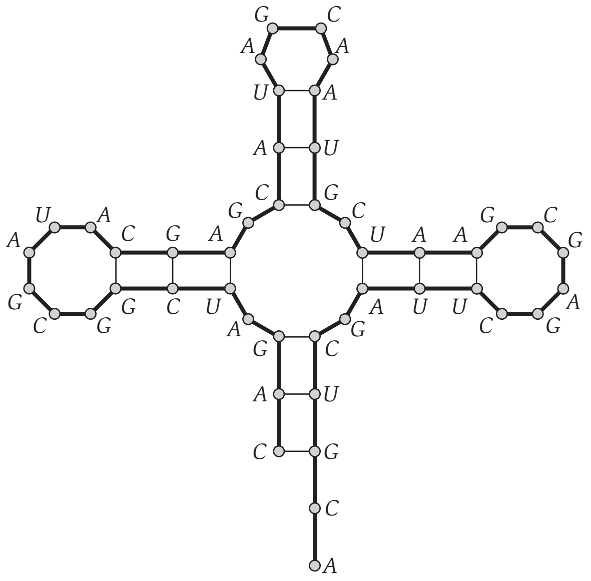


Figure 6.13 An RNA secondary structure. Thick lines connect adjacent elements of the sequence; thin lines indicate pairs of elements that are matched.

The Problem

As one learns in introductory biology classes, Watson and Crick posited that double-stranded DNA is "zipped" together by complementary base-pairing. Each strand of DNA can be viewed as a string of *bases*, where each base is drawn from the set $\{A, C, G, T\}$.[2] The bases A and T pair with each other, and the bases C and G pair with each other; it is these A-T and C-G pairings that hold the two strands together.

Now, single-stranded RNA molecules are key components in many of the processes that go on inside a cell, and they follow more or less the same structural principles. However, unlike double-stranded DNA, there's no "second strand" for the RNA to stick to; so it tends to loop back and form base pairs with itself, resulting in interesting shapes like the one depicted in Figure 6.13. The set of pairs (and resulting shape) formed by the RNA molecule through this process is called the *secondary structure*, and understanding the secondary structure is essential for understanding the behavior of the molecule.

[2] Adenine, cytosine, guanine, and thymine, the four basic units of DNA.

For our purposes, a single-stranded RNA molecule can be viewed as a sequence of n symbols (bases) drawn from the alphabet $\{A, C, G, U\}$.[3] Let $B = b_1 b_2 \cdots b_n$ be a single-stranded RNA molecule, where each $b_i \in \{A, C, G, U\}$. To a first approximation, one can model its secondary structure as follows. As usual, we require that A pairs with U, and C pairs with G; we also require that each base can pair with at most one other base—in other words, the set of base pairs forms a *matching*. It also turns out that secondary structures are (again, to a first approximation) "knot-free," which we will formalize as a kind of *noncrossing* condition below.

Thus, concretely, we say that a *secondary structure on B* is a set of pairs $S = \{(i, j)\}$, where $i, j \in \{1, 2, \ldots, n\}$, that satisfies the following conditions.

(i) *(No sharp turns.)* The ends of each pair in S are separated by at least four intervening bases; that is, if $(i, j) \in S$, then $i < j - 4$.

(ii) The elements of any pair in S consist of either $\{A, U\}$ or $\{C, G\}$ (in either order).

(iii) S is a matching: no base appears in more than one pair.

(iv) *(The noncrossing condition.)* If (i, j) and (k, ℓ) are two pairs in S, then we cannot have $i < k < j < \ell$. (See Figure 6.14 for an illustration.)

Note that the RNA secondary structure in Figure 6.13 satisfies properties (i) through (iv). From a structural point of view, condition (i) arises simply because the RNA molecule cannot bend too sharply; and conditions (ii) and (iii) are the fundamental Watson-Crick rules of base-pairing. Condition (iv) is the striking one, since it's not obvious why it should hold in nature. But while there are sporadic exceptions to it in real molecules (via so-called *pseudo-knotting*), it does turn out to be a good approximation to the spatial constraints on real RNA secondary structures.

Now, out of all the secondary structures that are possible for a single RNA molecule, which are the ones that are likely to arise under physiological conditions? The usual hypothesis is that a single-stranded RNA molecule will form the secondary structure with the optimum total free energy. The correct model for the free energy of a secondary structure is a subject of much debate; but a first approximation here is to assume that the free energy of a secondary structure is proportional simply to the *number* of base pairs that it contains.

Thus, having said all this, we can state the basic RNA secondary structure prediction problem very simply: We want an efficient algorithm that takes

[3] Note that the symbol T from the alphabet of DNA has been replaced by a U, but this is not important for us here.

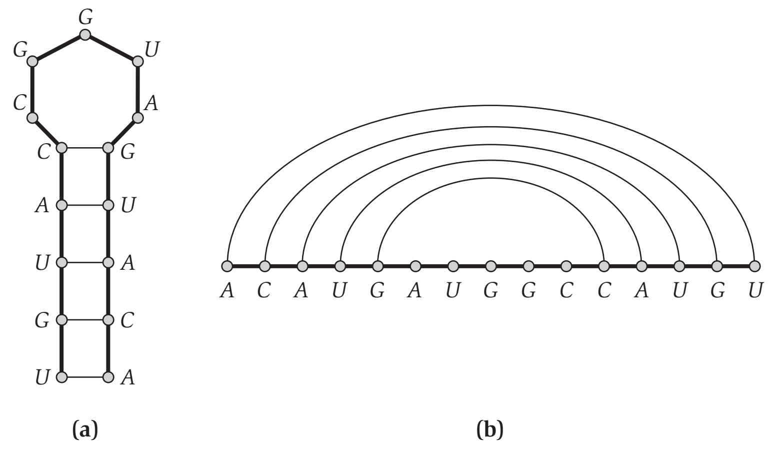


Figure 6.14 Two views of an RNA secondary structure. In the second view, (b), the string has been "stretched" lengthwise, and edges connecting matched pairs appear as noncrossing "bubbles" over the string.

a single-stranded RNA molecule $B = b_1 b_2 \cdots b_n$ and determines a secondary structure S with the maximum possible number of base pairs.

Designing and Analyzing the Algorithm

A First Attempt at Dynamic Programming The natural first attempt to apply dynamic programming would presumably be based on the following subproblems: We say that $\text{OPT}(j)$ is the maximum number of base pairs in a secondary structure on $b_1 b_2 \cdots b_j$. By the no-sharp-turns condition above, we know that $\text{OPT}(j) = 0$ for $j \le 5$; and we know that $\text{OPT}(n)$ is the solution we're looking for.

The trouble comes when we try writing down a recurrence that expresses $\text{OPT}(j)$ in terms of the solutions to smaller subproblems. We can get partway there: in the optimal secondary structure on $b_1 b_2 \cdots b_j$, it's the case that either

- j is not involved in a pair; or
- j pairs with t for some $t < j - 4$.

In the first case, we just need to consult our solution for $\text{OPT}(j-1)$. The second case is depicted in Figure 6.15(a); because of the noncrossing condition, we now know that no pair can have one end between 1 and $t-1$ and the other end between $t+1$ and $j-1$. We've therefore effectively isolated two new subproblems: one on the bases $b_1 b_2 \cdots b_{t-1}$, and the other on the bases $b_{t+1} \cdots b_{j-1}$. The first is solved by $\text{OPT}(t-1)$, but the second is not on our list of subproblems, because it does not begin with b_1.

