Plan du cours

1/ Rappels et compléments Java.

2/ Tableaux, boucles et invariants.

3/ Notions élémentaires de complexité.

4/ Récursion. **ICI**

5/ Structures de données et introduction aux types abstraits de données.

6/ Quelques compléments Java.

4/ Récursion.

1. Introduction & exemples.
2. Algorithmes récursifs.
3. Les différents types de récursivité.
4. Récursif ou itératif.
5. Complexités de nos exemples.

Programme récursif

Un **programme** est dit **`récursif`** s'il fait appel à lui-même (du latin *recurrere* $\sim$ courir en arrière).

Conséquence.

Pour pouvoir s'appeler, un programme récursif est donc forcément une **`fonction`** !

Un exemple de tous les jours: factoriel!

$$n! = n \times (n-1) \times \cdots \times 2 \times 1\,.$$

```
public static long factoriel (int n) {
        if(n<2) return 1;
        return n* factoriel (n-1);
}
```

Maximum/minimum d'un tableau

On veut calculer le maximum (resp. minimum) d'un tableau.

Propriétés

- $\max_{i\in[1;r]} t\,[i] = \max(\max_{i\in[1;m]} t\,[i]\,, \max_{i\in[m+1;r]} t\,[i])$
- $\max_{i\in[\ell;\ell]} t\,[i] = t\,[\ell]$.

En utilisant ces propriétés, on en déduit le programme récursif suivant :

Recherche récursive du max

```
// recherche du max par diviser pour régner
static int  max (int[] t, int ℓ, int r) {
        if(ℓ ==r) return t[ℓ];
        int m=(ℓ+r)/2;
        int a= max (t,ℓ,m);
        int b= max (t,m+1,r);
        return (a>b)?a:b;
}
```

L'exécution d'un algorithme récursif

Calcul de 4!:
factoriel(4) $\Rightarrow$ 4.factoriel(3) $\Rightarrow$ 4.3.factoriel(2) $\Rightarrow$ 4.3.2.factoriel(1) $\Rightarrow$ 4.3.2.1 = 24

Le principe de l'exécution

Le principe de l'exécution est simple : l'algorithme met de côté (**en fait dans une PILE**) ce qu'il ne sait pas faire quitte à les reprendre après!

Le principe de la programmation

Par conséquent, on doit dire à l'algorithme comment faire dans les cas les plus simples $\Rightarrow$ LES CONDITIONS D'ARRÊT ! Dans les cas complexes, l'algorithme va faire appel à lui-même en SIMPLIFIANT ces cas jusqu'à tomber sur l'une des conditions d'arrêt.

Exemples de programmes FAUX!

Si ces principes ne sont pas appliqués, l'algorithme risque soit de ne pas produire de résultat, soit de tourner à l'infini.

Pas de conditions d'arrêt $\Rightarrow$ CALCUL INFINI

```
public static long factoriel (int n) {
        return n* factoriel (n-1);
}
```

Pas de simplification du cas à traiter

```
 /* Dans le cas suivant,
** on ne dirige pas le programme vers les conditions
** d'arrêt */
public static long factoriel (int n) {
        if n<2 return(1);
        return factoriel (n+1)/(n+1);
}
```

Algorithmes récursifs : les règles de conception (1/2)

Première règle.

Tout algorithme récursif doit distinguer plusieurs cas, dont l'un au moins ne doit pas comporter d'appel récursif. Souvent ce sont les cas les plus simples.

Sans cela, on risque de tourner en rond et de faire des exécutions qui ne se finissent pas.

Conditions d'arrêt.

Ces cas non récursifs sont appelés **`cas de base`** et les conditions que doivent satisfaire les données dans ces cas de base sont appelées **`conditions d'arrêt`** ou de **`terminaison`** .

Algorithmes récursifs : les règles de conception (2/2)

Seconde règle.

On doit conduire le programme `vers` les cas de base : **tout appel récursif doit se faire avec des données plus proches des conditions de terminaison!**

Cette règle utilise le théorème suivant :

Théorème.

Il n'existe pas de suite infinie strictement décroissante d'entiers positifs ou nuls.

Ainsi, l'arrêt de l'exécution d'un programme récursif est **garanti** quand les deux règles sont appliquées!

Les différents types de récursivité

Dans ce cours, nous allons voir 3 types:

1. récursivité **multiple** ,
2. récursivité **croisée** ou **mutuelle** ,
3. récursivité **imbriquée** .

Définition récursivité multiple

Un programme récursif est **multiple** si l'un des cas qu'il traite se résout avec plusieurs appels récursifs.

Définition récursivité croisée/mutuelle

Deux algorithmes sont **mutuellement récursifs** si l'un fait appel à l'autre et vice-versa.

Définition récursivité imbriquée

Une fonction contient une **récursivité imbriquée** s'il contient comme paramètre un appel à lui-même

La fonction d'Ackermann

Sa récursivité est à la fois **binaire** et **imbriquée**. Elle a le mérite d'être une fonction qui croit très très vite!

Le code d'Ackermann

```
static int ack(int m, int n) {
if (m == 0) return n + 1;
else
        if (n == 0) return ack (m - 1, 1);
        else return ack (m - 1, ack (m, n - 1));
}
```

Une croissance phénoménale

On a $ack(0,n) = n+1$, $ack(1,n) = n+2$ mais $ack(5,1)$ est DÉJÀ TRÈS GRAND devant 2^{65536} qui est très grand par rapport à 10^{80}!

Remarque

Java passe les arguments des fonctions par valeur. Il calcule donc toujours l'argument avant de trouver le résultat d'une fonction.

Dans le cas de la fonction de Morris, on a le code suivant.

Le code

```
static int g(int m, int n) {
        if (m == 0) return 1;
        else return g(m - 1, g(m, n));
}
```

On peut se poser la question: `que vaut g(1,0)` ?

$$\mathbf{g(1,0) = g(0, g(1,0))}\,.$$

La suite de Syracuse

Le code

```
static int s(int n) {
        if (n==1) return 1;
        else {
                if (n % 2 == 0) return s(n/2);
                else return s(3*n+1);
        }
}
```

Problème ouvert

Sur plein d'exemples, on trouve `s(n) = 1` mais personne ne sait le prouver mathématiquement!

Dérécursion

Théoriquement on peut toujours **dérécursiver** un algorithme récursif, c'est-à-dire le transformer en **algorithme itératif**. En pratique, ce n'est pas toujours facile!

Factoriel itératif

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