Durée : 2h.
Aucun document ni calculatrice autorisé.
Les 3 exercices sont indépendants.

Important : Lorsqu'une fonction est demandée, celle-ci doit être écrite en OCaml.

Quand l'énoncé demande de coder une fonction, il n'est pas nécessaire de justifier que celle-ci est correcte. On suppose que la fonction n'est utilisée que sur des entrées satisfaisant les préconditions de la spécification. Il est acceptable (et même normal, pour certaines questions) qu'une fonction telle qu'écrite ait un type plus large que celui demandé (par exemple, une fonction de type `'a -> 'a` quand on demande le type `int -> int`) tant qu'elle respecte la spécification.

Une grande importance sera apporté à la clarté et la lisibilité des programmes. Il est recommandé de choisir des noms de variables intelligibles. Il n'est en revanche pas nécessaire de réécrire la spécification d'une fonction si celle-ci est donnée dans l'énoncé.

On rappelle qu'en OCaml, les listes sont immuables. Aucune réponse utilisant des listes comme si elles étaient mutables ne recevra de points, même partiels.

Exercice 1 :
On considère la fonction suivante :

```
1 let rec mystere p l =
2   match l with
      | [] -> false
4     | x::q -> p x || mystere p q
```

1. Déterminer, en justifiant, le type de `mystere`.
2. (a) Déterminer une spécification (c'est à dire une description de son résultat en fonction de l'entrée, sans préciser le détail de l'algorithme) de la fonction `mystere`.
   (b) Donner un exemple de calcul de `mystere p q` avec `p` de votre choix et `l` non vide. On précisera uniquement le résultat, il n'est pas nécessaire de détailler l'exécution.
   (c) Prouver la correction de `mystere` par rapport à la spécification de la question 2a.

Exercice 2 : Ensembles finis d'entiers sous forme de liste...

Pour $n \in \mathbb{N}$, on note $E_n = [\![1; n]\!]$ (donc en particulier $E_0 = \emptyset$.

On représente une partie A de E_n par une liste $[a_0; a_1; \ldots; a_{p-1}]$ avec $1 \leqslant a_0 < a_1 < \cdots < a_{p-1} \leqslant n$ et $A = \{a_0; a_1; \ldots; a_{p-1}\}$

1. Écrire une fonction `ajout : int list -> int -> int list` telle que si A est une partie de E_n représentée par la liste `l`, et $x \in E_n$, la liste `ajout l x` représente $A \cup \{x\}$.

   Par exemple :

   - `ajout [2; 3; 6] 4 = [2; 3; 4; 6]`
   - `ajout [2; 3; 6] 3 = [2; 3; 6]`

   Rappel : On ne s'intéresse pas à ce que fait la fonction si la liste n'est pas une représentation correcte d'une partie de E_n.

2. Similairement, écrire une fonction `retire : int list -> int -> int list` telle que si A est une partie de E_n représentée par la liste `l`, et $x \in E_n$, la liste `retire l x` représente $A \setminus \{x\}$.

   Par exemple :

   - `retire [2; 3; 6] 4 = [2; 3; 6]`
   - `retire [2; 3; 6] 3 = [2; 6]`

3. Écrire une fonction `intersection : int list -> int list -> int list` telle que si A_1 et A_2 sont des parties de E_n représentées respectivement par les listes `l1` et `l2`, la liste `intersection l1 l2` représente $A_1 \cap A_2$.

   Par exemple `intersection [2; 3; 6] [1; 2; 4; 5; 6; 42] = [2; 6]`.

4. On donne la fonction suivante

```
let rec union l1 l2 =
  match (l1, l2) with
    | ([], _)          -> l2
    | (_, [])          -> l1
    | (x1::q1, x2::q2) ->
        if x1 = x2 then x1::(union q1 q2)
        else if x1 < x2 then x1::(union q1 l2)
        else (* on a x2 < x1 *) x2::(union l1 q2)
```

   (a) Montrer par récurrence sur la taille totale des listes que si `l1` et `l2` sont des listes représentant A_1 et A_2 des parties de E_n, alors `union l1 l2` calcule la liste représentant $A_1 \cup A_2$.

