Consignes. Les calculatrices sont interdites. Numérotez vos feuilles et faites apparaître les questions dans l'ordre du sujet. Si vous repérez une erreur d'énoncé, signalez-la sur votre copie et poursuivez votre composition.

Les parties 2, 3, 4 sont indépendantes et utilisent les notations de la partie 1.

1 Introduction

Dans les questions de programmation, on manipulera des listes d'entiers dont les éléments sont distincts deux à deux. En OCaml, ces listes seront de type `parc` (pour "parcours").

```
(* Une liste de type parc ne contient pas de doublon *)
type parc = int list;;
```

De plus, on va s'intéresser à des arbres binaires dont les noeuds sont étiquetés par des entiers distincts deux à deux. En OCaml, ces arbres seront de type `arbre1`.

```
(* Les étiquettes d'un arbre sont distinctes deux à deux *)
type arbre1 =
  | Vide (* Arbre vide *)
  | N of (int * arbre1 * arbre1);;
```

Par exemple, l'arbre de la figure 1 est représenté par `a1_fig1`.

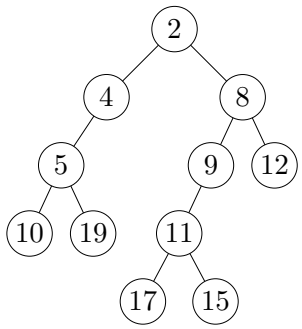

FIGURE 1

```
let a1_fig1 =
  N (2,
     N (4,
        N (5, N (10, Vide, Vide), N (19, Vide, Vide)),
        Vide),
     N (8,
        N (9,
           N (11, N (17, Vide, Vide), N (15, Vide, Vide)),
           Vide),
        N (12, Vide, Vide)));;
```

Les notions étudiées dans ce sujet sont fortement liées au parcours en profondeur infixe.

1. Écrire une fonction `inf: arbre1 -> parc` qui renvoie le parcours infixe de l'arbre en entrée. Votre fonction devra être de complexité linéaire en le nombre de noeuds.

Soit A un arbre. On dira que A est un ***arbre tournoi*** si pour tout noeud x différent de la racine, l'étiquette du père de x est strictement inférieure à l'étiquette de x. Par exemple, l'arbre de la figure 1 est un arbre tournoi.

2. (a) Écrire une fonction `tournoi: arbre1 -> bool` qui indique si l'arbre en entrée est un arbre tournoi.
   (b) Quelle est la complexité de votre fonction ? Justifier.

Soit A un arbre et « `li: parc` » une liste d'entiers distincts deux à deux. On dira que A est un ***arbre cartésien associé à `li`*** si les deux conditions suivantes sont respectées :

(i) A est un arbre tournoi.

(ii) Lorsqu'on liste les étiquettes de A dans l'ordre du parcours infixe, on obtient `li`.

Par exemple, l'arbre de la figure 1 est un arbre cartésien associé à :

```
li_fig1 = [10; 5; 19; 4; 2; 17; 11; 15; 9; 8; 12].
```

3. Dessiner un arbre cartésien associé à `[18, 6, 16, 4, 10, 19, 3, 8, 12, 5, 11]`.
4. Soit A un arbre tournoi non vide. Montrer par récurrence sur le nombre de noeuds de A que toutes les étiquettes de A sont supérieures ou égales à l'étiquette de la racine de A.

Soit « `li: parc` » une liste. En utilisant le résultat de la question 4, on peut démontrer par récurrence sur la longueur de `li` qu'il existe exactement un arbre cartésien associé à `li`. Jusqu'à la fin du sujet, on se permet donc de dire "l'arbre cartésien associé à `li`" à la place de "un arbre cartésien associé à `li`". L'objectif des parties qui suivent est d'étudier différentes méthodes pour construire cet arbre.

