ICC/Programming — Week 11, Mini-project B

CGC/SIE, 4 May 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 Mini-project-A you implemented tile-edge comparison (to reconstruct a shuffled puzzle) and a box blur (to smooth an image before reducing its resolution). Here you apply the same pixel-comparison idea to a new problem: automatically locating a small rectangular region anywhere inside a large map, using only pixel values.

The map is a 1991 Southern California tourist cartoon map by MetroGuide. It arrives as a checkerboard puzzle — alternating tiles have been removed and shuffled, exactly as in Mini-project-A . You must first reconstruct it, then convert it to grayscale, and only then run the pattern search. **Part 1** implements a brute-force search that checks every possible position. **Part 2** makes the same search dramatically faster using a coarse-to-fine strategy, and you will measure the speedup directly.

Preparation. Download **`miniproject-b-start.zip`** from Moodle and extract it. Then copy your file **`miniproject_a.py`** into the extracted folder.

The zip contains:

- **`miniproject_b.py`** — the file you will edit. Implement each function where you see a **`TODO`** comment. Do **not** change any function name or its parameters.
- **`test_miniproject_b.py`** — run this file at any time to check your progress. Do **not** modify it.
- **`miniprojectutils.py`** — helper functions. Do **not** modify.
- **`imgs/cartoon.jpg`** — the 1991 Southern California cartoon map.
- **`tiles/`** — checkerboard puzzle tiles of the cartoon map.

Note: Make sure **`miniproject-b-start`** folder contains your complete implementation of **`miniproject_a.py`**.

`miniproject_b.py` contains 8 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:

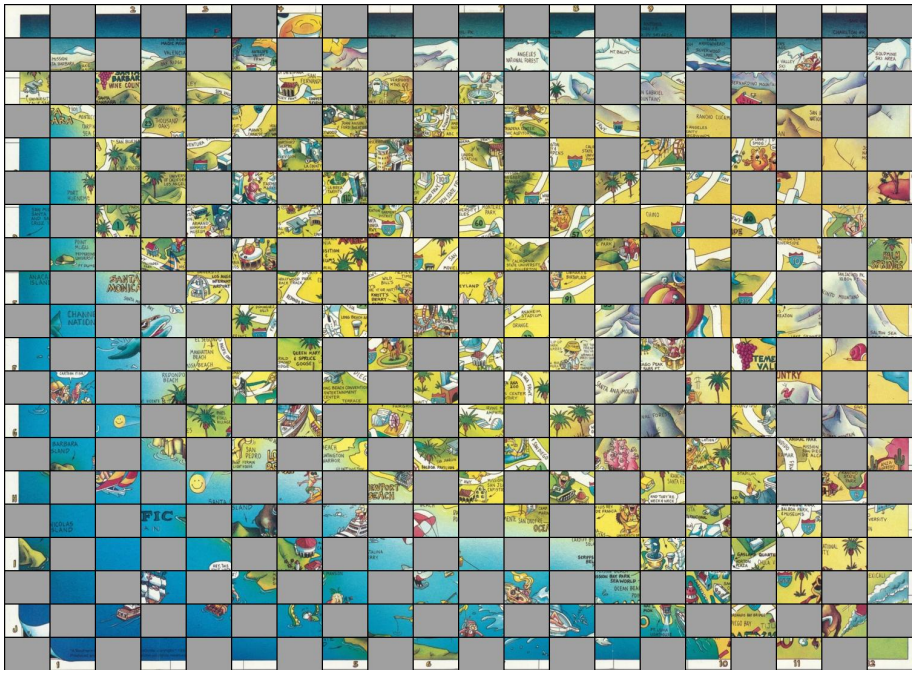
```
python3 test_miniproject_b.py
```

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 of the 9 sections are passing. Run the tests frequently — after every function you implement — so that mistakes are caught early.

