

Programmation

SIE/GCG, Cours 9

22 avril 2024

Jean-Philippe Pellet

```

class ProgramView(Canvas):
    def __init__(self, parent, view):
        Canvas.__init__(self, parent, height=2 * TOP_MARGIN + 4 * LINE_HEIGHT, highlightthickness=0, view)

    def redraw(
        self,
        program: Program,
        current_subprogram: List[Instruction],
        current_instruction_index: int,
    ) -> None:
        height = self.winfo_height()
        width = self.winfo_width()

        self.delete(ALL)
        self.create_rectangle(0, 0, width, height, fill=window_background_color, width=0)

        # boucle pour les 4 sous-programmes P1 à P4
        for l, subprogram in enumerate(
            [program.P1, program.P2, program.P3, program.P4]
        ):
            # dessin du titre
            instruction_center_y = TOP_MARGIN + 1 * LINE_HEIGHT + LINE_HEIGHT // 2
            self.create_text(LEFT_MARGIN // 2, instruction_center_y, text=f"P{l + 1}")

            # dessin de chaque instruction
            for j, instr in enumerate(subprogram):
                instruction_center_x = (
                    LEFT_MARGIN
                    + j * (INSTRUCTION_BOX_SPACING + INSTRUCTION_BOX_WIDTH)
                    + INSTRUCTION_BOX_WIDTH // 2
                )
                instruction_x = instruction_center_x - INSTRUCTION_BOX_WIDTH // 2
                instruction_y = instruction_center_y - INSTRUCTION_BOX_HEIGHT // 2
                instruction_width = INSTRUCTION_BOX_WIDTH
                instruction_height = INSTRUCTION_BOX_HEIGHT

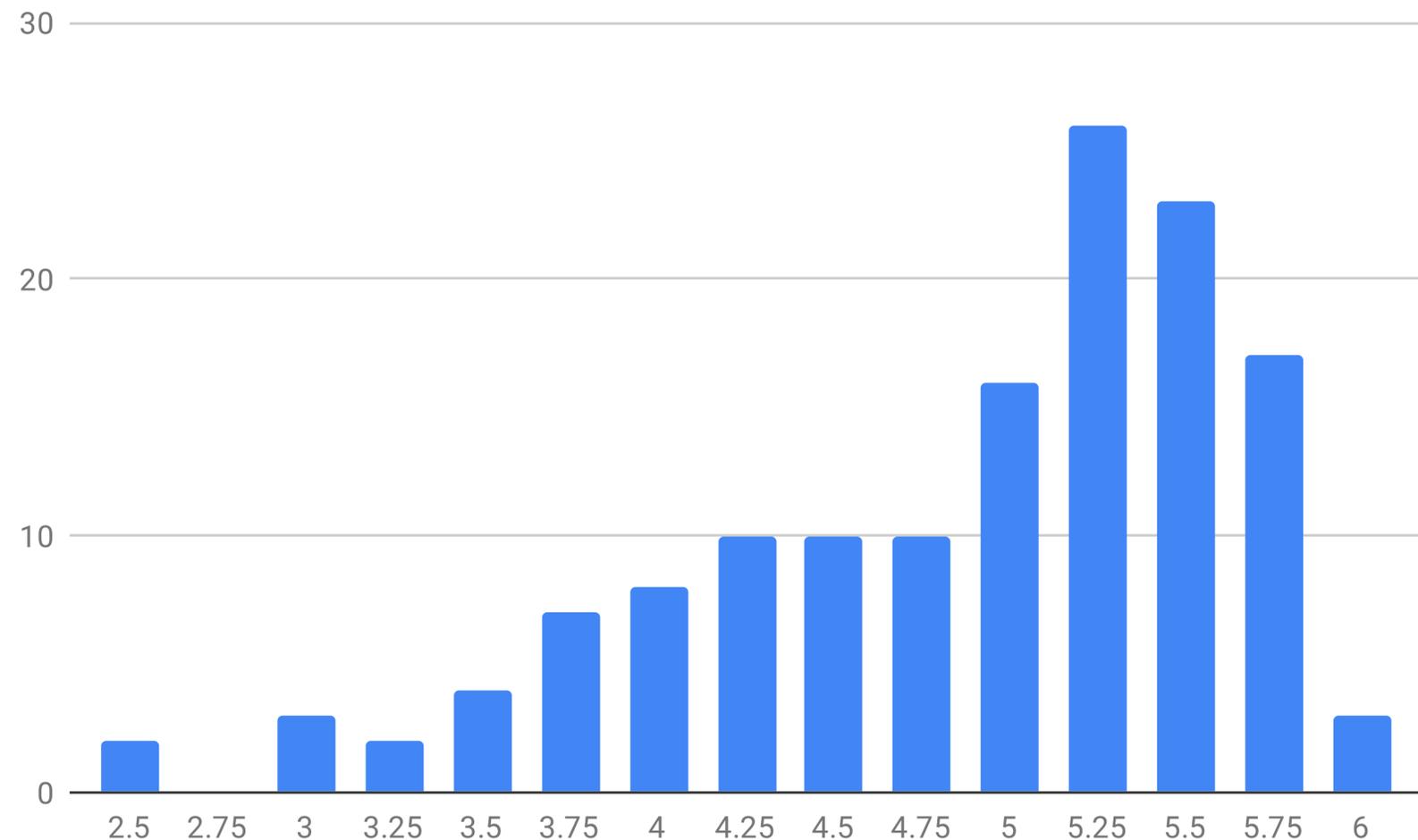
```

