Tp - Redimensionnement d'une photo

On s'intéresse dans ce tp à un redimensionnement « intelligent » d'une photo.

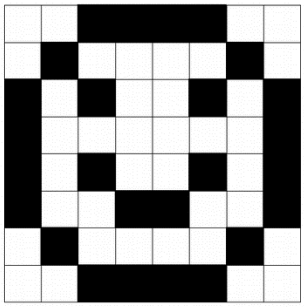
1- Découverte du dimensionnement « intelligent » d'une photo :

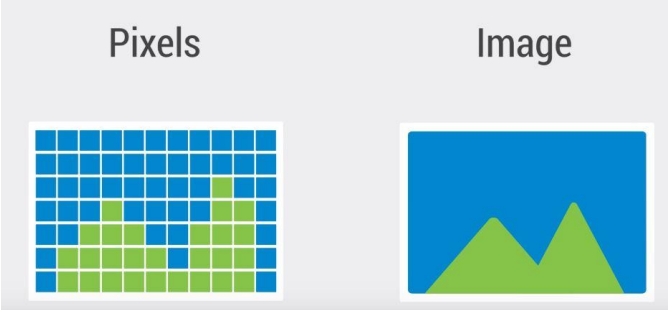
Sur un écran ou une image, le pixel est le plus petit élément constitutif d'une image :

- Chaque pixel va être codé avec un code couleur.
- Si on l'on regarde de nouveau cette image et que l'on zoome, on remarque qu'elle est composée de carrés de couleurs.

C'est l'organisation de ces pixels qui va former une image matricielle.

Chaque pixel est organisé dans un quadrillage, ou chaque case va contenir un code couleur. Au lieu de pixel, on parle aussi de « point » d'où le second nom de cette méthode de codage : la représentation **Bitmap** (pour « Carte de points »). Une image matricielle est donc un tableau de valeurs (ou matrice) qui sera encodé dans un format particulier (PBM, TIFF, PNG, JPG)





Une image étant une matrice de pixels. On peut alors affecter un « poids » à chaque pixel en fonctions de critères déterminés :

- même couleur que ses voisins = un poids faible (couleur du ciel),
- en fonction de sa couleur,
- en fonction des caractéristiques de ses voisins.

Au final chaque case du tableau de pixels aura un certain poids ou une certaine « énergie ».
Exemple de redimensionnement « intelligent » :

Image initiale	*Image rognée*	*Image redimensionnée*
		

Sur cet exemple l'image simplement rognée perd des informations importantes (la forme de la maison de plaisance de l'image et la position du personnage dans l'image).

L'image redimensionnée intelligemment garde les informations essentielles et supprime celles de moindre importance (partie du ciel et de la pelouse).

Le traitement d'image par cette méthode est détaillé dans cette vidéo : https://youtu.be/6NcIJXTlugc

2- OBJECTIF DU TP :

L'objectif de ce TD est de définir un algorithme qui permet de trouver des lignes descendantes traversant l'image de haut en bas et pour lesquelles la somme des poids des pixels traversés sera minimale.

Pour illustrer un principe de redimensionnement nous allons prendre un exemple modeste : une image de 4 x 4 pixels avec les poids suivants :

1	2	8	9
7	2	2	1
2	1	4	5
10	8	6	2

⇨ Solution optimale :

En analysant cette grille, on peut visuellement voir que la ligne « de poids minimal » sera la suivante : 1 ⇨ 2 ⇨ 4 ⇨ 2 pour un poids total de 9

1	2	8	9
7	**2**	2	1
2	1	**4**	5
10	8	6	**2**

Il s'agit à présent de définir un algorithme qui permette de trouver ces lignes automatiquement

⇨ Solution gloutonne :

