CSC110 Assignment 3: Loops, Mutation, and Applications

In this assignment, you'll take what you learned about using loops and mutation over the past few weeks, and apply these programming techniques to a few new problem domains. In Parts 1 and 2, you will generate English texts by creating and using simple computational models. In Part 3, you will learn about *sentiment analysis* and debug a small program that analyses the sentiments found in movie reviews. And finally, in Part 4 you will implement two mathematical models used by epidemiologists to model the spread of infectious diseases and process some real-life data based on COVID-19 statistics from the Province of Ontario. What an exciting assignment!

Logistics

- Due date: **Friday, October 30th before 12pm noon Eastern Time**. (*Note*: the deadline has been extended from the original deadline, Tuesday October 27.)
- This assignment can be done with one partner or individually.
- You will submit your assignment solutions on MarkUs (see submission instructions at the end of this handout).
- Please review the Assignment Guidelines and Policies page (https://q.utoronto.ca/courses/178986/pages/policies-and-guidelines-assignments?module_item_id=1705343) and the Course Syllabus section on Academic Integrity (https://q.utoronto.ca/courses/178986/assignments/syllabus#academic-integrity).

Starter files

To obtain the starter files for this assignment:

1. Download `a3.zip` (starter-files/a3.zip).
2. Extract the contents of this zip file into your `csc110/assignments/` folder. It will create a new `a3` folder for you, with all the starter files inside. This should look similar to what you had for Assignment 1 (../../a1/handout).

3. Mark the `a3` folder as "Sources Root" by right-clicking on it and selecting Mark Directory as -> Sources Root.

General instructions

This assignment contains a mixture of both written and programming questions. All of your written work should be completed in the `a3.tex` starter file using the LaTeX typesetting language. You went through the process of getting started with LaTeX in the Software Installation Guide (https://q.utoronto.ca/courses/160038/pages/setting-up-your-computer-start-here?module_item_id=1346385), but for a quick start we recommend using the online platform Overleaf (https://www.overleaf.com/) for LaTeX. Overleaf also provides many tutorials (https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes) to help you get started with LaTeX.

Your programming work should be completed in the different starter files provided (each part has its own starter file). We have provided code at the bottom of each file for running doctest examples and PythonTA on each file. We are *not* grading doctests on this assignment, but encourage you to add some as a way to understand each function we've asked you to complete. We *are* using PythonTA to grade your work, so please run that on every Python file you submit using the code we've provided.

Warning: one of the purposes of this assignment is to evaluate your understanding and mastery of the concepts that we have covered so far. So on this assignment, you may only use parts of the Python programming language that we have covered in the first five weeks of lecture (i.e., up to and including all of Chapter 5 in the notes). Other parts are not allowed, and parts of your submissions that use them may receive a grade as low as **zero** for doing so.

- You *may* use augmented assignment operations (e.g., `+=`)
- You *may* use the alternate method call syntax (e.g., `text.lower()` rather than `str.lower(text)`

Part 1: Text generation, uniformly random model

Computers have become very good at predicting the next word in a sentence. Consider, for example, the touch-based keyboards on a smartphone. The accuracy of these predictions are based on two things: the model and the data used to "train" that model. In Assignment 1, we saw how to use data (both real and

randomly generated) to train a linear regression model that could predict the value of a variable y varie linearly with an independent variable x. In Parts 1 and 2, we will see how we can use textual data to create a model that generates a series of semi-sensible sentences.

