1 Exercices des recursivité

1.1 Exercice 1 : Fonction factorielle

1.1.1 En utilisant les propriétés suivantes,

- $0! = 1$
- $(n+1)! = (n+1) \times n!$

écrire en Python une fonction `fac(n)` qui renvoie le factoriel d'un nombre entier positif n passé en paramètre. On donnera un algorithme récursif.

[ ]:
```
def fac(n) : # version récursive
```

[2]:
```
# Validation 5! = 120 ?
print(fac(5))
```

```
120
```

1.1.2 Version iterative

Proposer une version itérative de la fonction factorielle, `facIter(n)`.

[ ]:
```
def facIter(n) :  # version itérative de factorielle
```

[4]:
```
print(facIter(5))
```

```
120
```

1.1.3 Complexités temps et espace

Comparer les complexités en temps et en espace de ces algorithmes.

1.1.4 Récursivité terminale

Proposer une version en **récursivité terminale** de la fonction factorielle sur le modèle `factTerm(n,res = 1)`.

Rappel : une fonction récursive est dite *en récursivité terminale* si ** il n'y a aucun traitement entre l'appel récursif et le retour de la fonction**. Ainsi, l'appel récursif doit coïncider exactement à l'instruction `return`.

Par exemple : l'instruction `return n*f(n-1)` nécessite de conserver la valeur de la variable n en attendant que le resultat de la fonction $f(n-1)$ soit retournée. Ainsi, un algorithme contenant cette instruction n'est donc pas un algorithme en récursivité terminale.

L'idée est d'utiliser un second argument dans l'appel de la fonction qui sert à stocker le résultat des calculs intermédiaires. Un peu comme dans la version itérative, ce second argument, ici noté *res*, est multiplié progressivement de manière à

construire la valeur finale qui sera renvoyée par la fonction au niveau de l'appel de terminaison.

[ ]:
```
def factTerm(n, res = 1) # version en récursivité terminale de la factorielle
```

[6]:
```
facTerm(5)
```

[6]:
```
120
```

1.2 Exercice 2 : récursivité mutuelle

On considère les deux suites (u_n) et (v_n) définies par $u_0 = 1$ et $v_0 = -1$ et

$$\forall n \in \mathbb{N} \quad \begin{cases} u_{n+1} &= 3u_n - \frac{1}{2}v_n - 2 \\ v_{n+1} &= -\frac{2}{3}u_n + 3v_n \end{cases}$$

Ecrire deux fonctions en langage Python `suiteU(n)` et `suiteV(n)` qui traduisent la définition par récurrence des suites (u_n) et (v_n) donnée ci-dessus.

Tester votre algorithme en affichant les 8 premiers termes de la suite (u_n).

[ ]:
```
# Recursivité mutuelle
def suiteU(n):

def suiteV(n):
```

[8]:
```
# Validation
for k in range (8): # k varie de zéro à 7 soit 8 valeurs
    print('u ( ',k,' ) = ',suiteU(k))
```

```
u (  0  ) =  1
u (  1  ) =  1.5
u (  2  ) =  4.333333333333333
u (  3  ) =  17.0
u (  4  ) =  68.44444444444444
u (  5  ) =  267.3333333333333
u (  6  ) =  1014.8148148148148
u (  7  ) =  3776.0
```

1.3 Exercice 3 : suite de Fibonacci

On considère la suite de Fibonacci $\mathcal{F}_n$définie par:

$\mathcal{F}_0 = 0, \mathcal{F}_1 = 1$, et $\forall n > 1, \mathcal{F}_n = \mathcal{F}_{n-1} + \mathcal{F}_{n-2}$

1.3.1 Fibonacci récursive

Ecrire une fonction Pyhton `fibo(n)` qui calcule le nième de la suite à partir de l'entier n passé en argument en utilisant un algorithme récursif. Calculer la valeur $\mathcal{F}_{15}$

[ ]:
```
def fibo(n): # version récursive
```

[ ]:
```
for k in range(16):
    print(k,fibo(k))
```

1.3.2 Fibonacci : comptage du nombre d'additions.

Modifier le code précédent de manière à afficher le nombre total d'additions exécutées, noté $C(n)$, correspondant à la complexité en temps. Faire afficher $C(15)$. Indication : on utilisera une variable global nommée `nbAdd`.

[6]:
```
nbAdd = 0
fibo(15)
print(nbAdd)
```

```
986
```

1.3.3 Expression théorique de la complexité* Démontrer que la complexité en nombre d'additions est donné par :

$$\forall n \in \mathbb{N}, \quad C(n) = \frac{\varphi^{n+1}}{\sqrt{5}} - \frac{(-\varphi)^{-(n+1)}}{\sqrt{5}} - 1$$

où $\varphi = \frac{1+\sqrt{5}}{2}$ (aussi appelé "nombre d'or").

