COMPSCI 260, Introduction to Computational Genomics **2 September 2022**

Problem Set 1

Due: 5pm on 16 September 2022

The main purpose of this first problem set is to familiarize yourself with using the Python programming language to explore and analyze biological sequences. The tasks are designed to help you develop your skill in writing small but useful Python programs to work with this kind of data. You'll start very simple, but even in this first problem set, you'll work your way up to analyzing the SARS-CoV-2 genome! As you'll recall from PS0, you will only need to use packages from the Python standard library throughout this course; please don't use additional packages because the autograder will not recognize them.

In this problem set, we will consider an open reading frame (ORF) to be a stretch of DNA (or RNA in the case of an RNA virus like SARS-CoV-2) 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, in Problem 1, you'll be working with the yeast Aim2 gene, which you will find located between nucleotides 1001 and 1741 (inclusive) in the `aim2_plus_minus_1kb.fasta` file. The subsequence corresponding to the gene begins with a start codon and ends with a stop codon, but the last nucleotide in this subsequence that actually codes for an amino acid of the Aim2 protein is nucleotide 1738.

Problem 1: Attack of the clones (20 points)

List of files to submit:

1. `README.problem1.[txt/pdf]`
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 bazillions of 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 can then be taken up by host bacterial cells and replicated again and again. 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 that flank the yeast Aim2 gene so that it can be excised from its context, identify a compatible restriction site within the multiple cloning site of the pRS304 plasmid, and then computationally insert the excised gene into the cleaved location in the plasmid.

a) Let's start with a little context. 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?

c) Aim2 is a yeast gene without introns (in fact, most yeast genes do not possess introns). Based on this knowledge, what is the expected length of Aim2's peptide product after translation? If Aim2 did contain intronic regions within its nucleotide sequence, how would you expect the resulting peptide's length to change? 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 serve as the place where a new genomic sequences can potentially be inserted (see Figure 1). In the following questions, you will be asked to computationally identify the locations of potential restriction sites for six different restriction enzymes in the genomic region that includes the Aim2 gene (which we provided). Deciding on the best restriction enzyme to use is a necessary step in any cloning experiment, and you can assess this computationally.

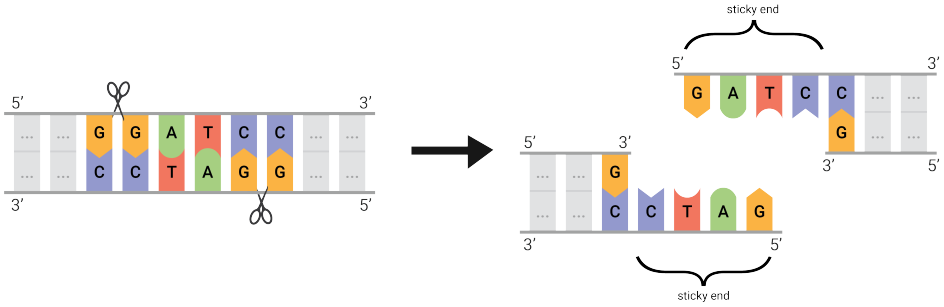

Figure 1: When the colored subsequence of the double-stranded DNA is recognized by the restriction enzyme BamHI, it binds temporarily and cleaves the two strands in the indicated locations, leaving behind two overhanging "sticky ends". The enzyme cleaves a covalent phosphodiester bond within the backbone of each strand (perhaps in offset locations, as illustrated here), and the few hydrogen bonds in between those cuts to the backbones of the two strands are generally not strong enough to keep the sticky ends stuck together for long.

The restriction site(s) recognized by a particular restriction enzyme may occur multiple times within any given genomic sequence, and if that happens, the enzyme can generally be assumed to cut all of them. In many 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 (why?). 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

Before you proceed, we have provided you with the `get_fasta_dict` function in `compsci260lib.py`, which you can use to read in a sequence stored in a file in FASTA format; look over the function and its signature to be sure you know how to call it and what it returns. The FASTA file `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. This means that the part of the sequence that corresponds to the Aim2 gene itself is nucleotides 1001–1741 (inclusive).[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**:

[1] Be careful: The nucleotides in a DNA or RNA sequence are numbered starting with 1 (i.e., the first nucleotide is nucleotide 1, which makes perfect sense), but when you are working with strings or arrays in Python, the first element will have the index 0 (this also makes perfect sense, assuming you like to count using bit vectors!). You will have to account for this discrepancy throughout the course.

