UNIVERSITÉ TOULOUSE III PAUL SABATIER

Université de Toulouse

Faculté des Sciences et d'Ingénierie

DIU NSI

Bloc 1

Programmation python

Récursivité, tris et types complexes

Armelle Bonenfant, Mathieu Raynal, Amal Sayah

Juin 2019

Récursivité sur les entiers

Objectifs

- Comprendre qu'il existe plusieurs façons d'aborder la notion d'itération en algorithmique et dans la plupart des langages de programmation.

- Apprendre à concevoir une solution récursive à un problème.

Introduction (1/3)

- Plusieurs façons d'aborder la notion d'itération en algorithmique et dans la plupart des langages de programmation
  - avec des boucles (while, for, loop, etc.)
  - avec la récursivité

- Montrer comment concevoir une solution récursive pour un problème.

- Une définition **récursive** est une définition dans laquelle intervient le nom qu'on est en train de définir.

Exemples

- Une personne A est un **descendant** d'une autre personne B si et seulement si
  - soit A est un enfant (fils ou fille) de B
  - soit A est un enfant (fils ou fille) d'un **descendant** de B

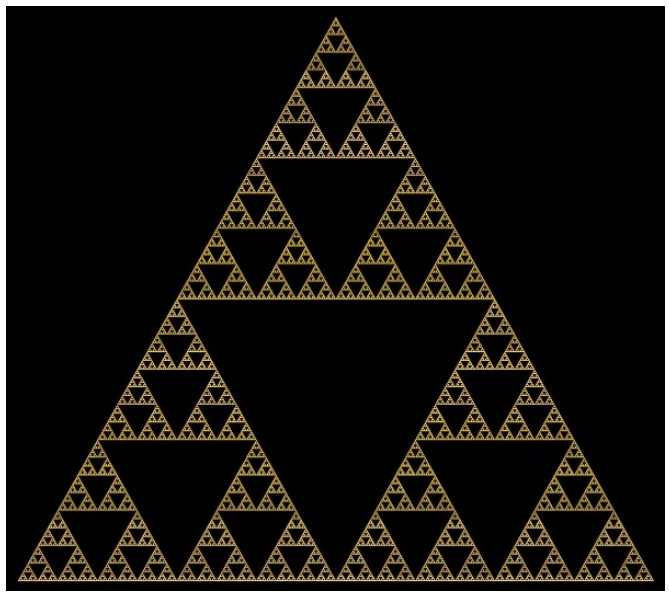
- Un **entier naturel positif ou nul** est
  - soit l'entier 0
  - soit le successeur d'un **entier naturel positif ou nul**.

Introduction (2/3)

- On trouve des définitions récursives dans beaucoup de domaines :
  - Linguistique :
    - Dictionnaire : chaque mot du dictionnaire est défini par d'autres mots eux-mêmes définis par d'autres mots dans ce même dictionnaire.
  - Art :
    - Œuvres d'Escher
    - Image contenant une image similaire
    - Les poupées russes
    - Caméra qui filme devant un miroir

Introduction (2/3)

- Biologie :
  - Motif de végétaux (fougère, cœur d'un tournesol, chou romanesco)
  - Processus de développement (Nautile)
- Mathématique :
  - Suites récurrentes
  - Fractales (triangle de Sierpinski)



- En informatique, on trouve :
  - des **fonctions récursives**, c'est-à-dire qui s'appellent elles-mêmes
  - des **structures de données récursives**, c'est-à-dire qui se définissent en fonction d'elles-mêmes

Introduction (3/3)

- Dans la plupart des langages de programmation, **une fonction peut s'appeler elle-même**, on parle alors de **fonction récursive**

- Les fonctions récursives, permettent, pour certains types de problèmes
  - d'écrire plus facilement les programmes
    → l'écriture se déduit de la définition de la fonction
  - de vérifier plus facilement que les programmes sont corrects
    → preuve par induction

- Il existe aussi la possibilité d'avoir des **définitions mutuellement récursives** c'est-à-dire une fonction qui en appelle une 2ème, qui en appelle une 3ème, …., qui appelle la première.

Exemple : Affichage de nombres

- On veut écrire un programme qui,
  - étant donné un entier n>0,
  - affiche les nombres de 1 à n par ordre *croissant*.

```
# avec une boucle while
n = eval(input())
i = 1
while (i <= n):
        print("i : ",i)
        i = i + 1
```

```
# avec une boucle for
n = eval(input())
for i in range(1,n+1):
        print("i : ",i)
```

Exemple : Affichage de nombres

- Avec une **fonction récursive** de paramètre n :
  - ce qu'on sait faire facilement : quand le paramètre **n vaut 1**
    → on **affiche 1** et c'est fini
  - quand n > 1,
    → on affiche les entiers de 1 à n-1
    → puis on affiche n

Exemple : Affichage de nombres

- le 1er cas est appelé **cas trivial** ou **cas d'arrêt de la récursivité**
  - on s'arrête quand n vaut 1

- le 2e cas est appelé le **cas récursif**

  →on ramène le problème à un sous-problème plus simple, c'est-à-dire de taille plus petite :
  - on fait un appel récursif à la fonction avec n-1

    → on affichera donc les entiers de 1 à n-1
  - on affiche ensuite n

Exemple : Affichage de nombres

```
# avec une fonction récursive
def affichage_ordre_croissant(n):
    if (n == 1):
        print("n : ",1)
    else:
        affichage_ordre_croissant(n-1)
        print("n : ",n)

affichage_ordre_croissant(5)
```

Exemple : Affichage de nombres – ordre décroissant

- Maintenant on veut écrire un programme qui,
  - étant donné un entier n > 0,
  - affiche les nombres de n à 1 par **ordre *décroissant***.

```
# avec une boucle while
n = eval(input())
i = n
while (i >= 1):
        print("i : ",i)
        i = i - 1
```

→On a dû changer

- l'initialisation de la variable de contrôle
- la condition de la boucle while

Exemple : Affichage de nombres – ordre décroissant

```
# avec une boucle for
n = eval(input())
for i in range(n,0,-1):
        print("i : ",i)
```

→ On a dû changer la séquence d'entiers que parcourt le for

Exemple : Affichage de nombres – ordre décroissant

```
# avec une fonction récursive
def affichage_ordre_decroissant(n):
    if (n == 1):
        print("n : ",1)
    else:
        print("n : ",n)
        affichage_ordre_decroissant(n-1)

affichage_ordre_decroissant(5)
```

→On a *seulement* inversé l'ordre d'affichage et l'appel récursif dans le cas où n > 1

Exemple : Affichage de nombres – ordre décroissant

- Pour **le cas trivial** ou **cas d'arrêt de la récursivité**
  - *on s'arrête quand n vaut 1*
  - c'est toujours ce qu'on sait faire facilement : quand le paramètre n vaut 1, on affiche 1 et c'est fini

