Chapter 1

Principe diviser pour régner

Le principe algorithmique dit diviser pour régner consiste à:

1. **diviser** l'instance d'un problème en sous-instances,
2. **traiter** indépendamment ces sous-instances,
3. **combiner** les différents résultats obtenus pour construire une solution au problème initial.

Les algorithmes qui s'appuient sur ce principe sont souvent récursifs et opèrent sur des instances de plus en plus petites en passant d'entrées de taille n à $\frac{n}{2}$ ou une fraction de n. La taille de l'instance à traiter finit par rendre la résolution du problème simple voire triviale. On peut dans ce cas considérer la méthode diviser pour régner comme un cas particulier de la récursivité où on ne se contente pas de réduire la taille du problème d'une unité.

Un exemple classique de problème que nous avons déjà étudié et dont la résolution s'appuie sur ce principe algorithmique est **la recherche dichotomique**. En effet, la partition divise la zone de recherche en deux parts de tailles égales et on se concentre uniquement sur la bonne moitié en reproduisant ce mécanisme sur une nouvelle moitié et ainsi de suite de manière à ramener la zone de recherche à une unité. Notons que dans ce cas, la résolution d'un des deux sous-problèmes se résume à ne rien faire.

Beaucoup d'algorithmes reposant sur ce principe sont **récursifs**, en effet si les sous-problèmes ne sont qu'une "réduction" du problème initial, il est tentant de leur appliquer à nouveau le principe et ainsi de suite.

Figure 1.1: Illustration David Revoy CC BY

1.1 Recherche dichotomique

L'algorithme vu en première est bien du type **diviser pour régner**. Adaptez-le avec une fonction récursive:

```
def recherche(tab, val, g, d):
    """
    :param: tab (list): liste triée
    renvoie une position de val dans tab[g...d]
    et -1 si elle ne s'y trouve pas.
    """
    if g>d:
        return ..........
    m = (g+d)//2
    if tab[m]<val:
        return recherche(..........)
    elif:
        return recherche(..........)
    else:
        return ..........
```

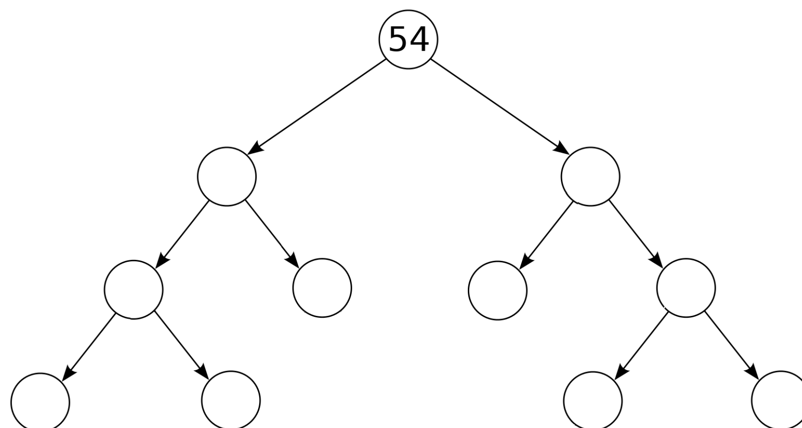
Sa complexité reste en $\Theta(log_2(n))$. De plus la quantité $d-g$ est un variant de notre fonction récursive, ce qui assure la **terminaison**.

1. Ecrire un script permettant de lancer cette fonction avec la liste:

$$L = [32, 1, 89, 78, 87, 90, 54, 7, 46, 91].$$

2. **Définition**: Un arbre est un arbre binaire de recherche (ABR) si pour tout sommet p, la valeur de p est strictement plus grande que les valeurs figurant dans son sous-arbre gauche et strictement plus petite que les valeurs figurant dans son sous-arbre droit.

   Compléter l'arbre binaire de recherche correspondant à une **recherche dichotomique** sur la liste [1, 7, 32, 46, 49, 54, 78, 87, 89, 90, 91], c'est à dire en y plaçant *tous* les milieux possbiles successifs de chacune des listes. La première valeur milieu 54 a été placée. Vous pouvez utiliser le symbole Δ ou écrire `NILL` pour indiquer qu'il n'y a pas de valeur sur le sommet. L'arbre peut être modifié.



   En passant par les milieux, quelle que soit la valeur recherchée, on obtient le résultat avec au maximum 4 tests.

