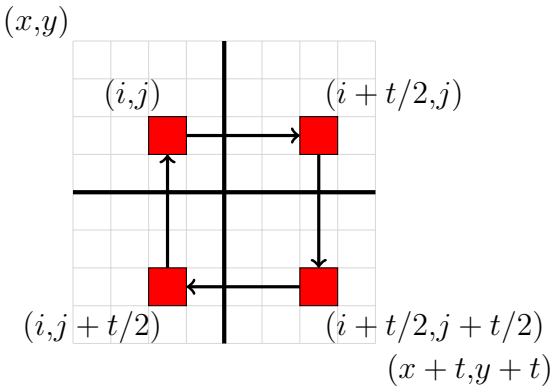
Diviser pour régner

Exercice 1 (Rotation d'une image bitmap d'un quart de tour)
Dans cet exercice, on cherche à écrire une fonction qui effectue la rotation d'une image de 90 degrés de manière récursive.
Il est à noter que cette méthode est en fait plus compliquée que la rotation directe, mais c'est un bel exemple de récursivité.
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 512×512.
L'idée de l'algorithme est de découper l'image en quatre et d'effectuer une rotation circulaire de chacun des quadrants obtenus. On recommence ensuite ce procédé avec chacune des quatre parties.
L'algorithme s'arrête lorsque l'image ne contient plus qu'un seul pixel : la rotation d'une telle image revient à ne rien faire.

Le module `Image` de la librairie `PIL` nous fournira la description des pixels d'une image à l'aide d'une matrice `px` utilisable de la manière suivante : la couleur du pixel de coordonnées (x,y) est donnée par `px[x,y]`.

1. La fonction `echange_circulaire(px, x, y, t)` réalise la permutation des quadrants de la portion carrée comprise entre les pixels `(x,y)` et `(x+t,y+t)`. Cette fonction effectue la permutation de chaque pixel comme dans la figure ci-dessous.
   Compléter le programme ci-dessous :

```
def echange_circulaire(px, x, y, t):
  n = t//2
    for i in range(x, x+n):
        for j in range(y, y+n):
            tmp = px[i,j]
            px[i,j] = ......
            ..........
            ..........
            ...... = tmp
```



2. Afin de procéder récursivement, on va définir une fonction `rotation_rec(px, x, y, t)` qui effectue la rotation de la portion carrée comprise entre les pixels `(x,y)` et `(x+t,y+t)`.

```
def rotation_rec(px, x, y, t):
  if ..........:
    return

  echange_circulaire(px, x, y, t)

  n = t//2
  rotation_rec(...,...,...,...)
  rotation_rec(...,...,...,...)
  rotation_rec(...,...,...,...)
  rotation_rec(...,...,...,...)
```

3. Compléter alors la fonction `rotation(im)` qui effectue la rotation de l'image entière en appelant la fonction récursive :

```
def rotation(im):
    size = im.width # largeur de l'image
    px = im.load() # la matrice de l'image
    rotation_rec(...,...,...,...)
```

4. On peut enfin tester notre fonction après avoir chargé l'image avec le module `Image` de la librairie `PIL`. On enregistre ensuite l'image dans un nouveau fichier :

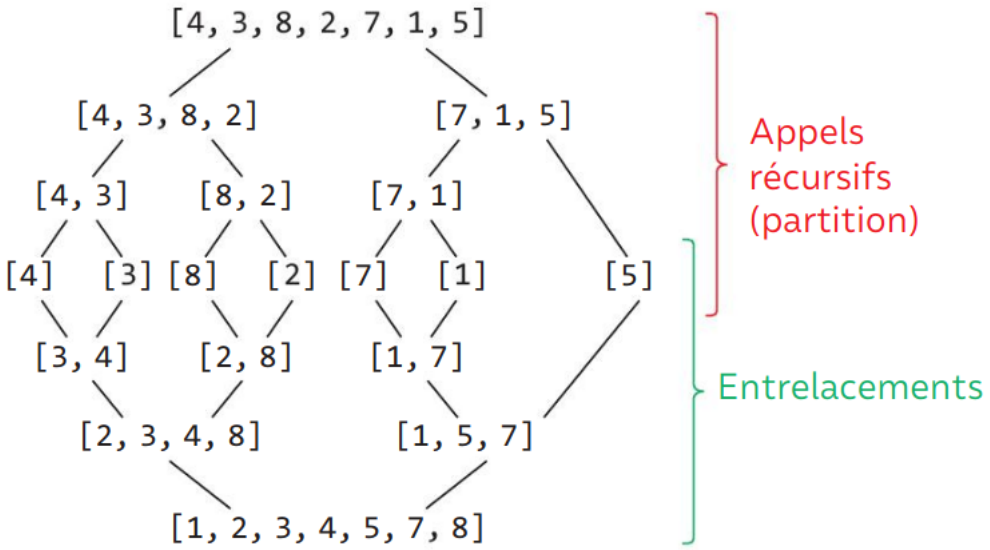
```
from PIL import Image

with Image.open("Turn_right.png") as im:
    rotation(im)
    im.save("Turn_right_rotation.png")
```

