# Exercice 1 : Filtrages sur les listes

**Q. 1** Définir la fonction de signature `len_comp_3 : 'a list -> int` qui donne

| | |
|---|---|
| 1 | si `xs` contient au moins 3 éléments |
| 0 | si `xs` contient exactement 3 éléments |
| -1 | sinon. |

**Q. 2** Définir la fonction de signature `swap_hd_snd : 'a list -> 'a list` qui donne la liste obtenue en inversant l'ordre des deux premiers éléments de `xs`. La fonction `swap_hd_snd` renvoie `xs` inchangée si elle contient moins de 2 éléments.

Exemples :
- `(swap_hd_snd [1;2;3;4])` donne la liste `[2;1;3;4]`

**Q. 3** Définir la fonction de signature `swap_hd_fst (xs:('a*'a) list) : ('a*'a) list` qui donne la liste obtenue en inversant l'ordre des valeurs dans le premier élément de `xs`. On aura que `(swap_hd_fst [])` = `[]`.

Exemples :
- `(swap_hd_fst [ (1,2); (3,4); (5,5) ])` donne la liste `[ (2,1); (3,4); (5,5) ]`

# Exercice 2 : Constructions de listes avec récursion sur entiers

**Q. 1** Définir la fonction `repeat : int -> 'a -> ('a list)` telle que `(repeat n x)` donne la liste contenant `n` fois `x`.

Exemples :
- `(repeat 7 true)` donne `[true; true; true; true; true; true; true]`
- `(repeat 0 "Hello")` donne `[]`
- `(repeat (-42) [1;2;3])` donne `[]`

**Q. 2** Définir la fonction `range : int -> int -> (int list)` telle que `(range i j)` donne la liste `[i; i+1; ..; j]`. Remarque : si `i` $>$ `j` alors `(range i j)` donne la liste vide.

**Q. 3** Variante de la fonction précédente : définir la fonction `range_bis : int -> int -> (int list)` telle que `(range_bis x n)` donne la liste `[x; x+1; ..:x+n-1]`. Si `n` $\leq 0$ alors `(range x n)` donne la liste vide.

## Exercice 3 : Manipulation de listes avec récursion sur listes

**Q. 1** Définir la fonction `double : ('a list) -> ('a list)` qui double la taille d'une liste en dupliquant chacun de ses éléments.

Exemples :
> `(double [])` donne `[]`
> `(double [1;2;3])` donne `[1;1;2;2;3;3]`

**Q. 2** Définir la fonction `produit : (int list) -> int` qui donne le produit des éléments de la liste passée en argument. Remarque : `(produit [])` donne `1`.

**Q. 3** Donnez une définition récursive terminale de `produit`.

**Q. 4** Définir la fonction `flatten : (('a list) list) -> ('a list)` qui concatène toutes les listes de son argument. Schématiquement, `(flatten [ [x1; ...; xn]; ..; [y1; ..; ym] ])` donne la liste `[x1; ...; xn] .. [y1; ..; ym]`, c-à-d, `[x1; ..; xn; ..; y1; ..;ym]`.

Exemples :
> `(flatten [])` donne `[]`
> `(flatten [[]])` donne `[]`
> `(flatten [[1;2;3]; []; [4;5;6;7]])` donne `[1;2;3;4;5;6;7]`

**Q. 5** Écrire une fonction `dernier : 'a list -> 'a` qui renvoie le dernier élément de la liste en argument. La fonction devra, le cas échéant, lever une exception.

**Q. 6** Définir une fonction `rm_doub : 'a list -> 'a list` supprimant les doublons de la liste passée en argument.

## Exercice 4 : Extraction de sous-listes

**Q. 1** Définir la fonction `drop : int -> ('a list) -> ('a list)` telle que `(drop n xs)` donne la liste obtenue en privant `xs` de ses `n` premiers éléments. Si `n` $\leq 0$, le résultat est `xs` ; si `n` est plus grand que la taille de `xs`, le résultat est la liste vide.

**Q. 2** Définir la fonction `take : int -> ('a list) -> ('a list)` telle que `(take n xs)` donne la liste des `n` premiers éléments de `xs`. Que faire si `n` est négatif ou plus grand que la taille de la liste ?

**Q. 3** Déduire de ce qui précède la définition de la fonction `sub : int -> int -> ('a list) -> ('a list)` telle que `(sub start len xs)` extrait la sous liste de `xs` de longueur `len` qui commence à la position `start`. On aura que `(sub 0 (List.length xs) xs)`=`xs`.

