ICC/Programmation — Semaine 11, Mini-projet B

CGC/SIE, 4 mai 2026

Le mini-projet est à réaliser en binôme ou individuellement. Les groupes peuvent échanger des idées ou des approches générales, mais pas du code directement.

***Note :** cet énoncé a été traduit automatiquement depuis la version anglaise. En cas d'ambiguïté ou d'incompréhension, référez-vous à la version anglaise ou consultez les assistants.*

Contexte. Dans le Mini-projet-A, vous avez implémenté la comparaison des bords de tuiles (pour reconstruire un puzzle mélangé) ainsi qu'un flou de boîte (pour lisser une image avant d'en réduire la résolution). Ici, vous appliquez la même idée de comparaison de pixels à un nouveau problème : localiser automatiquement une petite région rectangulaire dans une grande carte, en utilisant uniquement les valeurs des pixels.

La carte est une carte touristique animée de la Californie du Sud de 1991, réalisée par MetroGuide. Elle se présente sous forme d'un puzzle en damier — des tuiles alternées ont été retirées et mélangées, exactement comme dans le Mini-projet-A. Vous devez d'abord la reconstruire, puis la convertir en niveaux de gris, et seulement ensuite effectuer la recherche de motif. La **Partie 1** implémente une recherche par force brute qui vérifie toutes les positions possibles. La **partie 2** rend la recherche considérablement plus rapide grâce à une stratégie de basse résolution à haute résolution, et vous mesurerez directement l'accélération obtenue.

Préparation. Téléchargez **`miniproject-b-start.zip`** depuis Moodle et extrayez-le. Ensuite, copiez votre fichier **`miniproject_a.py`** dans le dossier extrait.

L'archive contient :

- **`miniproject_b.py`** — le fichier que vous allez modifier. Implémentez chaque fonction là où vous voyez un commentaire **`TODO`**. Ne modifiez **pas** les noms de fonctions ni leurs paramètres.
- **`test_miniproject_b.py`** — exécutez ce fichier à tout moment pour vérifier votre avancement. Ne le modifiez **pas**.
- **`miniprojectutils.py`** — fonctions utilitaires. Ne le modifiez **pas**.
- **`imgs/cartoon.jpg`** — la carte animée de la Californie du Sud de 1991.
- **`tiles/`** — les tuiles du puzzle en damier de la carte.

Remarque : Assurez-vous que le dossier **`miniproject-b-start`** contient votre implémentation complète de **`miniproject_a.py`**.

`miniproject_b.py` contient 8 fonctions à implémenter. Chaque fonction possède une docstring expliquant ce qu'elle doit faire, des commentaires décrivant les étapes, et un ou plusieurs marqueurs **`TODO`** indiquant précisément où ajouter votre code. Lorsque vous implémentez un **`TODO`**, **remplacez** la ligne **`raise NotImplementedError(...)`** par votre code, sauf indication contraire dans les commentaires.

Pour vérifier votre avancement à tout moment, exécutez :

