Devoir surveillé 1

3 Problèmes

Option informatique MPSI1 & MPSI2

Ce sujet comporte 3 problème indépendants ; on en traitera 2 au choix durant le D.S.
Les sujets sont de difficulté croissante mais chacun comptera pour la moitié des points ; il est suggéré de choisir les sujets en fonction de votre niveau actuel.

- Le premier sujet est un ensemble d'exercices qui se veulent proches du cours ; il est destiné à ceux qui souhaitent évaluer leur apprentissage de OCaml.
- Le deuxième sujet élabore une réponse informatique à l'usage des ensembles ; il est adapté d'un sujet de concours de niveau raisonnable.
- Le troisième sujet propose un problème plus abstrait ; il est adapté d'un concours qui propose des sujets ambitieux.

Le choix naturel est celui des deux premiers sujets, ne choisissez les sujets 2 et 3 que si vous trouvez que le risque de la difficulté possible peut être équilibré par le plaisir de l'abstraction.
Chaque question vaudra (au moins) 2 points dans la note totale.

I Premier sujet : exercices

Dans ces exercices il sera demandé de répondre à une même question sur des objets implémentés sous forme de liste ou sous forme de tableaux. Il est demandé de répondre de manière adaptée au type de données.

Par exemple si le but est de calculer la somme des termes d'une liste, on peut écrire un algorithme récursif

```
let rec sommeL liste =
  match liste with
  |[] -> 0
  |t::q -> t + (somme q);;

sommeL : int list -> int
```

Par contre, si on veut la somme des termes d'un tableau d'entier, on écrira une boucle

```
let sommeT t =
  let n = Array.length t in
  let s = ref 0 in
  for i = 0 to (n-1) do
    s := !s + t.(i) done;
  !s;;

sommeT : int array -> int
```

Remarques

- *On voit que, dans le cas des tableaux, on sera proche de l'usage de Python.*
- *Les solutions attendues ne sont pas longues ; si votre solution commence à nécessiter plus de 10 lignes, il est probable qu'elle soit fausse ou incompréhensible. Dans le titre des exercices le nombre de lignes de la solution proposée dans le corrigé est indiqué à titre indicatif.*
- *Les questions comportent la signature attendue, c'est-à-dire le type des arguments et de la réponse. Les fonctions ci-dessus sont suivies de leur signature. Il est impératif de respecter cette consigne ; en particulier il ne faut pas convertir les listes en tableaux ou réciproquement.*
- *Sans que ce soit une interdiction, on évitera l'usage de références (`let x = ref ... in` dans les fonctions portant sur les listes. Bien entendu l'usage des références est indispensable dans le cas des tableaux.*

Question 1 - Appartenance à un tableau. Solution en 7 lignes
Écrire une fonction `appartientT : int -> int array -> bool` telle que `appartientT k t` renvoie `true` si et seulement si `k` est un des éléments du tableau `t`, c'est-à-dire s'il existe i tel que `t.(i) = k`.

Il n'est pas exigé d'optimiser la recherche en l'interrompant dès l'élément trouvé.
Indication : contrairement à Python, OCaml ne permet pas de sortir d'une boucle avec un `return` (qui n'existe pas en OCaml).

Question 2 - Appartenance à une liste. Solution en 5 lignes
Écrire une fonction `appartientL : int -> int list -> bool` telle que `appartientL k liste` renvoie `true` si et seulement si `k` est un des éléments de la liste.

Indication : on ne peut pas utiliser le motif `|k::q -> ...` pour tester si une liste commence par la valeur k, on doit utiliser la syntaxe `|t::q when k = t -> ...`.

Question 3 - Maximum d'un tableau. Solution en 7 lignes
Écrire une fonction `maximumT : int array -> int` telle que `maximumT t` renvoie la valeur maximale des éléments du tableau `t`.

On supposera, sans avoir besoin de le vérifier, que la longueur du tableau est au moins 1.

Question 4 - Maximum d'une liste. Solution en 6 lignes
Écrire une fonction `maximumL : int list -> int` telle que `maximumL t` renvoie la valeur maximale des éléments d'une liste.

Indication : le cas terminal de la récursivité n'est pas la liste vide mais une liste avec un unique élément. Pour l'exhaustivité du pattern-matching on renverra une erreur pour une liste vide.

```
|[] -> failwith "Un message qui explique l'erreur"
|[a] ->
|t:: q ->
```

Question 5 - Occurences dans un tableau. Solution en 7 lignes
Écrire une fonction `nombreOccT : int -> int array -> int` telle que `nombreOccT k t` renvoie le nombre d'occurences de `k` dans le tableau `t` c'est-à-dire le nombre d'apparitions (qui peut être 0) de `k` parmi les éléments de `t`.

