

Chapitre # (ALGO) 4

Réversivité, Tri rapide et applications des tris

1 Généralités sur la programmation récursive.....

2 Utilisation de la programmation récursive.....

3 Tri récursif : le tri rapide (quicksort).....

4 Comparaison des tris & Applications

5 Solutions des exercices.....

Le flocon de KOCH est l'une des premières courbes fractales à avoir été décrites, bien avant l'invention du terme « fractale ».

— Le saviez-vous?

Résumé & Plan

La programmation récursive est un mode de programmation très différent de l'itératif, type de programmation rencontré jusque ici. Elle est adaptée par exemple à tous les objets mathématiques définis par récurrence. Les mathématiciens ont pu démontrer que : tout ce qui peut se programmer avec des boucles peut aussi se programmer avec des fonctions récursives, et inversement.

- Les chapitres d'Informatique sont composés de cours et d'exercices intégrés. Le cours sera projeté au tableau.
- Il n'est pas attendu que toute la classe aborde tous les exercices. Traitez donc en priorité les exercices présents dans la liste donnée à chaque début de séance.
- Exercices  / **Pour aller plus loin** : exercices plus difficiles, ou plus techniques. À ne regarder que si les autres sont bien compris.

1.

GÉNÉRALITÉS SUR LA PROGRAMMATION RÉCURSIVE

1.1. Présentation

Définition 1 | Fonction récursive

Une fonction informatique dont le résultat dépend d'elle-même est appelée *fonction récursive*. Une fonction non récursive est appelée *fonction itérative*.^a

- Un appel à la fonction dans le corps de ladite fonction s'appelle un *appel récursif*,
- l'ensemble des appels récursifs s'appelle la *pile d'exécution*.
- Chaque appel récursif s'appelle aussi un *empilement*, chaque **return** sans appel récursif s'appelle un *dépilement*.

Remarque 1 On parle de *pile d'exécution* à cause de la manière qu'à Python de gérer les appels récursifs : il les exécute selon le principe « premier arrivé, dernier exécuté », comme une pile d'assiettes. La dernière assiette posée sera aussi la première à être réutilisée ensuite.

⊗ Attention Une pile d'exécution est finie

Un nombre non fini d'appels récursifs conduira systématiquement à l'erreur `RecursionError: maximum recursion depth exceeded while ...`.

Exemple 1 (Pile non finie) Définir dans l'éditeur la fonction ci-après

```
def test():
    print("bonjour !")
    test() # réexécution de la fonction test
```

Dans cet exemple, les appels à la fonction `test()` s'enchaînent dans la pile d'exécution : il y a uniquement des empilements sans dépilement. L'accumulation de celles-ci provoque une erreur appelée *dépassement de pile* ou « *stack overflow* »

a. C'est ce type de fonction que nous avons utilisé jusqu'à maintenant

en anglais.

Exercice 1 | Fonction mystère *Solution* Sans taper le programme suivant dans un éditeur, essayez de prévoir le résultat de l'exécution du programme ci-dessous. Vérifiez votre hypothèse en exécutant le programme.

```
def mystere(n):
    if n == 0:
        return 0
    else:
        return n + mystere(n-1)
```

Expliquer le résultat renvoyé par `mystere(3)`, puis `mystere(10)`.

⊗ Attention

Il faut donc faire très attention au bon arrêt d'une fonction récursive, qui n'est pas automatique. Vous aurez quelques problèmes si dans la fonction précédente, vous omettez le test `if n > 0`, le même type d'erreur que dans la fonction test présentée en début de chapitre apparaîtra.

RÉCURSIF / ITÉRATIF : AVANTAGES ET INCONVÉNIENTS. Les fonctions récursives sont généralement plus simples à écrire, mais pas forcément plus rapides. La grande difficulté est cependant de bien comprendre les différents appels récursifs, et en quoi elle renvoie bien le bon résultat. Les inconvénients de la programmation récursive sont :

- la multiplication des mémoires allouées au stockage des résultats en attente, qui peut devenir rédhibitoire;
- le nombre de calculs effectués, souvent bien plus grand qu'en programmation itérative. Nous le constaterons dans un futur exercice.

1.2. Résumé : structure d'une fonction récursive

■ Structure type d'une fonction récursive

```
def f(par_1, par_2, ...):
    """
    fonction récursive
    """
    if condition arrêt:
        return valeur finale
```

```
instructions # dépend éventuellement de \
↪ f(par_1_prim,...)
return f(par_1_prim,...)
```

De manière générale, un algorithme est dit récursif quand sa mise en oeuvre utilise ce même algorithme. Pour être valide, cet algorithme doit impérativement vérifier les deux contraintes de terminaison :

1. existence d'un ou plusieurs cas de base où l'algorithme est directement effectif,
2. assurance qu'il n'y aura qu'un nombre fini d'appels récursifs avant de déboucher sur un cas de base.

2. UTILISATION DE LA PROGRAMMATION RÉCURSIVE

2.1. Pour les suites récurrentes

La conception d'une fonction récursive n'est pas éloignée du principe de démonstration par récurrence, et elle est bien adaptée pour décrire les objets mathématiques satisfaisant des relations de récurrence. Rappelons, au travers d'un exemple, les deux modes de définition d'une suite en Mathématiques.

