COMPSCI 260, Introduction to Computational Genomics **30 August 2019**

Problem Set 1

Due: 5pm on 13 September 2019

The main purpose of this first problem set is to familiarize yourself with the Python programming language. The tasks are designed to help you develop skill in writing small but useful Python programs to work with biological sequence data.

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 at the end of an ORF is considered part of that ORF.

You should note, however, 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 yeast Aim2 gene, which is located between nucleotides 1001 and 1741 (inclusive) in the `aim2_plus_minus_1kb.fasta` file (Problem 1). This sequence begins with a start codon and ends with a stop codon, but the part of this sequence that is actually coding for a protein is nucleotides 1001 through 1738.

Problem 1: Attack of the clones (30 points)

List of files to submit

1. `README.problem1.txt`
2. `cloning.py`

⋆ Step 1: Identification of restriction sites

Plan

Molecular cloning is a common practice in biology where bacterial cells are used to create many copies of a specific DNA fragment, often a particular gene of interest. This is accomplished by extracting the DNA fragment via polymerase chain reaction (PCR) or restriction enzyme digestion and inserting the fragment into a circular plasmid which is taken up and replicated by host bacterial cells. Those of you who have taken Biology 201 here at Duke have done molecular cloning before. In this problem, you will write code intended to simulate a simplified version of this process. In particular, you will write a Python program—modifying the starter code we provide in `cloning.py`—to locate restriction sites flanking the yeast Aim2 gene, identify a compatible restriction site within the multiple cloning site of the pRS304 plasmid, and extract the appropriate genomic fragment and insert it into the cleaved location in the plasmid.

a) Using the search function in the *Saccharomyces* genome database (SGD; `http://www.yeastgenome.org`), what is the function of the Aim2 protein?

b) What is the length of the Aim2 gene in nucleotides?

Aim2 is a yeast gene without introns. In fact, most yeast genes do not possess introns.

c) Based on this knowledge, what is the expected length of Aim2's peptide product? If Aim2 did contain intronic regions within its nucleotide sequence, would you expect the resulting peptide to be longer or shorter? Why?

Restriction sites are particular sequences of nucleotides recognized by *restriction enzymes*. The function of these enzymes is to cleave double-stranded DNA molecules at specific sites, leaving "sticky ends" of single-stranded DNA that can be used to clone genomic sequences from different origins together (see Figure 1). In the following questions, you will be asked to computationally identify the locations of six different restriction enzymes in the genomic region including the Aim2 gene. Deciding the best restriction enzyme to use is a necessary step in any cloning experiment.

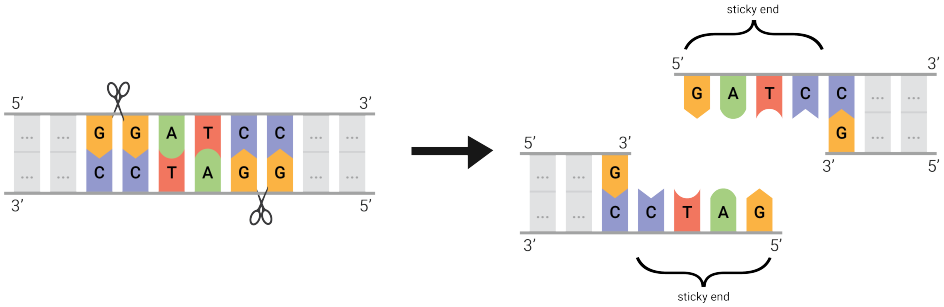

Figure 1: Double stranded DNA is cleaved by restriction enzyme BamHI leaving behind two overhanging "sticky ends". The enzyme cleaves a covalent phosphodiester bond within the backbone of each strand (perhaps in offset locations, as illustrated here), and the few hydrogen bonds in between those cuts are generally not strong enough to keep the sticky ends stuck together.

Restriction sites may occur multiple times within any given genomic sequence. In most cases, the site recognized by a restriction enzyme is palindromic. When trying to computationally identify palindromic restriction sites, it is sufficient to simply check for the location of a restriction site in one strand of the genome. However, in cases where the site is not palindromic, it may be necessary to check both the forward and reverse strands for that particular sequence.

Develop

`aim2_plus_minus_1kb.fasta` contains the full genomic sequence of the yeast Aim2 gene as well as an additional thousand base pairs upstream and downstream of the 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.) This means that the part of the sequence provided which corresponds to the Aim2 gene itself is nucleotides 1001–1741[1].

d) Write code to store the locations for the beginning and end of the Aim2 gene in newly defined variables. Additionally, have your code calculate and **report**:

- the starting and ending locations for the gene
- the length of the gene in nucleotides
- the length of the gene in amino acids

[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 or arrays in Python, the first element will have the index 0!