Important. Mini-project-B imports functions you wrote in Mini-project-A — specifically **`to_grayscale`**, **`simple_blur`**, **`get_edge`**, **`edge_difference`**, **`find_best_tile`**, and **`reconstruct`**. Make sure to complete Mini-project-A before proceeding to Mini-project-B.

Step 0. Reconstruct and prepare the map

The cartoon map is provided as a checkerboard puzzle in the **`tiles/`** folder (same format as Mini-project-A). Before you can search for a pattern you need to reassemble the map and convert it to grayscale. You do not need to write any new code here — this step is already implemented for you.



The cartoon map as given: given tiles are in place, grey squares mark the missing tiles to be reconstructed.

Run **`test_miniproject_b.py`**. The **Step 0** tests should pass immediately — reconstruction and grayscale conversion are already implemented for you using your function implementation from Mini-project-A. The test saves:

- **`imgs/out/cartoon_map_reconstructed.jpg`** — the reassembled colour map.
- **`imgs/out/cartoon_map_gray.jpg`** — the grayscale version used for all subsequent searches.

The correctly reassembled colour map (**`cartoon_map_reconstructed.jpg`**).

The grayscale working image (**`cartoon_map_gray.jpg`**) used for all pattern searches.

Open **`imgs/out/cartoon_map_gray.jpg`** and verify it looks like the image above. If the **Step 0** reconstruction test fails, fix **`find_best_tile`** in **`miniproject_a.py`** before continuing — the pattern search might give meaningless results on a broken map.

Part 1. Brute-Force Pattern Search

The idea. Given a small rectangular region cropped from the map — the pattern — the goal is to find where it came from. We do this by sliding the pattern over every possible position in the grayscale map and computing a similarity score at each position. The position with the lowest score is the best match.

The similarity score is the same mean absolute difference formula you used for **`edge_difference`** in Miniproject-A — the only difference is that now we compare full 2D regions instead of 1D edge strips.

(a) **Implement `pattern_difference(img_gray, pattern_gray, position)`.**

Computes how similar **`pattern_gray`** is to the rectangular region of **`img_gray`** whose top-left corner is at **`position = (row, col)`**.

- If the region would extend beyond the boundary of **`img_gray`** in any direction, return **`1.0`** immediately.
- Otherwise, extract the region and compute:

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

A score of 0 means the two regions are pixel-perfect identical; 1 means they are as different as possible.

No loops allowed — use numpy operations only (**`np.abs`**, **`np.mean`**, and array slicing to extract the region). This constraint is essential: the next function calls **`pattern_difference`** once per pixel position in the image, and any Python loop inside it would make the entire search orders of magnitude slower.

Hint: extracting the region is one slice: **`img_gray[r : r+ph, c : c+pw]`**, where **`ph`** and **`pw`** are the pattern's height and width.

(b) **Implement `find_similar_pattern_position(img_gray, pattern_gray)`.**

Tries every valid top-left position and returns the one with the lowest **`pattern_difference`** score. A valid position **`(r, c)`** is one where the pattern fits entirely inside the image: $0 \le r \le H_{\text{img}} - H_{\text{pat}}$ and $0 \le c \le W_{\text{img}} - W_{\text{pat}}$. Return **`(0, 0)`** if the pattern is larger than the image in either dimension. A loop over all valid positions is expected and fine here.

(c) **Implement `highlight_rectangle(img, corner, size, value, linewidth)`.**

Returns a **new** copy of **`img`** (the input must not be modified) with a rectangular border drawn on top. The border is drawn around the rectangle — the pixels inside are unchanged.

- **`corner = (row, col)`**: top-left pixel of the rectangle interior.
- **`size = (height, width)`**: dimensions of the interior in pixels.
- **`value`**: intensity of the border pixels (0 = black, 255 = white).
- **`linewidth`**: thickness of the border in pixels.

Use **`clamp(0, limit, x)`** from **`miniprojectutils.py`** whenever you compute a row or column index that might fall outside the image.