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

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

e) Write a function called `find_restriction_sites` that uses regular expressions to find restriction sites within a given sequence for the specific enzymes below. Here, we will use the function to look for restriction sites for these enzymes within the sequence `aim2_plus_minus_1kb.fasta`. The same function will be applied to a different sequence in a subsequent task, so we will pass the sequence in as a parameter.

Within the function call to `find_restriction_sites`, you see that it is passed `r_enzymes` as a parameter, a variable populated from a call to `get_restriction_enzymes_regex`. The job of `get_restriction_enzymes_regex` is to define the set of restriction enzymes and their regular expressions, returning them in the form of a dictionary. You need to expand on the skeleton contents of the `get_restriction_enzymes_regex` function so that it returns a dictionary with entries for each of the following enzymes:

- 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 `find_restriction_sites` function will take as input: a genomic sequence, the beginning and end location of a window within that genomic sequence, and a dictionary mapping restriction enzymes to their regex strings (as returned from `get_restriction_enzymes_regex` function). For the purpose of this subproblem, the input should be the genomic sequence that contains the Aim2 gene, and you should specify the window to be the coordinates of the Aim2 gene.

The function should return a dictionary that maps restriction enzymes to lists of dictionaries, with each dictionary in the list containing information about one of the sites recognized by that enzyme. The dictionary with information about each site should be:

`'start'`: start position (first nucleotide) of the site

`'end'`: end position (last nucleotide) of the site

`'sequence'`: actual nucleotide sequence that was recognized by the enzyme at this site

`'location'`: location of the site with respect to the specified window (in this case, the Aim2 gene); this is a string that can take values of 'upstream', 'downstream', or 'within'. To keep things simple, we have ensured that no restriction site will straddle a boundary of the Aim2 gene window.

The returned dictionary should look something like the following (*Note:* When we provide snippets like this to indicate what the dictionary should look like, we are describing only the *form* or *types* of what is returned; the specific *values* are just made up):

```
{
    'BamHI' : [
        {'start': 10, 'end': 15, 'sequence': 'GGATCC', 'location': 'upstream'},
        {'start': 2100, 'end': 2105, 'sequence': 'GGATCC', 'location': 'downstream'}
    ],
    'BstYI' : [
        {'start': 1230, 'end': 1235, 'sequence': 'AGATCC', 'location': 'within'},
        ...
    ],
    ...
}
```

Apply

f) **Report** the values described above for all identified sites for each enzyme after running the `find_restriction_sites` 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 are cutting within the gene? (You might think about organizing your reporting so that these questions are easy to answer.)

Reflect

g) *If you were running an experiment that involved cloning the Aim2 gene, 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 (by design). One key step in designing a cloning experiment is determining which restriction enzyme(s) to use so that when Aim2 is excised from its flanking sequence, its sticky ends will be compatible with the sticky ends where the plasmid MCS is cut. Specifically, the overhanging single-stranded DNA sequences (often 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.

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

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 is this not guaranteed to 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) Use the `find_restriction_sites` function developed in **part 1e** to determine if the enzymes in the list provided have restriction sites in the pRS304 plasmid, and which of those restriction sites are within the MCS. You should use the genomic sequence of pRS304 (which can be found in `pRS304.fasta`) and window settings that match the coordinates of the MCS region of the plasmid, 1891–1993.

Write another function called `report_pRS304_MCS_sites` that will **report** relevant summary information about the output. This function should take as input the dictionary returned from the above call to `find_restriction_sites`. It should **report** to the the user, for each restriction enzyme, how often that enzyme cuts the plasmid outside the MCS, how often it cuts the plasmid inside the MCS, and relevant details about any cut sites located inside the MCS.

Use this function to **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 enzymes and sites would you use? Where are they located? Note that to excise Aim2, you will have to cut at two restriction sites (one upstream and one downstream of the Aim2 gene). However, to integrate into the pRS304 plasmid, you should only have one cut on the entire plasmid. 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 excise the sequence fragment between the restriction sites of your choosing (the excised fragment should contain the entire Aim2 gene and some bits of additional sequence up- and downstream). Have your code **report** the nucleotide positions (with respect to the original sequence) for the beginning and end of the excised fragment (if there are multiple sites your chosen enzymes can cut, these should be the positions for the shortest fragment that contains the gene). Store the excised sequence in another variable.

n) Finally, write code to insert your excised sequence into the pRS304 plasmid at the proper location. Specifically, your code should produce the new plasmid sequence that would result after cloning in the chosen Aim2 gene fragment (and assuming that the backbone nicks for the sticky ends are properly ligated).

