Récursivité

I. Rappels

1. Principes de base

Une fonction *récursive* est une fonction qui peut être amenée à s'appeler elle-même. On signale la présence d'une fonction récursive en OCaml à l'aide du mot clé **rec**.

Un exemple, déjà vu plus tôt, est la fonction factorielle, à la version récursive est la traduction directe de la définition mathématique par récurrence :

```
let rec fact n =
  if n <= 1 then
    n
  else
    n * fact (n - 1)
;;
```

Méthodologie Lors de l'écriture d'une fonction récursive, on distingue en général deux types de branches d'exécution :

- ⋆ les branches **cas de base** qui ne comporte pas d'appel récursif ;
- ⋆ les branches avec **appels récursifs**.

Pour ces dernières, si l'on veut que l'appel de la fonction termine, on s'assurera que l'appel récursif se fait avec des arguments « plus petits » (nous préciserons cela dans le chapitre suivant).
Pour développer l'intuition, on pourra considérer pour l'instant comme arguments des entiers naturels, un appel récursif avec des arguments strictement plus petits ne pouvant se répéter infiniment.

Notons que l'on peut avoir des fonctions mutuellement récursives : chaque fonction appelle l'autre, comme illustré dans l'exemple totalement inintéressant suivant. On parle alors de **récursivité croisée**.

```
let rec est_pair n =
  match n with
  | 0 -> true
  | _ -> not (est_impair (n - 1))
and est_impair n =
  match n with
  | 0 -> false
  | _ -> not (est_pair (n - 1))
;;
```

2. Quelques exemples

On commence par deux exemples, simples, portant sur les entiers.

2.1. Multiplication

On peut commencer par une version récursive très simple (mais peu efficace) de la multiplication entre entiers naturels vue comme addition itérée.

```
let rec multiplication_naive x y =
  if x = 0 then
    0
  else
    y + multiplication_naive (x - 1) y
;;
```

On peut faire mieux avec la *multiplication russe* (qui utilise simplement la multiplication par 2 que l'on peut voir comme l'addition d'un nombre à lui-même) :

```
let rec multiplication_russe x y =
  if x = 0 then
    0
  else
    let x' = x / 2
    and y' = 2 * y in
    let prod = multiplication_russe x' y' in
    if x mod 2 = 1 then
      prod + y
    else
      prod
;;
```

2.2. Exponentiation

```
let rec exponentiation_rapide a n =
  if n = 0 then
    1
  else
    let r = exponentiation_rapide (a * a) (n / 2) in
    if n mod 2 = 1 then a * r else r
;;
```

On pourra noter qu'il s'agit du même algorithme que la multiplication russe (mais appliqué à un autre groupe).

3. Fonctionnement, pile d'appel

On peut retenir, pour comprendre le fonctionnement d'appels récursifs (mais, en fait, de tout appel de fonction), que lorsqu'une fonction en appelle une autre, elle...

1. interrompt ce qu'elle fait,
2. note toutes les informations nécessaires pour reprendre plus tard (où elle en est exactement, quelles sont les valeurs des différents arguments),
3. stocke ses informations dans la *pile d'appel*,
4. fait l'appel de fonctions proprement dit,
5. récupère les informations stockées le plus récemment dans la pile d'appel pour retrouver l'état précédent,
6. reprends où elle en était.

Pour comprendre un peu comment se déroulent les appels récursifs, on peut utiliser la fonctionnalité de *tracage* d'OCaml (les indentations ont été ajoutées à la main).

```
# #trace fact ;;
# fact 5 ;;
fact <-- 5
  fact <-- 4
    fact <-- 3
      fact <-- 2
        fact <-- 1
        fact --> 1
      fact --> 2
    fact --> 6
  fact --> 24
fact --> 120
- : int = 120
```

4. Récursivité terminale

On peut parfois optimiser les appels récursifs, en omettant les phases 5 et 6 d'un appel de fonction. Cela est possible si le résultat de l'appel récursif est directement utilisé comme résultat de l'appel de la fonction courante.

Exemple Considérons une fonction qui, étant donné une fonction f, un entier n et un élément x, calcule l'itéré n-ème de f appliqué à x. Une première version est la suivante :

```
let rec itere f n x =
    if n = 0 then x
    else f (itere f (n - 1) x)
;;
```

Cependant, après avoir exécuté l'appel `itere f (n - 1) x`, il faut encore effectuer un post-traitement qui consiste ici à appliquer `f` une nouvelle fois. On peut l'éviter en modifiant la fonction ainsi :

```
let rec itere2 f n x =
    if n = 0 then x
    else itere f (n - 1) (f x)
;;
```

Sans entrer dans des détails trop techniques, notons qu'une méthode usuelle pour obtenir une fonction terminale récursive est d'utiliser une fonction auxilliaire qui comporte un argument supplémentaire correspondant à un accumulateur contenant la valeur du résultat en cours de construction.

