3

SEQUENCE COMPARISON AND DATABASE SEARCH

In this chapter we present some of the most practical and widely used methods for sequence comparison and database search. Sequence comparison is undoubtedly the most basic operation in computational biology. It appears explicitly or implicitly in all subsequent chapters of this book. It has also many applications in other subareas of computer science.

3.1 BIOLOGICAL BACKGROUND

Why compare sequences? Why use a computer to do that? How to compare sequences using computers? These are the main questions that will concern us in this chapter. We will answer the first two in this section, and devote the rest of the chapter to the last one.

Sequence comparison is the most important primitive operation in computational biology, serving as a basis for many other, more complex, manipulations. Roughly speaking, this operation consists of finding which parts of the sequences are alike and which parts differ. However, behind this apparently simple concept, a great variety of truly distinct problems exist, with diverse formalizations and sometimes requiring completely different data structures and algorithms for an efficient solution.

As examples, we will give a list of problems that often appear in computational biology. In these examples we use two notions that will be precisely defined in later sections. One is the *similarity* of two sequences, which gives a measure of how similar the sequences are. The other is the *alignment* of two sequences, which is a way of placing one sequence above the other in order to make clear the correspondence between similar characters or substrings from the sequences. The examples (and the whole chapter) also use basic concepts about strings, which are defined in Section 2.1. Here are the examples.

1. We have two sequences over the same alphabet, both about the same length (tens of thousands of characters). We know that the sequences are almost equal, with

only a few isolated differences such as insertions, deletions, and substitutions of characters. The average frequency of these differences is low, say, one each hundred characters. We want to find the places where the differences occur.

2. We have two sequences over the same alphabet with a few hundred characters each. We want to know whether there is a prefix of one which is similar to a suffix of the other. If the answer is yes, the prefix and the suffix involved must be produced.
3. We have the same problem as in (2), but now we have several hundred sequences that must be compared (each one against all). In addition, we know that the great majority of sequence pairs are unrelated, that is, they will not have the required degree of similarity.
4. We have two sequences over the same alphabet with a few hundred characters each. We want to know whether there are two substrings, one from each sequence, that are similar.
5. We have the same problem as in (4), but instead of two sequences we have one sequence that must be compared to thousands of others.

Problems like (1) appear when, for instance, the same gene is sequenced by two different labs and they want to compare the results; or even when the same long sequence is typed twice into the computer and we are looking for typing errors. Problems like (2) and (3) appear in the context of fragment assembly in programs to help large-scale DNA sequencing. Problems like (4) and (5) occur in the context of searches for local similarities using large biosequence databases.

We will see in this chapter that a single basic algorithmic idea can be used to solve all of the above problems. However, this may not be the most efficient solution. Sometimes less general but faster methods are better suited to each task.

The use of computers hardly needs justification when we deal with large quantities of data, as in searches in large databases. Yet, even in cases where we could conceivably do comparisons "by hand," the use of computers is safer and more convenient, as the following examples, taken from a paper by Smith and coworkers [176], illustrate.

In a 1979 paper, Sims and colleagues examined similar regions from the DNA of two bacteriophages [173]. They presented an alignment between regions of the H-gene from phages St-1 and G4 containing 11 matches (that is, 11 columns in which there are equal characters in both sequences). The procedure they used to obtain good alignments was described as the "insertion of occasional gaps to maximize homology [number of identities]." Smith and colleagues [176], however, produced 12 matches in an alignment found by computer, which was certainly overlooked by the first group of researchers.

In another 1979 paper, Rosenberg and Court completed a study aligning 46 promoter sequences from various organisms [164]. The multiple alignment was constructed starting from the alignment of certain regions reputed relatively invariant and then extending this "seed" alignment in both directions without introducing gaps. Although the resulting alignment was good as a whole, the pairwise alignments obtained from it were relatively poor. This happens fairly frequently in multiple alignments, but in this case Smith and colleagues found, with the help of a computer, an alignment between two substrings of the sequences involved that had 44 identical characters over a total of 45. This is certainly an astonishingly good local alignment, and the fact that it is not even mentioned by Rosenberg and Court must mean again that they were not able to find it by hand.

These two examples show that the use of computers can reveal intriguing similarities that might go unnoticed otherwise.

3.2 COMPARING TWO SEQUENCES

In this section we study methods for comparing two sequences. More specifically, we are interested in finding the best alignments between these two sequences. Several versions of this problem occur in practice, depending on whether we are interested in alignments involving the entire sequences or just substrings of them. This leads to the definition of global and local comparisons. There is also a third kind of comparison in which we are interested in aligning not arbitrary substrings, but prefixes and suffixes of the given sequences. We call this third kind semiglobal comparison. All the problems mentioned can be solved efficiently by dynamic programming, as we will see in the sequel.

3.2.1 GLOBAL COMPARISON — THE BASIC ALGORITHM

Consider the following DNA sequences: GACGGATTAG and GATCGGAATAG. We cannot help but notice that they actually look very much alike, a fact that becomes more obvious when we align them one above the other as follows.

```
GA-CGGATTAG
GATCGGAATAG
```
(3.1)

The only differences are an extra T in the second sequence and a change from A to T in the fourth position from right to left. Observe that we had to introduce a space (indicated by a dash above) in the first sequence to let equal bases before and after the space in the two sequences align perfectly.

Our goal in this section is to present an efficient algorithm that takes two sequences and determines the best alignment between them as we did above. Of course, we must define what the "best" alignment is before approaching the problem. To simplify the discussion, we will adopt a simple formalism; later, we will see possible generalizations.

To begin, let us be precise about what we mean by an *alignment* between two sequences. The sequences may have different sizes. As we saw in the recent example, alignments may contain spaces in any of the sequences. Thus, we define an alignment as the insertion of spaces in arbitrary locations along the sequences so that they end up with the same size. Having the same size, the augmented sequences can now be placed one over the other, creating a correspondence between characters or spaces in the first sequence and characters or spaces in the second sequence. In addition, we require that no space in one sequence be aligned with a space in the other. Spaces can be inserted even in the beginning or end of sequences.

Given an alignment between two sequences, we can assign a *score* to it as follows. Each column of the alignment will receive a certain value depending on its contents and the total score for the alignment will be the sum of the values assigned to its columns. If a column has two identical characters, it will receive value $+1$ (a *match*). Different characters will give the column value -1 (a *mismatch*). Finally, a space in a column drops down its value to -2. The best alignment will be the one with maximum total score. This maximum score will be called the *similarity* between the two sequences and will be denoted by $\mathrm{sim}(s, t)$ for sequences s and t. In general, there may be many alignments with maximum score.

To exercise these definitions, let us compute the score of alignment (3.1). There are nine columns with identical characters, one column with distinct characters, and one column with a space, giving a total score of

$$9 \times 1 + 1 \times (-1) + 1 \times (-2) = 6.$$

Why did we choose these particular values ($+1$, -1, and -2)? This scoring system is often used in practice. We reward matches and penalize mismatches and spaces. In Section 3.6.2 we discuss in more detail the choice of these parameters.