CC-BY-SA

1.2 Rendu de monnaie

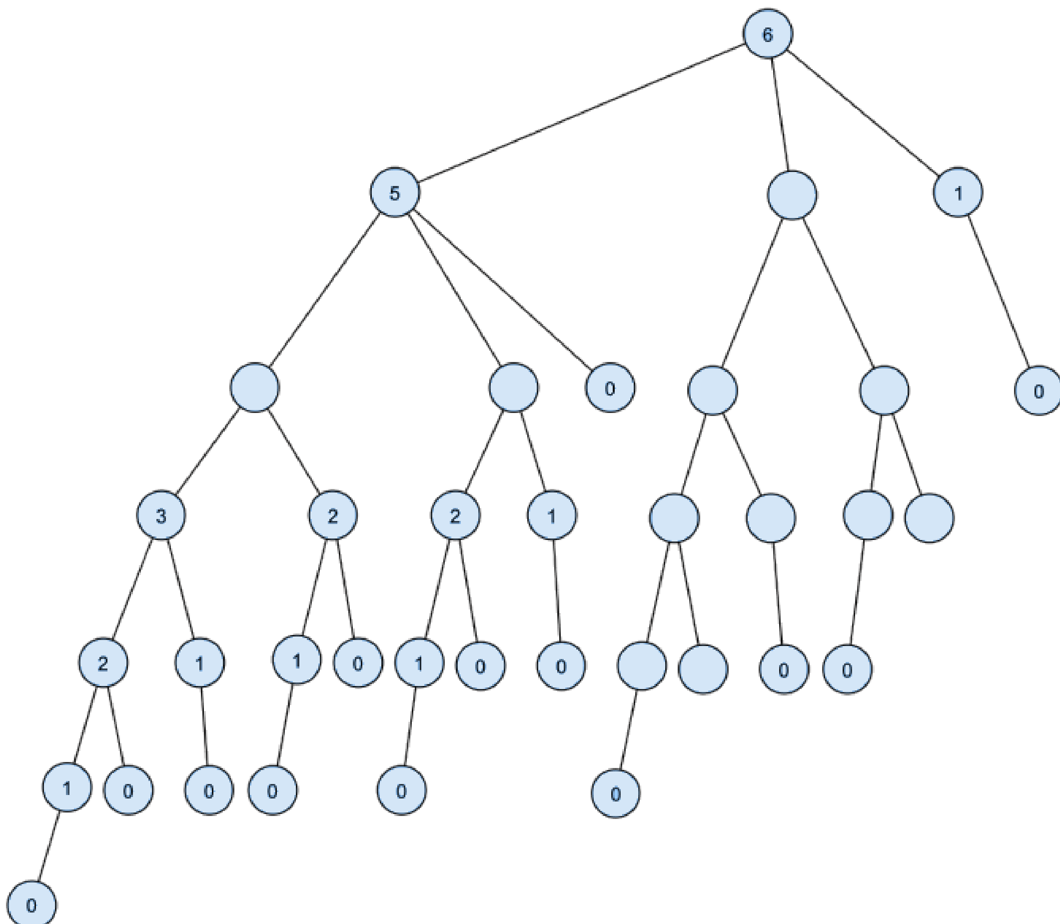
On peut utiliser ce paradigme pour écrire un algorithme de rendu de monnaie. Ce problème a déjà été résolu par méthode gloutonne, mais nous n'obtenions pas la solution optimale à chaque fois. Rappelons qu'on dispose d'un système de pièce `s`, et d'un montant à rendre `m`.

De façon récursive :

- Si le montant `m` est égal à zéro, on renvoie zéro.
- Sinon on effectue un appel récursif avec un montant `m` dimininué de la valeur de la pièce choisie. Vous pourrez fixer un minimum arbitrairement grand.

Écrire un algorithme `rendu_monnaie` prenant en paramètre un système de pièce `s` (`list`) et un montant `m` (`int`) avec le paradigme *diviser régner*.

```
def rendu_monnaie(m, s):
    if m == 0:
        ...........
    else:
        #minimum choisi arbitrairement grand
        minimum = ...........
        for piece in s:
            if piece <= m:
                #nb : nombre de pièce utilisé.
                nb = 1 + ...........
                #changer le minimum si plus optimal.
                if nb < minimum:
                    ...........
        return minimum
```



Nous reviendrons sur l'efficacité de cet algorithme dans le chapitre *programmation dynamique*. Compléter sur la *figure 1.2* l'arbre des appels récursifs avec s = (1, 2, 5, 10) et m = 6 :

1.3 Tri fusion

Le principe de cet algorithme repose lui aussi sur le principe *diviser pour régner*. Voici donc l'idée de l'algorithme du tri fusion :

1. **DIVISER :** Découper la liste à trier en deux listes d'égales longueurs (à un élément près).
2. **RÉGNER :** Trier chacune des deux listes obtenues récursivement, cas d'arrêt lorsque les listes sont de longueur 1.
3. **COMBINER :** Fusionner les deux listes triées pour former la liste voulue.

