Introduction to Computational Molecular Biology

http://bio5495.wustl.edu/

Fall 2001

Sean Eddy
Howard Hughes Medical Institute
Department of Genetics
Washington University School of Medicine
St. Louis, Missouri

Contents

Chapter 1

Sequence alignment algorithms

> *"Cryptography has contributed a new weapon to the student of unknown scripts.... the basic principle is the analysis and indexing of coded texts, so that underlying patterns and regularities can be discovered. If a number of instances are collected, it may appear that a certain group of signs in the coded text has a particular function..."*
>
> – John Chadwick, *The Decipherment of Linear B*[1]

DNA, RNA, and protein are an alien language. Essentially, this book is about methods for cryptographically attacking this language. As biologists, we want to decipher both its meaning and its history.

Conveniently, the language is written in a small finite alphabet. Nucleic acids are polymers composed of a sequence made from an alphabet four nucleotide bases: A, C, G, and T (adenine, cytosine, guanine, and thymine) for DNA; A, C, G, and U for RNA (uracil instead of thymine). Proteins are polymers composed of a sequence made from an alphabet of twenty amino acids.[1]

The notion that biology might be based on an alphabetic code is older than one might think. Friedrich Miescher, who first chemically isolated DNA, wrote to his brother in 1892 that perhaps genetic information might be encoded in a linear form using a few different chemical units, "just as all the words and concepts

[1] Like almost any simple statement about molecular genetics, this is a lie. Biology is full of exceptions to rules. Nucleic acids and proteins also contain modified residues besides the basic alphabets of four and twenty. Usually these modifications are added post-synthesis, after the molecule has been assembled from residues in the standard alphabet (though not always; remember, exceptions to everything). We won't need to worry about nonstandard residues in this book, but you should be aware that they exist.

Figure 1.1: Salvador Dali's *Galacidalacidesoxyribonucleicacid, Homage to Watson and Crick*, 1963. Note the figures in quartets to the right - signifying the tetranucleotide hypothesis that DNA was composed of a simple repeating unit of A, C, G, and T and was therefore too simple to encode genetic information - an idea that was outdated by the time Dali painted *Galacid*.

in all languages can find expression in twenty-four to thirty letters of the alphabet". Ironically, Miescher's DNA was thought to be too simple to be the carrier of hereditary information until the molecular genetics revolution of the 1940's and 1950's. The "tetranucleotide hypothesis" that DNA was composed of a simple repeating tetramer of A, C, G, T even became part of a Dali painting (Figure 1.1).

It is fortunate that the genetic code is alphabetic. It makes it susceptible to powerful forms of attack - string comparison and pattern recognition. We don't have to understand a language to recognize that some patterns recur, and to statistically associate certain patterns with particular effects. In the old science fiction movie *The Day the Earth Stood Still*, the alien words "klaatu barada nikto" keep the robot Gort from destroying the Earth; the movie never tells us what the phrase means, but that's fairly unimportant in the grand scheme of things. In biology,

```
(a)
  HBA_HUMAN  GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKL
             G+ +VK+HGKKV  A+++++AH+D++ +++++LS+LH  KL
  HBB_HUMAN  GNPKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKL
(b)
  HBA_HUMAN  GSAQVKGHGKKVADALTNAVAHV---D--DMPNALSALSDLHAHKL
             ++ ++++H+ KV   + +A  ++          +L+ L+++H+ K
  LGB2_LUPLU NNPELQAHAGKVFKLVYEAAIQLQVTGVVVTDATLKNLGSVHVSKG
(c)
  HBA_HUMAN  GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSD----LHAHKL
             GS+ + G +   +D L  ++ H+ D+  A +AL D    ++AH+
  F11G11.2   GSGYLVGDSLTFVDLL--VAQHTADLLAANAALLDEFPQFKAHQE
```

Figure 1.2: Three sequence alignments to a fragment of human alpha globin. (a) Clear similarity to human beta globin. (b) A structurally plausible alignment to leghaemoglobin from yellow lupin. (c) A spurious high-scoring alignment to a nematode glutathione S-transferase homologue named F11G11.2.

once we know that one sequence encodes a protein kinase (a protein that attaches phosphorus to specific target proteins, one of the main molecular signalling systems used in cells), we can guess that similar sequences also encode protein kinases - even though we may not know anything about how the proteins fold, what proteins they interact with, or where and when they act in the cell.

The concept of *similarity searching* is therefore central to computational sequence analysis. If two sequences are significantly similar to each other, we infer that they probably have the same function.

We have to define what we mean by "significantly similar". *unfinished...some introductory comments about sequence evolution and alignment, introducing the notions of substitution, insertion, and deletion (which we will model), and duplication and inversion (which we generally won't)...*

An example is shown in Figure 1.2.

Brute force comparison

Probably the simplest possible string comparison algorithm is to determine whether two equal-length strings are identical or not. We compare the first characters, sec-

ond characters, third characters, and so on, until we either find a mismatch (and thus the strings are not identical), or we run out of string (and thus the strings are identical). In pseudocode, this algorithm is:

Algorithm 1: STRINGS_IDENTICAL
Input: Two sequences x and y of length N.
Output: **TRUE** if x and y are identical; else **FALSE**.

```
(1)    for i = 1 to N
(2)        if x_i ≠ y_i
(3)            return FALSE
(4)    return TRUE
```

How many operations does this take to compute? If we're lucky, it can take us as little as 1 character comparison, which happens when the first characters in each string are different. If the strings are random sequences of characters chosen equiprobably from an alphabet of K letters, the probability of matching at any given position by chance is $\frac{1}{K}$; it usually won't take us more than a character or two to determine that two randomly chosen strings are different (the number of comparisons follows a geometric distribution, and the expected number of comparisons is $\frac{K}{K-1}$). If letters aren't equiprobably distributed, the number of comparisons will increase, because spurious matches will be more common. In the worst case (which is when the strings really are identical) our algorithm will take us N character comparisons. We'll say that the algorithm can take on the order of N steps in the worst case, and abbreviate this by saying this is an $O(N)$ algorithm (pronounced "order-N algorithm", or we can also say "this algorithm has a big-O of N".) We'll discuss "big-O" notation in more detail soon.