Apply

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: Let's start to stop COVID-19 (15 points)

List of files to submit:

1. `README.problem2.[txt/pdf]`
2. `orfs.py`

Plan

A critical step in stopping COVID-19 is understanding the biology of the SARS-CoV-2 virus that causes it. To do that, we can first identify all the open reading frames (ORFs) in the SARS-CoV-2 genome, demarcating them by their start and stop codons. These ORFs represent the potential proteins that the virus instructs our infected cells to make. After finding these ORFs, we can determine the amino sequences of the corresponding proteins, and continue to characterize them further. These proteins can (and have!) become targets for therapies and vaccines against SARS-CoV-2 infection.

As discussed in class, in genomes that lack introns, we can identify all the potential protein-coding genes by locating all of the ORFs. An ORF begins with a start codon (AUG in RNA viruses like SARS-CoV-2) and ends with a stop codon (UAG, UGA, or UAA) somewhere downstream in the same reading frame.

Note that AUG is the only triplet coding for methionine (Met). Thus, 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. For this reason, when searching for ORFs in a genome, please only return the longest ORF sharing a given stop codon (i.e., if the nucleotide sequence is ... AUG CGU AUG AAG AUG UCA UAG ..., return only the ORF: AUG CGU AUG AAG AUG UCA UAG, and not the nested substrings AUG AAG AUG UCA UAG or AUG UCA UAG).

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:

`'start'`: the start position of the ORF (the first nucleotide of the start codon)

`'stop'`: the stop position of the ORF (recall that we consider the stop codon part of the ORF, so the stop position will be the last nucleotide of the stop codon)

`'stopcodon'`: specific stop codon sequence (UAG, UGA, or UAA for RNA)

`'nlength'`: length of the ORF in nucleotides

`'aalength'`: length of the translated peptide in amino acids

`'frame'`: reading frame with respect to the start of the genome (0, 1, or 2)[3]

`'strand'`: the strand on which the ORF is found (`W` or `C`)[4]. *Note*: For single-stranded sequences, `'strand'` will always be set to `W`. However, the strand key will become more relevant in Problem 3 when we ask you to apply the ORF finder to a genomic sequence that is double-stranded.

The list returned should look something like:

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

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.

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

Apply

b) Apply your functions to the SARS-CoV-2 genome in `sars_cov2_wu.fasta`. Have your code **report** the number of ORFs if the minimum ORF length is 10 amino acids, 40 amino acids, or 60 amino acids. Also **report** the average length (in amino acids) of the identified ORFs for the three cases.

After you have found ORFs using your code, you may want to study characteristics such as their length and position in the genome, among other things. Since a genome is a linear sequence of nucleotides, you can imagine it as a straight line, with ORFs being subintervals of the line based on their start and end coordinates. A genome browser uses that sort of representation and is a powerful tool to visualize genomic interval data including ORFs, spliced genes, binding sites, etc. Here, you can use the UCSC genome browser for your analysis. Before you dive into the visualization analysis of the COVID-19 genome, first look at the human genome in the UCSC genome browser and take some time to learn how to navigate.

Open the following link in your internet browser to visualize the human genome: https://genome.ucsc.edu/cgi-bin/hgTracks?db=hg38&lastVirtModeType=default&lastVirtModeExtraState=&virtModeType=default&virtMode=0&nonVirtPosition=&position=chrX%3A15560138%2D15602945&hgsid=882870467_OyAJik6tZ8HJjDsE1N7AnXrrZqk1. You will notice that the link opens up to a segment of the human genome (chrX:15,560,138–15,602,945). Note which human gene is centered in this particular browser view (Why did we select this gene?). You can zoom in or out of the segment to look at finer details or to get a bigger picture. You can also click and drag to change the segment displayed. You can view any other segment of the genome by updating the genome coordinates to the segment you wish to look at. Aside

[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.