   (b) Donner, en justifiant, un ordre de grandeur (i.e. avec la notation $\mathcal{O}$) de la complexité en temps dans le pire cas de `union l1 l2` en fonction de m_1 et m_2 les tailles des listes.

5. (a) Écrire une fonction `parties : int -> int -> int list list` telle que pour tous entiers positifs n et p, `parties n p` calcule une liste (peu importe son ordre) des représentations des parties de E_n de taille p.

   Par exemple :

   - `parties 0 5 = [[]]`
   - `parties 5 0 = []`
   - `parties 5 6 = []`
   - `parties 5 5 = [[1; 2; 3; 4; 5]]`
   - `parties 2 1 = [[1]; [2]]`. Le résultat $|[[2]\,;\,[1]]|$ satisferait aussi la spécification.
   - `parties 3 2 = [[1; 2]; [1; 3]; [2; 3]]`. Un résultat avec un ordre différente des 3 parties (du moment que les entiers restent ordonnés selon l'ordre croissant au sein d'une partie) satisferait aussi la spécification.

On rappelle pour cela que `@` est l'opérateur de concaténation des listes en OCaml.

On rappelle également que l'ensemble des parties de E_n à p éléments se partitionne en l'ensemble des parties à p éléments qui contiennent n (donc n est leur plus grand élément) et l'ensemble des parties à p éléments de E_{n-1}.

(b) Détailler l'exécution de `parties 3 2`.

Exercice 3 : ... et sous forme de tableau

On pose maintenant, pour $n \in \mathbb{N}$, $F_n = [\![0; n-1]\!]$ (donc en particulier $F_0 = \{0\}$).

On représente maintenant une partie A de F_n par un tableau de n booléens `[|`b_0`;`b_1`;`$\cdots$`;`b_{n-1}`|]` où $\forall i \in F_n$, b_i est vrai si $i \in A$ et faux sinon.

On remarque que, contrairement à l'exercice précédent, la représentation de A dépend du n choisi : on ne représentera pas de la même façon l'ensemble $\{1;3\}$ en tant que partie de F_4 ou de F_5.

1. (a) Écrire une fonction `ajout : bool array -> int -> unit` telle que si A est une partie de F_n représentée par le tableau `t`, et $x \in F_n$, `ajout t x` modifie `t` pour qu'il représente $A \cup \{x\}$.

   Par exemple, si `t` est initialement le tableau `[|false; true; false; true; false|]` représentant $\{1;3\}$ en tant que partie de F_5 :

   - `ajout t 2` modifie `t` en `[|false; true; true; true; false|]`
   - `ajout t 3` ne modifie pas `t`

   (b) Similairement, écrire une fonction `retire : bool array -> int -> unit` telle que si A est une partie de F_n représentée par le tableau `t`, et $x \in F_n$, `retire t x` modifie `t` pour qu'il représente $A \setminus \{x\}$.

   Par exemple, si `t` est initialement le tableau `[|false; true; false; true; true|]` représentant $\{1;3;4\}$ en tant que partie de F_5 :

   - `retire t 3` modifie `t` en `[|false; true; false; false; true|]`
   - `retire t 2` ne modifie pas `t`

2. (a) Écrire une fonction `union : bool array -> bool array -> unit` talle que si A_1 et A_2 sont des parties de F_n représentées par les tableaux `t1` et `t2`, `union t1 t2` modifie `t1` pour qu'il représente $A_1 \cup A_2$.

   Par exemple, si `t1` est initialement `[|false; true; false; true|]`, alors `union t1 [|false; false; true; true|]` modifie `t1` en `[|false; true; true; true|]`.

   (b) Écrire une fonction `union_copie : bool array -> bool array -> bool array` talle que si A_1 et A_2 sont des parties de F_n représentées par les tableaux `t1` et `t2`, `union_copie t1 t2` renvoie un tableau représentant $A_1 \cup A_2$, sans modifier `t1` ni `t2`.

   Par exemple, `union_copie [|false; true; false; true|] [|false; false; true; true|] = [|false; true; true; true|]`.