The lowly(?) dotplot

Our brain is good at recognizing patterns in data. Some of the best tricks in a computational biologist's repertoire are visualization tools - the aim being to deliver the data efficiently to the highly evolved pattern recognition machinery in your brain. For recognizing similarities between two sequences, one of the most basic visualizations is the *dotplot*.

In a dotplot, we draw a matrix with one sequence (call it x) as the vertical axis and the other sequence (call it y) as the horizontal axis. Each cell of this dotplot

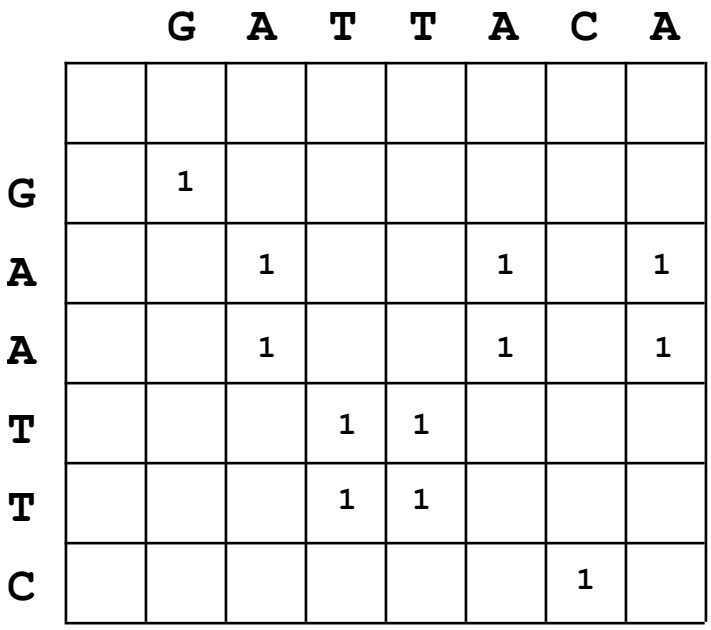

Figure 1.3: A dotplot of two fictitious sequences.

matrix $D(i, j)$ is going to represent the alignment of residue i of sequence x to residue j of sequence y. The simplest dotplotting rule is that for every identical pair of residues x_i and y_j, we color the corresponding cell $D(i, j)$ black. Significant local alignments appear as strong diagonal lines in the resulting plot, against a random background of uncorrelated dots. An example is shown in Figure 1.3.

We can use the dotplot procedure to better introduce the pseudocode notation we'll use for algorithms in this book. Given two sequences x and y, the algorithm to construct the dotplot matrix D is as follows:

Algorithm 2: A dotplot representation of sequence similarity.

```
DOTPLOT(x, y)
(1)   M ← LENGTH(x)
(2)   N ← LENGTH(y)
(3)   for i ← 1 to M
(4)      for j ← 1 to N
(5)         if x_i = y_j
(6)            D(i, j) ← 1
(7)         else
(8)            D(i, j) ← 0
```

Let's define some of the notation in this pseudocode in more detail:

- The $\leftarrow$ symbol indicates assignment. For example, $D(i, j) \leftarrow 1$ means "set $D(i, j)$ to 1".
- The $=$ symbol indicates comparison; $x_i = y_j$ evaluates to TRUE if x_i and y_j are the same character.
- A small cap font indicates calls to other functions. For example, LENGTH(x) returns the length of sequence x in residues. Sometimes we won't bother to define "obvious" functions, including LENGTH(), MIN(), and MAX(), but most will be defined somewhere in the text. For example, the above algorithm defines DOTPLOT(x, y), which takes two sequences x and y as arguments.
- A bold font indicates pseudocode logic constructs. You're probably familiar with these from some basic programming experience. Examples include **for** loops, **while** loops, and **if/then** and **if/then/else** conditional statements.

For the purposes of this chapter, we're just using DOTPLOT to introduce the notion of a matrix representation of sequence similarities, and our pseudocode notation for algorithms. Nonetheless, don't forget that dotplots can be very useful tools in bioinformatics! Figure 1.4 shows an example of real dotplots, produced with the software package **DOTTER** [12]. **DOTTER** uses a more sophisticated plotting algorithm than Algorithm 2. **DOTTER** uses a gray scale scheme to indicate points that are supported by other similar or identical residue pairs along the diagonal nearby, so that regions of consistent alignment are better highlighted.