Question 6 - Occurences dans une liste. Solution en 5 lignes
Écrire une fonction `nombreOccL : int -> int list -> int` telle que `nombreOccL k liste` renvoie le nombre d'occurences de `k` dans la liste.

Les deux fonctions suivantes ont le même but : repérer les éléments positifs d'une collection. Cependant le résultat sera donné sous une forme adaptée au type de données en entrée :

- on renvoie un tableau de booléen qui indique les indices pour lesquels `t.(i)` est positif dans le cas d'un tableau,
- on renvoie la liste des éléments positifs dans la cas d'une liste.

Question 7 - Filtrage dans un tableau. Solution en 7 lignes
Écrire une fonction `positifsT : int array -> bool array` telle que `positifsT t` renvoie un tableau de booléen `b` de même taille que `t` tel que `b.(i)` vaut `true` si et seulement si `t.(i)` est strictement positif.

`positifsT [|4; -6; -3; 7; 3; -2; 5|]` doit renvoyer `[|true; false; false; true; true; false; true|]`.
Indication : on doit initialiser (`Array.make`) le tableau avec une valeur booléenne (`true` ou `false`). On rappelle qu'on modifie un tableau avec `t.(i) <- valeur`.

Question 8 - Filtrage dans une liste. Solution en 5 lignes
Écrire une fonction `positifsL : int list -> int liste` telle que `positifsL liste` renvoie la liste des éléments strictement positifs de la liste.

`positifsT [4; -6; -3; 7; 3; -2; 5]` doit renvoyer `[|4; 7; 3; 5|]`

Question 9 - Croissance d'un tableau. Solution en 7 lignes
Écrire une fonction `croissantT : int array -> bool` telle que `acroissant t` renvoie `true` si et seulement si `t` est un tableau croissant, c'est-à-dire `t.(0) <= t.(1) <= ... <= t.(n-1)` si n est la longueur du tableau.

Il n'est pas exigé d'optimiser la recherche en l'interrompant dès que la croissance est contredite.
Indication : il n'y a que $n-1$ comparaisons à faire pour un tableau de taille n.

Question 10 - Croissance d'une liste. Solution en 6 lignes
Écrire une fonction `croissantL : int list -> bool` telle que `croissantL liste` renvoie `true` si et seulement si les éléments de la liste sont dans un ordre croissant.

Indication : on pourra utiliser un motif de la forme `a::b::q` pour repérer les deux premiers éléments d'une liste qui en comporte au moins 2.

II Deuxième sujet : ensembles finis, adapté de E3A 2012

On considérera des sous-ensembles finis d'entiers.
Un ensemble A sera représenté par une liste **strictement croissante** donc sans doublon.
Par exemple $A = \{4, 5, 3, -1, 3, 7\}$ est représenté par `[-1; 3; 4; 5; 7]`.

Les fonctions seront récursives, toute fonction qui emploie des tableaux sera refusée.

Question 11
Écrire une fonction `appartient` prenant en arguments un entier k et une liste représentant un ensemble A et qui renvoie `true` ou `false` selon que x appartient ou non à A.

```
appartient : int -> int list -> bool
appartient 7 [1; 4; 9] -> false
appartient 3 [2; 3; 11] -> true
```

N.B. La fonction est une amélioration de la fonction de la question 2 : on doit cesser de chercher l'élément quand il est plus petit que les termes de la liste, on doit utiliser la croissance de la liste.

Question 12
Écrire une fonction `add` prenant en arguments un entier k et une liste représentant un ensemble A et qui renvoie la liste représentant $A \cup \{x\}$.

```
add : int -> int list -> int list
add 7 [1; 4; 9] -> [1; 4; 7; 9]
add 3 [2; 3; 11] -> [2; 3; 11]
```

Question 13
Écrire une fonction `ensemble` prenant en arguments deux entiers a et b et qui renvoie la liste représentant $\{a, a+1, \ldots, b\}$.

```
ensemble: int -> int -> int list
ensemble (-2) 3 -> [-2; -1; 0; 1; 2; 3]
ensemble 4 3 -> []
```

Pour calculer l'union de deux ensembles on pourrait écrire la fonction suivante.

```
let rec union0 l1 l2 =
  match l1 with
  |[] -> l2
  |t1::q1 -> union1 q1 (add t1 l2);;
```

Question 14
Montrer que le nombre de comparaisons faites lors du calcul de l'union de deux ensembles A_1 et A_2 est majoré par $n_1.n_2$ où n_i est le cardinal de A_i.
Donner deux ensembles A_1 et A_2 de cardinal 3 pour lesquels on fait 9 comparaisons.

