Méthodes d'optimisation

Problème du rendu de monnaie

La société Sharp commercialise des caisses automatiques utilisées par exemple dans des boulangeries. Le client glisse directement les billets ou les pièces dans la machine qui se charge de rendre automatiquement la monnaie.

Objectifs : Afin de satisfaire les clients, on cherche à déterminer un algorithme qui va permettre de rendre le moins de monnaie possible.

La machine dispose de billets de 20€, 10€et 5€ ainsi que des pièces de 2€, 1€, 50, 20, 10, 5, 2 et 1 centimes. On se propose donc de concevoir un algorithme qui demande à l'utilisateur du programme la somme totale à payer ainsi que le montant donné par l'acheteur. L'algorithme doit alors afficher quels sont les billets et les pièces à rendre par le vendeur.
L'approche "gloutonne" de cet algorithme revient à considérer les pièces une par une, en commençant par les pièces de plus grande valeur, et de décroître le montant à rendre (en enlevant la valeur de la pièce que l'on considère à ce moment) jusqu'à ce que le montant soit inférieur à la valeur de la pièce que l'on regarde. On considère alors la plus grande valeur de pièce inférieure au montant, et on itère le raisonnement jusqu'à ce que le montant atteigne zéro.
Par exemple, ma machine veut rendre 1 euro 22 centimes.
Si nous avons à notre disposition des pièces de : 1 euro, 50 centimes, 10 centimes, 2 centimes , 1 centimes. Le principe de notre algorithme est le suivant :

- ✗ On regarde la pièce de 1 euro. Alors on peut enlever une pièce de 1 euro au montant : on obtient 22 centimes à rendre. Or 22 centimes est inférieur à 1 euro et à 50 centimes, donc on passe à la pièce de 10 centimes.
- ✗ On regarde la pièce de 10 centimes. Alors on peut enlever deux fois la pièce de 10 centimes pour obtenir 2 centimes à rendre (inférieur à 10 centimes et à 5 centimes).
- ✗ On regarde la pièce de 2 centimes. Il suffit d'enlever une pièce de 2 centimes pour rendre le montant total.
- ✗ Au final, l'algorithme retourne "rendre 1 pièce de 1 euro, 2 pièces de 10 centimes, 1 pièce de 2 centimes".

1. Pour un montant d'achat donné et pour une somme donnée par le client, **proposer** un algorithme en pseudo code permettant de rendre le minimum de monnaie au client. Cet algorithme devra détailler la somme à rendre (nombre de pièces et nombre de billets).

Méthode itérative

```
somme : float
pieces, rendu_monnaie : liste de float
indice_piece : int
pieces = [10, 5, 2, 1, 0,5, 0,2, 0,1]
indice_piece = 0
tant que somme > 0 faire :
    tant que piece[indice_piece]>somme faire:
        indice_piece = indice_piece+1
    rendu_monnaie = rendu_monnaie + [piece[indice_piece]]
    somme = somme - [piece[indice_piece]]
```

2. **Définir** la structure de donnée possible à utiliser pour gérer les valeurs des billets ou des pièces

Les pièces / billets sont définis dans une liste de floattants

3. **Déterminer** le type de variable à d'utiliser pour décompter l'argent. **Justifier** la réponse.

   Le décompte d'argent est effectué avec un float (réel pour les centimes)

4. **Implémenter** cet algorithme dans Python. **Vérifier** sur plusieurs cas que l'algorithme fonctionne

Tri fusion

Nous avons vu en classe de première deux méthodes de tri qui ont un coût quadratique($O(n^2)$) :

- ➔ Le tri par insertion
- ➔ Le tri par sélection

Nous allons voir maintenant un autre tri, le **tri fusion**. L'idée du tri par fusion repose sur la méthode « diviser pour régner ». Ainsi le problème est découpé en deux sous problèmes (une liste découpée en 2 sous listes), puis chaque sous problème est traité séparément puis les résultats sont fusionnés de manière intelligente.