One approach to computing the similarity between two sequences would be to generate all possible alignments and then pick the best one. However, the number of alignments between two sequences is exponential, and such an approach would result in an intolerably slow algorithm. Fortunately, a much faster algorithm exists, which we will now describe.

The algorithm uses a technique known as *dynamic programming*. It basically consists of solving an instance of a problem by taking advantage of already computed solutions for smaller instances of the same problem. Given two sequences s and t, instead of determining the similarity between s and t as whole sequences only, we build up the solution by determining all similarities between arbitrary *prefixes* of the two sequences. We start with the shorter prefixes and use previously computed results to solve the problem for larger prefixes.

Let m be the size of s and n the size of t. There are $m + 1$ possible prefixes of s and $n + 1$ prefixes of t, including the empty string. Thus, we can arrange our calculations in an $(m + 1) \times (n + 1)$ array where entry (i, j) contains the similarity between $s[1..i]$ and $t[1..j]$.

Figure 3.1 shows the array corresponding to $s =$ AAAC and $t =$ AGC. We placed s along the left margin and t along the top to indicate the prefixes more easily. Notice that the first row and the first column are initialized with multiples of the space penalty (-2 in our case). This is because there is only one alignment possible if one of the sequences is empty: Just add as many spaces as there are characters in the other sequence. The score of this alignment is $-2k$, where k is the length of the nonempty sequence. Hence, filling the first row and column is easy.

Now let us concentrate on the other entries. The key observation here is that we can compute the value for entry (i, j) looking at just three previous entries: those for $(i - 1, j)$, $(i - 1, j - 1)$, and $(i, j - 1)$. The reason is that there are just three ways of obtaining an alignment between $s[1..i]$ and $t[1..j]$, and each one uses one of these previous values. In fact, to get an alignment for $s[1..i]$ and $t[1..j]$, we have the following three choices:

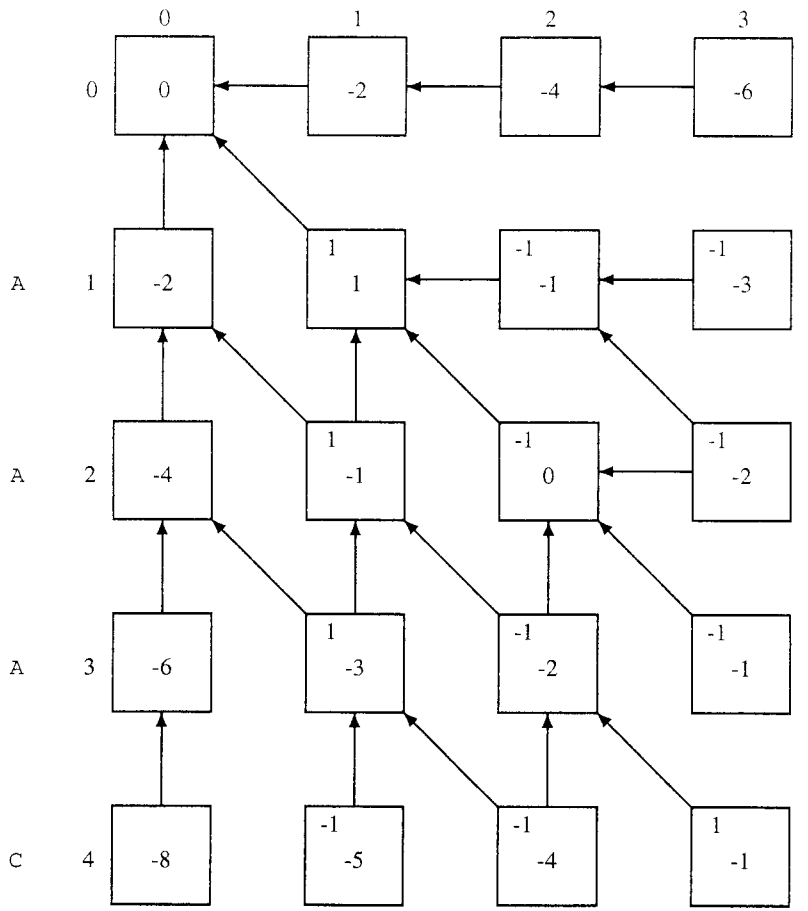

FIGURE 3.1

Bidimensional array for computing optimal alignments. The value in the upper left corner of cell (i, j) indicates whether $s[i] = t[j]$. Indices of rows and columns start at zero.

- Align $s[1..i]$ with $t[1..j - 1]$ and match a space with $t[j]$, or
- Align $s[1..i - 1]$ with $t[1..j - 1]$ and match $s[i]$ with $t[j]$, or
- Align $s[1..i - 1]$ with $t[1..j]$ and match $s[i]$ with a space.

These possibilities are exhaustive because we cannot have two spaces paired in the last column of the alignment. Scores of the best alignments between smaller prefixes are already stored in the array if we choose an appropriate order in which to compute the entries. As a consequence, the similarity sought can be determined by the formula

$$\mathrm{sim}(s[1..i], t[1..j]) = \max \begin{cases} \mathrm{sim}(s[1..i], t[1..j-1]) - 2 \\ \mathrm{sim}(s[1..i-1], t[1..j-1]) + p(i, j) \\ \mathrm{sim}(s[1..i-1], t[1..j]) - 2, \end{cases} \tag{3.2}$$

where $p(i, j)$ is $+1$ if $s[i] = t[j]$ and -1 if $s[i] \neq t[j]$. The values of $p(i, j)$ are written in the upper left corners of the boxes in Figure 3.1. If we denote the array by a, this equation can be rewritten as follows:

$$a[i, j] = \max \begin{cases} a[i, j-1] - 2 \\ a[i-1, j-1] + p(i, j) \\ a[i-1, j] - 2. \end{cases} \tag{3.3}$$

As we mentioned earlier, a good computing order must be followed. This is easy to accomplish. Filling in the array either row by row, left to right on each row, or column by column, top to bottom on each column, suffices. Any other order that makes sure $a[i, j-1]$, $a[i-1, j-1]$, and $a[i-1, j]$ are available when $a[i, j]$ must be computed is fine, too.

Finally, we drew arrows in Figure 3.1 to indicate where the maximum value comes from according to Equation (3.3). For instance, the value of $a[1, 2]$ was taken as the maximum among the following figures.

$$a[1, 1] - 2 = -1$$
$$a[0, 1] - 1 = -3$$
$$a[0, 2] - 2 = -6.$$

Therefore, there is only one way of getting this maximum value, namely, coming from entry (1, 1), and that is what the arrows show.

