LAB2: DATA VISUALIZATION & PROCESSING IN PYTHON

In this lab you will practice Python data processing and visualization.

Step 1: Access the datasets

You can access the data files for `Lab_2` on Canvas in the `Files/Lab 2` section of *TECHIN 510*, and then upload them to your local Jupyter server.

- Robot Faces data set
- People
- Pre-recorded accelerometer data

And you can incorporate them into your code using the sample code in the next cell.

In [ ]:
```
## SAMPLE CODE
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

faces = pd.read_csv('robot_faces.csv')
people = pd.read_csv('people.csv')
accel = pd.read_csv('accelerometer.csv', header=None)

# faces.shape
# faces.head()
# faces.info()
# faces.describe(include=object)
```

Step 2: Robot face data exploration

Let's start by exploring the data in `robot_faces.csv`. First, write the code for loading and preprocessing the data. After inspecting the different column names to better understand what the data includes, pose a specific question and write new code to answer that question. Some example questions are:

- How many robots both have a mouth and a nose?
- Which country has the highest fraction of robots with black face color?
- Do more robots built after 2012 have blue eyes than those built before?

Your code should print the question at the beginning and print the computed answer at the end. Your script should also create at least one visualization that allows a human to answer the same question without having to do calculations.

If you are interested, you can read more about the face data here.

