Lycée Faidherbe, 2020-2021

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

MP1 option informatique

Version du 20 février 2021

Table des matières

I Tableaux persistants **I-1**
1 Définition . . . I-1
2 Choix de l'implémentation . . . I-2
3 Les fonctions . . . I-3
4 Solutions . . . I-5

II Tracé d'arbres binaires **II-1**
1 Rectangles . . . II-2
2 Squelette . . . II-4
3 Nœuds . . . II-5
4 Valeur des nœuds . . . II-6
5 Solutions . . . II-7

III Arbres rouge-noir **III-1**
1 Présentation du problème . . . III-1
2 Premiers pas . . . III-2
3 Équilibrage . . . III-3
4 Solutions . . . III-6

IV Tas persistants **IV-1**
1 Présentation . . . IV-1
2 Fonctions . . . IV-1
3 Calcul des nombres de Hamming . . . IV-2
4 Nombres taupins . . . IV-3
5 Propriétés des arbres fortement équilibrés . . . IV-3
6 Solutions . . . IV-4

V IF-expressions **V-1**
1 IF-expressions . . . V-1
2 Décision sur les IF-expressions . . . V-3
3 Solutions . . . V-6

VI Bidons **VI-1**
1 Généralisation . . . VI-2
2 Solutions . . . VI-3

VII Mini Langage **VII-1**
1 Opérations simples . . . VII-1

2 Variables VII-2
3 Fonctions VII-3
4 Condionnelles VII-4
5 Solutions VII-5

VIII Algorithme de Tarjan **VIII-1**
1 Algorithme de Tarjan VIII-1
2 Quelques graphes VIII-5
3 Solutions VIII-6

IX Couplages **IX-1**
1 Couplages généraux IX-1
2 Couplages dans les graphes bipartis IX-2
3 Solutions IX-6

TP I

Tableaux persistants

Résumé *La notion de tableaux est centrale dans les langages informatiques. Le type usuel est une structure impérative : les changements ne créent pas un nouveau tableau mais modifient le tableau "en place". Nous allons définir le type de données tableau et en construire une implémentation persistante.*

1 Définition

On peut voir un tableau comme une fonction depuis un ensemble fini $\{0, 1, \ldots, n-1\}$ vers l'ensemble des valeurs du type choisi. Cependant, pour des raisons de simplicité nous allons définir des tableaux de taille n dont les indices décrivent $\{1, 2, \ldots, n\}$.
Voici les principales fonctions définies pour le type `Array` :

1. `Array.make n x` crée un tableau de taille n dont les éléments sont tous x.

```
Array.make : int -> 'a -> 'a array
```

2. `Array.length t` renvoie la longueur du tableau t

```
Array.length : 'a array -> int
```

3. `Array.get k t` renvoie l'élément d'indice k du tableau t, c'est l'équivalent de `t.(k)`

```
Array.get : int -> 'a array -> 'a
```

4. `Array.set k t y` met la valeur y à la position k dans le tableau t, c'est `t.(k) <- x`

```
Array.set : int -> 'a array -> 'a -> unit
```

Il y en a d'autres, moins utilisés.
Nous allons nous intéresser à l'initialisation par une fonction : `Array.init n f` renvoie le tableau de taille n dont les éléments sont tels que la valeur en i soit $f(i)$

```
Array.init : int -> (int -> 'a) -> 'a array
```

et à l'extraction d'une tranche : `Array.sub t i p` renvoie le tableau de taille p dont les éléments sont ceux de t, dans le même ordre, d'indice i, $i+1 \ldots i+p-1$.
C'est l'équivalent de `t[i, i+p]` dans python.

```
Array.sub : 'a array -> int -> int -> 'a array
```

Le caractère impératif du type se voit uniquement dans la la fonction `set` : celle-ci ne renvoie rien [1]

1. En fait, elle renvoie `()`, la valeur du type `unit`.

Nous allons définir un type de données persistant, **tableau**, avec les mêmes fonctions (dont on change le nom) :

1. `creer n x` crée un tableau de taille n dont les éléments sont tous x.

```
creer : int -> 'a -> 'a tableau
```

2. `longueur t` renvoie la longueur du tableau t

```
longueur : 'a tableau -> int
```

3. `valeur k t` renvoie l'élément d'indice k du tableau t

```
valeur : int -> 'a tableau -> 'a
```

4. `changer k t y` met la valeur y à la position k dans le tableau t en renvoyant un nouveau tableau, modifié et laissant inchangé le tableau initial.

```
changer : int -> 'a tableau -> 'a -> 'a tableau
```

5. `construire n f` renvoie le tableau des $f(1)$, $f(2)$, ..., $f(n)$

```
construire : int -> (int -> 'a) -> 'a tableau
```

6. L'extraction est trop difficile à écrire, on se contentera de
`debut t p` qui renvoie le tableau des p élément initiaux de t

```
debut : 'a tableau -> int ->  'a tableau
```

Bien entendu, on souhaite que les opérations se fassent rapidement, un temps constant est généralement difficile à obtenir pour des structures persistantes mais une complexité logarithmique est espérée.

Cela interdit de recopier un tableau classique quand on change un des éléments.

2 Choix de l'implémentation

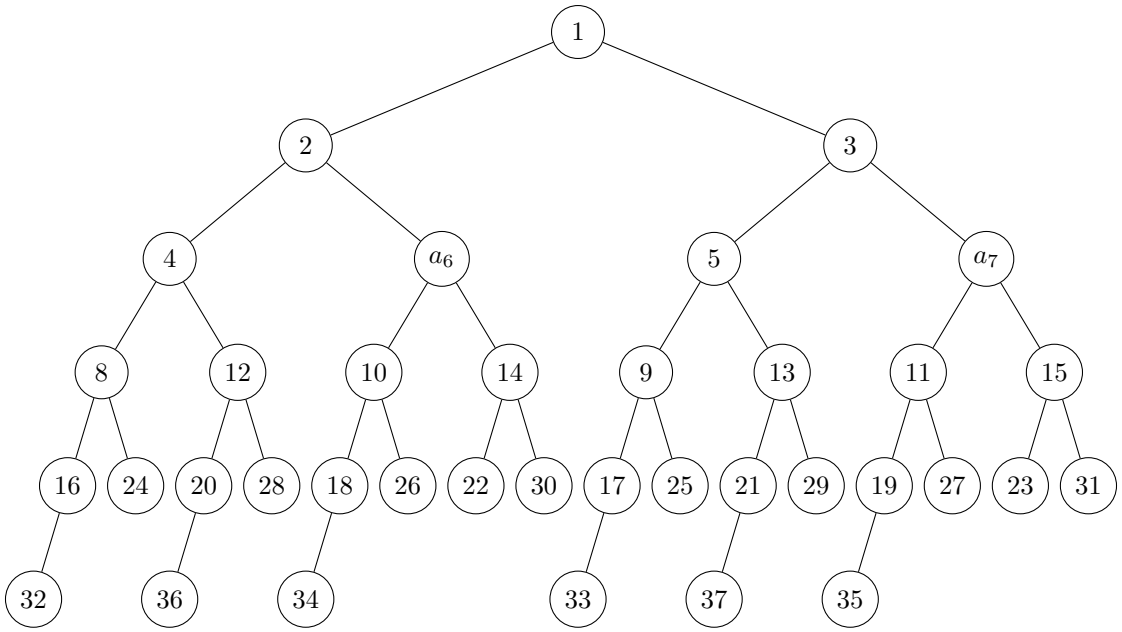
Une complexité logarithmique est possible avec un arbre à condition de savoir comment descendre les branches. On propose la stratégie suivante pour rechercher le nœud d'indice k :

- si $k = 1$ on a trouvé, l'élément est à la racine,
- si $k > 1$ avec k pair, on cherche `k/2` dans le fils gauche,
- si $k > 1$ avec k impair, on cherche `k/2` dans le fils droit.

Par exemple, l'élément d'indice 37 sera trouvé

- en cherchant 18 dans le fils droit
- puis en cherchant 9 dans le petit fils gauche
- puis 4 à droite
- puis 2 à gauche
- la valeur est la racine du fils gauche.

Dans le dessin suivant on a donné les positions d'un tableau à 37 éléments.



On va donc utiliser un arbre binaire qui sera accompagné de la taille pour que celle-ci soit directement calculable (et le contrôle des indices facile aussi).

```
type 'a arbre = Vide | Noeud of 'a arbre * 'a * 'a arbre;;
type 'a tableau = {taille : int;
                   data : 'a arbre};;
```

3 Les fonctions

3.1 Premières fonctions

Exercice 1 — Taille
Écrire la fonction `longueur`.

Exercice 2 — Accès
Écrire la fonction `valeur`.

Exercice 3 — Modification
Écrire la fonction `changer`.

Exercice 4 — Complexité
Quelle est la hauteur de l'arbre associée à un tableau de taille n ?
En déduire la complexité des fonctions ci-dessus.

3.2 Constructions

Exercice 5 — Nombres d'éléments
Prouver que l'arbre associée à un tableau est quasi-complet.
Combien y-a-t-il d'éléments dans les fils droits et gauches de l'arbre associée à un tableau de taille n ?.

Exercice 6 — Création
Écrire la fonction `creer`.

Exercice 7 — Initialisation
Écrire la fonction `construire`.
On essaiera d'écrire une fonction de complexité linéaire et non en $\mathcal{O}\big(n\log(n)\big)$.

Exercice 8 — Extraction
Écrire la fonction `debut`.

3.3 Nouvelles fonctions

Un avantage de cette implémentation est qu'elle permet de travailler dynamiquement : on peut ajouter un élément à la fin du tableau ou enlever le dernier élément.
Dans les questions n est la longueur des tableaux.

Exercice 9 — Ajouter
Écrire une fonction `plus t x` *qui ajoute l'élément* x *dans un nouveau tableau reprenant les éléments de* t.

Exercice 10 — Enlever
Écrire une fonction `moins t` *qui renvoie un couple formé du tableau des* $n-1$ *premiers éléments de* `t` *et du dernier élément.*

Il est souvent utile d'écrire les fonctions de conversion.

Exercice 11 — Tableau vers array
Écrire une fonction `vers_array t` *qui reçoit un tableau et renvoie l'objet de type* `array` *dont les éléments d'indices 0 à* $n-1$ *sont ceux d'indices 1 à* n *du tableau.*

Exercice 12 — Array vers tableau
Écrire une fonction `vers_tableau t` *qui reçoit objet de type* `array` *et renvoie le tableau dont les éléments d'indices 1 à* n *sont ceux d'indices 1 à* n *de* t.

4 Solutions

Solution de l'exercice 1 -

```
let longueur t = t.taille;;
```

Solution de l'exercice 2 -

```
let valeur k t =
  let rec lire p a =
    match a, p with
      |Vide, _ -> failwith "Indice trop grand"
      |_, p when p < 1 -> failwith "Indice trop petit"
      |Noeud(g, r, d), 1 -> r
      |Noeud(g, r, d), p when p mod 2 = 0 -> lire (p/2) g
      |Noeud(g, r, d), p -> lire (p/2) d in
    lire k t.data;;
```

Solution de l'exercice 3 -

```
let changer k t x =
  let rec ecrire p a =
    match a, p with
      |Vide, _ -> failwith "Indice trop grand"
      |_, p when p < 1 -> failwith "Indice trop petit"
      |Noeud(g, r, d), 1 -> Noeud(g, x, d)
      |Noeud(g, r, d), p when p mod 2 = 0
                          -> Noeud(ecrire (p/2) g, r, d)
      |Noeud(g, r, d), p -> Noeud(g, r, ecrire (p/2) d )in
    {taille = t.taille; data = ecrire k t.data};;
```

Solution de l'exercice 4 - La profondeur du nœud associé à 1 est 0. Si on note $p(k)$ la profondeur du nœud associé à k on a $p(k) = 1 + p(k/2)$. On en déduit par récurrence sur m que, pour $2^m \leqslant k < 2^{m+1}$, $p(k) = m$. Ainsi la hauteur est $\lfloor \log_2(n) \rfloor$.
La complexité de `taille` est constante et `valeur` et `changer` sont donc de complexité logarithmique.

Solution de l'exercice 5 - Pour $2^m \leqslant n < 2^{m+1}$, les entiers k tels que $2^h \leqslant k < 2^{h+1}$ remplissent le niveau de profondeur h pour $h < m$.
Pour n impair, $n = 2m+1$, il y a m termes pairs et m impairs entre 2 et n, et pour n pair, $n = 2m$, il y a m termes pairs et $m-1$ impairs entre 2 et n. Il y a donc `m/2` éléments dans le fils gauche et `(m-1) /2` éléments dans le fils droit.