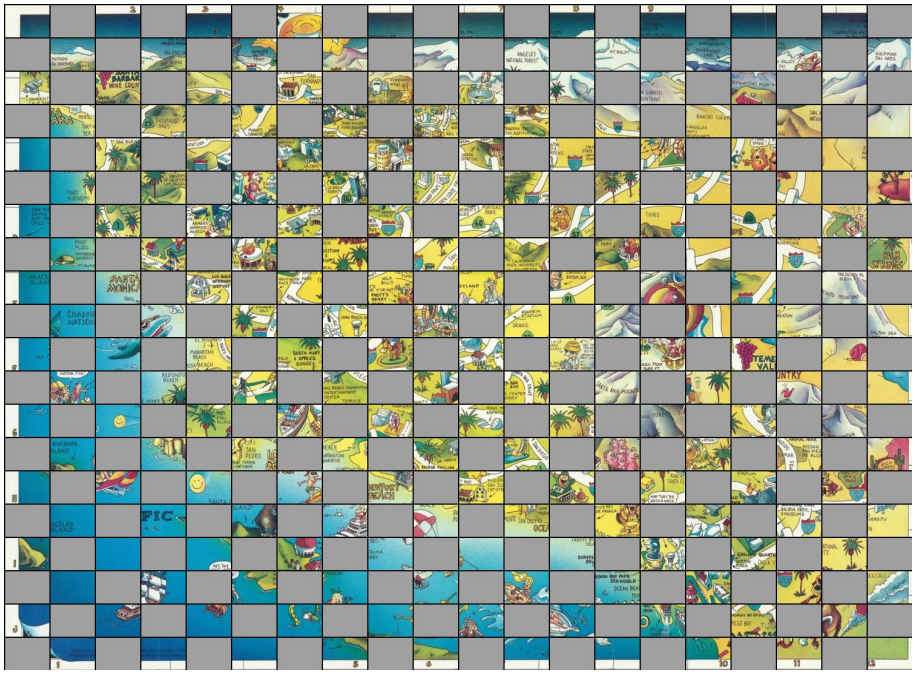
```
python3 test_miniproject_b.py
```

Cela lance tous les tests et affiche un ✓ à côté de chaque fonction correctement implémentée et un **x** à côté de celles qui ne sont pas encore implémentées ou contiennent un bogue. Un résumé final indique combien des 9 sections sont réussies. Lancez les tests fréquemment — après chaque fonction implémentée — afin de détecter les erreurs rapidement.

Important. Le Mini-projet-B importe des fonctions que vous avez écrites dans le Mini-projet-A — plus précisément **`to_grayscale`**, **`simple_blur`**, **`get_edge`**, **`edge_difference`**, **`find_best_tile`** et **`reconstruct`**. Assurez-vous de compléter le Mini-projet-A avant de passer au Mini-projet-B.

Étape 0. Reconstruire et préparer la carte

La carte animée est fournie sous forme de puzzle en damier dans le dossier **`tiles/`** (même format que le Mini-projet-A). Avant de pouvoir rechercher un motif, vous devez réassembler la carte et la convertir en niveaux de gris. Vous n'avez pas besoin d'écrire de nouveau code ici — cette étape est déjà implémentée pour vous.



La carte telle qu'elle est fournie : les tuiles présentes sont en place, les carrés gris marquent les tuiles manquantes à reconstruire.

Exécutez **`test_miniproject_b.py`**. Les tests de l'**Étape 0** devraient réussir immédiatement — la reconstruction et la conversion en niveaux de gris sont déjà implémentées pour vous à partir de vos fonctions du Mini-projet-A. Le test sauvegarde :

- **`imgs/out/cartoon_map_reconstructed.jpg`** — la carte couleur réassemblée.
- **`imgs/out/cartoon_map_gray.jpg`** — la version en niveaux de gris utilisée pour toutes les recherches suivantes.

La carte couleur correctement réassemblée (**`cartoon_map_reconstructed.jpg`**).

L'image en niveaux de gris (**`cartoon_map_gray.jpg`**) utilisée pour toutes les recherches de motif.

Ouvrez **`imgs/out/cartoon_map_gray.jpg`** et vérifiez qu'elle ressemble à l'image ci-dessus. Si le test de reconstruction de l'**Étape 0** échoue, corrigez **`find_best_tile`** dans **`miniproject_a.py`** avant de continuer — la recherche de motif pourrait donner des résultats sans signification sur une carte incorrecte.

Partie 1. Recherche de motif par force brute

L'idée. Étant donné une petite région rectangulaire découpée dans la carte — le motif — l'objectif est de retrouver d'où elle provient. Pour cela, on fait glisser le motif sur toutes les positions possibles dans la carte en niveaux de gris et on calcule un score de similarité à chaque position. La position avec le score le plus bas est la meilleure correspondance.

Le score de similarité est la même formule de différence absolue moyenne que celle utilisée pour **`edge_difference`** dans le Mini-projet-A — la seule différence est qu'on compare maintenant des régions 2D complètes plutôt que des bandes 1D de bords.

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

Calcule la similarité entre **`pattern_gray`** et la région rectangulaire de **`img_gray`** dont le coin supérieur gauche est à **`position = (ligne, colonne)`**.

— Si la région dépasserait les limites de **`img_gray`** dans n'importe quelle direction, retourner **`1.0`** immédiatement.
— Sinon, extraire la région et calculer :

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

Un score de 0 signifie que les deux régions sont identiques pixel par pixel ; 1 signifie qu'elles sont aussi différentes que possible.

Aucune boucle autorisée — utilisez uniquement des opérations numpy (**`np.abs`**, **`np.mean`**, et le découpage de tableaux pour extraire la région). Cette contrainte est essentielle : la fonction suivante appelle **`pattern_difference`** une fois par position de pixel dans l'image, et toute boucle Python à l'intérieur rendrait la recherche entière des ordres de grandeur plus lente.

