Informatique PCSI
Prérequis TP 5 : algorithmes récursifs

Algorithmes dichotomiques

Recherche dichotomique

Le programme qui suit est une traduction en récursif du programme itératif vu au TP 4. Son coût est du même ordre que celui de l'algorithme itératif. La terminaison se démontre comme pour l'algorithme itératif en étudiant les valeurs successives des expressions `d-g`, `d` et `g` étant ici des paramètres de la fonction récursive. La liste passée en paramètre est supposée non vide.

```
def dicho_rec(liste, x, g, d):
    if g == d:
        return liste[g] == x
    m = (g + d) // 2
    if x == liste[m]:
        return True
    elif x < liste[m]:
        if g == m:
            return False
        else:
            return dicho_rec(liste, x, g, m-1)
    else:
        return dicho_rec(liste, x, m+1, d)
```

On utilise la fonction avec un premier appel `dicho_rec(lst, x, 0, len(lst)-1)`. À chaque appel récursif, l'un des deux paramètres `g` ou `d` est modifié.

Exemples classiques

Premier exemple

L'un des exemples les plus classiques est une fonction mathématique, la fonction factorielle. On dit « factorielle n » et on note avec un point d'exclamation $n!$.

Cette fonction est définie « par récurrence » sur les entiers naturels : $0! = 1$ et pour tout entier naturel n non nul, $n! = n \times (n-1)!$

Autrement dit, pour tout n non nul, $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$.

Dans l'écriture $0! = 1$ et $n! = n \times (n-1)!$ pour n non nul, nous voyons les deux aspects d'une fonction récursive : un cas simple qui correspond à la condition d'arrêt et un cas complexe, celui de l'appel récursif pour obtenir la valeur de $(n-1)!$ afin de calculer $n!$.

L'écriture d'une fonction récursive consiste à traduire la définition mathématique.

```
def factorielle(n):
    """ n est de type int positif
        renvoie n!"""
    if n > 0:
        return n * factorielle(n-1)
    else:
        return 1
```

La condition `n > 0` au début de l'exécution de la fonction sert à déterminer s'il faut procéder à un appel récursif ou pas. Dès que la valeur de `n` est négative ou nulle, la valeur 1 est renvoyée. À cet effet, les valeurs passées en paramètres lors des appels récursifs constituent une suite décroissante qui en `n` étapes prend une valeur nulle si `n` est un entier positif.

Voici un exemple en Python d'une fonction qui calcule $n!$ de manière itérative.

```
def factorielle2(n):
    f = 1
    for i in range(2, n+1):
        f = f * i
    return f
```

Il est important de savoir passer d'une fonction récursive à une fonction itérative utilisant une boucle `while` ou une boucle `for`, et réciproquement.

Dans la fonction récursive, la dernière instruction est une multiplication par `n` qui ne peut être effectuée qu'après avoir obtenu la valeur renvoyée par l'appel récursif. Les différentes multiplications sont donc mises en attente.

La fonction peut être modifiée en utilisant un paramètre supplémentaire, un accumulateur, par lequel des informations sont passées dans les appels récursifs. On munit ainsi la fonction d'un peu de mémoire. C'est l'accumulateur qui est renvoyé lorsque la condition d'arrêt des appels récursifs est satisfaite. Il doit donc contenir le résultat.

```
def factorielle3(n, acc):
    if n > 0:
        return factorielle4(n-1, acc*n)
    else
        return acc
```

Cette fonction renvoie la valeur de `acc` $\times n!$. Donc `factorielle3(n, 1)` a pour valeur $n!$. Elle peut s'interpréter plus facilement comme une fonction itérative avec une boucle `while`. Il n'y a aucune opération en attente hormis les appels récursifs.

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 constituant une suite qui converge en un nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt. La correction se démontre à l'aide d'un raisonnement « par récurrence ». On vérifie qu'une propriété est vraie pour une valeur entière initiale puis on prouve que la propriété est héréditaire. Une propriété est héréditaire si la véracité pour un entier n quelconque entraîne la véracité pour l'entier suivant $n+1$.