Ce qui se résume dans le tableau suivant :

Exemple 1 Prenons la liste $\ell = [32, 1, 89, 78, 87, 90, 54, 7, 46, 91]$.

1. Découpons la liste en deux. Il y a de nombreuse façons de faire cela. En voici une, qui consiste à mettre un élément sur deux dans une liste et les autres dans l'autre :

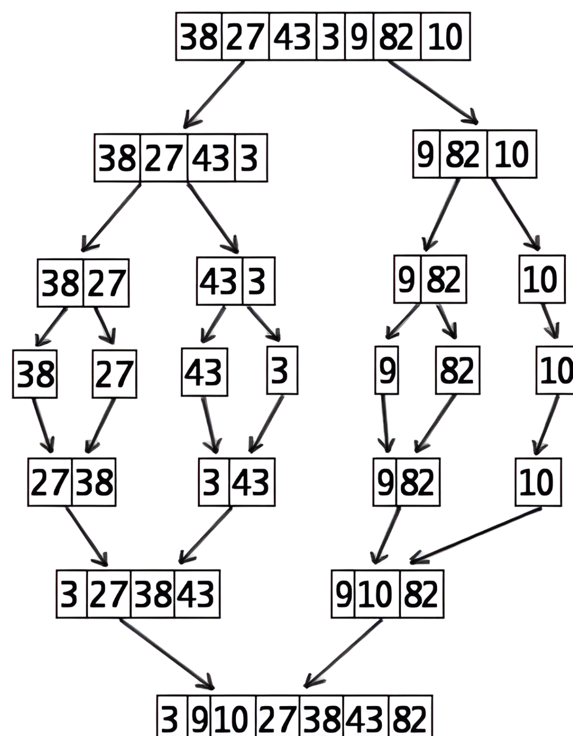
$$\ell_1 = [32,\ 1,\ 89,\ 78,\ 87] \text{ et } \ell_2 = [90,\ 54,\ 7,\ 46,\ 91].$$

2. Trions chacune de ces deux listes, et nous obtenons :

$$\ell_1 = [1, 32, 78, 87, 89] \text{ et } \ell_2 = [7, 46, 54, 90, 91].$$

3. Fusionnons ces deux listes triées tout en conservant l'ordre des éléments et nous obtenons la liste triée : [1, 7, 32, 46, 54, 78, 87, 89, 90, 91].

Exemple 2 Étapes du tri fusion de la liste [38, 27, 43, 3, 9, 82, 10] :



Source : Wikipédia

CC-BY-SA

1.3.1 Mise en oeuvre

Pour cette mise en oeuvre, nous couperons en deux la liste L en utilisant l'indice `len(L)//2`. L'algorithme de tri fusion utilise deux fonctions `moitie_gauche()` et `moitie_droite()` qui prennent en argument une liste L et renvoient respectivement :

- la sous-liste de L formée des éléments d'indice strictement inférieur à `len(L)//2` ;
- la sous-liste de L formée des éléments d'indice supérieur ou égal à `len(L)//2`.

Exemple

```
>>> L = [3, 5, 2, 7, 1, 9, 0]
>>> moitie_gauche(L)
[3, 5, 2]
>>> moitie_droite(L)
[7, 1, 9, 0]
>>> M = [4, 1, 11, 7]
>>> moitie_gauche(M)
[4, 1]
>>> moitie_droite(M)
[11, 7]
```

L'algorithme utilise aussi une fonction `fusion()` qui prend en argument deux listes triées L1 et L2 et renvoie une liste L triée et composée des éléments de L1 et L2. On donne ci-dessous le code python d'une fonction récursive `tri_fusion()` qui prend en argument une liste L et renvoie une nouvelle liste triée formée des éléments de L.

```python
def tri_fusion(L):
  n = len(L)
  if n<=1 :
    return L
  print(L)
  mg = moitie_gauche(L)
  md = moitie_droite(L)
  L1 = tri_fusion(mg)
  L2 = tri_fusion(md)
  return fusion(L1, L2)
```

1. Donner la liste des affichages produits par l'appel suivant:

   ```
   >>>tri_fusion([7, 4, 2, 1, 8, 5, 6, 3])
   ```

2. On s'intéresse désormais à différentes fonctions appelées par `tri_fusion`, à savoir `moitie_droite` et `fusion`. Écrire la fonction `moitie_droite`.

