List Toolbox

Les fonctions de ce TP seront nécessaires pour les deux prochains sujets !

Ne pas utiliser l'opérateur @ ni les fonctions du module `List`.

Par contre, n'hésitez pas à réutiliser les fonctions écrites...

Un seul fichier à rendre (**et à conserver !**) : `list_tools.ml`.

1 Basics

1.1 `length`

Écrire une fonction qui calcule la longueur d'une liste.

```
val length : 'a list -> int = <fun>

# length [0; 1; 0; 1; 0; 0; 0; 0; 1; 1] ;;
- : int = 10
```

1.2 `product`

Écrire la fonction `product` qui calcule le produit de tous les éléments d'une liste d'entiers.

```
val product : int list -> int = <fun>

# product [1; 2; 3; 4; 5] ;;
- : int = 120
# product [];;
- : int = 1
```

1.3 `nth`

Écrire une fonction qui donne la valeur du $i^{ème}$ élément d'une liste. La fonction devra déclencher une exception `Invalid_argument` si i est négatif, ou une exception `Failure` si la liste est trop courte.

```
val nth : int -> 'a list -> 'a = <fun>

# nth 5 [1; 2; 3; 4; 5] ;;
Exception: Failure "nth: list is too short".
# nth 0 ['a'; 'b'; 'c'] ;;
- : char = 'a'
# nth (-5) [] ;;
Exception: Invalid_argument "nth: index must be a natural".
```

1.4 `search_pos`

Écrire une fonction qui retourne la position d'un élément dans une liste.

```
val search_pos : 'a -> 'a list -> int = <fun>

# search_pos 0 [1; 5; -1; 0; 8; 0] ;;
- : int = 3
# search_pos 'z' ['r'; 'h'; 'j'; 'o'] ;;
Exception: Failure "search_pos: not found".
```

1.5 int_of_bigint

On peut représenter un entier naturel comme étant une liste de chiffres, commençant par le chiffre des unités, puis celui des dizaines...
Par exemple l'entier `987654321` sera représenté par la liste : `[1; 2; 3; 4; 5; 6; 7; 8; 9]`. Dans la suite cette représentation sera appelée *entier long*.

Écrire la fonction `int_of_bigint` qui convertit un *entier long* en entier.

```
val int_of_bigint : int list -> int = <fun>

# int_of_bigint [0; 1; 2; 3; 4; 5; 6; 7; 8; 9] ;;
- : int = 9876543210
# int_of_bigint [] ;;
- : int = 0
```

1.6 prefix

Écrire une fonction `prefix` qui prend en paramètre un couple de listes et retourne *true* si l'une des listes est préfixe de l'autre, *false* sinon.

```
val prefix : 'a list * 'a list -> bool = <fun>

# prefix ([1; 2; 3], [1; 2]) ;;
- : bool = true
# prefix ([1; 2], [1; 2; 3; 4]) ;;
- : bool = true
# prefix ([1; 2], [1; 3; 4]) ;;
- : bool = false
# prefix ([1; 2; 4], [1; 2; 3; 4; 5]) ;;
- : bool = false
# prefix ([], [1]) ;;
- : bool = true
```

2 Construire - Modifier

2.1 init_list

Écrire la fonction `init_list n x` qui crée une liste de `n` valeurs `x`.

```
val init_list : int -> 'a -> 'a list = <fun>

# init_list 5 0 ;;
- : int list = [0; 0; 0; 0; 0]
# init_list 0 'a' ;;
- : char list = []
# init_list (-5) 1.5 ;;
Exception: Invalid_argument "init_list: n must be a natural".
```

2.2 put_list

Écrire une fonction `put_list` *v i list* qui remplace la valeur en position *i* dans *list* par *v* si possible.

```
val put_list : 'a -> int -> 'a list -> 'a list = <fun>

# put_list 'x' 3 ['-'; '-'; '-'; '-'; '-'; '-'] ;;
- : char list = ['-'; '-'; '-'; 'x'; '-'; '-']
# put_list 0 10 [1; 1; 1; 1] ;;
- : int list = [1; 1; 1; 1]
```

2.3 bigint_of_int (see 1.5)

Écrire la fonction `bigint_of_int` qui convertit un entier naturel en sa représentation sous forme d'*entier long* (de liste de chiffres).

```
val bigint_of_int : int -> int list = <fun>

# bigint_of_int 9876543210 ;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
# bigint_of_int 0 ;;
- : int list = []
# bigint_of_int (-12) ;;
Exception: Invalid_argument "bigint_of_int: not a natural".
```

3 'a list list

Dans la suite nous appellerons *matrice* (`board`) une liste de listes avec toutes les sous-listes de longueurs identiques.

Pour toutes les prochaines fonctions, voir les exemples d'applications page suivante !

3.1 init_board

Écrire la fonction `init_board` $(l,\ c)$ *val* qui retourne une *matrice* de taille $l \times c$ remplie de *val*.

```
val init_board : int * int -> 'a -> 'a list list = <fun>
```

3.2 get_cell

Écrire la fonction `get_cell` $(x,\ y)$ *board* qui retourne la valeur en position $(x,\ y)$ dans la matrice *board*.

```
val get_cell : 'a * 'a -> 'a list list = <fun>
```

3.3 put_cell

Écrire la fonction `put_cell` *val* $(x,\ y)$ *board* qui remplace la valeur en $(x,\ y)$ dans la matrice *board* par la valeur *val*.
Si la case $(x,\ y)$ n'existe pas, *board* est retournée inchangée (pas d'exception).

```
val put_cell : 'a -> int * int -> 'a list list -> 'a list list =
 <fun>
```

3.4 count_neighbours

Écrire la fonction `sum_neighbours` $(x,\ y)$ *board* $(l,\ c)$ qui retourne la somme des valeurs autour de la cellule en $(x,\ y)$ dans *board* de taille $(l,\ c)$.

```
val count_neighbours : int * int -> int list list -> int * int ->
 int = <fun>
```

Exemples

```
# let board = init_board (5, 3) 0;;
val board : int list list =
  [[0; 0; 0]; [0; 0; 0]; [0; 0; 0]; [0; 0; 0]; [0; 0; 0]]

# let board = put_cell 1 (0, 0) board;;
val board : int list list =
  [[1; 0; 0]; [0; 0; 0]; [0; 0; 0]; [0; 0; 0]; [0; 0; 0]]

# let board = put_cell 2 (2, 1) board;;
val board : int list list =
  [[1; 0; 0]; [0; 0; 0]; [0; 2; 0]; [0; 0; 0]; [0; 0; 0]]

# get_cell (2, 1) board;;
- : int = 2

# count_neighbours (1, 0) board (5, 3);;
- : int = 3

# let board2 = init_board (3, 4) 1;;
val board2 : int list list = [[1; 1; 1; 1]; [1; 1; 1; 1]; [1; 1; 1; 1]]

# count_neighbours (1, 2) board2 (3, 4);;
- : int = 8

# count_neighbours (2, 3) board2 (3, 4);;
- : int = 3
```