INFO 190S Introduction to Programming

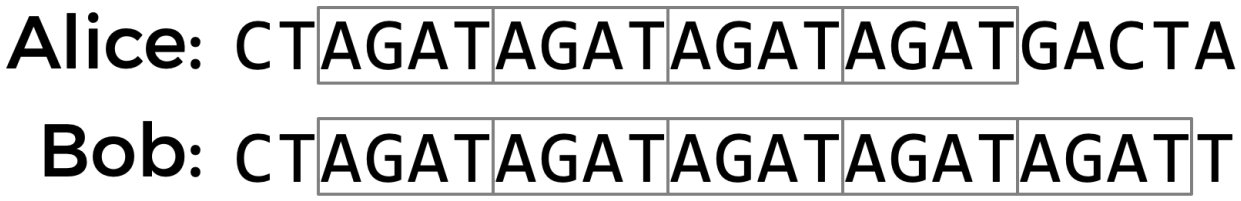
Project 3 DNA Profiler

Background

DNA, the carrier of genetic information in living things, has been used in criminal justice for decades. But how, exactly, does DNA profiling work? Given a sequence of DNA, how can forensic investigators identify to whom it belongs?

Well, DNA is really just a sequence of molecules called nucleotides, arranged into a particular shape (a double helix). Each nucleotide of DNA contains one of four different bases: adenine (A), cytosine (C), guanine (G), or thymine (T). Every human cell has billions of these nucleotides arranged in sequence. Some portions of this sequence (i.e. genome) are the same, or at least very similar, across almost all humans, but other portions of the sequence have a higher genetic diversity and thus vary more across the population.

One place where DNA tends to have high genetic diversity is in Short Tandem Repeats (STRs). An STR is a short sequence of DNA bases that tends to be repeated back-to-back numerous times at specific locations in DNA. The number of times any particular STR repeats varies a lot among different people. In the DNA samples below, for example, Alice has the STR AGAT repeated four times in her DNA, while Bob has the same STR repeated five times.



Using multiple STRs, rather than just one, can improve the accuracy of DNA profiling. If the probability that two people have the same number of repeats for a single STR is 5%, and the analyst looks at 10 different STRs, then the probability that two DNA samples match purely by chance is about 1 in 1 quadrillion (assuming all STRs are independent of each other). So if two DNA samples match in the number of repeats for each of the STRs, the analyst can be pretty confident they came from the same person. CODIS, The FBI's DNA database, uses 20 different STRs as part of its DNA profiling process.

What might such a DNA database look like? Well, in its simplest form, you could imagine formatting a DNA database as a CSV file, wherein each row corresponds to an individual, and each column corresponds to a particular STR. Here is an example of an STR file that records the name of individuals and the number of repeats for particular STRs.

```
name,AGAT,AATG,TATC
Alice,5,2,8
Bob,3,7,4
Charlie,6,1,5
```

The data in the above file indicates that Alice has the sequence AGAT repeated 5 times consecutively somewhere in her DNA, the sequence AATG repeated 2 times, and TATC repeated 8 times. Bob, meanwhile, has those same three STRs repeated 3 times, 7 times, and 4 times, respectively. And Charlie has those same three STRs repeated 6, 1, and 5 times, respectively.

So given a sequence of DNA, how might you identify to whom it belongs? Well, imagine that you looked through the DNA sequence for the longest consecutive sequence of repeated AGATs and found that the longest sequence was 3 repeats long. If you then found that the longest sequence of AATGs is 7 repeats long, and the longest sequence of TATC is 4 repeats long, that would provide pretty good evidence that the DNA was Bob's. Of course, it's also possible that once you take the counts for each of the STRs, it doesn't match anyone in your DNA database, in which case you have no match. This would mean that you either do not have this person in your database or you need to gather more DNA evidence.

In practice, since analysts know on which chromosome and at which location in the DNA an STR will be found, they can localize their search to just a narrow section of DNA. But we'll ignore that detail for this problem and assume that we are given this narrow section in text files that we can use for our analysis.

Your task is to write a program that will take a sequence of DNA and a CSV file containing STR counts for a list of individuals and then output to whom the DNA (most likely) belongs.

