BIOL591: Introduction to Bioinformatics
Alignment of pairs of sequences

Reading in text (Mount *Bioinformatics*):

I must confess that the treatment in Mount of sequence alignment does not seem to me a model of clarity. Nonetheless, the following sections might be of value. If they don't do much for you, however, don't worry.

- pp.300-309: Description of Blast and filtering
- pp. 53-57: Overview of sequence alignment
- pp. 64-72: Description of Smith-Waterman algorithm
  The text's web site gives a more helpful example of the algorithm in action. If you have a copy of the text, go to http://www.bioinformatcsonline.org and register yourself using the ID number on the inside cover of your book. Once inside, click on chapters, chapter 3, page 1.

Outline:

I. Relationship between Scenario 3 and sequence alignment
II. Different ways of aligning sequences
III. The Smith-Waterman algorithm for local alignments
IV. Word-based algorithms (FastA and Blast)

I. Relationship between Scenario 3 and sequence alignment

Take a look at Scenario 3 and DO it if you haven't already. In this scenario, to find out what a DNA sequence might be and where it might have come from, you ran it through BlastN. One might imagine that BlastN looks up sequences and tells you whether they've been seen before. This is not the case. All it knows how to do is attempt to match your sequence against a database of other sequences and pass on to you those sequence that are the best matches. How you interpret that information is up to you.

How does BlastN decide that two sequences match? The program declares a match if it can *align* the two sequences according to certain criteria. Of the two pairs of sequences below, it seems pretty obvious that the first pair match well (identity fraction of 16/17) and the second pair match not much better than what you'd expect from two random sequences (identity fraction of 6/21).

```
a. AATATTGACGCTTTACT                b. TTTACTACATCAGTCCATCGG
   |||||| ||||||||||                    |   | |    | |    |
   AATATTCACGCTTTACT                   GTACTTCCGATGGACTGATGT
```

But the latter pair actually isn't a bad match. If you invert the second sequence and shift it a bit you get:

```
b. TTTACTACATCAGTCCATCGG
         |||||||||||||||
         ACATCAGTCCATCGGAAGTAC
```

Clearly we need some machine help, to contort the sequences in all possible ways and to do this for the trillion or so nucleotides of sequences within GenBank.

II. Different ways of aligning sequences

Mount slices up sequence alignment in several ways:

a. Alignment of pairs of sequences vs multiple sequence alignment
   Alignment of pairs of sequences (which is what Blast does) is a much easier problem than multiple sequence alignment. The former is useful in identifying sequences. The latter is useful in finding conserved regions in a set of similar sequences. We'll consider here only pairwise alignment.

b. Global alignment vs local alignment
   Global alignment compares two sequences throughout their lengths. This is useful if you have reason to believe that the two sequences ought be similar from one end to the other. This is clearly not the case when you're comparing a short DNA sequence against an entire genome, and it is seldom the case with protein comparisons either. We'll consider here only local alignment, in which the ends of the two sequences being compared are not forced into the alignment.

c. Dot matrix analysis vs the dynamic programming algorithm vs word methods

   – Dot matrices are very useful for seeing at a glance regions of similarity between two sequences. However, computers don't know how to glance, and so the method requires a human to scrutinize every comparison. Since this makes the method unusable for mass comparisons (e.g. a sequence against GenBank), we'll not consider it further here.

   – Dynamic programming is a mathematical technique designed to aid in decision making by considering the last decision and working backwards. That's all you're going to hear on the subject in this course.[1] The Smith-Waterman algorithm for local alignments, discussed below, is based on dynamic programming. The algorithm is guaranteed to give the best local alignment between two sequences.

   – Word methods, discussed below, are short cuts designed to allow the Smith-Waterman algorithm to run in a reasonable length of time. Blast and FastA use such methods. The downside is that the algorithm is no longer guaranteed to find the best match. But it almost always does anyway.

III. The Smith-Waterman algorithm for local alignments

The Smith-Waterman algorithm examines every possible alignment of a sequence with respect to another, scoring them according to a set of penalties for mismatches and gaps and a reward for matches. The algorithm proceeds in two stages. First, two sequences are considered from left to right, producing a table of scores of possible alignments. Second, the two sequences are considered from right to left, starting with the match that ended with the highest score and retracing the steps that led to that match. The details are most easily understood by example.