The model

Consider the short sentence `'Hello Amy was here'`. One naive way to generate new sentences from this data is to *randomly select words* from this sentence and join them together to generate a new sentence. Here the start of a function to do so:

```
def generate_text_uniform(model: List[str], n: int) -> str:
    """Return a string of n randomly-generated words chosen from the given model.

    Each word in the returned string is separated by a single space.

    Preconditions:
        - n >= 0
        - model != []
        - all({str.isalnum(s) for s in model})
    """
```

For example, here is how we might call this function on the words in our given sentence:

```
>>> words = ['Hello', 'Amy', 'was', 'here']
>>> generate_text_uniform(words, 5)
'was Amy was was here'
>>> generate_text_uniform(words, 5)
'Hello Hello Amy here Hello'
```

This function makes `n` random choices from the given words, and can be implemented using a loop wit the following structure:

```
def generate_text_uniform(model: List[str], n: int) -> str:
    """Return a string of n randomly-generated words chosen from the given model.

    Each word in the returned string is separated by a single space.

    Preconditions:
        - n >= 0
        - model != []
        - all({len(s) > 0 for s in model})
        - all({str.isalnum(s) for s in model})
    """
    # ACCUMULATOR: a list of the randomly-generated words so far
    words_so_far = []

    for _ in range(0, n):
        random_word = ...  # Randomly choose a word from model
        list.append(words_so_far, random_word)

    return str.join(' ', words_so_far)
```

(Recall that `str.join` take a separator `str` and `List[str]`, and returns a new string containing all the words in the given list, separated by the separator. Try it in the Python console!)

1. Open `a3.tex` and answer the following questions.

    a. Complete the *loop accumulation table* for the example call to `generate_text_uniform(words, 5)` where `words = ['Hello', 'Amy', 'was', 'here'` and the final return value is `'Hello Hello Amy here Hello'`. This table doesn't show the loop variable, as it is not used inside the loop body.

    b. Explain why it wouldn't be a good idea to include this example for `generate_text_uniform` to check with `doctest`:

    ```
    """
    >>> words = ['Hello', 'Amy', 'was', 'here']
    >>> generate_text_uniform(words, 5)
    'Hello Hello Amy here Hello'
    """
    ```

c. What inputs to `generate_text_uniform` could we write doctest examples for? (I.e., inputs where the problem you described in part (b) doesn't apply.)

2. Open `a3_part1.py` and complete the functions `generate_text_uniform` and `create_model_uniform` according to their docstrings.

Part 2: Text Generation, One-Word Context Model

`generate_text_uniform` doesn't return meaningful sentences, whether we give it a short list of words or the entire collection of words in the English language. The reason is its model, which makes *independent random choices* for each word in the generated output. The reason independent word choices generate nonsensical sentences is that in English (and every other language), word order matters: each word in a sentence is very much tied to the previous and next words.

In this part, we'll develop and use a text model that stores not just the words from our input data, but some *context* of how each word is used as well.

The model

Consider the following sentence:

```
'I really really like chocolate'
```

The **one-word context model** for this text is a dictionary where:

- Each key is a word from the text.
- Each key's corresponding value is a set of words that *immediately follow* that word in the text. W call this set the *follow set* of the key.

The one-word context model for `'I really really like chocolate'` is the following:

```
{
    'I': {'really'},
    'really': {'really', 'like'},
    'like': {'chocolate'},
    'chocolate': set()
}
```

This model tells us:

- The word `'I'` can be followed by `'really'`. Its *follow set* is `{'really'}`.
- The word `'really'` can be followed by `'really'` or `'like'`.
- The word `'like'` can be followed by `'chocolate'`.
- The word `'chocolate'` has no words that follow it. Its follow set is empty.

0. (*Not to be handed in, but useful to complete*) answer the following questions:

    a. Write the one-word context model for the following string:

    ```
    'David is cool and funny and Mario is cool'
    ```

    b. Write a string whose one-word context model is the following:

    ```
    {
        'I': {'love'},
        'love': {'cats'},
        'hate': {'dogs'}
        'cats': {'and', 'hate'},
        'dogs': set(),
        'and': {'cats'}
    }
    ```

Before building this model, we'll add one additional piece of complexity: separating sentences in text. For this assignment we'll allow our input texts to only contain alphanumeric characters, spaces, and periods. Periods mark the end of a sentence, and are handled in a special way: whenever a word is followed by a period, the string `'.'` is added to the word's follow set in the one-word context model.