Remarque 2 (Modes de définition d'une suite) Considérons une suite classique que vous connaissez bien : la factorielle, que l'on note $(u_n) = (n!)$. La suite (u_n) peut être définie en Mathématiques de deux manières.

1. **[Mode explicite]** pour tout $n \in \mathbb{N}$,

$$u_n = 1 \times 2 \times \cdots \times n = \prod_{k=1}^n u_k \quad (\text{convention } u_0 = 1)$$

2. **[Mode par récurrence]** pour tout $n \in \mathbb{N}^*$, (u_n) vérifie la relation de récurrence :

$$u_n = n \times u_{n-1}, \quad u_0 = 1.$$

Ces deux points de vue se transposent en Informatique : le mode de définition par récurrence correspond aux fonctions récursives, le mode explicite aux fonctions itératives.

Exemple 2 (Suite récurrente d'ordre 1) On considère la suite $(u_n)_{n \in \mathbb{N}}$ définie par :

$$u_0 = 1, \quad \forall n \in \mathbb{N}, \quad u_{n+1} = u_n^2 + u_n.$$

La suite est bien définie et la relation de récurrence peut se réécrire :

$$\forall n \geq 1, \quad u_n = u_{n-1}^2 + u_{n-1}.$$

La fonction suivante permet d'obtenir le n -ième terme de la suite.

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

Exemple 3 (Suite récurrente d'ordre 2) On considère la suite $(u_n)_{n \in \mathbb{N}}$ définie par : $u_0 = 1, u_1 = 2, \quad \forall n \in \mathbb{N}, \quad u_{n+2} = u_{n+1}^2 + u_n$. La suite est bien définie et la relation de récurrence peut se réécrire :

$$\forall n \geq 2, \quad u_n = u_{n-1}^2 + u_{n-2}.$$

La fonction suivante permet d'obtenir le n -ième terme de la suite. Cette fois-ci il faut bien indiquer deux conditions d'arrêt, une pour chaque « condition initiale » permettant ainsi les dépilements.

```
def u(n):
    if n == 0:
        return 1
    elif n == 1:
        return 2
    else:
        return u(n-1)**2 + u(n-2)
```

Une fois ces deux exemples bien compris, exercez-vous sur les suites ci-après.

Exercice 2 | Suite récurrente ordre 1 [Solution](#)

1. Écrire une fonction récursive puis itérative d'en-tête $u(n)$, qui calcule le $n^{\text{ème}}$ terme de la suite définie par :

$$u_0 = 2, \quad \forall n \in \mathbb{N}^*, \quad u_n = \frac{1}{2} \left(u_{n-1} + \frac{3}{u_{n-1}} \right).$$

2. Même question avec les suites $(v_n), (w_n)$ définies par :

$$\text{2.1)} \quad v_0 = 1, \quad \forall n \in \mathbb{N}, \quad v_{n+1} = 2v_n + 1,$$

$$\text{2.2)} \quad w_0 = 1, \quad \forall n \in \mathbb{N}, \quad w_{n+1} = 2w_n + n.$$

Exercice 3 | Suite récurrente ordre 2 [Solution](#) Écrire une fonction récursive d'en-tête $u(n)$, qui calcule le $n^{\text{ème}}$ terme de la suite définie par :

$$u_0 = 1, \quad u_1 = 0, \quad u_{n+2} = u_{n+1}^2 + 2u_n.$$

Exercice 4 | Suite récurrente ordre 1 : la factorielle [Solution](#) Nous avons déjà rencontré une version itérative dans le [Chapitre \(ALGO\) 1](#), la revoici.

```
def factorielle(n):
```

```
    """
```

```
    retourne la factorielle de n de manière itérative
```

```
    """
```

```
    P = 1
```

```
    for k in range(1, n+1):
```

```
        P *= k
```

```
    return P
```

Pour $n \in \mathbb{N}^*$, rappeler la relation reliant $n!$ à $(n-1)!$. Quel est le cas terminal? Écrire alors une fonction récursive d'en-tête `factorielle_rec(n)` qui calcule la factorielle de l'entier n . Que se passe-t-il lorsque l'on appelle `factorielle_rec(n)` pour $n < 0$? Pour $n = 989$?

Exercice 5 | Suite récurrente d'ordre 2 : FIBONACCI [Solution](#) On considère la suite (F_n) définie par : $F_0 = a, \quad F_1 = b, \quad F_{n+2} = F_{n+1} + F_n$, où $a, b \in \mathbb{R}$.

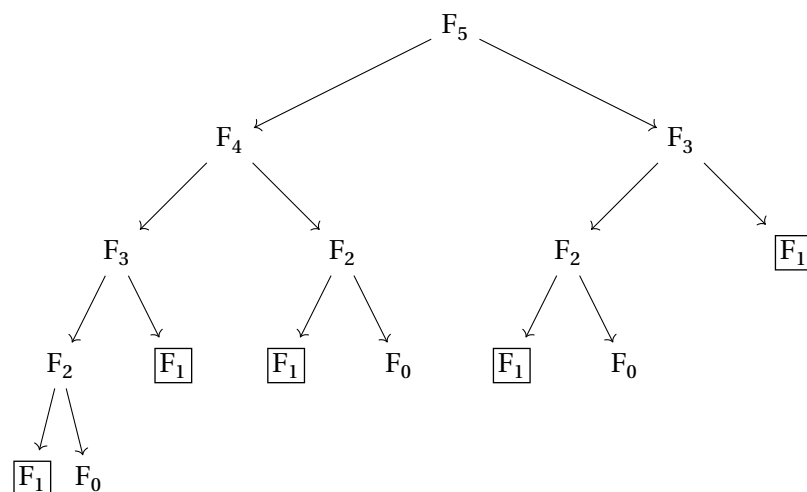
Écrire une fonction récursive d'en-tête `fibonacci(n, a, b)` qui calcule le $n^{\text{ème}}$ terme de la suite (F_n) . *Nous verrons une méthode itérative plus tard dans l'année, plus généralement pour n'importe quelle suite d'ordre 2.*