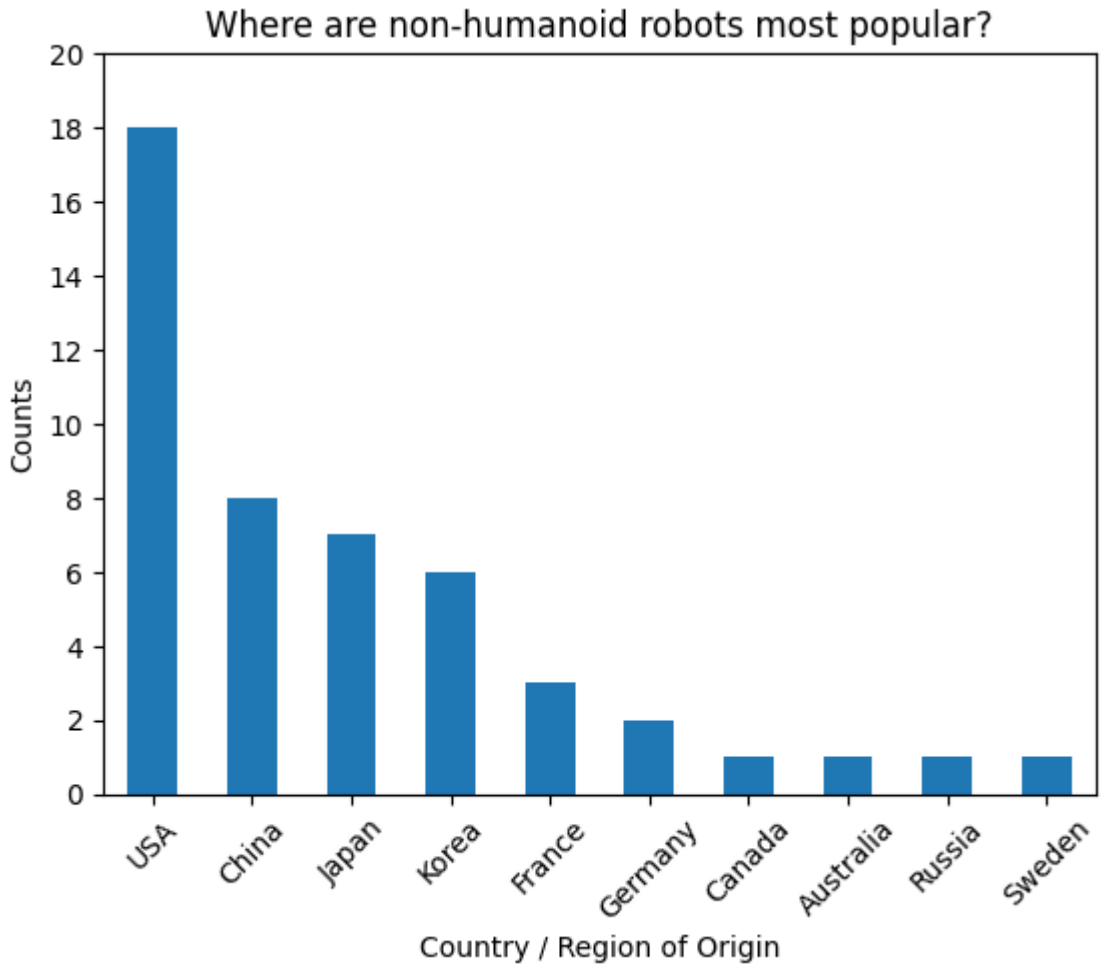
Question

- Rank the countries in terms of their preference for **non-humanoid** robots.
- ~~List **5** top categories of robots produced in and after **2015** and compare them with before.~~

- ~~Count how many types of motion can be made in total (e.g. ud, lr in eye motion count as 2).~~

In [ ]:
```
faces.replace(to_replace={'none': np.nan, 'no data': np.nan}, inplace=True)  # replac

non_hm = faces[faces['robot type'] != 'humanoid']  # filters out humanoid robots
count_non_hm = non_hm['country/region of origin'].value_counts()  # counts robots bas
plot_non_hm = count_non_hm.plot(kind='bar')  # plots bar chart of non-humanoid robot

plt.title('Where are non-humanoid robots most popular?')
plt.xlabel('Country / Region of Origin')
plt.ylabel('Counts')
plt.xticks(rotation=45)
plt.yticks(range(0, 21, 2))

print(
    f"What are the countries that made the most robots of a non-human shape?"
    f"\nThe top 4 countries leading the race are {', '.join(count_non_hm[:4].index.to
)
```

```
What are the countries that made the most robots of a non-human shape?
The top 4 countries leading the race are USA, China, Japan, Korea.
```



Step 3: Persons data exploration

We continue our exploration of the data by diving into a dataset of notional (100% fake) persons. Just as you did with the Robot data, write the code for loading and preprocessing the Person data. After inspecting the different column names to better understand what the data includes, pose a specific question and write new code to answer that question. Some example questions are:

- How many males are in the data set vs females?
- Who are the oldest people living in Chicago?
- Which city is most popular with people in their 30's?

- What are the top 5 US states represented in the data?

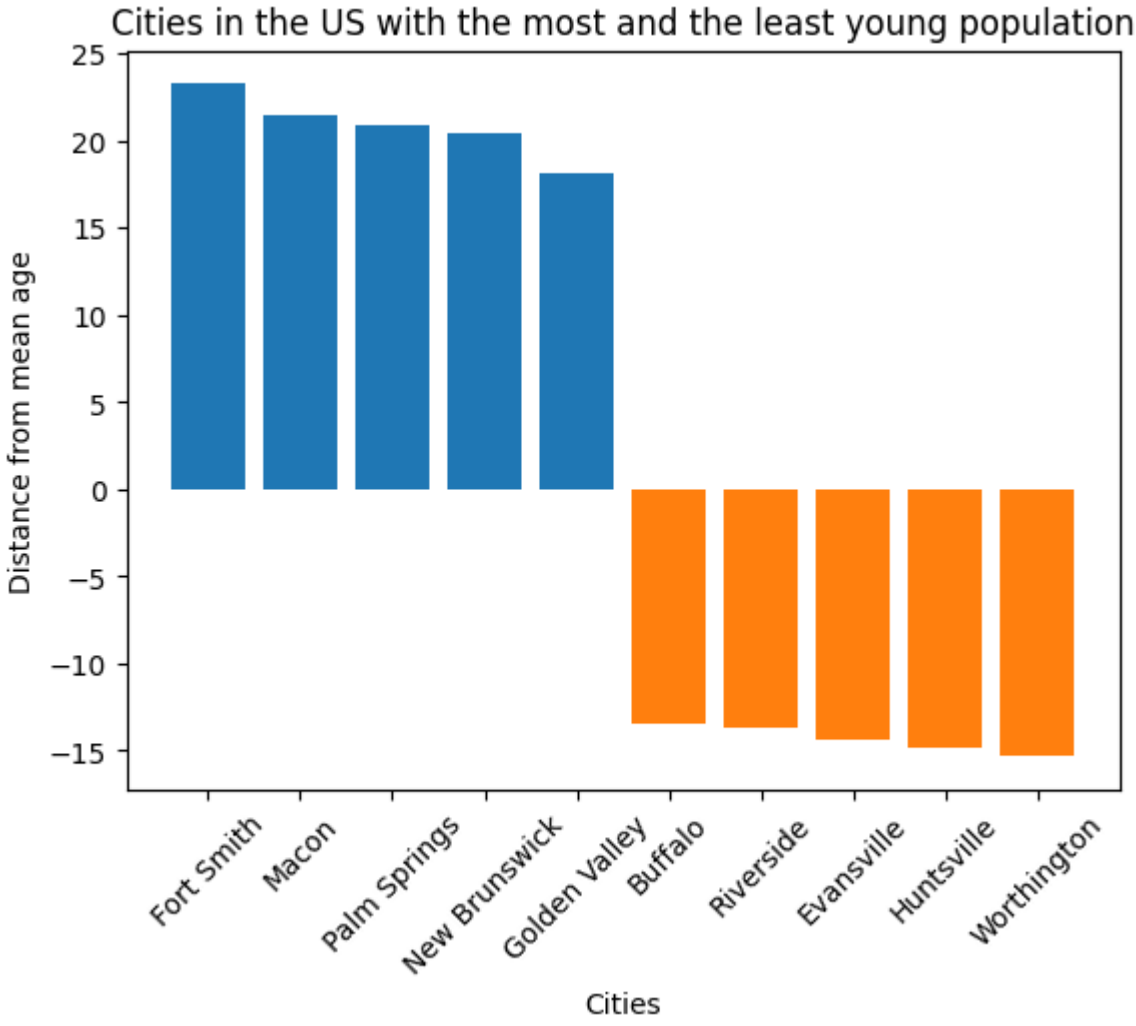
Your code should print the question at the beginning and print the computed answer at the end. Your script should also create at least one visualization that allows a human to answer the same question without having to do calculations.