2 Méthode de construction 1

Soit `li` une liste d'entiers distincts deux à deux. Pour construire l'arbre cartésien associé à `li`, on utilise une procédure récursive :

- Si `li` est vide, l'arbre cartésien associé est l'arbre vide.
- Sinon :
  - ⋆ On commence par construire deux nouvelles listes `li1` et `li2` en coupant `li` au niveau de son minimum. Par exemple, lorsqu'on coupe `li_fig1` (définie page 2) au niveau de son minimum (qui vaut `2`), on obtient :

    ```
    li1 = [10; 5; 19; 4]      et      li2 = [17; 11; 15; 9; 8; 12]
    ```

  - ⋆ On construit récursivement G l'arbre cartésien associé à `li1` et D l'arbre cartésien associé à `li2`.
  - ⋆ L'arbre cartésien associé à `li` est alors l'arbre dont la racine est étiquetée par le minimum de `li`, dont le sous-arbre gauche est G et dont le sous-arbre droit est D.

Programmons cette procédure.

5. (a) Écrire une fonction `mini: parc -> int` qui renvoie le minimum de la liste donnée en entrée.
   (b) Écrire une fonction `couper: parc -> int -> (parc * parc)` qui prend en entrée une liste `li` ainsi qu'un entier `m` appartenant à `li`, et renvoie les deux listes obtenues lorsqu'on coupe `li` au niveau de `m`. Par exemple avec les listes `li_fig1`, `li1`, `li2` définies ci-dessus :

   ```
   couper li_fig1 2 = (li1, li2)
   ```

   (c) Donner la complexité de la fonction `mini` et celle de la fonction `couper`. Justifier.
6. (a) Écrire une fonction `cart1: parc -> arbre1` qui prend en entrée une liste `li` sans doublon, et renvoie l'arbre cartésien associé à `li`. Vous devez utiliser la méthode de construction 1 décrite dans cette partie.
   (b) Quel est le pire cas pour la fonction `cart1` ? Calculer la complexité associée.
   (c) Même question pour le meilleur cas.

3 Méthode de construction 2 : algorithme diviser pour régner

3.1 Étude théorique

Dans cette partie, pour tout arbre A, la notation $I(A)$ désigne la liste des étiquettes de A dans l'ordre du parcours infixe. Lorsque A est non vide, $r(A)$ désigne l'étiquette de la racine de l'arbre, $G(A)$ désigne son sous-arbre gauche et $D(A)$ son sous-arbre droit. Enfin, pour toute liste `li` d'entiers deux à deux distincts, on note $T($`li`$)$ l'arbre cartésien associé à `li`. Par exemple si A est l'arbre de la figure 1, alors $I(A) =$ `li_fig1` (la liste `li_fig1` est définie page 2), $r(G(A)) = 4$ et $r(D(A)) = 8$. De plus, on remarque que :

- Pour toute liste « `li: parc` » : $I(T($`li`$)) =$ `li`.
- Pour tout arbre A, si A est un arbre tournoi, alors $T(I(A)) = A$.

Afin de construire un arbre cartésien, on va utiliser une stratégie de type "diviser pour régner" proche de l'algorithme du tri fusion. Pour cela, on a besoin d'une opération qui fusionne deux arbres cartésiens, c'est-à-dire d'une opération qui construit l'arbre $T(\texttt{li1 @ li2})$ à partir de $T(\texttt{li1})$ et $T(\texttt{li2})$.

7. Soient `li`, `li1`, `li2` trois listes d'entiers non vides de type `parc` telles que `li = li1 @ li2`. Soit A l'arbre tel que :
   - La racine de A est étiquetée par $r(T(\texttt{li1}))$.
   - Le sous-arbre gauche de A est $G(T(\texttt{li1}))$.
   - Le sous-arbre droit de A est $T\Big(I(D(T(\texttt{li1}))) \texttt{ @ li2}\Big)$.

   (a) Montrer que $I(A) = \texttt{li}$.
   (b) Supposons que $r(T(\texttt{li1})) < r(T(\texttt{li2}))$. Montrer que $A = T(\texttt{li})$.

