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

I.A. Local BlastN vs NCBI’s BlastN
In the last class we discovered the following:

1. We have in our hands a working program BlastN, written in Python, 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.

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 left 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 left 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?

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 left via NCBI’s pairwise Blast, after disabling filtering.
   b. Compare DG47 and left 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
4) (if no match: Return to Step 3a)
5) (if a match, continue)
   a) Extend match in both directions
   b) Calculate an E-value for the final match
   c) Save matches whose E-values are better than a given threshold
   d) Repeat steps 3a-3e until all possible query words have been tried
6) Rank the matches by their E-values
7) 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?

We discussed the meaning of gap-extension penalty. In our initial crude Smith-Waterman program only had one gap, regardless of whether it was a gap opening or a gap extension for each additional gap. 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 (in the same direction). Examination of the BlastN code indicates that there are separate arguments to the __init__ method

SQ6. Even though the __init__ method does indeed have variables (gap_open & gap_extend) to keep track of these separate concepts, does the remainder of the code follow through and correctly make use of these? Where in the code would look to know? What did you find that resolve this?

SQ7. Assuming that there is a problem with how these variables are used, what would you need to do to decide whether a gap is being opened or extended. Which data structure(s) in the code would you need to query to figure this out.

SQ8. If you are able, modify the program to appropriately incorporate the concepts of gap opening and gap extension.