Fonctions récursives

Algorithmique

Semestre 2 – 2019 / 2020

Objectifs

- Comprendre qu'il existe plusieurs façons d'aborder la notion d'itération en algorithmique et dans la plupart des langages de programmation.

- Apprendre à concevoir une solution récursive à un problème.

Introduction (1/3)

- Plusieurs façons d'aborder la notion d'itération en algorithmique et dans la plupart des langages de programmation
  - avec des boucles (while, for, loop, etc.)
  - avec la récursivité

- Montrer comment concevoir une solution récursive pour un problème.

- Une définition **récursive** est une définition dans laquelle intervient le nom qu'on est en train de définir.

Exemples

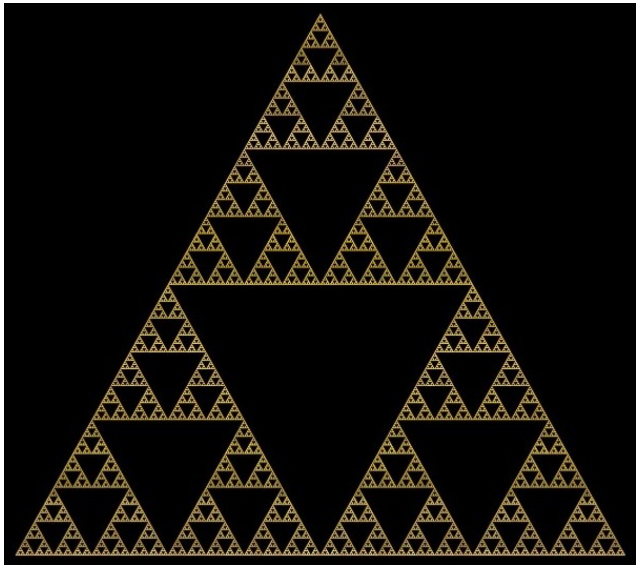
- Une personne A est un **descendant** d'une autre personne B si et seulement si
  - soit A est un enfant (fils ou fille) de B
  - soit A est un enfant (fils ou fille) d'un **descendant** de B
- Un **entier naturel positif ou nul** est
  - soit l'entier 0
  - soit le successeur d'un **entier naturel positif ou nul**.

Introduction (2/3)

- On trouve des définitions récursives dans beaucoup de domaines :
  - Linguistique :
    - Dictionnaire : chaque mot du dictionnaire est défini par d'autres mots eux-mêmes définis par d'autres mots dans ce même dictionnaire.
  - Art :
    - Œuvres d'Escher
    - Image contenant une image similaire
    - Les poupées russes
    - Caméra qui filme devant un miroir

Introduction (2/3)

- Biologie :
  - Motif de végétaux (fougère, cœur d'un tournesol, chou romanesco)
  - Processus de développement (Nautile)



- Mathématique :
  - Suites récurrentes
  - Fractales (triangle de Sierpinski)

- En informatique, on trouve :
  - des **fonctions récursives**, c'est-à-dire qui s'appellent elles-mêmes
  - des **structures de données récursives**, c'est-à-dire qui se définissent en fonction d'elles-mêmes

Introduction (3/3)

- Dans la plupart des langages de programmation, **une fonction peut s'appeler elle-même**, on parle alors de **fonction récursive**

- Les fonctions récursives, permettent, pour certains types de problèmes
  - d'écrire plus facilement les programmes
    - → l'écriture se déduit de la définition de la fonction
  - de vérifier plus facilement que les programmes sont corrects
    - → preuve par induction

- Il existe aussi la possibilité d'avoir des **définitions mutuellement récursives** c'est-à-dire une fonction qui en appelle une 2e, qui en appelle une 3e, ..., qui appelle la première.

Rappels sur les fonctions en Python (1/4)

- Une fonction permet de définir un calcul à un endroit du programme (**définition**) et de l'utiliser (**appel**) partout dans le reste du programme.