Question

- List the top 5 cities with the highest mean age and the bottom 5 with the lowest.

In [ ]:
```
valid = people.groupby('City').size() > 3  # validates city by 3 records in total fro
people_val = people[people.apply(lambda row: valid[row['City']], axis=1)]  # shows re
city_mean_age = people_val.groupby('City')['Age'].mean().sort_values()  # computes an

mean_age = people['Age'].mean()  # computes mean age of the us
highest_mean = city_mean_age[-5:][::-1]  # retrieves 5 oldest cities and reverses ord
lowest_mean = city_mean_age[:5][::-1]  # retrieves 5 youngest cities and reverses ord

plt.bar(highest_mean.index, highest_mean - mean_age)
plt.bar(lowest_mean.index, lowest_mean - mean_age)
plt.title('Cities in the US with the most and the least young population')
plt.xlabel('Cities')
plt.ylabel('Distance from mean age')
plt.xticks(rotation=45)

print(
    f"What are the cities in the US with the oldest and the youngest populations?"
    f"\nThe mean age of the country from the data is {mean_age:.2f}."
    f"\nPlotted on the chart are the distances from the mean age of those of the olde
)
```

```
What are the cities in the US with the oldest and the youngest populations?
The mean age of the country from the data is 52.35.
Plotted on the chart are the distances from the mean age of those of the oldest and y
oungest cities.
```



Step 4: Load and visualize accelerometer data

Next you will explore the accelerometer data in `accelerometer.csv` recorded from a mobile device.

The first thing your Python script should do is open the data file and parse its content into Python lists or arrays. Each row in the data file corresponds to one reading. The first value is the time in seconds, and the next three values are the x, y, z acceleration values from a mobile device accelerometer. Your goal for this part of the lab is to obtain four lists or arrays (of same length) each containing the different columns in the data file.

