BIOL591: Introduction to Bioinformatics
What's wrong with BlastN?

I. Interlude: Progress in solving the *Mystery of the Missing Match*

I.A. Local *BlastN* vs NCBI's *BlastN*

Friday in class we discovered the following:

1. We have in our hands a working program *BlastN*, written in Perl, that at least in some respects acts like *BlastN* implemented by NCBI.
2. The parameters set in the local program are the same as those set in the NCBI implementation (except for *gap dropoff* and *Expect threshold*)
3. The main loop of the program sequentially extracts words from the query sequence and finds matches between them and the target sequence.
4. A subroutine called within the main loop is responsible for constructing the scoring tables necessary to extend the exact match in both directions.
5. The local implementation differs from the NCBI implementation in at least the following particulars:
    a. The local *BlastN* does not filter the query sequence
    b. The local *BlastN* does not calculate a score related to the probability of finding a match as good or better than the match found.
    c. Therefore, the local *BlastN* does not throw away or rank sequences based on this score.
6. Unlike NCBI-implemented *BlastN*, the local program printed several matches when DG47 (the sequence of the PCR product) was blasted against M29081 (bona fide *lef* gene). All the matches were very short, except for one that extended the length of DG47.

I.B. Mysteries to be solved

1. There remains the main mystery: Why can't NCBI *BlastN* find the similarity between DG47 and *lef* that we can readily see by eye?
2. Now we have a new and possibly related mystery: Why does *BlastN* implemented by NCBI give different output from homegrown *BlastN* ?

There was a suggestion on Friday that perhaps the filtering performed by NCBI's version could account for the different output.

SQ1. Test whether filtering is the key difference in two ways:

a. Compare DG47 and *lef* via NCBI's pairwise *Blast*, after disabling filtering.

b. Compare DG47 and *lef* via local *BlastN* with filtering. You could teach the program how to recognize and mask repetitive sequences, but for our purposes that's way too much work. Find the repetitive sequence yourself by eye and then edit the query to replace the repetitive sequence with random nucleotides. Use this modified DG47 also in a pairwise Blast search using NCBI's *Blast*.

In both cases, is the output of the two programs now the same?

I.C. How to solve the ultimate mystery (of *BlastN*, I mean)

Filtering indeed makes a big difference in certain cases what output you get from *BlastN*. But even accounting for filtering out regions of low complexity, the main mystery still remains. I would ***still*** consider DG47 and *lef* a good match, even if I excluded regions of low complexity. Yet *BlastN* does not, neither NCBI's nor our own.

SQ2. Think seriously on why *BlastN* does not find the match. For starters, why do *you* think it *should* find a match?

a. **Extract a small region of DG47 and *lef* that *you* think should match and use those regions as query and subject. Does our *BlastN* still fail to find a match?**

b. **Make yourself into *BlastN*, and by hand go through all the steps of the *BlastN* algorithm using the small query you invented in 2a to search through the small target sequence. Do *you* still find a match?**

I.D. How to account for the extra sequences found by homegrown *BlastN*

When you accounted for filtering, you should have discovered that there are still matches found by local *BlastN* that are not found by NCBI's implementation. Why is that? Consider the steps of the *BlastN* algorithm:

1. Read the query and target sequences
2. (optional) Filter the query sequence to remove regions of low complexity
3. Find all query-target matches as follows:
    a. Extract a word from the query, using a sliding window
    b. Find an exact match of the word in the target sequence
    (if no match: Return to Step 3a)
    (if a match, continue)
    c. Extend match in both directions
    d. Calculate an E-value for the final match
    e. Save matches whose E-values are better than a given threshold
    f. Repeat steps 3a-3e until all possible query words have been tried
4. Rank the matches by their E-values
5. Print out the top matches

SQ3. Which of these steps are performed by our homegrown *BlastN*?

Certainly one major difference is that our *BlastN* does not calculate E-values. Consider how you would go about doing this – what you need to know and where in the program you need to know it.

SQ4. Add a line to the program that calculates the E-value for a match and then prints it. To do this, you will need to know (amongst other things) values for lambda and K. Grab those values from the output from a Blast search you do at NCBI. For now, don't bother saving the E-values or think about ranking matches based on E-values.

SQ5. After running your program, resolve the discrepancy between the different output of our program and NCBI's.

II. Does local *BlastN* make scoring tables properly?

I overheard a conversation between Jen and James regarding the meaning of *gap-extension penalty*. The issue, I think (I didn't hover over them to find out for sure), was whether the gap extension penalty was exacted on all mismatches or only starting from the second one. The Smith-Waterman method charges a gap-opening penalty for the first gap and a gap extension penalty for each *additional* gaps, but the online documentation for *Blast* does not make clear whether Blast computes penalties in this way or some other way. An alternative approach is to charge a gap-opening penalty *and* a gap extension penalty (or more accurately a gap-presence penalty) for the first gap and just a gap extension penalty for each additional gap. These two choices are illustrated in Table 1.

Table 1: Comparison of methods to calculate gap penalty

Method	**Formula for gap penalty**	**Example: penalty for** `AGGC` open → -5 `T--G` ext → -2
Smith-Waterman	Gap_opening + gap_extension · (gap_size-1)	7
Alternative (Blast?)	Gap_opening + gap_extension · gap_size	9

Which method does NCBI *BlastN* use? We'll be able to answer this better after a discussion of scoring, but we can answer right now what method local *BlastN* uses.

SQ6. One way to tell is to examine the program and look for the code that calculates gap penalties. Find that part of the program. Where is it and which method does the program use?

SQ7. Another approach is to print out the scoring matrix and examine the values in adjacent boxes. The notes for Wed Oct 8 describes how to write a subroutine to print out some values of the scoring matrix. First of all, where in *BlastN* would you put a call to this subroutine (print_score) so that the values printed out are meaningful?

SQ8. Complete the subroutine print_score started for you in Wednesday's notes, insert it at the end of *BlastN*, and call it at the appropriate time (determined in SQ3).