For example, the string `'I like chocolate. I really really like chocolate.'` has the model:

```
{
    'I': {'like', 'really'},
    'really': {'like', 'really'},
    'like': {'chocolate'},
    'chocolate': {'.'}
}
```

1. In `a3.tex`, write the one-word context model for the following string:

```
'Love is patient. Love is kind. It does not envy. It does not boast. It is not
        proud.'
```

2. In `a3_part2.py`, implement the functions `create_model_owc` and `update_follow_set`.

   Some details about the model:

   - The keys in the model are case-sensitive; so `'love'` and `'Love'` are treated as different words, and have different follow sets.
   - Every word in the input text is a key in the returned model.
   - `'.'` does not appear as a key in the dictionary.
   - Every word in the input text must have a non-empty follow set, except possibly the last word in the text.
   - The text does *not* need to end with a period; so it *is* possible for the last word in the text to have an empty follow set.

3. In `a3.tex`, complete the *loop accumulation table* to show how the loop in your `create_model_ov` function processes the input string `'I like chocolate. I really really like chocolate.'`

   You may modify the given table to suit your specific loop (e.g., the loop variable(s) or accumulator(s)).

Generating text

Now we're going to use our one-word context model to generate text, by implementing the following function:

```
def generate_text_owc(model: Dict[str, Set[str]], n: int) -> str:
    """Return a string containing n randomly generated words chosen from model.

    Preconditions:
        - n > 0
        - model is in the format described by the assignment handout
    """
    # ACCUMULATOR: a list of the randomly-generated words so far
    words_so_far = []

    for _ in range(0, n):
        # Randomly choose a word from model
        ...

        list.append(words_so_far, random_word)

    return str.join(' ', words_so_far)
```

This function has the same skeleton as `generate_text_uniform` from Part 1: generate `n` random word from the model and accumulate them into a list of words. Except now we aren't going to make random choices independent of each other—instead, whenever a word `w` is chosen, the *next* word to be chosen must come from the follow set of `w` stored in the `model`.

More concretely, `generate_text_owc` uses the following algorithm:

1. For the first word, pick a *random key* from the model.
2. Look up the follow set of the most recent word, and randomly choose an element from the follow set.
    - If the follow set is empty, instead choose a new random key from the model.
3. If the chosen element is a normal alphanumeric word, add the word to the accumulator and repea Step 2.
    - If the chosen element is `'.'`, instead add a `'.'` to the end of the most recent word and then choose a new random key from the model.
4. Repeat steps 2-3 until `n` words have been generated. The last word does *not* need to end with a period.

- Periods do not count as separate words.

Here are some examples of generated eight-word texts from our above "chocolate" text:

```
>>> my_model = {
...     'I': {'like', 'really'},
...     'really': {'like', 'really'},
...     'like': {'chocolate'},
...     'chocolate': {'.'}
... }
>>> generate_text_owc(my_model, 8)
'chocolate. I like chocolate. really really like chocolate'
>>> generate_text_owc(my_model, 8)
'really really like chocolate. I like chocolate. like'
>>> generate_text_owc(my_model, 8)
'I really really like chocolate. like chocolate. chocolate'
```

Neat! These sentences aren't exactly poetry, but they make more sense than the text generated by the `generate_text_uniform` from Part 1.

3. In `a3_part2.py`, complete the functions `generate_text_owc`, `generate_new_word`, and `generate_next_word` (the latter two are helper functions for `generate_text_owc`).

   Test your functions carefully in the Python console (you can copy-and-paste examples from this handout, and make up your own). At the bottom of the file we've given you some code that you ca use to read text from a "real" data file and use that to test your functions too.

   ```
   >>> text = load_text_data('data/texts/sample_text_clean.txt')
   >>> model = create_model_owc(text)
   >>> generate_text_owc(model, 20)
   'texts it is a few insidious Copy Writers ambushed her for their agency where
           they abused her seven versalia put'
   ```

Part 3: Loops and Mutation Debugging Exercise

A movie review typically includes a critic's comments and a score, but the score doesn't portray the sentiment and raw emotion in the critic's comments. Professor Mario has decided to build his own *movie review platform* that labels reviews as positive, negative, or neutral based on the intensity of the

words used in the review.

