Lycée Faidherbe, 2019-2020

T.P. et T.D. d'informatique

MP1 option informatique

Version du 14 mars 2020

Table des matières

TP I

Arbres AVL

Résumé

On se propose ici d'implémenter la structure d'arbre binaire de recherche en assurant une hauteur logarithmique. La structure choisie est celle définie par Georgy Adelson-Velsky and Evgenii Landis (d'où le nom AVL construit à partir des initiales) en 1962. On ajoute à un nœud l'indication de sa hauteur et on équilibre l'arbre à chaque opération.
On commencera par le dessin des arbres binaires afin de pouvoir visualiser les opérations.
La valeur des arbres sera réduite a la clé entière afin de simplifier l'étude mais la généralisation est immédiate.
On rappelle qu'un arbre est **équilibré** *si, pour tout nœud, la différence entre les hauteurs du fils droit et du fils gauche est au plus 1 en valeur absolue.*
Le but de ce TP est de permettre d'obtenir des arbres équilibrés lors de des opérations sur les arbres binaires de recherche.

Exercice 1 — Préliminaire mathématiques
Montrer que si a est un arbre équilibré de hauteur h alors il contient au moins $F_{h+3} - 1$ nœuds où (F_n) est la suite de Fibonacci définie par $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_{n+1} + F_n$.
En déduire que, pour un arbre équilibré de hauteur h et de taille n, il existe un réel A tel que $h \leqslant A \log_2(n)$.

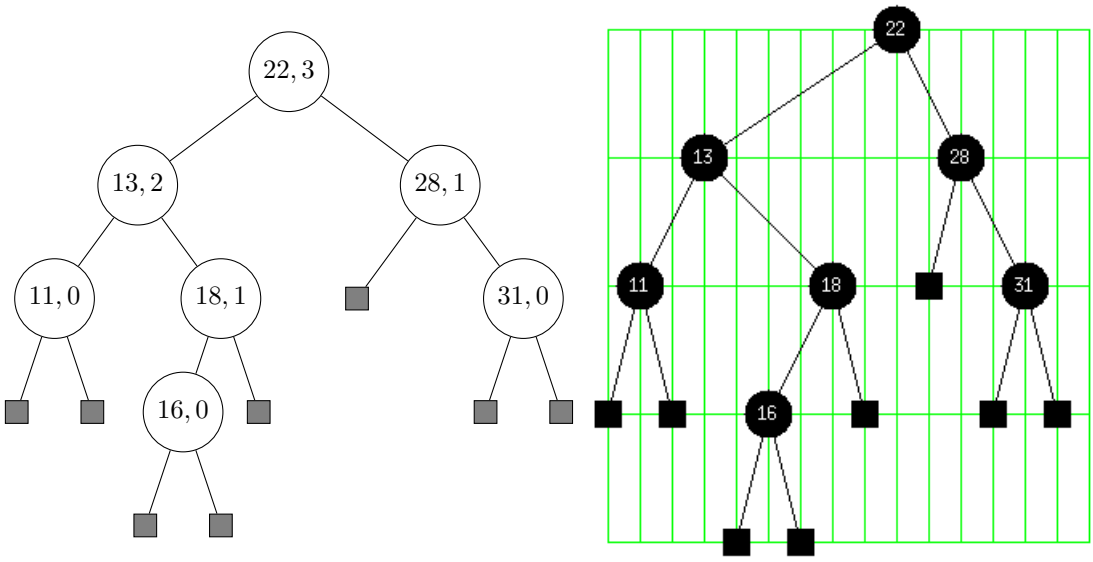
1 Dessin

On souhaite dessiner dessiner les arbres de telle manière que le parcours infixe consiste à lire les nœuds (et les feuilles) de gauche à droite ; on impose aussi que les nœuds de même profondeur soient alignés horizontalement. On placera donc les nœuds sur une grille.

Pour faire des représentations graphique on a besoin de la bibliothèque graphique.

```
#load "graphics.cma";;
open Graphics;;
open_graph " 800x900";;
```

1. La première ligne charge en mémoire le fichier de la bibliothèque.
2. La deuxième ligne permet d'utiliser les fonctions de la bibliothèque directement : on écrit `set_color` au lieu de `Graphics.set_color`
3. La troisième ligne ouvre une fenêtre graphique de 800 pixels de large et 900 pixels de haut. L'espace blanc avant le premier chiffre est obligatoire.
   L'origine est en bas à gauche.



La bibliothèque sait dessiner des segments de droites.
On a besoin des fonctions suivantes

- `set_color` sélectionne la couleur, les couleurs de base sont définies :
  `black, while, red, green, blue, yellow, cyan, magenta`.
- `set_line_width` sélectionne la largeur du trait.
- `moveto x y` déplace la position sans rien tracer vers (x, y).
- `lineto x y` trace un trait depuis la position actuelle vers (x, y) avec la couleur courante.
- `clear_graph ()` efface le dessin courant.
- `close_graph ()` ferme la fenêtre graphique.

Pour voir ce que l'on dessine le programme ne doit pas clore tout de suite. On pourra employer une fonction qui attend un clic de la souris. Il est utile ensuite de fermer la fenêtre.

```
wait_next_event [Button_down];;
close_graph()
```

Un exemple est donné dans le programme `dessin_test.ml`.

Exercice 2
Écrire une fonction qui permet la représentation du squelette d'un arbre.
On pourra donner comme paramètre les coordonnées du point supérieur gauche et considérer que les pas de la grille horizontaux (`dX`) et verticaux (`dY`) sont des variables globales (de valeurs 20 et -80, par exemple car on veut aller vers le bas).
Une idée possible est de dessiner l'arbre gauche en décalant l'ordonnée d'origine d'un pas vers le bas. Pour savoir où placer la racine, la fonction doit renvoyer le bord droit du dessin du fils.
On trace alors le fils droit en décalant encore vers la droite.
Pour tracer les segments depuis la racine, il faut aussi recevoir l'abscisse des racines des fils : la fonction de dessin, en plus de dessiner l'arbre devra renvoyer l'abscisse de la racine et l'abscisse du bord droit

Des fonctions supplémentaires possibles sont

- `fill_rect x0 y0 larg haut` qui dessine un rectangle dont le point en bas à gauche est aux coordonnées (x_0, y_0), de largeur `larg` et de hauteur `haut`. Le rectangle est rempli par la couleur courante.
- `fill_circle x0 y0 r` qui dessine un cercle rempli par la couleur courante, de centre (x_0, y_0) et de rayon `r`.
- `draw_string texte` qui écrit un texte avec la couleur courante depuis la position actuelle (elle correspond au point en bas à gauche).

Par exemple pour dessiner un disque rouge, centré en (x, y), de rayon r et marqué d'un chiffre $n < 100$, on peut écrire

```
let disqueRouge x y r n =
  set_color red;
  fill_circle x y r;
  set_color white;
  if n < 10
  then moveto (x - 2) (y - 5)
  else moveto (x - 6) (y - 5);
  draw_string (string_of_int n);;
```

Exercice 3
Écrire une fonction qui permet la représentation des nœuds d'un arbre.
On représentera les feuilles par des carrés de côté 16 et les nœuds internes par des disques de rayon 15.

Exercice 4
En déduire un tracé de l'arbre : les segments ne doivent pas passer par dessus les nœuds.
Peut-on faire la même chose en un seul passage ?

2 Équilibrage

On considère un type d'arbre binaires de recherche que l'on a augmenté de l'indication de la hauteur (en quatrième argument de la hauteur).
Le type utilisé sera

```
type arbreAVL = Vide|Noeud of  arbreAVL * int * arbreAVL * int
   ;;
```

Les constructions devront s'assurer que la hauteur est bien la valeur indiquée.

2.1 Premières fonctions

Exercice 5
Écrire une fonction qui teste l'appartenance d'une valeur à un arbre.
Il s'agit de transcrire simplement la fonction du cours.

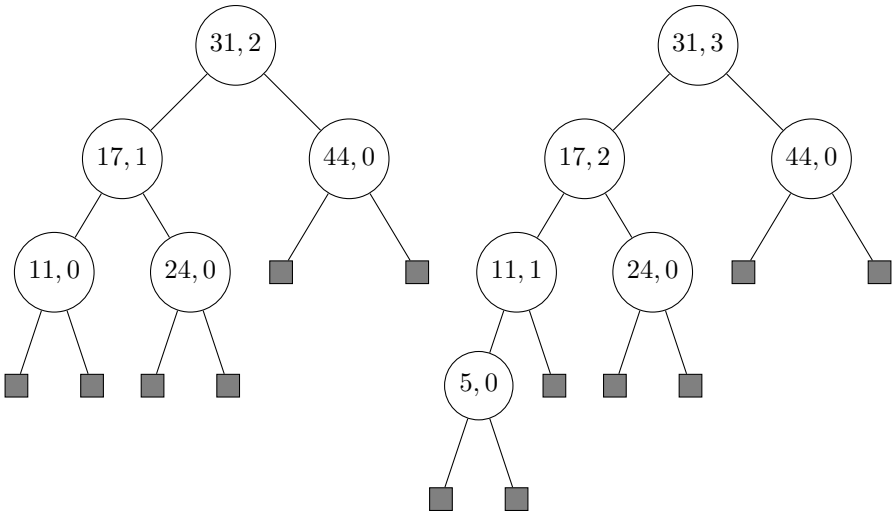
Exercice 6
Écrire une fonction qui crée une feuille à partir d'une valeur du nœud.

Exercice 7
Écrire une fonction `ht` *qui renvoie la hauteur d'un arbre.*
La fonction doit renvoyer -1 *dans le cas d'un arbre vide.*

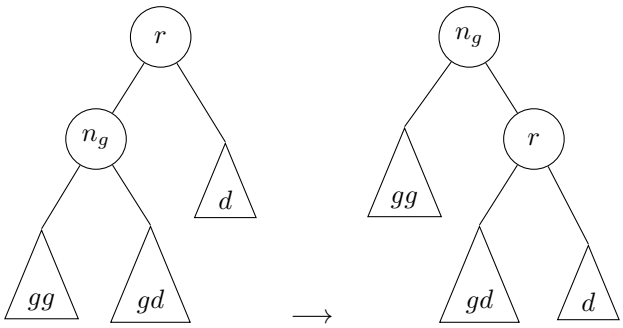
Exercice 8
Écrire une fonction `cons g n0 d` *avec* n_0 *valeur et* g *et* d *deux arbres AVL (non nécessairement équilibrés) qui renvoie un arbre AVL construit avec les paramètres (fils gauche, racine et fils droit respectivement).*

2.2 Rotations

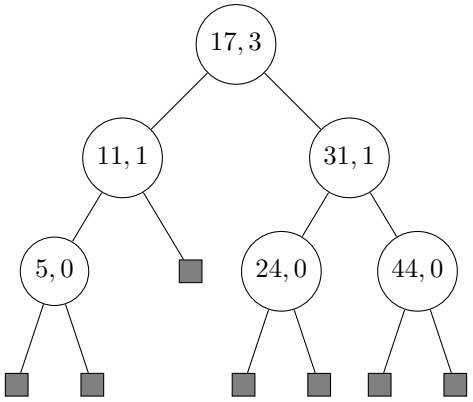
Lors de l'ajout ou le retrait d'un élément peut perturber l'équilibre.
Dans l'exemple ci-dessous, ou ajoute la valeur 5.



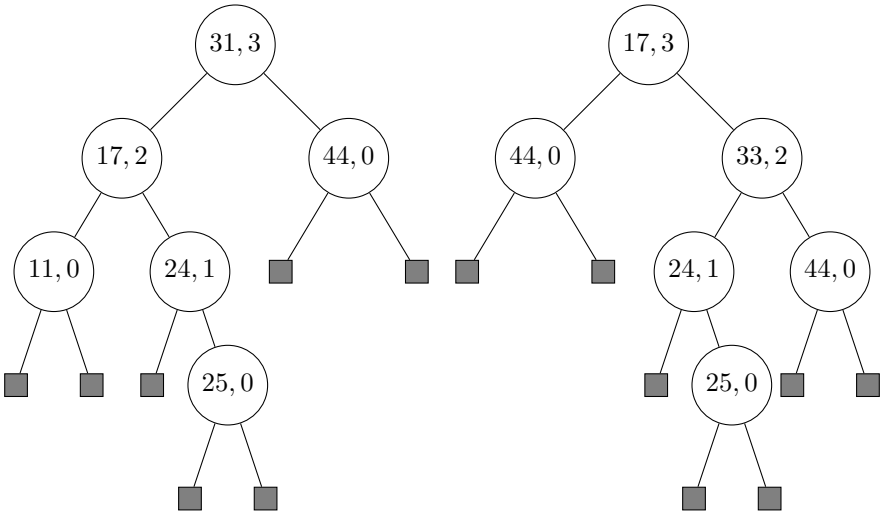
L'arbre n'est alors plus équilibré. Pour en rétablir l'équilibre on va faire tourner la liaison entre la racine et le fils gauche en faisant glisser le fils droit du fils gauche le long de cette liaison pour qu'il devienne le fils gauche de l'ancienne racine devenue fils droit. Cette opération est nommée rotation droite. On définit de même une rotation gauche.



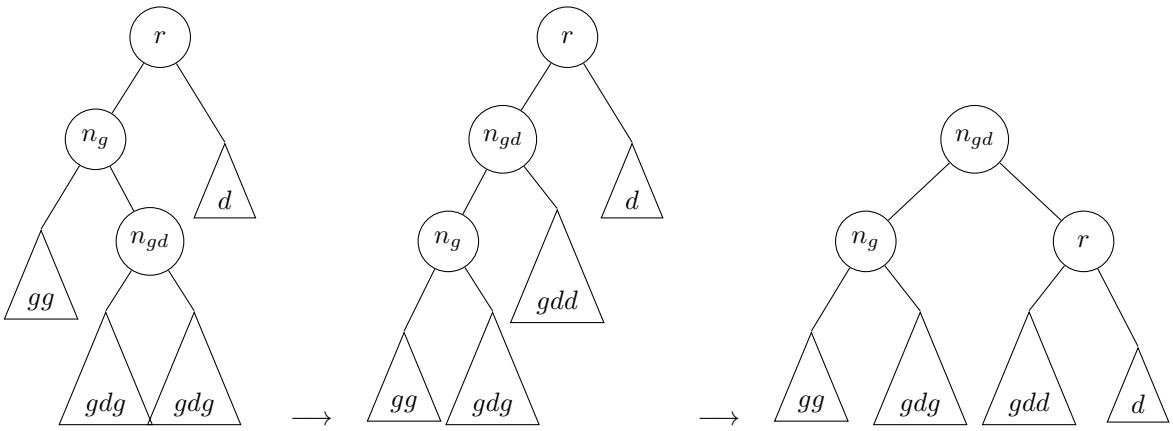
On obtient alors un arbre équilibré.



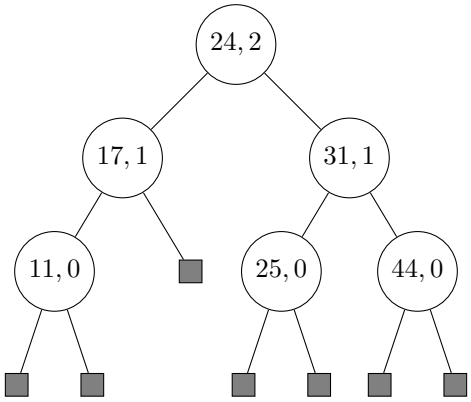
Par contre si on avait ajouté la valeur 25 à l'arbre initial on aurait obtenu aussi un déséquilibre vers la gauche mais une rotation droite n'aurait fait que le déplacer à droite.



La solution est d'effectuer une rotation gauche sur le fils gauche avant de réaliser la rotation droite.



On parvient ainsi à un arbre équilibré



Dans la pratique on n'équilibre pas l'arbre après l'avoir construit : on le construit équilibré.

Exercice 9
Écrire une fonction `noeud g n0 d` *avec* n_0 *valeur et* g *et* d *deux arbres AVL dont les hauteurs diffèrent de 2 au plus en valeur absolue, renvoie un arbre AVL contenant les valeurs des deux arbres ainsi que* n_0.

2.3 Arbres binaires de recherche

On remarque qu'une rotation d'un arbre binaire de recherche donne un arbre binaire de recherche ; une démonstration possible consiste à prouver que le parcours infixe est inchangé.
La construction ci-dessus permet donc de maintenir un arbre binaire de recherche.
Dans la suite on désignera un arbre binaire de recherche AVL équilibré simplement sous le nom **arbre AVL**.

Exercice 10
Écrire une fonction `insertion n arbre` *qui reçoit une valeur et un arbre AVL et qui renvoie un arbre AVL contenant les valeurs de l'arbre ainsi que* n_0*. On insérera la valeur dans une feuille. Si la valeur existait déjà dans l'arbre on renvoie le même arbre.*

Exercice 11
Écrire une fonction qui reçoit une liste de valeurs et qui affiche les arbres AVL construits en ajoutant successivement les valeurs de la liste à partir de l'arbre vide.

Exercice 12
Écrire une fonction `suppression n arbre` *qui reçoit une valeur et un arbre AVL et qui renvoie un arbre AVL contenant les valeurs de l'arbre à l'exception de* n_0 *s'il était présent.*

Exercice 13
Qu'est-ce qui rend difficile l'insertion à la racine ?
Proposer une fonction.

