CO7100/CO7200: Algorithms for Bioinformatics

Thomas Erlebach

Contents

1 Course overview

Processing biological data often requires complex computations on large data volumes. Efficient algorithms are needed to solve such processing tasks. This course teaches the basic concepts of algorithms in the context of bioinformatics. It covers design principles, techniques for the analysis of algorithms, and issues of algorithm implementation in Java. It also covers probabilistic models that underlie the formulation of various biological data processing tasks as computational problems.

The topics covered in the course include string matching, pairwise sequence alignment, multiple sequence alignment, hidden Markov models, restriction site mapping (the partial digest problem), building phylogenetic trees, and genome rearrangement (sorting by reversals). These notes cover the first part of the course (string matching and pairwise sequence alignment).

1.1 Literature

The course is mainly based on material from [DEKM], a textbook emphasising the probabilistic models underlying many computational problems in bioinformatics, and [JP], a recent textbook introducing algorithms in the context of bioinformatics. For further reading, a standard textbook covering algorithms in general (not only in the context of bioinformatics) is [CLRS], and algorithms and data structures for exact and inexact string matching and sequence analysis in general are covered in [G].

2 Introduction to algorithms

Loosely speaking, an algorithm is a set of rules for solving a problem in a finite number of steps. The origin of the word "algorithm" is "al-kawarizmi", the name of a ninth-century Persian mathematician. Algorithms are machine independent and language independent. Therefore, algorithms are usually described in natural language or given as *pseudo-code* (a semi-formal way of presenting an algorithm in a style similar to a real programming language, but where the syntax is flexible and one can also add statements in natural language etc.). An algorithm can be implemented as a computer program in any programming language, for example Java.

Using algorithms to solve computational problems involves the following aspects:

- Modelling the problem: One needs to determine how an input of the problem is given, what a solution to the problem is, and, if there is more than one possible solution, how the quality of a solution is to be determined. Often, especially in computational biology, statistical considerations are important in order to arrive at meaningful problem formulations.

- Design of algorithms: One needs to come up with an algorithm that solves the given problem correctly and efficiently. Many algorithms are obtained using design

principles such as divide-and-conquer, dynamic programming, greedy algorithms, or backtracking.

- Analysis of algorithms: One wants to analyse the behaviour of an algorithm, e.g., how long the algorithm takes on inputs of a certain size, how much memory space the algorithm needs, and how good the solutions computed by the algorithm are. Two fundamental types of analysis are *worst-case analysis*, considering how the algorithm behaves on worst possible inputs, and *average-case analysis*, considering how the algorithm behaves on average.

- Algorithm implementation: When one has found a good algorithm, one wants to implement it in a programming language and use the implementation to solve instances of the given problem.

- Experimental evaluation: If an algorithm has been implemented, one can run it on a variety of test inputs to see how the algorithm behaves in practice or to compare its performance to that of other algorithms.

2.1 Time complexity of algorithms

One of the most important properties of an algorithm is its running-time. Naturally, one is interested in algorithms that run as quickly as possible. If an algorithm solves a computational task within seconds, one can run the algorithm repeatedly in an interactive system without incurring excessive waiting times; if the algorithm takes several hours or days to compute its solution, interactive use of the algorithm becomes impossible and the algorithm is much less useful.

When one discusses the running-time of an algorithm, saying how long an implementation of it takes on a specific input on a specific computer is not very informative; such a statement would not tell us how the algorithm behaves on other (larger) inputs or how long it would run on other machines or if implemented in a different programming language. Therefore, the time complexity of an algorithm is usually given in terms of the number of elementary steps it makes, specified as a function of the size (or other parameters) of the input. Furthermore, one does not specify the number of steps exactly, but only up to constant factors. This is where the big-oh notation comes in. Let us assume that we have an algorithm that, on inputs of size n, makes a number of steps that is proportional to n^2. Then we would say that the algorithm has running-time $O(n^2)$, no matter whether the actual number of steps is $0.5 \cdot n^2 - 7$ or $20n^2 + 10n + 3$.

Formally, for functions f and g that map natural numbers to natural numbers, we say that f is $O(g)$ ("f is big-oh of g") if there are constants $n_0 \in \mathbb{N}$ and $c \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all n larger than n_0. Intuitively, "f is $O(g)$" means "f is smaller or equal to g multiplied by an appropriate factor, for all n that are larger than a certain value n_0." In the example above, $0.5n^2 - 7$ is $O(n^2)$ because

$$0.5n^2 - 7 \leq n^2$$

for all n, hence we can choose $c = 1$ and $n_0 = 1$. Furthermore, $20n^2 + 10n + 3$ is $O(n^2)$ because

$$20n^2 + 10n + 3 \leq 20n^2 + n^2 + n^2 = 22n^2$$

for all $n \geq 10$, hence we can choose $c = 22$ and $n_0 = 10$. We will also use the big-oh notation for functions of several variables; for example, if an algorithm has running-time $12n + 27m + 13$, we would say that its time complexity is $O(n + m)$.

Usually, we consider the running-time (or time complexity) of an algorithm as a function of the size or of other parameters of the input. For example, if the input to an algorithm is a string of symbols from a fixed alphabet, it would be natural to consider the running-time as a function of the length of the string. If the input consists of a string s and a natural number k, it might be appropriate to consider the complexity as a function of the length of s and of the number k. In general, we will be interested in the worst-case running-time of the algorithm. This means, we are looking for a function f such that the algorithm makes at most $f(n)$ steps on inputs of size n. If f is $O(g)$, we say that the algorithm has running-time $O(g)$.

Important classes of running-times (measured as a function of n or of n and m) are:

- linear running-time: $O(n)$ or $O(n + m)$
- quadratic running-time: $O(n^2)$ or $O(nm)$
- polynomial running-time: $O(n^k)$ for some constant k
- logarithmic running-time: $O(\log n)$
- exponential running time: $O(c^n)$ for some constant $c > 1$

For problems where the input has size n and the algorithm must look at the whole input, the best running-time we can hope for is linear, $O(n)$.

The important point is that the running-time of an algorithm with complexity $O(n)$ grows linearly with the size of the input; if the input is twice as large, the algorithm takes roughly twice as long. Linear-time algorithms can often deal with inputs of several million bytes in no time. A quadratic algorithm will need four times as long if the input size doubles; the running-time of quadratic algorithms is often acceptable for inputs of several thousand bytes, but becomes quite long (although still feasible) for larger inputs. Algorithms with exponential running-time are feasible only for very small inputs; an algorithm with running-time $O(2^n)$ will take twice as long if the input length increases only by one. Algorithms with logarithmic running-time often occur for problems such as searching in a preprocessed database; the running-time grows only logarithmically in the size of the input, and even for huge inputs the running-time is very small.

Similar to time complexity, one can also consider the amount of memory space an algorithm needs on inputs of a certain size. If a computer program uses more memory space than the physical main memory available, part of the memory needs to be swapped out to the hard disk, and the running-time increases tremendously. Therefore, the space complexity of an algorithm can also be very important.

2.2 NP-Hardness

Many computational problems fall in one of two categories: polynomial-time solvable problems, and NP-hard problems. For problems in the former category, there is an algorithm with polynomial time complexity that solves them optimally. For problems in the latter category, no such algorithm is known, and there is strong evidence (but no proof yet) that no such algorithm exists. The best known algorithms for problems of the latter type are of exponential time complexity, and it is often impossible to compute optimal solutions for large instances of such problems in reasonable time.

"NP" stands for "non-deterministic polynomial time". A problem that is NP-hard and can be solved non-deterministically in polynomial time (we will not go into details about what this means) is called NP-complete.

3 String matching

This section is based on parts of Chapter "String Matching" from [CLRS].

As the first example of a concrete computational problem for which we want to find good algorithms, let us consider the *string matching* problem. A string is a sequence of symbols over a fixed-size alphabet Σ. For DNA strings, we would take $\Sigma = \{A, G, C, T\}$, and for proteins, we would let Σ be a set of letters representing the twenty amino acids. The input for the string matching problem consists of two strings: one is called the *text*, the other is called the *pattern*. We denote the length of the text T by n, and the length of the pattern P by m.