  → INCHANGE par rapport à l'ordre croissant

- Pour le **cas récursif**
  - quand n > 1
    - *on affiche n*
    - *on affiche* ensuite *les entiers de 1 à n-1*

  → ON A INVERSE par rapport à l'ordre croissant

Remarques

- On constate qu'une fonction récursive comporte :
  - (au moins) un **cas trivial (cas d'arrêt)** pour lequel on a **directement le résultat**
  - (au moins) un **appel récursif** dans lequel on **ramène le problème à un sous-problème plus simple** dans lequel on aura diminué la **"taille du problème"**

Calcul de la puissance $n^{ième}$ d'un entier strictement positif

- Soient a>0 et n>=0, la puissance $n^{ième}$ de a est définie par
  - pour n=0 : $a^0 = 1$
  - pour n > 0 : $a^n = a \times a^{n-1}$

- Cette définition mathématique comporte
  - un point de départ *(n = 0)* pour lequel on a directement le résultat *($a^0 = 1$)*
  - un calcul de la puissance $n^{ième}$ de a qui fait appel au calcul de la puissance $(n-1)^{ième}$ de a.

Calcul de la puissance $n^{ième}$ d'un entier strictement positif

- C'est donc une définition récursive avec :
  - un cas **d'arrêt** de la récursivité : n = 0
  - un cas de calcul récursif $a^n = a \times a^{n-1}$ dans lequel on fait **diminuer la taille du problème** (n-1)

- On constate donc que, pour tout n>0, le calcul finira par s'arrêter quand n atteindra le cas d'arrêt et la valeur 0.

Calcul de la puissance n^{ième} d'un entier strictement positif

- On se calque sur la définition mathématique pour écrire en Python la définition de la fonction **puissance** à 2 paramètres a *(avec a>0)* et n *(avec n>=0)*
  - un cas d'arrêt de la récursivité : **n=0**
    → le résultat est 1
  - un cas de calcul récursif : **n>0**
    → pour calculer a^n, on multiplie n avec le résultat du calcul de a^{n-1}

Calcul de la puissance n^{ième} d'un entier strictement positif

```
# calcul de a à la puissance n, pour a>0 et n>=0
def puissance(a,n):
    if (n == 0):
        # cas d'arrêt
        return 1
    else:
        # n > 0 cas récursif
        return a * puissance(a,n-1)

puissance(2,4)
puissance(2,0)
```

Calcul de la puissance n^{ième} d'un entier strictement positif

```
# On peut aussi écrire sans le else :
def puissance_bis(a,n):
    if (n == 0):
        # cas d'arrêt
        return 1
    # n > 0 cas récursif
    return a * puissance_bis(a,n-1)

puissance_bis(2,4)
puissance_bis(2,0)
```

Démarche pour écrire une fonction récursive

- On commence par chercher le(s) cas simple(s), c'est-à-dire celui(ceux) qui ne nécessite(nt) pas de rappeler récursivement la fonction :
  - Pour l'affichage, c'est le cas où n=1 (on affiche 1)
  - Pour la puissance, c'est le cas où n=0 (on sait que $a^0 = 1$)

Démarche pour écrire une fonction récursive

- On cherche ensuite le(s) sous-problème(s) récursif(s) (sous-problème de taille réduite par rapport au problème) pour rappeler la fonction récursive pour chaque sous-problème à résoudre
  - Pour l'affichage, c'est afficher les n-1 entiers
  - Pour la puissance, c'est calculer a^{n-1}

- Vérifier que la fonction se termine

  → Atteint-on au moins un cas d'arrêt ?

Problème de la terminaison

- Une fonction récursive doit obligatoirement avoir au moins un cas d'arrêt

- Les appels récursifs doivent permettre d'atteindre ce(s) cas d'arrêt.

QUESTIONS OBLIGATOIRES

- Soit la définition de la fonction f ci-dessous.

```
def f(n):
    return n * f(n-1)
```

→ Que se passe t-il si on veut exécuter f(3)?

QUESTIONS OBLIGATOIRES

- Soit la définition de la fonction g ci-dessous.

```
def g(n):
    if(n==0):
        return 1
    else:
        return n * g(n+1)
```

→ Que se passe t-il si on veut exécuter g(3)?

QUESTIONS OBLIGATOIRES

- Soit la définition de la fonction h ci-dessous.

```
def h(n):
    if(n==0):
        return 1
    else:
        return n * h(n)-1
```

→ Que se passe t-il si on veut exécuter h(3)?

QUESTIONS OBLIGATOIRES

- Soit la définition de la fonction p ci-dessous.

```
def p(n):
    if(n==0):
        return 1
    else:
        return n * p(n-2)
```

→ Que se passe t-il si on veut exécuter p(3)?

Problème du domaine de définition

- Ecrire la fonction factorielle qui,
  - étant donné un entier n>=0,
  - calcule le $n^{ième}$ terme de la suite F_n définie par :
    - pour n=0 : $F_0 = 1$
    - pour n>0 : $F_n = n \times (n-1) \times \ldots \times 1 = n \times F_{n-1}$

Problème du domaine de définition

- Pour ce problème, l'écriture d'une fonction récursive en Python s'impose d'elle même en suivant la définition mathématique avec
  - un cas d'arrêt de la récursivité : **n=0**

    → le résultat est 1
  - un cas de calcul récursif : **n>0**

    → pour calculer F_n, on doit multiplier n avec le résultat du calcul de F_{n-1} *(sous-problème de taille réduite)*

Fonction factorielle

```
def factorielle(n):
    if (n == 0):
        return 1
    else:
        return n * factorielle(n-1)

print(factorielle(5))
```

Traces de la fonction factorielle

- On ajoute des impressions *(avec print)*

  → Permet de tracer à chaque appel la valeur de l'argument et le résultat calculé

```
def factorielle_trace_argument_et_resultat(n):
    # ajout des impressions de n et du résultat calculé jusque là
    if (n == 0):
        print('Pour n = ', n, ' , factorielle(',n,') est égal à :', 1)
        return 1
    else:
        resultat = n * factorielle_trace_argument_et_resultat(n-1)
        print('Pour n = ', n, ' , factorielle(',n,') est égal à :', resultat)
        return resultat

factorielle_trace_argument_et_resultat(5)
```

Qu'en est-il de factorielle(-5) ?

- La fonction factorielle est définie sur N donc l'argument doit être positif ou nul, sinon on va vouloir calculer factorielle(-1), puis de -2, etc. et on boucle sur les entiers négatifs…

- Ecrire une version qui permette de résoudre ce problème signifie qu'on va devoir **vérifier la validité de la donnée**.

