Devoir surveillé du Samedi 11 Octobre

Dans l'ensemble du devoir, toutes les fonctions seront :

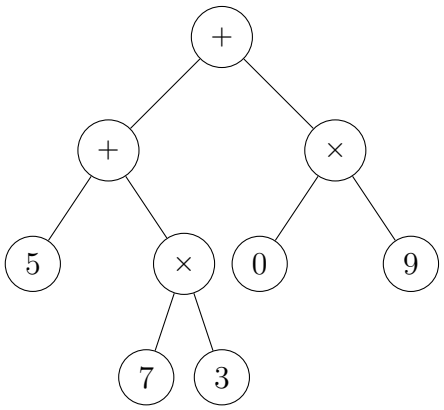
- *écrites en langage OCaml,*
- *précédées d'une explication des variables utilisées,*
- *précédées d'une explication de l'algorithme.*

Exercice 1 (Arbres binaires et polynômes)

On peut représenter une expression arithmétique par un arbre dont les noeuds sont de type `operation` et les feuilles sont de type `float`. Le type `operation` est le suivant :

```
type operation = Addition | Multiplication ;;
```

1. Quelle est l'expression représentée par l'arbre ci-dessous ? Quelle est sa valeur ?



2. On considère le type `expression_arithmetique` suivant :

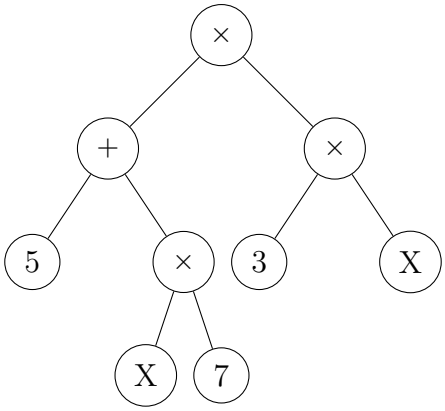
```
type expression_arithmetique =
  | F of float
  | N of operation * expression_arithmetique * expression_arithmetique
;;
```

   Définir l'arbre ci-dessus avec ce type.

3. Écrire une fonction qui calcule la valeur d'une expression arithmétique représentée par un tel arbre.

On s'intéresse maintenant aux polynômes à une indéterminée. On les représente toujours par des arbres, mais les feuilles sont maintenant de type `float` ou représentent l'indéterminée `X`.

On obtient ainsi des arbres tels que celui-ci :



4. Quel est le polynôme représentée par l'arbre ci-dessus ? Quelle est sa valeur en 2 ?
5. (a) Donner une définition du type `polynome`.
   (b) Définir l'arbre ci-dessus avec ce type.
6. Écrire une fonction qui prend en argument un polynôme P, une valeur x pour l'indéterminée et renvoie $P(x)$.
7. (a) Écrire une fonction `add_poly` qui, étant donné deux polynômes représentés sous forme de liste des coefficients par degré croissant, renvoie la liste correspondant à la somme des deux polynômes.

   Par exemple, pour deux polynômes $2 + X^2$ et $1 + 3X + 2X^3$:

   ```
   add_poly [2.; 0.;  1.] [1.; 3.; 0.; 2.] ;;
   - : float list = [3.; 3.; 1.; 2.]
   ```

   (b) Écrire une fonction `mult_poly` qui, étant donné deux polynômes représentés sous forme de liste des coefficients par degré croissant, renvoie la liste correspondant au produit des deux polynômes.

   Par exemple, pour deux polynômes $2 + X^2$ et $1 + 3X + 2X^3$:

   ```
   mult_poly [2.; 0.;  1.] [1.; 3.; 0.; 2.] ;;
   - : float list = [2.; 6.; 1.; 7.; 0.; 2.]
   ```

   (c) En déduire une fonction qui prend en argument un polynôme représenté sous forme d'un arbre et renvoie la liste des coefficients du polynôme sous forme développé, par degré croissant.

Exercice 2 (Les arbres AVL)

Nous noterons pour tout arbre binaire non vide A, $h(A)$ sa hauteur et $s(A)$ son nombre de sommets. Dans ce texte, l'arbre vide est de hauteur -1.