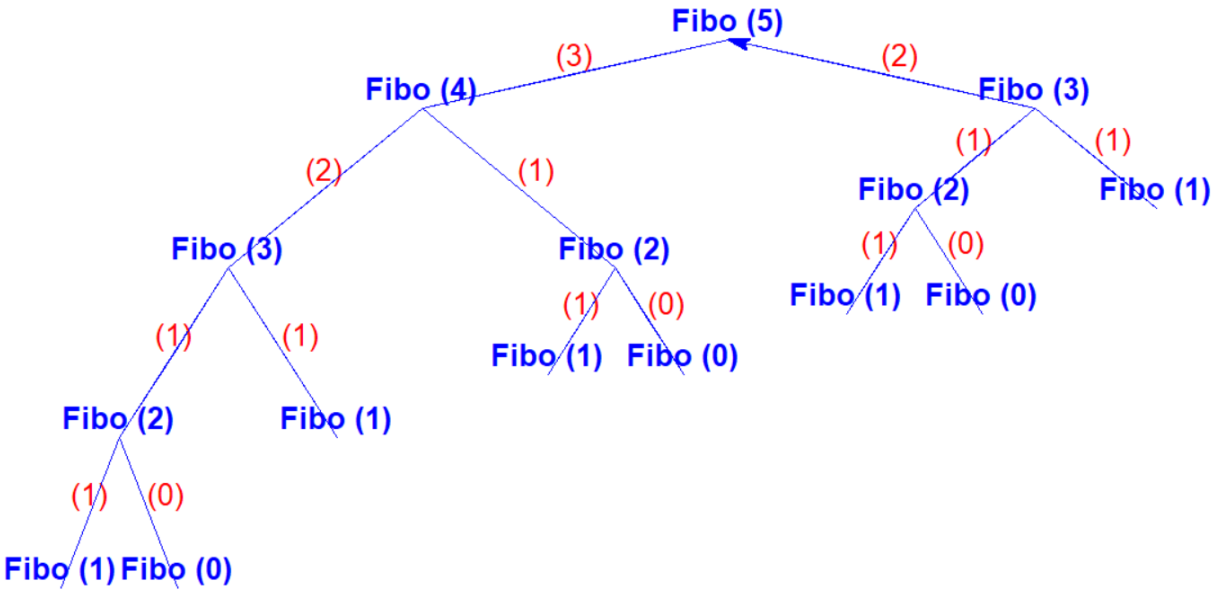
Vérifier numériquement la validité de cette expression pour $n = 6$. On définira une fonction `C(n)` qui renvoie la valeur numérique de la complexité pour un argument `n` entier.

[8]:
```
import numpy as np # fonctions mathématiques
phi = (1 + np.sqrt(5))/2
def C(n):
    return # A COMPLETER
```

[9]:
```
# Validation numérique de l'expression donnée pour n=15
C(15)
```

[9]:
```
986.0000000000005
```

Visualisation graphique de l'algorithme récursif. Avec la bibliothèque Python `turtle`.



1.3.4 Fibonacci itératif

Proposer une **version non récursive** d'une fonction calculant le n-ième terme de la suite de Fibonacci, `fiboIter(n)`.

L'idée est d'utiliser deux variables pour mémoriser les valeurs précédentes. Les variables sont initialisées à $\mathcal{F}_0$, et $\mathcal{F}_1$.

Pour $n > 1$, on effectue $n - 1$ itérations de manière à recalculer la valeur de $\mathcal{F}_n$ et à stocker la valeur $\mathcal{F}_{n-1}$ qui servira pour le calcul suivant.

Attention à ne pas se tromper sur le nombre d'itérations à effectuer : calculer $\mathcal{F}_2$ demande d'effectuer une seule somme, c'est-à-dire d'appliquer une seule fois la relation de récurrence.

