Notes by Felix Wu

- Longest Common Subsequence (LCS)
- Global sequence alignment
  - Recursive algorithm
  - Memoizing
  - Dynamic programming
  - Space efficiency
- Gap models
- Local alignment
- Multiple alignment
- Biological applications

1 Longest Common Subsequence

Definition: A **subsequence** is any subset of the elements of a sequence that maintains the same relative order. If A is a subsequence of B, this is denoted by $A \subset B$.

For example, if $A = a_1a_2a_3a_4a_5$, the sequence $A' = a_2a_4a_5$ is a subsequence of A since the elements appear in order, while the sequence $A'' = a_2a_1a_5$ is not a subsequence of A since the elements a_2 and a_1 are reversed.

The Longest Common Subsequence (LCS) of two sequences A and B is a sequence C such that $C \subset A$, $C \subset B$, and $|C|$ is maximized.

Example:

$A =$ `abacdac`; $B =$ `cadcddc`; $C =$ `acdc`

```
_abacdac
 |  || |
cad_cddc
```

In this example, C is a LCS of A and B since it is a subsequence of both and there are no subsequences of length 5.

Applications:

- The UNIX `diff` command works by finding the LCS of two files, where each line is treated as a character. It then prints out lines which are not in the LCS, indicating insertions, deletions, and changes.

- For screens A and B, we can define the edit distance of A and B, $D(A,B) = |A| + |B| - 2|LCS(A,B)|$. The edit distance gives the number of characters in either screen which are not in the LCS, and hence measures the number of commands needed to change the screen from A to B. Maximizing the LCS minimizes the edit distance.

2 Global Sequence Alignment

The global sequence alignment problem is a generalization of LCS; instead of only noticing whether characters are the same or different, we have an arbitrary cost function which defines the relationship between characters. (In what follows, we will be maximizing "costs"; hence, the cost function is really a measure of similarity.)

Let $c(a,b)$ be a cost function, giving a value for any pair a and b of alphabet symbols, including the blank character (_). We can formally define an alignment of A and B as a pair of sequences A' and B' which are just A and B with added blanks, and for which $|A'| = |B'| = l$. The value of an alignment is given by

$$V(A', B') = \sum_{i=1}^{l} c(A'[i], B'[i]).$$

Our goal then is to find an optimal alignment, i.e. one whose value is a maximum.

Note that if we define costs so that $c(x,y) = 1$ if $x = y$, and 0 otherwise, we get LCS as a special case.

Example:

A = abacdac, B = cadcddc

```
A'  = _abacdac          A'' = aba_cdac
       |  || |                  | || |
B'  = cad_cddc          B'' = _cadcddc
```

Both of these alignments have the same value when we use the LCS cost function, but for the following cost function (which treats b's and c's as being similar), (A'', B'') has a higher value.

	a	b	c	d	_
a	2	0	0	0	0
b	0	2	1	0	0
c	0	1	2	0	0
d	0	0	0	2	0
_	0	0	0	0	-1

Notice the negative cost of two blanks. This is necessary to prevent unnecessary padding of sequences with extra blanks.

2.1 Cost Functions for Protein Matching

With proteins, we have an alphabet of 20 amino acids, and hence a cost matrix with 210 entries. (We assume the matrix is symmetric, since we have no reason to order the proteins we are comparing.) There are a number of possible cost functions:

- **Identity:** We could use the LCS cost function and view amino acids as just being the same or different.
- **Genetic code:** We can assign costs which are inversely related to the minimum number of DNA changes required to convert a DNA triple for one amino acid into a DNA triple for the other.
- **Chemical similarity:** We can take into account the relative sizes, shapes, and charges of different amino acids.
- **Experimental:** We could use standard matrices (e.g. Dayhoft, Blosum) which are based on observed properties of amino acids and/or their observed relative effect on tertiary structure.

In practice the cost function of choice is application specific and a topic of hot debate.

