, \cdots, v_m$ such that $(v_i, v_{i+1}) \in E$ for every i, $1 \leq i < m$; we say this is a path from v_1 to v_m. The cost of this path is defined to be the sum of the costs of the $m - 1$ edges in the path; if $m = 1$ (so that the path consists of a single node and no edges) then the cost of the path is 0.

Given a directed, weighted graph, we wish to find, for every pair of vertices u and v, the cost of a cheapest path from u to v. This should be called the "all pairs cheapest path problem", and that is how we will refer to it from now on, but traditionally it has been called the "all pairs shortest path problem". We will give the Floyd-Warshall dynamic programming algorithm for this problem.

Let us assume that $V = \{1, \cdots n\}$, and that *all* edges are present in the graph. We are given n^2 costs $c(i, j) \in \mathbb{R}^{\geq 0} \cup \{\infty\}$ for every $1 \leq i, j \leq n$. Note that our costs are either nonnegative real numbers or the symbol "∞". We don't allow negative costs, since then cheapest paths might not exist: there might be arbitrarily small negative-cost paths from one vertex to another. We allow ∞ as a cost in order to denote that we really view that edge as not existing. (We do arithmetic on ∞ in the obvious way: ∞ plus ∞ is ∞, and ∞ plus any real number is ∞.)

For $1 \leq i, j \leq n$, define $D(i, j)$ to be the cost of a cheapest path from i to j. Our goal is to compute *all* of the values $D(i, j)$.

Step 1: Describe an array of values we want to compute.
For $0 \leq k \leq n$ and $1 \leq i, j \leq n$, define
$A(k, i, j)$ = the cost of a cheapest path from i to j, from among those paths from i to j whose *intermediate* nodes are all in $\{1, \cdots, k\}$. (We define the intermediate nodes of the path $v_1, \cdots, v_m$ to be the set $\{v_2, \cdots, v_{m-1}\}$; note that if m is 1 or 2, then this set is empty.)

Note that the values we are ultimately interested in are the values $A(n, i, j)$ for all $1 \leq i, j \leq n$.

Step 2: Give a recurrence.

- If $k = 0$ and $i = j$, then $A(k, i, j) = 0$.
  If $k = 0$ and $i \neq j$, then $A(k, i, j) = c(i, j)$.

  This part of the recurrence is obvious, since a path with *no* intermediate nodes can only consist of 0 or 1 edge.

- If $k > 0$, then $A(k, i, j) = \min\{A(k-1, i, j), A(k-1, i, k) + A(k-1, k, j)\}$.

The reason this equation holds is as follows. We are interested in cheapest paths from i to j whose intermediate vertices are all in $\{1, \cdots, k\}$. Consider a cheapest such path p. If p doesn't contain k as an intermediate node, then p has cost $A(k-1, i, j)$; if p does contain k as an intermediate node, then (since costs are nonnegative) we can assume that k occurs only once as an intermediate node on p. The subpath of p from i to k must have cost $A(k-1, i, k)$ and the subpath from k to j must have cost $A(k-1, k, j)$, so the cost of p is $A(k-1, i, k) + A(k-1, k, j)$.

We leave it as an exercise to give a more rigorous proof of this recurrence along the lines of the proof given for the problem of scheduling with deadlines, profits and durations.

Step 3: Give a high-level program.
We could compute the values we want using a 3-dimensional array $B[0..n, 1..n, 1..n]$ in a very straightforward way. However, it suffices to use a 2-dimensional array $B[1..n, 1..n]$; the idea is that after k executions of the body of the for-loop, $B[i, j]$ will equal $A(k, i, j)$. We will also use an array $B'[1..n, 1..n]$ that will be useful when we want to compute cheapest paths (rather than just costs of cheapest paths) in Step 4.

```
**All_Pairs_CP**
for $i : 1..n$ do
    $B[i, i] \leftarrow 0$
    $B'[i, i] \leftarrow 0$
    for $j : 1..n$ such that $j \neq i$ do
        $B[i, j] \leftarrow C(i, j)$
        $B'[i, j] \leftarrow 0$
    end for
end for

for $k : 1..n$ do
    for $i : 1..n$ do
        for $j : 1..n$ do
            if $B[i, k] + B[k, j] < B[i, j]$ then
                $B[i, j] \leftarrow B[i, k] + B[k, j]$
                $B'[i, j] \leftarrow k$
            end if
        end for
    end for
end for
```

We want to prove the following lemma about this program.

Lemma: For every k, i, j such that $0 \leq k \leq n$ and $1 \leq i, j \leq n$, after the kth execution of the body of the for-loop the following hold:

- $B[i,j] = A(k,i,j)$
- $B'[i,j]$ is the smallest number such that there exists a path p from i to j all of whose intermediate vertices are in $\{1, \cdots, B'[i,j]\}$, such that the cost of p is $A(k,i,j)$. (Note that this implies that $B'[i,j] \leq k$).

Proof: We prove this by induction on k. The base case is easy. To see why the induction step holds for the first part, we only have to worry about the fact that when we are computing the kth version of $B[i,j]$, some elements of B have already been updated. That is, $B[i,k]$ might be equal to $A(k-1,i,k)$, or it might have already been updated to be equal to $A(k,i,k)$ (and similarly for $B[k,j]$); however this doesn't matter, since $A(k-1,i,k) = A(k,i,k)$. The rest of the details, including the part of the induction step for the second part of the Lemma, are left as an exercise. $\square$

This Lemma implies that when the program has finished running, $B'[i,j]$ is the smallest number such that there exists a path p from i to j all of whose intermediate vertices are in $\{1, \cdots, B'[i,j]\}$, such that the cost of p is $D(i,j)$.

