TD1: Fonctions récursives

MPSI Lycée Pierre de Fermat

Exercice 1. *Récursivité terminale*

Q1. Écrire une fonction **récursive terminale** `max_liste: 'a list -> 'a` calculant le maximum d'une liste non-vide.

Q2. Déterminer, parmi les fonctions suivantes, lesquelles sont récursives terminales :

```
(* Fusionne l1, l2 deux listes triées en une unique liste triée *)
let rec fusion (l1: 'a list) (l2: 'a list) : 'b list =
  match l1, l2 with
  | [], _ -> l2
  | _, [] -> l1
  | x1 :: q1, x2 :: q2 ->
    if x1 < x2 then x1 :: fusion q1 l2
    else           x2 :: fusion l1 q2

(* Exécute f sur chaque élément de l *)
let rec iter (f: 'a -> unit) (l: 'a list) : unit =
  match l with
  | []    -> ()
  | x :: q -> f x; iter f q

(* Renvoie le i-ème élément de l *)
let rec nth (l: 'a list) (i: int) : 'a =
  match l with
  | [] -> failwith "erreur"
  | x :: q -> if i > 0 then nth q (i-1) else x

(* Sépare l en deux listes de tailles égales à 1 près *)
let rec separer (l: 'a list) : (a list * 'a list) =
  match l with
  | [] -> ([], [])
  | [x] -> ([x], [])
  | x :: y :: q ->
    let l1, l2 = separer q in
    (x :: l1, y :: l2)
```

Q3. Rendre les fonctions précédentes récursives terminales lorsqu'elles ne le sont pas.

Q4. Étant donnée une liste L d'entiers, on cherche à découper L en sous-listes d'éléments consécutifs de L dont la somme est inférieure à 10. Par exemple, pour $L = [2, 3, 9, 2, 1, 5, 3]$, un bon découpage de L serait $[[2, 3], [9], [2, 1, 5], [3]]$. Écrire une fonction récursive terminale calculant un découpage valide d'une liste. On pourra passer par la fonction auxiliaire suivante :

```
(* Découpe l en paquets d'éléments consécutifs de somme au plus 10.
   - p est le paquet en cours de construction,
   - r est l'espace restant dans p avant d'atteindre 10
   - res est la liste des paquets déjà construits. *)
let rec construire (l: int list) (p: int list) (r: int) (res: int list list)
    : int list list = ...
```

Exercice 2. *Compteur binaire*

Tout entier $n \in \mathbb{N}$ peut s'écrire comme somme de puissances de 2, $n = \sum_{i=0}^{l-1} x_i 2^i$, où chaque x_i vaut 0 ou 1. On parle d'**écriture en base 2** de n, ou encore d'écriture **binaire**. On notera $\overline{x_{l-1} \dots x_0}^2$ l'entier dont les chiffres en base 0 sont $x_{l-1}, \dots, x_0$. Par exemple, $\overline{10011}^2 = 16 + 2 + 1 = 19$, et $\overline{11111}^2 = 16+8+4+2+1 = 31$. On admettra qu'un entier n donné n'admet qu'une écriture binaire, à rajout de zéros près. Par exemple, 5 peut s'écrire 101, mais aussi 00101.

Q1. Donner les écritures binaires des entiers suivants :

a) $\overline{1000}^2 + 1$

b) $\overline{1011}^2 + 1$

c) $\overline{10111}^2 + 1$

d) $\overline{100100111}^2 + 1$ (ne repassez pas par le décimal !)

Q2. En déduire une règle générale pour calculer l'écriture binaire du successeur d'un entier $n = \overline{x_{l-1} \dots x_0}^2$.

En OCaml, on représentera l'écriture binaire d'un nombre entier par une liste de booléens. On stockera les chiffres en partant des unités, de telle sorte que le chiffre de poids fort sera à la fin de la liste. Par exemple, l'entier $19 = \overline{10011}^2$ sera représenté par la liste `[true; true; false; false; true]`. L'entier 0 sera représenté par la liste vide.

Q3. Écrire une fonction OCaml `succ: bool list -> bool list` qui, étant donnée l'écriture binaire d'un entier n, renvoie l'écriture binaire de $n + 1$.

Q4. (Optionnel) Écrire la fonction précédente en récursif terminal.

Q5. Quelle est la complexité pire cas de `succ` en fonction de l la taille de la liste d'entrée ?

Néanmoins, on peut remarquer que dans de nombreux cas, `succ` n'a besoin que de quelques étapes pour s'exécuter : seule la suite initiale de 1 doit être parcourue. Afin de trouver une mesure plus fine du temps d'exécution de `succ`, on se proposer d'étudier sa **complexité amortie**. On pose, pour $n \in \mathbb{N}$, $C(n)$ le nombre de chiffres qui diffèrent entre n et $n + 1$ lorsqu'ils sont écrits en binaire, et on admettra que $C(n)$ est un bonne estimation de la complexité de `succ`. On considère ensuite $S(n) = C(0) + C(1) + \dots + C(n-1)$, le coût **total** de l'application de `succ` n fois d'affilée depuis 0. Par exemple, pour appliquer `succ` 4 fois :

```
0000
0001 (1 chiffre  de différence)
0010 (2 chiffres de différence)
0011 (1 chiffres de différence)
0100 (3 chiffres de différence)
```

Au total, $S(4) = 1 + 2 + 1 + 3 = 7$.

Q6. Déterminer $S(7), S(15)$.

Q7. On pose $v_k = S(2^k - 1)$ pour $k \in \mathbb{N}$. Proposer une formule de récurrence sur v_k, puis en trouver une expression directe.

Q8. En déduire que $S(n) = \mathcal{O}(n)$. (*Indication : considérer* $k = \lceil \log_2(n) \rceil$.)

On dit que `succ` a une **complexité amortie constante** : dans une suite de n opérations, le coût total est linéaire, donc le coût moyen est constant ! Autrement dit, si un programme utilise cette représentation des entiers (pour faire un compteur par exemple), le coût moyenné sera le même que si `succ` était en $\mathcal{O}(1)$ pire cas.

Q9. (Optionnel) Écrire une fonction `add: bool list -> bool list -> bool list` permettant d'additionner deux nombres en binaire, puis une autre permettant de multiplier deux nombres en binaire.