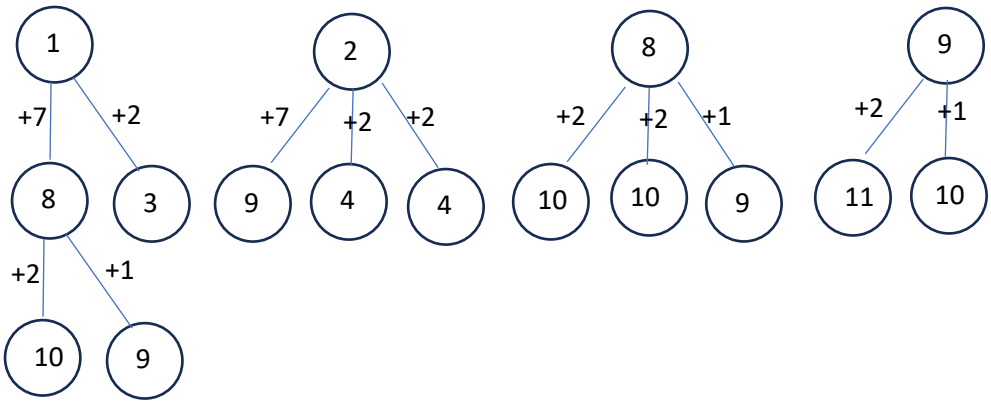
Le premier algorithme qui vient à l'esprit consiste à retenir pour chaque ligne traversée le pixel de poids minimal. On obtient ainsi un algorithme de type Glouton. En informatique, un algorithme glouton (greedy algorithm en anglais, parfois appelé aussi algorithme gourmand) est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local. Dans certains cas cette approche permet d'arriver à un optimum global, mais dans le cas général c'est une heuristique/règle empirique pris telle quelle.

1	2	8	9
7	**2**	2	1
2	**1**	4	5
10	8	**6**	2

Cet algorithme permet-il de trouver la solution optimale ? : 1 ⇨ 2 ⇨ 1 ⇨ 6 pour un poids total de 10

3- QUEL CODE NAÏF POUR DEFINIR LA SOLUTION OPTIMALE ? :

L'objectif de ce TD est de définir un algorithme qui permet de trouver des lignes descendantes traversant l'image de haut en bas et pour lesquelles la somme des poids des pixels traversés sera minimale.



⇨ Quelle est la complexité de ce type d'algorithme ? :

Si on suppose, pour simplifier, qu'à chaque nœud on a 3 choix possibles, pour une image de n pixels sur n pixels, le nombre de différents cas à étudier en bout d'arbre sera de $n \times 3^{n-1}$. On a une complexité qui est donc supérieure à $\mathcal{O}(3^n)$ si $n \geq 3$. Pour une image de $n = 100$ pixels, le nombre de cas à étudier serait de l'ordre de $3^{100} \approx 5 \,.\, 10^{47}$. Cette complexité est appelée complexité exponentielle.

4- CODE QUI UTILISE LE PARADIGME DE PROGRAMMATION DYNAMIQUE POUR DEFINIR LA SOLUTION OPTIMALE ? :

⇨ Méthode : Le gros inconvénient de la méthode naïve est sa complexité qui est exponentielle. En se plaçant dans un paradigme de programmation dynamique (décomposition du problème complet en sous-problèmes et mémoïsation des calculs intermédiaires), on obtient un code bien plus efficace.

⇨ Principe : On étudie les lignes les unes après les autres en partant du haut. On détermine un tableau de même dimension que celui qui contient les pixels de l'image. Il contient les poids cumulés minimaux pour un chemin qui part de la première ligne.

Tableau « pixels » :

1	2	8	9
7	2	2	1
2	1	4	5
10	8	6	2

Tableau « memo » :

