# Listes

## *Liste chainée en OCaml*

### Exercice 1 : Double

Écrire une fonction `double (u : 'a list) : 'a list` qui à une liste $[a_1; \dots; a_n]$ associe la liste $[a_1; a_1; \dots; a_n; a_n]$.

### Exercice 2 : Répète

1. Écrire une fonction `repete (u : 'a list) (n : int) : 'a list` qui renvoie la liste obtenue en concaténant $n$ listes $u$. On utilisera pour cela l'opérateur de concaténation `@`.

```
# repete [1; 2; 3] 4;;
- : int list = [1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3]
```

2. Après avoir rappelé la complexité de l'opérateur `@`, déterminer la complexité temporelle de cette fonction.

### Exercice 3 : Le dernier élément

1. (a) Écrire une fonction qui renvoie le dernier élément d'une liste, s'il existe, et une exception signalant que la liste est vide dans le cas contraire.
   (b) Quelle est la complexité temporelle de cette fonction ?
2. Faire de même avec une fonction renvoyant l'avant-dernier élément.

### Exercice 4 : Liste des préfixes

1. Écrire une fonction `prepend (x : 'a) (u : 'a list list) : 'a list list` qui ajoute $x$ en tête de chaque liste de la liste `u`. Quelle est la complexité temporelle de cette fonction ?
2. (a) En déduire une fonction `prefixes (u : 'a list) : 'a list list` calculant la liste de tous les préfixes de la liste `u`. Par exemple :

```
# prefixes [1; 2; 3; 4];;
- : int list list = [[1]; [1; 2]; [1; 2; 3]; [1; 2; 3; 4]]
```

   (b) Quelle est la complexité temporelle de `prefixes` ?

### Exercice 5 : Liste monotone

1. Écrire une fonction `est_croissante (u : 'a list) : bool` qui détermine si la liste u est croissante. Montrer que le nombre de comparaisons $C(u)$ effectuées par cette fonction vérifie $C(u) \leqslant |u|$.
2. (a) En déduire une fonction `est_monotone (u : 'a list) : bool` qui détermine si une liste est monotone. Majorer le nombre de comparaisons effectué par cette fonction.
   (b) On suppose qu'on dispose d'une fonction `compare (x : 'a) (y : 'a) : int` qui renvoie $-1$ si $x < y$, $0$ si $x = y$ et $1$ si $x > y$. Écrire une fonction `est_monotone (u : 'a list) : bool` ayant les mêmes spécifications que celle de la question précédente et telle que le nombre d'appels à `compare` est majoré par $|u|$.

### Exercice 6 : Partition

Écrire une fonction `partition (p : 'a -> bool) (u : 'a list) : ('a list * 'a list)` qui prend un prédicat $p$ et une liste $u$ et renvoie le couple $(v, w)$ où $v$ est la liste des éléments de $u$ vérifiant le prédicat et $w$ celle des éléments de $u$ ne le vérifiant pas. L'ordre des éléments devra être conservé.

```
# partition (fun x -> x > 0) [2; -1; -5; 4; 6; 0; -4];;
- : int list * int list = ([2; 4; 6], [-1; -5; 0; -4])
```

**Exercice 7 : Zip**

1. Écrire une fonction `zip (u : 'a list) (v : 'b list) : ('a * 'b) list` qui prend deux listes $[x_1; \dots; x_n]$ et $[y_1; \dots; y_n]$ et renvoie la liste $[(x_1, y_1); \dots; (x_n, y_n)]$. La fonction lèvera une exception si les listes ne sont pas de même longueur.

```
# zip [1; 2; 4] ["a"; "c"; "b"];;
- : (int * string) list = [(1, "a"); (2, "c"); (4, "b")]
# zip [1; 2; 4; 5] ["a"; "c"; "b"];;
Exception: Failure "Longueurs differentes"
```