3 Solutions

Solution de l'exercice 1 - On note $\mathcal{P}(h)$ la propriété :
tout arbre équilibré de hauteur h contient au moins $F_{h+2}-1$ nœuds.
Un arbre de hauteur 0 est un nœud de fils vides donc contient 1 nœud et $F_2-1=1$.
Un arbre de hauteur 1 est un nœud dont au moins un fils est non vide : il contient au moins 2 nœuds et $F_3-1=2$. Ainsi $\mathcal{P}(0)$ et $\mathcal{P}(1)$ sont vraies.
On suppose que $\mathcal{P}(h-1)$ et $\mathcal{P}(h)$ sont vraies avec $h \geqslant 1$.
a est un arbre équilibré de hauteur $h+1$.
Ses deux fils sont de hauteur h au plus et au moins l'un des deux est de hauteur h.
Comme l'arbre est équilibré l'autre fils est de hauteur $h+\varepsilon$ avec $|\varepsilon| \leqslant 1$ donc il est de hauteur $h-1$ au moins.
De plus les deux fils sont équilibrés donc l'un contient au moins $F_{h+2}-1$ nœuds et l'autre contient au moins $F_{h-1+2}-1$ nœuds.
On en déduit que la taille de a est minorée par $1+F_{h+2}-1+F_{h+1}-1=F_{h+3}-1$.
Ainsi $\mathcal{P}(h+1)$ est vrai donc la propriété est vraie pour tout arbre.

Les racines de X^2-X-1 sont $\alpha=\frac{1+\sqrt{5}}{2}$ et $\beta=\frac{1-\sqrt{5}}{2}$ et $F_n=\frac{1}{\sqrt{5}}\left(\alpha^{n+1}-\beta^{n+1}\right)$ donc, pour un arbre équilibré on a $n \geqslant \frac{1}{\sqrt{5}}\left(\alpha^{h+3}-\beta^{h+3}\right)-1$. On en déduit

$$\frac{n}{\alpha^h} \geqslant \frac{\alpha^3}{\sqrt{5}} - \left(\frac{\beta}{\alpha}\right)^h \frac{\beta^3}{\sqrt{5}} - \frac{1}{\alpha^h} \geqslant \frac{\alpha^3}{\sqrt{5}} - \left(\frac{\beta}{\alpha}\right)^h \frac{\beta^3}{\sqrt{5}} - 1 = F_2 - 1 = 1$$

d'où $\alpha^h \leqslant n$ puis $h \leqslant \dfrac{1}{\log_2(\alpha)} \log_2(n)$.

Solution de l'exercice 2 - On écrit une fonction auxiliaire qui reçoit les coordonnées du points supérieur gauche du rectangle contenant l'arbre et qui renvoie la nouvelle abscisse pour la suite ainsi que l'abscisse de la racine.

```
let rec squelette arbre x0 y0 =
  set_color black;
  match arbre with
  |Vide -> x0, x0
  |Noeud(k, g, r, d) -> let xrg, xbg = squelette g x0 (y0 - dY
     ) in
                        let xr = xbg + dX in
                        let xrd, xbd = squelette d (xr + dX) (
                           y0 - dY) in
                        moveto xrg (y0 - dY);
                        lineto xr y0;
                        lineto xrd (y0 - dY);
                        xr, xbd;;
```

Solution de l'exercice 3 - On s'inspire du squelette.

```
let cote = 16
let rayon = 15;;

let dessinNoeud arbre x y =
(* Dessin de la racine d'un arbre à la position (x,y)*)
match arbre with
|Vide -> set_color black;
fill_rect (x-cote/2) (y-cote/2) cote cote
|Noeud(_, n, _ , _) ->
set_color black;
fill_circle x y rayon;
set_color white;
if n < 10 then moveto (x - 2) (y - 5)
else moveto (x - 6) (y - 5);
draw_string (string_of_int n);;


let rec noeuds arbre x0 y0 =
match arbre with
|Vide -> dessinNoeud Vide x0 y0; x0
|Noeud(g, n, d, h) ->
(let xg = noeuds g x0 (y0 + dY) in
let xr = xg + dX in
let xd = noeuds d (xr + dX) (y0 + dY) in
dessinNoeud arbre xr y0 ;
```

Solution de l'exercice 4 - On dessine le squelette avant les nœuds.

```
open_graph " 800x900";;
squelette a0 x0 y0;;
noeuds a0 x0 y0;;
wait_next_event [Button_down];;
close_graph ();;
```

```
let dessin arbre x0 y0 =
  let rec auxDessin arbre x y =
  match arbre with
  |Vide -> (x + dX, x)
  |Noeud(g, n, d, h) ->
    let (xg, xrg) = auxDessin g x (y + dY) in
    let (xd, xrd) = auxDessin d (xg+dX) (y + dY) in
    set_color black;
    moveto xrg (y + dY);
    lineto xg y;
    lineto xrd (y + dY);
    dessinNoeud g xrg (y + dY);
    dessinNoeud d xrd (y + dY);
    (xd, xg) in
    let max, xr = auxDessin arbre x0 y0 in
  dessinNoeud arbre xr y0;
  wait_next_event [Button_down];;
```

Solution de l'exercice 5 -

```
let rec chercher n arbre =
  match arbre with
  |Vide -> false
  |Noeud(g, r, d, h) when (n = rx) -> true
  |Noeud(g, r, d, h) when (n < r) -> chercher n g
  |Noeud(g, r, d, h) -> chercher n d;;
```

Solution de l'exercice 6 -

```
let feuille r =  Noeud(Vide, r, Vide, 0);;
```

Solution de l'exercice 7 -

```
let ht arbre =
  match arbre with
  |Vide -> -1
  |Noeud(g, r, d, h) -> h;;
```

Solution de l'exercice 8 -

```
let cons g n0 d =
  let hg = ht g in
  let hd = ht d in
  let h = (max hg hd) + 1 in
  Noeud(g, n0, d, h);;
```

Solution de l'exercice 9 -

```
let noeud g n0 d =
  let hg = ht g in
  let hd = ht d in
  match (hg - hd) with
  |2 -> begin match g with
    |Noeud(gg, rg, gd, _) when ht gg >= ht gd
      -> cons gg rg (cons gd n0 d)
    |Noeud(gg, rg, Noeud(gdg, rgd, gdd, _), _)
      -> cons (cons gg rg gdg) rgd (cons gdd n0 d)
    | _ -> failwith "Ceci ne devrait pas arriver" end
  |(-2) -> begin match d with
    |Noeud(dg, rd, dd, _) when ht dd > ht dg
      -> cons (cons g n0 dg) rd dd
    |Noeud(Noeud(dgg, rdg, dgd, _), rd, dd, _)
      -> cons (cons g n0 dgg) rdg (cons dgd rd dd)
    | _ -> failwith "Ceci ne devrait pas arriver" end
  | _ -> cons g n0 d;;
```

Solution de l'exercice 10 -

```
let rec insertion n arbre =
  match arbre with
  |Vide -> feuille n
  |Noeud(g, r, d, h) when n = r -> arbre
  |Noeud(g, r, d, h) when n < r
                  -> noeud (insertion n g) r  d
  |Noeud(g, r, d, h) -> noeud g r (insertion n d);;
```

Solution de l'exercice 11 -

```
let evolution liste =
  let rec aux liste arbre =
    match liste with
    |[] -> ()
    |n::reste -> let a1 = insertion n arbre in
                 let _ = dessin a1 x0 y0 in
                 clear_graph();
                 aux reste a1 in
  aux liste Vide;
  close_graph();;
```

Solution de l'exercice 12 -

```
let rec maxArbre arbre =
  match arbre with
  |Vide -> raise(Failure "Arbre vide")
  |Noeud(_, r, Vide, _) -> r
  |Noeud(_, _, d, _) -> maxArbre d;;

let rec suppressionMax arbre =
  match arbre with
  |Vide -> failwith "Arbre vide"
  |Noeud(g, r, Vide, h) -> g
  |Noeud(g, r, d, h) -> noeud g r (suppressionMax d);;

let suppressionRacine arbre =
  match arbre with
  |Vide -> Vide
  |Noeud(Vide, r, d, h) -> d
  |Noeud(g, r, d, h) ->  let p = maxArbre g in
                         noeud (suppressionMax g) p d;;

let rec suppression n arbre =
  match arbre with
  |Vide -> Vide
  |Noeud(g, r, d, h) when n = r
                      -> suppressionRacine arbre
  |Noeud(g, r, d, h) when n < r
                      -> noeud (suppression n g) r d
  |Noeud(g, r, d, h) -> noeud g r (suppression n d);;
```

Solution de l'exercice 13 - Lors du découpage d'un arbre on peut obtenir deux arbres dont la différence de hauteur est supérieure à 2. Pour pouvoir équilibrer on peut remplacer la fonction `noeud` en rendant récursive et en traitant les cas de différences supérieures ou égales à 2 en valeur absolue.

```
let rec noeud g n0 d =
  let hg = ht g in
  let hd = ht d in
  if (hg - hd) > 1
  then begin match g with
    |Noeud(gg, rg, gd, _) when ht gg >= ht gd
      -> noeud gg rg (noeud gd n0 d)
    |Noeud(gg, rg, Noeud(gdg, rgd, gdd, _), _)
      -> noeud (noeud gg rg gdg) rgd (noeud gdd n0 d)
    | _ -> failwith "Ceci ne devrait pas arriver" end
  else if (hg - hd) < -1
  then begin match d with
    |Noeud(dg, rd, dd, _) when ht dd > ht dg
      -> cons (cons g n0 dg) rd dd
    |Noeud(Noeud(dgg, rdg, dgd, _), rd, dd, _)
      -> cons (cons g n0 dgg) rdg (cons dgd rd dd)
    | _ -> failwith "Ceci ne devrait pas arriver" end
  else cons g n0 d;;
```

```
let rec decoupage n arbre =
  match arbre with
  |Vide -> (Vide,Vide)
  |Noeud(g, r, d, h) when n = r -> (g, d)
  |Noeud(g, r, d, h) when n < r
                      -> let (gg, gd) = decoupage n g
                          in (gg, noeud gd r d)
  |Noeud(g, r, d, h) -> let (dg, dd) = decoupage n d
                        in (noeud g r dg, dd);;

let insertionRacine n arbre =
  let (g, d)= decoupage n arbre
  in noeud g n d;;
```

TP II

Codage de Huffman

Résumé

Pour coder les chaînes de caractères, le code le plus répandu est le code ASCII[a]*. Dans ce code, chaque caractère est codé par un octet, c'est-à-dire par un mot de 8 bits. Un texte de n caractères est donc codé sur $8n$ bits.*
Il est parfois utile de comprimer le codage : une méthode possible consiste à coder les caractères les plus fréquents des mots de longueur inférieure à 8, alors que d'autres seront codés par des mots plus longs. L'idée est que la longueur moyenne d'un texte de n caractères pourrait être alors strictement inférieure à $8n$.
Une telle méthode est proposée par le codage de Huffman.

a. American Standard Computer Information Interchange

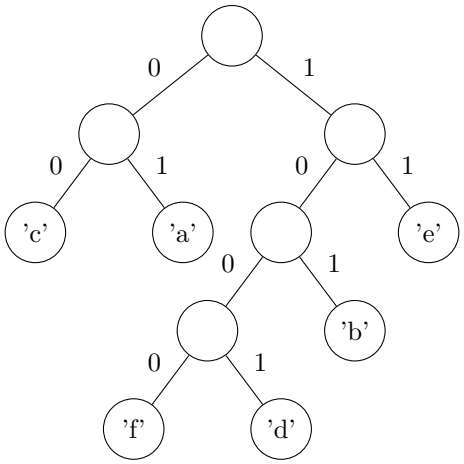
1 Arbre de Huffman

Pour qu'un codage soit utilisable il faut que le texte codé soit décodable.
La méthode de Huffman consiste à définir un code qui soit préfixe, c'est à dire qu'aucun code d'une lettre n'est le préfixe du code d'une autre lettre.
Par exemple, le code $a \mapsto 01$, $b \mapsto 101$, $c \mapsto 00$, $d \mapsto 1001$, $e \mapsto 11$, $f \mapsto 1000$ est préfixe et on peut décoder 01100010011011101001[1]

On passe d'un caractère à son code ASCII par le biais de la fonction `Char.code` ou `int_of_char`, dont l'inverse est `Char.chr` ou `char_of_int`.
N.B. Les versions récentes de OCaml utilisent le codage UTF les fonctions ci-dessus ne sont utilisables que pour les codes ASCII de 0 à 127.
Il faudra donc exclure les lettres accentuées et autres caractères spécifiques.
Les tableaux indexés par les codes pourront se limiter à une longueur 128.

Un tel code est représentable par un arbre T dont les feuilles sont étiquetées par les lettres et le code d'une lettre a, $C(a)$, est donné par le chemin qui relie la racine de T à la feuille d'étiquette a, une descente à gauche ou à droite correspondant respectivement à un caractère 0 ou 1. Le code défini ci-dessus est associé à l'arbre dessiné ci-après.
On supposera également que les nœuds internes des arbres ont exactement deux fils.

1. "afdbeac"



On ajoute aussi un poids (entier) à chaque nœud. On définit donc le type de données

```
type arbre = F of int * char | N of arbre * int * arbre;;
```

Le poids d'un arbre sera le poids de sa racine.

Exercice 1 — Poids
Écrire une fonction `poids : arbre -> int` *qui renvoie le poids d'un arbre.*
Le codage est adapté au texte à compacter.
L'arbre associé à ce codage est obtenu de la façon suivante.

1. En lisant une première fois le texte, on accorde à chaque caractère un poids : le nombre d'apparitions du caractère dans le texte. Ces poids sont stockés dans un tableau de longueur 128.
2. On construit une file de priorité initiale, dont les éléments sont les arbres de Huffman, réduits à des feuilles `N(char, poids)`, pour chaque lettre : l'**alphabet pondéré**.
3. Tant que la file contient au moins deux arbres :
   (a) on retire les deux arbres de plus faibles poids, h_1 et h_2 de poids respectifs w_1 et w_2,
   (b) on construit un nouvel arbre h (de Huffman) dont les sous-arbres gauche et droit sont h_1 et h_2 et de poids $w_1 + w_2$,
   (c) on ajoute ce nouvel arbre dans la file de priorité.
4. Quand la file est réduite à un unique arbre, on renvoie cet arbre. Le code associé est le **code de Huffman** du texte (on admet que ce code est bien optimal).

Le nombre d'éléments dans la file d'attente n'est pas très élevé, on se contentera de coder celle-ci par une liste triée par poids croissant.

Exercice 2 — Insertion
Écrire une fonction `inserer : arbre -> arbre list -> arbre list` *qui ajoute un arbre à une liste d'arbres ; on supposera que la liste initiale est triée par ordre croissant de poids et la liste renvoyée le sera également.*

Exercice 3 — Calcul des poids
Écrire une fonction `occ : string -> int array` *qui calcule le tableau des occurrences des 128 caractères dans un texte.*

Exercice 4 — Calcul de l'alphabet
Écrire une fonction `alphabet : int array -> arbre list` *qui calcule l'alphabet pondéré associé à tableau des poids.*
On ne mettra dans la liste que les caractères qui ont une fréquence non nulle.

Exercice 5 — Calcul de l'arbre
Écrire une fonction `arbre_huffman : arbre list -> arbre` *qui, appliquée à un alphabet pondéré, renvoie l'arbre du code de Huffman qui lui est associé.*

L'arbre associé à un texte sera donc calculé par

```
let t = arbre_huffman (alphabet (poids texte));;
```

Exercice 6 — Table de codage
Écrire une fonction `table_codes` *qui, appliquée à un arbre, renvoie le tableau de longueur 128 dont la i-ème entrée contient le code $C(a)$ où a est le caractère de code ASCII i. Le code sera codé sous la forme d'une chaîne de caractères* `'0'` *ou* `'1'` *et sera égal à la chaîne vide si a n'est pas codé par C.*

2 Codage-décodage

Exercice 7 — Codage avec table
Écrire une fonction `coder_table`*, qui prend en argument une table de code et une liste de caractères et qui renvoie le texte codé.*

Exercice 8 — Codage du texte
En déduire une fonction qui, à partir d'un texte, renvoie le couple formé de la table des occurrences (voir question 3) et du texte compacté.

Exercice 9 — Analyse
Quel défaut présente l'algorithme de Huffman ?
Proposer une fonction qui détermine l'efficacité de la compression.
On notera que, pour mesurer l'efficacité, la longueur de la chaîne de 0 et 1 peut être divisé par 8 car on pourra coder les 0/1 sur un seul bit au lieu d'un octet.

Exercice 10 — Décodage d'une lettre
Écrire une fonction prenant en argument un arbre de Huffman, une chaîne de 0 et 1 et un indice valide de cette chaîne et qui retourne le caractère décodé dont le codage commence à l'indice ainsi que l'indice du premier indice non utilisé.

Exercice 11 — Décodage
En déduire une fonction `decode : int array -> string -> string` *qui retourne le texte décodé à partir de la table des occurrences et du texte codé.*
`Char.escaped car` convertit un caractère en une chaîne de longueur 1.