2.2. La récursivité, oui mais...

Nous avons déjà plusieurs exemples où il est possible de comparer l'efficacité de la version itérative et de la version récursive.

Une fonction récursive peut s'appeler à plusieurs rangs inférieurs, comme dans le cas par exemple d'une suite récurrente d'ordre 2. Analysons de plus près les appels récursifs opérés pour le calcul de la suite de FIBONACCI (F_n) , vue dans un précédent exercice.

Si ce code renvoie bien la valeur de F_n , il le fait de manière extrêmement maladroite puisque les deux appels récursifs générés à chaque étape conduisent à recalculer plusieurs fois de nombreux termes de la suite, comme le montre l'exemple de l'appel `fibonacci(5)` (on dispose à chaque fois sur la ligne d'en-dessous les appels générés par ceux de l'étape précédente, et pour plus de concision F_n est écrit au lieu de `fibonacci(n)` :



Vous constatez que F_1 par exemple a été évalué 5 fois ! On peut aussi évaluer la médiocre performance de la version récursive à l'aide du module `time` ; cela sera fait dans un prochain TP (??).

2.3. Pour l'algorithmique

Exercice 6 | Somme des éléments d'une liste, le retour *Solution* Soit L une liste de longueur n . Nous avons déjà vu dans le **Chapitre (ALGO) 2** comment calculer la somme des éléments d'une liste, de manière itérative.

1. Que représente l'élément $L[-1]$ pour une liste ? Que représente la liste $L[: -1]$ et quelle est sa taille ? *On pourra tester dans une console afin de confirmer sa réponse.*
2. Écrire une fonction d'en-tête `somme_rec(L)` renvoyant la somme des éléments d'une liste d'entiers L , en version récursive.

Exercice 7 | Appartenance à une liste, le retour *Solution* Écrire une fonction récursive `appartient_rec(e, L)` renvoyant **True** si l'élément e appartient à la liste L , **False** sinon, selon le principe suivant :

- si L est vide, alors on retourne **False**.
- Sinon, on regarde si l'élément est présent en tête de liste, si c'est le cas on retourne **True**, sinon on recommence à la liste privée de son premier élément.

*Pour rappel, nous avons également déjà vu cette fonction, de manière itérative, dans le **Chapitre (ALGO) 2**.*

Exercice 8 | Mots palindromiques *Solution* On rappelle qu'un mot est un *palindrome* s'il est égal à lui-même quand on l'écrit de la droite vers la gauche. Par exemple, les mots "kayak" et "ressasser" sont des palindromes, le mot "professeur" ne l'est pas.

1. Écrire une fonction itérative d'en-tête `palindrome(mot)` renvoyant **True** si la chaîne de caractères `mot` est un palindrome, **False** sinon. Par exemple, `palindrome("kayak")` doit ainsi renvoyer **True**, `palindrome("professeur")` doit renvoyer **False**.
2. Même question, mais avec une version récursive.

Exercice 9 | Exponentiation naïve et rapide *Solution*

1. Écrivez une fonction récursive d'en-tête `expo_rec(x, n)` renvoyant x^n pour n entier naturel, en exprimant la définition sous la forme :

$$x^n = \begin{cases} 1 & \text{si } n = 0, \\ x \times x^{n-1} & \text{si } n \in \mathbb{N}^*. \end{cases}$$

Combien de multiplications sont-elles nécessaires pour effectuer le calcul de x^n avec cette version ? *On notera c_n ce nombre, et on cherchera une relation de récurrence sur la suite (c_n)*

2. Dans l'exemple précédent, comme sur celui de la factorielle, l'appel récursif se fait en diminuant la variable n de 1. Il est parfois possible de la réduire davantage (pour minimiser le nombre d'appels nécessaires), et souvent de la diviser par 2, en utilisant un principe *dichotomique*.

Par exemple,

- $x^{13} = x \cdot x^{12} = x \cdot (x^6)^2$. On peut donc se contenter de calculer x^6 , suivi de deux multiplications.
- Reste à appliquer la même idée sur x^6 : $x^6 = (x^3)^2$.
- Reste à appliquer la même idée sur x^3 : $x^3 = x \cdot x^2$.

