Kun-Mao Chao / Louxin Zhang
Sequence Comparison. Theory and Methods.
pp 17-62

Chapter 2
Basic Algorithmic Techniques

An *algorithm* is a step-by-step procedure for solving a problem by a computer. Although the act of designing an algorithm is considered as an art and can never be automated, its general strategies are learnable. Here we introduce a few frameworks of computer algorithms including greedy algorithms, divide-and-conquer strategies, and dynamic programming.

This chapter is divided into five sections. It starts with the definition of algorithms and their complexity in Section 2.1. We introduce the asymptotic O-notation used in the analysis of the running time and space of an algorithm. Two tables are used to demonstrate that the asymptotic complexity of an algorithm will ultimately determine the size of problems that can be solved by the algorithm.

Then, we introduce greedy algorithms in Section 2.2. For some optimization problems, greedy algorithms are more efficient. A greedy algorithm pursues the best choice at the moment in the hope that it will lead to the best solution in the end. It works quite well for a wide range of problems. Huffman's algorithm is used as an example of a greedy algorithm.

Section 2.3 describes another common algorithmic technique, called divide-and-conquer. This strategy divides the problem into smaller parts, conquers each part individually, and then combines them to form a solution for the whole. We use the mergesort algorithm to illustrate the divide-and-conquer algorithm design paradigm.

Following its introduction by Needleman and Wunsch, dynamic programming has become a major algorithmic strategy for many optimization problems in sequence comparison. The development of a dynamic-programming algorithm has three basic components: the recurrence relation for defining the value of an optimal solution, the tabular computation for computing the value of an optimal solution, and the backtracking procedure for delivering an optimal solution. In Section 2.4, we introduce these basic ideas by developing dynamic-programming solutions for problems from different application areas, including the maximum-sum segment problem, the longest increasing subsequence problem, and the longest common subsequence problem.

Finally, we conclude the chapter with the bibliographic notes in Section 2.5.

2.1 Algorithms and Their Complexity

An *algorithm* is a step-by-step procedure for solving a problem by a computer. When an algorithm is executed by a computer, the central processing unit (CPU) performs the operations and the memory stores the program and data.

Let n be the size of the input, the output, or their sum. The time or space complexity of an algorithm is usually denoted as a function $f(n)$. Table 2.1 calculates the time needed if the function stands for the number of operations required by an algorithm, and we assume that the CPU performs one million operations per second.

Exponential algorithms grow pretty fast and become impractical even when n is small. For those quadratic and cubic functions, they grow faster than the linear functions. The constant and log factor matter, but are mostly acceptable in practice. As a rule of thumb, algorithms with a quadratic time complexity or higher are often impractical for large data sets.

Table 2.2 further shows the growth of the input size solvable by polynomial and exponential time algorithms with improved computers. Even with a million-times faster computer, the 10^n algorithm only adds 6 to the input size, which makes it hopeless for handling a moderate-size input.

These observations lead to the definition of the O-notation, which is very useful for the analysis of algorithms. We say $f(n) = O(g(n))$ if and only if there exist two positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. In other words, for sufficiently large n, $f(n)$ can be bounded by $g(n)$ times a constant. In this kind of asymptotic analysis, the most crucial part is the order of the function, not the constant. For example, if $f(n) = 3n^2 + 5n$, we can say $f(n) = O(n^2)$ by letting $c = 4$ and $n_0 = 10$. By definition, it is also correct to say $n^2 = O(n^3)$, but we always prefer to choose a tighter order if possible. On the other hand, $10^n \neq O(n^x)$ for any integer x. That is, an exponential function cannot be bounded above by any polynomial function.

2.2 Greedy Algorithms

A greedy method works in stages. It always makes a locally optimal (*greedy*) choice at each stage. Once a choice has been made, it cannot be withdrawn, even if later we realize that it is a poor decision. In other words, this greedy choice may or may not lead to a globally optimal solution, depending on the characteristics of the problem.

Table 2.1 The time needed by the functions where we assume one million operations per second.

$f(n)$	$n = 10$	$n = 100$	$n = 100000$
$30n$	0.0003 second	0.003 second	3 seconds
$100n \log_{10} n$	0.001 second	0.02 second	50 seconds
$3n^2$	0.0003 second	0.03 second	8.33 hours
n^3	0.001 second	1 second	31.71 years
10^n	2.78 hours	3.17×10^{84} centuries	3.17×10^{99984} centuries

Table 2.2 The growth of the input size solvable in an hour as the computer runs faster.

$f(n)$	Present speed	1000-times faster	10^6-times faster
n	x_1	$1000x_1$	$10^6 x_1$
n^2	x_2	$31.62x_2$	$10^3 x_2$
n^3	x_3	$10x_3$	$10^2 x_3$
10^n	x_4	$x_4 + 3$	$x_4 + 6$

It is a very straightforward algorithmic technique and has been used to solve a variety of problems. In some situations, it is used to solve the problem exactly. In others, it has been proved to be effective in approximation.

What kind of problems are suitable for a greedy solution? There are two ingredients for an optimization problem to be exactly solved by a greedy approach. One is that it has the so-called greedy-choice property, meaning that a locally optimal choice can reach a globally optimal solution. The other is that it satisfies the principle of optimality, *i.e.*, each solution substructure is optimal. We use Huffman coding, a frequency-dependent coding scheme, to illustrate the greedy approach.

2.2.1 Huffman Codes

Suppose we are given a very long DNA sequence where the occurrence probabilities of nucleotides A (adenine), C (cytosine), G (guanine), T (thymine) are 0.1, 0.1, 0.3, and 0.5, respectively. In order to store it in a computer, we need to transform it into a binary sequence, using only 0's and 1's. A trivial solution is to encode A, C, G, and T by "00," "01," "10," and "11," respectively. This representation requires two bits per nucleotide. The question is "Can we store the sequence in a more compressed way?" Fortunately, by assigning longer codes for frequent nucleotides G and T, and shorter codes for rare nucleotides A and C, it can be shown that it requires less than two bits per nucleotide on average.

In 1952, Huffman [94] proposed a greedy algorithm for building up an optimal way of representing each letter as a binary string. It works in two phases. In phase one, we build a binary tree based on the occurrence probabilities of the letters. To do so, we first write down all the letters, together with their associated probabilities. They are initially the unmarked terminal nodes of the binary tree that we will build up as the algorithm proceeds. As long as there is more than one unmarked node left, we repeatedly find the two unmarked nodes with the smallest probabilities, mark them, create a new unmarked internal node with an edge to each of the nodes just marked, and set its probability as the sum of the probabilities of the two nodes.

The tree building process is depicted in Figure 2.1. Initially, there are four unmarked nodes with probabilities 0.1, 0.1, 0.3, and 0.5. The two smallest ones are

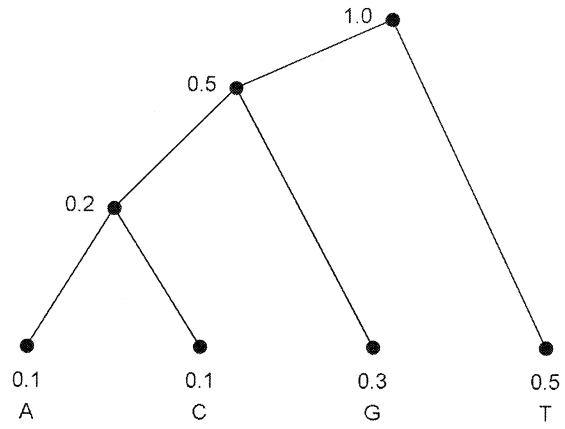

Fig. 2.1 Building a binary tree based on the occurrence probabilities of the letters.

with probabilities 0.1 and 0.1. Thus we mark these two nodes and create a new node with probability 0.2 and connect it to the two nodes just marked. Now we have three unmarked nodes with probabilities 0.2, 0.3, and 0.5. The two smallest ones are with probabilities 0.2 and 0.3. They are marked and a new node connecting them with probabilities 0.5 is created. The final iteration connects the only two unmarked nodes with probabilities 0.5 and 0.5. Since there is only one unmarked node left, *i.e.*, the root of the tree, we are done with the binary tree construction.

After the binary tree is built in phase one, the second phase is to assign the binary strings to the letters. Starting from the root, we recursively assign the value "zero" to the left edge and "one" to the right edge. Then for each leaf, *i.e.*, the letter, we concatenate the 0's and 1's from the root to it to form its binary string representation. For example, in Figure 2.2 the resulting codewords for A, C, G, and T are "000," "000," "01," and "1," respectively. By this coding scheme, a 20-nucleotide DNA sequence "GTTGTTATCGTTTATGTGGC" will be represented as a 34-bit binary sequence "01110111000100101011110001011010100 1." In general, since $3 \times 0.1 + 3 \times 0.1 + 2 \times 0.3 + 1 \times 0.5 = 1.7$, we conclude that, by Huffman coding techniques, each nucleotide requires 1.7 bits on average, which is superior to 2 bits by a trivial solution. Notice that in a Huffman code, no codeword is also a prefix of any other codeword. Therefore we can decode a binary sequence without any ambiguity. For example, if we are given "01110111000100101011110001011010100 1," we decode the binary sequence as "01" (G), "1" (T), "1" (T), "01" (G), and so forth.

The correctness of Huffman's algorithm lies in two properties: (1) greedy-choice property and (2) optimal-substructure property. It can be shown that there exists an optimal binary code in which the codewords for the two smallest-probability nodes have the same length and differ only in the last bit. That's the reason why we can contract them greedily without missing the path to the optimal solution. Besides, after contraction, the optimal-substructure property allows us to consider only those unmarked nodes.

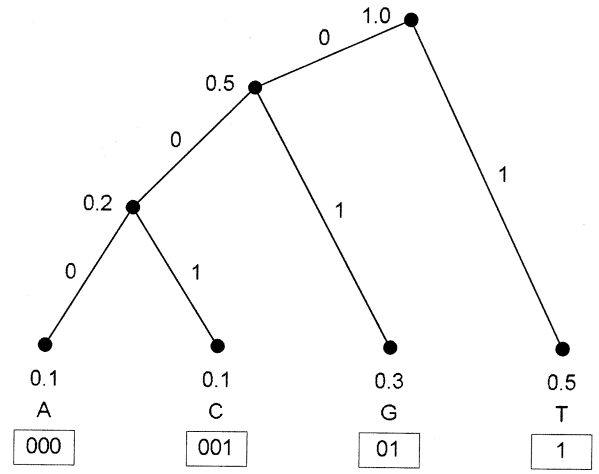

Fig. 2.2 Huffman code assignment.

Let n be the number of letters under consideration. For DNA, n is 4 and for English, n is 26. Since a heap can be used to maintain the minimum dynamically in $O(\log n)$ time for each insertion or deletion, the time complexity of Huffman's algorithm is $O(n \log n)$.

2.3 Divide-and-Conquer Strategies

The divide-and-conquer strategy *divides* the problem into a number of smaller subproblems. If the subproblem is small enough, it *conquers* it directly. Otherwise, it *conquers* the subproblem recursively. Once the solution to each subproblem has been done, it combines them together to form a solution to the original problem.

One of the well-known applications of the divide-and-conquer strategy is the design of sorting algorithms. We use mergesort to illustrate the divide-and-conquer algorithm design paradigm.

2.3.1 Mergesort

Given a sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$, the sorting problem is to sort these numbers into a nondecreasing sequence. For example, if the given sequence is $\langle 65, 16, 25, 85, 12, 8, 36, 77 \rangle$, then its sorted sequence is $\langle 8, 12, 16, 25, 36, 65, 77, 85 \rangle$.

To sort a given sequence, mergesort splits the sequence into half, sorts each of them recursively, then combines the resulting two sorted sequences into one

sorted sequence. Figure 2.3 illustrates the dividing process. The original input sequence consists of eight numbers. We first divide it into two smaller sequences, each consisting of four numbers. Then we divide each four-number sequence into two smaller sequences, each consisting of two numbers. Here we can sort the two numbers by comparing them directly, or divide it further into two smaller sequences, each consisting of only one number. Either way we'll reach the boundary cases where sorting is trivial. Notice that a sequential recursive process won't expand the subproblems simultaneously, but instead it solves the subproblems at the same recursion depth one by one.

How to combine the solutions to the two smaller subproblems to form a solution to the original problem? Let us consider the process of merging two sorted sequences into a sorted output sequence. For each merging sequence, we maintain a cursor pointing to the smallest element not yet included in the output sequence. At each iteration, the smaller of these two smallest elements is removed from the merging sequence and added to the end of the output sequence. Once one merging sequence has been exhausted, the other sequence is appended to the end of the output sequence. Figure 2.4 depicts the merging process. The merging sequences are $\langle 16,25,65,85\rangle$ and $\langle 8,12,36,77\rangle$. The smallest elements of the two merging sequences are 16 and 8. Since 8 is a smaller one, we remove it from the merging sequence and add it to the output sequence. Now the smallest elements of the two merging sequences are 16 and 12. We remove 12 from the merging sequence and append it to the output sequence. Then 16 and 36 are the smallest elements of the two merging sequences, thus 16 is appended to the output list. Finally, the resulting output sequence is $\langle 8,12,16,25,36,65,77,85\rangle$. Let N and M be the lengths of the

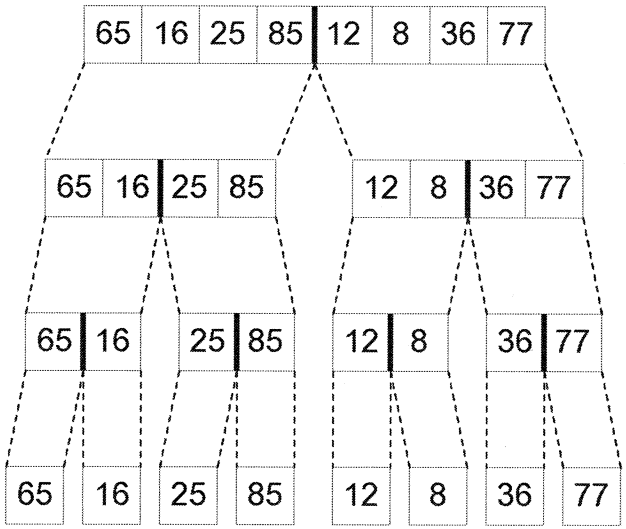

Fig. 2.3 The top-down dividing process of mergesort.