2. Écrire une fonction `unzip` telle que `unzip (zip u v)` renvoie `(u, v)`.

```
# unzip (zip [1; 2; 4] ["a"; "c"; "b"]);;
- : int list * string list = ([1; 2; 4], ["a"; "c"; "b"])
```

**Exercice 8 : Sommes cumulées**

On souhaite écrire une fonction `sommes_cumulees (u : int list) : int list` qui renvoie

$$[x_1; x_1 + x_2; \dots; x_1 + \cdots + x_n]$$

lorsqu'on l'appelle sur $[x_1; \dots; x_n]$.

```
# sommes_cumulees [2; 3; 5; 31];;
- : int list = [2; 5; 10; 41]
```

1. (a) Écrire une fonction `ajoute (x : int) (u : int list) : int list` qui ajoute $x$ à tous les éléments de la liste $u$.
   (b) En déduire une première version de `sommes_cumulees`. Quelle est sa complexité temporelle ?
2. Écrire une version de `sommes_cumulees` dont la complexité temporelle est en $\Theta(|u|)$.

**Exercice 9 : Run-length encoding**

Le principe du *run-length encoding* est de remplacer, dans une liste, $k$ occurrences consécutives d'une même valeur $x$ par le couple $(x, k)$. C'est la méthode de compression la plus simple que l'on puisse imaginer, ce qui ne l'empêche pas d'être très utilisée même si elle est très rarement utilisée seule.

1. Écrire une fonction `compresse : 'a list -> ('a * int) list` réalisant la « compression ».
2. Écrire une fonction `decompresse : ('a * int list) -> 'a list` réalisant la « décompression ».

```
# let l = compresse ["a"; "b"; "b"; "b"; "a"; "c"; "c"; "d"];;
val l : (string * int) list =
[("a", 1); ("b", 3); ("a", 1); ("c", 2); ("d", 1)]
# decompresse l;;
- : string list = ["a"; "b"; "b"; "b"; "a"; "c"; "c"; "d"]
```

**Exercice 10 : Listes associatives**

Dans cet exercice, l'idée est de créer une *liste associative* représentant le fait qu'on associe à des valeurs $x_k$ de type `'a` des valeurs $y_k$ de type `'b`. Plus précisément, une liste associative est une liste de type `('a * 'b) list`.

1. Écrire une fonction `assoc (x : 'a) (lst : 'a * 'b) : 'b` qui pour une valeur $x$ et une liste

$$[(x_1, y_1); \dots; (x_n, y_n)]$$

renvoie un élément $y_k$ tel que $x_k = x$, et lève une exception `Not_Found` si un tel élément n'existe pas. *On peut lever l'exception avec la syntaxe `raise UneException`.*

2. Écrire une fonction `assoc_opt (x : 'a) (lst : 'a * 'b) : 'b option` qui a le même comportement, mais renvoie `Some yk` au lieu de $y_k$ et `None` sinon.

**Exercice 11 : Une élimination poussée**

On considère le processus suivant :
- On part d'une liste d'entiers.
- À chaque fois que l'on trouve deux éléments consécutifs égaux, on les supprime tous les deux. Si l'on crée ainsi de nouvelles opportunités de suppression, car deux éléments égaux se retrouvent désormais côte à côte, on itère le procédé.
- On s'arrête quand il n'y a plus d'éléments consécutifs égaux.

Il est à noter que le résultat final ne dépend pas de l'ordre dans lequel on choisit de faire les suppressions. Voici un exemple pour illustrer le processus, en partant deux fois de la même liste mais en faisant les éliminations dans un ordre différent.