Figure 3.2 presents an algorithm for filling in an array as explained. This algorithm computes entries row by row. It depends on a parameter g that specifies the space penalty

```
Algorithm Similarity
    input: sequences s and t
    output: similarity between s and t
    m ← |s|
    n ← |t|
    for i ← 0 to m do
        a[i, 0] ← i × g
    for j ← 0 to n do
        a[0, j] ← j × g
    for i ← 1 to m do
        for j ← 1 to n do
            a[i, j] ← max(a[i − 1, j] + g,
                          a[i − 1, j − 1] + p(i, j),
                          a[i, j − 1] + g)
    return a[m, n]
```

FIGURE 3.2

Basic dynamic programming algorithm for comparison of two sequences.

(usually $g < 0$) and on a scoring function p for pairs of characters. We have used $g = -2$, $p(a, b) = 1$ if $a = b$, and $p(a, b) = -1$ if $a \neq b$ in Figure 3.1.

Optimal Alignments

We saw how to compute the similarity between two sequences. Here we will see how to construct an optimal alignment between them. The arrows in Figure 3.1 will be useful in this respect. All we need to do is start at entry (m, n) and follow the arrows until we get to (0, 0). Each arrow used will give us one column of the alignment. In fact, consider an arrow leaving entry (i, j). If this arrow is horizontal, it corresponds to a column with a space in s matched with $t[j]$; if it is vertical, it corresponds to $s[i]$ matched with a space in t; finally, a diagonal arrow means $s[i]$ matched with $t[j]$. Observe that the first sequence, s, is always placed along the vertical edge. Thus, an optimal alignment can be easily constructed from right to left if we have the matrix a computed by the basic algorithm. It is not necessary to implement the arrows explicitly — a simple test can be used to choose the next entry to visit.

Figure 3.3 shows a recursive algorithm for determining an optimal alignment given the matrix a and sequences s and t. The call *Align*(m, n, len) will construct an optimal alignment. The answer will be given in a pair of vectors *align-s* and *align-t* that will hold in the positions 1..*len* the aligned characters, which can be either spaces or symbols from the sequences. The variables *align-s* and *align-t* are treated as globals in the code. The

```
Algorithm Align
    input: indices i, j, array a given by algorithm Similarity
    output: alignment in align-s, align-t, and length in len
    if i = 0 and j = 0 then
        len ← 0
    else if i > 0 and a[i, j] = a[i − 1, j] + g then
        Align(i − 1, j, len)
        len ← len + 1
        align-s[len] ← s[i]
        align-t[len] ← -
    else if i > 0 and j > 0 and a[i, j] = a[i − 1, j − 1] + p(i, j) then
        Align(i − 1, j − 1, len)
        len ← len + 1
        align-s[len] ← s[i]
        align-t[len] ← t[j]
    else  // has to be j > 0 and a[i, j] = a[i, j − 1] + g
        Align(i, j − 1, len)
        len ← len + 1
        align-s[len] ← -
        align-t[len] ← t[j]
```

FIGURE 3.3

Recursive algorithm for optimal alignment.

length of the alignment is also returned by the algorithm in the parameter *len*. Note that $\max(|s|, |t|) \leq len \leq m + n$.

As we said previously, many optimal alignments may exist for a given pair of sequences. The algorithm of Figure 3.3 returns just one of them, giving preference to the edges leaving (i, j) in counterclockwise order (see Figure 3.4).

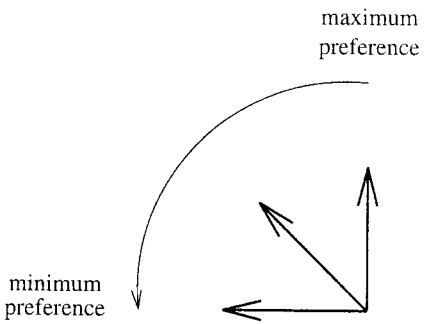

FIGURE 3.4

Arrow preference.

As a result, the optimal alignment returned by this algorithm has the following general characteristic: When there is choice, a column with a space in t has precedence over a column with two symbols, which in turn has precedence over a column with a space in s. For instance, when aligning $s =$ ATAT with $t =$ TATA, we get

```
-ATAT
TATA-
```

rather than

```
ATAT-
-TATA
```

which is the other optimal alignment for these two sequences. Similarly, when aligning $s =$ AA with $t =$ AAAA, we get

```
--AA
AAAA
```

although there are five other optimal alignments. This alignment is sometimes referred to as the *upmost* alignment because it uses the arrows higher up in the matrix. To reverse these preferences, we would reverse the order of the **if** statements in the code, obtaining the *downmost* alignment in this case. A column appearing in both the upmost and downmost alignments will be present in all optimal alignments between the two sequences considered.

It is possible to modify the algorithm to produce *all* optimal alignments between s and t. We need to keep a stack with the points at which there are options and backtrack to them to explore all possibilities of reaching (0, 0) through the arrows. However, there might be a very large number of optimal alignments. In these cases, it is often advisable to keep some sort of short representation of all or part of these alignments, for instance, the upmost and downmost ones.

Let us now determine the complexity of the algorithms described in this section. The basic algorithm of Figure 3.2 has four loops. The first two do the initialization and consume time $O(m)$ and $O(n)$, respectively. The last two loops are nested and fill the rest of the matrix. The number of operations performed depends essentially on the number of entries that must be computed, that is, the size of the matrix. Thus, we spend time $O(mn)$ in this part and this is the dominant term in the time complexity. The space used is also proportional to the size of the matrix. Hence, the complexity of the basic algorithm is $O(mn)$ both for time and space. If the sequences have the same or nearly the same length, say n, we get $O(n^2)$. That is why we say that these algorithms have quadratic complexity.

The construction of the alignment — given the already filled matrix — is done in time $O(len)$, where *len* is the size of the returned alignment, which is $O(m + n)$.

3.2.2 LOCAL COMPARISON

A *local alignment* between s and t is an alignment between a substring of s and a substring of t. In this section we present an algorithm to find the highest scoring local alignments between two sequences.

This algorithm is a variation of the basic algorithm. The main data structure is, as before, an $(m+1) \times (n+1)$ array. Only this time the interpretation of the array values is different. Each entry (i, j) will hold the highest score of an alignment between a *suffix* of $s[1..i]$ and a *suffix* of $t[1..j]$. The first row and the first column are initialized with zeros.

For any entry (i, j), there is always the alignment between the empty suffixes of $s[1..i]$ and $t[1..j]$, which has score zero; therefore the array will have all entries greater than or equal to zero. This explains in part the initialization above.

Following initialization, the array can be filled in the usual way, with $a[i, j]$ depending on the value of three previously computed entries. The resulting recurrence is

$$a[i, j] = \max \begin{cases} a[i, j-1] + g \\ a[i-1, j-1] + p(i, j) \\ a[i-1, j] + g \\ 0, \end{cases}$$

that is, the same as in the basic algorithm, except that now we have a fourth possibility, not available in the global case, of an empty alignment.

