Chapter 6

Dynamic programming

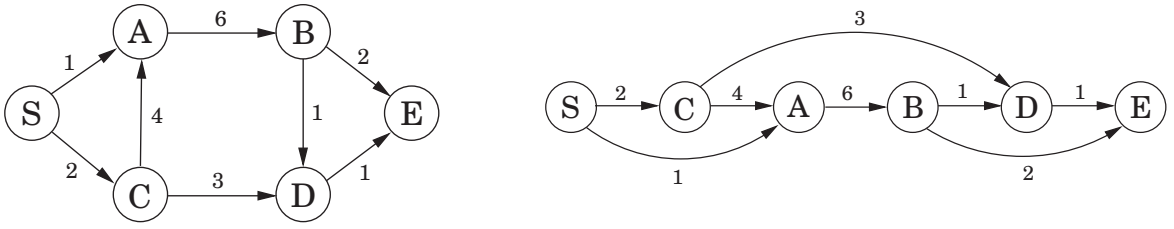
In the preceding chapters we have seen some elegant design principles—such as divide-and-conquer, graph exploration, and greedy choice—that yield definitive algorithms for a variety of important computational tasks. The drawback of these tools is that they can only be used on very specific types of problems. We now turn to the two *sledgehammers* of the algorithms craft, *dynamic programming* and *linear programming*, techniques of very broad applicability that can be invoked when more specialized methods fail. Predictably, this generality often comes with a cost in efficiency.

6.1 Shortest paths in dags, revisited

At the conclusion of our study of shortest paths (Chapter 4), we observed that the problem is especially easy in directed acyclic graphs (dags). Let's recapitulate this case, because it lies at the heart of dynamic programming.

The special distinguishing feature of a dag is that its nodes can be *linearized*; that is, they can be arranged on a line so that all edges go from left to right (Figure 6.1). To see why this helps with shortest paths, suppose we want to figure out distances from node S to the other nodes. For concreteness, let's focus on node D. The only way to get to it is through its

Figure 6.1 A dag and its linearization (topological ordering).



predecessors, B or C; so to find the shortest path to D, we need only compare these two routes:

$$\texttt{dist}(D) = \min\{\texttt{dist}(B) + 1, \texttt{dist}(C) + 3\}.$$

A similar relation can be written for every node. If we compute these `dist` values in the left-to-right order of Figure 6.1, we can always be sure that by the time we get to a node v, we already have all the information we need to compute $\texttt{dist}(v)$. We are therefore able to compute all distances in a single pass:

```
initialize all dist(·) values to ∞
dist(s) = 0
for each v ∈ V\{s}, in linearized order:
    dist(v) = min_{(u,v)∈E}{dist(u) + l(u, v)}
```

Notice that this algorithm is solving a collection of *subproblems*, $\{\texttt{dist}(u) : u \in V\}$. We start with the smallest of them, $\texttt{dist}(s)$, since we immediately know its answer to be 0. We then proceed with progressively "larger" subproblems—distances to vertices that are further and further along in the linearization—where we are thinking of a subproblem as large if we need to have solved a lot of other subproblems before we can get to it.

This is a very general technique. At each node, we compute some function of the values of the node's predecessors. It so happens that our particular function is a minimum of sums, but we could just as well make it a *maximum*, in which case we would get *longest* paths in the dag. Or we could use a product instead of a sum inside the brackets, in which case we would end up computing the path with the smallest product of edge lengths.

Dynamic programming is a very powerful algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until the whole lot of them is solved. In dynamic programming we are not given a dag; the dag is *implicit*. Its nodes are the subproblems we define, and its edges are the dependencies between the subproblems: if to solve subproblem B we need the answer to subproblem A, then there is a (conceptual) edge from A to B. In this case, A is thought of as a smaller subproblem than B—and it will always be smaller, in an obvious sense.

But it's time we saw an example.

6.2 Longest increasing subsequences

In the *longest increasing subsequence* problem, the input is a sequence of numbers $a_1, \ldots, a_n$. A *subsequence* is any subset of these numbers taken in order, of the form $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ where $1 \le i_1 < i_2 < \cdots < i_k \le n$, and an *increasing* subsequence is one in which the numbers are getting strictly larger. The task is to find the increasing subsequence of greatest length. For instance, the longest increasing subsequence of $5, 2, 8, 6, 3, 6, 9, 7$ is $2, 3, 6, 9$:

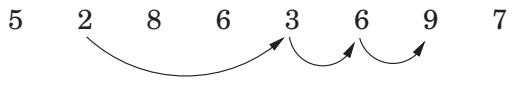
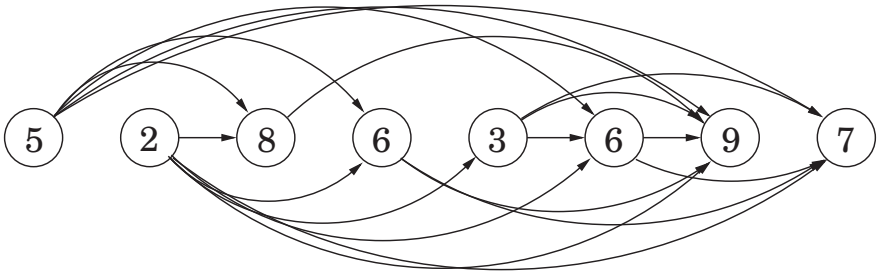


Figure 6.2 The dag of increasing subsequences.



In this example, the arrows denote transitions between consecutive elements of the optimal solution. More generally, to better understand the solution space, let's create a graph of *all* permissible transitions: establish a node i for each element a_i, and add directed edges (i, j) whenever it is possible for a_i and a_j to be consecutive elements in an increasing subsequence, that is, whenever $i < j$ and $a_i < a_j$ (Figure 6.2).

Notice that (1) this graph $G = (V, E)$ is a dag, since all edges (i, j) have $i < j$, and (2) there is a one-to-one correspondence between increasing subsequences and paths in this dag. Therefore, our goal is simply to find the longest path in the dag!