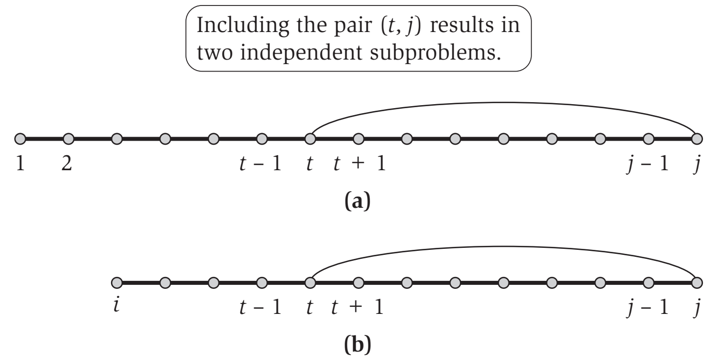

Including the pair (t, j) results in two independent subproblems.
1
2
$t-1$
t
$t+1$
$j-1$
j
(a)
i
$t-1$
t
$t+1$
$j-1$
j
(b)

Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

This is the insight that makes us realize we need to add a variable. We need to be able to work with subproblems that do not begin with b_1; in other words, we need to consider subproblems on $b_i b_{i+1} \cdots b_j$ for all choices of $i \le j$.

Dynamic Programming over Intervals Once we make this decision, our previous reasoning leads straight to a successful recurrence. Let $\text{OPT}(i, j)$ denote the maximum number of base pairs in a secondary structure on $b_i b_{i+1} \cdots b_j$. The no-sharp-turns condition lets us initialize $\text{OPT}(i, j) = 0$ whenever $i \ge j - 4$. (For notational convenience, we will also allow ourselves to refer to $\text{OPT}(i, j)$ even when $i > j$; in this case, its value is 0.)

Now, in the optimal secondary structure on $b_i b_{i+1} \cdots b_j$, we have the same alternatives as before:

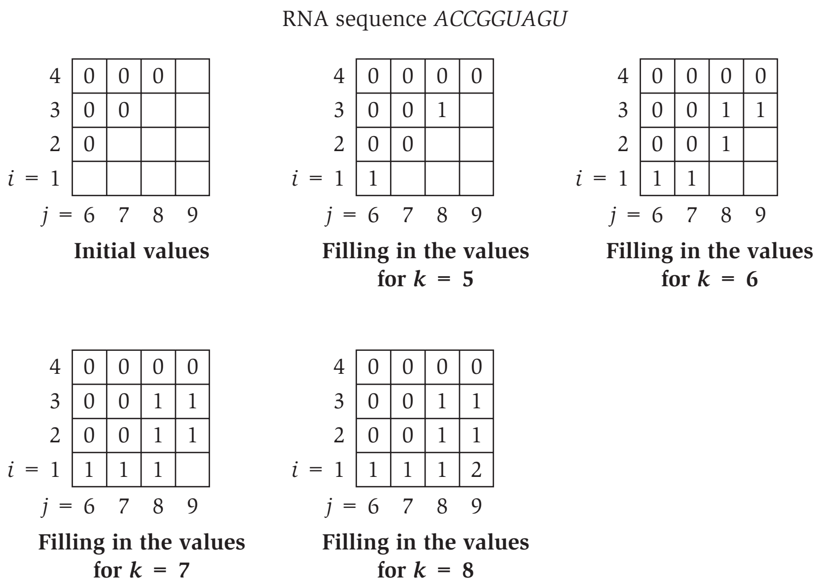
- j is not involved in a pair; or
- j pairs with t for some $t < j - 4$.

In the first case, we have $\text{OPT}(i, j) = \text{OPT}(i, j-1)$. In the second case, depicted in Figure 6.15(b), we recur on the two subproblems $\text{OPT}(i, t-1)$ and $\text{OPT}(t+1, j-1)$; as argued above, the noncrossing condition has isolated these two subproblems from each other.

We have therefore justified the following recurrence.

(6.13) $\text{OPT}(i, j) = \max(\text{OPT}(i, j-1), \max(1 + \text{OPT}(i, t-1) + \text{OPT}(t+1, j-1)))$, *where the* max *is taken over t such that b_t and b_j are an allowable base pair (under conditions (i) and (ii) from the definition of a secondary structure).*

Now we just have to make sure we understand the proper order in which to build up the solutions to the subproblems. The form of (6.13) reveals that we're always invoking the solution to subproblems on *shorter* intervals: those



RNA sequence *ACCGGUAGU*

Initial values

4	0	0	0	
3	0	0		
2	0			
i = 1				
	j = 6	7	8	9

Filling in the values for k = 5

4	0	0	0	0
3	0	0	1	
2	0	0		
i = 1	1			
	j = 6	7	8	9

Filling in the values for k = 6

4	0	0	0	0
3	0	0	1	1
2	0	0	1	
i = 1	1	1		
	j = 6	7	8	9

Filling in the values for k = 7

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
i = 1	1	1	1	
	j = 6	7	8	9

Filling in the values for k = 8

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
i = 1	1	1	1	2
	j = 6	7	8	9

Figure 6.16 The iterations of the algorithm on a sample instance of the RNA Secondary Structure Prediction Problem.

for which $k = j - i$ is smaller. Thus things will work without any trouble if we build up the solutions in order of increasing interval length.

```
Initialize OPT$(i, j) = 0$ whenever $i \geq j - 4$
For $k = 5,\ 6, \ldots, n-1$
  For $i = 1, 2, \ldots n - k$
    Set $j = i + k$
    Compute OPT$(i, j)$ using the recurrence in (6.13)
  Endfor
Endfor
Return OPT$(1, n)$
```

As an example of this algorithm executing, we consider the input *ACCGGUAGU*, a subsequence of the sequence in Figure 6.14. As with the Knapsack Problem, we need two dimensions to depict the array M: one for the left endpoint of the interval being considered, and one for the right endpoint. In the figure, we only show entries corresponding to $[i, j]$ pairs with $i < j - 4$, since these are the only ones that can possibly be nonzero.

It is easy to bound the running time: there are $O(n^2)$ subproblems to solve, and evaluating the recurrence in (6.13) takes time $O(n)$ for each. Thus the running time is $O(n^3)$.

As always, we can recover the secondary structure itself (not just its value) by recording how the minima in (6.13) are achieved and tracing back through the computation.

6.6 Sequence Alignment

For the remainder of this chapter, we consider two further dynamic programming algorithms that each have a wide range of applications. In the next two sections we discuss *sequence alignment*, a fundamental problem that arises in comparing strings. Following this, we turn to the problem of computing shortest paths in graphs when edges have costs that may be negative.

The Problem

Dictionaries on the Web seem to get more and more useful: often it seems easier to pull up a bookmarked online dictionary than to get a physical dictionary down from the bookshelf. And many online dictionaries offer functions that you can't get from a printed one: if you're looking for a definition and type in a word it doesn't contain—say, *ocurrance*—it will come back and ask, "Perhaps you mean *occurrence*?" How does it do this? Did it truly know what you had in mind?

Let's defer the second question to a different book and think a little about the first one. To decide what you probably meant, it would be natural to search the dictionary for the word most "similar" to the one you typed in. To do this, we have to answer the question: How should we define similarity between two words or strings?

