Milestone 1 (27 points)

Due date: March 10, 2021

Introduction

Lab03: "DNA" introduced you to writing functions that manipulate DNA strings. We'll kick off the project with similar problems. Your team will be building off parts of Lab03 to complete problems from the ROSALIND project. Each of the functions is defined below — it is up to you and your team to divide up the work and combine your code into one file.

Team Check-In & Progress Report (8 points)

All teammates will check in together with the TA once a week to present your progress report and updated task list. Delivery method will be up to your section TA.

Peer Evaluation (2 points)

Your TA will release a peer evaluation survey after the deadline. This survey will be completely anonymous and privy only to yourself and the TA. This is your chance to voice any concerns (or lack thereof) about your team so we can get things on track for the next leg of the project.

- Note that you will receive a **2-point deduction** to your *individual* milestone score if you do not submit a peer evaluation.

Coding Style (2 points)

You will be assessed one point for using good naming conventions for functions and variables (beyond the mandated names given below for your submission). You will be assessed another point based on your adherence to good Python style, as outlined in the Project Rubric.

Submission

The Integrator is responsible for submitting one Python file per group to Gradescope. Make sure you add your team members in your submission!

- **Late submissions:** 5 points will be deducted from your total Milestone score per day after the deadline.
- **Missing teammates:** 1 point will be deducted from your total Milestone score if you do not add your grade members to your Gradescope submission by the deadline!

Coding Assignment (15 points)

Create your own Python file in a code editor of choice and begin writing functions as defined below. Your function names, inputs, and outputs should be exactly as described. Please make sure you follow the coding guidelines from the main rubric!

```
def dna_count(dna):
```

This function takes in a string representing a DNA sequence and returns a list or tuple of four integers representing the number of times each nucleotide A, C, G, or T occur in the DNA string, in that order. (HINT: You did this in Lab03...)

Example:

- Sample input DNA string:
  `"AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC"`
- The returned list should be:
  `[20, 12 , 17 , 21]`

```
def dna2rna(dna):
```

This function takes in a string representing a DNA sequence and returns the transcribed RNA string. DNA sequences are transcribed to RNA sequences by replacing all occurrences of the "T" nucleotides with "U". (HINT: You did this in Lab03...)

Example:

- Sample input DNA string:
  `"GATGGAACTTGACTACGTAAATT"`
- The returned RNA string:
  `"GAUGGAACUUGACUACGUAAAUU"`

```
def reverse_complement(dna):
```

This function takes in a string representing a DNA sequence and returns the reverse complement as a new string. The reverse complement of a DNA string is formed by reversing the entire string, and then taking the complement of each nucleotide. Recall from Lab03 that "A" and "T" are complements of each other, as are "C" and "G". (HINT: You created complementary DNA sequences in Lab03...)

Example:

- Sample input DNA string:
  `"AAAACCCGGT"`

- The returned reverse complement string:
  `"ACCGGGTTTT"`

```
def mendels_law(hom, het, rec):
```

Gregor Mendel's laws of inheritance state that organisms possess a pair of alleles for a given trait. If those two alleles are the same, then it is called homozygous; if they are different, it is heterozygous. For a given trait, are two possible alleles: dominant and recessive. Only one dominant allele is necessary for the trait to be expressed. Watch this 3-minute TED-Ed video for a more detailed explanation of Mendel's laws.

The `mendels_law()` function takes in three positive integers representing the number of organisms that are homozygous dominant, heterozygous, and homozygous recessive. It should return the probability that the offspring of two randomly selected organisms will possess a dominant allele (AKA it displays the dominant phenotype).

Example:

- Sample function call with integer inputs:
  `mendels_law(2, 2, 2)`

- The returned probability:
  `0.78333`

```
def fibonacci_rabbits(n, k):
```

This function takes in two integers `n` and `k` and calculates the total number of rabbit pairs that will be present after `n` months, given every pair of mating rabbits produces `k` rabbit pairs in their next litter. Assume we begin with 1 pair, and that the reproduction of rabbits is represented by the Fibonacci sequence (which was actually a mathematical exercise to calculate the population of rabbits over time!):

$$F_n = F_{n-1} + F_{n-2}$$

$$\text{with } F_1 = F_2 = 1$$

Note that you'll have to account for the rabbits producing `k` pairs of offspring every generation. You will also need to use a coding method called recursion. You can watch the first 4 minutes of this tutorial for an overview of recursion (the rest is examples).

Example:

- Sample function call with integer inputs:
  `fibonacci_rabbits(5, 3)`

- The returned number of rabbit pairs:
  `19`

```
def GC_content(dna_list):
```

This function takes in a list of DNA strings and returns the index of the DNA string with the highest GC-content and its GC-content percentage as a tuple. The GC-content of a DNA string is the percentage of nucleotides in the string that are "C" or "G".