If you've done this properly, it should agree with your earlier answers, so you should be able to check your code.

e) Using regular expressions, write a function called `find_aim2_restriction_enzymes` to find the following restriction sites in the regions contained in `aim2_plus_minus_1kb.fasta`. You will need to define and call the regular expressions in the `get_restriction_enzymes_regex` function to be used in `find_aim2_restriction_enzymes`. These regular expressions will be used again for a subsequent task.

- BamHI (recognizes GGATCC, cutting between the G in the first position and the G in the second position). This cut occurs on each of the DNA strands, but in different locations, resulting in "sticky ends" (see Figure 1).
- BstYI (recognizes rGATCy—where r is either A or G, and y is either C or T—cutting between the nucleotide in the first position and the G in the second position).
- SalI (recognizes GTCGAC, cutting between the G in the first position and the T in the second position).
- SpeI (recognizes ACTAGT, cutting between the A in the first position and the C in the second position).
- SphI (recognizes GCATGC, cutting between the G in the fifth position and the C in the sixth position).
- StyI (recognizes CCwwGG—where w is either A or T—cutting between the C in the first position and the C in the second position).

The function will take as input: the beginning and end locations of the Aim2 gene, as well as the genomic sequence containing the Aim2 gene.

It should return a dictionary that maps restriction enzymes to lists of dictionaries containing information about the site recognized by each enzyme. The information about each site should be:

1. Start position
2. End position
3. Sequence recognized by the enzyme
4. Location with respect to the gene body ("upstream", "downstream", or "within"). To keep things simple, we have ensured that no restriction site will straddle a boundary of the Aim2 gene.

The returned dictionary should look something like:

```
{
    'BamHI' : [
        {'start': 10, 'end': 15, 'sequence': 'GGATCC', 'location': 'upstream'},
        {'start': 100, 'end': 105, 'sequence': 'GGATCC','location': 'downstream'},
        ...
    ],
    'BstYI' : [
        {'start': 30, 'end': 35, 'sequence': 'AGATCC', 'location': 'within'},
        ...
    ],
    ...
}
```

Check

f) **Report** the values described above for all identified sites for each enzyme after running the function on the Aim2 genomic sequence in `aim2_plus_minus_1kb.fasta`. What are the restriction enzymes that are cutting upstream and downstream of the gene? Are there any enzymes that cut within the gene body?

Reflect

g) If you were running a cloning experiment on the Aim2 protein, which restriction enzymes would you use? Which would you avoid? Why?

⋆ **Step 2:** Cloning Aim2 into a yeast integrating plasmid

Plan

pRS304.fasta contains the sequence for a yeast integrating plasmid and `pRS304_map.pdf` highlights several key features of this plasmid. Before moving on, do some brief research on how integrating plasmids are used, because that will help you understand the rest of this problem.

h) In light of your findings, consult the `pRS304_map.pdf` file and discuss the purpose of the following features: a multiple cloning site (MCS), an ampicillin resistance gene, a replication origin, and a gene required for tryptophan synthesis in yeast (referred to as an auxotrophic marker).

The multiple cloning site in pRS304 possesses a number of restriction sites. The key is to determine which restriction enzyme(s) we need to use on Aim2 in order to excise it from its flanking sequence, keeping in mind that we cannot use an enzyme that has a restriction site within the Aim2 gene body or it will not function. Ultimately, the overhanging single-stranded DNA sequences (sometimes referred to as "sticky ends") [2] generated at the restriction sites flanking the gene need to match up with the overhangs generated at the restriction site in the plasmid.

i) Consider the restriction enzymes from step 1. In most cases, the sticky ends generated at a particular site by one enzyme will be compatible with the sticky ends generated by the same enzyme at a different site. In which cases will this not be true?

j) Again, consider the restriction enzymes from step 1. In some cases, restriction enzymes will generate sticky ends that are compatible with sticky ends produced by a different restriction enzyme. Which of the restriction enzymes from the list above could exhibit this behavior?

Develop

k) Using the regular expressions in `get_restriction_enzymes_regex`, write a function called `find_pRS304_MCS_restriction_sites` to determine if there is a restriction site from the enzyme list provided in the pRS304 MCS (found between nucleotides 1887 and 1994 in `pRS304.fasta`). This code will be similar to the function to find restriction sites around the Aim2 gene previously described.

