1 Introduction

Credit: My source for this material was *Biological sequence analysis*, by Durbin, Eddy, Krogh, and Mitchison.

In biology, one wants to know how closely two sequences are related. For example, sequences of amino acids composing a protein molecule, or sequences of nucleic acids in a DNA molecule. This question can be reduced to computer science by considering a DNA molecule as a string in a four-letter alphabet, or a protein molecule as a string in a 22-letter alphabet. However, the exact matching algorithms we have studied so far do not help much, as the two sequences in question may differ by some deletions and insertions as well as some mismatched characters, so we have to study approximate matches.

The first problem we take up is finding the optimal alignment of two strings. That is, we are allowed to insert gaps (a blank character) in either or both of the strings, and we wish to do so in such a way to make the resulting strings an optimal match.

*Example:* Suppose the two strings are `HEAGAWGHEE` and `PAWHEAE`. One possible alignment is

```
HEAGAWGHE-E
--P-AW-HEAE
```

To say the match is optimal means that its score is maximum. The score is computed by adding up the scores for each pair of characters in corresponding position (after inserting some gaps). The score for a pair of non-blank characters is specified in advance, in a table $s(a,b)$, where $a$ and $b$ are any two characters in the alphabet. The size of this table (number of entries) is then $n(n-1)/2$, where $n$ is the size of the alphabet; 210 for proteins and 6 for DNA. We will not discuss how these entries are determined–just assume they are given. The entries can be positive or negative. Positive means the two characters in question are related; i.e. it is good to line up these two characters if we can; the more positive the better. Negative means it’s bad to line up these two characters, it’s better to keep them apart.

For the case of proteins, the scoring matrix most commonly used is called the BLOSUM50 substitution matrix. Here it is:
Gaps are bad; we insert them if we have to in order to make other things line up well, but we have to pay a penalty. When one string contains a gap of length $g$, the "gap penalty" is given by

$$p(g) = -d - (g - 1)e$$

for some constants $d$ and $e$. The constant $d$ is called the gap-open penalty and $e$ is called the gap-extension penalty. Usually $e$ should be less than $d$, allowing long insertions and deletions to be penalized less than the same number of blanks not occurring together.

If we use BLOSUM50 for scoring, with $e = d = 8$ as the gap penalties, then it turns out that the above example alignment is optimal. Check that you have understood the definitions by computing the score of this alignment and some other possible alignments, and see that indeed this one seems to have the maximum score.

### 2 The Algorithm

The idea is to use dynamic programming to efficiently implement a recursion. Given two input strings $x$ and $y$, we build a matrix $F$ such that the entry $F[i,j]$ is the score of the optimal alignment of $x[1..i]$ and $y[1..j]$. We initialize
\( F[0, 0] = 0 \). We then proceed to fill the matrix from top left to bottom right. Suppose we have filled in \( F[i - 1, j - 1] \), \( F[i - 1, j] \), and \( F[i, j - 1] \), the three entries above, to the left, and diagonally above and left of \( F[i, j] \), and we have the (or an) optimal alignment for each of those three pairs. Then there are three corresponding ways to complete these alignments to an alignment of \( x[1..i] \) and \( y[1..j] \). We can either align \( x[i] \) with \( y[j] \) (in the first case), or align \( x[i] \) with a new gap (in the second case), or align \( y[j] \) with a new gap (in the third case). Therefore we have the recursion:

\[
F[i, j] = \max \left\{ F[i - 1, j - 1] + s(x[i], y[j]), F[i - 1, j] - d, F[i, j - 1] - d \right\}
\]

At the top row and the left side, we must specify \( F[i, 0] \) and \( F[0, j] \). The value \( F[i, 0] \) represents assigning a prefix of \( x \) to a gap, so we should define \( F[i, 0] = id \). Similarly down the left side \( F[0, j] = jd \), corresponding to assigning a prefix of \( y \) to a gap.

This recursion is a classic example of what dynamic programming is good for. If it is implemented naively, many values will be recomputed exponentially many times. Instead, we use a matrix to store the values, as indicated, and compute them bottom-up in a double for-loop.

Since we want to obtain not only the optimal score, but the (or an) optimal alignment, we store in each cell of the matrix a “pointer” (not in the programming sense)—the pointer value is left, up, or diagonal. In an actual program we could keep these values in a separate matrix, the same size as \( F \). These pointer values tell us which of the three choices was used to compute \( F[i, j] \). In case two of the choices yield the same value of \( F[i, j] \), we just pick one arbitrarily. (If we want to return all optimal alignments, we have to allow for retaining more than one pointer value.)

When the matrix has been computed, we read the optimal score as the lower right corner entry \( F[n, m] \). Note that \( F[n, m] \) does not have to be the maximum entry along the right side or the bottom of the matrix \( F \), but starting there would mean omitting one or more characters from the end of one of the strings. We are only allowed to insert gaps, not to omit characters, so we must start in the corner at \( F[n, m] \).

How do we recover the optimal alignment itself? That will be given by two strings \( u \) and \( v \), such that \( u \) is formed from \( x \) by inserting some gaps, and \( v \) is formed from \( y \) by inserting some gaps. We initialize \( u \) and \( v \) as follows: \( u = x[n] \) and \( v = y[m] \).

We now use a “trace-back” process, following the pointers back starting from the cell with the maximum score. A diagonal pointer adds a character (at the left) to each string. An up pointer adds a gap to \( u \) and a character at the left of \( v \). A left pointer adds a gap to \( v \) and a character at the left of \( u \). In class we carried out this process for the example given above.