Université de Nice - Sophia Antipolis
Faculté des Sciences

DEUG MIAS MP1

Programmation 2000-01

3. RECURSIVITE (SUITE)

A. Procédures récursives

Dans le chapitre 2, nous avons traité des fonctions récursives. Parlons à présent des procédures récursives, qui ne retournent aucun résultat mais ont un effet résiduel, par exemple un affichage. Prenons l'exemple de la procédure `affRenverse(String str)` qui consiste à afficher une chaîne de caractères à l'envers. Pensons récursivement :

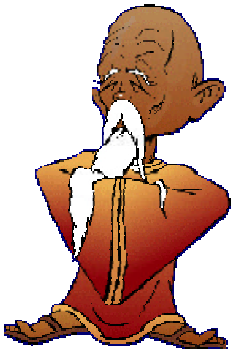
Pour afficher une chaîne non vide à l'envers :
- afficher le dernier caractère
- afficher le reste de la chaîne à l'envers

Exercice 3.1 En n'oubliant pas le cas de base [celui de la chaîne vide], écrivez dans Numerik la méthode `affRenverse(…)` de manière récursive. On rappelle que `str.substring(i,j)` retourne une copie de la sous-chaîne de `str` comprise entre les indices `i` et `j-1` inclus.

• Autre exemple, le **schéma de Hörner** : un algorithme récursif de calcul de la valeur d'un polynôme en un point. Sans faire de grosse classe pour simplifier, supposons que notre polynôme nous est fourni sous la forme d'un tableau de nombres approchés contenant tous les coefficients dans l'ordre des puissances croissantes. Par exemple, le polynôme $5 + 4x - x^2 + 2x^3$ sera représenté par le tableau noté symboliquement `{5,4,-1, 2}` et sa valeur en `x` se calcule récursivement ainsi : $5 + 4x - x^2 + 2x^3 == 5 + x(4 - x + 2x^2)$, autrement dit :

`horner({5,4,-1, 2},x) == 5 + x * horner({4,-1,2},x)`

Exercice 3.2 Programmez la fonction `horner`. Vous aurez sans doute besoin d'une fonction auxilliaire…



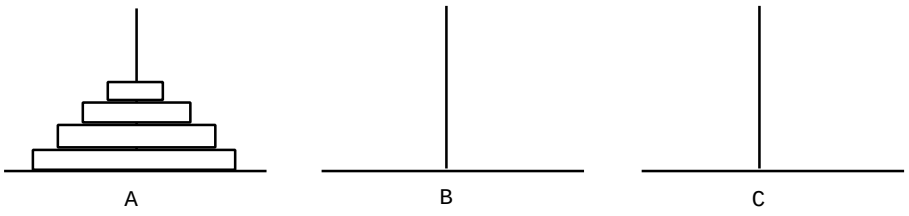
Exercice 3.3 Le jeu des **Tours de Hanoï** est un solitaire qui se joue avec 3 piliers A, B, C. Sur le pilier A se trouvent n disques de largeur décroissante. Il s'agit de transférer tous les disques du pilier A vers le pilier C, en utilisant le pilier B comme intermédiaire, en respectant les règles suivantes :

- on ne déplace qu'un seul disque à la fois.
- on ne peut déposer un disque que sur un disque plus grand.

Programmez une procédure récursive `hanoi(char p1, char p2, char p3, int n)` affichant à l'écran la suite des coups à jouer pour résoudre ce défi.

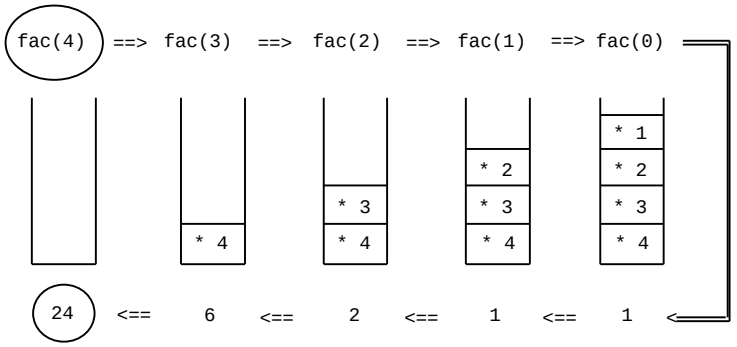
Par exemple, l'appel `hanoi('A', 'B', 'C', 3)` affichera les 7 coups nécessaires, disons sous la forme suivante [*attention, la solution est fausse, rectifiez-la…*] :

```
A->B  A->B  A->C  B->A  B->C  A->B
```



B. Mécanique de la récursivité

Lorsque Java calcule `4!` avec l'algorithme `fac` de la section 2.A, il réduit le calcul de `fac(4)` à celui de `4 * fac(3)`. Or il ne peut pas faire la multiplication par 4 qui *enveloppe* le calcul avant d'être en possession du résultat de `fac(3)`, qui peut prendre du temps. Il va donc plonger dans le calcul de `fac(3)` mais pour ne pas oublier qu'il faudra encore multiplier par `4`, il va mémoriser cette multiplication en la mettant en attente dans une « *pile* ». Il va procéder de la même façon avec `fac(3)` qu'il réduit à `3 * fac(2)`, etc. Lorsqu'il tombe sur le cas de base `fac(0)`, il sait que c'est égal à `1`. Il est sur le point de rendre ce `1` en résultat lorsqu'il s'aperçoit qu'il y a des opérations en attente sur la pile. Il effectue alors ces opérations sur le résultat courant en les « dépilant » au fur et à mesure. Vous notez que les opérations sont effectuées et dépilées en ordre inverse de l'ordre dans lequel elles ont été empilées.