[ ]:
```
def fiboIter(n) : # version itérative
```

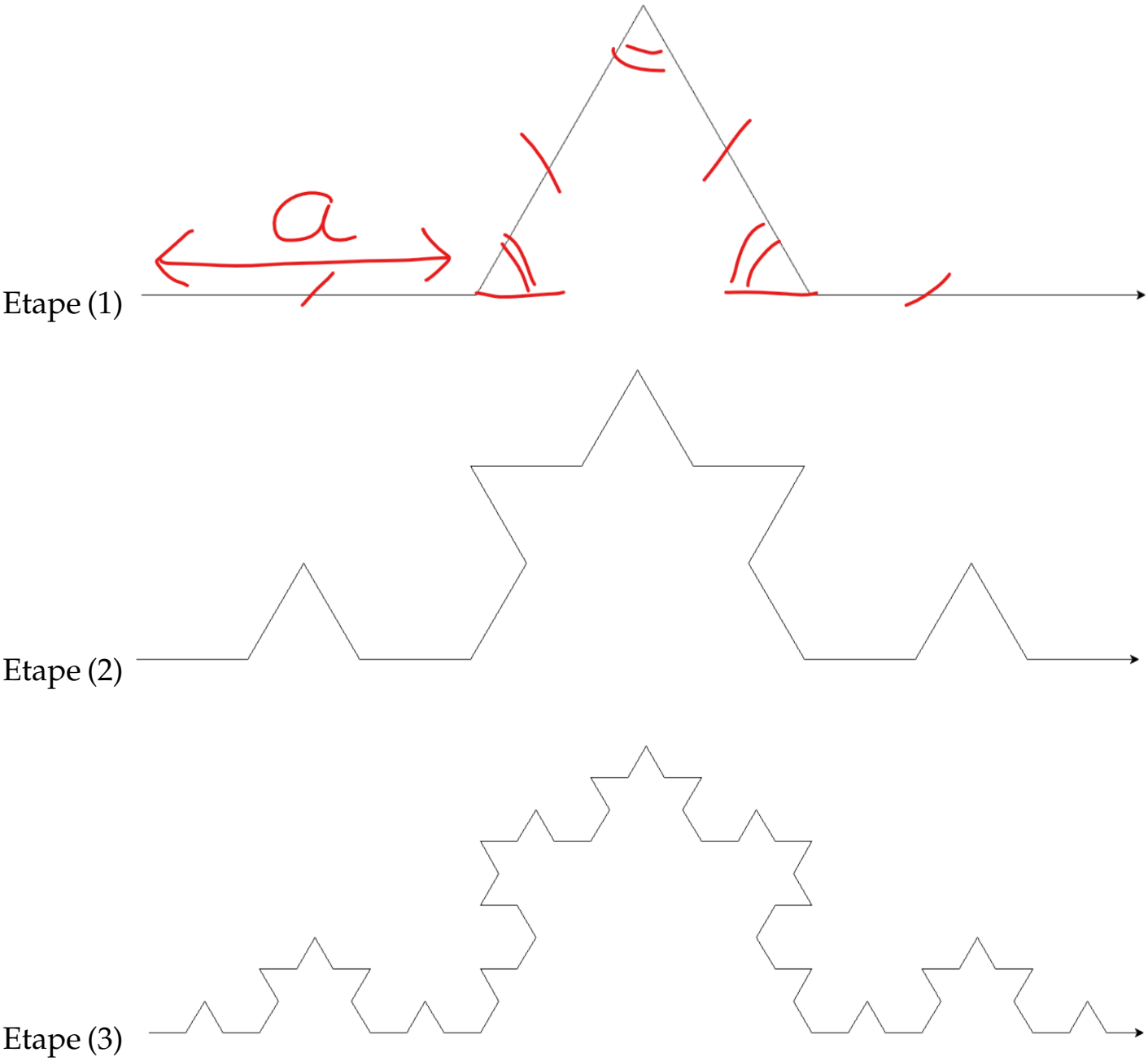
[17]:
```
for k in range(16):
    print(k,fiboIter(k))
```

```
0 0
1 1
2 1
3 2
4 3
5 5
6 8
7 13
8 21
9 34
10 55
11 89
12 144
13 233
```

```
14 377
15 610
```

1.4 Exercice 3 : le flocon de Koch

Il s'agit d'une structure fractale construite "*autour d'un triangle équilatéral*".



A l'aide du module `turtle`, on crée un objet Python qui permet de tracer des lignes brisées. On déplace ainsi la *tortue* en appelant des méthodes sur l'objet `t=turtle.Turtle()` qui sont les suivantes: - `t.forward(d)` : avance d'une longueur *d*, - `t.right(angle)` : tourne à droite de l'angle *angle* (rotation horaire), l'angle étant exprimé en degrés. - `t.left(angle)` : tourne à gauche l'angle *angle* (rotation dans le sens trigonométrique), l'angle étant exprimé en degrés.

Commpléter le code ci-dessous de manière à générer le flocon de Koch.

[18]:
```
# CODE à compléter
import turtle # module turtle
def koch(a, n):
    if n == 0: # condition d'arret
        t.forward(a)
        return
    # à compléter
    #
    #

# SCRIPT D'APPEL A LA FONCTION
turtle.TurtleScreen._RUNNING = True
w = turtle.Screen()

w.setup(1400, 700) # taille de la zone de tracée (PIXELS_X, PIXELS_Y)
w.clear()
w.visible = True
t = turtle.Turtle()
t.speed("fastest") # ou bien nombre entier entre 1 (lent) et 10␣
 ↪(rapide), zéro.
t.up()
t.setpos(-600, -200) # position initiale de la tortue
t.down()
koch(1200,1) # APPEL A LA FONCTION KOCH
w.exitonclick()
```

Réponse : il faut effectuer 4 appels récursifs à l'ordre $n-1$ et pour une longueur de segment divisée par 3.

1.5 Exercice 4 : recherche d'un élément dans une liste triée

On considère une liste triée par ordre croissant. On souhaite savoir si l'élément x est dans la liste.

On souhaite savoir si un élément x appartient à la liste en utilisant un algorithme récursif basé sur la principe de recherche par dichotomie.

1.5.1 Fonction qui renvoie un booléen

Ecrire une telle fonction Python `dansListe(L,x)` qui, à partir d'une liste `L` que l'on suppose triée par ordre croissant, renvoie *True* si l'élément est dans la liste.

Tester votre fonction sur la liste `L0` définie par:

`L0 = [ 2*k+1 for k in range(int(2e4)) if k %17 ==0 or k %3 ==0]` (Rappel : une telle liste est dite "définie *en compréhension*") en y recherchant les éléments x suivants: 561, 12409, 17477, 22501.