3 Sequence Alignment Algorithms

3.1 Recursive Algorithm

A recursive solution to the sequence alignment problem takes the last character of each sequence and tries all three possible alignments: either the characters are aligned together, the first character is aligned with a blank, or the second character is aligned with a blank. Taking the maximum over all three choices gives a solution. ML-style code for this would look like the following:

```
OptAlgn(_,_)     = C(_,_).
OptAlgn(_,A:a)   = C(_,a) + OptAlgn(_,A).
OptAlgn(B:b,_)   = C(b,_) + OptAlgn(B,_).
OptAlgn(B:b,A:a) = max(C(b,a) + OptAlgn(B,A),
                       C(b,_) + OptAlgn(B,A:a),
                       C(_,a) + OptAlgn(B:b,A)).
```

For the special case of LCS, all costs involving a blank are 0, and we know that it is always to our advantage to align characters if they match. Hence, we get the following code:

```
LCS(_,A:a)   = 0.
LCS(B:b,_)   = 0.
LCS(B:x,A:x) = 1 + LCS(B,A).
LCS(B:b,A:a) = max(LCS(B,A:a), LCS(B:b,A)).
```

A naive implementation of this recursive solution has an exponential running time, since each call spawns three recursive calls over which we take the maximum. This is easily rectified by memoizing, however.

3.2 Memoizing

If $|A| = n$ and $|B| = m$, note that there are only nm distinct calls we could make to `OptAlgn`. By simply recording the results of these calls as we make them, we get an $O(nm)$ algorithm. The code for LCS follows: (We assume the matrix M is initialized to `INVALID`.)

```
int LCS(int i, int j) {
  if (M[i,j] != INVALID) return M[i,j];
  if (i == 0 || j == 0) r = 0;
    else if (A[i] == B[j]) r = 1 + LCS(i-1, j-1);
    else r = max(LCS(i-1, j), LCS(i, j-1));
  return M[i,j] = r;
}
```

3.3 Dynamic Programming

Instead of filling in the matrix of values as they are computed, we can fill in the values row by row. This is the dynamic programming solution, for which we have the following code:

```
for i = 1 to n
  for j = 1 to m
    if (A[i] == B[j]) M[i,j] = 1 + M[i-1,j-1];
    else M[i,j] = max(M[i-1,j], M[i,j-1]);
```

For example, with $A =$ `tcat` and $B =$ `atcacac`, we get the following matrix:

	_	a	t	c	a	c	a	c
_	0	0	0	0	0	0	0	0
t	0	0	1	1	1	1	1	1
c	0	0	1	2	2	2	2	2
a	0	1	1	2	3	3	3	3
t	0	1	2	2	3	3	3	3

To actually find the optimal alignments, we can think of each entry as having edges to the neighboring entries from which it could have been calculated. That is, each entry can have edges pointing to its upper, left, and upper left neighbors, depending on whether it could have been calculated by adding one to its diagonal neighbor and whether its value is the same as either its upper or left neighbors. A portion of the induced graph for the matrix above looks like:

$$\begin{array}{c|ccccccc}
 & \mathrm{t} & & \mathrm{c} & & \mathrm{a} & & \mathrm{c} \\ \hline
\mathrm{t} & 1 & \leftarrow & 1 & \leftarrow & 1 & \leftarrow & 1 \\
 & \uparrow & \nwarrow & & & & \nwarrow & \\
\mathrm{c} & 1 & & 2 & \leftarrow & 2 & \leftarrow & 2
\end{array}$$

An optimal alignment is then a path from the lower right corner of the table to the upper left. In the matrix fragment above, the two paths correspond to aligning the `tc` as `tc_` or as `t_c`. Finding all possible paths gives us all optimal alignments. As with memoizing, this algorithm runs in $O(nm)$ time.

3.4 Space Efficiency

Our time bound is essentially the best possible for the general case[5], but we would like to use less space. In particular, notice that each row of the matrix depends only on the previous row. Hence, by filling in the matrix row-by-row and reusing the space for each row, we could compute the value of the optimal alignment using just $O(m)$ space. Without something clever, however, this does not give us an actual alignment since we throw away the path as we go (remember that we have to traverse backwards through the matrix to generate the alignment path).

