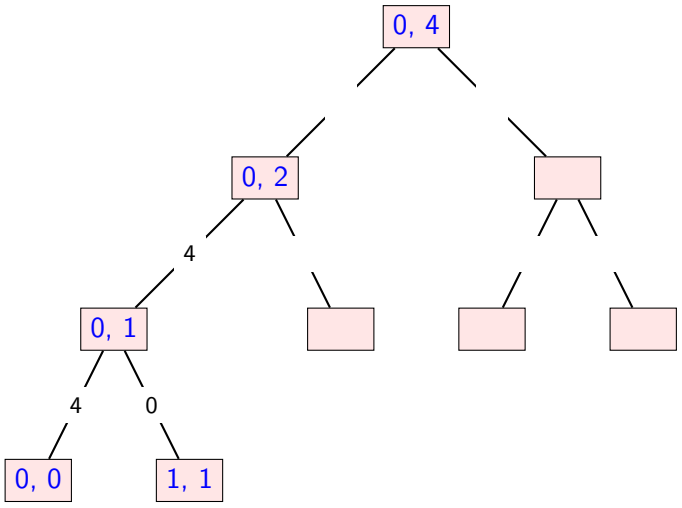
Méthode : « Diviser pour régner »

Exercices

Exercice 1

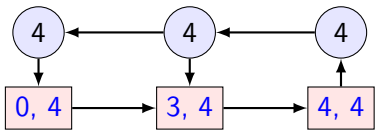
1. Programmer les fonctions `maximum_rec` et `maximum`.
2. Compléter l'arbre suivant décrivant les différents appels récursifs de la fonction `maximum_rec` lors de l'appel de la fonction `maximum` sur la liste `[4, 0, 1, 2, 3]`.

   Les valeurs indiquées dans les rectangles sont les différentes valeurs des variables `debut` et `fin`, et les valeurs indiquées sur les branches, sont les valeurs renvoyées par la fonction récursive.
3. Quel est le nombre d'appels récursifs de la fonction `maximum_rec` dans l'exemple précédent ?
4. Vérifier que si la taille de la liste est $8 = 2^3$ alors il y aura 14 appels récursifs.



Exercice 2

1. Programmer les fonctions `dichotomie_rec` et `dichotomie`.
2. Voici la pile des appels récursifs de la fonction `dichotomie_rec` lors de l'appel de la fonction `dichotomie` sur le tableau `[1, 3, 8, 9, 10]` pour la valeur `10`.



Les valeurs indiquées dans les rectangles sont les différentes valeurs des variables `g` et `d`, et les valeurs indiquées dans les cercles les valeurs renvoyées par la fonction récursive.

Écrire les piles des appels récursifs pour les valeurs `1` et `5`.

Exercice 3

```
class Maillon:
    """ Classe définissant un maillon
    d'une liste chaînée """

    def __init__(self, valeur, suivant):
        self.valeur = valeur
        self.suivant = suivant

class Liste:
    """ Liste chaînée """

    def __init__(self, liste = []):
        """ Constructeur """
        self.tete = None
        self.longueur = 0
        while liste != []:
            self.ajoute_tete(liste.pop())

    def est_vide(self):
        """ renvoie un booléen indiquant
        si la liste est vide """
        return self.tete is None

    def __len__(self):
        """ définit la méthode générique len
        qui retourne la longueur de la liste """

        return self.longueur
```

```
    def __repr__(self):
        """ affichage de la liste chaînée """

        affichage = []
        maillon = self.tete
        while maillon is not None:
            affichage.append(maillon.valeur)
            maillon = maillon.suivant
        return str(affichage)

    def ajoute_tete(self, valeur):
        """ ajoute l'élément valeur au début
        de la liste """

        self.longueur += 1
        maillon = Maillon(valeur, self.tete)
        self.tete = maillon

    def enleve_tete(self):
        """ enlève le maillon en tête de la liste et
        retourne la valeur contenue dans ce maillon
        """

        valeur = self.tete.valeur
        self.tete = self.tete.suivant
        self.longueur -= 1
        return valeur
```

1. En utilisant la classe `Liste` rappelée ci-dessus, implémenter la fonction `scinder_liste`. Vous testerez votre fonction sur la liste donnée en exemple dans le cours, d'une liste à un seul élément, puis dans le cas d'une liste à deux éléments.
2. Écrire cette fonction `fusion`, puis la tester dans les cas suivants :
   - une des deux listes est vide ;
   - deux listes à un élément ;
   - dans le cas des deux listes données ci-dessus ;
   - dans le cas des deux listes suivantes : `liste1 = [1, 5, 7, 9]` et `liste2 = [2, 3]`.
3. Implémenter la fonction `tri_fusion` et valider les tests pour la liste donnée en exemple dans le cours, la liste vide et un liste à un élément.

Exercice 4

Pour rappel, voici les différentes fonctions décrivant le tri par sélection et le tri par insertion :

```
def echange(t : list, i : int, j : int):
    """
    Echange deux éléments du tableau t
    situés aux indices i et j
    """
    temporaire = t[i]
    t[i] = t[j]
    t[j] = temporaire

def tri_par_selection(t : list):
    """
    Trie par ordre croissant le tableau t
    de taille n en utilisant le tri par sélection
    """
    n = len(t)
    for i in range(n):
        m = i
        for j in range(i+1, n):
            if t[j] < t[m]:
                m = j
        echange(t, i, m)
```

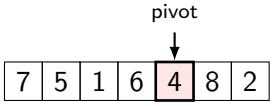
```
def insere(t : list, i : int):
    """
    Insère la valeur v = t[i] dans la partie du tableau
    t[0..i] en supposant t[0..i-1] déjà trié
    """
    j = i-1
    v = t[i]
    while t[j] > v and j >= 0:
        t[j+1] = t[j]
        t[j] = v
        j = j - 1
def tri_par_insertion(t : list):
    """
    Trie par ordre croissant le tableau t
    de taille n en utilisant le tri par sélection
    """
    n = len(t)
    for i in range(n):
        insere(t, i)
```

