TD n°5

Récursivité et tris

1 Récursivité simple

Une fonction (ou une procédure) *récursive* est une fonction qui s'appelle elle-même. Ainsi, la fonction factorielle peut être définie de manière plus concise à l'aide d'une fonction récursive :

```
class Factorielle {

    static int factorielleImperative(int n) {
        int res = 1;
        for (int i = 1; i <= n ; i++)
          res = res * i;
        return res;
    }

    static int factorielleRecursive(int n) {
        if (n == 0)
          return 1;
        else
          return n * factorielleRecursive(n-1);
    }

    static int factorielleRecursiveTerminale(int n, int a) {
        if (n <= 1)
          return a;
        else
          return factorielleRecursiveTerminale(n-1, n*a);
    }

    public static void main (String[] args) {
        System.out.println(factorielleImperative(10) + " "
                           + factorielleRecursive(10) + " "
                           + factorielleRecursiveTerminale(10, 1));
    }
}
```

Une fonction récursive `f` est *terminale* lorsque tout appel récursif est de la forme `return f(...);`. Autrement dit, la valeur retournée est directement la valeur obtenue par un appel récursif, sans qu'il n'y ait aucune opération sur cette valeur, c'est le cas de `factorielleRecursiveTerminale`.

Ainsi, dans le cas d'une fonction récursive terminale, le dépilement des valeurs de retour est direct, ceci aboutit à une version plus optimisée de la fonction.

Il faut faire attention à ce que la fonction ne boucle pas sur la même valeur, auquel cas le programme ne s'arrêterait jamais, comme dans cet exemple :

```
class Stupide {
    static int boucle(int n) {
        return boucle(n);
    }

    public static void main (String[] args) {
        int p;
        p = boucle(0);
        System.out.println("ce message n'arrivera jamais");
    }
}
```

Pour éviter les appels infinis :
– l'appel récursif doit toujours être fait avec un paramètre de valeur différente que l'appel initial, dans la plupart des cas ce paramètre est plus petit.
– il doit toujours y avoir un cas d'arrêt, i.e. sans appel récursif.
Un algorithme récursif est plus lent qu'un algorithme itératif car il y a la gestion des appels de fonctions (empilement et dépilement du contexte).

Exercice 1 Que fait le programme suivant sur des entrées positives ? Sur des entrées négatives ? Corrigez-le pour qu'il ait un comportement cohérent pour toutes les entrées. Dessiner l'arbre des appels pour ce programme si on rentre la valeur 4. Comment progresse la pile des appels récursifs ?

```
import fr.jussieu.script.Deug;

class Compteur {

    static void f(int n) {
        System.out.print(n + " ");
        if (n!=0)
          f(n-1);
        System.out.print(n + " ");
    }

    public static void main (String[] args) {
        System.out.println("entrer un entier");
        int p = Deug.readInt();
        f(p);
    }
}
```

Exercice 2 1. Écrire une fonction récursive `String repete(int n, String s)` renvoyant la chaîne de caractères s répétée n fois : `repete(3, "bla")` donne `"blablabla"`.

2. Écrire une fonction récursive `void pyramide(int n, String s)` qui écrit 1 fois la chaîne `s`sur la première ligne, 2 fois la chaîne `s`sur la deuxième ligne, et ainsi de suite jusque la `n`ième ligne. Ainsi `pyramide(5, "bla");` donnera
```
bla
blabla
blablabla
blablablabla
blablablablabla
```
3. Quand on lance `pyramide(n, s)`, combien y a-t-il d'appels récursifs à `pyramide`, combien y a-t-il d'appels récursifs à `repete` ? Pour vous aider à répondre dessiner l'arbre des appels pour `n` = 3.

Exercice 3 1. Écrire une fonction récursive `double puissance(double x, int p)` qui calcule la puissance x^p. Combien fait-on d'appels récursifs ?

2. En utilisant le fait que $x^{2q} = (x^2)^q$ et $x^{2q+1} = x.(x^2)^q$, proposer une fonction récursive `double puissanceRapide(double x, int p)` qui calcule la puissance x^p. Combien fait-on maintenant d'appels récursifs pour calculer x^{2^r} ? Quelle est la meilleure méthode ?

Exercice 4 1. Écrire une fonction récursive `int fibonacci(int n)` qui calcule la valeur F_n de la suite de Fibonacci en n :

$$F_0 = 0, \qquad F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2}, \quad \text{pour } n \geq 2$$

2. Combien fait-on d'appels récursifs pour calculer F_5 ? Combien de fois a-t-on calculé F_4, F_3 et F_2 ?
3. Pour résoudre ce problème, on définit $G_n = (F_n, F_{n+1})$. Que vaut G_0 ? Comment calculer G_{n+1} en fonction de G_n ?
4. Proposer une fonction `Couple fibonacciRapide(int n)` qui renvoie la valeur G_n (on ecrira une classe `Couple` avec deux attributs publics de type entier).
5. (Pour aller plus loin) En utilisant les mêmes idées que dans la question précédente, proposer une méthode pour calculer le $n$ième terme d'une suite récurrente ϕ_n définie par ses p premiers termes $\phi_0, \ldots, \phi_{p-1}$ et la formule de récurrence

$$\phi_{n+p} = a_{p-1}\phi_{n+p-1} + a_{p-2}\phi_{n+p-2} + \cdots + a_1\phi_{n+1} + a_0\phi_n.$$

Exercice 5 La fonction d'Ackermann croît extrêmement rapidement ; Ack(4,2) a déjà 19829 chiffres, soit bien plus que le nombre d'atomes dans l'univers actuel. Elle s'écrit ainsi :

$$Ack(0, p) = p + 1$$
$$Ack(n, 0) = Ack(n - 1, 1)$$
$$Ack(n, p) = Ack(n - 1, Ack(n, p - 1))$$

Cette fonction utilise-t-elle des appels infinis ? Écrivez un programme qui calcule $Ack(n, p)$. Dessiner l'arbre des appels pour `Ack(2,2)`.

2 Récursivité croisée

On parle de récursivité *croisée* lorsque deux fonctions s'appellent l'une l'autre récursivement.

Exercice 6 Les fonctions suivantes sont censées donner la parité d'un nombre entier : cela sera-t-il le cas pour toutes les valeurs entières positives ?

```
import fr.jussieu.script.Deug;

class PairImpair {

    static boolean pair (int n) {
        if (n == 0)
            return true;
        else
            return impair(n-1);
    }

    static boolean impair (int n) {
        if (n == 1)
            return true;
        else
            return pair(n-1);
    }

    public static void main (String[] args) {
        int p = Deug.readInt();
        System.out.println("pair ? "+ pair(p) + " impair ? " + impair(p));
    }
}
```

Morale de l'histoire : "Dans les recursions croisées, il vaut mieux que le cas d'arrêt soit le même pour toutes les fonctions".