We index the positions of a string S of length k from 0 to $k-1$, and denote the i-th symbol of S by $S[i]$. This means that $T[0]$ is the first symbol of the text and $P[m-1]$ is the last symbol of the pattern, for example. Furthermore, we denote the substring of S that starts in position i and ends in position j, $0 \leq i \leq j \leq n-1$, by $S[i..j]$. A substring of S beginning in position 0 is called a *prefix* of S, a substring ending in position $k-1$ of S is called a *suffix* of T. A string is always a prefix and a suffix of itself. A prefix or suffix of S is called *proper* if it is different from S.

We say that the pattern P occurs in text T at position i if $T[i..i+m-1] = P$, i.e., if the substring of T from position i to position $i+m-1$ is equal to the pattern. The goal of the string matching problem is to determine whether the pattern occurs in the text. If it does, one either wants to find the position of the first occurrence or the positions of all occurrences. We will consider algorithms that output all occurrences of the pattern in the text; if we wanted only the first occurrence, we could simply stop the algorithm after finding the first occurrence.

It is natural to express the running-time of a string matching algorithm as a function of n and m. We will first consider a simple algorithm with running-time $O(nm)$. After that, we will see that it is possible to design an algorithm with running-time $O(n+m)$, i.e., a linear-time algorithm. If we considered string matching problems for strings over

possibly large alphabets, the size of the alphabet could also be an important parameter, but we will only be concerned with alphabets of constant size.

3.1 A first approach

A natural approach to the string matching problem is as follows: try every possible starting position in the text, and check (comparing symbol by symbol) whether the pattern occurs at that position. A possible pseudo-code for this algorithm is shown as Algorithm 1.

Algorithm 1: Naive String Matching

Input: pattern P, text T
Output: all occurrences of P in T

```
m ← P.length
n ← T.length
for i ← 0 to n − m do
    f ← true
    for j ← 0 to m − 1 do
        if T[i + j] ≠ P[j] then
            f ← false
            break
    if f = true then
        print "pattern occurs in position" i
```

The algorithm uses two for-loops. The outer for-loop tries all possible starting positions i, beginning with $i = 0$ (check if the pattern occurs at the beginning of T) and ending with $i = n - m$ (check if the pattern occurs at the end of T). The inner for-loop uses variable j to iterate through all positions in the pattern and checks for each symbol of the pattern whether it occurs at the respective position in the text. If one of the symbols does not match, the flag f is set to false and the algorithm exits the inner for-loop with a **break** statement. If all symbols match, the flag f will be true at the end of the inner for-loop, and the algorithm prints a message saying that the pattern occurs in position i.

It should be clear that the algorithm solves the string matching problem correctly. What about its running-time? The outer for-loop has $n - m + 1$ iterations. The number of iterations of the inner for-loop can be anything between 1 and m, depending on when the first non-matching symbol is found. In the worst case, the number of iterations of the inner for-loop in m. Therefore, the total number of steps made by the algorithm is at most proportional to $(n - m + 1)m$. As $n - m + 1 \leq n$, we can say that the algorithm has time complexity $O(nm)$.

On some inputs (for example, if the text consists of A's and the pattern consists of C's), the inner for-loop will always be exited in the first iteration, so the total running-time will be $O(n)$ for such inputs. On other inputs (for example, if the text and the pattern consist

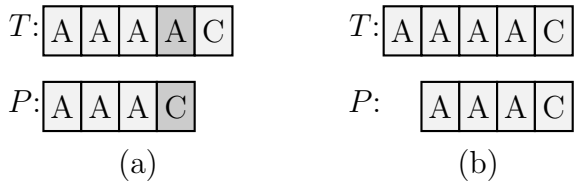


Figure 1: String matching example.

only of A's), however, the inner for-loop will always be executed m times, and the total running-time is indeed proportional to $(n - m + 1)m$. Furthermore, if $m \leq n/2$, this running-time is at least $nm/2$. Therefore, we cannot give a better worst-case bound than $O(nm)$ for this algorithm.

While this algorithm is perfectly fine for small inputs (say, a text of length 1,000 and a pattern of length 10), it can take a very long time on large inputs (e.g., a text of length 1,000,000 and a pattern of length 100,000). Therefore, we are interested in string matching algorithms that have a better time complexity.

3.2 The Knuth-Morris-Pratt algorithm

Now we will see how a string matching algorithm with linear time complexity can be obtained. Let us look at the string matching algorithm from the previous section. Does it perform steps that are not necessary? Consider the text $T = \mathsf{AAAAC}$ and the pattern $P = \mathsf{AAAC}$, for example. First, the algorithm will check whether P occurs at the beginning (position 0) of T. It sees that the three A's match, but then the C in the pattern does not match the fourth A in the string; see Figure 1(a). Next, the algorithm checks whether P occurs at position 1 of the pattern, see Figure 1(b). Here, it first checks whether there are A's in positions 1 and 2 of T. However, at this time we already *know* that there must be A's in these positions, because the first three symbols of P matched the first three symbols of T. If we could exploit this knowledge and avoid checking these symbols again, we could improve the running-time. This is the basic idea underlying the algorithm of Knuth, Morris and Pratt, which we will develop in the following.

Assume we have found that the first q symbols of P match T starting at position i, but the $(q + 1)$-th symbol does not match. Thus we know that P does not occur in T at position i. However, we know that the first q symbols of P match T at position i, and we want to exploit this information. In particular, we want to use this information to determine the next possible starting position (after i) at which P could possibly occur in T. For this purpose, we will use a table π (called the *prefix function*) for the pattern P, with the following meaning:

> For $q = 1, 2, \ldots, m$, $\pi[q]$ is the largest value of k such that $k < q$ and $P[0..k-1]$ is a suffix of $P[0..q-1]$.

For $k = 0$, $P[0..k-1]$ is the empty string, and the empty string is a suffix of every string.

Therefore, $\pi[q]$ is always well-defined and denotes the length of the longest proper suffix of $P[0..q-1]$ that is also a prefix of P.

Intuitively, the prefix function π helps us as follows: If we have found that q symbols of the pattern match the text from position i to position $j = i+q-1$, but there is a mismatch at position $j+1$ in the text, then the next possible match that we need to consider is the following: we consider the potential match where the pattern is aligned to the text in such a way that its first $\pi[q]$ symbols match positions $j-\pi[q]+1$ to j of the text, and the next thing we check is whether the next symbol of the pattern matches the symbol in position $j+1$ of the text. In this way we avoid comparing symbols of the text up to position j a second time. The pseudo-code for the resulting algorithm, called the KMP Algorithm, is shown as Algorithm 2.

Algorithm 2: KMP Algorithm

Input: pattern P, text T
Output: all occurrences of P in T

```
m ← P.length
n ← T.length
π ← Compute-Prefix-Function(P)
q ← 0
for i ← 0 to n − 1 do
    while q > 0 and P[q] ≠ T[i] do
        q ← π[q]
    if P[q] = T[i] then
        q ← q + 1
    if q = m then
        print "pattern occurs in position" i − m + 1
        q ← π[q]
```

First, the algorithm calls a subroutine `Compute-Prefix-Function` to compute the prefix function π for the pattern P; we will discuss this subroutine later on. It also initialises q to 0. The for-loop runs over all positions i of the text. At the beginning of an iteration of the for-loop, q represents the number of symbols at the beginning of the pattern that match the text up to position $i-1$. If $q > 0$ and the symbol $T[i]$ in the text does not match the next symbol $P[q]$ in the pattern, we know that this partial match (the first q symbols of the pattern matching $T[i-q..i-1]$) cannot be extended, and we try the next possible match by setting q to $\pi[q]$. This is repeated as long as $q > 0$ and $P[q] \neq T[i]$. The while-loop can end with $q = 0$ (which means that the next possible match is one starting in position i) or with $P[q] = T[i]$ (or both). If $P[q] = T[i]$, this means that the current partial match can be extended by one position, so we set q to $q+1$. If $q = m$, this means that we have found an occurrence of the pattern ending in position i of the text, and thus we output that we have found a match at position $i-m+1$. In this way, the algorithm finds and outputs all occurrences of the pattern in the text.