Learning Objectives

After completing this assignment, you will be able to:

- Apply branching and looping constructs more effectively.
- Use import statements to use other modules.
- Use the **csv** module to read CSV files into memory.
- Use the **sys** module to process command line arguments.
- Use functions to break a problem down into smaller problems.
- Evaluate a problem according to its functional decomposition.
- Run a Python program from the terminal and pass in program arguments.
- Convert between different types of Python data
- Test functions to determine their correctness.

Specification

Objective: Implement a program that identifies to whom a sequence of DNA belongs. You must implement your solution in a Python file named **dna.py**.

1. The program will accept command line arguments. The first command line argument is an STR file and the second command line argument is a DNA file. We provide the STR database in a file named **str_profiles.csv** and four DNA files that you can use to test your implementation.

2. Your program must open the CSV file and read its contents into memory. We recommend that you use Python's csv module to do this. In particular, we suggest that you use csv.DictReader to read the

CSV file into Python dictionaries. Although you could implement this yourself, it is an important skill to get used to importing and using libraries that already exist if we suggest that you do so.

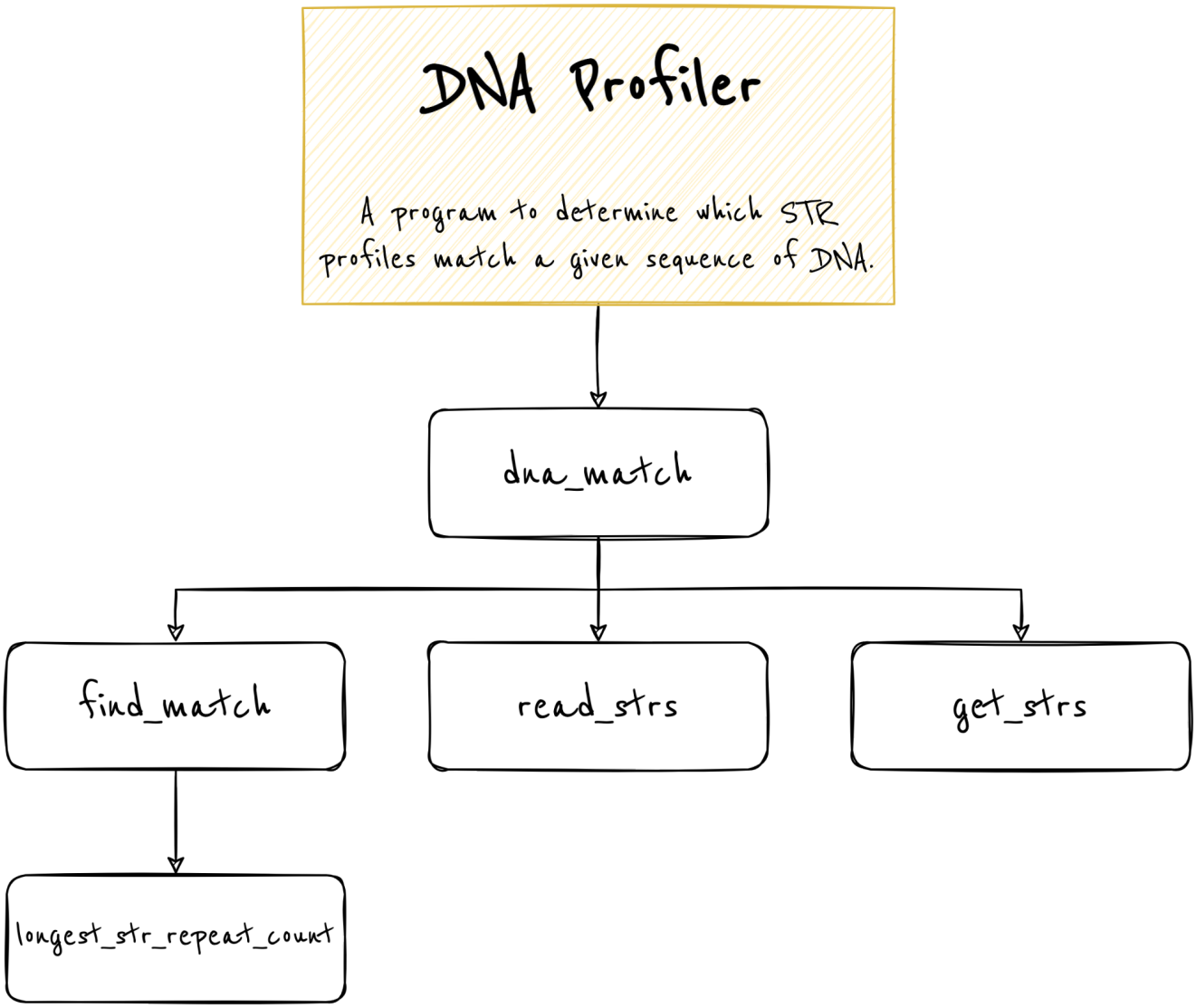
3. Your program must open the DNA sequence file and read its contents into memory.
4. For each of the STRs (after the first line of the CSV file), your program should compute the longest run of consecutive repeats of the STR in the DNA sequence we need to identify.
5. If the STR counts match exactly with any of the individuals in the CSV file, your program should print out the name of the matching individual. You may assume that the STR counts will not match more than one individual.
6. If the STR counts do not match exactly with any of the individuals in the CSV file, your program should print "No match".