Exemple Une version terminale récursive de la fonction factorielle est :

```
let fact2 n =
  let rec aux n acc =
    (* r est le résultat en cours de construction *)
    if n = 0 then
      acc
    else
      aux (n - 1) (n * acc)
  in
  aux n 1
;;
```

Une fonction récursive terminale est très efficace, elle est optimisée par OCaml et se comporte comme une boucle **while**. La fonction précédente pourrait se réécrire (avec des références) :

```
let fact2bis n =
  let acc = ref 1
  and i = ref n in
  while !i > 0 do
```

```
    acc := !acc * !i;
    i := !i - 1
  done;
  !acc
;;
```

II. Introduction à la complexité des fonctions récursives

Nous allons, pour l'instant, étudier la complexité des fonctions récursives dans le cas où pour une entrée de taille n, les appels récursifs se font sur ses entrées de taille $n-1$.

1. Notations de Landau

Étant donné deux suites (u_n) et (v_n), on définit :

- ⋆ $u_n = \mathrm{O}(v_n)$ s'il existe un $C > 0$ tel que $u_n \leq C v_n$ pour tout $n \in \mathbf{N}$ (ou, au moins, à partir d'un certain rang) ;
- ⋆ $u_n = \Omega(v_n)$ si $v_n = \mathrm{O}(u_n)$, autrement dit s'il existe un $C > 0$ tel que $C v_n \leq u_n$ pour tout $n \in \mathbf{N}$ (ou, au moins, à partir d'un certain rang) ;
- ⋆ $u_n = \Theta(v_n)$ si $u_n = \mathrm{O}(v_n)$ et $u_n = \Omega(v_n)$, autrement dit s'il existe des réels $0 < C \leq D$ tels que $C v_n \leq u_n \leq D v_n$ pour tout $n \in \mathbf{N}$ (ou, au moins, à partir d'un certain rang). Notons qu'il s'agit d'une relation d'équivalence.

2. Un seul appel récursif

On a vu pour l'instant des fonctions récursives basées sur des parcours de liste et sur les entiers. En terme de complexité, il est clair que l'on a comme relations :

Factoriel $c_n = c_{n-1} + \Theta(1)$

Tri par insertion $c_n = c_{n-1} + \Theta(n)$

Ce type de récurrence est facile à résoudre.

Proposition 1

Si (c_n) vérifie la relation

$$\forall\, n \in \mathbf{N}^\star,\ c_n = c_{n-1} + b_n$$

avec $b_n = \Theta\left(n^\alpha\right)$ pour un réel $\alpha \geq 0$ fixé, alors $c_n = \Theta\left(n^{\alpha+1}\right)$.

Preuve 1 — Raisonnements sur les sommes
Notons tout d'abord que :

$$\forall\, n \in \mathbf{N},\ c_n = c_0 + \sum_{k=1}^{n} b_k$$

Or, à une constante multiplicative près, on a $b_k \leq n^\alpha$ d'où $c_n \leq c_0 + n \times n^\alpha$
De plus, à une constante multiplicative près toujours, au moins $n/2$ termes de la somme sont supérieurs ou égaux à $(n/2)^\alpha$, d'où $c_n \geq (n/2)^{\alpha+1}$.

Preuve 2 — Encadrement par des intégrales
Pour tout $n \geq 1$, on a :

$$\int_{n-1}^{n} t^\alpha \,\mathrm{d}t \leq n^\alpha \leq \int_{n}^{n+1} t^\alpha \,\mathrm{d}t$$

On en déduit par relation de Chasles que :

$$\frac{n^{\alpha+1}}{\alpha+1} = \int_0^n t^\alpha \,\mathrm{d}t \leq \sum_{k=1}^{n} k^\alpha \leq \int_1^{n+1} t^\alpha \,\mathrm{d}t = \frac{(n+1)^{\alpha+1} - 1}{\alpha+1}$$

Ainsi, le calcul de la factoriel est de complexité $\Theta(n)$ (en nombre de multiplications), et le tri par insertion (ainsi que le tri par sélection) est en $\Theta(n^2)$.