Step 4: Compute an optimal solution.
For this problem, computing an optimal solution can mean one of two different things. One possibility is that we want to print out a cheapest path from i to j, for *every* pair of vertices (i,j). Another possibility is that after computing B and B', we will be given an arbitrary pair (i,j), and we will want to compute a cheapest path from i to j as quickly as possible; this is the situation we are interested in here.

Assume we have already computed the arrays B and B'; we are now given a pair of vertices i and j, and we want to print out the edges in some cheapest path from i to j. If $i = j$ then we don't print out anything; otherwise we will call PrintOpt(i,j). The call PrintOpt(i,j) will satisfy the following Precondition/Postcondition pair:
Precondition: $1 \leq i, j \leq n$ and $i \neq j$.
Postcondition The edges of a path p have been printed out such that p is a path from i to j, and such that all the intermediate vertices of p are in $\{1, \cdots, B'[i,j]\}$, and such that no vertex occurs more than once in p. (Note that this holds even if there are edges of 0 cost in the graph.)

The full program (assuming we have already computed the correct values into B') is as follows:

```
**procedure** PrintOpt($i, j$)
    $k \leftarrow B'[i,j]$
    if $k = 0$ then
        put "edge from",$i$,"to",$j$
    else
        PrintOpt($i, k$)
        PrintOpt($k, j$)
    end if
end PrintOpt

if $i \neq j$ then PrintOpt($i, j$) end if
```

Exercise:
Prove that the call PrintOpt(i, j) satisfies the above Precondition/Postcondition pair. Prove that if $i \neq j$, then PrintOpt(i, j) runs in time linear in the number of edges printed out; conclude that the whole program in Step 4 runs in time $O(n)$.

Analysis of the Running Time
The program in Step 3 clearly runs in time $O(n^3)$, and the Exercise tells us that the program in Step 4 runs in time $O(n)$. So the total time is $O(n^3)$. We can view the size of the input as n – the number of vertices, or as n^2 – an upper bound on the number of edges. In any case, this is clearly a polynomial-time algorithm. (Note that if in Step 4 we want to print out cheapest paths for *all* pairs i, j, this would still take just time $O(n^3)$.)

Remark: The recurrence in Step 2 is actually not as obvious as it might at first appear. It is instructive to consider a slightly different problem, where we want to find the cost of a *longest* (that is, most expensive) path between every pair of vertices. Let us assume that there is an edge between every pair of vertices, with a cost that is a real number. The notion of longest path is still not well defined, since if there is a cycle with positive cost, then there will be arbitrarily costly paths between every pair of points. It does make sense, however, to ask for the length of a longest *simple* path between every pair of points. (A simple path is one on which no vertex repeats.) So define $D(i, j)$ to be the cost of a most expensive simple path from i to j. Define $A(k, i, j)$ to be the cost of a most expensive path from i to j from among those whose intermediate vertices are in $\{1, 2, \ldots, k\}$. Then it is *not* necessarily true that $A(k, i, j) = \max\{A(k-1, i, j), A(k-1, i, k) + A(k-1, k, j)\}$. Do you see why?

Longest Common Subsequence

The input consists of two sequences $\vec{x} = x_1, \dots, x_n$ and $\vec{y} = y_1, \dots, y_m$. The goal is to find a longest common subsequence of $\vec{x}$ and $\vec{y}$, that is a sequence $z_1, \dots, z_k$ 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 $x_1 \dots x_i$ and $y_1 \dots y_j$ are prefixes of $\vec{x}$ and $\vec{y}$ respectively, and $x_i = y_j$, then the length of the longest common subsequence of $x_1 \dots x_i$ and $y_1 \dots y_j$ is one plus the length of the longest common subsequence of $x_1 \dots x_{i-1}$ and $y_1 \dots y_{j-1}$.

Step 1. We define an array to hold partial solution to the problem. For $0 \le i \le n$ and $0 \le j \le m$, $A(i,j)$ is the length of the longest common subsequence of $x_1 \dots x_i$ and $y_1 \dots y_j$. 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 $0 \le i \le n$ and $0 \le j \le m$, $A(i,0) = A(0,j) = 0$.
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 $1 \le i \le n$ and $1 \le j \le m$,

$A(i,j)$ 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 $1 \le i \le n$, 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 $\max\{A(i) \mid 1 \le i \le n\}$.

Step 2: Give a recurrence.
For $1 \le i \le n$,
$A(i) = 1 + \max\{A(j) \mid 1 \le j < i \text{ and } a_j < a_i\}$.
(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)	$\emptyset$	7	3	8	4	2	6
$\emptyset$	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 $O(n^2)$.

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

```
put $a_m$
while $A(m) > 1$ do
    $i \leftarrow m - 1$
    while not($a_i < a_m$ and $A(i) = A(m) - 1$) do
        $i \leftarrow i - 1$
    end while
    $m \leftarrow i$
    put $a_m$
end while
```

Dynamic Programming Summary:

- An algorithm follows the "bottom-up" approach, and it is based on a recurrence for filling in an array (or arrays). The array usually contains the value of an optimal solution.
- The correctness of the algorithm relies on the correctness of the recurrence, which in turn relies on the "principle of optimality" (and is usually proved by case analysis).
- Usually, we can get not just the value of an optimal solution, but also an optimal solution itself (sometimes, may need to use an additional array for that).