On donne les exemples suivants de fonction **pour la syntaxe uniquement**. Il est dans l'immense majorité des cas préférable d'utiliser des filtrages et le constructeur `::`, plutôt que ces fonctions (ou les fonctions de même nom déja implémentées dans le module `List`).

```
let cons x l = x::l

let hd l =
  match l with
    | [] -> failwith "liste vide"
    | x::_ -> x

let tl l =
  match l with
    | [] -> failwith "liste vide"
    | _::q -> q
```

Exercice 1 : **Quelques fonctions récursives de base**

1. Écrire une fonction récursive `length : 'a list -> int` qui renvoie le nombre d'éléments de la liste passée en arguments.
   Exemple : `length [1; 2; 3; 42] = 4`.
   À l'avenir, plutôt que réécrire cette fonction vous pouvez utiliser la fonction `List.length`.
2. Écrire une fonction récursive `nth : 'a list -> int -> 'a` telle que `nth l i` est l'élément d'indice `i` dans la liste `l` (on commence à l'indice 0).
   Exemple : `nth [1; 2; 3; 42] 2 = 3`.
   À l'avenir, plutôt que réécrire cette fonction vous pouvez utiliser la fonction `List.nth` **mais c'est peu recommandé car cette fonction est de complexité linéaire**.
3. Écrire une fonction récursive `map : ('a -> 'b) -> 'a list -> 'a list` telle que `map f l` est la liste des images par `f` des éléments de `l`.
   Exemple : `map Float.abs [-3.5; 0.; 4.2; -0.2] = [3.5; 0.; 4.2; 0.2]`.
   À l'avenir, plutôt que réécrire cette fonction vous pouvez utiliser la fonction `List.map`.

Exercice 2 :

1. Écrire une fonction récursive `retire : 'a list -> 'a -> 'a list` telle que `retire l x` soit une liste qui contient les mêmes éléments que `l` dans le même ordre, sauf tous ceux de valeur `x`.
   Exemple : `retire [3; 0; -1; 1; 61; 2; 1; 1; 8; 14] 1 = [3; 0; -1; 61; 2; 8; 14]`.
2. Cette fonction a-t-elle un effet de bord ? Si on représente `l` et `retire l 1` par des maillons, les deux listes ont-elles des choses en commun ?

Exercice 3 : Et quelques fonctions plus compliquées

1. On souhaite écrire une fonction pour calculer le renversé d'une liste (à l'avenir on pourra utiliser `List.rev` poue cela).
   (a) Pourquoi ne peut-on pas simplement écrire une fonction récursive `rev : 'a list -> 'a list` qui fait ce qu'on veut ?
   (b) Écrire une fonction récursive `rev_aux : 'a list -> 'a list -> 'a list` telle que `rev l1 l2` calcule le renversé de `l1` suivi de `l2`.
   Exemple : `rev_aux [1; 2; 42] [12; 18] = [42; 2; 1; 12; 18]`.
   (c) En utilisant la fonction ci-dessus, écrire `rev`.
2. Écrire une fonction récursive `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` telle que `fold_left f init l`, avec `l` $= [x_0; x_1; \ldots; x_{n-1}]$ est `f (f ... (f init` x_0`) ...` x_{n-2}`)` x_{n-1}.
   Exemple : `fold_left min max_int [3; 17; -2; 5] = -2`.
   À l'avenir, plutôt que réécrire cette fonction vous pouvez utiliser la fonction `List.fold_left`.