CSCI 311 - Fall 2022:
Programming Project: DNA Sequence Matching

Brian King Edward Talmage

Due: **On gradescope and in demo** by November 4, 2022
Based in part on problems in CLRS and other sources

Instructions

This is a group project. You have received an Overleaf share invitation to your group's report, and can find group members there. You will collectively submit one version of your solution. Be certain that the student who uploads your solution then adds all group members to the submission on gradescope.

1 Overview

DNA sequencing has made enormous technological strides in recent years. Once something reserved only for advanced multi-million dollar research, sequencing is now available for the public at a small fraction of the cost it once was. Moreover, research labs are increasingly sequencing a multitude of different organisms and viruses to aid in their research. The result is that we have an enormous amount of genetic sequence data available to aid in understanding practically any organism or virus you can think of. However, the big challenge is that only a very small portion of this data has experimental, lab-confirmed evidence to properly characterize the sequences with respect to their biological function.

How do researchers characterize a newly discovered genetic sequence? The process is actually based on a sound, long-standing principle known as the *Central Dogma of Molecular Biology*. It generally states that there is a very tight, highly conserved relationship between the sequence of a gene, the structure of the protein that the gene produces, and likewise the function that the protein performs in the organism. In other words, the sequence of the gene predicts the function of the protein! What does this mean for us? Simple: if you have a new, uncharacterized sequence, find a highly similar sequence with known structure and function, and there's a good chance your new sequence is similar, we can use the information from the known sequence to annotate the new sequence! This is known as **sequence similarity**.

Simple! Right?

Well, there are some complications. First, databases today are huge, but we're going to ignore that. The challenge for us lies in the search space of biological sequences. (Humans have well over 20,000 genes that work together to carry out all biological functions in our body! Even some tiny fleas have more than 30,000 genes!) The real challenge lies in the algorithm for assessing sequence similarity between two sequences.

Biological sequence analysis is a field in the broad field of *bioinformatics* and *computational biology* that focuses on the development of methods to analyze DNA and protein sequence data. The core of biological sequence analysis focuses on the development of methods to assess *similarity* between two arbitrary biological sequences. The reason is clear–as we said above, a new sequence that has high similarity with a known sequence is very likely to have similar (or even identical) function. Therefore, it is imperative that biological and biomedical researchers have access to highly accurate, efficient tools to assess sequence similarity.

How does this work in practice? Suppose a lab obtained a new gene sequence, and they have no idea what its function is. Researchers have access to multitudes of public databases that contain well-known, well-characterized sequences. So, what do they do? They search these databases for the most similar sequence to their target sequence of interest. The best sequence(s) is(are) used to hypothesize the function of their new sequence, and they can then test these hypotheses in the lab. The pseudocode looks like this:

Clearly the heart of your work is going to go into the algorithm you write for the *computeSim* function to compare two arbitrary DNA sequences, and return a quantitative (numeric) measure that best captures their similarity.

The field of computer science has made great strides in helping researchers get more accurate results on this problem and doing it efficiently. In the real world, these databases can contain millions of sequences to be searched! And consider that, on average, the typical DNA sequence for a gene can be about 1000 nucleotides, with some as long as 10,000 nucleotides or more. (If you are comparing entire chromosomes, or even genomes, then you are comparing sequences that can be millions of nucleotides or longer! We'll stick with comparing individual genes.) Moreover, it's not merely a matter of lining up two strings and counting the letters that are similar! How would you do that in a meaningful way when the strings have different length? That's the challenge. Fortunately we're pretty good at accurately representing a real DNA sequence in a computer.

Algorithm 1 Pseudocode to find the sequence in D that is most similar to t

```
1: function FindMostSimilarSeq(t: a sequence, D = s_1, ...s_n: a set of sequences to be searched )
2:     bestSim ← −Inf
3:     bestSeq ← null
4:     for i = 1 to n do
5:         sim ← computeSim(s_i, t)
6:         if sim > bestSim then
7:             bestSim ← sim
8:             bestSeq ← i
9:         end if
10:    end for
11:    return bestSeq, bestSim
12: end function
```

A single nucleotide is represented as a symbol over the set A, C, G, T, and a DNA sequence is nothing more than a string of these symbols. Therefore, your work on this project comes down to implementing different algorithms for comparing two strings, and returning a quantitative assessment of their similarity using the selected algorithm.

