TD : Récursivité

1 Exercices pratiques

Exercice 1. *Nombres de Catalan.* Les nombres de Catalan satisfont la relation $c_0 = 1$ et à la relation $c_n = c_{n-1}\frac{2(2n-1)}{n+1}$ pour tout $n \geq 1$. Écrire une fonction récursive `catalan: int -> int` renvoyant c_n.

Exercice 2. *Que font les fonctions ?.* Devinez ce que font les fonctions suivantes, dont l'argument n est supposé positif ou nul.

```
let rec a n = match n with
| 0 -> ()
| _ -> a (n-1); print_int n; print_string " "
and b n = match n with
| 0 -> ()
| _ -> print_int n; print_string " "; b (n-1)
and c n = match n with
| 0 -> ()
| _ -> print_int n; print_string " "; c (n-1); print_int n; print_string " "
```

Exercice 3. *Que fait cette fonction ?.* Deviner le type de la fonction `g`, et ce qu'elle fait :

```
let rec g f q= match q with
 | [] -> []
 | x::r when f x -> x::(g f r)
 | x::r -> (g f r)
;;
```

Exercice 4. *Que fait cette fonction ?.* Deviner le type de la fonction `h`, et ce qu'elle fait :

```
let rec h x = if x <= 1 then 1 else if x mod 2 = 0 then 2 * h (x / 2) else 1 + h (x + 1);;
```

Exercice 5. *Addition récursive terminale.* Écrire une fonction `add : int -> int -> int`, récursive terminale, et effectuant la somme de ses deux arguments (supposés ≥ 0). Les seules opérations autorisées sont l'addition ou la soustraction de 1 à un argument.

Exercice 6. *Multiplication récursive terminale.* Écrire une fonction `mul : int -> int -> int`, utilisant une fonction auxiliaire récursive terminale, et effectuant le produit de ses deux arguments (supposés ≥ 0). Les seules opérations autorisées sont l'addition, la multiplication par 2 et la division par 2 d'un nombre pair.

Exercice 7. *Itérées d'une fonction.* Écrire une fonction itere : `int -> ('a -> 'a) -> 'a -> 'a` calculant la valeur en un point x de la n-ième itérée $f \circ f \circ \cdots \circ f$ d'une fonction f.

Exercice 8. On considère la fonction `g` suivante (qui nécessite une fonction `f`).

```
let rec f q x=match q with
  | [] -> [x]
  | y::p -> y::(f p x)
and g q=match q with
  | [] -> []
  | y::p -> f (g p) y
;;
```

Montrer qu'elle termine, expliquer ce qu'elle fait (et le démontrer) et estimer sa complexité. Peut-on faire la même chose plus rapidement ?

2 Exercices théoriques

Exercice 9. *Des ordres.* **1.** Montrer que l'ordre lexicographique sur $\mathbb{N}^2$ est un bon ordre.

2. On définit l'ordre de Sharkovskii sur $\mathbb{N}^*$ de la façon suivante :

$$3 \prec 5 \prec 7 \prec 9 \prec 11 \prec \cdots \prec 2\times 3 \prec 2\times 5 \prec 2\times 7 \prec \cdots \prec 2^2\times 3 \prec \cdots \prec 2^n \prec 2^{n-1} \prec 2^{n-2} \prec 2 \prec 1$$

Est-ce un bon ordre sur $\mathbb{N}^*$?

Remarque : cet ordre a une utilité ! Le mathématicien Sharkovskii a montré en 1964 que si une fonction continue $f : [0,1] \to [0,1]$ admet un cycle d'ordre $p > 0$ (c'est-à-dire qu'il existe x tel que $f^p(x) = x$ mais $f^k(x) \neq x$ pour $0 < k < p$), alors il existe un n-cycle pour tout $n \succ p$. En particulier dès qu'une telle fonction possède un 3-cycle, elle possède un cycle d'ordre quelconque !

Exercice 10. *Suite de Fibonacci.* On définit la suite de Fibonacci comme

$$F_0 = F_1 = 1 \qquad \text{et} \qquad F_n = F_{n-1} + F_{n-2} \quad \text{pour} \quad n \geq 2.$$

Dans cet exercice, la complexité est comptée en nombre d'opérations arithmétiques.

1. Écrire une fonction récursive `fibo n` calculant F_n.

2. Donner sa complexité en fonction de n.

3. Écrire une version en $O(n)$ faisant usage d'un tableau.