Functional Decomposition

When solving problems computationally, it is important that we understand the problem we are trying to solve. Often, the definition of a problem is large and hard to understand exactly where to

start. We can use functional decomposition to break a complex problem down into smaller parts. Those smaller parts are easier to reason about and apply solutions to. If a smaller part is still too hard, we can further break it down into even smaller parts. We do so until we understand how to solve the problem.

In this assignment, we will demonstrate this process by providing the decomposition of the problem into functions that you are required to implement. However, you are welcome to implement additional functions if you feel that it would be helpful. We break the problem of DNA profiling down into the following functions:



As you can see in the above diagram, we break the DNA profiler problem down into a **dna_match** function which is then broken down into three additional functions that perform subtasks. You will also notice that we further break down the **find_match** problem down into the **longest_str_repeat_count** function. Below, we will describe what each function is supposed to do and you will need to implement each function to complete the DNA profiler program.

The above diagram helps us see the structure of the program which can be very helpful as programs become more complex. We encourage you to draw diagrams to help you understand what your code looks like. In addition, the above program tells you the order your functions must be defined in your Python code. Because a function must be defined before it is used, you can use this diagram to tell you which functions should be defined first in your code file. To do that, you work bottom up. The **longest_str_repeat_count** function must be defined before **find_match**. The **find_match**, **read_strs**, and **get_strs** functions must be

defined before **dna_match**. The order of definitions of the **find_match**, **read_strs**, and **get_strs** functions between each other doesn't matter as they do not depend on each other.

Project Setup

Create dna.py

The first thing you want to do to get started is create a new folder for this project and inside of it create a file called **dna.py** (the name of the file must be exact for the auto-grader to identify it). You must add the following contents, we will call the *main conditional*, to this file as a starting point:

```
if __name__ == '__main__':
    print('hello')
```

It is important that this code is always at the end of your file. So, any code that you write for the tasks below should be written above this code. You will also use this if block to write tests for the functions you implement to perform your own checks to make sure your functions are working according to the specifications.

Create the data folder

This program uses data contained in two different types of files. You should create a folder named **data** inside of your project folder. Next, download the data and place it in that data folder. You will use this data to test your functions and program.

Task 1: Reading DNA Files

The first task is to read a DNA text file into memory. A DNA text file contains only a single segment of DNA in ASCII format that looks like this:

```
AGACGGGTTACCATGACTATCTATCTATCTATCTATCTATCTATCTATCACGTACGTACGTATCGAGATAGATAGATAGATAGAT
CCTCGACTTCGATCGCAATGAATGCCAATAGACAAAA
```