Factorielle : Gestion des valeurs négatives

```
def factorielle_argument_valide(n):
    # tester si l'argument est positif ou nul pour faire les calculs
    if (n < 0):
        print('Pour n =', n, ', qui est négatif, factorielle(',n,') est non définie')
    else:
        if (n == 0):
            print('Pour n =', n, ', factorielle(',n,') est égal à :', 1)
            return 1
        else:
            resultat = n * factorielle_argument_valide(n-1)
            print('Pour n =', n, ', factorielle(',n,') est égal à :', resultat)
            return resultat

factorielle_argument_valide(-5)
```

Factorielle : Gestion des valeurs négatives

- Il ne semble pas judicieux de tester à chaque appel récursif si n < 0.
  - Si n < 0 dés le départ il faut signaler l'erreur de domaine
  - mais si n >= 0, le calcul permettra à n d'atteindre la valeur 0 (et donc arrêter le calcul) avant une valeur négative.

- On va **séparer** la vérification du domaine de validité de l'argument du calcul proprement dit quand l'argument est correct.

EXERCICES : Fonctions récursives sur les entiers

- **Pour chacune des fonctions suivantes, on supposera la validité de l'argument sans la vérifier**

Fibonacci

- Ecrire la fonction *Fibonacci* qui, étant donné un entier n>0, calcule Fib_n, le $n^{ième}$ terme de la suite définie par
  - $Fib_0 = 1$
  - $Fib_1 = 1$
  - $Fib_{n+2} = Fib_{n+1} + Fib_n$

- Quel est le résultat de l'appel suivant ?

```
fibonacci(12)
```

Fibonacci

- Cas trivial d'arrêt de la récursivité
  - n == 0 et n==1 : dans ce cas le résultat est 1

- Cas de calcul récursif : n > 1
  - pour calculer F_n, on doit additionner les résultats des calculs de F_{n-1} et F_{n-2}

Fibonacci

```
def fibonacci(n):
    # 2 cas d'arrêt : n = 0 et n=1 retournant 1
    if (n==0 or n==1):
        return 1
    # cas général
    # 2 appels récursifs avec la taille du problème diminuée
    return (fibonacci(n-1) + fibonacci(n-2))
```

Suite

- Ecrire la fonction *suite* qui, étant donné un entier n >= 0, calcule U_n, le $n^{ième}$ terme de la suite définie par
  - $U_0 = 2$
  - $U_{n+1} = (1 + 3 \times U_n) / (3 + U_n)$

- Quel est le résultat de l'appel suivant ?

```
suite(12)
```

Suite

- Cas trivial d'arrêt de la récursivité
  - n = 0, dans ce cas, le résultat est 2

- Cas de calcul récursif : n > 0
  - pour calculer U_n, on utilise le résultat du calcul de U_{n-1}

Suite

```
def suite(n):
    # 1 cas d'arrêt : n = 0 retournant 2
    if (n == 0):
        return 2
    # cas général
    # 2 appels récursifs avec la taille du problème diminuée
    return (1 + 3 * suite(n-1)) / (3 + suite(n-1))
```

Suite

- On remarque que *suite(n-1)* est calculée 2 fois à chaque appel récursif.

- On peut améliorer ce code en nommant le résultat de *suite(n-1)* calculé une **seule** fois pour l'utiliser 2 fois.

- La variable locale *resultat_intermédiaire* sera différente à chaque appel.

- Quel est le résultat de l'appel suivant ?

```
suite_bis(12)
```

Suite

```
def suite_bis(n):
    # cas d'arrêt : n = 0 retournant 2
    if (n == 0):
        return 2
    # cas général
    # 1 seul appel récursif
    # conserver le résultat dans une variable locale
    resultat_intermediaire = suite_bis(n-1)
    # on utilise le résultat sans le recalculer
    return (1+ 3 *resultat_intermediaire)/(3+resultat_intermediaire)
```

Suite

- Ecrire la fonction *pgcd* qui, étant donnés 2 entiers a > 0 et b > 0, calcule le plus grand commun diviseur de a et de b défini par :
  - pgcd(a,b) = a si a = b
  - pgcd(a,b) = pgcd(min(a,b),|a-b|) sinon

- Quels sont les résultats pour les appels suivants ?

pgcd(15,21)

pgcd(11,19)

pgcd(24, 48)

PGCD

- Cas trivial d'arrêt de la récursivité
  - a=b : dans ce cas le résultat est a

- Cas de calcul récursif : a != b
  - Si a>b, le résultat est le même que celui du pgcd de b et a-b
  - Si b>a, le résultat est le même que celui du pgcd de a et b-a

→C'est un sous-problème du PGCD initial

→Petit à petit les deux nombres diminuent

→La différence est forcément positive

→Si a et b toujours différents, la différence va aboutir à 1

PGCD

- PGCD(15,21)
  - →Plus petit : 15, différence : 6 → =PGCD(15,6)
  - →Plus petit : 6, différence : 9 → = PGCD(6,9)
  - →Plus petit : 6, différence : 3 → = PGCD(6,3)
  - →Plus petit : 3, différence : 3 → = PGCD(3,3)
  - →a et b sont égaux et valent 3 donc PGCD(15,21)=3

PGCD

```
def pgcd(a,b):
    # cas d'arrêt : a = b retournant a
    if (a == b):
        return a
    # cas général a != b,
    else: # on cherche le plus petit des 2
        if (a > b): # appel récursif
            # avec le plus petit(b) et la différence (a-b)
            return pgcd(b,a-b)
        else: # appel récursif
            # avec le plus petit(a) et la différence (b-a)
            return pgcd(a,b-a)
```

PGCD

- On peut aussi écrire les fonctions :
  - ***min*** qui, appliquée à 2 entiers, retourne le plus petit des 2
  - ***val_abs*** qui retourne la valeur absolue de son paramètre

et les composer

- Quels sont les résultats pour les appels suivants ?

```
pgcd_bis(15,21)
```

```
pgcd_bis(11,19)
```

```
pgcd_bis(24, 48)
```

PGCD

```
def min(a,b):
    if (a < b):
        return a
    else:
        return b

def val_abs(a):
    if (a >= 0):
        return a
    else:
        return(-a)
```

PGCD

```
def pgcd_bis(a,b):
    # cas d'arrêt : a = b retournant a
    if (a == b):
        return a
    # cas général a != b
    else: # appel récursif
        # avec le plus petit des deux et la valeur absolue de leur différence
        return pgcd_bis(min(a,b),val_abs(a-b))
```

Récursivité sur les listes

Tours de Hanoï

L'exemple présenté le plus souvent comme le plus percutant sur la récursivité est le problème dit des Tours de Hanoï car la solution récursive est logique, concise, élégante et . . . la seule viable !

- On dispose de **trois "tours" A, B et C** et de **n disques** troués en leur centre et de tailles différentes deux à deux.
- Au départ, les n disques sont empilés sur la tour A de telle sorte que chaque disque ait un diamètre inférieur au disque immédiatement en dessous de lui.
- Ainsi, le plus grand disque de la tour se situe à sa base et le plus petit sur son sommet.

