COMPSCI 260, Introduction to Computational Genomics **2 September 2016**

Problem Set 1

Due: 5pm on 16 September 2016

The main purpose of this first problem set is to familiarize yourself with the Python programming language. The tasks are designed to ensure that you develop some skills in writing small but useful Python programs.

Please comment your code efficiently to explain well the logic behind it; it is hard for us to give partial credit if we cannot determine what your logic is. If your code is sufficiently well documented, you need not write up an accompanying description in situations where that is all the problem requires. We will provide you with a skeleton `README.problemX.txt` file that you can populate with answers to various questions and extra comments about your code. The `README` files have some fields at the top that you **must** fill in to receive credit for your work.

As a suggestion to less experienced programmers, it often helps to outline your high-level solution clearly on paper or in your mind before you even start to develop your programs. Once your strategy is clear in your mind, then develop your code step-by-step; you can check that each step is performing correctly by printing some results to the screen. Be sure to test your programs thoroughly before submitting. You should always submit along with your code a short transcript of the program in operation to suggest its correctness on a few inputs (and to convince yourself it is working as expected).

We want you to think through new and difficult concepts and hopefully figure them out, but not to be stuck and frustrated for hours. Feel free to post questions to Piazza at any point, or come to office hours if you need more hands-on help.

As discussed in class, in this problem set we will consider an ORF to be a stretch of DNA (or RNA in the case of an RNA virus like SARS) that starts with a start codon (ATG; or AUG in the case of RNA) and continues until it reaches the first in-frame stop codon (TAA, TAG, TGA; or UAA, UAG, UGA in the case of RNA). Specifically, we will adopt the convention that the stop codon that follows an ORF is *not* considered part of that ORF.

You should also note that a stop codon will never be translated into any amino acid, while the start codon (ATG or AUG) *will* be translated into an amino acid (specifically, methionine). As an example, let's consider the nucleotides 11717–18680 of the p53 gene (Problem 1). This sequence starts with a start codon and ends with a stop codon, but the part of this sequence that is actually coding for a protein is a subset of the nucleotides between 11717 and 18677 (i.e., we exclude the stop codon; also, we say "a subset of" because there may be introns).

Problem 1: Exploring the p53 gene (30 points)

Assume you want to clone the genomic region coding for the p53 gene, including all relevant introns. `p53.fasta` contains the full genomic sequence of the p53 gene, in FASTA format. (You may have noticed that there is a function in `compsci260lib.py` called `get_fasta_dict` that can be used to read in a file stored in FASTA format—this will be useful here.) The part of the sequence that codes for protein (including several untranslated introns) is between nucleotides 11717–18677[1].

[1] Be careful: The nucleotides in a DNA (RNA) sequence are numbered starting with 1 (i.e., nucleotide 1 is the first nucleotide), but when you are working with strings and arrays in Python, the first element will have the index 0!

a) From only the information provided thus far, what can we predict about the length of the peptide product of the p53 gene?

For the remainder of the problem, write a Python program to accomplish the following tasks. You should modify the starter code we provide entitled `p53.py`.

b) Start by storing the locations of the beginning and end of the coding region in newly defined variables. You should use these variables for extracting the coding sequence.

c) Add code to extract the part of the sequence between the beginning and the end of the coding region (including the introns) and store it in another variable. We will refer to this whole sequence henceforth as the *coding plus intron sequence*.

d) Construct a new variable by concatenating the *coding plus intron sequence* with the next three nucleotides (nucleotides 18678–18680). Add code to validate that this new sequence starts with an ATG codon and ends with a stop codon (either TAA, TAG, or TGA) using a single regular expression (you can find a quick refresher on Python regular expressions at `http://docs.python.org/2/library/re.html` or `http://docs.python.org/2/howto/regex.html`).

e) Add code to print out the nucleotides from 17520 to 17522. Under what circumstances will this triplet function as a stop codon?

f) Add code to check whether the coding plus intron sequence contains a restriction site[2] for PmlI (cuts at CACGTG) using a single regular expression. If it does, have your program report the actual sequence in p53 that is recognized by this enzyme and the position (from the start of the coding plus intron sequence) at which it is located. Do the same for SgrAI (cuts at CrCCGGyG, where r is either G or A, and y is either C or T) and for OliI (cuts at CAC, then 4 nucleotides of any type, then GTG).