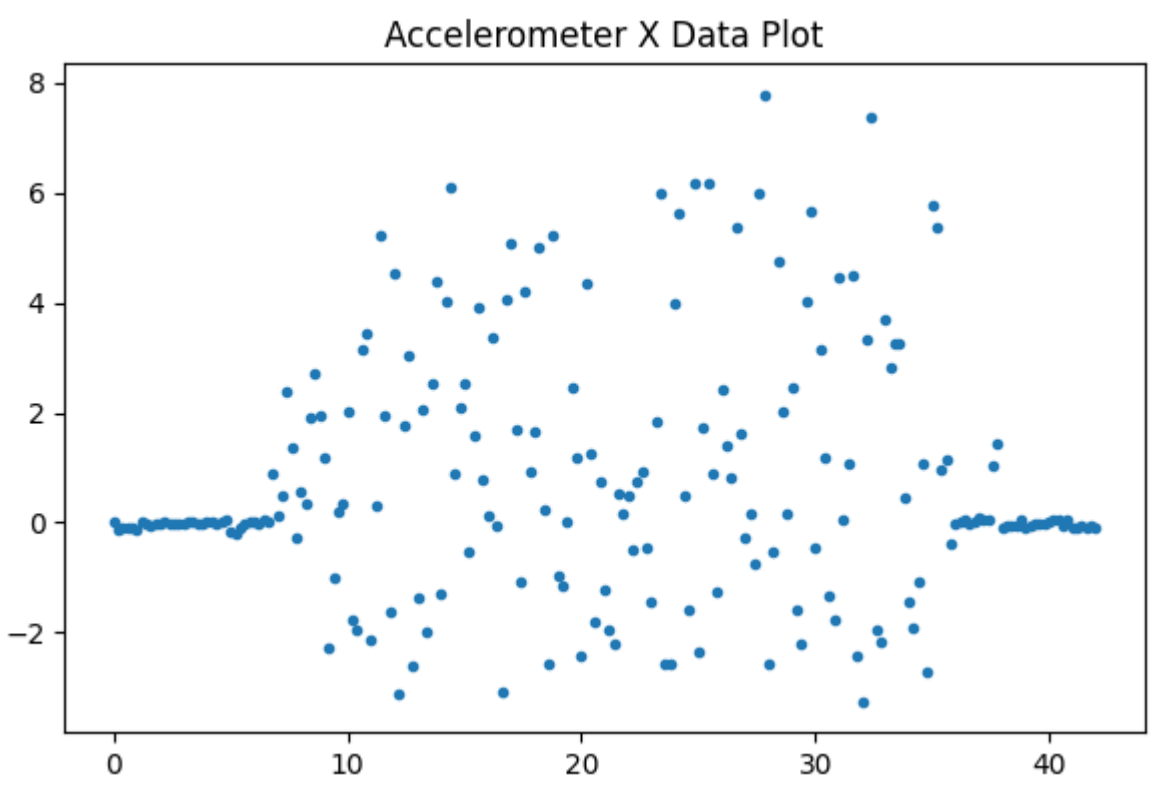
You can use the `pd.read_csv()`, but if you would like to practice some of the string operations we used during the `chatbot` exercise last week, you can open the file, read its content into a single string, and then use the split() function to split into lines (`data_string.split("\n")`) and elements (`data_string.split(",")`).