- $[1; 2; 2; 3; 1; 3; 1; 4; 4; 2; 1; 1; 2; 1]$
- $[1; 3; 1; 3; 1; 4; 4; 2; 1; 1; 2; 1]$
- $[1; 3; 1; 3; 1; 2; 1; 1; 2; 1]$
- $[1; 3; 1; 3; 1; 2; 2; 1]$
- $[1; 3; 1; 3; 1; 1]$
- $[1; 3; 1; 3]$

- $[1; 2; 2; 3; 1; 3; 1; 4; 4; 2; 1; 1; 2; 1]$
- $[1; 2; 2; 3; 1; 3; 1; 2; 1; 1; 2; 1]$
- $[1; 2; 2; 3; 1; 3; 1; 2; 2; 1]$
- $[1; 2; 2; 3; 1; 3; 1; 1]$
- $[1; 2; 2; 3; 1; 3]$
- $[1; 3; 1; 3]$

Écrire une fonction `elimine (lst : 'a list) : 'a list` qui renvoie le résultat final, en essayant si possible d'être efficace.

**Exercice 12 : Produit cartésien**

Écrire une fonction `produit (u : 'a list) (v : 'b list) : ('a * 'b) list` qui calcule le « produit cartésien » de deux listes. Autrement dit, `produit u v` doit renvoyer la liste des couples $(x, y)$ où $x$ est un élément de `u` et $y$ un élément de `v`, l'ordre d'apparition des couples étant quelconque. D'éventuelles répétitions dans l'une des deux listes donneront des répétitions dans le produit. On pourra utiliser avantageusement un « map ».

```
# produit [1; 2; 3] ["a"; "b"; "c"];;
- : (int * string) list =
[(1, "a"); (1, "b"); (1, "c"); (2, "a"); (2, "b"); (2, "c"); (3, "a");
 (3, "b"); (3, "c")]
# produit [1; 2; 1] [12; 24];;
- : (int * int) list = [(1, 12); (1, 24); (2, 12); (2, 24); (1, 12); (1, 24)]
```

**Exercice 13 : Ranger $n$ objets identiques dans $m$ tiroirs**

Si vous disposez de $n$ objets identiques à ranger dans $m$ tiroirs, et que le nombre d'objets par tiroir n'est pas limité, une manière possible de ranger les objets correspond à un $m$-uplet d'entiers positifs ou nuls dont la somme vaut $n$ : la valeur de la première composante indique combien il y a d'objets dans le premier tiroir, la deuxième composante le nombre d'objets dans le deuxième tiroir et ainsi de suite. Par exemple, pour $n = 2$ et $m = 3$, on obtient $(2, 0, 0)$, $(1, 1, 0)$, $(1, 0, 1)$, $(0, 2, 0)$, $(0, 1, 1)$ et $(0, 0, 2)$.

1. Dans le cas général, combien existe-t-il de possibilités de rangement ?
2. On souhaite écrire une fonction `enumere (n : int) (m : int) : int list` renvoyant la liste de toutes ces possibilités.

```
# enumere 2 4;;
- : int list list =
[[0; 0; 0; 2]; [0; 0; 1; 1]; [0; 0; 2; 0]; [0; 1; 0; 1]; [0; 1; 1; 0];
[0; 2; 0; 0]; [1; 0; 0; 1]; [1; 0; 1; 0]; [1; 1; 0; 0]; [2; 0; 0; 0]]
```

   (a) Écrire une fonction `range (a : int) (b : int) : int list` telle que l'appel `range a b` renvoie la liste $[a; a + 1; \dots ; b - 1]$.

   (b) Écrire une fonction `flatten (u : 'a list list) : 'a list` renvoyant la concaténation des listes composant la liste $u$. *Cette fonction est disponible dans la bibliothèque standard sous le nom de* `List.flatten`.

   (c) Écrire enfin une implémentation récursive de la fonction demandée

   `enumere (n : int) (m : int) : int list`

   On pourra écrire une fonction interne `recurr (p : int) : int list` qui renvoie la liste de toutes les répartitions de $n$ objets dans $m$ tiroirs, contenant $p$ objets dans le premier tiroir.