Tours de Hanoï

L'objectif est de déplacer tous les disques de la tour A vers la tour C en s'aidant uniquement de la tour B, tout en respectant les règles suivantes :

- On ne déplace qu'un seul disque à la fois ;
- Un disque ne peut pas être mis sur un disque plus petit ;
- À chaque étape, la règle d'organisation des tours décrite précédemment doit être respectée.

Tours de Hanoï

Bien sûr, on peut essayer à la main avec 2 disques, éventuellement 3 (surtout pas plus !) mais ce n'est peut être pas une bonne idée !

Mieux vaut penser récursivement la solution au problème quel que soit le nombre n de disques.

Pour déplacer n disques de la tour A vers la tour C en passant par B, il faudra :

- Déplacer(n-1) disques de A vers B en passant par C ;
- Puis déplacer 1 disque de A vers C ;
- Puis déplacer(n-1) disques de B vers C en passant par A.

Et c'est tout...

Tours de Hanoï

Pour cette fonction récursive, on ne calculera pas de valeurs, mais on fera des impressions :

```
1 print("déplacer un disque de ",tour_depart," vers ",tour_arrivee)
```

La fonction retournera None à la fin.

Tours de Hanoï

```
1 def hanoi(n, tour_depart, tour_arrivee, tour_intermediaire):
2     if n==1:
3     print("deplacer un disque de ", tour_depart , " vers " , tour_arrivee )
4   else:
5     hanoi(n-1,tour_depart, tour_intermediaire, tour_arrivee)
6     hanoi(1,tour_depart, tour_arrivee, tour_intermediaire)
7     hanoi(n-1,tour_intermediaire, tour_arrivee ,tour_depart)
```

Tours de Hanoï

Qu'affichent les appels suivants ?

```
1 hanoi(3, 'A', 'C', 'B')
```

```
1 hanoi(5, 'A', 'C', 'B')
```

Tours de Hanoï

Si on considère qu'on ne fait rien si le nombre de disques est 0, une autre solution est :

```
1 def hanoi_bis(n, tour_depart, tour_arrivee, tour_intermediaire):
2     if n>0:
3         hanoi_bis(n-1,tour_depart, tour_intermediaire, tour_arrivee)
4         print("deplacer un disque de ", tour_depart , " vers " , tour_arrivee )
5         hanoi_bis(n-1,tour_intermediaire, tour_arrivee ,tour_depart)
```

Introduction

Une solution récursive est particulièrement adaptée quand on doit traiter des structures de données récursives.

Les listes qu'on appelle chaînées (et d'autres structures de données que nous verrons plus tard) sont des structures récursives. En effet, on peut dire d'une liste qu'elle est :

- soit vide ;
- soit construite à partir d'un élément et d'une autre liste (ajout d'un élément dans la liste).

On peut donc en déduire que le traitement d'une liste suivra (le plus souvent) le schéma récursif suivant :

- La liste vide correspondra au cas d'arrêt (cas trivial, de base) de la récursivité.
- La liste résultant de l'ajout de l'élément element à la liste reste_liste correspondra au cas de calcul récursif avec la taille du problème diminuée si on fait le traitement sur reste_liste.

Bien entendu, il peut y avoir des traitements où nous aurons besoin d'autres cas d'arrêt (liste à un seul élément, par exemple).

Introduction

Jusqu'à présent, quand nous avons traité des listes, nous avons le plus souvent parcouru toute la liste, élément par élément pour faire le traitement adéquat. Suivant le modèle de solution choisi, le parcours des listes pouvait se faire :

- de la gauche vers la droite,
- de la droite vers la gauche,
- à partir du milieu,
- ...

Pour la construction et le traitement récursif des listes, il est généralement admis de faire le parcours de gauche à droite, c'est-à-dire que l'ajout d'un élément dans une liste se fait en tête (mais on aurait pu choisir un autre mode de solution).

Avertissement

Dans cette partie du cours, nous voulons montrer comment exprimer une solution sans se préoccuper de problème d'efficacité.

Pour ne pas "ajouter" de difficultés, nous avons pris le parti de ne pas modifier les listes données en argument des fonctions. Il est bien évidemment qu'on pourrait donner d'autres(s) version(s) des fonctions en modifiant les listes données en argument pour économiser de la mémoire.

Démarche pour écrire une fonction récursive sur les listes

Chercher le(s) cas simple(s), c'est-à-dire celui(ceux) qui ne nécessite(nt) pas de rappeler récursivement la fonction :

- Le plus souvent liste vide.
- Eventuellement liste à un élément.

Chercher le sous-problème récursif (sous-problème de taille réduite par rapport au problème) pour rappeler la fonction récursive pour ce sous-problème :

- la liste privée d'un élément,
- le plus souvent privée de son 1^{er} élément.

Démarche pour écrire une fonction récursive sur les listes

"Vérifier/Constater" que la fonction termine, c'est-à-dire qu'on atteint au moins un cas d'arrêt :

- le plus souvent, on a diminué la taille de la liste en enlevant un élément de celle-ci

Exemples

Ecrire la fonction récursive longueur_liste qui donne le nombre d'éléments d'une liste.

2 cas à traiter :

- Cas d'arrêt : la liste est vide
  - $\rightarrow$ On retourne 0.
- Cas récursif : la liste n'est pas vide
  - $\rightarrow$ On retourne la longueur de la liste privée d'un élément (par exemple, le 1^{er}) + 1.

Fonction récursive longueur_liste

```
1 def longueur_liste ( liste ):
2     if ( liste == []):
3         return 0
4     else :
5         return (1 + longueur_liste( liste [1:]) )
```

Exercices classiques sur les listes

Exercice 1

Ecrire la fonction récursive produit_elements_de_la_liste qui, étant donnée une liste d'entiers, calcule le produit des éléments de la liste.

Par convention, le produit des éléments d'une liste vide sera 1.

Fonction récursive produit_elements_de_la_liste

2 cas à traiter :

- Cas d'arrêt : la liste est vide
  → On retourne 1.
- Cas récursif : la liste n'est pas vide
  → On retourne le produit de l'élément en tête de liste avec le résultat de produit_elements_de_la_liste sur la liste privée de la tête de liste.

Fonction récursive produit_elements_de_la_liste

```
1 def produit_elements_de_la_liste( liste ):
2     if ( liste == []):
3         return 1
4     else :
5         return ( liste [0]*produit_elements_de_la_liste( liste [1:]) )
```

Exercice 2

Ecrire la fonction récursive dernier_element_de_la_liste qui, étant donnée une liste, retourne le dernier élément de la liste s'il existe.

Fonction récursive dernier_element_de_la_liste

3 cas à traiter :

- Cas d'arrêt : la liste est vide.
  → On retourne 1.
- Cas d'arrêt : la liste a un seul élément.
  → On retourne cet élément.
- Cas récursif : la liste a plus de 1 élément.
  → On retourne la liste privée de son 1^{er} élément.