Previously, on Programmation...

- **Types** de base en Python: `int`, `float`, `str`, `bool`
- **Méthodes, fonctions et slicing** pour calculer des valeurs dérivées
- **Conditions** pour exécuter du code selon la valeur d'une expression booléenne:
`if` `<condition>`: ... `else`: ... et ses variantes
- **Boucles** pour exécuter du code plusieurs fois:
 - Boucle `while` `<condition>`: ...
 - Boucle `for` `i in range(...)`: ...
- **Déclaration de fonctions** avec type de retour et paramètres:
 - `def` `calculate_area(r: float) -> float: return ...`
- Utilisation de **listes**, **sets** et **dictionnaires**
- Déclaration de **classes**: `@dataclass class` `Rectangle`: ...
- Création, chargement, manipulation et sauvegarde d'**images**

Midterm

- Résultats sur Moodle dans le **carnet de notes**, exercice par exercice
- Proposition de **corrigé** sur Moodle à la semaine 7
- Très bons résultats!
 - 87% de ≥ 4
- Les **points** font foi pour le calcul final de la note ICC
 - Note *indicative*

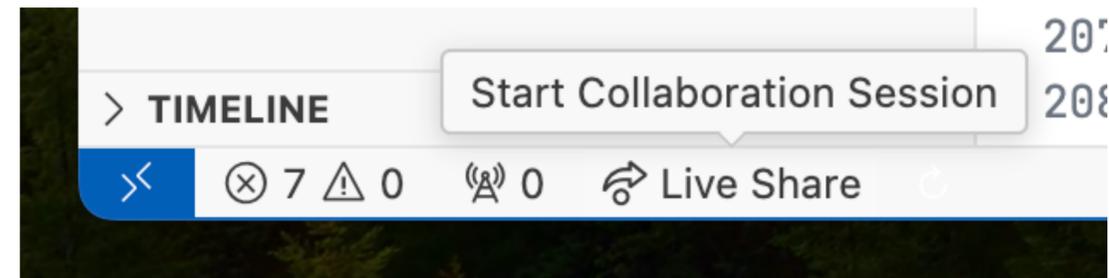


Examen final

- **24 juin 2024**, 15h15, *salles à confirmer*
- Durée: **2 heures et 40 minutes**
- Compte pour **75%** de la note
- Contenu: $\frac{1}{3}$ ICC-P, $\frac{2}{3}$ ICC-T

Travailler à deux sur VS Code

- L'hôte peut **partager son workspace** en live
 - Installer **Live Share**:
<https://marketplace.visualstudio.com/items?itemName=MS-vsliveshare.vsliveshare>
 - Faire un login GitHub ou Microsoft
 - Puis se partager le lien
 - Attention, cela ne change les fichiers que chez l'hôte!