5. **Regarder** la vidéo d'introduction du principe de l'algorithme
   https://www.youtube.com/watch?v=OEmlVnH3aUg

6. **Proposer** un script Python qui permet de découper une liste en deux parties équivalentes :

➔ avec des slice

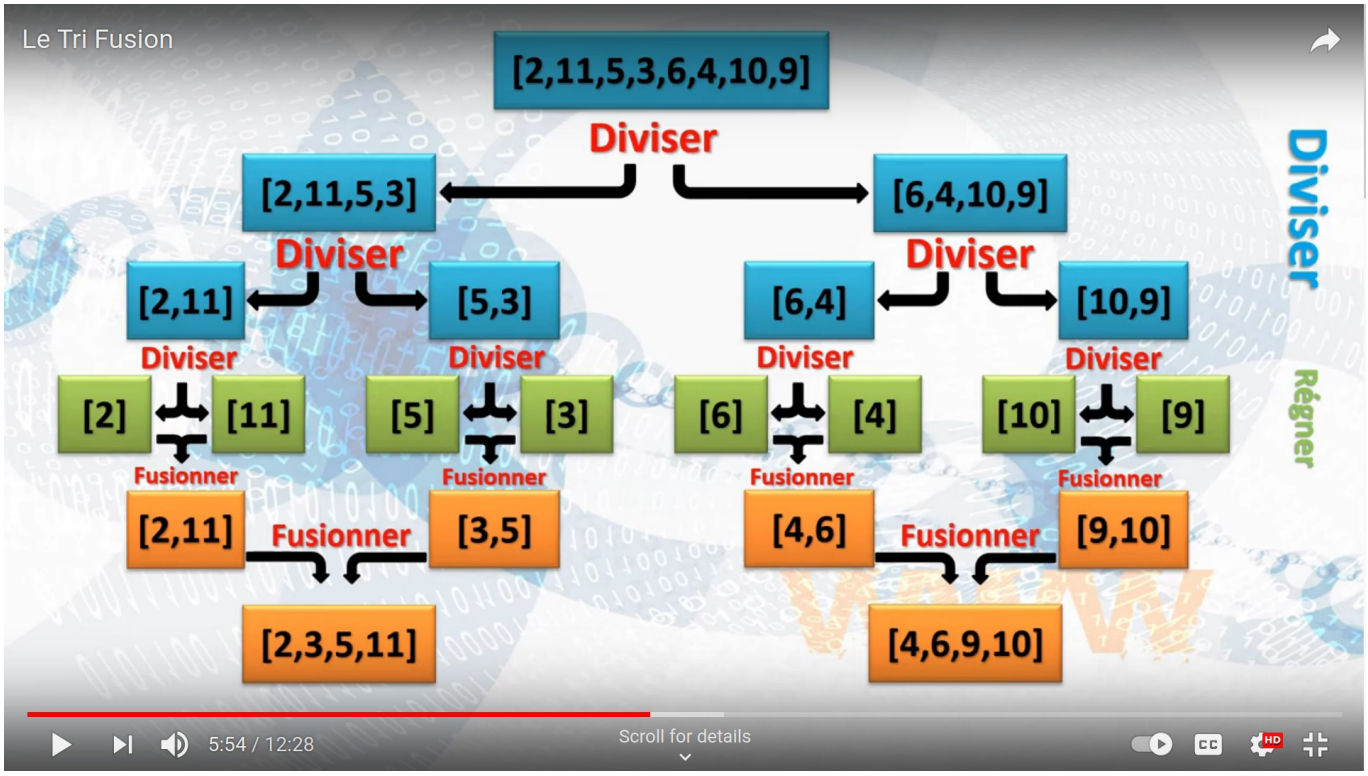
```
def decoupe(l1):
    '''Decoupe une liste l1 en deux parties egales avec slices '''
    milieu = len(l1)//2
    return l1[:milieu], l1[milieu:]
```

➔ sans slice sans compréhension de liste

```
def decoupe2(l1) :
    '''Decoupe une liste l1 en deux parties egales   par itération '''
    milieu = len(l1)//2
    l1a = []
    l1b = []
    for i in range(milieu) :
        l1a.append(l1[i])
    for i in range(milieu, len(l1)) :
        l1b.append(l1[i])
    return l1a, l1b
```

➔ sans slice avec compréhension de liste

```
def decoupe3(l1) :
    '''Decoupe l1 en deux parties egales par comprehension de liste '''
    milieu = len(l1)//2
    return [l1[i] for i in range(milieu)], [l1[i] for i in range(milieu,
len(l1))]
```



7. **Ecrire** une fonction `fusion(gauche,droite)` qui prend en argument deux listes triées gauche et `droite` et qui renvoie une liste triée correspondant à la fusion de gauche et `droite`.

```
def fusion(l1, l2) :
    '''Fusionne 2 listes triees l1 et l2'''
    l = []
    while len(l1)>0 and len(l2)>0 :
        if l1[0] > l2[0] : l.append(l2.pop(0))
        else : l.append(l1.pop(0))
    return l + l1 + l2
```