Question 15
Écrire une fonction `union` prenant en arguments deux listes représentant deux ensembles A et B et qui renvoie la liste représentant $A \cup B$. Cette fonction devra effectuer au plus n_1+n_2 comparaisons où n_1 et n_2 sont les cardinaux des ensembles représentés par les listes.

```
union : int list -> int list -> int list
union [1; 4; 7; 9] [1; 7; 8] -> [1; 4; 7; 8; 9]
```

Indication : on pourra comparer les têtes des listes dans le pattern-matching

```
let rec union l1 l2 =
  match l1, l2 with
  |[], _ -> ...
  |_, [] -> ...
  |t1::q1, t2::q2 when t1 < t2 -> t1::(union2 q1 t2)
  ...
```

Question 16
Écrire une fonction `intersection` prenant en arguments deux listes représentant deux ensembles A et B et qui renvoie la liste représentant $A \cap B$.

```
intersection : int list -> int list -> int list
intersection [1; 4; 7; 9] [1; 7; 8] -> [1; 7]
```

Question 17
Écrire une fonction `moins` prenant en arguments deux listes représentant deux ensembles A et B et qui renvoie la liste représentant $A \setminus B$, la différence ensembliste entre A et B.

```
moins : int list -> int list -> int list
moins [1; 4; 7; 9] [1; 7; 8] -> [4; 9]
```

On va considérer des familles d'ensembles, représentés par des listes de listes.

Question 18
Écrire une fonction `plusTete` telle que `plusTete k [l1; l2; ...]` renvoie `[k::l1; k::l2; ...]` où `l1`, `l2`, ...sont des listes d'entiers

```
plusTete : int -> int list list -> int list list
plusTete 0 [[1; 2]; [3]] -> [[0; 1; 2]; [0; 3]]
```

Pour conserver la structure croissante on n'emploiera cette fonction que lorsque k est strictement inférieur à tous les termes de toutes les listes `l1`, `l2`, ...

Question 19
Écrire une fonction `parties` telle que `parties a` renvoie la liste des parties représentant les sous-ensembles de l'ensemble A représenté par la liste `a`.

```
parties : int list -> int list list
parties [] -> [[]]
parties [2; 5] -> [[]; [5]; [2]; [2; 5]]
```

Indication : on pourra remarquer que les parties de $E = \{a\} \cup E'$ sont soit des parties de E', soit des parties de E' auxquelles on ajoute a.

Question 20
Écrire une fonction `combinaisons` telle que `combinaisons k a` renvoie la liste des parties représentant les sous-ensembles à k éléments de l'ensemble A représenté par la liste `a`.

```
combinaisons : int -> int list -> int list list
combinaisons 3 [4; 5] -> []
combinaisons 2 [-1; 4; 5] -> [[4; 5]; [-1; 5]; [-1; 4]]
```

Indication : on pourra remarquer que les parties de $E = \{a\} \cup E'$ à k éléments sont soit des parties à k éléments de E', soit des parties à $k-1$ éléments de E' auxquelles on ajoute a.

III Troisième sujet : librement adapté de X-ENS 2019

On s'intéresse dans ce sujet à la notion de **sous-suite** (suite extraite) parmi les suites finies.

Définition 1 - Sous-suite

Une suite $a = (a_0, a_1, \cdots, a_{m-1})$ est une sous-suite de $b = (b_0, b_1, \cdots, b_{n-1})$
s'il existe une suite strictement croissante d'entiers $0 \leq p_0 < p_1 < \cdots < p_{m-1} < n$
telle que $a_i = b_{p_i}$ pour tout $i \in \{0, 1, \ldots, m-1\}$.
On notera $a \preccurlyeq b$ si a est une sous-suite de b.
La suite $(p_0, p_1, \ldots, p_{m-1})$ peut être vue comme une application strictement croissante p, un **plongement**, de l'ensemble $\{0, 1, \ldots, m-1\}$ vers l'ensemble $\{0, 1, \ldots, n-1\}$.

Notons qu'il peut exister plusieurs façons différentes (ou aucune) de plonger a dans b.
Exemples :

- $(1, 2, 3)$ est une sous-suite de $(1, 5, 2, 3, 8, 2, 3)$ et il y a 3 plongements,
- $(1, 2, 3)$ n'est pas une sous-suite de $(1, 5, 3, 7, 8, 2)$,

Une liste sera représentée par une liste : $a = (4, 2, 7)$ est représentée par `a = [4; 2; 7]`.
On assimilera dans la suite du sujet suite et liste.