Professor Mario uses the VADER Lexicon (https://github.com/cjhutto/vaderSentiment), which we represent as a mapping from words to a positive/negative intensity score. For example, here are two ke words and their intensities:

Word	Intensity
awesome	3.1
awful	-2.0

The *polarity* of a review is one of `{'positive', 'negative', 'neutral'}`, and is calculated by finding the average intensity of the lexicon words used in the review. Words that don't appear in the VADER Lexicon are ignored when calculating the average.

Polarity	Average intensity
`positive`	≥ 0.05
`neutral`	-0.05 to 0.05, exclusive
`negative`	≤ -0.05

In `a3_part3.py`, Professor Mario has written a small program to read review data from a csv file, extract the lexicon words from the review, and then compute review's the average intensity and polarit Professor Mario has painstakingly gone through three reviews and manually calculated their polarity and average intensity. Unfortunately, when Professor Mario compares his calculated results to the return values from his program, he finds that his program has some errors!

Answer the following questions about this program and `pytest` report. Write your responses in `a3.te` except for 2(b).

1. Run the program to generate a `pytest` report.

   Based on the report, state which tests, if any, passed, and which tests failed. You can just state the name of the tests and whether they passed/failed, and do not need to give explanations here.

2. For *each* failing test from the `pytest` report:

   a. Explain what error is causing the test to fail.

Hint: each test refers to a data file under `data/reviews`; the last column of each csv file stores the test of the review.

b. Edit `a3_part3.py` by fixing the function code so that all tests pass. Unlike Assignment 1, the tests do *not* contain errors.

   The changes should be small and must be to fix errors only; the original purpose of all functions and tests must remain the same. The expected value in the tests is correct, do not change it.

3. For *each* test that passed on the original code (before your changes in question 2), explain why th test passed even though there were errors in the Python file.

Part 4: Modeling an Epidemic

In this section, we're going to learn about the mathematics behind modeling how a disease is spread during an epidemic. As we've learned all too well over the past several months, mathematical models o epidemics are complex and subtle objects, sensitive to many different parameters and unpredictable real-world phenomena and human behaviours. No mathematical model can perfectly predict how a disease might spread across a population.

So we have a fairly humble goal for this section: understanding simple types of disease models, implementing and visualizing these models in a computer program, and comparing these models against "real-world" data.

The SIR model

The **SIR model** is the simplest model used to simulate disease spread across a population. In this model, a population is partitioned into three groups:

- *Susceptible* people are ones who do not have the disease, but have the potential to be infected.
- *Infectious* people are ones who currently have the disease.
- *Recovered* people are ones who previously have had the disease and have recovered, *and are no longer susceptible*.

The goal of this model is to predict how many people are in each of these three groups as time passes during the epidemic. Mathematically, we define a constant N to represent the total population size, an three functions $S(t)$, $I(t)$, and $R(t)$ to represent the size of each of these groups at time t, measured in days.

The *initial state* of the model is represented by the numbers $S(0)$, $I(0)$, and $R(0)$, where typically $R(0) = 0$, $I(0) > 0$, and $S(0) = N - I(0)$, representing the fact that some people have been infected and no one has recovered yet.

The basic SIR model makes a few simplifying assumptions:

1. The total population number stays constant over time. For all times t, we have $S(t) + I(t) + R(t) = N$. (This ignores births and deaths, including disease-related fatalities.)
2. People who recover from the disease are no longer susceptible from the disease. (This is fairly accurate for some diseases, like measles, but not accurate for others.)
3. The likelihood of a susceptible person to contract the disease when they come into contact with a infected person is the same for every person (ignoring factors like age and underlying health conditions). Similarly, the likelihood of an infected person recovering is the same for every perso

The SIR model uses two **model parameters**, real numbers β and γ, which govern how the populatio numbers change between S, I, and R over time.