Démo

Miniprojet et manipulation de pixels — *rappel*

- La **lecture d'un pixel** via `img[x, y]` donne:
 - Un **nombre entier** si l'image est en niveaux de gris
 - Une **liste de trois nombres** si l'image est en RGB
- Dans l'**écriture d'un pixel** via `img[x, y] = p`, `p` doit être:
 - Un **nombre entier** si l'image est en niveaux de gris
 - Une **liste de trois nombres** si l'image est en RGB
- On écrit `img[x, y]` plutôt que `img[x][y]`
 - Plus efficace, fonctionne ainsi grâce à la bibliothèque **numpy**
 - Seulement pour les **images** du miniprojet, pas pour les listes de listes standard Python

Images RGB, images en niveaux de gris

- Attention, on ne peut pas traiter une **image RGB** comme une **image en niveaux de gris**
 - Avec numpy, on ne peut pas stocker un int représentant un niveau de gris dans un emplacement prévu **pour une liste de trois valeurs**
 - Vos fonctions `apply_kernel` et `to_grayscale` doivent **créer de nouvelles images**, pas modifier les données des images existantes passées en paramètres
- Si vous avez des soucis:
 - Mettez des **breakpoints**, décomposez vos calculs **sur plusieurs lignes**
 - Exécutez **ligne par ligne**; vérifiez à chaque ligne que ce que vous avez dans vos variables est ce que vous pensez
 - Testez avec des **images générées de petites tailles** (3×3 , 5×5), pour être sûr que ça marche
 - ➔ Démo plus tard

