

Chapter 6

Dynamic programming

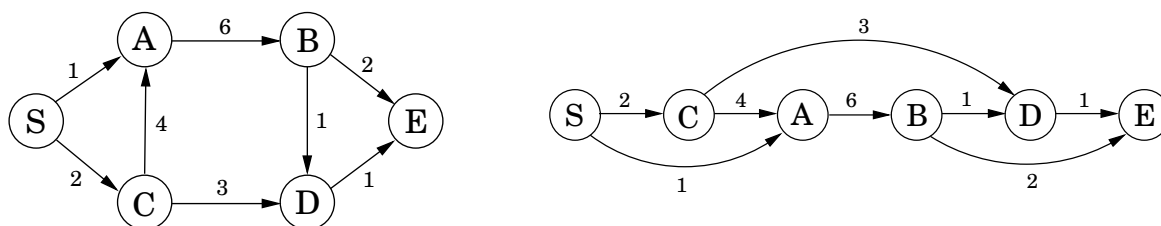
In the preceding chapters we have seen some elegant design principles—such as divide-and-conquer, graph exploration, and greedy choice—that yield definitive algorithms for a variety of important computational tasks. The drawback of these tools is that they can only be used on very specific types of problems. 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:

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

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 $\text{dist}(v)$. We are therefore able to compute all distances in a single pass:

```

initialize all  $\text{dist}(\cdot)$  values to  $\infty$ 
 $\text{dist}(s) = 0$ 
for each  $v \in V \setminus \{s\}$ , in linearized order:
     $\text{dist}(v) = \min_{(u,v) \in E} \{\text{dist}(u) + l(u,v)\}$ 

```

Notice that this algorithm is solving a collection of *subproblems*, $\{\text{dist}(u) : u \in V\}$. We start with the smallest of them, $\text{dist}(s)$, 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, \dots, a_n . A *subsequence* is any subset of these numbers taken in order, of the form $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ where $1 \leq i_1 < i_2 < \dots < i_k \leq n$, 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:

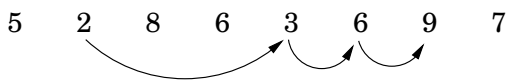
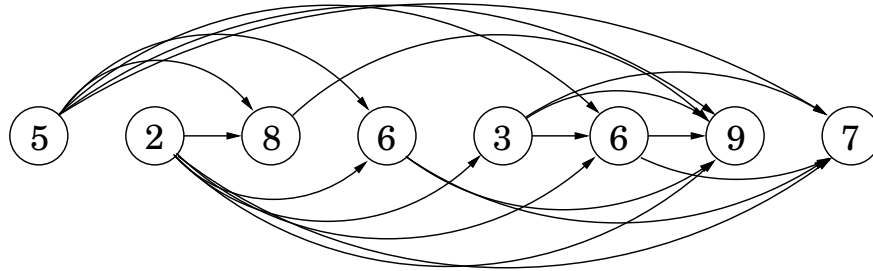


Figure 6.2 The dag of increasing subsequences.

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  $j = 1, 2, \dots, n$ :
     $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$ 
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 $\text{prev}(j)$, the next-to-last node on the longest path to j . The optimal subsequence can then be reconstructed by following these backpointers.

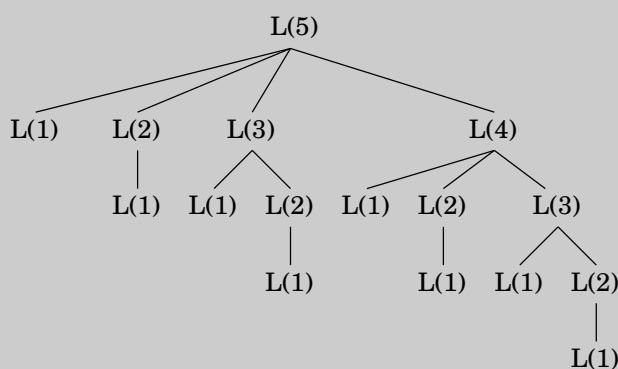
Recursion? No, thanks.

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, \dots, a_n is sorted. In that case, the formula for subproblem $L(j)$ becomes

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

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 → 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 $E(7, 5)$.

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

about looking at the edit distance between some *prefix* of the first string, $x[1 \dots i]$, and some *prefix* of the second, $y[1 \dots j]$? 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 $x[1 \dots i]$ and $y[1 \dots j]$? Well, its rightmost column can only be one of three things:

$$\begin{array}{ccccc} x[i] & & - & & x[i] \\ & \text{or} & & \text{or} & \\ - & & y[j] & & y[j] \end{array}$$

The first case incurs a cost of 1 for this particular column, and it remains to align $x[1 \dots i-1]$ with $y[1 \dots j]$. 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 $x[1 \dots i]$ with $y[1 \dots j-1]$. 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 $\text{diff}(i, j)$ 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:

$$\begin{array}{ccccc} O & & - & & O \\ & \text{or} & & \text{or} & \\ - & & L & & L \end{array}$$

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

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)

				$j-1$	j			n
$i-1$								
i								
m								GOAL

(b)