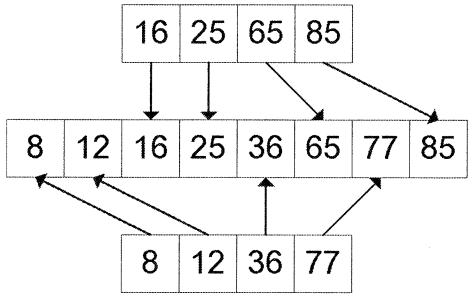

Fig. 2.4 The merging process of mergesort.

two merging sequences. Since the merging process scans the two merging sequences linearly, its running time is therefore $O(N+M)$ in total.

After the top-down dividing process, mergesort accumulates the solutions in a bottom-up fashion by combining two smaller sorted sequences into a larger sorted sequence as illustrated in Figure 2.5. In this example, the recursion depth is $\lceil \log_2 8 \rceil = 3$. At recursion depth 3, every single element is itself a sorted sequence. They are merged to form sorted sequences at recursion depth 2: $\langle 16,65\rangle$, $\langle 25,85\rangle$, $\langle 8,12\rangle$, and $\langle 36,77\rangle$. At recursion depth 1, they are further merged into two sorted sequences: $\langle 16,25,65,85\rangle$ and $\langle 8,12,36,77\rangle$. Finally, we merge these two sequences into one sorted sequence: $\langle 8,12,16,25,36,65,77,85\rangle$.

It can be easily shown that the recursion depth of mergesort is $\lceil \log_2 n \rceil$ for sorting n numbers, and the total time spent for each recursion depth is $O(n)$. Thus, we conclude that mergesort sorts n numbers in $O(n \log n)$ time.

2.4 Dynamic Programming

Dynamic programming is a class of solution methods for solving sequential decision problems with a compositional cost structure. It is one of the major paradigms of algorithm design in computer science. Like the usage in *linear programming*, the word "*programming*" refers to *finding an optimal plan* of action, rather than *writing programs*. The word "*dynamic*" in this context conveys the idea that choices may depend on the current state, rather than being decided ahead of time.

Typically, dynamic programming is applied to optimization problems. In such problems, there exist many possible solutions. Each solution has a value, and we wish to find a solution with the optimum value. There are two ingredients for an optimization problem to be suitable for a dynamic-programming approach. One is that it satisfies the principle of optimality, *i.e.*, each solution substructure is optimal. Greedy algorithms require this very same ingredient, too. The other ingredient is that it has overlapping subproblems, which has the implication that it can be solved

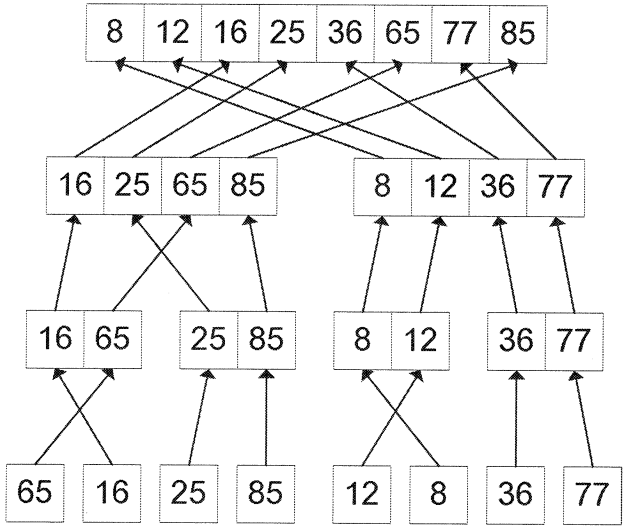

Fig. 2.5 Accumulating the solutions in a bottom-up manner.

more efficiently if the solutions to the subproblems are recorded. If the subproblems are not overlapping, a divide-and-conquer approach is the choice.

The development of a dynamic-programming algorithm has three basic components: the recurrence relation for defining the value of an optimal solution, the tabular computation for computing the value of an optimal solution, and the backtracking procedure for delivering an optimal solution. Here we introduce these basic ideas by developing dynamic-programming solutions for problems from different application areas.

First of all, the Fibonacci numbers are used to demonstrate how a tabular computation can avoid recomputation. Then we use three classic problems, namely, the maximum-sum segment problem, the longest increasing subsequence problem, and the longest common subsequence problem, to explain how dynamic-programming approaches can be used to solve the sequence-related problems.

2.4.1 Fibonacci Numbers

The Fibonacci numbers were first created by Leonardo Fibonacci in 1202. It is a simple series, but its applications are nearly everywhere in nature. It has fascinated mathematicians for more than 800 years. The *Fibonacci numbers* are defined by the following recurrence:

$$\begin{cases} F_0 = 0, \\ F_1 = 1, \\ F_i = F_{i-1} + F_{i-2} \text{ for } i \geq 2. \end{cases}$$

By definition, the sequence goes like this: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, and so forth. Given a positive integer n, how would you compute F_n? You might say that it can be easily solved by a straightforward divide-and-conquer method based on the recurrence. That's right. But is it efficient? Take the computation of F_{10} for example (see Figure 2.6). By definition, F_{10} is derived by adding up F_9 and F_8. What about the values of F_9 and F_8? Again, F_9 is derived by adding up F_8 and F_7; F_8 is derived by adding up F_7 and F_6. Working toward this direction, we'll finally reach the values of F_1 and F_0, *i.e.*, the end of the recursive calls. By adding them up backwards, we have the value of F_{10}. It can be shown that the number of recursive calls we have to make for computing F_n is exponential in n.

Those who are ignorant of history are doomed to repeat it. A major drawback of this divide-and-conquer approach is to solve many of the subproblems repeatedly. A tabular method solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Figure 2.7 explains that F_n can be computed in $O(n)$ steps by a tabular computation. It should be noted that F_n can be computed in just $O(\log n)$ steps by applying matrix computation.

2.4.2 The Maximum-Sum Segment Problem

Given a sequence of numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$, the maximum-sum segment problem is to find, in A, a consecutive subsequence, *i.e.*, a substring or segment, with the maximum sum. For each position i, we can compute the maximum-sum segment ending at that position in $O(i)$ time. Therefore, a naive algorithm runs in $\sum_{i=1}^{n} O(i) = O(n^2)$ time.

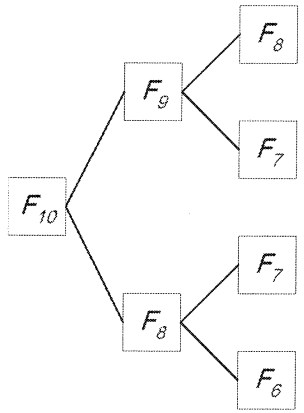

Fig. 2.6 Computing F_{10} by divide-and-conquer.

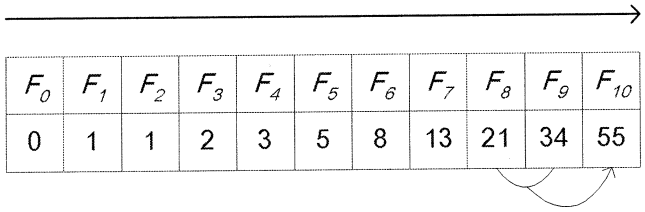

Fig. 2.7 Computing F_{10} by a tabular computation.

Now let us describe a more efficient dynamic-programming algorithm for this problem. Define $S[i]$ to be the maximum sum of segments ending at position i of A. The value $S[i]$ can be computed by the following recurrence:

$$S[i] = \begin{cases} a_i + \max\{S[i-1], 0\} & \text{if } i > 1, \\ a_1 & \text{if } i = 1. \end{cases}$$

If $S[i-1] < 0$, concatenating a_i with its previous elements will give a smaller sum than a_i itself. In this case, the maximum-sum segment ending at position i is a_i itself.

By a tabular computation, each $S[i]$ can be computed in constant time for i from 1 to n, therefore all S values can be computed in $O(n)$ time. During the computation, we record the largest S entry computed so far in order to report where the maximum-sum segment ends. We also record the traceback information for each position i so that we can trace back from the end position of the maximum-sum segment to its start position. If $S[i-1] > 0$, we need to concatenate with previous elements for a larger sum, therefore the traceback symbol for position i is "←." Otherwise, "↑" is recorded. Once we have computed all S values, the traceback information is used to construct the maximum-sum segment by starting from the largest S entry and following the arrows until a "↑" is reached. For example, in Figure 2.8, $A = \langle 3, 2, -6, 5, 2, -3, 6, -4, 2\rangle$. By computing from $i = 1$ to $i = n$, we have $S = \langle 3, 5, -1, 5, 7, 4, 10, 6, 8\rangle$. The maximum S entry is $S[7]$ whose value is 10. By backtracking from $S[7]$, we conclude that the maximum-sum segment of A is $\langle 5, 2, -3, 6\rangle$, whose sum is 10.

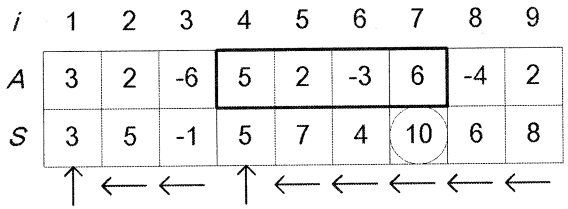

Fig. 2.8 Finding a maximum-sum segment.

Let *prefix sum* $P[i] = \sum_{j=1}^{i} a_j$ be the sum of the first i elements. It can be easily seen that $\sum_{k=i}^{j} a_k = P[j] - P[i-1]$. Therefore, if we wish to compute for a given position the maximum-sum segment ending at it, we could just look for a minimum prefix sum ahead of this position. This yields another linear-time algorithm for the maximum-sum segment problem.

2.4.3 Longest Increasing Subsequences

Given a sequence of numbers $A = \langle a_1, a_2, \ldots, a_n\rangle$, the longest increasing subsequence problem is to find an increasing subsequence in A whose length is maximum. Without loss of generality, we assume that these numbers are distinct. Formally speaking, given a sequence of distinct real numbers $A = \langle a_1, a_2, \ldots, a_n\rangle$, sequence $B = \langle b_1, b_2, \ldots, b_k\rangle$ is said to be a subsequence of A if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k\rangle$ of indices of A such that for all $j = 1, 2, \ldots, k$, we have $a_{i_j} = b_j$. In other words, B is obtained by deleting zero or more elements from A. We say that the subsequence B is increasing if $b_1 < b_2 < \ldots < b_k$. The longest increasing subsequence problem is to find a maximum-length increasing subsequence of A.

For example, suppose $A = \langle 4, 8, 2, 7, 3, 6, 9, 1, 10, 5\rangle$, both $\langle 2,3,6\rangle$ and $\langle 2,7,9,10\rangle$ are increasing subsequences of A, whereas $\langle 8,7,9\rangle$ (not increasing) and $\langle 2,3,5,7\rangle$ (not a subsequence) are not.

Note that we may have more than one longest increasing subsequence, so we use "*a* longest increasing subsequence" instead of "*the* longest increasing subsequence." Let $L[i]$ be the length of a longest increasing subsequence ending at position i. They can be computed by the following recurrence:

$$L[i] = \begin{cases} 1 + \max_{j=0,\ldots,i-1}\{L[j] \mid a_j < a_i\} & \text{if } i > 0, \\ 0 & \text{if } i = 0. \end{cases}$$

Here we assume that a_0 is a dummy element and smaller than any element in A, and $L[0]$ is equal to 0. By tabular computation for every i from 1 to n, each $L[i]$ can be computed in $O(i)$ steps. Therefore, they require in total $\sum_{i=1}^{n} O(i) = O(n^2)$ steps. For each position i, we use an array P to record the index of the best previous element for the current element to concatenate with. By tracing back from the element with the largest L value, we derive a longest increasing subsequence.

Figure 2.9 illustrates the process of finding a longest increasing subsequence of $A = \langle 4, 8, 2, 7, 3, 6, 9, 1, 10, 5\rangle$. Take $i = 4$ for instance, where $a_4 = 7$. Its previous smaller elements are a_1 and a_3, both with L value equaling 1. Therefore, we have $L[4] = L[1] + 1 = 2$, meaning that the length of a longest increasing subsequence ending at position 4 is of length 2. Indeed, both $\langle a_1, a_4\rangle$ and $\langle a_3, a_4\rangle$ are an increasing subsequence ending at position 4. In order to trace back the solution, we use array P to record which entry contributes the maximum to the current L value. Thus, $P[4]$ can be 1 (standing for a_1) or 3 (standing for a_3). Once we have computed all L and

P values, the maximum L value is the length of a longest increasing subsequence of A. In this example, $L[9] = 5$ is the maximum. Tracing back from $P[9]$, we have found a longest increasing subsequence $\langle a_3, a_5, a_6, a_7, a_9 \rangle$, *i.e.*, $\langle 2, 3, 6, 9, 10 \rangle$.

In the following, we briefly describe a more efficient dynamic-programming algorithm for delivering a longest increasing subsequence. A crucial observation is that it suffices to store only those smallest ending elements for all possible lengths of the increasing subsequences. For example, in Figure 2.9, there are three entries whose L value is 2, namely $a_2 = 8$, $a_4 = 7$, and $a_5 = 3$, where a_5 is the smallest. Any element after position 5 that is larger than a_2 or a_4 is also larger than a_5. Therefore, a_5 can replace the roles of a_2 and a_4 after position 5.