1. Montrer que, si A est un arbre binaire, alors :

$$\log_2(s(A)+1) - 1 \leqslant h(A) \leqslant s(A) - 1.$$

Nous aurons souvent intérêt à utiliser des arbres qui, pour une taille donnée, ont une hauteur minimale, autrement dit pour lesquels :

$$h(A) = O(\log_2(s(A)))\,.$$

2. Le cas idéal.

   Étudions tout d'abord le cas optimal où $h(A) = \log_2(s(A)+1) - 1$ correspondant aux arbres binaires complets.

   (a) Montrer que, pour qu'un arbre $A = (F_g, x, F_d)$ soit complet, il faut et il suffit que F_g et F_d soient complets et de même hauteur.

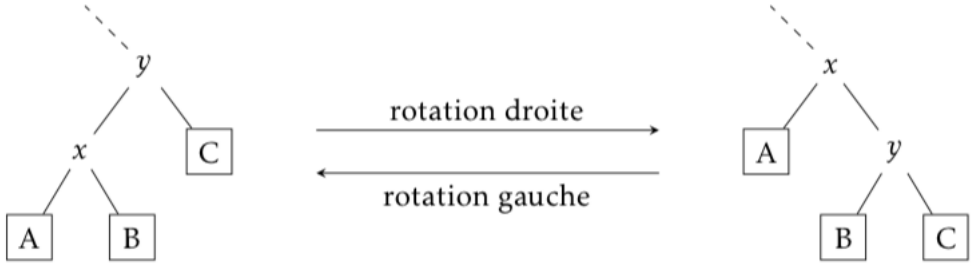
   (b) En déduire qu'un arbre binaire est complet si et seulement si toutes ses feuilles sont à la même profondeur.

Cependant, la plupart des arbres binaires que l'on rencontre dans la pratique ne sont pas complets, cette notion étant trop restrictive. On privilégie donc certaines catégories d'arbres qui garantissent l'équilibrage : arbres rouge-noir, arbres AVL, etc...

3. Les arbres binaires de recherche.

   Soit A un arbre binaire de recherche et y l'un de ses noeuds.

   Une rotation droite autour de y consiste à faire descendre ce noeud et à faire remonter son fils gauche x sans invalider l'ordre des éléments :

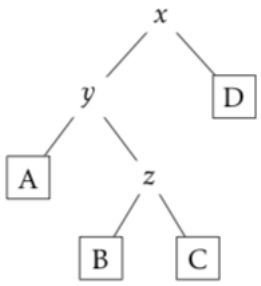


L'opération inverse s'appelle rotation gauche autour du sommet x.

(a) Montrer qu'une rotation préserve la structure d'arbre binaire de recherche.

(b) Écrire en OCaml les deux fonctions `rotation_d` et `rotation_g` correspondantes, de type `arbre -> arbre`. On précisera le type `arbre` utilisé.

(c) Appliquer une rotation gauche autour de y puis une rotation droite autour de x dans l'arbre suivant :



4. Les arbres AVL (d'après le nom de leurs inventeurs Adelson-Velsky et Landis).

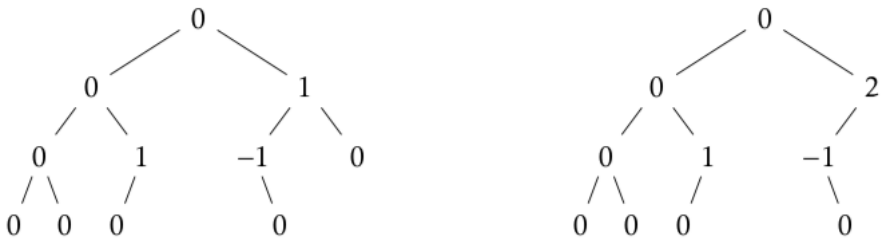
Définition. Le déséquilibre d'un arbre binaire non vide $A = (F_g, x, F_d)$ est égal à l'entier noté $\text{eq}(A)$ défini par :

$$\text{eq}(A) = h(F_g) - h(F_d)\,.$$

Par extension, le déséquilibre de A est aussi appelé le déséquilibre de x et noté $\text{eq}(x)$.

