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.
Activity Selection with Profits

The input is information about \( n \) activities; for the \( i \)th activity we are given three nonnegative real numbers \((s_i, f_i, g_i)\) where \( s_i \) is the start time of activity \( i \), \( f_i \) is the finish time of activity \( i \) (and \( s_i < f_i \)), and \( g_i \geq 0 \) is the profit we get if we schedule activity \( i \). For \( i, j \in \{1, 2, \ldots, n\} \), \( i \neq j \), we say \( i \) and \( j \) are compatible if the corresponding courses don’t overlap, that is, if \( f_i \leq s_j \) or \( f_j \leq s_i \). A (feasible) schedule is a set \( S \subseteq \{1, 2, \ldots, n\} \) such that every two distinct numbers in \( S \) are compatible. The profit of a schedule \( S \) is \( P(S) = \sum_{i \in S} g_i \). We want an algorithm for finding a schedule that maximizes profit. (Note that we are assuming, for convenience, that \( s_i < f_i \) for every activity \( i \), rather than merely \( s_i \leq f_i \). This is no great loss: if trivial activities – that is, activities of 0 duration – exist, we can always remove them, find an optimal schedule \( S \), and then add the trivial activities to \( S \).

Before starting the dynamic programming algorithm itself, we do some precomputation, as follows. We first sort the activities according to finish time, so that \( f_1 \leq f_2 \leq \cdots \leq f_n \). We also compute, for every activity \( i \), a number \( H(i) \) defined as

\[ H(i) = \max \{ l \in \{1, 2, \ldots, i-1\} \mid f_l \leq s_i \} \]

the maximum value of the empty set is 0. We can compute each value \( H(i) \) in time \( O(\log n) \) using binary search, so all of the precomputation can be done in time \( O(n \log n) \).

We now perform the four steps of the dynamic programming method.

**Step 1:** Describe an array of values we want to compute.

For every integer \( i \), \( 0 \leq i \leq n \), define

\[ A(i) = \text{the largest profit we can get by (feasibly) scheduling activities from } \{1, 2, \ldots, i\}. \]

The value we are ultimately interested in is \( A(n) \).

**Step 2:** Give a recurrence.

- \( A(0) = 0 \).
  
  This is true because if we are not allowed to schedule any activities at all, then the highest profit we can make is 0.

- Let \( 1 \leq i \leq n \). Then
  
  \[ A(i) = \max\{A(i-1), g_i + A(H(i))\}. \]

To see why this equality holds, consider a best possible schedule \( S \) among those containing activities from \( \{1, 2, \ldots, i\} \). If \( i \notin S \), then \( P(S) = A(i-1) \). Otherwise \( i \in S \); because the activities were sorted by finish time, the other activities in \( S \) have finish times \( \leq f_i \), and therefore must have finish times \( \leq s_i \); so the rest of \( S \) must be an optimal schedule among those that contain activities from \( \{1, 2, \ldots, H(i)\} \).

**Step 3:** Give a high-level program.

We leave it as an exercise to give a program for computing the values of \( A \) in time \( O(n) \).
Step 4: Compute an optimal solution.
This is left as an exercise. If one is careful, then using the array $A$, one can compute an optimal schedule in time $O(n)$.

The total time used by this algorithm is $O(n \log n)$. It is a polynomial-time algorithm.

Example

<table>
<thead>
<tr>
<th>Activity $i$:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start $s_i$:</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Finish $f_i$:</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>Profit $g_i$:</td>
<td>20</td>
<td>30</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>$H(i)$:</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$A(0) = 0$
$A(1) = \max\{0, 20 + A(H(1))\} = 20$
$A(2) = \max\{20, 30 + A(H(2))\} = 30$
$A(3) = \max\{30, 20 + A(H(3))\} = 40$
$A(4) = \max\{40, 30 + A(H(4))\} = 40$

The reader should note that the problem of scheduling activities with profits can be generalized to multiple processors, and an appropriate generalization of the above dynamic programming algorithm can be used to solve this more general problem. This algorithm will run in polynomial time, but only if the number of processors is fixed. (That is, if the running time is $O(n^c)$, then the exponent $c$ may depend on the number of processors.)
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 \( v_1, \ldots, v_m \) 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 \( V = \{1, \ldots, n\} \), 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 “\( \infty \)”. 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 \( \infty \) as a cost in order to denote that we really view that edge as not existing. (We do arithmetic on \( \infty \) in the obvious way: \( \infty + \infty \) is \( \infty \), and \( \infty + \) any real number is \( \infty \).)

