TP n°1 – Fonctions récursives

CORRECTION

Notions utiles :

Définition : Une **fonction récursive** est une fonction qui contient au moins un appel à elle-même.

Par exemple : La fonction suivante calcule la factorielle d'un entier naturel n.

```
def factorielle(n) :
    #cas trivial 0 ! = 1
    if n == 0 :
        return 1
    #cas récursif ou hérédité
    return n*factorielle(n-1)
```

Pour bien implémenter une fonction récursive il faut s'assurer de sa terminaison. Pour cela on suit toujours les règles suivantes :

- La fonction doit contenir au moins un cas trivial ne comportant pas d'appel récursif.
- Les appels récursifs à la fonction convergent toujours vers le cas trivial.

Lors de l'exécution de la fonction chaque appel récursif met « en pause » l'exécution en cours, en attente d'obtenir le résultat qui est déterminé par l'appel suivant. Concrètement :

- Les appels sont tour à tour mis « en pause » jusqu'au dernier appel qui fournit un résultat (cas trivial). On appelle cela **le dépliage** ou la descente.
- Ce résultat est ensuite transmis à l'appel précédent qui l'utilise pour calculer son propre résultat et le transmettre à l'appel précédent, et ainsi de suite jusqu'au premier appel qui peut calculer le résultat final. On appelle cette phase **l'évaluation** ou la remontée.

Du point de vue de l'ordinateur l'appel d'une fonction récursive fait appel à **une pile d'exécution**. Une pile est une structure de type **LIFO** (Last In First Out). Les variables locales de chaque appel sont empilées successivement sur la pile. Lorsqu'on obtient un résultat exploitable au sommet de pile alors on peut commencer à dépiler.

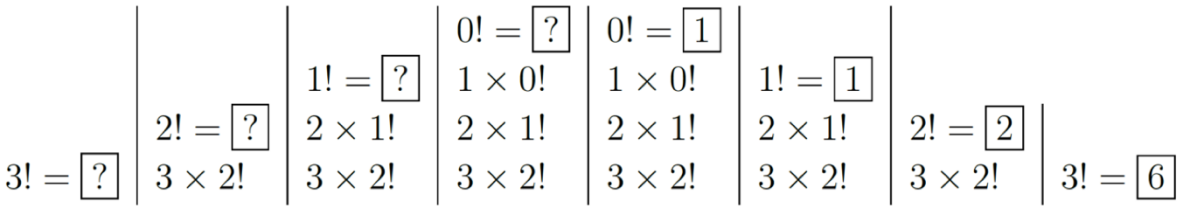

Figure 1 - Illustration du remplissage de la pile dans le cas de la fonction factorielle(3)

Attention ! La pile d'exécution est une structure de taille limitée. Lorsque beaucoup d'appels sont nécessaires Python renvoie une erreur. Il faut donc être attentif lors de l'utilisation des fonctions récursives.

Exercice 1 : Adoptez des lapins

1. A partir des données on voit apparaitre la relation de récurrence : $c_{n+2} = c_{n+1} + c_n$.
   Il s'agit de la suite de Fibonacci.

2. Pour calculer le nombre de couples après n mois, version itérative :

```
def LapinsIt(n):
    a,b = 1,1
    for i in range(n):
          a,b = b,a+b
    return a
```

3. Pour calculer le nombre de couples après n mois, version récursive :

```
def LapinsRec(n):
    if(n <= 1):
        return 1
    else:
        return (LapinsRec (n-1) + LapinsRec (n-2))
```

4. Avec la méthode itérative duree_it $= 3{,}1.10^{-6}$s et avec la méthode récursive duree_rec $= 2{,}1$s
5. Il n'y a qu'une seule boucle for donc la complexité est linéaire pour l'algorithme itératif. En revanche pour l'algorithme récursif la complexité est exponentielle !
6. En général un algorithme récursif est plus concis et plus adapté aux fonctions. En revanche comme l'illustre cet exemple la complexité temporelle explose rapidement dans le cas de l'algorithme récursif.

Exercice 2 : Diviser pour mieux régner

1.

```
def rechDicho(L,x,i,j):
        if i > j :
                return False
        m = (i+j)//2
        if L[m] == x :
                return m
        elif L[m] < x :
                return rechDicho(L,x,m+1, j)
        else :
                return rechDicho(L,x,i, m-1)
```

2.

```
def parcours_simple(L,x):
      for i in range(len(L)):
            if L[i] == x:
                  return i
      return False
```

Le pire des cas correspond ici à la recherche d'un élément qui n'est pas dans la liste. L'algorithme dichotomique est alors plus efficace. Sa complexité est logarithmique tandis que le parcours simple à une complexité linéaire.