Solution de l'exercice 6 -

```
let creer n x =
  let rec aux k =
    match k with
      |0 -> Vide
      |_ -> Noeud(aux (k/2), x, aux (k-1)/2) in
    if n < 1
    then failwith "La longueur doit être strictement positive"
    else {taille = n; data = aux n};;
```

Solution de l'exercice 7 -

```
let construire n f =
  let rec aux n f =
    match n with
      |0 -> Vide
      |_ -> let g = aux (n/2) (fun k -> f (2*k)) in
          let d = aux ((n-1)/2) (fun k -> f (2*k+1)) in
            Noeud(g, f 1, d) in
    if n < 1
    then failwith "La longueur doit être strictement positive"
    else {taille = n; data = aux n f};;
```

Solution de l'exercice 8 -

```
let debut t  p =
  if p > t.taille
  then failwith "Il n'y a pas assez d'éléments"
  else let rec extr ar k =
         match ar, k with
           |Vide, _ -> Vide
           |_, 0 -> Vide
           |Noeud(g, r, d), k ->
               Noeud(extr g (k/2), r, extr d ((k-1)/2)) in
       {taille = p; data = extr t.data p};;
```

Solution de l'exercice 9 -

```
let plus t x =
  let rec add p a =
    match a, p with
      |_, 1 -> Noeud(Vide, x, Vide)
      |Vide, _ -> failwith "Ceci ne devrait pas arriver"
      |Noeud(g, r, d), p when p mod 2 = 0 -> Noeud(add (p/2) g
         , r, d)
      |Noeud(g, r, d), p -> Noeud(g, r, add (p/2) d )
  and n  = t.taille +1 in
    {taille = n; data = add n t.data};;
```

Solution de l'exercice 10 -

```
let moins t =
  let rec del p a =
    match a, p with
      |Noeud(g, r, d), 1 -> Vide, r
      |Vide, _ -> failwith "Ceci ne devrait pas arriver"
      |Noeud(g, r, d), p when p mod 2 = 0 ->
          let g1, x = del (p/2) g in Noeud(g1, r, d), x
      |Noeud(g, r, d), p ->
          let d1, x = del (p/2) d in Noeud(g, r, d1), x
  and n  = t.taille in
  let a, x = del n t.data in {taille = n-1; data = a}, x;;
```

Solution de l'exercice 11 -

```
let vers_array t =
  let n = t.taille in
  let rec aux ar k =
    match ar with
      |Vide -> [||]
      |Noeud(g, r, d) ->
          let ag = aux g (k/2)
          and ad = aux d ((k-1)/2)
          and a = Array.make n r in
            for i = 0 to (k-2) do
              if i mod 2 = 0
              then a.(i+1) <- ag.(i/2)
              else a.(i+1) <- ad.(i/2) done;
            a
  in aux t.data n;;
```

Solution de l'exercice 12 -

```
let vers_tableau t =
  let f k = t.(k-1)
  and n = Array.length t in
    construire n f;;
```

TP II

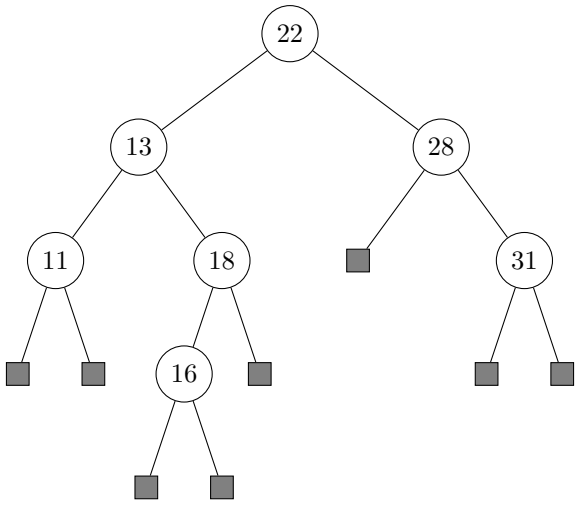
Tracé d'arbres binaires

Résumé

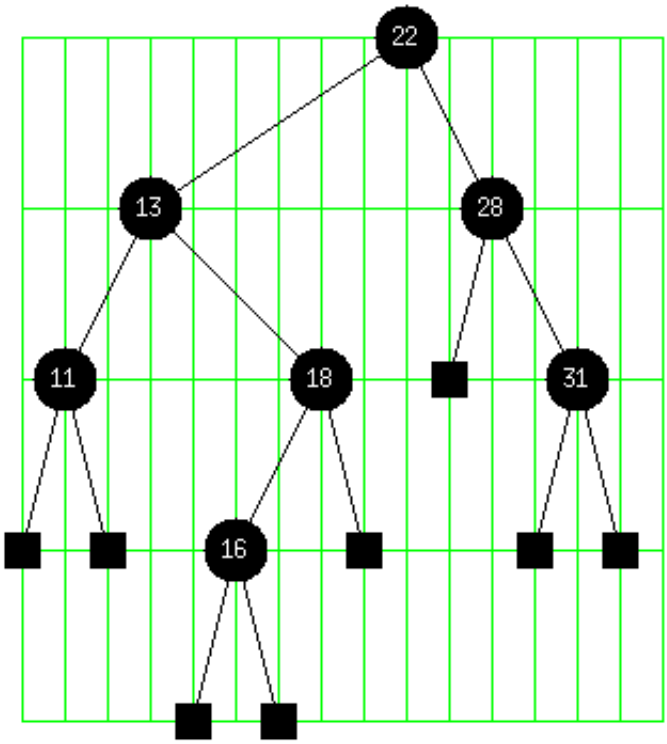
On se propose ici de représenter des arbres binaires afin de pouvoir visualiser les opérations que l'on effectue sur ces arbres. On souhaite dessiner les arbres de telle manière que le parcours infixe consiste à lire les nœuds (et les feuilles) de gauche à droite ; on impose aussi que les nœuds de même profondeur soient alignés horizontalement. On placera donc les nœuds sur une grille.

Dans un premier temps on ne fera porter aux nœuds qu'une valeur entière. Dans le TP suivant on pourra enrichir les valeurs des nœuds, on adaptera les fonctions aux cas particuliers.

L'obectif est de faire tracer à OCaml l'arbre suivant



sous la forme



Nous allons travailler par renforcements successifs d'une fonction : dans chaque partie elle sera améliorée.
Le type de donnée est celle du cours pour lequel les feuilles définies par `Vide`.

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

1 Rectangles

Dans cette partie nous allons encadrer chaque sous-arbre dans un rectangle.
Dans un premier temps on va calculer la taille de la grille qui contient l'arbre. On a besoin

- du nombre de pas verticaux, c'est la profondeur maximale des feuilles donc la hauteur augmentée de 1,
- du nombre de pas horizontaux, comme on dessine un nœud ou une feuille par pas, c'est le nombre total de nœuds/feuilles diminué de 1 donc le double du nombre de nœuds. (Rappel : le nombre de feuilles est le nombre de nœud plus 1.) Plus simplement si les deux fils d'un arbre non vide ont des largeurs p et q l'arbre aura une largeur de $p+q+2$ car il faut placer la racine entre les deux fils.

Exercice 1
Écrire une fonction qui calcul la taille sous la forme d'un couple `(largeur, hauteur)`.
On ne fera qu'une lecture de l'arbre

```
taille : arbre -> int * int
```

On va maintenant dessiner le cadre d'un arbre.
Pour faire des représentations graphique on a besoin de la bibliothèque graphique.

```
#load "graphics.cma";;
open Graphics;;
```

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 `draw` au lieu de `Graphics.draw`

Une documentation complète se trouve à
`https://ocaml.github.io/graphics/graphics/Graphics/index.html`

- Les dessins se font dans une fenêtre que l'on doit ouvrir puis fermer.

```
open_graph " 800x900";;
...
close\_graph ();;
```

  Ici on a ouvert une fenêtre graphique de 800 pixels de large et 900 pixels de haut.
  L'espace blanc avant le premier chiffre est obligatoire.
- Les coordonnées sont calculées en pixels (ce sont des entiers),
  l'origine est en bas à gauche.
- Pour observer le dessin avant de la fermer, on pourra utiliser l'instruction

```
let _ = wait_next_event [Button_down] in close_graph();
```

  Elle met le programme en pause en attendant un clic de souris.
  Effacer la fenêtre ensuite élimine des erreurs, je ne sais pas pourquoi.
- On définit les constantes

```
let pas_h = 20;;
let pas_v = 80;;
let x0 = 20;;
let y0 = 800;;
```

  `pas_h` et `pas_v` sont les tailles, respectivement horizontale et verticale, de la grille.
  x_0 et y_0 sont les coordonnées du point de départ qui correspond au point en haut et à gauche du rectangle, en effet l'arbre sera dessiné depuis sa racine.
- On peut dessiner un rectangle avec la fonction `draw_rect x y larg haut` : x et y sont les coordonnées du point en bas à droite, `larg` est la largeur, `larg` est la hauteur.

Exercice 2
Écrire une fonction `rectg x1 y1 x2 y2` *qui dessine un rectangle à partir des coordonnées des points supérieur gauche et inférieur droit.*

```
rectg : int -> int -> int -> int -> unit
```

Exercice 3
Écrire une fonction `bas_droite a b arbre` *qui renvoie les coordonnées du point en bas à droite du rectangle qui délimite l'arbre et dont le point en haut à gauche est de coordonnées a et b.*

```
bas_droite : int -> int -> arbre -> int * int
```

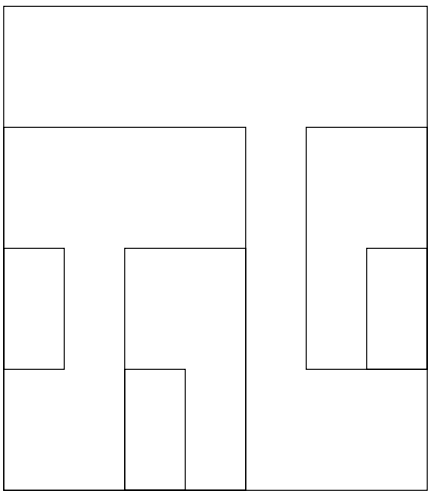
Voici une fonction qui dessine le rectangle délimitant un arbre.

```
let rectangle arbre =
  let x2, y2 = bas_droite x0 y0 arbre in
  open_graph " 800x900";
  rectg x0 y0 x2 y2;
  let _ = wait_next_event [Button_down] in close_graph ();;
```

Exercice 4
Écrire une fonction `rectangles arbre` *qui dessine tous les rectangles délimitant les sous-arbres non réduits à une feuille.*

```
rectangles : arbre -> unit
```

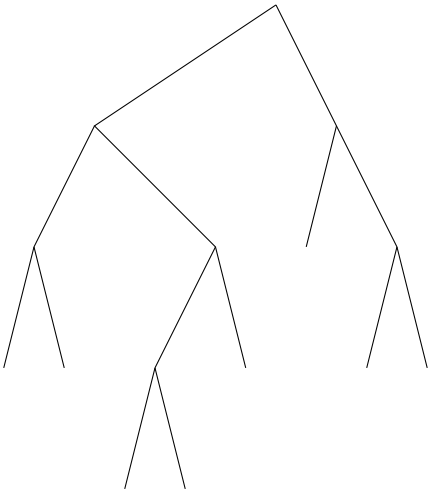
La fonction devra faire appel à une fonction récursive adaptée de `bas_droite`, afin de pouvoir encadrer la fonction récursive par l'ouverture et la fermeture de la fenêtre. Cette fonction récursive aura comme paramètres les coordonnées du coin supérieur gauche du rectangle qui délimite l'arbre, pour l'arbre global ces coordonnées sont x_0 et y_0.



2 Squelette

On souhaite maintenant tracer le squelette de l'arbre, c'est-à-dire l'ensemble des segments entre les nœuds et leurs fils (vides ou non).
Dessiner dessiner des segments on utilise 2 fonctions de déplacement :

- `moveto x y` déplace la position sans rien tracer jusqu'à (x, y)
- `lineto x y` trace un trait depuis la position actuelle jusqu'a (x, y) .

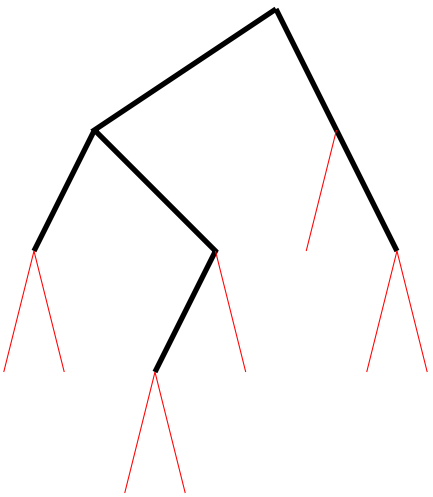


Exercice 5
Écrire une fonction `squelette arbre` *qui dessine tous les segments.*
Une fonction auxiliaire pourra renvoyer aussi la position de la racine.

```
squelette : arbre -> unit
```

On peut décorer les lignes (ces fonctions agissent aussi sur les rectangles) :

- `set_color` sélectionne la couleur, les couleurs de base sont pré-définies :
  `black, while, red, green, blue, yellow, cyan, magenta`.
- `set_line_width` sélectionne la largeur du trait.



Exercice 6
Écrire une fonction `squelette2 arbre` *qui dessine les segments.*
Les segments vers des nœuds non vides seront plus épais, ceux vers une feuille seront rouge.

```
squelette : arbre -> unit
```

3 Nœuds

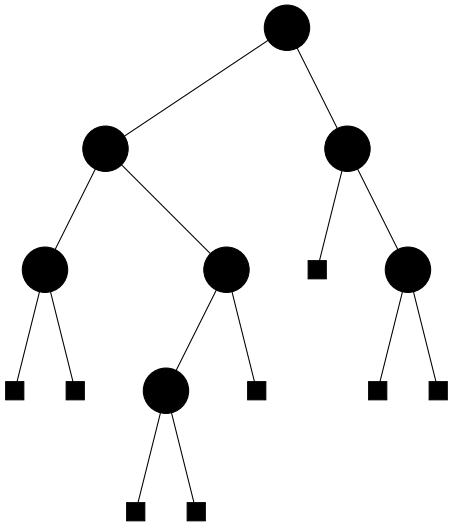
On va maintenant placer les nœuds et les feuilles. Un nœud sera représenté par un disque de rayon 15 (par exemple) et une feuille par un carré de longueur 16, ces formes seront centrées aux extrémités des segments.

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

Voici le début d'une fonction de dessin de la racine d'un arbre, complétez-la.

```
let dessinNoeud arbre x y =
  match arbre with
  |Vide ->  fill_rect (x-cote
     /2) (y-cote/2) cote cote
  |Noeud(g, r, d) ->  ...
```

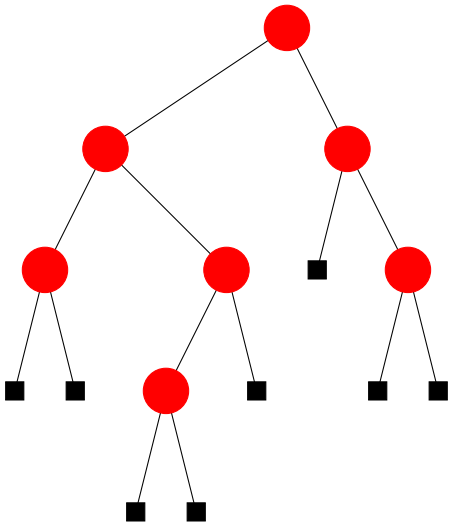
Le tracé est muet car on n'écrit pas la valeur des nœuds.