Fonction récursive dernier_element_de_la_liste

```
1 def dernier_element_de_la_liste_non_vide(liste):
2     if len(liste)==1:
3         return liste[0]
4     else:
5         return dernier_element_de_la_liste(liste[1:])
```

```
1 def dernier_element_de_la_liste(liste):
2     assert len(liste)>0, "Le dernier élément d'une liste vide n'est pas défini"
3     return dernier_element_de_la_liste_non_vide(liste)
```

Exercice 3

Ecrire la fonction récursive appartient_element_a_la_liste qui, étant donnés un élément et une liste, retourne True si l'élément appartient à la liste et False sinon.

Modifier cette fonction pour écrire la fonction récursive appartient_element_a_la_liste_croissante qui, étant donnés un élément et une liste supposée triée dans l'ordre croissant, retourne True si l'élément appartient à la liste et False sinon.

Fonction récursive appartient_element_a_la_liste

3 cas à traiter :

- Cas d'arrêt : la liste est vide.
  → On retourne False (l'élément n'appartient donc pas à la liste).
- Cas d'arrêt : l'élément recherché est le 1^{er} de la liste.
  → On retourne True.
- Cas récursif : l'élément recherché n'est pas le 1^{er} de la liste.
  → On cherche l'élément dans la liste privée de son 1^{er} élément.

Fonction récursive appartient_element_a_la_liste

```
1 def appartient_element_a_la_liste(element, liste ):
2     if liste == []:
3         return False
4     else :
5         if liste [0] == element:
6             return True
7         else :
8             return appartient_element_a_la_liste(element, liste [1:])
```

```
1 def appartient_element_a_la_liste_bis(element, liste ):
2     return ( liste != [] and ( liste [0] == element or
          appartient_element_a_la_liste_bis(element, liste [1:]) ))
```

Fonction récursive appartient_element_a_la_liste_croissante

4 cas à traiter :

- Cas d'arrêt : la liste est vide.
  → On retourne False (l'élément n'appartient donc pas à la liste).
- Cas d'arrêt : l'élément recherché est le 1^{er} de la liste.
  → On retourne True (l'élément recherché appartient à la liste).
- Cas d'arrêt : l'élément recherché est strictement supérieur au 1^{er} élément de la liste.
  → On retourne False (l'élément recherché n'appartient pas à la liste, car la liste est triée).
- Cas récursif : l'élément recherché n'est pas le 1^{er} de la liste et est inferieur au 1^{er} élément de la liste.
  → On cherche l'élément dans la liste privée de son 1^{er} élément.

Fonction récursive appartient_element_a_la_liste_croissante

```
1  def appartient_element_a_la_liste_croissante(element, liste):
2      if liste == []:
3          return False
4      else:
5          if liste[0] == element:
6              return True
7          else:
8              if liste[0] < element:
9                  return
                       appartient_element_a_la_liste_croissante(element, liste[1:])
10             else:
11                 return False
```

Exercice 4

Ecrire la fonction récursive est_palindrome qui, étant donnée une liste, retourne True si la liste est un palindrome et False sinon.

[] [1] [1, 2, 2, 1] [1, 2, 3, 2, 1] sont des palindromes.

Fonction récursive est_palindrome

3 cas à traiter :

- Cas d'arrêt : la liste est vide .
  → On retourne True (on a un palindrome de longueur paire).
- Cas d'arrêt : la liste a un seul élément.
  → On retourne True (on a un palindrome de longueur impaire).
- Cas récursif : la liste a plus de un élément.
  → Le 1^{er} et le dernier éléments de la liste sont égaux ;
  → On teste si la liste privée du 1^{er} et du dernier élément est un palindrome.

Fonction récursive est_palindrome

```
1 def est_palindrome( liste ):
2       taille = len( liste )
3       if taille == 0 or taille == 1:
4           return True
5       else:
6           if liste [0] == liste[-1]:
7               return est_palindrome( liste [1: taille -1])
8           else:
9               return False
```

Exercice 5

Ecrire la fonction récursive substituer_element_par_un_autre qui, étant donnés deux éléments, element_a_substituer et nouvel_element, et une liste, construit une nouvelle liste (c'est à dire que la liste initiale ne sera pas modifiée) où toutes les occurrences de l'élément à substituer auront été remplacées par le nouvel élément.

Fonction récursive substituer_element_par_un_autre

2 cas à traiter :

- Cas d'arrêt : la liste est vide.
  - → On retourne la liste vide (il n'y a pas d'élément à substituer).
- Cas récursif : la liste a, au moins, un élement.
  - → Si le 1^{er} élément de la liste est égal à l'élément à substituer, on le remplace par le nouvel élément ;
  - → On le concatène avec la nouvelle liste générée en appelant la fonction substituer_element_par_un_autre avec la liste privée de son 1^{er} élément.

Fonction récursive substituer_element_par_un_autre

```
1  def substituer_elt_par_un_autre(elt_a_sub, new_elt,  liste ):
2      if  liste  == []:
3          return  []
4      else :
5          reste_liste  =  liste [1:]
6          reste_liste_sub  = substituer_elt_par_un_autre(elt_a_sub, new_elt,
               reste_liste )
7          if  liste [0]  == elt_a_sub:
8              liste_substituee  = [new_elt] + reste_liste_sub
9          else :
10             liste_substituee  = [ liste [0]]  + reste_liste_sub
11         return  liste_substituee
```

Exercice 6

Le tri par insertion est un algorithme qui permet d'insérer un élément en bonne position dans une liste déjà triée dans l'ordre croissant. On construit une liste qui sera triée sans modifier la liste initiale.

Fonction récursive inserer_elt_dans_liste_triee

Il nous faut d'abord écrire une fonction récursive inserer_elt_dans_liste_triee qui insère un élément au bon endroit dans une liste déjà triée par ordre croissant.

Fonction récursive inserer_elt_dans_liste_triee

2 cas à traiter :

- Cas d'arrêt : la liste est vide.
  - → On retourne la liste contenant l'élément à insérer.
- Cas récursif : il faut comparer le 1^{er} élément de la liste avec l'élément à insérer.
  - Si l'élément à insérer est inférieur ou égal au 1^{er} élément de la liste
    - → On ajoute l'élément à insérer en tête de la liste résultat.
  - Sinon, on va construite la liste résultat en concaténant :
    - → le 1^{er} élément de la liste,
    - → le résultat de l'insertion de l'élément à insérer dans la liste privée du 1^{er} élément.

Fonction récursive inserer_elt_dans_liste_triee

```
1 def inserer_elt_dans_liste_triee(elt_a_inserer, liste):
2     if liste == []:
3         return [elt_a_inserer]
4     else:
5         if elt_a_inserer <= liste[0]:
6             return [elt_a_inserer]+liste
7         else:
8             return [liste[0]]+ inserer_elt_dans_liste_triee(elt_a_inserer,
                  liste[1:])
```

Fonction récursive tri_par_insertion

2 cas à traiter :

- Cas d'arrêt : la liste est vide.
  - $\rightarrow$ On retourne la liste vide.
- Cas récursif : la liste a, au moins, un élement.
  - $\rightarrow$ On insère le 1^{er} élément de la liste dans la liste privée de son 1^{er} élément triée préalablement.