In the end, it suffices to find the maximum entry in the whole array. This will be the score of an optimal local alignment. Any entry containing this value can be used as a starting point to get such an alignment. The rest of the alignment is obtained tracing back as usual, but stopping as soon as we reach an entry with no arrow going out. Alternatively, we can stop as soon as we reach an entry with value zero.

In general, when doing local comparison, we are interested not only in the optimal alignments, but also in near optimal alignments with scores above a certain threshold. References to methods for retrieving near optimal alignments are given in the bibliographic notes.

3.2.3 SEMIGLOBAL COMPARISON

In a semiglobal comparison, we score alignments ignoring some of the *end spaces* in the sequences. An interesting characteristic of the basic dynamic programming algorithm is that we can control the penalty associated with end spaces by doing very simple modifications to the original scheme.

Let us begin by defining precisely what we mean by *end spaces* and why it might be better to let them be included for free in certain situations. End spaces are those that appear before the first or after the last character in a sequence. For instance, all the spaces in the second sequence in the alignment below are end spaces, while the single space in the first sequence is not an end space:

```
CAGCA-CTTGGATTCTCGG
---CAGCGTGG--------
```
(3.4)

Notice that the lengths of these two sequences differ considerably. One has size 8, and the other has 18 characters. When this happens, there will be many spaces in any alignment, giving a large negative contribution to the score. Nevertheless, if we ignore end spaces, the alignment is pretty good, with 6 matches, 1 mismatch, and 1 space.

Observe that this is not the best alignment between these sequences. In alignment (3.5) below we present another alignment with a higher score (-12 against -19 of the previous one) according to the scoring system we have been using so far.

```
CAGCACTTGGATTCTCGG
CAGC-----G-T----GG
```
(3.5)

In spite of having scored higher and having matched all characters of the second sequence with identical characters in the first one, this alignment is not so interesting from the point of view of finding similar regions in the sequences. The second sequence was simply torn apart brutally by the spaces just for the sake of matching exactly its characters. If we are looking for regions of the longer sequence that are approximately the same as the shorter sequence, then undoubtedly the first alignment (3.4) is more to the point. This is reflected in the scores obtained when we disregard (that is, not charge for) end spaces: (3.4) gets 3 points against the same -12 for (3.5).

Let us now describe a variation of the basic algorithm that will ignore end spaces. Consider initially the case where we do not want to charge for spaces after the last character of s. Take an optimal alignment in this case. The spaces after the end of s are matched to a suffix of t. If we remove this final part of the alignment, the remaining is an alignment between s and a prefix of t, with score equal to the original alignment. Therefore, to get the score of the optimal alignment between s and t without charge for spaces after the end of s, all we need to do is to find the best similarity between s and a prefix of t. But we saw in Section 3.2.1 that the entry (i, j) of matrix a contains the similarity between $s[1..i]$ and $t[1..j]$. Hence, it suffices to take the maximum value in the last row of a, that is,

$$\mathrm{sim}(s, t) = \max_{j=1}^{n} a[m, j].$$

Notice that in this section we have changed the definition of $\mathrm{sim}(s, t)$ so that it now indicates similarity, ignoring final spaces in s. The expression above gives the score of the best alignment. To recover the alignment itself, we proceed just as in the basic algorithm, but starting at (m, k) where k is such that $\mathrm{sim}(s, t) = a[m, k]$.

An analogous argument solves the case in which we do not charge for final spaces in t. We take the maximum along the last column of a in this case. We can even combine the two ideas and seek the best alignment without charging for final spaces in either sequence. The answer will be found by taking the maximum along the border of the matrix formed by the union of the last row and the last column. In all cases, to recover an optimal alignment, we start at an array entry that contains the similarity value and follow the arrows until we reach (0, 0). Each arrow will give one column of the alignment as in the basic case.

Now let us turn our attention to the case of initial spaces. Suppose that we want the best alignment that does not charge for initial spaces in s. This is equivalent to the best alignment between s and a suffix of t. To get the desired answer, we use an $(m + 1) \times (n + 1)$ array just as in the basic algorithm, but with a slight difference. Each entry (i, j) now will contain the highest similarity between $s[1..i]$ and a suffix of a prefix, which is to say a suffix of $t[1..j]$.

Doing that, it is clear that $a[m, n]$ will be the answer. What is less clear, but nevertheless true, is that the array can be filled in using exactly the same formula as in the basic algorithm! That is Equation (3.3). The initialization will be different, however. The first row must be initialized with zeros instead of multiples of the space penalty because of the new meaning of the entries. We leave it to the reader to verify that Equation (3.3) works in this case.

We can apply the same trick and initialize the first column with zeros, and by doing this we will be forgiving spaces before the beginning of t. If in addition we initialize both the first row and the first column with zeros, and proceed with Equation (3.3) for the other entries, we will be computing in each entry the highest similarity between a suffix of s and a suffix of t. To find an optimal alignment, we follow the arrows from the maximum value entry until we reach one of the borders initialized with zeros, and then follow the border back to the origin.

Table 3.1 summarizes these variations. There are four places where we may not want to charge for spaces: beginning or end of s, and beginning or end of t. We can combine these conditions independently in any way and use the variations above to find the similarity. The only things that change are the initialization and where to look for the maximum value. Forgiving initial spaces translates into initializing certain positions with zero. Forgiving final spaces means looking for the maximum along certain positions. But filling in the array is always the same process, using Equation (3.3).

TABLE 3.1

Summary of end space charging procedures.

Place where spaces are not charged for	*Action*
Beginning of first sequence	Initialize first row with zeros
End of first sequence	Look for maximum in last row
Beginning of second sequence	Initialize first column with zeros
End of second sequence	Look for maximum in last column

EXTENSIONS TO THE BASIC ALGORITHMS

3.3

The algorithms presented in Section 3.2 are generally adequate for most applications. Sometimes, however, we have a special situation and we need a better algorithm. In this section we study a few techniques that can be employed in some of these cases.

One kind of improvement is related to the problem's computational complexity. We show that it is possible to reduce the space requirements of the algorithms from quadratic (mn) to linear ($m+n$), at a cost of roughly doubling computation time. On the other hand, we show a way of reducing also the time complexity, but this only works for similar sequences and for a certain family of scoring parameters.

Another improvement has to do with the biological interpretation of alignments. From the biological point of view, it is more realistic to consider a series of consecutive spaces instead of individual spaces. We study variants of the basic algorithms adapted to this point of view.

3.3.1 SAVING SPACE

The quadratic complexity of the basic algorithms makes them unattractive in some applications involving very long sequences or repeated comparison of several sequences. No algorithm is known that uses asymptotically less time and has the same generality, although faster algorithms exist if we restrict ourselves to particular choices of the parameters.

With respect to space, however, it is possible to improve complexity from quadratic to linear and keep the same generality. The price to pay is an increase in processing time, which will roughly double. Nevertheless, the asymptotic time complexity is still the same, and in many cases space and not time is the limiting factor, so this improvement is of great practical value. In this section we describe this elegant space-saving technique.