Suppose you want to find the best alignment of query sequence within a subject sequence:

Query: `AGATACCTACA` **Subject$_1$:** `TTAGATAAGCCTAGAG`

[1] You can find out more (and everything I know on the subject) by going to http://plus.maths.org/issue3/dynamic/

We'll use the (rather atypical) scoring parameters of:

Match reward = +1 Gap opening penalty = -3
Mismatch penalty = -2 Gap extension penalty = -1

For example, with these scoring parameters, the alignment:

```
  AAGATA--CCTACA
   |||||  |||| |
TTAGATAAGCCTAGAG
```

would produce the score of:

(10 matches x +1) + (1 mismatch x –2) + (1 gap opening x –5) + (1 gap extension x –2)

= +1

(note that the alignment begins with the first match shown, not at the beginning of the sequence). Negative scores are not allowed: any score below zero is set to zero.

The scoring table for this comparison would look as shown in Table 1 (next page). Note that the first row and first column consist solely of one's and zero's because at the beginning of a comparison, the score is either one (the two characters match) or zero (they don't). The score for all other boxes is taken from the maximum of three choices:

- **a.** No gap: Take the score in the box diagonal and to the upper left and either add a match reward or mismatch penalty, as appropriate (or zero if the sum is negative).
- **b.** Gap in query: Take the score in the box to the left and add a gap opening penalty or a gap extension penalty, as appropriate (or zero if the sum is negative).
- **c.** Gap in target: Take the score in the box above and add a gap opening penalty or a gap extension penalty, as appropriate (or zero if the sum is negative).

From two examples you might be able to see how the scores are figured out.

- (rule **a**) The score of two in the eighth row and second column arises because the match of **A** with **A** is preceded by another match with no gap. The red line highlights the connection.
- (rule **c**) The score of 2 in the eighth row and sixth column arises because of the insertion of a gap in the target (note that the match between **A** and **A** at that position is irrelevant: the **A** in the target is already used).

Once the scoring table is filled out, the first stage of the algorithm is complete.

SQ1. Fill out the scoring table to the right, using the same scoring parameters in the preceding example. Put in both scores and connecting lines.

SQ2. Make just one change in the query sequence used in Table 1 so that the first long diagonal match (AGATA) is connected to the second long diagonal match (CCTA).

	G	**C**	**A**	**A**	**T**
G					
C					
A					
A					
G					
T					

The second stage of the algorithm produces the best match. Start with the box that has the highest score (highlighted in green in Table 1). From that box, follow the lines backwards until a zero is reached. From that path, read upwards to get the query sequence and leftwards to get the target sequence. Any time a horizontal line is encountered, insert a gap in the query sequence, and any time a vertical line is encountered, insert a gap in the target sequence.

The match found from the scoring table in Table 1 would thus be:

```
Query:  AGATA
        |||||
Target: AGATA
```

Table 1: Scoring table for query vs $subject_1$

	A	A	G	A	T	A	C	C	T	A	C	A
T	0	0	0	0	1	0	0	0	1	0	0	0
T	0	0	0	0	1	0	0	0	1	0	0	0
A	1	1	0	1	0	2	0	0	0	2	0	1
G	0	0	2	0	0	0	0	0	0	0	0	0
A	1	1	0	3	0	1	0	0	0	1	0	1
T	0	0	0	0	4	0	0	0	1	0	0	0
A	1	1	0	1	0	5	0	0	0	2	0	1
A	1	2	0	1	0	2	3	0	0	1	0	1
G	0	0	3	0	0	0	0	1	0	0	0	0
C	0	0	0	1	0	0	1	1	0	0	1	0
C	0	0	0	0	0	0	1	2	0	0	1	0
T	0	0	0	0	1	0	0	0	3	0	0	0
A	1	1	0	1	0	2	0	0	0	4	0	1
G	0	0	2	0	0	0	0	0	0	1	2	0
A	1	1	0	3	0	1	0	0	0	1	0	3
G	0	0	2	0	0	0	0	0	0	0	0	0

SQ3. Would the change you proposed in answer to SQ2 produce a higher score than 5?

SQ4. Make *two* changes in the query sequence used in Table 1 to produce a higher score.