Let $SmallestEnd[k]$ denote the smallest ending element of all possible increasing subsequences of length k ending before the current position i. The algorithm proceeds for i from 1 to n. How do we update $SmallestEnd[k]$ when we consider a_i? By definition, it is easy to see that the elements in $SmallestEnd$ are in increasing order. In fact, a_i will affect only one entry in $SmallestEnd$. If a_i is larger than all the elements in $SmallestEnd$, then we can concatenate a_i to the longest increasing subsequence computed so far. That is, one more entry is added to the end of $SmallestEnd$. A backtracking pointer is recorded by pointing to the previous last element of $SmallestEnd$. Otherwise, let $SmallestEnd[k']$ be the smallest element that is larger than a_i. We replace $SmallestEnd[k']$ by a_i because now we have a smaller ending element of an increasing subsequence of length k'.

Since $SmallestEnd$ is a sorted array, the above process can be done by a binary search. A binary search algorithm compares the query element with the middle element of the sorted array, if the query element is larger, then it searches the larger half recursively. Otherwise, it searches the smaller half recursively. Either way the size of the search space is shrunk by a factor of two. At position i, the size of $SmallestEnd$ is at most i. Therefore, for each position i, it takes $O(\log i)$ time to determine the appropriate entry to be updated by a_i. Therefore, in total we have an $O(n \log n)$-time algorithm for the longest increasing subsequence problem.

Figure 2.10 illustrates the process of finding a longest increasing subsequence of $A = \langle 4, 8, 2, 7, 3, 6, 9, 1, 10, 5 \rangle$. When $i = 1$, there is only one increasing subsequence, *i.e.*, $\langle 4 \rangle$. We have $SmallestEnd[1] = 4$. Since $a_2 = 8$ is larger than $SmallestEnd[1]$, we create a new entry $SmallestEnd[2] = 8$ and set the backtracking

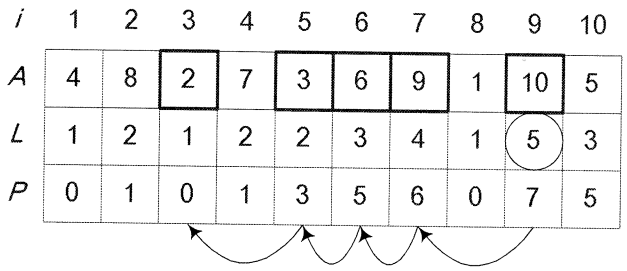

Fig. 2.9 An $O(n^2)$-time algorithm for finding a longest increasing subsequence.

pointer $P[2] = 1$, meaning that a_2 can be concatenated with a_1 to form an increasing subsequence $\langle 4, 8 \rangle$. When $a_3 = 2$ is encountered, its nearest larger element in $SmallestEnd$ is $SmallestEnd[1] = 4$. We know that we now have an increasing subsequence $\langle 2 \rangle$ of length 1. So $SmallestEnd[1]$ is changed from 4 to $a_3 = 2$ and $P[3] = 0$. When $i = 4$, we have $SmallestEnd[1] = 2 < a_4 = 7 < SmallestEnd[2] = 8$. By concatenating a_4 with $Smallest[1]$, we have a new increasing subsequence $\langle 2, 7 \rangle$ of length 2 whose ending element is smaller than 8. Thus, $SmallestEnd[2]$ is changed from 8 to $a_4 = 7$ and $P[4] = 3$. Continue this way until we reach a_{10}. When a_{10} is encountered, we have $SmallestEnd[2] = 3 < a_{10} = 5 < SmallestEnd[3] = 6$. We set $SmallestEnd[3] = a_{10} = 5$ and $P[10] = 5$. Now the largest element in $SmallestEnd$ is $SmallestEnd[5] = a_9 = 10$. We can trace back from a_9 by the backtracking pointers P and deliver a longest increasing subsequence $\langle a_3, a_5, a_6, a_7, a_9 \rangle$, *i.e.*, $\langle 2, 3, 6, 9, 10 \rangle$.

2.4.4 Longest Common Subsequences

A subsequence of a sequence S is obtained by deleting zero or more elements from S. For example, $\langle P, R, E, D \rangle$, $\langle S, D, N \rangle$, and $\langle P, R, E, D, E, N, T \rangle$ are all subsequences of $\langle P, R, E, S, I, D, E, N, T \rangle$, whereas $\langle S, N, D \rangle$ and $\langle P, E, F \rangle$ are not.

Recall that, given two sequences, the longest common subsequence (LCS) problem is to find a subsequence that is common to both sequences and its length is maximized. For example, given two sequences

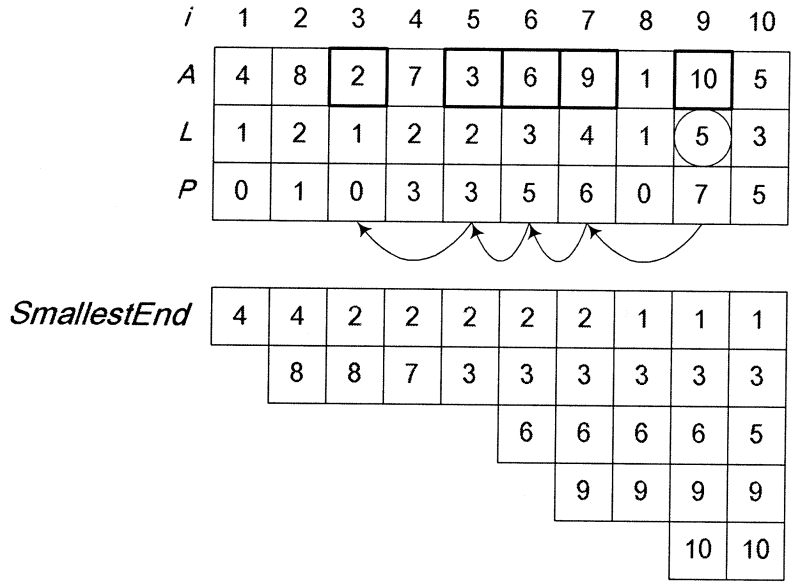

Fig. 2.10 An $O(n \log n)$-time algorithm for finding a longest increasing subsequence.

$$\langle P,R,E,S,I,D,E,N,T\rangle$$

and

$$\langle P,R,O,V,I,D,E,N,C,E\rangle,$$

$\langle P,R,D,N\rangle$ is a common subsequence of them, whereas $\langle P,R,V\rangle$ is not. Their LCS is $\langle P,R,I,D,E,N\rangle$.

Now let us formulate the recurrence for computing the length of an LCS of two sequences. We are given two sequences $A=\langle a_1,a_2,\ldots,a_m\rangle$, and $B=\langle b_1,b_2,\ldots,b_n\rangle$. Let $len[i,j]$ denote the length of an LCS between $\langle a_1,a_2,\ldots,a_i\rangle$ (a prefix of A) and $\langle b_1,b_2,\ldots,b_j\rangle$ (a prefix of B). They can be computed by the following recurrence:

$$len[i,j]=\begin{cases}0 & \text{if } i=0 \text{ or } j=0,\\ len[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } a_i=b_j,\\ \max\{len[i,j-1],len[i-1,j]\} & \text{otherwise.}\end{cases}$$

In other words, if one of the sequences is empty, the length of their LCS is just zero. If a_i and b_j are the same, an LCS between $\langle a_1,a_2,\ldots,a_i\rangle$, and $\langle b_1,b_2,\ldots,b_j\rangle$ is the concatenation of an LCS of $\langle a_1,a_2,\ldots,a_{i-1}\rangle$ and $\langle b_1,b_2,\ldots,b_{j-1}\rangle$ and a_i. Therefore, $len[i,j]=len[i-1,j-1]+1$ in this case. If a_i and b_j are different, their LCS is equal to either an LCS of $\langle a_1,a_2,\ldots,a_i\rangle$, and $\langle b_1,b_2,\ldots,b_{j-1}\rangle$, or that of $\langle a_1,a_2,\ldots,a_{i-1}\rangle$, and $\langle b_1,b_2,\ldots,b_j\rangle$. Its length is thus the maximum of $len[i,j-1]$ and $len[i-1,j]$.

Figure 2.11 gives the pseudo-code for computing $len[i,j]$. For each entry (i,j), we retain the backtracking information in $prev[i,j]$. If $len[i-1,j-1]$ contributes the maximum value to $len[i,j]$, then we set $prev[i,j]=$"$\nwarrow$." Otherwise $prev[i,j]$ is set to be "$\uparrow$" or "$\leftarrow$" depending on which one of $len[i-1,j]$ and $len[i,j-1]$ contributes the maximum value to $len[i,j]$. Whenever there is a tie, any one of them will

Algorithm LCS_LENGTH($A=\langle a_1,a_2,\ldots,a_m\rangle$, $B=\langle b_1,b_2,\ldots,b_n\rangle$)
begin
  for $i\leftarrow 0$ **to** m **do** $len[i,0]\leftarrow 0$
  for $j\leftarrow 1$ **to** n **do** $len[0,j]\leftarrow 0$
  for $i\leftarrow 1$ **to** m **do**
    for $j\leftarrow 1$ **to** n **do**
      if $a_i=b_j$ **then**
        $len[i,j]\leftarrow len[i-1,j-1]+1$
        $prev[i,j]\leftarrow$"$\nwarrow$"
      else if $len[i-1,j]\geq len[i,j-1]$ **then**
        $len[i,j]\leftarrow len[i-1,j]$
        $prev[i,j]\leftarrow$"$\uparrow$"
      else
        $len[i,j]\leftarrow len[i,j-1]$
        $prev[i,j]\leftarrow$"$\leftarrow$"
  return *len* and *prev*
end

Fig. 2.11 Computation of the length of an LCS of two sequences.

		A	L	I	G	N	M	E	N	T
	0	0	0	0	0	0	0	0	0	0
A	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1
L	0	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\leftarrow$ 2	$\leftarrow$ 2	$\leftarrow$ 2	$\leftarrow$ 2	$\leftarrow$ 2	$\leftarrow$ 2
G	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3	$\leftarrow$ 3	$\leftarrow$ 3	$\leftarrow$ 3	$\leftarrow$ 3
O	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3
R	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3
I	0	$\uparrow$ 1	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3
T	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
H	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 4
M	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3	$\uparrow$ 3	$\nwarrow$ 4	$\leftarrow$ 4	$\leftarrow$ 4	$\uparrow$ 4

Fig. 2.12 Tabular computation of the length of an LCS of $\langle A,L,G,O,R,I,T,H,M\rangle$ and $\langle A,L,I,G,N,M,E,N,T\rangle$.

work. These arrows will guide the backtracking process upon reaching the terminal entry (m,n). Since the time spent for each entry is $O(1)$, the total running time of algorithm LCS_LENGTH is $O(mn)$.

Figure 2.12 illustrates the tabular computation. The length of an LCS of

$$\langle A,L,G,O,R,I,T,H,M\rangle$$

and

$$\langle A,L,I,G,N,M,E,N,T\rangle$$

is 4.

Besides computing the length of an LCS of the whole sequences, Figure 2.12 in fact computes the length of an LCS between each pair of prefixes of the two sequences. For example, by this table, we can also tell the length of an LCS between $\langle A,L,G,O,R\rangle$ and $\langle A,L,I,G\rangle$ is 3.

Algorithm LCS_OUTPUT($A=\langle a_1,a_2,\ldots,a_m\rangle$, *prev*, *i*, *j*)
begin
  if $i=0$ or $j=0$ **then return**
  if $prev[i,j]=$"$\nwarrow$" **then**
    LCS_OUTPUT(*A*, *prev*, $i-1$, $j-1$)
    print a_i
  else if $prev[i,j]=$"$\uparrow$" **then** LCS_OUTPUT(*A*, *prev*, $i-1$, *j*)
  else LCS_OUTPUT(*A*, *prev*, *i*, $j-1$)
end

Fig. 2.13 Delivering an LCS.

Once algorithm LCS_LENGTH reaches (m,n), the backtracking information retained in array *prev* allows us to find out which common subsequence contributes $len[m,n]$, the maximum length of an LCS of sequences A and B. Figure 2.13 lists the pseudo-code for delivering an LCS. We trace back the dynamic-programming matrix from the entry (m,n) recursively following the direction of the arrow. Whenever a diagonal arrow "↖" is encountered, we append the current matched letter to the end of the LCS under construction. Algorithm LCS_OUTPUT takes $O(m+n)$ time in total since each recursive call reduces the indices i and/or j by one.

Figure 2.14 backtracks the dynamic-programming matrix computed in Figure 2.12. It outputs $\langle A,L,G,T\rangle$ (the shaded entries) as an LCS of

$$\langle A,L,G,O,R,I,T,H,M\rangle$$

and

$$\langle A,L,I,G,N,M,E,N,T\rangle.$$

		A	L	I	G	N	M	E	N	T
	0	0	0	0	0	0	0	0	0	0
A	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
L	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
G	0	↑ 1	↑ 2	↑ 2	↖ 3	← 3	← 3	← 3	← 3	← 3
O	0	↑ 1	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3
R	0	↑ 1	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3
I	0	↑ 1	↑ 2	↖ 3	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3
T	0	↑ 1	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3	↖ 4
H	0	↑ 1	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3	↑ 4
M	0	↑ 1	↑ 2	↑ 3	↑ 3	↑ 3	↖ 4	← 4	← 4	↑ 4

Fig. 2.14 Backtracking process for finding an LCS of $\langle A,L,G,O,R,I,T,H,M\rangle$ and $\langle A,L,I,G,N,M,E,N,T\rangle$.

2.5 Bibliographic Notes and Further Reading

This chapter presents three basic algorithmic techniques that are often used in designing efficient methods for various problems in sequence comparison. Readers can refer to algorithm textbooks for more instructive tutorials. The algorithm book (or "The White Book") by Cormen et al. [52] is a comprehensive reference of data structures and algorithms with a solid mathematical and theoretical foundation. Manber's

book [133] provides a creative approach for the design and analysis of algorithms. The book by Baase and Gelder [17] is a good algorithm textbook for beginners.

2.1

As noted by Donald E. Knuth [113], the invention of the O-notation originated from a number-theory book by P. Bachmann in 1892.

2.2

David A. Huffman [94] invented Huffman coding while he was a Ph.D. student at MIT. It was actually a term paper for the problem of finding the most efficient binary coding scheme assigned by Robert M. Fano.

