ICC/Programming — Week 9, Mini-project Part A CGC/SIE, April 20 2026

The mini-project is to be done in groups of two or individually. Groups may exchange ideas or general approaches, but not code directly.

Context. In this mini-project you will work with two historical maps. The first is *One Year After* (Sunday News, May 5, 1946), a newspaper map showing post-war Europe. The second is *Antiquae Urbis Romae*, a detailed bird's-eye engraving of ancient Rome published in 1588.

The project has two independent parts. **Part 1** uses the Europe map to get familiar with basic image manipulation: converting a colour image to grayscale, and applying a blur. **Part 2** uses the Rome map for a puzzle reconstruction task: some tiles have been removed from the map and shuffled, and you must write an algorithm that figures out where each one belongs by comparing pixel values along tile edges.

Mini-project Part B will build directly on the functions you implement here.

Preparation

Open Visual Studio Code, choose Terminal → New Terminal, and install the required libraries if needed:

```
test -d venv && source venv/bin/activate
python3 -m pip install numpy
```

Download `miniproject-a-start.zip` from Moodle and extract it into your workspace. It contains:

- `miniprojectutils.py` — helper functions (do **not** modify this file).
- `test_miniproject_a.py` — run this file to check your progress (do **not** modify this file).
- `miniproject_a.py` — the file you will edit. Implement each function where you see a `TODO` comment. Do **not** change any function name or its parameters.
- `imgs/europe.jpg` — the 1946 Europe map.
- `imgs/rome.jpg` — the ancient Rome engraving.
- `tiles/` — missing tiles from the Rome map.

Run `miniproject_a.py`. It should print one line and stop without errors. If it does not, call an assistant before continuing.

`miniproject_a.py` contains 7 functions for you to implement. Each function has a docstring explaining what it should do, inline comments describing the steps, and one or more `TODO` markers showing exactly where to add your code. When you implement a `TODO`, **replace** the `raise NotImplementedError(...)` line with your code, unless the comments in code specify otherwise.

To check your progress at any point, run `test_miniproject_a.py` This runs all tests and prints a ✓ next to each function that is working correctly and a **X** next to each one that is not yet implemented or contains a bug. A summary at the end shows how many sections are passing. Run the tests frequently — after every function you implement — so that mistakes are caught early before they affect the functions that depend on them.

How images are stored. A colour image is a numpy array of shape (*height*, *width*, 3), where the last dimension holds the red, green, and blue (RGB) channel values, each an integer between 0 and 255. A grayscale image has shape (*height*, *width*): a single brightness value per pixel. All the helper functions you need to load and save images are already provided in `miniprojectutils.py`.

Part 1. Image Manipulation

A. Conversion to Grayscale

(a) **Implement `rgb_to_gray(r, g, b)`.**

This function converts a single RGB colour to a single grayscale brightness value. It takes three integers `r`, `g`, `b` (each between 0 and 255) and must return an `int` between 0 and 255.

Averaging the three channels (`(r+g+b)/3`) gives a poor result because our eye is not equally sensitive to all colours — it perceives green as roughly 3.5 times brighter than blue. Use the following perceptual formula instead:

$$gray = 0.2126 \cdot r \ + \ 0.7152 \cdot g \ + \ 0.0722 \cdot b$$

Tip: test your function on a few specific values before continuing. For example, **`rgb_to_gray(255, 0, 0)`** should return 54, **`rgb_to_gray(0, 255, 0)`** should return 182, and **`rgb_to_gray(255, 255, 255)`** should return 255.

(b) **Implement `to_grayscale(img)`.**

This function converts a full colour image to grayscale. It receives a colour image (shape $H \times W \times 3$) and must return a new grayscale image of the same height and width, where each pixel has been converted using **`rgb_to_gray`**.

Uncomment the corresponding lines in **`grayscale_test`** and run the file. It saves **`imgs/out/europe_gray.jpg`**; open it and verify it looks like the right-hand image below.

