Exercice 1 : Profondeur d'une étiquette

Dans cet exercice, on considère les arbres binaires, définis en OCaml au moyen du type ci-dessous.

```
type 'a btree =
  | V
  | N of 'a * 'a btree * 'a btree
```

Q. 1 Définir une fonction `profondeur_etiquette : 'a btree -> 'a -> int option` prenant en paramètres un arbre binaire et une étiquette x et retournant la profondeur du nœud le plus profond contenant l'étiquette x. Si un tel nœud n'existe pas, on retournera `None`.

On considère à présent les arbres généraux, définis ci-dessous.

```
type 'a gtree = GN of 'a * 'a gtree list
```

Q. 2 Définir une fonction `profondeur_etiquette : 'a gtree -> 'a -> int option` prenant en paramètres un arbre général et une étiquette x et retournant la profondeur du nœud le plus profond contenant l'étiquette x. Si un tel nœud n'existe pas, on retournera `None`.

Exercice 2 : Sous-arbres

Dans cet exercice, on considère les arbres binaires, définis en OCaml au moyen du type ci-dessous.

```
type 'a btree =
  | V
  | N of 'a * 'a btree * 'a btree
```

Q. 1 Définir une fonction `egaux : 'a btree -> 'a btree -> bool` prenant en paramètres deux arbres et testant s'ils sont égaux.

On dit d'un arbre a que c'est un sous-arbre d'un arbre b dès lors que l'arbre a apparaît dans l'arbre b. Formellement, on peut définir l'ensemble des sous-arbres inductivement de la manière suivante :

$$\begin{aligned} \text{sous\_arbres}(\mathsf{E}) &= \{\mathsf{E}\} \\ \text{sous\_arbres}(\mathsf{N}(x,g,d)) &= \{\mathsf{N}(x,g,d)\} \cup \{\text{sous\_arbres}(g)\} \cup \{\text{sous\_arbres}(d)\}. \end{aligned}$$

Q. 2 Définir une fonction OCaml `sous_arbres : 'a btree -> 'a btree -> bool` permettant de tester si un arbre a est un sous-arbre d'un arbre b.

On considère à présent les arbres généraux, définis ci-dessous.

```
type 'a gtree = N of 'a * 'a gtree list
```

Q. 3 Définir une fonction OCaml `sous_arbres : 'a gtree -> 'a gtree -> bool` permettant de tester si un arbre général a est un sous-arbre d'un arbre général b. *Indication :* On rappelle l'existence de la fonctionnelle `List.exists : ('a -> bool) -> 'a list -> bool` prenant en paramètres un prédicat p et une liste l et testant s'il existe, dans l, un élément x tel que $p(x)$ est vrai.

Exercice 3 : Meilleure branche

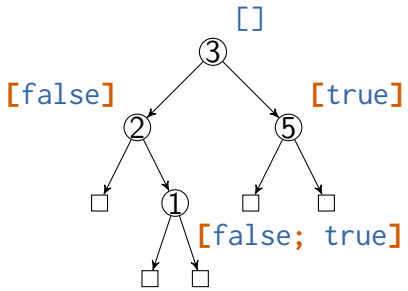
Dans cet exercice, on considère les arbres binaires, définis en OCaml au moyen du type ci-dessous.

```
type 'a btree =
  | V
  | N of 'a * 'a btree * 'a btree
```

Une *branche* dans un arbre binaire est un chemin entre la racine et une feuille de l'arbre. On définit la *valeur* d'une branche $(n_1, n_2, \ldots, n_p)$ comme la somme des étiquettes des nœuds des branches.

Q. 1 Définir une fonction OCaml `val_pg_branche: int btree -> int` calculant la valeur de la plus grande branche d'un arbre binaire étiqueté par des entiers.

On représente une branche, dans un arbre binaire, en OCaml, au moyen d'une liste de booléens, comme indiqué dans la figure ci-dessous.



Q. 2 Définir une fonction OCaml `pg_branche: int btree -> (int * (bool list))` calculant à la fois la valeur d'une plus grande branche et une plus grande branche d'un arbre binaire étiqueté par des entiers.

On considère à présent les arbres généraux, définis ci-dessous.

```
type 'a gtree = GN of 'a * 'a gtree list
```

On représente une branche, dans un arbre binaire, en OCaml, au moyen d'une liste d'entiers : le premier fils est indiqué par 0, le second par 1, le troisième par 2, ….

Q. 3 Définir une fonction OCaml `pg_branche_g: int gtree -> (int * (intr list))` calculant à la fois la valeur d'une plus grande branche et une plus grande branche d'un arbre général étiqueté par des entiers.