**Exercice 14 : Index d'un livre**

On souhaite écrire une fonction `creer_index : (int * string list) list -> (string * int list) list` qui fonctionne comme sur l'exemple suivant :

```
# creer_index [
      (1, ["suite"; "fonction"]);
      (2, ["suite"; "ensemble"]);
      (3, ["suite"; "partition"; "fonction"])
  ];;
- : (string * int list) list =
[("fonction", [3; 1]); ("suite", [3; 2; 1]); ("ensemble", [2]); ("partition", [3])]
```

La chaine `"fonction"` apparait aux « pages » 1 et 3, `"suite"` apparait à chaque page, `"ensemble"` seulement en page 2. Chaque couple de type `int * string list` représente une page du livre et l'objet de type `(string * int list) list` renvoyé par `creer_index` représente l'index du livre.

1. Écrire une fonction

   `ajouter_mot (n : int) (index : (string * int list) list) (s : string)`
   `: (string * int list) list`

   Ajoutant le mot $s$ trouvé à la page $n$ dans l'index du livre.

2. En déduire une fonction

   `ajouter_page (index : (string * int list) list) (p : int * string list)`
   `: (string * int list) list`

   ajoutant la page $p$ à l'index passé en argument en utilisant avantageusement un « fold ». Définir enfin la fonction demandée `creer_index`.

*Il y a des solutions plus efficaces à ce problème classique, utilisant des structures de données plus sophistiquées que les listes. Nous aurons l'occasion d'y revenir, mais l'idée ici est simplement de s'entrainer à la manipulation de listes en OCaml.*

**Exercice 15 : Éléments répétés**

1. Écrire une fonction `est_sans_doublons_triee (u : 'a list) : bool` qui prend en entrée une liste $u$ supposée triée (dans l'ordre croissant ou décroissant) et renvoie `true` si et seulement si les éléments de $u$ sont deux à deux distincts.
2. Écrire une fonction `elimine_doublons_triee (u : 'a list) : 'a list` qui prend en entrée une liste $u$ supposée triée et renvoie la liste des éléments distincts de $u$.

```
# elimine_doublons [1; 1; 2; 3; 3; 3; 4; 5; 5; 6; 6; 6; 7];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
```

3. Écrire une fonction `sans_doublons (u : 'a list) : bool` qui prend en entrée une liste `u` et détermine si ses éléments sont deux à deux distincts. On pourra utiliser la fonction `List.mem`.
4. Écrire une fonction `elimine_doublons (u : 'a list) -> 'a list` qui prend en entrée une liste $u$ et renvoie la liste des éléments distincts de $u$, dans l'ordre de leur dernière apparition dans $u$.

```
# elimine_doublons [1; 5; 2; 5; 3; 3; 7; 3; 7; 3; 1];;
- : int list = [2; 5; 7; 3; 1]
```

5. Combien de tests d'égalité la fonction `sans_doublons_triee` effectuera-t-elle au maximum si on l'appelle sur une liste de longueur $n$ ? Préciser dans quel cas ce maximum est atteint.
6. Mêmes questions pour la fonction `sans_doublons`.

## *Liste chainée en C*

**Exercice 16 : Le lièvre et la tortue**

1. Implémenter en C le type liste chainée permettant de stocker des valeurs entières. On devra pouvoir l'utiliser de la façon suivante :

```c
// Création de la liste [ 1; 2; 3 ]
cell *lst1 = cons(1, cons(2, cons(3, NULL)));
// Accès à l'élément suivant
cell *lst2 = lst1->next;
// Accès à la valeur
printf("%d", lst2->value);
```

2. Implémenter une fonction `void delete (cell *lst)` permettant de libérer la totalité de la mémoire allouée par la liste `lst`.

Rien n'empêche de modifier un champ `next` pour le faire revenir sur un autre chainon. Par exemple :

```c
cell *lst3 = lst2->next;
// lst2 pointe vers l'élément "2", lst3 pointe vers l'élément "3"
lst3->next = lst2;
```

3. Représenter par un schéma l'état de la mémoire.
4. Est-ce que votre fonction de libération de la mémoire fonctionne sur cette liste ?

