CPS 363: Introduction to Bioinformatics
Homework 1: Simple Gene Finder for Salmonella!
(20 Points)

Handed out: February 4, 2019 (Thursday)
Due: 11:59pm February 10, 2019 (Wednesday)
Late submissions accepted with penalty until 11:59pm February 11 (Thursday).

Introduction

Salmonella, the name of a group of bacteria, is one of the most common causes of food poisoning in the United States. Usually, symptoms last 4-7 days and most people get better without treatment. But, Salmonella can cause more serious illness in older adults, infants, and persons with chronic diseases. Researchers are interested in studying the Salmonella pathogenesis and have identified genomic regions present in Salmonella but not in closely related species such as E. coli. Further study revealed that some of these regions are essential to Salmonella-specific pathogenic processes. One particular region was found to be needed for the entry of the bacterium into the epithelial cells of the gut. The researchers studying it wished to identify the protein-coding genes in it, so they sequenced it and analyzed that sequence for genes. In this homework, you will develop a simple gene finder, and use it to examine this sequence. You will also examine a related "mystery" sequence, and try to determine its function.

Preparation work

In `load.py` file write a function called `loadSeq(fileName)`. This function will take one variable as input: a string with the name of the FASTA file that we'll be loading, and return a single string which is all of the DNA in the lines of the input file. Test your function thoroughly with the provide file (U81861.fa) or some other FASTA sample files you create to make sure it works correctly.

I n `dna.py` file write following helper functions for manipulating DNA sequences. These functions will be useful in subsequent problems where you'll analyze DNA data in various ways.

- ***Reverse complements***
  Write a function called `reverseComplement(DNA)` that takes a DNA string as input (in 5' to 3' order) and returns the sequence of its complementary DNA strand, also in 5' to 3' order.

  Here's an example:

```
>>> reverseComplement("TTGAC")
'GTCAA'
```

- ***DNA to amino acids***
  Write a function called `codingStrandToAA(DNA)` that takes a sequence of DNA nucleotides from the coding strand and returns the corresponding amino acids as a string. If the length of the DNA sequence is not divisible by 3, then print error message and return `None`. Assume that the reading frame begins with the first base and continues on to the last nucleotide (i.e., `codingStrandToAA` doesn't need to worry about start and stop codons). Hint: use the codon table to figure out how to translate DNA sequence to protein sequence.

  Here are some examples of what your `codingStrandToAA` function should be doing:

  ```
  >>> codingStrandToAA("AGTCCCGGGTTT")
  'SPGF'
  >>> codingStrandToAA("ATGCAACAGCTC")
  'MQQL'
  >>> codingStrandToAA("ATGCAACAGCTCT")
  'DNA sequence not divisible by 3!'
  ```

  You can then begin by creating a file called `GeneFinder.py`. You will make use of a number of functions we have already written above. You can give yourself access to the functions in your `load.py` and `dna.py` files by making sure they are present in the same directory as `GeneFinder.py`. Then include this at the top of `GeneFinder.py`:

  ```
  from load import *
  from dna import *
  ```

  Next, you should download the salmonella sequence from Canvas (`X73525.fa`), and move it to the same directory as your `GeneFinder.py` file.

A simple gene-finding strategy

A simple gene-finding strategy is to look for large open reading frames. An open reading frame (ORF) is the stretch of sequence between a start codon (with the sequence "ATG") and the next in frame stop codon. By "in frame" we mean that it is in a position that is a multiple of 3 nucleotides away.

For example, take a look at the string:

ATGCATAATTAGCT

There is an "ATG" at the beginning. Then there are two codons, "CAT" and "AAT", and then a stop codon "TAG". So "ATGCATAAT" is an open reading frame in this string. You might think for a moment that "ATGCATAA" is also an answer, but it's not an open reading frame because the "TAA" at the end of it is not "in frame" with the start codon "ATG" - that is, it's not a multiple of three nucleotides away from the leading "ATG".

This strategy depends on the assumption that protein-coding sequence has larger ORFs than noncoding sequence. In noncoding regions "Start" and "Stop" codons appear by random chance, and the ORFs between them are not long. In many circumstances protein-coding genes have ORFs that are longer than this.

To look for genes in a particular sequence, you can identify its longest ORFs. But how do you know if these are genes? To help decide, you can look at many known noncoding regions to obtain a distribution of open reading frame lengths for noncoding DNA. You can then compare the longest ORFs in our test sequence with this distribution. If the test sequence ORFs are longer, they are likely to be genes.