We begin by noticing that computing $\mathrm{sim}(s, t)$ can be easily done in linear space. Each row of the matrix depends only on the preceding one, and it is possible to perform the calculations keeping only one vector in memory, which will hold partly the new row being computed and partly the previous row. This is done in the code shown in Figure 3.5. Obviously, the same is valid for columns, and if $m < n$, using this trick with the columns uses less space. Notice that at the end of each iteration in the loop on i in Figure 3.5 the vector a contains the similarities between $s[1..i]$ and all prefixes of t. This fact will be used later.

The hard part is to get an optimal alignment in linear space. The algorithm we saw earlier depends on the whole matrix to do its job. To remove this difficulty, we use a *divide and conquer* strategy, that is, we divide the problem into two smaller subproblems and later combine their solutions to obtain a solution for the whole problem.

The key idea is the following. Fix an optimal alignment and a position i in s, and consider what can possibly be matched with $s[i]$ in this alignment. There are only two possibilities:

```
Algorithm BestScore
    input: sequences s and t
    output: vector a
    m ← |s|
    n ← |t|
    for j ← 0 to n do
        a[j] ← j × g
    for i ← 1 to m do
        old ← a[0]
        a[0] ← i × g
        for j ← 1 to n do
            temp ← a[j]
            a[j] ← max(a[j] + g,
                       old + p(i, j),
                       a[j − 1] + g)
            old ← temp
```

FIGURE 3.5

Algorithm for similarity in linear space. In the end, $a[n]$ contains $\mathrm{sim}(s, t)$.

1. The symbol $t[j]$ will match $s[i]$, for some j in $1..n$.
2. A space between $t[j]$ and $t[j+1]$ will match $s[i]$, for some j in $0..n$.

In the second case the index j varies between 0 and n because there is always one more position for spaces than for symbols in a sequence. We also abused notation when $j = 0$ or $j = n$. What we mean in these cases is that the space will be before $t[1]$ or after $t[n]$, respectively.

Let

$$Optimal\begin{pmatrix} x \\ y \end{pmatrix}$$

denote an optimal alignment between x and y. Every alignment between s and t, optimal or not, satisfies (1) or (2). In particular, our fixed optimal alignment must satisfy one of these as well. If it satisfies (1), to obtain all of it we must concatenate

$$Optimal\begin{pmatrix} s[1..i-1] \\ t[1..j-1] \end{pmatrix} + \begin{matrix} s[i] \\ t[j] \end{matrix} + Optimal\begin{pmatrix} s[i+1..m] \\ t[j+1..n] \end{pmatrix}, \tag{3.6}$$

while in case (2) we must concatenate

$$Optimal\begin{pmatrix} s[1..i-1] \\ t[1..j] \end{pmatrix} + \begin{matrix} s[i] \\ - \end{matrix} + Optimal\begin{pmatrix} s[i+1..m] \\ t[j+1..n]) \end{pmatrix}. \tag{3.7}$$

These considerations give us a recursive method to compute an optimal alignment, as long as we can determine, for a given i, which one of the cases (1) or (2) occurs and what is the corresponding value of j.

This can be done as follows. According to Equations (3.6) and (3.7) we need, for fixed i, the similarities between $s[1..i-1]$ and an arbitrary prefix of t, and also the similarities between $s[i+1..m]$ and an arbitrary suffix of t. If we had these values, we could

explicitly compute the scores of the j alignments represented in (3.6) and of the $j+1$ alignments represented in (3.7). By choosing the best among these, we will have the information necessary to proceed in the recursion.

As we saw earlier, it is possible to compute in linear space the best scores between a given prefix of s and all prefixes of t (see Figure 3.5). A similar algorithm exists for suffixes. Hence, our problem is almost solved. The only thing left is to decide which value of i to use in each recursive call. The best choice is to pick i as close as possible to the middle of the sequence. The complete code appears in Figure 3.6. In this code, the call

$$BestScore(s[a..i-1], t[c..d], pref\text{-}sim)$$

returns in *pref-sim* the similarities between $s[a..i-1]$ and $t[c..j]$ for all j in $c-1..d$. Analogously, the call

$$BestScoreRev(s[i+1..b], t[c..d], suff\text{-}sim)$$

returns in *suff-sim* the similarities between $s[i+1..b]$ and $t[j+1..d]$ for all j in $c-1..d$. The call

$$Align(1, m, 1, n, 1, len)$$

will return an optimal alignment in the global variables *align-s* and *align-t*, and the size of this alignment in *len*.

One last concern remains: Can the processing time go up too much with these additional calculations? Not really. In fact, the time roughly doubles, as we show below.

Let $T(m, n)$ be the number of times a maximum is computed in the internal loop of *BestScore* or *BestScoreRev* as a result of a call $Align(a, b, c, d, start, end)$ where $m = b - a + 1$ and $n = c - d + 1$. It is easy to see that the total processing time will be proportional to $T(m, n)$ plus linear terms due to control and initializations. We claim that $T(m, n) \leq 2mn$.

A proof can be developed by induction on m. For $m = 1$ no maximum computations will occur, so obviously $T(1, n) \leq 2n$. For $m > 1$ we will have a call to *BestScore* with at most $mn/2$ maximum computations, another such amount for *BestScoreRev*, and two recursive calls to *Align*, producing at most $T(m/2, j)$ and $T(m/2, n-j)$ maximum computations. Adding this all up, we have

$$\begin{aligned} T(m, n) &\leq \frac{mn}{2} + \frac{mn}{2} + T(m/2, j) + T(m/2, n-j) \\ &\leq mn + mj + m(n-j) \\ &= 2mn, \end{aligned}$$

proving the claim.

3.3.2 GENERAL GAP PENALTY FUNCTIONS

Let us define a **gap** as being a consecutive number $k > 1$ of spaces. It is generally accepted that, when mutations are involved, the occurrence of a gap with k spaces is more probable than the occurrence of k isolated spaces. This is because a gap may be due to a single mutational event that removed a whole stretch of residues, while separated spaces are most probably due to distinct events, and the occurrence of one event is more common than the occurrence of several events.

