Spécialité NSI en terminale
Validité et coût d'un algorithme récursif

Validité d'un algorithme récursif

Terminaison et correction

De manière générale, la terminaison d'une fonction récursive est assurée par la condition d'arrêt des appels récursifs et les valeurs passées en paramètres. La correction se prouve à l'aide d'un raisonnement « par récurrence ». Ce type de raisonnement consiste à vérifier qu'une propriété est vraie pour une valeur entière initiale puis à prouver que la propriété est héréditaire. Une propriété est héréditaire si la véracité pour un entier n quelconque entraîne la véracité pour l'entier suivant $n+1$. C'est le principe d'une échelle ; si on peut atteindre le premier barreau et si on peut passer d'un barreau quelconque au barreau suivant, alors on peut monter jusqu'en haut de l'échelle.

Considérons la fonction `fact_term` définie ci-dessous :

```
def fact_term(n, acc):
    if n > 0:
        return fact_term(n-1, acc*n)
    else:
        return acc
```

Nous allons prouver par récurrence que `fact_term(n, acc)` vaut `acc` $\times$ `n!` pour tout entier n.

— Initialisation
Pour $n = 0$, `fact_term(0, acc)` vaut `acc`, soit `acc` $\times$ $0!$. La propriété est donc vraie pour $n = 0$.

— Hérédité
Nous supposons que la propriété est vraie pour un entier n quelconque, c'est-à-dire que `fact_term(n, acc)` vaut `acc` $\times$ `n!`.
Alors `fact_term(n+1, acc)` vaut `fact_term(n, acc*(n+1))`, soit `acc` $\times$ `(n+1)` $\times$ `n!` d'après l'hypothèse. Donc `fact_term(n+1, acc)` vaut `acc` $\times$ `(n+1)!` et la propriété est vraie pour $n + 1$.

La propriété est donc démontrée pour tout entier n.

Pour la terminaison, nous avons parlé de la condition d'arrêt des appels récursifs et des valeurs des paramètres passés pour ces appels. Il est parfois nécessaire de compléter le code par un test qui permet de traiter tous les cas possibles.

Nous allons examiner cette question sur un exemple et pour cela nous considérons un code de la fonction `factorielle` définie ci-dessous.

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

Si le paramètre `n` passé à la fonction est négatif ou si ce n'est pas un entier naturel, les appels récursifs se poursuivent indéfiniment. En fait, ils se poursuivent tant que la machine le permet. On peut donc ajouter un test pour gérer ce type de problème, par exemple `elif n > 0`. Ce problème ne se pose pas avec l'écriture ci-dessous de la fonction factorielle. La terminaison est assurée pour n'importe quel nombre `n` passé en paramètre avec le code qui suit.

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

Coût d'un algorithme récursif

Le coût en temps d'un algorithme récursif est principalement déterminé par le nombre d'appels récursifs en fonction de n représentant le nombre ou la taille de l'objet en entrée. Il est nécessaire de préciser quelles sont les opérations comptabilisées dans l'évaluation du coût. Ce coût peut généralement s'exprimer par une relation de récurrence et il n'est pas souvent facile à obtenir de manière exacte.

Prenons l'exemple de la fonction `factorielle`. Nous notons c_n le coût en fonction de n et nous comptons les tests et les opérations mathématiques.

Pour $n = 0$, nous avons un test, soit $c_0 = 1$.

Pour $n > 0$, nous avons un test, une multiplication et l'appel de la fonction effectué avec le paramètre $n - 1$, soit $c_n = 2 + c_{n-1}$ (ou $c_n = 3 + c_{n-1}$ si nous comptons aussi l'appel).

Les nombres c_n pour n entier naturel constituent ce qu'on appelle en mathématiques une suite arithmétique : 1, 3, 5, ...

On montre que si $u_n = u_{n-1} + r$, alors $u_n = r \times n + u_0$. Nous obtenons donc $c_n = 2n + 1$.

Le coût de la fonction factorielle est linéaire en n.

Plus précisément, pour tout n, nous avons n appels récursifs, n multiplications et $n + 1$ tests.

De manière générale, si le programme a un coût constant c_0 lorsque la condition d'arrêt est satisfaite et c_n est de la forme $c_{n-1} + k$ où k est le coût constant des opérations effectuées en dehors de l'appel récursif, alors on obtient pour tout n : $c_n = kn + c_0$.

Le coût est donc linéaire en fonction de n.

Considérons un exemple plus complexe, celui de la suite de Fibonacci. Une fonction `fibo` est définie de manière récursive :

```
def fibo(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibo(n-1) + fibo(n-2)
```

Déterminer le coût ici est un peu plus difficile. Nous allons calculer séparément le nombre d'additions, le nombre de tests et le nombre d'appels récursifs.

Si nous notons a_n le nombre d'additions pour obtenir f_n, alors $a_n = a_{n-1} + 1 + a_{n-2}$.

Nous disposons d'une méthode pour déterminer l'expression de a_n en fonction de n.

Commençons par ajouter 1 des deux côtés de l'égalité : $a_n + 1 = a_{n-1} + 1 + a_{n-2} + 1$.