Le principe précédent, récursif, repose donc sur les égalités ci-après :

$$x^n = \begin{cases} (x * x)^{n//2} & \text{si } n \text{ est pair} \\ (x * x)^{n//2} * x & \text{si } n \text{ est impair} \end{cases}$$

- 2.1) Écrire une fonction récursive d'en-tête `expo_rapide_rec(x, n)` renvoyant x^n pour n entier naturel selon le principe précédent.
On rappelle que le quotient de la division euclidienne de n par 2 s'obtient par $n//2$ et qui est égale à $\frac{n}{2}$ si n est pair et à $\frac{n-1}{2}$ si n est impair.
- 2.2) Afin de bien vous approprier les fonctionnements, listez à la main les appels successifs engendrés par `expo_rapide_rec(2, 10)`.
- 2.3) On souhaite évaluer dans cette question le nombre de multiplications de l'exponentiation rapide. Écrire une fonction d'en-tête

`expo_rapide_rec_mult(n, m)` qui retourne le nombre de multiplications requises par la méthode d'exponentiation rapide appliquée à x^n , le paramètre m jouant le rôle de compteur de multiplications ((il est nécessaire de le passer en argument de la fonction, $m = 0$ en début de fonction récursive étant problématique)) La tester pour $n = 20, 50, 100$. Commentez.

2.4. Pour les dessins récursifs & fractales

Exercice 10 | Des étoiles plein les yeux [Solution](#)

1. Que va produire l'appel de `etoiles1(5)` pour la fonction récursive suivante? Essayez de le deviner d'abord sans taper la fonction, puis faites-le pour contrôler.

```
def etoiles1(n):
    if n > 0:
        print("*"*n)
        etoiles1(n-1)
```

2. Même question pour l'appel de `etoiles2(5)`:

```
def etoiles2(n):
    if n > 0:
        etoiles2(n-1)
        print("*"*n)
```

Avant de tracer une fractale, commençons par nous familiariser avec un module de dessin dynamique : le module `turtle`.

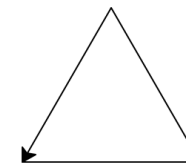
Exercice 11 | Apprendre à connaître le module `turtle` [Solution](#)

1. Exécuter les commandes ci-après dans la console, et annoter le sujet une fois leur effet constaté.

```
from turtle import *
fenetre = Screen()
fenetre.setup(400,800)
fenetre.setup(800,800)
forward(100)
left(90)
forward(50)
width(5)
forward(50)
color('green')
right(60)
```

```
forward(50)
up() # arrête le tracé
p,o = position(), heading()
p
o
setposition(-200,0)
p,o = position(), heading()
p
o
down() # remets le tracé
```

2. Dessiner à l'aide de `turtle` le triangle équilatéral ci-contre (de longueur de côté arbitraire).



DESSIN DYNAMIQUE, ET APPLICATION À LA FRACTALE DE KOCH. De manière générale, une figure fractale est un objet mathématique qui présente une structure similaire à toutes les échelles. C'est un objet géométrique « infiniment morcelé » dont des détails sont observables à une échelle arbitrairement choisie.

Exemple 4

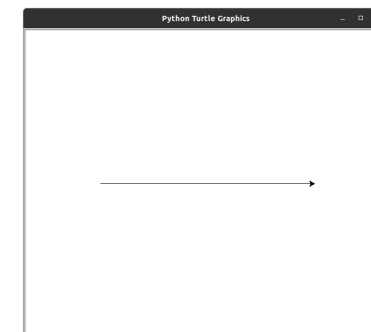
Il existe de nombreuses figures fractales dans la nature. Par exemple, le chou Romanesco.



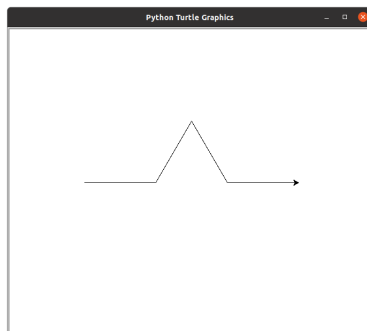
Une initiation à la récursivité ne saurait être complète sans un tracé de courbe de KOCH. Elle consiste :

•

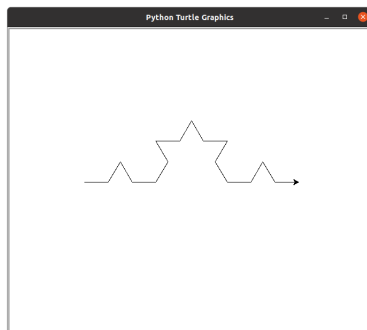
à la profondeur 0 à tracer un simple segment d'une longueur ℓ donnée (notez que le petit triangle au bout du segment est dû au module `turtle` que nous avons découvert précédemment).



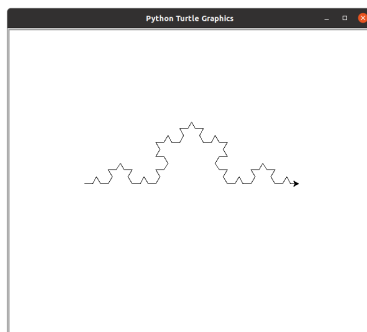
à la profondeur 1, on remplace le segment précédent par la ligne brisée suivante (chacun des segments de cette ligne a pour longueur le tiers de celui du segment initial).