Indice : extraire la région se fait en un seul découpage : **`img_gray[r : r+ph, c : c+pw]`**, où **`ph`** et **`pw`** sont la hauteur et la largeur du motif.

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

Essaie toutes les positions valides du coin supérieur gauche et retourne celle avec le score **`pattern_difference`** le plus bas. Une position valide **`(r, c)`** est une position où le motif tient entièrement dans l'image : $0 \le r \le H_{\text{img}} - H_{\text{pat}}$ et $0 \le c \le W_{\text{img}} - W_{\text{pat}}$. Retourner **`(0, 0)`** si le motif est plus grand que l'image

dans l'une ou l'autre dimension. Une boucle sur toutes les positions valides est attendue et tout à fait correcte ici.

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

Retourne une **nouvelle** copie de **`img`** (l'entrée ne doit pas être modifiée) avec un bord rectangulaire dessiné par-dessus. Le bord est dessiné autour du rectangle — les pixels à l'intérieur restent inchangés.

— **`corner = (ligne, colonne)`** : pixel supérieur gauche de l'intérieur du rectangle.
— **`size = (hauteur, largeur)`** : dimensions de l'intérieur en pixels.
— **`value`** : intensité des pixels du bord (0 = noir, 255 = blanc).
— **`linewidth`** : épaisseur du bord en pixels.

Utilisez **`clamp(0, limite, x)`** de **`miniprojectutils.py`** chaque fois que vous calculez un indice de ligne ou de colonne qui pourrait dépasser les limites de l'image.

Indice : pensez au bord comme quatre bandes rectangulaires — haut, bas, gauche, droite — chacune remplie par une seule affectation de slice numpy. Pour chaque bande, calculez ses limites en utilisant **`clamp`**, puis affectez la valeur du bord à cette slice.

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

Appelle **`find_similar_pattern_position`** pour trouver la meilleure correspondance, puis appelle **`highlight_rectangle`** pour dessiner un bord à cette position. Cette fonction fait deux lignes.

Choisir et tester votre motif

Les tests sont auto-vérifiants : ils découpent une région connue de la carte et demandent à votre algorithme de la localiser. Puisque la position de découpe est connue, le test peut vérifier automatiquement si la réponse est correcte.

Définir votre motif.
Remarque : Vous pouvez ignorer cette étape et utiliser les valeurs par défaut de **`PATTERN_CORNER`** et **`PATTERN_SIZE`** fournies dans le code — elles pointent vers une région distinctive de la carte qui fonctionne bien pour les tests.

Ouvrez **`imgs/out/cartoon_map_gray.jpg`** et choisissez une zone visuellement distinctive — un nom de lieu, un coude de côte, ou le contour distinctif d'un bâtiment. Évitez les grandes zones uniformes (pleine mer, ciel vide) où de nombreuses positions semblent également similaires. Définissez **`PATTERN_CORNER`** et **`PATTERN_SIZE`** en bas de **`miniproject_b.py`** avec les coordonnées en pixels de votre région choisie.

Exécuter les tests. Exécutez **`test_miniproject_b.py`**. Une fois les fonctions de la Partie 1 implémentées, les sections suivantes réussiront :

— **1A – pattern_difference** : vérifie les conditions aux limites, l'auto-comparaison (score $= 0$), et le cas de différence maximale (score $= 1$).
— **1B – find_similar_pattern_position** : découpe votre région choisie et vérifie que l'algorithme la retrouve exactement à la position de découpe.
— **1C – highlight_rectangle** : vérifie que les pixels intérieurs sont inchangés et que les pixels du bord ont la valeur correcte. Sauvegarde **`cartoon_map_pattern_origin.jpg`** — la carte avec votre région choisie encadrée.
— **1D – highlight_similar_pattern** : vérifie que l'adaptateur en deux lignes appelle correctement les deux fonctions. Sauvegarde **`cartoon_map_found_pattern.jpg`** — la carte avec la région correspondante surlignée en noir.

Exemple de sélection de motif : la région choisie est encadrée en blanc. L'algorithme doit retrouver cette position exacte à partir des pixels découpés seuls.