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

```

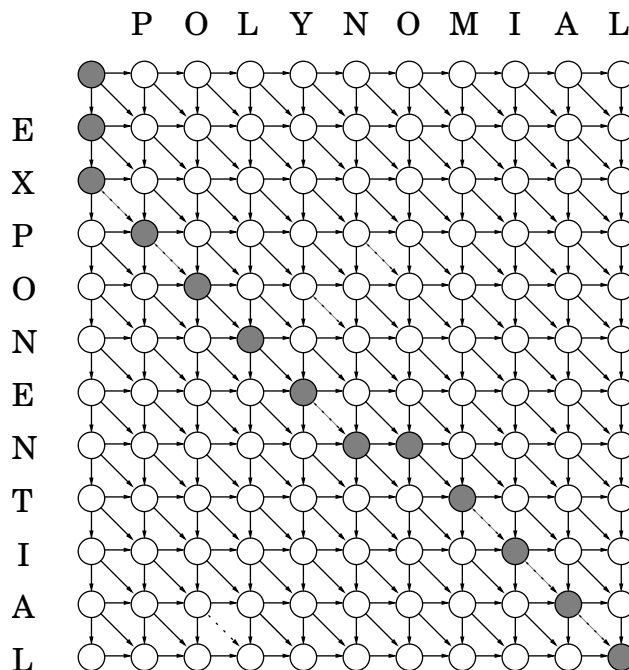
for  $i = 0, 1, 2, \dots, m$ :
     $E(i, 0) = i$ 
for  $j = 1, 2, \dots, n$ :
     $E(0, j) = j$ 
for  $i = 1, 2, \dots, m$ :
    for  $j = 1, 2, \dots, n$ :
         $E(i, j) = \min\{E(i-1, j) + 1, E(i, j-1) + 1, E(i-1, j-1) + \text{diff}(i, j)\}$ 
return  $E(m, n)$ 

```

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, \dots, u_k point to v means “subproblem v can only be solved once the answers to u_1, \dots, u_k are known.”

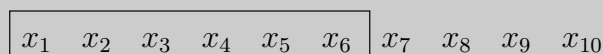
In our present edit distance application, the nodes of the underlying dag correspond to subproblems, or equivalently, to positions (i, j) in the table. Its edges are the precedence constraints, of the form $(i-1, j) \rightarrow (i, j)$, $(i, j-1) \rightarrow (i, j)$, and $(i-1, j-1) \rightarrow (i, j)$ (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

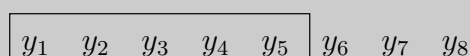
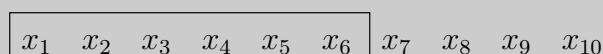
Finding the right subproblem takes creativity and experimentation. But there are a few standard choices that seem to arise repeatedly in dynamic programming.

- i. The input is x_1, x_2, \dots, x_n and a subproblem is x_1, x_2, \dots, x_i .



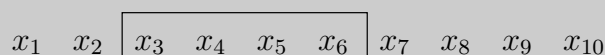
The number of subproblems is therefore linear.

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



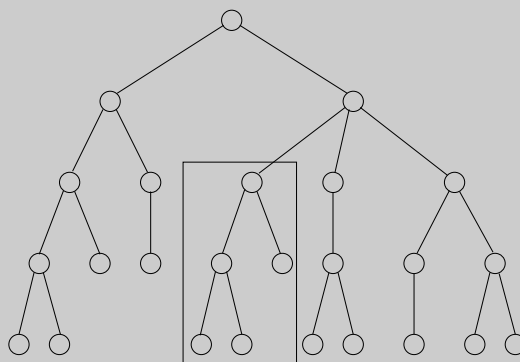
The number of subproblems is $O(mn)$.

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



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, \dots, w_n and dollar value v_1, \dots, 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 $1, 2, \dots, j$, 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.

¹If 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  $W$ :
     $K(w) = \max\{K(w - w_i) + v_i : w_i \leq w\}$ 
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, $0 \leq j \leq n$:

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

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?

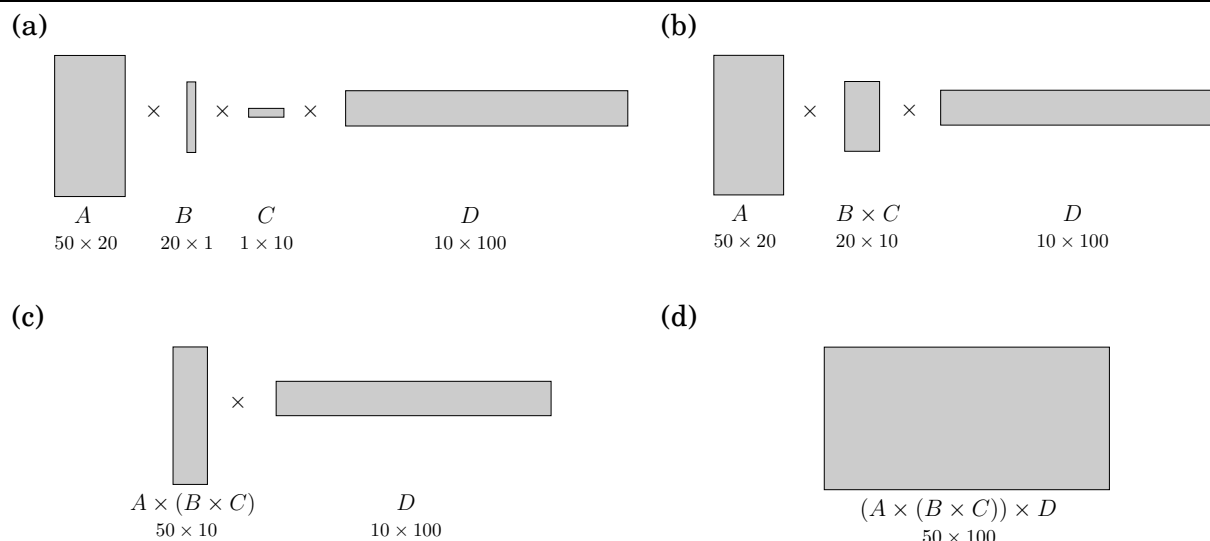
On the knapsack problem (with repetitions), such an algorithm would use a hash table (recall Section 1.5) to store the values of $K(\cdot)$ that had already been computed. At each recursive call requesting some $K(w)$, the algorithm would first check if the answer was already in the table and then would 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  $K(w)$ 
   $K(w) = \max\{\text{knapsack}(w - w_i) + v_i : w_i \leq w\}$ 
  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