Take as input the pRS304 genomic sequence and the start position of the MCS and return as output a dictionary mapping restriction enzymes to lists of sites. Each site will be formatted with the following keys:

1. Start position
2. End position
3. Sequence recognized by the enzyme

[2] Figure 1 gives an example of how sticky ends are generated in case of BamHI.

The returned dictionary should look something like:

```
{
    "BamHI" : [
        {'start': 10, 'end': 15, 'sequence': 'GGATCC'},
        ...
    ],
    ...
}
```

Report your findings.

Plan

l) Determine if there is a way to excise the Aim2 gene using one or more of the enzymes from step 1 such that you can incorporate it into the pRS304 plasmid. Which restriction sites would you use? Where are they located? Note that you will have to cut at two restriction sites (one upstream and one downstream of the Aim2 gene) but only once in the pRS304 MCS. And remember, these restriction sites need not be cut by the same enzyme in order to give compatible sticky ends.

Develop

m) Based on the plan you've devised above, write code to extract the Aim2 gene and whatever upstream and downstream regions are flanked by the restriction sites of your choosing. With respect to the original sequence, have your code **report** the nucleotide positions for the beginning and end of the **shortest** excised gene region. Store this sequence in another variable.

n) Finally, write code to insert your excised gene region into the pRS304 plasmid at the proper location. Specifically, your code should produce the new plasmid sequence after cloning in the chosen Aim2 gene fragment.

Check

o) What is the length of the plasmid after inserting the Aim2 gene?

Reflect

p) Consider the junctions between the plasmid sequence and the gene region sequence. Would you be able to use the same restriction enzyme(s) to cut at those locations again? Why or why not? (NOTE: Consider only the shortest excised gene region.)

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

List of files to submit

1. `README.problem2.txt`
2. `orfs.py`

Plan

As discussed in class, to identify genes in genomes that lack introns, we can start by locating all of the open reading frames (ORFs). An ORF begins with a start codon (AUG in RNA viruses like SARS). Since there are no introns, an ORF ends with a stop codon (UAG, UGA, or UAA in RNA viruses like SARS) somewhere downstream in the same reading frame.

Note, AUG is the only triplet coding for methionine (Met). Any time Met is required in a protein, we will find a corresponding AUG codon in the nucleotide sequence for that protein. This means start codons can (and most likely will) be found within ORFs. When searching for ORFs in a genome, please only return the longest ORF found in any given region (i.e. if the nucleotide sequence is ...AUG CGU AUG AAG AUG UCA UAG ..., return only the ORF: AUG CGU AUG AAG AUG UCA, and not the substrings AUG AAG AUG UCA or AUG UCA).

Develop

Submit a Python program to accomplish the following tasks. Modify the skeleton code provided in `orfs.py`.

a) Write a function called `find_orfs` that will take as input a genome sequence and the minimum ORF length in amino acids. It should return a list of dictionaries where each dictionary entry corresponds to one ORF and contains the following information describing that ORF:

1. Start position
2. Stop position (We consider the stop codon as part of the ORF, therefore the stop position will be the last position of the stop codon)
3. Stop codon (UAG, UGA, or UAA)
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 look something like:

```
[
{'frame': 0, 'stop': 13413, 'aalength': 4382, 'start': 265,
 'stopcodon': 'UAA', 'nlength': 13149},
{'frame': 0, 'stop': 27063, 'aalength': 221, 'start': 26398,
 'stopcodon': 'UAA', 'nlength': 666},
...
]
```

Additionally, we will need a quick way to summarize the ORFs we find. Write a function called `summarize_orfs` that takes in the ORF list from `find_orfs` and returns a tuple containing the number of ORFs found and the average ORF length in amino acids.

There are many ways of arriving at a solution, but some are easier than others. HINT: Judicious use of regular expressions may be useful, but are not required nor necessarily simplest.

[3]The reading frames 0, 1 and 2 are defined as:
0 = the reading frame starting with the first nucleotide of the genome.
1 = the reading frame starting with the second nucleotide of the genome (or shifted one position to the right).
2 = the reading frame starting with the third nucleotide of the genome (or shifted two positions to the right).
For example if the genome is CCAAUCACGGC... then reading frame 0 will begin with the codons CCA, AUC, ACG, reading frame 1 will begin with the codons CAA, UCA, CGG, and reading frame 2 will begin with the codons AAU, CAC, GGC.