Here is the algorithm:

```
for j = 1, 2, ..., n:
    L(j) = 1 + max{L(i) : (i, j) ∈ E}
return max_j L(j)
```

$L(j)$ is the length of the longest path—the longest increasing subsequence—ending at j (plus 1, since strictly speaking we need to count nodes on the path, not edges). By reasoning in the same way as we did for shortest paths, we see that any path to node j must pass through one of its predecessors, and therefore $L(j)$ is 1 plus the maximum $L(\cdot)$ value of these predecessors. If there are no edges into j, we take the maximum over the empty set, zero. And the final answer is the *largest* $L(j)$, since any ending position is allowed.

This is dynamic programming. In order to solve our original problem, we have defined a collection of subproblems $\{L(j) : 1 \leq j \leq n\}$ with the following key property that allows them to be solved in a single pass:

> (*) There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.

In our case, each subproblem is solved using the relation

$$L(j) \;=\; 1 + \max\{L(i) : (i, j) \in E\},$$

an expression which involves only smaller subproblems. How long does this step take? It requires the predecessors of j to be known; for this the adjacency list of the reverse graph G^R, constructible in linear time (recall Exercise 3.5), is handy. The computation of $L(j)$ then takes time proportional to the indegree of j, giving an overall running time linear in $|E|$. This is at most $O(n^2)$, the maximum being when the input array is sorted in increasing order. Thus the dynamic programming solution is both simple and efficient.

There is one last issue to be cleared up: the L-values only tell us the *length* of the optimal subsequence, so how do we recover the subsequence itself? This is easily managed with the same bookkeeping device we used for shortest paths in Chapter 4. While computing $L(j)$, we should also note down $\texttt{prev}(j)$, the next-to-last node on the longest path to j. The optimal subsequence can then be reconstructed by following these backpointers.

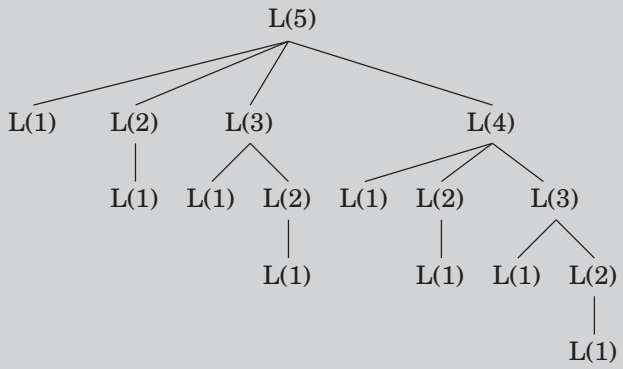
Recursion? No, thanks.

Returning to our discussion of longest increasing subsequences: the formula for $L(j)$ also suggests an alternative, recursive algorithm. Wouldn't that be even simpler?

Actually, recursion is a very bad idea: the resulting procedure would require exponential time! To see why, suppose that the dag contains edges (i, j) for *all* $i < j$—that is, the given sequence of numbers $a_1, a_2, \ldots, a_n$ is sorted. In that case, the formula for subproblem $L(j)$ becomes

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}.$$

The following figure unravels the recursion for $L(5)$. Notice that the same subproblems get solved over and over again!



For $L(n)$ this tree has exponentially many nodes (can you bound it?), and so a recursive solution is disastrous.

Then why did recursion work so well with divide-and-conquer? The key point is that in divide-and-conquer, a problem is expressed in terms of subproblems that are *substantially smaller*, say half the size. For instance, `mergesort` sorts an array of size n by recursively sorting two subarrays of size $n/2$. Because of this sharp drop in problem size, the full recursion tree has only logarithmic depth and a polynomial number of nodes.

In contrast, in a typical dynamic programming formulation, a problem is reduced to subproblems that are only slightly smaller—for instance, $L(j)$ relies on $L(j-1)$. Thus the full recursion tree generally has polynomial depth and an exponential number of nodes. However, it turns out that most of these nodes are repeats, that there are not too many *distinct* subproblems among them. Efficiency is therefore obtained by explicitly enumerating the distinct subproblems and solving them in the right order.

Programming?

The origin of the term *dynamic programming* has very little to do with writing code. It was first coined by Richard Bellman in the 1950s, a time when computer programming was an esoteric activity practiced by so few people as to not even merit a name. Back then programming meant "planning," and "dynamic programming" was conceived to optimally plan multistage processes. The dag of Figure 6.2 can be thought of as describing the possible ways in which such a process can evolve: each node denotes a state, the leftmost node is the starting point, and the edges leaving a state represent possible actions, leading to different states in the next unit of time.

The etymology of *linear programming*, the subject of Chapter 7, is similar.

6.3 Edit distance

When a spell checker encounters a possible misspelling, it looks in its dictionary for other words that are close by. What is the appropriate notion of closeness in this case?

A natural measure of the distance between two strings is the extent to which they can be *aligned*, or matched up. Technically, an alignment is simply a way of writing the strings one above the other. For instance, here are two possible alignments of SNOWY and SUNNY:

```
S - N O W Y        - S N O W - Y
S U N N - Y        S U N - - N Y
   Cost: 3            Cost: 5
```

The "–" indicates a "gap"; any number of these can be placed in either string. The *cost* of an alignment is the number of columns in which the letters differ. And the *edit distance* between two strings is the cost of their best possible alignment. Do you see that there is no better alignment of SNOWY and SUNNY than the one shown here with a cost of 3?

Edit distance is so named because it can also be thought of as the minimum number of *edits*—insertions, deletions, and substitutions of characters—needed to transform the first string into the second. For instance, the alignment shown on the left corresponds to three edits: insert U, substitute O $\rightarrow$ N, and delete W.

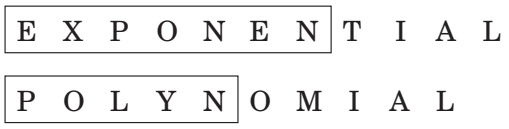
In general, there are so many possible alignments between two strings that it would be terribly inefficient to search through all of them for the best one. Instead we turn to dynamic programming.