On va s'intéresser à la détection de cycle dans une liste chainée. On veut savoir si une liste est ou non cyclique à partir d'un certain rang. Pour cela, on veut écrire une fonction `bool is_cyclic(cell *lst)` qui renvoie `true` si un cycle est détecté, et `false` sinon. On va utiliser l'algorithme du lièvre et de la tortue. L'idée est la suivante : on parcourt la liste avec deux pointeurs. À chaque itération, le premier pointeur avance d'un élément (c'est la tortue), et le second pointeur avance de deux éléments à la fois (c'est le lièvre). Si le lièvre atteint la fin de la liste (`NULL`), il n'y a pas de cycle. Si la tortue rattrape le lièvre après le départ, alors il y a un cycle.

5. Illustrer une exécution de l'algorithme sur un exemple.
6. Implémenter cet algorithme en C.
7. Justifier la correction et la terminaison de cet algorithme.

## *Tableau, tableau redimensionnable*

### Exercice 17 : Exercice

Écrire une fonction `appartient : 'a -> 'a array -> bool` telle que `appartient x t` renvoie `true` si et seulement si l'élément `x` apparaît dans le tableau `t`.

### Exercice 18 : Exercice

Écrire une fonction `croissant : 'a array -> bool` qui détermine si le tableau passé en argument est croissant (au sens large). Si possible, on arrêtera le parcours dès que la réponse est connue.

### Exercice 19 : Exercice

Pour une séquence $s = s_0, \dots, s_{n-1}$, on définit le *miroir* de $s$ par $\overline{s} = s_{n-1}, \dots, s_0$. Une séquence est un *palindrome* si $s = \overline{s}$. Écrire une fonction `est_palindrome : 'a array -> bool` qui détermine si le tableau passé en argument est un palindrome. On essaiera de limiter au maximum le nombre de comparaisons effectuées.

```ocaml
# est_palindrome [|2; 1; 1; 3; 1; 1; 2|];;
- : bool = true
# est_palindrome [|2; 1; 1; 3; 3; 1; 1; 2|];;
- : bool = true
# est_palindrome [|2; 1; 1; 3; 3; 1; 2; 2|];;
- : bool = false
```

### Exercice 20 : Exercice

Écrire une fonction `swap : 'a array -> int -> int -> unit` telle que, après l'appel `swap t i j`, les éléments de `t` d'indices `i` et `j` aient été échangés.

**Exercice 21 : Exercice**

Écrire une fonction `renverse : 'a array -> unit` telle que, si l'on a initialement `t = [| t_0; ...; t_{k-1} |]`, alors, après l'appel `renverse t`, on a `[| t_{k-1}; ...; t_0 |]`. On travaillera « en place » sans créer de tableau auxiliaire, et l'on fera bien attention à ce que la fonction soit correcte indépendamment de la parité de la longueur du tableau considéré.

```
# let t = [|4; 1; 2; 5|];;
val t : int array = [|4; 1; 2; 5|]
# renverse t;;
- : unit = ()
# t;;
- : int array = [|5; 2; 1; 4|]
```

**Exercice 22 : Exercice**

1. Écrire une fonction `cherche_somme (t : int array) (s : int) : (int * int) option` ayant la spécification suivante :
   — si la fonction renvoie `Some (i, j)`, alors on a $0 \leqslant i \leqslant j <$ t et `t.(i) + t.(j) = s` ;
   — si la fonction renvoie `None`, alors il n'existe pas de couple $(i, j)$ vérifiant $0 \leqslant i \leqslant j <$ t et `t.(i) + t.(j) = s`.

```
# cherche_somme [|15; 1; 3; 5; 6; 7; 10; 1; 8|] 11;;
- : (int * int) option = Some (6, 7)
(* Some (1, 6) ou Some (2, 8) conviennent également. *)

# cherche_somme [|15; 1; 3; 5; 6; 7; 10; 1; 8|] 14;;
- : (int * int) option = Some (5, 5)

# cherche_somme [|15; 1; 3; 5; 6; 7; 10; 1; 8|] 19;;
- : (int * int) option = None
```