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

Check

b) Apply your functions to the SARS genome in `sars.fasta`. Have your code **report** the number of ORFs if the minimum ORF length is 10 amino acids, 50 amino acids, or 70 amino acids. Also **report** the average length (in amino acids) of the identified ORFs for the three cases.

Reflect

c) Consider your results and answer the following questions. What is the expected relationship between the minimum ORF length and the number of ORFs you will identify? What is the expected relationship between the minimum ORF length and the average length of the ORFs you will identify?

d) Take a closer look at the ORFs identified when the minimum ORF length is set to 70. Compare these ORFs with those depicted in `SARS_genome_map.pdf`. How well do the ORFs identified in your function match those depicted in the genome map? *What are some possible reasons for any discrepancies you encounter?*

Problem 3: Plasmid assembly (35 points)

List of files to submit

1. `README.problem3.txt`
2. `plasmid.py`
3. `orfs.py`
4. `FLAG.txt`

⋆ Step 1: The genome assembly problem

Plan

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 identify and then combine overlapping sequences into longer sequences of bases.

a) Formulate an approach to solve the computational genome assembly problem: Suppose you are given an arbitrary number of sequence reads—each of which is a randomly located subsequence of the source genome—and you're required to computationally assemble them into a longer, continuous sequence. *Discuss your solution and the challenges that would arise in assembling the source genome using your approach.* (Some of the points that you may discuss in your answer are as follows: 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?)

Develop

In the following steps, you will write a Python program called `plasmid.py` to solve a very simple assembly problem, with one twist: the DNA whose sequence you are trying to determine is that of a circular bacterial plasmid.

Your input file, `plasmid.fasta`, contains fourteen overlapping reads from a circular bacterial plasmid. Though the length of each read is different, their average length is approximately 625 base pairs. To simplify the problem, you can assume that each read is from the same strand of the DNA (you need not worry about reverse complementation or opposite orientations), that the end of each read overlaps the start of another read by exactly 15 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).

b) In `plasmid.py`, write a function called `simple_assembler` that will reassemble a full plasmid sequence from a list of overlapping reads. The function will take as input the reads as a list of strings (but note that the reads are not in any particular order) and return the assembled string sequence. Because the assembled plasmid is circular, we could define its start anywhere. For consistency, your function should return the plasmid sequence assuming it starts with whatever is the first read that appears in its input list.

Call `simple_assembler` on the reads contained in `plasmid.fasta`. `plasmid.fasta` has one read named "start". Use the start of this read to define where the assembled sequence will begin (it may not always be the first read in the fasta file). **Report** the assembled plasmid sequence. (*Hint:* You should find the `get_fasta_dict` function useful for importing the reads.)

Check

c) How many nucleotides are in the plasmid sequence you assemble? (*Hint:* So you can double-check your work up to this point, the answer you get should be divisible by three.)

Reflect

d) Does your answer to the previous question make sense given the lengths of the original sequences? Explain how.

⋆ **Step 2:** Finding ORFs in a circular plasmid

Plan

Searching for ORFs in the SARS genome is somewhat simpler than searching for ORFs in other organisms without introns because the SARS genome is a single-stranded RNA genome. When searching for ORFs in double-stranded DNA genomes, one must scan both strands for ORFs, in each case being sure to take orientation correctly into account (recall that an mRNA is processed in a 5' to 3' direction by a ribosome, and that an mRNA is produced in a 5' to 3' direction by an RNA polymerase on the basis of complementation with a DNA template).

Develop

e) Write code to search for ORFs within your reconstructed-double stranded plasmid. Be sure that your search includes all 6 possible reading frames and permits you to find ORFs that overlap, in case any do (two ORFs cannot overlap within one reading frame, by construction, but they could overlap if they are in

different reading frames). The data structure you developed earlier can help you store the ORF information effectively. Remember that the plasmid is circular; thus, searching for and storing the ORFs you find will not be as simple as in Problem 2. Also remember that the length is a multiple of three so you need not worry about the reading frame changing as you roll around the plasmid.

Check

f) Within your reconstructed double-stranded plasmid, search for ORFs with a minimum length of 50 amino acids. **Report** the output of your search.

g) How many ORFs do you find? Do there exist ORFs that overlap other ORFs?

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

Reflect

i) 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 best match your query. In this case, you should find a perfect match. What protein is this and what does it do? Given that the protein does not come from bacteria, are you surprised to find the gene for it on a bacterial plasmid? *How could it have gotten there?*