(As an aside, note that if we were to run this algorithm in parallel, we would want to fill in the matrix diagonal by diagonal so that each entry would be independent of the others currently being worked on.)

In order to actually generate an alignment using only $O(m)$ space, we use the following divide-and-conquer algorithm:

1. Calculate the entries of the matrix row-by-row, discarding intermediate results, until we reach the middle row.

2. Once we get to the middle row as we calculate each additional row, we keep track of where in the middle row each new entry came from. (This is easily derived from the corresponding information about the previous row.) If we have more than one possibility, we choose arbitrarily. (Hence, this algorithm will only output one solution.)

3. When we reach the last row, we will know which entry in the middle row the bottom right corner came from (*i.e.* we will know the middle node along the solution path). Suppose that middle row entry is entry k. Now we recursively solve the problem of finding the path from the bottom right to $(n/2, k)$ and from $(n/2, k)$ to the upper left.

It may seem that this algorithm is doing redundant work since it recursively resolves the same solution over and over again. Although this is indeed true note, however, that in the recursion the algorithm only needs to recur on the upper left and bottom right quadrants. Hence, it is doing half as much work at each level of the recursion, leading to a constant factor increase in time, not a log factor as one might suppose. More formally, the recurrence we get is:

$$T(n, m) = T(n/2, k) + T(n/2, m - k) + O(nm) = O(nm).$$

[5]We will describe a more efficient method for small edit-distances in the next class.

Since we only keep track of an extra set of pointers one per column, this algorithm uses only $O(m)$ space. (A slight optimization is to orient our matrix so that m is the smaller of the two dimensions.) The algorithm was invented in the context of finding the longest common subsequence by Hirschberg ("A linear-space algorithm for computing maximal common subsequences", Communications of the ACM, 18:107–118, 1975) and generalized to alignment by Myers and Miller ("Optimal alignments in linear space", Computer Applications in the Biosciences, 4:11–17, 1988).

4 Gap Models

For many applications, consecutive indels (insertions or deletions, also called gaps) really need to be treated as a unit. In other words, a gap of 2 characters isn't really twice as bad as a gap of 1 character. There are various ways of dealing with this problem, depending on what sort of gap model we want to allow.

In the most general case, we can define arbitrary costs x_k for gaps of length k. For this we have the Waterman-Smith-Beyer algorithm:

$$\begin{aligned}
S_{00} &= 0 \\
S_{i0} &= x_i \\
S_{0j} &= x_j \\
S_{ij} &= \max \begin{cases} S_{i-1,j-1} + C(a_i, b_j) \\ \max_k(S_{i,j-k} + x_k) \\ \max_k(S_{i-k,j} + x_k) \end{cases}
\end{aligned}$$

This is similar to our previous algorithms, except that for each entry, instead of only looking at gaps of length 1, which corresponds to looking at the left and upper neighbors in the matrix, we look at all possible gap lengths. To calculate S_{ij} this requires us to look at the entire row i and column j generated so far. Hence, the running time of the algorithm is $O(nm^2 + n^2m)$. Furthermore, since we need all previous rows, we cannot make this algorithm space efficient.

In practice, our gap models are likely to be relatively simple, and for these simpler models, we can again achieve the time/space bounds we were able to get before. For example, with an affine gap model, we have $x_k = \alpha + \beta k$. (This is popular in computational biology, where something like $x_k = -(1 + k/3)$ is typical.) An algorithm due to Gotoh for the affine gap model is as follows:

$$\begin{aligned}
E_{ij} &= \max \begin{cases} E_{i-1,j} + \beta \\ S_{i-1,j} + \alpha + \beta \end{cases} && E_{0j} = -\infty \\
F_{ij} &= \max \begin{cases} F_{i,j-1} + \beta \\ S_{i,j-1} + \alpha + \beta \end{cases} && F_{i0} = -\infty \\
S_{ij} &= \max \begin{cases} S_{i-1,j-1} + C(a_i, b_j) \\ E_{ij} \\ F_{ij} \end{cases} && \begin{aligned} S_{0j} &= \alpha + \beta j \\ S_{i0} &= \alpha + \beta i \end{aligned}
\end{aligned}$$

The E_{ij} matrix calculates optimal alignments in which the second sequence ends in blanks, the F_{ij} matrix calculates optimal alignments in which the first sequence ends in blanks, and the S_{ij} matrix calculates overall optimal alignments. The basic idea is that since each additional blank (after the first one) has the same cost, we can essentially use the same algorithm as before, just using the E and F values to be sure we add in α when we introduce the first blank in a gap. This algorithm runs in $O(nm)$ time and can be made space efficient as before.