Intuitively, we'd like to say that *ocurrance* and *occurrence* are similar because we can make the two words identical if we add a *c* to the first word and change the *a* to an *e*. Since neither of these changes seems so large, we conclude that the words are quite similar. To put it another way, we can *nearly* line up the two words letter by letter:

```
o-currance
occurrence
```

The hyphen (-) indicates a *gap* where we had to add a letter to the second word to get it to line up with the first. Moreover, our lining up is not perfect in that an *e* is lined up with an *a*.

We want a model in which similarity is determined roughly by the number of gaps and mismatches we incur when we line up the two words. Of course, there are many possible ways to line up the two words; for example, we could have written

```
o-curr-ance
occurre-nce
```

which involves three gaps and no mismatches. Which is better: one gap and one mismatch, or three gaps and no mismatches?

This discussion has been made easier because we know roughly what the correspondence ought to look like. When the two strings don't look like English words—for example, abbbaabbbbaab and ababaaabbbbbab—it may take a little work to decide whether they can be lined up nicely or not:

```
abbbaa--bbbbaab
ababaaabbbbba-b
```

Dictionary interfaces and spell-checkers are not the most computationally intensive application for this type of problem. In fact, determining similarities among strings is one of the central computational problems facing molecular biologists today.

Strings arise very naturally in biology: an organism's *genome*—its full set of genetic material—is divided up into giant linear DNA molecules known as *chromosomes,* each of which serves conceptually as a one-dimensional chemical storage device. Indeed, it does not obscure reality very much to think of it as an enormous linear *tape*, containing a string over the alphabet $\{A, C, G, T\}$. The string of symbols encodes the instructions for building protein molecules; using a chemical mechanism for reading portions of the chromosome, a cell can construct proteins that in turn control its metabolism.

Why is similarity important in this picture? To a first approximation, the sequence of symbols in an organism's genome can be viewed as determining the properties of the organism. So suppose we have two strains of bacteria, X and Y, which are closely related evolutionarily. Suppose further that we've determined that a certain substring in the DNA of X codes for a certain kind of toxin. Then, if we discover a very "similar" substring in the DNA of Y, we might be able to hypothesize, before performing any experiments at all, that this portion of the DNA in Y codes for a similar kind of toxin. This use of computation to guide decisions about biological experiments is one of the hallmarks of the field of *computational biology*.

All this leaves us with the same question we asked initially, while typing badly spelled words into our online dictionary: How should we define the notion of *similarity* between two strings?

In the early 1970s, the two molecular biologists Needleman and Wunsch proposed a definition of similarity, which, basically unchanged, has become the standard definition in use today. Its position as a standard was reinforced by its simplicity and intuitive appeal, as well as through its independent discovery by several other researchers around the same time. Moreover, this definition of similarity came with an efficient dynamic programming algorithm to compute it. In this way, the paradigm of dynamic programming was independently discovered by biologists some twenty years after mathematicians and computer scientists first articulated it.

The definition is motivated by the considerations we discussed above, and in particular by the notion of "lining up" two strings. Suppose we are given two strings X and Y, where X consists of the sequence of symbols $x_1x_2 \cdots x_m$ and Y consists of the sequence of symbols $y_1y_2 \cdots y_n$. Consider the sets $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ as representing the different positions in the strings X and Y, and consider a matching of these sets; recall that a *matching* is a set of ordered pairs with the property that each item occurs in at most one pair. We say that a matching M of these two sets is an *alignment* if there are no "crossing" pairs: if $(i, j), (i', j') \in M$ and $i < i'$, then $j < j'$. Intuitively, an alignment gives a way of lining up the two strings, by telling us which pairs of positions will be lined up with one another. Thus, for example,

```
stop-
-tops
```

corresponds to the alignment $\{(2, 1), (3, 2), (4, 3)\}$.

Our definition of similarity will be based on finding the *optimal* alignment between X and Y, according to the following criteria. Suppose M is a given alignment between X and Y.

- First, there is a parameter $\delta > 0$ that defines a *gap penalty*. For each position of X or Y that is not matched in M—it is a *gap*—we incur a cost of δ.
- Second, for each pair of letters p, q in our alphabet, there is a *mismatch cost* of α_{pq} for lining up p with q. Thus, for each $(i, j) \in M$, we pay the appropriate mismatch cost $\alpha_{x_iy_j}$ for lining up x_i with y_j. One generally assumes that $\alpha_{pp} = 0$ for each letter p—there is no mismatch cost to line up a letter with another copy of itself—although this will not be necessary in anything that follows.
- The *cost* of M is the sum of its gap and mismatch costs, and we seek an alignment of minimum cost.

The process of minimizing this cost is often referred to as *sequence alignment* in the biology literature. The quantities δ and $\{\alpha_{pq}\}$ are external parameters that must be plugged into software for sequence alignment; indeed, a lot of work goes into choosing the settings for these parameters. From our point of view, in designing an algorithm for sequence alignment, we will take them as given. To go back to our first example, notice how these parameters determine which alignment of *ocurrance* and *occurrence* we should prefer: the first is strictly better if and only if $\delta + \alpha_{ae} < 3\delta$.

Designing the Algorithm

We now have a concrete numerical definition for the similarity between strings X and Y: it is the minimum cost of an alignment between X and Y. The lower this cost, the more similar we declare the strings to be. We now turn to the problem of computing this minimum cost, and an optimal alignment that yields it, for a given pair of strings X and Y.

One of the approaches we could try for this problem is dynamic programming, and we are motivated by the following basic dichotomy.

- In the optimal alignment M, either $(m, n) \in M$ or $(m, n) \notin M$. (That is, either the last symbols in the two strings are matched to each other, or they aren't.)

By itself, this fact would be too weak to provide us with a dynamic programming solution. Suppose, however, that we compound it with the following basic fact.

(6.14) *Let M be any alignment of X and Y. If $(m, n) \notin M$, then either the m^{th} position of X or the n^{th} position of Y is not matched in M.*

Proof. Suppose by way of contradiction that $(m, n) \notin M$, and there are numbers $i < m$ and $j < n$ so that $(m, j) \in M$ and $(i, n) \in M$. But this contradicts our definition of *alignment*: we have $(i, n), (m, j) \in M$ with $i < m$, but $n > i$ so the pairs (i, n) and (m, j) cross. ■

There is an equivalent way to write (6.14) that exposes three alternative possibilities, and leads directly to the formulation of a recurrence.

(6.15) *In an optimal alignment M, at least one of the following is true:*

(i) $(m, n) \in M$; or

(ii) the m^{th} position of X is not matched; or

(iii) the n^{th} position of Y is not matched.

Now, let $\text{OPT}(i, j)$ denote the minimum cost of an alignment between $x_1x_2 \cdots x_i$ and $y_1y_2 \cdots y_j$. If case (i) of (6.15) holds, we pay $\alpha_{x_my_n}$ and then align $x_1x_2 \cdots x_{m-1}$ as well as possible with $y_1y_2 \cdots y_{n-1}$; we get $\text{OPT}(m, n) = \alpha_{x_my_n} + \text{OPT}(m-1, n-1)$. If case (ii) holds, we pay a gap cost of δ since the m^{th} position of X is not matched, and then we align $x_1x_2 \cdots x_{m-1}$ as well as possible with $y_1y_2 \cdots y_n$. In this way, we get $\text{OPT}(m, n) = \delta + \text{OPT}(m-1, n)$. Similarly, if case (iii) holds, we get $\text{OPT}(m, n) = \delta + \text{OPT}(m, n-1)$.

Using the same argument for the subproblem of finding the minimum-cost alignment between $x_1x_2 \cdots x_i$ and $y_1y_2 \cdots y_j$, we get the following fact.