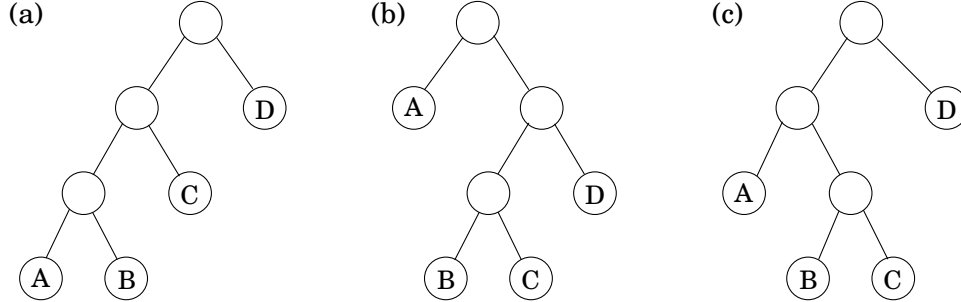
Suppose that we want to multiply four matrices, $A \times B \times C \times D$, of dimensions 50×20 , 20×1 , 1×10 , and 10×100 , respectively (Figure 6.6). This will involve iteratively multiplying two matrices at a time. Matrix multiplication is not *commutative* (in general, $A \times B \neq B \times A$), 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 $m_0 \times m_1, m_1 \times m_2, \dots, m_{n-1} \times m_n$, 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.

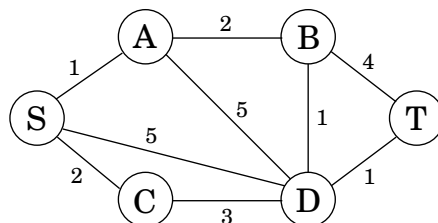
```

for  $i = 1$  to  $n$ :  $C(i, i) = 0$ 
for  $s = 1$  to  $n - 1$ :
  for  $i = 1$  to  $n - s$ :
     $j = i + s$ 
     $C(i, j) = \min\{C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j : i \leq k < j\}$ 
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 $i \leq k$, $\text{dist}(v, i)$ to be *the length of the shortest path from s to v that uses i edges*. The starting values $\text{dist}(v, 0)$ are ∞ for all vertices except s , for which it is 0. And the general update equation is, naturally enough,

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

Need we say more?

All-pairs shortest paths

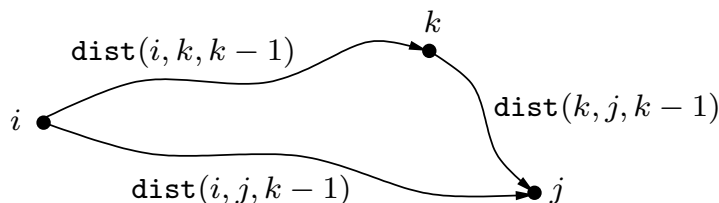
What if we want to find the shortest path not just between s and t but between *all* pairs of vertices? One approach would be to execute our general shortest-path algorithm from Section 4.6.1 (since there may be negative edges) $|V|$ times, once for each starting node. The total running time would then 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 $u \rightarrow w_1 \rightarrow \dots \rightarrow w_l \rightarrow v$ 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 $\{1, 2, \dots, n\}$, and let $\text{dist}(i, j, k)$ denote the length of the shortest path from i to j in which only nodes $\{1, 2, \dots, k\}$ can be used as intermediates. Initially, $\text{dist}(i, j, 0)$ 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

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

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

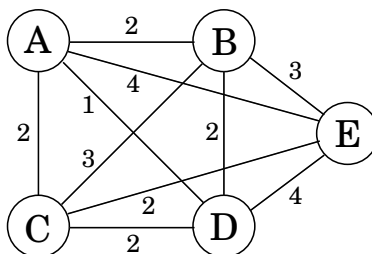
Here is the Floyd-Warshall algorithm—and as you can see, it takes $O(|V|^3)$ time.

