

DM Centrale 2019

Published on 2021-12-28 by **Marc de Falco**

Remarques globales

Certaines questions sont fausses ou mal posées (c'est comme ça parfois le jour du concours :/)

- Question 1 : on renvoie une nouvelle liste vu qu'on ne peut pas les modifier en OCaml
- Question 5 et 6 : très mal posées ainsi, vous les sautez
- Question 11, 12 et 13 : les portes logiques ne sont pas techniquement au programme, ce n'est pas très compliqué, mais si ça ne vous inspire pas, n'hésitez pas à sauter
- Question 16 : l'indication est fausse il faut chercher le plus *grand*
- Question 18 : C'est le nombre de combinaisons *après* être arrivé à la combinaison donnée qui importe
- Question 25 : L'ordre *précédent* est ambigu, je pense que c'est l'ordre lexicographique sur les uplets.

Questions de programmation

[Vous pouvez accéder à un énoncé ici](#)

Question 1

Quelques exemples significatifs, j'écris $a_1..a_n$ pour $[a_1; \dots; a_n]$ et $a \rightarrow \text{Some } b$ si suivant a vaut $\text{Some } b$ OU $a \rightarrow \text{None}$ si suivant a vaut None .

1. Comme $1011 \rightarrow \text{Some } 1100$ on a $01011 \rightarrow \text{Some } 01100$ et $11011 \rightarrow \text{Some } 11100$.
2. Comme $1111 \rightarrow \text{None}$ on a $01111 \rightarrow \text{Some } 10000$ mais $11111 \rightarrow \text{None}$.
3. $\emptyset$ avec 0 chiffres qui a une écriture vide n'a pas de successeur.

On voit donc :

- Qu'on peut raisonner pour suivant l quand l vaut $t::q$ avec suivant q .
- Qu'on va alors faire un cas selon que la valeur de retour soit None ou non, puis selon que t soit 1 ou 0.
- Qu'on aura sûrement besoin d'une fonction `let rec zeros (l : int list) : int list = ...` telle que `zeros q` renvoie une liste avec autant de 0 que d'éléments de q .
Note : on peut aussi le faire directement avec `List.init (List.length q) (fun _ -> 0)` mais à l'écrit c'est plutôt à vous de le faire.

On pourra donc avoir une structure telle que :

```

let rec suivant l = match l with
| [] -> None (* cas 3 *)
| t :: q -> match suivant q with
| None -> (* c'est le cas 2 *)
| Some v -> (* c'est le cas 1 *)

```

Si on voulait le faire en renvoyant un couple `int list * bool` indiquant le résultat et un booléen pour dire si on a réussi, on écrirait :

```

let rec zeros l = match l with
| [] -> []
| _::q -> 0 :: zeros q

let rec suivant l = match l with
| [] -> ([], false)
| t :: q -> let q', b = suivant q in
    if b
    then (t :: q', true)
    else if t = 1
    then (l, false)
    else (1 :: zeros q, true)

```

On aurait alors par exemple :

```

>>> suivant [0;1;1;0;1]
([0;1;1;1;0], true)

>>> suivant [0;1;1;1;1]
([1;1;1;1;1], true)

>>> suivant [1;1;1;1;1]
([1;1;1;1;1], false)

```

On voit que c'est maladroit de donner une valeur `int list` quand le booléen est `false` car cela signifie qu'on n'a pas réussi à calculer un successeur. C'est pourquoi , je propose plutôt de renvoyer `Some s` quand on a réussi à calculer le successeur `s` (c'est l'équivalent du couple `(s, true)`) et `None` sinon (c'est l'équivalent d'un couple quelconque `(_, false)` où `_` une liste quelconque d'entiers).