You must implement two functions named **read_dna** and **dna_length**. Given the name of a file, the **read_dna** function reads the text file and returns the DNA sequence as a Python string. The **dna_length** function returns the length of the DNA sequence contained in a given DNA file name. The **dna_length** function must use the **read_dna** function to get the DNA string and then compute the DNA length. Here are the two functions along with their docstrings you need to implement:

```
def read_dna(dna_filename):
    """
    Reads the DNA file

    This function reads the DNA sequence from the given `dna_filename`
```

```
    file. It returns the DNA sequence read from the file as a string.

    Parameters:
      dna_filename      The DNA file name

    Returns:
      str   A string that is the DNA sequence read from the file.
    """
    pass


def dna_length(dna_filename):
    """
    Returns the length of a DNA sequence in the file `dna_filename`

    Parameters:
      dna_filename     The DNA file name

    Returns:
      int   An integer length of the DNA sequence
    """
    pass
```

As you design and implement these functions, you should test them by printing their output. A good place to put these calls are in the *main conditional*:

```
# We check if this module is the "main" module. If it is, we run the
# code inside of the `if` block.
if __name__ == '__main__':
    print(read_dna('dna_1.txt'))
    print(dna_length('dna_1.txt'))
```

You should verify that what is printed is correct by comparing what is printed out to the contents of the file you read in. Only after you have verified that your functions are working correctly, should you move onto the next task.

Note: you will not use **dna_length** in the rest of this project. We ask that you define it to get some practice before continuing on into the tasks below. However, you must keep this function in your submission to pass all of the tests.

Task 2: Reading STR Files

Your next job is to read an STR file containing STR profile data of particular people in CSV format. The contents of the STR file we provide to you looks like this:

```
name,AGAT,AATG,TATC
Alice,5,2,8
Bob,3,7,4
```

```
Charlie,6,1,5
```

You are to write two functions, the first function **read_strs** will read an STR file with a given name into a Python dictionary where the keys are the items in the first row of the STR file and the values are each of the subsequent rows. This function returns that dictionary. For example, if we call the function **read_strs('str_profiles.csv')**, the dictionary return value would be:

```
[{'name': 'Alice', 'AGAT': '5', 'AATG': '2', 'TATC': '8'},
 {'name': 'Bob', 'AGAT': '3', 'AATG': '7', 'TATC': '4'},
 {'name': 'Charlie', 'AGAT': '6', 'AATG': '1', 'TATC': '5'}]
```

The second function, **get_strs**, returns a list of tuples where each tuple is a *(STR, repeat-count)* pair for a given person's STR profile in dictionary-format like one of the entries in the list returned by the **read_strs** function above. For example, given the dictionary value above referenced by the variable **suspects**, calling **get_strs(suspects)** returns the value:

```
[('AGAT', 5), ('AATG', 2), ('TATC', 8)]
```

Here are the two functions along with their docstrings (that you should read) you need to implement:

```
def read_strs(str_filename):
    """
    Reads the STRs from the given `str_filename` file

    The STR file is a CSV file containing STR repeats for certain
    people. An example of this file looks like this:

      name,AGAT,AATG,TATC
      Alice,5,2,8
      Bob,3,7,4
      Charlie,6,1,5

    This function must read the file using the `csv` module and return
    a list of dictionary objects that look like this:

      [{'name': 'Alice', 'AGAT': '5', 'AATG': '2', 'TATC': '8'},
       {'name': 'Bob', 'AGAT': '3', 'AATG': '7', 'TATC': '4'},
       {'name': 'Charlie', 'AGAT': '6', 'AATG': '1', 'TATC': '5'}]

    Parameters:
      str_filename      The STR file name

    Returns:
      list of dicts   A list of dictionary objects read from the CSV file
    """
    pass
```

```
def get_strs(str_profile):
    """
    Returns a tuple of (STR, repeats) pairs

    Given a dictionary representation of an STR profile that looks
    like this:

      {'name': 'Alice', 'AGAT': '5', 'AATG': '2', 'TATC': '8'}

    return a list of tuples that looks like this:

      [('AGAT', 5), ('AATG', 2), ('TATC', 8)]

    Note: the repeat is an `int`, not a `string`.

    Parameters:
      str_profile      A STR profile in dictionary form

    Returns:
      list of tuples   A list of (STR, repeats) pairs
    """
    pass
```