Définition. Un arbre binaire A est un arbre AVL lorsqu'il est vide ou égal à (F_g, x, F_d) avec :
— F_g et F_d sont des arbres AVL,
— le déséquilibre de A est égal à -1, 0 ou 1.
Autrement dit, un arbre AVL est un arbre dans lequel le déséquilibre de chaque sous-arbre est égal à -1, 0 ou 1.

Exemple. L'arbre de gauche suivant est un AVL, pas celui de droite :



(a) On considère la suite de Fibonacci $(f_n)_{n\in\mathbb{N}}$ définie par $f_0 = 0$, $f_1 = 1$ et la relation :

$$\forall n \in \mathbb{N}, \quad f_{n+2} = f_{n+1} + f_n\,.$$

Montrer que tout arbre AVL A de hauteur h contient au moins f_h sommets.

(b) En déduire que pour tout arbre AVL A, on a : $h(A) = O(\log_2(s(A)))$.

5. Les arbres binaires de recherche respectant la condition AVL.
   On considère désormais un arbre binaire de recherche qui respecte la condition AVL, à savoir que le déséquilibre de chaque noeud est égal à -1, 0 ou 1, et on considère une insertion qui a conduit à la création d'une feuille f.
   Seuls les ancêtres de f ont vu leur déséquilibre modifié ; on suppose que l'un au moins ne respecte plus la condition AVL et on note y le premier de ceux-là.

   (a) Montrer que : $\text{eq}(y) = \pm 2$.
   Sans perte de généralité, on suppose désormais que $\text{eq}(y) = 2$ et on note x le fils gauche de y.

   (b) Montrer que si $\text{eq}(x) = 0$ ou $\text{eq}(x) = 1$, alors une rotation suffit pour retrouver la condition AVL.

   (c) Montrer que si $\text{eq}(x) = -1$, alors deux rotations suffisent pour retrouver la condition AVL.

   Il reste à remonter dans la branche menant à la racine et réitérer éventuellement le processus pour rééquilibrer un arbre AVL après une insertion en un coût $O(h(A))$.

Exercice 3 (Algorithmique des mots sans facteur carré)

Dans cette exercice, l'objectif est de construire différents algorithmes pour vérifier si un mot comporte des facteurs carrés ou non.

Dans toute la suite, $\Sigma = \{a_1 < \ldots < a_p\}$ désigne un alphabet totalement ordonné comportant p lettres, ε représente le mot vide et Σ^* est l'ensemble des mots finis obtenus à partir de Σ. Pour tout réel x, on note $\lfloor x \rfloor$ la partie entière de x.

Partie 1 : Définitions

Définition. Soit $w = w_0 \ldots w_{n-1}$ un mot de Σ^*. La **longueur** n de w est notée $|w|$. Pour tout $0 \leqslant i \leqslant j < n$, $w[i, j]$ désigne le mot $w_i \ldots w_j$. Par convention, si $j < i$, $w[i, j]$ désigne ε.

Définition. Soit w un mot de Σ^*. On dit que w est un **carré** s'il existe un mot x tel que $w = x \cdot x$.

Définition. Soient v et w deux mots de Σ^*. On dit que v est un **facteur** de w s'il existe r et s deux mots (éventuellement vides) tels que $w = rvs$.

Définition. On dit qu'un mot w contient une **répétition** s'il contient un facteur carré différent de ε.

Dans la suite, un mot sera représenté en OCaml par la liste de ses lettres. Par exemple, le mot *baba* est représenté par la liste `['b'; 'a'; 'b'; 'a']` et le mot vide est représenté par la liste `[ ]`.

Partie 2 : Fonctions utiles sur les listes

1. Ecrire une fonction récursive OCaml de signature `longueur : 'a list -> int` qui renvoie la longueur de la liste.
2. Ecrire une fonction OCaml de signature `sous_liste : 'a list -> int -> int -> 'a list` où `sous_liste l k long` renvoie une liste `s` qui est la sous-liste de `l` commençant à l'indice `k` et de longueur `long`. On supposera que l'indexation des listes commence à 0.