Left: the original colour Europe map. Right: the same map converted to grayscale.

Debugging tip. If your output looks wrong, test on a small 3×3 pixel image first. You can create one with **`np.array([[[255, 0, 0], [0, 255, 0], ...]])`** and check individual pixel values with **`print`**. It is much easier to spot mistakes on small arrays than on a full-size image.

B. Box Blur

Processing large images can be slow. A common solution is to reduce the image resolution before processing — but simply dropping every other pixel loses information. A better approach is to blur first (averaging neighbouring pixels) and then downsample. In this section you implement the blurring step.

Implement `simple_blur(img_gray, ksize)`.

This function applies a box blur of size `ksize` × `ksize` to a grayscale image: each output pixel is the average of the `ksize` × `ksize` square of input pixels centred on it.

You will implement it as two separate 1D passes, which is more efficient than a full 2D loop:

1. **Horizontal pass.** Convolve each row independently with a 1D kernel of length **`ksize`** using **`np.convolve(row, kernel, mode='valid')`**. This shortens each row by `ksize` − 1 pixels; the height does not change.

2. **Vertical pass.** Convolve each column of the Pass 1 result with the same kernel using **`np.convolve(col, kernel, mode='valid')`**. This shortens each column by $\texttt{ksize} - 1$ pixels.

The kernel must be a **`float64`** numpy array of length **`ksize`** with all values equal to $1/\texttt{ksize}$ (so they sum to 1). Store the Pass 1 result as **`float64`** to avoid rounding errors. Round to the nearest integer at the end of Pass 2 and store as **`int16`**.

Example: `simple_blur(img, 4)` on a 100×80 grayscale image produces a 97×77 result ($100 - 3 = 97$ rows, $80 - 3 = 77$ columns).

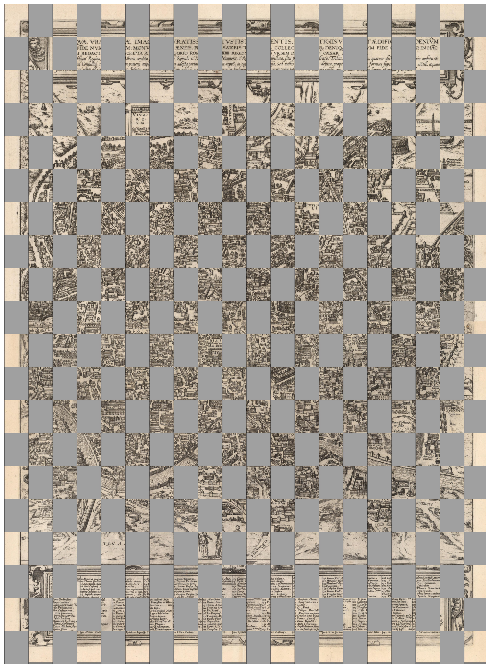
Uncomment the corresponding lines in **`blur_test`** and run the file. It saves **`imgs/out/europe_blurred.jpg`**; the result should look like the image below.

grayscale image after box blur (**`ksize=4`**)

Part 2. Map Puzzle Reconstruction

Setup

The ancient Rome map has been cut into a 20×20 grid of equal tiles. The tiles are arranged in a checkerboard pattern: tiles at even positions (where $row + col$ is even) are the **given tiles —** they are already placed and their positions are known. Tiles at odd positions ($row + col$ odd) are the **missing tiles —** they have been removed, shuffled into a random order, and saved with filenames that give no position hint.



Left: the puzzle as given (grey squares = missing tiles). Right: the correct completed reconstruction.

Key observation. Every missing tile is surrounded on all four sides by given tiles (or by the grid border). This means each missing tile can be matched independently: you never need to wait for another missing tile to be placed before you can score a candidate.

