

Problem 1.

```
import numpy as np
import matplotlib.pyplot as plt

def plot_sin(a):
    x = np.linspace(0, 2 * np.pi, 100)
    y = np.sin(a * x)
    plt.plot(x, y)
    plt.show()

if __name__ == "__main__":
    plot_sin(2)
```

Run this code in Pyzo. You may need to install matplotlib and scipy on ECF. In the terminal, enter the following:

```
python3 -m pip install --user matplotlib scipy numpy
```

Problem 2.

Here is code that generates a y using a “secret” value of a , and then tries to find a by minimizing the squared error between y and $\sin(ax)$.

```
import numpy as np
import matplotlib.pyplot as plt

def generate_data_noisy(a, num_points):
    x = np.linspace(0, 2 * np.pi, num_points)
    y = np.sin(a * x) + 0.1 * np.random.randn(num_points)
    return y

def generate_data(a, num_points):
    x = np.linspace(0, 2 * np.pi, num_points)
    y = np.sin(a * x)
    return y

def sum_squared_differences(y1, y2):
    s = 0
    for i in range(len(y1)):
        s = s + (y1[i] - y2[i]) ** 2
    return s

if __name__ == "__main__":
    secret_a = 2.5
    y = generate_data(secret_a)
    # Your code here: try different a's and find the one that makes
    # sum_squared_differences(y, generate_data(a, len(y))) small.

    # Your code here: plot y together with the best fit sin(ax).
```

Problem 3.

As promised, here is a problem touching on Prof. Mark Braverman's research, celebrating the event here: <http://www.fields.utoronto.ca/activities/25-26/markbraverman>

Professor Evil teaches a class of 12 students. Each student is given a black or a white hat. Each student can see the colors of the hats of all other students, but not their own hat color. The students are not allowed to communicate with each other in any way after they receive their hats. The professor then asks each student to simultaneously guess the color of their own hat. If all 12 students guess correctly, they all get an A+ in the class. Otherwise, they all get an F.

At first, this seems hopeless: everyone has a 50% chance of guessing correctly, so the probability that everyone guesses correctly is only $\frac{1}{2^{12}} = \frac{1}{4096}$. However, before the hats are placed on their heads, the students are allowed to get together and agree on a strategy. Can they do anything?

YES!

Here is an idea: the number of white hats must be either even or odd. The students agree to guess that the number is even.

With just that guess, every student can figure out the colour of their hat: if they see an odd number of white hats, they must be wearing a white hat themselves (to make the total number of white hats even). If they see an even number of white hats, they must be wearing a black hat.

If Professor Evil decides to give out an odd or even number of white hats at random, the students will get the answer right 50% of the time!

This is a simple example of a theme in Braverman's research: you can figure out clever mechanisms to get around having to communicate information you don't actually need to communicate.

This seems like magic, so let's implement it to make sure it works!

```
def generate_hats(num_students):
    import random
    hats = []
    for i in range(num_students):
        if random.random() < 0.5:
            hats.append('W')
        else:
            hats.append('B')
    return hats

def count_white_hats(hats):
    '''Return the number of white hats in the list hats.'''
    # Your code here

def make_guesses(hats):
    '''Return a list of guesses, one for each hat in hats.
    The guesses should be 'W' or 'B'.'''
    # Your code here
    # For the student hats[i], make a guess based on the number
    # of white hats that they see
    # You can use count_white_hats() to help you figure out the
    # number of white hats that a student sees

    guesses = []
```

```

# Your code here

def are_all_guesses_correct(hats, guesses):
    '''Return True if all guesses are correct.'''
    # Your code here

if __name__ == "__main__":
    num_students = 12
    hats = generate_hats(num_students)
    print("Hats: ", hats)
    guesses = make_guesses(hats)
    print("Guesses:", guesses)
    if are_all_guesses_correct(hats, guesses):
        print("All guesses correct!")
    else:
        print("Not all guesses correct.")

```

Problem 4. Greatest common divisor using exhaustive search

In this question, you will write a function with the signature `gcd(n, m)` which computes the greatest common divisor of the positive integers `n` and `m`. The greatest common divisor of `n` and `m` is a number `d` such that `n` is divisible by `d` and `m` is divisible by `d`.

Note that `k` is divisible by `d` if and only if the remainder of the division of `k` by `d` is 0, *i.e.*, `k % d == 0`.

There is an efficient algorithm for finding the greatest common divisor (more on this later), but in this question, you should use exhaustive search: trying every possible answer until you find the right one. This is a similar approach to the one we used for `is_perfect_square` in the Monday lecture <https://www.youtube.com/live/S-4raWYGyq8?si=jFqSDAME5tdA1nNQ&t=2523>

Implement both of the approaches below.

Part (a) Approach 1

Write the function by trying every possible divisor from 1, 2, 3, ..., *etc* (What is the largest guess you can try?). Keep track of the largest guess for the greatest common divisor that worked so far. Update a variable every time the divisor divides both `n` and `m`. Return the latest guess that worked.

```

def gcd(n, m):
    """Return the greatest common divisor of n and m.
    Use exhaustive search, trying divisors from 1 up to the maximum possible."""
    # Your code here

```

Part (b) Approach 2

Approach 1 is inefficient in that we always have to try all the possible guesses every time. If we tried the largest guess first and it worked, we would not need to try smaller guesses (explain why). Use a `while`-loop to try all the possible guesses, from the largest to the smallest, and use an early return technique to return from the function once you know the answer.

Hint: in the examples so far, we used something like this to count from 1 to `n`:

```
i = 1
while i < n:
    print(i)
    i = i + 1
```

Now, we want to count backward from a larger number to a smaller number. That means we want to change `i = 1` and `i = i + 1` to something else.

Note: it is also possible, and arguably better, to use a `for`-loop here (using material not covered until Week 4), but for this question, please use a `while`-loop.

```
def gcd_efficient(n, m):
    """Return the greatest common divisor of n and m.
    Use exhaustive search, trying divisors from the largest down to 1.
    Return immediately when the GCD is found."""
    # Your code here
```