What is the running-time of the KMP algorithm? We will see later that the prefix function can be computed in time $O(m)$. The for-loop has n iterations. Each iteration executes a constant number of steps, except for the while-loop that could have several iterations. Therefore, apart from the time spent in the while-loop, the running-time is $O(m + n)$. To analyse the while-loop, we observe that each iteration of the while-loop reduces the value of q (since it sets q to $\pi[q]$). On the other hand, q is increased at most by one for each symbol of the text. Therefore, over the whole run of the algorithm, q is increased by one at most n times, and thus it can be decreased at most n times as well (since it always remains non-negative). Therefore, the total running-time spent in the while-loop (over the whole run of the algorithm) is $O(n)$. In summary, we obtain that the running-time of the algorithm is $O(n + m)$, hence linear.

3.2.1 Computing the prefix function

It remains to show how we can compute the prefix function π for a given pattern P of length m. A naive approach would take time $O(m^2)$ or even $O(m^3)$. The pseudo-code of a clever algorithm doing it in linear time $O(m)$ is shown as Algorithm 3.

Algorithm 3: Compute-Prefix-Function

Input: pattern P
Output: prefix function π for P

```
m ← P.length
π[1] ← 0
k ← 0
for q ← 2 to m do
    while k > 0 and P[k] ≠ P[q − 1] do
        k ← π[k]
    if P[k] = P[q − 1] then
        k ← k + 1
    π[q] ← k
```

Initially, the algorithm sets $\pi[1]$ to 0, which is always correct. In the beginning of each iteration of the for-loop, k is the length of the longest proper suffix of $P[0..q-2]$ that is also a prefix of P. If $P[k] = P[q-1]$, that suffix can be extended by one symbol and is still a prefix of P. In that case, k is set to $k + 1$. If $k > 0$ and $P[k] \neq P[q-1]$, the suffix cannot be extended by one symbol in this way, and we must try the next longest suffix instead; this is done via the while-loop, which sets k to $\pi[k]$ repeatedly as long as $k > 0$ and $P[k] \neq P[q-1]$. The important point is that the values $\pi[k]$ that are used in the while-loop have already been computed in previous iterations of the for-loop. This algorithm correctly computes the prefix-function. Its running-time is $O(m)$, and this can be shown in a similar way as in the analysis of the KMP Algorithm above.

```
HBA_HUMAN  GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKL
           G+ +VK+HGKKV  A+++++AH+D++ +++++LS+LH  KL
HBB_HUMAN  GNPKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKL
```

Figure 2: Alignment of human alpha globin and human beta globin [DEKM].

```
HBA_HUMAN  GSAQVKGHGKKVADALTNAVAHV---D--DMPNALSALSDLHAHKL
           ++ ++++H+ KV   + +A  ++          +L+ L+++H+ K
LGB2_LUPLU NNPELQAHAGKVFKLVYEAAIQLQVTGVVVTDATLKNLGSVHVSKG
```

Figure 3: Alignment of human alpha globin and leghaemoglobin from yellow lupin [DEKM].

4 Sequence alignment

This section is based on [DEKM, Chapter 2].

A fundamental task in sequence analysis is determining whether two given sequences are related. This is usually done by aligning the sequences (or parts thereof) and using the score of the alignment to determine whether the sequences are indeed related.

Figure 2 shows an alignment of fragments of human alpha and beta globin, showing a clear similarity. The middle line indicates identical positions with a letter and "similar" positions with a + sign. Figure 3 shows a structurally plausible alignment of the same fragment of human alpha globin to leghaemoglobin from yellow lupin. Here, two gaps (indicated by sequences of - signs) have been introduced into the human alpha globin in order to derive the alignment.

Sometimes, spurious high-scoring alignments can be found between proteins with completely different function and structure. Therefore, one has to be very careful when using alignment scores to infer sequence similarity.

4.1 The scoring model

When we compare DNA or protein sequences (which we model as strings), we are looking for evidence that they have diverged from a common ancestor by a process of mutation and selection. The basic mutations that we consider are *substitutions* (replacing a symbol in the sequence by another), *insertions*, and *deletions*. In an alignment, insertions and deletions correspond to *gaps*. If gaps (sequences of - symbols) have been inserted into a string, we refer to the original symbols as *real symbols* and the - symbols as *gap symbols*.

A (global) alignment of two given strings S and T is obtained by possibly inserting gaps into both strings in such a way that the resulting strings S' and T' have the same length, and then writing the resulting strings one above the other, so that each (real or gap) symbol in the first string is aligned with the corresponding (real or gap) symbol of the second string; see Figures 2 and 3 for examples.

We will assign a score to an alignment by adding up scores for each pair of aligned real symbols and scores for each gap. We will later discuss a probabilistic interpretation in which the score of an alignment corresponds to the logarithm of the relative likelihood

that the sequences are related, compared to being unrelated. This interpretation assumes that mutations at different positions (sites) of the sequences occur independently, where insertions or deletions of arbitrary length are treated as a single mutation. For DNA and protein sequences, this assumption seems to be a reasonable approximation.

The scoring of an alignment has two components: a *substitution matrix*, which assigns a score to each pair of aligned real symbols, and *gap penalties*, which assign a (negative) score to gaps depending on their length.

4.1.1 Substitution matrices

A substitution matrix specifies a score for each pair (x, y) of symbols. We want the score to reflect the probability of observing this pair of symbols in related sequences, compared to the probability of observing them in unrelated sequences. If we assume that any symbol c occurs with probability q_c, then the probability of observing x and y in unrelated strings is $q_x q_y$. Furthermore, assume that we know the probability p_{xy} of observing symbol x aligned with symbol y in related sequences. Then the ratio of these two probabilities, known as the *odds ratio*, is:

$$\frac{p_{xy}}{q_x q_y}$$

To obtain the odds ratio for two given sequences, we would have to multiply the odds ratios for each pair of aligned symbols (assuming independence between the different sites). Since we want an additive scoring function, we take the logarithm of the odds ratio, obtaining the *log-odds ratio*:

$$s(x, y) = \log \frac{p_{xy}}{q_x q_y}$$

The score of an alignment (without gaps) is then obtained by summing up the scores $s(x, y)$ for each pair (x, y) of aligned symbols. The scores $s(x, y)$ can be arranged in a matrix, so that the entry in position i, j of the matrix represents the score of aligning the i-th symbol with the j-th symbol of the alphabet. This matrix is called substitution matrix or score matrix. For amino acids, it is a 20×20 matrix. Popular examples of substitution matrices for protein sequences are the BLOSUM and PAM matrices. The BLOSUM50 matrix is shown in Figure 4; the entries have been scaled and rounded to integers for computational efficiency.

Using the BLOSUM50 matrix, an ungapped alignment of ADC and RNC would get a score of $s(\mathsf{A}, \mathsf{R}) + s(\mathsf{D}, \mathsf{N}) + s(\mathsf{C}, \mathsf{C}) = -2 + 2 + 13 = 13$, for example.

4.1.2 Gap penalties

The standard method of assigning scores to gaps is to define a gap penalty $\gamma(g)$ for gaps of length g. Popular choices are a linear score

$$\gamma(g) = -gd$$

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	5	-2	-1	-2	-1	-1	-1	0	-2	-1	-2	-1	-1	-3	-1	1	0	-3	-2	0
R	-2	7	-1	-2	-4	1	0	-3	0	-4	-3	3	-2	-3	-3	-1	-1	-3	-1	-3
N	-1	-1	7	2	-2	0	0	0	1	-3	-4	0	-2	-4	-2	1	0	-4	-2	-3
D	-2	-2	2	8	-4	0	2	-1	-1	-4	-4	-1	-4	-5	-1	0	-1	-5	-3	-4
C	-1	-4	-2	-4	13	-3	-3	-3	-3	-2	-2	-3	-2	-2	-4	-1	-1	-5	-3	-1
Q	-1	1	0	0	-3	7	2	-2	1	-3	-2	2	0	-4	-1	0	-1	-1	-1	-3
E	-1	0	0	2	-3	2	6	-3	0	-4	-3	1	-2	-3	-1	-1	-1	-3	-2	-3
G	0	-3	0	-1	-3	-2	-3	8	-2	-4	-4	-2	-3	-4	-2	0	-2	-3	-3	-4
H	-2	0	1	-1	-3	1	0	-2	10	-4	-3	0	-1	-1	-2	-1	-2	-3	2	-4
I	-1	-4	-3	-4	-2	-3	-4	-4	-4	5	2	-3	2	0	-3	-3	-1	-3	-1	4
L	-2	-3	-4	-4	-2	-2	-3	-4	-3	2	5	-3	3	1	-4	-3	-1	-2	-1	1
K	-1	3	0	-1	-3	2	1	-2	0	-3	-3	6	-2	-4	-1	0	-1	-3	-2	-3
M	-1	-2	-2	-4	-2	0	-2	-3	-1	2	3	-2	7	0	-3	-2	-1	-1	0	1
F	-3	-3	-4	-5	-2	-4	-3	-4	-1	0	1	-4	0	8	-4	-3	-2	1	4	-1
P	-1	-3	-2	-1	-4	-1	-1	-2	-2	-3	-4	-1	-3	-4	10	-1	-1	-4	-3	-3
S	1	-1	1	0	-1	0	-1	0	-1	-3	-3	0	-2	-3	-1	5	2	-4	-2	-2
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	2	5	-3	-2	0
W	-3	-3	-4	-5	-5	-1	-3	-3	-3	-3	-2	-3	-1	1	-4	-4	-3	15	2	-3
Y	-2	-1	-2	-3	-3	-1	-2	-3	2	-1	-1	-2	0	4	-3	-2	-2	2	8	-1
V	0	-3	-3	-4	-1	-3	-3	-4	-4	4	1	-3	1	-1	-3	-2	0	-3	-1	5