(6.16) *The minimum alignment costs satisfy the following recurrence for $i \geq 1$ and $j \geq 1$:*

$$\text{OPT}(i, j) = \min[\alpha_{x_iy_j} + \text{OPT}(i-1, j-1), \delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1)].$$

Moreover, (i, j) is in an optimal alignment M for this subproblem if and only if the minimum is achieved by the first of these values.

We have maneuvered ourselves into a position where the dynamic programming algorithm has become clear: We build up the values of $\text{OPT}(i, j)$ using the recurrence in (6.16). There are only $O(mn)$ subproblems, and $\text{OPT}(m, n)$ is the value we are seeking.

We now specify the algorithm to compute the value of the optimal alignment. For purposes of initialization, we note that $\text{OPT}(i, 0) = \text{OPT}(0, i) = i\delta$ for all i, since the only way to line up an i-letter word with a 0-letter word is to use i gaps.

```
Alignment($X$,$Y$)
  Array $A[0 \ldots m, 0 \ldots n]$
  Initialize $A[i, 0] = i\delta$ for each $i$
  Initialize $A[0, j] = j\delta$ for each $j$
  For $j = 1, \ldots, n$
     For $i = 1, \ldots, m$
         Use the recurrence (6.16) to compute $A[i, j]$
     Endfor
  Endfor
  Return $A[m, n]$
```

As in previous dynamic programming algorithms, we can trace back through the array A, using the second part of fact (6.16), to construct the alignment itself.

Analyzing the Algorithm

The correctness of the algorithm follows directly from (6.16). The running time is $O(mn)$, since the array A has $O(mn)$ entries, and at worst we spend constant time on each.

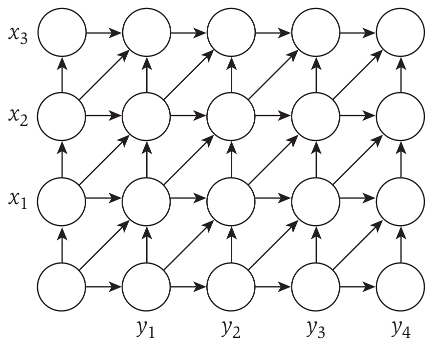


Figure 6.17 A graph-based picture of sequence alignment.

There is an appealing pictorial way in which people think about this sequence alignment algorithm. Suppose we build a two-dimensional $m \times n$ grid graph G_{XY}, with the rows labeled by symbols in the string X, the columns labeled by symbols in Y, and directed edges as in Figure 6.17.

We number the rows from 0 to m and the columns from 0 to n; we denote the node in the i^{th} row and the j^{th} column by the label (i, j). We put *costs* on the edges of G_{XY}: the cost of each horizontal and vertical edge is δ, and the cost of the diagonal edge from $(i-1, j-1)$ to (i, j) is $\alpha_{x_i y_j}$.

The purpose of this picture now emerges: the recurrence in (6.16) for $\text{OPT}(i, j)$ is precisely the recurrence one gets for the minimum-cost path in G_{XY} from $(0, 0)$ to (i, j). Thus we can show

(6.17) *Let $f(i, j)$ denote the minimum cost of a path from $(0, 0)$ to (i, j) in G_{XY}. Then for all i, j, we have $f(i, j) = \text{OPT}(i, j)$.*

Proof. We can easily prove this by induction on $i + j$. When $i + j = 0$, we have $i = j = 0$, and indeed $f(i, j) = \text{OPT}(i, j) = 0$.

Now consider arbitrary values of i and j, and suppose the statement is true for all pairs (i', j') with $i' + j' < i + j$. The last edge on the shortest path to (i, j) is either from $(i-1, j-1)$, $(i-1, j)$, or $(i, j-1)$. Thus we have

$$\begin{aligned} f(i,j) &= \min[\alpha_{x_i y_j} + f(i-1, j-1), \delta + f(i-1, j), \delta + f(i, j-1)] \\ &= \min[\alpha_{x_i y_j} + \text{OPT}(i-1, j-1), \delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1)] \\ &= \text{OPT}(i, j), \end{aligned}$$

where we pass from the first line to the second using the induction hypothesis, and we pass from the second to the third using (6.16). ■

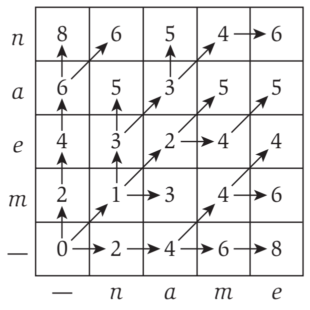


Figure 6.18 The OPT values for the problem of aligning the words *mean* to *name*.

Thus the value of the optimal alignment is the length of the shortest path in G_{XY} from $(0, 0)$ to (m, n). (We'll call any path in G_{XY} from $(0, 0)$ to (m, n) a *corner-to-corner path*.) Moreover, the diagonal edges used in a shortest path correspond precisely to the pairs used in a minimum-cost alignment. These connections to the Shortest-Path Problem in the graph G_{XY} do not directly yield an improvement in the running time for the sequence alignment problem; however, they do help one's intuition for the problem and have been useful in suggesting algorithms for more complex variations on sequence alignment.

For an example, Figure 6.18 shows the value of the shortest path from $(0, 0)$ to each node (i, j) for the problem of aligning the words *mean* and *name*. For the purpose of this example, we assume that $\delta = 2$; matching a vowel with a different vowel, or a consonant with a different consonant, costs 1; while matching a vowel and a consonant with each other costs 3. For each cell in the table (representing the corresponding node), the arrow indicates the last step of the shortest path leading to that node—in other words, the way that the minimum is achieved in (6.16). Thus, by following arrows backward from node $(4, 4)$, we can trace back to construct the alignment.

6.7 Sequence Alignment in Linear Space via Divide and Conquer

In the previous section, we showed how to compute the optimal alignment between two strings X and Y of lengths m and n, respectively. Building up the two-dimensional m-by-n array of optimal solutions to subproblems, OPT$(\cdot, \cdot)$, turned out to be equivalent to constructing a graph G_{XY} with mn nodes laid out in a grid and looking for the cheapest path between opposite corners. In either of these ways of formulating the dynamic programming algorithm, the running time is $O(mn)$, because it takes constant time to determine the value in each of the mn cells of the array OPT; and the space requirement is $O(mn)$ as well, since it was dominated by the cost of storing the array (or the graph G_{XY}).

The Problem

The question we ask in this section is: Should we be happy with $O(mn)$ as a space bound? If our application is to compare English words, or even English sentences, it is quite reasonable. In biological applications of sequence alignment, however, one often compares very long strings against one another; and in these cases, the $\Theta(mn)$ space requirement can potentially be a more severe problem than the $\Theta(mn)$ time requirement. Suppose, for example, that we are comparing two strings of 100,000 symbols each. Depending on the underlying processor, the prospect of performing roughly 10 billion primitive operations might be less cause for worry than the prospect of working with a single 10-gigabyte array.

Fortunately, this is not the end of the story. In this section we describe a very clever enhancement of the sequence alignment algorithm that makes it work in $O(mn)$ time using only $O(m+n)$ space. In other words, we can bring the space requirement down to linear while blowing up the running time by at most an additional constant factor. For ease of presentation, we'll describe various steps in terms of paths in the graph G_{XY}, with the natural equivalence back to the sequence alignment problem. Thus, when we seek the pairs in an optimal alignment, we can equivalently ask for the edges in a shortest corner-to-corner path in G_{XY}.

