ASSIGNMENT 2

Executive Summary:

Part I:

In this Nonogram solver, I start by reading an input file with row and column clues, which the program uses to set up a grid. The grid is initialized with all cells marked as unknown, and clues are organized into `row_clues` and `col_clues`. The core logic lies in the `solve_line` method, where dynamic programming is used to match each row or column with its clues and build a possible solution. The `propagate_constraints` method refines the solution by repeatedly applying clues to narrow down the grid. Once complete, the `display` method visually shows the solved puzzle, using `matplotlib` and `PyQt5` to draw the grid and display the clues. To handle larger puzzles like "example5," I added a timeout parameter, improved efficiency with dynamic programming, and optimized the display and file parsing. These updates ensure the solver can handle big puzzles without stalling. This resulted in a quicker execution of the code, but unfortunately this code could not solve example5 completely. I am attaching the intermediate solution I could produce in the zipped file with the name 'example_5_intermediate_nonogram'.

File name: nonogram.py
Example usage: python .\nonogram.py .\example4

References:
https://github.com/Arpanio/nonogram/blob/master/constraint_programming_solver.py
https://github.com/pythonik/Nonogram-game-solver/blob/master/source/constraint_propagation.py

Part II:

For this part of the assignment, I'm creating a Nonogram generator by converting an image into a black-and-white grid using a threshold-based method. First, I load the image and preprocess it by converting it to grayscale, simplifying it to a single intensity channel. I then enhance the contrast using histogram equalization, which makes the image clearer and improves the thresholding process. After preprocessing, I resize the image to fit the desired puzzle size, allowing flexibility in adjusting the final Nonogram grid's dimensions. Once the image is ready, I apply the thresholding method, which turns the image into a binary grid—either using the Otsu method for automatic thresholding or the mean method, where the pixel values above the average are set to white and the rest to black. This step plays a crucial role in shaping the final puzzle, determining which areas are defined and which are left blank. Next, I generate the row and column clues by counting the contiguous blocks of black pixels in each row and column, which will serve as the instructions for solving the Nonogram. Finally, I visualize the Nonogram by plotting the grid with the corresponding row and column clues and adding grid lines for clarity. Both the image and a text file containing the clues are saved, providing everything needed for the puzzle. This whole process involves turning an image into a solvable Nonogram by converting it into a binary grid, generating clues, and visualizing the result.

File name: nonogram.py
Example usage : python .\makenonogram.py '.\original image 1.jpg'

References:
https://theses.liacs.nl/pdf/2011-04SjoerdHenstra.pdf
https://github.com/OmarAlkousa/Image-Thresholding-Web-App/blob/main/README.md