Figure 4: The BLOSUM50 matrix.

for some constant d, or an affine score

$$\gamma(g) = -d - (g-1)e$$

for constants d and e, where d is the *gap-open* penalty and e is the *gap-extension* penalty; usually, e is chosen smaller than d. For use with the BLOSUM50 substitution matrix, one could use $d = 8$ in the linear case and $d = 12$, $e = 2$ in the affine case, for example.

The alignment

```
AR---N
DRAACN
```

has a score of

$$s(\mathsf{A},\mathsf{D}) + s(\mathsf{R},\mathsf{R}) + \gamma(3) + s(\mathsf{N},\mathsf{N}) = -2 + 7 + \gamma(3) + 7 = 12 + \gamma(3).$$

With the gap penalties chosen as above, the score is $12 - 24 = -12$ for the linear gap penalty and $12 - 12 - 2 \cdot 2 = -4$ for the affine gap penalty.

4.2 Alignment algorithms

Now that we know how to score alignments, we would like to compute the highest-scoring alignment for two given strings; if that alignment has a large score, the sequences are likely

to be related, otherwise they are not. A first approach would be to compute all possible alignments of the two strings, compute the score for each of them, and then output the alignment with largest score. Such an approach is infeasible due to the potentially exponential number of possible alignments (which arises from the many different possibilities where gaps can be introduced). Therefore, we need a faster algorithm to compute optimal alignments. In this section, we will derive such algorithms using an algorithm design principle called *dynamic programming*. In dynamic programming, one computes optimal solutions for smaller subproblems and then combines these to obtain the optimal solution for the original problem. Usually, the solutions of the smaller problems are stored in a matrix, and each entry of the matrix is computed by looking up previously computed entries and combining them in the right way. In the following, we will discuss dynamic programming algorithms for the computation of optimal alignments.

4.2.1 Needleman-Wunsch: Global alignment with linear gap penalty

First, we consider the problem of computing the optimal alignment (where gaps are allowed) for two given strings X and Y for scoring functions that use an arbitrary substitution matrix and a linear gap penalty $\gamma(g) = -gd$. Let X be of length n and Y be of length m. The positions in X (in Y) are indexed from 0 to $n-1$ (to $m-1$). For example, this means that the symbols of X are $X[0]$, $X[1]$, ..., $X[n-1]$.

The basic idea of the Needleman-Wunsch algorithm is to compute optimal alignment scores for all pairs of strings $X[0..i-1]$ and $Y[0..j-1]$ for $0 \leq i \leq n$, $0 \leq j \leq m$. In other words, we compute optimal alignments between prefixes of X and prefixes of Y. Let $F(i,j)$ be the score of an optimal alignment between $X[0..i-1]$ and $Y[0..j-1]$, where we take $X[0..-1]$ and $Y[0..-1]$ to be the empty string. Then the following equations hold:

$$\begin{aligned} F(0,0) &= 0 \\ F(0,j) &= -jd \\ F(i,0) &= -id \\ F(i,j) &= \max \begin{cases} F(i-1,j-1) + s(X[i-1], Y[j-1]) \\ F(i-1,j) - d \\ F(i,j-1) - d \end{cases} \end{aligned}$$

The first three equations are straightforward: aligning two empty strings has score 0, and aligning a string of length i to the empty string has score $-id$ (and similarly for length j). The last equation, which applies to the cases where $1 \leq i \leq n$ and $1 \leq j \leq m$, is the interesting one. It says that the largest possible score for aligning $X[0..i-1]$ and $Y[0..j-1]$ is obtained in one of three ways:

- $X[i-1]$ is aligned to $Y[j-1]$, and to the left of this aligned pair of symbols we use an optimal alignment of $X[0..i-2]$ and $Y[0..j-2]$.
- $X[i-1]$ is aligned with a gap symbol inserted into Y, and to the left of this we use an optimal alignment of $X[0..i-2]$ and $Y[0..j-1]$.

Algorithm 4: Global alignment with linear gap penalty (Needleman-Wunsch)

Input: strings X and Y, substitution matrix s, gap cost d
Output: score of optimal gapped alignment between X and Y

```
n ← X.length
m ← Y.length
for i = 0 to n do
    for j = 0 to m do
        if i = 0 and j = 0 then
            F(i,j) ← 0
        else if i = 0 then
            F(i,j) ← −jd
            P(i,j) ← L
        else if j = 0 then
            F(i,j) ← −id
            P(i,j) ← T
        else
            F(i,j) ← max{F(i−1,j−1) + s(X[i−1],Y[j−1]), F(i−1,j) − d,
                    F(i,j−1) − d}
            set P(i,j) to D, T, or L depending on which of the three terms in the
                    previous line yielded the maximum
return F(n,m)
```

- $Y[j-1]$ is aligned with a gap symbol inserted into X, and to the left of this we use an optimal alignment of $X[0..i-1]$ and $Y[0..j-2]$.

As the three cases cover all possibilities for the last position in the alignment of $X[0..i-1]$ and $Y[0..j-1]$, the best of the three scores must be the optimal score. Furthermore, if the entries of the matrix F are computed in order of increasing row and column indices (e.g., row by row), the three entries on the right-hand side of the equation have already been computed when $F(i,j)$ is computed. Therefore, we can compute all entries of F in time $O(mn)$. The entry $F(n,m)$ gives us the score of the optimal alignment.

To obtain the actual alignment (and not only its score), we remember for each entry $F(i,j)$ which of the three expressions on the right-hand side of the equation above has determined the value of $F(i,j)$. We could do this in a second matrix P, whose entries are:

$$P(i,j) = \begin{cases} \text{D} & \text{if } F(i,j) \text{ is set to } F(i-1,j-1) + s(X[i-1],Y[j-1]) \\ \text{T} & \text{if } F(i,j) \text{ is set to } F(i-1,j) - d \\ \text{L} & \text{if } F(i,j) \text{ is set to } F(i,j-1) - d \end{cases}$$

Here, D stands for *diagonal*, T for *top*, and L for *left*, indicating from which other entry of F the value of $F(i,j)$ was obtained. Consequently, $P(i,j) = \text{D}$ means that, in the optimal alignment of $X[0..i-1]$ and $Y[0..j-1]$, $X[i-1]$ and $Y[j-1]$ are aligned to each other.

$P(i,j) = \mathrm{T}$ means that $X[i-1]$ is aligned to a gap symbol. $P(i,j) = \mathrm{L}$ means that $Y[j-1]$ is aligned to a gap symbol. In the end, we can use the P matrix to construct the optimal alignment as follows: We start by looking up $P(n,m)$, which tells us the symbols in the last position of the optimal alignment. Depending on these symbols, we continue with either $P(n-1,m-1)$ or $P(n,m-1)$ or $P(n-1,m)$, and that entry will tell us the symbols in the next-to-last position of the optimal alignment. We continue in this way until we arrive at $P(0,0)$; by then we have constructed the optimal alignment. This method of constructing the optimal alignment from the P matrix is called *traceback*.