2.3

There are numerous sorting algorithms such as insertion sort, bubblesort, quicksort, mergesort, to name a few. As noted by Donald E. Knuth [114], the first program ever written for a stored program computer was the mergesort program written by John von Neumann in 1945.

2.4

The name "dynamic programming" was given by Richard Bellman in 1957 [25]. The maximum-sum segment problem was first surveyed by Bentley and is linear-time solvable using Kadane's algorithm [27].

Chapter 3
Pairwise Sequence Alignment

Pairwise alignment is often used to reveal similarities between sequences, determine the residue-residue correspondences, locate patterns of conservation, study gene regulation, and infer evolutionary relationships.

This chapter is divided into eight sections. It starts with a brief introduction in Section 3.1, followed by the dot matrix representation of pairwise sequence comparison in Section 3.2.

Using alignment graph, we derive a dynamic-programming method for aligning globally two sequences in Section 3.3. An example is used to illustrate the tabular computation for computing the optimal alignment score as well as the backtracking procedure for delivering an optimal global alignment.

Section 3.4 describes a method for delivering an optimal local alignment, which involves only a segment of each sequence. The recurrence for an optimal local alignment is quite similar to that for global alignment. To reflect the flexibility that an alignment can start at any position of the two sequences, an additional entry "zero" is added.

In Section 3.5, we address some flexible strategies for coping with various scoring schemes. Affine gap penalties are considered more appropriate for aligning DNA and protein sequences. To favor longer gaps, constant gap penalties or restricted affine gap penalties could be the choice.

Straightforward implementations of the dynamic-programming algorithms consume quadratic space for alignment. For certain applications, such as careful analysis of a few long DNA sequences, the space restriction is more important than the time constraint. Section 3.6 introduces Hirschberg's linear-space approach.

Section 3.7 discusses several advanced topics such as constrained sequence alignment, similar sequence alignment, suboptimal alignment, and robustness measurement.

Finally, we conclude the chapter with the bibliographic notes in Section 3.8.

3.1 Introduction

In nature, even a single amino acid sequence contains all the information necessary to determine the fold of the protein. However, the folding process is still mysterious to us, and some valuable information can be revealed by sequence comparison. Take a look at the following sequence:

```
THETR UTHIS MOREI MPORT ANTTH ANTHE FACTS
```

What did you see in the above sequence? By comparing it with the words in the dictionary, we find the tokens "FACTS," "IMPORTANT," "IS," "MORE," "THAN," "THE," and "TRUTH." Then we figure out the above is the sentence "The truth is more important than the facts."

Even though we have not yet decoded the DNA and protein languages, the emerging flood of sequence data has provided us with a golden opportunity of investigating the evolution and function of biomolecular sequences. We are in a stage of compiling dictionaries for DNA, proteins, and so forth. Sequence comparison plays a major role in this line of research and thus becomes the most basic tool of bioinformatics.

Sequence comparison has wide applications to molecular biology, computer science, speech processing, and so on. In molecular biology, it is often used to reveal similarities among sequences, determine the residue-residue correspondences, locate patterns of conservation, study gene regulation, and infer evolutionary relationships. It helps us to fish for related sequences in databanks, such as the GenBank database. It can also be used for the annotation of genomes.

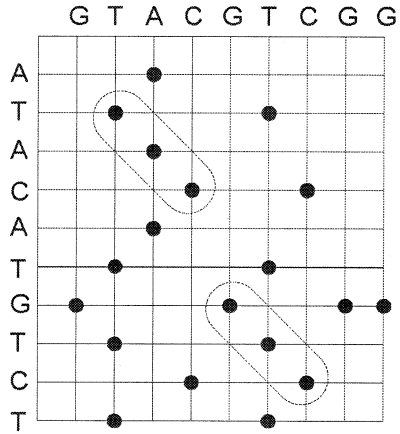

Fig. 3.1 A dot matrix of the two sequences ATACATGTCT and GTACGTCGG.

3.2 Dot Matrix

A dot matrix is a two-dimensional array of dots used to highlight the exact matches between two sequences. Given are two sequences $A = \langle a_1, a_2, \ldots, a_m \rangle$ (or $A = a_1 a_2 \ldots a_m$ in short), and $B = \langle b_1, b_2, \ldots, b_n \rangle$. A dot is plotted on the (i, j) entry of the matrix if $a_i = b_j$. Users can easily identify similar regions between the two sequences by locating those contiguous dots along the same diagonal. Figure 3.1 gives a dot matrix of the two sequences ATACATGTCT and GTACGTCGG. Dashed lines circle those regions with at lease three contiguous matches on the same diagonal.

A dot matrix allows the users to quickly visualize the similar regions of two sequences. However, as the sequences get longer, it becomes more involved to determine their most similar regions, which can no longer be answered by merely looking at a dot matrix. It would be more desirable to automatically identify those similar regions and rank them by their "similarity scores." This leads to the development of sequence alignment.

3.3 Global Alignment

Following its introduction by Needleman and Wunsch in 1970, dynamic programming has become a major algorithmic strategy for many optimization problems in sequence comparison. This strategy is guaranteed to produce an alignment of two given sequences having the highest score for a number of useful alignment-scoring schemes.

Given two sequences $A = a_1 a_2 \ldots a_m$, and $B = b_1 b_2 \ldots b_n$, an *alignment* of A and B is obtained by introducing dashes into the two sequences such that the lengths of the two resulting sequences are identical and no column contains two dashes. Let Σ denote the alphabet over which A and B are defined. To simplify the presentation, we employ a very simple scoring scheme as follows. A score $\sigma(a, b)$ is defined for each $(a, b) \in \Sigma \times \Sigma$. Each indel, *i.e.*, a column with a space, is penalized by a constant β. The score of an alignment is the sum of σ scores of all columns with no dashes minus the penalties of the gaps. An *optimal global alignment* is an alignment that

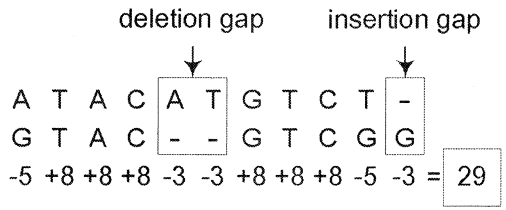

Fig. 3.2 An alignment of the two sequences ATACATGTCT and GTACGTCGG.

maximizes the score. By global alignment, we mean that both sequences are aligned globally, i.e., from their first symbols to their last.

Figure 3.2 gives an alignment of sequences ATACATGTCT and GTACGTCGG and its score. In this and the next sections, we assume the following simple scoring scheme. A match is given a bonus score 8, a mismatch is penalized by assigning score -5, and the gap penalty for each indel is -3. In other words, $\sigma(a,b) = 8$ if a and b are the same, $\sigma(a,b) = -5$ if a and b are different, and $\beta = -3$.

It is quite helpful to recast the problem of aligning two sequences as an equivalent problem of finding a maximum-scoring path in the alignment graph defined in Section 1.2.2, as has been observed by a number of researchers. Recall that the alignment graph of A and B is a directed acyclic graph whose vertices are the pairs (i,j) where $i \in \{0,1,2,\ldots,m\}$ and $j \in \{0,1,2,\ldots,n\}$. These vertices are arrayed in $m+1$ rows and $n+1$ columns. The edge set consists of three types of edges. The substitution aligned pairs, insertion aligned pairs, and deletion aligned pairs correspond to the diagonal edges, horizontal edges, and vertical edges, respectively. Specifically, a vertical edge from $(i-1,j)$ to (i,j), which corresponds to a deletion of a_i, is drawn for $i \in \{1,2,\ldots,m\}$ and $j \in \{0,1,2,\ldots,n\}$. A horizontal edge from $(i,j-1)$ to (i,j) , which corresponds to an insertion of b_j, is drawn for $i \in \{0,1,2,\ldots,m\}$ and $j \in \{1,2,\ldots,n\}$. A diagonal edge from $(i-1,j-1)$ to (i,j) , which corresponds to a substitution of a_i with b_j, is drawn for $i \in \{1,2,\ldots,m\}$ and $j \in \{1,2,\ldots,n\}$.

It has been shown that an alignment corresponds to a path from the leftmost cell of the first row to the rightmost cell of the last row in the alignment graph. Figure 3.3 gives another example of this correspondence.

Let $S[i,j]$ denote the score of an optimal alignment between $a_1a_2\ldots a_i$, and $b_1b_2\ldots b_j$. By definition, we have $S[0,0] = 0$, $S[i,0] = -\beta \times i$, and $S[0,j] = -\beta \times j$. With these initializations, $S[i,j]$ for $i \in \{1,2,\ldots,m\}$ and $j \in \{1,2,\ldots,n\}$ can be computed by the following recurrence.

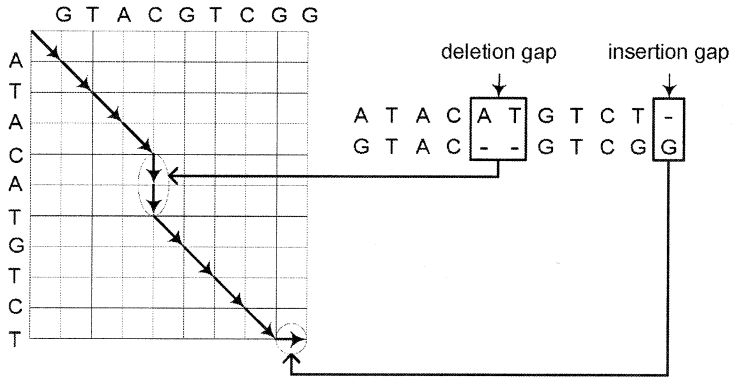

Fig. 3.3 A path in an alignment graph of the two sequences ATACATGTCT and GTACGTCGG.

$$S[i,j] = \max \begin{cases} S[i-1,j] - \beta, \\ S[i,j-1] - \beta, \\ S[i-1,j-1] + \sigma(a_i, b_j). \end{cases}$$

Figure 3.4 explains the recurrence by showing that there are three possible ways entering into the grid point (i,j), and we take the maximum of their path weights. The weight of the maximum-scoring path entering (i,j) from $(i-1,j)$ *vertically* is the weight of the maximum-scoring path entering $(i-1,j)$ plus the weight of edge $(i-1,j) \to (i,j)$. That is, the weight of the maximum-scoring path entering (i,j) with a deletion gap symbol at the end is $S[i-1,j] - \beta$. Similarly, the weight of the maximum-scoring path entering (i,j) from $(i,j-1)$ *horizontally* is $S[i,j-1] - \beta$ and the weight of the maximum-scoring path entering (i,j) from $(i-1,j-1)$ *diagonally* is $S[i-1,j-1] + \sigma(a_i,b_j)$. To compute $S[i,j]$, we simply take the maximum value of these three choices. The value $S[m,n]$ is the score of an optimal global alignment between sequences A and B.

Figure 3.5 gives the pseudo-code for computing the score of an optimal global alignment. Whenever there is a tie, any one of them will work. Since there are $O(mn)$ entries and the time spent for each entry is $O(1)$, the total running time of algorithm GLOBAL_ALIGNMENT_SCORE is $O(mn)$.

Now let us use an example to illustrate the tabular computation. Figure 3.6 computes the score of an optimal alignment of the two sequences ATACATGTCT and GTACGTCGG, where a match is given a bonus score 8, a mismatch is penalized by a score -5, and the gap penalty for each gap symbol is -3. The first row and column of the table are initialized with proper penalties. Other entries are computed in order. Take the entry $(5,3)$ for example. Upon computing the value of this entry, the following values are ready: $S[4,2] = -3$, $S[4,3] = 8$, and $S[5,2] = -6$. Since the edge weight of $(4,2) \to (5,3)$ is 8 (a match symbol "A"), the maximum score from $(4,2)$ to $(5,3)$ is $-3+8 = 5$. The maximum score from $(4,3)$ is $8-3 = 5$, and the

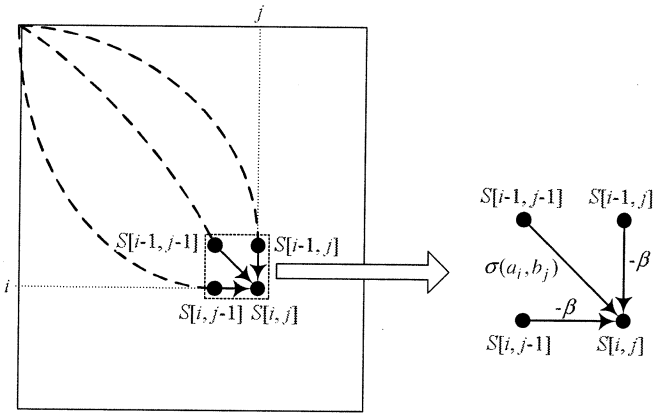

Fig. 3.4 There are three ways entering the grid point (i,j).

Algorithm GLOBAL_ALIGNMENT_SCORE($A = a_1a_2\ldots a_m$, $B = b_1b_2\ldots b_n$)
begin
  $S[0,0] \leftarrow 0$
  for $j \leftarrow 1$ **to** n **do** $S[0,j] \leftarrow -\beta \times j$
  for $i \leftarrow 1$ **to** m **do**
    $S[i,0] \leftarrow -\beta \times i$
    for $j \leftarrow 1$ **to** n **do**

$$S[i,j] \leftarrow \max \begin{cases} S[i-1,j]-\beta \\ S[i,j-1]-\beta \\ S[i-1,j-1]+\sigma(a_i,b_j) \end{cases}$$

  Output $S[m,n]$ as the score of an optimal alignment.
end

Fig. 3.5 Computation of the score of an optimal global alignment.

maximum score from $(5,2)$ is $-6-3=-9$. Taking the maximum of them, we have $S[5,3]=5$. Once the table has been computed, the value in the rightmost cell of the last row, *i.e.*, $S[10,9]=29$, is the score of an optimal global alignment.

