Programmation : boucles itératives et conditionnelles

Nicolas Gast - nicolas.gast@ens.fr

7 novembre 2005

1 Introduction

1.1 Structure d'un programme

En Maple, un programme est une fonction. La syntaxe pour écrire un programme prenant n argument en entrée est :

```
mon_programme := proc( arg1, arg2, ... argn)
# insérer ici le code du programme
end;
```

Notons que ni les retours à la ligne ni les espaces ne sont importants en Maple. Néanmoins il est vivement conseiller d'en mettre afin de rendre vos programmes lisibles.

Pour exécuter un programme déjà écrit, il suffit ensuite de taper l'instruction
`mon_programme(arg1, ... argn) ;`

1.2 Boucle `if`

L'instruction `if` permet d'exécuter une commande ou une autre selon une condition. La syntaxe est :

```
if condition
  then commande1;
  else commande2;
fi;
```

Ainsi si la `condition` est vraie, `commande1` sera exécutée, sinon `commande2` le sera.

Petite remarque sur les tests de condition : une condition est souvent de la forme `x = 3` (et non `x := 3`), ou `x < π`,... Mais Maple ne teste que des valeurs numériques. Ainsi si vous voulez savoir si $x > \pi$, il faut préciser `x > evalf(pi)`.

Exercice 1. Utilisation de `if`

Écrire un programme qui résout dans $\mathbb{R}$ les équations du second degré sans utiliser la commande `solve`

1.3 Boucles itérative : `for` et `while`

Comme dans la pluspart des langages de programmation, il existe deux manières d'effectuer des boucles itératives :

– **For -** Pour répéter une commande plusieurs fois avec un compteur qui augmente à chaque étape, la syntaxe est :

```
for i from debut to fin do
  ...
od;

#OU

for i from debut to fin by valeur do
  ...
od;
```

La variable début doit être un nombre entier, la variable fin aussi. Quand on exécute une telle boucle, les instructions écrites dans les `...` seront exécuté alors que la valeur `i` vaut debut puis debut+valeur puis debut+2*valeur... jusqu'à ce que `début+k*valeur > fin`.

– **While -** Il est aussi possible d'exécuter une commande tant qu'une condition est vraie. La syntaxe est :

```
while( condition ) do
  ...
od;
```

Remarque : on peut remplacer une boucle `for` par une boucle `while` en écrivant

```
i := debut;
while(i<fin) do
  ...
  i := debut+valeur;
od;
```

Exercice 2. Utilisation de `for`

Écrire un programme calculant le n$^{\text{ième}}$ terme de la série $u_{n+1} = 2003 - \frac{6002}{u_n} + \frac{4000}{u_n.u_{n+1}}$

1.4 Variables locale, globale

Lorsqu'on écrit un programme, il est souvent utile d'utiliser des variables à l'intérieur du programme. Généralement, ces valeurs ne servent que dans le programme, il faut donc les déclarer par la commande `local`. Si on veut que la valeur de cette variable soit modifiée, il faut la déclarer `global`.

Exemple :

```
> mon_prog := proc()
  local i;
  global j;
  i := 1;
  j := 1;
  end;
          mon_prog := proc() local i; global j; i := 1; j := 1 end proc
> mon_prog(): i; j;
                                        i
                                        1
```

Exercice 3. Utilisation de `global` et `local` : Échanger deux éléments

Écrire une commande `swap(i,j)` qui échange les éléments `i` et `j` du tableau `t`. Attention le tableau `t` n'est pas passé en argument mais est une variable globale qui doit être modifiée par la procedure.

Exemple du résultat que votre procedure doit vous donner :

```
> t := [1,2,3];
                                  t := [1, 2, 3]
> swap(1,2): t;
                                     [2, 1, 3]
```

2 Exercices

Exercice 4. Le Tri bulle

Le "tri bulle" est une méthode simple permettant de trier un tableau. L'idée est de parcourir le tableau et d'échanger deux éléments i et $i+1$ si le premier est plus grand que le second. Après avoir répété l'oppération n fois, la tableau est trié.

a. Montrer qu'au bout d'une itération, le plus grand élément du tableau se trouve en dernière position.

b. En déduire qu'au bout de n itérations, le tableau est entièrement trié.

c. Programmer l'algorithme de tri-bulle

d. Combien votre algorithme fait-il de comparaisons ?

e. Proposer une amélioration (simple) permettant de passer le nombre de comparaisons à $n(n-1)/2$

f. (pour les informaticiens en herbe) A votre avis, peut-on faire moins d'opérations que ça ? Quelle est le nombre minimal d'opérations qu'il faut faire pour trier un tableau ?

Exercice 5. Mélange de cartes

Imaginez que vous vouliez mélanger un jeu de carte par un ordinateur. Une méthode simple est de tirer une carte au hasard et la mettre sur le dessus du paquet puis d'en tirer une parmis les cartes restantes, etc...

L'algorithme correspondant est :

1. Tirer un nombre au hasard a entre 1 et n et échangez l'éléments n avec l'élément a
2. Tirer un nombre au hasard a entre 1 et $n-1$ et échangez l'éléments $n-1$ avec l'élément a
3. ... Continuer en remplaçant $n-1$ par $n-2$ puis $n-3$, ... jusuqu'à 2.

a. Programmer l'algorithme en Maple prenant en entrée un tableau et rendant le même tableau dont les éléments ont été permutés aléatoirement. (note : pour tirer un nombre aléatoire entre `0` et `n-1` inclus, utilisez la commande `rand(n)()`)

b. Quel est la complexité de votre algorithme ? (*ie* le nombre d'oppérations)

c. Montrer que votre algorithme tire bien une permutation aléatoire *uniformement* parmis l'ensemble des permutations de taille n

Remarque : cet algorithme est dit "en place" car le résultat obtenu est situé dans la même zone mémoire que celle passé en entrée.

Exercice 6. Quick sort

`Quick sort` consiste à placer le premier élément d'un tableau d'éléments à trier (appelé pivot) à sa place définitive en permutant tous les éléments de telle sorte que tous ceux qui lui sont inférieurs soient à sa gauche et que tous ceux qui lui sont supérieurs soient à sa droite. Cette opération s'appelle partitionnement. Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soient triés.

```
tri_rapide(tableau t, entier premier, entier dernier)
  début
    si premier < dernier alors
      pivot <- choix_pivot(t,premier,dernier)
      tableau <- partitionner(t,premier,dernier,pivot)
      tri_rapide(t,premier,pivot-1)
      tri_rapide(t,pivot+1,dernier)
    finsi
  fin
```

a. Programmer cet algorithme en Maple. Le choix du pivot est laissé libre, vous pouvez par exemple prendre le premier élément.

b. Quelle est la complexité de cet algorithme ?

c. Déterminer la complexité de cet algorithme dans le pire des cas.

d. Tester cet algorithme sur des tableaux aléatoires de 1000 puis 10000 éléments et comparer les résultats à ceux obtenus par le tri "bulle". Que peut-on en déduire quand à la complexité moyenne ?

Si les algorithmes de tri vous intéressent, vous pourez trouver beaucoup d'exemples sur `wikipedia`. `Quick sort` est l'algorithme le plus utilisé car sa complexité moyenne est excellente mais d'autres algorithmes peuvent avoir une meilleur complexité dans le pire cas.