A dynamic programming solution

When solving a problem by dynamic programming, the most crucial question is, *What are the subproblems?* As long as they are chosen so as to have the property (*) from page 171. it is an easy matter to write down the algorithm: iteratively solve one subproblem after the other, in order of increasing size.

Our goal is to find the edit distance between two strings $x[1 \cdots m]$ and $y[1 \cdots n]$. What is a good subproblem? Well, it should go part of the way toward solving the whole problem; so how

Figure 6.3 The subproblem $E(7,5)$.



about looking at the edit distance between some *prefix* of the first string, $x[1 \cdots i]$, and some *prefix* of the second, $y[1 \cdots j]$? Call this subproblem $E(i,j)$ (see Figure 6.3). Our final objective, then, is to compute $E(m,n)$.

For this to work, we need to somehow express $E(i,j)$ in terms of smaller subproblems. Let's see—what do we know about the best alignment between $x[1 \cdots i]$ and $y[1 \cdots j]$? Well, its rightmost column can only be one of three things:

$$\begin{matrix} x[i] \\ - \end{matrix} \quad \text{or} \quad \begin{matrix} - \\ y[j] \end{matrix} \quad \text{or} \quad \begin{matrix} x[i] \\ y[j] \end{matrix}$$

The first case incurs a cost of 1 for this particular column, and it remains to align $x[1 \cdots i-1]$ with $y[1 \cdots j]$. But this is exactly the subproblem $E(i-1,j)$! We seem to be getting somewhere. In the second case, also with cost 1, we still need to align $x[1 \cdots i]$ with $y[1 \cdots j-1]$. This is again another subproblem, $E(i,j-1)$. And in the final case, which either costs 1 (if $x[i] \neq y[j]$) or 0 (if $x[i] = y[j]$), what's left is the subproblem $E(i-1,j-1)$. In short, we have expressed $E(i,j)$ in terms of three *smaller* subproblems $E(i-1,j)$, $E(i,j-1)$, $E(i-1,j-1)$. We have no idea which of them is the right one, so we need to try them all and pick the best:

$$E(i,j) \;=\; \min\{1 + E(i-1,j),\ 1 + E(i,j-1),\ \text{diff}(i,j) + E(i-1,j-1)\}$$

where for convenience diff(i,j) is defined to be 0 if $x[i] = y[j]$ and 1 otherwise.

For instance, in computing the edit distance between EXPONENTIAL and POLYNOMIAL, subproblem $E(4,3)$ corresponds to the prefixes EXPO and POL. The rightmost column of their best alignment must be one of the following:

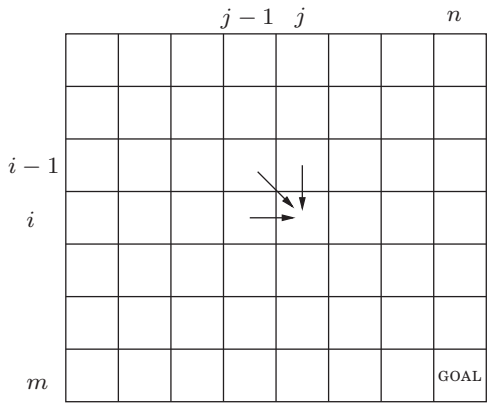
$$\begin{matrix} \texttt{O} \\ - \end{matrix} \quad \text{or} \quad \begin{matrix} - \\ \texttt{L} \end{matrix} \quad \text{or} \quad \begin{matrix} \texttt{O} \\ \texttt{L} \end{matrix}$$

Thus, $E(4,3) = \min\{1 + E(3,3),\ 1 + E(4,2),\ 1 + E(3,2)\}$.

The answers to all the subproblems $E(i,j)$ form a two-dimensional table, as in Figure 6.4. In what order should these subproblems be solved? Any order is fine, as long as $E(i-1,j)$, $E(i,j-1)$, and $E(i-1,j-1)$ are handled *before* $E(i,j)$. For instance, we could fill in the table one row at a time, from top row to bottom row, and moving left to right across each row. Or alternatively, we could fill it in column by column. Both methods would ensure that by the time we get around to computing a particular table entry, all the other entries we need are already filled in.

Figure 6.4 (a) The table of subproblems. Entries $E(i-1, j-1)$, $E(i-1, j)$, and $E(i, j-1)$ are needed to fill in $E(i, j)$. (b) The final table of values found by dynamic programming.

(a)



(b)

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

With both the subproblems and the ordering specified, we are almost done. There just remain the "base cases" of the dynamic programming, the very smallest subproblems. In the present situation, these are $E(0, \cdot)$ and $E(\cdot, 0)$, both of which are easily solved. $E(0, j)$ is the edit distance between the 0-length prefix of x, namely the empty string, and the first j letters of y: clearly, j. And similarly, $E(i, 0) = i$.

At this point, the algorithm for edit distance basically writes itself.

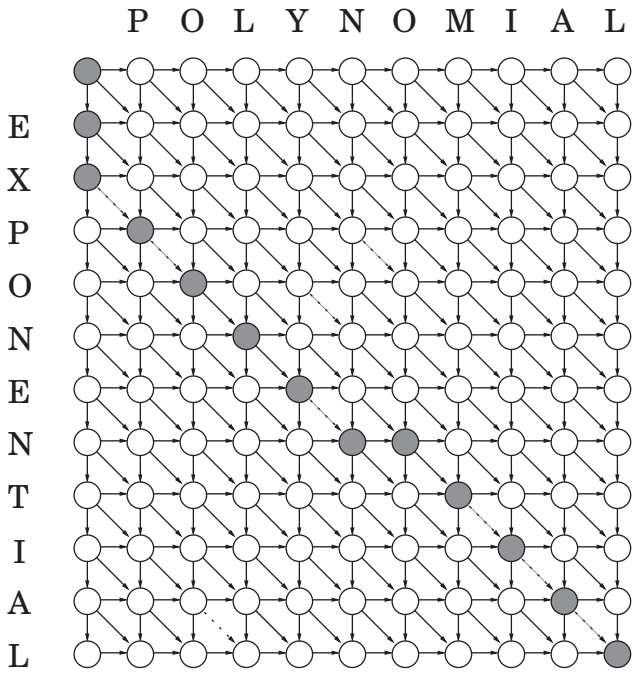
```
for i = 0, 1, 2, ..., m:
    E(i, 0) = i
for j = 1, 2, ..., n:
    E(0, j) = j
for i = 1, 2, ..., m:
    for j = 1, 2, ..., n:
        E(i, j) = min{E(i − 1, j) + 1, E(i, j − 1) + 1, E(i − 1, j − 1) + diff(i, j)}
return E(m, n)
```

This procedure fills in the table row by row, and left to right within each row. Each entry takes constant time to fill in, so the overall running time is just the size of the table, $O(mn)$.