In Section 2.4.4, we have shown that if a backtracking information is saved for each entry while we compute the dynamic-programming matrix, an optimal solution can be derived following the backtracking pointers. Here we show that even if we don't save those backtracking pointers, we can still reconstruct an optimal so-

		G	T	A	C	G	T	C	G	G
	0	-3	-6	-9	-12	-15	-18	-21	-24	-27
A	-3	-5	-8	2	-1	-4	-7	-10	-13	-16
T	-6	-8	3	0	-3	-6	4	1	-2	-5
A	-9	-11	0	11	8	5	2	-1	-4	-7
C	-12	-14	-3	8	19	16	13	10	7	4
A	-15	-17	-6	5	16	14	11	8	5	2
T	-18	-20	-9	2	13	11	22	19	16	13
G	-21	-10	-12	-1	10	21	19	17	27	24
T	-24	-13	-2	-4	7	18	29	26	24	22
C	-27	-16	-5	-7	4	15	26	37	34	31
T	-30	-19	-8	-10	1	12	23	34	32	29

Fig. 3.6 The score of an optimal global alignment of the two sequences ATACATGTCT and GTACGTCGG, where a match is given a bonus score 8, a mismatch is penalized by a score -5, and the gap penalty for each gap symbol is -3.

lution by examining the values of an entry's possible contributors. Figure 3.7 lists the pseudo-code for delivering an optimal global alignment, where an initial call GLOBAL_ALIGNMENT_OUTPUT(A, B, S, m, n) is made to deliver an optimal global alignment. Specifically, we trace back the dynamic-programming matrix from the entry (m,n) recursively according to the following rules. Let (i,j) be the entry under consideration. If $i=0$ or $j=0$, we simply output all the insertion pairs or deletion pairs in these boundary conditions. Otherwise, consider the following three cases. If $S[i,j]=S[i-1,j-1]+\sigma(a_i,b_j)$, we make a diagonal move and output a substitution pair $\begin{pmatrix} a_i \\ b_j \end{pmatrix}$. If $S[i,j]=S[i-1,j]-\beta$, then we make a vertical move and output a deletion pair $\begin{pmatrix} a_i \\ - \end{pmatrix}$. Otherwise, it must be the case where $S[i,j]=S[i,j-1]-\beta$. We simply make a horizontal move and output an insertion pair $\begin{pmatrix} - \\ b_j \end{pmatrix}$. Algorithm GLOBAL_ALIGNMENT_OUTPUT takes $O(m+n)$ time in total since each recursive call reduces i and/or j by one. The total space complexity is $O(mn)$ since the size of the dynamic-programming matrix is $O(mn)$. In Section 3.6, we shall show that an optimal global alignment can be recovered even if we don't save the whole matrix.

Figure 3.8 delivers an optimal global alignment by backtracking from the rightmost cell of the last row of the dynamic-programming matrix computed in Figure 3.6. We start from the entry $(10,9)$ where $S[10,9]=29$. We have a tie there because both $S[10,8]-3$ and $S[9,8]-5$ equal to 29. In this illustration, the horizontal move to the entry $(10,8)$ is chosen. Interested readers are encouraged to try the

Algorithm GLOBAL_ALIGNMENT_OUTPUT($A = a_1a_2\ldots a_m$, $B = b_1b_2\ldots b_n$, S, i, j)
begin
  if $i=0$ or $j=0$ **then**
    if $i>0$ **then for** $i' \leftarrow 1$ **to** i **do print** $\begin{pmatrix} a_{i'} \\ - \end{pmatrix}$
    if $j>0$ **then for** $j' \leftarrow 1$ **to** j **do print** $\begin{pmatrix} - \\ b_{j'} \end{pmatrix}$
    return
  if $S[i,j]=S[i-1,j-1]+\sigma(a_i,b_j)$ **then**
    GLOBAL_ALIGNMENT_OUTPUT(A, B, S, $i-1$, $j-1$)
    print $\begin{pmatrix} a_i \\ b_j \end{pmatrix}$
  else if $S[i,j]=S[i-1,j]-\beta$ **then**
    GLOBAL_ALIGNMENT_OUTPUT(A, B, S, $i-1$, j)
    print $\begin{pmatrix} a_i \\ - \end{pmatrix}$
  else
    GLOBAL_ALIGNMENT_OUTPUT(A, B, S, i, $j-1$)
    print $\begin{pmatrix} - \\ b_j \end{pmatrix}$
end

Fig. 3.7 Backtracking procedure for delivering an optimal global alignment.

		G	T	A	C	G	T	C	G	G
	0	-3	-6	-9	-12	-15	-18	-21	-24	-27
A	-3	-5	-8	2	-1	-4	-7	-10	-13	-16
T	-6	-8	3	0	-3	-6	4	1	-2	-5
A	-9	-11	0	11	8	5	2	-1	-4	-7
C	-12	-14	-3	8	19	16	13	10	7	4
A	-15	-17	-6	5	16	14	11	8	5	2
T	-18	-20	-9	2	13	11	22	19	16	13
G	-21	-10	-12	-1	10	21	19	17	27	24
T	-24	-13	-2	-4	7	18	29	26	24	22
C	-27	-16	-5	-7	4	15	26	37	34	31
T	-30	-19	-8	-10	1	12	23	34	32	29

```
A T A C A T G T C T -
G T A C - - G T C G G
-5 +8 +8 +8 -3 -3 +8 +8 +8 -5 -3 = 29
```

Fig. 3.8 Computation of an optimal global alignment of sequences ATACATGTCT and GTACGTCGG, where a match is given a bonus score 8, a mismatch is penalized by a score -5, and the gap penalty for each gap symbol is -3.

diagonal move to the entry $(9,8)$ for an alternative optimal global alignment, which is actually chosen by GLOBAL_ALIGNMENT_OUTPUT. Continue this process until the entry $(0,0)$ is reached. The shaded area depicts the backtracking path whose corresponding alignment is given on the right-hand side of the figure.

It should be noted that during the backtracking procedure, we derive the aligned pairs in a reverse order of the alignment. That's why we make a recursive call before actually printing out the pair in Figure 3.7. Another approach is to compute the dynamic-programming matrix backward from the rightmost cell of the last row to the leftmost cell of the first row. Then when we trace back from the leftmost cell of the first row toward the rightmost cell of the last row, the aligned pairs are derived in the same order as in the alignment. This approach could avoid the overhead of reversing an alignment.

3.4 Local Alignment

In many applications, a global (i.e., end-to-end) alignment of the two given sequences is inappropriate; instead, a local alignment (i.e., involving only a part of each sequence) is desired. In other words, one seeks a high-scoring local path that need not terminate at the corners of the dynamic-programming matrix.

Let $S[i,j]$ denote the score of the highest-scoring local path ending at (i,j) between $a_1a_2\ldots a_i$, and $b_1b_2\ldots b_j$. $S[i,j]$ can be computed as follows.

$$S[i,j] = \max \begin{cases} 0, \\ S[i-1,j]-\beta, \\ S[i,j-1]-\beta, \\ S[i-1,j-1]+\sigma(a_i,b_j). \end{cases}$$

The recurrence is quite similar to that for global alignment except the first entry "zero." For local alignment, we are not required to start from the source $(0,0)$. Therefore, if the scores of all possible paths ending at the current position are all negative, they are reset to zero. The largest value of $S[i,j]$ is the score of the best local alignment between sequences A and B.

Figure 3.9 gives the pseudo-code for computing the score of an optimal local alignment. Whenever there is a tie, any one of them will work. Since there are $O(mn)$ entries and the time spent for each entry is $O(1)$, the total running time of algorithm LOCAL_ALIGNMENT_SCORE is $O(mn)$.

Now let us use an example to illustrate the tabular computation. Figure 3.10 computes the score of an optimal local alignment of the two sequences ATACATGTCT and GTACGTCGG, where a match is given a bonus score 8, a mismatch is penalized by a score -5, and the gap penalty for each gap symbol is -3. The first row and column of the table are initialized with zero's. Other entries are computed in order. Take the entry $(5,5)$ for example. Upon computing the value of this entry, the following values are ready: $S[4,4] = 24$, $S[4,5] = 21$, and $S[5,4] = 21$. Since the edge weight of $(4,4) \rightarrow (5,5)$ is -5 (a mismatch), the maximum score from $(4,4)$ to $(5,5)$ is $24-5 = 19$. The maximum score from $(4,5)$ is $21-3 = 18$, and the maximum score from $(5,4)$ is $21-3 = 18$. Taking the maximum of them, we have $S[5,5] = 19$. Once

Algorithm LOCAL_ALIGNMENT_SCORE($A = a_1a_2\ldots a_m$, $B = b_1b_2\ldots b_n$)
begin
  $S[0,0] \leftarrow 0$
  $Best \leftarrow 0$
  $End_i \leftarrow 0$
  $End_j \leftarrow 0$
  for $j \leftarrow 1$ **to** n **do** $S[0,j] \leftarrow 0$
  for $i \leftarrow 1$ **to** m **do**
    $S[i,0] \leftarrow 0$
    for $j \leftarrow 1$ **to** n **do**

$$S[i,j] \leftarrow \max \begin{cases} 0 \\ S[i-1,j]-\beta \\ S[i,j-1]-\beta \\ S[i-1,j-1]+\sigma(a_i,b_j) \end{cases}$$

      if $S[i,j] > Best$ **then**
        $Best \leftarrow S[i,j]$
        $End_i \leftarrow i$
        $End_j \leftarrow j$
  Output $Best$ as the score of an optimal local alignment.
end

Fig. 3.9 Computation of the score of an optimal local alignment.

		G	T	A	C	G	T	C	G	G
	0	0	0	0	0	0	0	0	0	0
A	0	0	0	8	5	2	0	0	0	0
T	0	0	8	5	3	0	10	7	4	1
A	0	0	5	16	13	10	7	5	2	0
C	0	0	2	13	24	21	18	15	12	9
A	0	0	0	10	21	19	16	13	10	7
T	0	0	8	7	18	16	27	24	21	18
G	0	8	5	3	15	26	24	22	32	29
T	0	5	16	13	12	23	34	31	29	27
C	0	2	13	11	21	20	31	42	39	36
T	0	0	10	8	18	17	28	39	37	34

Fig. 3.10 Computation of the score of an optimal local alignment of the sequences ATACATGTCT and GTACGTCGG.

the table has been computed, the maximum value, *i.e.*, $S[9,7] = 42$, is the score of an optimal local alignment.

Figure 3.11 lists the pseudo-code for delivering an optimal local alignment, where an initial call LOCAL_ALIGNMENT_OUTPUT(A, B, S, End_i, End_j) is made to deliver an optimal local alignment. Specifically, we trace back the dynamic-programming matrix from the maximum-score entry (End_i, End_j) recursively according to the following rules. Let (i, j) be the entry under consideration. If $S[i, j] = 0$, we have reached the beginning of the optimal local alignment. Otherwise, consider the following three cases. If $S[i, j] = S[i-1, j-1] + \sigma(a_i, b_j)$, we make a diagonal move and output a substitution pair $\begin{pmatrix} a_i \\ b_j \end{pmatrix}$. If $S[i, j] = S[i-1, j] - \beta$, then we make a vertical move and output a deletion pair $\begin{pmatrix} a_i \\ - \end{pmatrix}$. Otherwise, it must be the case where $S[i, j] = S[i, j-1] - \beta$. We simply make a horizontal move and output an insertion pair $\begin{pmatrix} - \\ b_j \end{pmatrix}$. Algorithm LOCAL_ALIGNMENT_OUTPUT takes $O(m+n)$ time in total since each recursive call reduces i and/or j by one. The space complexity is $O(mn)$ since the size of the dynamic-programming matrix is $O(mn)$. In Section 3.6, we shall show that an optimal local alignment can be recovered even if we don't save the whole matrix.

Figure 3.12 delivers an optimal local alignment by backtracking from the maximum scoring entry of the dynamic-programming matrix computed in Figure 3.6.

Algorithm LOCAL_ALIGNMENT_OUTPUT($A = a_1 a_2 \ldots a_m$, $B = b_1 b_2 \ldots b_n$, S, i, j)
begin
  if $S[i, j] = 0$ **then return**
  if $S[i, j] = S[i-1, j-1] + \sigma(a_i, b_j)$ **then**
    LOCAL_ALIGNMENT_OUTPUT(A, B, S, $i-1$, $j-1$)
    print $\begin{pmatrix} a_i \\ b_j \end{pmatrix}$
  else if $S[i, j] = S[i-1, j] - \beta$ **then**
    LOCAL_ALIGNMENT_OUTPUT(A, B, S, $i-1$, j)
    print $\begin{pmatrix} a_i \\ - \end{pmatrix}$
  else
    LOCAL_ALIGNMENT_OUTPUT(A, B, S, i, $j-1$)
    print $\begin{pmatrix} - \\ b_j \end{pmatrix}$
end

Fig. 3.11 Computation of an optimal local alignment.

We start from the entry $(9,7)$ where $S[9,7] = 42$. Since $S[8,6] + 8 = 34 + 8 = 42 = S[9,7]$, we make a diagonal move back to the entry $(8,6)$. Continue this process until an entry with zero value is reached. The shaded area depicts the backtracking path whose corresponding alignment is given on the right-hand side of the figure.

Further complications arise when one seeks k best alignments, where $k > 1$. For computing an arbitrary number of non-intersecting and high-scoring local alignments, Waterman and Eggert [198] developed a very time-efficient method. It records those high-scoring candidate regions of the dynamic-programming matrix in the first pass. Each time a best alignment is reported, it recomputes only those entries in the affected area rather than recompute the whole matrix. Its linear-space implementation was developed by Huang and Miller [92].

On the other hand, to attain greater speed, the strategy of building alignments from alignment fragments is often used. For example, one could specify some fragment length w and work with fragments consisting of a segment of length at least w that occurs exactly or approximately in both sequences. In general, algorithms that optimize the score over alignments constructed from fragments can run faster than algorithms that optimize over all possible alignments. Moreover, alignments constructed from fragments have been very successful in initial filtering criteria within programs that search a sequence database for matches to a query sequence. Database sequences whose alignment score with the query sequence falls below a threshold are ignored, and the remaining sequences are subjected to a slower but higher-resolution alignment process. The high-resolution process can be made more efficient by restricting the search to a "neighborhood" of the alignment-from-fragments. Chapter 4 will introduce four such homology search programs: FASTA, BLAST, BLAT, and PatternHunter.