Exercice 7
Écrire une fonction `muet arbre` *qui dessine les segments et les nœuds.*

```
muet : arbre -> unit
```

On va parfois dessiner les nœuds en couleur.



Exercice 8
Écrire une fonction `muet_r arbre` *qui dessine les segments et les nœuds avec les nœuds en rouge et les feuilles en noir.*

```
muet_r : arbre -> unit
```

Si on n'y prend pas garde les segment sont dessinés au-dessus des nœuds, il faudra adapter le dessin.

4 Valeur des nœuds

Il reste alors à écrire la valeur du nœud dans le cercle.
On utilise les fonctions

- `draw_string texte` qui écrit un texte avec la couleur courante depuis la position actuelle (elle correspond au point en bas à gauche).
- `string_of_int` qui convertit un entier en chaîne de caractères.

Exercice 9
Écrire une fonction `dessin arbre` *qui dessine les segments et les nœuds ainsi que les valeurs, supposées entières, des nœuds.*

```
dessin : arbre -> unit
```

5 Solutions

Solution de l'exercice 1 - Comme la largeur est le nombre de nœuds/feuilles moins un, on doit ajouter 2 lors de la construction.

```
let rec taille arbre =
  match arbre with
  |Vide -> 0, 0
  |Noeud(g, r, d) -> let lg, hg = taille g
                    and ld, hd = taille d in
                    (lg + ld + 2, max hd hg + 1);;
```

Solution de l'exercice 2 -

```
let rectg x1 y1 x2 y2 =
draw_rect x1 y2 (x2 - x1) (y1 - y2);;
```

Solution de l'exercice 3 -

```
let rec bas_droite a b arbre =
  match arbre with
  |Vide -> a, b
  |Noeud(g, r, d)->
            let xg, yg = bas_droite a (b - pas_v) g in
            let xd, yd = bas_droite (xg + 2*pas_h) (b - pas_v)
                d in
  xd, min yg yd;;
```

Solution de l'exercice 4 -

```
let rectangles arbre =
    let rec aux x y a =
        match a with
        |Vide -> x, y
        |Noeud(g, r, d)->
                let x1, y1 = aux x (y - pas_v) g in
                let x2, y2 = aux (x1 + 2*pas_h) (y - pas_v) d
                   in
                let yy = min y1 y2 in
                rectg x y x2 yy;
                x2, yy in
    open_graph " 800x900";
    let _ = aux x0 y0 arbre in
    let _ = wait_next_event [Button_down] in close_graph ();;
```

Solution de l'exercice 5 -

```
let squelette arbre =
    let rec aux x y a =
        match a with
        |Vide -> x, y, x
        |Noeud(g, r, d)->
                let x1, y1, xr1 = aux x (y - pas_v) g in
                let x2, y2, xr2 = aux (x1 + 2*pas_h) (y -
                    pas_v) d in
                moveto xr1 (y -pas_v);
                lineto (x1 + pas_h) y;
                lineto xr2 (y -pas_v);
                x2, (min y1 y2), (x1 + pas_h) in
    open_graph " 800x900";
    let _ = aux x0 y0 arbre in
    let _ = wait_next_event [Button_down] in close_graph ();;
```

Solution de l'exercice 6 - Il est utile d'écrire une fonction séparée pour tracer un segment selon sa destination.

```
let segment x1 y1 x2 y2 a =
    match a with
    |Vide -> moveto x1 y1;
             set_color red;
             set_line_width 1;
             lineto x2 y2
    |_ -> moveto x1 y1;
          set_color black;
          set_line_width 2;
          lineto x2 y2;;
```

```
let squelette2 arbre =
    let rec aux x y a =
        match a with
        |Vide -> x, y, x
        |Noeud(g, r, d)->
                let y0 = y - pas_v in
                let x1, y1, xr1 = aux x y0 g in
                let x2, y2, xr2 = aux (x1 + 2*pas_h) y0 d in
                let xr = x1 + pas_h in
                segment xr y xr1 y0 g;
                segment xr y xr2 y0 d;
                dessinNoeud a xr y;
                x2, (min y1 y2), xr in
    open_graph " 800x900";
    let _ = aux x0 y0 arbre in
    let _ = wait_next_event [Button_down] in  close_graph ();;
```

Solution de l'exercice 7 -

```
let muet arbre =
    let rec aux x y a =
        match a with
        |Vide -> dessinNoeud a x y;
                 x, y, x
        |Noeud(g, r, d)->
                let y0 = y - pas_v in
                let x1, y1, xr1 = aux x 0 g in
                let x2, y2, xr2 = aux (x1 + 2*pas_h) y0 d in
                let xr = x1 + pas_h in
                moveto xr1 y0;
                lineto xr y;
                lineto xr2 y0;
                dessinNoeud a xr y;
                 x2, (min y1 y2), xr in
    open_graph " 800x900";
    let _ = aux x0 y0 arbre in
    let _ = wait_next_event [Button_down] in  close_graph ();;
```

Solution de l'exercice 8 -
On commence par modifier le dessin des nœuds.

```
let dessinNoeud_r arbre x y =
    match arbre with
    |Vide ->  fill_rect (x-cote/2) (y-cote/2) cote cote
    |Noeud(g, r, d) ->  set_color red;
                        fill_circle x y rayon;
                        set_color black;;
```

Ensuite on dessine les nœuds des fils au lieu de celui de la racine dans la fonction, il faut alors dessiner la racine à la fin.

```
let muet_r arbre =
    let rec aux x y a =
        match a with
        |Vide -> x, y, x
        |Noeud(g, r, d)->
                let y0 = y - pas_v in
                let x1, y1, xr1 = aux x 0 g in
                let x2, y2, xr2 = aux (x1 + 2*pas_h) y0 d in
                let xr = x1 + pas_h in
                moveto xr1 y0;
                lineto xr y;
                lineto xr2 y0;
                dessinNoeud_r g xr1 y0;
                dessinNoeud_r d xr2 y0;
                 x2, (min y1 y2), xr in
    open_graph " 800x900";
    let a, b, xr = aux x0 y0 arbre in dessinNoeud_r arbre xr
       y0;
    let _ = wait_next_event [Button_down] in  close_graph ();;
```

Solution de l'exercice 9 -
On commence par modifier le dessin des nœuds pour inclure le texte.

```
let toutNoeud_r arbre x y =
    match arbre with
    |Vide ->  fill_rect (x-cote/2) (y-cote/2) cote cote
    |Noeud(g, r, d) ->  fill_circle x y rayon;
                        set_color white;
                        let ch = string_of_int r in
                        let k = String.length ch in
                        moveto (x - 3*k + 1) (y - 5);
                        draw_string ch;
                        set_color black;;
```

Ensuite on dessine les nœuds des fils au lieu de celui de la racine dans la fonction, il faut alors dessiner la racine à la fin.

```
let dessin arbre =
    let rec aux x y a =
        match a with
        |Vide -> x, y, x
        |Noeud(g, r, d)->
                let y0 = y - pas_v in
                let x1, y1, xr1 = aux x 0 g in
                let x2, y2, xr2 = aux (x1 + 2*pas_h) y0 d in
                let xr = x1 + pas_h in
                moveto xr1 y0;
                lineto xr y;
                lineto xr2 y0;
                toutNoeud_r g xr1 y0;
                toutNoeud_r d xr2 y0;
                 x2, (min y1 y2), xr in
    open_graph " 800x900";
    let a, b, xr = aux x0 y0 arbre in toutNoeud_r arbre xr y0;
    let _ = wait_next_event [Button_down] in  close_graph ();;
```

TP III

Arbres rouge-noir

1 Présentation du problème

Résumé

On se propose ici de donner une structure d'arbre qui aura une hauteur en $\mathcal{O}(\log(n))$ dans le pire des cas (plutôt qu'en moyenne). Plusieurs solutions existent qui ajoutent de la structure aux arbres binaires : les arbres AVL ajoutent la mesure du déséquilibre, les arbres 2-3 permettent des nœuds à clé double qui ont trois fils et les arbres rouges-noir, qui sont notre sujet, ajoutent une couleur aux nœuds.
Nous nous contenterons de valeurs entières pour simplifier l'étude mais la généralisation est immédiate.

1.1 Définition

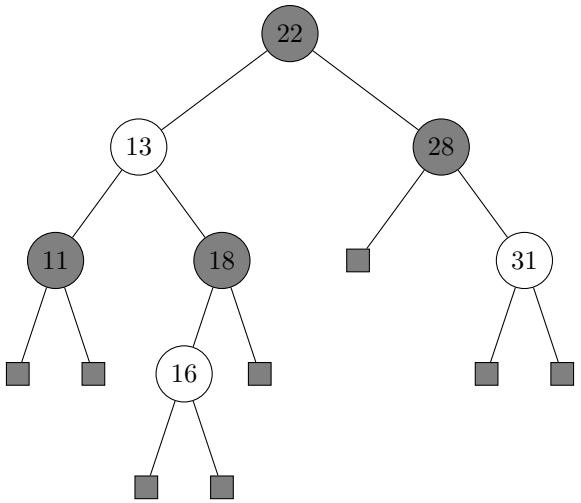
Nos arbres rouge-noir sont définis par les types :

```
type couleur = Rouge|Noir;;
type arbreRN = Vide|Noeud of couleur * arbreRN * int * arbreRN
    ;;
```

Cette structure sera utilisée avec des conditions supplémentaires.

- Toutes les feuilles vides sont considérées comme noires.
- La racine doit être noire.
- Toutes les branches doivent comporter le même nombre de nœuds noirs.
- Un nœud rouge ne peut avoir que des fils noirs (donc son père est noir).

Voici un exemple d'arbre rouge-noir, les nœuds noirs sont grisés, les nœuds rouges sont blancs.



Cet arbre est donné, ainsi qu'une adaptation du tracé des arbres, dans le fichiers `TP03_debut.ml`.

2 Premiers pas

2.1 Fonctions

On commence par s'intéresser aux fonctions usuelles sans se préoccuper des conditions de couleurs : on dit que les arbres sont colorés.

Exercice 1
Écrire les fonctions du cours en les adaptant au nouveau type.
On écrira les fonctions `chercher x arbre`, `ajoutFeuille x a`, `maxArbre arbre`, `suppressionMax arbre`, `suppressionRacine arbre` *et* `suppression x arbre`.
Lors de la création d'un nœud il sera rouge.

2.2 Hauteur noire

On nomme hauteur noire d'un nœud `n` appartenant à un arbre bicolore, notée $h_N(n)$, le nombre de nœuds noirs sur les chemins partant de ce nœud et aboutissant à une feuille (sans prendre en compte `n`). Cette fonction est bien définie en raison de la propriété d'équilibre. On nomme hauteur noire d'un arbre bicolore la hauteur noire de sa racine.

Exercice 2
Montrer qu'un sous-arbre de racine `n` *d'un arbre bicolore contient au moins* $2^{h_N(n)} - 1$ *nœuds internes.*

Exercice 3
Montrer qu'un arbre bicolore comportant l *nœuds internes a une hauteur au plus égale à* $2\log_2(l+1)$.

Cela permet donc d'assurer une complexité logarithmique pour les fonctions sur les arbres si on sait les écrire en conservant les conditions.

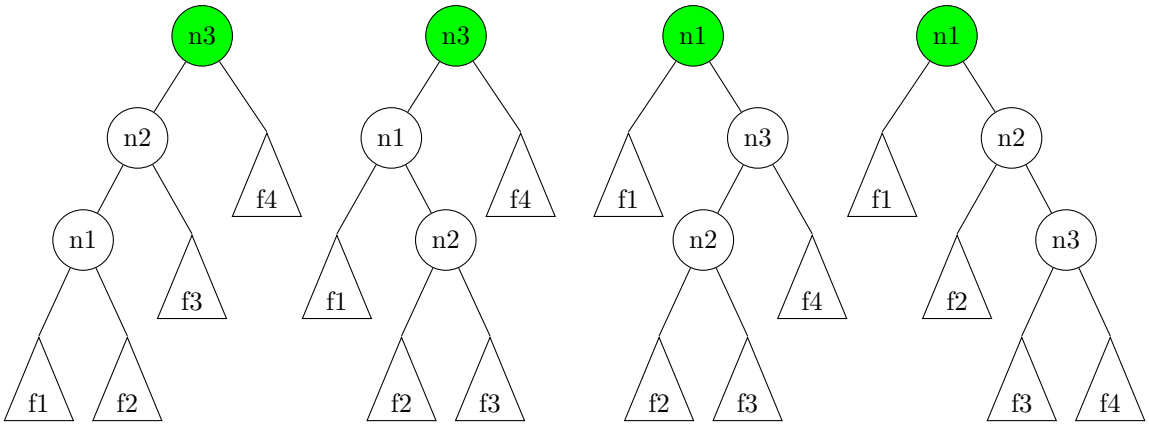
3 Équilibrage

La fonction de recherche ne pose pas de problème.

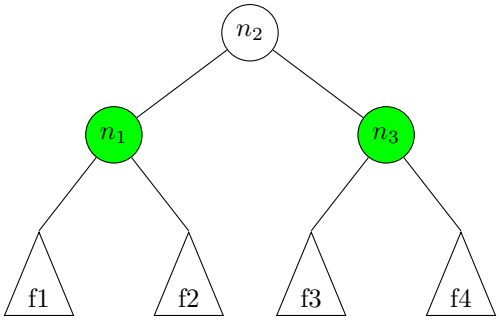
3.1 Insertion aux feuilles

Lors de l'insertion à une feuille on a créé un nœud rouge pour ne pas modifier la hauteur noire.
Cependant on peut créer un conflit si un nœud est rouge ainsi que son père.
C'est le fils rouge qui vient d'être modifié donc le père du père, s'il existe, est noir.
Pour résoudre le conflit on a donc 5 cas.

- Si le père rouge est la racine, on le change en noir.
- Il reste 4 cas selon que le père rouge est un fils droit ou gauche et le fils rouge est un fils droit ou gauche.



