TD 4 : Récursion

Novembre 2012

Principe de la récursion

On dit qu'une fonction f est récursive lorsqu'elle s'appelle elle-même dans sa définition.

La notion de fonction récursive est en fait assez omniprésente dans le monde de l'informatique et des mathématiques, pensez par exemple à la définition de la multiplication dans l'axiomatique de Peano ou à la factorielle :

$$m \times n = \begin{cases} 0 & \text{si } m = 0 \\ \underbrace{(m-1)}_{m'} \times n + n & \text{si } m = m' + 1 \end{cases} \qquad n! = \begin{cases} 1 & \text{si } n = 0 \\ \underbrace{(n-1)}_{n'}! \times n & \text{si } n = n' + 1 \end{cases}$$

Pour écrire une fonction récursive en maple, il suffit de rappeler la fonction dans sa définition. Par exemple, on peut écrire la multiplication sur les entiers de la manière suivante :

```
multiplication := proc(m,n)
  if m = 0
  then 0;
  else multiplication(m-1, n) + n;
  end if;
end proc;
```

L'appel à `multiplication` à la 5ème ligne fait référence à la fonction que l'on est en train de définir tout comme ce qui est écrit juste au-dessus.

Exercice 1

En vous inspirant de l'exemple de la multiplication, écrivez une fonction factorielle en maple qui suive le schéma donné ci-dessus.

Exercice 2 (*Exponentiation rapide*)

En remarquant que

$$p^q = \begin{cases} 1 & \text{si } q = 0 \\ \left(p^{\frac{q}{2}}\right)^2 & \text{si } q \text{ est pair} \\ \left(p^{\lfloor \frac{q}{2} \rfloor}\right)^2 \times p & \text{si } q \text{ est impair} \end{cases}$$

écrivez un algorithme efficace d'exponentiation. Estimez la complexité de votre algorithme.

Structures de données récursives

Certaines structures de données se prètent particulièrement bien à l'emploi dans une fonction récursive : il s'agit des structures basées sur un schéma d'induction comme les entiers, les listes ou les arbres.

Définition inductive des entiers

Il s'agit de la définition classique dans l'axiomatique de Peano, aussi appelés entiers de Church. Un entier est :

- ▷ Soit la constante `0`
- ▷ Soit le successeur `S(n) = n + 1` de `n`

Schéma d'induction pour les listes

On peut définirvoir les listes comme :

- ▷ Soit une liste vide `[]`
- ▷ Soit un élément `x` ajouté au début d'une liste `l`, c'est à dire `[x, op(l)]`

Cette définition donne automatiquement la structure récursive suivante :

```
generic_list_process := proc(l)
  if l = []
  then
    faire quelque chose dans le cas de base
  else
    dans ce cas l = [x, op(l’)] donc on peut travailler
    avec x=l[1] puis appeler generic_list_process sur l[2..-1]
  end if;
end proc;
```

Exercice 3

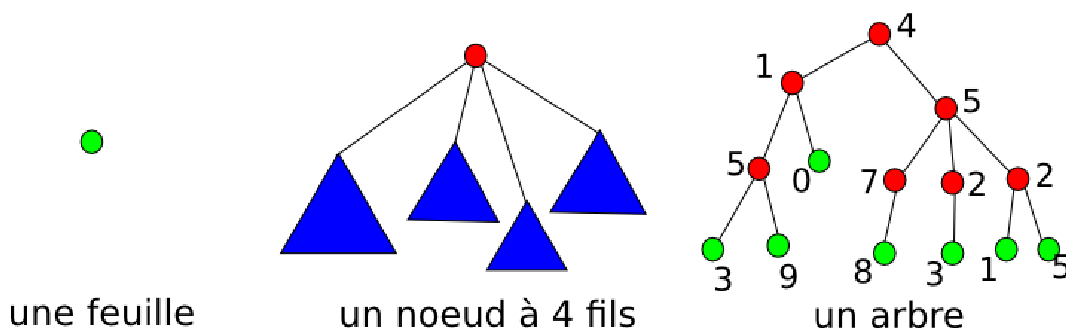
En suivant le schéma précédent écrivez une fonction `len := proc(l)` *qui calcule la longueur d’une liste.*

Les arbres

Un arbre est une structure de données définie récursivement :

▷ Soit c’est une feuille

▷ Soit c’est un nœud avec une liste de fils, chaque fils étant lui même un arbre.