Définition 2 - Concaténation

Si $a = (a_0, a_1, \cdots, a_{m-1})$ et $b = (b_0, b_1, \cdots, b_{n-1})$ sont deux suite, leur concaténation est $a + b = (a_0, a_1, \cdots, a_{m-1}, b_0, b_1, \cdots, b_{n-1})$.
Cela correspond à `a @ b` pour les représentations en listes.
On notera aussi $k \oplus a$ à la place de $(k) + a$, la concaténation d'un élément à une suite, cela correspond à `k::a` pour les représentations en listes.

Cette dernière écriture permettra de définir une liste non vide $a = (a_0, a_1, \cdots, a_{m-1})$ sous la forme $a = a_0 \oplus a'$ avec $a' = (a_1, a_2, \cdots, a_{m-1}) = (a'_0, a'_1, \ldots, a'_{m-2})$.

Question 21 - Première étape
On considère deux suites non vides, $a = a_0 \oplus a'$ et $b = b_0 \oplus b'$.

- Si $a_0 = b_0$, prouver que $a \preccurlyeq b$ équivaut à $a' \preccurlyeq b'$.
- Si $a_0 \neq b_0$, à quelle condition sur a et b', a-t-on $a \preccurlyeq b$?

Question 22 - Fonction de test
En déduire une fonction `sousSuite : int list -> int list -> bool` telle que `sousSuite l1 l2` renvoie `true` si et seulement si `l1` est une sous-suite de `l2`.

On note $\binom{b}{a}$ le nombre de plongements de b dans a, de sorte que $a \preccurlyeq b$ si et seulement si $\binom{b}{a} > 0$.
Si on note ε la suite vide : $\varepsilon = ()$, on admet qu'on a $\binom{a}{\epsilon} = 1$ pour tout suite b (même vide).
On admet aussi qu'on $\binom{\epsilon}{a} = 0$ pour tout suite a non vide.

Question 23 - Exemples

- Calculer $\binom{b}{a}$ si $a = (1, 2)$ et $b = (1, 2, 1, 2)$.
- Calculer $\binom{b}{a}$ si $a = (1, 1, \ldots, 1)$ (m fois 1) et $b = (1, 1, \ldots, 1)$ (n fois 1)

Question 24 - Réduction
On considère deux suites non vides, $a = a_0 \oplus a'$ et $b = b_0 \oplus b'$.

- Si $a_0 = b_0$, prouver que $\binom{b}{a} = \binom{b'}{a} + \binom{b'}{a'}$.
- Si $a_0 \neq b_0$„ prouver que $\binom{b}{a} = \binom{b'}{a}$.

Question 25 - Comptage des plongements
En déduire une fonction `nbPlongements : int list -> int list -> int` telle que `nbPlongements l1 l2` renvoie le nombre de plongement de `l1` dans `l2`.

On cherche maintenant à dénombrer les sous-suites d'une suite a.
On note $\downarrow a$ pour $\{u \mid u \preccurlyeq a\}$. Par exemple $(1,2,1,2)$ a 12 sous-suites distinctes :
$\downarrow(1,2,1,2) = \{\epsilon, (1), (2), (1,1), 1,2), (2,1), (2,2), (1,1,2), (1,2,1), (1,2,2), (2,1,2), (1,2,1,2)\}$.

On généralise l'opérateur $\oplus$ à un ensemble :
si u_1, u_2, ...sont des suites, $k \oplus \{u_1, u_2, \ldots\} = \{k \oplus u1, k \oplus u_2, \ldots\}$

Question 26
Montrer que pour deux suite commençant par la même valeur $v = k \oplus v'$ et $w = k \oplus w'$,

$$\downarrow(v+w) \;= \downarrow(v'+w) \;\cup\; k \oplus \big(\downarrow(v'+w) \setminus \downarrow w'\big) \tag{1}$$

Question 27
Montrez que l'union $\downarrow(v'+w) \;\cup\; k \oplus \big(\downarrow(v'+w) \setminus \downarrow w'\big)$ est disjointe si et seulement si la suite v' ne contient pas la valeur k.

Quand l'union est disjointe dans (1), on peut obtenir $\mathrm{Card}\big(\downarrow(v+w)\big)$
en combinant $\mathrm{Card}\big(\downarrow(v'+w)\big)$ et $\mathrm{Card}(\downarrow w')$.

Question 28
Si $u = k \oplus u'$ avec la suite u' ne contenant pas la valeur k, déterminer $\mathrm{Card}(\downarrow u)$ en fonction de $\mathrm{Card}(\downarrow u')$.

Question 29
En se basant sur les résultats précédents, écrire une fonction `nbSousSuites : int list -> int` telle que `nbSousSuites liste` renvoie le nombre de sous-suites de la suite représentée par la liste.