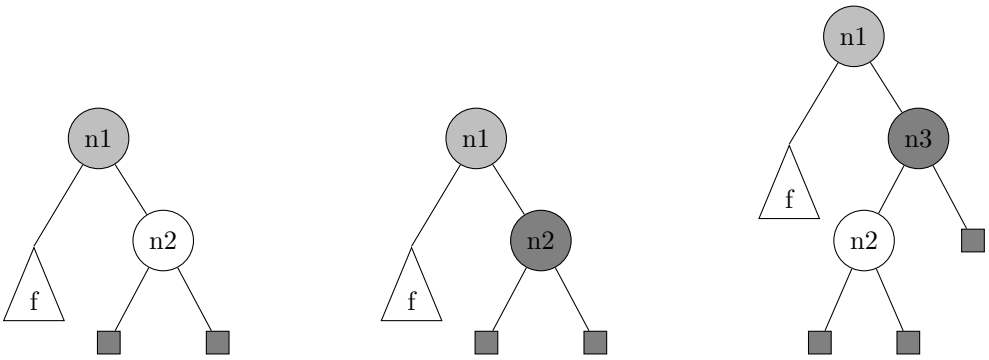
Ces 4 cas sont équilibrés avec le même résultat



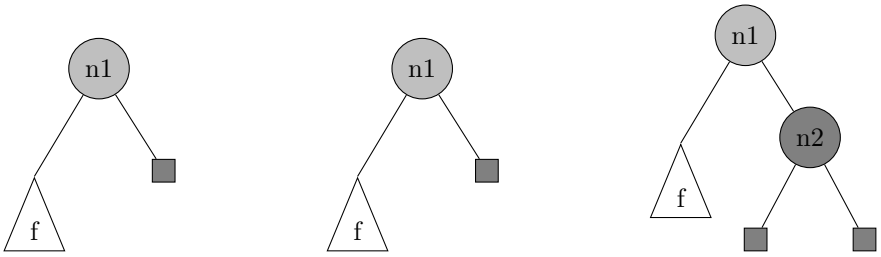
Exercice 4
Modifier l'insertion en ajoutant l'appel à une fonction d'équilibrage à chaque construction de nœud et en colorant la racine en noir. On écrira, bien sur, la fonction d'équilibrage.

3.2 Suppression de la racine

Pour supprimer la racine on doit supprimer le maximum du fils gauche, s'il existe, et le mettre à la place de la racine.
Le maximum du fils gauche est un nœud (rouge ou noir) dont le fils droit est vide donc, en raison de la hauteur noire, le fils gauche est vide ou est un nœud rouge avec deux fils vides. Il y a donc 3 cas (le quatrième, rouge/rouge, est impossible). La couleur de `n1` n'est pas signifiante pour l'instant.

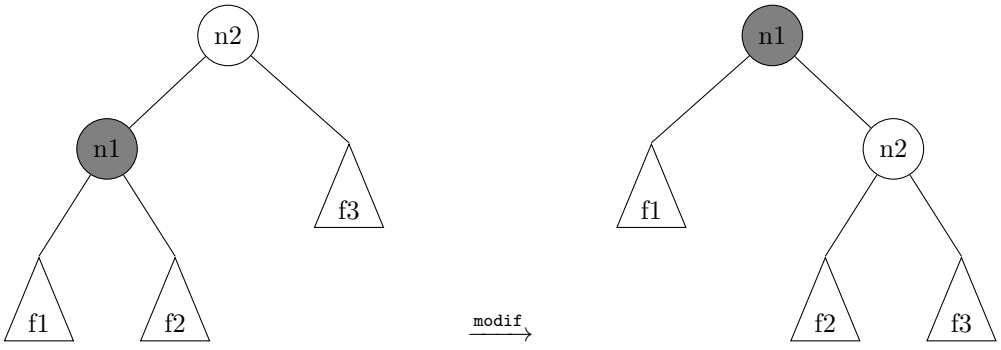


En voici des résultats possible après suppression.



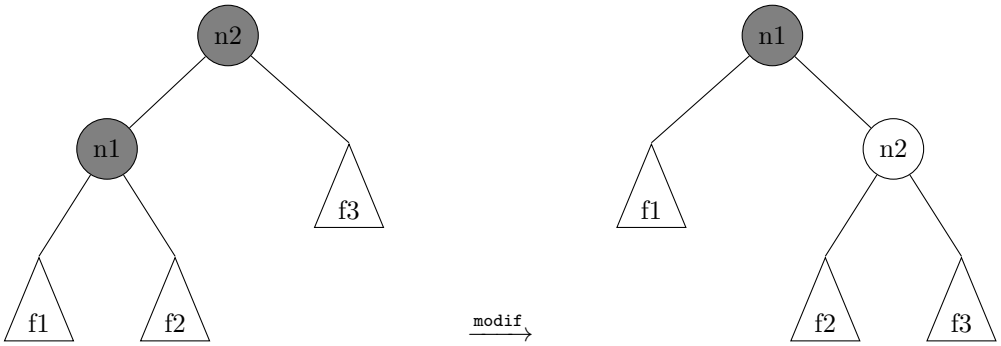
Dans le second cas l'arbre a perdu une unité dans la hauteur noire. Il y a un déséquilibre de hauteur noire qui apparaît : un nœud a deux fils qui sont des arbres rouge-noir mais dont les hauteurs noires diffèrent de 1. On va essayer de remonter ce déséquilibre. Si on parvient à la racine le problème disparaît. Pour l'instant c'est le fils droit qui est diminué en hauteur noire car on remonte depuis le maximum.
On se place donc dans le cas d'un fils gauche de hauteur noire supérieure de 1 (donc au moins 2) à celle du fils droit ; ce fils n'est donc pas vide. On va alors chercher un nœud noir pour en remonter la couleur et ainsi équilibrer les hauteurs noires.

- Si la racine est rouge alors son fils gauche est noir ; on peut corriger le déséquilibre et rétablir la hauteur noire



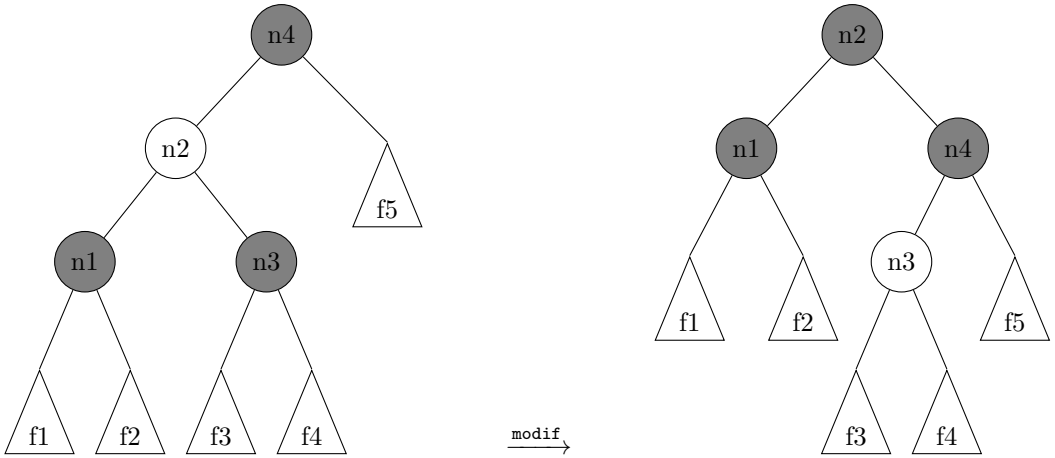
On remarque qu'on peut créer un conflit entre `n2` et `f2` si la racine de `f2` est rouge : il faudra équilibrer comme dans le cas de l'ajout.

- Si la racine et le fils gauche sont noirs on peut corriger le déséquilibre mais on propage le déficit de noir.



Ici encore il faudra équilibrer à cause du conflit possible entre `n2` et `f2`.

- Si la racine est noire et le fils gauche est rouge alors ce fils gauche a des fils qui sont noirs, en particulier le fils droit. On peut corriger le déséquilibre en rétablissant la hauteur noire.



Cependant le fils droit doit être équilibré (et non l'arbre entier) à cause d'un conflit possible entre `n3` et `f3` ou `f4`. Dans le cas où les racines de `f3` et `f4` sont rouges on peut remarquer que la correction du conflit entre `n3` et l'un des deux supprime aussi le conflit avec l'autre. On remarque aussi que l'arbre de racine `n1` est inchangé.

Exercice 5
Ré-écrire la fonction `suppressionMax arbre`.

Exercice 6
Ré-écrire la fonction `suppression x arbre`.

4 Solutions

Solution de l'exercice 1 -

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

let rec ajoutFeuille x arbre =
  match arbre with
  |Vide -> Noeud(Rouge,Vide,x,Vide)
  |Noeud(k,g,n,d) when (n = x) -> arbre
  |Noeud(k,g,n,d) when (x < n) -> Noeud(k,(ajoutFeuille x g),n
     ,d)
  |Noeud(k,g,n,d) ->  Noeud(k,g,n,(ajoutFeuille x d));;

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

let rec suppressionMax arbre =
  match arbre with
  |Vide -> raise(Failure "Arbre vide")
  |Noeud(_,g,_,Vide) -> g
  |Noeud(k,g,n,d) -> Noeud(k,g,n,suppressionMax d);;

let suppressionRacine arbre =
  match arbre with
  |Vide -> Vide
  |Noeud(_,g,_,Vide) -> g
  |Noeud(k,g,n,d) ->  let r = maxArbre g in
                      Noeud(k,suppressionMax g,r,d);;

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

Solution de l'exercice 2 -
On note $\mathcal{P}(\mathtt{a})$ la propriété que l'arbre `a` contient au moins $2^{h_N(a)} - 1$ nœuds internes.
Si `a` est vide, sa hauteur noire est nulle (la racine n'est pas prise en compte) et il admet $0 = 2^0 - 1$ nœud interne. Ainsi $\mathcal{P}(\mathtt{Vide})$ est vraie.
Pour `a = Noeud(k, n, g, d)` on a $h_N(\mathtt{g}) = h_N(\mathtt{a})$ si la couleur de `g` est blanche et $h_N(\mathtt{g}) = h_N(\mathtt{a}) - 1$ si la couleur de `g` est noire. De même pour `d`.
Le nombre de nœuds internes de `a`, $|\mathtt{a}|$, est $|\mathtt{a}| = |\mathtt{g}| + |\mathtt{d}| + 1$ d'où

$$|\mathtt{a}| \geqslant 2^{h_N(\mathtt{g})} - 1 + 2^{h_N(\mathtt{d})} - 1 + 1 \geqslant 2^{h_N(\mathtt{a})-1} + 2^{h_N(\mathtt{a})-1} - 1 = 2^{h_N(\mathtt{a})} - 1$$

La propriété est vérifiée pour tout arbre par induction structurelle.

Solution de l'exercice 3 -
D'après la question précédente la hauteur noire vérifie $l \geqslant 2^{h_N} - 1$ donc $h_N \leqslant \log_2(l+1)$.
Le long d'une branche il y a au maximum un nœud blanc entre deux nœuds noirs donc la hauteur est majorée par $2h_N$ d'où le résultat demandé.

Solution de l'exercice 4 -

```
let equR arbre =
  match arbre with
  |Noeud(Noir,
         Noeud(Rouge,Noeud(Rouge,f1,n1,f2),n2,f3),
         n3,
         f4) -> Noeud(Rouge,
                      Noeud(Noir,f1,n1,f2),
                      n2,
                      Noeud(Noir,f3,n3,f4))
  |Noeud(Noir,
         Noeud(Rouge,f1,n1,Noeud(Rouge,f2,n2,f3)),
         n3,
         f4) -> Noeud(Rouge,
                      Noeud(Noir,f1,n1,f2),
                      n2,
                      Noeud(Noir,f3,n3,f4))
  |Noeud(Noir,
         f1,
         n1,
         Noeud(Rouge,Noeud(Rouge,f2,n2,f3),n3,f4))
             -> Noeud(Rouge,
                      Noeud(Noir,f1,n1,f2),
                      n2,
                      Noeud(Noir,f3,n3,f4))
  |Noeud(Noir,
         f1,
         n1,
         Noeud(Rouge,f2,n2,Noeud(Rouge,f3,n3,f4)))
             -> Noeud(Rouge,
                      Noeud(Noir,f1,n1,f2),
                      n2,
                      Noeud(Noir,f3,n3,f4))
  |_ -> arbre;;

let racineNoire arbre =
  match arbre with
  |Vide -> Vide
  |Noeud(_,g,n,d) -> Noeud(Noir,g,n,d);;

let ajoutFeuille x arbre =
  let rec aux arbre =
    match arbre with
    |Vide -> Noeud(Rouge,Vide,x,Vide)
    |Noeud(k,g,n,d) when (n = x) -> arbre
    |Noeud(k,g,n,d) when (x < n) -> equR (Noeud(k,(aux g),n,d)
       )
    |Noeud(k,g,n,d) ->  equR (Noeud(k,g,n,(aux d))) in
  racineNoire (aux arbre);;
```

Solution de l'exercice 5 -

```
let equN_d arbre bon =
  match arbre, bon with
  |Noeud(Rouge,Noeud(_,f1,n1,f2),n2,f3), false
    -> equR (Noeud(Noir,f1,n1,Noeud(Rouge,f2,n2,f3))), true
  |Noeud(Noir,Noeud(Noir,f1,n1,f2),n2,f3), false
    -> equR (Noeud(Noir,f1,n1,Noeud(Rouge,f2,n2,f3))), false
  |Noeud(Noir,Noeud(Rouge,f1,n1,Noeud(_,f2,n2,f3)),n3,f4),
     false
    -> Noeud(Noir,f1,n1,equR (Noeud(Noir,f2,n2,Noeud(Rouge,f3,
       n3,f4)))), true
  |_ -> arbre, true;;

let rec suppressionMax arbre =
  match arbre with
  |Vide -> raise(Failure "Arbre vide")
  |Noeud(Rouge,g,n,Vide) -> Vide, true
  (* g ne peut être que Vide *)
  |Noeud(Noir,Vide,n,Vide) -> Vide, false
  |Noeud(Noir,Noeud(k,g1,n1,d1),n,Vide) -> Noeud(Noir,g1,n1,d1
     ), true
  (* Comme le fils droit est Vide, k est rouge et g1 et d1
     sont vides *)
  |Noeud(k,g,n,d) ->  let d',bon = suppressionMax d in
                      equN_d (Noeud(k,g,n,d')) bon ;;
```

Solution de l'exercice 6 -

```
let equN_g arbre bon =
  match arbre, bon with
  |Noeud(Rouge,f1,n1,Noeud(_,f2,n2,f3)), false
    -> equR (Noeud(Noir,Noeud(Rouge,f1,n1,f2),n2,f3)), true
  |Noeud(Noir,f1,n1,Noeud(Noir,f2,n2,f3)), false
    -> equR (Noeud(Noir,Noeud(Rouge,f1,n1,f2),n2,f3)), false
  |Noeud(Noir,f1,n1,Noeud(Rouge,Noeud(_,f2,n2,f3),n3,f4)),
     false
    -> Noeud(Noir,equR (Noeud(Noir,Noeud(Rouge,f1,n1,f2),n2,f3
       )),n3,f4), true
  |_ -> arbre, true;;

let suppression x arbre =
  let rec aux arbre =
    match arbre with
    |Vide -> Vide, true
    |Noeud(k,Vide,n,d) when n = x -> d, (k = Rouge)
    |Noeud(k,g,n,d) when n = x -> let n' = maxArbre g in
                                  let g',bon = suppressionMax
                                     g in
                                  equN_g (Noeud(k,g',n',d))
                                     bon
    |Noeud(k,g,n,d) when n < x -> let d',bon = aux d in
                                  equN_d (Noeud(k,g,n,d')) bon
    |Noeud(k,g,n,d) -> let g',bon = aux g in
                       equN_g (Noeud(k,g',n,d)) bon in
  let a, _ = aux arbre in racineNoire a;;
```

TP IV

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 revenir au type récursif d'arbre binaire.

