TD 6

ENTRAÎNEMENT

1 Ensembles représentés par des listes

Dans cette partie, on représente des parties finies de $\mathbb{Z}$ par des listes strictement croissantes d'entiers. Une telle représentation est unique ; on notera $U, V, \ldots$ les parties et $u, v, \ldots$ les listes qui leur correspondent.

À chaque fois que l'on vous demande d'écrire une fonction renvoyant une liste représentant un ensemble, il est impératif que la liste renvoyée soit bien strictement croissante.

Les types donnés dans l'énoncé correspondent au cas des listes d'entiers, mais vos fonctions devraient logiquement avoir un type plus général (`'a list` plutôt que `int list`) : ce n'est bien sûr pas un problème.

EXERCICE 1 **Quelques fonctions usuelles sur les listes**

On appelle prédicat *une fonction de type* `'a -> bool`. *On dit alors que* `x` *vérifie le prédicat* `pred` *si* `pred x = true`.

1. *Écrire une fonction* `pour_tout : ('a -> bool) -> 'a list -> bool` *telle que* `pour_tout pred u` *soit vrai si et seulement si tous les éléments de u vérifient pred.*
   On évitera les calculs inutiles.
2. *Écrire une fonction* `existe : ('a -> bool) -> 'a list -> bool` *telle que* `existe pred u` *soit vrai si et seulement si il existe $x \in u$ vérifiant pred.*
   On évitera les calculs inutiles.
3. *Écrire une fonction* `filtre : ('a -> bool) -> 'a list -> 'a list` *telle que* `filtre pred u` *renvoie la liste des éléments de u vérifiant pred (ces éléments seront dans le même ordre que dans la liste u).*

EXERCICE 2 **Opérations ensemblistes, versions élémentaires**

Essayez de répondre aux questions de cet exercice sans utiliser de `let rec` *(et, évidemment, sans références !).*

1. *Écrire une fonction* `appartient : int list -> int -> bool` *qui détermine si un entier est élément d'une liste.*
2. *Écrire une fonction* `inclus : int list -> int list -> bool` *telle que* `inclus u v` *renvoie* `true` *si et seulement si $U \subset V$.*
   Quelle est la complexité de cette fonction (dans le pire des cas, en fonction de $|u|$ et $|v|$).
3. *Que penser de l'implémentation suivante de la fonction testant l'égalité de deux ensembles ?*

   ```
   let egal u v = (inclus u v) && (inclus v u)
   ```

   Proposer une version plus efficace.
4. *Écrire une fonction* `inter : int list -> int list -> int list` *renvoyant l'intersection de deux listes.*

EXERCICE 3 **Opérations ensemblistes, versions « optimisées »**

Écrire des versions efficaces (ici, cela signifie que leur complexité doit être proportionnelle à la longueur de leurs arguments) des fonctions `inclus`*,* `union` *et* `inter`*. On pourra commencer par reprendre la fonction que nous avions écrite pour réaliser l'étape de fusion du tri fusion et la modifier pour obtenir une fonction* `union` *correcte, puis faire les modifications nécessaires pour les deux autres fonctions.*

2 Combinatoire

Dans cette partie, on s'intéresse à la génération des différents objets combinatoires élémentaires associés à un ensemble : parties, combinaisons, permutations, *p*-listes et combinaisons avec répétitions.

Je vous conseille de définir la fonction suivante , qui s'avère très utile :

```
let map_prefixe x = List.map (fun u -> x :: u)
(*
  # map_prefixe 3 [[1; 2]; []; [3]];;
  - : int list list = [[3; 1; 2]; [3]; [3; 3]]
*)
```

Dans toute la partie, on pourra supposer sans le vérifier que les listes que l'on reçoit en argument sont sans répétition.

EXERCICE 4

Définir une fonction `parties : 'a list -> 'a list list` *qui renvoie la liste des parties de u (l'ordre dans lequel les parties sont renvoyées n'a pas d'importance).*

```
# parties [1; 2; 5];;
- : int list list = [[]; [5]; [2]; [2; 5]; [1]; [1; 5]; [1; 2]; [1; 2; 5]]
```

EXERCICE 5

1. *Définir une fonction* `combinaisons : 'a list -> int -> 'a list list` *telle que* `combinaisons u n` *renvoie la liste des combinaisons de n éléments de u (l'ordre dans lequel les combinaisons sont renvoyées n'a pas d'importance).*

```
# combinaisons [1; 2; 3; 6] 2;;
- : int list list = [[3; 6]; [2; 6]; [2; 3]; [1; 6]; [1; 3]; [1; 2]]
```

2. *Donner une autre version de la fonction* `parties` *utilisant la fonction* `combinaisons`*.*
3. *Définir une fonction* `avec_repetitions : 'a list -> int -> 'a list list` *telle que* `avec_repetitions u n` *renvoie la liste des combinaisons avec répétitions de n éléments de u :*

```
# avec_repetitions [1; 2; 3; 7] 3;;
- : int list list =
[[1; 1; 1]; [1; 1; 2]; [1; 1; 3]; [1; 1; 7]; [1; 2; 2]; [1; 2; 3]; [1; 2; 7];
 [1; 3; 3]; [1; 3; 7]; [1; 7; 7]; [2; 2; 2]; [2; 2; 3]; [2; 2; 7]; [2; 3; 3];
 [2; 3; 7]; [2; 7; 7]; [3; 3; 3]; [3; 3; 7]; [3; 7; 7]; [7; 7; 7]]
```