```

for  $i = 1$  to  $n$ :
  for  $j = 1$  to  $n$ :
     $\text{dist}(i, j, 0) = \infty$ 

```

Figure 6.9 The optimal traveling salesman tour has length 10.



```

for all  $(i, j) \in E$ :
     $\text{dist}(i, j, 0) = \ell(i, j)$ 
for  $k = 1$  to  $n$ :
    for  $i = 1$  to  $n$ :
        for  $j = 1$  to  $n$ :
             $\text{dist}(i, j, k) = \min\{\text{dist}(i, k, k-1) + \text{dist}(k, j, k-1), \text{dist}(i, j, k-1)\}$ 

```

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 $1, \dots, n$, 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 $S \subseteq \{1, 2, \dots, n\}$ 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) + dij : i ∈ S, i ≠ j}
return minj C({1, ..., n}, j) + dj1

```

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 $\text{dist}(i, j, k)$ is not needed once the $\text{dist}(\cdot, \cdot, k+1)$ 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 $\text{dist}(i, j, k)$, we overwrite $\text{dist}(i, j, k-2)$.

(And let us not forget that, as always in dynamic programming, we also need one more array, $\text{prev}(i, j)$, storing the next to last vertex in the current shortest path from i to j , a value that must be updated with $\text{dist}(i, j, k)$. 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, $O(|V| + |E|)$.

Figure 6.10 The largest independent set in this graph has size 3.

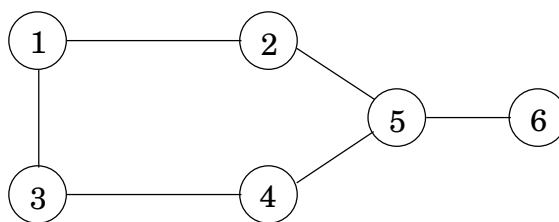
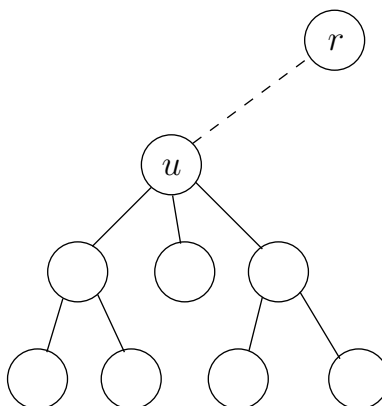


Figure 6.11 $I(u)$ is the size of the largest independent set of the subtree rooted at u . Two cases: either u is in this independent set, or it isn't.



Exercises

- 6.1. A *contiguous subsequence* of a list S is a subsequence made up of consecutive elements of S . For instance, if S is

$$5, 15, -30, 10, -5, 40, 10,$$

then $15, -30, 10$ is a contiguous subsequence but $5, 15, 40$ is not. Give a linear-time algorithm for the following task:

Input: A list of numbers, a_1, a_2, \dots, a_n .

Output: The contiguous subsequence of maximum sum (a subsequence of length zero has sum zero).

For the preceding example, the answer would be $10, -5, 40, 10$, with a sum of 55.

(*Hint:* For each $j \in \{1, 2, \dots, n\}$, consider contiguous subsequences ending exactly at position j .)

- 6.2. You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \dots < a_n$, where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance a_n), which is your destination.

You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the *penalty* for that day is $(200 - x)^2$. You want to plan your trip so as to minimize the total penalty—that is, the sum, over all travel days, of the daily penalties.

Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

- 6.3. Yuckdonald's is considering opening a series of restaurants along Quaint Valley Highway (QVH). The n possible locations are along a straight line, and the distances of these locations from the start of QVH are, in miles and in increasing order, m_1, m_2, \dots, m_n . The constraints are as follows:

- At each location, Yuckdonald's may open at most one restaurant. The expected profit from opening a restaurant at location i is p_i , where $p_i > 0$ and $i = 1, 2, \dots, n$.
- Any two restaurants should be at least k miles apart, where k is a positive integer.

Give an efficient algorithm to compute the maximum expected total profit subject to the given constraints.

- 6.4. You are given a string of n characters $s[1 \dots n]$, which you believe to be a corrupted text document in which all punctuation has vanished (so that it looks something like “itwasthebestoftimes...”). You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function $\text{dict}(\cdot)$: for any string w ,

$$\text{dict}(w) = \begin{cases} \text{true} & \text{if } w \text{ is a valid word} \\ \text{false} & \text{otherwise.} \end{cases}$$

- (a) Give a dynamic programming algorithm that determines whether the string $s[\cdot]$ can be reconstituted as a sequence of valid words. The running time should be at most $O(n^2)$, assuming calls to dict take unit time.
 - (b) In the event that the string is valid, make your algorithm output the corresponding sequence of words.
- 6.5. *Pebbling a checkerboard.* We are given a checkerboard which has 4 rows and n columns, and has an integer written in each square. We are also given a set of $2n$ pebbles, and we want to place some or all of these on the checkerboard (each pebble can be placed on exactly one square) so as to maximize the sum of the integers in the squares that are covered by pebbles. There is one constraint: for a placement of pebbles to be legal, no two of them can be on horizontally or vertically adjacent squares (diagonal adjacency is fine).
- (a) Determine the number of legal *patterns* that can occur in any column (in isolation, ignoring the pebbles in adjacent columns) and describe these patterns.