1 Présentation

On veut implémenter le type de données abstraite de file de priorité. Dans ce TP, la priorité maximale correspondra à une clé minimale. On manipule des ensembles de couples *clé*×*valeur* où la clé est un entier et la valeur est de type α (`'a`).
Dans ce TP nous simplifierons l'étude en ne manipulant que les clés entières.
On a donc besoin d'un type `fdp` avec les fonctions

1. `fpVide : int -> fdp`,
2. `estVide : fdp -> bool`,
3. l'ajout est `ajouter : fdp -> int -> fdp`,
   on crée un nouvel objet
4. `premier : fdp -> int` ,
   permet de voir l'élément de priorité maximale (de clé minimale) et sa priorité,
5. le retrait de l'élément prioritaire est `enlever : fdp -> fdp`.

Le type sera classiquement

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

La taille d'un arbre a est notée $|a|$.
Notre structure de file de priorité sera implémentée par des tas de Braun croissants ; ce sont des arbres binaires qui vérifient les deux propriétés suivantes.
croissance : l'étiquette de tout nœud est inférieure aux étiquettes de ses fils,
équilibre fort : pour tout nœud, `Noeud(g,x,d)`, $|d| \leqslant |g| \leqslant |d| + 1$.
Ces propriétés devront être maintenues dans les constructions.

2 Fonctions

Exercice 1 — Test
Écrire une fonction `estBraun : fdp -> bool` *qui renvoie true ou* `false` *selon que l'arbre passé en paramètre est ou non un tas de Braun.*
On pourra écrire une fonction auxiliaire qui renvoie ce booléen, la taille de l'arbre et sa racine.

Exercice 2 — Premières fonctions
Écrire les fonctions `fpVide`, `estVide` *et* `premier`.

Ajout

On remarque que, pour garder la condition d'équilibre fort lors de l'ajout, on ne peut ajouter un élément que sur le fils de droite mais qu'alors celui-ci devient le fils de plus grande taille et il faut inverser les fils.

Exercice 3 — Ajouter
Écrire une fonction `ajouter`.

Retrait

Pour pouvoir enlever l'élément prioritaire on aura besoin de reconstruire un tas de Braun à partir de deux fils et d'un entier. On suppose, dans la question suivante que `a1` et `a2` sont deux tas de braun tels que $|a_2| \leqslant |a_1| \leqslant |a_2| + 1$.

Exercice 4 — Enlever un élément
Écrire une fonction `extractionGauche : arbre -> int * arbre` *qui reçoit un tas de Braun et qui renvoie l'élément le plus à gauche et un tas de Braun contenant les éléments restants.*

Exercice 5 — Reconstitution
Écrire une fonction `union : int -> arbre -> arbre -> arbre` *qui reçoit un entier k et deux tas de Braun de taille adéquate et qui renvoie un tas de Braun contenant les étiquettes des deux arbres et k en plus.*

Exercice 6 — Enlever la racine
En déduire une fonction `enlever : arbre -> arbre` *qui renvoie un tas de Braun obtenu en enlevant la racine du tas passé en paramètre.*

Exercice 7 — Question facultative
Déterminer une autre manière d'enlever l'élément prioritaire en remontant la racine d'un fils.

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 8 — Une première tentative
Programmer cet algorithme.

Les 20 premiers nombres de Hamming sont
1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36.
L'algorithme précédent va produire
1, 2, 3, 4, 5, 6, 6, 8, 9, 10, 10, 12, 12, 15, 15, 16, 18, 18, 18.
Le nombre 900 apparaît 90 fois ! En effet les éléments sont insérés plusieurs fois.

Exercice 9 — Correction
Écrire une fonction qui renvoie vraiment le n-ième nombre de Hamming.
On pourra vérifier : le 1000-ième nombre de Hamming est 51 200 000.

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 10 — 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 11 — 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 12 — 3 filières
Déterminer le premier nombre taupin qui peut peut l'être avec 3 chemins.

Exercice 13 — 4 filières
Déterminer le premier nombre taupin qui peut peut l'être avec 4 chemins.

5 Propriétés des arbres fortement équilibrés

Dans cette partie on suppose que `a` est un arbre fortement équilibré, c'est-à-dire qu'il vérifie la condition d'équilibre fort.

Exercice 14

1. *Déterminer la taille des fils d'un arbre fortement équilibré a de taille n.*
2. *Prouver qu'un arbre fortement équilibré de taille 2^k-1 est complet.*
3. *Prouver qu'un arbre fortement équilibré de taille $n \geqslant 1$ avec $2^k \leqslant n < 2^{k+1}$ est de hauteur k.*
4. *Prouver qu'un arbre fortement équilibré est quasi-complet.*
5. *Montrer que la feuille la plus à gauche d'un arbre fortement équilibré est à la profondeur maximale.*

6 Solutions

Solution de l'exercice 1 -

```
let racine arbre =
   match arbre with
   |Vide -> failwith "L'arbre est vide"
   |Noeud(_,x,_) -> x;;

let estBraun arbre =
   let rec aux arbre =
      match arbre with
      |Vide -> true, 0, max_int
      |Noeud(Vide,x,Vide) -> true, 1, x
      |Noeud(g,x,Vide) -> let b, n, y = aux g in (n=1), (n+1),
          x
      |Noeud(Vide, x, d) -> let b, n, y = aux d in false, (n
         +1), x
      |Noeud(g, x, d) -> let bg, ng, xg = aux g in
                         let bd, nd, xd = aux d in
                         (xg >= x) && (xd >= x) && bg && bd &&
                             (nd <= ng) && (nd <= ng+1),
                         ng + nd + 1,
                         x
  in let b, n, x = aux arbre in b;;
```

Solution de l'exercice 2 -

```
let fpVide n = Vide;;

let estVide tas =
   tas = Vide;;

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

Solution de l'exercice 3 -

```
let rec ajouter x t =
   match t with
   |Vide -> Noeud(Vide, x, Vide)
   |Noeud(g,y,d) ->  if x > y
                     then Noeud(ajouter x d, y, g)
                     else Noeud(ajouter y d, x, g);;
```

Solution de l'exercice 4 -

```
let rec extractionGauche arbre =
   match arbre with
   |Vide -> failwith "L'arbre est vide"
   |Noeud(Vide, x,_) -> x, Vide
   |Noeud(g, x, d) -> let y, g1 = extractionGauche g in y,
      Noeud(d, x, g1);;
```

Solution de l'exercice 5 - On commence par les cas où la racine finale n'est pas k, les autres cas se construisent simplement.

```
let rec union k a1 a2 =
  match a1, a2 with
  |Noeud(g1, r1, d1), Noeud(g2, r2, d2) when (r1 < k) && (r1 <
      r2)
        -> Noeud(union k g1 d1, r1, a2)
  |Noeud(g1, r1, d1), Noeud(g2, r2, d2) when (r2 < k) && (r2
     <= r1)
        -> Noeud(a1, r2, union k g2 d2)
  |Noeud(Vide, r, Vide), Vide when (r < k)
        -> Noeud(Noeud(Vide, k, Vide), r, Vide)
  |_, _ -> Noeud(a1, k, a2);;
```

Solution de l'exercice 6 -

```
let enlever arbre =
  match arbre with
  |Vide -> failwith "L'arbre est vide"
  |Noeud(Vide, _, _) -> Vide
  |Noeud(g, x, d) -> let y, g1 = extractionGauche g in union y
      d g1;;
```

Solution de l'exercice 7 - On peut enlever récursivement la racine (`rg`) du fils gauche en modifiant celui-ci en `g1`.
On note `d = Noeud(gd, rd, dd)` le fils droit. Si `rg <= rd` on renvoie `Noeud(d, rg, g1)` sinon on renvoie `Noeud(union rg gd dd, rd, g1)`.

```
let rec enlever1 arbre =
   match arbre with
   |Vide -> failwith "L'arbre est vide"
   |Noeud(g, r, Vide) -> g
   |Noeud(g, r, (Noeud(gd, rd, dd) as d))
           -> let rg = premier g in
              let rd = premier d in
              let g1 = enlever1 g in
              if rg < rd then Noeud(d, rg, g1)
                         else Noeud(union rg gd dd, rd, g1);;
```

Solution de l'exercice 8 -

```
let rec hamming n =
   let rec aux n tas =
      match n with
      |1 -> premier tas
      |n -> let p = premier tas in
            let t1 = enlever tas in
            let t2 = ajouter (5*p) t1 in
            let t3 = ajouter (3*p) t2 in
            let t4 = ajouter (2*p) t3 in
            aux (n-1) t4
  in let t0 = ajouter 1 (fpVide 0) in aux n t0;;
```

Solution de l'exercice 9 - 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 tas =
      match n with
      |1 -> premier tas
      |n -> let p = premier tas in
            let t1 = enlever tas in
            let t2 = ajouter (5*p) t1 in
            if p mod 5 = 0
            then aux (n-1) t2
            else let t3 = ajouter (3*p) t2 in
                 if p mod 3 = 0
                 then aux (n-1) t3
                 else let t4 = ajouter (2*p) t3 in
                      aux (n-1) t4
  in let t0 = ajouter 1 (fpVide 0) in aux n t0;;
```

Solution de l'exercice 10 -

```
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 11 -

```
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 12 -

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

On trouve 20480

Solution de l'exercice 13 - 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 !

Solution de l'exercice 14 -

1. La taille des fils est n_g et n_d avec $n_d \leqslant n_g \leqslant n_d + 1$ et $n_g + n_d + 1 = n$.
   On a donc $2n_g \leqslant n_g + n_d + 1 \leqslant n$ et $2n_g \geqslant n_g + n_d = n - 1$.
   Si n est pair, $n = 2p$, on a donc n_g entier avec $p - \frac{1}{2} \leqslant n_g \leqslant p$ puis $n_g = p$ puis $n_d = n - n - g - 1 = p - 1$.
   Si $n = 2p + 1$ est impair, alors $p \leqslant n_g \leqslant p + \frac{1}{2}$ donne $n_g = p$ et $n_d = p$.
   On remarque qu'on a $n_g = \left\lfloor \frac{n}{2} \right\rfloor$.
2. On démontre le résultat par récurrence sur k.
   Pour $k = 1$ on a un arbre réduit à un seul nœud.
   Si la propriété est vrai pour les arbres fortement équilibré de taille $2^{k-1} - 1$ on conclut pour un arbre fortement équilibré de taille $2^k - 1$ cas ses fils sont fortement équilibrés de taille $2^{k-1} - 1$ donc complets.
3. On prouve le résultat par récurrence sur n.
   Pour $n = 1$, la hauteur est 0 et $2^0 \leqslant 1 < 2^1$.

Pour $n = 2$, la hauteur est 1 et $2^1 \leqslant 1 < 2^2$.
On suppose la propriété vraie pour tout arbre fortement équilibré de taille $m < n$.
On a $2^k \leqslant n < 2^{k+1}$ donc $2^{k-1} \leqslant n_g = \left\lfloor \frac{n}{2} \right\rfloor < 2^{k+1}$ donc le fils gauche est de hauteur $k-1$. Sauf dans le cas $n = 2^k$, la taille du fils droit vérifie la même inégalité donc le fils droit est aussi de hauteur $k-1$. Ainsi l'arbre est de hauteur k et ses deux fils sont de hauteur $k-1$. Si $n = 2^k$ son fils gauche est de hauteur $k-1$ et son fils droit est de hauteur $k-2$ donc l'arbre est de hauteur k.

4. Ici encore on procède par récurrence sur la taille.
On a vu ci-dessus que les deux fils d'un arbre fortement équilibré de hauteur h sont de même hauteur $h-1$ sauf dans le cas d'une taille 2^h. Dans ce dernier cas le fils droit est de taille $2^{h-1} - 1$ donc est complet de hauteur $h-2$. Dans tous les cas les fils d'un arbre fortement équilibré de hauteur h n'ont que des feuilles à la hauteur $h-1$ ou $h-2$: l'arbre est quasi-complet.
5. Toujours une récurrence.

TP V

IF-EXPRESSIONS

Résumé

Dans ce T.P. nous allons utiliser un nouveau connecteur logique qui va permettre d'implémenter un algorithme de décision qui reconnaît les tautologie, cet algorithme est différent des tables de vérité classiques. Il nous donnera de plus une réfutation lorsque la proposition n'est pas valide, c'est-à-dire une valuation qui donne la valeur **Faux** *lorsqu'on l'applique à la proposition.*
On testera la satisfiabilité d'une formule F en appliquant le test à $\neg F$.