What you are given. The `tiles/` folder contains:

- `tile_given_r_c.jpg` — one file per given tile. The position (r, c) is encoded in the filename.
- `tile_missing_0.jpg`, `tile_missing_1.jpg`, ...— the missing tiles in shuffled order. The file number is **only an identifier** — it is not the correct grid position.
- `partial_grid.jpg` — a preview with given tiles placed and grey rectangles for missing slots.
- `config.json` — read automatically by `miniprojectutils.py`. You do not need to open it.

The helper functions `load_given_tiles` and `load_missing_tiles` in `miniprojectutils.py` load the tiles for you. `load_given_tiles` returns a dictionary mapping `(row, col)` to the tile image. `load_missing_tiles` returns a plain list indexed by file number.

Functions to implement

(a) **Implement `get_edge(tile, side)`.**

Extracts one border strip of a tile as a numpy array. `side` is one of `'top'`, `'bottom'`, `'left'`, `'right'`. Tiles are colour images, so each pixel has three values (R, G, B):

`'top'`	first row → shape $(width,\ 3)$
`'bottom'`	last row → shape $(width,\ 3)$
`'left'`	first column → shape $(height,\ 3)$
`'right'`	last column → shape $(height,\ 3)$

No loops — use numpy slicing only. *Hint:* `tile[0, :]` gives the first row; `tile[:, -1]` gives the last column.

Self-test: `get_edge(tile, 'top').shape` should equal `(tile.shape[1], 3)`.

(b) **Implement `edge_difference(edge1, edge2)`.**

Computes a score measuring how different two edges are. Both arrays have the same shape (same number of pixels, same 3 colour channels).

$$score \ = \ \frac{\text{mean}(\,|\,edge1 - edge2\,|\,)}{255} \ \in \ [0,\ 1]$$

A score of 0 means the two edges are pixel-perfect identical; a score of 1 means they are as different as possible. No loops — use `np.abs` and `np.mean`.

Self-test: `edge_difference(e, e)` must return exactly `0.0` for any edge `e`.

(c) **Implement `find_best_tile(given_tiles, missing_tiles, r, c, grid, placed)`.**

Given the empty slot at position (r, c), this function finds which unplaced missing tile fits it best.

Parameters:

- `given_tiles` — dict mapping `(row, col)` to tile image for all fixed tiles.
- `missing_tiles` — list of all missing tile images (by file index).
- `r, c` — grid coordinates of the empty slot.
- `grid` — total number of rows (= number of columns).
- `placed` — dict mapping file index to `(row, col)` for missing tiles that have already been placed. Do **not** use a tile that is already in `placed`.

Algorithm. For each unplaced missing tile, compute an average `edge_difference` score using all four direct neighbours of slot (r, c) that are already known:

- Left neighbour at $(r, c-1)$: compare its `'right'` edge with the candidate's `'left'` edge.
- Top neighbour at $(r-1, c)$: compare its `'bottom'` edge with the candidate's `'top'` edge.
- Right neighbour at $(r, c+1)$: compare the candidate's `'right'` edge with its `'left'` edge.
- Bottom neighbour at $(r+1, c)$: compare the candidate's `'bottom'` edge with its `'top'` edge.

A neighbour is "known" if it is a given tile or an already-placed missing tile. Skip any direction where the neighbour is not known (grid boundary, or not yet placed).

Return the **file index** of the candidate with the lowest average score.

Hint: We have already built a reverse lookup `pos_to_file = {v: k for k, v in placed.items()}` to quickly find which file index occupies a given position.

(d) **Implement `reconstruct(given_tiles, missing_tiles, grid)`.**

Calls `find_best_tile` for every empty slot and assembles the full placement.

Algorithm:

1. Build the list of empty slots: all (r, c) where $(r + c)$ is odd.
2. Initialise `placed = {}`.
3. For each empty slot, call `find_best_tile` and record the result in `placed`.

Return `placed`: a dict mapping each file index to its assigned (r, c) position.

Uncomment the lines in `reconstruct_test` and run the file. The console will print how many tiles were placed correctly and save `imgs/out/rome_reconstructed.jpg`. If all tiles are correct it prints ✓ **Correct!**