4. Donner une expression du vecteur $\binom{F_{n-1}}{F_n}$ comme produit d'une matrice 2×2 par le vecteur $\binom{F_{n-2}}{F_{n-1}}$, et en déduire (sans coder) une approche en $O(\log n)$.

Exercice 11. *Les Tours de Hanoï, variante.* On reprend l'exemple des tours de Hanoï, mais on s'interdit tout déplacement du piquet A vers le piquet C, les autres règles étant conservée. Montrer que le jeu possède toujours une solution, et majorer le nombre de mouvements à effectuer pour déplacer une pile de taille n.

Exercice 12. *Tri rapide sur les listes.* On s'intéresse à un autre tri efficace, différent du tri fusion.

1. Écrire une fonction `partition x q` de type `'a -> 'a list -> 'a list * 'a list`, renvoyant un couple de listes (q_1, q_2) contenant les éléments de q, les éléments de q_1 étant inférieurs ou égaux à x, et ceux de q_2 strictement supérieurs. La complexité doit être linéaire en la taille de q.

2. En déduire une fonction `tri_rapide q` de tri d'une liste :
- si la liste a au plus 1 élément, on la renvoie à l'identique ;
- sinon, on partitionne la queue de la liste autour de la tête de liste, on trie récursivement les deux listes de la partition, et on concatène les trois morceaux pour obtenir une liste triée.

3. Justifier brièvement la terminaison et la correction.

4. Estimer la complexité dans le pire cas et le meilleur cas.

Usuellement, on trie plutôt des tableaux que des listes chaînées. Il est possible d'écrire une version du tri fusion pour les tableaux de type `'a array -> unit`, triant un tableau (en le modifiant) avec une complexité en $O(n \log n)$. Cependant, une étape de ce tri est la fusion de deux portions triées du tableau, qui est très difficile à réaliser *en place* (c'est-à-dire sans faire usage d'un tableau annexe). C'est pour cette raison qu'en pratique, on préfère souvent le tri rapide, qui peut s'écrire en place, et est donc un peu plus rapide en pratique.

Exercice 13. *Variante de l'algorithme de Strassen.* Imaginons qu'on sache multiplier deux matrices de taille 3×3 avec p multiplications. Quel est la valeur maximale de p qui permettrait de faire mieux (asymptotiquement) que l'algorithme de Strassen ? *Remarque : l'existence d'un tel algorithme est un problème ouvert !*

Exercice 14. *Suites de Goodstein (faibles).* On définit une suite $(u_b)_{b\geq 2}$ de la façon suivante.

- $u_2 \in \mathbb{N}$ est fixé. Dans l'exemple, nous prendrons $u_2 = 266$, dont la décomposition en binaire est $u_2 = 2^8+2^3+2^1$;
- pour $b \geq 2$, on obtient u_{b+1} à partir de u_b en remplaçant dans la décomposition de u_b en base b la base par $b+1$, et en retranchant 1. Par exemple avec $u_2 = 266$, on trouve successivement $u_3 = 3^8+3^3+3^1-1 = 3^8+3^3+2\cdot 3^0 (= 6590)$, $u_4 = 4^8+4^3+2\cdot 4^0-1 = 4^8+4^3+4^0 (= 65601)$, $u_5 = 5^8+5^3+5^0-1 = 5^8+5^3 (= 390750)$, $u_6 = 6^8+6^3-1 = 6^8+5\cdot 6^2+5\cdot 6^1+5\cdot 6^0 (= 1679831)$, etc...

Il semblerait que dans ce processus, $u_b \underset{b\to+\infty}{\to} +\infty$, au moins dans cet exemple. Montrer pourtant que quel que soit $u_2 \in \mathbb{N}$ choisi, il existe un rang u_b pour lequel $u_b = 0$.

Remarque : ne cherchez pas à coder, sauf pour de très petites valeurs de u_2 : la convergence vers 0 est extrêmement lente à obtenir ! Si le résultat est déja surprenant, on peut montrer le même résultat pour les suites de Goodstein *fortes*, ou même les exposants sont décomposés dans la base b de manière répétée. Par exemple, pour $u_2 = 266 = 2^{2^{2^1+1}} + 2^{2^1+1} + 2^1$, on a $u_3 = 3^{3^{3^1+1}} + 3^{3^1+1} + 3^1 - 1 \simeq 4\times 10^{38}$... L'arithmétique des entiers ne suffit plus pour les suites de Goostein fortes, il faut avoir recours à la théorie des ensembles pour montrer que $u_b = 0$ pour un certain b.