Prouvons que `factorielle3(n, acc)` vaut `acc` $\times n!$ pour tout entier n.

- Initialisation
  Pour $n = 0$, `factorielle3(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 `factorielle3(n, acc)` vaut `acc` $\times n!$.
  Alors `factorielle3(n+1, acc)` vaut `factorielle3(n, acc*(n+1))`. D'après l'hypothèse, cette valeur est `acc` $\times(n+1) \times n!$. Donc `factorielle3(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.

Deuxième exemple

Dans les exemples précédents, le corps de la fonction récursive ne contient qu'un seul appel récursif. Mais il y a des cas où le corps de la fonction contient deux appels récursifs ou plus.

Considérons la suite de Fibonacci. Il s'agit d'une suite d'entiers naturels définie par récurrence.

Les deux premiers termes sont 0 et 1, puis un terme est la somme des deux termes précédents. On obtient ainsi la suite de nombres 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

La définition mathématique est : $f_0 = 0$, $f_1 = 1$ et $f_n = f_{n-1} + f_{n-2}$ pour $n > 1$.

Le n-ième terme peut se calculer avec $n-2$ additions. On peut donc écrire une fonction, utilisant une boucle `while` ou une boucle `for`, qui prend en paramètre un entier `n` et renvoie le terme f_n. Exemple avec une boucle for :

```
def fibo1(n):
    u, v = 0, 1
    for i in range(n):
        u, v = v, u + v
    return u
```

Pour écrire une version récursive, il suffit de « traduire » la formule de calcul.

```
def fibo2(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibo2(n-1) + fibo2(n-2)
```

Cette version récursive est clairement moins efficace que la version itérative. En effet, le nombre d'appels récursifs est « presque multiplié par deux » chaque fois qu'on augmente `n` d'une unité. Il suffit de constater que $2^{30} \simeq 10^9$, pour conclure qu'avec des valeurs supérieures à 30, le calcul devient lent, et sûrement trop long pour des valeurs supérieures à 40.

Il est intéressant d'effectuer des tests dans l'interpréteur Python avec différentes valeurs à partir de 35. On peut arrêter l'exécution avec « ctrl+C » si l'attente est trop longue !

Validité de l'algorithme

Prouvons que la valeur de `fibo2(n)` est bien celle du nombre de Fibonacci f_n pour tout $n \in \mathbb{N}$.

- Initialisation
  Pour $n = 0$, `fibo2(0)` vaut 0, et pour $n = 1$, `fibo2(1)` vaut 1. La propriété est donc vraie pour $n = 0$ et pour $n = 1$.
- Hérédité
  Supposons que pour un entier k quelconque `fibo2(k)` vaut f_k et `fibo2(k+1)` vaut f_{k+1}, alors : `fibo2(k+2)` vaut `fibo2(k+1)` + `fibo2(k)`, soit $f_{k+1} + f_k$ égal à f_{k+2}. La propriété est donc vraie pour $k + 2$.

Nous pouvons conclure que la propriété est vraie pour tout entier naturel n.

On peut transformer cette fonction récursive comme cela a été fait pour `factorielle`. Il convient ici d'ajouter deux accumulateurs dans les paramètres. Avec des valeurs par défaut, l'appel initial de la fonction est plus simple. On obtient alors, par exemple, le nombre f_{300} sans souci.

```
def fibo3(n, a=0, b=1):
    if n == 0:
        return a
    else:
        return fibo3(n-1, b, a+b)
```

Coût d'un algorithme récursif

Le coût en temps d'un algorithme récursif est lié au nombre d'appels récursifs en fonction de n représentant le nombre ou la taille de l'objet en entrée. 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 le nombre de tests et d'opérations arithmétiques.

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 si 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, $c_n = kn + c_0$, donc un coût linéaire.