Call two patterns *compatible* if they can be placed on adjacent columns to form a legal placement. Let us consider subproblems consisting of the first k columns $1 \leq k \leq n$. Each subproblem can be assigned a *type*, which is the pattern occurring in the last column.

- (b) Using the notions of compatibility and type, give an $O(n)$ -time dynamic programming algorithm for computing an optimal placement.
- 6.6. Let us define a multiplication operation on three symbols a, b, c according to the following table; thus $ab = b$, $ba = c$, and so on. Notice that the multiplication operation defined by the table is neither associative nor commutative.

	<i>a</i>	<i>b</i>	<i>c</i>
<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>
<i>b</i>	<i>c</i>	<i>b</i>	<i>a</i>
<i>c</i>	<i>a</i>	<i>c</i>	<i>c</i>

Find an efficient algorithm that examines a string of these symbols, say *bbbbaac*, and decides whether or not it is possible to parenthesize the string in such a way that the value of the resulting expression is *a*. For example, on input *bbbbaac* your algorithm should return *yes* because $((b(bb))(ba))c = a$.

- 6.7. A subsequence is *palindromic* if it is the same whether read left to right or right to left. For instance, the sequence

A, C, G, T, G, T, C, A, A, A, A, T, C, G

has many palindromic subsequences, including *A, C, G, C, A* and *A, A, A, A* (on the other hand, the subsequence *A, C, T* is *not* palindromic). Devise an algorithm that takes a sequence $x[1 \dots n]$ and returns the (length of the) longest palindromic subsequence. Its running time should be $O(n^2)$.

- 6.8. Given two strings $x = x_1x_2 \dots x_n$ and $y = y_1y_2 \dots y_m$, we wish to find the length of their *longest common substring*, that is, the largest k for which there are indices i and j with $x_i x_{i+1} \dots x_{i+k-1} = y_j y_{j+1} \dots y_{j+k-1}$. Show how to do this in time $O(mn)$.

- 6.9. A certain string-processing language offers a primitive operation which splits a string into two pieces. Since this operation involves copying the original string, it takes n units of time for a string of length n , regardless of the location of the cut. Suppose, now, that you want to break a string into many pieces. The order in which the breaks are made can affect the total running time. For example, if you want to cut a 20-character string at positions 3 and 10, then making the first cut at position 3 incurs a total cost of $20 + 17 = 37$, while doing position 10 first has a better cost of $20 + 10 = 30$.

Give a dynamic programming algorithm that, given the locations of m cuts in a string of length n , finds the minimum cost of breaking the string into $m + 1$ pieces.

- 6.10. *Counting heads*. Given integers n and k , along with $p_1, \dots, p_n \in [0, 1]$, you want to determine the probability of obtaining exactly k heads when n biased coins are tossed independently at random, where p_i is the probability that the i th coin comes up heads. Give an $O(n^2)$ algorithm for this task.² Assume you can multiply and add two numbers in $[0, 1]$ in $O(1)$ time.
- 6.11. Given two strings $x = x_1x_2 \dots x_n$ and $y = y_1y_2 \dots y_m$, we wish to find the length of their *longest common subsequence*, that is, the largest k for which there are indices $i_1 < i_2 < \dots < i_k$ and $j_1 < j_2 < \dots < j_k$ with $x_{i_1}x_{i_2} \dots x_{i_k} = y_{j_1}y_{j_2} \dots y_{j_k}$. Show how to do this in time $O(mn)$.
- 6.12. You are given a convex polygon P on n vertices in the plane (specified by their x and y coordinates). A *triangulation* of P is a collection of $n - 3$ diagonals of P such that no two diagonals intersect (except possibly at their endpoints). Notice that a triangulation splits the polygon's interior into $n - 2$ disjoint triangles. The cost of a triangulation is the sum of the lengths of the diagonals in it. Give an efficient algorithm for finding a triangulation of minimum cost. (*Hint*: Label the vertices of P by $1, \dots, n$, starting from an arbitrary vertex and walking clockwise. For $1 \leq i < j \leq n$, let the subproblem $A(i, j)$ denote the minimum cost triangulation of the polygon spanned by vertices $i, i + 1, \dots, j$.)

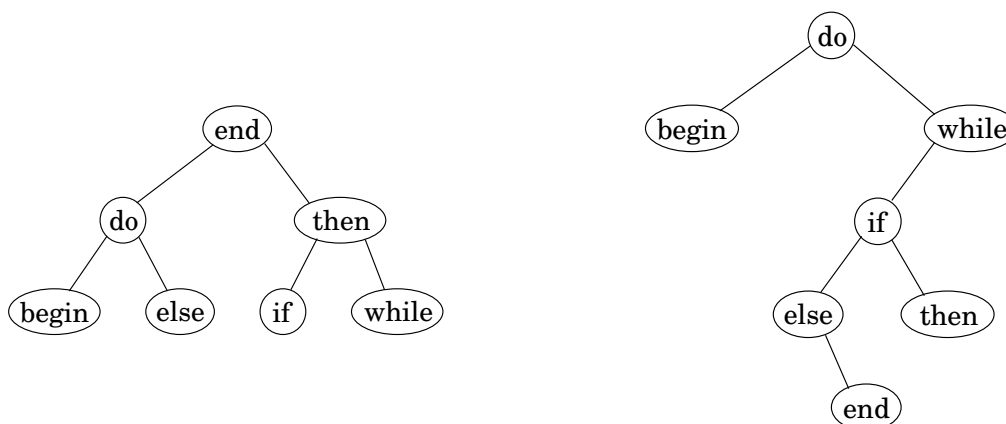
²In fact, there is also a $O(n \log^2 n)$ algorithm within your reach.