à la profondeur 2, chacun des segments de la figure précédente est elle-même remplacée par une figure similaire à l'échelle correspondante et orientée dans sa direction.



et ainsi de suite, à la profondeur 3 :



Exercice 12 | Tracé complet de la fractale [Solution](#) Le code ci-dessous vous permet de tracer la ligne correspondant à la profondeur 1 pour une longueur ℓ valant ici 400.

```
from turtle import *
```

```
resetscreen() # efface l'écran
setposition(-200,0)
```

```
# tracé de la courbe en profondeur 1 pour une longueur 400
```

```
 $\ell$  = 400
```

```
forward( $\ell/3$ ) # en avant
```

```
left(60)
```

```
forward( $\ell/3$ ) # en avant
```

```
right(120)
```

```
forward( $\ell/3$ ) # en avant
```

```
left(60)
```

```
forward( $\ell/3$ ) # en avant
```

```
exitonclick()
```

Écrire une fonction récursive d'en-tête `koch( $\ell$ , p)` effectuant le tracé pour une longueur ℓ et à une profondeur p .

3. TRI RÉCURSIF : LE TRI RAPIDE (QUICKSORT)

LE PRINCIPE : PARTAGER LA LISTE EN DEUX SOUS-LISTES D'ÉLÉMENTS INFÉRIEURS/SUPÉRIEURS OU ÉGAUX À UN ÉLÉMENT CHOISI AU HASARD, PUIS RECOMMENCER AVEC LES DEUX SOUS-LISTES. Le tri rapide est un algorithme récursif basé sur le principe « diviser pour régner ». Étant donnée une liste L , on commence par choisir le premier élément comme *pivot*, et on sépare la liste entre deux sous-listes : la première contient des éléments inférieurs ou égaux au pivot, et la seconde contient des éléments supérieurs (strictement) au pivot. Puis on applique le tri rapide aux deux sous-listes obtenues.

Exercice 13 | Programmation du tri rapide [Solution](#)

1. Écrire alors une fonction récursive `tri_rapide_rec(L)` qui trie une liste L avec le tri rapide, et de manière récursive. On complètera, par exemple, le code ci-dessous.

■ ■ Tri Rapide, version récursive

```
def tri_rapide_rec(L):
```

```
    if len(L) == 0:
```

```
        # Condition d'arrêt
```

```
        return []
```

```

pivot = L[0]
inf_pivot = []
sup_pivot = []
for x in L[1:]:
    if _____:
        _____
    else:
        _____
return _____ # on recommence

```

- Proposer une autre version où inf_pivot et sup_pivot sont complétées par compréhension.
- Ce tri a-t-il lieu en place? Est-il itératif? récursif? comparatif?
- Comment faire pour obtenir une liste triée par ordre décroissant?

Il est possible aussi de programmer le tri rapide de manière itérative, mais le principe même de ce tri invite à utiliser de la récursivité.

4. COMPARAISON DES TRIS & APPLICATIONS

4.1. Comparaison

Nous allons dans cette partie mesurer empiriquement les temps d'exécution de chacun des tris.

■ Générer une liste de 10 entiers, et les mélanger

```

>>> import random as rd
>>> n = 10
>>> L = list(range(n))
>>> rd.shuffle(L) #Mélange des éléments de L en place
>>> L
[0, 3, 1, 8, 7, 4, 9, 6, 2, 5]

```

On décide à présent de comparer tous les tris présents dans la liste fonctions ci-dessous. On a :

- Une fonction qui génère des listes d'entiers (fonction genere_liste),
- pour chaque tri, on : génère une liste d'entiers de taille n, puis on chronomètre l'exécution du tri et on effectue une moyenne des temps sur 10**2 essais.

- Enfin, on trace le temps d'exécution moyen en fonction de l'entier n (taille de la liste).

```

fonctions = [tri_insertion, tri_selection, tri_bulles, \
↳ tri_rapide_rec]
noms = {tri_insertion : "tri insertion", tri_selection : "tri
sélection", tri_bulles : "tri à bulles", tri_rapide_rec : "tri
rapide"} # affichage d'une légende

```

```

import matplotlib.pyplot as plt
import random as rd

```

```

def genere_liste(n):
    """
    retourne une liste d'entiers aléatoires entre 1 et n
    """
    L = list(range(n))
    rd.shuffle(L)
    return L

```

```

import time as ti

```

```

def eval_temps_tri(n, nom_du_tri):
    """
    retourne le temps d'exécution moyen pour le tri nom_du_tri
    """
    somme = 0
    for _ in range(10**2):
        L = genere_liste(n)
        t_1 = ti.time()
        nom_du_tri(L)
        somme += ti.time()-t_1
    return somme/10**2

```

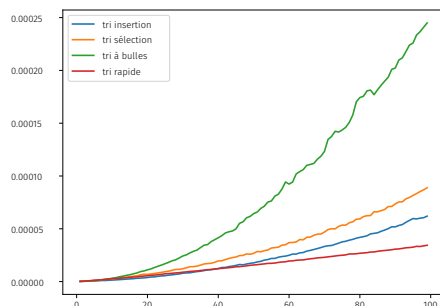
```

def trace_temps():
    X = list(range(1, 100))
    Y = []
    for nom_du_tri in fonctions:
        for n in X:
            Y.append(eval_temps_tri(n, nom_du_tri))
    plt.plot(X, Y, label = noms[nom_du_tri])

```

```
Y = []
plt.legend()
```

```
trace_temps()
```



On constate que c'est le tri rapide, qui semble être le meilleur en terme de temps d'exécution.