On peut placer des données dans un arbre au niveau des nœuds et/ou des feuilles. Pour représenter un arbre en maple, on peut considérer qu’un nœud est une paire composée d’une donnée et d’une liste d’arbre qui sont ses fils. Les feuilles sont alors des nœuds sans fils, c’est à dire des paires composées d’une donnée et d’une liste vide.

Par exemple l’arbre ci-dessus donnerait :

```
mon_arbre := [4, [
                    [1, [
                           [5, [[3,[]], [9, []]]],
                           [0, []]
                        ]],
                    [5, [
                           [7, [8, []]],
                           [2, [3, []]],
                           [2, [[1, []], [5, []]]]
                        ]]
                 ]];
```

Il est en fait plus simple de définir de *constructeurs* :

```
leaf := x -> [x, []];
node := (x,l) -> [x, l]
```

et d’écricre

```
mon_arbre_bis := node(4, [node(1, [node(5, [leaf(3), leaf(9)]), leaf(0)]),
                          node(5, [node(7, [leaf(8)]), node(2, [leaf(3)]),
                                   node(2, [leaf(1), leaf(5)])])]);
```

Le squelette d'un programme travaillant sur un tel arbre aura trés souvent la forme suivante :

```
tree_process := proc(t)
  local val, fils;
  val := t[1];
  fils := t[2];
  if fils = []
  then
     on est arrivé à une feuille, on la traite comme telle
  else
     on est à un noeud avec une liste d'arbre dans fils,
     on va surement appeler tree_process(fils[i]) pour i entre 1 et nops(fils)
  end if;
end proc;
```

Par exemple, si on veut compter le nombre de feuilles d'un arbre on peut faire un des deux programmes suivant (la version à droite utilise la fonction `add` de maple pour alléger un peu le code) :

```
count_leaf := proc(t)
  local val, fils, compte;
  val := t[1];
  fils := t[2];
  compte := 0;
  if fils = []
  then
     compte := 1;
  else
     for i from 1 to nops(fils) do
        compte := compte +
                       count_leaf(fils[i]);
     end do;
  end if;
end proc;
```

```
count_leaf := proc(t)
  local val, fils;
  val := t[1];
  fils := t[2];
  if fils = []
  then
     1;
  else
     add(count_leaf(fils[i]),
                   i=1..nops(fils));
  end if;
end proc;
```

Exercice 4

Sur le modèle de l'exemple précédent écrivez une fonction qui compte le nombre de nœud et de feuilles dans l'arbre.

Complexité des fonctions récursives

Terminaison

Mais la récursion ne se limite pas à définir des fonctions par induction (ce que l'on a fait ci-dessus). On peut trés facilement fabriquer des fonctions qui ne terminent pas

```
vers_l_infini_et_au_dela := proc()
  vers_l_infini_et_au_dela()
end proc;
```

ou au contraire qui terminent sans que ça paraîsse évident

```
ack := proc(m,n)
  if m = 0
  then n + 1;
  elif n = 0
  then ack(m-1,1);
  else ack(m-1,ack(m,n-1));
  end if;
end proc;
```

Attention cette fonction croît **trés trés** vite par rapport à ses arguments (surtout m) ! Si vous voulez l'essayer, cantonnez vous à des valeurs de m et n petites (≤ 3).

Pour montrer qu'une fonction récursive termine, il est parfois nécessaire d'utiliser des arguments un peu lourd, typiquement il s'agit de donner un ordre $\prec$ sur l'ensemble des arguments tel qu'il n'existe pas de chaines infinies décroissantes (on appelle un tel ordre un bon ordre).

En pratique, la quasi-totalité des fonctions que nous allons manipuler cette année sont assez simple et termineront assez clairement (ou ne terminerons clairement pas en cas d'erreur lors de l'écriture du code).