Hint: think of the border as four rectangular bands — top, bottom, left, right — each filled with a single numpy slice assignment. For each band, compute its extents using **`clamp`**, then assign **`value`** to that slice.

(d) **Implement `highlight_similar_pattern(img_gray, pattern_gray, value, linewidth)`.**

Calls **`find_similar_pattern_position`** to find the best match, then calls **`highlight_rectangle`** to draw a border at that position. This function is two lines.

Choosing and testing your pattern

The tests are self-verifying: they crop a known region from the map and ask your algorithm to locate it. Since the crop position is known, the test can check automatically whether the answer is correct.

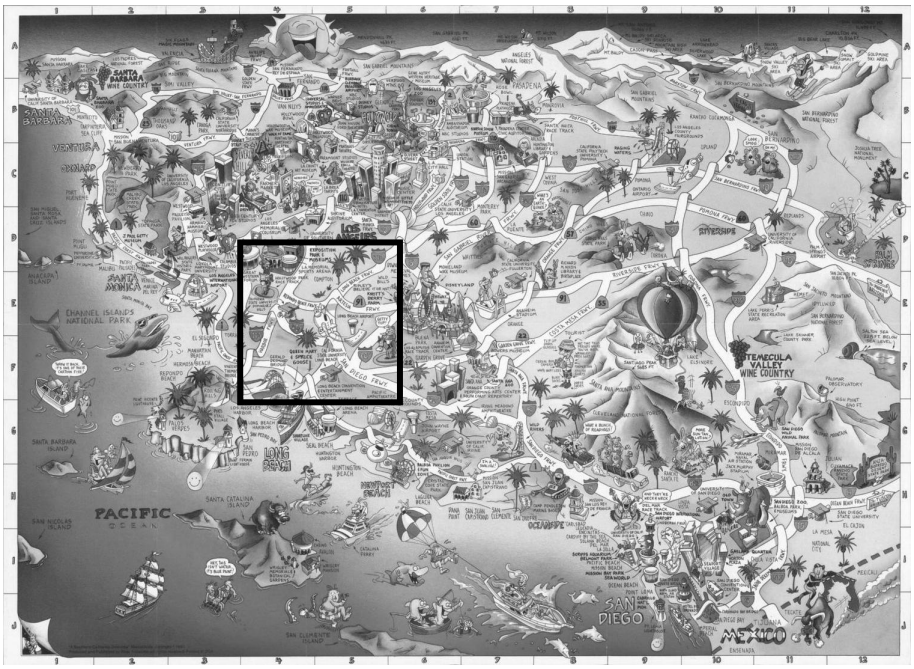
Setting your pattern.
Note: You can skip this step and use the default values of **`PATTERN_CORNER`** and **`PATTERN_SIZE`** provided in the code — they point to a distinctive region of the cartoon map that works well for testing.

Open **`imgs/out/cartoon_map_gray.jpg`** and choose a visually distinctive area — a landmark label, a coastline bend, or a distinctive building outline. Avoid large uniform areas (open sea, blank sky) where many positions look equally similar. Set **`PATTERN_CORNER`** and **`PATTERN_SIZE`** near the bottom of **`miniproject_b.py`** to the pixel coordinates of your chosen region.

Running the tests. Run **`test_miniproject_b.py`**. Once Part 1 functions are implemented, the following sections will pass:

- **1A – pattern_difference**: checks boundary conditions, self-comparison (score $= 0$), and the max-difference case (score $= 1$).
- **1B – find_similar_pattern_position**: crops your chosen region and verifies the algorithm finds it at the exact crop position.
- **1C – highlight_rectangle**: checks that the interior pixels are unchanged and the border pixels are set to the correct value. Saves **`cartoon_map_pattern_origin.jpg`** — the map with your chosen region outlined.
- **1D – highlight_similar_pattern**: verifies the two-line wrapper calls both functions correctly. Saves **`cartoon_map_found_pattern.jpg`** — the map with the best-matching region highlighted in black.