Example:

- Sample input list
  `["CCTGCGGAAGATCGGCACTAGAATAGCCAGAACCGTTTCTCTGAGGCTTCCGGCCTTCCCTCCCACTAATAATTCTGAGG","CCATCGGTAGCGCATCCTTAGTCCAATTAAGTCCCTATCCAGGCGCTCCGCCGAAGGTCTATATCCATTTGTCAGCAGACACGC","CCACCCTCGTGGTATGGCTAGGCATTCAGGAACCGGAGAACGCTTCAGACCAGCCCGGACTGGGAACCTGCGGGCAGTAGGTGGAAT"]`

- The returned tuple:
  `(2, 60.919540)`

```
def rna2codon(rna):
```

This function takes in a string representing an RNA sequence, and returns the corresponding amino acid string, as transcribed by this codon table. You do not need to transcribe the stop codon. (HINT: You already did this in Lab03, go open it up and figure out how to incorporate it here!)

Example:

- Sample RNA string:
  `"AUGGCCAUGGCGCCCAGAACUGAGAUCAAUAGUACCCGUAUUAACGGGUGA"`

- The returned protein string:
  `"MAMAPRTEINSTRING"`

```python
def locate_substring(dna_snippet, dna):
```

This function takes in two strings, `dna_snippet` and `dna`, where `dna_snippet` is a substring of `dna`, and returns all locations of the substring as a list of integers. In other words, `dna_snippet` may exist in multiple locations within `dna`; you should return the beginning index position of each occurrence of `dna_snippet` inside of `dna`.

Example:

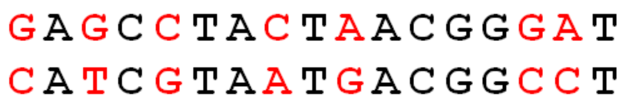
- Sample DNA snippet (substring):
  `"ATAT"`
- Sample DNA string:
  `"GATATATGCATATACTT"`
- The returned position list:
  `[1, 3, 9]`

```python
def hamming_dist(dna1, dna2):
```

This function takes in two strings representing DNA sequences of equal length and returns the Hamming distance between the two strings as an integer. The Hamming distance refers to the number of corresponding symbols in the same position that differ between two sequences. In other words, you should find the number of differences between two DNA sequences.

Example:

- Sample input DNA strings:
  `"GAGCCTACTAACGGGAT"`
  `"CATCGTAATGACGGCCT"`
- The returned Hamming distance:
  `7`



There are 7 mismatched symbols colored in red. Thus, the Hamming distance is 7.

```python
def count_dom_phenotype(genotypes):
```

This function takes in a list of six nonnegative integers corresponding to the number of couples in a population possessing a specific genotype pairing, such as AA-AA (both homozygous dominant) or Aa-aa (heterozygous and homozygous recessive). The list of integers represent the number of couples having the following genotypes, in this order:

1. AA-AA
2. AA-Aa
3. AA-aa
4. Aa-Aa
5. Aa-aa
6. aa-aa

The function should return the expected number of offspring displaying the dominant phenotype in the next generation, assuming that every couple has exactly two offspring.

Example:

- Sample input list:
  `[1, 0, 0, 1, 0, 1]`

- The returned number of dominant phenotype offspring:
  `3.5`

```
def source_rna(protein):
```

This function takes in a string representing a sequence of proteins and returns the total number of different source RNA strings, modulo 1,000,000. Essentially, you are calculating how many strands of mRNA from which this particular protein could have been translated. (HINT: You"ll want to use/manipulate the codon table from `rna2codon()`!)

Example:

- Sample input protein string:
  `"MA"`

- The returned number of possible source RNA sequences:
  `12`

```
def splice_rna(dna, intron_list):
```

This function takes in a string representing a DNA sequence and a list of strings representing introns. The process of transcribing DNA into RNA involves translating the DNA to RNA and then performing RNA splicing, where the sequence is chopped into smaller segments called introns and exons. Introns are segments of the gene not used for protein translation, so they should be removed from the sequence. Exons are the remaining segments, which are then transcribed sequentially into a protein string.

`splice_rna()` should return a protein string that results from transcribing and translating the exons of the given string. (HINT: You should use some of your previous functions. It's similar to the last part of Lab03!)

Example:

- Sample input DNA string:
  `"ATGGTCTACATAGCTGACAAACAGCACGTAGCAATCGGTCGAATCTCGAGAGGCATATGGTCACATGATCGGTCGAGCGTGTTTCAAAGTTTGCGCCTAG"`

- Sample intron list:
  `["ATCGGTCGAA", "ATCGGTCGAGCGTGT"]`

- The returned protein string:
  `"MVYIADKQHVASREAYGHMFKVCA"`