Exercice 12 — Codage compacté
Coder un texte sous forme compactée en une chaîne de caractères.
Pour simplifier on n'utilisera que les caractères de code ASCII entre 0 et 127 donc on regroupera les chiffres 7 par 7, ce qui donne un taux de compression moins avantageux.
Pour gérer une taille qui n'est pas un multiple de 7 on pourra ajouter un caractère sentinelle au texte original, par exemple `Char.chr 0` *et ajouter des caractères 0 à la chaîne de 0 et 1 pour obtenir une longueur multiple de 7.*

3 Solutions

Solution de l'exercice 1 -

```
let poids a =
  match a with
  |F(k, c) -> k
  |N(g, k, d) -> k;;
```

Solution de l'exercice 2 -

```
let rec inserer a liste =
  match liste with
  |[] -> [a]
  |t::reste when poids(a) > poids(t) -> t :: (inserer a reste)
  |_ -> a::liste;;
```

Solution de l'exercice 3 -

```
let occ texte =
  let t = Array.make 128 0 in
  for i = 0 to (String.length u -1) do
      let k = Char.code u.[i] in
      t.(k) <- t.(k) + 1 done;
 t;;
```

Solution de l'exercice 4 -

```
let alphabet t =
 let alphabet = ref [] in
 for k = 0 to 255 do
   if t.(k)<>0
   then alphabet := inserer (F(t.(k), Char.chr k))
                            !alphabet done;
 !alphabet;;
```

Solution de l'exercice 5 -

```
let rec arbre_huffman alphabet =
  match alphabet with
  |[] -> failwith "Le texte est-il vide ?"
  |[a] -> a
  |a::b::q -> let t = N(a, (poids a) + (poids b), b) in
              arbre_huffman (inserer t q);;
```

Solution de l'exercice 6 - On définit une fonction auxiliaire qui ajoute une chaîne (qui sera `"0"` ou `"1"`) à toutes les chaînes d'une liste (les codes d'un fils).

```
let rec prefixe pref liste =
  match liste with
  |[] -> []
  |(car, ch)::reste -> (car, pref^ch)::(prefixe pref reste);;
```

On peut aussi écrire

```
let prefixe pref = List.map (fun (car, ch) -> (car, pref^ch))
   ;;
```

On commence par lire l'arbre pour obtenir la liste des couples caractère-code.

```
let rec lecture arbre =
  match arbre with
  |F(_, ch) -> [(ch, "")]
  |N(g, _, d) -> (prefixe "0" (lecture g))
                 @(prefixe "1" (lecture d));;
```

On peut alors remplir un tableau

```
let table_code arbre =
  let liste = lecture arbre in
  let c = Array.make 128 "" in
  let rec appliquer liste =
    match liste with
    |[] -> ()
    |(car, code)::reste -> let k = Char.code car in
                           c.(k) <- code;
                           appliquer reste in
  appliquer liste;
  c;;
```

On peut aussi écrire

```
let table_code arbre =
  let liste = lecture arbre in
  let c = Array.make 128 "" in
  List.iter (fun (a, b) -> c.(Char.code a) <- b) liste;
  c;;
```

Solution de l'exercice 7 -

```
let coder_table table  texte =
  let n = Array.length texte in
  let comp = ref "" in
  for i = 0 to (n-1) do
    let k = Char.code texte.[i] in
    comp := !comp^table.(k) done;
  !comp;;
```

Solution de l'exercice 8 -

```
let codage texte =
  let t = occ texte in
  let arbre  = arbre_huffman (alphabet t) in
  let table = table_code (lecture arbre) in
  t, (codage_table table texte);;
```

Solution de l'exercice 9 - On doit lire le texte deux fois, cela implique que l'on doit connaître tout le texte **avant** de le compresser. On ne peut donc pas compresser "à la volée".

```
let taux_compression texte =
  let _, comp = codage text in
  100*(String.length v)/(String.length u)/8;;
```

Donne le pourcentage de compression (un entier).

Solution de l'exercice 10 -

```
let decode huff ch i =
  let rec descente arbre ind =
     match arbre with
     |F(_, car) -> car
     |N(g, _, _) when ch.[ind] = '0' -> descente g (ind + 1)
     |N(_, _, d) -> descente d (ind + 1) in
  descente huff i;;
```

Solution de l'exercice 11 -

```
let decode t comp =
  let texte = ref "" in
  let n = String.length comp in
  let ind = ref 0 in
  let a = arbre_huffman (alphabet t) in
  while !ind < n do
    let car, k = decode a comp !ind in
    ind := k;
    texte := !texte ^ (Char.escaped car) done;
  !texte;;
```

Solution de l'exercice 12 -

TP III

Tas persistants

Résumé

Dans le cours nous avons défini le type de tas qui a été implémenté à l'aide d'un tableau. On a ainsi utilisé une structure de données itératives, c'est-à-dire non persistante.
Nous allons dans ce travail utiliser le type récursif d'arbre croissant récursive et y définir les fonctions des files de priorité.

1 Cas général

On veut implémenter le type de données abstraite de file de priorité. On manipule des ensembles de couples *clé*×*valeur* où la clé est un entier et la valeur est de type α (`'a`).
On a donc besoin d'un type `'a pQueue` avec les fonctions

1. `createPQ : 'a -> 'a pQueue`,
2. `isEmptyPQ : 'a pQueue -> bool`,
3. l'ajout est `add : 'a pQueue -> 'a -> int -> 'a pQueue`,
   on crée un nouvel objet
4. `next : 'a pQueue -> 'a * int` ,
   permet de voir l'élément de priorité minimale (de clé minimale) et sa priorité,
5. le retrait de l'élément prioritaire est `remove : 'a pQueue -> 'a pQueue`.

Pour simplifier l'étude on ne manipulera ici que des données vides, le couple sera réduit à sa clé.
Le type sera simplement

```
type pQueue = Vide|Noeud of pQueue*int*pQueue
```

On devra maintenir la condition de croissance : les valeurs des fils sont supérieures à la valeur du père.

Exercice 1 — Premières fonctions
Écrire les fonctions `createPQ`, `isEmptyPQ` *et* `next`.

On suppose donnée une fonction `fusion pQueue -> pQueue -> pQueue` qui calcule un nouveau tas à partir de deux tas en conservant les éléments.

Exercice 2 — Dernières fonctions
Écrire les fonctions `add` *et* `remove`.

2 Tas auto-équilibrants

Pour la fusion, le cas où un des tas est vide est évident et, sinon, la racine doit être la plus petite des deux racines. Il reste alors 3 arbres :

1. les deux fils de l'arbre de racine minimale,
2. l'autre arbre.

On en choisit deux que l'on fusionne et on reconstitue un arbre.

Les tas **maxiphobiques**[1] choisissent de fusionner les deux arbres de tailles minimales. Pour les définir, on change le type pour qu'il contienne la taille de l'arbre. On prouve alors que la complexité de la fusion est logarithmique par rapport à la taille.

En 1986, Sleator et Tarjan ont proposé de considérer les deux fils d'un arbre comme une sorte de file d'attente : dans la situation décrite ci-dessus, le fils gauche est associé à l'autre tas et le fils droit devient un fils gauche

Exercice 3 — Fusion
Écrire La fonction fusion correspondante.

La complexité est alors difficile à calculer, en fait on ne peut pas assurer une complexité logarithmique à chaque opération. Cependant on peut assurer une complexité amortie logarithmique.
La complexité amortie se calcule en calculant la moyenne des complexités pour une suite d'opérations.
On compte la complexité de `fusion` en nombre d'appels récursifs à cette fonction `fusion`.
Dans le cas des tas auto-équilibrants on sépare les nœuds en 2 catégories :

1. les nœuds lourds ont le fils gauche de taille strictement supérieure à celle du fils droit,
2. les nœuds légers ont le fils droit de taille supérieure à celle du fils gauche.

Pour calculer la complexité amortie on va donner, à chaque appel initial à la fonction `fusion`, un certain nombre de "*crédits de complexité*" qui ne sont pas forcément dépensés lors de la fonction. De fait, chaque nœud lourd devra être muni d'un crédit à dépenser.

Exercice 4 — Cas des nœuds lourds
Montrer que si le nœud de taille minimale est lourd, le résultat d'une fusion est un nœud léger.

Ainsi l'étape de fusion avec décomposition d'un nœud lourd utilise le crédit du nœud. Il faut donc sufisamment de crédit pour la fusion qui décomposent un nœud léger. Il faut, dans ce cas, 2 crédits :

1. un pour payer la fusion,
2. un autre pour munir le nœud construit, qui peut être lourd.

Exercice 5 — Nombre de fusions avec nœuds légers
Montrer que le nombre de fusions qui décomposent un nœud léger lors de l'appel de fusion à deux tas de tailles respectives n_1 et n_2 est au plus $\log_2(n_1) + \log_2(n_2) + 2$.

Exercice 6 — Crédits à fournir
En déduire qu'il suffit de créditer chaque appel initial à la fonction `fusion` *de* $4\big(\log_2(n)+1\big)$ *crédits pour que la fusion soit toujours possible ; n est la somme des taille des arbres à fusionner.*

1. qui n'aiment pas les gros

3 Calcul des nombres de Hamming

Les nombres de Hamming sont les entiers qui n'admettent que 2, 3 et 5 comme diviseurs premiers, ce sont les entiers de l forme $n = 2^a.3^b.5^c$.
On remarque que si x est un nombre de Hamming alors $2x, 3x, 5x$ le sont aussi, et que tous les nombres de Hamming à part 1 s'obtiennent en multipliant par 2 ou 3 ou 5 un nombre de Hamming plus petit.
Pour obtenir les nombres de Hamming successifs, on peut

1. Initialiser une file de priorité avec le nombre 1,
2. répéter n fois : extraire le premier élément de la file, x et insérer $2x$, $3x$ et $5x$ dans la file.

Exercice 7
Programmer cet algorithme pour calculer le 1000-ième nombre de Hamming

Exercice 8
Normalement le 1000-ième nombre de Hamming est 51 200 000*, mais ce n'est pas ce que l'on a trouvé. En effet les éléments sont insérés plusieurs fois. Comment l'éviter ?*

4 Nombres taupins

Dans les temps anciens, chaque taupin avait un numéro secret, son **nombre taupin**.
Les premiers étudiants avaient le numéro 1.
Pour les étudiants suivant le nombre était attribué par le parrain.

- Si le parrain, de numéro n, est carré il attribue le numéro $2n$ à son filleul.
- Si le parrain est cube (de numéro n) il attribue le numéro $3n - 1$ à son filleul : un cube sait faire 2 opérations à la fois.

On peut remarquer que tous les entiers ne sont pas attribués : par exemple 7 ne peut pas être de la forme $2n$ ni de la forme $3n - 1$.
La question se pose de savoir si un entier est un nombre taupin possible : par exemple 1000 est obtenu par la suite $1 \to 2 \to 5 \to 14 \to 28 \to 56 \to 167 \to 500 \to 1000$.
On peut remarquer que toutes les suites commencent par $1 \to 2$.

Exercice 9 — Test de taupinalité
Écrire une fonction `est_taupin(n)` qui renvoie `true` ou `false` selon que n est ou non un nombre taupin.

On aurait aimé pouvoir suivre la généalogie d'un nombre taupin, en calculant le nombre taupin du parrain. malheureusement il peut exister des nombres qui sont obtenus de plusieurs manières. Par exemple
$1 \to 2 \to 4 \to 8 \to 16 \to 32$ et $1 \to 2 \to 4 \to 11 \to 32$.

Exercice 10 — Nombre de filières
Écrire une fonction `filieres n` qui renvoie le nombre de manières (qui peut être 0) d'obtenir k à partir de 2 avec les transformations $k \mapsto 2k$ et $k \mapsto 3k - 1$ pour tout k compris entre 0 et n.

Exercice 11 — 3 filières
Déterminer le premier nombre taupin qui peut peut l'être avec 3 chemins.

Exercice 12 — 4 filières
Déterminer le premier nombre taupin qui peut peut l'être avec 4 chemins.

5 Solutions

Solution de l'exercice 1 -

```
let createPQ () = Vide;;

let isEmptyPQ tas = tas = Vide;;

let next tas =
  match tas with
  |Vide -> failwith "Le tas est vide"
  |Noeud(_, n, _) -> n;;
```

Solution de l'exercice 2 -

```
let add tas n = fusion tas (Noeud(Vide, n, Vide));;

let remove tas =
  match tas with
  |Vide -> failwith "Le tas est vide"
  |Noeud(g, _, d) -> fusion g d;;
```

Solution de l'exercice 3 -

```
let rec fusion tas1 tas2 =
  match tas1, tas2 with
  |Vide, _ -> tas2
  |_, Vide -> tas1
  |Noeud(g1, n1, d1), Noeud(g2, n2, d2)
       -> if n1 < n2
          then Noeud(d1, n1, fusion g1 tas2)
          else Noeud(d2, n2, fusion g2 tas1);;
```

Solution de l'exercice 4 - Si $|g1| > |d1|$ alors la taille de `fusion g1 tas2` est de taille supérieure à celle de $d1$ donc de celle de g_1 d'où `Noeud(d1, n1, fusion g1 tas2)` est léger.

Solution de l'exercice 5 - Dans le cas où `t = Noeud(g1, x, d)` est léger alors $|g_1| \leqslant \frac{1}{2}|t|$. Ainsi, lors de chaque fusion la taille d'une partie diminue d'un facteur 2. Cela ne peut se produire que $\log_2(n_1) + 1 + \log_2(n_2) + 1$ fois.

Solution de l'exercice 6 - $\log_2(n_1) \leqslant \log_2(n_1 + n2) = \log_2(n)$

Solution de l'exercice 7 -

```
let rec hamming n =
  let rec aux n file =
    match n with
    |1 -> next file
    |n -> let p = next file in
          aux (n-1) (add (add (add (remove file) (2*p)) (3*p))
              (5*p))
  in aux n (add (createPQ()) 1);;
```

Solution de l'exercice 8 - x est un nombre de HAMMING extrait de la file.
Si x est divisible par 5, $x = 5y$, alors $2x = 5.2y$ a été inséré auparavant car $2y < x$.
De même $3x = 5.3y$ est aussi présent. Il est donc inutile d'insérer ces nombres, seul $5x$ est nouveau.
Lorsque x n'est pas divisible par 5 mais l'est par 3 alors un raisonnement analogue montrer qu'il est inutile d'insérer $2x$.

```
let rec hamming n =
  let rec aux n file =
    match n with
    |1 -> next file
    |n -> let p = next file in
          if p mod 5 = 0
          then aux (n-1) (add (remove file) (5*p))
          else if p mod 3 = 0
               then aux (n-1) (add (add (remove file) (3*p))
                  (5*p))
               else aux (n-1) (add (add (add (remove file) (2*
                  p)) (3*p)) (5*p))
  in aux n (add (createPQ()) 1);;
```

Solution de l'exercice 9 -

```
let rec est_taupin n =
  if n <= 2
  then true
  else begin
    if n mod 2 = 0
    then begin
      if n mod 3 = 2
      then est_taupin (n/2) || est_taupin ((n+1)/3)
      else est_taupin (n/2) end
    else begin
      if n mod 3 = 2
      then est_taupin ((n+1)/3)
      else false end end;;
```

Solution de l'exercice 10 -

```
let rec filieres n =
  if n <= 2
  then 1
  else begin
    if n mod 2 = 0
    then begin
      if n mod 3 = 2
      then filieres (n/2) + filieres ((n+1)/3)
      else filieres (n/2)
      end
    else begin
      if n mod 3 = 2
      then filieres ((n+1)/3)
      else 0
      end
    end;;
```

Solution de l'exercice 11 -

```
let n = ref 1;;
while filieres !n < 4 do n := !n + 1 done;;
print_int !n;;
```

On trouve 20480

Solution de l'exercice 12 - Ici le calcul est long, 30 minutes. On peut accélérer en employant une file de priorité.

```
let k_filieres k =
  let rec aux nb n file =
    if nb = k
    then n
    else begin
      let n1 = next file in
      let new_f = add (add (remove file) (2*n1)) (3*n1 - 1) in
      if n1 = n
      then aux (nb + 1) n new_f
      else aux 1 n1 new_f end
  in aux 1 1 (add (createPQ()) 2);;
```

On trouve 3 988 094 144 mais il faut 7 minutes !

TP IV

Logique

1 Présentation du problème

On se propose ici de gérer les propositions logiques de manière élémentaire et d'étudier le problème de la satisfiabilité.
Quelques applications ludiques sont proposées en conclusion.

- Les propositions booléennes seront codées à l'aide du type

```
type propBool =  X of int
                |Non of propBool
                |Ou of propBool*propBool
                |Et of propBool*propBool;;
```

- `X k` désignera une variable booléenne indexée par k que l'on notera x_k.
  Nous n'utiliserons que des variables indexées par des entiers supérieurs ou égaux à 1.
- Par exemple la proposition $x_1 \vee \big(\neg(x_2 \vee \neg x_1) \wedge x_3\big)$ sera comprise par

```
let p_ex = Ou (X 1, Et (Non (Ou (X 2, Non (X 1))), X 3));;
```

- Si les variables booléennes ont un indice majoré par n on représentera une valuation par un tableau de booléens de taille n, `val`. La valeur attribuée à `X k` sera `val.(k-1)`.

2 Outils

On veut pouvoir écrire les expression booléennes de manière plus lisible.
Par exemple, la formule `p_ex` pourra être imprimée sous la forme

```
x1 V (!(x2 V !x1) & x3))
```

Exercice 1
Écrire une fonction `montrer` *qui permet cette visualisation.*
La concaténation de chaînes se fait avec `str1^str2`
et on convertit un entier en chaîne par `string_of_int`.