Le coût de la fonction `fibo2` pour le calcul des nombres de Fibonacci est plus complexe.

Les calculs du nombre d'additions, du nombre de tests et du nombre d'appels récursifs sont faits séparément.

Soit a_n le nombre d'additions pour obtenir f_n, alors $a_n = a_{n-1} + 1 + a_{n-2}$. On ajoute 1 des deux côtés de l'égalité : $a_n + 1 = a_{n-1} + 1 + a_{n-2} + 1$.

On pose alors $b_n = a_n + 1$, et on obtient : $b_n = b_{n-1} + b_{n-2}$, 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 `fibo2` qui procède à 14 appels récursifs. Le nombre de tests est égal au nombre d'appels de la fonction `fibo2`, 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 $5f_{n+1} - 4$. Or, on démontre en mathématiques 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 disons que le coût en fonction de n est exponentiel.

Lorsqu'un calcul exact du coût n'est pas possible, il faut envisager d'établir un encadrement.

Lien avec les piles

Il existe deux types de structures linéaires courantes en informatique, les files et les piles. Pour parler de la structure de file, on peut penser à une file d'attente comme dans une boulangerie : premier entré,

premier sorti ! Pour une pile, on peut penser à une pile de livres déposés sur une table. On peut prendre un livre sur la pile et le déposer ailleurs, par exemple sur une autre pile. Les livres ne sont manipulés qu'un par un : dernier entré, premier sorti.

On considère à nouveau la fonction récursive `factorielle`. Pour son exécution, les instructions en attente sont placées dans une pile. Exemple avec `factorielle(3)` :

- 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`.

Les instructions sont alors retirées de la pile et les produits sont effectués.

- n vaut 1 : `return 1;`
- n vaut 2 : `return 2;`
- n vaut 3 : `return 6.`

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

En *notation polonaise inverse*, l'expression s'écrit : $3\ 2\ 1\ 1\ \times\ \times\ \times$ et se lit 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.

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`, puis `sys.setrecursionlimit(5000)` (par exemple). Ce nombre maximal doit rester raisonnable.

Permutations et parties d'un ensemble fini

Génération des permutations d'un ensemble fini

```
def permute(liste, ind, resultats):
    """Les éléments d'indice 0 à ind-1 sont fixés,
    les autres sont permutés,
    resultats contient les différentes permutations"""
    if ind == len(liste)-1: #cas liste vide
        resultats.append(list(liste))
    else:
        for i in range(ind, len(liste)):
            liste[i], liste[ind] = liste[ind], liste[i]
            permute(liste, ind+1, resultats)
            liste[i], liste[ind] = liste[ind], liste[i]

def permutations(L):
    res = []
    permute(L, 0, res)
    return res
```

On peut tester dans l'interpréteur Python avec `ensemble = ['a', 'b', 'c']`.

```
>>> permutations(ensemble)
[['a', 'b', 'c'], ['a', 'c', 'b'], ['b', 'a', 'c'], ['b', 'c', 'a'],
['c', 'b', 'a'], ['c', 'a', 'b']]
```

Génération des parties d'un ensemble fini

On distingue un élément, par exemple le dernier. Une partie contient ou pas cet élément.

```
def parties(liste, resultats):
    """énumère les parties de liste"""
    n = len(liste)
    if n == 0:
        resultats.append([])
    else:
        sousliste = liste[0:n-1]
        parties(sousliste, resultats) #sans le dernier élément
        for k in range(len(resultats)): #avec le dernier élément
            partie = resultats[k] + [liste[n-1]]
            resultats.append(partie)

def ens_parties(ensemble):
    rep = []
    parties(ensemble, rep)
    return rep
```

On teste la fonction avec `ensemble = [1, 2, 3, 4]` :

```
>>> ens_parties(ensemble)
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3], [4], [1, 4],
[2, 4], [1, 2, 4], [3, 4], [1, 3, 4], [2, 3, 4], [1, 2, 3, 4]]
```