5. **(facultatif)** Définir une fonction qui effectue la rotation de manière directe.
   On pourra comparer les temps d'exécution en utilisant la fonction `process_time()` du module `time`.
   Indiquer la complexité de la fonction de rotation directe en fonction de la largeur n de l'image. La rotation récursive a, quant à elle, une complexité en $O(\log_2(n) \times n^2)$.

Exercice 2 (Tri fusion)

Le principe du tri fusion, *merge sort* en anglais, repose sur la stratégie diviser pour régner.
La liste à trier est partagée en deux parties de tailles égales à une unité près, un appel récursif est alors réalisé sur chacune des deux parties. Lorsque ces deux parties sont triées, elles sont fusionnées en une liste triée.



Partager une liste en deux est simple. L'algorithme repose en fait surtout sur la fonction `fusion` qui prend deux listes `l1` et `l2` d'entiers triées par ordre croissant et les fusionne en une liste triée `l` qu'elle retourne.
Par exemple :

```
>>> fusion([1,6,10],[0,7,8,9])
[0, 1, 6, 7, 8, 9, 10]
```

1. Recopier et compléter le code suivant de la fonction `fusion` :

```
def fusion(l1, l2):
    l = []
    i, j = 0, 0
    #
    while i < len(l1) and ......:
        if l1[i] < l2[j]:
            l.append(......)
            i = ......
        else:
            l.append(l2[j])
            j = ......
    # s'il reste des éléments dans l1
    while i < len(l1):
        l.append(......)
        i = ......
    # s'il reste des éléments dans l2
    while j < len(l2):
        l.append(......)
        j = ......
    return l
```

2. Pour effectuer un tri, la fonction `fusion` est appelée de manière récursive sur des parties du tableau à trier.
   Recopier et compléter la fonction de tri `tri_fusion` suivante :

```
def tri_fusion(l):
    # si le tableau contient 0 ou 1 élément
    if ......:
        return l
    else:
        milieu = len(l)//2
        # partie gauche du tableau
        l1 = l[:milieu]
        # partie droite du tableau
        l2 = l[milieu:]
        return fusion(......,......)
```

3. Tester alors la fonction.

Exercice 3 (Somme maximale des sous-tableaux)
Le but de cet exercice est de résoudre le problème suivant :
Étant donné un tableau d'entiers, trouver la somme maximale parmi tous les sous-tableaux possibles.

On appelle sous-tableau d'un tableau une tranche (un *slice*) de valeurs consécutives du tableau.
Exemple : La somme maximale des sous-tableaux de `[2, -4, 1, 9, -6, 7, -3]` est 11.
En effet, la somme des éléments du sous-tableau `[1, 9, -6, 7]` est 11, et c'est la plus grande possible.
La description générale de l'algorithme est la suivante :

- Trouver la somme maximale incluant l'élément du milieu ;
- Obtenir récursivement la somme maximale des sous-tableaux de la partie à gauche du milieu ;
- Obtenir récursivement la somme maximale des sous-tableaux de la partie à droite du milieu ;
- Retourner le maximum des trois sommes précédentes ;