A pseudo-code implementation of the algorithm computing the F and P matrices is shown as Algorithm 4. It is easy to see that the running-time of the algorithm is $O(mn)$. Pseudo-code for constructing the optimal alignment from the P matrix (and returning it in form of the strings X' and Y' that are obtained from X and Y after the insertion of gaps) is shown as Algorithm 5. This has running-time $O(m+n)$, as it executes a constant number of steps for each position of the constructed alignment. So the total running-time of the Needleman-Wunsch algorithm is $O(mn)$. Usually, n and m are approximately the same, hence one also says that the algorithm has time complexity $O(n^2)$. As the algorithm stores the matrices F and P, it has a memory requirement (space complexity) of $O(n^2)$.

4.2.2 A linear-space version of the Needleman-Wunsch algorithm

For very long input sequences, the quadratic space complexity of the Needleman-Wunsch algorithm can become a problem. Therefore, we are interested in alignment algorithms that require only linear space. First, we notice that we can modify the Needleman-Wunsch algorithm (Algorithm 4) so that it does not store the whole matrix F, but only the current row and the previous one. This suffices for the computation of $F(n,m)$, as each value $F(i,j)$ depends only on values in the same row or in the previous row of the matrix. This modified version of the algorithm needs only linear space. Hence, if we only want to know the score of an optimal alignment (but not the alignment itself), we already have a linear-space algorithm. To construct the alignment, however, we need the whole P matrix, and thus it looks as if we cannot avoid using quadratic space. Fortunately, there is a way around this. The main idea needed for a linear-space algorithm computing the optimal alignment is to apply the divide-and-conquer principle: divide the problem into two smaller problems, solve each of them separately, and combine the solutions into a solution for the whole problem. More concretely, we proceed as follows:

- If n is small (bounded by a constant), we execute the original Needleman-Wunsch algorithm to compute the optimal alignment. Since n is $O(1)$, the running time is $O(m)$ and thus linear in this case.

- Otherwise, we consider the middle row of the matrix, row $u = n/2$. First, we determine a column v such that the optimal alignment for the whole sequences X and Y contains an optimal alignment of $X[0..u-1]$ with $Y[0..v-1]$; this is the case if the traceback (Algorithm 5) in the original Needleman-Wunsch algorithm visits $P(u,v)$, but we will determine v without storing the whole P matrix.

Algorithm 5: Constructing the optimal alignment from the matrix P

Input: strings X and Y, matrix P
Output: optimal gapped alignment (X', Y') between X and Y

```
n ← X.length
m ← Y.length
X' ← empty string
Y' ← empty string
i ← n
j ← m
while i + j > 0 do
    if P(i, j) = D then
        insert X[i − 1] in the beginning of X'
        insert Y[j − 1] in the beginning of Y'
        i ← i − 1
        j ← j − 1
    else if P(i, j) = T then
        insert X[i − 1] in the beginning of X'
        insert - in the beginning of Y'
        i ← i − 1
    else
        insert - in the beginning of X'
        insert Y[j − 1] in the beginning of Y'
        j ← j − 1
return (X', Y')
```

- Then, we recursively compute optimal alignments of $X[0..u-1]$ with $Y[0..v-1]$ and of $X[u..n-1]$ with $Y[v..m-1]$. The optimal alignment of X with Y is the concatenation of these two optimal alignments.

To determine v, we compute a value $V(i,j)$ for each $i \geq u$ and $0 \leq j \leq m$. We want to have $V(i,j) = t$ if the traceback from $P(i,j)$ would visit $P(u,t)$. $V(i,j)$ can be computed as follows:

$$V(i,j) = \begin{cases} j & \text{if } i = u, \\ V(i',j') & \text{if } i > u \text{ and } F(i,j) \text{ is determined from } F(i',j') \end{cases}$$

The entry $V(n,m)$ gives us the desired value v. We can compute $F(n,m)$ and $V(n,m)$ in such a way that we always store only the current row and the previous row of these two matrices; hence, the computation needs only linear space $O(m)$. Thus, the total space requirement of the algorithm is $O(m)$.

What is the running-time of this approach? If we denote the running-time for input strings of length m and n by $T(n, m)$, we obtain the following recurrence:

$$T(n,m) \leq \begin{cases} O(m) & \text{if } n \text{ is bounded by a constant} \\ hnm + T(n/2, v) + T(n/2, m-v) & \text{otherwise} \end{cases}$$

Here, the term hnm (with a suitable constant h) represents the time needed to compute v as well as the time for combining the two optimal alignments obtained from the recursive calls. The recursive calls are for a problem with one string of length $n/2$ and the other of length v or $m - v$. It is not difficult to prove by induction that the above inequality implies that $T(n, m)$ is $O(nm)$. Hence, the time complexity of the algorithm is larger than that of the original Needleman-Wunsch algorithm only by a constant factor. In summary, we have obtained an algorithm constructing optimal global alignments using linear space $O(n + m)$ and with time complexity $O(nm)$.

4.2.3 Global alignments with general gap penalties

The sequence alignment algorithms we have discussed so far were designed for the case of linear gap penalties. However, it is not difficult to generalise the basic Needleman-Wunsch algorithm to the case of arbitrary gap penalties $\gamma(g)$. Here, we assume that the gap penalty function γ satisfies the natural property $\gamma(g_1) + \gamma(g_2) \leq \gamma(g_1 + g_2)$, for all $g_1, g_2 \geq 1$. Intuitively, this means that a gap does not get cheaper if we treat it as two consecutive smaller gaps.

We simply use the following main equation to determine the $F(i, j)$ values (for $i, j > 0$):

$$F(i,j) = \max \begin{cases} F(i-1, j-1) + s(X[i-1], Y[j-1]) \\ \max\{F(k,j) + \gamma(i-k) \mid 0 \leq k \leq i-1\} \\ \max\{F(i,k) + \gamma(j-k) \mid 0 \leq k \leq j-1\} \end{cases}$$

Besides, we set $F(0, 0) = 0$, $F(i, 0) = \gamma(i)$, and $F(0, j) = \gamma(j)$. Again, the basic idea is to consider all possible cases for the last position in the optimal alignment between $X[0..i-1]$ and $Y[0..j-1]$. If $X[i-1]$ and $Y[j-1]$ are aligned, the score is $F(i-1, j-1) + s(X[i-1], Y[j-1])$ as before. If a gap in Y is aligned with $X[i-1]$, we consider all possibilities k for the first symbol in X that is aligned to that gap; each possibility gives a candidate score of $F(k, j) + \gamma(i - k)$, and we take the largest of these. The third expression on the right-hand side of the equation for $F(i, j)$ deals with the case that a gap in X is aligned with $Y[j-1]$ in an analogous way. All possible cases are covered, and so $F(i, j)$ gives the optimal score of an alignment of $X[0..i-1]$ and $Y[0..j-1]$. The optimal alignment score for X and Y is $F(n, m)$. By storing traceback information (e.g., by using a separate matrix P to remember for each $F(i, j)$ from which $F(i', j')$ its value was derived) we can construct the optimal alignment.

Since we have to try all possible values of k to compute $F(i, j)$, the total running-time is $O(n^3)$ instead of $O(n^2)$ as before (here we assume that m and n are of similar length). Furthermore, the method we used to convert the Needleman-Wunsch algorithm into a

linear-space algorithm cannot be applied directly to this algorithm as it needs to look up values $F(k, j)$ and $F(i, k)$ from all previous rows and columns.

For some special cases of gap penalties, we can still get a quadratic alignment algorithm. The next section shows this for the case of affine gap penalties.

4.2.4 Global alignments with affine gap penalties

Consider a scoring scheme with affine gap penalties $\gamma(g) = -d-(g-1)e$. To get an $O(nm)$ algorithm for this case, we keep three matrices instead of the single matrix F:

- $M(i, j)$ is the largest score of an alignment between $X[0..i-1]$ and $Y[0..j-1]$ for which $X[i-1]$ is aligned to $Y[j-1]$.
- $I_X(i, j)$ is the largest score of an alignment between $X[0..i-1]$ and $Y[0..j-1]$ for which $X[i-1]$ is aligned to a gap.
- $I_Y(i, j)$ is the largest score of an alignment between $X[0..i-1]$ and $Y[0..j-1]$ for which $Y[j-1]$ is aligned to a gap.

