

A / Fonctions récursives pour un calcul de nombre

[niveau facile]

A.1 / Factorielle

Q1 –  Écrire une fonction `facto(n)` prenant en argument un entier naturel n et renvoyant la valeur de $n!$ calculée avec une boucle `while`. Tester la fonction avec, par exemple, les valeurs 3 et 4.

Q2 –  Même question, écrire et tester la fonction `facto_r(n)`, mais cette fois avec un algorithme récursif.

A.2 / Puissance

Q3 –  Écrire une fonction `puiss(x, k)` prenant en argument un flottant x et un entier naturel k et renvoyant la valeur de x^k calculée avec une boucle `while`. L'algorithme doit initialiser un produit, qui à chaque itération (k fois) est multiplié par x .

Tests :

```
print(puiss(2,3)) # doit afficher 2³ donc 8
print(puiss(2,4)) # doit afficher 2⁴ donc 16
```

Q4 –  Même question, écrire et tester la fonction `puiss_r(x, k)`, mais cette fois avec un algorithme récursif.

B / Fonction récursive sur une liste

[niveau moyen]

B.1 / Comptage d'occurrences

Commençons par un test pour (re)découvrir la méthode `Liste.pop(index_element)`. Dans un terminal, déclarez la variable `L_test = [1,2,3,4,5]`

Q5 –  Que renvoie la commande `L_test.pop(0)` ? Que vaut `L_test` après cet appel ?
Que renvoie `L_test.pop(-1)` ? Que vaut `L_test` après ce second appel ?

Remarque : cette fonction nous permet ici de faire ce qu'on appelle du dépilage par la gauche et par la droite.

On souhaite écrire une fonction prenant en argument une valeur a et une liste L , et que compte le nombre d'occurrences de la valeur a dans cette liste L . Vous aviez déjà eu à écrire cette fonction auparavant, et l'aviez probablement écrite :

```
def occur(a, L):
    compteur = 0 # initialisation du compteur d'occurrences
    for i in range(len(L)): # balayage des éléments de la liste
        if L[i] == a: # si on rencontre l'élément cherché
            compteur += 1 # incrémentation du compteur
    return compteur # en fin de balayage : renvoi du compteur
```

Exemples :

```
print(occur(3, [1,3,5,7,3])) # affiche 2
print(occur(2, [1,3,5,7,3])) # affiche 0
print(occur(1, [1,3,5,7,3])) # affiche 1
```

Q6 –  Copier-coller puis compléter le code de la fonction **récursive** suivante `occurr_r(a, L, i = 0)`, qui renvoie le nombre d'occurrences de a dans la sous-liste extraite `L[i:]`. L'argument optionnel i vaut par défaut 0, ce qui fait que l'appel `occurr_r(a, L)` renvoie le nombre total d'occurrences de a dans `L[0:] = L`.

Petite remarque contre-intuitive, mais utile pour déterminer le cas de base :

```
L = [1,3,5]
L[ len(L)-1:] # renvoie [5]
L[ len(L):]   # renvoie [] (liste vide)
```

```
def occur_r(a, L, i = 0) : # va compter les occurrences de a dans L[i:]
    if ... : # cas de base, on a atteint la fin de la liste (vide)
        return ...
    else :
        element = ... # on récupère la valeur du ième élément de la liste
        if element == a : # s'il vaut a (élément recherché)
            return ... # alors on incrémente le nombre d'occurrences
        else :
            return ... # sinon, on laisse le nombre d'occurrences inchangé
```

Q7 – Écrire une fonction **impérative** `occur_pile(a, L)` renvoyant le nombre d'occurrences de l'élément *a* dans la liste *L*, et qui s'appuie sur une méthode de pile. C'est-à-dire :

