Université de Nice - Sophia Antipolis
Faculté des Sciences

DEUG MIAS MP1

Programmation 2000-01

2. FONCTIONS RECURSIVES

A. Le principe de récurrence

Le *principe de récurrence*, utilisé intuitivement par Grassmann dès 1861 et formalisé par Peano vers 1889, est l'une des méthodes les plus puissantes des mathématiques pour prouver une propriété générale portant sur un entier naturel.

Exercice 2.1 Prouver par récurrence que pour tout n ∈ **N**, on a : $1^2 + 2^2 + 3^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$

• Prenons l'exemple de la factorielle d'un entier naturel n, qui n'est autre que le produit $n! = 1 \times 2 \times 3 \times \ldots \times n$. Le calcul de $n!$ se ramène au calcul de $(n-1)!$ grâce à la relation de récurrence $n! = n \times (n-1)!$ qui fait décroître la quantité n. Cette décroissance de n risquant de faire diverger le calcul dans les entiers négatifs, on la bloque en donnant une définition spéciale pour $n = 0$: la factorielle de 0 vaut 1. En effet, 0! est un produit vide, qui donne l'élément neutre 1 de la multiplication [on peut trouver d'autres raisons…]. D'où deux règles de calcul :

$$\boxed{\begin{array}{ll} 0! = 1 & \\ n! = n \times (n-1)! & si \quad n > 0 \end{array}}$$

Les deux cas seront traités en Java par une conditionnelle :

```
static int fac (int n)              // pour n = 0, calcule n!
{  if (n == 0) return 1;
   else return n * fac(n – 1);      // le 'else' est ici optionnel
}
```

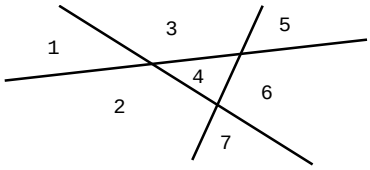
Une telle fonction est dite **récursive** [⇔programmée par récurrence]. Syntaxiquement, cela se voit au fait qu'elle s'utilise elle-même dans sa propre définition, ce qui pour un matheux est d'une grande banalité, n'est-ce pas ? Oui, à condition de bien s'assurer que le calcul converge et que la relation de récurrence générale finira par buter sur le cas de base, ici $n = 0$.

Exercice 2.2 Programmer une fonction récursive `somCarrés(int n)` retournant $1^2 + 2^2 + 3^2 + \ldots + n^2$.

B. Premiers pas en terrain récursif

Comme le mathématicien, le programmeur dispose donc de la récurrence comme outil fondamental, à côté des boucles et de l'itération. Nous reviendrons plus loin sur les rapports qu'elles entretiennent. Dans l'immédiat, nous vous proposons de programmer quelques fonctions récursives. N'essayez pas de comprendre « comment ça marche », contentez-vous de trouver la relation de récurrence, qui relie le cran *n* au cran *n-1*, ainsi que le cas de base, et l'ordinateur fera le reste…

Exercice 2.3 Programmer une fonction récursive `nbRégions(int n)` retournant le nombre de régions du plan, bornées ou pas, délimitées par *n* droites « en position générale » [cela signifie qu'il n'y a pas de couples de droites parallèles ni de triplet de droites concourrantes].



`nbRégions(3) == 7`

Exercice 2.4 a) Programmer une fonction récursive `binomial(int n, int p)` prenant deux entiers `0 = p = n`, et retournant le coefficient binomial C_n^p du triangle de Pascal. Contrairement à ce qui a été fait en Tronc Commun, on ne passera pas par un tableau, et l'on programmera directement la relation de récurrence bien connue $C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$ en faisant très attention au cas de base.

b) Si l'on comptabilise le nombre d'additions, montrez que la *complexité* de ce calcul de C_n^p est exponentielle !

Remarque. *Il est facile de chronométrer un algorithme en Java. La fonction `System.currentTimeMillis()` retourne un nombre de millisecondes à partir d'une date inconnue, sous la forme d'un entier `long` [le type primitif `long` contient les entiers signés sur 64 bits, donc entre -2^{63} et $2^{63}-1$] :*

```
long t0 = System.currentTimeMillis();
int n = binomial(20,10);
long t1 = System.currentTimeMillis();
System.out.println(n + " [time = " + (t1 - t0) + "ms]");
```

Exercice 2.5 Le PGCD [Plus Grand Commun Diviseur] de deux entiers naturels *a* et *b* était déjà calculé par Euclide de la manière suivante :

Le PGCD de a et b n'est autre que le PGCD de b et du reste de la division de a par b.

a) Calculez à la main le PGCD de *a = 8* et *b = 12*. Quand le calcul stoppe-t-il ?

b) Programmer dans la classe `Numerik` une fonction récursive `pgcd(int a, int b)` en Java.

Exercice 2.6 Le **principe de dichotomie** donne lieu à un grand classique dans la résolution des équations. Soit `f` une fonction continue sur `[a,b]` telle que `f(a) < 0` et `f(b) > 0`. Faites un dessin. Alors [théorème des valeurs intermédiaires] il existe au moins un point `c` ∈ `[a,b]` tel que `f(c) = 0`. On se propose de calculer une valeur approchée de `c`, par exemple telle que $|f(c)| \leq e$. Programmez dans `Numerik` une fonction :

```
racineDicho(double a, double b, double epsilon)
```

retournant une telle valeur approchée. La fonction `f` sera définie à part dans la classe `Numerik`.

Application : Trouvez une valeur approchée des constantes e=1.71… , π=3.14... et $\sqrt[3]{2}$ = 1.25… avec cette méthode.

N.B. L'inconvénient de l'exercice précédent tient au fait que l'on ne peut pas dans le `main(…)` demander deux calculs de racines puisque la fonction `f` est décrite « en dur » comme méthode séparée. Il faudrait pouvoir programmer la méthode `racineDicho(f,a,b,e)` pour toute fonction f telle que `f(a) < 0 < f(b)`. Nous verrons dans une feuille ultérieure comment ceci peut être réalisé en Java avec des « classes anonymes ».