Exercice 2
Écrire une fonction `multiOu` *de type* `propBool List -> propBool` *qui, en recevant une liste non vide de propositions logiques* `[P1; P2; ...; Pn]`*, renvoie la disjonction des* P_i.
Écrire de même une fonction `multiEt`

Exercice 3
Écrire une fonction `implique` *qui calcule une proposition logiquement équivalente à* `P1` $\Rightarrow$ `P1` *à partir de deux propositions* `P1` *et* `P2`.
Écrire de même des fonction `equivaut` *et* `ouExclusif`.

Exercice 4
Écrire une fonction (récursive) `uneSeule` *qui prend pour argument une liste de propositions et calcule une proposition qui est vraie si et seulement s'il y a exactement une des propositions de la liste qui est vraie.*

3 Évaluation

Exercice 5
Écrire une fonction `eval` *qui renvoie la valeur booléenne d'une proposition booléenne pour une valuation. Par exemple*

```
eval p_ex [|true; false; true|];;
#- : bool = true
```

Exercice 6
Écrire une fonction `indiceMax` *qui renvoie l'indice de variable maximal rencontré dans une proposition logique.*

Pour pouvoir décrire toutes les valuations nous allons utiliser une fonction `incremente` qui modifie un tableau de booléens (sans renvoyer sa valeur) et qui renvoie `true` sauf quand le tableau n'est pas incrémentable. On suit l'algorithme qui correspond à l'incrémentation d'un entier exprimé en base 2 :

- on modifie les indices les plus petits de `true` à `false`
- s'il existe, le premier `false` est modifé en `true` et la fonction renvoie `true`
- s'il n'y avait que des `true` la fonction renvoie `false` pour signifier que toutes les valuations ont été parcourues.

Exercice 7
Écrire la fonction `incremente`.

Exercice 8
Écrire une fonction `possibles` *qui renvoie la liste des tableaux satisfaisant une proposition logique. On copiera les tableaux à l'aide de la fonction* `Array.copy`.

Exercice 9
Écrire deux fonctions `satisfiable` *et* `tautologie` *qui déterminent respectivement si une proposition logique est satisfiable et si c'est une tautologie.*

Exercice 10
Vérifier que les propositions suivantes sont des tautologies.

1. $\neg\neg x_1 \Leftrightarrow x_1$
2. $(x_1 \Rightarrow x_2) \Leftrightarrow (\neg x_2 \Rightarrow \neg x_1)$
3. $(x_1 \Rightarrow x_2) \wedge (x_2 \Rightarrow x_3) \Rightarrow (x_1 \Rightarrow x_3)$

4 Exemples

4.1 Le concours

Vous êtes admissible à l'Université de Logique Médiévale (ULM) et vous devez passer l'épreuve de logique pratique. Depuis le Hall d'entrée il y a 5 couloirs et vous savez qu'un seul mène à la salle d'examen.
Chaque couloir est indiqué par un panneau et vous savez aussi que le panneau qui mène à votre salle dit la vérité mais que les autres peuvent mentir.
Voici ce que disent les cinq panneaux.

1. Ce couloir vous mène à l'échec mais pas le quatrième.
2. C'est un chemin de numéro impair qui mène dans la salle.
3. Le deuxième panneau dit la vérité ou le cinquième ment.
4. Ce panneau ment mais pas le premier.
5. Le troisième panneau ment.

Exercice 11
Comment arriver à votre salle ?

4.2 Le cadeau

Vous êtes admis dans l'école de vos rêves ! Pour vous accueillir les camarades de deuxième année vous offrent un Kinder. Mais il faut le mériter : vous avez devant vous 9 boîtes dont une seule contient l'œuf. De plus chacune a une étiquette et votre parrain vous annonce que la boîte qui contient votre cadeau porte une étiquette qui dit la vérité les autres peuvent mentir.
Voici les énoncés.

1. C'est une boîte de numéro impair qui contient la récompense
2. Ouvre-moi si tu veux du chocolat.
3. La cinquième étiquette dit la vérité ou la septième ment. Mais la neuvième ment.
4. La première étiquette ment.
5. Parmi la deuxième et la quatrième étiquette, il y en a au moins une qui dit la vérité.
6. La troisième étiquette ment.
7. La première boîte est vide.
8. Je ne dis pas la vérité et la neuvième boîte contient le cadeau.
9. Il ne faut pas croire la sixième étiquette.

Exercice 12
Quelle boîte ouvrez-vous ?

5 Solutions

Solution de l'exercice 1 -

```
let symOu = " V ";;
let symEt = " & ";;
let symNon = "!";;
let rec montrer X =
  match X with
  |Var(k) -> "x"^(string_of_int k)
  |Non(q) -> symNon^(montrer q)
  |Ou(q,r) -> "("^(montrer q)^symOu^(montrer r)^")"
  |Et(q,r) -> "("^(montrer q)^symEt^(montrer r)^")";;
```

Solution de l'exercice 2 -

```
let rec multiOu listeX =
  match listeX with
  |[] -> failwith "La liste est vide"
  |[p] -> p
  |t::q -> Ou(t, multiOu q);;

let rec multiEt listeX =
  match listeX with
  |[] -> failwith "La liste est vide"
  |[p] -> p
  |t::q -> Et(t, multiEt q);;
```

Solution de l'exercice 3 -

```
let implique X q = Ou(Non p, q);;

let equivaut X q = Ou(Et(p, q), Et(Non p, Non q));;

let ouExclusif X q = Ou(Et(p, Non q), Et(Non p, q));;
```

Solution de l'exercice 4 -

```
let rec uneSeule listeX =
  match listeX with
  |[] -> failwith "La liste est vide"
  |[p] -> p
  |t::q -> Ou(Et(Non t, uneSeule q), Et(t, Non (multiOu q)));;
```

Solution de l'exercice 5 -

```
let rec eval X t =
  match X with
  |Var(k) -> t.(k-1)
  |Non(q) -> not (eval q t)
  |Ou(q,r) -> (eval q t) || (eval r t)
  |Et(q,r) -> (eval q t) && (eval r t);;
```

Solution de l'exercice 6 -

```
let rec indiceMax X =
  match X with
  |Var(k) -> k
  |Non(q) -> indiceMax q
  |Ou(q,r) -> max (indiceMax q) (indiceMax r)
  |Et(q,r) -> max (indiceMax q) (indiceMax r);;
```

Solution de l'exercice 7 -

```
let incremente t =
  let rec aux_inc pos =
    if pos = Array.length t
    then false
    else if t.(pos)
         then (t.(pos) <- false;  aux_inc (pos+1))
         else (t.(pos) <- true; true)
  in aux_inc 0;;
```

Solution de l'exercice 8 -

```
let possibles X =
  let n = indiceMax X in
  let t = Array.make n false in
  let sol = ref [] in
  if eval X t then sol := t::(!sol);
  while (incremente t)
  do if eval X t
     then sol := (Array.copy)::(!sol) done;
  !sol;;
```

Solution de l'exercice 9 -

```
let satisfiable X = possibles p <> [];;

let tautologie X = not (satisfiable (Non(p)));;
```

Solution de l'exercice 10 -

```
let x1 = X 1;;
let x2 = X 2;;
let x3 = X 3;;

let p1 = implique (Non(Non x1)) x1;;
let p2 = equivaut (implique x1 x2)
                  (implique (Non x2) (Non x1));;
let p3 = implique (Et (implique x1 x2,implique x2 x3))
                  (implique x1 x3);;

tautologie p1;;
tautologie p2;;
tautologie p3;;
```

Solution de l'exercice 11 -

```
let x1 = X 1;;
let x2 = X 2;;
let x3 = X 3;;
let x4 = X 4;;
let x5 = X 5;;
let p1 = X 6;;
let p2 = X 7;;
let p3 = X 8;;
let p4 = X 9;;
let p5 = X 10;;

let c1 = implique x1 p1;;
let c2 = implique x2 p2;;
let c3 = implique x3 p3;;
let c4 = implique x4 p4;;
let c5 = implique x5 p5;;

let u = uneSeule [x1; x2; x3; x4; x5];;

let e1 = equivaut p1 (Et (Non x1, x4));;
let e2 = equivaut p2 (multiOu [x1; x3; x5]);;
let e3 = equivaut p3 (Ou (p2, Non p5));;
let e4 = equivaut p4 (Et (Non p4, p1));;
let e5 = equivaut p5 (Non p3);;

let p = multiEt [c1; c2; c3; c4; c5; u; e1; e2; e3; e4; e5];;

possibles jeu1;;
```

Prendre le troisième couloir.

Solution de l'exercice 12 -

```
let x1 = X 1;;
let x2 = X 2;;
let x3 = X 3;;
let x4 = X 4;;
let x5 = X 5;;
let x6 = X 6;;
let x7 = X 7;;
let x8 = X 8;;
let x9 = X 9;;
let p1 = X 10;;
let p2 = X 11;;
let p3 = X 12;;
let p4 = X 13;;
let p5 = X 14;;
let p6 = X 15;;
let p7 = X 16;;
let p8 = X 17;;
let p9 = X 18;;

let c1 = implique x1 p1;;
let c2 = implique x2 p2;;
let c3 = implique x3 p3;;
let c4 = implique x4 p4;;
let c5 = implique x5 p5;;
let c6 = implique x6 p6;;
let c7 = implique x7 p7;;
let c8 = implique x8 p8;;
let c9 = implique x9 p9;;

let u = uneSeule [x1; x2; x3; x4; x5; x6; x7; x8; x9];;

let e1 = equivaut p1 (multiOu [x1; x3; x5; x7; x9]);;
let e2 = equivaut p2 x2;;
let e3 = equivaut p3 (Et (Ou (p5, Non p7), Non p9));;
let e4 = equivaut p4 (Non p1);;
let e5 = equivaut p5 (Ou (p2, p4));;
let e6 = equivaut p6 (Non p3);;
let e7 = equivaut p7 (Non x1);;
let e8 = equivaut p8 (Et (Non p8, x9));;
let e9 = equivaut p9 (Non p6);;

let jeu3 = multiEt [c1; c2; c3; c4; c5; c6; c7; c8; c9; u; e1;
    e2; e3; e4; e5; e6; e7; e8; e9];;

possibles jeu3;;
```

La septième.

TP V

Coloriages

Préliminaires

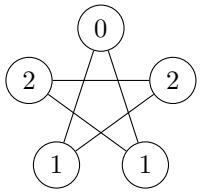
- On se fixe dans cet énoncé une représentation des graphes par matrices d'adjacence de booléens. Un graphe non-orienté $G = (S, A)$ avec $S = \{0, \ldots, n-1\}$ est représenté par une valeur `gphe` de type `bool array array` noté `graphe` telle que pour tous sommets $i, j \in S$, on ait `gphe.(i).(j) = true` si et seulement si $(i, j) \in A$.
- Étant donné un graphe $G = (S, A)$,
  le **sous-graphe induit** par un ensemble de sommets $T \subseteq S$ est $(T, A \cap (T \times T))$.
- On note $V(s)$ l'ensemble des voisins de s. Le **degré** $d(s)$ de s est le cardinal de $V(s)$.
  Le **degré** $d(G)$ de G est le maximum des degrés de ses sommets.
- Un graphe est dit **étiqueté** lorsque l'on dispose d'une fonction, dite d'étiquetage, de l'ensemble de ses sommets vers l'ensemble des entiers ; il est défini par un tableau d'entiers dont le type sera noté `etiquetage`.
- On dit qu'une fonction d'étiquetage L est un **coloriage** des sommets de $G = (S, A)$ lorsque deux sommets voisins ont toujours deux étiquettes distinctes (alors appelées **couleurs**), c'est-à-dire lorsque L vérifie $\forall s, t \in S, (s, t) \in A \Rightarrow L(s) \neq L(t)$
- Un graphe G est dit k-coloriable s'il admet un coloriage avec au plus k couleurs.
  Un graphe est dit colorié s'il est $k - coloriable$ pour un $k > 0$.
- Le **nombre chromatique** d'un graphe non-orienté G, noté $\chi(G)$, est le nombre minimal k tel que G est k-coloriable.

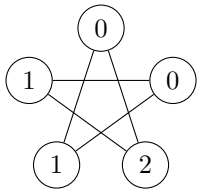
Pour définir un graphe induit de (S, A) avec $S = \{0, \ldots, n-1\}$, on définit une partie de S par un tableau `sg` de taille $p \leqslant n$ à valeurs dans S et sans répétition. On obtient ainsi une application injective de $T = \{0, 1, \ldots, p-1\}$ dans S.

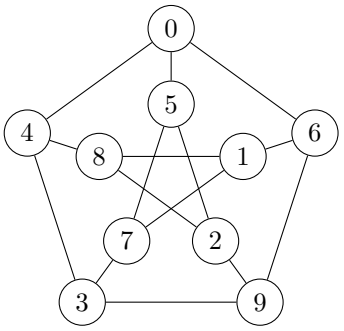
Exercice 1
Écrire une fonction `sous_graphe : graphe -> int array -> graphe` *telle que*
`sous_graphe gphe sg` *renvoie la matrice d'adjacence du graphe de sommets T qui a une arête entre les sommets* `s` *et* `t` *si et seulement si* `gphe.(sg.(s)).(sg.(t)) = true`.

1 Coloriage

Exercice 2
Indiquer, pour chacun des deux premiers graphes suivants, s'il est colorié.







Exercice 3
Donner le nombre chromatique, ainsi qu'un exemple de coloriage correspondant, pour le **graphe de Petersen** *(le troisième) ci-dessus, les numéros sont les sommets*

Exercice 4
Écrire une fonction `est_col : graphe -> etiquetage -> bool`*, telle que* `est_col gphe etiq` *renvoie* `true` *si et seulement si* `etiq` *est un coloriage de* `gphe`*. Dans le cas où la taille de l'étiquetage est strictement inférieure au nombre de sommets du graphe, la fonction renvoie* `false`*.*
On demande une complexité quadratique en le nombre de sommets du graphe.

Exercice 5
Démontrer que le calcul du nombre chromatique d'un graphe peut s'effectuer en temps exponentiel en le nombre de sommets.

Exercice 6
Soit $k > 0$. Montrer que si G est $(k+1)$-coloriable, alors pour tout sommet s de G le sous-graphe induit par $V(s)$ est k-coloriable.

2 2-coloriage

On considère dans cette partie le cas du 2-coloriage.

Exercice 7
Démontrer qu'un graphe G est 2-coloriable si et seulement s'il est **biparti***, c'est-à-dire si l'ensemble de ses sommets S peut être divisé en deux sous-ensembles disjoints T et U, tels que chaque arête a une extrémité dans T et l'autre dans U.*

Pour programmer la vérification de la 2-colorabilité d'un graphe on effectue un parcours du graphe en profondeur au cours duquel on construit une 2-coloration du graphe.

1. On étiquette tous les sommets à -1.
2. On choisit un sommet s d'étiquette -1.
3. On colorie les sommets rencontrés lors du parcours en profondeur à partir de s, en alternant entre les couleurs 0 et 1 à chaque incrémentation de la profondeur, et en vérifiant si les sommets déjà coloriés rencontrés sont d'une couleur compatible.
4. Enfin, s'il reste des sommets d'étiquette -1, alors on revient au point (2).

Exercice 8
Écrire une fonction `deux_col : graphe -> etiquetage` *telle que* `deux_col gphe` *renvoie un* 2*-coloriage de* `gphe` *si* `gphe` *est* 2*-coloriable. Le coloriage utilisera les couleurs* 0 *et* 1*. On demande une complexité quadratique en le nombre de sommets du graphe. Le comportement de la fonction est laissé libre lorsque* `gphe` *n'est pas* 2*-coloriable.*

3 Algorithmes gloutons

Dans cette partie, nous allons étudier deux algorithmes permettant de colorier un graphe en temps polynomial, mais donnant en général un coloriage sous-optimal : le coloriage obtenu peut dans certains cas utiliser plus de couleurs que le coloriage optimal.
Ces deux algorithmes prennent en paramètre un ordre sur les sommets du graphe, que l'on appelera **ordre de numérotation** ; pour un graphe `gphe` à n sommets, on implémente un ordre de numérotation de ses sommets par un tableau `num` de n valeurs entières, tel que `num.(k) = j` si et seulement si le sommet `j` apparait en $(k+1)$-ème position dans l'ordre.
Nous commençons par l'**algorithme glouton** de coloriage. Cet algorithme construit un coloriage L d'un graphe G en utilisant au plus $d(G)+1$ couleurs. Son principe est le suivant :
On parcourt la liste des sommets du graphe, dans l'ordre de numérotation des sommets donné. Pour chaque sommet s parcouru :

1. On calcule l'ensemble $C(s) = \{L(t) | t \in V(s)\}$ des couleurs déjà données aux voisins de s.
2. On cherche le plus petit entier naturel c qui n'appartient pas à $C(s)$.
3. On pose $L(s) = c$.

Exercice 9
Considérons le graphe de Petersenn et les deux ordres de numérotation :
`num1 = [|1;3;4;0;2;6;5;9;8;7|]` *et* `num2 = [|0;7;2;5;4;6;8;1;3;9|]`.
Donner les coloriages obtenus par l'algorithme glouton décrit ci-dessus pour le graphe de Petersen et chacun de ces deux ordres de numérotation, ainsi que les nombres de couleurs correspondants.

