La récursivité

Algorithmie

Principe de Hofstadter : il faut toujours plus de temps que prévu, même en tenant compte du principe de Hofstadter.

1 Notions sur la récursivité

1.1 Définition

Une fonction est récursive si elle s'appelle elle même.

1.2 Illustration simple

On désire programmer une fonction permettant de calculer *de manière native* $\sum_{i=1}^{n} i$. Pour vérifier que la fonction donne le bon résultat, on le comparera au résultat bien connu : $\sum_{i=1}^{n} i = \frac{n.(n+1)}{2}$.

1.2.1 Version impérative

Activité 1 **Vérification de la conformité de l'algorithme**

Vérifier **sans l'exécuter** que la fonction suivante renvoie bien la valeur de $\sum_{i=1}^{n} i$. Exécutez ensuite le programme.

```
1 def somme_boucle(n):
      s=0
3     for i in range (n+1):
          s=s+i
5     return s
```

Pour information, un tel algorithme est dit **impératif** (par opposition à **récursif**).

1.2.2 Version récursive

Activité 2 **Vérification de la conformité de l'algorithme**

Vérifier **sans l'exécuter** que la fonction suivante renvoie bien la valeur de $\sum_{i=1}^{n} i$. Exécutez ensuite le programme.

```
1 def somme_recursion(n):
      if  n == 1:
3         return 1
      else :
5         return n+somme_recursion(n-1)
```

On constate bien que la fonction est **récursive**, car elle s'appelle elle-même. Ce n'était pas le cas dans la fonction impérative.

De plus, on constate qu'il existe une condition d'arrêt dans la fonction récursive. L'existence de la **condition d'arrêt** (ici : si n == 1, alors return 1) est une **nécessité** : si il n'y a pas de condition d'arrêt, la fonction boucle à l'infini, puisqu'elle s'auto-appelle ...

1.3 Avantages et inconvénients de la récursivité

1.3.1 Avantages

Voici quelques avantages d'un programme récursif. Ces avantages sont principalement liés au fait que le style récursif est plus proche des maths.

- Le code est souvent plus court, et plus clair ;
- En conséquence, il y a moins de risque d'erreur : un programme récursif est plus sûr qu'un programme impératif ;
- Toute personne habituée à faire des raisonnements par récurrence en mathématiques, et à manipuler des suites définies par récurrence, trouvera les programmes récursifs parfaitement naturels ;
- La conception d'un programme récursif est souvent plus simple.

1.3.2 Inconvénients

- Il y de nombreuses circonstances où un programme impératif est plus naturel ;
- La récursion n'est pas naturelle pour tout le monde ...
- Les ordinateurs actuels sont optimisés pour les programmes impératifs (c'est-à-dire avec des boucles), de sorte qu'un programme récursif sera souvent légèrement plus lent qu'un programme impératif ;
- La récursivité nécessite beaucoup de mémoire vive, surtout lorsque le nombre de récursions est élevé. L'ordinateur doit garder en mémoire toutes les opération à effectuer, et les exécuter ensuite (au lieu de les faire au fur et à mesure).

Pour illustrer le dernier point, on essaie d'imaginer ce que fait l'ordinateur pour calculer la valeur de `somme_recursion(4)`. Au premier passage, il sait qu'il doit calculer `4+somme_recursion(3)`. Il appelle `somme_recursion(3)`. Il doit donc calculer `3+somme_recursion(2)`. Il appelle `somme_recursion(2)`. Et ainsi de suite, jusqu'à tomber sur la condition d'arrêt.

Ensuite, il reprend les calculs à effectuer, dans le sens inverse, et annonce le résultat final.

2 Exemples de fonctions recursives

2.1 Calcul de n !

2.1.1 Version impérative

Activité 3 **Ecriture d'une fonction**

nom : `factorielle_boucle`
arguments : n (int)
effet : doit retourner la valeur de n ! avec un algorithme impératif (boucle for ici).

Testez ensuite votre fonction.

2.1.2 Version récursive

Activité 4 **Ecriture d'une fonction**

nom : `factorielle_recursion`
arguments : n (int)
effet : doit retourner la valeur de n ! avec un algorithme récursif.

Testez ensuite votre fonction.

2.2 Méthode de Héron

2.2.1 Calcul approché de racines carrées

Soit $x \in \mathbb{R}^{+*}$. Soit $u \in \mathbb{R}^{\mathbb{N}}$. La suite vérifiant $u_0 = 1$ et $\forall n \in \mathbb{N}, u_{n+1} = \frac{u_n}{2} + \frac{x}{2u_n}$. On démontre que u converge très rapidement vers $\sqrt{x}$.

Le but est d'écrire une fonction prenant en argument x et n et calculant u_n, et ce par une méthode impérative et récursive.

2.2.2 Version impérative

Activité 5 **Ecriture d'une fonction**

nom : `Heron_boucle`

arguments : n,x (int,float)

effet : doit retourner la valeur de u_n, qui est une valeur approchée de $\sqrt{x}$. La fonction doit être écrite de manière impérative.

Tester votre fonction.

2.2.3 Version récursive

Activité 6 **Ecriture d'une fonction**

nom : `Heron_recursion`

arguments : n,x (int,float)

effet : doit retourner la valeur de u_n, qui est une valeur approchée de $\sqrt{x}$. La fonction doit être écrite de manière récursive.

Tester votre fonction.

Attention à ne pas prendre n trop élevé. Classiquement, dans ce cas, il est prudent de ne pas dépasser 25 (sauf si vous avez programmé en faisant très attention à la complexité de l'algorithme : dans ce cas, félicitations !!). Cette subtilité sur la problématique dépasse le cadre de votre programme.

2.3 La dichotomie

L'idée est de reprendre la problématique de la résolution de f(x) = 0 par la méthode de la dichotomie, mais sous l'angle récursif. On présuppose que la fonction renvoyant f(x) est programmée. Par exemple :

```
1 def f(x):
      return x**2-2
```

Activité 7 **Ecriture d'une fonction**

nom : `dicho`

arguments : a,b,epsilon (float,float,float)

effet : doit retourner la valeur de la racine de f(x)=0 sur l'intervalle [a,b] avec une précision epsilon. La fonction doit être écrite de manière récursive.

Testez ensuite votre fonction.