Problem 2: Hunting for ORFs in SARS (35 points)

As discussed in class, one of the initial steps in the process of identifying genes in genomes that lack introns is to locate all of the open reading frames (ORFs). An ORF begins with a start codon (AUG in the case of an RNA virus like SARS), and because there are no introns, it must have a stop codon somewhere downstream in the same reading frame. Remember that unlike stop codons, start codons can (and most likely will) be found inside the ORFs: since AUG is the only triplet coding for methionine (Met), every time Met is required in a protein, we will find a corresponding AUG codon in the sequence coding for that protein. When looking for ORFs in a genome, please report only the longer ORFs (i.e., if the nucleotide sequence is ... AUG CGU AUG AAG AUG UCA UAG ..., report only the ORF: AUG CGU AUG AAG AUG UCA, and not the substrings AUG AAG AUG UCA or AUG UCA).

Submit a Python program to accomplish the following tasks, modifying the starter code we provide entitled `orfs.py`.

a) Write a procedure that will take as input a genome sequence and the minimum length of an ORF in amino acids, and return **a list of dictionaries**, each entry of the list corresponding to one ORF, and each dictionary containing entries providing information about that ORF's:

1. Start position
2. Stop position (since we don't consider the stop codon as part of the ORF, the stop position will be the position just before the stop codon)
3. Particular stop codon (UAG, UGA, or UAA, since SARS is an RNA virus)

[2] Restriction sites are particular sequences of nucleotides that are recognized by *restriction enzymes* as sites where to cut a double-stranded DNA molecule.

4. Length in nucleotides
5. Length of the translated peptide (in amino acids)
6. Reading frame with respect to the start of the genome (0, 1, or 2)[3].

The list returned should be something like:

```
[
{'frame': 0, 'stop': 13410, 'aalength': 4382, 'start': 265,
 'stopcodon': 'UAA', 'nlength': 13147, 'strand': '+'},
{'frame': 0, 'stop': 27060, 'aalength': 221, 'start': 26398,
 'stopcodon': 'UAA', 'nlength': 664, 'strand': '+'},
...
]
```

There are many ways of arriving at a solution, but some are easier than others. Judicious use of regular expressions may be useful, but not be required, or even necessarily simplest.

Extra challenge: Modify or write your code to take as input either a DNA or an RNA sequence. Can you identify different strategies to implement this change in your procedure?

b) Now apply your procedure to the SARS genome in `sars.fasta`. How many ORFs does your procedure find if the minimum ORF length is 10 amino acids? 40 amino acids? 70 amino acids? What can you necessarily say about the numbers you will compute, even before you compute them?

c) In each of the three cases above, what is the average length (in amino acids) of the identified ORFs? What can you necessarily say about the numbers you will compute, even before you compute them?

d) Write a function `random_amino_acid` that takes an amino acid as input and randomly changes it into another (different) amino acid by mutation. Write another function `random_nucleotide` that takes a nucleotide as input and randomly changes it into another (different) nucleotide. Now use these two functions in a program to try two things. Using the third (*w.r.t.* the beginning of the sequence, regardless of reading frame) ORF in the 5' to 3' direction, found when the minimum ORF length is 40 amino acids:

1. Select a random position in the sequence and apply the `random_nucleotide` mutation function to that position. Repeat this procedure 10 or 20 times, accumulating mutations as you go and printing out the corresponding proteins that arise as each mutation is applied (the `translate` function will be useful to compute the corresponding proteins).
2. Now translate the initial ORF sequence into the corresponding protein and then repeatedly select random positions in the protein sequence and apply the `random_amino_acid` mutation function, accumulating mutations as you go and printing out the proteins that arise as each mutation is applied.

In the long run (as you repeat this infinitely many times), will the overall distribution of protein sequences that you see from these two approaches be the same? Why or why not?