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
\[
A(k, i, j) = \text{the cost of a cheapest path from } i \text{ to } j, \text{ from among those paths from } i \text{ to } j \text{ whose intermediate nodes are all in } \{1, \ldots, k\}. \quad (\text{We define the intermediate nodes of the path } v_1, \ldots, v_m \text{ to be the set } \{v_2, \ldots, v_{m-1}\}; \text{ note that if } m \text{ is 1 or 2, then this set is empty.})
\]

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 \( \{1, \ldots, k\} \). 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.

\textit{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.

\textbf{All_Pairs_CP}

\begin{verbatim}
for i : 1..n do
  B[i, i] ← 0
  B'[i, i] ← 0
  for j : 1..n such that j ≠ i do
    B[i, j] ← C(i, j)
    B'[i, j] ← 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] ← B[i, k] + B[k, j]
        B'[i, j] ← k
      end if
    end for
  end for
end for
\end{verbatim}

We want to prove the following lemma about this program.

\textbf{Lemma:} For every \( k, i, j \) such that \( 0 \leq k \leq n \) and \( 1 \leq i, j \leq n \), after the \( k \)th execution of the body of the for-loop the following hold:

\begin{itemize}
  \item \( B[i, j] = A(k, i, j) \)
\end{itemize}
• $B'[i, j]$ is the smallest number such that there exists a path $p$ from $i$ to $j$ all of whose intermediate vertices are in $\{1, \ldots, B'[i, j]\}$, 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 $k$th 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. □

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 $\{1, \ldots, B'[i, j]\}$, 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 $\{1, \ldots, B'[i, j]\}$, 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 $\text{PRINTOPT}(i, j)$ end if

**Exercise:**
Prove that the call $\text{PRINTOPT}(i, j)$ satisfies the above Precondition/Postcondition pair.
Prove that if $i \neq j$, then $\text{PRINTOPT}(i, j)$ 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 Increasing Subsequence

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

**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 $\bar{a}$ 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 $O(n \log n)$.)

**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 $\bar{a}$. 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 $O(n^2)$.

\[
\begin{align*}
m &\leftarrow 1 \\
\text{for } i : 2..n & \quad \text{put } a_m \\
\quad \text{if } A(i) > A(m) & \quad \text{while } A(m) > 1 \text{ do} \\
\quad \quad m &\leftarrow i \\
\quad \quad \text{end if} \\
\text{end for} \\
\quad \text{while not}(a_i < a_m \text{ and } A(i) = A(m) - 1) & \quad \text{do} \\
\quad \quad i &\leftarrow i - 1 \\
\quad \quad \text{end while} \\
\quad m &\leftarrow i \\
\quad \text{put } a_m \\
\text{end while}
\end{align*}
\]

LCS and LIS arrays for the example

<table>
<thead>
<tr>
<th>A(i,j)</th>
<th>0</th>
<th>7</th>
<th>3</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

| A(i)   | 1  | 1  | 2  | 2  | 1  | 3  |

8
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 subsequence is not always substring: 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 substring it would have to be $j' = j + 1$.

For example, let $\vec{x}$ and $\vec{y}$ be two DNA strings $\vec{x} = \text{TGACTA}$ and $\vec{y} = \text{GTGCATG}$; $n = 6$ and $m = 7$. Then one common subsequence would be $\text{GTA}$. However, it is not the longest possible common subsequence: there are common subsequences $\text{TGCA}$, $\text{TGAT}$ and $\text{TGCT}$ 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 $1 \leq j \leq m$,

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

**Step 3.** Skipped.

**Step 4.** As before, just retrace the decisions.