And in our example, the edit distance turns out to be 6:

```
E  X  P  O  N  E  N  −  T  I  A  L
−  −  P  O  L  Y  N  O  M  I  A  L
```

Figure 6.5 The underlying dag, and a path of length 6.



The underlying dag

Every dynamic program has an underlying dag structure: think of each node as representing a subproblem, and each edge as a precedence constraint on the order in which the subproblems can be tackled. Having nodes $u_1, \ldots, u_k$ point to v means "subproblem v can only be solved once the answers to $u_1, \ldots, u_k$ are known."

In our present edit distance application, the nodes of the underlying dag correspond to subproblems, or equivalently, to positions (i, j) in the table. Its edges are the precedence constraints, of the form $(i-1, j) \rightarrow (i, j)$, $(i, j-1) \rightarrow (i, j)$, and $(i-1, j-1) \rightarrow (i, j)$ (Figure 6.5). In fact, we can take things a little further and put weights on the edges so that the edit distances are given by shortest paths in the dag! To see this, set all edge lengths to 1, except for $\{(i-1, j-1) \rightarrow (i, j) : x[i] = y[j]\}$ (shown dotted in the figure), whose length is 0. The final answer is then simply the distance between nodes $s = (0, 0)$ and $t = (m, n)$. One possible shortest path is shown, the one that yields the alignment we found earlier. On this path, each move down is a deletion, each move right is an insertion, and each diagonal move is either a match or a substitution.

By altering the weights on this dag, we can allow generalized forms of edit distance, in which insertions, deletions, and substitutions have different associated costs.

Common subproblems

Finding the right subproblem takes creativity and experimentation. But there are a few standard choices that seem to arise repeatedly in dynamic programming.

i. The input is $x_1, x_2, \ldots, x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

The number of subproblems is therefore linear.

ii. The input is $x_1, \ldots, x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$.

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

$$\boxed{y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5} \quad y_6 \quad y_7 \quad y_8$$

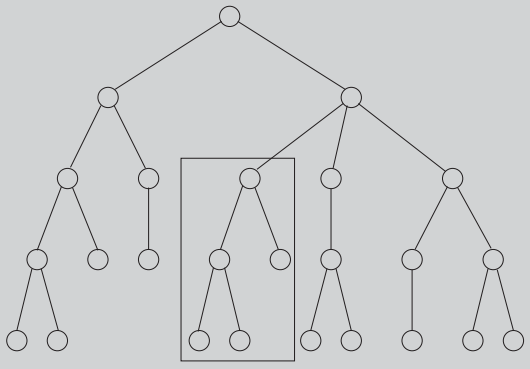
The number of subproblems is $O(mn)$.

iii. The input is $x_1, \ldots, x_n$ and a subproblem is $x_i, x_{i+1}, \ldots, x_j$.

$$x_1 \quad x_2 \quad \boxed{x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

The number of subproblems is $O(n^2)$.

iv. The input is a rooted tree. A subproblem is a rooted subtree.



If the tree has n nodes, how many subproblems are there?

We've already encountered the first two cases, and the others are coming up shortly.

Of mice and men

Our bodies are extraordinary machines: flexible in function, adaptive to new environments, and able to interact and reproduce. All these capabilities are specified by a program unique to each of us, a string that is 3 billion characters long over the alphabet $\{A, C, G, T\}$—our DNA.

The DNA sequences of any two people differ by only about 0.1%. However, this still leaves 3 million positions on which they vary, more than enough to explain the vast range of human diversity. These differences are of great scientific and medical interest—for instance, they might help predict which people are prone to certain diseases.

DNA is a vast and seemingly inscrutable program, but it can be broken down into smaller units that are more specific in their role, rather like subroutines. These are called *genes*. Computers have become a crucial tool in understanding the genes of humans and other organisms, to the extent that *computational genomics* is now a field in its own right. Here are examples of typical questions that arise.

1. When a new gene is discovered, one way to gain insight into its function is to find known genes that match it closely. This is particularly helpful in transferring knowledge from well-studied species, such as mice, to human beings.

   A basic primitive in this search problem is to define an efficiently computable notion of when two strings approximately match. The biology suggests a generalization of edit distance, and dynamic programming can be used to compute it.

   Then there's the problem of searching through the vast thicket of known genes: the database GenBank already has a total length of over 10^{10}, and this number is growing rapidly. The current method of choice is BLAST, a clever combination of algorithmic tricks and biological intuitions that has made it the most widely used software in computational biology.

2. Methods for *sequencing* DNA (that is, determining the string of characters that constitute it) typically only find fragments of 500–700 characters. Billions of these randomly scattered fragments can be generated, but how can they be assembled into a coherent DNA sequence? For one thing, the position of any one fragment in the final sequence is unknown and must be inferred by piecing together overlapping fragments.

   A showpiece of these efforts is the draft of human DNA completed in 2001 by two groups simultaneously: the publicly funded Human Genome Consortium and the private Celera Genomics.

3. When a particular gene has been sequenced in each of several species, can this information be used to reconstruct the evolutionary history of these species?

We will explore these problems in the exercises at the end of this chapter. Dynamic programming has turned out to be an invaluable tool for some of them and for computational biology in general.