Hint: Python has a built-in pseudo-random number generator in the package `random`. To call it, import the `random` package first (`import random`) and then call `random.randint(a,b)`. It will return an integer uniformly distributed between `a` and `b`, inclusive.

[3]The reading frames 0, 1 and 2 can be defined as:
0 = the reading frame that starts at the first nucleotide of the genome.
1 = the reading frame that starts at the second nucleotide of the genome (in other words we shift one position to the right).
2 = the reading frame that starts at the third nucleotide of the genome (in other words we shift two positions to the right).
For example if the genome is CCAAUCACGGC... then reading frame 0 will contain the codons CCA, AUC, ACG and so on, reading frame 1 will contain the codons CAA, UCA, CGG and so on, and reading frame 2 will contain the codons AAU, CAC, GGC and so on.

Problem 3: Genome assembly (35 points)

When the human genome was sequenced, researchers didn't put it into a machine and wait for a sequence of 3 billion base pairs to come out. DNA sequencing technology (of the Sanger sequence variety) only permits the determination of around 500 to 800 base pairs at a time, so to solve what is known as the *assembly problem*, algorithms are used to combine many small overlapping reads into a longer sequence of bases. In this problem, you will write a Python program to solve a very simple assembly problem.

a) Before writing any code, take some time to consider the genome assembly problem at hand. Given an arbirtrary number of reads of a certain average length, how would you assemble them into a longer, continuous sequence? What could make this endeavor more challenging? How might an algorithm deal with these potential obstacles? Would your approach need to change in response to variations in the number of reads, the average read length, or the degree to which the reads overlap? How might your thinking be influenced by any knowledge you may (or may not) have about the genome from which the reads were generated?

b) Now, you will be given an input file `ssRNA.fasta` that contains twenty overlapping reads from a linear RNA molecule. Each read is approximately 500 base pairs in length. The problem is made simpler by the fact that each read is from a single-stranded RNA molecule (you need not worry about reverse complementation or opposite orientations), that the end of each read overlaps the start of another read by exactly 20 base pairs (you need not worry about sliding one read against the others in all possible positions), and that there are no sequencing errors whatsoever (you need only consider perfect matches). Submit a Python program, modifying our starter program entitled `ssRNA.py`, that will reassemble the full RNA sequence from the given input file. You should find the `get_fasta_dict` function useful for importing the reads. How many nucleotides are in the RNA sequence you assemble? (*Hint:* So you can double-check your work up to this point, the answer you get should be divisible by three.) Does this make sense given the lengths of the original sequences? Explain how.

c) Within your reconstructed single-stranded RNA sequence, search for ORFs with a minimum length of 70 amino acids. You should consider overlapping ORFs only if they are in different reading frames. You may find the data structure you developed earlier will help you store the ORF information effectively.

How many ORFs do you find and what are their lengths?

Do there exist ORFs that overlap other ORFs?

Compute the fraction of the genome that is coding (in more precise terms: the fraction of the total length of the sequence that is made up of nucleotides which participate in at least one coding sequence). Remember that the stop codon is not considered a part of the coding sequence.

d) Take the sequence of the largest ORF you find and translate it into an amino acid sequence (the `translate` function in `compsci260lib.py` will come in handy here). Visit the NCBI web site, find their BLAST page, and select the 'protein blast' program. It should take you to the *blastp* page. Paste your amino acid sequence into the search box, choose the 'Reference Proteins (refseq_protein)' database, leave all the other parameters at their default values, ensure that the algorithm selected under 'program selection' is *blastp* (protein-protein blast), and submit your BLAST query. When you receive a response, you'll see a picture with your sequence depicted under the graphic summary section, and then a number of matches and partial matches depicted beneath it. Click on the topmost match and you'll jump down the page to a collection of sequences that are similar to each other and best match your query. In this case, you should find a perfect match present.

What protein is this? To which organism does it belong? Discuss, in detail, the structure of this protein. Is it subjected to any post-translational modification? How, specifically, does this protein support the organism in which it is found? You may need to consult additional resources in answering these questions (i.e., feel free to use the information gleaned from the NCBI entry in order to inform additional research).