La meilleure correspondance trouvée par la recherche par force brute, surlignée en noir (**`cartoon_map_found_pattern_highlighted.jpg`**). La console affiche le temps écoulé — notez-le pour comparaison avec la Partie 2.

Remarque : la recherche par force brute vérifie chaque position de pixel dans l'image et peut prendre environ 3 à 4 minutes. Si elle est trop lente, choisissez un ***`PATTERN_SIZE`*** *plus petit.*

Partie 2. Recherche de motif accélérée

Pourquoi la recherche par force brute est-elle lente ?

Pour comprendre concrètement le problème, considérons les chiffres. Supposons que la carte en niveaux de gris fasse 800×600 pixels et que le motif fasse 50×50 pixels. La recherche par force brute vérifie $(800 - 50) \times (600 - 50) = 412\,500$ positions. À chaque position, elle calcule la moyenne de $50 \times 50 = 2\,500$ différences absolues. Cela représente plus d'un milliard d'opérations individuelles. Si l'image est deux fois plus grande, la recherche prend quatre fois plus de temps.

La stratégie grossier-à-fin

Au lieu de chercher dans l'image complète à pleine résolution en un seul passage, la recherche accélérée fonctionne en deux étapes :

Étape 1 — recherche grossière. Réduire l'image et le motif d'un facteur **`factor`** (par ex. facteur 4) pour obtenir des images `factor`^{2} fois plus petites. Effectuer la recherche par force brute sur les petites versions. Garder les **`top_n`** meilleures positions candidates plutôt que la seule meilleure, car la petite image est une approximation et la vraie meilleure position pourrait ne pas arriver en première place à basse résolution.

Étape 2 — affinage fin. Pour chaque position candidate, la ramener aux coordonnées de l'image originale et chercher uniquement dans un petit voisinage autour d'elle. Ce voisinage a un rayon de $\lfloor \texttt{factor}/2 \rfloor$ pixels dans chaque direction. Le nombre total d'appels à **`pattern_difference`** est une infime fraction du nombre de la force brute.

Comment réduire l'image

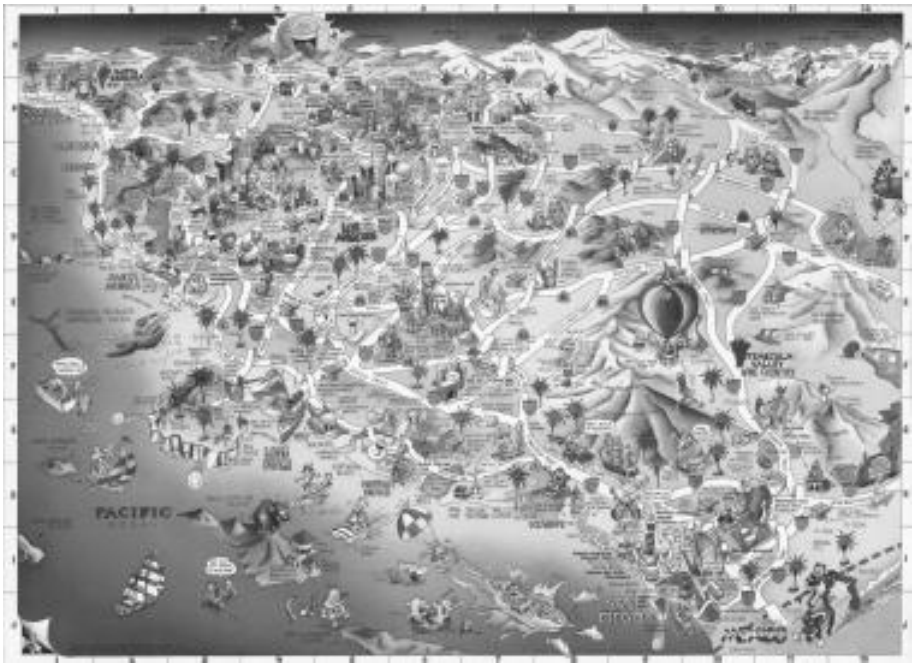
Pour réduire l'image pour l'Étape 1, on utilise deux étapes : flou, puis sous-échantillonnage. Appliquer le flou en premier moyenne chaque pixel avec ses voisins, de sorte que chaque pixel original contribue à l'image réduite — aucune information n'est silencieusement perdue. Le sous-échantillonnage ne garde ensuite qu'un pixel sur **`factor`** dans chaque dimension. Vous avez implémenté **`simple_blur`** dans le Mini-projet-A ; vous le réutilisez ici sans modification.