SQ5. What would be the effect on the original Table 1 of changing the open-gap-penalty from 3 to 2? Based on this observation, how would you modify the claim that the Smith-Waterman algorithm finds "the best local alignment between two sequences"?

SQ6. How big would the scoring table have to be to accommodate a comparison of a 1000-nucleotide sequence and the 4.6 million nucleotide genome of *E. coli* K12?

Your answer to SQ6 should highlight the problem with the Smith-Waterman algorithm. The requirement for an *m*x*n* matrix (where *m* is the length of the query sequence and *n* is the length of the target sequence) is overwhelming for large sequences.

IV. Word-based algorithms

The Smith-Waterman algorithm gives the best local match (according to the specific set of parameters used), but it does so by demanding huge chunks of computer memory and time, making the method impractical for use in comparisons of sequences with large databases.[2] Routine searches demands an alternative method to reduce these requirements. Two programs, Blast and FastA, take similar shortcuts to complete searches about 1000-times faster than the Smith-Waterman algorithm. The main trick comes from the realization that the majority of the scoring matrix calculated by Smith-Waterman is stray zeros and ones of no use to the ultimate best match. Blast and FastA attempt to calculate only those parts of the matrix that bear on the question of how far a good match should extend.

---

[2] Recently, tricks have been used to enable the Smith-Waterman algorithm to run on very fast computers to do exhaustive searches of databases, catching some hits that Blast misses, but this facility is not available to the general public.

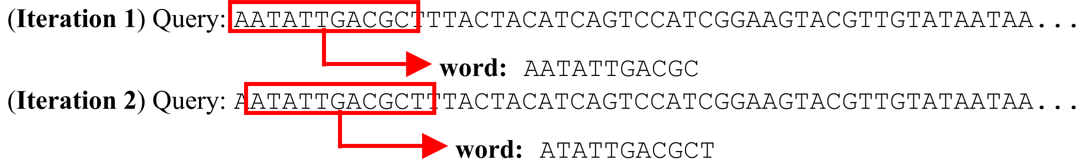

Fig. 1: Extraction of words from query sequence, beginning from the beginning of the sequence and proceeding by sliding the window to the right, to the end.

The algorithm used by Blast and FastA proceeds by the following steps:

1. Filter the query sequence to remove repetitive regions (FastA doesn't do this and it is optional in Blast). Filtering is explained at the end of this section.
2. Find all matches between the query sequence and the target sequence In Perl-speak: *foreach* ***word*** *starting (beginning of query . . end of query) {*
   a. Extract a subsequence from the query, called a **word**, by sliding a window along the length of the query, as shown in Fig. 1.

   b. Find an *exact* match of the word in the target sequence. If no match is found, go back to Step **a** and get the next word from the query. If a match is found, continue to Step **c**.

   c. Use a modified Smith-Waterman algorithm to extend the word match in both directions, according to a set of reward and penalties.

   d. Calculate a score related to the probability of finding a match as good or better than the final match.

   e. Save those matches whose scores are better than a given threshold.

   f. Repeat Steps **a** through **e** until there are no more words remaining to try within the query.
3. Rank the matches by their scores.
4. Print out the top matches.

This procedure saves an enormous amount of memory and calculation time, as shown in Figure 2. The attempt to extend a word-match proceeds only until a zero score is found in a cell. Thus the number of cells with scores that need to be calculated is bounded by 0 scores. Of course it is possible that a good match might be missed by the trick of searching first for exact matches. One can guard against this possibility by reducing the size of the word, but that increases the number of word-matches and slows down the search. At the limit case, where the word size is one nucleotide, the procedure is essentially the same as the full Smith-Waterman algorithm.

SQ7: What word size would you use in order to detect the match shown in Table 1?

Blast filters queries before extracting words from them. This is to guard against the large number of spurious matches that usually result from searches using queries containing regions like:

AAAAAAAAAA... or CACACACACACA...