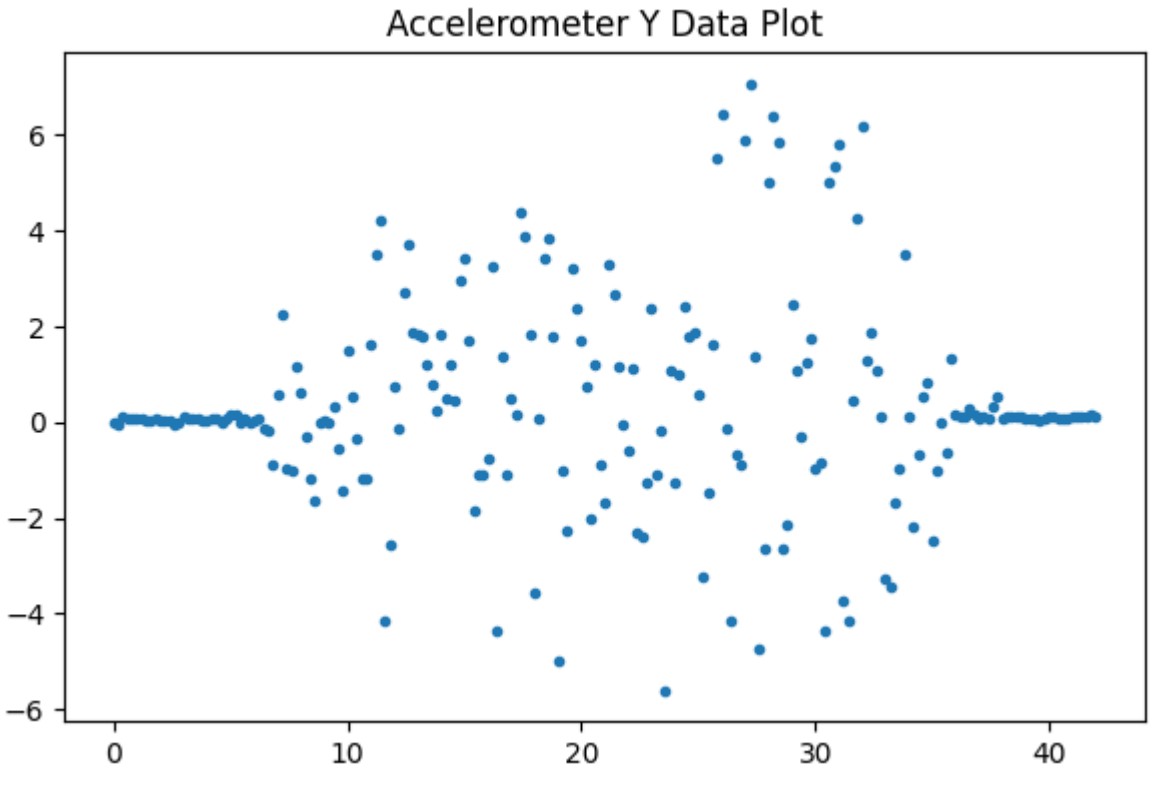
Before starting to process the data, visualize it to get a better sense of what is in the data. Keep visualization steps in your script for your lab submission.

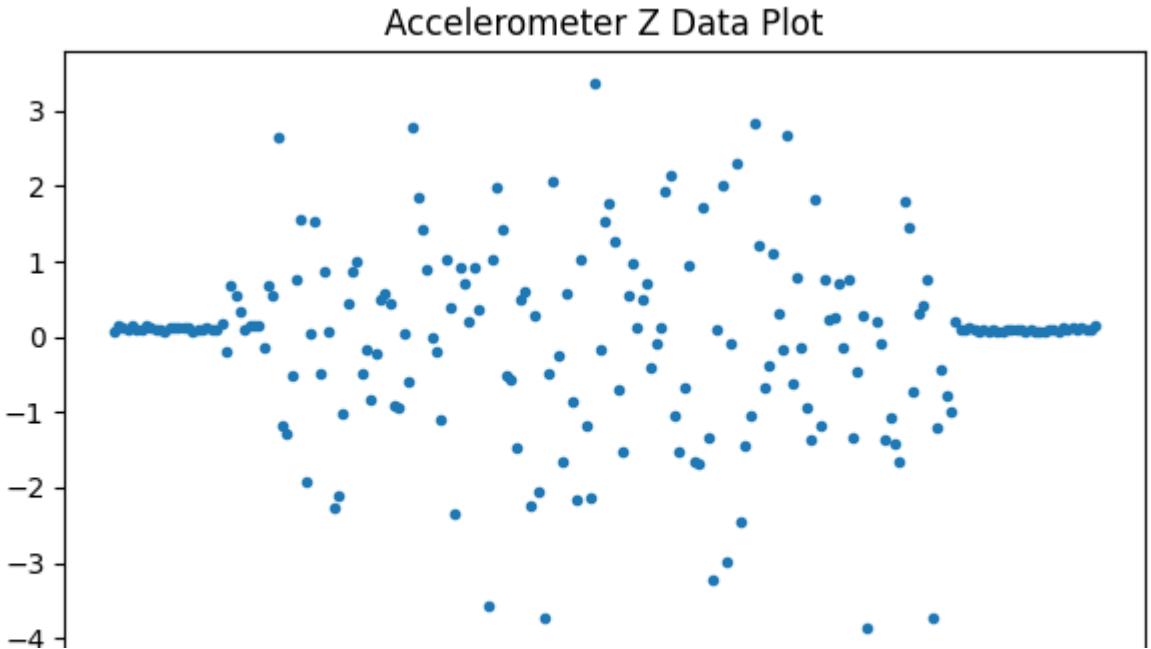
In [ ]:
```
accel.set_axis(['Time', 'X', 'Y', 'Z'], axis=1)  # sets header for data

print(
    f"Four lists or arrays requested can be obtained by indexing the column in the da
    f"\nThe visulization of the accelerometer data breaks it into three 2D plots for
)

plt.figure(figsize=(6, 12))
for data, order, axis in zip((accel["X"], accel["Y"], accel["Z"]), range(311, 314), '
```

```
    plt.subplot(order)
    plt.plot(accel['Time'], data, ".")  # plots a scatter chart with Time as x-axis a
    plt.title(f"Accelerometer {axis} Data Plot")
plt.tight_layout()
```