Fonction récursive tri_par_insertion

```
1 def tri_par_insertion ( liste ):
2     if liste == []:
3         return []
4     else :
5         return
              inserer_elt_dans_liste_triee ( liste [0], tri_par_insertion ( liste [1:]) )
```

Exercice 7

Le tri par fusion est un algorithme qui propose de découper la liste en 2 parties de longueur égale (à un élément près) que l'on doit ensuite trier pour enfin fusionner ces 2 listes triées.

La fonction récursive tri_par_fusion aura donc besoin de deux fonctions :

- partager_liste_en_2 qui, étant donnée une liste, retourne un couple de listes de longueurs équivalentes formées des éléments de la liste à partager ;
- fusionner_2_listes_triees qui, étant données 2 listes triées dans l'ordre croissant, construit dans l'ordre croissant la liste des éléments de ces 2 listes.

Fonction récursive partager_liste_en_2

3 cas à traiter :

- Cas d'arrêt : la liste est vide.
  - → On retourner le couple formé de 2 listes vides.
- Cas d'arrêt : la liste a un seul élément.
  - → On retourne le couple formé d'une liste vide et la liste à un élément.
- Cas récursif : la liste a, au moins, 2 éléments.
  - → il faut partager en 2 la liste privée de ses 2 premiers éléments ;
  - → On retourne ensuite le couple formé du 1^{er} élément à la $1^{ère}$ composante du couple, et le 2^{e} élément à la 2^{e} composante du couple.

Fonction récursive partager_liste_en_2

```
1 def partager_liste_en_2( liste ):
2     if liste == []:
3         return ([],[])
4     else :
5         if len( liste ) == 1:
6             return ([], liste )
7         else :
8             ( liste_1 , liste_2 ) = partager_liste_en_2( liste [2:])
9             return ([ liste [0]]+ liste_1 ,[ liste [1]]+ liste_2 )
```

Fonction récursive fusionner_2_listes_triees

2 cas à traiter :

- Cas d'arrêt : au moins l'une des 2 listes est vide.
  - → On retourne l'autre.
- Cas récursif : les 2 listes sont non vides.
  - → On compare le 1^{er} élément de chacune des 2 listes ;
  - → On retourne la liste avec le plus petit de ces 2 éléments ajouté en tête du résultat de la fusion de la liste privée de ce 1^{er} élément et de l'autre liste inchangée.

Fonction récursive partager_liste_en_2

```
1  def fusionner_2_listes_triees(liste_1, liste_2):
2      if liste_1 == []:
3          return liste_2
4      else:
5          if liste_2 == []:
6              return liste_1
7          else:
8              if liste_1[0] <= liste_2[0]:
9                  liste = fusionner_2_listes_triees(liste_1[1:], liste_2)
10                 return([liste_1[0]]+ liste)
11             else:
12                 liste = fusionner_2_listes_triees(liste_1, liste_2[1:])
13                 return([liste_2[0]]+ liste)
```

Fonction récursive tri_par_fusion

2 cas à traiter :

- Cas d'arrêt : la liste est vide.
  - → On retourne la liste vide.
- Cas récursif : la liste n'est pas vide.
  - → On partage la liste en 2 listes ;
  - → On trie ces 2 listes ;
  - → On retourne la fusion de ces 2 listes triées.

Fonction récursive tri_par_fusion

```
1 def tri_par_fusion( liste ):
2     if len( liste )<=1:
3         return liste
4     else :
5         ( liste_1 , liste_2 ) = partager_liste_en_2( liste )
6         ( liste_1_triee , liste_2_triee ) = (tri_par_fusion( liste_1 ) ,
              tri_par_fusion( liste_2 ))
7         return( fusionner_2_listes_triees ( liste_1_triee , liste_2_triee ))
```

Exercice 8

Le tri rapide est un algorithme de tri très utilisé pour sa relative simplicité et sa rapidité.

On peut en donner une solution récursive :

- Si la liste a moins de 2 éléments, c'est fini, la liste est déjà triée.
- Sinon
  - On choisit un élément de la liste au hasard qu'on appelle pivot ; ici on prendra le 1^{er} élément de la liste pour simplifier ;
  - On partitionne la liste en 2 listes : la liste des éléments qui sont inférieurs au pivot et la liste des éléments qui sont supérieurs au pivot ;
  - On trie ensuite ces 2 listes suivant la même méthode (2 appels récursifs) ;
  - On construit la liste résultat en concaténant les 2 listes triées sans oublier le pivot entre les 2.

Fonction récursive partitionner_liste_suivant_pivot

```
1  def  partitionner_liste_suivant_pivot ( liste , pivot ):
2      if  liste == []:
3          return  ([],[])
4      else :
5          ( inferieur , superieur ) =
                  partitionner_liste_suivant_pivot ( liste [1:], pivot )
6          if  liste [0] > pivot:
7              superieur = [ liste [0]]+ superieur
8          else :
9              inferieur = [ liste [0]]+ inferieur
10         return ( inferieur , superieur )
```

Fonction récursive tri_pivot

```
1 def tri_pivot(liste):
2     if len(liste) < 2:
3         return liste
4     else:
5         (inferieur, superieur) = partitionner_liste_suivant_pivot(liste[1:],
                liste[0])
6         inferieur_trie = tri_pivot(inferieur)
7         superieur_trie = tri_pivot(superieur)
8         return inferieur_trie + [liste[0]] + superieur_trie
```

Exercice 9

L'objectif est d'écrire une fonction récursive qui calcule la moyenne d'une liste NON VIDE ainsi que le nombre d'éléments de cette liste qui sont strictement supérieurs à cette moyenne.

Pour cela la fonction va calculer la moyenne lors des appels récursifs et comptera le nombre d'éléments plus grands lors du retour des appels récursifs.

Fonction récursive nb_superieurs

On vous demande donc de créer une fonction appelée nb_superieurs prenant trois arguments :

- la liste restant à parcourir,
- la somme des éléments déjà parcourus lors des appels récursifs précédents (à 0 lors du 1^{er} appel),
- le nombre des éléments déjà parcourus lors des appels récursifs précédents (à 0 lors du 1^{er} appel).

Fonction récursive nb_superieurs

2 cas à traiter :

- Cas d'arrêt : la liste est vide.
  - → On calcule la moyenne et on renvoie qu'il n'y a aucun élément plus grand que la moyenne dans la liste vide.
- Cas récursif : la liste n'est pas vide.
  - → On fait l'appel récursif sur la liste privée du 1^{er} élément ;
  - → On compare ce 1^{er} élément par rapport à la moyenne renvoyée par l'appel récursif pour renvoyer le bon couple de valeurs.

