TD n°4

Récursivité et tris

1 Recursivite 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 :

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

class Fact{

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

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

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

    public static void main (String[] args){
        int p;
        p = Deug.readInt();
        Deug.println(fact_imperative(p) + "   " + fact_recursive(p) + " " +
                     fact_recursive_terminale(p,1));
    }
}
```

Une fonction récursive f est dite 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 la fonction `fact_recursive_terminale`. 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 :

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

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



    public static void main (String[] args){
        int p;
        p = boucle(0);
        Deug.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 entrees positives ? Sur des entrees negatives ? Corrigez-le pour qu'il ait un comportement coherent pour toutes les entrees. Dessiner l'arbre des appels pour ce programme si on rentre la valeur 4. Comment progresse la pile des appels recursifs ?*

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

class Compteur{

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

```
    public static void main (String[] args) {
        int p = Deug.readInt();
        f(p);
    }
}
```

Exercice 2 *1. Ecrire une fonction recursive* `String repete( int n, String s)` *renvoyant la chaîne de caracteres* s *repetee* n *fois :* `repete(3, "bla")` *donne* `"blablabla"`.

2. *Ecrire une fonction* `void pyramide( int n, String s)` *qui ecrit sur la premiere ligne, 1 fois la chaîne* `s`, *sur la deuxieme, 2 fois la chaîne* `s`, *et ainsi de suite jusque la derniere ligne, ou il y aura* `n` *fois la chaîne* `s`. *Ainsi* `pyramide(5, "bla");` *donnera*

```
bla
blabla
blablabla
blablablabla
blablablablabla
```

3. *Quand on lance* `pyramide(n, s)`, *combien y a-t-il d'appels a* `pyramide`, *combien y a-t-il d'appels a* `repete` *? Pour vous aider a repondre dessiner l'arbre des appels pour* `n = 3`.

Exercice 3 *Ecrire une fonction qui calcule la puissance* n^p, n *et* p *etant deux entiers.*

Exercice 4 *Ecrire une fonction qui calcule la valeur* F_n *de la suite de Fibonacci en* n *:*

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

Exercice 5 *La fonction d'Ackermann croît extrêmement rapidement ; Ack(4,2) a deja 19829 chiffres, soit bien plus que le nombre d'atomes dans l'univers actuel. Elle s'ecrit 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 in nis ? Ecrivez un programme qui calcule $Ack(n, p)$. *Dessiner l'arbre des appels pour* `Ack(2,2)`.

2 Recursivite croisee

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

Exercice 6 *Les fonctions suivantes sont censees donner la parite d'un nombre entier : cela sera-t-il le cas pour toutes les valeurs entieres 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();
        Deug.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

– On modifie compte de façon à avoir dans `compte[i]` le nombre d'éléments de s ayant une valeur inférieur ou égale à i.

– On utilise le contenu de compte pour construire la suite triée dans un deuxième tableau auxiliare t en utilsant l'instruction suivante :

pour tout i allant de n à 1 par pas de -1 faire :

```
t[compte[ s[i] ] - 1] = s[i];
compte[ s[i] ] = compte[ s[i] ] - 1;
```

finpour

– On recopie la suite triée dans le tableau s.

1. Dérouler l'algorithme de tri par dénombrement sur le tableau :
$$A = [7, 1, 3, 1, 2, 4, 5, 7, 2, 4, 3]$$
2. Écrire une méthode pour calculer la fréquence des éléments d'une suite de n entiers positifs ayant une valeur inférieure à k.
3. Écrire une méthode qui effectue le tri par dénombrement d'une suite de n entiers positifs ayant valeur inférieure à k.