1.1 Example

Let's suppose you had the following sequences in D:

1. s_1 = `CCGAC`
2. s_2 = `CGGACAT`

Now, suppose your query sequence is t = `CGACT` How could you algorithmically determine which is most similar? Well, you could line them up:

```
CGACT
|
CCGAC
```

And that results in only one match at the first position. However, what if you were allowed to introduce *insert* and *delete* operations to try to maximize the number of matching symbols?

```
C GACT
| |||
CCGAC
```

Now you have four matches between t and s_1 by inserting a space in the target! And, consider the second sequence, s_2. You clearly would need to allow some gaps to be inserted in the sequence to maximize the alignment:

```
CG AC T
|| || |
CGGACAT
```

Here, we have five matches!

And, this is pretty much how real sequence alignment algorithms work in practice. There are many different approaches out there to try and identify the best biologically-meaningful alignment between sequences. In fact, most methods will also introduce a scoring system to quantify an alignment between two sequences. The simplest methods will count up the number of matches and be done. However, that is not going to result in the best measure. Better measures use a scoring system that "rewards" matched symbols, but penalizes and insertions and deletions that had to be introduced, to ensure the most similar sequences come up that had the best alignment without introducing these operations. If you don't implement a good scoring system, then merely counting matches would place. For example, using our same target of `CGACT`, suppose we had obtained alignments with two arbitrary sequences, `CGGACAT` and `CGGGGTTTGGGGGACAT`:

```
CG AC T
|| || |
CGGACAT
```

and

```
CG          AC T
||          || |
CGGGGTTTGGGGGACAT
```

Though they both have five matches, these are hardly biologically equivalent alignments! The introduction of so many gaps in the second example should cause the first alignment to score significantly higher than the second.

2 Your Task

Your task is to implement a basic sequence alignment system. We will supply you with a dataset of known DNA sequences, denoted as D, and your code will accept a single query sequence, denoted as t. Your job is to determine which sequences in D are most similar to t. A sample dataset and query are on moodle as `DNA_sequences.txt` and `DNA_query.txt`.

How will you determine the best match? We will get you started with a couple of very basic techniques as detailed below, but the best approach is going to require you to write a dynamic programming method to perform a real **sequence alignment**.

3 Requirements

- Interface: Start with a simple interface that asks the user for the name of the file that contains the query sequence, and the name of the file that contains all of the sequences to be searched against the query. Then be sure to present a menu for the user to select the algorithm. Perform the search, then report the sequence most similar. Your code must have at least three of the below algorithms as options. Bonus points are available for more than 3 algorithms implemented.
- Algorithms: Remember the pseudocode above - you are performing a pairwise sequence comparison with the query sequence against each sequence in D, and reporting the sequence in D that is most "similar". The definition of similarity will depend on the method used. For simplicity, we will use s and t as the two sequences that are being compared. The methods are listed from the easiest to the most challenging:
  - **Longest Common Substring** - identify the longest substring that is common to both s and t. The sequence with the longest substring in common will be most similar.
  - **LCS - Longest-Common Subsequence**, as presented in the dynamic programming unit. This is a generalization of the Longest Common Substring, where you now allow non-contiguous matches between both sequences to maximize the number of matches. The sequence has the longest length match will be the most similar.
  - **Edit distance** - this is another interesting approach used in practice (and also widely used in the field of *computational linguistics*). Given two sequences, s and t, how many "edit operations" are required to transform one sequence into the other? The set of edit operations you would consider are **insertion**, **deletion** and **substitution**. See `https://en.wikipedia.org/wiki/Edit_distance` for more information.
  - **Needleman-Wunsch Algorithm** - this is a classic bottom-up dynamic programming solution that is still use in bioinformatics tools today for aligning two sequences. The following has a good description of the recurrence and subsequent algorithm: `https://en.wikipedia.org/wiki/Needleman\OT1\textendashWunsch_algorithm`
  - **Your own design** - Come up with any idea, method, trick, or combination thereof to get the best matching you can. Even if you can't outperform the methods above, I want to see something unique. Be sure to discuss your ideas, especially what did or didn't work how you expected, in your writeup.