- γ represents the rate of recovery of infected people as a number between 0 and 1. If at day t there are $I(t)$ infected people, then at day $t + 1$ the model assumes that $\gamma \cdot I(t)$ people have recovered.

  For example, if $\gamma = 0.05$, this means that 5% of the infected people recover each day.

- β represents the average number of contacts that *one* infected person has per day that spread the disease.

  At day t there are $I(t)$ infected people, and $\beta \cdot I(t)$ contacts on that day that can spread the disease. However, because only susceptible people can contract the disease, we multiply this quantity by the fraction of susceptible people in the population ($\frac{S(t)}{N}$) to obtain the total number new infections: $\beta \cdot I(t) \cdot \frac{S(t)}{N}$.

  For example, if $\beta = 1.5$ and $S(t) \approx N$, then on average each infected person spreads the disease 1.5 people per day. On the other hand, if $\beta = 1.5$ and $S(t) \approx N/2$ (i.e., only half of the populatio is susceptible), then on average each infected person spreads the disease to 0.75 people per day.

Together, β and γ govern how functions S, I, and R change over time:

$$S(t+1) = S(t) - \min\left\{\beta \cdot I(t) \cdot \frac{S(t)}{N}, S(t)\right\}$$
$$I(t+1) = I(t) + \min\left\{\beta \cdot I(t) \cdot \frac{S(t)}{N}, S(t)\right\} - \gamma \cdot I(t)$$
$$R(t+1) = R(t) + \gamma \cdot I(t)$$

Note that these equations specify how S, I, and R change over time, but do not give exact formulas for these quantities. However, this is enough information to calculate these values over time using—you guessed it!—loops.

1. Your first task in this part is to implement and visualize the basic SIR model.

    a. In `a3_part4.py`, implement the function `sir_model`, which takes a population number `n`, a initial infected number (corresponding to `I(0)`), values for `beta` and `gamma`, and a number o days `num_days`. This function returns a dictionary with three keys `'S'`, `'I'`, and `'R'`, whose corresponding values are a list of integers of the size of the group over the days from 0 to `num_days - 1`.

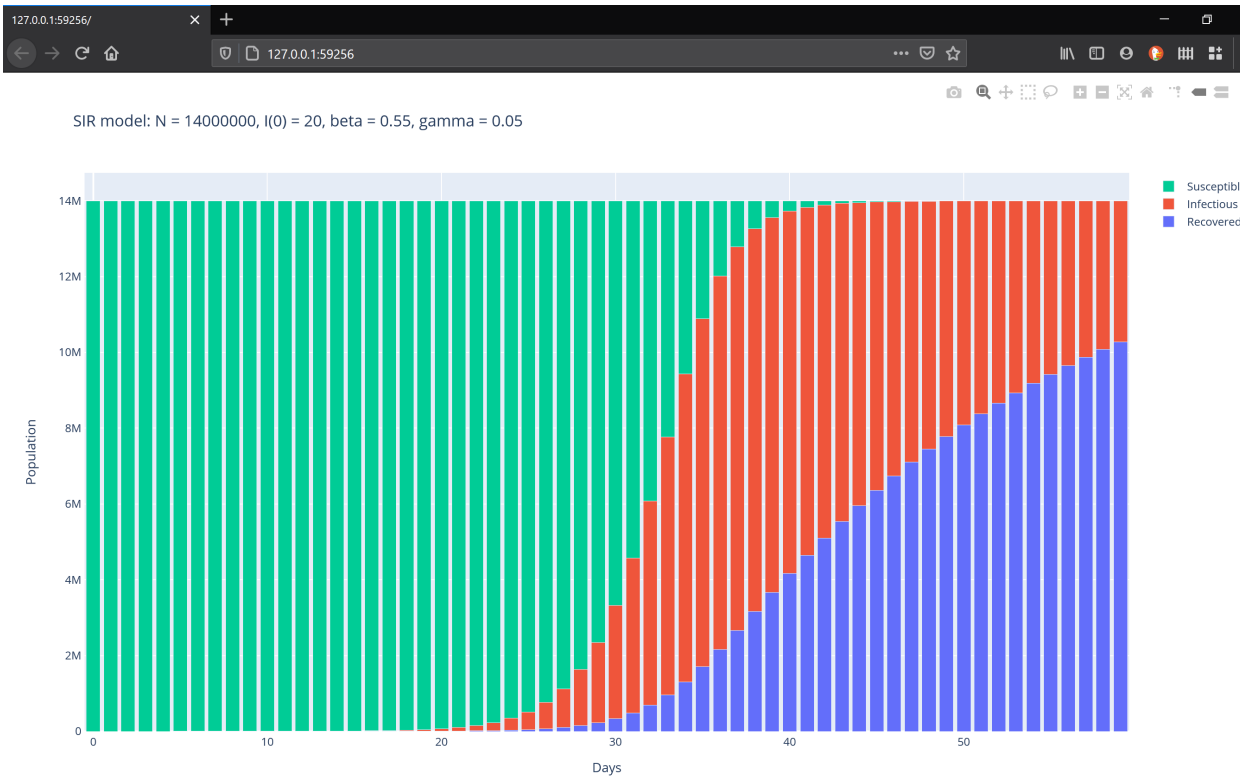
    Note: because of rounding error, only use the above equations to calculate S and I. To calculate R, first round S and I down by calling `int()` on them, and then subtract these values from the total population size N.