6.4 Knapsack

During a robbery, a burglar finds much more loot than he had expected and has to decide what to take. His bag (or "knapsack") will hold a total weight of at most W pounds. There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$. What's the most valuable combination of items he can fit into his bag?[1]

For instance, take $W = 10$ and

Item	Weight	Value
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

There are two versions of this problem. If there are unlimited quantities of each item available, the optimal choice is to pick item 1 and two of item 4 (total: \$48). On the other hand, if there is one of each item (the burglar has broken into an art gallery, say), then the optimal knapsack contains items 1 and 3 (total: \$46).

As we shall see in Chapter 8, neither version of this problem is likely to have a polynomial-time algorithm. However, using dynamic programming they can both be solved in $O(nW)$ time, which is reasonable when W is small, but is not polynomial since the input size is proportional to $\log W$ rather than W.

Knapsack with repetition

Let's start with the version that allows repetition. As always, the main question in dynamic programming is, what are the subproblems? In this case we can shrink the original problem in two ways: we can either look at smaller knapsack capacities $w \leq W$, or we can look at fewer items (for instance, items $1, 2, \ldots, j$, for $j \leq n$). It usually takes a little experimentation to figure out exactly what works.

The first restriction calls for smaller capacities. Accordingly, define

$$K(w) = \text{maximum value achievable with a knapsack of capacity } w.$$

Can we express this in terms of smaller subproblems? Well, if the optimal solution to $K(w)$ includes item i, then removing this item from the knapsack leaves an optimal solution to $K(w - w_i)$. In other words, $K(w)$ is simply $K(w - w_i) + v_i$, for some i. We don't know which i, so we need to try all possibilities.

$$K(w) = \max_{i: w_i \leq w} \{K(w - w_i) + v_i\},$$

where as usual our convention is that the maximum over an empty set is 0. We're done! The algorithm now writes itself, and it is characteristically simple and elegant.

[1] If this application seems frivolous, replace "weight" with "CPU time" and "only W pounds can be taken" with "only W units of CPU time are available." Or use "bandwidth" in place of "CPU time," etc. The knapsack problem generalizes a wide variety of resource-constrained selection tasks.

```
K(0) = 0
for w = 1 to W:
    K(w) = max{K(w − w_i) + v_i : w_i ≤ w}
return K(W)
```

This algorithm fills in a one-dimensional table of length $W + 1$, in left-to-right order. Each entry can take up to $O(n)$ time to compute, so the overall running time is $O(nW)$.

As always, there is an underlying dag. Try constructing it, and you will be rewarded with a startling insight: this particular variant of knapsack boils down to finding the longest path in a dag!

Knapsack without repetition

On to the second variant: what if repetitions are not allowed? Our earlier subproblems now become completely useless. For instance, knowing that the value $K(w - w_n)$ is very high doesn't help us, because we don't know whether or not item n already got used up in this partial solution. We must therefore refine our concept of a subproblem to carry additional information about the items being used. We add a second parameter, $0 \leq j \leq n$:

$$K(w, j) = \text{maximum value achievable using a knapsack of capacity } w \text{ and items } 1, \ldots, j.$$

The answer we seek is $K(W, n)$.

How can we express a subproblem $K(w, j)$ in terms of smaller subproblems? Quite simple: either item j is needed to achieve the optimal value, or it isn't needed:

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}.$$

(The first case is invoked only if $w_j \leq w$.) In other words, we can express $K(w, j)$ in terms of subproblems $K(\cdot, j - 1)$.

The algorithm then consists of filling out a two-dimensional table, with $W + 1$ rows and $n + 1$ columns. Each table entry takes just constant time, so even though the table is much larger than in the previous case, the running time remains the same, $O(nW)$. Here's the code.

```
Initialize all K(0, j) = 0 and all K(w, 0) = 0
for j = 1 to n:
    for w = 1 to W:
        if w_j > w:   K(w, j) = K(w, j − 1)
        else:   K(w, j) = max{K(w, j − 1), K(w − w_j, j − 1) + v_j}
return K(W, n)
```

Memoization

In dynamic programming, we write out a recursive formula that expresses large problems in terms of smaller ones and then use it to fill out a table of solution values in a bottom-up manner, from smallest subproblem to largest.

The formula also suggests a recursive algorithm, but we saw earlier that naive recursion can be terribly inefficient, because it solves the same subproblems over and over again. What about a more intelligent recursive implementation, one that remembers its previous invocations and thereby avoids repeating them?

On the knapsack problem (with repetitions), such an algorithm would use a hash table (recall Section 1.5) to store the values of $K(\cdot)$ that had already been computed. At each recursive call requesting some $K(w)$, the algorithm would first check if the answer was already in the table and then would proceed to its calculation only if it wasn't. This trick is called *memoization*:

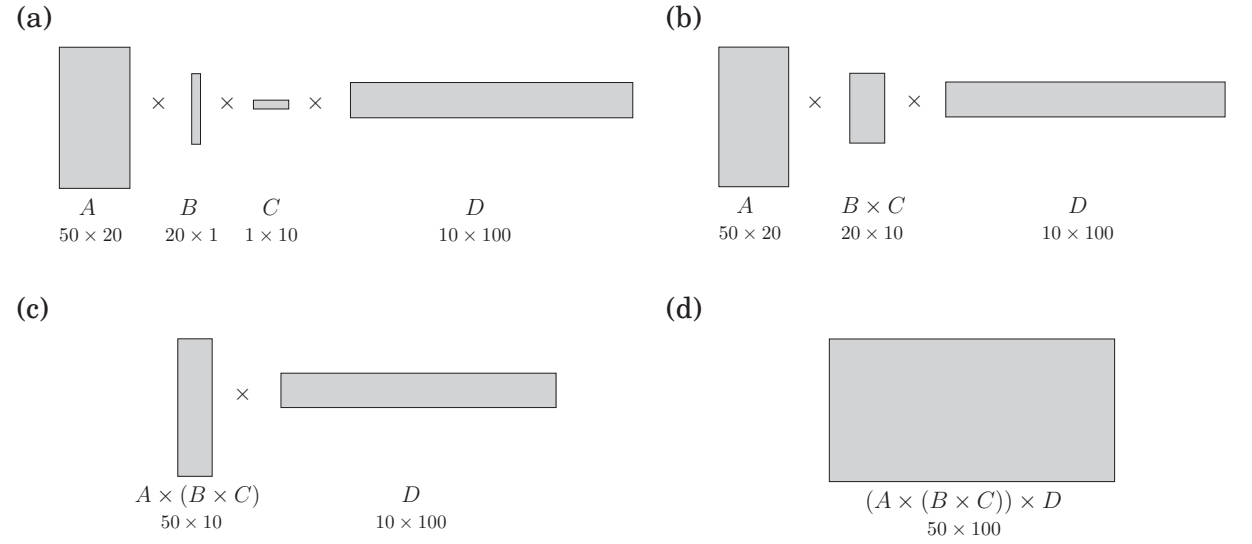
```
A hash table, initially empty, holds values of K(w) indexed by w

function knapsack(w)
if w is in hash table:  return K(w)
K(w) = max{knapsack(w − w_i) + v_i : w_i ≤ w}
insert K(w) into hash table, with key w
return K(w)
```

