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 $\alpha_{x_i y_j}$ 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 $\alpha_{x_m y_n}$ and then align $x_1x_2 \cdots x_{m-1}$ as well as possible with $y_1y_2 \cdots y_{n-1}$; we get $\text{OPT}(m, n) = \alpha_{x_m y_n} + \text{OPT}(m-1, n-1)$. 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 $y_1 y_2 \cdots y_n$. 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 $x_1 x_2 \cdots x_i$ and $y_1 y_2 \cdots y_j$, we get the following fact.

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

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

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...m, 0...n]
  Initialize A[i, 0] = iδ for each i
  Initialize A[0, j] = jδ for each j
  For j = 1, ..., n
    For i = 1, ..., 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.

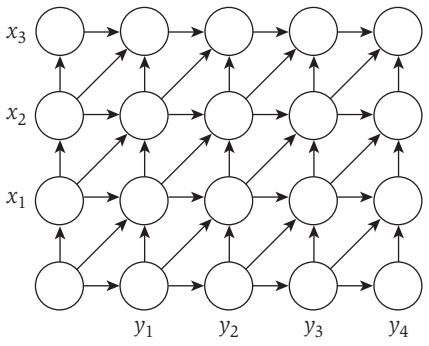

Figure 6.17 A graph-based picture of sequence alignment.

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). ■

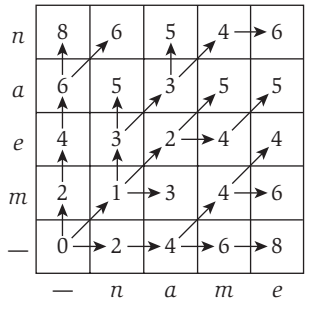

Figure 6.18 The OPT values for the problem of aligning the words *mean* to *name*.

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...m, 0...1]
  Initialize B[i, 0] = iδ for each i (just as in column 0 of A)
  For j = 1, ..., n
     B[0, 1] = jδ (since this corresponds to entry A[0, j])
     For i = 1, ..., m
          B[i, 1] = min[α_{x_i y_j} + B[i − 1, 0],
                        δ + B[i − 1, 1],  δ + 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

this backward version, naturally enough, as `Backward-Space-Efficient-Alignment`.

Combining the Forward and Backward Formulations So now we have symmetric algorithms which build up the values of the functions f and g. The idea will be to use these two algorithms in concert to find the optimal alignment. First, here are two basic facts summarizing some relationships between the functions f and g.

(6.19) *The length of the shortest corner-to-corner path in G_{XY} that passes through (i, j) is $f(i, j) + g(i, j)$.*

Proof. Let ℓ_{ij} denote the length of the shortest corner-to-corner path in G_{XY} that passes through (i, j). Clearly, any such path must get from $(0, 0)$ to (i, j) and then from (i, j) to (m, n). Thus its length is at least $f(i, j) + g(i, j)$, and so we have $\ell_{ij} \geq f(i, j) + g(i, j)$. On the other hand, consider the corner-to-corner path that consists of a minimum-length path from $(0, 0)$ to (i, j), followed by a minimum-length path from (i, j) to (m, n). This path has length $f(i, j) + g(i, j)$, and so we have $\ell_{ij} \leq f(i, j) + g(i, j)$. It follows that $\ell_{ij} = f(i, j) + g(i, j)$. ■

(6.20) *Let k be any number in $\{0, \ldots, n\}$, and let q be an index that minimizes the quantity $f(q, k) + g(q, k)$. Then there is a corner-to-corner path of minimum length that passes through the node (q, k).*

Proof. Let ℓ^* denote the length of the shortest corner-to-corner path in G_{XY}. Now fix a value of $k \in \{0, \ldots, n\}$. The shortest corner-to-corner path must use *some* node in the k^{th} column of G_{XY}—let's suppose it is node (p, k)—and thus by (6.19)

$$\ell^* = f(p, k) + g(p, k) \geq \min_q f(q, k) + g(q, k).$$

Now consider the index q that achieves the minimum in the right-hand side of this expression; we have

$$\ell^* \geq f(q, k) + g(q, k).$$

By (6.19) again, the shortest corner-to-corner path using the node (q, k) has length $f(q, k) + g(q, k)$, and since ℓ^* is the minimum length of *any* corner-to-corner path, we have

$$\ell^* \leq f(q, k) + g(q, k).$$

It follows that $\ell^* = f(q, k) + g(q, k)$. Thus the shortest corner-to-corner path using the node (q, k) has length ℓ^*, and this proves (6.20). ■

Using (6.20) and our space-efficient algorithms to compute the *value* of the optimal alignment, we will proceed as follows. We divide G_{XY} along its center column and compute the value of $f(i, n/2)$ and $g(i, n/2)$ for each value of i, using our two space-efficient algorithms. We can then determine the minimum value of $f(i, n/2) + g(i, n/2)$, and conclude via (6.20) that there is a shortest corner-to-corner path passing through the node $(i, n/2)$. Given this, we can search for the shortest path recursively in the portion of G_{XY} between $(0, 0)$ and $(i, n/2)$ and in the portion between $(i, n/2)$ and (m, n). The crucial point is that we apply these recursive calls sequentially and reuse the working space from one call to the next. Thus, since we only work on one recursive call at a time, the total space usage is $O(m + n)$. The key question we have to resolve is whether the running time of this algorithm remains $O(mn)$.