[ ]:
```
def dansListe(L, x) : # recherche dichotomique dans une liste triée à␣
 ↪compléter
    n = len(L)
    if n == 0 :
        # à compléter
    m = # "milieu" de la liste

    if  # élémet trouvé à l'indice m
        return True
    elif # condition de dichotomie
         # recherche dans la sous-liste de drotie
        return dansListe(L[## ], x) #slicing
    else : # sous liste de 'gauche'
        return dansListe(L[##], x) #slicing
```

1.5.2 Comptage du nombre d'appels

Modifier le programme proposé de manière à compter le nombre d'appels à la fonction pour chacun des cas recherchés. Afficher le cardinal de la liste *L0*. Commenter.

[21]:
```
# CODE DE VALIDATION
L0 = [ 2*k+1 for k in range(int(2e4)) if k %17 == 0 or k %3 == 0]
xListe = [561, 12409, 17471, 22501, 39997]
print('len(L0) = ',len(L0))
for x in xListe:
    nbAppel=0
    dans = dansListe(L0,x)
    print(dans, nbAppel)
```

```
len(L0) =  7451
False 14
True 11
False 14
True 4
True 12
```

1.6 Exercice 5 : Triangle de Pascal et ensemble des parties d'un ensemble

On note $\binom{n}{k}$ le nombre de parties d'un ensemble à n éléments qui contiennent k éléments.

1.6.1 Construction récursive du triangle de Pascal

Construire de manière récursive le triangle de Pascal en utilisant les propriétés suivantes:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Et pour un ensemble à $n = 1$ élément, il n'y a qu'une seule partie qui contienne un nombre nombre k d'éléments ($k = 0$ ou $k = 1$). Compléter le code ci-dessous.

[ ]:
```
def C(k,n):
    if (k < 0) or (k > n): # k doit être compris entre 0 et n au sens␣
 ↪large
        # A COMPLETER
        return ...
    if (n == 1): # pour un ensemble à 1 élément, il n'y a une manière
                 #unique de construire les parties (à zéro ou 1␣
 ↪élément)
        return ...
    return ...
```

[17]:
```
# COMPARAISON AVEC L'EXPRESSION N!/((N-k)!k!)
print(C(5,11))
print(fac(11)/((fac(5)*fac(11-5))))
```

```
462
462.0
```

1.6.2 Ensemble des parties d'un ensemble à k éléments *Proposer l'algorithme d'une* fonction récursive* `parties(L,k)` qui donne toutes les parties à k éléments de l'ensemble L à n éléments passé comme argument.

Attention: cette fonction renvoie une **liste de parties**, chaque *partie* étant elle-même un sous-ensemble (= une sous-liste) de la liste initiale. Le résultat de la fonction est donc une **liste de listes**.

Indications :

- lorsque $k = 0$, il n'y a qu'une seule partie possible - c'est *l'ensemble vide*. La fonction renverra donc la liste `[[]]`, c'est-à-dire une liste ne contenant que la liste vide.
- lorsque $k = n$, il n'y a encore qu'une seule partie possible. C'est la liste L au complet. La fonction renverra donc une liste ne contenant que cette partie.
- dans les autres cas, on utilise le fait que l'on peut obtenir l'ensemble des parties contenant k éléments à partir de :
  - l'ensemble des parties contenant k éléments de la sous-liste $L[0:-1]$

– l'ensemble des parties contenant $k-1$ éléments de la sous-listes $L[0:-1]$ en ajoutant à chacune de ces parties le dernier élément de la liste L.

Pour ajouter un élément à chaque partie, on pourra parcourir la liste des parties `parties` à l'aide d'une boucle de la forme :

```
for p in parties :          p.append(...)
```

[ ]:
```
def parties(L,k): # ensemble des parties à k éléments

```

[22]:
```
# VALIDATION
N = 4 # ensemble à 4 éléments
L0 = [j for j in range(N)]
for k in range(N + 1):
    ensParties = parties(L0,k)
    print('k=', k,' nb = ', len(ensParties), C(k,N))
    print(ensParties)
```

```
k= 0  nb =  1 1
[[]]
k= 1  nb =  4 4
[[0], [1], [2], [3]]
k= 2  nb =  6 6
[[0, 1], [0, 2], [1, 2], [0, 3], [1, 3], [2, 3]]
k= 3  nb =  4 4
[[0, 1, 2], [0, 1, 3], [0, 2, 3], [1, 2, 3]]
k= 4  nb =  1 1
[[0, 1, 2, 3]]
```

1.6.3 Ensemble des parties d'un ensemble

Proposer une fonction `ensPartieTotal(L)` qui renvoie l'ensemble des parties d'un ensemble à *n=len(L)* éléments à partir d'une liste L fournie en argument. On utilisera un algorithme récursif ne faisant pas appel à la fonction `parties(L,k)` précédemment définie mais se basant sur des propriétés suivantes :

1) L'ensemble des parties d'un ensemble à zéro élement est l'ensemble vide.

2) L'ensemble des parties d'un ensemble à $n > 0$ éléments est la réunion de :

- l'ensemble P des parties de cet ensemble lorsqu'il est privé de son dernier élément, `L[-1]`;
- l'ensemble des éléments de P auxquels on ajoute l'élément `L[-1]`.