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

Garde un pixel sur **`factor`** dans chaque dimension en partant de $(0, 0)$. Pas de boucles — une seule slice numpy. Appelez toujours **`simple_blur(ksize=factor)`** sur l'image avant de la passer à **`downsample`**.

Exécutez **`test_miniproject_b.py`**. Une fois implémentée, la section **2A – downsample** réussira et sauvegardera **`imgs/out/cartoon_map_downsampled.jpg`** pour inspection visuelle.

Après **`simple_blur`** (ksize=4) : l'image est lissée et légèrement plus petite en raison du mode de convolution **`'valid'`**.



Après **`downsample`** (factor=4) : l'image est réduite à une fraction de sa taille originale tout en préservant les détails grâce au flou préalable.

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

Même recherche que **`find_similar_pattern_position`** de la Partie 1, mais retourne les **`top_n`** positions avec les scores les plus bas, triées du meilleur (plus bas) au moins bon, construites en un **seul passage**.

Indice : maintenez une liste d'au plus **`top_n`** paires **`(score, position)`**, triée par score à tout moment. Lorsqu'un nouveau score bat l'entrée actuellement la plus mauvaise, remplacez-la et retriez.

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

Étape 1 — recherche grossière :

1. Appliquer le flou à **`img_gray`** avec **`simple_blur(ksize=factor)`**.
2. Appliquer le flou à **`pattern_gray`** avec **`simple_blur(ksize=factor)`**.
3. Sous-échantillonner les deux résultats floutés par **`factor`**.
4. Appeler **`find_most_likely_positions`** sur la petite image et le petit motif pour collecter les **`top_n`** meilleures positions candidates à basse résolution.

Étape 2 — recherche fine : Pour chaque position candidate (sr, sc), revenir à l'image originale avec $r_0 = sr \times \texttt{factor}$, $c_0 = sc \times \texttt{factor}$. Essayer tous les décalages (dr, dc) avec $dr, dc \in [-\lfloor \texttt{factor}/2 \rfloor, +\lfloor \texttt{factor}/2 \rfloor]$. Ignorer les positions où le motif sort de l'image. Retourner la meilleure position trouvée.

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

Comme **`highlight_similar_pattern`** de la Partie 1, mais utilise **`faster_search`** à la place de **`find_similar_pattern_positi`**
Deux lignes.

Comparaison des deux méthodes

Exécutez **`test_miniproject_b.py`**. Une fois les fonctions de la Partie 2 implémentées, les sections suivantes réussiront :

- **2B – find_most_likely_positions** : vérifie que la liste de résultats est triée et que la vraie position apparaît parmi les 5 meilleures candidates.
- **2C – faster_search** : vérifie que la position correcte est trouvée et affiche le temps écoulé. Sauvegarde **`imgs/out/cartoon_map_faster_search_pattern_highlighted.jpg`**.
- **2D – faster_highlight** : vérifie l'adaptateur en deux lignes.

Le résumé affiché dans la console indique le temps pris par chaque méthode et le ratio d'accélération pour une comparaison directe.

Le résultat grossier-à-fin (**`test_faster_search.jpg`**) doit surligner exactement la même position que le résultat par force brute, mais être calculé en une fraction du temps.

Les deux méthodes doivent trouver la même position. Si elles diffèrent, augmentez **`top_n`** ou réduisez **`factor`** dans **`PATTERN_CORNER`** / **`PATTERN_SIZE`** en bas de **`miniproject_b.py`**.

Instructions de soumission

— **Inscription du groupe sur Moodle :** date limite avant le lundi 27 april 2026. Les étudiants travaillant seuls doivent également s'inscrire.
— **Date limite de soumission :** 23 mai 2026 à 23h59. Vous pouvez resoumettre autant de fois que vous le souhaitez avant la date limite.
— **Fichiers à soumettre :** `miniproject_a.py` et `miniproject_b.py`. Une seule personne par groupe doit soumettre.
— Ne modifiez pas les signatures de fonctions ni `miniprojectutils.py`.
— Toute fonction auxiliaire que vous ajoutez doit apparaître dans `miniproject_a.py` ou `miniproject_b.py`.
— La notation évalue uniquement les fonctions listées ci-dessus, pas les assistants de test.