4 File Format

Below is an example of a file of three genetic sequences. This is called the **FASTA** file format, a standard still used today in bioinformatics. It's actually quite a simplistic data format:

- Every sequence will have a header that starts with `>`. All text after `>` should be stored as it represents the name of the sequence and often has useful, interesting characterizations about the sequence, such as annotating its biological function. When reporting the sequence that is most similar, print this header.
- Every subsequent line after the header is concatenated together to form a single sequence. The last line of the sequence is determined by the next line being a blank link, a header line for the next sequence, or the end of file.
- All data will be DNA, and thus all symbols will be over the alphabet A, C, G or T. However, symbols could be upper or lower case letters
- Each line for a sequence may be of varying length.

Here is an example of three DNA sequences:

```
>HSGLTH1 Human theta 1-globin gene
CCACTGCACTCACCGCACCCGGCCAATTTTTGTGTTTTTAGTAGAGACTAAATACCATATAGTGAACACCTAAGA
CGGGGGGCCTTGGATCCAGGGCGATTCAGAGGGCCCCGGTCGGAGCTGTCGGAGATTGAGCGCGCGCGGTCCCGG
GATCTCCGACGAGGCCCTGGACCCCCGGGCGGCGAAGCTGCGGCGCGGCGCCCCCTGGAGGCCGCGGGACCCCTG
GCCGGTCCGCGCAGGCGCAGCGGGGTCGCAGGGCGCGGCGGGTTCCAGCGCGGGGATGGCGCTGTCCGCGGAGGA
CCGGGCGCTGGTGCGCGCCCTGTGGAAGAAGCTGGGCAGCAACGTCGGCGTCTACACGACAGAGGCCCTGGAAAG
GTGCGGCAGGCTGGGCGCCCCCGCCCCCAGGGGCCCTCCCTCCCCAAGCCCCCCGGACGCGCCTCACCCACGTTC
CTCTCGCAGGACCTTCCTGGCTTTCCCCGCCACGAAGACCTACTTCTCCCACCTGGACCTGAGCCCCGGCTCCTC
ACAAGTCAGAGCCCACGGCCAGAAGGTGGCGGACGCGCTGAGCCTCGCCGTGGAGCGCCTGGACGACCTACCCCA
CGCGCTGTCCGCGCTGAGCCACCTGCACGCGTGCCAGCTGCGAGTGGACCCGGCCAGCTTCCAGGTGAGCGGCTG
CCGTGCTGGGCCCCTGTCCCCGGGAGGGCCCCGGCGGGGTGGGTGCGGGGGGCGTGCGGGGCGGGTGCAGGCGAG
TGAGCCTTGAGCGCTCGCCGCAGCTCCTGGGCCACTGCCTGCTGGTAACCCTCGCCCGGCACTACCCCGGAGACT
TCAGCCCCGCGCTGCAGGCGTCGCTGGACAAGTTCCTGAGCCACGTTATCTCGGCGCTGGTTTCCGAGTACCGCT
GAACTGTGGGTGGGTGGCCGCGGGATCCCCAGGCGACCTTCCCCGTGTTTGAGTAAAGCCTCTCCCAGGAGCAGC
CTTCTTGCCGTGCTCTCTCGAGGTCAGGACGCGAGAGGAAGGCGC
>BTBSCRYR
tgcaccaaacatgtctaaagctggaaccaaaattactttctttgaagacaaaaactttca
aggccgccactatgacagcgattgcgactgtgcagatttccacatgtacctgagccgctg
caactccatcagagtggaaggaggcacctgggctgtgtatgaaaggcccaattttgctgg
gtacatgtacatcctaccccggggcgagtatcctgagtaccagcactggatgggcctcaa
cgaccgcctcagctcctgcagggctgttcacctgtctagtggaggccagtataagcttca
gatctttgagaaaggggattttaatggtcagatgcatgagaccacggaagactgcccttc
catcatggagcagttccacatgcgggaggtccactcctgtaaggtgctggagggcgcctg
gatcttctatgagctgcccaactaccgaggcaggcagtacctgctggacaagaaggagta
ccggaagcccgtcgactggggtgcagcttccccagctgtccagtctttccgccgcattgt
ggagtgatgatacagatgcggccaaacgctggctggccttgtcatccaaataagcattat
aaataaaacaattggcatgc
>random sequence 1 consisting of 1000 bases.
acctctggggcgactaatggcgtttgggcccctgtagtgacaccccgaatcgtcctaaag
cgtctccaaaccctactcaggccatattaggatatgaaagagcgatcgcttgcattttcg
tgtctatgttggtttgaacccatcagttcggcagtgcgagccgttatacttctaacggtc
aatcagctggcatattgtcaccgttgtgagctttcttcaccttggtcctcactcatacgg
cttaccagctgactcctggcgaacgttaatacccgcgggccagaacctagatatcctaaa
tcgactgtccgcatggatatatcgatacaatatagaaatacggtcagggagtcgtgttct
atattagggttcccattctcatcccttcgtcgcttagaacaaggcctgtcgctcaatcgc
gatgtcatgtaaggaaagagtcatcgccctctgttaatgcccaacggaagataagactac
cctaatattataaacgagcattctcgctccatactacaggtgcggtcgacgtatggggct
gcctggttagttttcattcgagcatgcgccatcaaggtttaggattcggggaacccccta
cgtggttcacagccgtactatatgatctttgtataagccagggtcacgtttgtgcgctat
ctactgtaccgagttggtcgaagcaccgtaaggcaaatctctcgaccctcgactcttgcg
aaggcgatgtaggacttcacgcaaacaatagcggtatcacaaaggactgggagcagtgac
gagaatttcgttcagacgtgtccatatctgatcacccaatgaactccattctaccttagt
cggcgttctggcactgctgatgtaagctattttgacgggccaacagttctactgaactct
ctaattacgactactgaggctcacgacaacacgcgcgtaccacgccatgccctatctcgt
actttagagccacgcaatcatgctcggttctttgcagttc
```