If we have computed these values, the maximum of the three values $M(n, m)$, $I_X(n, m)$ and $I_Y(n, m)$ gives us the score of an optimal alignment. The following equations allow us to compute the three matrices:

$$
\begin{aligned}
M(i,j) &= \max \begin{cases} M(i-1,j-1) + s(X[i-1], Y[j-1]) \\ I_X(i-1,j-1) + s(X[i-1], Y[j-1]) \\ I_Y(i-1,j-1) + s(X[i-1], Y[j-1]) \end{cases} \\
I_X(i,j) &= \max \begin{cases} M(i-1,j) - d \\ I_X(i-1,j) - e \end{cases} \\
I_Y(i,j) &= \max \begin{cases} M(i,j-1) - d \\ I_Y(i,j-1) - e \end{cases}
\end{aligned}
$$

These equations assume that we are not interested in alignments where an insertion is followed directly by a deletion, or vice versa. (If we wanted such alignments, we would have to modify the equations slightly.) Furthermore, we do not discuss the boundary conditions (i.e., the matrix entries for $i = 0$ or $j = 0$) here; these need to be defined appropriately, but there is no difficulty.

The equation for $M(i, j)$ says that the optimal score is obtained by taking $s(X[i-1], Y[j-1])$ and adding the cost for the optimal alignment of $X[0..i-2]$ and $Y[0..j-2]$ (which can be one of the three possible types). The equation for $I_X(i, j)$ considers two cases: either $X[i-1]$ is the first symbol aligned to the gap in Y, so we take $M(i-1, j)$ and add $-d$, the gap-open penalty; or $X[i-1]$ is not the first symbol aligned to the gap in Y, so we take $I_X(i-1, j)$ and add $-e$, the gap-extension penalty.

The equation for $I_Y(i, j)$ can be explained similarly. The equations deal with all possible cases and thus the algorithm indeed gives the optimal alignment. (To construct the alignment, we store traceback pointers for each value $M(i, j)$, $I_X(i, j)$ and $I_Y(i, j)$ that tell us which case gave the maximum in the computation of that entry.)

4.2.5 Local alignments: Smith-Waterman

So far we have considered the computation of global alignments for two given sequences X and Y. In practice, one is often interested in identifying a substring X' of X and a substring Y' of Y such that X' and Y' have a high-scoring alignment. Such alignments are called *local alignments*. This issue arises for example when it is suspected that two protein sequences may share a common domain, or when comparing extended sections of genomic DNA sequence. It is also useful for detecting similarity in two highly diverged sequences; often, part of a sequence may be under strong enough selection to preserve detectable similarity, while the rest accumulates noise via mutations and is no longer alignable.

We want to solve the problem of computing the best local alignment for two given strings X and Y. This means we want to compute substrings X' of X and Y' of Y such that the largest-scoring alignment of X' and Y' maximises the score, among all choices of X' and Y'. We assume that the score of an alignment is computed using a substitution matrix and linear gap penalties. It turns out that this problem can then be solved with dynamic programming, using a slight variation of the Needleman-Wunsch algorithm, called the Smith-Waterman algorithm.

Let n be the length of X, and m be the length of Y. For $0 \leq i \leq n$, let $X_i = X[0..i-1]$ denote the prefix of X of length i; note that X_0 is the empty string. Define Y_j for $0 \leq j \leq m$ similarly. We define a dynamic programming matrix F whose entries have the following interpretation:

> For $0 \leq i \leq n$ and $0 \leq j \leq m$, $F(i,j)$ is the largest score of an alignment between any suffix X' of X_i and any suffix Y' of Y_j.

If we can compute such values, then the maximum value $F(i,j)$ in the matrix gives us the score of the best local alignment, and we can use traceback from position (i,j) to construct the alignment.

The values $F(i,j)$ can be computed using the following equations:

$$\begin{aligned} F(0,0) &= 0 \\ F(i,0) &= 0 \\ F(0,j) &= 0 \\ F(i,j) &= \max \begin{cases} 0 \\ F(i-1,j-1) + s(X[i-1], Y[j-1]) \\ F(i-1,j) - d \\ F(i,j-1) - d \end{cases} \end{aligned}$$

Compared to the Needleman-Wunsch algorithm, the difference is that we do not assign negative values to any matrix entry; if an entry were to be negative, we replace it by 0. This is meaningful because $F(i,j)$ is the score of the best alignment of a suffix X' of X_i and a suffix Y' of Y_j, and we can always choose X' and Y' to be the empty string (giving an alignment with score 0). Note that we can initialise the first row and column of F with 0 for this reason.

The equation for $F(i,j)$ can be justified as before by considering the possibilities for the last position in the optimal alignment of a suffix X' of X_i with a suffix Y' of Y_j:

- The optimal alignment is obtained when X' and Y' are the empty string, giving $F(i,j) = 0$.
- The optimal alignment is obtained when $X[i-1]$ is aligned with $Y[j-1]$ in the last position, and to the left of it there is an optimal alignment of a suffix of X_{i-1} and a suffix of Y_{j-1}. This possibility gives $F(i,j) = F(i-1,j-1) + s(X[i-1], Y[j-1])$.
- The optimal alignment is obtained when $X[i-1]$ is aligned with a gap in Y in the last position, and to the left of it there is an optimal alignment of a suffix of X_{i-1} and a suffix of Y_j. This leads to $F(i,j) = F(i-1,j) - d$.
- The optimal alignment is obtained when $Y[j-1]$ is aligned with a gap in X in the last position, and to the left of it there is an optimal alignment of a suffix of X_i and a suffix of Y_{j-1}. This leads to $F(i,j) = F(i,j-1) - d$.

The four possibilities cover all cases, and thus the largest of the four values is the optimal score of an alignment of any suffix of X_i with any suffix of Y_j.

We can compute all values $F(i,j)$ using the equations above by filling the matrix row by row, for example. We can also create a second matrix P in which $P(i,j)$ tells us which of the four possible expressions gave the maximum in the computation of $F(i,j)$. In the end, we find the largest entry of the matrix F. Assume that $F(i_0,j_0)$ is the largest entry. This shows that the score of the best local alignment is $F(i_0,j_0)$, and that this score is achieved by an alignment of a suffix of X_{i_0} with a suffix of Y_{j_0}. To obtain the alignment itself, we use the P matrix for a traceback starting at $P(i_0,j_0)$ and ending at the first position (i_1,j_1) with $F(i_1,j_1) = 0$.

As an example, assume that we want to find the best local alignment for protein sequences PAWHEAE and HEAGAWGHEE, using the BLOSUM50 substitution matrix and linear gap penalties with $d = 8$ as the scoring model. The resulting F matrix is shown in Figure 5. For non-zero entries, the arrows indicate the contents of the P matrix; an arrow pointing from $F(i,j)$ to $F(i-1,j-1)$, for example, indicates that in the computation of $F(i,j)$ the maximum was achieved by $F(i-1,j-1) + s(X[i-1], Y[j-1])$. Entries with value 0 do not have an arrow. The largest value in the matrix is $F(5,9) = 28$. The traceback from there (indicated by the bold arrows) yields the best local alignment:

```
AWGHE
AW-HE
```

The score of that alignment is 28.

Note that the algorithm could be adapted to affine gap penalties or general gap penalties in a similar fashion as in the case of global alignments.

There are some subtle issues concerning suitable scoring models for local alignments. Essentially, it is required that the expected score for the alignment of a pair of random

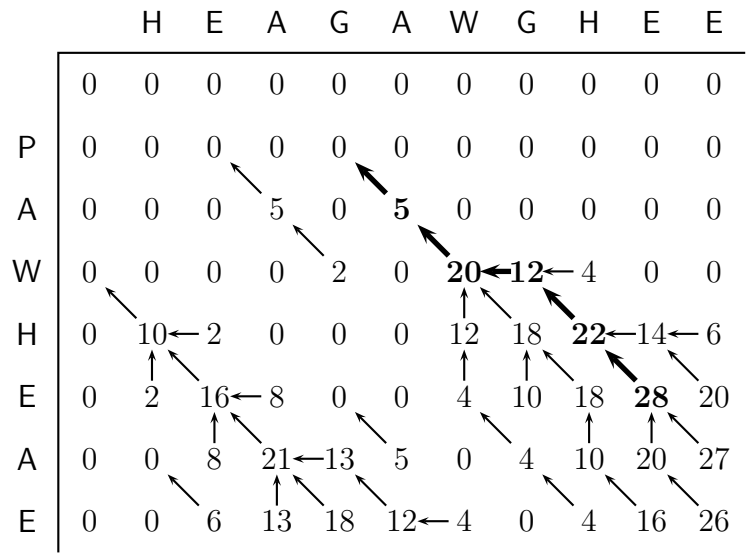


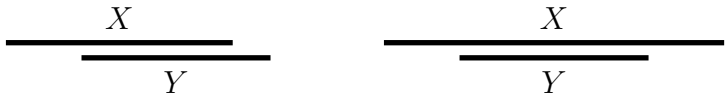
Figure 5: Dynamic programming matrix for local alignment.

symbols be negative. If this is not the case, longer alignments will tend to have larger scores than shorter ones, even if they are not biologically meaningful, and so the algorithm is likely to simply produce a global or nearly global alignment as the best local alignment. Therefore, one has to be careful when choosing a scoring model for the computation of local alignments.