1. Recopier et compléter le code récursif suivant qui doit retourner la somme maximale des sous-tableaux pris entre les indices `g` et `d` :

```
def somme_max_rec(tab, g, d):

    # Si le tableau ne contient qu'un seul élément
    # (ou si d est inférieur à g)
    if g >= d:
        # on retourne l'élément d'indice g de tab
        return ......

    # Trouver l'indice de l'élément du milieu
    m = ......

    # Trouver la somme maximale
    # depuis l'élément du milieu
    # en allant à gauche
    max_gauche = tab[m]
    somme = max_gauche
    for i in range(m - 1, g - 1, -1):
        somme = ......
        if somme > max_gauche:
            max_gauche = ......

    # Trouver la somme maximale
    # depuis l'élément après le milieu
    # en allant à droite
    max_droit = float('-inf') # nombre - infini
    somme = 0
    for i in range(m + 1, d + 1):
        somme = ......
        if somme > max_droit:
            max_droit = ......

    # Trouver la somme maximale
    # incluant l'élément du milieu
    max_milieu = ......

    # Trouver récursivement la somme maximale
    # pour le tableau de gauche
```

```
    # et le tableau de droite
    # excluant l'élément du milieu
    max_gauche = ......
    max_droite = ......

    # Retourner le maximum des trois sommes
    return max(max_gauche, max_droite, max_milieu)
```

2. Définir alors la fonction `somme_max(tab)` qui retourne la valeur recherchée :

```
def somme_max(tab):
    g = ......
    d = ......
    return somme_max_rec(tab, g, d)
```

3. Tester enfin la fonction avec l'exemple initial.

Exercice 4 (Extrait des épreuves pratiques ; fusion)
La fonction `fusion` prend deux listes `L1` et `L2` d'entiers triées par ordre croissant et les fusionne en une liste triée `L12` qu'elle renvoie.
Le code Python de la fonction est :

```
def fusion(L1,L2):
  n1 = len(L1)
  n2 = len(L2)
  L12 = [0]*(n1+n2)
  i1 = 0
  i2 = 0
  i = 0
  while i1 < n1 and ...... :
    if L1[i1] < L2[i2] :
      L12[i] = ......
      i1 = ......
    else:
      L12[i] = L2[i2]
      i2 = ......
    i += 1
  while i1 < n1:
    L12[i] = ......
    i1 = i1 + 1
    i = ......
  while i2 < n2:
    L12[i] = ......
    i2 = i2 + 1
    i = ......
  return L12
```

Compléter le code.
Exemple :

```
>>> fusion([1,6,10],[0,7,8,9])
[0, 1, 6, 7, 8, 9, 10]
```

Exercice 5 (Extrait des épreuves pratiques ; dichotomie)
Soit `T` un tableau non vide d'entiers dans l'ordre croissant et `n` un entier.
La fonction `chercher`, donnée ci-dessous, doit renvoyer un indice où la valeur `n` apparaît éventuellement dans `T`, et `None` sinon.
Les paramètres de la fonction sont :

- `T`, le tableau dans le quel s'effectue la recherche ;
- `n`, l'entier à chercher dans le tableau ;
- `i`, l'indice de début de la partie du tableau où s'effectue la recherche ;
- `j`, l'indice de fin de la partie du tableau où s'effectue la recherche.

La fonction `chercher` est une fonction récursive basée sur le principe « diviser pour régner ».
Le code de la fonction commence par vérifier si `0 <= i` et `j <= len(T)`. Si cette condition n'est pas vérifiée, elle affiche `"Erreur"` puis renvoie `None`.
Recopier et compléter le code de la fonction `chercher` proposée ci-dessous :

```
def chercher(T,n,i,j):
  if i < 0 or ...... :
    print("Erreur")
    return None
  if i > j :
    return None
  m = (i+j) // ......
  if T[m] < ...... :
    return chercher(T, n, ......, ......)
  elif ...... :
    return chercher(T, n, ......, ......)
  else:
    return ......
```

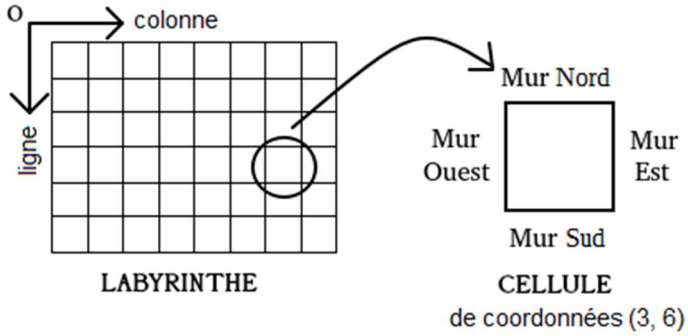
L'exécution du code doit donner :

```
>>> chercher([1,5,6,6,9,12],7,0,10)
Erreur
>>> chercher([1,5,6,6,9,12],7,0,5)
>>> chercher([1,5,6,6,9,12],9,0,5)
4
>>> chercher([1,5,6,6,9,12],6,0,5)
2
```

1. Annales

EXERCICE 5 (4 points)

Cet exercice aborde la programmation objet et la méthode diviser pour régner.