1	2	8	9
8	3	4	9
5	4	7	9
14	12	10	9

La fonction *optimalRec()* ci-dessous prend en paramètre la liste de listes « pixels » qui contient l'image. Il renvoie le tableau « memo » qui contient, pour chaque pixel, le poids minimal cumulé d'un chemin provenant de la ligne du haut.

```
def optimalRec(pixels):
    def optimal(pixels,i) :
        n = len(pixels)
        if i == 0 :
            memo[i] = pixels[0]
            return memo[i]
        memo[i] = [0 for j in range(n)]
        for j in range(n) :
            if j == 0 :
              memo[i][j] = pixels[i][j] + min(optimal(pixels,i-1)[j] ,optimal(pixels,i-1)[j+1])
            elif j == n-1 :
              memo[i][j] = pixels[i][j] + min(optimal(pixels,i-1)[j-1] ,optimal(pixels,i-1)[j])
            else :
              memo[i][j]=pixels[i][j]+min(optimal(pixels,i-1)[j-1],optimal(pixels,i-1)[j],optimal(pixels,i-1)[j+1])
        return memo[i]

    n = len(pixels)
    memo = [[0 for j in range(n)] for i in range(n)]
    optimal(pixels,n-1)
    return memo
```

En exécutant :

```
pixels = [
            [1,2,8,9],
            [7,2,2,1],
            [2,1,4,5],
            [10,8,6,2]
        ]
print(optimalRec(pixels))
```

on obtient la liste memo suivante :

```
[[1, 2, 8, 9], [8, 3, 4, 9],
[5, 4, 7, 9], [14, 12, 10, 9]
]
```

Question : Ecrire le code python de la fonction optimalIter() qui a en paramètre la liste pixes et qui renvoie la liste memo en utilisant à présent une méthode itérative.

```python
def optimalIter(pixels):
    n = len(pixels)
    memo = []
    for i in range(n) :
        if i == 0 : memo.append(pixels[0])
        else :
            ligne = [0 for j in range(n)]
            for j in range(n):
                if j == 0 :
                    ligne[j] = pixels[i][j] + min(memo[i-1][j] ,memo[i-1][j+1])
                elif j == n-1 :
                    ligne[j] = pixels[i][j] + min(memo[i-1][j-1] ,memo[i-1][j])
                else :
                    ligne[j] = pixels[i][j] + min(memo[i-1][j-1] , memo[i-1][j] ,memo[i-1][j+1])
            memo.append(ligne)
    return memo
```

5- CODE QUI PERMET DE RECONSTITUER LE CHEMIN DE POIDS MINIMAL :

a. FONCTION INTERMEDIAIRE :

La fonction *indice_min()* ci-dessous est incomplète. Elle a comme paramètres une liste l, et 2 entiers iG et iD, avec iG < iD. Elle renvoie l'indice $iG \leq i \leq iD$ du minimum $l[i]$.

On donne en exemple l'exécution ci-contre :

```
>>> indice_min([4,8,0,-3],1,3)
3
```

```python
def indice_min(l,iG,iD) :
    min = l[iG]
    indice_min = iG
    for i in range(iG,iD+1) :
        if l[i] <= min :
            min = l[i]
            indice_min = i
    return indice_min
```

⇨ Compléter ce code.

b. FONCTION QUI DONNE LE CHEMIN DE POIDS MINIMAL :

La fonction solution*()* ci-dessous est incomplète. Elle a comme paramètres les listes pixels et memo. Elle renvoie une liste qui contient le chemin de poids minimal.

On donne en exemple l'exécution ci-dessous :

```python
pixels = [
            [1,2,8,9],
            [7,2,2,1],
            [2,1,4,5],
            [10,8,6,2]
        ]
memo = optimalIter(pixels)
chemin = solution(memo,pixels)
print(chemin)
```

On obtient :

```
>>> (executing file "tp1.py")
[2, 4, 2, 1]
```

1	2	8	9
7	2	2	1
2	1	4	5
10	8	6	2

⇨ Compléter le code de cette fonction :

```
def solution(memo, pixels):
    n = len(memo)
    i = n-1
    j = indice_min(memo[i],0,n-1)
    chemin = []
    chemin.append(pixels[i][j])
    i = n-2
    while i >=0 :
        if j == 0 : j = indice_min(memo[i],0,1)
        elif j == n-1 : j = indice_min(memo[i],n-2,n-1)
        else : j = indice_min(memo[i],j-1,j+1)
        chemin.append(pixels[i][j])
        i = i - 1
    return chemin
```