4.2.6 Other variants

We have seen dynamic programming algorithms that compute highest-scoring global alignments and local alignments. In fact, minor variations of this approach can be used to solve a multitude of other alignment problems, for example:

- Repeated matches: Here, we want to find many different local alignments of parts of one sequence with non-overlapping parts of the other sequence. As an example, we might be interested in finding many copies of a repeated domain or motif in a protein.
- Overlap matches: This type of search is appropriate if we expect that one sequence contains the other, or that they overlap. This often occurs when comparing fragments of genomic DNA sequence to each other, or to larger chromosomal sequences. Here, we essentially want a global alignment, but we do not penalise overhanging ends. This means that we are interested in configurations like the following:



In many such cases, a minor adaptation of the equations used for the computation of the F matrix suffices to get an algorithm to compute optimal alignments of the desired type.

We refer to [DEKM] for details.

4.3 Heuristic alignment algorithms

So far we have discussed algorithms that compute global or local alignments using dynamic programming. These algorithms were *exact* algorithms, in the sense that they are guaranteed to output an alignment of largest score. The running-time of these algorithms was $O(nm)$ (or even $O(nm(n+m))$ in the case of general gap penalties). In this section, we briefly consider *heuristic* algorithms for computing alignments. These are algorithms that run faster than the exact methods, but can no longer guarantee to find optimal solutions. Nevertheless, the results they produce are often good enough to make them useful in practice.

If one studies a new sequence and wants to check an existing database of known sequences for high-scoring local alignments with the new sequence, the use of exact methods leads to a running-time that is proportional to the product of the size of the database and the length of the new sequence; this may mean that one has to wait several hours for the results. In such situations, fast heuristic methods are often used instead of the exact methods.

Two of the best-known heuristic alignment algorithms are BLAST and FASTA. The BLAST (Basic Local Alignment Search Tool) package is a collection of programs for comparing gene and protein sequences against others in public databases. It is available here:

`http://www.ncbi.nlm.nih.gov/BLAST/`

The idea behind BLAST is that good local alignments are likely to include short stretches of identities or very high scoring matches. Therefore, BLAST initially looks for such short stretches and uses them as 'seeds' that are then extended in search of a good local alignment.

BLAST first makes a list of all 'neighbourhood words' of a fixed length (by default, 3 for protein sequences, 11 for nucleic acids) that would match the query sequences somewhere with a score higher than a threshold. It then scans through the database, and whenever it finds a word in this set, it starts a 'hit extension' process to extend the possible match in both directions.

Initial versions of BLAST produced only alignments without gaps, but more recent versions can also compute gapped alignments.

FASTA (pronounced 'fast-aye') is another widely used heuristic sequence searching package. It is available here:

`http://fasta.bioch.virginia.edu/`

FASTA uses a multistep approach to finding high-scoring local alignments between two sequences: it starts from exact short word matches, finds maximal scoring ungapped extensions, and finally combines them into gapped alignments. The first step uses a lookup table to locate all identically matching words of length *ktup* between the two sequences

- $\Pr[A]$: probability of event A
- $\Pr[A, B]$: probability of event A and event B occurring together
- $\Pr[A \mid B] = \dfrac{\Pr[A, B]}{\Pr[B]}$: conditional probability of event A, assuming that B has occurred
- $\Pr[A, B] = \Pr[A \mid B] \cdot \Pr[B]$
- Bayes' Theorem:
$$\Pr[A \mid B] = \frac{\Pr[B \mid A] \cdot \Pr[A]}{\Pr[B]}$$
- Marginal probability: $\Pr[A] = \sum_i \Pr[A, B_i] = \sum_i \Pr[A \mid B_i] \cdot \Pr[B_i]$, where the B_i are disjoint events covering all possible outcomes

Figure 6: Some probability theory.

(where *ktup* is a parameter that is typically 1 or 2 for protein sequences and 4 or 6 for DNA). It then looks for diagonals (as in the dynamic programming matrix F) with many such matches. The best diagonals are then pursued further in the second step, which extends the short matches into maximal scoring ungapped regions (possibly joining several short matches). In the last step, FASTA attempts to combine several of these ungapped regions into gapped regions. The highest-scoring gapped alignments are then realigned using a modified dynamic programming algorithm.

4.4 Significance of scores

By now we know how to compute high-scoring local or global alignments between two given sequences. Now we are interested in assessing the significance of the score of an alignment. How do we know if the produced alignment is a biologically meaningful alignment giving evidence for a homology, or just the best alignment between two entirely unrelated sequences?

We will discuss two approaches to this issue, both based on statistical considerations. Some basic notations and laws of probability theory are shown in Figure 6.

4.4.1 The Bayesian approach: Model comparison

We consider two probabilistic models for generating an alignment: a model M (match model) for alignments of related sequences, and a model R (random model) for alignments of unrelated sequences.

In the following, log refers to the natural logarithm (logarithm to base e). We ignore gaps in this discussion, i.e., we consider only ungapped alignments.

Let X and Y be the aligned sequences, both of length n, and let $X = x_1x_2\cdots x_n$ and $Y = y_1y_2\cdots y_n$. For the model R, we assume that each symbol c has a probability q_c of appearing in any position of any of the two sequences. This means that each x_i and each y_i is equal to c with probability q_c. The probability of observing X and Y in the model R is therefore the product of all q_{x_i} and q_{y_i}:

$$\Pr[X, Y \mid R] = \prod_{i=1}^{n} q_{x_i} q_{y_i}$$

In the model M for matching sequences, we assume that for every pair of symbols c and d, there is a probability p_{cd} of observing c aligned to d. Therefore, the probability of observing X aligned to Y for related sequences is the product of all $p_{x_iy_i}$ for $i = 1, \ldots, n$:

$$\Pr[X, Y \mid M] = \prod_{i=1}^{n} p_{x_iy_i}$$

When we introduced scoring models for alignments, we already mentioned that these models are motivated by the log-odds ratio S:

$$S = \log \frac{\Pr[X, Y \mid M]}{\Pr[X, Y \mid R]} = \log \frac{\prod_{i=1}^{n} p_{x_iy_i}}{\prod_{i=1}^{n} q_{x_i} q_{y_i}} = \sum_{i=1}^{n} \log \frac{p_{x_iy_i}}{q_{x_i} q_{y_i}}$$

This is why we have defined the score $s(x, y)$ for aligning symbols x and y as $s(x, y) = \log \frac{p_{xy}}{q_x q_y}$.

Intuitively, what we are really interested in is the probability that X and Y are related. This probability can be written as $\Pr[M \mid X, Y]$. Furthermore, we assume that there is an *a priori* probability $\Pr[M]$ of observing two related sequences and $\Pr[R]$ of observing two unrelated sequences. We can then calculate $\Pr[M \mid X, Y]$, using Bayes' Theorem:

$$\begin{aligned}
\Pr[M \mid X, Y] &= \frac{\Pr[X, Y \mid M] \cdot \Pr[M]}{\Pr[X, Y]} \\
&= \frac{\Pr[X, Y \mid M] \cdot \Pr[M]}{\Pr[X, Y \mid M] \Pr[M] + \Pr[X, Y \mid R] \Pr[R]} \\
&= \frac{\Pr[X, Y \mid M] \cdot \Pr[M] / (\Pr[X, Y \mid R] \cdot \Pr[R])}{1 + \Pr[X, Y \mid M] \Pr[M] / (\Pr[X, Y \mid R] \Pr[R])}
\end{aligned}$$

Note that $S = \log \frac{\Pr[X,Y|M]}{\Pr[X,Y|R]}$ implies:

$$\log \frac{\Pr[X, Y \mid M] \cdot \Pr[M]}{\Pr[X, Y \mid R] \cdot \Pr[R]} = S + \log \frac{\Pr[M]}{\Pr[R]}$$

If we let $S' = S + \log \frac{\Pr[M]}{\Pr[R]}$, we get:

$$\Pr[M \mid X, Y] = \frac{e^{S'}}{1 + e^{S'}} = \sigma(S')$$

with $\sigma(x) = \frac{e^x}{1+e^x}$, known as the *logistic* function. Note that $\sigma(S')$ is 0.5 for $S' = 0$ and greater than 0.5 for $S' > 0$.