Example pattern selection: the chosen region is outlined in white. The algorithm must find this exact position from the cropped pixels alone.



The best match found by the brute-force search, highlighted in black (**`cartoon_map_found_pattern_highlighted.jpg`**). The console prints the time taken — note it for comparison with Part 2.

Note: the brute-force search checks every pixel position in the image and may take around 3-4 minutes. If it is too slow, choose a smaller ***`PATTERN_SIZE`****.*

Part 2. Accelerated Pattern Search

Why the brute-force search is slow

To understand the problem concretely, consider the numbers. Suppose the grayscale map is 800×600 pixels and the pattern is 50×50 pixels. The brute-force search checks $(800 - 50) \times (600 - 50) = 412{,}500$ positions. At each position it computes the mean of $50 \times 50 = 2{,}500$ absolute differences. That is over one billion individual operations. If the image is twice as large, the search takes four times longer.

The coarse-to-fine strategy

Instead of searching the full image at full resolution in one pass, the accelerated search works in two stages:

Stage 1 — coarse search. Shrink both the image and the pattern by a factor of **`factor`** (e.g. factor 4) to get images that are `factor`^{2} times smaller. Run the brute-force search on the small versions. Keep the **`top_n`** best candidate positions rather than just the single best, because the small image is an approximation and the true best position might not rank first at low resolution.

Stage 2 — fine refinement. For each candidate position, scale it back to the original image coordinates and search only a small neighbourhood around it. This neighbourhood has radius $\lfloor$`factor`$/2\rfloor$ pixels in each direction. The total number of calls to **`pattern_difference`** is a tiny fraction of the brute-force count.

How we shrink the image

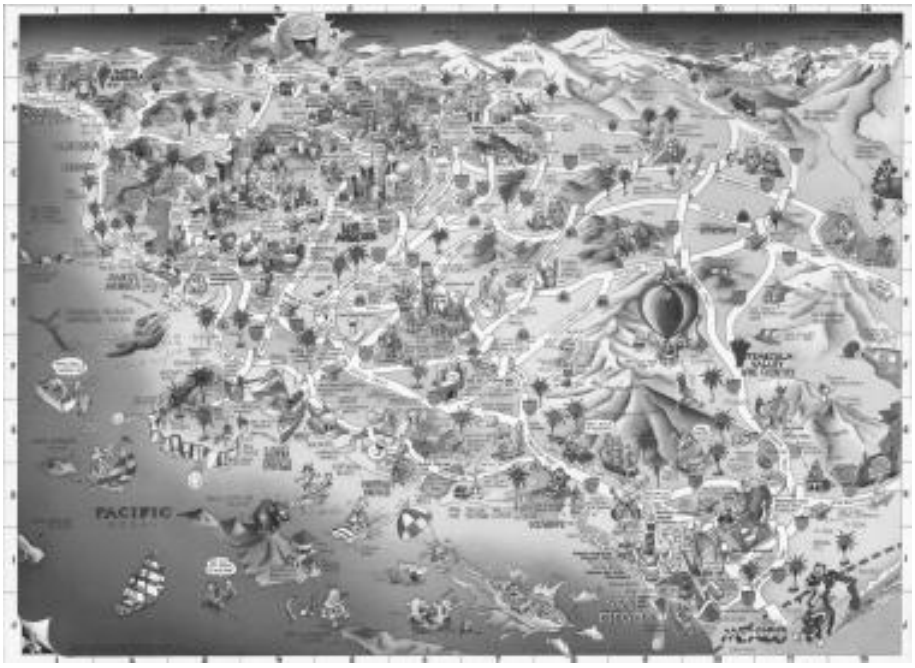
To reduce the image for Stage 1 we use two steps: blur, then downsample. Blurring first averages every pixel with its neighbours so that every original pixel contributes to the reduced image — no information is silently discarded. Downsampling then keeps only every **`factor`**-th pixel in each dimension. You implemented **`simple_blur`** in Mini-project-A ; you will reuse it here without change.