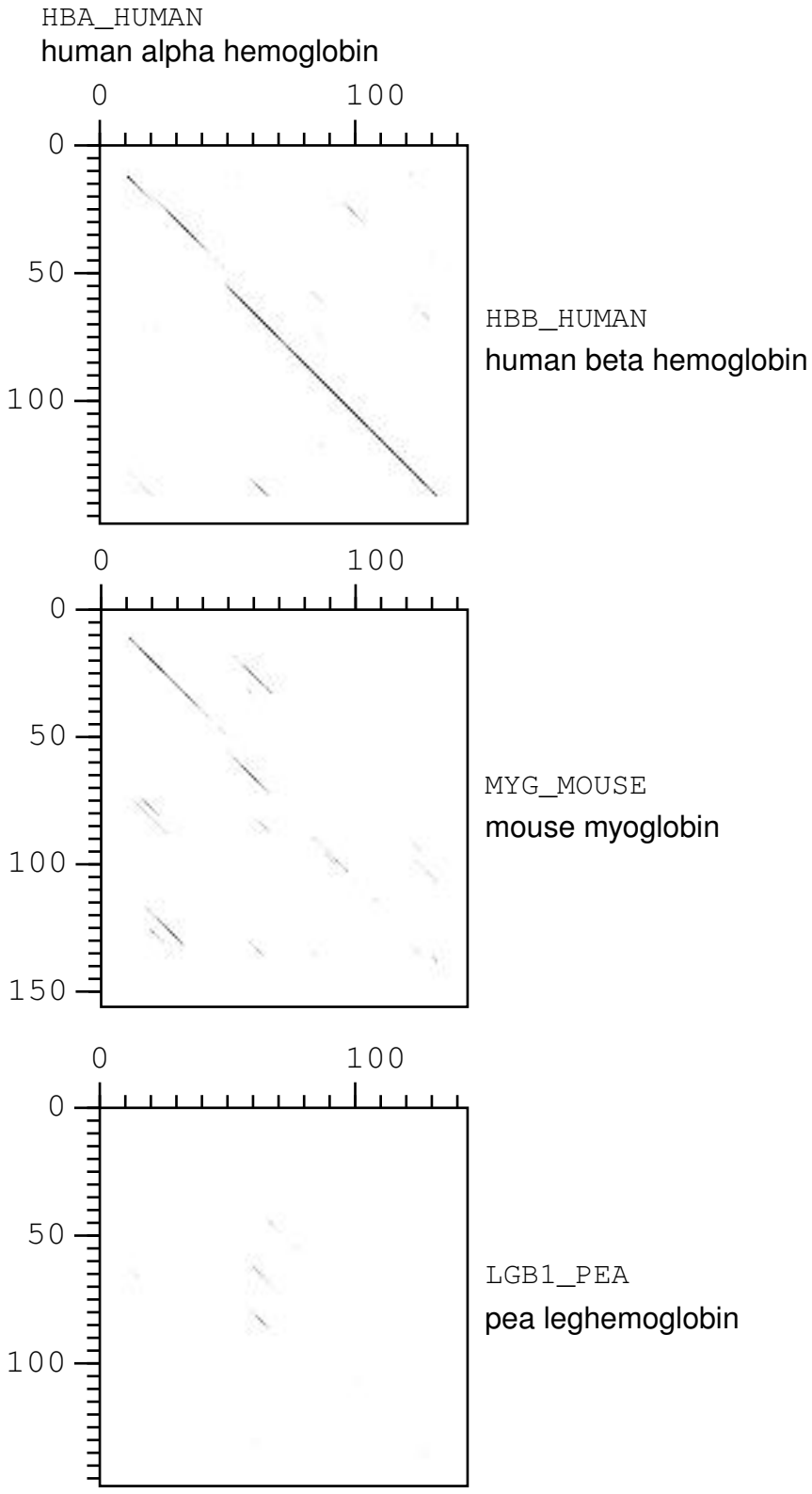

Figure 1.4: Three examples of real dotplots produced by **DOTTER**, illustrating similarities of human α-hemaglobin (horizontal axis) to structurally homologous globin sequences of progressively less obvious sequence similarity: human β-globin, mouse myoglobin, and the distantly related pea leghemoglobin (vertical

sidebar - Big-O Notation

We're often concerned with comparing the efficiency of algorithms. A natural way to measure the efficiency of an algorithm is to show how required computational resources (both running time and memory) will scale as the size of the problem increases. Problem sizes can often be measured in number of units, n. For instance, in sequence alignment, we will measure problem sizes as the length of one or more sequences in residues.

Generally we will discuss the efficiency of an algorithm by analyzing its *asymptotic worst-case running time*. If an algorithm takes time proportional to the square of the number of units in the problem, for instance, we will say that the algorithm is $O(n^2)$ ("order n-squared") in time. This $O()$ notation is called "big-oh" notation.

Let's be more precise. Mathematically what we're saying with big-O notation is as follows. The real running time of an algorithm is a function $f(n)$ of the problem size n. We could (and often do) measure $f(n)$ empirically, by actually implementing an algorithm in a program and timing the program on problems of varying size n. We expect $f(n)$ to increase monotonically with increasing n, but aside from that $f(n)$ can be a rather complex and arbitrary function because some of the factors it depends on are independent of our algorithm – most notably, how fast our computer executes individual lines of code, how long it takes to execute any necessary overhead in our program compared to the core of the algorithm itself, and whether this particular problem of size n is a hard case or an easy case for our algorithm to solve. At least at first glance we're not interested in such details; we're more concerned with the behavior of the essence of the algorithm, in the realm of larger problem sizes where its demands (rather than trivial programmatic details) dominate $f(n)$. When we say an algorithm is $O(g(n))$, we mean that there are constants c and n_0 such that $cg(n) >= f(n)$ for all problems of size $n > n_0$.

We will call an $O(k^n)$ algorithm (for some integer k) an *exponential time* algorithm, and we will be reluctant if not horrified to use it; sometimes we will go so far as to call the problem *intractable* if this is the best algorithm we have to solve it. We will call an $O(n^k)$ algorithm a *polynomial time* algorithm, and so long as k is reasonable (3 or less) we will usually speak of the problem that this algorithm solves as *tractable*. Some problems have no known polynomial time solutions. We will run across this later in a discussion of so-called NP (nondeterministic polynomial) problems.

In general we will tend to believe that an $O(2^n)$ algorithm is inferior to an $O(n^3)$ algorithm , which is inferior to an $O(n^2)$ algorithm, which is inferior to

an $O(n \log n)$ algorithm, which is inferior to an $O(n)$ algorithm... and so on. However, $O()$ notation, though ubiquitous and useful, does need to be used with caution. A superior $O()$ does not necessarily imply a faster running time, and equal $O()$'s do not imply the same running time.