- **DÉFINITION d'une fonction**

```
def nom(param1,param2,... paramN):
  corps
```

  - la définition est enregistrée pour son utilisation ultérieure

Rappels sur les fonctions en Python (2/4)

- **APPEL d'une fonction**

```
nom(arg1,arg2,... argN)
```

  – à l'appel de `nom(arg1,arg2,...,argN)`, *`corps`* est exécuté après avoir remplacé respectivement chaque *`param`* par la valeur de *`arg`*

- Si la fonction doit produire une valeur, **`return`** va permettre de sortir de la fonction, et remplacer l'appel de fonction par la valeur retournée.
  – S'il n'y a pas de `return` la fonction se termine après la dernière instruction (et renvoie `None`).

Rappels sur les fonctions en Python (3/4)

- Qu'affichent les programmes suivants ?

```
def pair_ou_impair(a):
  if (a % 2 == 0):
    return "pair"
  else:
    return "impair"

print(pair_ou_impair(2))
```

Rappels sur les fonctions en Python (4/4)

```python
#on peut aussi écrire
def successeur(a):
  return a + 1

print(successeur(2))
```

```python
# composition de fonctions
print(pair_ou_impair(successeur(20)))
print(pair_ou_impair(successeur(19)))
```

Exemple : Affichage de nombres

- On veut écrire un programme qui,
  - étant donné un entier n>0,
  - affiche les nombres de 1 à n par ordre *croissant*.

```
# avec une boucle while
n = eval(input())
i = 1
while i <= n:
  print("i : ",i)
  i = i + 1
```

```
# avec une boucle for
n = eval(input())
for i in range(1,n+1):
  print("i : ",i)
```

Exemple : Affichage de nombres

- Avec une **fonction récursive** de paramètre n :
  - ce qu'on sait faire facilement : quand le paramètre **n vaut 1**
    - → on **affiche 1** et c'est fini
  - quand n > 1,
    - → on affiche les entiers de 1 à n-1
    - → puis on affiche n

Exemple : Affichage de nombres

- le 1er cas est appelé **cas trivial** ou **cas d'arrêt de la récursivité**
  - on s'arrête quand n vaut 1

- le 2e cas est appelé le **cas récursif**

  → on ramène le problème à un sous-problème plus simple, c'est-à-dire de taille plus petite :
    - on fait un appel récursif à la fonction avec n-1

      → on affichera donc les entiers de 1 à n-1
    - on affiche ensuite n

Exemple : Affichage de nombres

```python
# avec une fonction récursive
def affichage_ordre_croissant(n):
  if n == 1:
    print("n : ",1)
  else:
    affichage_ordre_croissant(n-1)
    print("n : ",n)

affichage_ordre_croissant(5)
```

Exemple : Affichage de nombres – ordre décroissant

- Maintenant on veut écrire un programme qui,
  - étant donné un entier n > 0,
  - affiche les nombres de n à 1 par **ordre *décroissant***.

```
# avec une boucle while
n = eval(input())
i = n
while i >= 1:
  print("i : ",i)
  i = i - 1
```

→On a dû changer
  - l'initialisation de la variable de contrôle
  - la condition de la boucle while

Exemple : Affichage de nombres – ordre décroissant

```
# avec une boucle for
n = eval(input())
for i in range(n,0,-1):
  print("i : ",i)
```

→ On a dû changer la séquence d'entiers que parcourt le for

Exemple : Affichage de nombres – ordre décroissant

```
# avec une fonction récursive
def affichage_ordre_decroissant(n):
  if n == 1:
    print("n : ",1)
  else:
    print("n : ",n)
    affichage_ordre_decroissant(n-1)

affichage_ordre_decroissant(5)
```

→On a *seulement* inversé l'ordre d'affichage et l'appel récursif dans le cas où n > 1

Exemple : Affichage de nombres – ordre décroissant

- Pour **le cas trivial** ou **cas d'arrêt de la récursivité**
  - *on s'arrête quand n vaut 1*
  - c'est toujours ce qu'on sait faire facilement : quand le paramètre n vaut 1, on affiche 1 et c'est fini

  → INCHANGÉ par rapport à l'ordre croissant

- Pour le **cas récursif**
  - quand n > 1
    - *on affiche n*
    - *on affiche* ensuite *les entiers de 1 à n-1*