Up to now, we have not made any distinction between clustered or isolated spaces. This means that a gap is penalized through a linear function. Denoting by $w(k)$, for $k \geq 1$, the penalty associated with a gap with k spaces, we have


```
Algorithm Align
    input: sequences s and t, indices a, b, c, d, start position start
    output: optimal alignment between s[a..b] and t[c..d] placed in vectors align-s
        and align-t beginning at position start and ending at end
    if s[a..b] empty or t[c..d] empty then
        // Base case: s[a..b] empty or t[c..d] empty
        Align the nonempty sequence with spaces
        end ← start + max(|s|, |t|)
    else
        // General case
        i ← ⌊(a + b)/2⌋
        BestScore(s[a..(i − 1)], t[c..d], pref-sim)
        BestScoreRev(s[(i + 1)..b], t[c..d], suff-sim)

        posmax ← c − 1
        typemax ← SPACE
        vmax ← pref-sim[c − 1] + g + suff-sim[c − 1]
        for j ← c to d do
            if pref-sim[j − 1] + p(i, j) + suff-sim[j] > vmax then
                posmax ← j
                typemax ← SYMBOL
                vmax ← pref-sim[j − 1] + p(i, j) + suff-sim[j]

            if pref-sim[j] + g + suff-sim[j] > vmax then
                posmax ← j
                typemax ← SPACE
                vmax ← pref-sim[j] + g + suff-sim[j]

        if typemax = SPACE then
            Align(a, i − 1, c, posmax, start, middle)
            align-s[middle] ← s[i]
            align-t[middle] ← SPACE
            Align(i + 1, b, posmax + 1, d, middle + 1, end)
        else // typemax = SYMBOL
            Align(a, i − 1, c, posmax − 1, start, middle)
            align-s[middle] ← s[i]
            align-t[middle] ← t[posmax]
            Align(i + 1, b, posmax + 1, d, middle + 1, end)
```

FIGURE 3.6

An optimal alignment algorithm that uses linear space.

$$w(k) = bk,$$

where b is the absolute value of the score associated with a space.

In this section we present an algorithm that computes similarities with respect to general gap penalty functions w. The algorithm has time complexity $O(n^3)$ for sequences of length n and is therefore slower than the basic algorithm. The algorithm still has a lot in common with the basic algorithm. The main difference is the fact that the scoring scheme is not *additive,* in the sense that we cannot break an alignment in two parts and expect the total score to be the sum of the partial scores. In particular, we cannot separate the last column of an alignment and expect the alignment score to be the sum of the score for this last column and the score for the remaining prefix of the alignment. However, score additivity is still valid if we break the alignment in block boundaries. Every alignment can be uniquely decomposed into a number of consecutive *blocks.* There are three kinds of blocks, enumerated below.

1. Two aligned characters from Σ
2. A maximal series of consecutive characters in t aligned with spaces in s
3. A maximal series of consecutive characters in s aligned with spaces in t

The term *maximal* means that it cannot be extended further. Figure 3.7 shows an alignment and its blocks. Blocks in the first category above receive score $p(a, b)$, where a and b are the two aligned characters. Blocks in categories (2) and (3) receive score $-w(k)$, where k is the length of the gap series.

```
s1   AAC---AATTCCGACTAC
s2   ACTACCT------CGC--

s1   A | A | C | --- | A | TATCCG | A | C | T | AC
s2   A | C | T | ACC | T | ------ | C | G | C | --
```

FIGURE 3.7

An alignment and its blocks.

The scoring of an alignment is not done at the column level now but rather at the block level. Accordingly, only if we break in block boundaries can we expect score additivity to hold. This implies some significant changes in our dynamic programming algorithm to compute the similarity under a general gap penalty function. Instead of reasoning on the last column of the alignment, we must reason on the last block. Furthermore, blocks cannot follow other blocks arbitrarily. A block of type 2 or 3 above cannot follow another block of the same type. This requires that we keep, for each pair (i, j), not only the best score of an alignment between prefixes $s[1..i]$ and $t[1..j]$, but rather the best score of these prefixes that ends in a particular type of block.

To compare sequence s of length m to sequence t of length n, we use three arrays of size $(m+1) \times (n+1)$, one for each type of ending block. Array a is used for alignments ending in character-character blocks; b is used for alignments ending in spaces in s; and c is used for alignments ending with spaces in t.

Initialization of the first row and column is done as follows, according to the meaning of the arrays: 9

$$a[0, 0] = 0$$
$$b[0, j] = -w(j)$$
$$c[i, 0] = -w(i).$$

All other values in the initial row and column should be set to $-\infty$ to make them harmless in the maximum computations that use them.

We now ask the standard question — What type of block terminates our optimal alignment? The answer determines which array, a, b, or c, will be updated. The recurrence relations are as follows:

$$a[i, j] = p(i, j) + \max \begin{cases} a[i-1, j-1] \\ b[i-1, j-1] \\ c[i-1, j-1] \end{cases}$$

$$b[i, j] = \max \begin{cases} a[i, j-k] - w(k), & \text{for } 1 \le k \le j \\ c[i, j-k] - w(k), & \text{for } 1 \le k \le j \end{cases}$$

$$c[i, j] = \max \begin{cases} a[i-k, j] - w(k), & \text{for } 1 \le k \le i \\ b[i-k, j] - w(k), & \text{for } 1 \le k \le i. \end{cases}$$

As usual, $p(i, j)$ indicates the score of a matching between $s[i]$ and $t[j]$.

Notice that entries in arrays b and c depend on more than one earlier value, because the last block can have variable length. Also, when computing an entry of b, we do not look at previous b entries, because type 2 blocks cannot immediately follow type 2 blocks. Likewise, c entries do not depend directly on earlier c entries. To obtain the final answer, that is, the value of $\text{sim}(s, t)$, we take the maximum among $a[m, n]$, $b[m, n]$, and $c[m, n]$.

The time complexity of the algorithm is $O(mn^2 + m^2n)$. To see this, count the number of times some entry of some array is read, based on the given formulas. This is the dominant term in the time complexity. It is easy to see that to compute $a[i, j]$, $b[i, j]$, and $c[i, j]$, we need to perform

$$3 + 2j + 2i$$

accesses to previous array entries, which sum up to

$$\sum_{i=1}^{m} \sum_{j=1}^{n} (3 + 2j + 2i)$$

accesses for the entire arrays. The sum above can be computed in closed form as follows.

$$\sum_{j=1}^{n} (2i + 2j + 3) = 2ni + n(n+1) + 3n = 2ni + n^2 + 4n$$

$$\sum_{i=1}^{m} (2ni + n^2 + 4n) = nm(m+1) + mn^2 + 4mn = m^2n + 5mn + mn^2.$$

Getting an optimal alignment is straightforward. The same ideas used in the previous algorithms work. We trace back in the arrays the entries that contributed to the

maximum computations, at the same time keeping track of which array we used.

Thus, we see that it is possible to use any gap penalty function. However, the price to pay for this generality is a considerable increase in computing time and in storage space (the three auxiliary arrays). This can be critical when sequence comparison by dynamic programming is the bottleneck of a larger computation.

3.3.3 AFFINE GAP PENALTY FUNCTIONS

We have seen that the basic algorithm with a linear gap function runs in $O(n^2)$ time, for sequences of length n; we have also seen that allowing general gap functions makes the running time go up to $O(n^3)$. The question we now pose is: If we use a less general gap penalty function but one that still charges less for a gap with k spaces than for k isolated spaces, can we still get an $O(n^2)$ algorithm? In this section we show that this is in fact possible.

If we want to introduce the idea that k spaces together are more probable than k isolated spaces (or at least equally probable), we must have

$$w(k) \leq kw(1)$$

or, in general,

$$w(k_1 + k_2 + \cdots + k_n) \leq w(k_1) + w(k_2) + \cdots + w(k_n). \tag{3.8}$$

A function w that satisfies Equation (3.8) is called a *subadditive function*. An *affine function* is a function w of the form $w(k) = h + gk$, $k \geq 1$, with $w(0) = 0$. It will be subadditive if $h, g > 0$.