In this problem the approach you will take is the following. You will compare open reading frames in our Salmonella sequence with open reading frame lengths from known noncoding sequences. How do you get known noncoding sequences? One way is to make them randomly. Here you will do this by randomly shuffling (reordering) our genomic sequences. More on that shortly!

Part 1: Finding Open Reading Frames

Now we need to write some functions to handle open reading frames.

OneFrame

In `GeneFinder.py` write a function `oneFrame(DNA)` that starts at the 0 position of DNA and searches forward in units of three looking for start codons. When it finds a start codon, oneFrame should take the slice of DNA beginning with that "ATG" for the open reading frame that begins there until the stopping codon (TAG, TGA, or TAA). If there is no in frame stopping code, then it should assume that the reading frame extends through the end of the sequence and simply return the entire sequence from ATG till the end . It should store that sequence in a list and then continue searching for the next "ATG" start codon and repeat this process. Ultimately, this function returns the list of all ORFs that it found.

Here are some examples of oneFrame in action:

```
>>> oneFrame("CCCATGTTTTGAAAAATGCCCGGGTAAA")
['ATGTTT', 'ATGCCCGGG']

>>> oneFrame("CCATGTAGAAATGCCC")
[]

>>> oneFrame("ATGCCCATGGGGAAATTTTGACCC")
['ATGCCCATGGGGAAATTT', 'ATGGGGAAATTT']
```

Fully test your function before move on to the next problem.

OneFrame2

Next you will write a modified version of `oneFrame` called `oneFrameV2(DNA)` in `GeneFinder.py` In our previous version of `oneFrame`, we searched the given reading frame for every start codon, and returned the ORF corresponding to it. This sometimes results in nested ORFs, and can be seen in the following example:

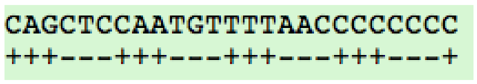
```
>>> oneFrame("ATGCCCATGGGGAAATTTTGACCC")
['ATGCCCATGGGGAAATTT', 'ATGGGGAAATTT']
```

Both ORFs end in the same stop codon. The second ORF is shorter, because it begins with an ATG which is internal to the first ORF. For our gene-finder application it is desirable to instead only return the largest ORF from a set of nested ORFs like this.

```
>>> oneFrameV2("ATGCCCATGGGGAAATTTTGACCC")
['ATGCCCATGGGGAAATTT']
```

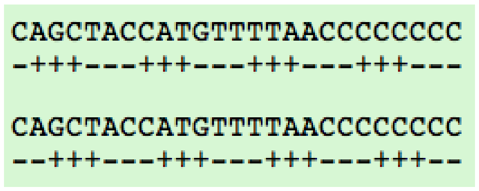
longest Open Reading Frame on one strand

Consider some sequence of interest. Imagine we start at the 0 position and count off in units of 3 nucleotides. This defines a particular reading frame for the sequence. Here is an illustration, where alternating +++ and --- are used to indicate the units of 3 nucleotides.



```
CAGCTCCAATGTTTTAACCCCCCCC
+++---+++---+++---+++---+
```

Considering just the given sequence (and not the reverse complement), we can define two other reading frames on this sequence, starting at either the 1 or the 2 positions.



```
CAGCTACCATGTTTTAACCCCCCCC
-+++---+++---+++---+++---

CAGCTACCATGTTTTAACCCCCCCC
--+++---+++---+++---+++--
```

Every open reading frame between an ATG and a stop must fall in one of these three reading frames. A useful way to look for genes involves searching each of these frames separately for open reading frames.

In `GeneFinder.py` create a `longestORF(DNA)` function which calls `oneFrameV2`.to find the longest open reading frame of all 3 reading frames. Here are some examples of `longestORF`:

```
>>> longestORF('ATGAAATAG')
'ATGAAA'
>>> longestORF('CATGAATAGGCCCA')
'ATGAATAGGCCCA'
>>> longestORF('CTGTAA')
''
>>> longestORF('ATGCCCTAACATGAAAATGACTTAGG')
'ATGAAAATGACT'
```

Fully test your function.

<u>Part 2: longestORFBothStrands(DNA)</u>

We are given a DNA sequence in 5' to 3' order. A gene might appear on this strand or its reverse complement. Thus, our next function is a very short one called `longestORFBothStrands(DNA)`. This function takes a DNA string as input and finds the longest ORF on that DNA string and its reverse complement. This won't be hard! You can use the `longestORF` function you have already written. First ask it for the longest ORF in the given DNA and then ask it for the longest ORF on its reverse complement (use your `reverseComplement` function to find the reverse complement). The longer of those two is the longest ORF possible (break ties arbitrarily).

