Question 1:

Part 1: Write a Python class named `Equation` which will represent a linear equation `y = ax + b`. The class should include the following:

- Constructor `__init__(self, a, b)` which will set the coefficients of the equation.
- Method `compute(self, x)` which will compute the value of y for a given x.

Part 2: Inherit from the `Equation` class and create a `QuadraticEquation` class to represent a quadratic equation `y = ax^2 + bx + c`.

- The constructor `__init__(self, a, b, c)` should set the coefficients of the equation.
- Override the `compute(self, x)` method to compute the value of y for a given x.

Part 3: Write an asynchronous function named `find_root_binary_search(equation, left, right, precision)`. This function should find the root of the given `equation` object using binary search. It should start by checking the midpoint between `left` and `right`, and update the boundaries according to the sign of y.

- `equation` is an object of either `Equation` or `QuadraticEquation` class.
- `left` and `right` are the initial boundaries of the root search. You can assume that the root is within these boundaries.
- `precision` is the precision of the found root. The function should stop when the width of the search interval is less than `precision`.

Use `await asyncio.sleep(0.01)` to simulate a delay in the calculation process.

Part 4: Now, write a Python script that creates a list of 3 equations (some of them linear, some quadratic) and finds their roots asynchronously. You should create at least three equations and print the roots of each one asyncronously.

Here are the equations:

- `y = 2x - 3` (root is 1.5)
- `y = x^2 - 4` (roots are -2 and 2)
- `y = 3x^2 - 2x - 1` (roots are -0.333 and 1)
- `precision = 0.001`

For simplicity, you can find only one root for each equation. For the quadratic ones, select the initial boundaries in a way that you find different roots.

Hints:

- To find the root of an equation means to find the x for which `y = 0`.
- In binary search, if y at the midpoint is positive, it means the root is on the left side. Otherwise, it's on the right side. Therefore, you should update the `left` or `right` boundary accordingly.
- To make asynchronous tasks in Python, you can use the `asyncio` module.
- To run asynchronous tasks concurrently, you can use `asyncio.gather()`.
- Use `asyncio.run(main())` to run the main function, where `main` is the function that starts all your asynchronous tasks.

Question 2:

Part 1: Define a decorator named `timing` that measures the time it takes to run a function. The decorator should print the elapsed time in seconds.

Part2: Create a decorator named `memoize` that caches the results of function calls, so that if the function is called again with the same arguments, it returns the cached result instead of recomputing the result. Assume that the functions this decorator is applied to only take positional arguments and no keyword arguments.

Part 3: Write a decorator named `enforce_types` that checks if the function is called with arguments of correct types. The decorator should take as argument a list of types, and it should raise a `TypeError` if the function is called with arguments of other types. Assume that the decorated function takes only positional arguments and no keyword arguments. For example, `@enforce_types([int])` is used when function should receive only 1 `int` as input argument.

Part 4: Write a function named `compute_series` that computes the sum of the series `1/k^2` for k from 1 to `n`, where `n` is a positive integer. This series converges to `pi^2/6`. Decorate this function with all three decorators from the previous questions. The decorators should be applied in the following order from top to bottom: `timing`, `memoize`, `enforce_types`.

The `enforce_types` decorator should enforce that `n` is of type `int`.

Part 5: Demonstrate the `compute_series` function in action. Call it with `n = 10^6`, then with `n = 10^6` again (should return from cache), then with `n = "10^6"` to check `enforce_types` (catch exception to continue), and finally with `n = 10^7`.

Hints:

- A decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it.
- Decorators are applied to functions using the `@decorator_name` syntax, placed directly above the function definition.
- When multiple decorators are applied to a function, they are applied from bottom to top. The decorator closest to the function is applied first, and so on.
- For the `memoize` decorator, you will need to use a dictionary to store the results of function calls. The keys of this dictionary should be all arguments with which the function is called, and the values should be the corresponding results.
- For the `enforce_types` decorator, remember that you can check the type of a variable with the `type()` function or `isinstance()` function.
- In Python, `10^6` is not one million, but the bitwise XOR of 10 and 6. Write one million as `10**6` instead.