The algorithm itself will be a nice application of divide-and-conquer ideas. The crux of the technique is the observation that, if we divide the problem into several recursive calls, then the space needed for the computation can be reused from one call to the next. The way in which this idea is used, however, is fairly subtle.

Designing the Algorithm

We first show that if we only care about the *value* of the optimal alignment, and not the alignment itself, it is easy to get away with linear space. The crucial observation is that to fill in an entry of the array A, the recurrence in (6.16) only needs information from the current column of A and the previous column of A. Thus we will "collapse" the array A to an $m \times 2$ array B: as the algorithm iterates through values of j, entries of the form $B[i, 0]$ will hold the "previous" column's value $A[i, j-1]$, while entries of the form $B[i, 1]$ will hold the "current" column's value $A[i, j]$.

```
Space-Efficient-Alignment($X$,$Y$)
  Array $B[0 \ldots m, 0 \ldots 1]$
  Initialize $B[i,0] = i\delta$ for each $i$ (just as in column 0 of $A$)
  For $j = 1, \ldots, n$
     $B[0,1] = j\delta$ (since this corresponds to entry $A[0,j]$)
     For $i = 1, \ldots, m$
         $B[i,1] = \min[\alpha_{x_i y_j} + B[i-1,0],$
                       $\delta + B[i-1,1], \quad \delta + B[i,0]]$
     Endfor
     Move column 1 of $B$ to column 0 to make room for next iteration:
         Update $B[i,0] = B[i,1]$ for each $i$
  Endfor
```

It is easy to verify that when this algorithm completes, the array entry $B[i, 1]$ holds the value of $\text{OPT}(i, n)$ for $i = 0, 1, \ldots, m$. Moreover, it uses $O(mn)$ time and $O(m)$ space. The problem is: where is the alignment itself? We haven't left enough information around to be able to run a procedure like `Find-Alignment`. Since B at the end of the algorithm only contains the last two columns of the original dynamic programming array A, if we were to try tracing back to get the path, we'd run out of information after just these two columns. We could imagine getting around this difficulty by trying to "predict" what the alignment is going to be in the process of running our space-efficient procedure. In particular, as we compute the values in the j^{th} column of the (now implicit) array A, we could try hypothesizing that a certain entry has a very small value, and hence that the alignment that passes through this entry is a promising candidate to be the optimal one. But this promising alignment might run into big problems later on, and a different alignment that currently looks much less attractive could turn out to be the optimal one.

There is, in fact, a solution to this problem—we will be able to recover the alignment itself using $O(m + n)$ space—but it requires a genuinely new idea. The insight is based on employing the divide-and-conquer technique that we've seen earlier in the book. We begin with a simple alternative way to implement the basic dynamic programming solution.

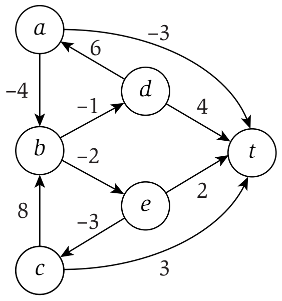
A Backward Formulation of the Dynamic Program Recall that we use $f(i, j)$ to denote the length of the shortest path from $(0, 0)$ to (i, j) in the graph G_{XY}. (As we showed in the initial sequence alignment algorithm, $f(i, j)$ has the same value as $\text{OPT}(i, j)$.) Now let's define $g(i, j)$ to be the length of the shortest path from (i, j) to (m, n) in G_{XY}. The function g provides an equally natural dynamic programming approach to sequence alignment, except that we build it up in reverse: we start with $g(m, n) = 0$, and the answer we want is $g(0, 0)$. By strict analogy with (6.16), we have the following recurrence for g.

(6.18) *For $i < m$ and $j < n$ we have*

$$g(i, j) = \min[\alpha_{x_{i+1}y_{j+1}} + g(i + 1, j + 1), \delta + g(i, j + 1), \delta + g(i + 1, j)].$$

This is just the recurrence one obtains by taking the graph G_{XY}, "rotating" it so that the node (m, n) is in the lower left corner, and using the previous approach. Using this picture, we can also work out the full dynamic programming algorithm to build up the values of g, *backward* starting from (m, n). Similarly, there is a space-efficient version of this backward dynamic programming algorithm, analogous to `Space-Efficient-Alignment`, which computes the value of the optimal alignment using only $O(m + n)$ space. We will refer to

```
Shortest-Path($G, s, t$)
  $n=$ number of nodes in $G$
  Array $M[0 \ldots n-1, V]$
  Define $M[0,t]=0$ and $M[0,v]=\infty$ for all other $v \in V$
  For $i=1, \ldots, n-1$
    For $v \in V$ in any order
      Compute $M[i,v]$ using the recurrence (6.23)
    Endfor
  Endfor
  Return $M[n-1,s]$
```



(a)

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	−3	−3	−4	−6	−6
b	∞	∞	0	−2	−2	−2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

(b)

Figure 6.23 For the directed graph in (a), the Shortest-Path Algorithm constructs the dynamic programming table in (b).

The correctness of the method follows directly by induction from (6.23). We can bound the running time as follows. The table M has n^2 entries; and each entry can take $O(n)$ time to compute, as there are at most n nodes $w \in V$ we have to consider.

(6.24) *The* Shortest-Path *method correctly computes the minimum cost of an s-t path in any graph that has no negative cycles, and runs in $O(n^3)$ time.*

Given the table M containing the optimal values of the subproblems, the shortest path using at most i edges can be obtained in $O(in)$ time, by tracing back through smaller subproblems.

As an example, consider the graph in Figure 6.23(a), where the goal is to find a shortest path from each node to t. The table in Figure 6.23(b) shows the array M, with entries corresponding to the values $M[i, v]$ from the algorithm. Thus a single row in the table corresponds to the shortest path from a particular node to t, as we allow the path to use an increasing number of edges. For example, the shortest path from node d to t is updated four times, as it changes from d-t, to d-a-t, to d-a-b-e-t, and finally to d-a-b-e-c-t.

Extensions: Some Basic Improvements to the Algorithm

An Improved Running-Time Analysis We can actually provide a better running-time analysis for the case in which the graph G does not have too many edges. A directed graph with n nodes can have close to n^2 edges, since there could potentially be an edge between each pair of nodes, but many graphs are much sparser than this. When we work with a graph for which the number of edges m is significantly less than n^2, we've already seen in a number of cases earlier in the book that it can be useful to write the running-time in terms of both m and n; this way, we can quantify our speed-up on graphs with relatively fewer edges.

If we are a little more careful in the analysis of the method above, we can improve the running-time bound to $O(mn)$ without significantly changing the algorithm itself.