For example,

```
>>> longestORFBothStrands('CTATTTCATG')
'ATGAAA'
```

Pause here to make sure you understand how we got that answer.

<u>Part 3: longestORFNoncoding(DNA, numReps)</u>

To assess whether long ORFs are genes, a researcher would ask the question, "is this sequence length indicative of a coding region or would I expect to see sequences this long in garbage?" By "garbage", of course, we mean just random sequences of nucleotides.

We'll test this by generating a bunch of "garbage" sequences of the same length as our test DNA sequence, and measuring the maximum ORF length in each. Then, we'll ask the following question. Is the very longest ORF among these still shorter than some ORFs we observe in our real DNA? If the real DNA ORFs are significantly longer than what we see in the garbage sequence, that is a very strong indicator that we did in fact find genes in our original DNA!

Our first task is to write a function `longestORFNoncoding(DNA, numReps)`in `GeneFinder.py` that makes a bunch of garbage sequences, finds the very longest ORF in all of these, and returns its length. Note: this function returns a ***number*** rather than a DNA string.

OK, so now it's time to generate garbage sequences. We could generate totally random strings of the same length as our DNA string, but that might not be a very accurate test since our DNA string might have more nucleotides of one type and fewer of another. To be fair, our garbage strings should have the same nucleotides but just reordered or "shuffled" randomly.

To do that, you'll first need to take your DNA string and turn it into a list of its constituent symbols. There is a built-in function called list that takes as input a string and returns the list of symbols in that string.

```
>>> myList = list("hello")
>>> myList
['h', 'e', 'l', 'l', 'o']
```

Now, in the random package (get it by including import random at the top of your file) there is a function called shuffle that scrambles the list. Unlike other functions that we've used until now, this shuffle function doesn't return a new list, but rather it actually changes the list that you gave it. Here's an example:

```
>>> import random
>>> myList = list("hello")
>>> myList
['h', 'e', 'l', 'l', 'o']
>>> random.shuffle(myList)
>>> myList                <-- myList will be changed now!
['o', 'l', 'h', 'e', 'l']
```

In other words, do not do this in your code...

```
myList = random.shuffle(myList)
```

but instead do just this...

```
random.shuffle(myList)  # this will actually change myList by scrambling its contents.
```

Oh, wait! But now we want that shuffled list to be glued back together as a string, since after all we're dealing with DNA strings. That's easy, write a function called `collapse(L)` that takes as input a list of symbols and returns back the string that we get by gluing those symbols together.

```
>>> collapse(['o', 'l', 'h', 'e', 'l'])
'olhel'
```

For each garbage sequence you make, calculate the longest ORF using `longestORFBothStrands`. You should repeat this process `numReps` times, and return a number indicating the length of the longest ORF you see in all those repetitions.

You can test your `longestORFNoncoding` function on our real Salmonella sequence.

```
>>> X73525=loadSeq("X73525.fa")
```

Now run `longestORFNoncoding` a few times. Note that because it makes use of randomness, it will not give you exactly the same number each time. It will, however, be consistent enough for our purposes.

```
>>> longestORFNoncoding(X73525,50)
624
>>> longestORFNoncoding(X73525,50)
693
```

<u>Part 4: findORFs</u>

Our next step is to write a function `findORFs(DNA)` that will identify all the ORFs in the real (un-shuffled) DNA and return them as a list. If there are none, it should return an empty list.

Once again, this task is easy because we can make use of functions we have already written. `findORFs` should call `oneFrameV2` in each of the three possible reading frames of the sequence. It should then combine all of the ORFs found in each frame and return them. Note that this strategy makes it possible to find overlapping reading frames. findORFs will likely be very similar to the `longestORF` that you wrote above.

```
>>> findORFs("ATGGGATGAATTAACCATGCCCTAA")
['ATGGGA', 'ATGCCC', 'ATGAAT']
>>> findORFs("GGAGTAAGGGGG")
[]
```

<u>Part 5: findORFsBothStrands</u>

Next write a function called `findORFsBothStrands( DNA )` that searches both the forward and reverse complement strands for ORFs and returns a list with all the ORFs found. For example:

```
>>> findORFsBothStrands('ATGAAACAT')
['ATGAAACAT', 'ATGTTTCAT']
```

<u>Part 6: getCoordinates</u>

The functions we've written so far return the sequence of an ORF. However another bit of information we might like to know is its coordinates in the original DNA sequence. We'll now write a function `getCoordinates(orf,DNA)` which returns the beginning and end coordinates of an ORF in DNA. We will follow the convention of reporting everything in forward strand coordinates (even if the ORF is actually on the reverse complemented strand).