CC-BY-SA

3. On donne ci-dessous une version incomplète de la fonction `fusion`.

```python

def fusion(L1, L2):
  L = []
  n1 = len(L1)
  n2 = len(L2)
  i1 = 0
  i2 = 0
  while i1 < n1 or i2 < n2 :
    if i1 >= n1:
      L.append(L2[i2])
      i2 = i2 + 1
    elif i2 >= n2:
      L.append(L1[i1])
      i1 = i1 + 1
    else:
      e1 = L1[i1]
      e2 = L2[i2]

  ...............
  ...............

  return L
```

Dans cette fonction, les entiers `i1` et `i2` représentent respectivement les indices des éléments des listes L1 et L2 que l'on souhaite comparer :

- Si aucun des deux indices n'est valide, la boucle `while` est interrompue ;
- Si `i1` n'est plus un indice valide, on ajoute à L les éléments de L2 à partir de l'indice `i2` ;
- Si `i2` n'est plus un indice valide, on ajoute à L les éléments de L1 à partir de l'indice `i1` ;
- Sinon, le plus petit élément non encore traité est ajouté à L et on décale l'indice correspondant.

Écrire les instructions manquantes des lignes 17 à 22 permettant d'insérer dans la liste L les éléments des listes L1 et L2 par ordre croissant.

Complexité Soit $T(n)$ la complexité de l'algorithme en fonction de n. La phase de division récursive a une complexité logarithmique, et la fusion des parties triées a une complexité linéaire. La relation de récurrence correspondante est :

$$T(n) = 2T(\frac{n}{2}) + n \tag{1.1}$$

on montre alors que :

$$T(n) = \Theta(n \log_2(n)) \tag{1.2}$$

1.4 Exercices

Exercice 1 Maximum On propose l'algorithme suivant pour la recherche du maximum des éléments d'une liste :

- Partager la liste en deux moitiés L_1 et L_2.
- Chercher les maximums m_1 de L_1 et m_2 de L_2.
- En déduire le maximum m de `L`.

1. Expliquer pourquoi cet algorithme fait partie de la méthode *diviser pour régner*.
2. Cet algorithme est-il récursif ? Justifier.
3. En utilisant la méthode **diviser pour régner** , implémenter une fonction `maximum(L)` renvoyant le maximum d'une liste `L`.

CC-BY-SA

Exercice 2 Tri rapide Le tri rapide s'appuie sur la méthode *diviser pour régner* en découpant la liste suivant des indices a et b. On choisit une valeur de la liste appelée pivot, souvent la dernière, et on partage le tableau en deux parties : les valeurs inférieures au pivot au début, les valeurs supérieures à la fin. Ce partage s'effectue en échangeant les valeurs.

On trie récursivement ces parties. La complexité moyenne du tri rapide en sélectionnant le dernier élément de la liste comme pivot est $O(n\log(n))$). (Wikipédia)

- Réalisez l'activité sur la forge de l'AEIF.

Figure 1.2: Illustration David Revoy CC-BY-SA

Exercice 3 Coefficients binomiaux Vous savez développer $(a+b)^2$, peut-être vous êtes vous déjà demandé s'il existait une formule pour développer $(a+b)^n$? Nous avons besoin pour cela des coefficients du binôme de Newton notés $\mathcal{C}_{k,n}$: c'est le nombre de combinaisons pour choisir k entiers parmi n. Ils sont définis par récurrence:

$$\mathcal{C}_{k,n} = \begin{cases} 1 & \text{si } k=0 \text{ } ou \text{ } k=n \\ \mathcal{C}_{k-1,n-1} + \mathcal{C}_{k,n-1} & \text{sinon.} \end{cases}$$

On peut utiliser la représentation appelée triangle de Pascal pour mémoriser la formule:

n\k	0	1	2	3	4	5	6				
0	1										
1	1	1									
2	1	2	1								
3	1	3	3	1							
4	1	4	6	4	1						
5	1	5	10	10	5	1					
6	1	6	15	20	15	6	1				
7	1	7	21	35	35	21	7	1			
8	1	8	28	56	70	56	28	8	1		
9	1	9	36	84	126	126	84	36	9	1	
10	1	10	45	120	210	252	210	120	45	10	1

Et $(a+b)^3 = \mathbf{1}\times a^3 + \mathbf{3}\times a^2b + \mathbf{3}\times ab^2 + \mathbf{1}\times b^3$

Ecrire une fonction récursive `coeff_bin(k,n)` retournant la valeur du coefficient $\mathcal{C}_{k,n}$.