4.2. Applications des tris

4.2.1. Calcul d'une médiane

Exercice 14 | [Solution](#) Écrire une fonction d'en-tête `mediane(L)` qui étant donnée une série statistique correspondante à `L` retourne la médiane de cette série. Indication : Si vous avez oublié ce qu'est une médiane, documentez-vous sur internet

4.2.2. Recherche dichotomique dans une liste triée

Le mot *dichotomie* signifie « division, opposition » (entre deux éléments, deux idées). Nous rencontrerons ce mot plusieurs fois au cours de l'année, une technique similaire sera déployée en Mathématiques pour approcher des points d'annulation de fonctions.

LE PRINCIPE. On s'intéresse ici à la recherche d'un élément noté `x` dans un tableau ou une liste dans laquelle les éléments de même type ont été préalablement triés par ordre croissant. Cette situation, bien qu'*a priori* particulière, se rencontre fréquemment en Informatique.

Dans un précédent TP, pour un tableau ou une liste non trié `L`, on a vu que l'on pouvait employer une méthode « par balayage » en utilisant un parcours simple de `L`. Cet algorithme dit « naïf » a pour propriétés :

- il s'applique à tout tableau ou liste `L`, sans nécessiter un ordre particulier entre les éléments de `L`,
- il met en jeu n tests dans le pire des cas, ce qui ne le place pas parmi les algorithmes très rapides.

Nous allons nous intéresser ici uniquement à des tableaux ou listes triés par ordre croissant, et appliquer un principe de recherche dichotomique qui conduit à un résultat avec beaucoup moins de comparaisons. La recherche par dichotomie de l'élément `x` dans `L`, consiste de manière itérative à :

1. initialiser deux indices `i_g = 0` (comme *indice gauche*), et `i_d = len(L) - 1` (comme *indice droite*).
2. Calculer alors `i_m = int((i_g+i_d)/2)`. L'élément `L[i_m]` est alors un élément *au milieu* de la liste `L`.
 - Si `L[i_m] = x`, l'algorithme est terminé.
 - Si `L[i_m] < x`, on recommence le processus en changeant `i_g` en `i_d = i_m + 1`. On a « tapé trop à gauche ».
 - Si `L[i_m] > x`, on recommence le processus en changeant `i_d` en `i_d = i_m - 1`. On a « tapé trop à droite ».
3. On arrête le processus lorsque `x` a été trouvé ou alors lorsque `i_g > i_d` (si `x` n'est pas présent, au dernier changement d'indice l'ordre entre les deux variables est inversé à cause du ± 1).

Pourquoi cet algorithme est beaucoup plus rapide? Car les tailles des sous-listes où l'on recommence la recherche diminue très vite (divisée par deux à chaque étape). En revanche, le tri utilisé coûte un peu en temps, on ne peut pas gagner sur tous les tableaux.

Exercice 15 | **Appartenance d'un élément dans une liste. Méthode itérative par dichotomie.** [Solution](#)

1. Étant donnée une liste `L`, écrire une fonction `recherche_dicho(x, L)` implémentant ce principe. Indication : On pourra compléter le script à trous ci-après

```
def recherche_dicho(x, L):
    """
    recherche un élément par méthode dichotomique
    """
    i_g = 0
    i_d = _____
```

```

while _____:
    i_m = _____
    if L[i_m] == x:
        return _____
    elif L[i_m] < x:
        i_g = i_m+1
    else:
        i_d = _____
return False # x non trouvé dans L

```

2. Que se passe-t-il avec `i_g`, `i_d` lorsque `x` n'est pas présent dans `L`?

COMPARAISON AVEC LA MÉTHODE PAR BALAYAGE. On peut comparer, comme nous l'avons fait pour les tris, l'efficacité de la recherche par balayage et de la recherche dichotomique.

```

fonctions = [appartient, recherche_dicho]
noms = {appartient : "balayage", recherche_dicho : "dichotomie"} \
↳ # affichage d'une légende
import time as ti

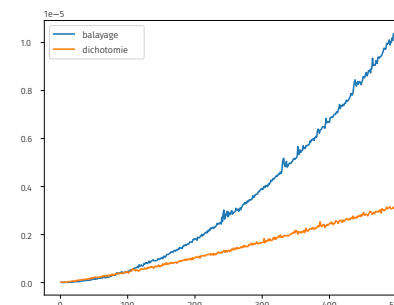
def eval_temps_recherche(n, nom_methode):
    """
    retourne le temps d'exécution moyen pour le tri nom_du_tri
    """
    somme = 0
    L = list(range(n))
    for x in range(n):
        t_1 = ti.time()
        nom_methode(x, L)
        somme += ti.time() - t_1
    return somme/10**2

def trace_temps():
    X = list(range(1, 500))
    Y = []
    for nom_methode in fonctions:
        for n in X:
            Y.append(eval_temps_recherche(n, nom_methode))
    plt.plot(X, Y, label = noms[nom_methode])
    Y = []

```

```
plt.legend()
```

```
trace_temps()
```



On constate que la recherche dichotomique semble être plus rapide.