## Exercice 5 : Sous-listes lacunaires, listes sous-jacentes

On appellera *sous-listes pleines* les sous listes calculées à l'exercice précédent. On s'intéresse à différentes notions de sous listes dans cet exercice.

**Q. 1** Définir une fonction `sous_listes: 'a list -> 'a list -> bool` permettant de tester si une liste est une sous-liste pleine d'une autre.

On dit qu'une liste `xs` est *sous-liste lacunaire* d'une liste `ys` lorsque tous les éléments de `xs` sont présents dans `ys` et qu'ils ont le même ordre dans les deux listes. Par exemple `[3; 13; 23]` est sous-liste lacunaire de `[2; 3; 5; 7; 11; 13; 17; 19; 23; 27]`. On peut avoir des répétitions, par exemple `[1; 1]` est sous-liste lacunaire de `[1; 0; 0; 1; 0]`. La liste vide est sous-liste lacunaire de toute liste. La liste vide n'admet que la liste vide comme sous liste lacunaire.

**Q. 2** Définir la fonction de signature `sublac 'a list -> 'a list -> bool` qui donne `true` si et seulement si son premier argument est une sous-liste lacunaire de son second.

On dit qu'une liste `xs` est *liste sous-jacente pour* $y$ de `ys` si on peut retrouver `ys` à partir de `xs` en remplaçant dans `xs` les $y$ par des éléments de `ys`. Par exemple `[`$y$`; 3; `$y$`; `$y$`; `$y$`; 13; `$y$`; `$y$`; 23; `$y$`]` est une liste sous-jacente de `[2; 3; 5; 7; 11; 13; 17; 19; 23; 27]`. Contre-exemple : `[3; `$y$`; `$y$`; 13; `$y$`; 23; `$y$`]` n'est pas une liste sous-jacente de `[2; 3; 5; 7; 11; 13; 17; 19; 23; 27]` : il manque un $y$ au début et un autre entre `3` et `13`.
Notez qu'une liste et sa liste-sous-jacente ont la même taille.

**Q. 3** Définir la fonction de signature `sublying (xs:'a list) (ys:'a list) (y: 'a): bool` qui donne `true` si et seulement si `xs` est liste sous-jacente de `ys` pour `y`.

**Q. 4** Définir la fonction de signature `stretch (xs:'a list) (ys:'a list) (y: 'a): 'a list` qui donne la liste sous-jacente de `ys` construite à partir de `xs`. Si cela n'est pas possible, c'est-à-dire, si `xs` n'est pas une sous-liste lacunaire de `ys`, la fonction `stretch` déclenche l'exception `Invalid_argument "stretch"`.

Exemples :

> `(stretch ([3; 5; 7]) ([1; 2; 3; 4; 5; 6; 7; 8; 9]) 0)` donne la liste `[0; 0; 3; 0; 5; 0; 7; 0; 0]`
> `(stretch ([3; 42; 7]) ([1; 2; 3; 4; 5; 6; 7; 8; 9]) 0)` déclenche `Invalid_argument "stretch"`.

Il est inutile d'utiliser `sublac` pour définir `stretch`.

## Exercice 6 : Cartes à jouer

Deux joueurs s'affrontent dans un (mauvais) jeu de cartes (imaginaire) dans lequel ils doivent comparer les cartes qu'ils ont en main. Une main correspond à un ensemble de 5 cartes qui valent chacune un certain nombre de points. Le joueur vainqueur est celui qui a le plus de points.
Les points sont calculés de la façon suivante : Un as vaut 150 points, Un roi vaut 100 points, Une dame vaut 80 points, Un valet vaut 50 points, Une carte avec un numéro vaut la valeur affichée sur la carte. Les cartes rouges font perdre leur valeur (ex : le dix de coeur fait perdre 10 points), les cartes noires font gagner leur valeur (ex : l'as de trèfle fait gagner 150 points).

**Q. 1** Définissez le type `card` (et les autres types utiles) pour représenter les cartes à jouer.

**Q. 2** Définissez la fonction `points_of_card : card -> int` qui calcule le nombre de points associé à chaque carte.

**Q. 3** Définissez la fonction `points_of_hand : card list -> int` qui calcule le nombre de points associé à une main.

**Q. 4** Définissez la fonction `compare_mains : card list -> card list -> int` qui, pour deux mains données, calcule un entier : $-1$ si la première main est moins forte que la deuxième, $0$ si elle est équivalente et $1$ sinon.