Question 30
Écrire une fonction `sousSuites : int list -> int list list` telle que `sousSuites u` renvoie la liste des listes représentant les sous-suites de u.

Solutions

Solution de la question 1 - Appartenance à un tableau. Solution en 7 lignes

```
let appartientT k t =
  let n = Array.length t in
  let reponse = ref false in
  for i = 0 to (n-1) do
    if k = t.(i)
    then reponse := true done;
  !reponse;;
```

Si on veut cesser dès qu'on a trouvé avec une écriture un peu astucieuse

```
let appartientT k t =
  let n = Array.length t in
  let i = ref 0 in
  while !i < n && k <> t.(!i) do
    i := !i + 1 done; (* incr i *)
  !i < n;;
```

Solution de la question 2 - Appartenance à une liste. Solution en 5 lignes

```
let rec appartientL k liste
  match liste with
  |[] -> false
  |t::q when t = k -> true
  |t::q -> appartient k q;;
```

Solution de la question 3 - Maximum d'un tableau. Solution en 7 lignes

```
let maximumT t =
  let n = Array.length t in
  let reponse = ref t.(0) in
  for i = 1 to (n-1) do
    if t.(i) > !reponse
    then reponse := t.(i) done;
  !reponse;;
```

Solution de la question 4 - Maximum d'une liste. Solution en 6 lignes

```
let rec maximumL liste =
  match liste with
  |[] -> failwith "Pas de maximum dans une liste vide"
  |[a] -> a
  |t:: q -> let m = maximumL q in
            max t m;;
```

Solution de la question 5 - Occurences dans un tableau. Solution en 7 lignes

```
let nombreOccT k t =
  let n = Array.length t in
  let reponse = ref 0 in
  for i = 0 to (n-1) do
    if t.(i) = k
    then incr reponse done;
  !reponse;;
```

Solution de la question 6 - Occurences dans une liste. Solution en 5 lignes

```
let rec nombreOccT k liste =
  match liste with
  |[] -> 0
  |t:: q -> when t = k -> 1 + nombreOccL k q
  |t:: q -> nombreOccL k q;;
```

Solution de la question 7 - Filtrage dans un tableau. Solution en 7 lignes

```
let positifsT t =
  let n = Array.length t in
  let b = Array.make n false in
  for i = 0 to (n-1) do
    if t.(i) > 0
    then b.(i) <- true done;
  b;;
```

Solution de la question 8 - Filtrage dans une liste. Solution en 5 lignes

```
let rec positifsL liste =
  match liste with
  |[] -> []
  |t:: q -> when t > 0 -> t :: (positifsL  q)
  |t:: q -> positifsL  q;;
```

Solution de la question 9 - Croissance d'un tableau. Solution en 7 lignes

```
let croissantT t =
  let n = Array.length t in
  let reponse = ref true in
  for i = 0 to (n-2) do
    if t.(i) > t.(i+1)
    then reponse := false done;
  !reponse;;
```

Si on veut cesser dès qu'on sait que ce n'est pas croissant

```
let croissantT t =
  let n = Array.length t in
  let i = ref 0 in
  while !i < n-1 && t.(!i) t.(!i+1) do
    i := !i + 1 done;
  !i = n - 1;;
```

Solution de la question 10 - Croissance d'une liste. Solution en 6 lignes

```
let rec croissantL liste
  match liste with
  |[] -> true
  |[a] -> true
  |a::b::q when a > b -> false
  |a::b::q -> croissantL b::q;;
```

Solution de la question 11

```
let rec appartient x liste =
  match liste with
  |[] -> false
  |t::q when t = x -> true
  |t::q -> appartient x q;;
```

Solution de la question 12

```
let rec add x liste =
  match liste with
  |[] -> [x]
  |t::q when t = x -> liste
  |t::q when x < t -> x::liste
  |t::q -> t::(add x q);;
```

Solution de la question 13

```
let rec ensemble a b =
  if a > b
  then []
  else a :: (ensemble (a+1) b);;
```

Solution de la question 14

La fonction `add` peut comparer x à chaque élément de la liste ; c'est le cas quand x est supérieur à tous les élément de la liste.

Ainsi `union` peut comparer chaque élément de `l1` à chaque élément de `l2` ce qui fait $n_1.n_2$ comparaisons. Ce nombre est atteint quand tous les éléments de `l1` sont supérieurs aux éléments de `l2` : on peut choisir `l1 = [3; 4; 5]` et `l2 = [0; 1; 2]`.