On pourra dans la suite de l'énoncé utiliser les fonctions `longueur` et `sous_liste`.

Partie 3 : Algorithme naïf

3. Préciser si les mots suivants contiennent ou non une répétition.

   (a) $aabfa$ (b) $abfdanq$ (c) $ababa$ (d) $avba$

4. Soit w un mot contenant au plus deux lettres différentes. Montrer que si $|w| \geqslant 4$ alors w contient au moins une répétition.
5. Ecrire une fonction OCaml de signature `estCarre : 'a list -> bool` prenant en argument une liste w et retournant `true` si w est un carré et `false` sinon.
6. Déterminer la complexité en nombre de comparaisons de lettres de la fonction `estCarre`.
7. Ecrire une fonction OCaml de signature `contientRepetitionAux : 'a list -> int -> bool` prenant en argument une liste w et un entier m et retournant `true` si w contient une répétition de la forme xx avec x de longueur m et `false` sinon.
8. Montrer que toute répétition d'un mot w de longueur n est de la forme xx avec $|x| \leqslant \dfrac{n}{2}$.
9. En déduire une fonction OCaml de signature `contientRepetition : 'a list -> bool` prenant en argument une liste w et retournant `true` si w contient une répétition et `false` sinon.
10. Quelle est la complexité en nombre de comparaisons de caractères de la fonction `contientRepetition` ?

Partie 4 : Algorithme de Main-Lorentz

L'algorithme de Main-Lorentz permet de détecter de manière plus efficace des répétitions d'un mot w. Il comporte essentiellement deux parties :

— la première consiste à voir si étant donné deux mots u et v, le mot uv contient un carré non nul issu de la concaténation ;
— la deuxième s'appuie sur le principe de "diviser pour régner".

Remarquons qu'un mot uv contient une répétition si et seulement si u ou v contiennent une répétition ou uv contient des répétitions provenant de la concaténation. Pour déterminer si un mot uv contient de nouvelles répétitions, on commence par effectuer des prétraitrements consistant à calculer des tables de valeurs de u et de v qui sont généralement appelées tables de préfixes (ou suffixes). Avant de présenter des algorithmes permettant de générer ces tables, on commence par justifier leur application dans la détection de répétitions.

Définition. Soient u et v deux mots. On dit que uv contient un **carré centré** sur u (respectivement sur v) s'il existe un mot w non vide et des mots u', v'', w', w'' tels que $u = u'ww'$, $v = w''v''$, $w = w'w''$ (respectivement $u = u'w'$, $v = w''wv''$, $w = w'w''$).

Définition. Soient u et v deux mots de Σ^*. Le **plus long préfixe** (respectivement suffixe) **commun** de u et v est le plus long mot w tel qu'il existe deux mots r et s tels que $u = wr$ et $v = ws$ (respectivement $u = rw$ et $v = sw$). On le note $\mathrm{lcp}(u, v)$ (respectivement $\mathrm{lcs}(u, v)$).

A propos des carrés centrés

11. Dans cette question, $\Sigma = \{a, b\}$. Soient $u = abababaa$ et $v = ababaaa$. Déterminer le plus grand préfixe commun de u et v.
12. Soient u et v deux mots de Σ^*. Montrer que uv contient un carré centré sur u si et seulement s'il existe $i \in \{0, \ldots, |u| - 1\}$ tel que

$$|\mathrm{lcs}(u[0, i-1], u)| + |\mathrm{lcp}(u[i, |u|-1], v)| \geqslant |u| - i.$$

De la même manière, on peut montrer que uv contient un carré centré sur v si et seulement s'il existe $j \in \{0, \dots, |v|-1\}$ tel que

$$|\mathrm{lcs}(v[0, j-1], u)| + |\mathrm{lcp}(v, v[j, |v|-1])| \geqslant |v| - j.$$