Tester votre fonction à l'aide du script de test qui permet d'afficher l'ensemble des parties de la liste `L0=['a','b','c','d']`

Vérifier également que le nombre de parties de cet ensemble à n élément est égal à 2^n.

```
[ ]: def ensPartieTotal(L): # ensemble des parties de l'ensemble L
       (récursivité)
         if # condition d'arrêt
             return ## ensemble ne contenant que l'ensemble vide...
         res = # appel récursif pour déterminer P
         if len(res)>0 : # ajout du dernier élément à P
             for el in res :
                 ## à compléter
         return ## à compléter
```

```
[27]: L0 = ['a','b','c','d']
```

```
[28]: rr=ensPartieTotal(L0)
      #print(rr)
      print(rr)
      print('Card(Ensemble des Parties) = ',len(rr))
```

```
[[], ['a'], ['b'], ['a', 'b'], ['c'], ['a', 'c'], ['b', 'c'], ['a',
 'b', 'c'],
['d'], ['a', 'd'], ['b', 'd'], ['a', 'b', 'd'], ['c', 'd'], ['a', 'c',
 'd'],
['b', 'c', 'd'], ['a', 'b', 'c', 'd']]
Card(Ensemble des Parties) =  16
```

1.7 Exercice 6 : Ensemble des permutations

Ecrire une fonction python `perm(L)` qui renvoie l'ensemble des permutations d'un ensemble à n éléments fourni en argument. Utiliser un algorithme récursif.

Par exemple, pour `L=['a','b','c']`, la fonction renvoie la liste suivante :

```
[['c', 'b', 'a'], ['c', 'a', 'b'], ['b', 'c', 'a'], ['a', 'c', 'b'],
['b', 'a', 'c'], ['a', 'b', 'c']]
```

Indication. Utiliser la propriété suivante : les permutations d'un ensemble à n éléments s'obtiennent à partir des permutations p_k de l'ensemble à $n-1$ éléments obtenu en retirant le dernier élément $L[-1]$ de cet ensemble et en insérant cet élément $L[-1]$ dans **l'une quelconque des n positions possibles** pour chaque permutations p_k contenant $n-1$ éléments:

- AVANT le premier élément
- entre l'élément 1 et l'élément 2, ...
- entre l'élément k et l'élément $k+1$, ...
- APRES le dernier élément.

Note:

- on pourra utiliser l'instruction `L.insert(k,toAdd)` qui insère l'élément `toAdd` en k-ième position dans la liste `L`.

- on prendra garde au caractère *mutable* de l'objet liste Python qui interdit notamment d'effectuer successivement plusieurs insersions du même type dans la liste `L` sous peine d'accroître sa taille au délà...

[30]:
```
# Pour un ensemble à 3 éléments
L0=['a','b','c']
nbCall=0
res=perm1(L0)
print(res)
print(len(res),nbCall)
print(nbCall/len(res))
```

```
[['c', 'b', 'a'], ['c', 'a', 'b'], ['b', 'c', 'a'], ['a', 'c', 'b'],␣
 ↪['b', 'a',
'c'], ['a', 'b', 'c']]
6 12
2.0
```

1.8 Exercice 7 : combinaisons d'entiers

Soit n un entier naturel non nul, une **combinaison** de l'entier n est une liste $(n_1, ..., n_p)$ d'entiers supérieurs ou égaux à 1 dont la somme fait n, *i.e.* la décomposition $3 = 2 + 1$ diffère de la combinaison $(1, 2)$, *i.e.* $3 = 1 + 2$.

Le but de cet exercice est de générer toutes les combinaisons d'un entier n.

Une combinaison sera représentée sous forme de liste et donc l'ensemble des combinaisons sous forme de liste de listes.

Pour $n = 1$, la seule combinaison est (1), l'ensemble des combinaisons est donc $\{(1)\}$, ce qui s'écrit en Pyhon `[[1]]`.

Pour $n = 2$, les combinaisons sont (2) et $(1, 1)$, ce qui s'écrit $\{(2), (2, 1)\}$ ou en Python:

```
[[2],[1,1]]
```

On peut générer l'ensemble des combinaisons de n de la manière suivante : - Si $n = 1$, la seule combinaison est `[[1]]` - Si $n > 1$, l'une des combinaisons est simplement (n), et les autres sont obtenues à partir d'un entier $i \in [\![1, n-1]\!]$: on génère toutes les combinaisons de i et on ajoute $n - 1$ à la fin pour obtenir une combinaison de n.

On obtient ainsi toutes les combinaisons de n à l'aide d'un algorithme récursif.

Mettre en oeuvre cet algorithme en créant une fonction `getCombinaisons(n)` qui renvoie les combinaisons de l'entier n passé en argument.

Afficher les combinaisons pour les 5 entiers naturels positifs afin de tester votre programme.