Posons alors $b_n = a_n + 1$.
Nous obtenons : $b_n = b_{n-1} + b_{n-2}$, avec $b_0 = a_0 + 1 = 1$ et $b_1 = a_1 + 1 = 1$.
Nous constatons alors que $b_n = f_{n+1}$ et donc $a_n = f_{n+1} - 1$.
Par exemple, pour calculer f_5 le nombre d'additions à effectuer est $f_6 - 1 = 7$.

Pour une addition, nous avons deux appels récursifs. Le nombre d'appels récursifs pour obtenir f_n est donc $2f_{n+1} - 2$. Par exemple pour calculer f_5, nous appelons la fonction `fibo` qui procède à 14 appels récursifs. Le nombre de tests est égal au nombre d'appels de la fonction `fibo`, soit $2f_{n+1} - 1$. Nous comptons un test pour l'appel initial et un test pour chaque appel récursif. Pour calculer f_5, nous procédons donc à 15 tests.

En conclusion, le coût du calcul de f_n en terme de nombre d'additions, de tests et d'appels récursif est $5f_{n+1} - 4$.

On démontre que le nombre f_n est égal à l'entier le plus proche de $\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$.

Nous avons donc ici un coût exponentiel.

Lorsqu'un calcul exact n'est pas possible, nous pouvons envisager d'établir un encadrement du coût. Par exemple, si nous observons la représentation du calcul de f_5, en notant les différents niveaux de 0 à 4 du haut vers le bas, nous pouvons conjecturer pour le nombre d'appels de la fonction noté c_5 l'encadrement suivant : $2^0 + 2^1 + 2^2 \leq c_5 \leq 2^0 + 2^1 + 2^2 + 2^3 + 2^4$, soit $2^3 - 1 \leq c_5 \leq 2^5 - 1$.
Et de manière générale : $2^{\text{Ent}(n/2)+1} - 1 \leq c_n \leq 2^n - 1$, où Ent est la partie entière.

Lien avec les piles

Il existe deux types de structures linéaires courantes en informatique, les files et les piles. Ces structures sont présentées plus tard. Il suffit ici pour parler de la structure de file de penser à une file d'attente. Dans une boulangerie, la première personne dans la file d'attente commande son pain et ressort : premier entré, premier sorti !

Pour une pile, on peut penser à des livres que l'on a déposés un par un sur une table. Ils forment une pile. On peut prendre un livre sur la pile, (nous dirons « dépiler »), et le déposer ailleurs, par exemple sur une autre pile, (nous dirons « empiler »). Les livres ne sont manipulés qu'un par un : dernier entré, premier sorti.

Reprenons une définition récursive de la fonction `factorielle` :

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

Un produit ne peut être effectué que lorsque la valeur d'une factorielle a été renvoyée.
Voyons comment est utilisée une pile pour exécuter ces calculs. Nous empilons les instructions en attente.

— `n` vaut 4 : empilement de `return n * factorielle(n-1)` ;
— `n` vaut 3 : empilement de `return n * factorielle(n-1)` ;
— `n` vaut 2 : empilement de `return n * factorielle(n-1)` ;
— `n` vaut 1 : empilement de `return n * factorielle(n-1)`.

Ensuite `n` vaut 0 donc renvoi de `factorielle(0)` : `return 1`.
Le dépilement et les produits s'exécutent alors.

— n vaut 1 : `return 1;`
— n vaut 2 : `return 2;`
— n vaut 3 : `return 6;`
— n vaut 4 : `return 24.`

Les produits sont effectués ainsi : $4 \times (3 \times (2 \times (1 \times 1)))$.

En notation polonaise inverse, l'expression s'écrit : 4 3 2 1 1 $\times$ $\times$ $\times$ $\times$. La lecture se fait de gauche à droite. Cela revient à empiler tous les opérandes. Puis à chaque signe d'opération, on dépile deux opérandes, on effectue l'opération et on empile le résultat. À la fin, on dépile le résultat.

Considérons maintenant la fonction `fact_term` définie ci-dessous :

```
def fact_term(n, acc):
    if n > 0:
        return fact_term(n-1, acc*n)
    else:
        return acc
```

La suite des appels pour calculer $4!$ est :
`fact_term(4,1)`, puis `fact_term(3, 4)`, puis `fact_term(2, 12)`, puis `fact_term(1, 24)`, et enfin `fact_term(0, 24)`.

Cette fois, les produits au niveau de l'accumulateur sont effectués de la façon suivante :
$(((1 \times 4) \times 3) \times 2) \times 1$.

En notation polonaise inverse, l'expression à calculer s'écrit : 1 4 $\times$ 3 $\times$ 2 $\times$ 1 $\times$.

En Python le nombre d'appels récursifs en attente est limité. Par défaut, cette limite est de l'ordre du millier d'appels. S'il y a un dépassement, un message d'erreur est affiché, et le programme ne termine pas : "*RuntimeError : maximum recursion depth exceeded in comparison*".

Nous pouvons modifier le nombre maximal d'appels récursifs avec les instructions :
`import sys`
`sys.setrecursionlimit(5000)` (par exemple)
Attention : ce nombre maximal doit rester raisonnable !