8. **Ecrire** une fonction `tri_fusion(l)` qui prend en argument une liste `l` et qui renvoie la liste `l` triée avec la méthode de tri par fusion.

```
def tri_fusion(l) :
    '''Trie la liste l par la methode tri par fusion'''
    l1, l2 = decoupe(l)
    if len(l1) > 1 : l1 = tri_fusion(l1)
    if len(l2) > 1 : l2 = tri_fusion(l2)
    return fusion(l1,l2)
```

Rotation d'une image

Dans ce problème, on considère une image « bitmap » implémentée par un tableau de tableau, les éléments du tableau étant des entiers indiquant la couleur de l'élément. Pour éviter les complications inutiles, on considère que l'image a autant de pixels en largeur qu'en hauteur et de plus, on considère que cette dimension commune est une puissance de 2. Par exemple, voici une image 256 x 256 (figure 1)

Figure 1: Image test

9. **Calculer** le nombre de pixels que comporte l'image

On aimerait faire tourner cette image d'un quart de tour dans le sens des aiguilles d'une montre. L'algorithme proposé fonctionne ainsi :

- ➔ Découper l'image 256 x 256 pixels en 4 images de 128×128 pixels,
- ➔ effectuer une permutation circulaire dans le sens horaire des quatre quadrants
- ➔ appliquer récursivement le même processus à chaque quadrant, jusqu'à obtenir des quadrants contenant un seul pixel.

Le principe de découpage et rotation est décrit sur la figure 2

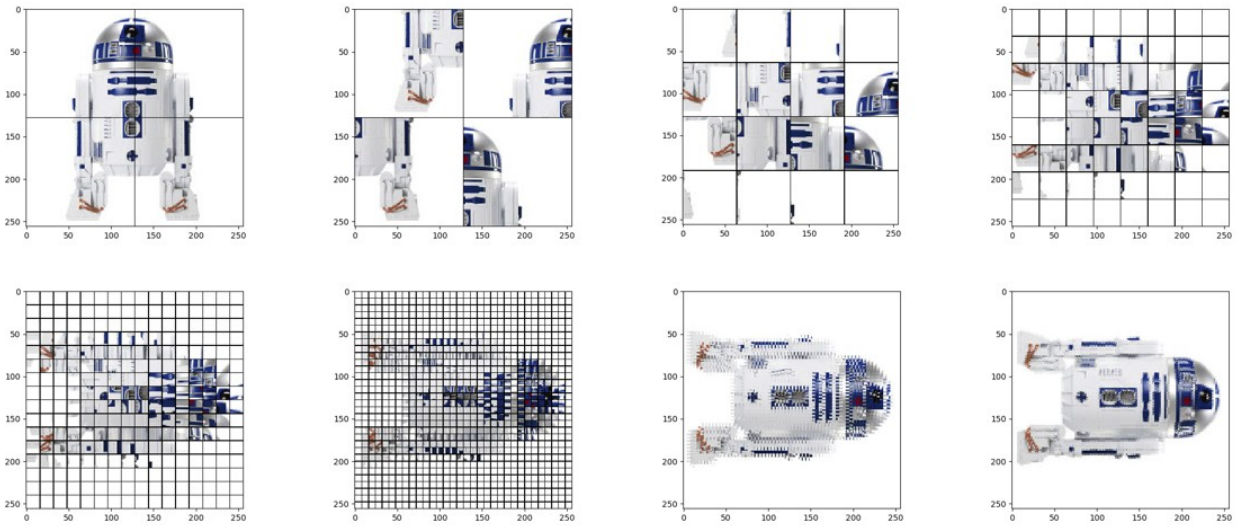


Figure 2: Détail du découpage et rotation récursif

L'ouverture, le chargement et l'affichage de cette image dans Python peut être effectué à l'aide des instructions suivantes :

```
from matplotlib.pyplot import imread, imshow, show
image = imread('r2d2.jpg')
imshow(image, cmap=('gray'))
show()
```

Pour évaluer la complexité de cet algorithme, on ne compte que les décalages. Par exemple, lors des premiers appels récursifs, on a décalé 4 images 128x128, ce qui représente 4 décalages. Pour tourner les images 128x128, on effectue 4 décalages d'images 128×128 pour chaque image 256×256, soit 4×4=16 décalages d'images 128×128 en tout.