		G	T	A	C	G	T	C	G	G
	0	0	0	0	0	0	0	0	0	0
A	0	0	0	8	5	2	0	0	0	0
T	0	0	8	5	3	0	10	7	4	1
A	0	0	5	16	13	10	7	5	2	0
C	0	0	2	13	24	21	18	15	12	9
A	0	0	0	10	21	19	16	13	10	7
T	0	0	8	7	18	16	27	24	21	18
G	0	8	5	3	15	26	24	22	32	29
T	0	5	16	13	12	23	34	31	29	27
C	0	2	13	11	21	20	31	42	39	36
T	0	0	10	8	18	17	28	39	37	34

T A C A T G T C
T A C - - G T C
+8 +8 +8 -3 -3 +8 +8 +8 = 42

Fig. 3.12 Computation of an optimal local alignment of the two sequences ATACATGTCT and GTACGTCGG.

3.5 Various Scoring Schemes

In this section, we shall briefly discuss how to modify the dynamic programming methods to copy with three scoring schemes that are frequently used in biological sequence analysis.

3.5.1 Affine Gap Penalties

For aligning DNA and protein sequences, affine gap penalties are considered more appropriate than the simple scoring scheme discussed in the previous sections. "Affine" means that a gap of length k is penalized $\alpha + k \times \beta$, where α and β are both nonnegative constants. In other words, it costs α to open up a gap plus β for each symbol in the gap. Figure 3.13 computes the score of a global alignment of the two sequences ATACATGTCT and GTACGTCGG under affine gap penalties, where a match is given a bonus score 8, a mismatch is penalized by a score -5, and the penalty for a gap of length k is $-4-k\times 3$.

In order to determine if a gap is newly opened, two more matrices are used to distinguish gap extensions from gap openings. Let $D(i,j)$ denote the score of an optimal alignment between $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$ ending with a deletion. Let $I(i,j)$ denote the score of an optimal alignment between $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$ ending with an insertion. Let $S(i,j)$ denote the score of an optimal alignment between $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$.

By definition, $D(i,j)$ can be derived as follows.

-4 -4
A T A C A T G T C T -
G T A C - - G T C G G
-5 +8 +8 +8 -3 -3 +8 +8 +8 -5 -3 = 29

29-4-4=21

Fig. 3.13 The score of a global alignment of the two sequences ATACATGTCT and GTACGTCGG under affine gap penalties.

$$\begin{aligned}
D(i,j) &= \max_{0\le i'\le i-1}\{S(i',j)-\alpha-(i-i')\times\beta\}\\
&= \max\{\max_{0\le i'\le i-2}\{S(i',j)-\alpha-(i-i')\times\beta\}, S(i-1,j)-\alpha-\beta\}\\
&= \max\{\max_{0\le i'\le i-2}\{S(i',j)-\alpha-((i-1)-i')\times\beta-\beta\}, S(i-1,j)-\alpha-\beta\}\\
&= \max\{D(i-1,j)-\beta, S(i-1,j)-\alpha-\beta\}.
\end{aligned}$$

This recurrence can be explained in an alternative way. Recall that $D(i,j)$ denotes the score of an optimal alignment between $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$ ending with a deletion. If such an alignment is an extension of the alignment ending at $(i-1,j)$ with a deletion, then it costs only β for such a gap extension. Thus, in this case, $D(i,j)=D(i-1,j)-\beta$. Otherwise, it is a new deletion gap and an additional gap-opening penalty α is imposed. We have $D(i,j)=S(i-1,j)-\alpha-\beta$.

In a similar way, we derive $I(i,j)$ as follows.

$$\begin{aligned}
I(i,j) &= \max_{0\le j'\le j-1}\{S(i,j')-\alpha-(j-j')\times\beta\}\\
&= \max\{\max_{0\le j'\le j-2}\{S(i,j')-\alpha-(j-j')\times\beta\}, S(i,j-1)-\alpha-\beta\}\\
&= \max\{I(i,j-1)-\beta, S(i,j-1)-\alpha-\beta\}.
\end{aligned}$$

Therefore, with proper initializations, $D(i,j)$, $I(i,j)$ and $S(i,j)$ can be computed by the following recurrences:

$$D(i,j)=\max\begin{cases}D(i-1,j)-\beta,\\ S(i-1,j)-\alpha-\beta;\end{cases}$$

$$I(i,j)=\max\begin{cases}I(i,j-1)-\beta,\\ S(i,j-1)-\alpha-\beta;\end{cases}$$

$$S(i,j) = \max \begin{cases} D(i,j), \\ I(i,j), \\ S(i-1,j-1) + \sigma(a_i, b_j). \end{cases}$$

3.5.2 Constant Gap Penalties

Now let us consider the constant gap penalties where each gap, regardless of its length, is charged with a nonnegative constant penalty α.

Let $D(i,j)$ denote the score of an optimal alignment between $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$ ending with a deletion. Let $I(i,j)$ denote the score of an optimal alignment between $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$ ending with an insertion. Let $S(i,j)$ denote the score of an optimal alignment between $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$. With proper initializations, $D(i,j)$, $I(i,j)$ and $S(i,j)$ can be computed by the following recurrences. In fact, these recurrences can be easily derived from those for the affine gap penalties by setting β to zero. A gap penalty is imposed when the gap is just opened, and the extension is free of charge.

$$D(i,j) = \max \begin{cases} D(i-1,j), \\ S(i-1,j) - \alpha; \end{cases}$$

$$I(i,j) = \max \begin{cases} I(i,j-1), \\ S(i,j-1) - \alpha; \end{cases}$$

$$S(i,j) = \max \begin{cases} D(i,j), \\ I(i,j), \\ S(i-1,j-1) + \sigma(a_i, b_j). \end{cases}$$

3.5.3 Restricted Affine Gap Penalties

Another interesting scoring scheme is called the restricted affine gap penalties, in which a gap of length k is penalized by $\alpha + f(k) \times \beta$, where α and β are both nonnegative constants, and $f(k) = \min\{k, \ell\}$ for a given positive integer ℓ.

In order to deal with the free long gaps, two more matrices $D'(i,j)$ and $I'(i,j)$ are used to record the long gap penalties in advance. With proper initializations, $D(i,j)$, $D'(i,j)$, $I(i,j)$, $I'(i,j)$, and $S(i,j)$ can be computed by the following recurrences:

$$D(i,j) = \max \begin{cases} D(i-1,j) - \beta, \\ S(i-1,j) - \alpha - \beta; \end{cases}$$

VERITAS JUSTITIA LIBERTAS
Fachbereich Mathematik u. Informatik
FREIE UNIVERSITÄT BERLIN

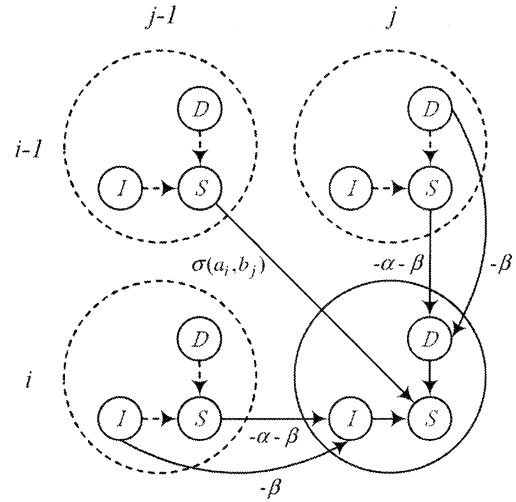

Fig. 3.14 There are seven ways entering the three grid points of an entry (i,j).

$$D'(i,j) = \max \begin{cases} D'(i-1,j), \\ S(i-1,j) - \alpha - \ell \times \beta; \end{cases}$$

$$I(i,j) = \max \begin{cases} I(i,j-1) - \beta, \\ S(i,j-1) - \alpha - \beta; \end{cases}$$

$$I'(i,j) = \max \begin{cases} I'(i,j-1), \\ S(i,j-1) - \alpha - \ell \times \beta; \end{cases}$$

$$S(i,j) = \max \begin{cases} D(i,j), \\ D'(i,j), \\ I(i,j), \\ I'(i,j), \\ S(i-1,j-1) + \sigma(a_i, b_j). \end{cases}$$

3.6 Space-Saving Strategies

Straightforward implementation of the dynamic-programming algorithms utilizes quadratic space to produce an optimal global or local alignment. For analysis of long DNA sequences, this space restriction is more crucial than the time constraint.

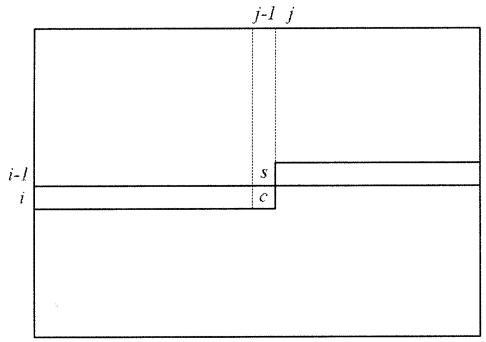

Fig. 3.15 Entry locations of S^- just before the entry value is evaluated at (i, j).

Because of this, different methods have been proposed to reduce the space used for aligning globally or locally two sequences.

We first describe a space-saving strategy proposed by Hirschberg in 1975 [91]. It uses only "linear space," *i.e.*, space proportional to the sum of the sequences' lengths. The original formulation was for the longest common subsequence problem that is discussed in Section 2.4.4. But the basic idea is quite robust and works readily for aligning globally two sequences with affine gap costs as shown by Myers and Miller in 1988 [150]. Remarkably, this space-saving strategy has the same time complexity as the original dynamic programming method presented in Section 3.3.

To introduce Hirschberg's approach, let us first review the original algorithm presented in Figure 3.5 for aligning two sequences of lengths m and n. It is apparent that the scores in row i of dynamic programming matrix S are calculated from those in row $i-1$. Thus, after the scores in row i of S are calculated, the entries in row $i-1$ of S will no longer be used and hence the space used for storing these entries can be recycled to calculate and store the entries in row $i+1$. In other words, we can get by with space for two rows, since all that we ultimately want is the single entry $S[m,n]$ in the rightmost cell of the last row.

In fact, a single array S^- of size n, together with two extra variables, is adequate. $S^-[j]$ holds the most recently computed value for each $1 \le j \le n$, so that as soon as the value of the jth entry of S^- is computed, the old value at the entry is overwrited. There is a slight conflict in this strategy since we need the old value of an entry to compute a new value of the entry. To avoid this conflict, two additional variables, say s and c, are introduced to hold the new and old values of the entry, respectively. Figure 3.15 shows the locations of the scores kept in S^- and in variables s and c. When $S^-[j]$ is updated, $S^-[j']$ holds the score in the entry (i, j') in row i for each $j' < j$, and it holds the score in the entry $(i-1, j')$ for any $j' \ge j$. Figure 3.16 gives the pseudo-code for computing the score of an optimal global alignment in linear space.

In the dynamic programming matrix S of aligning sequences $A = a_1a_2\ldots a_m$ and $B = b_1b_2\ldots b_n$, $S[i,j]$ denotes the optimal score of aligning $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$

Algorithm FORWARD_SCORE($A = a_1a_2\ldots a_m$, $B = b_1b_2\ldots b_n$)
begin
  $S^-[0] \leftarrow 0$
  for $j \leftarrow 1$ **to** n **do** $S^-[j] \leftarrow S^-[j-1] - \beta$
  for $i \leftarrow 1$ **to** m **do**
    $s \leftarrow S^-[0]$
    $c \leftarrow S^-[0] - \beta$
    $S^-[0] \leftarrow c$
    for $j \leftarrow 1$ **to** n **do**

$$c \leftarrow \max \begin{cases} S^-[j] - \beta \\ c - \beta \\ s + \sigma(a_i, b_j) \end{cases}$$

      $s \leftarrow S^-[j]$
      $S^-[j] \leftarrow c$
  Output $S^-[n]$ as the score of an optimal alignment.
end

Fig. 3.16 Computation of the optimal score of aligning sequences of lengths m and n in linear space $O(n)$.

or, equivalently, the maximum score of a path from $(0,0)$ to the cell (i,j) in the alignment graph. By symmetry, the optimal score of aligning $a_{i+1}a_{i+2}\ldots a_m$ and $b_{j+1}b_{j+2}\ldots b_n$ or the maximum score of a path from (i,j) to (m,n) in the alignment graph can be calculated in linear space in a backward manner. Figure 3.17 gives the pseudo-code for computing the score of an optimal global alignment in a backward manner in linear space.

In what follows, we use $S^-[i,j]$ and $S^+[i,j]$ to denote the maximum score of a path from $(0,0)$ to (i,j) and that from (i,j) to (m,n) in the alignment graph, respectively. Without loss of generality, we assume that m is a power of 2. Obviously, for each j, $S^-[m/2,j] + S^+[m/2,j]$ is the maximum score of a path from $(0,0)$ to (m,n) through $(m/2,j)$ in the alignment graph. Choose j_{mid} such that

$$S^-[m/2, j_{mid}] + S^+[m/2, j_{mid}] = \max_{1 \le j \le n} S^-[m/2,j] + S^+[m/2,j].$$

Then, $S^-[m/2, j_{mid}] + S^+[m/2, j_{mid}]$ is the optimal alignment score of A and B and there is a path having such a score from $(0,0)$ to (m,n) through $(m/2, j_{mid})$ in the alignment graph.

Hirschberg's linear-space approach is first to compute $S^-[m/2,j]$ for $1 \le j \le n$ by a forward pass, stopping at row $m/2$ and to compute $S^+[m/2,j]$ for $1 \le j \le n$ by a backward pass and then to find j_{mid}. After j_{mid} is found, recursively compute an optimal path from $(0,0)$ to $(m/2, j_{mid})$ and an optimal path from $(m/2, j_{mid})$ to (m,n).