Why not? First of all, there's that constant c. Two $O(n)$ algorithms may have quite different constants, and therefore run in quite different times. Second, there's the notion (embodied in the constant n_0) that $O()$ notation indicates an *asymptotic* bound, as the problem size increases. An $O(n)$ algorithm may require so much precalculation and overhead relative to an $O(n^2)$ algorithm that its asymptotic speed advantage is only apparent on unrealistically large problem sizes. Third, there's the notion that $O()$ notation is a *worst-case* bound. We will see examples of algorithms that have poor $O()$ because of a rare worst case scenario, but their average-case behavior on typical problems is much better.

We will use $O()$ notation to describe the asymptotic worst-case memory requirement of an algorithm, as well as its asymptotic worst-case running time. For example, we will often say something like "this algorithm is $O(n^3)$ in time and $O(n^2)$ in memory".

We will sometimes see examples of trading time for memory (or vice versa). Especially when memory is the major factor limiting us, we may favor an algorithm with a worse constant or even a worse $O()$ in time, in exchange for an improved $O()$ in memory. In general, in computational biology, we will be reasonably happy if an algorithm is $O(n^2)$ or better in time and memory, and quite pleased to have $O(n)$ in either or both. We will present a few algorithms that are $O(n^3)$ in time or even higher. Only rarely will we see algorithms that are $O(n^3)$ in memory or higher; gene sequences are usually in the ballpark of length 100-10000 residues, and on current computers, though we may survive square matrices with up to $10000^2 \simeq 10^8$ numbers in them (400 megabytes), we generally won't be able to deal with cubic lattices with $10000^3 \simeq 10^{12}$ numbers in them (4 terabytes).

Analyzing algorithms to determine their $O()$ can be a bit of an art form, but in this book will usually be straightforward. For example, the depth of nesting of a dynamic programming algorithm's inner loop gives away its running time; "for $i = 1..n, j = 1..n, k = 1..n$, do something constant" is obviously $O(n^3)$ in time. Similarly, the memory usage of a dynamic programming algorithm is usually implicit in the notation for the score being calculated; a calculation of a square matrix $F(i, j)$, where both i and j range from $1..n$, is obviously $O(n^2)$ in memory.

Problem sizes are not necessarily expressed with a single number n. In pairwise alignment, for instance, we will define the problem size in terms of *two*

sequence lengths, M and N, and talk about algorithms that are $O(MN)$ in time and memory.

There are various additional subtleties to $O()$ notation, related asymptotic notations, and algorithmic analysis in computer science. For example, we will casually assume for simplicity that our computing machine performs its basic operations in $O(1)$ time, though pedantically speaking this isn't true – the number of bits it requires to store a number n scales with $\log n$, so for extremely large n, when numbers exceed the native range of 32-bit or 64-bit arithmetic on our computing machine, it will no longer take $O(1)$ time to perform storage, retrieval, and arithmetic operations. But these concerns leave the realm of computational biology and enter the realm of careful computer science theory. The reader is advised to consult Cormen et al. [2] or Knuth [7] for more formal details on algorithmic analysis.

Longest common substring

With a small modification in the dotplot algorithm, we can convert it into an algorithm that finds the longest common substring (LCS) that the two strings x and y have in common. For example, the LCS of GATTACA and GAATTC is ATT, 3 letters long. The LCS algorithm is as follows:

Algorithm 3: Longest common substring.
LCS(x, y)
(1) **Initialization:**
(2) $D(0,0) = 0.$
(3) **for** $i = 1$ **to** M
(4) $D(i,0) = 0.$
(5) **for** $j = 1$ **to** N
(6) $D(0,j) = 0.$
(7) **Recursion:**
(8) **for** $i = 1$ **to** M
(9) **for** $j = 1$ **to** N
(10) **if** $x_i = y_j$
(11) $D(i,j) \leftarrow D(i-1,j-1) + 1$
(12) **else**
(13) $D(i,j) \leftarrow 0$

		G	A	T	T	A	C	A
	0	0	0	0	0	0	0	0
G	0	1	0	0	0	0	0	0
A	0	0	2	0	0	1	0	1
A	0	0	1	0	0	1	0	1
T	0	0	0	2	1	0	0	0
T	0	0	0	1	3	0	0	0
C	0	0	0	0	0	0	1	0

Figure 1.5: Finding the LCS of two fictitious sequences.

The score (length) of the LCS is given by $\max_(i, j)D(i, j)$. A D matrix calculated by this algorithm is shown in Figure 1.5.

This now introduces some key new concepts:

Dynamic programming. Scoring $D(i, j)$ requires that we know $D(i - 1, j - 1)$. Our solution is *recursive*. We solve smaller problems first, and use those solutions to solve larger problems. We know what problems are smaller (and already calculated) by our position in the tabular matrix D. Computer scientists refer to a recursive method that stores the results of intermediate solutions in a tabular matrix as a *dynamic programming* method. (Here the word "programming" refers to the tabular matrix, not to computer code – another place this usage might confuse you is the term "linear programming", a tabular method for solving systems of equations.) Dynamic programming methods are pivotal in computational biology.

Initialization. Dynamic programming methods have to worry about boundary conditions. You can't start recursing until you've defined the smallest solutions non-recursively. We have to make sure that if the recursion is going to look into a particular cell (i, j), that cell must always have a valid number in it. In the LCS algorithm, this means initializing the 0 row and the 0 column to 0, so $D(i-1, j-1)$ never accesses an uninitialized cell.