Since this algorithm never repeats a subproblem, its running time is $O(nW)$, just like the dynamic program. However, the constant factor in this big-O notation is substantially larger because of the overhead of recursion.

In some cases, though, memoization pays off. Here's why: dynamic programming automatically solves every subproblem that could conceivably be needed, while memoization only ends up solving the ones that are actually used. For instance, suppose that W and all the weights w_i are multiples of 100. Then a subproblem $K(w)$ is useless if 100 does not divide w. The memoized recursive algorithm will never look at these extraneous table entries.

Figure 6.6 $A \times B \times C \times D = (A \times (B \times C)) \times D$.



6.5 Chain matrix multiplication

Suppose that we want to multiply four matrices, $A \times B \times C \times D$, of dimensions 50×20, 20×1, 1×10, and 10×100, respectively (Figure 6.6). This will involve iteratively multiplying two matrices at a time. Matrix multiplication is not *commutative* (in general, $A \times B \neq B \times A$), but it is *associative*, which means for instance that $A \times (B \times C) = (A \times B) \times C$. Thus we can compute our product of four matrices in many different ways, depending on how we parenthesize it. Are some of these better than others?

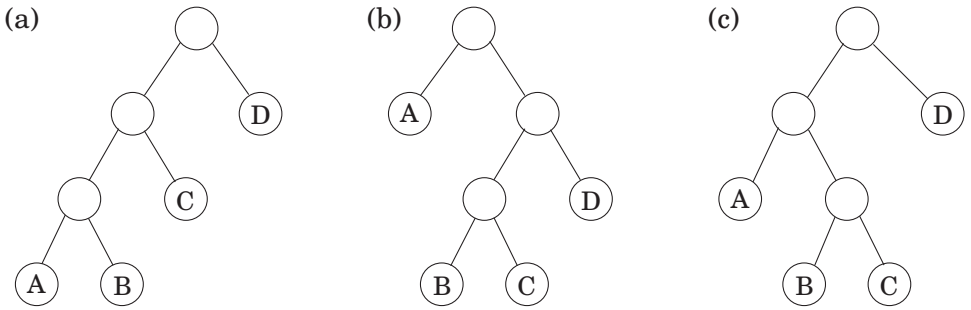
Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes mnp multiplications, to a good enough approximation. Using this formula, let's compare several different ways of evaluating $A \times B \times C \times D$:

Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	$120,200$
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	$60,200$
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	$7,000$

As you can see, the order of multiplications makes a big difference in the final running time! Moreover, the natural *greedy* approach, to always perform the cheapest matrix multiplication available, leads to the second parenthesization shown here and is therefore a failure.

How do we determine the optimal order, if we want to compute $A_1 \times A_2 \times \cdots \times A_n$, where the A_i's are matrices with dimensions $m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$, respectively? The first thing to notice is that a particular parenthesization can be represented very naturally by a binary tree in which the individual matrices correspond to the leaves, the root is the final

Figure 6.7 (a) $((A \times B) \times C) \times D$; (b) $A \times ((B \times C) \times D)$; (c) $(A \times (B \times C)) \times D$.



product, and interior nodes are intermediate products (Figure 6.7). The possible orders in which to do the multiplication correspond to the various full binary trees with n leaves, whose number is exponential in n (Exercise 2.13). We certainly cannot try each tree, and with brute force thus ruled out, we turn to dynamic programming.

The binary trees of Figure 6.7 are suggestive: for a tree to be optimal, its subtrees must also be optimal. What are the subproblems corresponding to the subtrees? They are products of the form $A_i \times A_{i+1} \times \cdots \times A_j$. Let's see if this works: for $1 \leq i \leq j \leq n$, define

$$C(i,j) = \text{minimum cost of multiplying } A_i \times A_{i+1} \times \cdots \times A_j.$$

The size of this subproblem is the number of matrix multiplications, $|j - i|$. The smallest subproblem is when $i = j$, in which case there's nothing to multiply, so $C(i,i) = 0$. For $j > i$, consider the optimal subtree for $C(i,j)$. The first branch in this subtree, the one at the top, will split the product in two pieces, of the form $A_i \times \cdots \times A_k$ and $A_{k+1} \times \cdots \times A_j$, for some k between i and j. The cost of the subtree is then the cost of these two partial products, plus the cost of combining them: $C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j$. And we just need to find the splitting point k for which this is smallest:

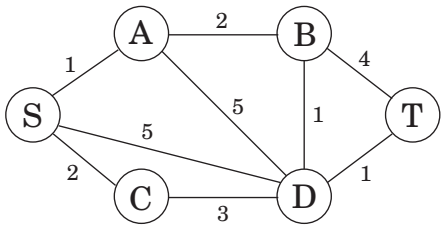
$$C(i,j) = \min_{i \leq k < j} \{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}.$$

We are ready to code! In the following, the variable s denotes subproblem size.

```
for i = 1 to n:   C(i,i) = 0
for s = 1 to n-1:
  for i = 1 to n-s:
    j = i+s
    C(i,j) = min{C(i,k) + C(k+1,j) + m_{i-1} · m_k · m_j : i ≤ k < j}
return C(1,n)
```

The subproblems constitute a two-dimensional table, each of whose entries takes $O(n)$ time to compute. The overall running time is thus $O(n^3)$.

