Before you begin

- Download `msd.py` and `counter.py` from canvas and put them in your project
- Install the Python packages scipy and matplotlib if you do not already have them.
  - If you are using a Mac, you might have to put the following lines at the top of your `lab4.py` file to make things work properly (depending on how you installed Python)
    ```
    import matplotlib as mpl
    mpl.use('TkAgg')
    ```
  - You might have to install some extra Python packages for this lab (matplotlib and its dependencies). If you do not know how to do this, ask the TA in the lab session for help.
- Check out the plotting folder in github for examples
- Work through the pyplot tutorial

Learning Objectives:

You should be able to do the following:

- Plot and graph data using matplotlib
- Analyze your code for time complexity – O(n), etc

Thoughts (come back and read these after you've read the problems):

- The `random` module in Python provides many utilities for generating numbers. Be sure to read and verify the behavior of the functions you're using, e.g. some integer generation functions may exclude the endpoint, while others will include them.
- To plot the displacement of the mass, you are going to have to extract the displacements from the list of states returned by the simulation. A list comprehension might be helpful here.
- You should include the saved plot figures in your final submission (otherwise we will not be able to grade those points)
- The `time` function in the `time` module gives you a timestamp on your computer in seconds. You should think about how you can use this to get the time it takes to run a line of code on your computer. There is no need to use more complicated utilities like `timeit` for this assignment.
- Your times can be approximate, in the sense that you can have other stuff running on the computer. The goal of this part of the lab is to show you how the times scale as the number of items being counted in your list grows, not to get the fastest times or show exact timing information. Also, for very small runtimes, Windows systems may show a runtime of 0 because the Windows clock runs slower than the code runs; this is OK.

Grading Rubric: [17 points]

In regards each item, please see associated footnotes for each item *

- Problem 1, spring damper system
  - [1 points] Image of plot included in submission
  - [1 points] Plot system displacement
  - [1 points] Plot includes title and axes
- Problem 2, histogram
  - [2 points] `simulate_gachapon` produces the correct distribution of results *

*Indicates that the autograder will tell you if this problem is correct on our set of test cases and assign you credit.

  - [1 points] Image of plot included in submission
  - [1 points] Histogram is plotted with the correct shape
  - [1 points] Histogram has title and axes

- Problem 3, runtimes
  - [1 points] `random_list` works *
  - [1 points] Properly records runtime of `get_element_counts`
  - [1 points] Image of plot included in submission
  - [1 points] Produces a plot with the right characteristics
  - [1 points] Plot has a title and axes

- Standard Grading
  - [2 points] Code passes PEP8 checks with 10 errors max *
  - [2 points] Code passes TA-reviewed style checks for cleanliness, layout, readability, etc.

Problem 1: Plot spring damper system

For this assignment, create two files: `generate_plots.py` and `utils.py`. Each of the 3 problems will ask you to output plots; all of your plot-generating code should go in `generate_plots.py`, while all functions which are not directly related to plotting should go in `utils.py`.

Download the file `msd.py` and take a look at the code inside of it. Do not modify this file when doing the assignment.

The `MassSpringDamper` class simulates a mass-spring-damper system, using the ODE integrator that is part of the `scipy` scientific python package. Do not worry if you cannot completely understand what is going on right now, since some things, such as classes and lambdas, haven't been covered yet. For now, read the usage information in the docstrings on how to use the system. Then, for this problem, inside of `generate_plots.py`:

- Use the `MassSpringDamper` class to simulate a system with a mass of 1.0, a spring constant (k) of 5.0, and a damper value (c) of 2.5 with a starting position of 1 and a starting velocity of -1. It should go for 40 seconds, using 0.01s time increments.
- Plot the resulting system **displacement** (i.e. position). Ensure that the following specifications are met:
  - The x-axis time values are consistent with the time simulated (i.e. they should be between 0 and 40)
  - Your plot has a title and labelled axes.
- Save the plot with the file name `"Problem1.png"` and include it in your final submission

Problem 2: Histogram of the gachapon problem

This problem considers the following scenario:

> **Your favorite brand of cereal announces a new promotion where each box of cereal you buy contains a random prize from a set of N total prizes. How many boxes of cereal do you have to buy to get them all?**

We will write a function to simulate this, and then plot the resulting distribution. Note that the description here should be sufficient to implement the gachapon problem; you don't need anything beyond a uniform random number generator.

- In `utils.py`, write a function called `simulate_gachapon(n)` which simulates a game with n total prizes. Each iteration, randomly generate an integer from 0 to n-1 and add it to your "prize pool". Then, check if you have obtained all the prizes. If not, iterate again. Once you're done, return the number of iterations it took to obtain all the prizes.
- The random module in Python provides many utilities for random number generation. Read the first bullet in Thoughts in the beginning for some tips.
- (Sanity check: `simulate_gachapon(n)` should never return a number less than n.)

- In `generate_plots.py`, write some code which runs `simulate_gachapon(15)` 1000 times. Store down each of the results, and then plot the values in this list as a histogram. Be sure to add a meaningful title and axes. Also be sure the left edge of the plot starts at 0. (Bin size does not matter so long as the shape of the distribution is clear enough.)
- Save the plot with the file name `"Problem2.png"` and include it in your final submission

Problem 3: Algorithmic runtimes

For this problem, we are going to take a look at how different code implementations can lead to certain algorithm complexities. Download the provided file `counter.py`, which has several functions in it; the one we care about is `get_element_counts()`. You should look at the function for a bit to try to understand what it is doing.

- In `utils.py`, write a function called `random_list()` which takes in an integer `n`. It should return a random list of length n with integers uniformly sampled from 0 to n-1 (inclusive).
```
# Example:
random_list(10) # May return [0, 0, 3, 9, 5, 1, 4, 3, 1, 2]
```
- In `generate_plots.py`, write some code which does the following. For all values 50, 100, ..., up to 2500 (call each value n), it should first create a list with `random_list(n)`. It should then time how long it takes for `get_element_counts(your_list)` to run by using the `time()` function in the time module. Store the runtimes in a list.
- Create a scatter plot which plots the runtimes on the y-axis against the list lengths on the x-axis. Be sure to have a title and labelled axes.
- Save the plot with the file name `"Problem3.png"` and include it in your final submission