8. Soient `li`, `li1`, `li2` trois listes d'entiers non vides de type `parc` telles que `li = li1 @ li2`. Supposons que $r(T(\texttt{li1})) > r(T(\texttt{li2}))$. En s'inspirant de la question précédente, construire un arbre A tel que $A = T(\texttt{li})$.

3.2 Implémentation

9. Écrire une fonction `sep: parc -> int -> (parc * parc)` qui prend en entrée une liste `li` ainsi qu'un entier `k` et renvoie deux listes `li1`, `li2` telles que `li1` soit de longueur `k` et `li = li1 @ li2`. Par exemple, pour la liste `li_fig1` définie page 2 :

```
sep li_fig1 6 = ([10; 5; 19; 4; 2; 17], [11; 15; 9; 8; 12])
```

10. Soient `li`, `li1`, `li2` trois listes d'entiers de type `parc` telles que `li = li1 @ li2`. À l'aide des questions 7 et 8, écrire une fonction `fusion: arbre1 -> arbre1 -> arbre1` qui prend en entrée $T(\texttt{li1})$ ainsi que $T(\texttt{li2})$, et renvoie $T(\texttt{li1 @ li2})$. Votre fonction devra être de complexité $\mathcal{O}(n)$ où n est le nombre de noeuds de $T(\texttt{li1 @ li2})$.

11. (a) En s'inspirant du tri fusion, écrire une fonction `cart2: parc -> arbre1` qui construit l'arbre cartésien associé à la liste donnée en entrée.
    (b) Quelle est la complexité de votre fonction ?

4 Méthode de construction 3

4.1 Représentation alternative des arbres

Pour cette méthode, on aura besoin de représenter les arbres en tant qu'objets du type `arbre2` et en tant qu'objets du type `arbre3`. La partie "Rappels" à la fin du sujet rappelle la définition du type « `'a option` ».

```
type arbre2 = {
    etiq: int array;
    r: int;
    fils: (int option * int option) array
  };;
```

```
type arbre3 = {
    etiq: int array;
    pere: int option array;
  }
```

Soit A un arbre avec `n` $\in \mathbb{N}^*$ noeuds. On numérote les noeuds de A de `0` à `n-1` en suivant l'ordre du parcours infixe. Une variable « `a2: arbre2` » représente A lorsque :

- Le tableau `a2.etiq` est de taille `n` et `a2.etiq.(i)` est l'étiquette du noeud numéro `i`. En d'autres termes, `a2.etiq` est le tableau des étiquettes de A dans l'ordre du parcours infixe.
- L'entier `a2.r` $\in [\![\texttt{0 ; n} - \texttt{1}]\!]$ est le numéro de la racine de A.
- Le tableau `a2.fils` est de taille `n` et `a2.fils.(i)` correspond aux numéros des deux fils de `i`. On utilisera l'objet `None` pour gérer les cas où `i` a 0 ou 1 fils.

Par exemple, l'arbre de la figure 1 est représenté par `a2_fig1`.

```
let a2_fig1 = {
    etiq = [|10; 5; 19; 4; 2; 17; 11; 15; 9; 8; 12|];
    r = 4;
    fils = [|(None, None); (Some 0, Some 2); (None, None); (Some 1, None);
             (Some 3, Some 9); (None, None); (Some 5, Some 7); (None, None);
             (Some 6, None); (Some 8, Some 10); (None, None)|]
  }
```

Une variable « `a3: arbre3` » représente A lorsque :

- Le tableau `a3.etiq` est de taille `n` et `a3.etiq.(i)` est l'étiquette du noeud numéro `i`. En d'autres termes, `a3.etiq` est le tableau des étiquettes de A dans l'ordre du parcours infixe.
- Le tableau `a3.pere` est de taille `n` et `a3.pere.(i)` correspond au numéro du père de `i`. On utilisera l'objet `None` pour indiquer que `i` n'a pas de père (c'est-à-dire que c'est la racine de A).