As the problem is partitioned further, there is a need to have an algorithm that is capable of delivering an optimal path for any specified two ends. In Figure 3.18, algorithm LINEAR_ALIGN is a recursive procedure that delivers a maximum-scoring path from (i_1, j_1) to (i_2, j_2). To deliver the whole optimal alignment, the two ends are initially specified as $(0,0)$ and (m,n).

Now let us analyze the time and space taken by Hirschberg's approach. Using the algorithms given in Figures 3.16 and 3.17, both the forward and backward pass take $O(nm/2)$-time and $O(n)$-spaces. Hence, it takes $O(mn)$-time and $O(n)$-spaces to find j_{mid}. Set $T = mn$ and call it the size of the problem of aligning A and B. At each recursive step, a problem is divided into two subproblems. However, regardless of where the optimal path crosses the middle row $m/2$, the total size of the two resulting subproblems is exactly half the size of the problem that we have at the recursive step (see Figure 3.19). It follows that the total size of all problems, at all levels of recursion, is at most $T + T/2 + T/4 + \cdots = 2T$. Because computation time is directly proportional to the problem size, Hirschberg's approach will deliver an optimal alignment using $O(2T) = O(T)$ time. In other words, it yields an $O(mn)$-time, $O(n)$-space global alignment algorithm.

Hirschberg's original method, and the above discussion, apply to the case where the penalty for a gap is merely proportional to the gap's length, i.e., $k \times \beta$ for a k-symbol gap. For applications in molecular biology, one wants penalties of the form $\alpha + k \times \beta$, i.e., each gap is assessed an additional "gap-open" penalty α. Actually, one can be slightly more general and substitute residue-dependent penalties for β. In Section 3.5.1, we have shown that the relevant alignment graph is more complicated. Now at each grid point (i, j) there are three nodes, denoted $(i, j)_S$, $(i, j)_D$, and $(i, j)_I$, and generally seven entering edges, as pictured in Figure 3.14. The alignment problem is to compute a highest-score path from $(0,0)_S$ to $(m,n)_S$. Fortunately, Hirschberg's strategy extends readily to this more general class of alignment scores [150]. In essence, the main additional complication is that for each defining corner of a subproblem, we need to specify one of the grid point's three nodes.

Another issue is how to deliver an optimal *local* alignment in linear space. Recall that in the local alignment problem, one seeks a highest-scoring alignment where the end nodes can be arbitrary, *i.e.*, they are not restricted to $(0,0)_S$ and $(m,n)_S$. In fact, it can be reduced to a global alignment problem by performing a linear-space

Algorithm BACKWARD_SCORE($A = a_1 a_2 \ldots a_m$, $B = b_1 b_2 \ldots b_n$)
begin
  $S^+[n] \leftarrow 0$
  for $j \leftarrow n-1$ **down to** 0 **do** $S^+[j] \leftarrow S^+[j+1] - \beta$
  for $i \leftarrow m-1$ **down to** 0 **do**
    $s \leftarrow S^+[n]$
    $c \leftarrow S^+[n] - \beta$
    $S^+[n] \leftarrow c$
    for $j \leftarrow n-1$ **down to** 0 **do**
      $c \leftarrow \max \begin{cases} S^+[j] - \beta \\ c - \beta \\ s + \sigma(a_{i+1}, b_{j+1}) \end{cases}$
      $s \leftarrow S^+[j]$
      $S^+[j] \leftarrow c$
  Output $S^+[0]$ as the score of an optimal alignment.
end

Fig. 3.17 Backward computation of the score of an optimal global alignment in linear space.

score-only pass over the dynamic-programming matrix to locate the first and last nodes of an optimal local alignment, then delivering a global alignment between these two nodes by applying Hirschberg's approach.

Algorithm LINEAR_ALIGN($A = a_1 a_2 \ldots a_m$, $B = b_1 b_2 \ldots b_n$, i_1, j_1, i_2, j_2)
begin
  if $i_1 + 1 \le i_2$ **or** $j_1 + 1 \le j_2$ **then**
    Output the aligned pairs for the maximum-score path from (i_1, j_1) to (i_2, j_2)
  else
    $i_{mid} \leftarrow \lfloor (i_1 + i_2)/2 \rfloor$
    // Find the maximum scores from (i_1, j_1)
    $S^-[j_1] \leftarrow 0$
    for $j \leftarrow j_1 + 1$ **to** j_2 **do** $S^-[j] \leftarrow S^-[j-1] - \beta$
    for $i \leftarrow i_1 + 1$ **to** i_{mid} **do**
      $s \leftarrow S^-[j_1]$
      $c \leftarrow S^-[j_1] - \beta$
      $S^-[j_1] \leftarrow c$
      for $j \leftarrow j_1 + 1$ **to** j_2 **do**
        $c \leftarrow \max \begin{cases} S^-[j] - \beta \\ c - \beta \\ s + \sigma(a_i, b_j) \end{cases}$
        $s \leftarrow S^-[j]$
        $S^-[j] \leftarrow c$
    // Find the maximum scores to (i_2, j_2)
    $S^+[j_2] \leftarrow 0$
    for $j \leftarrow j_2 - 1$ **down to** j_1 **do** $S^+[j] \leftarrow S^+[j+1] - \beta$
    for $i \leftarrow i_2 - 1$ **down to** i_{mid} **do**
      $s \leftarrow S^+[j_2]$
      $c \leftarrow S^+[j_2] - \beta$
      $S^+[j_2] \leftarrow c$
      for $j \leftarrow j_2 - 1$ **down to** j_1 **do**
        $c \leftarrow \max \begin{cases} S^+[j] - \beta \\ c - \beta \\ s + \sigma(a_{i+1}, b_{j+1}) \end{cases}$
        $s \leftarrow S^+[j]$
        $S^+[j] \leftarrow c$
    // Find where maximum-score path crosses row i_{mid}
    $j_{mid} \leftarrow$ value $j \in [j_1, j_2]$ that maximizes $S^-[j] + S^+[j]$
    LINEAR_ALIGN(A,B, i_1, j_1, i_{mid}, j_{mid})
    LINEAR_ALIGN(A,B, i_{mid}, j_{mid}, i_2, j_2)
end

Fig. 3.18 Computation of an optimal global alignment in linear space.

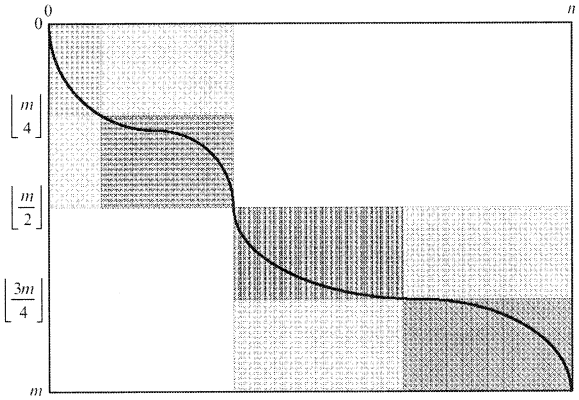

Fig. 3.19 Hirschberg's linear-space approach.

3.7 Other Advanced Topics

In this section, we discuss several advanced topics such as constrained sequence alignment, similar sequence alignment, suboptimal alignment, and robustness measurement.

3.7.1 Constrained Sequence Alignment

Rigorous sequence alignment algorithms compare each residue of one sequence to every residue of the other. This requires computational time proportional to the product of the lengths of the given sequences. Biologically relevant sequence alignments, however, usually extend from the beginning of both sequences to the end of both sequences, and thus the rigorous approach is unnecessarily time consuming; significant sequence similarities are rarely found by aligning the end of one sequence with the beginning of the other.

As a result of the biological constraint, it is frequently possible to calculate an optimal alignment between two sequences by considering only those residues that are within a diagonal band in which each row has only w cells. With sequences $A = a_1a_2\ldots a_m$ and $B = b_1b_2\ldots b_n$, one can specify constants $\ell \leq u$ such that aligning a_i with b_j is permitted only if $\ell \leq j - i \leq u$. For example, it rarely takes a dozen insertions or deletions to align any two members of the globin superfamily; thus, an optimal alignment of two globin sequences can be calculated in $O(nw)$ time that is identical to the rigorous alignment that requires $O(nm)$ time.

Alignment within a band is used in the final stage of the FASTA program for rapid searching of protein and DNA sequence databases (Pearson and Lipman, 1988; Pearson, 1990). For optimization in a band, the requirement to "start at the beginning, end at the end" is reflected in the $\ell \leq \min\{0, n-m\}$ and $u \geq \max\{0, n-m\}$ constraints. "Local" sequence alignments do not require that the beginning and end of the alignment correspond to the beginning and end of the sequence, *i.e.*, the aligned sequences can be arbitrary substrings of the given sequences, A and B; they simply require that the alignment have the highest similarity score. For a "local" alignment in a band, it is natural to relax the requirement to $\ell \leq u$. Algorithms for computing an optimal local alignment can utilize a global alignment procedure to perform subcomputations: once locally optimal substrings A' of A and B' of B are found, which can be done by any of several available methods, a global alignment procedure is called to align A' and B'. Appropriate values of ℓ' and u' for the global problem are inferred from the ℓ and u of the local problems. In other situations, a method to find unconstrained local alignments, *i.e.*, without band limits, might determine appropriate values of ℓ and u before invoking a global alignment procedure within a band.

Although the application of rigorous alignment algorithms to long sequences can be quite time-consuming, it is often the space requirement that is limiting in practice. Hirschberg's approach, introduced in Section 3.6, can be easily modified to find a solution locating in a band. Unfortunately, the resulting time required to produce the alignment can exceed that of the score-only calculation by a substantial factor. If T denotes the number of entries in the band of the dynamic programming matrix, then $T = O(nw)$. Producing an alignment involves computing as many as $T \times \log_2 n$ entries (including recomputations of entries evaluated at earlier steps). Thus, the time to deliver an alignment exceeds that for computing its score in a band by a log factor.

To avoid the log factor, we need a new way to subdivide the problem that limits the subproblems to some fraction, $\alpha < 1$, of the band. Figure 3.20 illustrates the idea. The score-only backward pass is augmented so that at each point it computes the next place where an optimal path crosses the mid-diagonal, *i.e.*, diagonal $(\ell + u)/2$. Using only linear space, we can save this information at every point on the "current row" or on the mid-diagonal. When this pass is completed, we can use the retained information to find the sequence of points where an optimal solution crosses the mid-diagonal, which splits the problem into some number of subproblems. The total area of these subproblems is no more than half of the original area for a narrow band with widely spaced crossing points; in other cases it is even less.

It should be noted that this band-aligning algorithm could be considered as a generalization of Hirschberg's approach by rotating the matrix partition line. The idea of partition line rotation has been exploited in devising parallel sequence comparison algorithms. Nevertheless, the dividing technique proposed in this section, which produces more than two subproblems, reveals a new paradigm for space-saving strategies.

Another extension is to consider the situations where the ith entry of the first sequence can be aligned to the jth entry of the second sequence only if $L[i] \leq j \leq U[i]$, for given left and right bounds L and U. As in the band alignment problem, we can apply the idea of defining a midpoint *partition line* that bisects the region

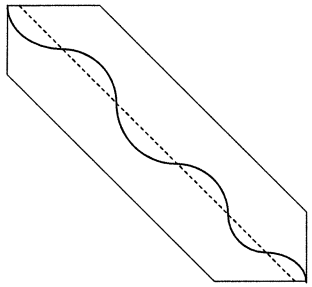

Fig. 3.20 Dividing a band by its middle diagonal.

into two nearly equal parts. Here we introduce a more general approach that can be easily utilized by other relevant problems.

Given a narrow region R with two boundary lines L and U, we can proceed as follows. We assume that L and U are non-decreasing since if, e.g., $L[i]$ were larger than $L[i+1]$, we could set $L[i+1]$ to equal $L[i]$ without affecting the set of constrained alignments. Enclose as many rows as possible from the top of the region in an upright rectangle, subject to the condition that the rectangle's area at most doubles the area of its intersection with R. Then starting with the first row of R not in the rectangle, we cover additional rows of R with a second such rectangle, and so on.

A score-only backward pass is made over R, computing S^+. Values of S^+ are retained for the top line in every rectangle (the top rectangle can be skipped). It can be shown that the total length of these pieces cannot exceed three times the total number of columns, as required for a linear space bound. Next, perform a score-only forward pass, stopping at the last row in the first rectangle. A sweep along the boundary between the first and second rectangles locates a crossing edge on an optimal path through R. That is, we can find a point p on the last row of the first rectangle and a point q on the first row of the second rectangle such that there is a vertical or diagonal edge e from p to q, and e is on an optimal path. Such an optimal path can be found by applying Hirschberg's strategy to R's intersection with the first rectangle (omitting columns following p) and recursively computing a path from q through the remainder of R. This process inspects a grid point at most once during the backward pass, once in a forward pass computing p and q, and an average of four times for applying Hirschberg's method to R's intersection with a rectangle.

3.7.2 Similar Sequence Alignment

If two sequences are very similar, more efficient algorithms can be devised to deliver an optimal alignment for them. In this case, we know that the maximum-scoring path in the alignment graph will not get too far away from the diagonal of the source

$(0,0)$. One way is to draw a constrained region to restrict the diversion and run the constrained alignment algorithm introduced in Section 3.7.1. Another approach is to grow the path greedily until the destination (m,n) is reached. For this approach, instead of working with the maximization of the alignment score, we look for the minimum-cost set of single-nucleotide changes (*i.e.*, insertions, deletions, or substitutions) that will convert one sequence to the other. Any match costs zero, which allows us to have a free advance. As for penalty, we have to pay a certain amount of cost to get across a mismatch or a gap symbol.

