Deuxième devoir surveillé d'informatique

1 Peignes

On suppose défini le type arbre de la manière suivante :

```
type arbre =
   |Feuille of int
   |Noeud of arbre * arbre ;;
```

On dit qu'un arbre est un *peigne* si tous les noeuds à l'exception éventuelle de la racine ont au moins une feuille pour fils. On dit qu'un peigne est un *strict* si sa racine a au moins une feuille pour fils, ou s'il est réduit à une feuille. On dit qu'un peigne est *rangé* si le fils droit d'un noeud est toujours une feuille. Un arbre réduit à une feuille est un peigne rangé.

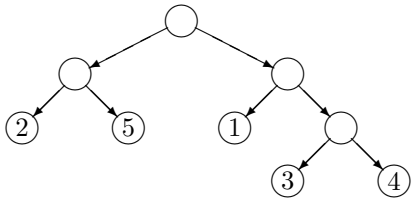

Figure 1 : un peigne à cinq feuilles

1. Représenter un peigne rangé à 5 feuilles.
2. La *hauteur* d'un arbre est le nombre de noeuds maximal que l'on rencontre pour aller de la racine à une feuille (la hauteur d'une feuille seule est 0). Quelle est la hauteur d'un peigne rangé à n feuilles ? On justifiera la réponse.
3. Ecrire une fonction `est_range : arbre → bool` qui renvoie `true` si l'arbre donné en argument est un peigne rangé.
4. Ecrire une fonction `est_peigne_strict : arbre → bool` qui renvoie `true` si l'arbre donné en argument est un peigne strict. En déduire une fonction `est_peigne : arbre → bool` qui renvoie `true` si l'arbre donné en argument est un peigne.
5. On souhaite ranger un peigne donné. Supposons que le fils droit N de sa racine ne soit pas une feuille. Notons A_1 le sous-arbre gauche de la racine, f l'une des feuilles du noeud N et A_2 l'autre sous-arbre du noeud N. On va utiliser l'opération de *rotation* qui construit un nouveau peigne où
   - le fils droit de la racine est le sous-arbre A_2 ;
   - le fils gauche de la racine est un noeud de sous-arbre gauche A_1 et de sous-arbre droit la feuille f.

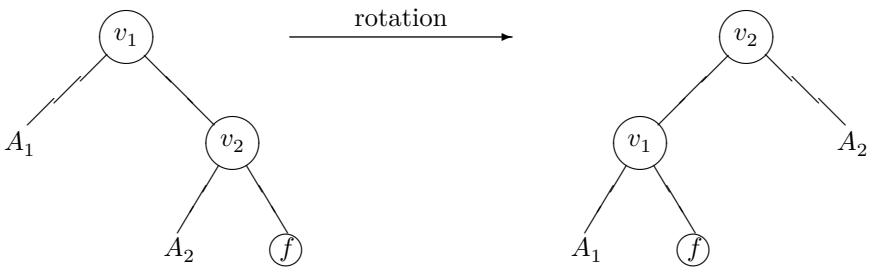

Figure 2 : une rotation

(a) Donner le résultat d'une rotation sur l'arbre de la figure 1.

(b) Ecrire une fonction `rotation : arbre` $\to$ `arbre` qui effectue l'opération décrite ci-dessus. La fonction renverra l'arbre initial si une rotation n'est pas possible.

(c) Ecrire une fonction `rangement : arbre` $\to$ `arbre` qui range un peigne donné en argument, c'est à dire qu'il renvoie un peigne rangé ayant les mêmes feuilles que celui donné en argument. La fonction renverra l'arbre initial si celui-ci n'est pas un peigne.

2 Exercice : hauteur d'un arbre

Pour cet exercice, on prend des notations « à la CCP ».

Définition 2.1. *Un arbre binaire d'entiers a est une structure qui peut soit être vide (notée $\emptyset$), soit être un nœud qui contient une étiquette entière (notée $\mathcal{E}(a)$), un sous-arbre gauche (noté $\mathcal{G}(a)$) et un sous-arbre droit (noté $\mathcal{D}(a)$) qui sont tous deux des arbres binaires d'entiers. L'ensemble des étiquettes de tous les nœuds de l'arbre a est noté $\mathcal{C}(a)$.*

L'ensemble des arbres binaires d'entiers est noté $\mathcal{B}$.