## Exercice 7 : Type option

Il est parfois nécessaire de pouvoir représenter l'absence de résultat pour une fonction. Par exemple la fonction de division de deux entiers n'est pas définie lorsque le second argument est nul. On est donc invité à définir un type étant :

— Soit rien, `None`
— Soit un élément `Some(x)`

Un tel type est défini au moyen du type :

```
type 'a option =
  | None
  | Some of 'a
```

**Q. 1** Définir une fonction `div : int -> int -> int option` calculant lorsque cela est possible la division des deux éléments qui lui sont passés en arguments et `None` sinon.

**Q. 2** Définir une fonction `pop : 'a list -> 'a option` prenant en argument une liste liste et s'évaluant en son premier élément s'il existe et en `None` sinon.

**Q. 3** Définir une fonction `map : ('a -> 'a option) -> 'a option -> 'a option` prenant en arguments une fonction `f` et une option `x` et s'évaluant à `None` si `x` ou `f x` est `None` et à `Some (f x)` sinon.

## Exercice 8 : Entiers naturels

Le type des entiers naturels de *Peano* est défini de la façon suivante : un entier (de *Peano*) est :
– soit *zéro*, représenté par `Z`
– soit le *successeur* `S(n)` d'un entier de *Peano* `n`.

**Q. 1** Définir le type `nat` des entiers naturels de Peano.

**Q. 2** Définir deux fonctions `nat_of_int` et `int_of_nat` permettant de convertir les entiers de *Peano* en entiers OCaml, et *vice-versa*.

**Q. 3** Définir deux fonctions `nat_add` et `nat_mult` permettant de calculer la somme et le produit de deux entiers de *Peano*. **On n'utilisera pas les fonctions `nat_of_int` et `nat_of_int`**

**Q. 4** Définir la fonction `nat_prec:nat -> nat option` retournant le prédécesseur d'un entier de *Peano*. Bien sûr, le zéro n'a pas de prédécesseur, la fonction est donc bien partielle.

**Q. 5** Définir une fonction `safe_int_of_nat: nat option -> int` qui convertit un entier de *Peano* si celui-ci est défini, mais qui lance une exception si l'entier n'est pas défini.

## Exercice 9 : Exercices complémentaires

**Q. 1** Donner une fonction `pack: 'a list -> 'a list list` qui rassemble les occurrences consécutives identiques d'une liste en une sous liste.

Exemples :
```
pack [1; 1; 1; 2; 2; 3; 7; 7; 1; 1] = [[1; 1; 1]; [2; 2]; [3]; [7; 7]; [1; 1]]
```

**Q. 2** Donner une fonction `slice: 'a list -> int -> int -> int -> 'a list` telle que `slice l d f o` calcule la sous-liste de `l` formée des éléments dont l'indice dans la liste est dans l'ensemble

$$\{\mathsf{d} + i \times \mathsf{o} \mid i \in \mathbb{N} \text{ et } \mathsf{d} + i \times \mathsf{o} < \mathsf{f}\}$$

Exemples :
```
slice ['a';'b';'c';'d';'e';'f';'g';'h';'i';'j'] 3 7 2 = ['d'; 'f']
```

**Q. 3** Donner une fonction `rotate: 'a list -> int -> 'a list` telle que `rotate l n` est la liste obtenue par rotation de `n` places sur la gauche de `l`.

Exemples :

```
rotate [1; 2; 3; 4; 5; 6] 3 = [4; 5; 6; 1; 2; 3]
rotate [1; 2; 3; 4; 5; 6] (-1) = [6; 1; 2; 3; 4; 5]
```

**Q. 4** Donner une fonction `rand_select: (l: 'a list) (n: int): 'a list` qui tire uniformément et sans remise `n` éléments de la liste `l`.

Exemples :

```
rand_select [1; 2; 3; 4; 5; 6] 3 = [6; 2; 1]
rand_select [1; 2; 3; 4; 5; 6] 3 = [5; 1; 4]
```

**Q. 5** Même question mais on s'efforcera de ne traverser qu'une fois la liste

**Q. 6** Donner une fonction `permutation: 'a list -> 'a list list` qui étant donnée une liste `l` produit la liste de toutes les permutations de `l`.

Exemples :

```
permutation [1; 2; 3] = [[1;2;3]; [1;3;2]; [2;1;3]; [2;3;1]; [3;1;2];
[3;2;1]]
```

**Q. 7** Même question mais en récursif terminal.