Exercice 10
Écrire une fonction `min_couleur : graphe -> etiquetage -> int -> int` *telle que pour un graphe* `gphe` *à* n *sommets, un étiquetage* `etiq` *à valeurs dans* $\{-1, \dots, n-1\}$ *et un sommet* `s` *de* `gphe`*, l'appel de* `min_couleur gphe etiq s` *renvoie le plus petit entier naturel n'appartenant pas à l'ensemble des étiquettes des voisins de* s*. On demande une complexité en* $O(n)$*.*

Exercice 11
Écrire une fonction `glouton : graphe -> int array -> etiquetage`*, telle que, pour un graphe* `gphe` *et un ordre de numérotation* `num` *de ses sommets,* `glouton gphe num` *renvoie le coloriage glouton de* `gphe`*, avec au plus* $d+1$ *couleurs, où* d *est le degré de* `gphe`*. On demande une complexité en* $O(n^2)$*, où* n *est le nombre de sommets de* `gphe`*.*
Dans le cas où le tableau `num` *contient autre chose qu'un ordre de numérotation des sommets de* `gphe`*, le résultat de la fonction est laissé au choix du rédacteur.*

Exercice 12
Montrer que l'algorithme de coloriage glouton construit toujours un coloriage, et que ce coloriage utilise au plus $d+1$ *couleurs, où* d *est le degré du graphe en entrée.*

Exercice 13
Soit G *un graphe. Montrer que pour tout coloriage* L *de* G*, il existe un ordre de numérotation des sommets tel que le coloriage glouton* L' *associé vérifie* $L'(s) \leq L(s)$ *pour tout sommet* s *de* G*. En déduire qu'il existe une numérotation des sommets telle que l'algorithme glouton renvoie un coloriage optimal.*

L'efficacité de l'algorithme glouton est en grande partie dépendante de l'ordre dans lequel on choisit de parcourir les sommets du graphe. Pour essayer de l'améliorer, une alternative est donnée par l'optimisation de Welsh-Powell.
L'idée est de parcourir l'ensemble des sommets du graphe par ordre de degré décroissant.
Le tri des sommets par degré décroissant ne prend pas plus de temps que le parcours glouton, mais permet d'obtenir un algorithme raisonnablement efficace en pratique.

Exercice 14
Écrire une fonction `tri_degre : graphe -> int array`*, qui calcule le tableau des sommets d'un graphe trié par ordre décroissant de leurs degrés.*
En déduire une fonction `welsh_powell : graphe -> etiquetage` *qui implémente l'optimisation de Welsh-Powell, et justifier le choix de votre algorithme de tri pour la fonction* `tri_degre`*.*

4 Algorithme de Wigderson

L'algorithme de Wigderson permet, pour un graphe G supposé 3-coloriable, de trouver en temps polynomial en n un coloriage de G en $O(\sqrt{n})$ couleurs (au sens où il existe $C > 0$ tel que pour tout n suffisamment grand, ce coloriage ait au plus $C\sqrt{n}$ couleurs).
Voici son principe de l'algorihtme de Wigderson (G a n sommets et est 3-coloriable).

1. On se donne comme couleur initiale $c = 0$.
2. Pour chaque sommet s de G pas encore colorié avec au moins $\sqrt{n}$ voisins pas encore coloriés :
   (a) On 2-colorie, avec les couleurs c et $c+1$, le sous-graphe induit par l'ensemble des voisins de s pas encore coloriés.
   (b) On incrémente c du nombre de couleurs utilisées en (a).
3. Enfin, on utilise l'algorithme glouton (avec un ordre de numérotation quelconque) pour colorier, avec des couleurs supérieures ou égales à c, le sous-graphe induit par l'ensemble des sommets par encore coloriés.

Exercice 15
Montrer que l'algorithme de Wigderson appliqué à un graphe 3-coloriable construit toujours un coloriage, et que ce coloriage utilise un nombre de couleur en $O(\sqrt{n})$, où n est le nombre de sommets du graphe.

Exercice 16
Écrire une fonction `voisins_non_colories : graphe -> etiquetage -> int -> int list` *telle que* `voisins_non_colories gphe etiq s` *renvoie la liste des voisins* `t` *de* `s` *tels que* `etiq.(t) = -1`
En déduire une fonction `degre_non_colories : graphe -> etiquetage -> int -> int` *telle que* `degre_non_colories gphe etiq s` *renvoie le nombre de voisins* `t` *de* `s` *tels que* `etiq.(t) = -1`*.*

Exercice 17
Écrire une fonction `non_colories : graphe -> etiquetage -> int list` *telle que* `non_colories gphe etiq` *renvoie la liste des sommets* `s` *de* `gphe` *tels que* `etiq.(s) = -1`*.*

Exercice 18
Écrire une fonction `wigderson : graphe -> etiquetage` *telle que si* `gphe` *est 3-coloriable, alors* `wigderson gphe` *renvoie un coloriage de gphe obtenu par l'algorithme de Wigderson décrit plus haut. On demande une complexité polynomaile en le nombre de sommets de* `gphe`*. De plus, les propriétés sur le coloriage établies à l'exercise 15 devront être respectées et justifiées.*
Le comportement de la fonction est laissé au choix du rédacteur lorsque `gphe` *n'est pas 3-coloriable.*

Exercice 19 — Gag !
Comment pourrait-on étendre l'algorithme de Wigderson à des graphes de nombre chromatique connu et strictement supérieur à 3 ?

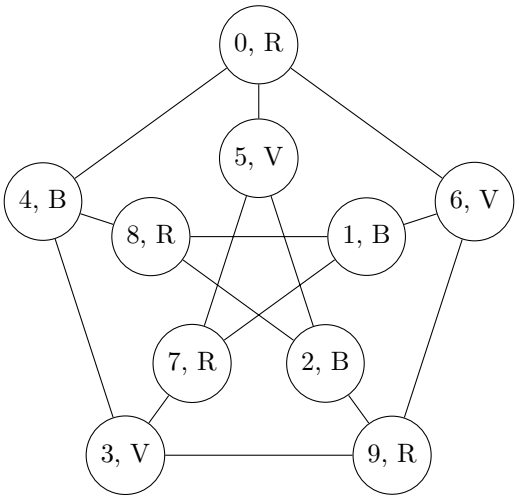
5 Solutions

Solution de l'exercice 1 -

```
let sous_graphe gphe sg =
    let p = Array.length sg in
    let ss_gphe = Array.make_matrix p p false in
    for i = 0 to (p-1) do
        let si = sg.(i) in
        for j = 0 to (p-1) do
            let sj = sg.(j) in
            ss_gphe.(i).(j) <- gphe.(si).(sj) done done;
    ss_gphe;;
```

Solution de l'exercice 2 - Le premier graphe admet deux sommets adjacent qui ont la même couleur ; l'étiquetage proposé n'est pas un coloriage. Cependant il admet un 3-coloriage avec l'étiquetage du second graphe donc il est colorié au sens du sujet.
Le second est colorié car l'étiquetage proposé est un 3-coloriage.

Solution de l'exercice 3 - Le graphe contient des cycles de longueur impaire, par exemple (0, 4, 3, 9, 6) : il n'est pas 2-coloriable. En effet il faudrait que les couleurs soient alternées dans le cycle et on arriverait à deux couleurs égales pour les sommets 0 et 6 qui sont reliés.
Par contre il est 3-coloriable :



Solution de l'exercice 4 -

```
let est_col gphe etiq =
    let n = Array.length gphe in
    let p = Array.length etiq in
    let coloriable = ref true in
    if n <> p
    then coloriable := false
    else for i = 0 to (n-2) do
           for j = (i+1) to (n-1) do
               if gphe.(i).(j) && etiq.(i) = etiq.(j)
               then coloriable := false done done;
    !coloriable;;
```

Solution de l'exercice 5 - Un graphe de n sommet possède un n-coloriage, il suffit de donner une couleur distincte à chaque sommet.
Pour tester son nombre chromatique il suffit de tester, pour k variant de 2 à $n-1$ s'il admet un coloriage à k couleurs.
Pour cela on peut tester tous les coloriages : il y en a k^n.
La complexité est alors majorée par

$$An^2\left(\sum_{k=2}^{n-1} k^n\right) \leqslant An^2.n.n^n = A2^{(n+3)\log_2(n)} \leqslant B2^{n^2}$$

la complexité est exponentielle.

Solution de l'exercice 6 - Supposons que le graphe G est $(k+1)$-coloriable, et soit s un de ses sommets. On considère une $(k+1)$-coloration de G. Alors aucun sommet de $V(s)$ n'est de la même couleur que s (ce serait contradictoire). Sur le sous-graphe induit par $V(s)$, cette coloration reste valide, et n'utilise donc que k couleurs au maximum (celle de s n'est pas employée).
Notons au passage qu'on a forcément $s \notin V(S)$, sans quoi aucune coloration ne serait possible.

Solution de l'exercice 7 - Si G est biparti, alors ses sommets peuvent se diviser en deux sous-ensembles T et U. On attribue alors la couleur 1 aux sommets de T, et 2 aux sommets de U. La propriété de coloration est alors instantanément vérifiée.
Inversement, si G possède une 2-coloration, alors on peut appeler T les sommets recevant la couleur 1 et U les sommets recevant la couleur 2. Aucune arête ne peut relier deux sommets de couleur 1 (ou 2), et les arêtes vont donc d'un sommet de T vers un sommet de U.

Solution de l'exercice 8 - Le sous-programme `explo i k` réalise le parcours en profondeur à partir du sommet `i`, en fixant sa couleur à `k`.

```
let deux_col gphe =
    let n = Array.length gphe in
    let etiq = Array.make n (-1) in
    let rec explo i k =
        etiq.(i) <- k;
        for j = 0 to n-1 do
            if gphe.(i).(j) && etiq.(j) = -1 then
                explo j (1-k) done
   in
    for i = 0 to n-1 do
        if etiq.(i) = -1 then
            explo i 0 ;
    done;
    etiq;;
```

Ici, si le graphe n'est pas 2-coloriable, alors le programme renvoie une coloration fausse (on ne détecte pas les erreurs si le sommet j est déjà colorié).
Le programme `explo` ne peut être lancé qu'une seule fois par sommet au maximum, et sa complexité est en $O(n)$. On arrive donc à une complexité en $O(n^2)$ dans le pire des cas.

Solution de l'exercice 9 - Avec le premier ordre, on trouve comme colorations pour les sommets : $(0;0;0;0;1;1;1;2;2;2)$, donc trois couleurs.
Avec le second, on trouve comme colorations : $(0;3;0;2;1;1;1;0;2;3)$, donc quatre couleurs.

Solution de l'exercice 10 -

```
let min_couleur_possible gphe etiq s =
    let n = Array.length gphe in
    let coul = Array.make n false in
    for i = 0 to (n-1) do
        if gphe.(s).(i) && etiq.(i) <> -1
        then coul.(etiq.(i)) <- true done;
    let c = ref 0 in
    while coul.(!c) do c := !c + 1 done;
    !c;;
```

Solution de l'exercice 11 -

```
let glouton gphe num =
    let n = Array.length gphe in
    let coul = Array.make n (-1) in
    for i = 0 to (n-1) do
        let k = num.(i) in
        coul.(k) <- min_couleur_possible gphe coul k done;
    coul;;
```

Solution de l'exercice 12 - La fonction `min_couleur_possible` renvoie une couleur qui n'a pas été affectée aux voisins du sommet passé en paramètre. Lorsqu'une couleur est affectée à un sommet elle ne pourra plus être affectée aux voisins qui suivent dans l'ordre de numération. De plus chaque sommet reçoit une couleur lorsque `num` est un ordre de numération.
On a bien construit un coloriage du graphe.

Dans l'algorithme glouton chaque sommet est colorié par une couleur non employée par ses voisins déjà coloriés, comme il admet au plus $d(G)$ voisins, la couleur choisie est la plus petite parmi une ensemble d'au plus $d(G)$ entiers positif : elle est donc majorée par $d(G)$.
Ainsi la coloration construite admet au plus $d(G) + 1$ couleurs.

Solution de l'exercice 13 - Soit L un coloriage de G. On considère une numération des sommets qui les classe par numéro de couleur croissante.
Comme deux sommets de même couleur ne sont pas adjacents, lors de l'appel de `min_couleur_possible` les seuls sommets voisins de s qui seront considérés auront une couleur strictement inférieure à celle de s, $L(s)$. La couleur choisie, $L'(s)$, ne pourra donc pas être strictement supérieure à $L(s)$: on a $L'(s) \leqslant L(s)$ pour tout s.
Si on part d'un coloriage optimal pour construire la numération des sommets on aboutit donc à un coloriage dont le maximum est majoré par celui du coloriage optimal : ce sera donc aussi un coloriage optimal.

Solution de l'exercice 14 -
Comme l'algorithme `glouton` est de complexité quadratique on peut choisir un algorithme lui-même quadratique pour construire une numération des sommets sans augmenter la complexité. On commence par construire un tableau avec les degrés.

```
let degres gphe =
    let n = Array.length gphe in
    let deg = Array.make n 0 in
    for i = 0 to (n-1) do
        for j = 0 to (n-1) do
            if gphe.(i).(j)
            then deg.(i) <- deg.(i) + 1 done done;
    deg;;
```

Les outils utilisés ensuite (dont `range`)

```
let echanger tab i j =
    let temp = tab.(i) in
    tab.(i) <- tab.(j);
    tab.(j) <- temp;;

let range n =
    Array.init n (fun x -> x);;
```

La création de la numération suit la méthode du tri par sélection

```
let tri_degre gphe =
    let n = Array.length gphe in
    let deg = degres gphe in
    let num = range n in
    for i = 0 to (n-2) do
        let max = ref i in
        let deg_max = ref deg.(num.(i)) in
        for j = (i+1) to (n-1) do
        let d =  deg.(num.(j)) in
            if d > !deg_max
            then (max := j; deg_max := d) done;
        echanger num i !max done;
    num;;
```

Il suffit alors de tout rassembler.

```
let welsh_powell gphe =
    glouton gphe (tri_degre gphe);;
```

Solution de l'exercice 15 - Fixons les notations.
On pose $G_0 = G$ de taille n.
L'étape (a) se décompose en étapes.

1. Si G_i admet un sommet de degré supérieur à $\sqrt{n}$,
2. on choisit un tel sommet, s_i, par exemple celui de degré maximum,
3. on colorie avec les couleurs $2i$ et $2i+1$ $V(s_i)$, ce qui est possible d'après la question précédente. En effet G_i est induit d'un graphe 3-coloriable donc est 3-coloriable d'où $V(s_i)$ est 2-coloriable.
4. On définit G_{i+1} comme le graphe induit de G_i en enlevant les sommets appartenant à $V(s_i)$.

Après au plus $\sqrt{n}$ itérations G_p n'admet plus de sommet de degré supérieur à $\sqrt{n}$ et on le colorie par l'algorithme glouton.
On a ainsi partitionné l'ensemble des sommets : $S = S_0 \cup S_1 \cup \cdots S_p$,
où $S_i = V(s_i)$ pour $i < p$ et S_p est l'ensemble des sommets de G_p.
Chaque S_i admet un coloriage :

1. pour $i < p$, $V(s_i)$ est colorié par $2i$ et $2i + 1$,
2. G_p est colorié par q couleurs qui sont supérieures ou égales à $2p$. De plus, d'après la question **10**, on a $q \leqslant \sqrt{n} + 1$.

Comme chaque partie est coloriée par des couleurs distinctes on obtient un coloriage de G avec $2p + q$ couleurs qui vérifient $2p + q \leqslant 2\sqrt{n} + \sqrt{n} + 1 = \mathcal{O}(\sqrt{n})$.

Solution de l'exercice 16 -

```
let voisins_non_colories gphe etiq s =
  let n = Array.length gphe in
  let vnc = ref [] in
  for j = 0 to (n-1) do
      if gphe.(s).(j) && etiq.(j) = -1
      then vnc := j :: !vnc done;
  !vnc;;

let degre_non_colories gphe etiq s =
    List.length (voisins_non_colories gphe etiq s);;
```

Solution de l'exercice 17 -

```
let non_colories gphe etiq =
  let n = Array.length gphe in
  let nc = ref [] in
  for i = 0 to (n-1) do
      if etiq.(i) = -1
      then nc := i :: !nc done;
  !nc;;
```

Solution de l'exercice 18 - Pour ne pas écrire un programme démesurément long, on sort quelques fonctions. La première détermine le degré en sommets non coloriés maximal. Le test de majoration de $\sqrt{n}$ est renvoyé par le type optionnel : la fonction renvoie `None` s'il n'existe pas de sommets avec suffisamment de voisins non coloriés et sinon elle renvoie un sommet k vérifiant cette proprité sous la forme `Some k`.

```
let sous_degre_maximum gphe etiq =
    let n = Array.length gphe in
    let r = int_of_float (sqrt (float_of_int n)) in
    let deg_max = ref (degre_non_colories gphe etiq 0) in
    let ind_max = ref 0 in
    for i = 1 to (n-1) do
        let d = degre_non_colories gphe etiq i in
        if  d > !deg_max
            then (deg_max := d; ind_max := i) done;
if !deg_max > r then Some !ind_max else None;;
```

La seconde modifie le tableau des couleurs (`etiq`) à partir des couleurs calculées (`ss_etiq`) pour un sous-graphe (`sg`)

```
let ajouter_couleurs etiq sg ss_etiq =
    let p = Array.length sg in
    for i = 0 to (p-1) do
        etiq.(sg.(i)) <- ss_etiq.(i) done;;
```

Le programme consiste alors à répéter la recherche de sommets avec suffisamment de voisins non coloriés tant qu'il en existe.

```
let wigderson gphe =
    let n = Array.length gphe in
    let etiq = Array.make n (-1) in
    let rec aux () =
        match sous_degre_maximum gphe etiq with
        |None -> let sg = Array.of_list (non_colories gphe
           etiq) in
                 let ss_gphe = sous_graphe gphe sg in
                 let ss_etiq = glouton ss_gphe (range (Array.
                    length ss_gphe)) in
                 ajouter_couleurs etiq sg ss_etiq
        |Some i -> let sg = Array.of_list (
           voisins_non_colories gphe etiq i) in
                   let ss_gphe = sous_graphe gphe sg in
                   let ss_etiq = deux_col ss_gphe in
                   ajouter_couleurs etiq sg ss_etiq;
                   aux ()
    in aux (); etiq;;
```

Toutes les fonctions auxiliaires sont de complexité polynomiale en la taille du graphe et elles sont appelées au plus $\sqrt{n}$ fois : la complexité est polynomiale.