Un labyrinthe est composé de cellules possédant chacune quatre murs (voir ci-dessus). La cellule en haut à gauche du labyrinthe est de coordonnées (0, 0). On définit la classe `Cellule` ci-dessous. Le constructeur possède un attribut `murs` de type `dict` dont les clés sont `'N'`, `'E'`, `'S'` et `'O'` et dont les valeurs sont des booléens (`True` si le mur est présent et `False` sinon).

```
class Cellule:
    def __init__(self, murNord, murEst, murSud, murOuest):
        self.murs={'N':murNord,'E':murEst,
                   'S':murSud,'O':murOuest}
```

1. Recopier et compléter sur la copie l'instruction Python suivante permettant de créer une instance `cellule` de la classe `Cellule` possédant tous ses murs sauf le mur Est.

```
cellule = Cellule(...)
```

2. Le constructeur de la classe `Labyrinthe` ci-dessous possède un seul attribut grille. La méthode `construire_grille` permet de construire un tableau à deux dimensions `hauteur` et `longueur` contenant des cellules possédant chacune ses quatre murs. Recopier et compléter sur la copie les lignes 6 à 10 de la classe `Labyrinthe`.

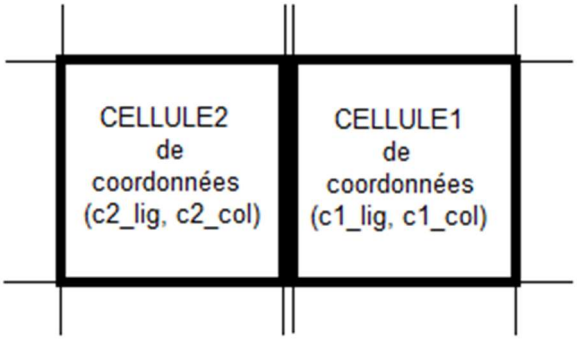
```
1  class Labyrinthe:
2      def __init__(self, hauteur, longueur):
3          self.grille=self.construire_grille(hauteur, longueur)
4      def construire_grille(self, hauteur, longueur):
5          grille = []
6          for i in range(...):
7              ligne = []
8              for j in range(...):
9                  cellule = ...
10                 ligne.append(...)
11             grille.append(ligne)
12         return grille
```

Pour générer un labyrinthe, on munit la classe `Labyrinthe` d'une méthode `creer_passage` permettant de supprimer des murs entre deux cellules ayant un côté commun afin de créer un passage. Cette méthode prend en paramètres les coordonnées `c1_lig`, `c1_col` d'une cellule notée `cellule1` et les coordonnées `c2_lig`, `c2_col` d'une cellule notée `cellule2` et crée un passage entre `cellule1` et `cellule2`.

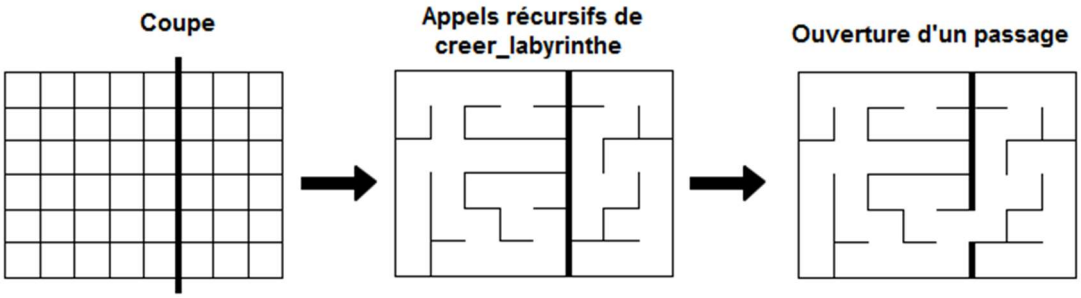
```
13      def creer_passage(self, c1_lig, c1_col, c2_lig, c2_col):
14          cellule1 = self.grille[c1_lig][c1_col]
15          cellule2 = self.grille[c2_lig][c2_col]
16          # cellule2 au Nord de cellule1
17          if c1_lig - c2_lig == 1 and c1_col == c2_col:
18              cellule1.murs['N'] = False
19              ....
20          # cellule2 à l'Ouest de cellule1
21          elif ....
22            ....
23            ....
```

3. La ligne 18 permet de supprimer le mur Nord de `cellule1`. Un mur de `cellule2` doit aussi être supprimé pour libérer un passage entre `cellule1` et `cellule2`. Écrire l'instruction Python que l'on doit ajouter à la ligne 19.