Solution and traceback. The LCS algorithm only gives us a score matrix $D(i, j)$. To get the LCS itself, we have to do additional operations on that matrix. The length of the LCS is obtained by $\max_(i, j)D(i, j)$. The LCS itself is obtained by a *traceback* from the starting cell $(i', j') = \text{argmax}_(i, j)D(i, j)$ - we follow the optimal path back along the diagonal until we reach a 0, halting the traceback of the LCS.

This LCS algorithm is $O(n^2)$ for two strings of lengths $\sim n$. Much more efficient $O(n)$ algorithms are possible using suffix trees [5]. The reason to show this relatively less efficient LCS algorithm is purely pedagogical, a stepping stone on our way to explaining dynamic programming sequence alignment algorithms.

sidebar: The basics of scoring alignments

It would be pretty straightforward to convert the exact-match LCS algorithm into an algorithm that found the *optimal scoring* local ungapped alignment, if we defined a scoring function $\sigma(a, b)$ for aligning two residues a, b. LCS essentially imposes a scoring function of $\sigma(a, b) = 1$ if $a = b$, $\sigma(a, b) = -\infty$ if $a \neq b$. If, say, we instead set $\sigma(a, b) = -1$ for $a \neq b$, the algorithm would allow matching substrings that included some mismatches.

Since we're about to introduce dynamic programming optimal alignment algorithms, it's worth thinking a little more deeply about scoring functions. We need a procedure for assigning numerical scores to alignments, so we can rank them and find the best.

Actually enumerating all possible pairwise alignments and ranking them is unrealistic. There are

$$\frac{2^{2L}}{\sqrt{2\pi L}}$$

possible global alignments between two sequences of length L [3, p.18]. Since we are concerned with biological sequences where L might range from hundreds (for protein sequences) to millions(for chromosomal DNA sequences), this number of alignments is clearly too large to handle.

Therefore we are concerned with the functional form of our alignment scoring procedure. We will want to make independence assumptions, so that we solve optimal alignment problems efficiently using recursive dynamic programming algorithms, without having to enumerate all possible alignments.

A very simple functional form for an alignment scoring procedure is to score $+1$ for every identical pair of residues. Maximizing this score amounts to finding the pairwise alignment of maximal percent identity. Though molecular biol-

ogy papers routinely cite "pairwise sequence identity" as if it were a meaningful statistic, in fact it's not a good idea to use sequence identity as the sole criterion for choosing optimal alignments (and no popular bioinformatics program does).

First, we want to use more information about residue conservation than just simple identity. In proteins, for example, we want to capture information about structural and chemical similarity between amino acids – for example, a K/R "mismatch" should be penalized fairly little (and maybe even rewarded) because lysine (K) and arginine (R) are both positively charged amino acids with similar roles in protein structures, whereas an R/V mismatch should be penalized fairly strongly because arginine and valine (V) are dissimilar amino acids that don't substitute for each other well. So, more generally, we can assign a score $\sigma(a, b)$ to a residue pair a, b - $\sigma(a, b)$ can be high if a and b are an acceptable substitution, and low if not. We'll call σ a *substitution matrix*.

We also need a scoring function for gaps. If we don't penalize our use of gaps somehow, our "optimal" alignments can get pretty unrealistic; the computer will insert lots of gap symbols trying to maximize the apparent similarity of the residues. In a pairwise sequence alignment, we can't tell the difference between an insertion in one sequence versus a deletion in the other; therefore we sometimes refer to gaps as *indels*, short for insertion/deletion. Most generally we can subtract a gap score penalty $\gamma(n)$ every time we invoke a gap of length n in an alignment. We will see that generalized gap score functions are rarely used. Instead, we will more often use two simple functional forms that are more compatible with efficient recursive solutions. *Linear gap penalties* have the form $\gamma(n) = nA$ - a cost of A units per gap character. *Affine gap penalties* have the form $\gamma(n) = nA + B$ - B is a *gap-open* cost to start the gap in the first place, and A is a *gap-extend* cost per residue in the gap.

For now we will defer the important issue of where these scores $\sigma(a, b)$ and $\gamma(n)$ come from. In the examples below, we'll use some simple values just to illustrate how the alignment algorithms work. In a later chapter, we'll work on the scoring functions themselves, using probability theory.

Global alignment: Needleman/Wunsch/Sellers

Let's look at the problem of *global sequence alignment*: what is the best pairwise alignment between two complete sequences (including gaps), and what is its score?

The first dynamic programming algorithm for global sequence alignment was

introduced in 1970 by Needleman and Wunsch. We still refer to global dynamic programming alignment algorithms as *Needleman/Wunsch algorithms*, although this is a bit imprecise. Many people have even forgotten that the original 1970 Needleman/Wunsch algorithm is $O(N^3)$ in time, and haven't noticed that their "Needleman/Wunsch" implementation is actually a different algorithm. Sellers (1974) introduced a simpler version of the algorithm using linear gap penalties that runs in time $O(N^2)$. Needleman/Wunsch/Sellers is also a slightly easier algorithm to understand, so we'll start with it. The version that is actually most often used is the affine gap version of the algorithm, due to Gotoh (1982), and we'll see it later.

We are given two sequences x and y. x contains M residues numbered $1..M$; we will refer to the i'th residue in x as x_i. Similarly, y is composed of N residues, referred to as y_j, for $j = 1..N$. We are given a scoring system $\sigma(a, b)$ and $\gamma(n)$. Finally, we make a crucial simplifying assumption: the gap penalty $\gamma(n)$ is a linear gap penalty, $\gamma(n) = nA$.