[ ]:
```
def getCombinaisons(n): # version récursive

```

[37]:
```
# Validation du programme
for k in range(1,5):
    res = getCombinaisons(k)
    print(k,res)
```

```
1 [[1]]
2 [[2], [1, 1]]
3 [[3], [1, 2], [2, 1], [1, 1, 1]]
4 [[4], [1, 3], [2, 2], [1, 1, 2], [3, 1], [1, 2, 1], [2, 1, 1], [1,␣
↪1, 1, 1]]
```

1.9 Exercice 8 : Tri récursif par insertion

1.9.1 Insertion récursive

Ecrire en langage Python une fonction `triRecInsert(L)` qui renvoie une liste triée à partir de la liste d'éléments de la liste L passé en argument selon la relation d'ordre totale $<$ sur les éléments de L. On utilisera les propriétés suivantes :

- si la liste contient **moins de deux éléments** ($n = len(L) \leq 1$), elle est considérée comme triée
- si la liste convient au moins deux éléments, on isole le premier élément que l'on insère en parcourant linéairement la sous-liste des $n-1$ derniers éléments de L.

Note: faire attention au caractère *mutable de l'objet liste* en Python dans la conception de votre algorithme.

Tester la fonction ainsi créée sur la liste de nombres entiers:

`L0=[1,-3,2,91,4,5,8,-12,102,43,28,32,-1000,4,5,18]`

[ ]:
```
def triRecInsert(L): # tri récursif par insertion
    n = len(L)
    if # condition d'arrêt
        return L
    Lsub = # sous-liste triée des n-1 éléments restants
    for k in range(n-1): # recherche d'une position d'insertion
        if L[0] < Lsub[k]: # condition d'insertion
            return # insertion à l'emplacement k par concaténation
    return  #cas où l'élément est inséré en fin de liste
```

[39]:
```
L0 = [1,-3,2,91,4,5,8,-12,102,43,28,32,-1000,4,5,18]
res = triRecInsert(L0)
print(res)
```

```
[-1000, -12, -3, 1, 2, 4, 4, 5, 5, 8, 18, 28, 32, 43, 91, 102]
```

1.9.2 Application pour une matrice aléatoire

Application au tri des lignes d'une matrice aléatoire

Importer la bibliothèque *numpy* avec l'alias *np*. Utiliser sa méthode `random.rand(n,m)` qui crée une matrice (= objet Python `numpy.ndarray`) possédant n lignes et m colonnes pour tester le fonctionnement de la fonction `triRecInsert(L)` sur les m lignes de cette matrice.

Pour cela, on utisera : 1. La méthode `np.shape(M)` qui renvoie *tuple* contenant la structure de l'objet.

Exemple, avec `nx,ny=np.shape(M)` *nx* est la taille de la dimension relative au premier indice de M et *ny* est la taille de la dimension relative au second indice de M.

2. La méthode `X.tolist()` qui renvoie un *numpy array* converti en liste (ou en liste de listes pour les matrices).

3. La comparaison entre deux listes :

`La==Lb` est vraie si et seulement si - les deux listes ont le même nombre n d'éléments - et

$$\forall k = 0 \dots n-1, \quad \texttt{La[k]} = \texttt{Lb[k]}$$

.

4. Le *slicing* sur les objets de type *numpy array*.

Exemple si

```
M=np.array([[i+j*4 for i in range(4)] for j in range(5)])
```

soit `array([[ 0, 1, 2, 3], [ 4, 5, 6, 7], [ 8, 9, 10, 11], [12, 13, 14, 15], [16, 17, 18, 19]])`

`M[1,:]` renvoie la sous-matrice de la deuxième ligne (d'indice python i=1):

```
array([4, 5, 6, 7])
```

`M[:,2:4]` renvoie la sous-matrice correspondant aux troisième et quatrième colonnes (d'indices python 2 et 3):

```
array([[ 2, 3], [ 6, 7], [10, 11], [14, 15], [18, 19]])
```

[40]:
```
import numpy as np
M=np.random.rand(5,10)
n,m=np.shape(M)
for k in range(m):
    L0=M[:,k]
    L0=L0.tolist()
```

```
    La=L0.copy()
    La.sort()
    if La==triRecInsert(L0):
        print('ok')
    else:
        print('!')
```

```
ok
ok
ok
ok
ok
ok
ok
ok
ok
ok
```

1.10 Exercice 9 : Exponentiation et exponentiation rapide

On considère la fonction suivante écrite en Python:

[41]:
```
def puissance1(a,n):
    """Fonction récursive qui calcule a^n:
    entrées : a, nombre réel, n nombre entier naturel
    sorties : a^n    """
    if n==0 :
        return 1
    return a*puissance1(a,n-1)
```

1.10.1 Théorie de l'exponentiation

Décrire l'algorithme présenté : est-il récursif ? S'agit-il de récursivité simple, multiple, mutuelle?

Prouver: - la **terminaison** de l'algorithme - sa **correction** (= montrer qu'il renvoie effectivement a^n)

Calculer sa complexité en temps, $C(n)$, en terme de nombre d'appels récursifs.

1.10.2 Version terminale

Reformuler l'algorithme précédent en version **terminale** sur le modèle `puissanceTerm(a,n,res=1)`.

Le tester pour $a = 2$ et $n = 5$, puis pour $a = 3$ et $n = 0$.