Solution de la question 15

```
let rec union l1 l2 =
  match l1, l2 with
  |[], _ -> l2
  |_, [] -> l1
  |t1::q1, t2::q2 when t1 = t2 -> t1::(union2 q1 q2)
  |t1::q1, t2::q2 when t1 < t2 -> t1::(union2 q1 l2)
  |t1::q1, t2::q2 -> t2::(union2 l1 q2);;
```

Solution de la question 16

```
let rec intersection l1 l2 =
  match l1, l2 with
  |[], _ -> []
  |_, [] -> []
  |t1::q1, t2::q2 when t1 = t2 -> t1::(intersection q1 q2)
  |t1::q1, t2::q2 when t1 < t2 -> intersection q1 l2
  |t1::q1, t2::q2 -> intersection l1 q2;;
```

Solution de la question 17

```
let rec imoins l1 l2 =
  match l1, l2 with
  |[], _ -> []
  |_, [] -> l2
  |t1::q1, t2::q2 when t1 = t2 -> moins q1 q2
  |t1::q1, t2::q2 when t1 < t2 -> t1 :: (moins q1 l2)
  |t1::q1, t2::q2 -> moins l1 q2;;
```

Solution de la question 18

```
let rec plusTete k liste =
  match liste with
  |[] -> []
  |t::q -> (k :: t) :: (plusTete k q);;
```

Solution de la question 19

```
let rec parties liste =
  match liste with
  |[] -> [[]]
  |t::q -> let l = parties q in
           l @ (plusTete t l);;
```

Solution de la question 20

```
let rec combinaisons k liste =
  match k, liste with
  |0, _ -> [[]]
  |k, [] -> []
  |k, t::q -> (combinaisons k q) @  (plusTete t (combinaisons (k
     -1) q));;
```

Solution de la question 21 - Première étape
On note $a' = (a_1, \dots, a_{m-1}) = (a'_0, a'_1, \dots, a'_{m-2}) : a'_i = a_{i+1}$.
et $b' = (b_1, \dots, b_{n-1}) = (b'_0, b'_1, \dots, b'_{n-2}) : b'_i = b_{i+1}$.

Cas $a_0 = b_0$

- Si $a \preccurlyeq b$, on note p un plongement associé.
  On a $a_i = b_{p_i}$ pour tout i donc $a'_i = a_{i+1} = b_{p_{i+1}}$.
  Or $p_{i+1} \geqslant p_1 > p_0 \geqslant 0$ donc $p_{i+1} \geqslant 1$ puis $a'_i = b_{p_{i+1}-1+1} = b'_{p_{i+1}-1}$.
  Si on note $q_i = p_{i+1} - 1$, q définit un plongement de a' d ans b' ; on a bien $a' \preccurlyeq b'$.

- Si $a' \preccurlyeq b'$, on note q un plongement associé. La démonstration ci-dessus suggère de poser $p_i = q_{i-1} + 1$ pour $i \geqslant 1$: on aura $b_{p_i} = b_{q_{i-1}+1} = b'_{q_{i-1}} = a'_{i-1} = a_i$. On peut alors compléter par $p_0 = 0$ d'où $b_{p_0} = b_0 = a_0$ donc p est un plongement de a dans b ; $a \preccurlyeq b$.
- On a bien prouvé l'équivalence $a \preccurlyeq b \iff a' \preccurlyeq b'$ dans le cas $a_0 = b_0$.

Cas $a_0 \neq b_0$ Prouvons qu'on a alors $a \preccurlyeq b \iff a \preccurlyeq b'$.

- Si $a \preccurlyeq b'$, on note q un plongement associé. On a $a_i = b'_{q_i} = b_{q_i+1}$ pour tout i
  Ainsi, en posant $p_i = q_i + 1$, on a un plongement de a dans b : $a \preccurlyeq b$.
- Si $a \preccurlyeq b$, le plongement associé, p vérifie $b_{p_0} = a_0 \neq b_0$ donc $p_0 \geqslant 1$.
  En posant $q_i = p_i - 1$ on obtient $a_i = b_{p_i} = b'_{p_i-1} = b'_{q_i}$: q définit un plongement de a dans b' d'où $a \preccurlyeq b'$.

Solution de la question 22 - Fonction de test

```
let rec sousSuite a b =
  match a, b with
  |[], _ -> true  (* a  est vide *)
  |_, [] -> false (* Une suite non vide n'est pas sous-suite du
     vide *)
  |ta::qa, tb:: qb when ta = tb -> sousSuite qa qb
  |ta::qa, tb:: qb -> sousSuite a qb;;
```

Solution de la question 23 - Exemples

- Il y a 3 plongement de $(1,2)$ dans $(1,2,1,2)$: $(\underline{1},\underline{2},1,2)$, $(\underline{1},2,1,\underline{2})$ et $(1,2,\underline{1},\underline{2})$: $\binom{b}{a} = 3$.
- Pour $m > n$, il n'y a pas de plongement possible : $\binom{b}{a} = 0$.
  Pour $m \leqslant n$, un plongement est défini par les positions de ses images, c'est-à-dire par le choix de m indices parmi n, il y a $\binom{n}{m}$ plongements.