Ainsi, pour pouvoir déterminer s'il existe un carré centré sur u ou v, on peut utiliser les valeurs :

$$|\mathrm{lcs}(u[0, i-1], u)|, |\mathrm{lcp}(u[i, |u|-1], v)|, |\mathrm{lcs}(v[0, j-1], u)|, |\mathrm{lcp}(v, v[j, |v|-1])|.$$

Dans la suite, étant donné deux mots u et v, on note pref_u, $\mathrm{pref}_{u,v}$, suff_u, $\mathrm{suff}_{u,v}$ les tableaux vérifiant, pour tout $i \in \{0, \dots, |u|-1\}$:

$$\begin{aligned} \mathrm{pref}_u[i] &= |\mathrm{lcp}(u[i, |u|-1], u)|, \\ \mathrm{pref}_{u,v}[i] &= |\mathrm{lcp}(u[i, |u|-1], v)|, \\ \mathrm{suff}_u[i] &= |\mathrm{lcs}(u[0, i], u)|, \\ \mathrm{suff}_{u,v}[i] &= |\mathrm{lcs}(u[0, i], v)|. \end{aligned}$$

Calcul de table de préfixes

On présente ci-dessous un algorithme permettant le calcul de la table pref_u ainsi que sa complexité en nombre de comparaisons de caractères :

```
Entrées : une chaîne de caractères u
Sorties : un tableau pref_u
i ← 0, pref ← tableau de taille |u| initialisé à 0, pref[i] ← |u|, g ← 0
pour i allant de 1 à |u| − 1 faire
    si i < g et pref[i − f] < g − i alors
        pref[i] ← pref[i − f]
    fin
    sinon si i < g et pref[i − f] > g − i alors
        pref[i] ← g − i
    fin
    sinon
        (f, g) ← (i, max(g, i))
        tant que g < |u| et u[g] == u[g − f] faire
            g ← g + 1
        fin
        pref[i] ← g − f
    fin
fin
```

On admet que la complexité est de $O(|u|)$ en nombre de comparaisons de caractères.

En adaptant cet algorithme, il est également possible de calculer la table $\mathrm{pref}_{u,v}$ en $O(|u|)$ de comparaisons de caractères.

13. On pose $u = aabbba$ et $v = abbaab$. Déterminer les tableaux pref_u et $\mathrm{pref}_{u,v}$ sans justification.
14. En déroulant l'algorithme de calcul de la table pref_u donné précédemment appliqué au mot $u = aaabaaabaaab$, compléter le tableau

$i =$	f	g	pref$[i]$
0	–	0	12
1	1	3	2
2	·	·	·
⋮	⋮	⋮	⋮
11	4	12	0

de la façon suivante : pour une valeur i donnée, on indique les valeurs de $f, g, \text{pref}[i]$ à l'issue des instructions internes de la boucle.

Par exemple, à l'initialisation, $i = 0$, f n'est pas définie, g vaut 0 et $\text{pref}[0] = 12$. Pour $i = 1$, à l'issue des instructions internes à la boucle, on a $f = 1, g = 3$ et $\text{pref}[1] = 2$.

15. Déduire de l'algorithme de calcul de la table pref_u donné précédemment une procédure calculant suff_u.

Dans la suite, on suppose que l'algorithme `tabpref(u,v)` qui prend en argument deux chaînes de caractères u et v et qui renvoie la table $\text{pref}_{u,v}$ nous est donné. On admet que la complexité de cet algorithme est de $O(|u|)$ en nombre de comparaisons de caractères.

16. Déduire des questions précédentes un algorithme qui, étant donnés deux mots u et v, renvoie `VRAI` s'il existe un carré centré sur u et `FAUX` sinon.
17. Quelle est la complexité de cet algorithme en nombre de comparaisons de caractères ?

Application des tables

18. Déduire des questions précédentes un algorithme récursif qui prend en argument une chaîne de caractères et qui renvoie `VRAI` si la chaîne contient une répétition et `FAUX` sinon.
19. Déterminer la complexité de cet algorithme en nombre de comparaisons de caractères.