Solution de l'exercice 19 - Comme on sait définir un coloriage dans le cas d'un graphe 3-coloriable on peut poursuivre le raisonnement de manière semblable.
Si un graphe est 4-coloriable, les voisins d'un sommets sont 3-coloriables ; on va donc pouvoir définir un coloriage pour les "gros" voisinages puis conclure par un algorithme glouton.
On considère la propriété $\mathcal{P}(k)$: on peut définir un coloriage de $A_k.n^{a_k}$ couleurs d'un graphe k-coloriable avec $a_k < 1$.
$\mathcal{P}(2)$ est vraie avec $a_2 = 1$ et $A_2 = 2$.
$\mathcal{P}(3)$ est vraie avec $a_2 = \frac{1}{2}$ et $A_3 = 3$.
On suppose $\mathcal{P}(k)$ vraie. G est un graphe $k+1$-coloriable.
Pour chaque sommet s de G ayant au moins n^r voisins pas encore coloriés on colorie ceux-ci avec au plus $A_k.n^{a_k}$ couleurs non encore utilisées. On colorie le reste avec au plus $n^r + 1$ couleurs non utilisées.
Le nombre de sommets avec plus de n^r voisins est au plus $\frac{n}{n^r}$ donc on a utilisé au plus $n^{1-r}.A_k.n^{a_k} + n^r$ couleurs.
On peut choisir r tel que l'expression ci-dessus devienne homogène : $1 - r + a_k = r$ donc $r = \frac{1+a_k}{2}$.
On obtient ainsi un coloriage avec au plus $(A_k + 1).n^r$ couleurs.
La récurrence $a_{k+1} = \frac{1+a_k}{2}$ avec $a_2 = 0$ donne $a_k = 1 - 2^{2-k}$. donc
la propriété $\mathcal{P}(k)$ est donc vraie avec $A_k = k$ et $a_k = 1 - 2^{2-k}$.

Dans le calcul ci-dessus on n'a pas tenu compte du fait qu'on appliquait $\mathcal{P}(k)$ à des graphes de tailles inférieures à n.
Si les tailles auxquelles on applique $\mathcal{P}(k)$ sont m_1, m_2, ..., m_p, le nombre de couleurs employées est en fait majoré par $\displaystyle\sum_{i=1}^{p} A_k.m_i^{a_k} + n^r$

La fonction $x \mapsto n^{a_k}$ est concave donc $\dfrac{\sum_{i=1}^{p} m_i^{a_k}}{p} \leqslant \left(\displaystyle\sum_{i=1}^{p} \frac{m_i}{p}\right)^{a_k}$.

Ainsi $\displaystyle\sum_{i=1}^{p} A_k.m_i^{a_k} + n^r \leqslant p^{1-a_k} A_k \left(\sum_{i=1}^{p} m_i\right)^{a_k} + n^r$.

Comme on a $\displaystyle\sum_{i=1}^{p} m_i \leqslant n$ et $p \leqslant n^{1-r}$, le nombre de couleurs utilisées est majoré par $n^{(1-r)(1-a_k)} A_k n^{a_k} + n^r$.
On choisit $a_{k+1} = r$ tel que $(1-r)(1-a_k) + a_k = r$ d'où $r = \frac{1}{2-a_k}$.
$a_2 = 0$ donne bien $a_3 = \frac{1}{2}$.
On calcule ensuite $a_4 = \frac{2}{3}$, $a_5 = \frac{3}{4}$ et, par récurrence, $a_k = \frac{k-2}{k-1} = 1 - \frac{1}{k-1}$ qui donne moins de couleurs que la valeurs $1 - 2^{2-k}$.

TP VI

Sudoku

Le jeu de Sudoku se joue sur une grille 9x9.
On remplit les cases par des chiffres entre 1 et 9 de telle sorte que chaque ligne, chaque colonne et chacun des 9 carrés 3x3 contienne les 9 chiffres.

1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	3	4	5	6	7	8	9	1
5	6	7	8	9	1	2	3	4
8	9	1	2	3	4	5	6	7
3	4	5	6	7	8	9	1	2
6	7	8	9	1	2	3	4	5
9	1	2	3	4	5	6	7	8

1 Définition

Nous allons utiliser un constructeur somme pour les cases :

```
type contenu = Vide | Pleine of int ;;
```

On pourra représenter les grilles par des tableaux de tableaux, on nommera `sudoku` ce type.

Exercice 1
Écrire une fonction qui crée une grille vide. Sa signature sera `init : unit -> sudoku`

Exercice 2
Écrire une fonction qui copie une grille dans une grille indépendante.

Exercice 3
Créer la grille présentée au début.

Exercice 4
Écrire une fonction `imprime : sudoku -> unit` qui imprime une grille.

```
 --- --- ---
|5xx|xxx|xxx|
|xxx|x7x|xxx|
|xx8|xxx|xxx|
 --- --- ---
|xxx|xxx|xxx|
|xxx|xxx|xxx|
|xxx|xxx|xxx|
 --- --- ---
|xxx|xxx|xxx|
|xxx|xxx|xxx|
|xxx|xxx|xxx|
 --- --- ---
```

2 Jouer

Exercice 5
Écrire une fonction `possible grille i j n` qui renvoie un booléen indiquant s'il est licite de placer la valeur n à la ligne i et la colonne j de la grille. On devra vérifier les 3 conditions.
À présent nous allons résoudre des grilles avec contraintes, c'est-à-dire partiellement remplies. Pour cela on utilisera une méthode assez naïve qui va essayer toutes les solutions possible jusqu'à en trouver une bonne. On appelle cette méthode "backtracking" car on revient en arrière quand on tombe sur un cul-de-sac.

- On tente d'insérer un 1 dans la première case vide.
- Si c'est possible alors on passe à la case suivante
- Quand on aboutit à une impossibilité on passe à 2 puis 3 ...
- Si on arrive à 9 sans trouver de solution, on retourne à la case précédente pour y essayer le nombre suivant

Exercice 6
Écrire la résolution d'une grille.
On pourra utiliser une exception pour arrêter dès qu'on a trouvé une solution.

```
exception Solution;;

let resoudre grille =
  ...
  let aux ...
    ...
    if ...
    then raise Solution
    ...
  try aux ...
      ...
  with Solution -> imprime g;;
```

3 Aller plus loin

On peut compter le nombre de solutions
On peut alors construire des grilles qui n'ont qu'une solution.
`Random.int (n+1)` renvoie un nombre au hasard entre 0 et n.

4 Solutions

Solution de l'exercice 1 -

```
let init () =
  Array.make_matrix 9 9 Vide;;
```

Solution de l'exercice 2 -

```
let copie grille =
  let g = init() in
  for i = 0 to 8 do
    for j = 0 to 8 do
      g.(i).(j) <- grille.(i).(j) done; done;
  g;;
```

Solution de l'exercice 3 -

```
let exemple =
  let g = init() in
  let un = [|1; 4; 7; 2; 5; 8; 3; 6; 9|] in
  for i = 0 to 8 do
    for j = 0 to 8 do
      g.(i).(j) <- Val ((un.(i)+j-1) mod 9 + 1) done; done;
  g;;
```

Solution de l'exercice 4 -

```
let imprime grille =
  let h = " --- --- ---\n" in
  print_newline ();
  print_string h;
  for i = 0 to 8 do
    print_string "|";
    for j = 0 to 8 do
      (match grille.(i).(j) with
       |Vide -> print_string "x"
       |Val(n) -> print_int n);
      if (j+1) mod 3 = 0
      then print_string "|" done;
    print_newline ();
    if (i+1) mod 3 = 0
    then print_string h done;;
```

Solution de l'exercice 5 -

```
let verif grille lg col n =
  let rep = ref true in
  for i = 0 to 8 do
    if i <> lg && (case grille i col) = Val(n)
    then rep := false;
    if i <> col && (case grille lg i) = Val(n)
    then rep := false done;
  let a = lg/3 and b = col/3 in
  for i = 0 to 2 do
```

```
    let u = 3*a + i in
    for j = 0 to 2 do
      let v  = 3*b +j in
        if (u <> lg || v <> col) && (case grille u v) = Val(n)
        then rep := false done; done;
  !rep;;
```

Solution de l'exercice 6 -

```
exception Solution;;

let suivant (i,j) =
  if j = 8
  then (i + 1,0)
  else (i,j + 1);;

let resoudre grille =
  let g = copie grille in
  let rec aux g (lg,col) =
    if lg = 9
    then raise Solution
    else if case g lg col <> Vide
         then aux g (suivant (lg,col))
         else for n = 1 to 9 do
                if possible g lg col n
                then (g.(lg).(col) <- Val n;
                      aux g (suivant (lg,col));
                      g.(lg).(col) <- Vide) done in
  try aux g (0,0);
      failwith "pas de solution"
  with Solution -> imprime g;;
```

TP VII

Algorithme de Tarjan

On cherche à déterminer les composantes fortement connexes d'un graphe **non orientés** *: ce sont les classes d'équivalence pour la relation définie par $s \sim t$ si et seulement si il existe un chemin de s vers t et un chemin de t vers s.*
On pourra illustrer les fonctions écrites pour les deux graphes suivants : Ils correspondent au même

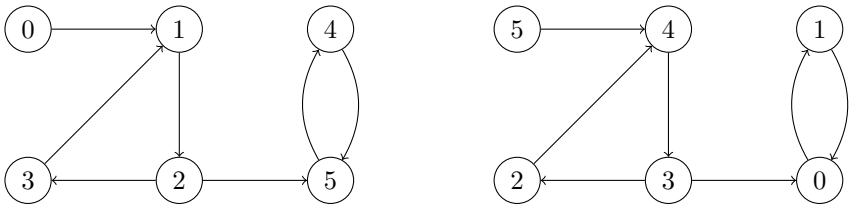


Figure VII.1 – a) Graphe G_1, b) Graphe G_2

graphe avec des numérotations des sommets différentes.

La représentation des graphes devra donner les fonctions

```
taille : graphe -> int
voisins : graphe -> int -> int list
```

Dans les exemples on supposera que les voisins sont donnés par ordre croissant.

Pour les calculs de complexité, on supposera que les graphes sont définis par des tableaux d'adjacence, le type `graphe` *sera alors* `int list array`*. Le graphe G_1 sera alors représenté par*

```
let g1 = [|[1]; [2]; [3; 5]; [1]; [5]; [4]|];;
```

1 Algorithme de Tarjan

Nous allons écrire l'algorithme de Tarjan par étapes.

1.1 Composantes connexes d'un graphe non orienté

On part du parcours en profondeur classique (sans utiliser `List.iter`) pour déterminer les composantes connexes d'un graphe non orienté..

```
let composantes g =
  let n = taille g in
  let cc = Array.make n (-1) in
  let num = ref 0 in
  let rec voir s =
    let rec aux liste =
      match liste with
      |[] -> ()
      |t::q -> if cc.(t) = -1 then voir t;
               aux q in
    cc.(s) <- !num;
    aux (voisins g s) in
  for i = 0 to (n-1) do
    if pref.(i) = -1
    then begin voir i;
               num := !num + 1 end done;
  cc;;
```

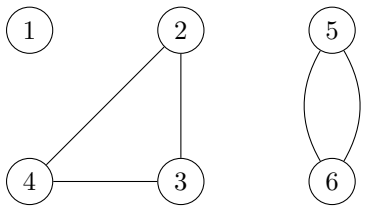


Figure VII.2 – Graphe non orienté G_0

Le tableau des numéros des composantes sert de contrôle des éléments vus.
Pour le graphe G_0, la fonction renvoie `[|0; 1; 1; 1; 2; 2|]`

1.2 Ordre préfixe

On commence part modifier le parcours en profondeur pour numéroter les sommets dans l'ordre où on commence à les explorer, c'est l'ordre infixe.

Exercice 1
Écrire une fonction `prefixe : graphe -> int array` *qui calcule cette numérotation.*

```
# prefixe g1
- : int array = [|0; 1; 2; 3; 5; 4|]
# prefixe g2
- : int array = [|0; 1; 2; 4; 3; 5|]
```

1.3 Minimum de retour

On peut remarquer que, parmi les sommets d'une même composante fortement connexe (CFC), il en existe un particulier, c'est celui qui est visité en premier (son numéro est minimum), c'est la racine de la composante. Bien entendu la racine d'une composante dépend de l'ordre de parcours des sommets.
Lors de l'exploration depuis la racine, tous les sommet de la CFC vont être visités et le parcours va, de plus, explorer des sommets de la CFC déjà vus car il existe un chemin vers la racine.
On va changer le numéro en numéro minimum, associé à chaque sommet :

- quand un sommet est visité, on attribue initialement la valeur d'ordre de visite au minimum

- quand on explore ses voisins, les voisins non encore vus sont visités ce qui calcule une valeur pour le minimum, les voisins déjà vus ont déjà un minimum
- à chaque voisin de s on met à jour le minimum de s en le remplaçant par celui du voisin s'il est inférieur.

Par exemple l'exploration de G_1 donne

	mini
0	[| 0; -1; -1; -1; -1; -1|]
0 → 1	[| 0; 1; -1; -1; -1; -1|]
0 → 1 → 2	[| 0; 1; 2; -1; -1; -1|]
0 → 1 → 2 → 3	[| 0; 1; 2; 3; -1; -1|]
0 → 1 → 2 → 3 → 1	[| 0; 1; 2; 3; -1; -1|]
0 → 1 → 2 → 3	[| 0; 1; 2; 1; -1; -1|]
0 → 1 → 2	[| 0; 1; 1; 1; -1; -1|]
0 → 1 → 2 → 5	[| 0; 1; 1; 1; 4; -1|]
0 → 1 → 2 → 5 → 4	[| 0; 1; 1; 1; 4; 5|]
0 → 1 → 2 → 5 → 4 → 5	[| 0; 1; 1; 1; 4; 5|]
0 → 1 → 2 → 5 → 4	[| 0; 1; 1; 1; 4; 4|]
0 → 1 → 2 → 5	[| 0; 1; 1; 1; 4; 4|]
0 → 1 → 2	[| 0; 1; 1; 1; 4; 4|]
0 → 1	[| 0; 1; 1; 1; 4; 4|]
0	[| 0; 1; 1; 1; 4; 4|]

Exercice 2
Écrire une fonction `minimum : graphe -> int array` *qui calcule cette numérotation.*

```
# minimum g1
- : int array = [|0; 1; 1; 1; 4; 4|]
```

1.4 Restriction des cas

Malheureusement la fonction ci-dessus ne donne pas les composantes fortement connexes dans tous les cas.

```
# minimum g2
- : int array = [|0; 0; 0; 0; 0; 0|]
```

Voici le détail du parcours

	mini
0	[| 0; -1; -1; -1; -1; -1|]
0 → 1	[| 0; 1; -1; -1; -1; -1|]
0 → 1 → 0	[| 0; 1; -1; -1; -1; -1|]
0 → 1	[| 0; 0; -1; -1; -1; -11|]
0	[| 0; 0; -1; -1; -1; -1|]
2	[| 0; 0; 2; -1; -1; -1|]
2 → 4	[| 0; 0; 2; -1; 3; -1|]
2 → 4 → 3	[| 0; 0; 2; 4; 3; -1|]
2 → 4 → 3 → 0	[| 0; 0; 2; 4; 3; -1|]
2 → 4 → 3	[| 0; 0; 2; 0; 3; -1|]
2 → 4 → 3 → 2	[| 0; 0; 2; 0; 3; -1|]
2 → 4 → 3	[| 0; 0; 2; 0; 3; -1|]
2 → 4	[| 0; 0; 2; 0; 0; -1|]
2	[| 0; 0; 0; 0; 0; -1|]
5	[| 0; 0; 0; 0; 0; 5|]
5 → 4	[| 0; 0; 0; 0; 0; 5|]
5	[| 0; 0; 0; 0; 0; 0|]

Le problème ici est que, lors de l'exploration de 3, le sommet à un voisin 0 qui a déjà été complètement exploré et qui ne peut donc pas accéder à 3 ; de même pour 5 qui admet 4 comme voisin.
Pour éviter le problème on peut ajouter un tableau de booléen qui permet de marquer comme finis les sommets dont l'exploration est achevée ; on ne mettra à jour le minimum avec un voisin que lorsque le voisin n'est pas fini.
Cette dernière règle est trop rigide, lorsque l'on parcourt un sommet t depuis s avec `mini.(t) = -1`, le sommet t aura été complètement visité au moment de la mise à jour de `mini.(s)`, alors que, dans ce cas, il est nécessaire de mettre à jour.