1 IF-expressions

Formules classiques

On considère les formules logiques vues en cours.
On ajoute 3 nouveaux symboles.

- Les propositions VRAI et FAUX. Sémantiquement ce sont des constantes qui prennent la valeur **Vrai** ou **Faux** respectivement pour toute valuation.
  Pour toute formule F, VRAI est équivalente à $F \vee \neg F$ (resp. à $F \wedge \neg F$).
  Elles seront considérées comme des formules élémentaires, au même titre que les variables.
- Le connecteur $\rightarrow$. $F \rightarrow G$ est équivalent à $G \vee \neg F$.
  Son but est de permettre des écritures simplifiées.

On représentera les propositions par le type Caml suivant :

```
type proposition =
  |Var of string
  |Vrai
  |Faux
  |Neg of proposition
  |Donc of proposition * proposition
  |Et of proposition * proposition
  |Ou of proposition * proposition;;
```

IF-expressions

Pour réaliser cet objectif on introduit la notion de **IF-expression** :
une IF-expression est soit une variable, soit VRAI ou FAUX, soit une proposition de la forme ITE(M, P, Q) où M, P et Q sont des IF-expression.
La sémantique d'une IF-expression ITE(M, P, Q) est déterminée récursivement comme prenant la valeur de P si M est évaluée à **Vrai** et la valeur de Q sinon.

Autrement dit ITE (M, P, Q) se traduit par "*si M alors P sinon Q*".
ITE est un acronyme pour If Then Else.
On représentera les IF-expressions par le type Caml suivant :

```
type ifExpr =
  |Var_ite of string
  |Vrai_ite
  |Faux_ite
  |Ite of ifExpr * ifExpr * ifExpr;;
```

1.1 Traduction

Exercice 1
Exprimer $\neg P$, $P \wedge Q$, $P \vee Q$ et $(P \to Q)$ par des formules ne comportant un seul connecteur ITE .

Exercice 2
En déduire une fonction `prop2if` *qui transforme une proposition en une IF-expression équivalente.*

Exercice 3
Donner les IF-expressions correspondant aux trois propositions suivantes :

$$\begin{aligned} P_1 &: (A \wedge B) \to A \\ P_2 &: A \to (A \wedge B) \\ P_3 &: \big(A \to (A \to B)\big) \to B \end{aligned}$$

1.2 Forme normale

On dit qu'une IF-expression est NORMALE si elle est
- une variable,
- VRAI ou FAUX
- ou ITE (s, P, Q) où s est une variable et P et Q sont deux IF-expressions normales.

Exercice 4
Écrire une fonction `estNormale : ifExpr -> bool`
qui teste si une IF-expression est normale.

On considère la fonction ϕ définie récursivement par
- $\phi(\text{VRAI}) = \phi(\text{FAUX}) = 1$
- $\phi(s) = 1$ pour toute variable s
- $\phi\big(\text{ITE}\,(M, P, Q)\big) = \phi(M)\big(1 + \phi(P) + \phi(Q)\big)$

Exercice 5
Prouver que les transformations suivantes diminuent la valeur de ϕ.

$$\begin{aligned} \text{ITE}\,(\text{VRAI}, P, Q) &\rightsquigarrow P \\ \text{ITE}\,(\text{FAUX}, P, Q) &\rightsquigarrow Q \\ \text{ITE}\,(\text{ITE}\,(M, P, Q), R, S) &\rightsquigarrow \text{ITE}\,\big(M, \text{ITE}\,(P, R, S), \text{ITE}\,(Q, R, S)\big) \end{aligned}$$

En déduire toute IF-expression est équivalente à une IF-expression normale.

Exercice 6
Écrire une fonction `normalise : ifExpr -> ifExpr`
qui transforme toute IF-expression en une IF-expression normale équivalente.

Exercice 7
Donner les IF-expressions normales correspondant à P_1, P_2 et P_3.

2 Décision sur les IF-expressions

2.1 Valuation partielle

On appelle `valuation partielle` une fonction des variables dans $\{\mathbf{Vrai}, \mathbf{Faux}\}$ dont le domaine est fini. On représentera une valuation partielle par une liste d'association :

```
type valuation = (string * bool) list;;
```

Exercice 8
Écrire une fonction `defini : string -> valuation -> bool` *qui renvoie* `true` *ou* `false` *selon que la valuation partielle est ou non définie pour une variable représentée par une chaîne de caractères.*

Exercice 9
Écrire une fonction `valeur : string -> valuation -> bool` *qui renvoie la valeur prise par une valuation partielle pour une variable représentée par une chaîne de caractères.*

Une valuation partielle α définit une valeur booléenne pour toute proposition P dont les variables sont dans le domaine de définition de α. On dit alors que α est compatible avec P.
On prolonge la notion de tautologie par la notion de α-tautologie de la manière suivante :

Définition :
Une expression P est une α-tautologie si elle est vraie pour toutes les valuations partielles qui prolongent α et qui sont compatibles avec P.

Ainsi P est une tautologie si et seulement si elle est une ω-tautologie où ω est la valuation partielle de domaine vide.

2.2 Algorithme

Nous allons étudier un algorithme qui détermine si une IF-expression normale P est une α-tautologie ou, si elle ne l'est pas, détermine une valuation partielle compatible avec P, qui prolonge α et qui réfute P.
Cet algorithme est décrit par les règles suivantes.

1. Si p est `True` ou `False` le résultat est immédiat.
2. Si p est une variable `s` alors trois cas se présentent :
   - si $\alpha(s)$ est défini et vaut $\mathbf{V}$ alors P est une α-tautologie,
   - si $\alpha(s)$ est défini et vaut $\mathbf{F}$ alors P n'est pas une α-tautologie, réfutée par α,
   - si $\alpha(s)$ n'est pas défini alors P n'est pas une α-tautologie
     et on obtient une réfutation en étendant α par $\alpha(s) = \mathbf{F}$.
3. Si P est une IF-expression `ite(s, Q, R)` alors
   - si $\alpha(s)$ est défini et vaut Vrai
     alors P est une α-tautologie si et seulement si Q est une α-tautologie,
   - si $\alpha(s)$ est défini et vaut Faux
     alors P est une α-tautologie si et seulement si R est une α-tautologie,
   - si $\alpha(s)$ n'est pas défini alors on définit deux prolongements de α, α_v et α_f, par $\alpha_v(s) = \mathbf{V}$ et $\alpha_f(s) = \mathbf{F}$. P est une α -tautologie si et seulement si Q est une α_v-tautologie et R est une α_f-tautologie.

Exercice 10
Si `P = ite(s, Q, R)` *n'est pas une α-tautologie donner une réfutation dans chacun des 3 cas précédents.*

On définit donc le type suivant pour gérer le résultat de l'algorithme.

```
type resultat = Tautologie | Refutation of valuation;;
```

Exercice 11
Écrire une fonction `decisionP : valuation -> ifExpr -> resultat` *qui décide si une IF-expression normale est une tautologie par rapport à une valuation (partielle) et donne une réfutation si elle ne l'est pas.*

Exercice 12
En déduire une fonction `decisionIF : ifExpr -> resultat` *qui décide si une IF-expression normale est une tautologie et donne une réfutation si elle ne l'est pas.*

Exercice 13
Écrire enfin une fonction `decision : proposition -> resultat` *qui décide si une proposition est une tautologie et donne une réfutation si elle ne l'est pas.*

2.3 Applications

Exercice 14
Tester les propositions P_1, P_2 et P_3.

Les dahuts

Le dahut est une espèce très rare de bouquetin qui a la particularité d'avoir les deux pattes d'un coté plus courtes que les autres. Il vit donc dans la montagne en ayant toujours le sommet du coté de ses pattes courtes. Un dahut est appelé de dextrogyre ou lévogyre selon les cas.
Ils ont d'autres particularités :

- Tout dahut non lévogyre a des rayures noires.
- Tout dahut qui a des oreilles blanches est lévogyre et vit dans les forêts.
- Tout dahut a des oreilles blanches ou n'a pas de rayures noires.
- Les dahuts qui vivent dans les forêts ne mangent pas de mulots.
- Un dahut mange des mulots si et seulement s'il est lévogyre.
- Tout dahut lévogyre a des oreilles blanches.

Exercice 15
Prouver que les dahuts n'existent pas.

La princesse

Un chevalier doit partir délivrer des princesses. Arrivé à une intersection, il a le choix entre trois chemins, chacun précédé d'un panneau. Le gardien des lieux lui déclare :
"*Parmi ces trois chemins, l'un mène à une princesse, et son panneau dit la vérité.*
Quant aux deux autres, ils aboutissent à une mort certaine.
Au moins l'un des panneaux ment."
Voici ce qui est écrit à l'entrée de chaque chemin :

- Le deuxième chemin mène à une mort certaine.
- Ce chemin mène à une mort certaine.
- Le premier chemin mène à une mort certaine.

Exercice 16
Comment délivrer la princesse ?

3 Solutions

Solution de l'exercice 1 - $\neg P \equiv \text{ITE}(P, \text{FAUX}, \text{VRAI})$,
$P \wedge Q \equiv \text{ITE}(P, Q, \text{FAUX})$,
$P \vee Q \equiv \text{ITE}(P, \text{VRAI}, Q)$,
$P \to Q \equiv \text{ITE}(P, Q, \text{VRAI})$.

Solution de l'exercice 2 -

```
let rec prop2if p =
  match p with
  |Var s      -> Var_ite s
  |Vrai       -> Vrai_ite
  |Faux       -> Faux_ite
  |Neg q      -> Ite (prop2if q, Faux_ite , Vrai_ite)
  |Donc (q,r) -> Ite (prop2if q, prop2if r, Vrai_ite)
  |Et (q,r)   -> Ite (prop2if q, prop2if r, Faux_ite )
  |Ou (q,r)   -> Ite (prop2if q, Vrai_ite , prop2if r);;
```

Solution de l'exercice 3 - $P_1 \equiv \text{ITE}\big(\text{ITE}(A, B, \text{FAUX}), A, \text{VRAI}\big)$,
$P_2 \equiv \text{ITE}(A, \text{ITE}(A, B, \text{FAUX}), \text{VRAI})$,
$P_3 \equiv \text{ITE}(\text{ITE}(A, \text{ITE}(A, B, \text{VRAI}), \text{VRAI}), B, \text{VRAI})$

Solution de l'exercice 4 -

```
let rec estNormale p =
  match p with
  |Ite (Var_ite _, p, q) -> (estNormale p) || (estNormale q)
  |Ite (_,  _,  _) -> false
  |_ -> true;;
```

Solution de l'exercice 5 - Chacune de ces transformation donne une IF-expression équivalente à l'expression initiale.
On a bien $\phi(P) < 1 + \phi(P) + \phi(Q)$, $\phi(Q) < 1 + \phi(P) + \phi(Q)$ et
$\phi(M)\big(1 + \phi(P)K + \phi(Q)K\big) < \phi(M)\big(1 + \phi(P) + \phi(Q)\big)K$ avec $K = \big(1 + \phi(R) + \phi(S)\big)$.
Si une If-expression n'est pas normale elle contient une composante de la forme $\text{ITE}(M, P, Q)$ avec $M = \text{VRAI}$, $M = \text{FAUX}$ ou $M = \text{ITE}(M', P', Q')$.
On peut alors appliquer une des transformations en diminuant strictement la valeur de ϕ.
Si les transformations ne permettaient jamais d'arriver à une forme normale on aurait une suite de formules équivalentes M_n et la suite des $\big(\phi(M_n)\big)$ formerait une suite strictement décroissante infinie d'entiers positifs ce qui est impossible.
On parvient donc à une IF-expression normale après un nombre fini de transformations.

Solution de l'exercice 6 -

```
let rec normalise prop =
  match prop with
  |Ite (Var_ite s, p, q) -> Ite (Var_ite s, normalise p,
     normalise q)
  |Ite (Vrai_ite, p, _) -> normalise p
  |Ite (Faux_ite , _, q) -> normalise q
  |Ite (Ite (m, p, q), r, s) -> normalise (Ite (m, Ite (p, r,
     s), Ite (q, r, s)))
  |p -> p;;
```

Solution de l'exercice 7 - $P_1 \equiv \text{ITE}(A, \text{ITE}(B, A, \text{VRAI}), \text{VRAI})$,
$P_2 \equiv \text{ITE}(A, \text{ITE}(A, B, \text{FAUX}), \text{VRAI})$,
$P_3 \equiv (A, \text{ITE}(A, \text{ITE}(B, B, \text{VRAI}), B), B)$.

Solution de l'exercice 8 -

```
let rec defini x v =
  match v with
  |[] -> false
  |(s, _)::vv -> (s = x) || (defini x vv);;
```

Solution de l'exercice 9 -

```
let rec valeur x v =
  match v with
  |[] -> failwith "non valué"
  |(s,b)::vv ->  if s = x then b else valeur x vv;;
```

Solution de l'exercice 10 -

- Si $\alpha(s) = \mathbf{V}$ alors une réfutation de Q est une réfutation de P.
- Si $\alpha(s) = \mathbf{F}$ alors une réfutation de R est une réfutation de P.
- si $\alpha(s)$ n'est pas défini alors une réfutation de P est une réfutation de Q si Q n'est pas une α_v-tautologie ou une réfutation de R si R n'est pas une α_f-tautologie .

Solution de l'exercice 11 - On applique l'algorithme

```
let rec decisionP valu prop =
  match prop with
  |Vrai_ite -> Tautologie
  |Faux_ite  -> Refutation valu
  |Var_ite s when defini s valu -> if valeur s valu
                                   then Tautologie
                                   else Refutation valu
  |Var_ite s -> Refutation ((s, false)::valu)
  |Ite (Var s, p, q) when defini s valu
     -> if valeur s valu
        then decisionP valu p
        else decisionP valu q)
  |Ite (Var s, p, q)
     -> let res = decisionP ((s, true)::valu) p in
        if res  = Tautologie
        then decisionP ((s, false)::valu) q
        else res
  |_ -> failwith "ceci ne devrait pas arriver ";;
```

Solution de l'exercice 12 -

```
let decisionIF p = decisionP [] p;;
```

Solution de l'exercice 13 -

```
let decision p = decisionIF (normalise (prop2if p));;
```

Solution de l'exercice 14 - P_1 est une tautologie,
P_2 est réfutée par $A = \text{VRAI}$ et $B = \text{FAUX}$,
P_3 est réfutée par $A = \text{FAUX}$ et $B = \text{FAUX}$.