5. SOLUTIONS DES EXERCICES

Solution (exercice 1) Énoncé Lors de chaque appel récursif `mystere(n-1)`, l'entier n est abaissé de 1, jusqu'à au final valoir 0. C'est alors à ce moment-là que la fonction s'arrête. On somme au fur et à mesure chaque résultat donc à la fin la fonction retournera $S = n + (n-1) + \dots + 1$ soit l'entier $\frac{n(n+1)}{2}$.
Par exemple, la valeur renvoyée par `mystere(3)` est 6. Essayons de comprendre en détail ce qui permet d'aboutir à ce résultat :

1. il y a un premier empilement qui correspond au 1er appel de la fonction `mystere()` avec le paramètre $n = 3$.
2. Comme $n > 0$, il y a donc un 2ème appel de la fonction `mystere()` avec le paramètre $n = 2$ et donc un 2ème empilement.
3. Comme $n > 0$, il y a donc un 3ème appel de la fonction `mystere()` avec le paramètre $n = 1$ et donc un 3ème empilement.
4. Comme $n > 0$, il y a donc un 4ème appel de la fonction `mystere()` avec le paramètre $n = 0$ et donc un 4ème empilement.
5. Comme $n = 0$, l'appel $n^o 4$ de la fonction `mystere()` renvoie $S = 0$ et il y a dépilement.
6. Le 3ème appel (avec $n = 1$) peut alors poursuivre son exécution et renvoie la valeur $1+0$, c'est-à-dire 1. Comme l'exécution de ce 3ème appel est terminée, il y a à nouveau dépilement.
7. Le 2ème appel (avec $n = 2$) peut alors poursuivre son exécution et renvoie la valeur $2+1$, c'est-à-dire 3. Comme l'exécution de ce 2ème appel est terminée, il y a dépilement.
8. Le 1er appel (avec $n = 3$) peut alors poursuivre son exécution et renvoie la valeur $3+3$, c'est-à-dire 6.

Finalement, la valeur renvoyée par `mystere(3)` est bien 6.

Solution (exercice 2) Énoncé

```
def u(n):
    if n == 0:
        return 2
    return 1/2*(u(n-1)+3/u(n-1))

def u_it(n):
    u = 2
    for _ in range(1, n+1):
        u = 1/2*(u+3/u)
    return u
```

```
def v(n):
    if n == 0:
        return 1
    return 2*v(n-1)+1

def v_it(n):
    u = 1
    for _ in range(1, n+1):
        u = 2*u + 1
    return u

def w(n):
    if n == 0:
        return 1
    return 2*w(n-1)+n-1

def w_it(n):
    u = 1
    for i in range(1, n+1):
        u = 2*u + i-1
    return u

>>> u(4)
1.7320508075688772
>>> u_it(4)
1.7320508075688772
>>> v(4)
31
>>> v_it(4)
31
>>> w(4)
27
>>> w_it(4)
27
```

Solution (exercice 3) Énoncé

```
def u(n):
    if n == 0:
        return 1
    if n == 1:
```

```

    return 0
    return u(n-1)**2 + 2*u(n-2)

```

Solution (exercice 4) [Énoncé](#)

```

def factorielle_rec(n):
    """
    Retourne la factorielle de n, de manière récursive
    """
    if n == 0:
        return 1
    return n*factorielle_rec(n-1)

```

Solution (exercice 5) [Énoncé](#)

■ Version récursive

```

def fibo(n, a, b):
    if n == 0:
        return a
    elif n == 1:
        return b
    else:
        return fibo(n-1,a,b) + fibo(n-2,a,b)
>>> fibo(5, 1, 1)
8

```

Solution (exercice 6) [Énoncé](#)

```

>>> L = [1, 3, -1, 2]
>>> L[-1] # dernier élément
2
>>> L[:-1] # liste extraite jusqu'à l'avant-dernier élément
[1, 3, -1]

```

Pour la fonction `somme_rec`, on peut extraire le dernier élément de la liste et l'ajouter à `somme_rec` exécutée sur la liste `L[:-1]`. La condition d'arrêt sera : lorsque la liste est nulle, on retourne 0.

```

def somme_rec(L):
    if len(L) == 0:
        return 0
    else:
        return somme_rec(L[:-1]) + L[-1]
>>> L = [1, 3, -1, 2]

```

```

>>> somme_rec(L)
5

```

Solution (exercice 7) [Énoncé](#) Principe similaire ici pour la fonction `appartient_rec`. L'arrêt est cependant ici : le cas d'une liste vide (e non trouvé), ou alors celui d'une liste avec e présent en dernier élément (on a donc trouvé e).

```

def appartient_rec(e, L):
    if len(L) == 0:
        return False # élément non trouvé
    elif L[0] == e:
        return True
    else:
        return appartient_rec(e, L[1:])
>>> L = [1, 3, -1, 2]
>>> appartient_rec(-1, L)
True
>>> appartient_rec(-2, L)
False

```

Solution (exercice 8) [Énoncé](#) Aspect logique : un mot n'est pas un palindrome si et seulement si il existe i tel que $\text{mot}[i] \neq \text{mot}[-1-i]$ où n est la longueur du mot.

```

def palindrome(mot):
    for i in range(len(mot)):
        if mot[i] != mot[-1-i]:
            return False
    return True

```

Pour la version récursive, on peut analyser si la première lettre est égale à la dernière, si c'est le cas on recommence le processus avec `mot[1:-1]` (mot privé de la 1ère et la dernière lettre).

```

def palindrome_rec(mot):
    if len(mot) == 0:
        return True
    elif mot[0] != mot[-1]:
        return False
    else:
        return palindrome_rec(mot[1:-1])
>>> palindrome("kayak")
True

```

```
>>> palindrome_rec("kayak")
True
>>> palindrome("professeur")
False
>>> palindrome_rec("professeur")
False
```

