Fonction et recursivite

1 Rappels sur les fonctions

1.1 Paramètres

La syntaxe de défintion d'une fonction est :

```
def fonction(param1, param2):  # nombre de paramètre arbitraire
    # corps de la fonction
    return # le résultat calculé
```

```
type(fonction)
```

```
function
```

Pour utiliser cette fonction on l'appelera sous la forme fonction(a, b) où a et b ont des valeurs ou sont directement des objets python.

Dans le corps de la fonction param1 et param2 auront les valeurs de a et de b respectivement.

```
def somme(a, b):
    return a + b

c = somme(5, 12)
d = 4
somme(c, d)
```

```
21
```

1.2 Variables locales

Toute variable définie dans le corps d'une fonction est une variable locale, qui n'est accessible que pendant l'éxécution de chaque appel, et cache toute variable de même nom définie à l'exérieur de la fonction.

```
a = 12
def f(n):
    a = 5
    b = n
    return a * b
```

```
f(10), a
```

1.3 Exercices

— Définir une fonction "factorielle" qui prend un entier naturel n comme paramètre et retourne n !
— Définir une fonction suite qui prend un nombre u0 et un entier naturel n comme paramètres et retourne la valeur du terme d'indice n de la suite de premier terme u0 et définie par $u_p = \frac{u_{p-1}}{2} + \frac{1}{u_{p-1}}$ pour tout $p \geq 1$

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

```
def suite(u0, p):
```

```
suite(1,5)
```

```
1.414213562373095
```

1.4 Combiner les fonctions

Quelle est la différence entre les exemples suivants ? Que vaut g(4)

```
def f(n):
    return 2 * n

def g(n):
    a = 1
    for i in range(n):
        a = f(a)
    return a

b=g(4)
```

```
b
```

```
def h(n):
    a = 1

    def h_int(k):
        return 2 * k

    for i in range(n):
        a = h_int(a)
    return a

b = h(4)
```

```
b
```

```
h_int
```

```
 ␣
↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
↪last)

       <ipython-input-9-27bc5741341c> in <module>
   ----> 1 h_int


       NameError: name 'h_int' is not defined
```

Les fonctions sont des objets python comme les autres. En particulier on peut les passer comme paramètre ou les renvoyer comme valeur de retour.

```
def multi_par(a):
```

```
        def interne(n):
            return a * n

        return interne
```

```
multi_5 = multi_par(5)
type(multi_5), multi_5(5)
```

```
(function, 25)
```

2 Fonctions récursives

Exemples de structures récursives :

2.1 Principe général

Une fonction f est dite récursive si, pour calculer sa valeur de retour, elle utilise une valeur de f.

Typiquement, pour une fonction f qui a un paramètre entier n, pour calculer f(n) on va d'abord calculer f(n-1).

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

```
f(5)
```

Evidemment la fonction précédente crée une boucle infinie.

Pour toute les fonctions récursives, il faut un cas d'arrêt, c'est à dire une valeur du ou des paramètres qui n'implique plus d'appel à f.

Prennons la définition suivante de n ! : - $0! = 1$ - $n! = n \times (n-1)!$ si $n > 0$

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

facto(3)
```

```
6
```

```
facto(10000)
```

2.2 Exercice

— Reprendre la suite précédente et créer une fonction suite_rec qui répond à la question de manière récursive

— Créer une fonction puissance(a, n) qui calcule a^n de manière récursive pour a un nombre et n un entier naturel.

2.3 Cas non trivial

Certain algorithmes s'écrivent naturellement sous forme récursive, et beaucoup moins facilement sous forme itérative.

Reprenons l'exemple du calcul de puissance. On définit a^n pour un nombre a et un entier naturel n par

1. $a^0 = 1$
2. Pour les $n \geq 1$, il y a deux cas : $a^n = (a^{\frac{n}{2}})^2$ pour les n pairs et $a^n = a \times a^{n-1}$ pour les n impairs.

```
def puissance_rec(a, n):
    if n == 0:
        return 1
    elif n % 2 == 0:

        return
    else:
        return
puissance_rec(2, 50)
```

```
1125899906842624
```

L'intérêt majeur ici est la réduction du nombre de multiplication. Une fois n'est pas coutume, nous allons utiliser une variable globale pour compter le nombre d'opérations effectuées.

```
def p(a, n):
    .... même chose mais avec un compteur

compteur = 0
p(2, 20000000)
compteur
```

```
32
```

Tâchons d'évaluer la complexité de notre fonction.

Le cas le pire est quand n est impair, et à chaque fois que l'on divise par 2, on trouve un entier impair (car dans ce cas, on réduit seulement l'exposant de 1 avant de diviser par 2).

En notant $c(n)$ le nombre d'opérations algébrique (+ appel de fonction) nécessaires au calcul de a^n, on a donc (on compte ne compte pas la division par 2 ni le reste, et les multiplications + 2 appels de fonctions)

$$c(n) = 4 + c(n/2) = 4 + c(n/2) = 12 + c(n/4) = 6k + c(\frac{n}{2^k})$$

On arrête lorsque $\frac{n}{2^2} < 1$ ie $2^k > n$ ou encore $k > \log_2(n)$. Ainsi $c(n) \approx 6\log_2(n) = O(\ln(n))$

Dans la première version itérative ou récursive, on obtient plutôt une complexité en $O(n)$ qui devient vite impraticable.

2.4 Version itérative

Donnons maintenant une version itérative de cette fonction. Par exemple, pour calculer a^{13}, on va en fait écrire $a^{13} = a \times (a^6)^2 = \cdots = a^{1+4+8}$.

Il s'agit d'utiliser la décomposition binaire de n, l'exposant.

L'idée de l'algorithme itératif est de maintenir une variable valant $a^{(2^i)}$ et d'effectuer la multiplication seulement si 2^i intervient dans la décomposition en binaire de n.

Tout d'abord un rappel sur la décomposition en base 2

```
    def decompose(n, b=2):  # b  est la base utilisée, 2 par défaut.
        L = []
        while n > 0:
            L.append(n % b)
            n = n // b
        return L  # L est la liste des chiffres, dans l'ordre de poids␣
↪croissant.
```

```
    decompose(13), decompose(32)
```

```
([1, 0, 1, 1], [0, 0, 0, 0, 0, 1])
```

```
    def puissance_rapide_iter(a, n):
        p = a # les puissances a^(2^i)
        res = 1
        i = 0 # pour se fixer les idées, on pense au numéro de l'itération.
        while n > 0:
            if n % 2 == 1:  # le ième bit est à 1
                res = res * p  # dans ce cas on multiplie le résultat par␣
↪a^(2^i)
            p = p * p  # on passe à a^(2^(i + 1))
            n = n // 2  # oublions le dernier bit de l'écriture de n
            i = i + 1
        print(i)  # le nombre de passages dans la boucle.
        return res
```

```
    puissance_rapide_iter(2, 80000)
    compteur = 0
    p(2, 80000)
    compteur
```

```
17
```

```
    21
```

```
    puissance_rapide_iter(2, 20000)
    compteur = 0
    p(2, 20000)
    compteur
```

```
15
```

```
    19
```

Pour poursuivre : inverser les éléments d'un tableau, recherche dichotomique.