1. La fonction `shuffle` de la bibliothèque `random` mélange un tableau de manière pseudo- aléatoire. À l'aide de celle-ci, écrire une fonction `tableau_melange` retournant un tableau dynamique (ou liste Python) d'entiers, mélangé de taille `n`. (on pourra commencer par générer par compréhension un tableau contenant les entiers allant de `1` à `n`).
2. Réaliser une fonction similaire, `liste_melange` pour une liste chaînée.
3. La fonction `time` de la bibliothèque `time` donne, en secondes, le temps écoulé depuis le 1er janvier 1970. En l'appelant et en conservant la valeur retournée à deux instants d'un programme, elle permet de mesurer le temps écoulé durant ces deux instants. Écrire une fonction `mesure_temps_tri` prenant pour paramètres la fonction de tri `f`, la taille du `n` tableau(ou de la liste chaînée) à trier et revoyant le temps de calcul.
4. Vérifier l'ordre de grandeur des valeurs indiquée dans le tableau du cours à l'aide de cette fonction.

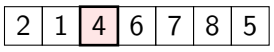
Exercice 5 — *Tri rapide*

Le tri rapide est créé en 1960 par Tony Hoare, et est également un algorithme utilisant la méthode diviser pour régner. Cette algorithme se décrit naturellement sur un tableau et le tri se fera en place :

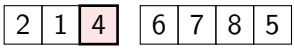
- on choisit un élément du tableau comme **pivot**, ce choix est aléatoire en « espérant » que ce soit le meilleurs ;



- on déplace les éléments plus grand que ce pivot à sa droite et les autres éléments à sa gauche :



- on recommence le processus sur chacun des sous tableaux situés à gauche et à droite du pivot (le premier contiendra le pivot) :



La partie la plus difficile décrire est le deuxième point, on l'appelle souvent le partitionnement ou le pivotage. On utilisera trois pointeurs : `p` l'indice du pivot, `d` l'indice du premier du tableau `T` (ou sous tableau) et `f` l'indice du dernier élément. Voici l'algorithme :

Définition *partionnement(T, p, d, f)*:
    pivot ← T[p]
    echange(T, p, f)
    f ← f - 1
    Tant que *d* < *f* **faire**
        Tant que *T[d]* <= *pivot et d* < *f* **faire**
            d ← d + 1
        FinTantque
        Tant que *T[f]* > *pivot et d* < *f* **faire**
            f ← f - 1
        FinTantque
        echange(T, d, f)
    FinTantque
    echange(T, p, f)
    renvoyer l'emplacement f du pivot
FinDéfinition

1. Appliquer cet algorithme « à la main », au tableau donné ci-dessus afin d'obtenir le partitionnement donné précédemment.
2. Implémenter cet algorithme en Python en programmant la fonction `partionnement`, puis valider quelques tests.
3. Écrire la fonction `tri_rapide` qui se déduit de cette description (la condition d'arrêt de la récursivité est `d >= f`, dans ce cas le tableau n'a qu'un élément ou est vide). La fonction ne renvoie rien : elle trie le tableau en place.
4. À l'aide de l'exercice précédent, comparer l'efficacité du tri rapide avec les autres algorithmes (on pourra changer la manière de choisir le pivot : le premier élément du tableau, ou bien la valeur du milieu du tableau).

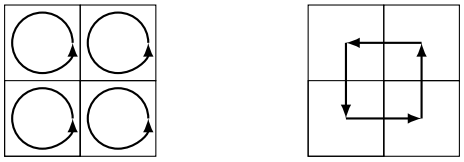
Exercice 6 — *Rotation d'une image*

On cherche à écrire une fonction qui effectue la rotation d'un image de 90 degrés en utilisant le principe « diviser pour régner ». Pour manipuler une image en Python, on peut utiliser la bibliothèque PIL (Python Image Library) et plus précisément son module `Image`. Avec les quatre lignes suivantes :

```
from PIL import Image
im = Image.open("lena.png")
largeur, hauteur = im.size
px = im.load()
```

on charge l'image contenu dans le fichier `lena.png`, la variable `px` est la matrice des pixels constituant l'image. Pour $0 \leqslant$ `x` $\leqslant$ `largeur` et $0 \leqslant$ `y` $\leqslant$ `hauteur`, la couleur du pixel `(x, y)` est donnée par `px[x,y]`. On peut modifier la couleur d'un pixel avec une affectation de la forme $px[x, y] = c$ où `c` est une couleur.

On supposera que l'image est carrée et que sa dimension est **une puissance de 2**, par exemple 512×512. L'idée consiste à découper l'image en quatre, à effectuer la rotation de 90 degrés de chacun des quatre morceaux, puis les déplacer vers leur position finale. On peut illustrer les deux étapes ainsi :



Afin de pouvoir procéder récursivement, on va définir une fonction `rotation_aux(px, x, y, t)` qui effectue la rotation de la portion carrée de l'image comprise entre les pixels `(x, y)` et `(x+t, y+t)`. Cette fonction ne renvoie rien, elle modifie le tableau `px` pour effectuer la rotation de cette portion d'image, au même endroit. On suppose que `t` est une puissance de 2.

1. Implémenter la fonction `rotation_aux`.
2. En déduire une fonction `rotation(im)` qui effectue une rotation d'image. Une fois la rotation effectuée, on pourra sauvegarder l'image dans un autre fichier avec la commande `im.save("rotation_lena.png")`.