DS2 PCE 2024

December 12, 2024

[3]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import os
```

Ce devoir est tiré de l'excellent travail de Mickaël Péchaud.

[4]:
```
guepier = (mpimg.imread('./guepiersNB.jpg')).tolist()
```

1 Introduction

[9]:
```
#Q1
def dim(img:list)->tuple:
    return (len(img[0]),len(img))

dim(image)
```

[9]:
```
(690, 460)
```

[11]:
```
#Q2.a
def inv(img:list)->list:
    return [[255-e for e in ligne] for ligne in img]
```

Seuiller par un entier s une image en niveau de gris consiste à appliquer à chaque pixel (donc une valeur entre 0 et 255) la fonction suivante

$$f : x \mapsto \begin{cases} 225 \text{ si } x \geq s \\ 0 \text{ sinon} \end{cases}$$

[13]:
```
#Q2.b
def seuil(img:list,s)->list:
    f = lambda x: 255 if x >= s else 0
    return [[f(e) for e in ligne] for ligne in img]
```

1.0.1 Q2c

[17]:
```
#Q2.c

def symh(img:list)->list:
    return img[::-1]
```

```python
def symh(img:list)->list:
    """sans slicing"""
    w,h=dim(img)
    return [img[h-i-1][:] for i in range(h)]
```

Le principe est le même pour la symétrie horizontale. On évite le slicing, pour varier les plaisirs.

[19]:
```python
#Q2.d
def symv(img:list)->list:
    mat=[]
    for ligne in img:
        line=[ligne[len(ligne)-1-i] for i in range(len(ligne))]
        mat.append(line)
    return mat
```

2 Réduction de largeur naïve

[21]:
```python
#Q3
def reduction_moitie_ligne(l:list) -> list:
    assert(len(l) % 2 == 0)
    n=len(l)
    return [(l[2*i]+l[2*i+1])//2 for i in range(n//2)]#O(len(l))
```

La complexité est linéaire en la longueur de la ligne de pixels. Comme il y w colonnes, la complexité est en $O(w)$

[24]:
```python
#Q4
def reduction_moitie_image(img:list) -> None:
    for i in range(len(img)):
        img[i] = reduction_moitie_ligne(img[i])
```

Dans la boucle il y a h passages. En interne, il y a une autre boucle cachée par l'appel à `reduction_moitie_ligne`.

Au total $O(wh)$

Une méthode de réduction de largeur d'un facteur autre que 2 Question `Q5`

Pour $k > 1$, on contracte k pixels consécutifs en un, en faisant la moyenne des k pixels (on remplace par `s//k` où `s` désigne la somme des pixels.

Pour le reste des pixels de la ligne (il y en a strictement moins que k), on les remplace par leur moyenne. Ce n'est par parfait, mais ça ne concerne au final qu'une colonne de l'image compressée.

On peut aussi répartir autrement les r pixels restants (pour $w = qk + r$; $0 \leq r < k$). Sur les r premiers regroupement de pixels de la lignes, on fait la moyenne de $k+1$ pixels, ensuite on regroupe par paquet de k.

3 Calcul de l'énergie d'un pixel

L'énergie verticale d'un pixel non situé sur un bord se calcule en divisant la différence de valeur des deux pixels verticaux qui l'encadrent par leur distance; c'est à dire 2. On procède de même pour l'énergie horizontale.

Lorsqu'un pixel est sur un bord horizontal (donc haut ou bas), l'énergie verticale est le quotient de (la différence entre la valeur du pixel étudié et celle de son unique voisin à la verticale) et la distance entre ces deux pixels; c'est à dire 1.

L'énergie horizontale d'un pixel sur un bord vertical se calcule de la même manière.

[26]:
```python
#Q6
def energie(img:list) -> list:
    '''Calcule l'énergie d'une image img.'''
    (w, h) = dim(img)
    e = [[0 for j in range(w)] for i in range(h)]#matrice nulle
    for i in range(h):
        for j in range(w):
            if i == 0:
                vx = (img[1][j] - img[0][j])/2
            elif i == len(e) - 1:
                vx = (img[i][j] - img[i-1][j])/2
            else:
                vx = (img[i+1][j] - img[i-1][j])/2
            if j == 0:
                vy = (img[i][1] - img[i][0])/2
            elif j == len(e[0]) - 1:
                vy = (img[i][j] - img[i][j-1])/2
            else:
                vy = (img[i][j+1] - img[i][j-1])/2
            e[i][j] = abs(vx)+abs(vy)
    return e
```

On parcourt tous les pixels, et pour chacun on fait 2 accès (au plus), 2 affectations (au plus), quelques opérations arithmétiques dont deux comparaisons. Pour chaque pixel, le coût est en $O(1)$.

La complexité totale est $O(wh)$

4 Réduction ligne par ligne

Beaucoup d'élèves ont utilisé `l.pop(i)`.

Il y a deux problèmes avec cette approche :

- on n'est pas sûr que le jury autorise la méthode `pop`
- si on n'a pas fait une copie de `l`, la liste est alors modifiée. Or le type de retour (`list` et pas `None`) devrait décourager les effets de bords.

[46]:
```
#Q7
def enlever(l:list,i:int) -> list:# avec slicing
    return [l[j] for j in range(len(l)) if i!=j]