1.5 TP:Quart de tour d'une image dans le sens horaire.

1.5.1 Le module PIL

On importe la classe `Image` du module PIL permettant de manipuler de façon simple les images. L'image utilisée est image.png .

1. Tester le code suivant:

```
from PIL import Image
im = Image.open("image.png")
largeur, hauteur = im.size
im.show()
```

2. Quelle est la couleur du pixel de coordonnée (50;80) ?
3. Remplacer ce pixel par un pixel de couleur RGB (100,25,200).

   Notez qu'une nouvelle image est créée par: `im2 = Image.new("RGB", (largeur,hauteur))`, elle est sauvegardée par `im2.save(nom_du_fichier)`.

1.5.2 Méthode naïve

On déplace les pixels de départ vers leur position d'arrivée en utilisant leurs coordonnées et la propriété suivante:

Propriété: Pour une image carrée de taille $n \times n$, le pixel de coordonnées (x, y) après rotation était initialement celui de coordonnées (y, n-1-x).

Ecrire un algorithme permettant de tourner une image d'un quart de tour dans le sens horaire où vous devrez:

- Ouvrir une image
- Récupérer sa largeur et sa hauteur
- Créer l'image vierge sur laquelle "imprimer"la rotation.
- Parcourir l'image pour "imprimer"les pixels.

1.5.3 Méthode Diviser pour régner

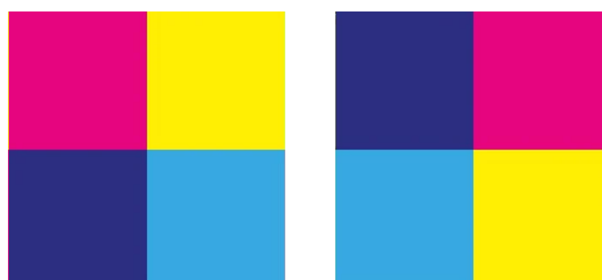
On veut maintenant réaliser un algorithme de quart de tour d'une image sans utiliser d'espace mémoire supplémentaire. L'image est carré et sa taille doit être une puissance de 2.

1.5.4 Principe

La méthode utilisée de type diviser pour régner consiste à:

- diviser l'image en quatre quadrants.
- tourner récursivement chaque quadrant.
- effectuer une permutation de ces quadrants.

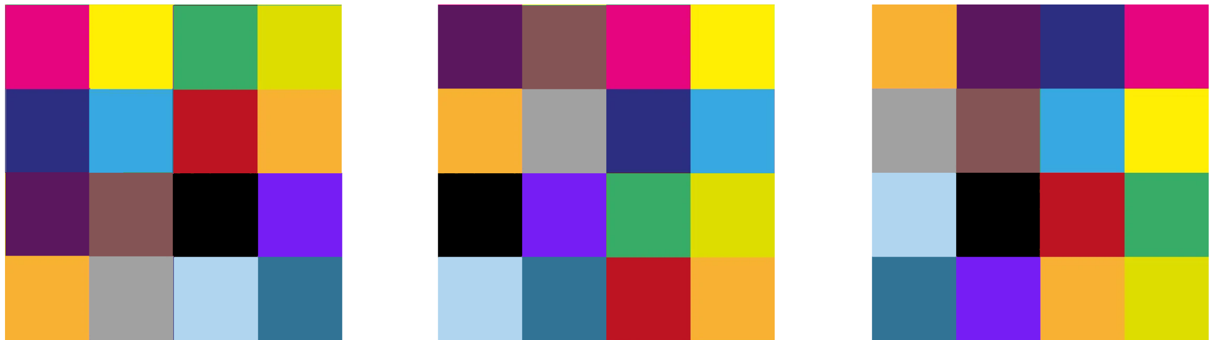
Illustration: Soit une image carrée constituée de 2×2 pixels. Si on fait glisser chaque pixel dans le sens des aiguilles d'une montre alors on fait une rotation d'un quart de tour de cette image.



CC-BY-SA

Cas le plus élémentaire: 2×2 pixels.