2. On suppose maintenant que le tableau `t` est trié par ordre croissant. Écrire une fonction `cherche_somme_croissant` ayant la même spécification que `cherche_somme` mais de complexité linéaire en la taille du tableau.

```
# cherche_somme_croissant [|1; 1; 3; 5; 6; 7; 7; 10; 12; 15|] 13;;
- : (int * int) option = Some (0, 8)
# cherche_somme_croissant [|1; 1; 3; 5; 6; 7; 7; 10; 12; 15|] 17;;
- : (int * int) option = Some (3, 8)
```

**Exercice 23 : Exercice**

Écrire une fonction `indices_mini (t : 'a array) : int list` renvoyant la liste des indices $i$ tels que `t.(i)` $=$ min `t`. On commencera par écrire la version la plus naturelle puis on pourra essayer d'écrire une version ne faisant qu'un seul parcours du tableau `t`.

```
# indices_mini [|10; 2; 7; 1; 7; 1; 3; 7|];;
- : int list = [3; 5]
```

**Exercice 24 : Exercice**

Écrire une fonction `filtre (pred : 'a -> bool) (t : 'a array) -> 'a list` qui renvoie la liste des `t.(i)` tels que `pred t.(i) = true`, classée dans l'ordre des $i$ croissants.

```
# filtre (fun n -> n mod 2 = 0) [|1; 4; 2; 5; 7; 8; 7; 10; 4; 5|];;
- : int list = [4; 2; 8; 10; 4]
```

**Exercice 25 : Section équilibrée**

On considère un tableau de booléens $t$ de longueur $n$. Pour $0 \leqslant a \leqslant b \leqslant n$, on note $t[a, b[$ la séquence $t_a, \ldots, t_{b-1}$ (elle est vide si $a = b$). La longueur de $t[a, b[$ est définie comme $b - a$. On définit alors

— $\text{vrai}(a, b)$ comme le nombre de `true` dans la section $t[a, b[$

$$\text{vrai}(a, b) = \text{Card}\{i \in [\![a, b[\![ \mid \texttt{t.(i) = true}\}.$$

— $\text{faux}(a, b)$ comme le nombre de `false` dans la section $t[a, b[$

$$\text{faux}(a, b) = \text{Card}\{i \in [\![a, b[\![ \mid \texttt{t.(i) = false}\}.$$

— $\delta(a, b)$ comme l'écart entre les deux

$$\delta(a, b) = \text{vrai}(a, b) - \text{faux}(a, b).$$

Une section $t[a, b[$ sera dite *équilibrée* lorsque $\delta(a, b) = 0$, c'est-à-dire lorsqu'elle contient autant de `true` que de `false`. Le but de l'exercice est de déterminer efficacement la longueur maximale d'une section équilibrée d'un tableau $t$.

1. Écrire une fonction `est_equilibree (t : bool array) (a : int) (b : int) : bool` qui détermine si la section $t[a, b[$ est équilibrée.
2. Écrire une fonction `max_equilibree (t : bool array) : int` qui calcule la longueur maximale d'une section équilibrée de `t` en testant toutes les possibilités. Quelle est la complexité de cette fonction ?
3. Écrire une fonction `max_depuis (t : bool array) (a : int) : int` qui renvoie la longueur maximale d'une section équilibrée de $t$ de la forme $t[a, b[$ (où $a$ est fixé car donné en argument). Cette fonction doit avoir une complexité linéaire en `t`.
4. En déduire une version plus efficace de `max_equilibree` et déterminer sa complexité.
5. Trouver une relation entre $\delta(a, b)$, $\delta(0, a)$ et $\delta(0, b)$ et en déduire une version de `max_equilibree` de complexité linéaire en la taille de $t$.