Avec un script utilisant cette fonction calculer la somme $S_n(a)$ des n premières puissances de a, et a^0 étant considéré comme le premier terme :

$$S_n = a^0 + a^1 + a^2 + \ldots + a^{(n-1)}$$

Donner la valeur numérique $n = 10$ et $a = 3$.

[ ]:
```
def puissanceTerm(a, n, res = 1): # version récursivité terminale
    if : # condition d'arrêt
        return #
    return # à compléter
```

[43]:
```
print(puissanceTerm(2,5))
print(puissanceTerm(3,0))
somme=0
for k in range(10):
    somme += puissanceTerm(3,k)
print(somme)
```

```
32
1
29524
```

Remarque : on vérifie bien l'égalité

$$S_n(a) = \sum_{k=0}^{n-1} a^k = \frac{1-a^n}{1-a}$$

[44]:
```
(1-3**10)/(1-3) # pour a=3 et n=10
```

[44]:
```
29524.0
```

1.10.3 Exponentiation rapide récursive.

L'algorithme d'exponentiation rapide (en anglais *square-and-multiply*) est basée la propriété suivante :

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ a^{n/2} \times a^{n/2} & \text{si } n \text{ est pair} \\ a \times a^{(n-1)/2} \times a^{(n-1)/2} & \text{si } n \text{ est impair} \end{cases}$$

Ecrire en langage Python la fonction `puissanceRapide(a,n)` qui utilise cette propriété de manière récursive.

La tester pour calculer 3^{128}.

[ ]:
```
def puissanceRapide(a,n): # exponentiation rapide récursive

```

[46]:
```
print(puissanceRapide(3,128))
print(3**128)
```

```
11790184577738583171520872861412518665678211592275841109096961
11790184577738583171520872861412518665678211592275841109096961
```

Calculer la complexité en temps de cet algorithme (on supposera que n s'ecrit sous la forme 2^p), et comptera le nombre d'appels à la fonction.

Comme n est une puissance de 2, la fonction `puissanceRapide` est appelée lors des récurrences successives avec un argument n qui est pair.

Ainsi, au premier appel, l'argument est la valeur initiale de n, notée n_0.

Au second appel, n est divisée par 2 et vaut $2^{p-1} = n_0/2$, au troisième appel, n est encore divisée et vaut $2^{p-2} = n_0/2^2$, etc ... Lorsque n vaut 1, c'est-à-dire au bout de p divisions par 2, et donc de $p+1$ appels, la fonction est appelée une dernière fois avec l'argument $n//2$ qui vaut donc zéro. Cela constitue la condition d'arrêt de la fonction récursive qui renvoie 1 dans ce cas.

In fine, pour $n = 2^p$, il y a $(p+2)$ appels à la fonction.

Or, de $n = 2^p$, on tire $p = \log(n)/\log(2)$, d'où la complexité:

$$C(n) = \frac{\log(n)}{\log(2)} + 2 = \log_2(n) + 2$$

L'algorithme possède donc une complexité logarithmique.

Remarque : une autre manière d'évaluer la complexité est de considérer que les appels à fonction vont se faire avec les arguments suivants: $n = 0$, $n = 1$, $n = 2$, $n = 2^2$, $n = 2^3$, ..., $n = 2^p$. On voit bien que la liste comprend $(p+2)$ valeurs.

Modifier le programme en ajoutant une variable globale de manière à compter le nombre d'appels à la fonction `puissanceRapide(a,n)` lors du calcul de 3^{128}.

1.10.4 Critique d'un code existant

Un élève propose l'algorithme suivant. Est-il correct? En faire la critique. Calculer la complexité $C(n)$ en nombre d'appels à la fonction pour un argument n de la forme $n = 2^p$, p étant un entier naturel.

[48]:
```
def puissance2(a,n):
    global nbAppels
    nbAppels+=1
    if n==0:
        return 1
    if (n%2==0):
```

```
        return puissance2(a,n//2)*puissance2(a,n//2)
    else:
        return a*puissance2(a,n//2)*puissance2(a,n//2)
```

[49]:
```
nbAppels = 0
print(puissance2(3,128))
print(3**128)
print(nbAppels)
```

```
11790184577738583171520872861412518665678211592275841109096961
11790184577738583171520872861412518665678211592275841109096961
511
```

[50]:
```
4*128-1
```

[50]: 511

1.10.5 Exponentiation rapide en récursivité terminale

Proposer une version en **récursivité terminale** de l'algorithme d'exponentiation rapide sur le modèle de fonction `puissanceRapideTerm(a,n,res=1)`.