In running the algorithm, we maintain a globally accessible list P which will hold nodes on the shortest corner-to-corner path as they are discovered. Initially, P is empty. P need only have $m + n$ entries, since no corner-to-corner path can use more than this many edges. We also use the following notation: $X[i : j]$, for $1 \le i \le j \le m$, denotes the substring of X consisting of $x_i x_{i+1} \cdots x_j$; and we define $Y[i : j]$ analogously. We will assume for simplicity that n is a power of 2; this assumption makes the discussion much cleaner, although it can be easily avoided.

```
Divide-and-Conquer-Alignment(X,Y)
  Let m be the number of symbols in X
  Let n be the number of symbols in Y
  If m ≤ 2 or n ≤ 2 then
     Compute optimal alignment using Alignment(X,Y)
  Call Space-Efficient-Alignment(X,Y[1 : n/2])
  Call Backward-Space-Efficient-Alignment(X,Y[n/2 + 1 : n])
  Let q be the index minimizing f(q, n/2) + g(q, n/2)
  Add (q, n/2) to global list P
  Divide-and-Conquer-Alignment(X[1 : q],Y[1 : n/2])
  Divide-and-Conquer-Alignment(X[q + 1 : n],Y[n/2 + 1 : n])
  Return P
```

As an example of the first level of recursion, consider Figure 6.19. If the *minimizing index* q turns out to be 1, we get the two subproblems pictured.

Analyzing the Algorithm

The previous arguments already establish that the algorithm returns the correct answer and that it uses $O(m + n)$ space. Thus, we need only verify the following fact.

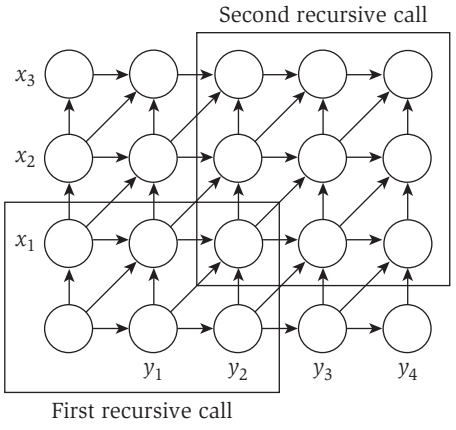

Figure 6.19 The first level of recurrence for the space-efficient `Divide-and-Conquer-Alignment`. The two boxed regions indicate the input to the two recursive cells.

(6.21) *The running time of* `Divide-and-Conquer-Alignment` *on strings of length m and n is $O(mn)$.*

Proof. Let $T(m, n)$ denote the maximum running time of the algorithm on strings of length m and n. The algorithm performs $O(mn)$ work to build up the arrays B and B'; it then runs recursively on strings of size q and $n/2$, and on strings of size $m - q$ and $n/2$. Thus, for some constant c, and some choice of index q, we have

$$T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2)$$
$$T(m, 2) \leq cm$$
$$T(2, n) \leq cn.$$

This recurrence is more complex than the ones we've seen in our earlier applications of divide-and-conquer in Chapter 5. First of all, the running time is a function of two variables (m and n) rather than just one; also, the division into subproblems is not necessarily an "even split," but instead depends on the value q that is found through the earlier work done by the algorithm.

So how should we go about solving such a recurrence? One way is to try guessing the form by considering a special case of the recurrence, and then using partial substitution to fill out the details of this guess. Specifically, suppose that we were in a case in which $m = n$, and in which the split point q were exactly in the middle. In this (admittedly restrictive) special case, we could write the function $T(\cdot)$ in terms of the single variable n, set $q = n/2$ (since we're assuming a perfect bisection), and have

$$T(n) \leq 2T(n/2) + cn^2.$$

This is a useful expression, since it's something that we solved in our earlier discussion of recurrences. Specifically, this recur-rence implies $T(n) = O(n^2)$.

So when $m = n$ and we get an even split, the running time grows like the square of n. Motivated by this, we move back to the fully general recurrence for the problem at hand and guess that $T(m, n)$ grows like the product of m and n. Specifically, we'll guess that $T(m, n) \leq kmn$ for some constant k, and see if we can prove this by induction. To start with the base cases $m \leq 2$ and $n \leq 2$, we see that these hold as long as $k \geq c/2$. Now, assuming $T(m', n') \leq km'n'$ holds for pairs (m', n') with a smaller product, we have

$$T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2)$$

$$\begin{aligned} &\leq cmn + kqn/2 + k(m - q)n/2 \\ &= cmn + kqn/2 + kmn/2 - kqn/2 \\ &= (c + k/2)mn. \end{aligned}$$

Thus the inductive step will work if we choose $k = 2c$, and this completes the proof. ■