[4]The strands are defined as:
`W` = Watson strand (the default strand when calling `find_orfs` on a single strand of DNA or RNA)
`C` = Crick strand

from gene information, the genome browser can be used to visualize other types of experimental data, such as histone acetylation and sequence conservation, as seen in this browser window. In this way, a genome browser can be used to visualize the similarity between the genome sequences of multiple organisms (we will explore this at a later point in the course). For now, just develop a basic understanding of how to navigate around in a genome browser; you do not need to write anything in your `README` about your exploration.

c) Next, open the UCSC genome browser to visualize the SARS-CoV-2 genome using the following link: https://genome.ucsc.edu/s/sneha/SARS%2DCoV%2D2. The genome browser will display the actual annotated ORFs in the SARS-CoV-2 genome as displayed in Figure 2. How many ORFs are there in the genome? You can click on the ORFs to learn more information about them. What are the coordinates of ORF10 and what is its length in nucleotides? What are the coordinates of the ORF for the spike protein (S), and how many amino acids are in the spike protein after it is translated from the SARS-CoV-2 genome?

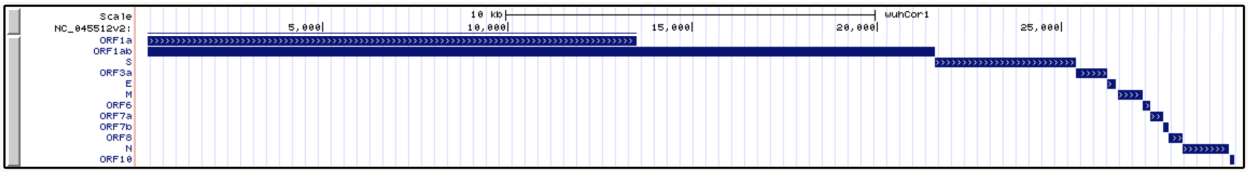

Figure 2: SARS-CoV-2 genome map of annotated ORFs visualized in UCSC genome browser.

Reflect

d) Consider the results generated by your code 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?

e) Take a closer look at the ORFs identified when the minimum ORF length is set to 60. Compare these ORFs with the annotated ORFs depicted in the UCSC genome browser (https://genome.ucsc.edu/s/sneha/SARS%2DCoV%2D2). 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?*

Extra challenge 1: Can you improve your code to address these discrepancies?

Extra challenge 2: Can you improve your code to translate the various ORFs into amino acid sequences? A function in `compsci260lib.py` should make this easy. In particular, can you report the amino acid sequence of the spike protein? Recall from the news that the spike protein has been evolving since the start of the pandemic, with the latest omicron variants containing a few mutations compared to the original strain you're studying here. Perhaps you can even find an article describing the locations of those mutations and compare them to what you found here.

Problem 3: Plasmid assembly (25 points)

List of files to submit:

1. `README.problem3.[txt/pdf]`
2. `plasmid.py`
3. `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, which we will discuss in more detail later) only permits the determination of around 500 to 800 base pairs at a time, and these short stretches of determined sequence are called "reads". But we can collect these reads from bazillions of random locations in the genome. Then, to solve what is known as the *genome assembly problem*, algorithms are used to computationally scan for reads (blue segments in Figure 3) whose sequences partially overlap those of other reads, and then computationally merge the sequences of the overlapping reads into longer sequences of contiguous bases, nicknamed 'contigs' (red segments in Figure 3, which exist contiguously in the genome, but collectively may not include all of the genome based on which reads were collected).

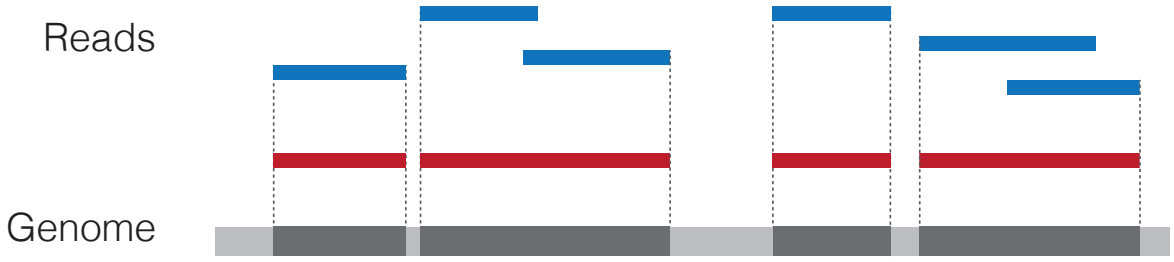

Figure 3: Genome assembly

a) Sketch (describe at a high level) a computational procedure that takes an arbitrary number of collected sequence reads—each of which is a randomly located subsequence of the source genome—and assembles them into a longer, continuous sequence.