Pour tout $a \in \mathcal{B}$, on notera N_a son nombre de nœuds.

Définition 2.2. ***(hauteur d'un arbre)*** *On définit la fonction $a \mapsto h_a$ par :*

$$\begin{cases} h_\emptyset = 0 \\ \forall a \in \mathcal{B} \setminus \{\emptyset\},\ h_a = 1 + \max\left(h_{\mathcal{G}(a)}, h_{\mathcal{D}(a)}\right) \end{cases}$$

N.B. Ce n'est danc pas la même définition que dans le cours !

1. Selon cette définition, quelle est la hauteur d'une feuille ?
2. Soit a un arbre binaire d'entiers de hauteur k. Montrer que $N_a \leqslant 2^k - 1$. Quel est son nombre minimal de nœuds ? On attend ici une démonstration complète par récurrence.
3. On fixe $n \in \mathbb{N}$. Considérons un arbre binaire d'entiers de n nœuds. Quelle est la forme de l'arbre dont la hauteur est maximale ? Quelle est la forme de l'arbre dont la hauteur est minimale ? Calculer la hauteur de l'arbre en fonction de n dans ces deux cas. Vous justifierez vos réponses.

   Indication : Pour la hauteur minimale : Soit a de hauteur minimale à n nœuds. Soit k sa hauteur. Encadrer n entre deux entiers dépendant de k (selon que le dernier étage est plein ou pas...)

3 Problème : Deux points du plan les plus proches

Le but est de calculer la distance minimale entre deux points d'un ensemble de points du plan. Un repère orthormé étant fixé, chaque point sera représenté par ses coordonnées, donc un couple de flottants.

```
1 type point = float*float;;
```

La méthode utilisée sera de type « diviser pour régner ». Le cas d'arrêt sera celui d'un ensemble d'au plus 3 points.

- **Diviser :** Soit `l` une liste d'au moins quatre points. On découpe `l` en deux listes `lg` de longueur $\left\lfloor \frac{n}{2} \right\rfloor$ et `ld` de longueur $\left\lfloor \frac{n+1}{2} \right\rfloor$ de telle manière qu'il existe $x_m \in \mathbb{R}$ tel que les points de `ld` ont une abscisse $\leqslant x_m$ et les points de `lg` on une abscisse $\geqslant x_m$.
- **Appels récursifs :** On récupère la distance minimale entre deux points de `lg` d'une part et entre deux points de `ld` d'autre part.
- **Régner :** C'est la partie délicate, elle est étudiée partie 3.2.

3.1 Fonctions préliminaires

1. Écrire une fonction `dist` de type `float * float -> float * float -> float` qui prend deux points et qui renvoie la distance entre ces deux points.

   Indication : On peut commencer par `let dist (a,b) (c,d) =...`

   Cela signifie que les deux arguments doivent étre des couples et qu'on a déjà récupéré abscisse et ordonnée de chacun, en les nommant `a,c,b,d`.
2. **Méthode naïve :** Le méthode naïve consiste à considérer toutes les paires de points de `l` et à garder le minimum.

(a) Écrire une fonction `dist_points_suivant` de type `float * float -> (float * float) list -> float` qui prend un point et une liste de points et qui renvoie la distance minimale entre `p` et un élément de `l` ↪ .

(b) En déduire une fonction pour calculer la distance minimale entre deux points d'une liste de points.

(c) Quelle est la complexité de celle-ci ?

3. **Tri :**

(a) Programmer le tri fusion. Cependant on y apportera une petite variante : votre programme prendra en entrée un argument supplémentaire : la relation d'ordre à utiliser. Celle-ci sera une fonction nommée `pp` (pour «plus petit») de type `'a -> 'a -> bool` telle que pour tout x, y, `pp x y` est vrai ssi $x \leqslant y$.

(b) En déduire une fonction `tri_ord` pour trier une liste de point par ordre croissant d'ordonnée, puis une fonction `tri_absc` pour trier une liste de points par ordre d'abscisses croissantes.

(c) *Cours :* Quelle est la complexité du tri fusion ?

4. **Diviser :** Écrire une fonction `decoupe` de type `('a * 'b) list -> int -> ('a * 'b) list * ('a * 'b) list * 'a` qui prend en entrée une liste `l` et un entier i et qui renvoie un triplet formé de :

- les i premiers éléments de `l` ;
- le reste de `l` ;
- l'abscisse x_m du premier point de la deuxième liste renvoyée.

