Exercice 1 : Zipper sur les listes

Un zipper sur une liste est la donnée d'un *curseur* positionné sur un élément. On le représente par le type suivant :

```
type 'a list_zipper = {
  before : 'a list;  (* éléments avant le curseur, en ordre inverse *)
  after  : 'a list;  (* éléments à partir du curseur (tête = élément courant) *)
}
```

Par exemple, le zipper représentant la liste `[1; 2; 3; 4; 5]` avec le curseur positionné sur `3` est : `{ before = [2; 1]; after = [3; 4; 5] }`

Q. 1 Écrire la fonction `of_list : 'a list -> 'a list_zipper option` qui crée un zipper positionné sur le premier élément d'une liste, et renvoie `None` si la liste est vide.

Q. 2 Écrire la fonction `current : 'a list_zipper -> 'a option` qui renvoie l'élément courant, ou `None` si le curseur est en dehors de la liste.

Q. 3 Écrire la fonction `to_list : 'a list_zipper -> 'a list` qui reconstruit la liste complète à partir d'un zipper, indépendamment de la position du curseur.

Q. 4 Écrire `move_right : 'a list_zipper -> 'a list_zipper option`, qui déplace le curseur d'un cran vers la droite. La fonction renvoie `None` si le curseur est déjà en fin de liste.

Q. 5 Écrire `move_left : 'a list_zipper -> 'a list_zipper option`, symétriquement.

Q. 6 Écrire `set : 'a -> 'a list_zipper -> 'a list_zipper option` qui remplace l'élément courant par la valeur donnée. Renvoie `None` si la liste est vide.

Q. 7 Écrire `insert_before : 'a -> 'a list_zipper -> 'a list_zipper` qui insère un élément juste avant l'élément courant. Le curseur reste positionné sur l'ancien élément courant.

Q. 8 Écrire `delete : 'a list_zipper -> 'a list_zipper option` qui supprime l'élément courant. Le curseur se positionne alors sur l'élément suivant s'il existe, ou sur le précédent sinon. Renvoie `None` si la liste est vide.

Exercice 2 : Permutations

Dans cet exercice on considère uniquement des tableaux T de taille $n \geqslant 0$, contenant une et une seule fois chaque entier de $[\![0, n-1]\!]$. Étant donné un tableau et un entier $i \in [\![0, n-1]\!]$, on note $o_T(i)$ (on l'appelle l'ordre de i dans le tableau T) le plus petit entier $p > 0$ tel que $\underbrace{T[T[T[\ldots[T}_{p \text{ fois}}[i]]]]] = i$.

Q. 1 Définir une fonction `o : int array -> int -> int` prenant en paramètres un tableau T et un entier i et calculant $o_T(i)$.

Q. 2 Définir une fonction `max_o : int array -> int` prenant en paramètres un tableau T et retournant l'élément d'ordre maximal dans le tableau. On pourra supposer le tableau non vide.

Exercice 3 : Matrice creuse

On considère dans cet exercice des matrices carrées d'entiers relatifs : $M \in \mathcal{M}_n(\mathbb{Z})$. Par exemple :

$$\begin{pmatrix} 0 & 3 & 5 \\ 0 & 0 & 7 \\ -1 & 0 & 0 \end{pmatrix}$$

On représente de telles matrices de deux manières :

- Ou bien par un tableau de tableau d'entiers relatifs, par exemple : `[|[|0; 3; 5|]; [|0; 0; 7|]; [|-1; 0; 0|]|]`.
- Ou bien par une liste de triplets : (i, j, v) indiquant que dans la case d'indice (i, j) de la matrice se trouve la valeur v. Si une case n'est pas mentionnée par une telle liste, c'est que la matrice représentée contient 0. Par exemple pour la matrice ci-dessus `[(0, 1, 3); (0, 2, 5); (1, 2, 7); (2, 0, -1)]`.

Q. 1 Donner des fonctions de conversion d'une représentation dans l'autre.

Exercice 4 : Découpage en deux de listes en C

On dispose (dans le fichier joint) d'une définition d'un type liste en C, que l'on rappelle ci-dessous.

```c
struct list_s {
  int elem ;                    /* L'élément en tête de liste */
  struct list_s* next ;         /* Le suivant */
};

typedef struct list_s* liste ;
```

Q. 1 Écrire une fonction C `void decoupe_en_deux(liste entree, liste* r1, liste* r2)` qui partage la liste `entree` en deux moitiés en place, sans allouer de nouveaux nœuds ni copier de valeurs. Après l'exécution `*r1` devra donc pointer vers le début d'une liste contenant les premiers éléments de `entree`, `*r1` devra pointer vers le début d'une liste contenant les derniers éléments de `entree`, Si la liste a un nombre impair d'éléments, la première moitié est la plus courte.

Indication : Utiliser deux curseurs, l'un avançant d'un nœud à chaque étape, l'autre de deux. Quand le curseur rapide atteint la fin de la liste, le curseur lent se trouve au milieu.

Q. 2 Écrire la fonction C `void decoupe_en_deux(liste entree, liste* r1, liste* r2)` qui partage la liste `entree` en deux moitiés en place, sans allouer de nouveaux nœuds ni copier de valeurs. Les nœuds doivent être répartis selon leur position : les nœuds aux positions paires forment `*r1`, ceux aux positions impaires forment `*r2`. L'ordre relatif des nœuds est conservé dans chaque sous-liste. Aucun nœud n'est alloué ou copié.