**COMPSCI 260, Introduction to Computational Genomics** **26 January 2021**

# Problem Set 1

*Due: 5pm on 2 February 2021*

The main purpose of this first problem set is to familiarize yourself with using the Python programming language to manipulate 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 by the end of the problem set, you'll be analyzing the SARS-CoV-2 genome!

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, let's consider the yeast Aim2 gene, which is located between nucleotides 1001 and 1741 (inclusive) in the `aim2_plus_minus_1kb.fasta` file (you will be working with this file in Problem 1). The subsequence corresponding to the gene begins with a start codon and ends with a stop codon, but the portion of this subsequence that actually codes for a protein is only nucleotides 1001 through 1738.

## Problem 1: Attack of the clones (25 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 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 can then be 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?

**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? 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 potential restriction sites for six different restriction enzymes in the genomic region we provided which includes the Aim2 gene. Deciding on the best restriction enzyme to use is a necessary step in any cloning experiment, and you can assess this computationally.

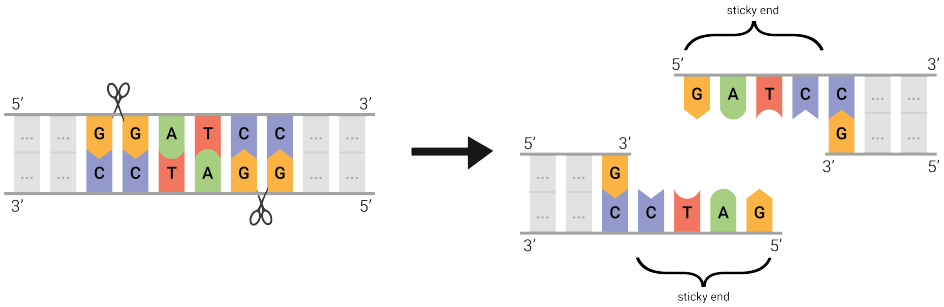


Figure 1: When the colored subsequence of the double-stranded DNA is recognized by the restriction enzyme BamHI, it binds temporarily and cleaves the two strands in the indicated locations, leaving behind two overhanging "sticky ends". The enzyme cleaves a covalent phosphodiester bond within the backbone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

Restriction sites may occur multiple times within any given genomic sequence. 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. 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. Look over the function and its signature to be sure you know how to call it and what it returns.) This means that the part of the sequence provided which 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**:

- 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 or 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 that your code is reporting correctly.

**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 Aim2 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 (*Note:* When we provide snippets like this to indicate what the dictionary should look like, we are describing only the form of the answer; the values shown below are 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 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? (You might think about organizing your report 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. 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 any enzyme that has a restriction site within the Aim2 gene body or the gene will no longer function. Ultimately, 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.

**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)** Using the regular expressions in `get_restriction_enzymes_regex`, write a function called `find_pRS304_restriction_sites` to determine if any of the enzymes in the list provided have a restriction site within the pRS304 plasmid (whose sequence can be found in `pRS304.fasta`).

*Note:* This function will be similar to the previously written `find_aim2_restriction_enzymes`. So you may find it helpful to borrow some of that code.

Your function should take as input the genomic sequence of pRS304 and return as output a dictionary mapping restriction enzymes to lists of sites. Each site will be formatted with the following keys:

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

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

The returned dictionary should look something like:

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

Once that function is ready, write another function called `report_pRS304_MCS_sites` that can **report** relevant summary information about the output. This function should take as input the dictionary returned above, along with the start and end coordinates of the MCS region of the plasmid (whose location is nucleotides 1887–1994). 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 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 extract the sequence between the restriction sites of your choosing; the sequence should contain the entire Aim2 gene and some bits of additional sequence up- and downstream. Have your code **report** the nucleotide positions for the beginning and end of the *shortest* excised gene region (with respect to the original sequence). 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.

## 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: Putting a stop to 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 to understand the workings 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 by demarcating them by their start and stop codons. These ORFs represent the potential proteins that the virus instructs our infected cells to make. After find these ORFs, we could 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, 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. 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 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:

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].
7. The strand of the found ORF (+ or −). Note: for single-stranded RNA sequences, this field will always be set to +. This field will become relevant in the next problem set when we ask you to apply the ORF finder to a genomic sequence that is double-stranded DNA[4].

[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:
`+` = Watson strand (the default strand when calling `find_orfs` on a single strand of DNA or RNA)
`-` = Crick strand

The list returned should look something like:

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

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 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.

## 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 long sequence of nucleotides, you could imagine writing it down on a straight line and then marking the ORFs on it using on their start and end coordinates. A genome browser does just that and is a powerful tool to visualize genomic data such as ORFs, spliced genes, 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:15,569,258–15,612,065). You can view another segment of the genome by updating the genome coordinates to the segment that you wish to look at. You can also 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. Aside from gene information, the genome browser can also be used to visualize other types of experimental data, such as DNase-seq, as seen in this browser window. The genome browser can also be used to analyze the similarity between the genome sequences of multiple organisms (we will explore this at a later point in the course). This paragraph is here to help you understand how the UCSC genome browser works; you do not need to write anything in your README related to this paragraph.

**c)** 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 will be in the spike protein?

## 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

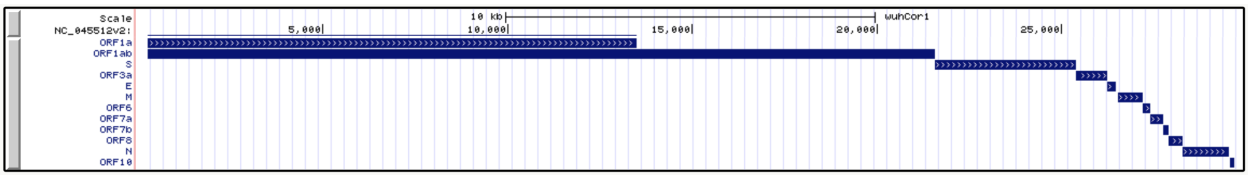


Figure 2: SARS-CoV-2 genome map of annotated ORFs visualized in UCSC genome browser.

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*: How can you improve your code to address these discrepancies?