3.2 Partie théorique

Soit `l` une liste d'au moins quatre points. Soient `lg`, `ld`, x_m obtenu après l'étape «diviser».

On suppose qu'on a trouvé les distances mininales entre deux points de `lg` et entre deux points de `ld` par appel récursif : notons le minimum de ces deux valeurs.

On note B la bande constituée de tous les points d'abscisse dans $[x_m - \delta, x_m + \delta]$.

Notons la distance minimale entre deux points de `l`.

1. Dans quelle situation est-ce que $\gamma < \delta$?
2. On suppose pour cette question que c'est le cas. En particulier $\delta > 0$. On note $P_g \in$ `lg` et $P_d \in$ `ld` tels que $\gamma = d(P_g, P_d)$. On note (x_g, y_g) et (x_d, y_d) les coordonnées de P_g et P_d respectivement.

(a) Montrer que $(P_g, P_d) \in B^2$.

Comme les ordonnées de P_g et P_d différent d'au plus , on trouve un intervalle J tel que le rectangle $[x_m - \delta, x_m + \delta] \times J$ contient P_g et P_d. On note R ce rectangle.

(b) Faire un dessin.

(c) Combien de points de `l` peut-on trouver au maximum dans R ? Refaire un dessin ! Attention : il est possible que `l` contienne des doublons (auquel cas la valeur renvoyée au final sera 0).

3. On revient au cas général. Soit η la distance minimale obtenue en comparant chaque point de $\texttt{l} \cap B$ à ses 7 suivants en prenant les points par ordre d'ordonnée croissante. Démontrer que $\gamma = \min(\delta, \eta)$.

Indication : On pourra distinguer deux cas, selon $\delta = \gamma$ ou pas.

3.3 Programmation finale

1. Écrire une fonction `bande` qui prend x_m, δ et une liste de points `l` et qui renvoie la liste des points de `l` dont l'abscisse est dans $[x_m - \delta, x_m + \delta]$.

Indication : Si vous savez utiliser `List.filter`, vous pouvez le faire en une ligne.

2. Écrire une fonction `parcourt_bande` qui prend une liste de points `b` triée par ordonnées croissantes et qui renvoie le minimum des distances obtenue en calculant la distance de chaque élément de `b` à ses 7 suivants.

Indication : Il faudra sans doute faire une fonction préliminaire avant la fonction demandée elle-même.

3. Écrire une fonction `dist_min_aux` pour appliquer l'algorithme qu'on a décrit ci-dessus. Cette fonction prendra en entrée la liste des points déjà triée par abscisses croissantes, ainsi que la longueur de la liste.

Pour le cas d'arrêt : lorsque la longueur est $\leqslant 3$, on utilise la méthode naïve.

4. Écrire alors la fonction finale, qui prend en entrée seulement la liste des points dans un ordre quelconque.
5. Quel est l'intérêt d'avoir fait cette fonction auxiliaire ?

3.4 Complexité

On pose pour tout $n \in \mathbb{N}$, C_n le nombre maximal de comparaisons pour appliquer l'algorithme à un ensenble d'au plus n points.

1. Montrer qu'il existe une constante K telle que pour tout $n \in [\![4, \infty[\![$,

$$C_n \leqslant C_{\lfloor \frac{n}{2} \rfloor} + C_{\lfloor \frac{n+1}{2} \rfloor} + Kn\log_2(n) \tag{1}$$

   Remarque : Autrement dit, $C_n \leqslant C_{\lfloor \frac{n}{2} \rfloor} + C_{\lfloor \frac{n+1}{2} \rfloor} + O(n\log_2(n))$.

2. **Cas d'une puissance de deux :**
   (a) Vérifier que pour tout $k \in \mathbb{N}$,

$$\frac{C_{2^k}}{2^k} \leqslant \frac{C_{2^{k-1}}}{2^{k-1}} + kK$$

   (b) En déduire que $C_{2^k} = O_{k\to\infty}(k^2 2^k)$.
3. **Cas général :** En déduire que $C_n = O_{n\to\infty}(n\log(n)^2)$.
4. Comparer à la méthode naïve.