(6.25) *The* `Shortest-Path` *method can be implemented in* $O(mn)$ *time.*

Proof. Consider the computation of the array entry $M[i, v]$ according to the recurrence (6.23); we have

$$M[i, v] = \min(M[i-1, v], \min_{w \in V}(M[i-1, w] + c_{vw})).$$

We assumed it could take up to $O(n)$ time to compute this minimum, since there are n possible nodes w. But, of course, we need only compute this minimum over all nodes w for which v has an edge to w; let us use n_v to denote this number. Then it takes time $O(n_v)$ to compute the array entry $M[i, v]$. We have to compute an entry for every node v and every index $0 \le i \le n-1$, so this gives a running-time bound of

$$O\left(n \sum_{v \in V} n_v\right).$$

In Chapter 3, we performed exactly this kind of analysis for other graph algorithms, and used (3.9) from that chapter to bound the expression $\sum_{v \in V} n_v$ for undirected graphs. Here we are dealing with directed graphs, and n_v denotes the number of edges *leaving* v. In a sense, it is even easier to work out the value of $\sum_{v \in V} n_v$ for the directed case: each edge leaves exactly one of the nodes in V, and so each edge is counted exactly once by this expression. Thus we have $\sum_{v \in V} n_v = m$. Plugging this into our expression

$$O\left(n \sum_{v \in V} n_v\right)$$

for the running time, we get a running-time bound of $O(mn)$. ■

Improving the Memory Requirements We can also significantly improve the memory requirements with only a small change to the implementation. A common problem with many dynamic programming algorithms is the large space usage, arising from the M array that needs to be stored. In the Bellman-Ford Algorithm as written, this array has size n^2; however, we now show how to reduce this to $O(n)$. Rather than recording $M[i, v]$ for each value i, we will use and update a single value $M[v]$ for each node v, the length of the shortest path from v to t that we have found so far. We still run the algorithm for iterations $i = 1, 2, \ldots, n-1$, but the role of i will now simply be as a counter; in each iteration, and for each node v, we perform the update

$$M[v] = \min(M[v], \min_{w \in V}(c_{vw} + M[w])).$$

We now observe the following fact.

(6.26) *Throughout the algorithm $M[v]$ is the length of some path from v to t, and after i rounds of updates the value $M[v]$ is no larger than the length of the shortest path from v to t using at most i edges.*

Given (6.26), we can then use (6.22) as before to show that we are done after $n-1$ iterations. Since we are only storing an M array that indexes over the nodes, this requires only $O(n)$ working memory.

Finding the Shortest Paths One issue to be concerned about is whether this space-efficient version of the algorithm saves enough information to recover the shortest paths themselves. In the case of the Sequence Alignment Problem in the previous section, we had to resort to a tricky divide-and-conquer method to recover the solution from a similar space-efficient implementation. Here, however, we will be able to recover the shortest paths much more easily.

To help with recovering the shortest paths, we will enhance the code by having each node v maintain the first node (after itself) on its path to the destination t; we will denote this first node by $\mathit{first}[v]$. To maintain $\mathit{first}[v]$, we update its value whenever the distance $M[v]$ is updated. In other words, whenever the value of $M[v]$ is reset to the minimum $\min_{w \in V}(c_{vw} + M[w])$, we set $\mathit{first}[v]$ to the node w that attains this minimum.

Now let P denote the directed "pointer graph" whose nodes are V, and whose edges are $\{(v, \mathit{first}[v])\}$. The main observation is the following.

(6.27) *If the pointer graph P contains a cycle C, then this cycle must have negative cost.*

Proof. Notice that if $\mathit{first}[v] = w$ at any time, then we must have $M[v] \geq c_{vw} + M[w]$. Indeed, the left- and right-hand sides are equal after the update that sets $\mathit{first}[v]$ equal to w; and since $M[w]$ may decrease, this equation may turn into an inequality.

Let $v_1, v_2, \ldots, v_k$ be the nodes along the cycle C in the pointer graph, and assume that (v_k, v_1) is the last edge to have been added. Now, consider the values right before this last update. At this time we have $M[v_i] \geq c_{v_i v_{i+1}} + M[v_{i+1}]$ for all $i = 1, \ldots, k-1$, and we also have $M[v_k] > c_{v_k v_1} + M[v_1]$ since we are about to update $M[v_k]$ and change $\mathit{first}[v_k]$ to v_1. Adding all these inequalities, the $M[v_i]$ values cancel, and we get $0 > \sum_{i=1}^{k-1} c_{v_i v_{i+1}} + c_{v_k v_1}$: a negative cycle, as claimed. ∎

Now note that if G has no negative cycles, then (6.27) implies that the pointer graph P will never have a cycle. For a node v, consider the path we get by following the edges in P, from v to $first[v] = v_1$, to $first[v_1] = v_2$, and so forth. Since the pointer graph has no cycles, and the sink t is the only node that has no outgoing edge, this path must lead to t. We claim that when the algorithm terminates, this is in fact a shortest path in G from v to t.

(6.28) *Suppose G has no negative cycles, and consider the pointer graph P at the termination of the algorithm. For each node v, the path in P from v to t is a shortest v-t path in G.*

Proof. Consider a node v and let $w = first[v]$. Since the algorithm terminated, we must have $M[v] = c_{vw} + M[w]$. The value $M[t] = 0$, and hence the length of the path traced out by the pointer graph is exactly $M[v]$, which we know is the shortest-path distance. ■

Note that in the more space-efficient version of Bellman-Ford, the path whose length is $M[v]$ after i iterations can have substantially more edges than i. For example, if the graph is a single path from s to t, and we perform updates in the reverse of the order the edges appear on the path, then we get the final shortest-path values in just one iteration. This does not always happen, so we cannot claim a worst-case running-time improvement, but it would be nice to be able to use this fact opportunistically to speed up the algorithm on instances where it does happen. In order to do this, we need a stopping signal in the algorithm—something that tells us it's safe to terminate before iteration $n - 1$ is reached.

Such a stopping signal is a simple consequence of the following observation: If we ever execute a complete iteration i in which *no* $M[v]$ value changes, then no $M[v]$ value will ever change again, since future iterations will begin with exactly the same set of array entries. Thus it is safe to stop the algorithm. Note that it is not enough for a *particular* $M[v]$ value to remain the same; in order to safely terminate, we need for all these values to remain the same for a single iteration.

6.9 Shortest Paths and Distance Vector Protocols

One important application of the Shortest-Path Problem is for routers in a communication network to determine the most efficient path to a destination. We represent the network using a graph in which the nodes correspond to routers, and there is an edge between v and w if the two routers are connected by a direct communication link. We define a cost c_{vw} representing the delay on the link (v, w); the Shortest-Path Problem with these costs is to determine the path with minimum delay from a source node s to a destination t. Delays are naturally nonnegative, so one could use Dijkstra's Algorithm to compute the shortest path. However, Dijkstra's shortest-path computation requires global knowledge of the network: it needs to maintain a set S of nodes for which shortest paths have been determined, and make a global decision about which node to add next to S. While routers can be made to run a protocol in the background that gathers enough global information to implement such an algorithm, it is often cleaner and more flexible to use algorithms that require only local knowledge of neighboring nodes.

If we think about it, the Bellman-Ford Algorithm discussed in the previous section has just such a "local" property. Suppose we let each node v maintain its value $M[v]$; then to update this value, v needs only obtain the value $M[w]$ from each neighbor w, and compute

$$\min_{w \in V}(c_{vw} + M[w])$$

based on the information obtained.

We now discuss an improvement to the Bellman-Ford Algorithm that makes it better suited for routers and, at the same time, a faster algorithm in practice. Our current implementation of the Bellman-Ford Algorithm can be thought of as a *pull-based* algorithm. In each iteration i, each node v has to contact each neighbor w, and "pull" the new value $M[w]$ from it. If a node w has not changed its value, then there is no need for v to get the value again; however, v has no way of knowing this fact, and so it must execute the pull anyway.