(a) **Implement `downsample(img_gray, factor)`.**

Keeps every **`factor`**-th pixel in each dimension starting from $(0, 0)$. No loops — one numpy slice. Always call **`simple_blur(ksize=factor)`** on the image before passing it to **`downsample`**.

Run **`test_miniproject_b.py`**. Once implemented, the **2A – downsample** section will pass and save **`imgs/out/cartoon_map_downsampled.jpg`** for visual inspection.

After **`simple_blur`** (ksize=4): the image is smoothed and slightly smaller due to the **`'valid'`** convolution mode.



After **`downsample`** (factor=4): the image is reduced to a fraction of its original size while preserving detail through prior blurring.

(b) **Implement `find_most_likely_positions(img_gray, pattern_gray, top_n)`.**

Same search as **`find_similar_pattern_position`** from Part 1, but returns the **`top_n`** positions with the lowest scores, sorted from best (lowest) to worst, built in a **single pass**.

Hint: maintain a list of at most **`top_n`** pairs **`(score, position)`**, sorted by score at all times. When a new score beats the current worst entry, replace it and re-sort.

(c) **Implement `faster_search(img_gray, pattern_gray, factor, top_n)`.**

Stage 1 — coarse search:

1. Blur **`img_gray`** with **`simple_blur(ksize=factor)`**.
2. Blur **`pattern_gray`** with **`simple_blur(ksize=factor)`**.
3. Downsample both blurred results by **`factor`**.
4. Call **`find_most_likely_positions`** on the small image and pattern to collect the **`top_n`** best candidate positions at low resolution.

Stage 2 — fine search: For each candidate position (sr, sc), scale back to the original image at $r_0 = sr \times \texttt{factor}$, $c_0 = sc \times \texttt{factor}$. Try all offsets (dr, dc) with $dr, dc \in [-\lfloor \texttt{factor}/2 \rfloor, +\lfloor \texttt{factor}/2 \rfloor]$. Skip positions where the pattern falls outside the image. Return the overall best position found.

(d) **Implement `faster_highlight(img_gray, pattern_gray, value, linewidth, factor, top_n)`.**

Like **`highlight_similar_pattern`** from Part 1, but uses **`faster_search`** instead of **`find_similar_pattern_position`**. Two lines.

Comparing the two methods

Run **`test_miniproject_b.py`**. Once Part 2 functions are implemented, the following sections will pass:

- **2B – find_most_likely_positions**: verifies the result list is sorted and that the true position appears in the top 5 candidates.
- **2C – faster_search**: verifies the correct position is found and prints the time taken. Saves **`imgs/out/cartoon_map_faster_search_pattern_highlighted.jpg`**.
- **2D – faster_highlight**: verifies the two-line wrapper.

The summary printed to the console shows the time taken by each method and the speedup ratio so you can compare directly.

The coarse-to-fine result (**`test_faster_search.jpg`**) should highlight exactly the same position as the brute-force result, but be computed in a fraction of the time.

Both methods must find the same position. If they differ, increase **`top_n`** or reduce **`factor`** in **`PATTERN_CORNER`** / **`PATTERN_SIZE`** at the bottom of **`miniproject_b.py`**.

Submission Instructions

- **Group registration on Moodle:** deadline before Monday, April 27, 2026. Students working alone must still register.
- **Submission deadline:** May 23, 2026 at 11:59 PM. You may resubmit as many times as you like before the deadline.
- **Files to submit:** `miniproject_a.py` and `miniproject_b.py`. Only one person per group needs to submit.
- Do not modify function signatures or `miniprojectutils.py`.
- Any helper functions you add must appear in `miniproject_a.py` or `miniproject_b.py`.
- Grading evaluates only the functions listed above, not the test helpers.