EXERCICE 6

1. *Écrire une fonction* `insertions : 'a -> 'a list -< 'a list list` *telle que* `insertions x u` *renvoie la liste des listes obtenues en insérant x à une certaine position dans u :*

```
# insertions 7 [2; 5; 3];;
- : int list list = [[7; 2; 5; 3]; [2; 7; 5; 3]; [2; 5; 7; 3]; [2; 5; 3; 7]]
```

2. *Écrire une fonction* `applatit : 'a list list -> 'a list` *qui renvoie la concaténation d'une liste de listes (c'est la fonction* `List.flatten` *de la bibliothèque standard) :*

```
# applatit [[1; 2; 3]; [7; 12]; [4; 1]];;
- : int list = [1; 2; 3; 7; 12; 4; 1]
```

3. *Écrire une fonction* `permutations : 'a list -> 'a list list` *renvoyant la liste des permutations de son argument.*

```
# permutations [2; 3; 5];;
- : int list list = [[2;3;5]; [3;2;5]; [3;5;2]; [2;5;3]; [5;2;3]; [5;3;2]]
```

EXERCICE 7

Écrire une fonction `plistes : 'a list -> int -> 'a list list`.

```
# plistes [1; 2; 3] 3;;
- : int list list =
[[1; 1; 1]; [1; 1; 2]; [1; 1; 3]; [1; 2; 1]; [1; 2; 2]; [1; 2; 3]; [1; 3; 1];
 [1; 3; 2]; [1; 3; 3]; [2; 1; 1]; [2; 1; 2]; [2; 1; 3]; [2; 2; 1]; [2; 2; 2];
 [2; 2; 3]; [2; 3; 1]; [2; 3; 2]; [2; 3; 3]; [3; 1; 1]; [3; 1; 2]; [3; 1; 3];
 [3; 2; 1]; [3; 2; 2]; [3; 2; 3]; [3; 3; 1]; [3; 3; 2]; [3; 3; 3]]
```

3 Permutations représentées par des tableaux

EXERCICE 8

Dans cette partie, on représente des permutations de l'intervalle d'entiers $[0, n[$ *(c'est-à-dire une bijection de cet ensemble sur lui-même) par un tableau t à n éléments. t représente l'application qui à* $i \in [0, n[$ *associe* $t.(i)$.

1. *Écrire une fonction* `estPermutation` *qui prend un tableau d'entiers en argument et retourne un booléen disant si ce tableau représente bien une permutation (de* $[0; |t|[$*).*

Par la suite, on confond «permutation» et «tableau d'entiers représentant une permutation».

2. *Écrire une fonction* `composer` *prenant deux permutations (sur le même ensemble, c'est-à-dire représentées par des tableaux de même taille) en arguments, et renvoyant la composée de ces permutations.*

3. *Écrire une application* `inverser` *prenant une permutation en argument et renvoyant la permutation réciproque.*

4. *t étant une permutation et k un entier, on note* t^k *la permutation* $t \circ t \cdots \circ t$ *(où t apparaît k fois), avec la convention que* t^k *désigne l'identité si* $k = 0$ *et* $(t^{-1})^{|k|}$ *si* $k < 0$.

Écrire une fonction `puissance` *prenant en entrée une permutation t et un entier k et renvoyant* t^k.

5. *On définit l'ordre d'une permutation t comme le plus petit entier k strictement positif tel que* t^k *est l'identité.*

Vu que je ne sais pas exactement ce que vous avez déjà fait en mathématiques, je précise que l'ordre existe forcément, et que la seule chose qu'on puisse en dire *a priori* est qu'il divise $n!$, ce qui ne sert à rien ici.

Écrire une fonction `ordre` *donnant l'ordre d'une permutation.*

6. *La période d'un indice i pour t est le plus petit* $k > 0$ *tel que* $t^k(i) = i$.

L'ordre existe toujours (et c'est un diviseur de n, ce qui ne nous est pas utile).

Écrire une fonction `periode` *prenant en argument une permutation t, un indice i et retournant la période de i pour t.*

7. *L'orbite de i pour t est l'ensemble des j tels qu'il existe k vérifiant* $t^k(i) = j$.

Écrire une fonction `estDansOrbite` *prenant en arguments un tableau t et deux indices i et j et retournant un booléen disant si j est dans l'orbite de i.*

8. *Une transposition est une permutation qui échange deux éléments* distincts *et laisse les autres inchangés.*

Écrire une fonction `estTransposition` *prenant en argument une permutation et retournant un booléen indiquant s'il s'agit d'une transposition.*

9. *Un cycle (simple) est une permutation dont exactement une des orbites est de taille strictement supérieure à un. Toutes les autres orbites, s'il y en a, sont réduites à des singletons.*

Écrire une fonction `estCycle` *qui prend une permutation t en argument et retournant un booléen indiquant s'il s'agit d'un cycle.*