Exercice 5 (*Expressivité de la récursion*)

1. *Écrivez une fonction récursive* `boucle_for := proc(i,j,f)` *qui prend en argument deux entiers* $i \leq j$ *et une fonction* f : `Int` $\to$ `Int` *et qui simule le programme suivant :*

```
for k from i to j do
   print(f(k));
end do;
```

2. *Écrivez à présent une fonction récursive* `boucle_while := proc(s,b,f)` *imitant la structure de contrôle* `while` *où* s : `State` *est un contexte,* b : `State` $\to$ `Bool` *est une condition sur le contexte et* f : `State` $\to$ `State` *est une fonction manipulant le contexte. Par contexte, on entend ici une structure de donnée "opaque" c'est à dire sur laquelle on ne sait rien (et on ne veut rien savoir d'ailleurs). On veut donc simuler :*

```
while b(s) do
   s := f(s);
end do;
```

3. Bonus : *On vient de montrer que l'on pouvait encoder le* `for` *et le* `while` *avec la récursion. Montrez que réciproquement, toute fonction récursive peut s'écrire à l'aide du* `for` *ou de* `while`*. Le* `while` *seul suffit en fait, il s'agit d'un modèle de calcul universel, ou aussi turing-complet, comme la récursion.*

Calcul pratique

Dans le cas de la programmation impérative avec des boucles, le calcul de la complexité est souvent relativement simple : soit les boucles sont imbriquées et on multiplie leurs complexités respectives, soit elles ne le sont pas et on doit alors les additionner.

Pour les fonctions récursives, le calcul est rarement aussi facile. Le moyen le plus simple pour calculer la complexité d'une fonction récursive f dont l'entrée est de taille n est d'introduire une autre fonction récursive T_f qui donne "le temps de calcul" de f sur une entrée de taille n en fonction de $T_f(p)$ pour $p \leq n$. Pour cela, il faut bien sur que f ne s'appelle que sur des arguments plus petits.

Par exemple, considérons le cas du tri-fusion. On suppose que l'on posséde une fonction `split` qui prend une liste et la coupe en deux ainsi qu'une fonction `merge` qui prend deux listes triées et qui les fusionne en une seule liste triée. Le tri-fusion s'écrit alors :

```
merge_sort := proc(l)
  local l1, l2;
  (l1, l2) := split(l);
  merge(merge_sort(l1), merge_sort(l2));
end proc;
```

On peut estimer la complexité de `split` de l'ordre de $\mathcal{O}(n)$ et celle de `merge` de l'ordre de $\mathcal{O}(n+m)$. On a alors pour la complexité du `merge_sort`

$$T(n) = \underbrace{\mathcal{O}(n)}_{\texttt{split(l)}} + \underbrace{\mathcal{O}(\overbrace{\frac{n}{2}}^{\text{taille de l1,}} + \overbrace{\frac{n}{2}}^{\text{de l2}})}_{\texttt{merge(...)}} + \underbrace{T(\frac{n}{2}) + T(\frac{n}{2})}_{\text{appels récursifs à }\texttt{merge_sort}}$$

c'est à dire $T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$ ce qui se résout en posant

$$\begin{aligned} U(p) &= T(2^p) \\ &= 2T(2^{p-1}) + \mathcal{O}(2^p) = 2U(p-1) + \mathcal{O}(2^p) \\ &= \sum_{i=0}^{p} 2^i \mathcal{O}(2^{p-i}) = \mathcal{O}(p2^p) = \mathcal{O}(n \log n) \end{aligned}$$

Il faut ensuite un argument de croissance de la fonction T pour conclure que $T(n) = \mathcal{O}(n \log n)$.

Bonus : Quelques compléments sur la récursion

Comment ça marche ?

Les fonctions récursives, c'est cool, ça marche super bien... sauf que ça a quelques limitations. Pour comprendre en quoi consiste ces limitations, il faut un peu s'intéresser au mécanisme d'appel de fonctions dans un ordinateur. Lorsque l'on appelle une fonction, on place les arguments de la fonction sur une partie la mémoire appelée la *pile*. A la fin de l'appel, les variables utilisées par la fonction sont dépilées et on revient à la fonction appelante.