  → ON A INVERSÉ par rapport à l'ordre croissant

Remarques

- On constate qu'une fonction récursive comporte :
  - (au moins) un **cas trivial (cas d'arrêt)** pour lequel on a **directement le résultat**
  - (au moins) un **appel récursif** dans lequel on **ramène le problème à un sous-problème plus simple** dans lequel on aura diminué la **"taille du problème"**

Calcul de la puissance $n^{ième}$ d'un entier strictement positif

- Soient a>0 et n>=0, la puissance $n^{ième}$ de a est définie par
  - pour n=0 : $a^0 = 1$
  - pour n > 0 : $a^n = a \times a^{n-1}$

- Cette définition mathématique comporte
  - un point de départ *(n = 0)* pour lequel on a directement le résultat *($a^0 = 1$)*
  - un calcul de la puissance $n^{ième}$ de a qui fait appel au calcul de la puissance $(n-1)^{ième}$ de a.

Calcul de la puissance n^{ième} d'un entier strictement positif

- C'est donc une définition récursive avec :
  - un cas **d'arrêt** de la récursivité : n = 0
  - un cas de calcul récursif $a^n = a \times a^{n-1}$ dans lequel on fait **diminuer la taille du problème** (n-1)

- On constate donc que, pour tout n>0, le calcul finira par s'arrêter quand n atteindra le cas d'arrêt et la valeur 0.

Calcul de la puissance $n^{ième}$ d'un entier strictement positif

- On se calque sur la définition mathématique pour écrire en Python la définition de la fonction **puissance** à 2 paramètres a *(avec a>0)* et n *(avec n>=0)*
  - un cas d'arrêt de la récursivité : **n=0**
    - → le résultat est 1
  - un cas de calcul récursif : **n>0**
    - → pour calculer a^n, on multiplie n avec le résultat du calcul de a^{n-1}

Calcul de la puissance n^{ième} d'un entier strictement positif

```
# calcul de a à la puissance n, pour a>0 et n>=0
def puissance(a,n):
  if n == 0:
    # cas d'arrêt
    return 1
  else:
    # n > 0 cas récursif
    return a * puissance(a,n-1)

puissance(2,4)
puissance(2,0)
```

Calcul de la puissance $n^{ième}$ d'un entier strictement positif

```
# On peut aussi écrire sans le else :
def puissance_bis(a,n):
  if (n == 0):
    # cas d'arrêt
    return 1
  # n > 0 cas récursif
  return a * puissance_bis(a,n-1)

puissance_bis(2,4)
puissance_bis(2,0)
```

Démarche pour écrire une fonction récursive

- On commence par chercher le cas simple, c'est-à-dire celui qui ne nécessite pas de rappeler récursivement la fonction :
  - Pour l'affichage, c'est le cas où n=1 (on affiche 1)
  - Pour la puissance, c'est le cas où n=0 (on sait que $a^0 = 1$)

Démarche pour écrire une fonction récursive

- On cherche ensuite le sous-problème récursif (sous-problème de taille réduite par rapport au problème) pour rappeler la fonction récursive pour chaque sous-problème à résoudre
  - Pour l'affichage, c'est afficher les n-1 entiers
  - Pour la puissance, c'est calculer a^{n-1}

- Vérifier que la fonction se termine

  → Atteint-on au moins un cas d'arrêt ?

Problème de la terminaison

- Une fonction récursive doit obligatoirement avoir au moins un cas d'arrêt

- Les appels récursifs doivent permettre d'atteindre ce cas d'arrêt.

QUESTIONS OBLIGATOIRES

- Soit la définition de la fonction f ci-dessous.

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

→ Que se passe t-il si on veut exécuter f(3)?

QUESTIONS OBLIGATOIRES

- Soit la définition de la fonction f ci-dessous.

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

→Que se passe t-il si on veut exécuter f(3)?

On a oublié le cas d'arrêt !

f(3) appelle f(2) puis f(1) ; f(0) ; f(-1)…

QUESTIONS OBLIGATOIRES

- Soit la définition de la fonction g ci-dessous.

```
def g(n):
  if n==0:
    return 1
  else:
    return n * g(n+1)
```

→ Que se passe t-il si on veut exécuter g(3)?