Solution de la question 24 - Réduction

Cas $a_0 \neq b_0$ Dans ce cas, pour tout plongement p de a dans b , on a $p_0 \neq 0$ donc $p_0 \geqslant 1$.
On peut, comme ci-dessus, définit $q_i = p_i + 1$ et q définit un plongement de a dans b'.
Réciproquement, tout plongement de a dans b' définit un plongement de a dans b.
Il a donc autant de plongement de a dans b que de plongement de a dans b' ; $\binom{b}{a} = \binom{b'}{a}$.

Cas $a_0 = b_0$ Il y a deux sortes de plongements de a dans b.

- Soit $p_0 = 0$ et alors $q_i = p_{i+1} - 1$ permet de définir un plongement de a' dans b'.
  On a vu, inversement, que tout plongement de a' dans b' se prolongeait en un plongement de a dans b avec $p_0 = 0$.
  Il a donc autant de plongement de a dans b tels que $p_0 = 0$ que de plongement de a' dans b'.
- Soit $p_0 \neq 0$; on a vu qu'alors il a donc autant de plongement de a dans b que de plongement de a dans b'.

Ainsi $\binom{b}{a} = \binom{b'}{a} + \binom{b'}{a'}$.

Solution de la question 25 - Comptage des plongements

```
let rec nbPlongements a b =
  match a, b with
  |[], _ -> 1
  |_, [] -> 0
  |ta::qa, tb:: qb when ta = tb
    -> nbPlongements a qb + nbPlongements qa qb
  |ta::qa, tb:: qb -> nbPlongements a qb;;
```

Solution de la question 26

Inclusion inverse On a $v+w = k \oplus (v'+w)$ donc toute sous-suite de $v'+w$ est une sous-suite de $v+w$: $\downarrow(v'+w) \subset \downarrow(v+w)$. Si on a $a \preccurlyeq b$ avec un plongement p alors $k \oplus a \preccurlyeq a \oplus b$ avec le plongement q défini par $q_0 = 0$ et $q_{i+1} = p_i + 1$. Ainsi on a $k \oplus (\downarrow b) \subset \downarrow(k \oplus b)$ d'où
$k \oplus \big(\downarrow(v'+w)\big) \subset \downarrow\big(k \oplus (v'+w)\big) = \downarrow\big((k \oplus v') + w\big) = \downarrow(v+w)$.
Comme on a $k \oplus \big(\downarrow(v'+w) \setminus \downarrow w'\big) \subset k \oplus \big(\downarrow(v'+w)\big)$ on aboutit à

$$\downarrow(v'+w) \ \cup \ k \oplus \big(\downarrow(v'+w) \setminus \downarrow w'\big) \ \subset \downarrow(v+w)$$

Inclusion directe Soit u une sous-suite de $v+w = k \oplus (v'+w)$.

- Si u ne commence pas par k, c'est en fait une sous-suite de $v'+w$: $u \in \downarrow(v'+w)$.
- Si u commence par k, on peut écrire $u = k \oplus u'$ et alors u' est une sous-suite de $v'+w$ donc $u \in k \oplus \big(\downarrow(v'+w)\big)$.
  Si on a $u \in k \oplus (\downarrow w')$ alors u' est une sous-suite de w' donc u est une sous-suite de w puis u est une sous-suite de $v'+w$. On en déduit que $u \in \downarrow(v'+w)$ ou
  $u \in k \oplus \big(\downarrow(v'+w)\big) \setminus k \oplus (\downarrow w') = k \oplus \big(\downarrow(v'+w) \setminus \downarrow w'\big)$.
- Dans tous les cas $u \in \downarrow(v'+w) \ \bigcup \ k \oplus \big(\downarrow(v'+w) \setminus \downarrow w'\big)$ d'où

$$\downarrow(v+w) \ \subset \downarrow(v'+w) \ \cup \ k \oplus \big(\downarrow(v'+w) \setminus \downarrow w'\big)$$

On a prouvé l'égalité.