Another way of thinking about a function w as above is by saying that the first space in a gap costs $h + g$, while the other spaces cost g. This cost is based on the difference $\Delta w(k) = w(k) - w(k-1)$ of the penalty from $k-1$ spaces to k spaces.

Let us now describe the dynamic programming algorithm for this case. The main difference between the basic algorithm and this one is that here we need to make a distinction between the first space in a gap and the others, so that they can be penalized accordingly. This is accomplished by the use of three arrays, a, b, and c, as was done for the general gap function case. The entries in each one of these arrays have the following meaning:

$a[i, j]$ = maximum score of an alignment between $s[1..i]$ and $t[1..j]$ that ends in $s[i]$ matched with $t[j]$.
$b[i, j]$ = maximum score of an alignment between $s[1..i]$ and $t[1..j]$ that ends in a space matched with $t[j]$.
$c[i, j]$ = maximum score of an alignment between $s[1..i]$ and $t[1..j]$ that ends in $s[i]$ matched with a space.

The entries (i, j) of these arrays depend on previous entries according to the following formulas, valid for $1 \leq i \leq m$ and $1 \leq j \leq n$:

$$a[i, j] = p(i, j) + \max \begin{cases} a[i-1, j-1] \\ b[i-1, j-1] \\ c[i-1, j-1] \end{cases}$$

10

$$b[i, j] = \max \begin{cases} -(h+g) + a[i, j-1] \\ -g + b[i, j-1] \\ -(h+g) + c[i, j-1] \end{cases}$$

$$c[i, j] = \max \begin{cases} -(h+g) + a[i-1, j] \\ -(h+g) + b[i-1, j] \\ -g + c[i-1, j]. \end{cases}$$

As before, $p(i, j)$ indicates the score of a matching between $s[i]$ and $t[j]$.

Let us understand the preceding formulas. The formula for $a[i, j]$ includes the term $p(i, j)$, which is the score of the last column, plus the best score of an alignment between the prefixes $s[1..i-1]$ and $t[1..j-1]$.

For $b[i, j]$, we know that the last column will contain a space. We still have to check whether this is the first space of a gap or a continuation space, so that we can penalize it correctly. Furthermore, we need to read from the arrays the values of the best scores for the prefixes involved, namely, $s[1..i]$ and $t[1..j-1]$. That is why we always look at entries $(i, j-1)$ in the formulas for $b[i, j]$. Alignments corresponding to $a[i, j-1]$ and $c[i, j-1]$ do not end with a space in s, hence the final space must be penalized as the first of a gap, that is, subtracting $h + g$. Alignments corresponding to $b[i, j-1]$ already have a space in s at the end, so the space matched with $t[j]$ must be penalized as a continuation space, that is, subtracting g. These three possibilities are exhaustive, so taking the maximum among them gives the correct value. A similar argument explains the formula for $c[i, j]$.

The initialization of the arrays requires some care. Again, it will depend on whether we want to charge for initial spaces. To make it concrete, let us assume that we want to charge for all spaces, leaving to the reader the task of adapting the algorithm to the other cases.

The entries that need initialization are those with indices of the form $(i, 0)$ for $0 \leq i \leq m$ or $(0, j)$ for $0 \leq j \leq n$. The initialization has to be done according to the contents of each array as given in the definition. Thus, for array b, for instance, we have

$$b[i, 0] = -\infty \quad \text{for} \quad 0 \leq i \leq m \qquad \text{and}$$
$$b[0, j] = -(h + gj) \quad \text{for} \quad 1 \leq j \leq n.$$

We assign the value $-\infty$ to $b[i, 0]$ to be consistent with the definition, because there is no alignment between $s[1..i]$ and $t[1..0]$ (an empty sequence) in which a space is matched with $t[0]$, since there is no $t[0]$. Hence, if there are no alignments, the maximum value must be $-\infty$, which is the identity for maximum computations. On the other hand, there is exactly one alignment ending with a space in s between $s[1..0]$ (empty sequence again) and $t[1..j]$, and this alignment has score $-(h + gj)$.

In an analogous manner we initialize c letting

$$c[i, 0] = -(h + gi) \quad \text{for} \quad 1 \leq i \leq m \qquad \text{and}$$
$$c[0, j] = -\infty \quad \text{for} \quad 0 \leq j \leq n,$$

and, finally,

$$a[0, 0] = 0,$$
$$a[i, 0] = -\infty \quad \text{for} \quad 1 \leq i \leq m \qquad \text{and}$$
$$a[0, j] = -\infty \quad \text{for} \quad 1 \leq j \leq n.$$

In the initialization of a we used alignments *that do not end in spaces,* generalizing the given definition. To get the final result, that is, the similarity, it suffices to take the maximum among $a[m, n]$, $b[m, n]$, and $c[m, n]$, given that this covers all possibilities.

We can construct an optimal alignment in a manner analogous to the case of general gap functions. Just trace back from the final position (m, n) the entries that were chosen as maxima in the process of filling the arrays until the initial position $(0, 0)$ is reached. During this traceback, it is necessary to remember not only the current position but also which array (a, b, or c) it belongs to.

We leave to the reader the task of verifying that the time complexity of this algorithm is indeed $O(mn)$. The version just described has space complexity $O(mn)$ as well, but linear-space versions have been developed (see the bibliographic notes).

3.3.4 COMPARING SIMILAR SEQUENCES

Two sequences are similar when they "look alike" in some sense. We have already developed our intuition about the concept of similarity and have formalized it through alignments and scores. So, for us, two sequences are similar when the scores of the optimal alignments between them are very close to the maximum possible (where "maximum possible" will be made precise). This section is about faster algorithms to find good alignments in this case. We analyze global alignments only.

Let us first treat the case where the two sequences of interest, s and t, have the same length n. This is a fair assumption, given that we are focusing on similar sequences. If s and t have the same length, the dynamic programming matrix is a square matrix and its main diagonal runs from position $(0, 0)$ to (n, n). Following this diagonal corresponds to the unique alignment without spaces between s and t. If this is not an optimal alignment, we need to insert some spaces in the sequences to obtain a better score. Notice that spaces will always be inserted in pairs, one in s and one in t.

As we insert these space pairs, the alignment is thrown off the main diagonal. Consider for a moment the following two sequences:

$$s = \text{GCGCATGGATTGAGCGA}$$
$$t = \text{TGCGCCATGGATGAGCA}$$

The optimal alignments correspond to paths that reach up to the diagonal twice removed from the main one, as in Figure 3.8. The best alignments include two pairs of spaces. In this case, the number of space pairs is equal to how far the alignment departed from the main diagonal, but this does not always happen. One thing is certain, though: The number of space pairs is greater than or equal to the maximum departure.