4. Recopier et compléter sur la copie le code Python des lignes 21 à 23 qui permettent le traitement du cas où `cellule2` est à l'Ouest de `cellule1` :

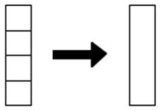


Pour créer un labyrinthe, on utilise la méthode diviser pour régner en appliquant récursivement l'algorithme `creer_labyrinthe` sur des sous-grilles obtenues en coupant la grille en deux puis en reliant les deux sous-labyrinthes en créant un passage entre eux.



La méthode `creer_labyrinthe` permet, à partir d'une grille, de créer un labyrinthe de hauteur `haut` et de longueur `long` dont la cellule en haut à gauche est de coordonnées (`ligne`, `colonne`).
Le cas de base correspond à la situation où la grille est de hauteur 1 ou de largeur 1. Il suffit alors de supprimer tous les murs intérieurs de la grille.

Exemple avec
haut = 1 et long = 4



Exemple avec
haut = 4 et long = 1

5. Recopier et compléter sur la copie les lignes 25 à 30 de la méthode `creer_labyrinthe` traitant le cas de base.

```
24      def creer_labyrinthe(self, ligne, colonne, haut, long):
25          if haut == 1 : # Cas de base
26                for k in range(...):
27                    self.creer_passage(ligne, k, ligne, k+1)
28          elif long == 1: # Cas de base
29                for k in range(...):
30                    self.creer_passage(...)
31          else: # Appels récursifs
32             # Code non étudié (Ne pas compléter)
```

6. Dans cette question, on considère une grille de hauteur `haut` = 4 et de longueur `long` = 8 dont chaque cellule possède tous ses murs.
On fixe les deux contraintes supplémentaires suivantes sur la méthode `creer_labyrinthe` :

- Si `haut` ≥ `long`, on coupe horizontalement la grille en deux sous-labyrinthes de même dimension.
- Si `haut` < `long`, on coupe verticalement la grille en deux sous-labyrinthes de même dimension.

L'ouverture du passage entre les deux sous-labyrinthes se fait le plus au Nord pour une coupe verticale et le plus à l'Ouest pour une coupe horizontale.

Dessiner le labyrinthe obtenu suite à l'exécution complète de l'algorithme `creer_labyrinthe` sur cette grille.

EXERCICE 4 (4 points)

Cet exercice porte sur l'algorithme de tri fusion, qui s'appuie sur la méthode dite de « diviser pour régner ».

1. **a.** Quel est l'ordre de grandeur du coût, en nombre de comparaisons, de l'algorithme de tri fusion pour une liste de longueur n ?

b. Citer le nom d'un autre algorithme de tri. Donner l'ordre de grandeur de son coût, en nombre de comparaisons, pour une liste de longueur n. Comparer ce coût à celui du tri fusion. Aucune justification n'est attendue.

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`.

On rappelle que la syntaxe `a//b` désigne la division entière de `a` par `b`.

Par 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`.

```
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)
```

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

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

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

3. Écrire la fonction `moitie_droite`.

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

```
1. def fusion(L1, L2):
2.     L = []
3.     n1 = len(L1)
4.     n2 = len(L2)
5.     i1 = 0
6.     i2 = 0
7.     while i1 < n1 or i2 < n2 :
8.         if i1 >= n1:
9.             L.append(L2[i2])
10.            i2 = i2 + 1
11.        elif i2 >= n2:
12.            L.append(L1[i1])
13.            i1 = i1 + 1
14.        else:
15.            e1 = L1[i1]
16.            e2 = L2[i2]
17.
18.
19.
20.
21.
22.
23.     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 va ajouter à `L` les éléments de `L2` à partir de l'indice `i2` ;
- Si `i2` n'est plus un indice valide, on va ajouter 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 sur la copie 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.