Par exemple, l'arbre de la figure 1 est représenté par `a3_fig1`.

```
let a3_fig1 = {
    etiq = [|10; 5; 19; 4; 2; 17; 11; 15; 9; 8; 12|];
    pere = [|Some 1; Some 3; Some 1; Some 4; None; Some 6;
             Some 8; Some 6; Some 9; Some 4; Some 9|]
  }
```

12. Écrire une fonction `to_array : parc -> int array` qui convertit une liste d'entiers en un tableau d'entiers. Par exemple, pour la liste `li_fig1` définie page 2 :

    `to_array li_fig1 = [|10; 5; 19; 4; 2; 17; 11; 15; 9; 8; 12|]`

13. Écrire une fonction `racine: arbre3 -> int` qui renvoie le numéro de la racine de l'arbre donné en entrée. Par exemple, « `racine a3_fig1` » vaut `4`.
14. Écrire une fonction `a3_to_a2: arbre3 -> arbre2` qui convertit du type `arbre3` vers le type `arbre2`. Par exemple, « `a3_to_a2 a3_fig1` » vaut `a2_fig1`.
15. Écrire une fonction `a2_to_a1: arbre2 -> arbre1` qui convertit du type `arbre2` vers le type `arbre1`. Par exemple, « `a2_to_a1 a2_fig1` » vaut `a1_fig1`.

4.2 Voisins d'un noeud

Soit « `tab: int array` » un tableau dont les éléments sont distincts deux à deux. Pour tout indice `i`, on appelle ***voisin gauche de i*** le plus grand indice `j` tel que `j < i` et `tab.(j) < tab.(i)`. Dans le cas où aucun indice `j` ne vérifie cette condition, on dira que `i` n'a pas de voisin gauche. Par exemple, pour :

`tab_fig1 = [|10; 5; 19; 4; 2; 17; 11; 15; 9; 8; 12|]`.

On obtient (une croix signifie que i n'a pas de voisin gauche) :

i	0	1	2	3	4	5	6	7	8	9	10
Voisin gauche de i	×	×	1	×	×	4	4	6	4	4	9

Afin de calculer les voisins gauches de tous les indices, on va utiliser un algorithme dont voici le pseudo-code :

Pseudo-code 1
Entrées: le tableau `tab` dont les éléments sont distincts deux à deux.
Sortie: cette procédure calcule le voisin gauche de `i` pour chaque indice `i` de `tab`.

Soit P une pile vide et n la longueur de `tab`
pour chaque $i \in [\![0 ; n-1]\!]$ **faire**
    `(*` Dans la ligne qui suit, `j` est l'élément se trouvant au sommet de P `*)`
    tant que P est non vide et `tab.(j) > tab.(i)` **faire**
        Dépiler un élément de P
    fin tant que
    si P est vide **alors**
        L'indice i n'a pas de voisin gauche
    sinon
        L'indice i a pour voisin gauche l'élément se trouvant au sommet de P
    fin si
    On empile i dans P
fin pour

16. On exécute le pseudo-code 1 en lui donnant en entrée le tableau `tab_fig1` défini page 4. Dessiner la pile à la fin de chacun des n tours de la boucle "**pour**".

Pour manipuler des piles en OCaml, on utilisera le module `Stack` (voir la partie "Rappels" à la fin du sujet).

17. (a) Écrire une fonction `top: 'a Stack.t -> 'a` qui renvoie l'élément se trouvant au sommet de la pile. Contrairement à la fonction `Stack.pop`, votre fonction ne doit pas modifier la pile en entrée.
    (b) Écrire une fonction `voisG : int array -> int option array` qui implémente le pseudo-code 1. On pourra supposer sans le vérifier que les éléments de `tab` sont distincts deux à deux. Par exemple, pour le tableau `tab_fig1` défini page 4, « `voisG tab_fig1` » renvoie :

```
[|None; None; Some 1; None; None; Some 4; Some 4; Some 6; Some 4; Some 4; Some 9|]
```

    (c) Quelle est la complexité de `voisG` ?