1. Faire une copie de *L* (pour ne pas la modifier à portée globale)
2. Dépiler (`L.pop(0)`) la copie par la gauche jusqu'à ce qu'elle soit vide (boucle `while` obligatoire)
 - ➔ À chaque élément dépilé, tester si c'est l'élément recherché
 - ➔ Si tel est le cas, on incrémente un compteur d'occurrences

Pour tester votre fonction, utilisons la liste :

```
L_tst = [1,3,4,6,7,9,0,1,3,4,6,1,6]
# et compter le nombre de 1 dans cette liste
```

Q8 – Écrire une fonction équivalente (même méthode de dépilage et test jusqu'à obtenir une liste vide), en complétant le code suivant. On s'interdit ici l'usage de `pop` et de la copie, pour privilégier des extractions de listes.

```
def occur_pile_r(a, L): # récursive
    if ... : # cas de base : L vide
        return ... # il ne peut pas y avoir la valeur a dans une liste vide !
    else :
        element = ... # 1er élément de la liste
        new_L = ... # on va faire l'appel récursif sur "la suite de la liste"
        if element == a :
            return ...
        else :
            return ...
```

B.2 / Somme des éléments d'une liste

Pour calculer la somme des éléments d'une liste *L*, vous pouvez écrire :

```
def somme_elements(L) :
    somme = 0
    for element in L :
        somme += element
    return somme
```

ou en utilisant un index balayant la liste :

```
def somme_elements(L) :
    somme = 0
    for i in range(len(L)) :
        somme += L[i]
    return somme
```

Q9 – En vous inspirant fortement du code que vous avez écrit pour la fonction `occur_r()` codée en **Q6**, écrire une fonction **récursive** `somme_r(L, i=0)` qui renvoie la valeur de la somme de la sous-liste extraite `L[i:]`. L'argument optionnel *i* vaut par défaut 0, ce qui fait que l'appel `somme_r(L)` renvoie la somme de `L[0:] = L`.

```
Test: L_tst = [1,3,4,6,7,9,0,1,3,4,6,1,6]
print(somme_r(L_tst)) # affiche 51
```

Q10 –  En vous inspirant fortement du code que vous aviez écrit pour la fonction `occure_pile(a, L)` codée en **Q7**, écrire une fonction **impérative** `somme_pile(L)` s'appuyant sur une boucle `while` et une méthode de dépile.

Il faudra donc :

1. Faire une copie de L (pour ne pas la modifier à portée globale)
2. Dépiler la copie (par la gauche ou par la droite) jusqu'à ce qu'elle soit vide (boucle `while` obligatoire)
→ sommer les éléments ainsi dépilés

Q11 –  En vous inspirant fortement du code que vous avez écrit pour la fonction `occure_pile_r()` codée en **Q8**, écrire une fonction **réursive** `somme_pile_r(L)` (même méthode de dépile et test jusqu'à obtenir une liste vide). On s'interdit l'usage de `pop` et de la copie, pour privilégier des extractions de listes. On prendra pour cas de base que la somme des éléments d'une liste vide vaut 0.

C / Principe et intérêt de la mémoïsation

[niveau moyen]

Prenons la suite de Fibonacci décrite par : $u_0 = 1 ; u_1 = 1 ; \text{ et } \forall n \geq 2, u_n = u_{n-1} + u_{n-2}$

Q12 –  • Écrire une fonction **réursive** `fibonacci(n)` prenant en argument un entier n et renvoyant la valeur de u_n .
• À l'aide de la commande `tic = time.time()` et `tac = time.time()`, (il faut importer au préalable `time`) regarder quel temps de processeur prend le calcul de u_{20}, u_{25}, u_{30} , etc jusqu'à atteindre des temps de calcul de plusieurs secondes.

On admet que le temps processeur nécessaire au calcul d'un rang n de la suite, est exponentiel. C'est-à-dire : $\exists \alpha \in \mathbb{R}_+^*$ tel qu'au-delà d'un certain rang, le temps de calcul soit proportionnel à α^n .