Solution (exercice 9) [Énoncé](#)

```
def expo_rec(x, n):
    if n == 0:
        return 1
    else:
        return x*expo_rec(x, n-1)
```

Si on note c_n le nombre de multiplications, alors $c_n = 1 + c_{n-1}$ et $c_0 = 0$, donc $c_n = n$.

```
def expo_rapide_rec(x,n):
    if n == 0:
        return 1
    else:
        if n%2 == 0:
            return expo_rapide_rec(x,n//2)**2
        else:
            return expo_rapide_rec(x,n//2)**2*x
```

```
>>> expo_rec(2, 4)
16
>>> expo_rapide_rec(2, 4)
16
```

```
def expo_rapide_rec_mult(n, m):
    if n == 0:
        return m
    else:
        if n%2 == 0:
            return expo_rapide_rec_mult(n//2, m+1)
        else:
            return expo_rapide_rec_mult(n//2, m+2)
```

```
>>> expo_rapide_rec_mult(20, 0)
7
>>> expo_rapide_rec_mult(50, 0)
9
```

```
>>> expo_rapide_rec_mult(100, 0)
10
```

On constate un nombre de multiplications bien inférieur à la première méthode! Qui nécessiterait, pour $n = 100$, 99 multiplications (au lieu de 10 ici).

Solution (exercice 10) [Énoncé](#)

```
>>> etoiles1(5)
*****
****
***
**
*

>>> etoiles2(5)
*
**
***
****
*****
```

Solution (exercice 11) [Énoncé](#)

```
from turtle import *
fenetre = Screen()
t = Turtle()
t.forward(120)
t.left(120)
t.forward(120)
t.left(120)
t.forward(120)
```

Solution (exercice 12) [Énoncé](#)

```
def koch(n, l):
    resetscreen() # efface l'écran
    setposition(-200,0)
    if n == 0 :
        forward(l)
    else :
        koch(n-1,l/3)
        left(60)
        koch(n-1,l/3)
        right(120)
```

```
koch(n-1,l/3)
left(60)
koch(n-1,l/3)
```

Solution (exercice 13) [Énoncé](#) Le second programme est à associer à la seconde question.

```
def tri_rapide_rec(L):
    """
    Trie la liste L selon le tri rapide (Non en place)
    """
    if len(L) == 0:
        return []
    pivot = L[0]
    inf_pivot = []
    sup_pivot = []
    for x in L[1:]:
        if x < pivot:
            inf_pivot.append(x)
        else:
            sup_pivot.append(x)
    return tri_rapide_rec(inf_pivot) + [pivot] + \
        tri_rapide_rec(sup_pivot)

def tri_rapide_rec_aux(L):
    """
    Trie la liste L selon le tri rapide (Non en place)
    """
    if len(L) == 0:
        return []
    pivot = L[0]
    inf_pivot = [L[i] for i in range(1, len(L)) if L[i] < \
        tri_rapide_rec_aux(L)]
    sup_pivot = [L[i] for i in range(1, len(L)) if L[i] >= \
        tri_rapide_rec_aux(L)]
    return tri_rapide_rec(inf_pivot) + [pivot] + \
        tri_rapide_rec(sup_pivot)
```

Le tri présenté ici est récursif, non en place car une nouvelle liste est créée dans le **return**. Pour trier dans l'ordre décroissant, on peut par exemple modifier l'ordre des listes dans le **return** :

```
def tri_rapide_dec_rec(L):
```

```
"""
Retourne une liste triée par ordre croissant des éléments \
↳ de L,
selon le tri rapide. Version récursive.
Tri non en place
"""
if L == []:
    return []
pivot = L[0]
inf_pivot = []
sup_pivot = []
for x in L[1:]:
    if x < pivot:
        inf_pivot.append(x)
    else:
        sup_pivot.append(x)
return tri_rapide_rec(sup_pivot) + [pivot] + \
    tri_rapide_rec(inf_pivot)

>>> L = [1, -4, 7, 4, 2]
>>> tri_rapide_rec(L)
[-4, 1, 2, 4, 7]
>>> tri_rapide_dec_rec(L)
[2, 4, 7, 1, -4]
```

Solution (exercice 14) [Énoncé](#)

```
from tri_rapide_rec import *
def mediane(L):
    """
    Cherche la médiane d'une liste, après tri rapide des \
    ↳ observations
    """
    L_tri = tri_rapide_rec(L)
    n = len(L)
    if n % 2 == 1:
        # Nombre impair d'observations
        return L_tri[n//2]
    else:
        # Nombre pair d'observations
        return (L_tri[n//2-1] + L_tri[n//2])/2
```

Solution (exercice 15) [Énoncé](#)

```
def recherche_dicho(x, L):  
    """  
    recherche un élément par méthode dichotomique  
    L est supposée triée  
    """  
    i_g = 0  
    i_d = len(L)-1  
    while i_g <= i_d:  
        i_m = int((i_g + i_d)/2)  
        if L[i_m] == x:  
            return True  
        elif L[i_m] < x:  
            i_g = i_m+1  
        else:  
            i_d = i_m-1  
    return False # x non trouvé dans L  
>>> L = [1, 2, 4, 5]  
>>> recherche_dicho(2, L)  
True  
>>> recherche_dicho(4, L)  
True
```

Lorsque x n'est pas présent, on a à la fin $i_g > i_d$.