Remarque Concernant la complexité de la factorielle, la taille des entiers manipulés croît très vite, donc on ne peut considérer que les multiplications s'effectuent en temps constant. Cependant, dans un grand nombre de cas pratiques, on effectuera ces calculs modulo un entier fixé, auquel cas les multiplications sont bien en temps constant.

3. Cas général

On s'intéresse maintenant aux récurrences de la forme :

$$u_{n+1} = a \cdot u_n + b_n$$

avec $a \in \mathbf{N}^\star$ et (b_n) à valeurs positives.

On peut se ramener au cas $a = 1$ en posant $v_n = \dfrac{u_n}{a^n}$ pour tout $n \in \mathbf{N}$ puisqu'alors on a pour tout n, $v_{n+1} = v_n + \dfrac{b_n}{a^n}$ et donc

$$\forall\, n \in \mathbf{N},\ v_n = v_0 + \sum_{k=0}^{n-1} \frac{b_k}{a^k} \qquad \textbf{et} \qquad \forall\, n \in \mathbf{N},\ u_n = a^n \left(u_0 + \sum_{k=0}^{n-1} \frac{b_k}{a^k} \right)$$

On en déduit que :

- ⋆ si la série de terme général $\dfrac{b_n}{a^n}$ converge, alors $u_n = \Theta(a^n)$;
- ⋆ si $\dfrac{b_n}{a^n} = \Theta(1)$, alors $u_n = \Theta(na^n)$;
- ⋆ si $\dfrac{b_n}{a^n} = \Theta(\lambda^n)$ pour $\lambda > 1$, alors $u_n = \Theta((a\lambda)^n)$.

En s'inspirant du cas pour $a = 1$, on peut généralisers le cas intermédiaire : si $\dfrac{b_n}{a^n} = \Theta(n^\alpha)$, alors $u_n = \Theta(n^{\alpha+1} a^n)$.

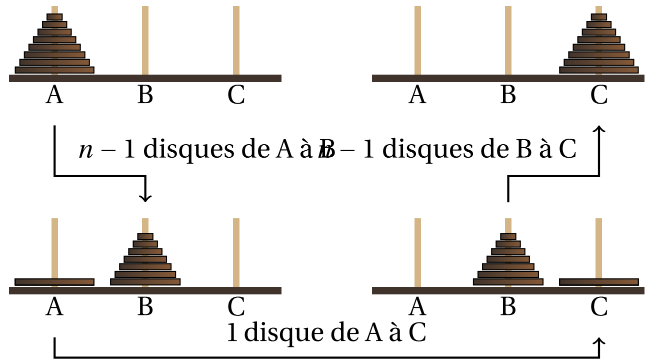
4. Un exemple : les tours de Hanoï

Il s'agit d'un casse-tête où l'on a trois tiges A, B et C et n disques sur la tige A, de taille croissante du haut vers le bas. Le but est de transférer, en moins de coups possible, tous les disques vers la tige C en respectant les règles suivantes :

- ⋆ on ne peut déplacer plus d'un disque à la fois ;
- ⋆ on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

Pour résoudre le problème avec n disques, il suffit de savoir le faire avec $n-1$ disques. En effet,

1. on transfère $n-1$ disques de la tige A à la tige B,
2. on transfère le disque n de la tige A à la tige C,
3. on transfère les $n-1$ autres disques de la tige B à la tige C.



Le programme correspondant, qui affiche les déplacements des disques, est alors très simple :

```
let rec hanoi a b c n =
  if n >= 1 then begin
    hanoi a c b (n - 1);
    Printf.printf "%s -%d-> %s\n" a n c;
    hanoi b a c (n - 1)
  end
;;
```

Si c_n désigne le nombre d'étapes nécessaires pour transférer n disques, alors clairement :

$$c_0 = 0 \qquad \forall\, n \in \mathbf{N},\ c_{n+1} = c_n + 1 + c_n = 2c_n + 1$$

On peut résoudre exactement cette récurrence :

$$\forall\, n \in \mathbf{N},\ c_n = 2^n - 1$$

Néanmoins, à titre d'exercice, on peut utiliser la méthode précédente. On a :

$$\forall\, n \in \mathbf{N},\ c_n = 2c_{n-1} + 1 = ac_{n-1} + b_n$$

avec $a = 2$ et $b_n = \Theta(1) = \Theta(1^n)$. Ainsi, on a

$$\sum_{k=1}^{n} \frac{b_k}{2^k} = \Theta(1)$$

et donc

$$c_n = 2^n \left(c_0 + \sum_{k=1}^{n} \frac{b_k}{2^k} \right) = \Theta(2^n)$$