TD 4 : Récursivité (1)

Info1.Algo1

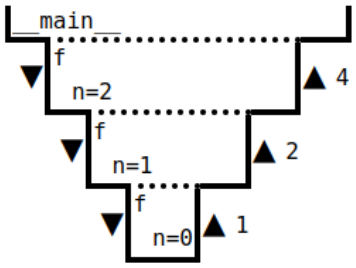
2022-2023 Semestre Impair

Exercices : Récursivité

On appelle **fonction récursive** une fonction qui s'appelle *(directement ou indirectement)* elle-même.

```
def f(n):
  if n==0: # Cas d'arret
      return 1
  else: # Cas recursif
      return 2*f(n-1) # Diminution taille du probleme
```

Un **schéma temps-mémoire** trace les **appels** de fonction avec la **valeur des paramètres et du retour**. Exemple pour l'appel de `f(2)`



Exercice 1 : analyse d'une fonction récursive

On considère la fonction suivante :

```
def calcul(n):
  if n==0:
    return 0
  else:
    s = calcul(n-1)+n
    return s
```

1) a) Déterminer la valeur retournée par les appels suivants :
- `calcul(0)`
- `calcul(1)`
- `calcul(2)`
- `calcul(3)`
- `calcul(10)`

b) Décrire par une phrase la valeur retournée lors de l'appel `calcul(n)`.
2) Faire la représentation temps/mémoire de l'appel `calcul(3)`.

Exercice 2 : problèmes d'arrêt

On considère les 4 fonctions suivantes :

```
def f1(n):
  return n*f1(n-1)

def f2(n):
  if n==0:
    return 1
  else:
    return n*f2(n+1)

def f3(n):
  if n==0:
    return 1
  else:
    return n*f3(n)-1

def f4(n):
  if n==0:
    return 1
  else:
    return n*f4(n-2)
```

Que se passe-t-il lorsqu'on appelle chacune de ces fonctions avec l'entier `3` ?

Exercice 3 : récursivité mutuelle

On considère les deux fonctions mutuellement récursives suivantes :

```
def est_pair(n):
  if n==0:
    return True
  else:
    return est_impair(n-1)

def est_impair(n):
  if n==0:
    return False
  else:
    return est_pair(n-1)
```

1) Déterminer la valeur retournée par ces deux fonctions lorsque le paramètre est `n=0`, puis `n=1`, puis `n=2`.
2) Faire une représentation temps/mémoire de l'appel `est_pair(3)`.

Exercice 4 : suite de Fibonacci

La suite de Fibonacci (F_n) (pour n entier positif) est définie par récurrence de la façon suivante :

$$\begin{cases} F_n = 0 & \text{si } n = 0 \\ F_n = 1 & \text{si } n = 1 \\ F_n = F_{n-1} + F_{n-2} & \text{sinon.} \end{cases}$$

Afin de calculer le terme de cette suite, on propose la fonction récursive suivante :

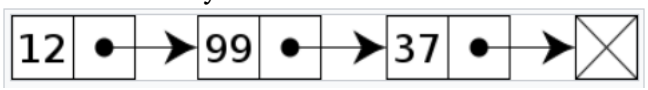
```
def fibonacci(n):
  if n<=1:
    return n
  else:
    return fibonacci(n-1)+fibonacci(n-2)
```

1) a) Déterminer la valeur retournée par les appels suivants :
- `fibonacci(0)`
- `fibonacci(1)`
- `fibonacci(2)`
- `fibonacci(3)`
- `fibonacci(4)`
- `fibonacci(5)`

b) La fonction `fibonacci` retourne-t-elle le résultat attendu ?
2) Faire une représentation temps/mémoire de l'appel `fibonacci(4)` et expliquer en quoi cette fonction est mal écrite.
3) Réécrire une nouvelle fonction récursive `fibonacci` prenant en paramètre un entier `n` et retournant le tuple constitué des 2 termes F_n et F_{n+1}.

Exercices : Listes chaînées

Une **liste chaînée** est une **succession de cellules** constituées chacune :
- d'une valeur associée à l'élément (la **tête** de la liste)
- d'un moyen d'accéder à la cellule suivante (la **queue** de la liste)



L'**implémentation** utilisée est la suivante :

```
def creer_liste_vide():
  return None

def creer_liste(t,q):
  return t,q

def tete(liste):
  return liste[0]

def queue(liste):
  return liste[1]

def est_vide(liste):
  return liste==None
```

Exercice 5 : produit des éléments

Écrire la fonction récursive `produit_elements` qui accepte en paramètre une liste chaînée d'entiers et retourne le produit des éléments de cette liste.
Indication : le produit des éléments d'une liste vide est `1`.

Exercice 6 : dernier élément

Écrire la fonction récursive `dernier_element` qui accepte en paramètre une liste chaînée et retourne la valeur du dernier élément de cette liste, s'il existe.
Indication : Si la liste est vide, une erreur est générée.

Exercice 7 : appartient

1) Écrire la fonction récursive `appartient` qui accepte en paramètres une liste chaînée ainsi qu'une valeur du même type que les éléments de cette liste.
La fonction retourne `True` si la valeur appartient à la liste et `False` sinon.
2) Modifier la fonction `appartient` afin d'écrire la fonction récursive `appartient_liste_triee` qui accepte en paramètres une liste chaînée **triée par ordre croissant** ainsi qu'une valeur du même type que les éléments de cette liste.

Exercice 8 : lecture i-ème élément

On souhaite écrire la fonction `lire_element` qui accepte en paramètres une liste chaînée ainsi qu'un indice `i` entier. La fonction retourne la valeur de l'élément situé à l'indice `i` dans la liste.

L'indice `0` *correspond, comme dans un tableau, au premier élément (c'est-à-dire la tête), de la liste chaînée.*

1. Supposer tout d'abord que l'indice `i` est valide afin de pouvoir écrire la fonction récursive `lire_element`.
2. Rajouter des assertions permettant de vérifier la validité de l'indice `i`. On pourra être amené à séparer la fonction en deux, dont une partie non récursive chargée de la vérification du domaine de validité.

Exercice 9 : nombres d'éléments supérieurs à la moyenne

L'objectif est d'écrire une fonction récursive qui accepte en paramètre une liste **non vide** d'entiers et retourne le nombre d'éléments de cette liste qui sont strictement supérieurs à la moyenne des éléments.
Écrire la fonction `nb_superieurs` prenant trois arguments :

— La liste restant à parcourir,
— La somme des éléments déjà parcourus lors des appels récursifs précédents (à 0 lors du 1^{er} appel),
— Le nombre des éléments déjà parcourus lors des appels récursifs précédents (à 0 lors du 1^{er} appel).

Indications :

— Calculer la somme et le nombre des éléments lors des appels récursifs.
— En déduire la moyenne lors du cas d'arrêt.
— Effectuer le comptage de éléments supérieurs à la moyenne lors du retour des appels récursifs.