Solution de l'exercice 15 -

```
let l  = Var "Levogyre";;
let rn = Var "RayuresNoires";;
let ob = Var "OreillesBlanches";;
let f  = Var "vitForet";;
let mm = Var "mangeMulot";;

let d1 = Donc (Neg l, rn);;
let d2 = Ou (ob, Neg rn);;
let d3 = Donc (f, Neg mm);;
let d4 = Et(Donc (mm, l), Donc (l, mm));;
let d5 = Donc (ob, Et (l, f));;
let d6 = Donc (l, ob);;

let p = Neg (Et (Et (Et (d1, d2), Et (d3, d4)), Et (d5, d6)))
   ;;
```

p est une tautologie donc les conditions donnent une contradiction.

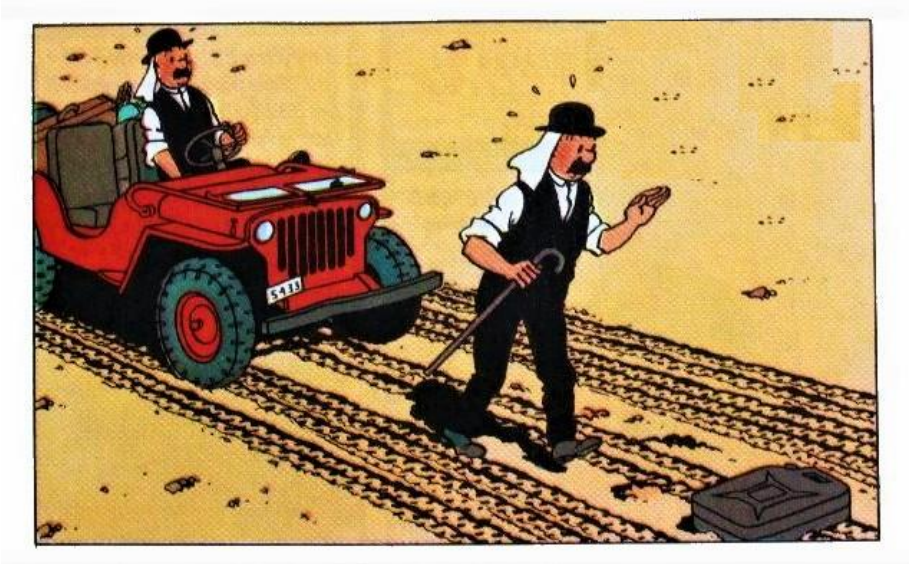
Solution de l'exercice 16 -

```
let p1 = Var "Princesse en 1";;
let p2 = Var "Princesse en 2";;
let p3 = Var "Princesse en 3";;

let e1 = Neg p2;;
let e2 = Neg p2;;
let e3 = Neg p1;;

let o1 = Et(Et (p1, e1), Et (Neg p2, Neg p3));;
let o2 = Et(Et (p2, e2), Et (Neg p3, Neg p1));;
let o3 = Et(Et (p3, e3), Et (Neg p1, Neg p2));;

let test = Neg (Ou (Ou (o1, o2), o3));;

decision test;;

Refutation ["Princesse en 3", false; "Princesse en 2", false;
            "Princesse en 1", true]
```

TP VI

Bidons

Dupont et Dupond se sont perdus dans le désert et leur jeep est en panne.



Il leur reste un bidon de huit litres d'eau, ainsi que deux bidons vides dont les capacités respectives sont de cinq et trois litres.
Dupont pense qu'il faut marcher vers le nord, mais Dupond penche pour le sud.
Ils décident donc de se séparer, mais ils veulent partager l'eau équitablement.
Nous allons les aider !
Associons aux trois bidons un triplet (a, b, c) qui indique le nombre de litres d'eau que chacun contient ; au départ on a $(a, b, c) = (8, 0, 0)$. Une opération élémentaire consiste à verser de l'eau d'un bidon x dans un bidon y jusqu'à ce que y soit plein ou que x soit vide. Par exemple, on peut passer de la situation initiale $(8, 0, 0)$ à la situation $(3, 5, 0)$ ou à la situation $(5, 0, 3)$ selon que l'on décide de remplir le deuxième ou le troisième bidon.
On note `init` la situation initiale $(8, 0, 0)$.

Derrière ce problème de partage se profile donc un graphe : ses sommets sont les valeurs possibles de (a, b, c) compatibles avec la capacité initiale. Ce graphe est orienté : un arc mène de (a, b, c) à (a', b', c') si l'on peut passer de la première situation à la deuxième au moyen d'une opération élémentaire. Nous noterons ceci $(a, b, c) \to (a', b', c')$.

Exercice 1
Combien ce graphe compte-t-il de sommets ?

Exercice 2
Résoudre le problème à la main.

Exercice 3
Montrez que , pour tout état t distinct de `init` *avec 8 litres au total,*
alors soit $t \to$ `init` *soit il existe un état t' tel que $t \to t'$ et $t' \to$* `init`*.*

1 Généralisation

On définit les types

```
type bidon = {maxi : int ; actu : int};;
type etat = bidon array;;
```

Le champ `maxi` permet de traiter le problème avec des bidons de capacités quelconques.

Exercice 4
Écrire une fonction `opElem (x,y)` *qui effectue l'opération élémentaire consistant à déverser le bidon* `x` *dans le bidon* `y`*.*

```
opElem : bidon * bidon -> bidon * bidon
```

Exercice 5
Écrire une fonction `voisins s` *qui construit la liste des états t tels que $s \to t$ avec $s \neq t$.*

```
voisins : etat -> etat list
```

On dit qu'il existe un chemin de s à t s'il existe une suite $(t_k)_{0 \leqslant k \leqslant n}$ de sommets telle que : $t_0 = s$, $t_n = t$ et $t_{k-1} \to t_k$ pour tout tel que $1 \leqslant k \leqslant n$.

Exercice 6
Pour un état e donné, déterminer les états f tels qu'il existe un chemin de e vers f.
On pourra utiliser une liste de sommets vus ; on cherchera un sommet dans cette miste avec la fonction

```
List.mem : 'a -> 'a liste -> bool
```

2 Solutions

Solution de l'exercice 1 -
Un état est déterminé par la capacité des petits bidons, b et c, car alors le gros bidon a une capacité de $8-b-c$. $b \in \{0,1,2,3,4,5\}$ et $c \in \{0,1,2,3\}$ donc il y a 24 états.

Solution de l'exercice 2 -
Voici une solution, l'autre est obtenue en changeant le deuxième état en $(5,0,3)$:
$(8,0,0) \to (3,5,0) \to (3,2,3) \to (6,2,0) \to (6,0,2) \to (1,5,2) \to (1,4,3) \to (4,4,0)$

Solution de l'exercice 3 - On vide les deux petits bidons dans le grand.

Solution de l'exercice 4 -

```
let opElem (x, y) =
  let q = min (y.maxi - y.actu) x.actu in
  ({maxi = x.maxi ; actu = x.actu - q},
   {maxi = y.maxi ; actu = y.actu + q});;
```

Solution de l'exercice 5 -

```
let voisins e =
   let n = Array.length e in
   let a = ref [] in
   for i = 0 to (n-1) do
      for j = 0 to (n-1) do
         if i <> j
         then begin
            let (x, y) = opElem (e.(i), e.(j)) in
            if (x, y) <> (e.(i), e.(j))
            then begin
               let e1 = Array.copy e in
               e1.(i) <- x;
               e1.(j) <- y;
               a := e1 :: !a end end done done;
   !a;;
```

Solution de l'exercice 7 - Il faut modifier la recherche de composante en insérant le chemin.

```
let joli e =
   let n = Array.length e in
   let t = Array.make n 0 in
   for i = 0 to (n-1) do
      t.(i) <- e.(i).actu done;
   t;;

let chemin e f =
   let rec faire vus a_voir =
      match a_voir with
      |[] -> failwith "Il n'y a pas de chemin"
      |(chm, s)::reste when List.mem s vus -> faire vus reste
      |(chm, s)::reste when s = f -> List.rev ((joli s) :: chm
         )
      |(chm, s)::reste -> let chm' = (joli s) :: chm in
                          let plus x = (chm', x) in
```

```
                        let suite = List.map plus (voisins s
                            ) in
                        faire (s::vus) (reste@suite)
    in faire [] [e];;
```

TP VII

Mini Langage

Nous allons définir un langage simple qui permet de travailler avec des fonctions d'une variable entière.
Ce langage sera défini par adjonctions successives d'objets.

1. On commence par les expressions arithmétiques simples, que l'on pourra évaluer.
2. On ajoute alors la notion de variable et d'environnement de ces variables ; la valeur d'une variable sera définie par une expression et les expression pourront utiliser une variable déjà définie.
3. On introduit ensuite les fonctions : elles seront pensées comme pouvant associer une valeur à une valeur donnée en entrée, cette dernière sera le résultat d'une expression. Les expression pourront alors utiliser des fonctions déjà définies.
4. La programmation est introduite par les conditionnelles `si ... alors ... sinon` qui permettent de définir des fonctions plus générales.
5. On finit par la récursivité qui est le moyen de plus simple d'effectuer des répétitions.

On omettra la partie difficile qui consiste à interpréter une commande en texte en un programme compréhensible par l'ordinateur : nous écrirons directement les programmes sous forme d'arbres.
On généralise le type classique des expression sous forme d'arbre binaire :

```
type op = Plus | Moins | Fois;;
type expr = Val of int  |Ope of expr*op*expr
```

en un type plus général dont les composantes seront définies petit à petit :

```
type op = Plus | Moins | Fois;;
type expr = |Val of int
            |Ope of expr*op*expr
            |Var of string           (* partie 2 *)
            |Fn of string*expr       (* partie 3 *)
            |Ite of expr*expr*expr;; (* partie 4 *)
```

Les fonctions dont le paramètre est une expression utiliseront un pattern-matching et traiteront les composantes non définies par une dernière ligne de la forme :

```
  |_ -> failwith "non encore utilisable"
```

1 Opérations simples

On définit (provisoirement) une expression par un arbre binaire :
Dans cette ne expression peut être :

- une valeur entière, une constante, représentée par `Val k`,

- une opération simple entre deux expression représentée par un arbre.

Il permettent de représenter les opérations simples ; par exemple

```
let e1 = Ope(Ope(Val 7,Moins,Val 3),
             Fois,
             Ope(Val 2,Plus,Val 4));;
```

représente $(7-3)*(2+4)$

Exercice 1
Représenter $11-(2*(8-3))$.

Exercice 2
Écrire une fonction `eval` *telle que* `eval t` *détermine la valeur de l'expression* `t`.

```
eval : expr -> int = <fun>
```

2 Variables

On va maintenant incorporer les variables dans nos expressions.
Les variables sont des valeurs "à définir", représentées par une chaîne de caractères.
L'expression $(7-3)*(2+y)$ sera donc représentée par

```
let e3 = Ope(Ope(Val 7, Moins, Val 3),
             Fois,
             Ope(Val 2, Plus, Var "y")
            );;
```

Les valeurs possibles des variables sont définies par une liste d'environnement dont les éléments sont des couples formés d'un nom de variable (`string`) et d'une valeur (`int`).
Un environnement est donc un dictionnaire de type `(string*int) list`. Par exemple,

```
let env0 = [("x",3); ("y",2)];;
```

définit un environnement dans lequel 2 variables sont définies.

Exercice 3
Écrire une fonction `valeur` *telle que* `valeur var env` *renvoie l'entier associé à la* **première apparition** *de la variable* `var` *dans la liste* `env`.

Par exemple `valeur "x" env` doit renvoyer 3.

```
valeur : 'a -> ('a * 'b) list -> 'b = <fun>
```

Pour ajouter une variable à un environnement on ajoute le couple `(nom, valeur)` en tête de l'environnement pour produire un nouvel environnement. On fait de même pour changer la valeur d'une variable : comme la valeur retournée est la première trouvée, on aura ainsi le bon résultat. Cela peut engendrer de nombreuses variables redondantes mais cela permet aussi de créer facilement des valeurs locales aux variables.

Exercice 4
Écrire une fonction `eval2` *telle que* `eval2 t env` *détermine la valeur de l'expression* `t` *dans l'environnement* `env`.

```
eval2 : expr -> (string * int) list -> int = <fun>
```

Par exemple l'évaluation de `e3` avec l'environnement ci-dessus donne 16

3 Fonctions

On se restreint aux fonctions d'une variable.
Une fonction est définie par un triplet `("f", "x", expF)` où `expF` est une expression. La valeur de `f` en `k` est obtenue en donnant à la variable `x` la valeur `k` dans `expF`.
Par exemple, si `c = Ope(Var "x", Fois, Var "x")`, le triplet `("f", "x", c)` représente la fonction carré.
Les fonctions sont gérées par une liste de fonctions. Par exemple

```
let fonc0 = [("f", "x", Ope(Val 2, Fois, Var "x"));
             ("g", "x", Ope(Var "x", Plus, Val 1))];;
```

Un environnement de fonctions est donc un dictionnaire (généralisé) de type
`(string * string * expr) list`.

Exercice 5
Écrire une fonction `defin` *telle que* `defin f fonc` *renvoie le nom de la variable (une chaîne de caractères) et l'expression associés à une fonction de nom* `f` *dans la liste de fonctions.*

```
defin : 'a -> ('a * 'b * 'c) list -> 'b * 'c = <fun>
```

On utilise donc le constructeur `Fn` dans les expression.
Les paramètres sont une chaîne de caractères, le nom de la fonction, et une expression `expV`.
Le résultat de l'expression `expV` sera la valeur envoyée à f.
Par exemple, avec l'environnement de fonctions ci-dessus,
`Fn("f", Ope(Val 7, Moins, Var "x"))` sera évalué en 8.
On notera que x est associé à 3 lors du calcul de `Ope(Val 7, Moins, Var "x"))`
mais qu'il prend la valeur 4 lors de l'évaluation de f

Exercice 6
Écrire une fonction `eval3` *telle que* `eval t env fonc` *détermine la valeur de l'expression* `t` *dans l'environnement* `env` *avec les fonctions de* `fonc`.

