Math 2300 **DISCRETE MATH** Fall 2015
Dr. Duval **Project 2–REVISED**
due Wednesday, October 28

Sequence alignment

In this project, we'll look at the Smith-Waterman algorithm to find alignments in sequences of symbols. Though there are more powerful algorithms now available, Smith-Waterman is relatively simple, and still effective.

Recall from Project 1 that it is important to compare strings of DNA, the genetic code, from one organism to another, and that we can consider DNA to be strings of letters chosen from the alphabet $\{A, C, G, T\}$. Often the same string of DNA will repeat, but, through years of evolution, some errors may have crept in, and two strings will not match exactly, which will make such comparisons harder.

If the only errors were the replacement of one letter by another (for instance, changing $GACCTGAG$ to $GACGTGAG$ by changing the second C to a G), it would not be hard to find such stretches of very good matches; in this case, we could just count the number of matching letters over every interval, and look for stretches with many such matches. The problem is that sometimes the error is instead an "indel" (from Project 1), inserting or deleting a letter, for instance changing $GACCTGAG$ to $GACCGTGAG$ by inserting an extra G before the T. If we simply line up these two strings, and match letters, the extra G ruins the potential matches of all the subsequent letters in the string:

G	A	C	C	T	G	A	G	–
G	A	C	C	G	T	G	A	G

The extra spaces introduce the difficulty of knowing where to line up the strings to make our comparisons. Although it's clear to us, in this small example, what happened, it's harder to spot in longer strings (hundreds of letters long), with more errors. Furthermore, we'd like to do this analysis systematically, so that a computer could do it for us.

One simple but effective approach is to assign a score to a potential alignment by rewarding spots where the strings match, and penalizing spots where the strings disagree, or where spaces have to be inserted into one string or another, to keep up the alignment. However, there are so many possibilities of where the spaces might be inserted that a brute force search through all the possibilities, looking for the best score, is infeasible (even with a computer – remember the strings are very long). The Smith-Waterman algorithm is a relatively simple and elegant recursive way to find the best score in long strings of letters.

We'll get to Smith-Waterman soon, but first we'll have to agree on the details of our scoring. There is no universal rule for how much to reward matches, and how much to penalize insertions or mismatches, but for this assignment, we'll use a simple rule, rewarding matches by $+3$, penalizing insertions by -2, and mismatches by -1 (much more complicated rules are possible). So, for instance, our strings above, with the alignment shown would have a score of 6:

G	A	C	C	T	G	A	G	–
G	A	C	C	G	T	G	A	G
+3	+3	+3	+3	-1	-1	-1	-1	-2

The better alignment

G	A	C	C	–	T	G	A	G
G	A	C	C	G	T	G	A	G
+3	+3	+3	+3	-2	+3	+3	+3	+3

gives a score of 21, much better.

The Smith-Waterman algorithm finds these good matches, even when they are more hidden than the example above. We'll illustrate each step with a small example, comparing the strings $CACTT$ and $AGTGT$. We start by building a matrix M whose columns are indexed by the first string, which we denote by x, and whose rows are indexed by the second string, which we denote by y. We also put in an extra space at the beginning of each string, in order to simplify the description of the algorithm.

We will fill in each entry $M(i, j)$ of M with the best score we can get with some segment ending with an alignment at the corresponding spots in the two strings. We determine this best score by first figuring out the score of nearby previously filled-in entries in M. In particular, each entry $M(i, j)$ is the **maximum** of the following four quantities:

- 0 [in other words, never let the score get negative, since, in the worst case, you could simply start the alignment from scratch at any point];
- $M(i-1, j-1) + s$, where $s = 3$ if $x_i = y_j$, but $s = -1$ if $x_i \neq y_j$ (look at the entry immediately diagonally above and to the left of the current entry);
- $M(i-1, j) - 2$ (look at the entry immediately above of the current entry);
- $M(i, j-1) - 2$ (look at the entry immediately to the left of the current entry);

Also, to get the matrix started, the first row and column is all 0's, since matching up the first letter of either string with a space will give a score of 0 (but allowing this space is what lets us start the strings at different places).

In our example the resulting matrix is the following

	–	C	A	C	T	T
–	0	0	0	0	0	0
A	0	0	3	1	0	0
G	0	0	1	2	0	0
T	0	0	0	0	5	3
G	0	0	0	0	3	4
T	0	0	0	0	3	6

The 3 in the 3rd column of the 2nd row is from adding 3 to the 0 entry above and to the left of it, since both strings have an A in that position. The 1 immediately to its right comes from subtracting 2 from the 3 entry immediately to its left (the 3 we just filled in), since the first string has a C in that position, and the second string has an A in that position. The entry to the far right of this same row (6th column of 2nd row) is 0 because looking at the three entries immediately above, to the left, and diagonally above and to the left of it are all 0's, and strings disagree in the corresponding positions (T in the column, A in the row),

so they would all give a negative number. But the rules say that, in this case, the entry is simply 0.

After the first row and column are filled with 0's, the rest of the entries can be filled in row by row, top to bottom, and left to right within each row (or column by column), as long as each entry is not filled out until the three entries immediately above, to the left, and diagonally above and to the left of it are all filled in.

Once the matrix is done, we identify the largest number in the matrix (in this case, 4 in the lower-right), and trace it backwards from where it came from. In this case, tracing it back yields the following entries:

	–	C	A	C	T	T
–	0	0	0	0	0	0
A	0	0	**3**	1	0	0
G	0	0	1	**2**	0	0
T	0	0	0	0	**5**	3
G	0	0	0	0	**3**	4
T	0	0	0	0	3	**6**

Every step back we take that goes diagonally means we align the strings there (even if there is a mismatch); every step that goes up or to the left means one of the strings puts in a space (to preserve the alignment). In this case, we get the alignment

A	C	T	–	T
A	G	T	G	T

1. Perform the Smith-Waterman algorithm on the following pairs of strings (each pair is a new problem). Show your steps clearly. You must use Smith-Waterman to receive any credit.

   (a) $CTAG$ and $GCTG$

   (b) $ACCTAG$ and $CCATCG$

   (c) $CGACCGGTAGT$ and $AGCGCTGTCTG$

2. Explain why the Smith-Waterman algorithm is **recursive**.