This wastefulness suggests a symmetric *push-based* implementation, where values are only transmitted when they change. Specifically, each node w whose distance value $M[w]$ changes in an iteration informs all its neighbors of the new value in the next iteration; this allows them to update their values accordingly. If $M[w]$ has not changed, then the neighbors of w already have the current value, and there is no need to "push" it to them again. This leads to savings in the running time, as not all values need to be pushed in each iteration. We also may terminate the algorithm early, if no value changes during an iteration. Here is a concrete description of the push-based implementation.

```
Push-Based-Shortest-Path($G, s, t$)
  $n =$ number of nodes in $G$
  Array $M[V]$
  Initialize $M[t] = 0$ and $M[v] = \infty$ for all other $v \in V$
  For $i = 1, \ldots, n-1$
    For $w \in V$ in any order
      If $M[w]$ has been updated in the previous iteration then
      For all edges $(v, w)$ in any order
        $M[v] = \min(M[v], c_{vw} + M[w])$
        If this changes the value of $M[v]$, then $first[v] = w$
      Endfor
    Endfor
    If no value changed in this iteration, then end the algorithm
  Endfor
  Return $M[s]$
```

In this algorithm, nodes are sent updates of their neighbors' distance values in rounds, and each node sends out an update in each iteration in which it has changed. However, if the nodes correspond to routers in a network, then we do not expect everything to run in lockstep like this; some routers may report updates much more quickly than others, and a router with an update to report may sometimes experience a delay before contacting its neighbors. Thus the routers will end up executing an *asynchronous* version of the algorithm: each time a node w experiences an update to its $M[w]$ value, it becomes "active" and eventually notifies its neighbors of the new value. If we were to watch the behavior of all routers interleaved, it would look as follows.

```
Asynchronous-Shortest-Path($G, s, t$)
  $n =$ number of nodes in $G$
  Array $M[V]$
  Initialize $M[t] = 0$ and $M[v] = \infty$ for all other $v \in V$
  Declare $t$ to be active and all other nodes inactive
  While there exists an active node
    Choose an active node $w$
       For all edges $(v, w)$ in any order
         $M[v] = \min(M[v], c_{vw} + M[w])$
         If this changes the value of $M[v]$, then
           $first[v] = w$
           $v$ becomes active
       Endfor
       $w$ becomes inactive
  EndWhile
```

One can show that even this version of the algorithm, with essentially no coordination in the ordering of updates, will converge to the correct values of the shortest-path distances to t, assuming only that each time a node becomes active, it eventually contacts its neighbors.

The algorithm we have developed here uses a single destination t, and all nodes $v \in V$ compute their shortest path to t. More generally, we are presumably interested in finding distances and shortest paths between all pairs of nodes in a graph. To obtain such distances, we effectively use n separate computations, one for each destination. Such an algorithm is referred to as a *distance vector protocol*, since each node maintains a vector of distances to every other node in the network.

Problems with the Distance Vector Protocol

One of the major problems with the distributed implementation of Bellman-Ford on routers (the protocol we have been discussing above) is that it's derived from an initial dynamic programming algorithm that assumes edge costs will remain constant during the execution of the algorithm. Thus far we've been designing algorithms with the tacit understanding that a program executing the algorithm will be running on a single computer (or a centrally managed set of computers), processing some specified input. In this context, it's a rather benign assumption to require that the input not change while the program is actually running. Once we start thinking about routers in a network, however, this assumption becomes troublesome. Edge costs may change for all sorts of reasons: links can become congested and experience slow-downs; or a link (v, w) may even fail, in which case the cost c_{vw} effectively increases to ∞.

Here's an indication of what can go wrong with our shortest-path algorithm when this happens. If an edge (v, w) is deleted (say the link goes down), it is natural for node v to react as follows: it should check whether its shortest path to some node t used the edge (v, w), and, if so, it should increase the distance using other neighbors. Notice that this increase in distance from v can now trigger increases at v's neighbors, if they were relying on a path through v, and these changes can cascade through the network. Consider the extremely simple example in Figure 6.24, in which the original graph has three edges (s, v), (v, s) and (v, t), each of cost 1.

Now suppose the edge (v, t) in Figure 6.24 is deleted. How does node v react? Unfortunately, it does not have a global map of the network; it only knows the shortest-path distances of each of its neighbors to t. Thus it does

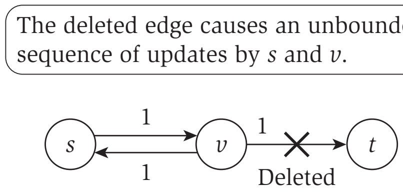

The deleted edge causes an unbounded sequence of updates by s and v.
s
v
t
1
1
1
Deleted

Figure 6.24 When the edge (v, t) is deleted, the distributed Bellman-Ford Algorithm will begin "counting to infinity."

Exercises

1. Let $G = (V, E)$ be an undirected graph with n nodes. Recall that a subset of the nodes is called an *independent set* if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here we'll see that it can be done efficiently if the graph is "simple" enough.

   Call a graph $G = (V, E)$ a *path* if its nodes can be written as $v_1, v_2, \ldots, v_n$, with an edge between v_i and v_j if and only if the numbers i and j differ by exactly 1. With each node v_i, we associate a positive integer *weight* w_i.

   Consider, for example, the five-node path drawn in Figure 6.28. The *weights* are the numbers drawn inside the nodes.

   The goal in this question is to solve the following problem:

   Find an independent set in a path G whose total weight is as large as possible.

   (a) Give an example to show that the following algorithm *does not* always find an independent set of maximum total weight.

```
The "heaviest-first" greedy algorithm
  Start with $S$ equal to the empty set
  While some node remains in $G$
    Pick a node $v_i$ of maximum weight
    Add $v_i$ to $S$
    Delete $v_i$ and its neighbors from $G$
  Endwhile
  Return $S$
```

   (b) Give an example to show that the following algorithm also *does not* always find an independent set of maximum total weight.

```
Let $S_1$ be the set of all $v_i$ where $i$ is an odd number
Let $S_2$ be the set of all $v_i$ where $i$ is an even number
(Note that $S_1$ and $S_2$ are both independent sets)
Determine which of $S_1$ or $S_2$ has greater total weight,
  and return this one
```

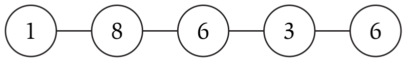


Figure 6.28 A paths with weights on the nodes. The maximum weight of an independent set is 14.

(c) Give an algorithm that takes an n-node path G with weights and returns an independent set of maximum total weight. The running time should be polynomial in n, independent of the values of the weights.

2. Suppose you're managing a consulting team of expert computer hackers, and each week you have to choose a job for them to undertake. Now, as you can well imagine, the set of possible jobs is divided into those that are *low-stress* (e.g., setting up a Web site for a class at the local elementary school) and those that are *high-stress* (e.g., protecting the nation's most valuable secrets, or helping a desperate group of Cornell students finish a project that has something to do with compilers). The basic question, each week, is whether to take on a low-stress job or a high-stress job.

If you select a low-stress job for your team in week i, then you get a revenue of $\ell_i > 0$ dollars; if you select a high-stress job, you get a revenue of $h_i > 0$ dollars. The catch, however, is that in order for the team to take on a high-stress job in week i, it's required that they do no job (of either type) in week $i - 1$; they need a full week of prep time to get ready for the crushing stress level. On the other hand, it's okay for them to take a low-stress job in week i even if they have done a job (of either type) in week $i - 1$.