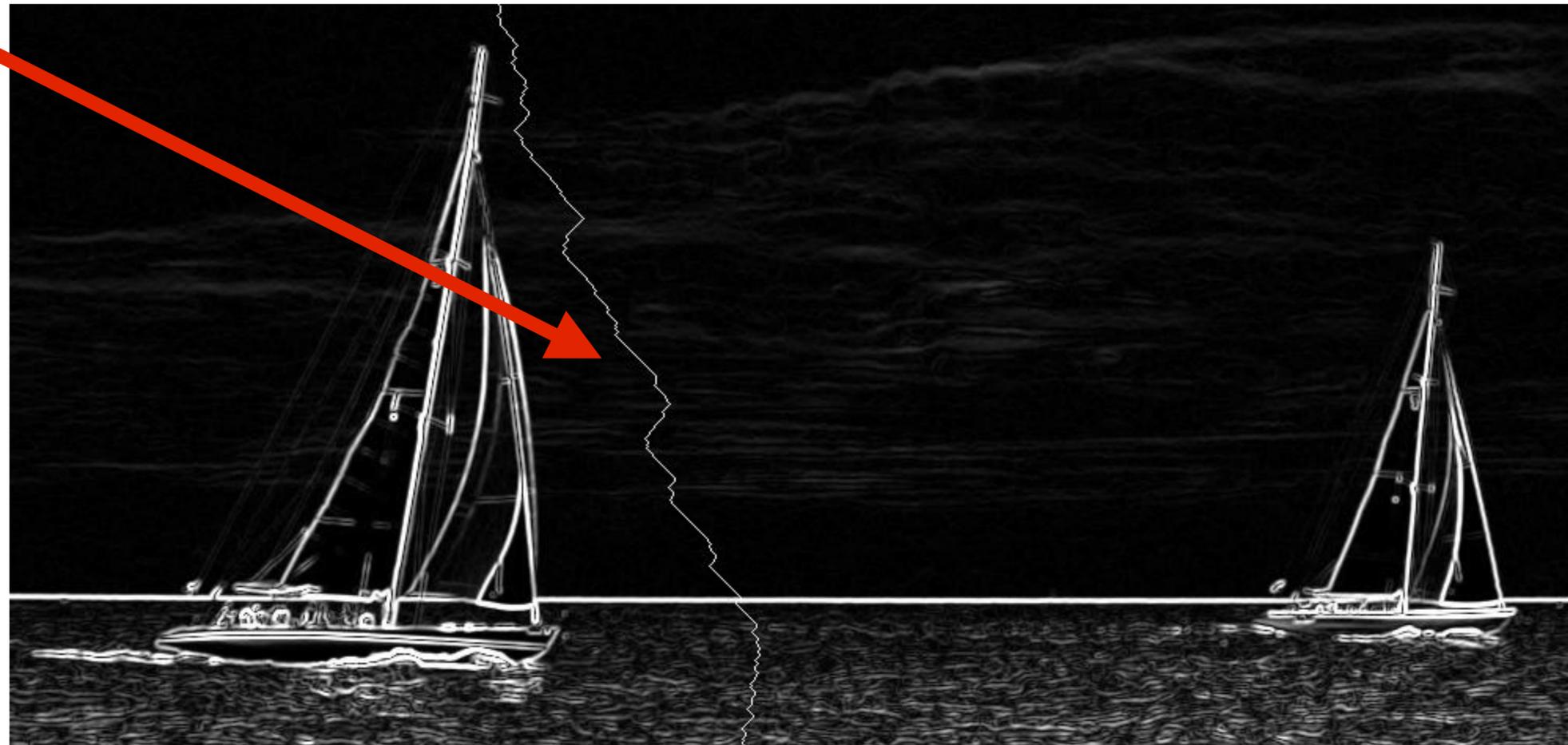
Cours de cette semaine

Algorithme pour trouver le seam de moindre énergie

Processus



niveaux de gris,
lissage, Sobel



recherche
d'un seam

Miniprojet — structure de base, rappel

```
def seam_carving(img_path: str, num_cols: int) -> None:
```

```
    name, ext = split_name_ext(img_path)
```

```
    folder = name + os.path.sep
```

```
    os.makedirs(folder, exist_ok=True)
```

```
    img = load_image(img_path)
```

```
    img_grey = to_grayscale(img)
```

```
    save_image(img_grey, folder + "grey" + ext)
```

```
    img_grey = smoothen(img_grey)
```

```
    save_image(img_grey, folder + "smooth" + ext)
```

```
    img_grey = sobel(img_grey)
```

```
    save_image(img_grey, folder + "sobel" + ext)
```

```
    img_grey = np.uint8(img_grey)
```

```
    for i in range(num_cols):
```

```
        seam = find_seam(img_grey)
```

```
        img_highlight = highlight_seam(img, seam)
```

```
        save_image(img_highlight, folder + f"highlight_{i}" + ext)
```

```
        img_grey_highlight = highlight_seam(img_grey, seam)
```

```
        save_image(img_grey_highlight, folder + f"highlight_{i}_grey" + ext)
```

```
        img = remove_seam(img, seam)
```

```
        save_image(img, folder + f"step_{i}" + ext)
```

```
        img_grey = remove_seam(img_grey, seam)
```

Semaine passée

Boucle qui se répète autant de fois qu'on veut enlever de colonnes

On cherche un *seam* — notre préoccupation principale maintenant!

On enregistre des images intermédiaires où on montre où se trouve le *seam* détecté

On supprime le *seam* de l'image (et aussi de l'image en gris) et on enregistre le résultat, puis on recommence la boucle si nécessaire

Tâche: trouver un seam

Un pixel par ligne, on peut se déplacer d'une ligne à l'autre en bougeant d'un pixel à gauche ou à droite

129	137	25	147	44	64	192	247	11	81
143	164	152	220	1	155	242	152	205	116
189	16	53	115	248	102	157	234	76	188
122	79	48	186	96	78	32	230	170	1
106	123	17	78	8	27	123	145	231	70