def enlever(l:list,i:int) -> list:#quasiment à la main
    res=[]
    for j in range(len(l)):
        if j!=i: res.append(l[j])
    return res

def enlever(l:list,i:int) -> list:# avec slicing
    return l[:i]+l[i+1:]
```

Complexité en $O(w)$

[47]:
```
enlever([1,2,3,4,5,6,7],2)
```

[47]:
```
[1, 2, 4, 5, 6, 7]
```

[48]:
```
#Q8
def indice_min(l:list) -> int:
    indice = 0
    for i in range(len(l)):
        if l[indice]>l[i]:
            indice=i
    return indice
```

Complexité en $O(w)$

[50]:
```
l = [10,2,86,1,8,1]
indice_min(l)
```

[50]:
```
3
```

4.0.1 Q9

Trop d'élèves ont calculé l'énergie à chaque tour de boucle. Avec cette approche, la complexité est de $O(h^2w)$.

Il faut calculer la matrice d'energie avant la boucle une fois pour toute. Puis, pour chque ligne, enlever le pixel d'énergie minimale.

[51]:
```
#Q9
def reduction_ligne_par_ligne(img:list)->None:
    w,h=dim(img)
    e=energie(img)
    for i in range(h):
        m=indice_min(e[i])#O(w)
        img[i]=enlever(img[i],m)#O(w)
```

Complexité en $O(wh)$

[34]:
```
#Q10
def itere_reduction_ligne_par_ligne(img:list, n:int)->None:
    for i in range(n):
        reduction_ligne_par_ligne(img)
```

Q10b

Complexité en $O(nwh)$

[59]:
```
flam = (mpimg.imread('./flamantsNB.jpg')).tolist()
```

Q10 c

Les pixels enlevés peuvent être très éloignés entre une ligne et la suivante, créant des décalages de parties de lignes et donc des distorsions importantes. En d'autres termes, on enlève un pixel sur une ligne sans soucis de cohérence avec le pixel enlevé sur la ligne voisine. Cela peut briser des lignes verticales.

5 Réduction par colonne

[36]:
```
#fonction auxiliaire
def somme_energie(e,n,k):#somme energie col k
    r=0
    for i in range(n):#pour toute les lignes
        r+=e[i][k]
    return r

#Q11
def meilleure_colonne(e:list)->int:
    w,h=dim(e)
    r = somme_energie(e,h,0)#somme energie col 0
    indice = 0
    for i in range(1,w):#pour chaque colonne
        c = somme_energie(e,h,i)
        if c<r:
            indice = i
            r=c
    return indice
```

On inspecte chaque colonne et dans cette colonne chaque pixel une fois (dans le calcul de somme énergie). La somme des énergies par colonne chaque traitement se fait en $O(h)$.

Au total $O(wh)$ car w colonnes.

[38]:
```python
#Q12
def reduction_meilleure_colonne(img:list)->None:
    w,h=dim(img)
    e=energie(img)#O(w*h)
    k = meilleure_colonne(e)#O(wh)
    for i in range(h):# h passages
        img[i]=enlever(img[i],k)#O(w)

#Q13
def itere_reduction_meilleure_colonne(img:list, n:int)->None:
    for i in range(n):
        reduction_meilleure_colonne(img)
```

Complexités

`reduction_meilleure_colonne` est en $O(wh)$ et `itere_reduction_meilleure_colonne` l'appelle n fois. $O(nwh)$

`Q14`

Dans le deux premières images, certaines colonnes (celles de petites énergies) croisent peu de détails intéressants. Ce n'est pas gênant de les supprimer. D'autres colonnes (de haute énergie) croisent un détail intéressant (comme un cheval ou un flamand) : elles sont conservées.

Dans la photo de rue, où les détails sont nombreux, les colonnes (y compris celles de plus basses énergie) croisent toutes un détail intéressant. Notre problème vient du fait qu'on enlève une colonne dans sa globalité, au lieu de n'enlever que les pixels les moins intéressant.

6 Seam carving

`Q15`

$$e = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 3 & 3 & 2 & 2 \\ 2 & 0 & 1 & 2 \end{bmatrix}$$

- la ligne L_0 (celle du haut) est $2, 1, 1, 0$ sans changement
- pour les meilleurs chemins finissant ligne 1, on calcule la somme du coeff courant avec le meilleur chemin d'une des 2 ou 3 cases accessibles à la ligne du dessus. On obtient $4, 4, 2, 2$
- Pour la ligne 2, on agit de même et on obtient : $6, 2, 3, 4$

Au final

$$E = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 4 & 2 & 2 \\ 6 & 2 & 3 & 4 \end{bmatrix}$$

Un chemin d'énergie minimal est `[(0, 3), (1, 2), (2, 1)]`

Méthode de calcul de E `Q16`

Soit e de dimension (w, h)

- Pour une case de la ligne du haut, le meilleur chemin est la case elle-même
- Pour une ligne qui n'est pas celle du haut :
  - pour chaque case, on calcule la somme du coeff courant avec le minimum des 2 ou 3 cases accessibles à la ligne du dessus.
- on passe à la ligne du dessous, s'il y en a une.

Complexité : pour chaque pixel, on accède aux (au plus) 3 coefficients accessibles à la ligne du dessus et on fait quelques additions et comparaisons en $O(1)$. Donc complexité en $O(wh)$

trouver un chemin d'énergie minimale `Q17`

A partir de E :

On se place sur la dernière ligne à une case d'énergie minimale.

Tant qu'on n'est pas arrivé en haut, depuis la case courante on se rend sur celle des 2 ou 3 cases accessibles à la ligne au dessus qui a la plus petite énergie.

Le chemin cherché est la liste des cases parcourues.

Complexité

Un chemin est de longueur h. Pour chaque case de ce chemin, on accède aux (au plus) 3 coefficients de la ligne du dessus et on fait quelques additions et comparaisons en $O(1)$. Bref, complexité en $O(h)$ pour la boucle. Mais on doit ajouter la recherche de minimum en dernière ligne $O(w)$. Conclusion : $O(w + h)$.

[61]:
```python
#Q18
#fonction auxiliaire : elles donnent les indices de colonnes voisines
# de la colonne j
def accessible(j,w):#O(1)
        #colonnes accessibles sur la ligne au dessus.
        if j==0: return 0,1
        elif j==w-1: return j-1,j
        else : return j-1,j,j+1