C'est cette structure LIFO [*Last In, First Out*] qui caractérise cette structure de pile si importante en programmation. La *discipline de pile* consiste à ne considérer que le sommet de pile, sans chercher à farfouiller en-dessous. Les **4 opérations de base sur une pile** sont donc :

- empiler un élément
- tester si la pile est vide
- dépiler un élément [erreur si la pile est vide]
- consulter le sommet de pile dans le dépiler [erreur si la pile est vide]

• On voit que l'exécution d'une récurrence en Java s'accompagne dans tous les cas d'une consommation de mémoire, précisément d'*espace de pile*. C'est la contrepartie de la facilité d'écriture des programmes récursifs. Dans certains cas, par exemple une dichotomie, la hauteur de pile n'est que logarithmique et il n'y a pas de raison de se priver de la puissance de la pensée récursive. Pour chercher le nombre d'apparitions d'un élément dans un tableau d'un million d'éléments, il faudrait par contre y renoncer[1].

C. La récursivité terminale

Pour marcher jusqu'à épuisement, deux solutions :

```
static void marcher()
{  do faireUnPasEnAvant();
   while (true);
}
```

```
static void marcher()
{  faireUnPasEnAvant();
marcher();
   }
```

La procédure de gauche est une *itération*, sous la forme d'une boucle infinie. Elle ne terminera jamais. La procédure de droite est son exacte réplique récursive. L'appel récursif à `marcher()` étant en dernière position [il n'a pas d'*enveloppe*], il exprime un mécanisme itératif. Hélas, le compilateur Java qui n'est pas ici très malin, ne s'en aperçoit pas et utilise une pile, qui finira par déborder. La procédure de droite terminera donc sur un épuisement de mémoire[2].

Cette *récursivité terminale* [sans enveloppe] est donc intellectuellement une itération, mais chaque langage de programmation décidera si le mécanisme de calcul dans l'ordinateur est effectivement itératif, i.e. utilisera une pile pour ses besoins propres[3]. En Java, on peut donc écrire un algorithme sur papier de manière récursive terminale, mais il faudra au moment de le coder le transformer en véritable boucle `while`, `do…while` ou `for`, à moins d'être sûr que la hauteur de pile ne sera pas conséquente, par exemple logarithmique comme dans un algorithme par dichotomie…

---

[1] Ou renoncer à manipuler des tableau à un million d'éléments au profit de structures de données accessibles en temps logarithmique, mais ceci est une autre histoire, wait and see…

[2] Si vous voulez vous en persuader, remplacez l'instruction *faireUnPasEnAvant()* par un *System.out.println("En avant !")*.

[3] Vous étudierez en 2ème année un autre langage de programmation, Scheme, qui optimise la récursivité terminale et la traitera comme une véritable boucle *while*.

D. L'heure de la morale

i) De manière générale, il est sain dans toute science de ramener un problème à des sous-problèmes plus simples. Ce que dit la récursivité, c'est qu'il est puissant de ramener un problème à un sous-problème qui est le problème lui-même mais avec des données plus simples.
ii) Et que face à un problème, si on veut l'attaquer de manière récursive, on peut envisager souvent avec profit de le *généraliser*. Cette activité hautement intelligente est hélas le frein qui empêche des avancées significatives dans la démonstration automatique ou dans la programmation automatique en Intelligence Artificielle. L'intelligence humaine réside en grande partie dans sa capacité à généraliser. Il est paradoxalement souvent plus facile de résoudre un problème général qu'un problème particulier, car on se donne du même coup une hypothèse de récurrence plus forte qui permet d'avancer, de prouver ou de construire…
iii) Dans cette optique, le principe de récurrence s'énonce souvent de la manière suivante avec une hypothèse de récurrence renforcée, ce qui est tout bénéfice pour la preuve ou la programmation :

> Pour prouver une propriété `P(n)` portant sur un entier naturel `n` :
> - on commence par la prouver pour `n = 0`
> - on montre que si `P(0)`,…,`P(n-1)` sont vraies, alors on peut en déduire `P(n)`

Exercice 3.4 Programmez récursivement *par dichotomie* la fonction `Numerik.pow(double a, int n)` prenant un réel `a` et un entier `n = 0`, et retournant le nombre a^n. Quelle est sa complexité en nombre de multiplications ?

Exercice 3.5 [sur papier] Quel est l'affichage produit par l'appel de chacune des procédures suivantes avec a=3 ?

```
static void truc (int a)
{  if (a == 0) System.out.println("Fini");
   else {  truc(a-1); System.out.println(a);  }
}
static void machin (int a)
{  if (a == 0) System.out.println("Fini");
   else {  System.out.println(a); machin(a-1);  }
}
static void chose (int a)
{  if (a == 0) System.out.println("Fini");
   else {  System.out.println(a); chose(a-1); System.out.println(a);  }
}
static void gasp (int a)
{  if (a == 0) System.out.println("Fini");
   else {  gasp(a-1); System.out.println(a);  gasp(a-1);  }
}
```

Exercice 3.6 Ecrivez une procédure récursive `sablier(int n)` affichant à l'écran une série de lignes représentant un sablier avec des caractères `'x'`. Par exemple, l'appel `sablier(11)` affichera :

```
xxxxxxxxxxx
 xxxxxxxxx
  xxxxxxx
   xxxxx
    xxx
     x
    xxx
   xxxxx
  xxxxxxx
 xxxxxxxxx
xxxxxxxxxxx
```

Remarque : rien ne vous interdit dans ce genre de problème, et dans tout problème en général d'ailleurs, d'introduire des fonctions ou procédures auxilliaires. Mais ici, aucune boucle !…