Coût: 206

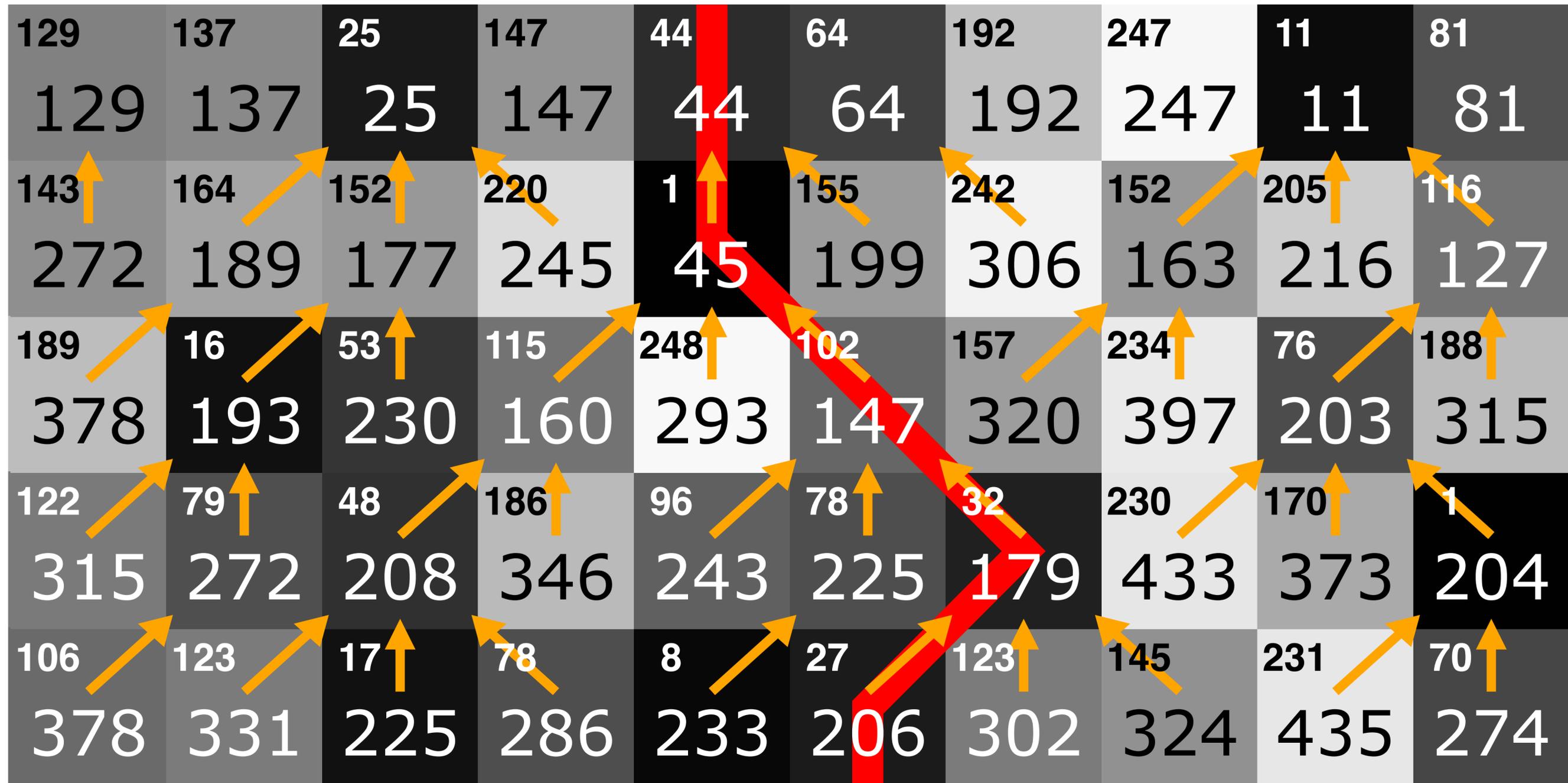
Ne commence pas forcément par la plus petite valeur, ne finit pas forcément par la plus petite valeur...

Comment trouver ce chemin?

Algorithme: idée générale

- Variante de l'**algorithme du plus court chemin** de Dijkstra (1956)
- On commence par la **ligne du haut** et on considère tous les chemins de longueur 1 qui commence à chaque pixel. Leur **coût** est égal à **la valeur du pixel** en question.
- Pour **chaque ligne** suivante:
 - Pour chaque pixel de la ligne, on cherche, parmi les **3 prédécesseurs possibles**, quel serait le **chemin de moindre coût** pour arriver à ce pixel-ci, en additionnant la valeur du pixel aux coûts des chemins menant aux 3 prédécesseurs. On garde ensuite le meilleur.
 - ➔ On se rappelle non seulement le **coût** du meilleur chemin, mais également **quel pixel du rang précédent nous a donné ce meilleur coût** — sinon on ne saura pas revenir en arrière
- À la fin, on sélectionne **le pixel d'arrivée qui a le meilleur coût total** en regardant la dernière ligne, et on **revient en arrière** pour trouver le chemin complet