- 6.13. Consider the following game. A “dealer” produces a sequence $s_1 \cdots s_n$ of “cards,” face up, where each card s_i has a value v_i . Then two players take turns picking a card from the sequence, but can only pick the first or the last card of the (remaining) sequence. The goal is to collect cards of largest total value. (For example, you can think of the cards as bills of different denominations.) Assume n is even.
- Show a sequence of cards such that it is not optimal for the first player to start by picking up the available card of larger value. That is, the natural *greedy* strategy is suboptimal.
 - Give an $O(n^2)$ algorithm to compute an optimal strategy for the first player. Given the initial sequence, your algorithm should precompute in $O(n^2)$ time some information, and then the first player should be able to make each move optimally in $O(1)$ time by looking up the precomputed information.
- 6.14. *Cutting cloth.* You are given a rectangular piece of cloth with dimensions $X \times Y$, where X and Y are positive integers, and a list of n products that can be made using the cloth. For each product $i \in [1, n]$ you know that a rectangle of cloth of dimensions $a_i \times b_i$ is needed and that the final selling price of the product is c_i . Assume the a_i , b_i , and c_i are all positive integers. You have a machine that can cut any rectangular piece of cloth into two pieces either horizontally or vertically. Design an algorithm that determines the best return on the $X \times Y$ piece of cloth, that is, a strategy for cutting the cloth so that the products made from the resulting pieces give the maximum sum of selling prices. You are free to make as many copies of a given product as you wish, or none if desired.
- 6.15. Suppose two teams, A and B , are playing a match to see who is the first to win n games (for some particular n). We can suppose that A and B are equally competent, so each has a 50% chance of winning any particular game. Suppose they have already played $i + j$ games, of which A has won i and B has won j . Give an efficient algorithm to compute the probability that A will go on to win the match. For example, if $i = n - 1$ and $j = n - 3$ then the probability that A will win the match is $7/8$, since it must win any of the next three games.
- 6.16. The *garage sale problem* (courtesy of Professor Lofti Zadeh). On a given Sunday morning, there are n garage sales going on, g_1, g_2, \dots, g_n . For each garage sale g_j , you have an estimate of its value to you, v_j . For any two garage sales you have an estimate of the transportation cost d_{ij} of getting from g_i to g_j . You are also given the costs d_{0j} and d_{j0} of going between your home and each garage sale. You want to find a tour of a *subset* of the given garage sales, starting and ending at home, that maximizes your total benefit minus your total transportation costs. Give an algorithm that solves this problem in time $O(n^2 2^n)$. (*Hint:* This is closely related to the traveling salesman problem.)
- 6.17. Given an unlimited supply of coins of denominations x_1, x_2, \dots, x_n , we wish to make change for a value v ; that is, we wish to find a set of coins whose total value is v . This might not be possible: for instance, if the denominations are 5 and 10 then we can make change for 15 but not for 12. Give an $O(nv)$ dynamic-programming algorithm for the following problem.

Input: $x_1, \dots, x_n; v$.

Question: Is it possible to make change for v using coins of denominations x_1, \dots, x_n ?

- 6.18. Consider the following variation on the change-making problem (Exercise 6.17): you are given denominations x_1, x_2, \dots, x_n , and you want to make change for a value v , but you are allowed to use each denomination *at most once*. For instance, if the denominations are 1, 5, 10, 20, then you can make change for $16 = 1 + 15$ and for $31 = 1 + 10 + 20$ but not for 40 (because you can't use 20 twice).

Figure 6.12 Two binary search trees for the keywords of a programming language.

Input: Positive integers x_1, x_2, \dots, x_n ; another integer v .

Output: Can you make change for v , using each denomination x_i at most once?

Show how to solve this problem in time $O(nv)$.

- 6.19. Here is yet another variation on the change-making problem (Exercise 6.17).

Given an unlimited supply of coins of denominations x_1, x_2, \dots, x_n , we wish to make change for a value v using at most k coins; that is, we wish to find a set of $\leq k$ coins whose total value is v . This might not be possible: for instance, if the denominations are 5 and 10 and $k = 6$, then we can make change for 55 but not for 65. Give an efficient dynamic-programming algorithm for the following problem.

Input: x_1, \dots, x_n ; k ; v .

Question: Is it possible to make change for v using at most k coins, of denominations x_1, \dots, x_n ?

- 6.20. *Optimal binary search trees.* Suppose we know the frequency with which keywords occur in programs of a certain language, for instance:

begin	5%
do	40%
else	8%
end	4%
if	10%
then	10%
while	23%

We want to organize them in a *binary search tree*, so that the keyword in the root is alphabetically bigger than all the keywords in the left subtree and smaller than all the keywords in the right subtree (and this holds for all nodes).