First consider the case where the ORF falls on the forward (non reverse complemented) strand. In that case, we can obtain its beginning coordinate with the following syntax:

```
>>> testDNA="ACGTTCGA"
>>> testORF="GTT"
>>> testDNA.find(testORF)
2
```

We can then get the end coordinate by adding the length of the ORF to the start coordinate.

But what if the ORF is actually on the reverse complemented strand? Notice what happens when we use .find with a sequence that is not present:

```
>>> testDNA="ACGTTCGA"
>>> testORF="GAA"
>>> testDNA.find(testORF)
-1
```

The method returns a `-1`. If we search the forward strand of DNA with an ORF and get a `-1`, then we should try searching with the reverse complement of orf. In our example, that would look like this:

```
>>> testDNA="ACGTTCGA"
>>> revCompTestORF=reverseComplement("GAA")
>>> testDNA.find(revCompTestORF)
3
```

Here are some examples of getCoordinates:

```
>>> getCoordinates("GTT", "ACGTTCGA")
[2, 5]
>>> getCoordinates("CGAA", "ACGTTCGA")
[3, 7]
```

<u>Part 7: Gene finding at last!</u>

Write a function called `geneFinder(DNA, minLen)` that identifies ORFs longer than `minLen`, and returns a list with information about each.

`geneFinder` should first call `findORFsBothStrands` to obtain a list of ORFs in the input DNA. It should then run through this list, keeping only those ORFs which are longer than `minLen`.

For each ORF which is long enough, `geneFinder` should calculate

- The beginning and end positions of the ORF in DNA using getCoordinates.
- The protein sequence of the ORF using `codingStrandToAA.`

These should then be placed in a list:

[beginningCoord, endCoord, proteinSequence]

There will be a list like this for every ORF that is long enough. You will collect these lists in another list (a list of lists). A final step is to sort this list of lists before returning it.

Say our final list of lists is called `finalOutputList`. Its elements are lists of this type: [beginningCoord, endCoord, proteinSequence]. We can sort `finalOutputList` by beginning coordinate, and return it like this:

```
finalOutputList.sort()
return finalOutputList
```

<u>Part 8: Using our gene finder</u>

Now its time to apply our method to data and see what we get.

A. Load X73525.fa into python

```
>>> X73525 = loadSeq("X73525.fa")
```

and apply our gene-finding strategy to it.

The first step is to decide on a threshold for `geneFinder`. Our strategy is to determine what the longest ORF we see in noncoding sequence is, and then to define ORFs longer than this as putative genes. Run `longestORFNoncoding(X73525,1500)`. This should be a relatively conservative way to pick a threshold.

Now run `geneFinder` using the threshold value you get for minLen.

```
>>> geneList = geneFinder(X73525, put_your_minLen_value_here )
```

Next write a short function `printGenes(geneList)` which prints the output of `geneFinder` in a nice human-readable form. Use it to print your results.

Note that our gene-finding strategy is very simple, and is also relatively conservative. As a result, we are likely to miss some true genes which are short. But we hope that the genes our method does find are real ones.

Paste your `printGenes` output into a file called `GeneFinder.txt`.

B. Pick one of the genes from your output and blast its protein sequence. To start, open this link in a separate window: NCBI Blast (http://www.ncbi.nlm.nih.gov/blast/Blast.cgi).

Go down to Web Blast and follow the Protein Blast link.

There should be a large box into which you can now input/paste a protein sequence. Then scroll down to the bottom of the page and click the "Blast" button.

When you blast a sequence, the application carries out a search through its databases for known sequences that either match your sequence or are close to it, and returns those to you in a list.

The BLAST output page will contain some graphics at the top. Scroll down past these to the section labelled "Descriptions". The first links in this section will be the closest sequences BLAST could find to your original input. Clicking on one of these will take you to a page with information about that sequence, including the name and what organism it can be found in. The right pane on that page has many useful links. Based on what you learn from these top BLAST hits, briefly describe the likely function of your gene inside GeneFinder.txt.