Q12 –  • À l'aide des valeurs renvoyées par `tac-tic` estimer la valeur de α pour votre machine.
• Quel serait l'ordre de grandeur du temps nécessaire pour calculer `fibonacci(2000)` ?

Le principe de la mémoïsation consiste à stocker tous les calculs intermédiaires dans une liste ou un dictionnaire. En l'occurrence, la récursivité au rang n demande simultanément le calcul de u_{n-1} et de u_{n-2} , or l'appel de u_{n-1} fait appel à u_{n-2} (possiblement déjà calculé), et de u_{n-3} , qui a par ailleurs été nécessaire au calcul de u_{n-2} . Il est donc ici possible d'économiser une quantité astronomique de calculs qui seraient refait de l'ordre de 2^n fois, le gain de temps croît considérablement avec n .

Nous allons donc stocker dans un dictionnaire, les valeurs des u_k à chaque fois qu'on les calcule : pour la clé k ($k < n$), on stockera la valeur u_k . Un tel dictionnaire est appelé `cache` (programmation dynamique).

Q14 –  Écrire une fonction `fibonacci_m(n, dico = ...)` qui prendra en argument n (rang auquel on souhaite élever la suite) et un argument optionnel `dico`, dont vous préciserez la valeur par défaut, et qui applique le principe de mémoïsation décrit ci-dessus.

Exécuter `fibonacci_m(2000)` et constater le temps de calcul.

Vous avez déjà, dans un précédent TD 9 sur les dictionnaires, écrit une fonction `comptage(L)` qui renvoie un dictionnaire dont les clés sont les éléments de L et les valeurs associées sont leur nombre d'occurrence. Voici le corrigé de ce devoir :

```
def comptage(L):
    dico = {} # dictionnaire vide
    for element in L: # element prend toutes les valeurs successives de L
        if element not in dico: # element est ici considere comme une "clé" du dictionnaire
            dico[element] = 1 # si la clé n'est pas déjà dans le dictionnaire, alors on
                              # l'ajoute, en comptant que c'est la 1ère occurrence de
                              # cette clé qu'on trouve
        else: # sinon, ca veut dire que la clé est déjà existante
            dico[element] += 1 # auquel cas on incrémente le compteur de nombre de fois
                              # où on a vu cette clé
    return dico # en fin de balayage de la liste, on renvoie le dictionnaire de comptage
               # des occurrences
```

Exemple :

```
ventes = [0, 100, 100, 80, 70, 80, 20, 10, 100, 100, 80, 70, 10, 30, 40]
print(comptage(ventes))
# affiche {0: 1, 100: 4, 80: 3, 70: 2, 20: 1, 10: 2, 30: 1, 40: 1}
```

Q15 – 🧩 En vous inspirant des codes des fonctions écrites en **Q8** et **Q11**, écrire une fonction `comptage_r(...)` qui renvoie le même dictionnaire que `comptage(L)`, mais avec algorithme **récuratif**, en s'appuyant sur la même méthode de pile.

E / Recherche du maximum d'une liste par un algorithme de type « tournoi »

[niveau difficile mais important]

Supposons une liste L , de taille une puissance de 2 (donc 2, 4, 8, 16, 32, ... éléments) contenant des nombres. On souhaite chercher le plus grand élément de L en appliquant une méthode de « tournoi ».

Vous trouverez sur mon site (<http://remyduperrayphysiquechimie.fr>) une petite vidéo d'une quinzaine de minute expliquant l'algorithme en détails, et où le code en Python est expliqué de bout par bout. Cette vidéo s'appelle « S2-TD19-*Algorithme_tournoi.mkv* » (qui s'ouvre très bien [avec VLC](#) si vous n'y arrivez pas).

Q16 – 🧩 Visionnez la vidéo attentivement, puis fermez-la et essayez de reproduire le code de la fonction récursive `rech_max(L)` qui a été présenté dans la vidéo sous le nom « *decoupe(L)* ».