5 Deliverables

You may use any language you want, as long as I can run it without jumping through too many hoops. Ideally something I can run on the command line would be best, as that allows me to run it over ssh while working from home if needed. Java, python, and the C family would all be fine.

- You will need to submit all source code, along with a plain-text README describing how I can compile, run, and interact with your code. In return for me allowing you to use your language of choice, it is imperative that you strive for simplicity with your instructions! You may add features to your code after the demo, but make sure they're **clearly** documented.
- You will need to schedule a time for a short (about 10-15 minutes) demo **before** the submission deadline. You'll briefly

walk me through the algorithms you used and show how your code behaves. These demos can be either in person or on zoom, as scheduling allows. You will need to bring your own device or share your screen and run your code yourself.

- You will submit a short (try to keep it no more than 1-2 pages) description of the algorithmic structure of your project. Include asymptotic runtimes, though you don't need complete analyses, just enough detail to convince me that your analysis is accurate.

6 Workflow

I suggest you start with coding your file handling and data structures, Longest Common Subsequence detector, and interface. You may organize your team as you wish, but it may make sense to designate one member to each of these tasks and one to start the write up. Once you have these pieces (within the first week!), you can assemble them and have a working prototype. Then you can get started on interesting and distinguishing features.

7 Grading

You will be graded on the following:

- README present and clear (1pt)
- Code runs (1pt)
- Functionality: Code reports similarity using different algorithms (3pts, one per algorithm)
- Algorithmic implementation (3pts, one per algorithm)
- Report (2pts)
- Demo (2pts)
- Functionality and implementation for more than 3 algorithms (bonus points available)
- Extra interface features (bonus points available)
- Any additional algorithms implemented (bonus points available)