QUESTIONS OBLIGATOIRES

- Soit la définition de la fonction g ci-dessous.

```
def g(n):
  if n==0:
    return 1
  else:
    return n * g(n+1)
```

→ Que se passe t-il si on veut exécuter g(3)?

n est croissant donc n'atteint pas 0 s'il est positif

g(3) appelle g(4) puis g(5)…

QUESTIONS OBLIGATOIRES

- Soit la définition de la fonction h ci-dessous.

```
def h(n):
  if n==0:
    return 1
  else:
    return n * h(n)-1
```

→ Que se passe t-il si on veut exécuter h(3)?

QUESTIONS OBLIGATOIRES

- Soit la définition de la fonction h ci-dessous.

```
def h(n):
   if n==0:
      return 1
   else:
      return n * h(n)-1
```

→Que se passe t-il si on veut exécuter h(3)?

attention : h(n) – 1 != h(n-1)

h(n) rappelle indéfiniment le calcul de h(n)

QUESTIONS OBLIGATOIRES

- Soit la définition de la fonction p ci-dessous.

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

→Que se passe t-il si on veut exécuter p(3)?

On n'atteint le cas d'arrêt que si p est pair : p(3) appelle p(1) puis p(-1)…

Problème du domaine de définition

- Écrire la fonction factorielle qui,
  - étant donné un entier n>=0,
  - calcule le n[ième] terme de la suite F_n définie par :
    - pour n=0 : $F_0 = 1$
    - pour n>0 : $F_n = n \times (n-1) \times \ldots \times 1 = n \times F_{n-1}$

Problème du domaine de définition

- Pour ce problème, l'écriture d'une fonction récursive en Python s'impose d'elle même en suivant la définition mathématique avec
  - un cas d'arrêt de la récursivité : **n=0**

    → le résultat est 1
  - un cas de calcul récursif : **n>0**

    → pour calculer F_n, on doit multiplier n avec le résultat du calcul de F_{n-1} *(sous-problème de taille réduite)*

Fonction factorielle

```
def factorielle(n):
   if n == 0:
      return 1
   else:
      return n * factorielle(n-1)

print(factorielle(5))
```

Traces de la fonction factorielle

- On ajoute des impressions *(avec print)*

→ Permet de tracer à chaque appel la valeur de l'argument et le résultat calculé

```
def factorielle_trace_argument_et_resultat(n):
  # ajout des impressions de n et du résultat calculé jusque là
  if n == 0:
    resultat = 1
  else:
    resultat = n * factorielle_trace_argument_et_resultat(n-1)
  print('Pour n = ', n, ' , factorielle(‘, n,') est égal à :’,
      resultat)
  return resultat

factorielle_trace_argument_et_resultat(5)
```

Traces de la fonction factorielle

```
def factorielle_trace_argument_et_resultat(n):
  # ajout des impressions de n et du résultat calculé jusque là
  if n == 0:
    print('Pour n = ', n, ' , factorielle(‘,n,
      ') est égal à :', 1)
    return 1
  else:
    resultat = n * \
      factorielle_trace_argument_et_resultat(n-1)
    print('Pour n = ', n, ‘ ,
      factorielle(',n,') est égal à :', resultat)
    return resultat

factorielle_trace_argument_et_resultat(5)
```

Pour n = 0 , factorielle(0) est égal à : 1

Pour n = 1 , factorielle(1) est égal à : 1

Pour n = 2 , factorielle(2) est égal à : 2

Pour n = 3 , factorielle(3) est égal à : 6

Pour n = 4 , factorielle(4) est égal à : 24

Pour n = 5 , factorielle(5) est égal à : 120

Qu’en est-il de factorielle(-5) ?

Qu’en est-il de factorielle(-5) ?

- La fonction factorielle est définie sur N donc l'argument doit être positif ou nul, sinon on va vouloir calculer factorielle(-1), puis de -2, etc. et on boucle sur les entiers négatifs...

- Écrire une version qui permette de résoudre ce problème signifie qu'on va devoir **vérifier la validité de la donnée**.