Figure 6.12 has a nicely-balanced example on the left. In this case, when a keyword is being looked up, the number of comparisons needed is at most three: for instance, in finding “while”, only the three nodes “end”, “then”, and “while” get examined. But since we know the frequency

with which keywords are accessed, we can use an even more fine-tuned cost function, the *average number of comparisons* to look up a word. For the search tree on the left, it is

$$\text{cost} = 1(0.04) + 2(0.40 + 0.10) + 3(0.05 + 0.08 + 0.10 + 0.23) = 2.42.$$

By this measure, the best search tree is the one on the right, which has a cost of 2.18.

Give an efficient algorithm for the following task.

Input: n words (in sorted order); frequencies of these words: p_1, p_2, \dots, p_n .

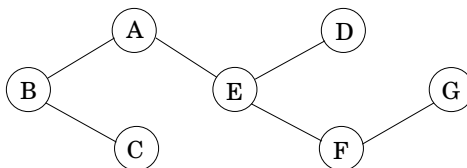
Output: The binary search tree of lowest cost (defined above as the expected number of comparisons in looking up a word).

- 6.21. A *vertex cover* of a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ that includes at least one endpoint of every edge in E . Give a linear-time algorithm for the following task.

Input: An undirected tree $T = (V, E)$.

Output: The size of the smallest vertex cover of T .

For instance, in the following tree, possible vertex covers include $\{A, B, C, D, E, F, G\}$ and $\{A, C, D, F\}$ but not $\{C, E, F\}$. The smallest vertex cover has size 3: $\{B, E, G\}$.



- 6.22. Give an $O(nt)$ algorithm for the following task.

Input: A list of n positive integers a_1, a_2, \dots, a_n ; a positive integer t .

Question: Does some subset of the a_i 's add up to t ? (You can use each a_i at most once.)

(*Hint:* Look at subproblems of the form “does a subset of $\{a_1, a_2, \dots, a_i\}$ add up to s ?”)

- 6.23. A mission-critical production system has n stages that have to be performed sequentially; stage i is performed by machine M_i . Each machine M_i has a probability r_i of functioning reliably and a probability $1 - r_i$ of failing (and the failures are independent). Therefore, if we implement each stage with a single machine, the probability that the whole system works is $r_1 \cdot r_2 \cdots r_n$. To improve this probability we add redundancy, by having m_i copies of the machine M_i that performs stage i . The probability that all m_i copies fail simultaneously is only $(1 - r_i)^{m_i}$, so the probability that stage i is completed correctly is $1 - (1 - r_i)^{m_i}$ and the probability that the whole system works is $\prod_{i=1}^n (1 - (1 - r_i)^{m_i})$. Each machine M_i has a cost c_i , and there is a total budget B to buy machines. (Assume that B and c_i are positive integers.)

Given the probabilities r_1, \dots, r_n , the costs c_1, \dots, c_n , and the budget B , find the redundancies m_1, \dots, m_n that are within the available budget and that maximize the probability that the system works correctly.

- 6.24. *Time and space complexity of dynamic programming.* Our dynamic programming algorithm for computing the edit distance between strings of length m and n creates a table of size $n \times m$ and therefore needs $O(mn)$ time and space. In practice, it will run out of space long before it runs out of time. How can this space requirement be reduced?

- (a) Show that if we just want to compute the value of the edit distance (rather than the optimal sequence of edits), then only $O(n)$ space is needed, because only a small portion of the table needs to be maintained at any given time.
- (b) Now suppose that we also want the optimal sequence of edits. As we saw earlier, this problem can be recast in terms of a corresponding grid-shaped dag, in which the goal is to find the optimal path from node $(0, 0)$ to node (n, m) . It will be convenient to work with this formulation, and while we're talking about convenience, we might as well also assume that m is a power of 2.

Let's start with a small addition to the edit distance algorithm that will turn out to be very useful. The optimal path in the dag must pass through an intermediate node $(k, m/2)$ for some k ; show how the algorithm can be modified to also return this value k .

- (c) Now consider a recursive scheme:

```

procedure find-path $((0, 0) \rightarrow (n, m))$ 
  compute the value  $k$  above
  find-path $((0, 0) \rightarrow (k, m/2))$ 
  find-path $((k, m/2) \rightarrow (n, m))$ 
  concatenate these two paths, with  $k$  in the middle

```

Show that this scheme can be made to run in $O(mn)$ time and $O(n)$ space.

- 6.25. Consider the following 3-PARTITION problem. Given integers a_1, \dots, a_n , we want to determine whether it is possible to partition of $\{1, \dots, n\}$ into three disjoint subsets I, J, K such that

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{1}{3} \sum_{i=1}^n a_i$$

For example, for input $(1, 2, 3, 4, 4, 5, 8)$ the answer is *yes*, because there is the partition $(1, 8)$, $(4, 5)$, $(2, 3, 4)$. On the other hand, for input $(2, 2, 3, 5)$ the answer is *no*.

Devise and analyze a dynamic programming algorithm for 3-PARTITION that runs in time polynomial in n and in $\sum_i a_i$.