Lorsque l'on utilise des fonctions récursives on peut faire face à deux problèmes :

1. La pile est un morceau de mémoire de taille limitée et l'accumulation d'arguments empilés les uns au dessus des autres rique de faire dépasser cette taille limite.
2. L'appel de la fonction nécessite souvent quelques opérations pour “préparer” le terrain pour la fonction appelée. Par rapport aux boucles, l'emploi de fonctions récursives et légèrement plus couteux.

Appel terminal, fonction récursive terminale (tail-rec)

On dit qu'un appel d'une fonction f est en position terminal quand aucun calcul n'est effectué avec le résultat de f avant de le retourner. Un appel terminal correspond donc à un appel où on retourne directement le résultat de l'appel. Une fonction est récursive terminale si tous les appels récursifs sont en position terminale.

Par rapport au paragraphe précédent, ça veut dire qu'avant de faire un appel en position terminale, la fonction appelante peut dépiler son environnement avant de faire l'appel, il n'y a alors plus d'accumulation des différents niveaux d'appel. Dans ce cas là, le surcoût associé à l'emploi de fonctions récursives est trés faible cependant toutes les fonctions ne peuvent pas être écrite de manière récursive terminale (voir la fonction `foldr` par exemple).

Fonctions mutuellement récursives

La notion de récursivité peut s'étendre à plusieurs fonctions : on dit alors que les fonctions sont mutuellement récursives. Plus précisément, deux fonctions f et g sont mutuellement récursives si f appelle g dans sa définition et g appelle f dans la sienne. Un exemple d'emploi de fonctions mutuellement récursive est donné ci-dessous :

```
is_length_even := proc(l)
  if l = []
  then true;
  else not (is_length_odd(l[2..-1]));
  end if;
end proc;

is_length_odd := proc(l)
  if l = []
  then false ;
  else not (is_length_even(l[2..-1])) ;
  end if
end proc;
```

Exercices supplémentaires

Exercice 6 (***opérateurs classiques sur des listes***)

En reprennant le principe de l'exercice 3, écrivez :

1. *une fonction* `map := proc(f,l)` *qui à la fonction f et la liste $l = [x_1, \dots, x_n]$ associe $map(f, l) = [f(x_1), \dots, f(x_n)]$*
2. *une fonction* `foldl := proc(f,acc,l)` *qui à la fonction f, l'accumulateur acc et la liste $l = [x_1, \dots, x_n]$ associe $foldl(f, acc, l) = f(\dots f(f(acc, x_1), x_2), \dots x_n)$*
3. *une fonction* `unfold := proc(seed, p, f)` *tel que*

$$\texttt{unfold(seed, p, f)} = \begin{cases} \texttt{[]} & \text{si } \texttt{p(seed) = false} \\ \texttt{[x, op(unfold(seed', p, f))]} & \text{si } \texttt{p(seed) = true} \text{ et } \texttt{f(seed) = (x, seed')} \end{cases}$$

Exercice 7 (*Arbre de recherche*)

Un arbre de recherche est un arbre binaire (c'est à dire que tous les nœuds ont degré 2 ou encore ont deux fils) contenant des nombres que l'on appelle clé *à chaque nœuds et tel que la clé a un nœud donné est supérieure à toutes les clés de son sous-arbre gauche et inférieure à toutes les clés de son sous-arbre droit.*

Votre mission est d'écrire des fonctions `insert_key`, `search_key` *et* `remove_key` *qui utilisent ce principe et d'en analyser la complexité moyenne (on pourra utiliser le fait que la profondeur d'un arbre de recherche est en moyenne logarithmique en son nombre de nœud si on suppose une distribution uniforme sur l'entrée).*

Bonus : *proposez un mécanisme pour obtenir des arbres de recherche de profondeur logarithmique dans tous les cas.*

Exercice 8 (*Programmation dynamique*)

Donner un algorithme pour trouver la plus longue sous-séquence monotone croissante d'une séquence de n nombres qui soit en temps $\mathcal{O}(n^2)$.