Factorielle : Gestion des valeurs négatives

```
def factorielle_argument_valide(n):
  # tester si l'argument est positif ou nul pour faire les calculs
  if n < 0:
    print('Pour n =', n,
      ', qui est négatif, factorielle(',n,') est non définie')
  else:
    if (n == 0):
      print('Pour n =', n, ', factorielle(',n,') est égal à :', 1)
      return 1
    else:
      resultat = n * factorielle_argument_valide(n-1)
      print('Pour n =', n, ', factorielle(',n,') est égal à :', resultat)
      return resultat

factorielle_argument_valide(-5)
```

Factorielle : Gestion des valeurs négatives

- Avec utilisation d'un **assert**

```
def factorielle_argument_valide_bis(n):
   # si l'argument est négatif le programme s'arrête
   assert n >= 0,
     "Pour n négatif, factorielle(n) est non définie"
   # argument positif ou nul, on peut faire les calculs
   if n == 0:
     print('Pour n =', n,
       ', factorielle(',n,') est égal à :', 1)
     return 1
   else:
     resultat = n * factorielle_argument_valide_bis(n-1)
     print('Pour n =', n,
       ', factorielle(',n,') est égal à :', resultat)
     return resultat

factorielle_argument_valide_bis(-5)
```

Factorielle : Gestion des valeurs négatives

- Il ne semble pas judicieux de tester à chaque appel récursif si n < 0.
  - Si n < 0 dés le départ il faut signaler l'erreur de domaine
  - mais si n >= 0, le calcul permettra à n d'atteindre la valeur 0 (et donc arrêter le calcul) avant une valeur négative.

- On va **séparer** la vérification du domaine de validité de l'argument du calcul proprement dit quand l'argument est correct.

Factorielle : Gestion des valeurs négatives

```
def factorielle_verification_du_domaine(n):
  # tester si l'argument est positif ou nul pour faire les calculs
  # cas d'erreur, pas de calcul de n!
  assert n >= 0, "Pour n négatif, factorielle(n) est non définie"
  # n est dans le domaine, on lance le calcul avec la fonction factorielle
  return factorielle(n)

def factorielle(n):
  # fonction sur N avec argument validé donc n >= 0
  if n == 0:
    return 1
  else:
    return n * factorielle(n-1)

factorielle_verification_du_domaine(-5)
factorielle_verification_du_domaine(5)
```

EXERCICES : Fonctions récursives sur les entiers

- **Pour chacune des fonctions suivantes, on supposera la validité de l'argument sans la vérifier**

Fibonacci

- Écrire la fonction *Fibonacci* qui, étant donné un entier n>0, calcule Fib_n, le $n^{ième}$ terme de la suite définie par
  - $Fib_0 = 0$
  - $Fib_1 = 1$
  - $Fib_{n+2} = Fib_{n+1} + Fib_n$

- Quel est le résultat de l'appel suivant ?

```
fibonacci(12)
```

Fibonacci

- Cas trivial d'arrêt de la récursivité
  - `n == 0` : dans ce cas le résultat est 1
  - `n == 1` : dans ce cas le résultat est 1

- Cas de calcul récursif : n > 1
  - pour calculer F_n, on doit additionner les résultats des calculs de F_{n-1} et F_{n-2}

Fibonacci

```python
def fibonacci(n):
    # 2 cas d'arrêt : n == 0 retournant 0
    # et n == 1 retournant 1
    if n == 0:
      return 0
    if n == 1:
      return 1
    # cas général
    # 2 appels récursifs avec la taille du problème diminuée
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Suite

- Écrire la fonction *suite* qui, étant donné un entier n >= 0, calcule U_n, le $n^{\text{ième}}$ terme de la suite définie par
  - $U_0 = 2$
  - $U_{n+1} = (1 + 3 \times U_n) / (3 + U_n)$

- Quel est le résultat de l'appel suivant ?