Fonction récursive nb_superieurs

```
1 def nbSuperieurs( liste , somme, nombre):
2     if liste ==[]:
3         return (0, somme/nombre)
4     partiel , moyenne = nbSuperieurs(liste [1:], somme+liste[0], nombre+1)
5     if liste [0]>moyenne:
6         return ( partiel +1, moyenne)
7     else :
8         return ( partiel , moyenne)
```

Bilan

La récursivité est une solution puissante et finalement assez simple pour exprimer la résolution d'un problème en isolant les cas simples, où le résultat est directement trouvé, et les cas où le problème est découpé en sous-problèmes plus petits (donc plus simples à résoudre) qui seront traités avec des appels récursifs.

Bien évidemment, il faut que le problème s'y prête bien (structures de données naturellement récursives qui s'expriment en fonction d'elles mêmes), ce qui est souvent le cas.

Bilan

La récursivité est une solution à un problème qui s'exprime facilement et assure souvent une bonne lisibilité. Cependant, on lui reproche son inefficacité à cause du nombre d'appels récursifs qu'elle peut engendrer (exemple de Fibonacci . . .).

Il existe des méthodes qui, pour certaines fonctions récursives, permettent de palier le problème en "dérécursivant" la solution pour avoir une solution dite "récursive terminale" qui sera ensuite transformée en boucle while.

Tris

Pourquoi trier ?

- Accès rapide à une donnée
- Visualisation
- Etude de propriétés
- Classement
- …

Comment trier ?

- Objectif : modifier l'ordre des données
- Besoin d'un critère de comparaison
  - 3 < 7
  - "abc" < "ade"
  - Janvier < Mars
- Traiter chaque élément
  - Parcourir tout le tableau
  - 1 seul parcours ?
- Déplacer les éléments mal placés
  - Insertion : placer entre deux objets
  - Substitution: remplacer par (généralement : permuter)

Une grande famille d'algorithmes

On verra dans ce cours :

- Insertion
- Sélection
- Fusion
- Rapide

- Des comparaisons avec d'autres algorithmes

Tri par insertion

- Idée : rajout d'une carte de jeu dans une « main »
  - Autrement dit : prendre un élément et le mettre à sa place dans un ensemble déjà trié
  - Le programme n’accède aux cartes que séquentiellement

Tri par insertion : Principe

- Parcourir le tableau de la gauche vers la droite :
  - Traiter l'élément à la position i
    - Placer ce i^{ème} élément à sa place dans la partie gauche déjà triée tab[0 : i-1]

→ La tranche tab[0 : i] devient, à présent, aussi triée

6 5 3 1 8 7 2 4

Tri par insertion

- Conditions initiales
  - Un tableau contenant une seule valeur est forcément trié !
    - indice = 1

→ tab[0 : indice] est trié

0	1	2	3	4	5
3	2	5	4	9	1

- Condition de boucle
  - Il reste au moins une valeur à placer

- Au début du premier tour de boucle
  - on va placer la 2ème valeur

0	1	2	3	4	5
3	2	5	4	9	1

Tri par insertion

- Après 1 tour de boucle
  - La 2ème valeur a été placée par rapport à la première,

    indice = indice + 1 # indice = 2

  → tab[0 : indice] est trié

0	1	2	3	4	5
2	3	5	4	9	1

- Après 2 tours de boucle
  - La 3ème valeur a été placée par rapport aux précédentes

    indice = indice + 1 # indice = 3

  → tab[0 : indice] est trié

0	1	2	3	4	5
2	3	5	4	9	1

- Après i tours de boucle
  - La ième valeur a été placée par rapport aux précédentes,

    indice = indice + 1 # indice = i +

  → tab[0 : indice] est trié

0	1	2	i	...	len(tab) - 1
2	3	4	5	9	1

Tri par insertion

- Pour un tableau de n éléments, après avoir exécuté les (n-1) tours de boucle
  - indice varie de 1 à len(tab)
  - Sortie de la boucle car il n'y a plus de valeurs à classer
    - indice = = len(tab)
  - Etat final à la sortie de la boucle
    - le tableau est entièrement trié

    → tab[0 : indice] est trié donc

    → tab[0 : len(tab)] est trié

Tri par insertion

Ecrire la fonction tri_insertion (non récursive)

```
def tri_insertion(tab):
    lg = len(tab)
    if (lg == 0):
        return

    #  pour toutes les cartes à classer faire
    for indAClasser in range(1, lg):
        # enlever la carte à classer de la main : crée une place entre les cartes
        carteAClasser = tab[indAClasser]

        # la carte courante est celle juste à gauche de la carte à classer
        indCourant = indAClasser - 1

        # tant que il y a une carte à gauche et cette carte > carte à classer Faire
        while (indCourant >= 0) and (tab[indCourant] > carteAClasser):
            # décalage de cette carte vers la droite
            tab[indCourant + 1] = tab[indCourant]
            indCourant -= 1

        # insertion à la position trouvée : "dégagée" par les décalages
        tab[indCourant + 1] = carteAClasser
```

Tri par insertion récursif

- Il nous faut d'abord écrire une fonction récursive *inserer_elt_dans_liste_triee* qui insère un élément au bon endroit dans une liste déjà triée par ordre croissant

Insérer un élément dans une liste déjà triée

- Cas d'arrêt
  - La liste est vide
  - → On retourne la liste contenant l'élément à insérer

- Cas général récursif : il faut comparer le 1er élément de la liste avec l'élément à insérer
  - si l'élément à insérer est inférieur ou égal au 1er élément de la liste
    - on ajoute l'élément à insérer en tête de la liste résultat.
  - Sinon
    - on va construite la liste résultat en concaténant
      - le 1er élément de la liste
      - le résultat de l'insertion de l'élément à insérer dans la liste privée du 1er élément.

Tri par insertion récursif

Ecrire la fonction *inserer_elt_dans_liste_triee* (récursive)

```
def inserer_elt_dans_liste_triee(elt_a_inserer, liste):
    if liste == []:
        return [elt_a_inserer]
    else:
        if elt_a_inserer <= liste[0]:
            return [elt_a_inserer]+liste
        else:
            return [liste[0]]+ inserer_elt_dans_liste_triee(elt_a_inserer, liste[1:])
```

Tri par insertion récursif

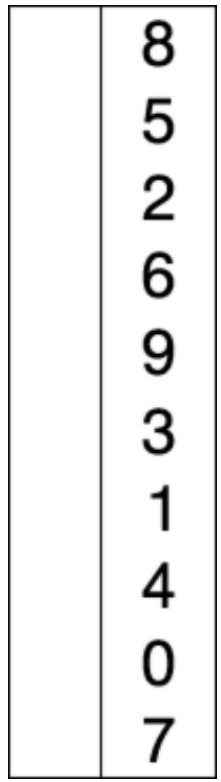
- Cas d'arrêt : la liste est vide
  - On retourne la liste vide