So, given a sequence of n weeks, a *plan* is specified by a choice of "low-stress," "high-stress," or "none" for each of the n weeks, with the property that if "high-stress" is chosen for week $i > 1$, then "none" has to be chosen for week $i - 1$. (It's okay to choose a high-stress job in week 1.) The *value* of the plan is determined in the natural way: for each i, you add ℓ_i to the value if you choose "low-stress" in week i, and you add h_i to the value if you choose "high-stress" in week i. (You add 0 if you choose "none" in week i.)

The problem. Given sets of values $\ell_1, \ell_2, \ldots, \ell_n$ and $h_1, h_2, \ldots, h_n$, find a plan of maximum value. (Such a plan will be called *optimal*.)

Example. Suppose $n = 4$, and the values of ℓ_i and h_i are given by the following table. Then the plan of maximum value would be to choose "none" in week 1, a high-stress job in week 2, and low-stress jobs in weeks 3 and 4. The value of this plan would be $0 + 50 + 10 + 10 = 70$.

	Week 1	Week 2	Week 3	Week 4
ℓ	10	1	10	10
h	5	50	5	1

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
For iterations $i = 1$ to $n$
  If $h_{i+1} > \ell_i + \ell_{i+1}$ then
    Output "Choose no job in week $i$"
    Output "Choose a high-stress job in week $i+1$"
    Continue with iteration $i+2$
  Else
    Output "Choose a low-stress job in week $i$"
    Continue with iteration $i+1$
  Endif
End
```

To avoid problems with overflowing array bounds, we define $h_i = \ell_i = 0$ when $i > n$.

In your example, say what the correct answer is and also what the above algorithm finds.

(b) Give an efficient algorithm that takes values for $\ell_1, \ell_2, \ldots, \ell_n$ and $h_1, h_2, \ldots, h_n$ and returns the *value* of an optimal plan.

3. Let $G = (V, E)$ be a directed graph with nodes $v_1, \ldots, v_n$. We say that G is an *ordered graph* if it has the following properties.

(i) Each edge goes from a node with a lower index to a node with a higher index. That is, every directed edge has the form (v_i, v_j) with $i < j$.

(ii) Each node except v_n has at least one edge leaving it. That is, for every node v_i, $i = 1, 2, \ldots, n-1$, there is at least one edge of the form (v_i, v_j).

The length of a path is the number of edges in it. The goal in this question is to solve the following problem (see Figure 6.29 for an example).

Given an ordered graph G, find the length of the longest path that begins at v_1 and ends at v_n.

(a) Show that the following algorithm does not correctly solve this problem, by giving an example of an ordered graph on which it does not return the correct answer.

```
Set $w = v_1$
Set $L = 0$
```

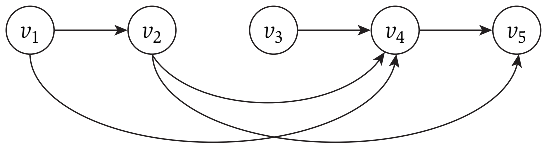


Figure 6.29 The correct answer for this ordered graph is 3: The longest path from v_1 to v_n uses the three edges (v_1, v_2),(v_2, v_4), and (v_4, v_5).

```
While there is an edge out of the node $w$
  Choose the edge $(w, v_j)$
     for which $j$ is as small as possible
  Set $w = v_j$
  Increase $L$ by 1
end while
Return $L$ as the length of the longest path
```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes an ordered graph G and returns the *length* of the longest path that begins at v_1 and ends at v_n. (Again, the *length* of a path is the number of edges in the path.)

4. Suppose you're running a lightweight consulting business—just you, two associates, and some rented equipment. Your clients are distributed between the East Coast and the West Coast, and this leads to the following question.

Each month, you can either run your business from an office in New York (NY) or from an office in San Francisco (SF). In month i, you'll incur an *operating cost* of N_i if you run the business out of NY; you'll incur an operating cost of S_i if you run the business out of SF. (It depends on the distribution of client demands for that month.)

However, if you run the business out of one city in month i, and then out of the other city in month $i + 1$, then you incur a fixed *moving cost* of M to switch base offices.

Given a sequence of n months, a *plan* is a sequence of n locations—each one equal to either NY or SF—such that the i^{th} location indicates the city in which you will be based in the i^{th} month. The *cost* of a plan is the sum of the operating costs for each of the n months, plus a moving cost of M for each time you switch cities. The plan can begin in either city.

The problem. Given a value for the moving cost M, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, find a plan of minimum cost. (Such a plan will be called *optimal*.)

Example. Suppose $n = 4$, $M = 10$, and the operating costs are given by the following table.

	Month 1	**Month 2**	**Month 3**	**Month 4**
NY	1	3	20	30
SF	50	20	2	4

Then the plan of minimum cost would be the sequence of locations

$$[NY, NY, SF, SF],$$

with a total cost of $1 + 3 + 2 + 4 + 10 = 20$, where the final term of 10 arises because you change locations once.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
For $i$ = 1 to $n$
  If $N_i$ < $S_i$ then
    Output "NY in Month $i$"
  Else
    Output "SF in Month $i$"
End
```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an example of an instance in which every optimal plan must move (i.e., change locations) at least three times.

Provide a brief explanation, saying why your example has this property.

(c) Give an efficient algorithm that takes values for n, M, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, and returns the *cost* of an optimal plan.

5. As some of you know well, and others of you may be interested to learn, a number of languages (including Chinese and Japanese) are written without spaces between the words. Consequently, software that works with text written in these languages must address the *word segmentation problem*—inferring likely boundaries between consecutive words in the text. If English were written without spaces, the analogous problem would consist of taking a string like “meetateight” and deciding that the best segmentation is “meet at eight” (and not “me et at eight,” or “meet ate ight,” or any of a huge number of even less plausible alternatives). How could we automate this process?

A simple approach that is at least reasonably effective is to find a segmentation that simply maximizes the cumulative “quality” of its individual constituent words. Thus, suppose you are given a black box that, for any string of letters $x = x_1x_2 \cdots x_k$, will return a number $quality(x)$. This number can be either positive or negative; larger numbers correspond to more plausible English words. (So $quality$(“*me*”) would be positive, while $quality$(“*ght*”) would be negative.)

Given a long string of letters $y = y_1y_2 \cdots y_n$, a segmentation of y is a partition of its letters into contiguous blocks of letters; each block corresponds to a word in the segmentation. The *total quality* of a segmentation is determined by adding up the qualities of each of its blocks. (So we’d get the right answer above provided that $quality$(“*meet*”) + $quality$(“*at*”) + $quality$(“*eight*”) was greater than the total quality of any other segmentation of the string.)

Give an efficient algorithm that takes a string y and computes a segmentation of maximum total quality. (You can treat a single call to the black box computing $quality(x)$ as a single computational step.)

(*A final note, not necessary for solving the problem:* To achieve better performance, word segmentation software in practice works with a more complex formulation of the problem—for example, incorporating the notion that solutions should not only be reasonable at the word level, but also form coherent phrases and sentences. If we consider the example “theyouthevent,” there are at least three valid ways to segment this into common English words, but one constitutes a much more coherent phrase than the other two. If we think of this in the terminology of formal languages, this broader problem is like searching for a segmentation that also can be parsed well according to a grammar for the underlying language. But even with these additional criteria and constraints, dynamic programming approaches lie at the heart of a number of successful segmentation systems.)

6. In a word processor, the goal of “pretty-printing” is to take text with a ragged right margin, like this,

```
Call me Ishmael.
Some years ago,
never mind how long precisely,
```