```
suite(12)
```

Suite

- Cas trivial d'arrêt de la récursivité
  - $n = 0$, dans ce cas, le résultat est 2
- Cas de calcul récursif : $n > 0$
  - pour calculer U_n, on utilise le résultat du calcul de U_{n-1}

Suite

```python
def suite(n):
    # 1 cas d'arrêt : n = 0 retournant 2
    if (n == 0):
        return 2
    # cas général
    # 2 appels récursifs avec la taille du problème diminuée
    return (1 + 3 * suite(n-1)) / (3 + suite(n-1))
```

Peut-on améliorer cette solution ?

Suite

- On remarque que `suite(n-1)` est calculée 2 fois à chaque appel récursif.

- On peut améliorer ce code en mémorisant le résultat de `suite(n-1)` calculé une **seule** fois pour l'utiliser 2 fois.

- La variable locale `resultat_intermediaire` sera différente à chaque appel.

- Quel est le résultat de l'appel suivant ?

```
suite_bis(12)
```

Suite

```
def suite_bis(n):
    # cas d'arrêt : n = 0 retournant 2
    if (n == 0):
        return 2
    # cas général
    # 1 seul appel récursif
    # conserver le résultat dans une variable locale
    resultat_intermediaire = suite_bis(n-1)
    # on utilise le résultat sans le recalculer
    return (1 + 3 * resultat_intermediaire) / \
      (3 + resultat_intermediaire)
```

Suite

- Ecrire la fonction *pgcd* qui, étant donnés 2 entiers a > 0 et b > 0, calcule le plus grand commun diviseur de a et de b défini par :
  - pgcd(a,b) = a si a = b
  - pgcd(a,b) = pgcd(min(a,b),|a-b|) sinon

- Ecrire la fonction pgcd sans utiliser les fonctions python min et abs

- Quels sont les résultats pour les appels suivants ?

```
pgcd(15,21)
```

```
pgcd(11,19)
```

```
pgcd(24, 48)
```

PGCD

- Cas trivial d'arrêt de la récursivité
  - a=b : dans ce cas le résultat est a

- Cas de calcul récursif : a != b
  - Si a>b, le résultat est le même que celui du pgcd de b et a-b
  - Si b>a, le résultat est le même que celui du pgcd de a et b-a

→C'est un sous-problème du PGCD initial

→Petit à petit les deux nombres diminuent

→La différence est forcément positive

→Si a et b toujours différents, la différence va aboutir à 1

PGCD

- PGCD(15,21)
  - →Plus petit : 15, différence : 6 → =PGCD(15,6)
  - →Plus petit : 6, différence : 9 → = PGCD(6,9)
  - →Plus petit : 6, différence : 3 → = PGCD(6,3)
  - →Plus petit : 3, différence : 3 → = PGCD(3,3)
  - →a et b sont égaux et valent 3 donc PGCD(15,21)=3

PGCD

```
def pgcd(a,b):
    # cas d'arrêt : a = b retournant a
    if a == b:
        return a
    # cas général a != b,
    else: # on cherche le plus petit des 2
        if (a > b): # appel récursif
            # avec le plus petit(b) et la différence (a-b)
            return pgcd(b,a-b)
        else: # appel récursif
            # avec le plus petit(a) et la différence (b-a)
            return pgcd(a,b-a)
```

PGCD

- On peut aussi écrire les fonctions :
  - ***min*** qui, appliquée à 2 entiers, retourne le plus petit des 2
  - ***val_abs*** qui retourne la valeur absolue de son paramètre

et les composer

- Quels sont les résultats pour les appels suivants ?

```
pgcd_bis(15,21)
```

```
pgcd_bis(11,19)
```

```
pgcd_bis(24, 48)
```

PGCD

```
def min(a,b):
    if a < b:
        return a
    else:
        return b

def val_abs(a):
    if a >= 0:
        return a
    else:
        return -a
```

PGCD

```
def pgcd_bis(a,b):
    # cas d'arrêt : a = b retournant a
    if a == b:
        return a
    # cas général a != b
    else: # appel récursif
        # avec le plus petit des deux et la valeur absolue de leur différence
        return pgcd_bis(min(a,b),val_abs(a-b))
```

- mais le test a > b est ici effectué 2 fois :
  - si a > b alors val_abs(a-b) = a-b
  - sinon val_abs(a-b) = b-a

```
def pgcd_ter(a,b):
    # cas d'arrêt : a = b retournant a
    if a == b:
        return a
    # cas général a != b
    if a > b:
      grand, petit = a, b
    else:
      petit, grand = a, b
    return pgcd_ter(petit, grand - petit)
```