Soit « `tab: int array` » un tableau dont les éléments sont distincts deux à deux. Pour tout indice `i`, on appelle ***voisin droit de i*** le plus petit indice `j` tel que `i < j` et `tab.(j) < tab.(i)`. Dans le cas où aucun indice `j` ne vérifie cette condition, on dira que `i` n'a pas de voisin droit. Par exemple, pour le tableau `tab_fig1` défini page 4 on obtient (une croix signifie que i n'a pas de voisin droit) :

i	0	1	2	3	4	5	6	7	8	9	10
Voisin droit de i	1	3	3	4	×	6	8	8	9	×	×

La définition de "voisin droit" étant similaire à celle de "voisin gauche", on écrit la fonction suivante :

```
let voisD (tab: int array): int option array =
  f2 (voisG (f1 tab));;
```

18. On souhaite faire en sorte que la fonction `voisD` renvoie un tableau contenant le voisin droit de i pour chaque indice i de `tab`. En particulier, on veut que « `voisD tab_fig1` » renvoie :

```
[|Some 1; Some 3; Some 3; Some 4; None; Some 6; Some 8; Some 8; Some 9; None; None|]
```

    (a) Écrire la fonction `f1 : 'a array -> 'a array`.
    (b) Écrire la fonction `f2 : int option array -> int option array`.

4.3 Construction de l'arbre cartésien

On reprend les notations des parties 4.1 et 4.2. On se place dans le cas où le tableau `tab` évoqué au début de la partie 4.2 contient le parcours infixe de l'arbre A. Pour tout $i \in [\![0\,; n-1]\!]$, soit V_i l'ensemble contenant le voisin gauche et le voisin droit de i. Notons que V_i est de cardinal 0, 1 ou 2 car le voisin gauche ou le voisin droit peut ne pas exister. Pour l'arbre de la figure 1 :

$$V_0 = \{1\} \qquad V_4 = \{\} \qquad V_7 = \{6; 8\} \qquad V_9 = \{4\}$$

Par ailleurs, on note E_i le plus petit ensemble tel que :

- $i \in E_i$.
- Si $j \in E_i$ et que j n'est pas la racine de A, alors le père de j appartient à E_i.

Les éléments de E_i sont appelés les ***ancêtres*** de i. Pour l'arbre de la figure 1 :

$$E_0 = \{1; 3; 4\} \qquad E_4 = \{4\} \qquad E_7 = \{4; 6; 7; 8; 9\} \qquad E_9 = \{4; 9\}$$

19. Soit $i \in [\![0\,; n-1]\!]$. Montrer que V_i est vide si et seulement si i est la racine de A.
20. Soit $i \in [\![0\,; n-1]\!]$ tel que V_i est non vide.
    (a) Montrer que le père de i appartient à V_i
    (b) Montrer que $V_i \subset E_i$.
    (c) Montrer que le père de i est l'élément de V_i dont l'étiquette est maximale.
21. À l'aide des résultats des questions 19 et 20, écrire une fonction `make_a3 : int array -> arbre3` qui renvoie l'arbre cartésien associé au tableau donné en entrée.
22. (a) À l'aide des fonctions définies dans cette partie, écrire une fonction `cart3: parc -> arbre1` qui construit l'arbre cartésien associé à la liste donnée en entrée.
    (b) Quelle est la complexité de `cart3` ?

Rappels

Le type « `'a option` » de la bibliothèque standard d'OCaml est défini par :

```
type 'a option =
  | None       (* Absence de valeur *)
  | Some of 'a (* Présence de valeur *)
```

Dans le module `Stack` de la bibliothèque standard d'OCaml :

- « `'a Stack.t` » désigne le type des piles contenant des éléments de type `'a`.
- « `Stack.create: unit -> 'a Stack.t` » renvoie une pile vide.
- « `Stack.is_empty: 'a Stack.t -> bool` » teste si une pile est vide.
- « `Stack.push: 'a -> 'a Stack.t -> unit` » empile un élément.
- « `Stack.pop: 'a Stack.t -> 'a` » dépile un élément.