def ajouter_ligne(e:list,E:list)->list:
    i = len(E)
    assert(i>0)
    w,h = dim(e)
    res = []
    for j in range(w):
        best = min(E[i-1][k] for k in accessible(j,w))
        res.append(best + e[i][j])
    return res
```

Pour chaque pixel de la ligne, on accède à ses 3 voisins du dessus et on calcule un min. Donc $O(1)$ pour chaque pixel de la ligne reproduit w fois : $O(w)$

[62]:
```
e = [[1,1,0,3],[4,1,2,4],[1,2,2,1],[4,1,1,0]]
E= [[1,1,0,3]]
ajouter_ligne(e,E)
```

[62]:
```
[5, 1, 2, 4]
```

[63]:
```
#Q19
def construire_E(e:list)->list:
    E = [list(e[0])]
    w,h=dim(e)
    for i in range(1,h):
        E.append(ajouter_ligne(e,E))
    return E
```

On appelle h fois `ajouter_ligne` en $O(w)$.

Donc $O(wh)$

[64]:
```
E=construire_E(e)
for l in E:
    print(l)
```

```
[1, 1, 0, 3]
[5, 1, 2, 4]
[2, 3, 3, 3]
[6, 3, 4, 3]
```

[65]:
```
#Q20
def chemin_minimal(E:list)->list:
    w,h=dim(E)
    i = h-1
    p=[]#path
    b = indice_min(E[i])#best
    p.append((i,b))
    while i>0:
        j = p[-1][1]
        l = [(E[i-1][k],k) for k in accessible(j,w)]
        b = l[indice_min(l)][1]
        i-=1
        p.append((i,b))
    return p[::-1]
```

On part du pixel d'energie min dans la ligne du bas, cela coûte $O(w)$ pour le trouver. Puis on remonte jusqu'au bord haut, en choisissant le pixel du dessus parmi les 3 voisins du pixel d'en dessous.

On obtient donc un chemin de h pixels dont chacun a un coût de construction en $O(1)$.

Complexité $O(h+w)$

[66]:
```
chemin_minimal(E)
```

[66]:
```
[(0, 2), (1, 1), (2, 0), (3, 1)]
```

[138]:
```
#Q21
def reduction_meilleure_energie(img:list)->None:
    w,h=dim(img)
    e=energie(img)#O(w*h)
    E = construire_E(e)# O(wh)
    p = chemin_minimal(E)#O(h)
    for i in range(h):# h passages
        img[i]=enlever(img[i],p[i][1])#O(w)
```

Complexité en $O(wh)$

[139]:
```
#Q22
def itere_reduction_meilleure_energie(img:list, n:int)->None:
    for i in range(n):#n passages
        reduction_meilleure_energie(img)#O(wh)
```

Complexité en $O(nwh)$

Conclusion

La construction de la matrice des énergies minimales puis celle du chemin min sont des algorithmes **gloutons** (ils font le meilleur choix à chaque instant) optimaux : ils fournissent la meilleure réponse possible.

D'une certaine manière c'est aussi de la **programmation dynamique** puisque :

Considérant le pixel p d'énergie minimale dans la ligne du bas, on a un problème : construire un chemin min qui part de la ligne du haut jusqu'à p.

On sait le résoudre si on sait tracer un chemin min qui va de la ligne du haut jusqu'à un des prédecesseurs de p, bref si on sait résoudre un sous-problème pour trouver un chemin moins long.