La récursivité

Introduction

La récursivité est une méthode de résolution de problèmes qui consiste à décomposer le problème en sous-problèmes identiques de plus en plus petits jusqu'à obtenir un problème suffisamment petit pour qu'il puisse être résolu de manière triviale.

Définition de itérable : adjectif singulier invariant en genre
Qui peut subir une itération, c'est à dire être répété.

Exemple :

Étant donné une liste d'entiers `L=[2,12,1,8,5,10,20]`, calculer la somme des éléments de cette liste. Comme les listes sont **itérables**, nous pouvons simplement résoudre ce problème avec l'un de ces algorithmes que l'on dit **itératif** :

Données : liste ← une liste d'entiers
fonction somme(liste)
S← 0
n← longueur de la liste
Pour i allant de 0 à n **faire**
  S←S+liste[i]
renvoyer S

Données : liste ← une liste d'entiers
fonction somme(liste)
S← 0
Pour chaque element de liste **faire**
  S←S+element
renvoyer S

Supposons maintenant que nous n'ayons pas la possibilité de faire de "boucles"
On peut alors aborder le problème sous un autre angle:
La somme des termes de cette liste est:
2 + (la somme des termes de [12,1,8,5,10,20])
Soit : 2 + (12 + (la somme des termes de [1,8,5,10,20])
et ainsi de suite...
jusqu'à : 2 + (12 + (1 + (8 + (5 + (10 + (la somme des termes de [20])))))))
il est clair que : la somme des termes de [20] est 20
Au final le calcul à faire est : 2 + (12 + (1 + (8 + (5 + (10 + (20)))))) = 58

Considérons alors une fonction `sommeliste(liste)` et qui renvoie le résultat de la somme des éléments de la liste.
L'algorithme ci-dessous que l'on dit **récursif** réalise cette seconde approche:

Données : liste ← une liste d'entiers
fonction sommeliste(liste)
Si la longueur de la liste est égale à 1 **alors**
  renvoyer liste[0]
Sinon
  renvoyer liste[0] + sommeliste(liste[1:])

Exercice 1:

Écrire en Python cette fonction, et la tester avec plusieurs exemples.

Exercice 2:

Comparer les vitesses d'exécutions des programmes version itérative et version récursive de l'exemple précédent.
Pour tester la vitesse d'exécution d'une fonction on utilise le module `timeit`, comme le montre le code ci-dessous pour une liste de 1000 entiers choisis aléatoirement entre 0 et 100 avec le module `random`.

```
from timeit import default_timer as timer
from random import randint

# les deux fonctions ici

L=[randint(0,100) for i in range(1000)]

debut=timer()
print(sommeliste(L))
fin=timer()
print(fin-debut)

debut=timer()
print(somme(L))
fin=timer()
print(fin-debut)
```

Question 1:

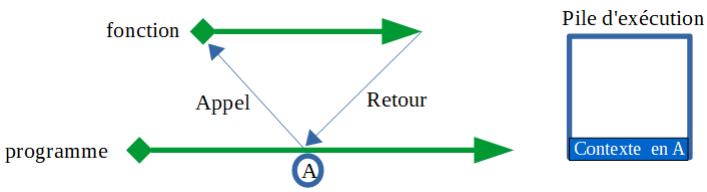
Que pouvez-vous en conclure *(pour cet exemple)*?

Votre réponse

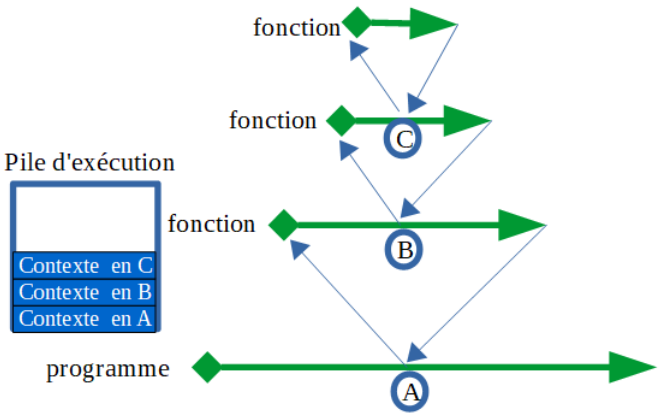
Comment fonctionne un programme récursif?

Un programme est une suite d'instructions, son exécution peut être représentée par un parcours de chemin ayant une origine et une extrémité.