- Cas général récursif
  - On insère le premier élément de la liste dans la liste privée de son premier élément triée préalablement

Ecrire la fonction tri_insertion_rec (récursive)

```
def tri_par_insertion(liste):
    if liste == []:
        return []
    else:
        return inserer_elt_dans_liste_triee(liste[0],tri_par_insertion(liste[1:]))
```

Tri par sélection



- Idée : mettre le plus petit en premier
- Principe
  - Trouver le plus petit élément
  - Le substituer avec le premier élément du tableau
  - Trouver le 2nd plus petit élément
  - Le substituer avec le 2nd élément du tableau
  - …
- Combien de parcours du tableau ?
- Algorithme simple mais peu efficace sur de grandes quantités de données

Tri par sélection

- Conditions initiales
  - indice = 0
  - Un tableau vide est forcément trié !
  - tab[0 : indice] est trié

0	1	2	3	4	5
3	2	5	4	9	1

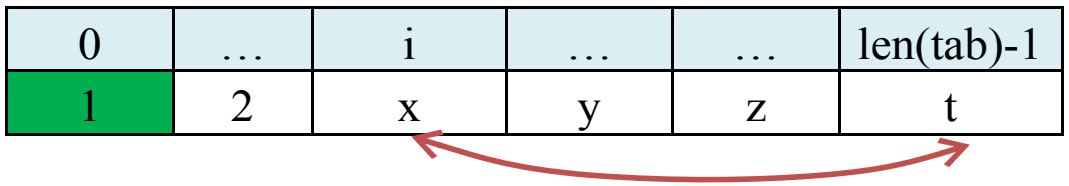
- Condition de boucle
  - Il reste au moins une valeur à placer

- Premier tour de boucle
  - On cherche la plus petite valeur dans tab[indice : len(tab)]
  - Substituer avec tab[indice]
  - indice = indice + 1
  - tab[0 : indice] est trié

0	1	2	3	4	5
1	2	5	4	9	3

Tri par sélection

- $i^{ème}$ tour de boucle
  - On cherche la plus petite valeur dans tab[indice : len(tab)]
  - Substituer avec tab[indice]
  - indice = indice + 1
  - tab[0 : indice] est trié



- Question : Etat du tableau après avoir exécuté les (n-1) tours de boucle ?

- Question : Condition d'arrêt ?
  - Sortie de la boucle car il n'y a plus de valeurs à classer
    - tab[0 : len(tab)] trié
  - Etat final à la sortie de la boucle
    - le tableau est entièrement trié

Tri par sélection non récursif

Ecrire la fonction *tri_selection* (non récursive)

```
def tri_selection(tab):
    lg = len(tab)
    if (lg == 0):
        return
    for indCourant in range(0, lg-1):
        indMin = indCourant
        for i in range(indCourant, lg):
            if tab[i]<tab[indMin]:
                indMin = i
        tab[indCourant],tab[indMin]=tab[indMin], tab[indCourant]
```

Tri par fusion (merge sort)

- Le tri par fusion est un algorithme qui propose de découper la liste en 2 parties de longueur égale (à un élément près) que l'on doit ensuite trier pour enfin fusionner ces 2 listes triées.

- La fonction récursive tri_par_fusion aura donc besoin de deux fonctions :
  - partager_liste_en_2 qui, étant donnée une liste, retourne un couple de listes de longueurs équivalentes formées des éléments de la liste à partager
  - fusionner_2_listes_triees qui, étant données 2 listes triées dans l'ordre croissant, construit dans l'ordre croissant la liste des éléments de ces 2 listes.

Partager la liste en 2

- Cas d'arrêt :
  - si la **liste est vide** on doit retourner le couple formé de 2 listes vides
  - si la **liste a un seul élément** on doit retourner le couple formé d'une liste vide et la liste à un élément

- Cas général récursif
  - si la **liste a au moins 2 éléments**, il faut partager en 2 la liste privée de ses 2 premiers éléments et ajouter ensuite, le 1er élément à la 1ère composante du couple, et le 2e élément à la 2e composante du couple.

```
def partager_liste_en_2(liste):
    if liste == []:
        return ([],[])
    else:
        if len(liste) == 1:
            return([],liste)
        else:
            (liste_1, liste_2) = partager_liste_en_2(liste[2:])
            return ([liste[0]]+liste_1,[liste[1]]+liste_2)
```

Fusionner 2 listes déjà triées

- Cas d'arrêt
  - si **au moins l'une des 2 listes est vide,** c'est fini, on retourne l'autre.

- Cas général récursif
  - si les **2 listes sont non vides**, on compare le 1er élément de chacune des 2 listes, on construit la liste avec le plus petit de ces 2 éléments ajouté en tête du résultat de la fusion de la liste privée de ce 1er élément et de l'autre liste inchangée.

```
def fusionner_2_listes_triees(liste_1, liste_2):
    if liste_1 == []:
        return liste_2
    else:
        if liste_2 == []:
            return liste_1
        else:
            if liste_1[0] <= liste_2[0]:
                liste = fusionner_2_listes_triees(liste_1[1:], liste_2)
                return([liste_1[0]]+liste)
            else:
                liste = fusionner_2_listes_triees(liste_1, liste_2[1:])
                return([liste_2[0]]+liste)
```

Tri par fusion

- Cas d'arrêt
  - si la **liste est vide**, c'est fini, on retourne la liste vide.

- Cas général récursif
  - si la **liste n'est pas vide**, on partage la liste en 2 listes, on trie ces 2 listes et on fusionne ensuite les 2 listes triées.

Préparation aux séances de TP

Plateforme PIXAL : les plus

- AUTONOMIE ! ! !
  - marche d'entrée minimale
  - pas d'installation nécessaire
  - n'importe quel appareil connecté
- compétence ciblée
- agilité numérique inclue dans l'apprentissage
- "gamifié" - ludique - gratifiant
- suivi pédagogique par groupe
- pour l'examen : alignement pédagogique (avoir interpréteur)

Plateforme PIXAL : les moins

- pas la "vraie vie"
- un peu trop dans l'essai-erreur qui manque de réflexion
- pas de contrôle de la "propreté" de l'écriture
- jeux de tests pas toujours "couvrants"
- obligation de connexion
- organisation examen ! ! !

Plateforme PIXAL : démo

http://pixal.univ-tlse3.fr

- Pour vous, extraits de la base de données des exercices proposés aux étudiants de L1
- Du temps pour faire le tour (accès aux exercices jusque juin 2020)
- Ne pas faire tous les exercices. Approfondir là où vous avez besoin
- Un répertoire "évaluation" qui reprend "un peu de chaque"

Préparation aux séances de TP - A faire pour jeudi

Messages d'erreur

- lister les messages d'erreurs que vous rencontrez dans le glossaire sur moodle
- jeudi, on les classifiera