Figure 6.8 We want a path from s to t that is both short *and* has few edges.



6.6 Shortest paths

We started this chapter with a dynamic programming algorithm for the elementary task of finding the shortest path in a dag. We now turn to more sophisticated shortest-path problems and see how these too can be accommodated by our powerful algorithmic technique.

Shortest reliable paths

Life is complicated, and abstractions such as graphs, edge lengths, and shortest paths rarely capture the whole truth. In a communications network, for example, even if edge lengths faithfully reflect transmission delays, there may be other considerations involved in choosing a path. For instance, each extra edge in the path might be an extra "hop" fraught with uncertainties and dangers of packet loss. In such cases, we would like to avoid paths with too many edges. Figure 6.8 illustrates this problem with a graph in which the shortest path from S to T has four edges, while there is another path that is a little longer but uses only two edges. If four edges translate to prohibitive unreliability, we may have to choose the latter path.

Suppose then that we are given a graph G with lengths on the edges, along with two nodes s and t and an integer k, and we want the shortest path from s to t *that uses at most* k *edges*.

Is there a quick way to adapt Dijkstra's algorithm to this new task? Not quite: that algorithm focuses on the length of each shortest path without "remembering" the number of hops in the path, which is now a crucial piece of information.

In dynamic programming, the trick is to choose subproblems so that all vital information is remembered and carried forward. In this case, let us define, for each vertex v and each integer $i \leq k$, $\texttt{dist}(v, i)$ to be *the length of the shortest path from* s *to* v *that uses* i *edges*. The starting values $\texttt{dist}(v, 0)$ are ∞ for all vertices except s, for which it is 0. And the general update equation is, naturally enough,

$$\texttt{dist}(v, i) = \min_{(u,v) \in E} \{\texttt{dist}(u, i-1) + \ell(u, v)\}.$$

Need we say more?

All-pairs shortest paths

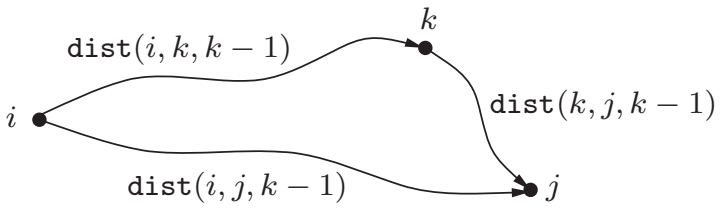
What if we want to find the shortest path not just between s and t but between *all* pairs of vertices? One approach would be to execute our general shortest-path algorithm from Section 4.6.1 (since there may be negative edges) $|V|$ times, once for each starting node. The total running time would then be $O(|V|^2|E|)$. We'll now see a better alternative, the $O(|V|^3)$ dynamic programming-based *Floyd-Warshall algorithm*.

Is there is a good subproblem for computing distances between all pairs of vertices in a graph? Simply solving the problem for more and more pairs or starting points is unhelpful, because it leads right back to the $O(|V|^2|E|)$ algorithm.

One idea comes to mind: the shortest path $u \to w_1 \to \cdots \to w_l \to v$ between u and v uses some number of intermediate nodes—possibly none. Suppose we disallow intermediate nodes altogether. Then we can solve all-pairs shortest paths at once: the shortest path from u to v is simply the direct edge (u, v), if it exists. What if we now gradually expand the *set of permissible intermediate nodes*? We can do this one node at a time, updating the shortest path lengths at each stage. Eventually this set grows to all of V, at which point all vertices are allowed to be on all paths, and we have found the true shortest paths between vertices of the graph!

More concretely, number the vertices in V as $\{1, 2, \ldots, n\}$, and let $\texttt{dist}(i, j, k)$ denote the length of the shortest path from i to j in which only nodes $\{1, 2, \ldots, k\}$ can be used as intermediates. Initially, $\texttt{dist}(i, j, 0)$ is the length of the direct edge between i and j, if it exists, and is ∞ otherwise.

What happens when we expand the intermediate set to include an extra node k? We must reexamine all pairs i, j and check whether using k as an intermediate point gives us a shorter path from i to j. But this is easy: a shortest path from i to j that uses k along with possibly other lower-numbered intermediate nodes goes through k just once (why? because we assume that there are no negative cycles). And we have already calculated the length of the shortest path from i to k and from k to j using only lower-numbered vertices:



Thus, using k gives us a shorter path from i to j if and only if

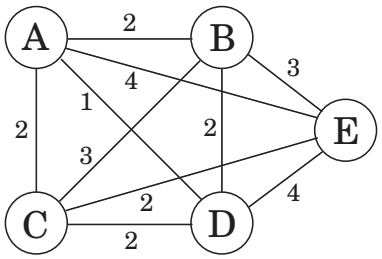
$$\texttt{dist}(i, k, k-1) + \texttt{dist}(k, j, k-1) \;<\; \texttt{dist}(i, j, k-1),$$

in which case $\texttt{dist}(i, j, k)$ should be updated accordingly.

Here is the Floyd-Warshall algorithm—and as you can see, it takes $O(|V|^3)$ time.

```
for i = 1 to n:
    for j = 1 to n:
        dist(i, j, 0) = ∞
```

Figure 6.9 The optimal traveling salesman tour has length 10.




```
for all (i, j) ∈ E:
    dist(i, j, 0) = ℓ(i, j)
for k = 1 to n:
    for i = 1 to n:
        for j = 1 to n:
            dist(i, j, k) = min{dist(i, k, k − 1) + dist(k, j, k − 1),  dist(i, j, k − 1)}
```

The traveling salesman problem

A traveling salesman is getting ready for a big sales tour. Starting at his hometown, suitcase in hand, he will conduct a journey in which each of his target cities is visited exactly once before he returns home. Given the pairwise distances between cities, what is the best order in which to visit them, so as to minimize the overall distance traveled?