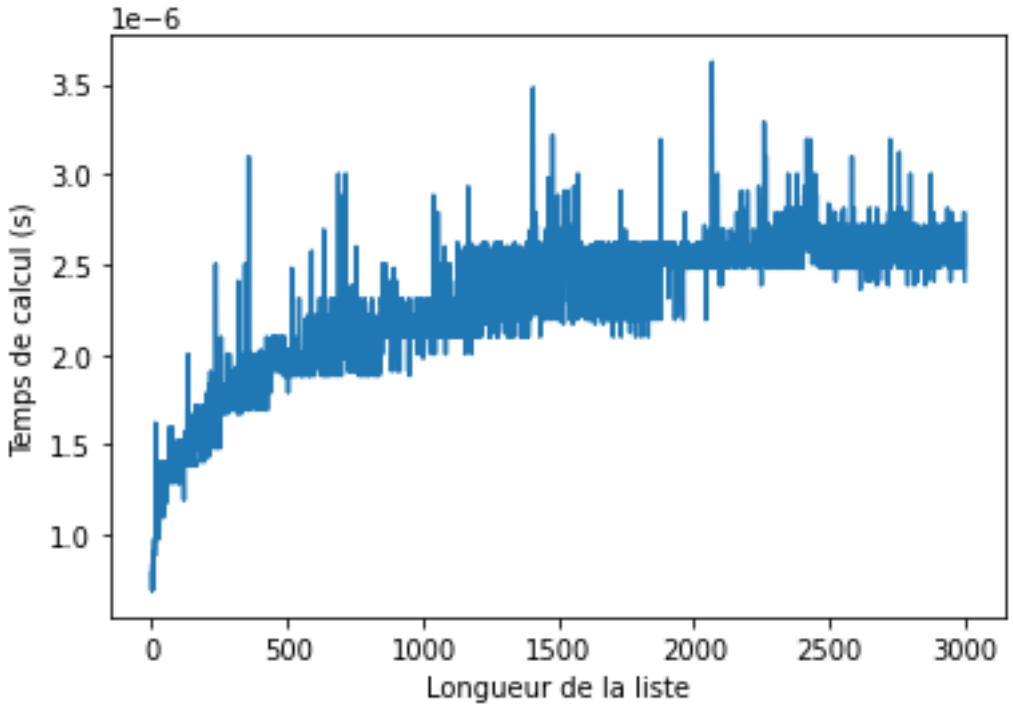

Figure 2 - Évolution du temps de calcul en fonction du nombre d'éléments dans la liste pour l'algorithme récursif.

Exercice 3 : Lire à l'envers

1.

```
def inverse_chaine(chaine):
    resultat = ''
    for caractere in chaine:
        resultat = caractere + resultat
    return resultat
```

2.

```
def est_palindrome(chaine):
    '''Renvoie un booléen indiquant si la chaine
    est un palindrome'''
    inverse = inverse_chaine(chaine)
    return chaine == inverse

def palindrome(mot):
  if len(mot) <= 1:
      return True

  if mot[0] == mot[-1]:
      return palindrome(mot[1:-1])
  else:
    return False
```

Exercice 4 : Avez-vous la monnaie ?

1.

```
valeurs = [100,50,20,10,5,2,1]

def rendu_monnaie(a_rendre, rang):
    if a_rendre == 0:
        return []
    v = valeurs[rang]
    if v <= a_rendre :
        return [v] + rendu_monnaie(a_rendre - v, rang)
    else :
        return rendu_monnaie(a_rendre, rang + 1)
```

Exercice 5 : Le flocon de Van Koch

1.

```
import turtle as tl


def courbeVonKoch(  n, cote  ) :
      if n == 0 :
            tl.forward(cote)
      else :
            courbeVonKoch(n-1, cote/3)
            tl.left(60)
            courbeVonKoch(n-1, cote/3)
            tl.left(-120)
            courbeVonKoch(n-1, cote/3)
            tl.left(60)
            courbeVonKoch(n-1, cote/3)


def flocon(n, cote) :
      for i in range(3) :
            courbeVonKoch(  n, cote  )
            tl.left(-120)
            i = i+1

tl.setheading(0) # orientation intiale de la tête : vers la droite de
l'écran
tl.hideturtle() # on cache la tortue
tl.speed(0)  # on accélère la tortue
tl.color('green')
flocon(  n = 3, cote = 200  )
tl.exitonclick() # pour  garder ouverte la fenêtre
```