Lors de l'appel d'une fonction, cette suite est interrompue le temps de cette fonction, avant de reprendre à l'endroit où le programme s'est arrêté.
Au moment où débute cette bifurcation, le processeur sauvegarde un certain nombre d'informations : adresse de retour, état des variables, etc. Toutes ces données forment ce qu'on appelle le contexte du programme, et elles sont stockées dans ce qu'on appelle la pile d'exécution.



À la fin de l'exécution de la fonction, le contexte est sorti pour permettre la poursuite de l'exécution du programme.



Lors de l'exécution d'une fonction récursive, chaque appel récursif conduit, au moment où il se produit, à un empilement du contexte dans la pile d'exécution.
Lorsqu'au bout de n appels se produit la condition d'arrêt, les différents contextes sont progressivement dépilés pour poursuivre l'exécution de la fonction.

Il est important de prendre conscience qu'une fonction récursive s'accompagne d'une complexité qui va croître avec le nombre d'appels récursifs (en général linéairement, mais ce n'est pas une règle, cela dépend du contenu du contexte).

Les trois règles...

Tout comme les robots d'Azimov, les algorithmes récursifs doivent obéir à trois règles...

• Un algorithme récursif doit avoir un " état trivial " , cela permet d'avoir une condition d'arrêt.
Dans notre exemple , il s'agit de : si la liste est de longueur 1 alors on renvoie le seul élément de la liste.

• Un algorithme récursif doit conduire vers cet " état d'arrêt ", cela permet de ne pas faire une infinité d'appels récursifs.
Dans notre exemple, à chaque appel récursif, la liste est diminuée d'un élément donc nécessairement elle finira par n'en n'avoir plus qu'un.

• Un algorithme récursif s'appelle lui même...

Itératif vs Récursif

Il n'existe pas de réponse définitive à la question de savoir si un algorithme récursif est préférable à un algorithme itératif ou le contraire. Il a été prouvé que ces deux paradigmes de programmation sont équivalents ; autrement dit, tout algorithme itératif possède une version récursive, et réciproquement.

Le choix du langage peut aussi avoir son importance : un langage fonctionnel tel Caml est conçu pour exploiter la récursivité et le programmeur est naturellement amené à choisir la version récursive de l'algorithme qu'il souhaite écrire. À l'inverse, Python, même s'il l'autorise, ne favorise pas l'écriture récursive (limitation basse (1000 par défaut) du nombre d'appels récursifs).

Enfin, le choix d'écrire une fonction récursive ou itérative peut dépendre du problème à résoudre : certains problèmes se résolvent particulièrement simplement sous forme récursive

Un autre exemple...

Le problème : Déterminer le minimum d'une liste d'entiers.

Supposons que nous ayons une fonction `minimum(a,b)` qui renvoie le plus petit des entiers a et b. et une liste L = $[a_0,a_1,a_2,...a_n]$ d'entiers dont il faut déterminer le minimum.

Version itérative

On initialise le minimum à mini=a_0
On parcours la liste en appelant à chaque étape : `minimum(mini,`a_i`)`

```
fonction miniIt
Données : L ← une liste d'entiers
mini← L[0]
Pour i allant de 1 à n faire
    mini=minimum(mini,L[i])
renvoyer mini
```

Version récursive

Le minimum de la liste L est le minimum entre a_0 et le minimum de la liste L' = $[a_1,a_2,...a_n]$ qui est lui même le minimum entre a_1 et le minimum de la liste $[a_2,a_3,...a_n]$ et ainsi de suite...
La condition d'arrêt étant : s'il n'y a qu'un seul élément dans la liste alors le minimum de la liste est cet élément.

```
fonction miniRec
Données : L ← une liste d'entiers
Si la longueur de la liste est égale à 1 alors
    renvoyer L[0]
Sinon
    renvoyer minimum(L[0] , miniRec(L[1:])
```

Exercice 3:

Implémenter ces deux fonctions en Python (sans oublier la fonction `minimum(a,b)`) et comparer leurs temps d'exécution sur plusieurs exemples.
Imaginer un autre découpage de L, pour implémenter une récursivité du type :
`minimum(miniRec(L1),miniRec(L2))`

Exercices

Exercice 4 :

Écrire les versions itérative et récursive de fonctions renvoyant le maximum d'une liste d'entiers.

Exercice 5 :

Écrire les versions itérative et récursive de fonctions calculant x^n où $n \in \mathbb{N}$ et $x \in \mathbb{R}$.

Exercice 6 :

Écrire les versions itérative et récursive de fonctions calculant $n!$ où $n \in \mathbb{N}$
sachant que : $n! = n \times (n-1) \times (n-2) \times ... \times 2 \times 1$.
Remarque : par définition $0! = 1$