The idea of the algorithm is to find the optimal alignment of the sequence prefix $x_1..x_i$ to the sequence prefix $y_1..y_j$. Let us call the score of this optimal alignment $F(i, j)$. (F here stands for "Forward" – later you'll see why.) The key observation is this: I can calculate $F(i, j)$ easily if I already know F for smaller alignment problems. The optimal alignment of $x_1..x_i$ to $y_1..y_j$ can only end in one of three ways:

1. x_i is aligned to y_j. $F(i, j)$ will be the score for aligning x_i to y_j) - $\sigma(x_i, y_j)$) - plus the optimal alignment score I'd already calculated for the rest of the alignment not including x_i and y_j - $F(i - 1, j - 1)$.

2. x_i is aligned to a gap character. This means we're adding a gapped residue (which costs us a penalty of A) to the optimal alignment we've already calculated for $x_1..x_{i-1}$ to $y_1..y_j$: so the score of this alternative is $F(i - 1, j) - A$.

3. y_j is aligned to a gap character. This is analogous to the case above. This alternative for $F(i, j)$ has the score $F(i, j - 1) - A$.

The optimal $F(i, j)$ is the maximum of these three alternatives.

So, I've defined a recursive solution, meaning that I've defined $F(i, j)$ strictly in terms of solutions of smaller subproblems $F(i - 1, j - 1)$, $F(i - 1, j)$, and $F(i, j - 1)$. Now I need to arrange my calculations so that I make sure I calculate $F(i - 1, j - 1)$, $F(i - 1, j)$, and $F(i, j - 1)$ before I try to calculate $F(i, j)$. An obvious way to do this is to arrange the numbers $F(i, j)$ in a matrix, start

at the upper left corner for the smallest alignment problem, $F(0,0)$, and work my way towards the lower right corner, $F(M,N)$. When I'm through, $F(M,N)$ contains the optimal score for aligning the prefix $x_1..x_M$ to $y_1..y_N$ ... in other words, the score for aligning all of x and y globally to each other. We call this a *global alignment*. We'll see an example of *local alignment* (and revisit the LCS algorithm) in a little while.

The first stage of the Needleman/Wunsch/Sellers algorithm is then as follows:

Algorithm 4: Needleman/Wunsch/Sellers, fill
Input: Two sequences x and y of length M and N, respectively; scoring matrix $\sigma(a,b)$; linear gap cost A.
Output: Dynamic programming matrix F.

(1) **Initialization:**
(2) $F(0,0) = 0$.
(3) **for** $i = 1$ **to** M
(4) $F(i,0) = -iA$.
(5) **for** $j = 1$ **to** N
(6) $F(0,j) = -jA$.
(7) **Recursion:**
(8) **for** $i = 1$ **to** M
(9) **for** $j = 1$ **to** N
(10)
$$F(i,j) = \max \begin{cases} F(i-1,j-1) + \sigma(x_i, y_j), \\ F(i-1,j) - A, \\ F(i,j-1) - A. \end{cases}$$

The algorithm is $O(MN)$ in time and memory. When we're done, $F(M,N)$ contains the optimal alignment score. Figure 1.6 shows an example.

However, we don't yet have the optimal alignment. To get that, we perform a *traceback* in F to find the path that led us to $F(M,N)$.

One way to traceback is by recapitulating the score calculations from algorithm 4. We start in cell $(i = M, j = N)$ and find which cell(s) we could have come from, by recalculating the three scores for coming from $(i-1, j-1)$, $(i-1, j)$, or $(i-1, j)$ and seeing which alternative(s) give us $F(i,j)$ and are thus on the optimal alignment path. We then choose one of these possible alternatives (either at random, but more usually arbitrarily, by order of evaluation), and move to that cell. If the optimal path moves to $(i-1, j-1)$ we align x_i to y_j; if the optimal path moves to $(i-1, j)$ we align x_i to a gap; and if the optimal path moves

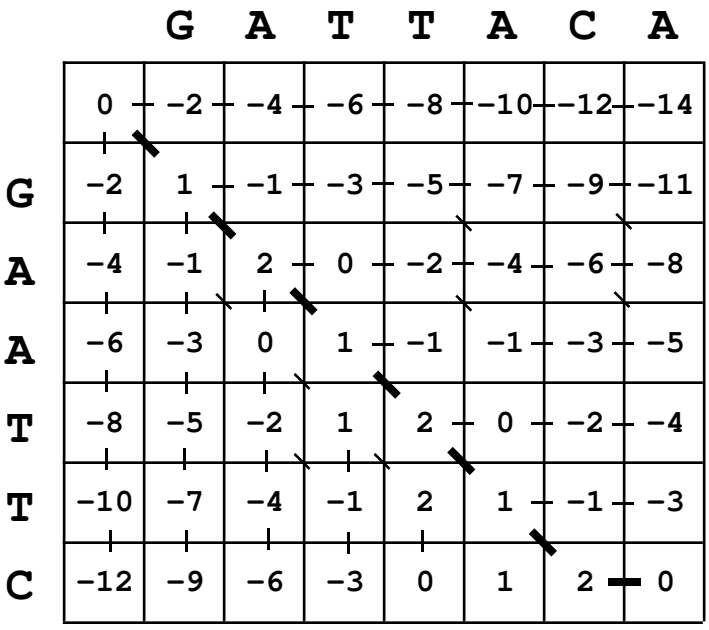

Figure 1.6: The dynamic programming matrix calculated for two strings GATTACA and GAATTC, using a scoring system of $+1$ for a match, -1 for a mismatch, and -1 per gap.

to $(i, j-1)$, we align y_j to a gap. We iterate this process until we've recovered the whole alignment.

There's an implementation detail we need to worry about. The output of our algorithm will be two aligned strings (let's call the aligned strings $\bar{x}$ and $\bar{y}$) of equal length L. We don't know L when we start the traceback, so we don't know where to start storing characters in $\bar{x}$ and $\bar{y}$. A simple trick is to build $\bar{x}$ and $\bar{y}$ backwards, then reverse them when we're done and we know L.