Utiliser pour cela la définition par récurrence de la puissance $n$ième d'un réel a par:

$$\text{puissance}(a,n) = \begin{cases} 1 & \text{si } n = 0 \\ a & \text{si } n = 1 \\ \text{puissance}(a^2, n/2) & \text{si } n \text{ est pair} \\ \text{puissance}(a^2, (n-1)/2) \times a & \text{si } n \text{ est impair} \end{cases}$$

Indications :

- Commencer par proposer un algorithme terminal récursif valable dans le cas où n est une puissance de 2, $n = 2^p$.
- L'idée est d'utiliser l'argument auxiliaire *res* tout comme pour l'algorithme de récursivité terminale (`puissanceTerm`(a,n)) . Avec cet argument on mémorise, les multiplications sucessives par l'argument a qui n'entrent pas dans le schéma de récurrence simple où (`puissanceTerm`$(a*a, n/2)$) lorsque n est une puissance de 2.
- Considérer le schéma ci-dessous :
- 3^{11} = ? On pose $11 = 2 \times p + 1$, avec $p = 5$, soit $3^{11} = (3^5)^2 \times 3$
- 3^5 = ? On pose $5 = 2 \times p + 1$, avec $p = 2$ soit $3^5 = (3^2)^2 \times 3$
- 3^2 = ? On pose $2 = 2 \times p$, avec $p = 1$ soit $3^2 = (3^1)^2$
- 3^1 = ? Ici, l'exposant vaut 1 donc on utilise le fait que $a^1 = a$

```
[ ]: def puissanceRapideTermAux(a,n): # n doit être une puissance de 2
```

```
[ ]: def puissanceRapideTerm(a,n,res=1): # exponentiation rapide en␣
     ↪récursivité terminale
         if n ==0 : # condition d'arrêt N°1

         if n ==1 : # condition d'arrêt N°2

             return a* res # la variable res est prise en compte à la fin
         if (n % 2 == 0): # parité?
             return  # n pair
         else:
             return # n impair
```

```
[ ]: for k in range(8): # VALIDATION
         print(3**k,puissanceRapideTerm(3,k))
```

1.10.6 Dérécursivation

Dérécursivation de l'algorithme d'exponentiation rapide.

Proposition une version itérative `puissanceIterRapide(a,n)` de l'algorithme d'exponentiation rapide.

Indication : pour calculer 3^{23}, considérer l'écriture en binaire de $n = 23$.

$n = 16 + 4 + 2 + 1$, ainsi le nombre 3^{23} peut s'écrire comme un produit de puissances de 3 dont les exposants sont des puissances de 2 : 3^{2^k}.

$$3^{23} = 3^{16} \times 3^4 \times 3^2 \times 3^1$$

L'idée est donc d'utiliser une boucle sur les puissances 3^{2^k} et de multiplier une variable auxiliaire par ces puissances lorsque les termes interviennent dans la décomposition.

```
[ ]: def puissanceIterRapide(a,n): # exponentiation rapide dérécursivée
```

```
[ ]: for k in range(12):
         print('3 ^( ',k,' ) = ',3**k,'   ', puissanceIterRapide(3,k))
```

Prouver la terminaison, la correction de cet algorithme et calculer la complexité $C(n)$ en terme de multiplications dans les cas: - $n=2^p$, p entier naturel. - $n = 2^p - 1$, p entier naturel.

1.11 Exercice 10* : les tours de Hanoï

Le matériel de ce casse-tête classique est constitué de trois piquets, notés A, B, C et de n rondelles de diamètres différents percées en leur centre de sorte qu'elles peuvent être enfilées sur les piquets.

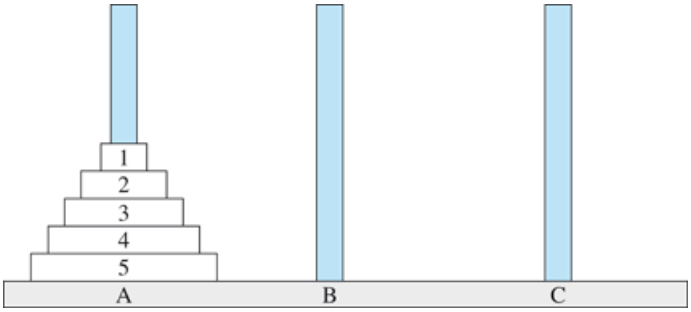
Dans la position initiale, les rondelles sont enfilées par ordre de diamètre décroissant sur le piquet 1 ; le but du jeu est de les transférer vers le piquet 3 en respectant la règle suivante : "déplacer une seule rondelle à la fois, de sorte qu'à aucun moment il n'y ait une rondelle posée sur une autre de diamètre inférieur".

Écrire une fonction récursive **affichant une succession de déplacements** permettant de résoudre ce problème sous la forme $[i \rightarrow j]$, où les indices i, j $((i,j) \in [\![1,3]\!] \times [\![1,3]\!])$ représentent les piquets.

La fonction créée possèdera la syntaxe : `deplacer(n,i,f)` (n : nombre de rondelles empilées, i: position initiale, f: position finale).

Déterminer par le calcul le nombre de déplacements $C(n)$ effectués en fonction de n. Vérifier la relation obtenue pour $C(n)$ à l'aide de la fonction Python que vous modifierez en conséquence.

Afficher les valeurs de $C(n)$ pour n allant de 1 à 10.



[ ]:
```
# ON POURRA S'AIDER DE LA TRAME A COMPLETER
def deplacer(n,i,f):
    if ... :
        inter = ...
        deplacer(..., ..., ...)
        print( .. ,' --> ',...)
        deplacer(...,...,...)
```