10. **Calculer** le nombre de décalages effectué au total par cet algorithme récursif pour tourner une image 256x256

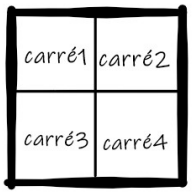
256x256 → 128x128 → 64x64 → 32x32 → 16x16 → 8x8 → 4x4 → 2x2 → 1x1

N = 4x4x4x4x4x4x4x4 = 4^8 = 65 536

Avec cette méthode il faut effectuer 65536 décalages.

11. **Comparer** le coût en temps de cet algorithme récursif par rapport celui d'un l'algorithme consistant à déplacer chaque pixel après avoir calculé ses nouvelles coordonnées.

Cette rotation est effectuée en 8 déplacements de l'ensemble des pixels alors qu'avec une méthode calculée la rotation est effectuée en 1 déplacement. La méthode diviser pour régner n'est donc pas la plus efficace dans ce cas.



Implémentation de l'algorithme de rotation d'image d'un quart de tour

12. **Ecrire** une fonction `echangePixel(image,x1,y1,x2,y2)` qui échange la valeur des pixels de coordonnées `(x1,y1)` et `(x2,y2)` dans l'image nommée `image`

```
def echangePixel(image, x1,y1,x2,y2) :
    image[y1][x1],image[y2][x2] = image[y2][x2],image[y1][x1]
```

13. **Ecrire** une fonction `echangeCarre(image,x1,y1,x2,y2,n)` qui échange des parties de l'image carrées de taille `n` dont les coordonnées de leur premier pixel est `(x1,y1)` et `(x2,y2)`. On utilisera la fonction `echangePixel`.

```
def echangeCarre(image,x1,y1,x2,y2,n) :
    for col in range(n) :
        for ligne in range(n) :
            echangePixel(image, x1+ligne, y1+col, x2+ligne, y2+col)
```

La permutation circulaire des carrés est effectuée de la façon décrite sur la figure 3.

Vous disposez d'une fonction `echange(Carré_a,Carré_b)` qui permet d'échanger deux carrés dans une image.

14. **Ecrire** la procédure qui permet de réaliser l'échange circulaire des 4 carrés suivants :

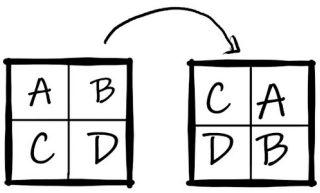
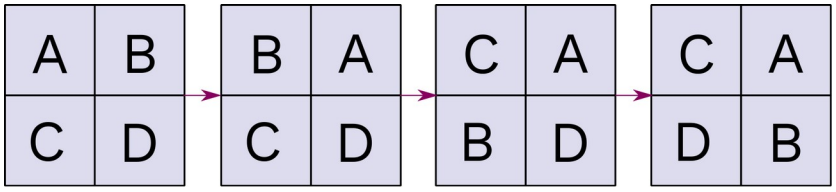


Figure 3: Ordre des permutations

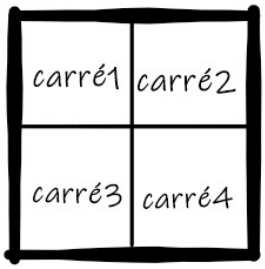
```
echange(carre_a, carre_b)
echange(carre_b, carre_c)
echange(carre_b, carre_d)
```



L'image est de taille m×m. Le premier pixel de carré1 a pour coordonnées (x,y). Les quatre carrés ont la même dimension.

15. **Donner** les coordonnées du premier pixel des `carré2`, `carré3` et `carré4` en fonction de `x,y` et `m`.

Carré	abscisse	ordonnée
Carré 1	x	y
Carré 2	x+m//2	y
Carré 3	x	y+m//2
Carré 4	x+m//2	y+m//2



16. **Ecrire** la procédure récursive `tourneCarre(image,x,y,n)` tourne d'un quart en sens horaire le carré de l'image de taille `n` dont le premier pixel a pour coordonnées `(x,y)`.

```
def tourneCarre(image,x,y,n) :
    if n>1 :
        echangeCarre(image, x, y, x+n//2, y, n//2)
        tourneCarre(image, x+n//2, y, n//2)
        echangeCarre(image, x, y, x, y+n//2, n//2)
        tourneCarre(image, x, y, n//2)
        echangeCarre(image, x, y+n//2, x+n//2, y+n//2, n//2)
        tourneCarre(image, x, y+n//2, n//2)
        tourneCarre(image, x+n//2, y+n//2, n//2)
```

17. **Ecrire** la fonction `rotation(image,n)` qui réalise la rotation de image de taille n d'un quart de tour.
18. **Tester** cet algorithme avec l'image `R2d2.png` fournie dans le dossier ressource.