Algorithme: exemple



Implémentation: représentation de la grille de recherche

- Les valeurs de gris sont directement tirées des données de l'image
- Les coûts cumulés et les prédécesseurs pour chaque cellule doivent être gérés par vous séparément
- Base: une `list[list[PixelData]]` à créer

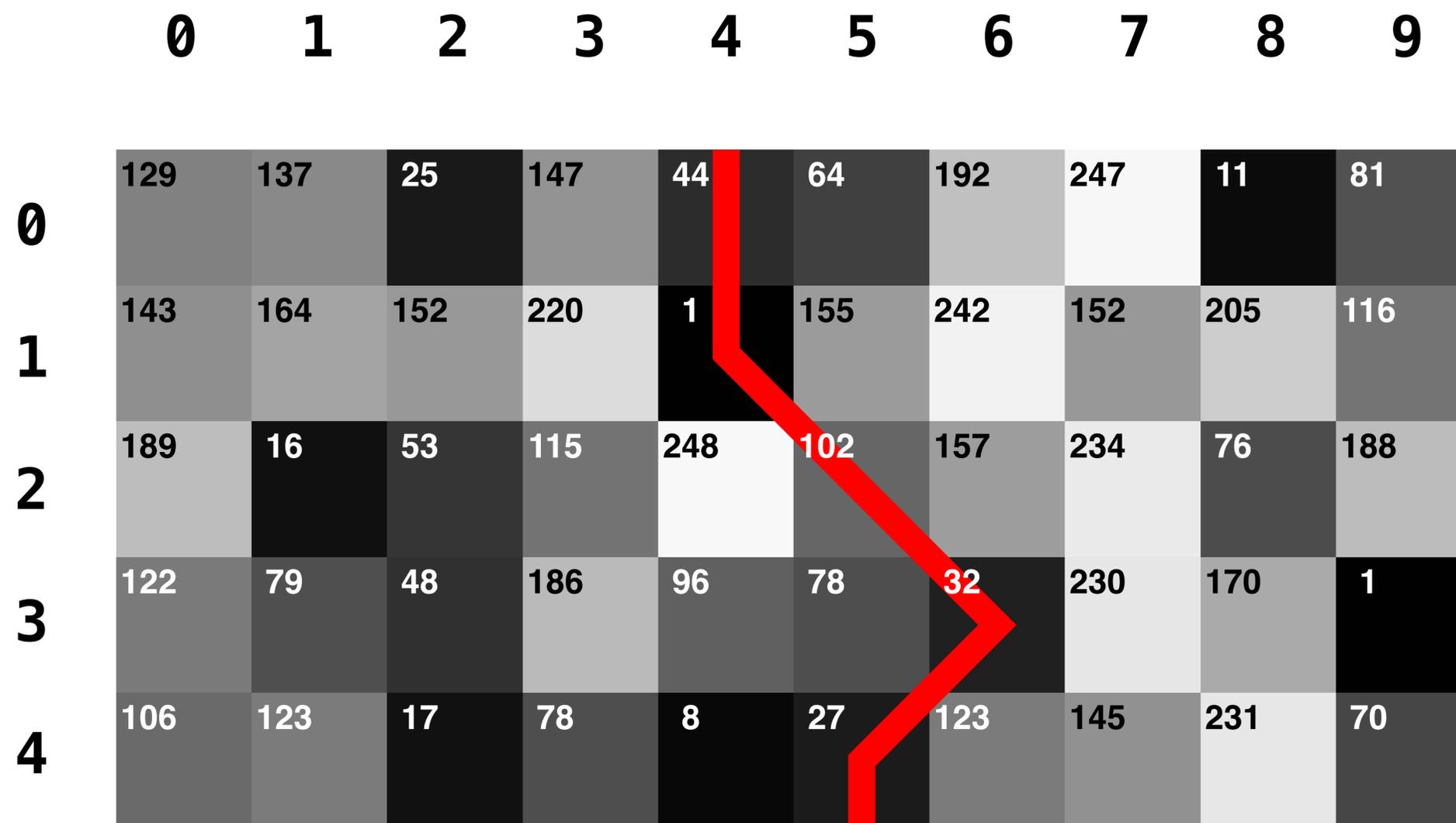
```
@dataclass
class PixelData:
    min_energy: int = Infinity
    parent: Position = (-1, -1)
```