This traceback algorithm is as follows:

Algorithm 5: Needleman/Wunsch/Sellers, traceback
Input: Two sequences x and y of length M and N; scoring matrix $\sigma(a, b)$; linear gap cost A; and matrix F calculated with algorithm 4.
Output: Aligned sequences $\bar{x}$ and $\bar{y}$, containing gap characters, of equal length L.

(1) $k = 0$
(2) $i = M$
(3) $j = N$
(4) **while** $i > 0$ **or** $j > 0$
(5) $\quad k = k + 1$
(6) $\quad$ **if** $i > 0$ **and** $j > 0$ **and** $F(i, j) = F(i - 1, j - 1) + \sigma(x_i, x_j)$
(7) $\quad\quad \bar{x}_k = x_i$
(8) $\quad\quad \bar{y}_k = y_j$
(9) $\quad\quad i = i - 1$
(10) $\quad\quad j = j - 1$
(11) $\quad$ **else if** $i > 0$ **and** $F(i, j) = F(i - 1, j) - A$
(12) $\quad\quad \bar{x}_k = x_i$
(13) $\quad\quad \bar{y}_k =$ -
(14) $\quad\quad i = i - 1$
(15) $\quad$ **else if** $j > 0$ **and** $F(i, j) = F(i, j - 1) - A$
(16) $\quad\quad \bar{x}_k =$ -
(17) $\quad\quad \bar{y}_k = y_j$
(18) $\quad\quad j = j - 1$
(19) $L = k$
(20) **for** $k = 1$ **to** $L/2$
(21) $\quad$ SWAP($\bar{x}_k, \bar{x}_{L-k+1}$)
(22) $\quad$ SWAP($\bar{y}_k, \bar{y}_{L-k+1}$)

Lines 20 to 22 are just the routine for reversing the order of the two aligned strings. The traceback takes us $O(M + N)$ time and memory (not counting the $O(MN)$ memory of the input dynamic programming matrix F). Because the traceback is linear-time, and the fill stage was quadratic-time, we don't pay a significant computational cost to recapitulate the calculations of the fill stage to trace the optimal path.

Local alignment: Smith/Waterman

A *local alignment* is the best (highest scoring) alignment of a *substring* of x to a *substring* of y, as opposed to a global alignment of the entire strings. Local alignment is usually more biologically useful. It is usually the case that only part of two sequences is significantly similar, so those parts must be identified.

The local alignment problem could be solved by brute force, using the Needleman/Wunsch/Sellers global alignment algorithm. For a sequence of length N, there are about N^2 substrings (choose a starting point $a = 1..N$ and an endpoint $b = i..N$). We could globally align all $\sim N^2M^2$ combinations of substrings $x_a..x_b$ and $y_c..y_d$ using the $O(MN)$ global alignment algorithm, and finding the choice of a, b, c, d start and end points that give us the optimal alignment score. This yields an $O(M^3N^3)$ local alignment algorithm, which is pretty ugly. At least we know the problem is solvable in polynomial time!

We can do that a bit better, by noting that we really only have to consider all possible start points $a = 1..N$, defining all possible suffixes of each sequence. If I use N/W/S to align a suffix $x_a..x_N$ to $y_c..y_M$, I can find the best endpoints b, d by finding the maximum score in the F matrix,

$$(a, b) = \text{argmax}_{i,j} F(i, j),$$

instead of just looking at the score in the bottom right cell. Now I only need to run the $O(MN)$ global alignment algorithm on $O(MN)$ combinations of suffixes, so I've improved to a $O(M^2N^2)$ local alignment algorithm. We already used this idea of searching the whole matrix to find the optimal alignment endpoints in the LCS algorithm, and it's one of the two keys to dynamic programming local alignment.

Now how do we avoid having to test all possible suffixes starting at positions a and b? The amazing thing is that optimal local alignment can be done in $O(MN)$ time by a second tiny modification to the global alignment algorithm, one that's also akin to what we did in the LCS algorithm: allow a fourth choice for $F(i, j)$ of 0. The 0 allows the alignment to restart at any internal start point in x and y. Remarkably, this tweak wasn't introduced until 11 years after Needleman/Wunsch; Temple Smith and Michael Waterman published it in 1981 [11]. Local dynamic programming alignment algorithms are called *Smith/Waterman* algorithms.

In pseudocode, the Smith/Waterman algorithm is as follows:

Algorithm 6: Smith/Waterman, fill
Input: Two sequences x and y of length M and N, respectively; scoring matrix $\sigma(a, b)$; linear gap cost A.
Output: Dynamic programming matrix F.

(1) **Initialization:**
(2) $F(0, 0) = 0$.
(3) **for** $i = 1$ **to** M
(4) $F(i, 0) = 0$.
(5) **for** $j = 1$ **to** N
(6) $F(0, j) = 0$.
(7) **Recursion:**
(8) **for** $i = 1$ **to** M
(9) **for** $j = 1$ **to** N

(10) $$F(i,j) = \max \begin{cases} 0, \\ F(i-1, j-1) + \sigma(x_i, x_j), \\ F(i-1, j) - A, \\ F(i, j-1) - A. \end{cases}$$

After filling the F matrix, we find the optimal alignment score and the optimal end points by finding the highest scoring cell, $\max_{i,j} F(i, j)$. To recover the optimal alignment, we trace back from there, terminating the traceback when we reach a cell with score 0.

Variants of the core DP alignment algorithms

Affine gap penalties: the Gotoh algorithm

Algorithm 7: Affine Smith/Waterman, fill
Input: Two sequences x and y of length M and N, respectively; scoring matrix $\sigma(a, b)$; gap-open cost A; gap-extend cost B.
Output: Dynamic programming matrices M, I_x, and I_y.

