DIVISER POUR REGNER

LA METHODE DIVISER POUR REGNER

En programmation, diviser pour régner est une méthode de conception d'algorithmes réduisant récursivement un problème en un ou plusieurs sous-problèmes du même type.

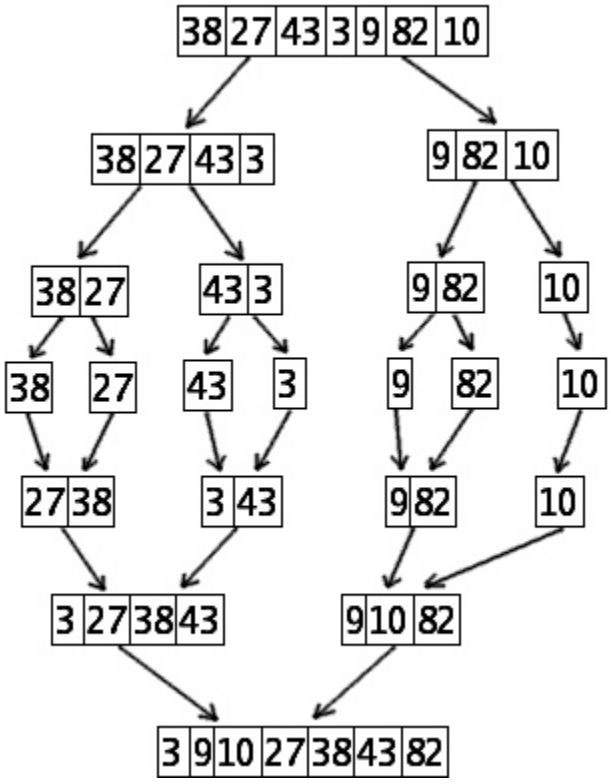
Cette méthode est utilisée notamment pour le **tri par dichotomie**, le **tri par fusion** et le **tri rapide** (quick sort).

Les **3 étapes de la méthode diviser pour régner** sont : **diviser** (découper), **régner** (résoudre les sous-problèmes), **combiner** (les solutions des sous problèmes pour résoudre le problème général).

PRINCIPE DU TRI PAR FUSION

Les trois étapes (diviser, régner, combiner) dans le tri par fusion d'un tableau.

DIVISER	REGNER	COMBINER
On découpe le tableau à trier T[1, .. n] en deux sous-tableaux T[1, .. n/2] et T[n/2 +1,..n]	**On trie les deux sous-tableaux** T[1, .. n/2] et T[n/2 +1,..n] (récursivement, ou on ne fait rien s'ils sont de taille 1)	**On fusionne** les deux sous-tableaux triés T[1, .. n/2] et T[n/2 +1,..n].



Exemple de tri par fusion appliqué à un tableau de sept éléments (source : Wikipédia)

D'où le programme récursif Python (en supposant programmée la fusion) :

```
def tri(t):
    n = len(t)
    if n < 2:
        return t
    else:
        m = n // 2
    return fusion(tri(t[:m]),tri(t[m:]))
```

La terminaison est justifiée par la décroissance stricte de n à chaque appel récursif.

La fusion se prête également à une programmation récursive :

```
def fusion(t1,t2):
    if t1 == []:
        return t2
    elif t2 == []:
        return t1
    elif t1[0] < t2[0]:
        return [t1[0]]+fusion(t1[1:],t2)
    else:
        return [t2[0]]+fusion(t1,t2[1:])
```

Ce programme est séduisant sur le papier mais il pose de gros problèmes de mémoire pour traiter de grand tableaux (en particulier sous Python).

Une version itérative est plus efficace :

```
def fusion(t1,t2):
    i1, i2, n1, n2 = 0, 0, len(t1), len(t2)
    t=[]
    while i1 < n1 and i2 < n2:
        if t1[i1] < t2[i2]:
            t.append(t1[i1])
            i1+=1
        else:
            t.append(t2[i2])
            i2+=1
    if i1 == n1:
        t.extend(t2[i2:])
    else:
        t.extend(t1[i1:])
    return t
```

On notera au passage quelques **bonnes pratiques** :

• utiliser les variables **`n1`**, **`n2`** plutôt que de recalculer sans cesse **`len(t1)`** et **`len(t2)`**

• utiliser **`i+=1`** plutôt que **`i=i+1`**

• utiliser **`t.append(x)`** plutôt que **`t=t+[x]`** (la première version ajoute x à la fin de t, tandis que la seconde recopie toutes les valeurs de t dans une nouvelle instance de t avant d'y adjoindre x. . .)

• de même, utiliser **`t.extend(u)`** plutôt que **`t=t+u`**.

ALGORITHME DE ROTATION D'IMAGE

Principe : soit une image carrée de constituée de 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.

On peut 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.

On peut ainsi, récursivement effectuer une rotation d'un quart de tour sur des images de toute taille.

```
def rotate(img, x, y, width):

    """x : ligne, y : colonne, width : largeur"""
    #si l'image ne contient qu'un pixel la rotation est finie
    if width < 2:
        return img

    width = width // 2
    #glissement des quatres quarts d'image
    for i in range(y, y + width):
        for j in range(x, x + width):
            pixtemp = img[i][j]
            img[i][j] = img[i + width][j]
            img[i + width][j] = img[i + width][j + width]
            img[i + width][j + width] = img[i][j + width]
            img[i][j + width] = pixtemp

    #appel récursif sur les quatres quarts d'image
    img = rotate(img, x, y, width)
    img = rotate(img, x + width, y, width)
    img = rotate(img, x + width, y + width, width)
    img = rotate(img, x, y + width, width)
    return img
```