129	137	25	147	44	64	192	247	11	81
143	164	152	220	1	155	242	152	205	116
189	16	53	115	248	102	157	234	76	188
122	79	48	186	96	78	32	230	170	1
106	123	17	78	8	27	123	145	231	70

129	137	25	147	44	64	192	247	11	81
129	137	25	147	44	64	192	247	11	81
272	189	177	245	45	199	306	163	216	127
189	16	53	115	248	102	157	234	76	188
378	193	230	160	293	147	320	397	203	315
122	79	48	186	96	78	32	230	170	1
315	272	208	346	243	225	179	433	373	204
106	123	17	78	8	27	123	145	231	70
378	331	225	286	233	206	302	324	435	274

Implémentation: représentation d'un seam

- Un *seam* est simplement une **liste de ints**: l'**index de la colonne** de chaque pixel, de haut en bas



seam = [4, 4, 5, 6, 5]

Démo

miniprojectutils.py: fonctions utiles

- **def save_image_greyscale_details(...)**
 - Sauvegarde une image en niveaux de gris avec les valeurs écrites dessus (comme dans mes slides)
 - On peut également passer un seam comme argument supplémentaire pour qu'il soit dessiné
- **def new_image_grey_with_data(data: list[list[int]]) -> Image**
 - Permet de créer une petite image en niveaux de gris avec directement une matrice de valeurs de 0 à 255
- **def new_random_grey_image(height: int, width: int) -> Image**
 - Crée une nouvelle image avec des niveaux de gris aléatoires et les dimensions données
 - ➔ Utile pour tester votre fonction sur de petites instances inspectables facilement

PixelData



```
@dataclass
class PixelData:
    min_energy: int = Infinity
    parent: Position = (-1, -1)
```

- Durant la recherche, ces données vont être représentée par des PixelData, un par pixel
- Stocke l'énergie cumulée minimale pour attendre ce pixel (`min_energy`)
- Stocke aussi dans `parent` les coordonnées du pixel du haut
- Utile pour recréer le seam depuis le bas

find_seam — structure de base

```
@dataclass
class PixelData:
    min_energy: int = Infinity
    parent: Position = (-1, -1)

def find_seam(img_grey: Image) -> Seam:
    """Find the seam with the lowest energy."""
    print("    Finding seam...")

    # 5.1 Initialize the grid of empty PixelData and fill the first row
    cells: list[list[PixelData]] = find_seam__init_pixel_data(img_grey)

    # 5.2 Fill the rest of the grid
    find_seam__fill_pixel_data(cells, img_grey)

    # 5.3 Find the minimum energy on the last row
    min_col = find_seam__find_min_column_on_last_row(cells)
    print(f"    Min energy at col {min_col} is {cells[-1][min_col].min_energy}")

    # 5.4 Backtrack the seam
    seam = find_seam__backtrack_seam(cells, min_col)
    return seam
```

Doit créer une matrice de PixelData, un par pixel, pour l'image donnée, et initialisation de la première ligne selon les valeurs de gris des pixels eux-mêmes

Remplit les autres lignes de cells en recherchant à gauche et à droite dans la ligne d'avant pour chaque pixel

Trouve le meilleur pixel d'arrivée sur la dernière ligne

Recrée le seam depuis le pixel de la dernière ligne en remontant à travers les parents successifs jusqu'à la première ligne

Résumé Cours 9

- Un *seam* (pour le miniprojet) est un chemin du haut de l'image vers le bas avec un pixel par ligne, qui peut se décaler d'un pixel à gauche ou à droite à chaque ligne
- Pour réduire l'image en largeur d'une unité, on va chercher le *seam* dont la somme des valeurs des pixels est la plus petite (qui représente donc le moins d'information)
- Pour le chercher, on applique un algorithme qui est un cas particulier de l'algorithme du plus court chemin de Dijkstra
- La recherche se fait en deux grandes phases: recherche progressive puis *backtracking*