The above discussion shows that it is meaningful to use a score of $S' = S + \log \frac{\Pr[M]}{\Pr[R]}$ for an alignment, where S is the score defined via the log-odds ratio. Then we can in principle compare S' with 0 to indicate whether the two sequences X and Y are related.

As an example, assume that we are looking for alignments of a query string X to strings in a database containing N strings. If we assume that X will match only one of the N strings in the database, this corresponds to $\Pr[M] = 1/N$ and $\Pr[R] = 1 - 1/N$. In this case, $\log \frac{\Pr[M]}{\Pr[R]} = \log \frac{1/N}{1-1/N} \approx \log \frac{1}{N} = -\log N$. Hence, we should *subtract* $\log N$ from the log-odds score of each alignment between X and a database string. Equivalently, only alignments with a log-odds score of more than $\log N$ should be considered significant.

4.4.2 Classical approach: Extreme value distribution

A second approach to assessing the significance of alignment scores is based on a more classical statistical framework. We sketch this approach here only briefly; for details, see [DEKM].

Again, consider the situation that we compute scores for alignments of a query string X with strings in a database of N strings. Let M_N denote the random variable giving the maximum of the alignment scores of X with N random strings. Assume that the optimal alignment of X with a string Y in the database has a score of S. If the probability that M_N is greater or equal to S is small (e.g., at most 0.05 or 0.01), we can conclude that it is unlikely that the sequences X and Y are actually unrelated. In certain cases, the distribution of M_N can be approximated using a so-called *extreme value distribution*:

$$\Pr[M_N \leq x] \simeq \exp(-KNe^{\lambda(x-\mu)})$$

where K, λ are constants and μ is the expected score of an alignment of X with a random string. This formula can then be used to calculate the probability that M_N is greater than S.

4.5 Deriving score parameters from alignment data

The basis of our scoring models is the log-odds ratio, which assigns a score of $s(x, y) = \log \frac{p_{xy}}{q_x q_y}$ to the alignment of symbol x with symbol y. To compute $s(x, y)$, we need to know the values p_{xy} and q_x and q_y. How can we estimate these values? First, we could take a number of known sequences and use the frequency of each symbol x in these sequences as an estimate of q_x. Furthermore, we could take a number of known alignments of related sequences, and use the frequency of each aligned pair of symbols (x, y) in these alignments as an estimate of p_{xy}. Unfortunately, there are problems with this approach. First, it is difficult to obtain a representative sample of sequences and alignments of pairs of related sequences to estimate these parameters. Second, related sequences will lead to very different values of p_{xy} depending on whether the sequences have diverged recently or a long

time ago. In fact, it may be appropriate to use different scoring models depending on the expected divergence of the sequences we are comparing.

4.5.1 PAM matrices

Dayhoff, Schwartz and Orcutt have defined PAM (Point Accepted Mutations) matrices in 1978. They obtain substitution data from alignments between very similar proteins, and then extrapolate this information to longer evolutionary distances. They use alignments of neighbouring protein sequences in a phylogenetic tree to estimate the probabilities $\Pr[b \mid a, 1]$ that a is substituted by b after divergence time 1 (where divergence time 1 corresponds to an expected number of substitutions of 1%). From these one can calculate $\Pr[b \mid a, t]$, $t = 1, 2, \ldots$, the probability that a is substituted by b after divergence time t. For each t, the scores $s(a, b \mid t)$ are defined by:

$$s(a, b \mid t) = \log \frac{\Pr[b \mid a, t] q_a}{q_a q_b} = \log \frac{\Pr[b \mid a, t]}{q_b}$$

The resulting substitution matrices are called PAMt matrices. The most widely used matrix is PAM250 (which is scaled by $3/\log 2$).

PAM matrices have been widely used, but they also have some weaknesses. One problem is that the scores for larger values of t are calculated from those for $t = 1$ without taking into account the differences between short time substitutions and long term ones. For example, the former are dominated by amino acid substitutions that arise from single base changes in codon triples, whereas the latter show all kinds of codon changes.

4.5.2 BLOSUM matrices

After the introduction of PAM matrices, databases with multiple alignments of more distantly related proteins have been made, and these can be used to derive score matrices more directly. One such set of matrices are the BLOSUM matrices introduced by Henikoff and Henikoff in 1992. These matrices were derived from a set of aligned, ungapped regions from protein families called the BLOCKS database. The sequences from each block were clustered, putting two sequences in the same cluster when the percentage of identical residues exceeded L%. Then the frequencies A_{ab} of observing residue a in one cluster aligned to b in another cluster were calculated (normalised by the sizes of the clusters). Then q_a and p_{ab} were estimated using

$$q_a = \frac{\sum_b A_{ab}}{\sum_{cd} A_{cd}}$$

and

$$p_{ab} = \frac{A_{ab}}{\sum_{cd} A_{cd}}.$$

Then scores were calculated as $s(a, b) = \log \frac{p_{ab}}{q_a q_b}$, scaled, and rounded to integers. Two popular BLOSUM matrices are BLOSUM50 for gapped alignments and BLOSUM62 for

ungapped alignments. The BLOSUM50 matrix was obtained for $L = 50\%$ and scaled by $3/\log 2$, the BLOSUM62 matrix for $L = 62\%$ and scaled by $2/\log 2$. Note that lower values of L correspond to longer evolutionary time, and are applicable for more distant searches.

4.6 The power of DNA sequence comparison

In this section, we briefly discuss two historical examples where sequence alignments led to important discoveries. The examples are taken from Section 6.1 of [JP].

After a new gene is found, biologists usually have no idea about its function. A common approach to inferring the function of a newly sequenced gene is to use alignment algorithms to find similarities with genes whose function is known. A biological discovery made through a similarity search happened in 1984 when scientists compared the newly discovered cancer-causing ν-sis oncogene with known genes. To their astonishment, the cancer-causing gene matched a normal gene involved in growth and development, called platelet-derived growth factor (PDGF). After discovering this similarity, scientists became suspicious that cancer might be caused by a normal growth gene being switched on at the wrong time.

Another example of a successful similarity search was the discovery of the cystic fibrosis gene. Cystic fibrosis is a fatal disease associated with abnormal secretions. It is diagnosed in children at a rate of 1 in 3900. A defective gene causes the body to produce abnormally thick mucus that clogs the lungs and leads to life-threatening lung infections. More than 10 million Americans are unknown and symptomless carriers of the defective cystic fibrosis gene. Each time two carriers have a child, there is a 25% chance that the child will have cystic fibrosis.

In 1989, the search for the cystic fibrosis gene was narrowed to a region of 1 million nucleotides on the chromosome 7, but the exact location was unknown. When that region was sequenced, biologists compared it against a database of known genes and discovered similarities between some segment within the region and a known gene that codes for *adenosine triphosphate (ATP) binding proteins*. These proteins span the cell membrane multiple times as part of the ion transport channel; this seemed a plausible function for a cystic fibrosis gene, given the fact that the disease involves sweat secretions with abnormally high sodium content. So the similarity analysis shed light on a damaged mechanism in faulty cystic fibrosis genes.

References

[DEKM] **R. Durbin, S. Eddy, A. Krogh, G. Mitchison: Biological sequence analysis – Probabilistic models of proteins and nucleic acids. Cambridge University Press, 1998. ISBN 0-521-62971-3.**

[JP] **N.C. Jones, P.A. Pevzner: An introduction to bioinformatics algorithms. MIT Press, 2004. ISBN 0-262-10106-8.**

[CLRS] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: Introduction to Algorithms, Second Edition. MIT Press, 2001. ISBN 0-262-03293-7.

[G] D. Gusfield: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, 1997. ISBN 0-521-58519-8.