Four lists or arrays requested can be obtained by indexing the column in the datafram
e with its header.
The visulization of the accelerometer data breaks it into three 2D plots for easier i
nterpretation.


Accelerometer X Data Plot
8
6
4
2
0
−2
0
10
20
30
40


Accelerometer Y Data Plot
6
4
2
0
−2
−4
−6
0
10
20
30
40


Accelerometer Z Data Plot
3
2
1
0
−1
−2
−3
−4

Step 5: Detecting lack of movement

As a first data processing exercise, iterate over the lists or arrays you created to compute the: 1) total amount of time; and 2) percentage of time during which the person holding the mobile device was not moving (e.g., absolute acceleration smaller than ~0.2m/sec^2). Visualize parts of the data where lack of movement is detected together with the original data to verify that it works correctly.

In [ ]:
```
## TODO: Code for detecting lack of movement (will be run after running code from Ste
```

Step 6: Counting steps

Next you will iterate over the data to count how many steps were taken. There are alternative ways to do that but here we will outline an approximation of the "zero-crossing" method. Walking with an accelerometer results in cyclic patterns characterized by pairs of peaks and valleys in acceleration in some directions. In the provided data you can focus on the z dimension, since the mobile device was held in fixed orientation. To determine the peaks and valleys, iterate over the z values; compare each element in the list to the value of the previous and the next element (note that you cannot do this for the first and last elements of the list); if it is greater than or smaller than both of those, it corresponds to a peak or a valley. Create a separate list, of same length as the data lists, that has the value +1 where peaks occur, -1 where valleys occur, and 0 otherwise. Visualize the peaks and the valleys on the same plot as your data to verify that your algorithm works correctly.

You will see that even very small variations in acceleration cause peaks and valleys, so we need to be stricter in detecting peaks and valleys that correspond to an actual step. For that, you can extend the condition for peaks and valleys to include a threshold on the absolute value (`math.fabs()`) of the acceleration in the z direction (e.g., ~1m/sec^2). Visually inspect the number of peaks and valleys with this stricter criteria.

A rough approximation of the number of steps would be the `math.min()` of the numbers of peaks and number of valleys. If you are out of time for this lab, you can stop here. However, to count pairs of peaks and valleys more strictly, you need to iterate over the list of peaks and valleys to determine the number of times a peak is followed by a valley (or vice versa) within one second or so (i.e., -1 and 1 separated by no more than four 0s in the peak/valley list). So if you have time, implement counting of pairs as a more accurate approximation of the number of steps.

Your script should print the counted number of steps on the terminal at the end.

In [ ]:
```
## TODO: Code for counting steps in accelerometer data (will be run after Step 4)
```

Bonus

If you have extra time on this lab, go over your code and add error checks for things that might go wrong, such as if the data file does not have the expected format or if an operation returns empty lists. Instead of throwing errors in these situations, your script should print informative error messages (remember `try` / `except` clauses). In addition, you can go over your code to refactor it into potentially re-usable functions. Finally, you can use the ipywidgets we covered in class to make

your figures interactive. Do this part on the code above and list all the improments you made in the text here.

Step 7: Submit your code on Canvas

Complete this lab by submitting this file (`lab2.ipynb`) on Canvas, by January 24, 23:59. We will test your code by manually running them and inspecting the code to verify that:

- Your face data analysis prints out a clearly stated question and the answer to the question onto the terminal. The code for computing the answer correctly represents the intended question. It also creates an interpretable visualization of parts of the data that would allow a person to answer the same question.
- Your step counter code visualizes the accelerometer data, clearly showing times where there is no movement and where peaks and valleys are detected by your algorithm. It prints the detected number of steps on the terminal at the end. The detected number is reasonable compared to ground truth and the code corresponds to the described algorithm.

See Canvas for a grading rubric.