The following is a summary of some different gap models and the running times of their associated algorithms.

Gap function form	**Running time**
General	$O(nm^2 + n^2m)$
Linear ($x_k = \alpha + \beta k$)	$O(nm)$
Logarithmic ($x_k = \alpha + \beta \log k$)	$O(nm)$
Concave downwards	$O(nm \log n)$
Piecewise linear, l segments	$O(lnm)$

5 Local Alignment

Sometimes instead of aligning two entire sequences, we want to align a smaller sequence at multiple locations within a larger one. For example, to find areas of DNA which might have similar functionality to some segment, we can find the best matches to our segment within the DNA.

Global alignment algorithms are easily converted into local ones by simply adding an extra argument 0 to the max function computing the optimal alignment. So, for example, for the general gap model

$$S_{ij} = \max \begin{cases} S_{i-1,j-1} + C(a_i, b_j) \\ \max_k(S_{i,j-k} + x_k) \\ \max_k(S_{i-k,j} + x_k) \\ 0 \end{cases}$$

This has the effect of not penalizing an alignment for the large gaps at the ends of the sequence. For example, with

$$C(a,b) = \begin{cases} 1 & \text{if } a = b \\ -1/3 & \text{if } a \neq b \end{cases}$$

and $x_k = -(1 + k/3)$, we get the following alignment matrix.

	_	a	t	c	a	c	a	c
_	0	0	0	0	0	0	0	0
t	0	0	1	0	0	0	0	0
c	0	0	0	2	2/3	1	0	1
a	0	1	0	2/3	3	5/3	2	2/3
t	0	0	2	2/3	5/3	8/3	4/3	5/3

In this case, the best match aligns `tca` with `tca` (the one 3 in the matrix). In general, we might want to find multiple matches with high scores. In this case we might pick out the highest k entries from the matrix, although we probably don't want to include entries that are just extensions of previous matches we have found. In the above example we might not want to pick the value 8/3 that follows 3, but instead pick the 2.

6 Multiple Alignment

Another problem is to align multiple sequences at once so as to maximize the number of pairs of aligned symbols. This is used to reconstruct a sequence from its overlapping fragments; the overlaps will not be perfect because of imprecision in the sequencing of the fragments. Alternatively, we can also use multiple alignment with sequences from different family members, in order to find genes shared by the family members.

Unfortunately, multiple alignment is NP-hard. Hence, there are two types of algorithms: those that take exponential time and those that are approximate. In exponential time, we can extend our earlier dynamic programming algorithms to work in a p-dimensional matrix, where p is the number of sequences we are trying to align. This takes $O(n^p)$ time and space and is impractical for p more than about 4.

Alternatively, we can extend the pairwise methods hierarchically to get an approximate solution. For this, we do all pairwise comparisons, cluster the results, and then build an alignment bottom-up from the cluster tree.

7 Biological Applications

One of the most intensive uses of pattern matching is in database search engines used by biologists to compare new sequences with databases of old ones. These databases can have as many as 100,000s of sequences with querying by web or email. Such shared search engines are useful because algorithms and data are updated more rapidly and more centrally, and considerable computing power can be brought to bear on the problem. Both local and global alignment algorithms are used.

On a smaller scale, pattern matching is also used for genome sequencing. This can be done in one of two ways. The first method involves sequencing the 600 bp (base pair) ends of the fragments of a sequence, which are then assembled using multiple alignment. Alternatively, one piece can be fully sequenced, and the rest of the sequence can be constructed outward using single alignment.