Exercice 3
Écrire une fonction `minimum1 : graphe -> int array` *qui calcule la numérotation avec le test d'achèvement du parcours en modifiant la règle rigide.*

```
# minimum1 g1
- : int array = [|0; 1; 1; 1; 4; 4|]
# minimum1 g2
- : int array = [|0; 0; 2; 2; 2; 5|]
```

1.5 Une pile

Encore une fois, les efforts faits ne suffisent pas. On le constate sur le graphe G_3 : il y a plusieurs indices mais une seule CFC.

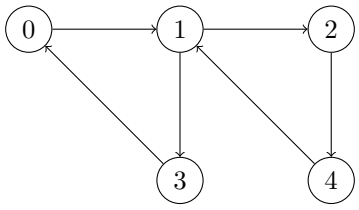


Figure VII.3 – Graphe G_0

Le problème ici est que, lors de l'exploration de 1, le parcours visite 2 qui revient à 1 alors que 1 n'a pas encore vu son minimum diminuer : il le sera en visitant 3 puis 0.
L'idée suivante est de calculer directement les CFC en utiliser les valeurs du minimum pour reconnaître les racines. En effet celles-ci ne peuvent pas voir leur valeur du minimum diminuer, elles sont caractérisées par l'égalité entre le minimum et la valeur de l'ordre préfixe (il faudra donc la conserver). Les sommets de la CFC associée sont alors qui succèdent à la racine **mais qui ne sont pas encore dans une autre CFC**.
Pour obtenir ces points on va empiler les sommets visités et, lorsque l'on a un sommet `s` pour lequel `pref(s) = mini.(s)` on dépile les sommets de la pile jusqu'à `s` : cela forme la CFC.

Exercice 4
Écrire une fonction `tarjan : graphe -> int list list` *qui calcule la liste des composantes fortement connexes (codées sous forme de listes).*

- La preuve de l'algorithme est subtile (lire : je ne sais pas la faire).
- Par contre la complexité est assez simple : on fait un parcours en profondeur et on empile et dépile (une seule fois) les sommet on obtient une complexité en $\mathcal{O}\big(|S| + |A|\big)$, linéaire.

2 Quelques graphes

2.1 Les catégories de Roget

Peter Mark Roget a publié, en 1851, un livre, le *Thesaurus of English Words ans Phrases*, qui renversait le principe d'un dictionnaire. Au lieu d'associer des catégories aux mots, le thesaurus associe des mots aux catégories. Le livre contient ainsi 1022 catégories (dans la seconde édition) qui contiennent des listes de mots les concernant.
Ce qui nous intéresse c'est que les catégories peuvent renvoyer à d'autres, sans que cela soit forcement réversible : on obtient ainsi un graphe orienté.
Ces données ont été traduites dans deux fichiers :

- le fichiers `roget_gr.ml` définit le graphe associé, `g`, sous la forme d'un tableau d'adjacence
- le fichiers `roget_noms.ml` définit le tableau des noms des concepts, `noms`, `noms.(k)` contient le nom du concept correspondant au sommet k du graphe.

On intègre les définitions par les instructions

```
#use "chemin/vers/les/fichiers/roget_gr.ml";;
#use "chemin/vers/les/fichiers/roget_noms.ml";;
```

On pourra donc calculer les composantes fortement connexes

```
let cfc_roget = tarjan g;;
```

Exercice 5
Déterminer la taille de la plus grande CFC.

Exercice 6
Déterminer les catégories des composantes à trois éléments.

2.2 Divisibilité et permutations

On considère l'ensemble S des entiers de 0 à 999.
Les arêtes du graphes sont les couples (p, q) tels que

- p et q sont des anagrammes $173 \to 317$, $7 \to 700$
- ou p divise q, $153 \to 612$.

Exercice 7
Déterminer le graphe.

Exercice 8
Déterminer la CFC de taille maximale.

3 Solutions

Solution de l'exercice 1 - Le principal changement avec le calcul des composantes connexes est le moment où on incrémente le numéro.

```
let prefixe g =
  let n = taille g in
  let pref = Array.make n (-1) in
  let num = ref 0 in
  let rec voir s =
    let rec aux liste =
      match liste with
      |[] -> ()
      |t::q -> if pref.(t) = -1 then voir t;
               aux q in
    pref.(s) <- !num;
    num := !num +1;
    aux (voisins g s) in
  for i = 0 to (n-1)
    do if pref.(i) = -1 then voir i done;
  pref;;
```

Solution de l'exercice 2 - Une ligne à ajouter

```
let minimum g =
  let n = taille g in
  let mini = Array.make n (-1) in
  let num = ref 0 in
  let rec voir s =
    let rec aux liste =
      match liste with
      |[] -> ()
      |t::q -> if mini.(t) = -1 then voir t;
               if mini.(t) < mini.(s) then mini.(s) <- mini.(t
                  );
               aux q in
    mini.(s) <- !num;
    num := !num +1;
    aux (voisins g s) in
  for i = 0 to (n-1)
    do if mini.(i) = -1 then voir i done;
  mini;;
```

Solution de l'exercice 3 -
Une nouvelle variable, un test supplémentaire et une nouvelle affectation.
On ne fait le test de `fini.(t)` que pour les sommets dont l'exploration à déjà commencé.

```
let minimum1 g =
  let n = taille g in
  let mini = Array.make n (-1) in
  let fini = Array.make n false in
  let num = ref 0 in
  let rec voir s =
    let rec aux liste =
      match liste with
      |[] -> ()
      |t::q -> if mini.(t) = -1
               then begin voir t;
                    if mini.(t) < mini.(s)
                    then mini.(s) <- mini.(t) end
                else if not fini.(t) && mini.(t) < mini.(s)
                     then mini.(s) <- mini.(t);
               aux q in
    mini.(s) <- !num;
    num := !num +1;
    aux (voisins g s);
    fini.(s) <- true in
  for i = 0 to (n-1)
    do if mini.(i) = -1 then voir i done;
  mini;;
```

Solution de l'exercice 4 - Beaucoup de variables ...
On utilise une liste référencée comme pile.
On commence par une fonction qui découpe une pile selon un sommet lui appartenant.

```
let jusqua x pile en_cours =
  let rec aux fait reste =
    match reste with
    |[] -> [], fait
    |t::q when t = x -> en_cours.(t) <- false; t::fait,q
    |t::q -> en_cours.(t) <- false; aux (t::fait) q
  in  aux [] pile;;
```

On peut alors écrire la fonction

```
let tarjan g =
  let n = taille g in
  let pref = Array.make n (-1) in
  let mini = Array.make n (-1) in
  let en_cours = Array.make n false in
  let num = ref 0 in
  let cfc = ref [] in
  let pile = ref [] in
  let rec voir s =
    let rec aux liste =
      match liste with
      |[] -> ()
      |t::q -> if pref.(t) = -1 then voir t;
               if en_cours.(t) && mini.(t) < mini.(s)
               then mini.(s) <- mini.(t);
               aux q
    in pref.(s) <- !num;
    mini.(s) <- !num;
    num := !num +1;
    pile := s :: !pile;
    en_cours.(s) <- true;
    aux (voisins g s);
    if pref.(s) = mini.(s)
    then begin let debut, fin = jusqua s !pile en_cours in
         cfc := debut :: !cfc;
         pile := fin end
  in for i = 0 to (n-1)
    do if pref.(i) = -1 then voir i done;
  !cfc;;
```

Solution de l'exercice 5 -

```
let rec max_lg liste k =
match liste with
|[] -> k
|t::q -> max_lg q (max k (List.length t));;

max_lg cfc_roget 0;;

904
```

Solution de l'exercice 6 -

```
let imprimer3 liste =
if List.length liste = 3
then begin List.iter (fun k -> print_string (noms.(k)^" "))
   liste;
print_newline() end;;


List.iter imprimer3 cfc_roget;;
```

```
adolescence man woman
plurality fraction zero
idolatry pseudo-revelation revelation
interpreter oracle sorcerer
consanguinity paternity posterity
```

Solution de l'exercice 7 -

Solution de l'exercice 8 -

TP VIII

Langages dérivés

Dans ce sujet on considère un alphabet fini $\mathcal{A}$.

Définition :
: langages dérivés
Si L est un langage sur $\mathcal{A}$ et si u est un mot ($u \in \mathcal{A}^*$) le langage dérivé (à gauche) de L par u est $u^{-1}.L = \{v \in \mathcal{A}^* \mid u.v \in L\}$. On dit aussi que $u^{-1}.L$ est un **résiduel** *de L.*

1 Propriétés

Exercice 1 — Première propriétés
Prouver les propriétés
1. ε appartient à $u^{-1}.L$ si et seulement si $u \in L$.
2. $\varepsilon^{-1}.L = L$.
3. $u^{-1}.\mathcal{A}^* = A^*$.
4. $v^{-1}.(u^{-1}.L) = (u.v)^{-1}.L$.

Exercice 2 — Un exemple
On note L_1 le langage sur $\mathcal{A} = \{a, b\}$ des mots ayant un nombre pair de b et L_2 le langage des mots ayant un nombre impair de b.
Calculer les dérivés $u^{-1}.L_1$ et $u^{-1}.L_2$ pour un mot de $\mathcal{A}^*$.

Exercice 3 — Un autre exemple
On pose $L = \{a^n b^n \ ; \ n \in \mathbb{N}\}$ sur $\mathcal{A} = \{a, b\}$.
Prouver que $(a^p)^{-1}.L = \{a^n b^{n+p} \ ; \ n \in \mathbb{N}\}$ pour $p \geqslant 1$.
Prouver que $(a^p.b^q)^{-1}.L = \{b^{p-q}\}$ pour $p \geqslant q \geqslant 1$.

Exercice 4 — Opérations rationnelles
L_1 et L_2 sont des langages sur $\mathcal{A}$ et a est une lettre de $\mathcal{A}$.
1. Prouver que $a^{-1}(L_1 \cup L_2) = a^{-1}L_1 \cup a^{-1}L_2$.
2. Prouver que si $\varepsilon \notin L_1$ alors $a^{-1}.(L_1.L_2) = (a^{-1}.L_1).L_2$.
3. Prouver que si $\varepsilon \in L_1$ alors $a^{-1}.(L_1.L_2) = a^{-1}.L_2 \cup (a^{-1}.L_1).L_2$.
4. Prouver que $a^{-1}.(L_1^*) = (a^{-1}.L_1).L_1^*$.

Exercice 5 — Rationalité
Prouver que $u^{-1}.L$ est rationnel si L est rationnel.

2 Automates

L est un langage reconnaissable.
$Q = (\mathcal{A}, S, \delta, s_0, T)$ est un automate déterministe complet reconnaissant L.

Exercice 6 — Reconnaissabilité des dérivés
Prouver que $u^{-1}.L$ est le langage reconnu par $Q_u = (\mathcal{A}, S, \delta, s_0.u, T)$.

Exercice 7 — Critère de reconnaissabilité
Prouver qu'un langage reconnaissable a un nombre fini de résiduels.

Exercice 8 — Un langage non reconnaissable
Prouver que $L = \{a^n b^n \ ; \ n \in \mathbb{N}\}$ n'est pas reconnaissable.

Si un langage L sur l'alphabet $\mathcal{A}$ admet un nombre fini de langages dérivés on définit un automate, l'**automate dérivé**, $Q_L = (\mathcal{A}, \Lambda, \delta_L, L, \Lambda_T)$ avec

- Λ est l'ensemble des résiduels de L, il contient $L = \varepsilon^{-1}.L$,
- Λ_T est l'ensemble des résiduels contenant ε,
- $\delta_L(\lambda, x) = x^{-1}.\lambda$ pour $x \in \mathcal{A}$ et pour tout résiduel λ.

Exercice 9 — Langage reconnu
Prouver que Q_L reconnaît L.

On a donc la démonstration de la réciproque du

Théorème :
Un langage est reconnaissable si et seulement si il admet un nombre fini de résiduels.

Le sens direct a été prouvé dans l'exercice 7.

Exercice 10 — Minimalité
Prouver que l'automate dérivé est de cardinal minimal parmi les automates déterministes complets reconnaissant L.

Définition :
: automates équivalents
Deux automates sur un même langage $\mathcal{A}$, $Q = (\mathcal{A}, S, \delta, s_0, T)$ et $Q' = (\mathcal{A}, S', \delta', s'_0, T')$, sont équivalents
s'il existe une bijection p de S vers S' telle que

1. $p(s_0) = s'_0$,
2. $p(T) = T'$ *et*
3. $\delta'\big(p(s), x\big) = p\big(\delta(s, x)\big)$ *pour tout $s \in S$ et pour tout $x \in \mathcal{A}$.*

p est un **isomorphisme** *de Q vers Q'.*

Exercice 11 — Unicité
Prouver que si un automate $Q = (\mathcal{A}, S, \delta, s_0, T)$ reconnaît L et vérifie $|S| = |\Lambda|$
alors Q est équivalent à l'automate dérivé.

3 Solutions

Solution de l'exercice 1 -

1. v appartient à $u^{-1}.L$ si et seulement si $u.v$ appartient à L donc $\varepsilon \in u^{-1}.L$ si et seulement si $u = u.\varepsilon \in L$.
2. $u \in \varepsilon^{-1}.L$ si et seulement si $u = \varepsilon.u \in L$: $\varepsilon^{-1}.L = L$.
3. $u^{-1}.\mathcal{A}^* \subset A^*$; si $v \in \mathcal{A}^*$ alors $u.v \in \mathcal{A}^*$ donc $v \in u^{-1}.\mathcal{A}^*$ d'où $\mathcal{A}^* \subset u^{-1}.\mathcal{A}^*$ puis $\mathcal{A}^* = u^{-1}.\mathcal{A}^*$.
4. $w \in v^{-1}.(u^{-1}.L) \iff v.w \in u^{-1}.L \iff u.(v.w) \in L \iff (u.v).w \in L \iff w \in (u.v)^{-1}.L$.

Solution de l'exercice 2 - En comptant le nombre de b on montre que $a^{-1}.L_1 = L_1$, $b^{-1}.L_1 = L_2$, $a^{-1}.L_2 = L_2$ et $b^{-1}.L_2 = L_1$. En appliquant le résultat ci-dessus on en déduit que $u^{-1}.L_1 = L_1$ et $u^{-1}.L_2 = L_2$ si $u \in L_1$; $u^{-1}.L_1 = L_2$ et $u^{-1}.L_2 = L_1$ si $u \in L_2$.

Solution de l'exercice 3 -
$u \in (a^p)^{-1}.L \iff a^p.u \in L \iff \exists n \in N,\ a^p.u = a^n.b^n \iff u = a^{n-p}.b^n$
$(a^p.b^q)^{-1}.L = (b^q)^{-1}.\left((a^p)^{-1}.L\right)$ or le seul mot de $(a^p)^{-1}.L$ qui commence par un b est b^p. L'ensemble est non vide si et seulement si $p \geqslant q$ et alors son seul élément est b^{p-q}.

Solution de l'exercice 4 -

1. Si u appartient à $a^{-1}(L_1 \cup L_2)$ alors $a.u$ appartient à $L_1 \cup L_2$ donc
   soit $a.u \in L_1$ d'où $u \in a^{-1}L_1 \subset a^{-1}L_1 \cup a^{-1}L_2$,
   soit $a.u \in L_2$ d'où $u \in a^{-1}L_2 \subset a^{-1}L_1 \cup a^{-1}L_2$.
   Dans les deux cas on a $u \in a^{-1}L_1 \cup a^{-1}L_2$ d'où $a^{-1}(L_1 \cup L_2) \subset a^{-1}L_1 \cup a^{-1}L_2$.

   Inversement on a $L_1 \subset L_1 \cup L_2$ donc $a^{-1}L_1 \subset a^{-1}(L_1 \cup L_2)$.
   De même $a^{-1}L_2 \subset a^{-1}(L_1 \cup L_2)$ d'où $a^{-1}L_1 \cup a^{-1}L_2 \subset a^{-1}(L_1 \cup L_2)$.
   On en déduit l'égalité demandée.
2. Soit u appartenant à $(a^{-1}.L_1).L_2$.
   $u = u_1.u_2$ avec $u_2 \in L_2$ et $u_1 \in a^{-1}.L_1$ donc $a.u_1 \in L_1$.
   On a alors $a.u = (a.u_1).u_2 \in L_1.L_2$ donc $u \in a^{-1}.(L_1.L_2)$.
   On a prouvé $(a^{-1}.L_1).L_2 \subset a^{-1}.(L_1.L_2)$ sans condition sur L_1.