    Your implementation must use a loop with a dictionary accumulator variable, and use mutating operations to modify the dictionary at east loop iteration.

    b. Next, implement `plot_sir_model` using `plotly`'s stacked bar charts (https://plotly.com/python/bar-charts/#stacked-bar-chart) to generate a chart showing thi data. Concretely, the input to your function should be the same parameters as `sir_model`, and your function should call `sir_model` and then process the dictionary to feed it into `plotly`.

    Here's what we obtain after calling this function in the Python console:

```
>>> plot_sir_model(14000000, 20, 0.55, 0.05, 60)
```



The SIRD model

Next, we'll look at one variation of the SIR model that adds a fourth group of people: those who die as a result of contracting the disease. We do this by adding a fourth function, $D(t)$, as well as third model parameter μ representing the fatality rate (between 0 and 1) for the disease. We'll assume $D(0) = 0$ in our model. We also use $N(t)$ to represent the total living population at time t (this is no longer constant in our model). We update the equations that govern the change of these four categories over time:

$$S(t+1) = S(t) - \min\left\{\beta \cdot I(t) \cdot \frac{S(t)}{N}, S(t)\right\}$$
$$I(t+1) = I(t) + \min\left\{\beta \cdot I(t) \cdot \frac{S(t)}{N}, S(t)\right\} - \gamma \cdot I(t) - \mu \cdot I(t)$$
$$R(t+1) = R(t) + \gamma \cdot I(t)$$
$$D(t+1) = D(t) + \mu \cdot I(t)$$

And we define $N(t) = S(t) + I(t) + R(t)$ for all times t.

2. In `a3_part4.py`:

    a. Implement the new function `sird_model` based on this new set of equations. It is analogous to the `sir_model` functions from the previous question but takes one additional parameter `m` for the mortality rate.

Note: to avoid rounding error, use the equations to compute S, I, and R, and then round these numbers down (call `int()`) and compute $D(t+1) = D(t) + N(t) - S(t+1) - I(t+1) - R(t+1)$.

b. Implement the function `plot_sird_model`, which is analogous to `plot_sir_model`, except displaying all four of S, I, R, and D.