Which are found in biological DNA sequences in frequencies higher than one would find in a random sequence. Such regions are said to have low complexity. Blast accomplishes filtering by

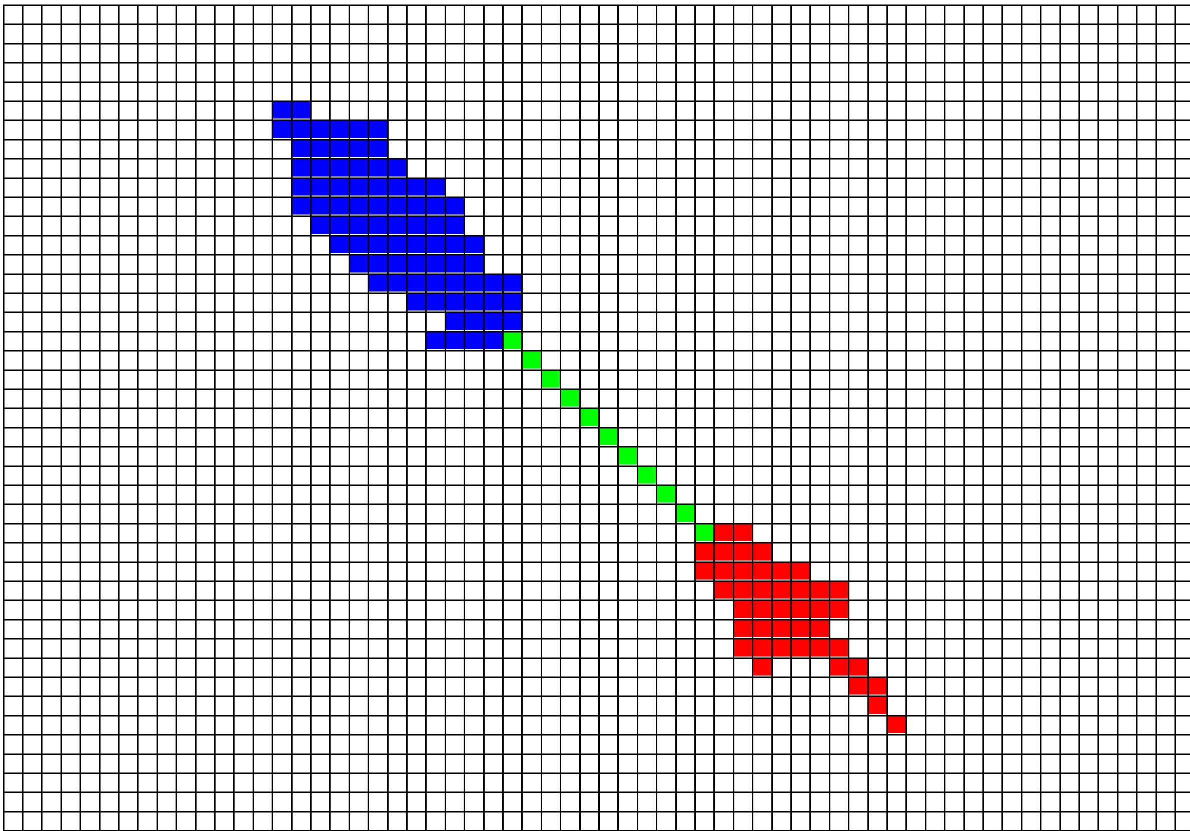

Fig. 1: Cells of Smith-Waterman scoring table that need to be calculated. Shown is just a tiny portion of an *m*x*n* scoring table of a query *m* nucleotides long compared to a target *n* nucleotides long. The 11-nucleotide word match is highlighted in green. Scores for red cells and blue cells are calculated to determine how far the match may be extended in the forward direction and backward directions, respectively. Since 11-nucleotide exact matches are quite rare, a very small fraction of the table must be calculated.

masking out low complexity regions, replacing the nucleotides with "N", which match nothing in the target sequence. Blast can also filter for common repetitive sequences in mammalian DNA. By default, Blast filters queries, but it is possible to turn filtering off or specify special filtering.

SQ8: I took from GenBank a random piece of noncoding human DNA and used BlastN to find similar sequences. You do the same thing:

a. **Get to BlastN (nucleotide vs nucleotide) in the NCBI site (see Links page in course website).**
b. **Put in the Search box the accession number AF397423.**
c. **Type 2041 in the From: box and 5040 in the To: box.**
d. **Submit this sequence in three ways:**
    1. **On the Choose filter line, specify Low complexity (default); click on BLAST.**
    2. **On the Choose filter line, specify Human repeats; click on BLAST.**
    3. **Remove all checks from the Choose filter line; click on BLAST.**

e. Compare the output: What matches were found in one search but not another? Why?
(More on this later!)