   Inversement si u appartient à $a^{-1}.(L_1.L_2)$ alors $a.u \in L_1.L_2$
   donc on peut écrire $a.u = u_1.u_2$ avec $u_1 \in L_1$ et $u_2 \in L_2$.
   Comme u_1 ne peut être le mot vide, il doit commencer par a donc on peut écrire $u_1 = a.u_1'$.
   On a alors $u_1' \in a^{-1}.L_1$ puis $u = u_1'.u_2 \in (a^{-1}.L_1).L_2$ ce qui prouve la seconde inclusion puis l'égalité.
3. Prouver que si $\varepsilon \in L_1$ alors $L_2 \subset L_1.L_2$ donc $a^{-1}L_2 \subset a^{-1}.(L_1.L_2)$.
   L'inclusion $(a^{-1}.L_1).L_2 \subset a^{-1}.(L_1.L_2)$ prouvée ci-dessus donne alors
   $a^{-1}.L_2 \cup (a^{-1}.L_1).L_2 \subset a^{-1}.(L_1.L_2)$

   Comme ci-dessus on écrit, pour $u \in a^{-1}.(L_1.L_2)$ $a.u = u_1.u_2$.
   Si $u_1 \neq \varepsilon$ on a encore $u \in (a^{-1}.L_1).L_2$.
   Si $u_1 = \varepsilon$ on a $a.u = u_2 \in L_2$ donc $u \in a^{-1}L_2$.
   On a donc $u \in a^{-1}.L_2 \cup (a^{-1}.L_1).L_2$ ce qui prouve la seconde inclusion puis l'égalité.
4. Prouver que $a^{-1}.(L_1^*) = (a^{-1}.L_1).L_1^*$.

Solution de l'exercice 5 - On procède par induction structurelle

Solution de l'exercice 6 - $v \in u^{-1}.L \iff uv \in L \iff s_0.(uv) \in T \iff (s_0.u).v \in T$.

Solution de l'exercice 7 - $u^{-1}.L$ est reconnu par $(\mathcal{A}, S, \delta, s_0.u, T)$: $s_0.u$ est un élément de S : il n'y a qu'un nombre fini de tels automates.

Solution de l'exercice 8 - Il admet une infinité de dérivés, par exemple les singletons $\{b^r\}$, donc il ne peut pas être reconnaissable.

Solution de l'exercice 9 - Par récurrence sur $|u|$ on montre que $\lambda.u = u^{-1}.\lambda$ pour tout λ.
En particulier $L.u = u^{-1}.L$ donc les mots reconnus sont les mots tels que $u^{-1}.L \in \Lambda_0$ c'est-à-dire les mots u tels que $\varepsilon \in u^{-1} \cdot L$. On a vu que cela caractérisait les mots de L donc L est le langage reconnu par Q.

Solution de l'exercice 10 - Soit $Q = (\mathcal{A}, S, \delta, s_0, T)$ un automate qui reconnaît L.
L'émondage donne un automate $Q = (\mathcal{A}, S', \delta', s_0, T')$ qui reconnaît L aussi avec $|S'| \leqslant |S|$.
Pour tout $s \in S'$ on note $L(s)$ le langage reconnu par $(\mathcal{A}, S', \delta', s, T')$
Il existe $u \in \mathcal{A}^*$ tel que $s = s_0.u$; on a vu qu'alors le langage reconnu on a $L(s) = u^{-1}.L$.
Ainsi l'application qui à $s \mapsto L(s)$ est à valeur dans Λ.
Cette application est surjective car $u^{-1}.L = L(s_0.u)$.
Ainsi $|\Lambda| \leqslant |S'| \leqslant |S|$.

Solution de l'exercice 11 - On va prouver que $\pi : s \mapsto L(s)$ est un isomorphisme d'automates.

1. On a vu que π est surjective de Λ vers S.
   Comme ces deux ensembles sont finis de même cardinal, π est une bijection.
   Si $\lambda = u^{-1}.L$ on a $\lambda = \pi(s_0.u)$ donc $\pi^{-1}(\lambda) = s_0.u$: le résultat est indépendant de u.
   On a prouvé, dans ce cas, la réciproque de l'exercice précédent
   $u^{-1}.L = v^{-1}.L$ implique $s_0.u = s_0.v$.
2. Si s est terminal alors $s = s_0.u$ avec $u \in L$.
   On a donc $\pi(s) = u^{-1}.L$ qui contient ε : $\pi(s)$ appartient à Λ_T.
   Inversement si $\pi(s)$ est terminal dans Q_L alors il contient ε ; pour $s = s_0.u$ cela signifie qu'on a $\varepsilon \in u^{-1}.L$ donc $u \in L$ et $s \in T$.
   On a bien $\pi(T) = \Lambda_T$.
3. Soit $s = s_0.u \in S$, $\pi(s) = u^{-1}.L$.
   $\delta_L\big(\pi(s), x\big) = x^{-1}.\pi(s) = x^{-1}.\big(u^{-1}.L\big) = (u.x)^{-1}.L$
   $\delta(s, x) = s.x = (s_0.u).x = s_0.(u.x)$ donc $\pi\big(\delta(s, x)\big) = \pi\big(s_0.(u.x)\big) = (u.x)^{-1}.L$.
   On a bien $\delta_L\big(\pi(s), x\big) = \pi\big(\delta(s, x)\big)$.

Les automates sont bien équivalents.

TP IX

Expressions régulières

On veut, dans ce TP, utiliser des expressions régulières.
Une expression régulière sera une chaîne de caractères dont les seuls caractères admis sont `0, 1, (, ), +, ., *` et les lettres de l'alphabet. On convient que `0` représente le vide et `1` le mot vide (ε). Le produit doit être noté avec un point.
On codera en Caml une expression régulière par l'arbre qui lui est associé. Le type est

```
type arbreER =  Vide
              |Eps
              |Feuille of char
              |Etoile of arbreER
              |Plus of arbreER*arbreER
              |Fois of arbreER*arbreER;;
```

Exercice 1
Écrire une fonction `ecrire : arbreER -> string` *qui renvoie l'expression régulière associée à un arbre en utilisant le parcours infixe parenthésé.*
La concaténation de chaînes de caractère s'écrit `^`.
La fonction `Char.escaped` permet de convertir un caractère [1] en une chaîne de longueur 1.

Par exemple `"(1+(a.(b*)))"` est le résultat de `ecrire` appliqué à
`Plus (Eps, Fois (Feuille 'a', Etoile (Feuille 'b'))).`

On peut exécuter l'opération inverse ; dans le cas d'une écriture bien parenthésée, un algorithme "élémentaire" est possible.
On définit une exception qui renvoie l'endroit où se situe un problème.

```
exception ChaineFautive of int;;
```

On définit une fonction auxiliaire récursive qui permet de lire une portion de chaîne qui a un sens, entre deux parenthèses associées, et qui renvoie aussi la position du caractère suivant. Ce caractère devra être un opérateur `'+'`, `'*'` ou `'.'`.

1. Seulement les lettres basiques, pas les lettres accentuées.

```
let lire chaine =
  let rec aux i =
    match chaine.[i] with
    |'0' -> Vide, i+1
    |'1' -> Eps, i+1
    |'(' -> begin let fg, k = aux (i+1) in
                  match chaine.[k] with
                  |')' -> fg, k+1
                  |'*' -> if chaine.[k+1] = ')'
                          then Etoile fg, k+2
                          else raise (ChaineFautive (k+1))
                  |'+' -> let fd, p = aux (k+1) in
                          if chaine.[p] =  ')'
                          then Plus (fg, fd), p+1
                          else raise (ChaineFautive p)
                  |'.' -> let fd, p = aux (k+1) in
                          if chaine.[p] =  ')'
                          then Fois (fg, fd), p+1
                          else raise (ChaineFautive p)
                  |_ -> raise (ChaineFautive k) end
    |c -> Feuille c, i+1 in
  let arbre, n = aux 0 in
    if n = String.length chaine
    then arbre
    else raise (ChaineFautive n);;
```

1 Premières fonctions

Exercice 2
Écrire une fonction `est0 : arbreER -> bool` *qui reçoit une expression régulière (sous forme d'un arbre) et qui renvoie* `true` *ou* `false` *selon que le langage dénoté par l'expression régulière est vide ou non.*

Exercice 3
Écrire une fonction `contient1 : arbreER -> bool` *qui reçoit une expression régulière (sous forme d'un arbre) et qui renvoie* `true` *ou* `false` *selon que le langage dénoté par l'expression régulière contient ou non le mot vide* ε.

Exercice 4
Écrire les fonctions qui permettent de calculer

1. *les premières lettres des mots non vides de* L
2. *les dernières lettres des mots non vides de* L
3. *les facteurs de 2 lettres des mots de longueur au moins 2 de* L

où L *est le langage dénoté par une expression régulière passée en paramètre.*
Les résultats devront être des listes de chaînes de caractères.

2 Rationalité de langages transformés

On reprend un exercice du chapitre.
On suppose que L est un langage rationnel sur $\mathcal{A}$ ($a \in \mathcal{A}$).
On va prouvef que chacun des langages suivants est rationnel.

1. Le langage des mots préfixes des mots de L.
2. L'ensemble des mots de L qui ne contiennent pas a.
3. L'ensemble des mots de L qui contiennent a.
4. L'ensemble des mots obtenus en enlevant le premier a dans les mots de L contenant a
5. L'ensemble des mots obtenus en enlevant un a dans les mots de L contenant a

Exercice 5
Pour chacune de ces questions écrire une fonction qui reçoit un arbre associé à une expression régulière dénotant L et qui renvoie une expression régulière dénotant le langage transformé.

3 Normalisations

On reprend les théorèmes du cours

Exercice 6
Écrire une fonction `elim0 : arbreER -> arbreER` *qui renvoie une expression régulière (sous forme d'un arbre) équivalente à celle passée en paramètre et qui est* `Vide` *ou un arbre ne contenant pas la feuille* `Vide`.

Exercice 7
Écrire une fonction `elimination1 : arbreER -> arbreER` *qui renvoie une expression régulière (sous forme d'un arbre) équivalente à celle passée en paramètre*
et qui est soit `Vide`*, soit* `Eps`*, soit* `Plus (Eps, arbre)` *soit* `arbre`
où arbre ne contient ni la feuille `Vide` *ni la feuille* `Eps`.

4 Solutions

Solution de l'exercice 1 -

```
let rec ecrire arbre =
  match arbre with
  |Vide -> "0"
  |Eps -> "1"
  |Feuille c -> Char.escaped c
  |Etoile f ->  "(" ^ (ecrire f) ^ "*)"
  |Plus (fg, fd) -> "(" ^ (ecrire fg) ^ "+" ^ (ecrire fd) ^ ")
     "
  |Fois (fg, fd) -> "(" ^ (ecrire fg) ^ "." ^ (ecrire fd) ^ ")
     ";;
```

Solution de l'exercice 2 -

```
let rec est0 arbre =
  match arbre with
  |Vide -> true
  |Eps -> false
  |Feuille c -> false
  |Etoile f -> true
  |Plus (fg, fd) -> est0 fg && est0 fd
  |Fois (fg, fd) -> est0 fg || est0 fd;;
```

Solution de l'exercice 3 -

```
let rec contient1 arbre =
  match arbre with
  |Feuille c -> false
  |Vide -> false
  |Eps -> true
  |Etoile f -> true
  |Plus (fg, fd) -> contient1 fg || contient1 fd
  |Fois (fg, fd) -> contient1 fg && contient1 fd;;
```

Solution de l'exercice 4 - On commence par des fonctions d'union de liste.

```
let rec inserer x liste =
  match liste with
  |[] -> [x]
  |t::q when t = x -> liste
  |t::q -> t :: (inserer x q);;
```

```
let rec fusion liste1 liste2 =
  match liste1 with
  |[] -> liste2
  |t::q -> fusion q (inserer t liste2);;
```

```
let rec prefixes arbre =
  match arbre with
  |Vide -> []
  |Eps -> []
  |Feuille c -> [c]
  |Etoile f -> prefixes f
  |Plus (fg, fd) -> fusion (prefixes fg) (prefixes fd)
  |Fois (fg, fd) -> if contient1 fg
                    then fusion (prefixes fg) (prefixes fd)
                    else prefixes fg;;
```

```
let rec suffixes arbre =
  match arbre with
  |Vide -> []
  |Eps -> []
  |Feuille c -> [c]
  |Etoile f -> suffixes f
  |Plus (fg, fd) -> fusion (suffixes fg) (suffixes fd)
  |Fois (fg, fd) -> if contient1 fd
                    then fusion (suffixes fg) (suffixes fd)
                    else suffixes fd;;
```

```
let rec coller x liste =
  match liste with
  |[] -> []
  |t::q -> (x ^ t) :: (coller x q);;
```

```
let rec collage liste1 liste2 =
  match liste1 with
  |[] -> []
  |t::q -> (coller t liste2) @ (collage q liste2);;
```

```
let rec facteurs arbre =
  match arbre with
  |Vide -> []
  |Eps -> []
  |Feuille c -> []
  |Etoile f -> union (facteurs f) (collage (suffixes f) (
     prefixes f))
  |Plus (fg, fd) -> union (facteurs fg) (facteurs fd)
  |Fois (fg, fd) -> union (collage (suffixes fg)
                                   (prefixes fd))
                          (union (facteurs fg)
                                 (facteurs fd));;
```

Solution de l'exercice 5 -

```
let rec pref arbre =
  match arbre with
  |Vide -> Vide
  |Eps -> Eps
  |Feuille c -> Plus (Eps, Feuille c)
  |Etoile f -> Fois (Etoile f, pref f)
  |Plus (fg, fd) -> Plus (pref fg, pref fd)
  |Fois (fg, fd) -> Plus (Fois (fg, pref fd),
                          pref fg);;
```

```
let rec sans arbre x=
  match arbre with
  |Vide -> Vide
  |Eps -> Eps
  |Feuille c -> if c = x then Vide else Feuille x
  |Etoile f -> Etoile (sans f x)
  |Plus (fg, fd) -> Plus (sans fg x, sans fd x)
  |Fois (fg, fd) -> Fois (sans fg x, sans fd x);;
```

```
let rec avec arbre x=
  match arbre with
  |Vide -> Vide
  |Eps -> Vide
  |Feuille c -> if c = x then Feuille x else Vide
  |Etoile f -> Fois (Fois (Etoile f, avec f x),
                     Etoile f)
  |Plus (fg, fd) -> Plus (avec fg x, avec fd x)
  |Fois (fg, fd) -> Plus (Fois (avec fg x, fd),
                          Fois (fg, avec fd x));;
```

```
let rec oter1 arbre x=
  match arbre with
  |Vide -> Vide
  |Eps -> Vide
  |Feuille c -> if c = x then Eps else Vide
  |Etoile f -> Fois (Fois (Etoile f, oter1 f x),
                     Etoile f)
  |Plus (fg, fd) -> Plus (oter1 fg x, oter1 fd x)
  |Fois (fg, fd) -> Plus (Fois (oter1 fg x, fd),
                          Fois (fg, oter1 fd x));;
```

```
let rec oterPrem arbre x=
  match arbre with
  |Vide -> Vide
  |Eps -> Vide
  |Feuille c -> if c = x then Eps else Vide
  |Etoile f -> Fois (Fois (Etoile (sans f x),
                           oterPrem f x),
                     Etoile f)
  |Plus (fg, fd) -> Plus (oterPrem fg x, oterPrem fd x)
  |Fois (fg, fd) -> Plus (Fois (oterPrem fg x, fd),
                          Fois (sans fg x,
                                oterPrem fd x));;
```

Solution de l'exercice 6 -

```
let rec elim0 arbre =
  match arbre with
  |Feuille c -> Feuille c
  |Vide -> Vide
  |Eps -> Eps
  |Etoile f -> if elim0 f = Vide
                 then Eps
                 else Etoile (elim0 f)
  |Plus (fg, fd) -> begin match (elim0 fg), (elim0 fd) with
                          |Vide, Vide -> Vide
                          |Vide, f -> f
                          |f, Vide -> f
                          |fg, fd -> Plus(fg, fd) end
  |Fois (fg, fd) -> begin match (elim0 fg), (elim0 fd) with
                            |Vide, f -> Vide
                            |f, Vide -> Vide
                            |fg, fd -> Fois(fg, fd) end;;
```

Solution de l'exercice 7 -

```
let elimination1 arbre =
  let rec aux arbre =
    match arbre with
    |Vide -> Vide
    |Eps -> Eps
    |Feuille c -> Feuille c
    |Etoile f
      -> begin match aux f with
               |Eps -> Eps
               |Plus(Eps, f) -> Etoile f
               |f -> Etoile f end
    |Plus (fg, fd)
      -> begin match (aux fg), (aux fd) with
               |Eps, Eps -> Eps
               |Eps, Plus(Eps, f) -> Plus(Eps, f)
               |Plus(Eps, f), Eps  -> Plus(Eps, f)
               |Plus(Eps, f), Plus(Eps, g)
                 -> Plus(Eps, Plus (f,g))
               |Plus(Eps, f), g -> Plus(Eps, Plus (f,g))
               |f, Eps -> Plus(Eps, f)
               |f, Plus(Eps, g) -> Plus(Eps, Plus (f,g))
               |f, g -> Plus(f, g) end
    |Fois (fg, fd)
      -> begin match (aux fg), (aux fd) with
               |Eps, f -> f
               |f, Eps -> f
               |Plus(Eps, f), Plus(Eps, g)
                  -> Plus(Eps, Plus (Plus (f,g), Fois(f,g)))
               |Plus(Eps, f), g -> Plus(g, Fois (f,g))
               |f, Plus(Eps, g)-> Plus(f, Fois (f,g))
               |f, g -> Fois(f, g) end
  in aux (elim0 arbre);;
```