Discuss your approach to assembling the original source genome from the reads, and the challenges that might arise using your approach. Some of the points that you may discuss in your answer are as follows: What factors would make this endeavor more challenging than it first seems, and 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, errors in the reads, or the degree to which the reads overlap? How might your thinking be influenced by any prior knowledge you may (or may not) have about the genome from which the reads were generated? Your discussion may include computational and/or biological aspects.

Develop

In the following steps, you will write a Python program called `plasmid.py` to solve an assembly problem that we have made much simpler than what occurs in reality, but with one interesting twist (so to speak): the DNA whose sequence you are trying to determine is that of a circular bacterial plasmid.

We provide you with an input file, `plasmid.fasta`, that contains fourteen partially 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 when looking for overlaps), 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 when looking for overlaps). In a real assembly task, none of these assumptions would be true so you'd need something a bit more sophisticated.

b) In `plasmid.py`, write a function called `simple_assembler` that will reassemble a full circular plasmid sequence from any 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 that your code should handle a list of reads that is arbitrarily long, not specifically 14) and return the assembled string sequence. Because the assembled plasmid is circular, we could define its start anywhere; for consistency, your function should adopt the convention that it returns the plasmid sequence assuming it starts with whatever is the first read that appears in its input list. So the first nucleotide of the first read in its input is also the first nucleotide of its output.

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, so you'll have to ensure it goes to the front of the list before you call `simple_assembler`). **Report** the complete assembled plasmid sequence. (*Hint:* You should find the `get_fasta_dict` function useful for importing all the reads from the file.)

Apply

c) How many nucleotides are in the plasmid sequence you assemble? (*Hint:* So you can double-check your work up to this point, we'll tell you that the answer you get should be a number 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-CoV-2 genome is somewhat simpler than searching for ORFs in organisms with DNA genomes because the SARS-CoV-2 genome is single-stranded. 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.

A further complication is that SARS-CoV-2 had a linear genome, whereas your reconstructed plasmid is circular, which means that the plasmid sequence you read in from a file actually wraps around on itself and an ORF could span across that (i.e., the ORF could start near the end of the plasmid sequence given in the file and end somewhere near the start of the plasmid sequence). All prokaryotic organisms (i.e., bacteria and archaea) possess circular genomes (and circular plasmid sequences), so this is not an uncommon problem.

Develop

e) Write a function called `find_orfs_circular_double_stranded` to search for ORFs within your reconstructed double-stranded plasmid. Remember that the plasmid is circular; thus, searching for and storing the ORFs you find will not be as simple as in Problem 2. One simplifying fact is that we have ensured the

plasmid's length is a multiple of three so your code need not worry about the reading frame changing as you roll around the plasmid (another thing that would make this a little harder in real life applications). Since this function will end up being a fancier version of `find_orfs` from Problem 2, you can copy your final code from that function into `plasmid.py` as a starting point.

Be sure that your search includes all 6 possible reading frames (3 frames on each strand) and permits you to find ORFs that overlap one another, if any do (by construction, two ORFs cannot overlap within one reading frame, but they could overlap if they are in different reading frames).

`find_orfs_circular_double_stranded` 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:

`'start'`: start position (first nucleotide) of the ORF

`'stop'`: stop position of the ORF (we consider the stop codon part of the ORF, therefore the stop position will be the last nucleotide of the stop codon)

`'stopcodon'`: specific stop codon sequence (TAG, TGA, or TAA since we are looking in DNA)

`'nlength'`: length of the ORF in nucleotides

`'aalength'`: length of the translated peptide in amino acids

`'frame'`: reading frame with respect to the start of the strand (0, 1, or 2)[5]

`'strand'`: the strand on which the ORF is found (`W` or `C`)[6]

Note that the start and stop positions should be reported *in relation to the 5' end of the strand on which the ORF was identified*. This should allow you to use a single helper function to process each of the two strands (if you use this strategy, you'll need to pass in to the helper function whether the strand it's processing is Watson or Crick, and the helper function will still need to remember that each strand is part of a circular plasmid).

Apply

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 even though we consider the stop codon to be part of an ORF, it is *not* considered a part of a coding sequence since it does not code for an amino acid.

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

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

[6] The strands are defined as:
`W` = Watson strand, here the one that is provided as input
`C` = Crick strand, here the reverse complement of the one provided as input

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 a bacterium, are you surprised to find the gene for it on a bacterial plasmid? *How could it have gotten there?*