As you design and implement these functions, you should test them by adding calls to these functions in your program file such as:

```
if __name__ == '__main__':
    profiles = read_strs('str_profiles.csv')
    print(profiles)
    print(get_strs(profiles[0]))
    print(get_strs(profiles[0])[0])
```

You should verify that what is printed is correct by comparing what is printed out to the contents of the file read in. You may also want to construct tests that are a bit more specific:

```
print(get_strs(profiles[0])[0] == ('AGAT', 5))
```

This will print **True** if the first tuple in the return value from **get_strs** is equal to the tuple **('AGAT', 5)**. This is known as an *assertion*. We are asserting that the result returned from the **get_strs** function must be equal to the value we specify. Indeed, this is so common that Python provides a built-in mechanism for specifying such things:

```
```

```
assert(get_strs(profiles[0])[0] == ('AGAT', 5))
```

When you use the **assert** function, if your assertion is *not true*, the program will fail with an error message. If what you assert *is true*, the program doesn't print anything. It is common to use **assert** in your code to ensure that the values your programs produce as it executes are what you expect they are. You are encouraged to use them in your functions to verify that they are executing properly. If there are no problems, they have no effect on the logic of your program.

Task 3: Longest STR Repeat Counts

This task requires you to implement a single function. It is likely the most difficult function out of all of them. Given an STR such as TATC you need to find the longest repeat count of that STR in a given DNA sequence. For example, given the following DNA sequence:

```
AGACGGGTTACCATGACTATCTATCTATCTATCTATCTATCTATCTATCACGTACGTACGTATCGAGATAGATAGATAGATAGAT
CCTCGACTTCGATCGCAATGAATGCCAATAGACAAAA
```

The longest repeated count of the STR TATC is 8. You can see this more clearly if we separate the DNA string slightly:

```
AGACGGGTTACCATGAC  TATC TATC TATC TATC TATC TATC TATC TATC  ACGTACGTACG  TATC
GAGATAGATAGATAGATAGATCCTCGACTTCGATCGCAATGAATGCCAATAGACAAAA
```

Your function must look for the start of a TATC fragment. Once it finds one, it needs to determine how many TATC fragments are repeated one after the other. Note, a fragment may start at any point in the DNA sequence and there may be more than one repeated sequence.

Here is the sole function, **longest_str_repeat_count**, along with its docstring (that you should read) that you need to implement:

```
def longest_str_repeat_count(str_frag, dna_seq):
    """
    Finds the longest match of a given STR DNA fragment in the given
    DNA sequence.

    This function returns the longest repeated occurance of the given
    STR fragment, `str_frag`, in the DNA sequence `dna_seq`. For
    example, given the STR AGAT and the DNA sequence:

    AGACGGGTTACCATGACTATCTATCTATCTATCTATCTATCTATCTATCACGTACGTACGTA
    TCGAGATAGATAGATAGATAGATCCTCGACTTCGATCGCAATGAATGCCAATAGACAAAA

    this function returns 5.
```

```
    Hints:

    1. You will want to loop over the `dna_seq` character by
       character using a while loop with an index
    2. You may find using string slicing convenient for this
       function. For example, `dna_seq[i:i+4]` will evaluate to a
       substring of `dna_seq` starting from i to i+4 exclusive.
    3. Do not use the `count` string method. It doesn't return the
       longest match, it returns the count. This would also be
       cheating.

    Parameters:
      str_frag     A fragment of DNA in a STR profile (e.g., AGAT)
      dna_seq      A DNA sequence

    Returns:
      int          The longest repeated occurance of the STR fragment
    """
    pass
```

If you find that part of this function could be written in a separate "helper" or "utility" function and called from this function to perform a subtask then we encourage you to do that. We will not test you on any additional functions you write, only the ones we name in this document.

You should think of ways in which you can test this function that might help you determine if you are implementing it correctly. We suggest calling this function with STR fragments and DNA sequences that you know will have particular counts to see if it works. Try to trick your function and break it. How might you be able to do that? Use the **print** statement to see what it produces. Use the **assert** statement to make assertions about what must be true to see if your function passes the test.