Denote the cities by $1, \ldots, n$, the salesman's hometown being 1, and let $D = (d_{ij})$ be the matrix of intercity distances. The goal is to design a tour that starts and ends at 1, includes all other cities exactly once, and has minimum total length. Figure 6.9 shows an example involving five cities. Can you spot the optimal tour? Even in this tiny example, it is tricky for a human to find the solution; imagine what happens when hundreds of cities are involved.

It turns out this problem is also difficult for computers. In fact, the traveling salesman problem (TSP) is one of the most notorious computational tasks. There is a long history of attempts at solving it, a long saga of failures and partial successes, and along the way, major advances in algorithms and complexity theory. The most basic piece of bad news about the TSP, which we will better understand in Chapter 8, is that it is highly unlikely to be solvable in polynomial time.

How long does it take, then? Well, the brute-force approach is to evaluate every possible tour and return the best one. Since there are $(n-1)!$ possibilities, this strategy takes $O(n!)$ time. We will now see that dynamic programming yields a much faster solution, though not a polynomial one.

What is the appropriate subproblem for the TSP? Subproblems refer to partial solutions, and in this case the most obvious partial solution is the initial portion of a tour. Suppose we have started at city 1 as required, have visited a few cities, and are now in city j. What

information do we need in order to extend this partial tour? We certainly need to know j, since this will determine which cities are most convenient to visit next. And we also need to know all the cities visited so far, so that we don't repeat any of them. Here, then, is an appropriate subproblem.

> For a subset of cities $S \subseteq \{1, 2, \ldots, n\}$ that includes 1, and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.

When $|S| > 1$, we define $C(S, 1) = \infty$ since the path cannot both start and end at 1.

Now, let's express $C(S, j)$ in terms of smaller subproblems. We need to start at 1 and end at j; what should we pick as the second-to-last city? It has to be some $i \in S$, so the overall path length is the distance from 1 to i, namely, $C(S - \{j\}, i)$, plus the length of the final edge, d_{ij}. We must pick the best such i:

$$C(S, j) = \min_{i \in S: i \neq j} C(S - \{j\}, i) + d_{ij}.$$

The subproblems are ordered by $|S|$. Here's the code.

```
C({1}, 1) = 0
for s = 2 to n:
    for all subsets S ⊆ {1, 2, ..., n} of size s and containing 1:
        C(S, 1) = ∞
        for all j ∈ S, j ≠ 1:
            C(S, j) = min{C(S − {j}, i) + d_ij : i ∈ S, i ≠ j}
return min_j C({1, ..., n}, j) + d_j1
```

There are at most $2^n \cdot n$ subproblems, and each one takes linear time to solve. The total running time is therefore $O(n^2 2^n)$.

6.7 Independent sets in trees

A subset of nodes $S \subset V$ is an *independent set* of graph $G = (V, E)$ if there are no edges between them. For instance, in Figure 6.10 the nodes $\{1, 5\}$ form an independent set, but nodes $\{1, 4, 5\}$ do not, because of the edge between 4 and 5. The largest independent set is $\{2, 3, 6\}$.

Like several other problems we have seen in this chapter (knapsack, traveling salesman), finding the largest independent set in a graph is believed to be intractable. However, when the graph happens to be a *tree*, the problem can be solved in linear time, using dynamic programming. And what are the appropriate subproblems? Already in the chain matrix multiplication problem we noticed that the layered structure of a tree provides a natural definition of a subproblem—as long as one node of the tree has been identified as a root.

So here's the algorithm: Start by rooting the tree at any node r. Now, each node defines a subtree—the one hanging from it. This immediately suggests subproblems:

$$I(u) \;=\; \text{size of largest independent set of subtree hanging from } u.$$

On time and memory

The amount of time it takes to run a dynamic programming algorithm is easy to discern from the dag of subproblems: in many cases *it is just the total number of edges in the dag!* All we are really doing is visiting the nodes in linearized order, examining each node's inedges, and, most often, doing a constant amount of work per edge. By the end, each edge of the dag has been examined once.

But how much computer *memory* is required? There is no simple parameter of the dag characterizing this. It is certainly possible to do the job with an amount of memory proportional to the number of vertices (subproblems), but we can usually get away with much less. The reason is that the value of a particular subproblem only needs to be remembered until the larger subproblems depending on it have been solved. Thereafter, the memory it takes up can be released for reuse.

For example, in the Floyd-Warshall algorithm the value of $\texttt{dist}(i, j, k)$ is not needed once the $\texttt{dist}(\cdot, \cdot, k+1)$ values have been computed. Therefore, we only need two $|V| \times |V|$ arrays to store the `dist` values, one for odd values of k and one for even values: when computing $\texttt{dist}(i, j, k)$, we overwrite $\texttt{dist}(i, j, k-2)$.

(And let us not forget that, as always in dynamic programming, we also need one more array, $\texttt{prev}(i, j)$, storing the next to last vertex in the current shortest path from i to j, a value that must be updated with $\texttt{dist}(i, j, k)$. We omit this mundane but crucial bookkeeping step from our dynamic programming algorithms.)

Can you see why the edit distance dag in Figure 6.5 only needs memory proportional to the length of the shorter string?

Our final goal is $I(r)$.

Dynamic programming proceeds as always from smaller subproblems to larger ones, that is to say, bottom-up in the rooted tree. Suppose we know the largest independent sets for all subtrees below a certain node u; in other words, suppose we know $I(w)$ for all descendants w of u. How can we compute $I(u)$? Let's split the computation into two cases: any independent set either includes u or it doesn't (Figure 6.11).

$$I(u) = \max \left\{ 1 + \sum_{\text{grandchildren } w \text{ of } u} I(w), \sum_{\text{children } w \text{ of } u} I(w) \right\}.$$

If the independent set includes u, then we get one point for it, but we aren't allowed to include the children of u—therefore we move on to the grandchildren. This is the first case in the formula. On the other hand, if we don't include u, then we don't get a point for it, but we can move on to its children.

The number of subproblems is exactly the number of vertices. With a little care, the running time can be made linear, $O(|V| + |E|)$.