Now we briefly describe the approach based on the diagonalwise monotonicity of the cost tables. The following cost function is employed. Each match costs 0, each mismatch costs 1, and a gap of length k is penalized at the cost of $k+1$. Adding 1 to a gap's length to derive its cost decreases the likelihood of generating gaps that are separated by only a few paired nucleotides. The edit graph for sequences A and B is a directed graph with a vertex at each integer grid point (x,y), $0 \le x \le m$ and $0 \le y \le n$. Let $I(z,c)$ denote the x value of the farthest point in diagonal z $(=y-x)$ that can be reached from the source (*i.e.*, grid point $(0,0)$) with cost c and that is *free* to open an insertion gap. That is, the grid point can be (1) reached by a path of cost c that ends with an insertion, or (2) reached by any path of cost $c-1$ and the gap-open penalty of 1 can be "paid in advance." (The more traditional definition, which considers only case (1), results in the storage of more vectors.) Let $D(z,c)$ denote the x value of the farthest point in diagonal z that can be reached from the source with cost c and is *free* to open a deletion gap. Let $S(z,c)$ denote the x value of the farthest point in diagonal z that can be reached from the source with cost c. With proper initializations, these vectors can be computed by the following recurrence relations:

$$I(z,c) = \max\{I(z-1,c-1), S(z,c-1)\},$$

$$D(z,c) = \max\{D(z+1,c-1)+1, S(z,c-1)\},$$

$$S(z,c) = \mathit{snake}(z, \max\{S(z,c-1)+1, I(z,c), D(z,c)\}),$$

where $\mathit{snake}(z,x) = \max\{x, \max\{t : a_x \ldots a_{t-1} = b_{x+z} \ldots b_{t-1+z}\}\}$.

Since vectors at cost c depend only on those at costs c and $c-1$, it is straightforward to derive a dynamic-programming algorithm from the above recurrence relations.

3.7.3 Suboptimal Alignment

Molecular biology is rapidly becoming a data-rich science with extensive computational needs. More and more computer scientists are working together on developing efficient software tools for molecular biologists. One major area of potential interaction between computer scientists and molecular biologists arises from the need for analyzing biological information. In particular, optimal alignments mentioned in

previous sections have been used to reveal similarities among biological sequences, to study gene regulation, and even to infer evolutionary trees.

However, biologically significant alignments are not necessarily mathematically optimized. It has been shown that sometimes the neighborhood of an optimal alignment reveals additional interesting biological features. Besides, the most strongly conserved regions can be effectively located by inspecting the range of variation of suboptimal alignments. Although rigorous statistical analysis for the mean and variance of optimal global alignment scores is not yet available, suboptimal alignments have been successfully used to informally estimate the significance of an optimal alignment.

For most applications, it is impractical to enumerate all suboptimal alignments since the number could be enormous. Therefore, a more compact representation of all suboptimal alignments is indispensable. A 0-1 matrix can be used to indicate if a pair of positions is in some suboptimal alignment or not. However, this approach misses some connectivity information among those pairs of positions. An alternative is to use a set of "canonical" suboptimal alignments to represent all suboptimal alignments. The kernel of that representation is a minimal directed acyclic graph (DAG) containing all suboptimal alignments.

Suppose we are given a threshold score that does not exceed the optimal alignment score. An alignment is suboptimal if its score is at least as large as the threshold score. Here we briefly describe a linear-space method that finds all edges that are contained in at least one path whose score exceeds a given threshold τ. Again, a recursive subproblem will consist of applying the alignment algorithm over a rectangular portion of the original dynamic-programming matrix, but now it is necessary that we continue to work with values S^- and S^+ that are defined relative to the original problem. To accomplish this, each problem to be solved is defined by specifying values of S^- for nodes on the upper and left borders of the defining rectangle, and values of S^+ for the lower and right borders.

To divide a problem of this form, a forward pass propagates values of S^- to nodes in the middle row and the middle column, and a backward pass propagates values S^+ to those nodes. This information allows us to determine all edges starting in the middle row or middle column that are contained in a path of score at least τ. The data determining any one of the four subproblems, *i.e.*, the arrays of S values on its borders, is then at most half the size of the set of data defining the parent problem. The maximum total space requirement is realized when recursion reaches a directly solvable problem where there is only the leftmost cell of the first row of the original grid left; at that time there are essentially $2(m+n)$ S-values saved for borders of the original problem, $m+n$ values on the middle row and column of the original problem, $(m+n)/2$ values for the upper left subproblem, $(m+n)/4$ values for the upper-left-most subsubproblem, etc., giving a total of about $4(m+n)$ retained S-values.

3.7.4 Robustness Measurement

The utility of information about the reliability of different regions within an alignment is widely appreciated, see [192] for example. One approach to obtaining such information is to determine suboptimal alignments, *i.e.*, some or all alignments that come within a specified tolerance of the optimum score, as discussed in Section 3.7.3. However, the number of suboptimal alignments, or even alternative optimal alignments, can easily be so large as to preclude an exhaustive enumeration.

Sequence conservation has proved to be a reliable indicator of at least one class of regulatory elements. Specifically, regions of six or more consecutive nucleotides that identical across a range of mammalian sequences, called "phylogenetic footprints," frequently correspond to binding sits for sequence-specific nuclear proteins. It is also interesting to look for longer, imperfectly conserved (but stronger matching) regions, which may indicate other sorts of regulatory elements, such as a region that binds to a nuclear matrix or assumes some altered chromatin structure.

In the following, we briefly describe some interesting measurements of the robustness of each aligned pair of a pairwise alignment. The first method computes, for each position i of the first sequence, the lower and upper limits of the positions in the second sequence to which it can be aligned and still come within a specified tolerance of the optimum alignment score. Delimiting suboptimal alignments this way, rather than enumerating all of them, allows the computation to run in only a small constant factor more time than the computation of a single optimal alignment.

Another method determines, for each aligned pair of an optimal alignment, the amount by which the optimum score must be lowered before reaching an alignment not containing that pair. In other words, if the optimum alignment score is s and the aligned pair is assigned the robustness-measuring number r, then any alignment scoring strictly greater than $s-r$ aligns those two sequence positions, whereas some alignment of score $s-r$ does not align them. As a special case, this value

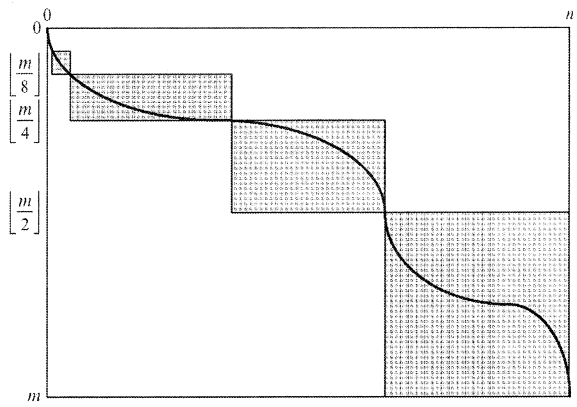

Fig. 3.21 The total number of the boundary entries in the active subproblems is $O(m+n)$.

tells whether the pair is in all optimal alignments (namely, the pair is in all optimal alignments if and only if its associated value is non-zero). These computations are performed using dynamic-programming methods that require only space proportional to the sum of the two sequence lengths. It has also been shown on how to efficiently handle the case where alignments are constrained so that each position, say position i, of the first sequence can be aligned only to positions on a certain range of the second sequence.

To deliver an optimal alignment, Hirschberg's approach applies forward and backward passes in the first nondegenerate rectangle along the optimal path being generated. Within a subproblem (i.e., rectangle) the scores of paths can be taken relative to the "start node" at the rectangle's upper left and the "end node" at the rightmost cell of the last row. This means that a subproblem is completely specified by giving the coordinates of those two nodes. In contrast, methods for the robustness measurement must maintain more information about each pending subproblem. Fortunately, it can be done in linear space by observing that the total number of the boundary entries of all pending subproblems of Hirschberg's approach is bounded by $O(m+n)$ (see Figure 3.21).

3.8 Bibliographic Notes and Further Reading

Sequence alignment is one of the most fundamental components of bioinformatics. For more references and applications, see the books by Sankoff and Kruskal [175], Waterman [197], Gusfield [85], Durbin et al. [61], Pevzner [165], Jones and Pevzner [98], and Deonier et al. [58], or recent survey papers by Batzoglou [23] and Notredame [154].

3.1

We compile a list of pairwise alignment tools in Table C.1 of Appendix C.

3.2

It is very easy to visualize in a dot-matrix representation certain sequence similarities such as insertions, deletions, repeats, or inverted repeats.

3.3

The global alignment method was proposed by Needleman and Wunsch [151]. Such a dynamic-programming method was independently discovered by Wagner and Fischer [194] and workers in other fields. For a survey of the history, see the book by Sankoff and Kruskal [175].

3.4

The local alignment method is so-called the *Smith-Waterman algorithm* [180]. In most applications of pairwise alignment, affine gap penalties are used [78].

3.5

Biologists need more general measurements of sequence relatedness than are typically considered by computer scientists. The most popular formulation in the computer science literature is the "longest common subsequence problem," which is equivalent to scoring alignments by simply counting the number of exact matches. For comparing protein sequences, it is important to reward alignment of residues that are similar in functions [70].

For both DNA and protein sequences, it is standard to penalize a long "gap," *i.e.*, a block of consecutive dashes, less than the sum of the penalties for the individual dashes in the gap. In reality, a gap would most likely represent a single insertion or deletion of a block of letters rather than multiple insertions or deletions of single letters [71]. This is usually accomplished by charging $\alpha + k \times \beta$ for a gap of length k. Thus the "gap-open penalty" α is assessed for every gap, regardless of length, and an additional "gap-extension penalty" β is charged for every dash in the gap. Such penalties are called *affine* gap penalties. Gotoh [78] showed how to efficiently compute optimal alignments under such a scoring scheme.

Even more general models for quantifying sequence relatedness have been proposed. For example, it is sometimes useful to let the penalty for adding a symbol to a gap depend on the position of the gap within the sequence [81], which is motivated by the observation that insertions in certain regions of a protein sequence can be much more likely than at other regions. Another generalization is to let the incremental gap score $\delta_i = c_{i+1} - c_i$, where a k-symbol gap scores c_k, be a monotone function of i, *e.g.*, $\delta_1 \geq \delta_2 \geq \cdots$ [139, 196].

Gotoh [79] proposed the piecewise linear gap penalties to allow long gaps in a resulting alignment. Huang and Chao [93] generalized the global alignment algorithms to compare sequences with intermittent similarities, an ordered list of similar regions separated by different regions.

Most alignment methods can be extended to deal with free end gaps in a straightforward way.

3.6

Readers can refer to [40, 41, 43] for more space-saving strategies.

3.7

Readers should also be aware that the hidden Markov models are a probabilistic approach to sequence comparison. They have been widely used in the bioinformatics community [61]. Given an observed sequence, the Viterbi algorithm computes the most probable state path. The forward algorithm computes the probability that a given observed sequence is generated by the model, whereas the backward algorithm computes the probability that a given observed symbol was generated by

a given state. The book by Durbin et al. [61] is a terrific reference book for this paradigm.

Alignment of two genomic sequences poses problems not well addressed by earlier alignment programs.

PipMaker [178] is a software tool for comparing two long DNA sequences to identify conserved segments and for producing informative, high-resolution displays of the resulting alignments. It displays a percent identity plot (pip), which shows both the position in one sequence and the degree of similarity for each aligning segment between the two sequences in a compact and easily understandable form. The alignment program used by the PipMaker network server is called BLASTZ [177]. It is an independent implementation of the Gapped BLAST algorithm specifically designed for aligning two long genomic sequences. Several modifications have been made to BLASTZ to attain efficiency adequate for aligning entire mammalian genomes and to increase the sensitivity.

MUMmer [119] is a system for aligning entire genomes rapidly. The core of the MUMmer algorithm is a suffix tree data structure, which can be built and searched in linear time and which occupies only linear space. DisplayMUMs 1.0 graphically presents alignments of MUMs from a set of query sequences and a single reference sequence. Users can navigate MUM alignments to visually analyze coverage, tiling patterns, and discontinuities due to misassemblies or SNPs.

The analysis of genome rearrangements is another exciting field for whole genome comparison. It looks for a series of genome rearrangements that would transform one genome into another. It was pioneered by Dobzhansky and Sturtevant [60] in 1938. Recent milestone advances include the works by Bafna and Pevzner [19], Hannenhalli and Pevzner [86], and Pevzner and Tesler [166].

Chapter 4
Homology Search Tools

The alignment methods introduced in Chapter 3 are good for comparing two sequences accurately. However, they are not adequate for homology search against a large biological database such as GenBank. As of February 2008, there are approximately 85,759,586,764 bases in 82,853,685 sequence records in the traditional GenBank divisions. To search such kind of huge databases, faster methods are required for identifying the homology between the query sequence and the database sequence in a timely manner.

One common feature of homology search programs is the filtration idea, which uses exact matches or approximate matches between the query sequence and the database sequence as a basis to judge if the homology between the two sequences passes the desired threshold.

This chapter is divided into six sections. Section 4.1 describes how to implement the filtration idea for finding exact word matches between two sequences by using efficient data structures such as hash tables, suffix trees, and suffix arrays.

FASTA was the first popular homology search tool, and its file format is still widely used. Section 4.2 briefly describes a multi-step approach used by FASTA for finding local alignments.

BLAST is the most popular homology search tool now. Section 4.3 reviews the first version of BLAST, *Ungapped BLAST*, which generates ungapped alignments. It then reviews two major products of BLAST 2.0: Gapped BLAST and Position-Specific Iterated BLAST (PSI-BLAST). Gapped BLAST produces gapped alignments, yet it is able to run faster than the original one. PSI-BLAST can be used to find distant relatives of a protein based on the profiles derived from the multiple alignments of the highest scoring database sequence segments with the query segment in iterative Gapped BLAST searches.

Section 4.4 describes BLAT, short for "BLAST-like alignment tool." It is often used to search for the database sequences that are closely related to the query sequences such as producing mRNA/DNA alignments and comparing vertebrate sequences.

PatternHunter, introduced in Section 4.5, is more sensitive than BLAST when a hit contains the same number of matches. A novel idea in PatternHunter is the