- 6.26. *Sequence alignment.* When a new gene is discovered, a standard approach to understanding its function is to look through a database of known genes and find close matches. The closeness of two genes is measured by the extent to which they are *aligned*. To formalize this, think of a gene as being a long string over an alphabet $\Sigma = \{A, C, G, T\}$. Consider two genes (strings) $x = ATGCC$ and $y = TACGCA$. An alignment of x and y is a way of matching up these two strings by writing them in columns, for instance:

–	A	T	–	G	C	C
T	A	–	C	G	C	A

Here the “–” indicates a “gap.” The characters of each string must appear in order, and each column must contain a character from at least one of the strings. The score of an alignment is specified by a scoring matrix δ of size $(|\Sigma| + 1) \times (|\Sigma| + 1)$, where the extra row and column are to accommodate gaps. For instance the preceding alignment has the following score:

$$\delta(-, T) + \delta(A, A) + \delta(T, -) + \delta(-, C) + \delta(G, G) + \delta(C, C) + \delta(C, A).$$

Give a dynamic programming algorithm that takes as input two strings $x[1 \dots n]$ and $y[1 \dots m]$ and a scoring matrix δ , and returns the highest-scoring alignment. The running time should be $O(mn)$.

- 6.27. *Alignment with gap penalties.* The alignment algorithm of Exercise 6.26 helps to identify DNA sequences that are close to one another. The discrepancies between these closely matched sequences are often caused by errors in DNA replication. However, a closer look at the biological replication process reveals that the scoring function we considered earlier has a qualitative problem: nature often inserts or removes entire substrings of nucleotides (creating long gaps), rather than editing just one position at a time. Therefore, the penalty for a gap of length 10 should not be 10 times the penalty for a gap of length 1, but something significantly smaller.

Repeat Exercise 6.26, but this time use a modified scoring function in which the penalty for a gap of length k is $c_0 + c_1k$, where c_0 and c_1 are given constants (and c_0 is larger than c_1).

- 6.28. *Local sequence alignment.* Often two DNA sequences are significantly different, but contain regions that are very similar and are *highly conserved*. Design an algorithm that takes an input two strings $x[1 \dots n]$ and $y[1 \dots m]$ and a scoring matrix δ (as defined in Exercise 6.26), and outputs substrings x' and y' of x and y , respectively, that have the highest-scoring alignment over all pairs of such substrings. Your algorithm should take time $O(mn)$.

- 6.29. *Exon chaining.* Each gene corresponds to a subregion of the overall genome (the DNA sequence); however, part of this region might be “junk DNA.” Frequently, a gene consists of several pieces called exons, which are separated by junk fragments called introns. This complicates the process of identifying genes in a newly sequenced genome.

Suppose we have a new DNA sequence and we want to check whether a certain gene (a string) is present in it. Because we cannot hope that the gene will be a contiguous subsequence, we look for partial matches—fragments of the DNA that are also present in the gene (actually, even these partial matches will be approximate, not perfect). We then attempt to assemble these fragments.

Let $x[1 \dots n]$ denote the DNA sequence. Each partial match can be represented by a triple (l_i, r_i, w_i) , where $x[l_i \dots r_i]$ is the fragment and w_i is a weight representing the strength of the match (it might be a local alignment score or some other statistical quantity). Many of these potential matches could be false, so the goal is to find a subset of the triples that are consistent (nonoverlapping) and have a maximum total weight.

Show how to do this efficiently.

- 6.30. *Reconstructing evolutionary trees by maximum parsimony.* Suppose we manage to sequence a particular gene across a whole bunch of different species. For concreteness, say there are n species, and the sequences are strings of length k over alphabet $\Sigma = \{A, C, G, T\}$. How can we use this information to reconstruct the evolutionary history of these species?

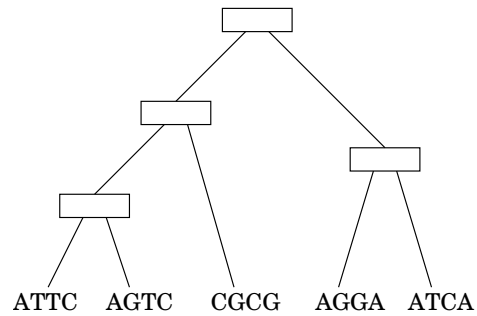
Evolutionary history is commonly represented by a tree whose leaves are the different species, whose root is their common ancestor, and whose internal branches represent speciation events (that is, moments when a new species broke off from an existing one). Thus we need to find the following:

- An evolutionary tree with the given species at the leaves.
- For each internal node, a string of length k : the gene sequence for that particular ancestor.

For each possible tree T , annotated with sequences $s(u) \in \Sigma^k$ at each of its nodes u , we can assign a score based on the principle of *parsimony*: fewer mutations are more likely.

$$\text{score}(T) = \sum_{(u,v) \in E(T)} (\text{number of positions on which } s(u) \text{ and } s(v) \text{ disagree}).$$

Finding the highest-score tree is a difficult problem. Here we will consider just a small part of it: suppose we know the structure of the tree, and we want to fill in the sequences $s(u)$ of the internal nodes u . Here's an example with $k = 4$ and $n = 5$:



- (a) In this particular example, there are several maximum parsimony reconstructions of the internal node sequences. Find one of them.
- (b) Give an efficient (in terms of n and k) algorithm for this task. (*Hint:* Even though the sequences might be long, you can do just one position at a time.)