The basic idea then goes as follows. If the sequences are similar, the best alignments have their paths near the main diagonal. To compute the optimal score and alignments, it is not necessary to fill in the entire matrix. A narrow band around the main diagonal should suffice.

The algorithm *KBand* in Figure 3.9 performs the matrix fill-in in a band of horizontal (or vertical) width $2k + 1$ around the main diagonal. At the end, the entry $a[n, n]$ contains the highest score of an alignment confined to that band. This algorithm runs in time $O(kn)$, which is a big win over the usual $O(n^2)$ if k is small compared to n.


```
-GCGC-ATGGATTGAGCGA
TGCGCCATGGAT-GAGC-A
```

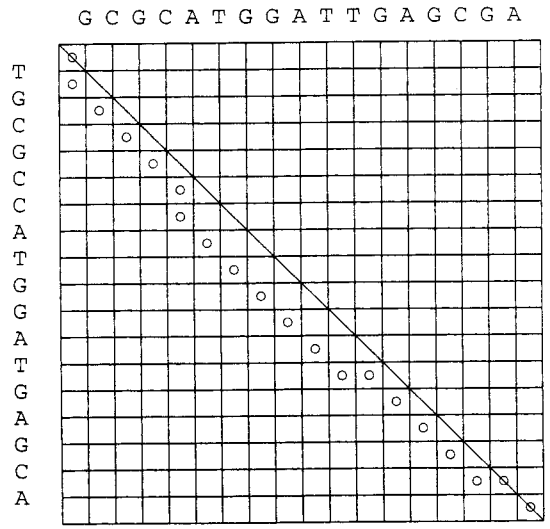

FIGURE 3.8

An optimal alignment and its corresponding path in the dynamic programming matrix. A line is drawn along the main diagonal.

Observe that we do not use entries outside of the k-strip at all. We do not initialize them, and we do not consult them for maximum computations. To test whether a certain position (i, j) is inside the k-strip, we use the following criterion:

$$InsideStrip(i, j, k) = (-k \leq i - j \leq k).$$

As in the other dynamic programming algorithms, each entry $a[i, j]$ depends on $a[i - 1, j]$, $a[i - 1, j - 1]$, and $a[i, j - 1]$. In the code, we need not test position $(i - 1, j - 1)$, because it is in the same diagonal as (i, j), so it will always be inside the strip. Tests must be performed for $(i - 1, j)$ and $(i, j - 1)$, which may be outside the strip when (i, j) is in the border.

How can we use algorithm *KBand*? We choose a value for k and run the algorithm. If $a[n, n]$ is greater than or equal to the best score that could possibly come from an alignment with $k + 1$ or more space pairs, we are lucky: We have found an optimal alignment with just $O(kn)$ steps. The best possible score, given that we have at least $k + 1$ space pairs, is

$$M(n - k - 1) + 2(k + 1)g, \tag{3.9}$$

which is computed assuming that there are exactly $k + 1$ space pairs and that the other pairs are matches. Here M is the score of a match, and g is added for each space. We will assume that $M > 0$ and $g \leq 0$.

```
Algorithm KBand
    input: sequences s and t of equal length n, integer k
    output: best score of alignment at most k diagonals away from main diagonal
    n ← |s|
    for i ← 0 to k do
        a[i, 0] ← i × g
    for j ← 0 to k do
        a[0, j] ← j × g
    for i ← 1 to n do
        for d ← −k to k do
            j ← i + d
            if 1 ≤ j ≤ n do
                // compute maximum among predecessors
                a[i, j] ← a[i − 1, j − 1] + p(i, j)
                if InsideStrip(i − 1, j, k) then
                    a[i, j] ← max(a[i, j], a[i − 1, j] + g)
                if InsideStrip(i, j − 1, k) then
                    a[i, j] ← max(a[i, j], a[i, j − 1] + g)
    return a[n, n]
```

FIGURE 3.9

Algorithm for k-strip around main diagonal.

If $a[n, n]$ is smaller than the quantity (3.9), we double k and run the algorithm again. If the initial value of k is 1, further values will be powers of two. The stop condition becomes

$$a_k[n, n] \geq M(n - k - 1) + 2(k + 1)g,$$

which is equivalent to

$$k \geq \frac{Mn - a_k[n, n]}{M - 2g} - 1.$$

At this point, we have already run the algorithm many times, each time with a larger k, so that the total complexity is

$$n + 2n + 4n + \cdots + kn \leq 2kn,$$

assuming we use powers of two. To bound this total complexity, we need an upper bound on k. Now, we did not stop earlier, so

$$\frac{k}{2} < \frac{Mn - a_{k/2}[n, n]}{M - 2g} - 1.$$

If $a_k[n, n] = a_{k/2}[n, n]$, then this is the optimum score $\mathrm{sim}(s, t)$, and our bound is

$$k < 2\left(\frac{Mn - \mathrm{sim}(s, t)}{M - 2g} - 1\right).$$

If $a_k[n, n] > a_{k/2}[n, n]$, then optimal alignments have more than $k/2$ pairs of spaces, and therefore

$$\mathrm{sim}(s, t) \leq M\left(n - \frac{k}{2} - 1\right) + 2\left(\frac{k}{2} + 1\right)g,$$

which leads to

$$k \leq 2\left(\frac{Mn - \mathrm{sim}(s, t)}{M - 2g} - 1\right),$$

almost the same bound as above.

Observe that $M - 2g$ is a constant. The time complexity is then $O(dn)$, where d is the difference between the maximum possible score Mn — the score of two identical sequences — and the optimal score. Thus, the higher the similarity, the faster the answer.

It is straightforward to extend this method to general sequences, not necessarily with the same length. Space-saving versions can also be easily derived.

3.4 COMPARING MULTIPLE SEQUENCES

So far in this chapter we have concentrated on the comparison between a pair of sequences. However we often are given several sequences that we have to align simultaneously in the best possible way. This happens, for example, when we have the sequences for certain proteins that have similar function in a number of different species. We may want to know which parts of these sequences are similar and which parts are different. To get this information, we need to build a **multiple alignment** for these sequences, and that is the topic of this section.

The notion of multiple alignment is a natural generalization of the two-sequence case. Let $s_1, \ldots, s_k$ be a set of sequences over the same alphabet. A multiple alignment involving $s_1, \ldots, s_k$ is obtained by inserting spaces in the sequences in such a way as to make them all of the same size. It is customary to place the extended sequences in a vertical list so that characters — or spaces — in corresponding positions occupy the same column. We further require that no column be made exclusively of spaces. Figure 3.10 shows a multiple alignment involving four short amino acid sequences. (Multiple alignments are more common with proteins, so in some of our examples in this section we use sequences of amino acids.)

One important issue to decide in multiple alignment is the precise definition of the

```
MQPILLL
MLR-LL-
MK-ILLL
MPPVLIL
```

FIGURE 3.10

Multiple alignment of four amino acid sequences.