Solution de la question 27
On va prouver la contraposée : v' contient k si et seulement si l'union n'est pas disjointe, c'est-à-dire si et seulement si $\downarrow(v'+w) \ \cap \ k \oplus \big(\downarrow(v'+w) \setminus \downarrow w'\big) \neq \emptyset$

- Si v' contient k, alors $u = k \oplus w$ est une sous-suite de $v'+w$: $u \in \downarrow(v'+w)$.
  De plus w est une sous-suite de $v'+w$ mais n'est pas une sous-suite de w' (en raison, par exemple, de la taille) donc $w \in \downarrow(v'+w) \setminus \downarrow w'$ puis $u = k \oplus w \in k \oplus \big(\downarrow(v'+w) \setminus \downarrow w'\big)$.
  Ainsi $k \oplus w \in \downarrow(v'+w) \ \cap \ k \oplus \big(\downarrow(v'+w) \setminus \downarrow w'\big)$, l'union n'est pas disjointe.
- Inversement, si $u \in \downarrow(v'+w) \ \cap \ k \oplus \big(\downarrow(v'+w) \setminus \downarrow w'\big)$ alors u est une sous-suite de $v'+w$ et $u = k \oplus u'$ avec u' sous-suite de $v'+w$ mais u' n'est pas une sous-suite de w'.
  Ainsi $u = k \oplus u'$ n'est pas une sous-suite de w mais c'est une sous-suite de $v'+w$ donc sa première valeur, k, se plonge dans v' : v' contient k.

On a prouvé l'équivalence.

Solution de la question 28
Si $u = k \oplus u'$, u' ne contenant pas k alors les sous-suites de u sont

- soit de la forme $v = k \oplus v'$ avec $v' \preccurlyeq u'$ et tout sous-suite de u' permet de définir, de manière unique une telle sous-suite ; il y en donc $\mathrm{Card}(\downarrow u')$,
- soit elle ne commence pas par k est c'est une des sous-suites de v' ; il y en a encore $\mathrm{Card}(\downarrow u')$.

On a donc $\mathrm{Card}(\downarrow u) = 2\mathrm{Card}(\downarrow u')$ dans ce cas.

L'union disjointe permet de calculer
$\mathrm{Card}\big(\downarrow(v+w)\big) = \mathrm{Card}\big(\downarrow(v'+w)\big) + \mathrm{Card}\big(k \oplus \big(\downarrow(v'+w) \setminus \downarrow w'\big)\big)$
Or $\mathrm{Card}\big(k \oplus \big(\downarrow(v'+w) \setminus \downarrow w'\big)\big) = \mathrm{Card}\big(\downarrow(v'+w) \setminus \downarrow w'\big) = \mathrm{Card}\big(\downarrow(v'+w)\big) - \mathrm{Card}(\downarrow w')$
car $\downarrow w'$ est inclus dans $\downarrow(v'+w)$ d'où $\mathrm{Card}\big(\downarrow(v+w)\big) = 2\mathrm{Card}\big(\downarrow(v'+w)\big) - \mathrm{Card}(\downarrow w')$ pour $v = k \oplus v'$, v' ne contenant pas k et $w = k \oplus w'$

Solution de la question 29

La première étape est de déterminer si la première valeur apparaît de nouveau et, si oui, de déterminer la décomposition $u = (k \oplus v') + (k \oplus w')$ v'ne contenant pas k utilisée ci-dessus.
On écrit donc une fonction `decoupage k liste` qui renvoie un booléen indiquant si k était présent dans la liste et la liste suivant la première apparition de k (la liste vide si k n'est pas présent).

```
let rec decoupage k liste =
  match liste with
  |[] -> false, []
  |t::q when t = k -> true, q
  |t::q-> decoupage k q;;

let rec nbSousSuites liste =
  match liste with
  |[] -> 1 (* La suite vide est une sous-suite de la suite vide *)
  |t::q -> let pres, w' = decoupage t q in
           if pres
           then 2*nbSousSuites q - nbSousSuites w'
           else 2*nbSousSuites q;;
```

Solution de la question 30

a. On a besoin de calculer une union disjointe : `@` convient,

b. d'ajouter une valeur devant tous les listes d'une liste de listes

```
let rec ajouter k liste =
  match liste with
  |[] -> []
  |t::q -> (k::t) :: (ajouter k q);;
```

c. de calculer une différence ensembliste (de listes non triées)

```
let rec dans k liste =
  match liste with
  |[] -> false
  |t::q when t = k -> true
  |t::q -> dans k q;;

let rec moins l1 l2 =
  match l1 with
  |[] -> []
  |t::q when dans t l2 -> moins q l2
  |t::q -> t :: (moins q l2);;
```

On peut alors écrire

```
let rec sousSuites liste =
  match liste with
  |[] -> [[]]
  |t::q -> let pres, w' = decoupage t q in
           let l = sousSuites q in
           if pres then let l2 = sousSuites w' in
                        l @ (ajouter t (moins l l2))
                   else l @ (ajouter t l);;
```