On peut alors exécuter cette opération sur une image carrée de 16px en faisant glisser de la même manière chaque quart d'image. Puis, en appliquant le glissement les 4 pixels constituant chaque quart d'image on retrouve l'image initiale avec une rotation d'un quart de tour



Cas 16×16: on utilise le cas précédent.

1.5.5 Echanger les quadrants

Nous réalisons l'écriture d'une procédure qui échange deux quadrants.

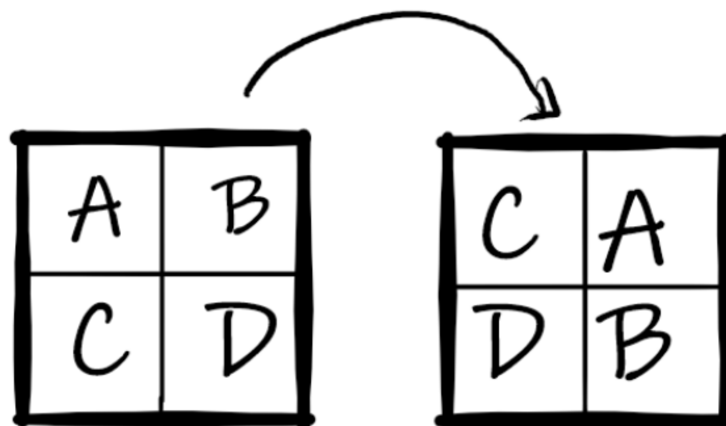
1. Ecrire une *procédure* `echange_pix(image, x0, y0, x1, x1)` qui échange les pixels de coordonnées (x0,y0) et (x1, y1) dans `image`.

2. Expliquer la *procédure* `echange_quadrant(image, x0, y0, x1, x1)` ci-dessous.

```
for i in range(n):
    for j in range(n):
        echange_pix(image, x0+i, y0+j, x1+i, y1+j)
```

3. Vous disposez d'une fonction `echange(Carre_a,Carre_b)` qui permet d'échanger deux carrés dans une image.

   Quelle est la procédure qui permet de réaliser l'échange ci-dessous (indiquez les échanges successifs) ?



A, B, C et D sont des carrés.

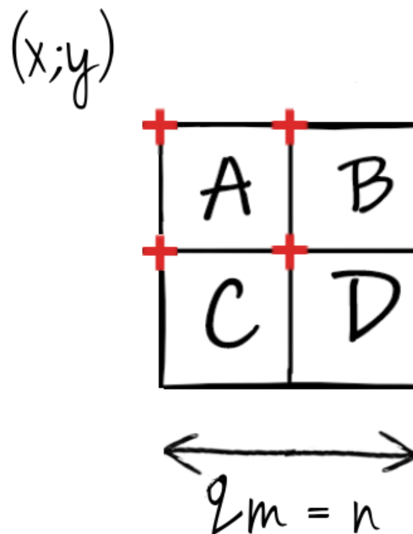
On peut diviser l'image en quatre, effectuer une permutation circulaire des quatre quadrants, puis appliquer récursivement le même processus à chaque quadrant, jusqu'à obtenir des quadrants contenant un seul pixel, il n'y a alors rien à faire.

CC-BY-SA

1.5.6 Fonction récursive de rotation

On peut donc effectuer récursivement une rotation d'un quart detour sur des images de toute taille. Notre image est divisée en quatre à chaque appel, il faut donc y effectuer l'échange de ces quadrants. La taille des quadrants dépend donc de la largeur n de l'image qui les définit. La question suivante a pour but de vous faire écrire les coordonnées des sommets pour chaque quadrants.

1. L'image est de taille m×m. Le premier pixel du carré A a pour coordonnées (x;y). Les quatre carrés ont la même dimension. Donner les coordonnées du premier pixel des carré B, carré C et carré D en fonction de x,y et m.



2. Ecrire la procédure récursive de rotation en complétant le script ci-dessous.

```
1  def rotation(image, x0, y0, n):
2  if n>=2:
3      m = ..........
4      rotation(image,x0, y0, m)
5      rotation(image,x0, y0.........., m)
6      rotation(image,x0.........., y0, m)
7      rotation(image,x0.........., y0.........., m)
8      echange_quadrant(image, x0, y0, x0+m, y0, m)
9      echange_quadrant(image, x0, y0, x0+m, y0+m m)
10     echange_quadrant(image, x0, y0, x0, y0+m m)
11
```

3. Tester avec l'image de vore choix.

1.6 Références

CC-BY-SA