(1) **Initialization:**
(2) **for** $i \leftarrow 0$ **to** M
(3) $M(i, 0) \leftarrow 0$,
(4) $I_x(i, 0) \leftarrow -\infty$
(5) $I_y(i, 0) \leftarrow -\infty$
(6) **for** $j \leftarrow 1$ **to** N
(7) $M(0, j) \leftarrow 0$
(8) $I_x(0, j) \leftarrow -\infty$
(9) $I_y(0, j) \leftarrow -\infty$
(10) **Recursion:**
(11) **for** $i \leftarrow 1$ **to** M
(12) **for** $j \leftarrow 1$ **to** N

(13) $$M(i,j) \leftarrow \max \begin{cases} 0, \\ M(i-1,j-1) + \sigma(x_i, x_j), \\ I_x(i-1,j-1) + \sigma(x_i, x_j), \\ I_y(i-1,j-1) + \sigma(x_i, x_j) \end{cases}$$

(14) $$I_x(i,j) \leftarrow \max \begin{cases} M(i-1,j) - A, \\ I_x(i-1,j) - B \end{cases}$$

(15) $$I_y(i,j) \leftarrow \max \begin{cases} M(i,j-1) - A, \\ I_y(i,j-1) - B \end{cases}$$

Arbitrary gap penalties: the Needleman/Wunsch algorithm

The original 1970 Needleman/Wunsch algorithm didn't assume any particular functional form for the gap penalty. It could work with an arbitrary gap penalty function $\gamma(n)$. For an arbitrary $\gamma(n)$, we have to examine all possible gap lengths to decide how we optimally reach $F(i, j)$, not just the cells in F to our immediate top and left.

For instance, instead of only considering the possibility $F(i,j) = F(i-1,j) - A$ for aligning x_i to a gap symbol with cost A, I actually need to know how many consecutive gap symbols are in $\bar{y}$ before I can assign a score to the gap. This means finding the best scoring alternative for aligning y_j to some residue x_h, for $h < i$, then aligning all the rest of the residues $x_h+1..x_i$ to gaps. The length of the gap is $i-h$. The score of this possible alternative for $F(i,j)$ is $F(\hat{h},j)+\gamma(i-\hat{h})$, for the optimal choice of $\hat{h}$, which I find by maximizing $F(h,j)+\gamma(i-h)$ over all $0 <= h < i$. I treat gaps in $\bar{x}$ analogously.

The original Needleman/Wunsch algorithm is then as follows:

Algorithm 8: Needleman/Wunsch, fill
Input: Two sequences x and y of length M and N, respectively; scoring matrix $\sigma(a,b)$; gap penalty function $\gamma(n)$.
Output: Dynamic programming matrix F.

(1) **Initialization:**
(2) $F(0,0) = 0$.
(3) **for** $i = 1$ **to** M
(4) $F(i,0) = \gamma(i)$.
(5) **for** $j = 1$ **to** N
(6) $F(0,j) = \gamma(j)$.
(7) **Recursion:**
(8) **for** $i = 1$ **to** M
(9) **for** $j = 1$ **to** N

(10)
$$F(i,j) = \max \begin{cases} F(i-1,j-1) + \sigma(x_i, x_j), \\ \max_{0<=h<i} F(h,j) + \gamma(i-h), \\ \max_{0<=k<j} F(i,k) + \gamma(j-k). \end{cases}$$

The traceback algorithm for this, by analogy to algorithm 5, should be obvious.

This algorithm is $O(MN(M+N))$ in time and $O(MN)$ in memory. The extra time factor of $(M+N)$ comes from finding optimal h and k by looking back over entire rows and columns. Since usually $M \simeq N$ (the sequences are of similar lengths), we more loosely say Needleman/Wunsch is $O(n^3)$ in time and $O(n^2)$ in memory.

We can extend the Smith/Waterman local alignment algorithm to arbitrary gap penalties in the same manner.

Linear memory: score-only

Reduced memory: shadow traceback matrices

Linear memory: the Myers/Miller algorithm

History and further reading

Dynamic programming sequence alignment was introduced by Saul Needleman and Christian Wunsch [9]. The more efficient $O(n^2)$ version of the algorithm, for linear gap cost, was introduced by Peter Sellers [10]. Local alignment was introduced by Temple Smith and Michael Waterman [11]. The $O(n^2)$ algorithms for affine gap alignment that are the most commonly used form today were introduced by Osamu Gotoh [4]. Michael Waterman and Mark Eggert described one of the best known "declumping" algorithms for obtaining multiple suboptimal independent local alignments [13]. The memory-efficient "Myers/Miller" algorithm was introduced to computational biology by Gene Myers and Webb Miller [8] based on earlier computer science work by Dan Hirschberg [6].

Bibliography

[1] J. Chadwick. *The Decipherment of Linear B.* Cambridge Univ. Press, Cambridge, UK, 1958.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990. ISBN: 0262031418. 1028 pages.

[3] R. Durbin, S. R. Eddy, A. Krogh, and G. J. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge UK, 1998.

[4] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982.

[5] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. ISBN 0521585198. 534 pages.

[6] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18:341–343, 1975.

[7] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison–Wesley, Reading, Massachusetts, second edition, 1973.

[8] Eugene W. Myers and Webb Miller. Optimal alignments in linear space. *Comput. Applic. Biosci.*, 4(1):11–17, 1988.

[9] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.

[10] P. Sellers. On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.*, 26:787–793, 1974.

[11] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.

[12] E. L. L. Sonnhammer and R. Durbin. A workbench for large scale sequence homology analysis. *Comput. Applic. Biosci.*, 10:301–307, 1994.

[13] Michael S. Waterman and Mark Eggert. A new algorithm for best subsequence alignments with application to tRNA–rRNA comparisons. *J. Mol. Biol.*, 197:723–728, 1987.