Task 4: Find a Match

The next task is to implement a function called **find_match**. It is significantly more straightforward compared to the previous function. Given an STR profile and a DNA sequence, match each STR fragment and its repeat value to the given DNA sequence and the number of actual repeat counts in that DNA sequence. A match is found if all repeat values in the profile are equal to the number of times that STR fragment appears in sequence in the DNA sequence.

If you are thinking that the **longest_str_repeat_count** function will come in handy, then you guessed right! Here is the function along with its docstring (that you should read) that you need to implement:

```
def find_match(str_profile, dna_seq):
    """
    Find a match given a specific STR profile

    This function compares the repeat values for each STR dna fragment
    X in the given `str_profile` to the count of that same X dna
```

```
    fragment in the provided DNA sequence `dna_seq`.

    For example, if we have a profile like this (a list of tuples):

      [('AGAT', 5), ('AATG', 2), ('TATC', 8)]

    We want to determine if the number of repeats for the STR
    fragments AGAT, AATG, and TATC for this profile, which is 5, 2,
    and 8, are the same number of repeats in the DNA sequence. If the
    repeat count in the DNA sequence for AGAT, AATG, and TATC are
    identical to this profile, then we have matched the profile to the
    DNA sequence.

    Hints:

      1. You want to use the `longest_str_repeat_count` function to
         find the longest count of repeats for each STR fragment in
         the DNA sequence. This will require you to iterate over the
         `str_profile` list.

    Parameters:
      str_profile  A list of tuples representing a person's STR
                   profile

    Returns:
      boolean      `True` if a match is found; `False` otherwise
    """
    pass
```

At this point you should follow the same pattern as the previous functions you have implemented to check your code. This will help you think critically about what your function does or is supposed to do.

Task 5: DNA Match

You are finally at the end! Assuming that you have tested the functions in the previous tasks and you are feeling pretty confident that they are working, then this task should be a breeze. This task requires you to implement a function called **dna_match** that will receive as arguments two file names. The first is the file name for the CSV file containing STR profile data and the second is the name of a file containing a DNA sequence. You want to use the following functions you defined previously to implement this function:

- **read_strs**
- **read_dna**
- **get_strs**
- **find_match**

This function uses the above functions and the two file names to determine if any profiles in the STR profile file match the DNA sequence. If it does, then we simply print the name of the person whose profile matches.

If it does not, we print "No Match". You can assume that the two files exist. Here is the function and its docstring:

```
def dna_match(str_filename, dna_filename):
    """
    Compares STRs to a DNA sequence

    This function reads the STRs in the `str_filename` file
    and the DNA sequence in the `dna_filename` file and compares
    the STRs to the DNA sequence to determine who the DNA sequence
    likely belongs to.

    Parameters:
      str_filename      The STR file name
      dna_filename      The DNA file name

    Returns:
      str   A string that is either the person's name in the STR file
            that matches the DNA sequence in the DNA file or
            'No match' if a match does not exist.
    """
    pass
```

Task 6: Driver Code

Now that you have completed all of the necessary functions for the DNA profiler program, you will need to implement code to “drive” the program. In other words, we want to provide a user interface for our program to make it easy for people who are unfamiliar with the Python language to run DNA profile matching. Your last task is to replace everything in your *main conditional* with code that will use values passed in from the console to run the DNA profiler program. The program accepts two arguments. The first argument is the name of the STR profile CSV file and the second argument is the name of a DNA sequence file. This code should ensure that the proper number of arguments have been passed to your program. If it did not receive the correct number of arguments it should print out the following on a single line and end its execution:

```
'Usage: python dna.py STR_FILE DNA_FILE'
```

Otherwise, the two file names should be passed to the **dna_match** function and print out the result of this function.

For more information about solving this task, check out Zybook chapter 10 section 5, or chapter 8 section 14.

Submission

Submit your **dna.py** solution file to Gradescope by the assigned due date.

The Gradescope autograder will test your submission and report failed tests with helpful messages to guide you towards a solution. If you are struggling, please visit office hours and post questions to the online discussion forum. Do not wait until the last minute to work on this project as you may not be able to get the answers you need soon enough to complete the assignment successfully.