Estimating parameters

In this final part of the assignment, we'll look at real COVID-19 data from the Government of Ontario (https://covid-19.ontario.ca/data). You can find this data in the csv file `data/modeling/ontario_covid_cases_2020_10_17.csv` (we have lightly simplified the original dataset found on the website). Here are the first few rows of the csv file:

```
Reported Date,Resolved,Deaths,Total Cases
3/2/2020,3,0,18
3/3/2020,3,0,20
3/4/2020,3,0,20
3/5/2020,4,0,22
```

After the header row, each row of data contains the number of cases resolved (recovered), the number deaths, and the total number of cases recorded, which includes both of the previous numbers. All of these numbers are cumulative, meaning they show the totals up to that date, not the daily rate of change. These columns correspond to the categories in our SIRD model as follows:

- **R**: equal to the number of resolved cases
- **D**: equal to the number of deaths
- **I**: equal to the total number of cases minus the number of resolved cases and deaths
- **S**: the rest of the popoulation

3. a. Your task first here is to complete the function `read_csv_data`, which converts the data found in the given csv file (or any other file that has the same format) into the same SIRD dictionary format returned by `sird_model` from the previous question.

   b. Then, implement the function `plot_csv_data`, which is analogous to `plot_sird_model`, except it calls `read_csv_data` to generate the data to plot instead. Also, this function does *n* show the number of susceptible people, but instead only the "IRD" parts, since they're much

much smaller than the total number of susceptible people in Ontario (thankfully this still holds true!).

And finally, your last task for this assignment is to try to estimate the SIRD model parameters from the real Ontario data that you' processed in the previous question. However, coming up with a single estimate for each parameter doesn't make a lot of sense in this context: we know that the transmission rate has fluctuated wildly as Ontario has shutdown, reopened in stages, shutdown, etc. Similarly, we know that the death rate is greatly affected by who was infected, and to our shame one of the earliest classes of outbreaks in the spring occurred in long-term care facilities across the province, housing son of our most vulnerable citizens.

So instead of computing a single number, you'll compute a *daily estimate* of each of the three parameters β, γ, and μ. First, recall the formulas from the SIRD model, now simplified by assuming th $\beta \cdot I(t)/N(t)$ is always less than 1:

$$\begin{aligned} S(t+1) &= S(t) - \beta \cdot I(t) \cdot \frac{S(t)}{N(t)} \\ I(t+1) &= I(t) + \beta \cdot I(t) \cdot \frac{S(t)}{N(t)} - \gamma \cdot I(t) - \mu \cdot I(t) \\ R(t+1) &= R(t) + \gamma \cdot I(t) \\ D(t+1) &= D(t) + \mu \cdot I(t) \end{aligned}$$

We can rearrange the first, third, and fourth equations to write β, γ, and μ in terms of the group sizes c two consecutive days. For example:

$$\mu = \frac{D(t+1) - D(t)}{I(t)}$$

4. a. (*Not to be handed in*) Isolate β in the equation for $S(t+1)$ and γ in the equation for $R(t+1)$.
   b. In `a3_part4.py`, implement the function `estimate_parameters`, which takes a SIRD data dictionary and returns a new dictionary containing daily estimates for each of the three mod parameters.
   c. (*Not to be handed in*) Finally, try calling the function `plot_estimates` that we've provided for you. You should see a pretty cool visualization of your parameter estimates from the previous part!

Submission instructions

Please **proofread**, **test**, and **fix PythonTA errors** in your work before your final submission!

1. Login to MarkUs (https://markus.teach.cs.toronto.edu/csc110-2020-09).
2. Go to Assignment 3, then the "Submissions" tab.
3. Submit the following files: `a3.tex`, `a3.pdf` (which must be generated from your `a3.tex` file), `a3_part1.py`, `a3_part2.py`, `a2_part3.py`, and `a2_part4.py`. Please note that MarkUs is picky with filenames, and so your filenames must match these exactly, including using lowercase letters

   Note: for your Python code files, please make sure they run (in the Python console) before submitting them! Code submissions that do not run will receive a grade of *zero* for that part, whic we do not want to happen to any of you, so please check your work carefully.
4. Refresh the page, and then *download each file* to make sure you submitted the right version.

Remember, you can submit your files multiple times before the due date. So you can aim to submit you work early, and if you find an error or a place to improve before the due date, you can still make your changes and resubmit your work.

After you've submitted your work, please give yourself a well-deserved pat on the back and go take a re or do something fun or eat some chocolate!