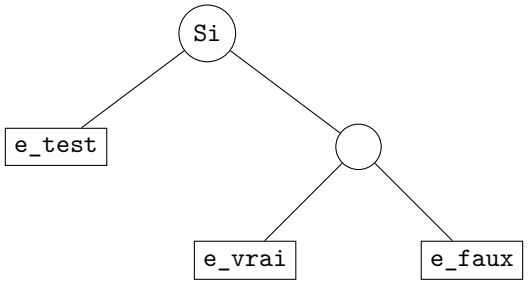
```
eval3 : expr  -> (string * int) list
              -> (string * string * expr) list
              -> int = <fun>
```

Exemple

```
let e4 = Ope(Fn("f", Ope(Val 7, Moins,Val 3)),
               Fois,
               Ope(Val 2, Plus, Var "y"));;

eval3 e4 env0 fonc0;;

- : int = 32
```

4 Condionnelles

On peut définir une instruction de branchement par un arbre



où `e_test`, `e_vrai` et `e_faux` sont des expressions.
Si `e_test` est évaluée à une valeur n avec $n > 0$ alors le résultat est l'évaluation de `e_vrai`, sinon le résultat est l'évaluation de `e_faux`.
On utilise donc le dernier constructeur : `Ite`.
L'arbre ci-dessus sera codé `Ite(e_test, e_vrai, e_faux)`.

Exercice 7
Écrire une fonction `signe x` *sous forme d'un triplet* `("signe", "x", e_signe)` *qui renvoie 1 pour* $x > 0$, -1 *pour* $x < 0$ *et 0 si* x *est nul.*

Exercice 8
Modifier la fonction d'évaluation en `eval4` *pour tenir compte de cet opérateur.*

On peut alors introduire des fonctions récursives, une conditionnelle permettant de gérer le cas d'arrêt.

Exercice 9
Écrire la fonction factorielle.

5 Solutions

Solution de l'exercice 1 -

```
let e2 = Ope (Val 11,
                Moins,
                Ope(Val 2,
                      Fois,
                      Ope(Val 8, Moins, Val 3)
                     )
               );;
```

Solution de l'exercice 2 -

```
let rec eval e =
  match e with
  |Val n -> n
  |Ope(e1, Plus, e2)  ->  eval e1 + eval e2
  |Ope(e1, Moins, e2) ->  eval e1 - eval e2
  |Ope(e1, Fois, e2)  ->  eval e1 * eval e2;;
```

Solution de l'exercice 3 -

```
let rec valeur var env =
  match env with
  |[] -> failwith "Variable non liee"
  |(v, n)::reste when v = var -> n
  |c::reste -> valeur var l;;
```

Solution de l'exercice 4 -

```
let rec eval2 e env =
  match e with
  |Val n -> n
  |Var ch -> valeur ch env
  |Ope(e1, Plus, e2)  ->  eval2 e1 env + eval2 e2 env
  |Ope(e1, Moins, e2) ->  eval2 e1 env - eval2 e2 env
  |Ope(e1, Fois, e2)  ->  eval2 e1 env * eval2 e2 env;;
```

Solution de l'exercice 5 -

```
let rec defin f fonc =
  match fonc with
  |[] -> failwith "Fonction non liee"
  |(fn, var, exp)::_ when fn = f -> (var,exp)
  |_::reste -> defin f reste;;
```

Solution de l'exercice 6 -

```
let rec eval3 t env fonc =
  match t with
  |Val n -> n
  |Var ch -> valeur ch env
  |Ope(t1, Plus, t2)  ->  eval3 t1 env fonc + eval3 t2 env
     fonc
```

```
  |Ope(t1, Moins, t2) ->  eval3 t1 env fonc - eval3 t2 env
     fonc
  |Ope(t1, Fois, t2)  ->  eval3 t1 env fonc * eval3 t2 env
     fonc
  |Fn(f, expV) -> let var, expF = defin f fonc in
                  let env1 = (var, eval expV env fonc)::env in
                  eval3 expF env1 fonc;;
```

Solution de l'exercice 7 -

```
e_signe = Ite(Var "x", Val 1, Ite(Ope(Val 0, Moins, Var "x"),
   Val (-1), Val 0))
```

Solution de l'exercice 8 -

```
let rec eval4 e env fonc =
  match e with
  |Val(n) -> n
  |Var(v) -> valeur v env
  |Ope(e1, Plus, e2)  ->  eval4 e1 env fonc + eval4 e2 env
     fonc
  |Ope(e1, Moins, e2) ->  eval4 e1 env fonc - eval4 e2 env
     fonc
  |Ope(e1, Fois, e2)  ->  eval4 e1 env fonc * eval4 e2 env
     fonc
  |Fn(f, expV) -> let v, expF = defin f fonc in
                  let env1 = (v, eval4 expV env fonc)::env in
                  eval4 expF env1 fonc
  |Ite(eT, eV, eF) -> if eval4 eT env fonc > 0
                      then eval4 eV env fonc
                      else eval4 eF env fonc;;
```

Solution de l'exercice 9 -

```
let e_boucle = Ope(Var "n", Fois, Fn("fact", Ope(Var "n",
   Moins, Val 1)));;

let e_fact = Ite(Var "n", e_boucle, Val 1);;

let fonc1 = ("fact", "n", e_fact)::fonc0;;
```

TP VIII

Algorithme de Tarjan

On cherche à déterminer les composantes fortement connexes d'un graphe **orienté** *: 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

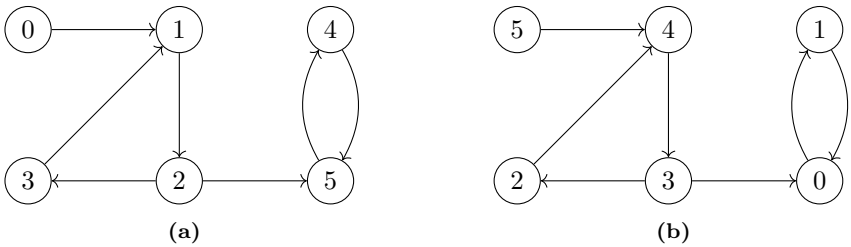


Figure VIII.1 – a) Graphe G_1, b) Graphe G_2

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;;
```

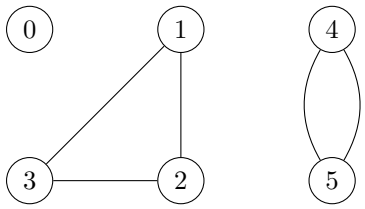


Figure VIII.2 – Graphe non orienté G_0

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 **préfixe**.

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; -1; 4|]
0 → 1 → 2 → 5 → 4	[| 0; 1; 1; 1; 5; 4|]
0 → 1 → 2 → 5 → 4 → 5	[| 0; 1; 1; 1; 5; 4|]
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.

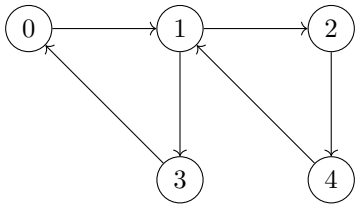


Figure VIII.3 – Graphe G_0

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 ceux 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 IX

Couplages

1 Couplages généraux

$G = (S, A)$ est un graphe non orienté, S est l'ensemble de ses sommets, A l'ensemble de ses arêtes.
On note $\sigma(a)$ l'ensemble des deux sommets reliés par une arête a.
On supposera que G est sans point isolé : $\bigcup_{a \in A} \sigma(a) = S$.

1.1 Recouvrement par les sommets

Un **stable** de G est une partie S' de S ne contenant aucune paire de sommets adjacents :

$$\forall a \in A \quad |\sigma(a) \cap S'| \leqslant 1$$

Le cardinal maximum d'un stable de G est noté $\alpha(G)$.
Un **transversal** est une partie T de S contenant un sommet adjacent à toute arête :

$$\forall a \in A \quad |\sigma(a) \cap T| \geqslant 1$$

Le cardinal minimum d'un transversal de G est noté $\beta(G)$.

Exercice 1 — Une première dualité
Prouver les assertions suivantes.

1. *S' est un stable si et seulement si $S \setminus S'$ est un transversal.*
2. *$\alpha(G) + \beta(G) = |S|$*

1.2 Recouvrement par les arêtes

Un **couplage** de G est une partie C de A ne contenant aucune paire d'arêtes adjacentes :

$$\forall a, a' \in C \quad \sigma(a) \cap \sigma(a') = \emptyset$$

Le cardinal maximum d'un couplage de G est noté $\alpha'(G)$.
Un **recouvrement** est une partie R de A contenant une arête adjacente à tout sommet :

$$\sigma(R) := \bigcup_{a \in R} \sigma(a) = S$$

Le cardinal minimum d'un recouvrement de G est noté $\beta'(G)$.

Exercice 2 — Comparaisons
Prouver qu'on a $2\alpha'(G) \leqslant |S| \leqslant 2\beta'(G)$.

Exercice 3 — Théorème de Gallai

1. *Si C est un couplage de cardinal maximal prouver que $S \setminus \sigma(C)$ est un stable.*
2. *Si R est un recouvrement de cardinal minimal prouver que toute arête de R admet au moins un sommet qui n'est adjacent qu'à une seule arête de R.*
3. *Prouver qu'on a $\alpha'(G) + \beta'(G) = |S|$*

Couplages parfaits

Un couplage est **parfait** s'il est en même temps un recouvrement.

Exercice 4 — Un critère

Prouver que G admet un couplage parfait si et seulement si $2\alpha'(G) = |S| = 2\beta'(G)$.

1.3 Augmentations

G est un graphe muni d'un couplage C.
Un sommet est couplé s'il appartient à $\sigma(a)$ pour une arête $a \in C$.
Une chaîne **alternée** relativement au couplage est une suite finie de sommets distincts $(x_0, x_1, \dots, x_p)$ telle que

- x_0 n'est pas couplé,
- (x_{2k}, x_{2k+1}) est une arête de G n'appartenant pas à C et (x_{2k-1}, x_{2k}) appartient à C pour tout $k \leqslant n \div 2$.

Une chaîne alternée est **augmentante** si, de plus, x_n n'est pas couplé.

Exercice 5 — Augmentation

Prouver que si un couplage C admet une chaîne alternée augmentante $(x_0, x_1, \dots, x_p)$ alors p est impair et C n'est pas de cardinal maximum.

Différence ensembliste

Si X et Y sont deux ensembles leur **différence ensembliste** est

$$X \Delta Y := (X \setminus Y) \cup (Y \setminus X) = (X \cup Y) \setminus (X \cap Y)$$

Si C est un couplage et si $(x_0, x_1, \dots, x_p)$ est une chaîne augmentante pour C, la preuve de l'exercice précédent consiste à prouver que $C \Delta \Gamma$ est toujours un couplage avec $\Gamma = \big\{(x_0, x_1), (x_2, x_2), \dots, (x_{p-1}, x_p)\big\}$.

Exercice 6 — Existence de chaînes augmentantes

Si C et C sont deux couplages de G avec $|C| < |C'|$, prouver que $C \Delta C'$ contient une chaîne augmentante pour C.

Exercice 7 — Théorème de Berge

Prouver qu'un couplage C est de cardinal maximum si et seulement si il n'admet pas de chaîne alternée augmentante.

2 Couplages dans les graphes bipartis

Un graphe non orienté (S, A) est **biparti** si l'ensemble des sommet peut s'écrire sous une forme d'union disjointe $S = U \sqcup V$ telle que toute arête ne peut joindre qu'un sommet de U à un sommet de V : $A \subset U \times V \cup V \times U$.

Un graphe biparti est **équilibré** si $|U| = |V|$.

Comme les chaînes alternées relativement à un couplage C alternent aussi les sommets dans U et V et qu'une chaîne alternée augmentante a un nombre pair de sommets on en déduit que ses extrémités appartiennent l'une à U et l'autre à V. On ne considérera ici que les chaînes alternées augmentantes d'origine dans U et d'extrémité dans V.

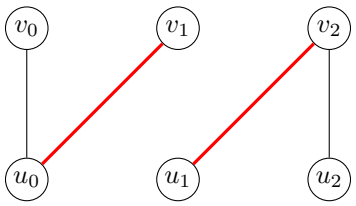
$$x_0 \to x_1 \to x_2 \to \cdots \to x_{p-1} \to x_p$$

- p est impair
- $x_i \in U$ pour i pair, $x_i \in V$ pour i impair,
- $(x_i, x_{i+1}) \notin C$ pour i pair, $(x_i, x_{i+1}) \in C$ pour i impair.

2.1 Dénombrements

C est un couplage d'un graphe biparti $(U \sqcup V, A)$.

- U_0 est l'ensemble (éventuellement vide) des sommets de U qui ne sont pas extrémité d'une arête de C.
- Z est l'ensemble des extrémités finales de chemins alternants commençant par un sommet de U_0.
- $T = \big(U \setminus (U_0 \cup Z)\big) \cup (V \cap Z)$.



Dans cet exemple, $C = \{(u_0, v_1), (u_1, v_2)\}$, $U_0 = \{u_2\}$, $Z = \{u_1, v_2\}$ et $T = \{u_0, v_2\}$.

Exercice 8 — Lemme
Montrer que T est un transversal.

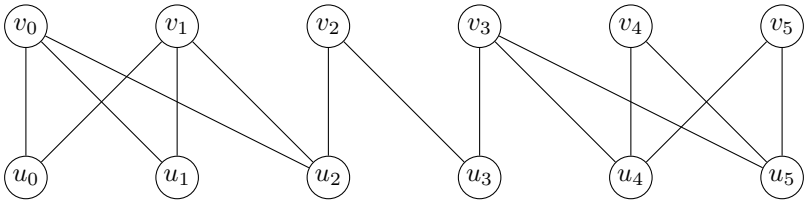
Exercice 9 — Théorème de König
Montrer que T
Montrer que, dans un graphe biparti, $\alpha'(G) = \beta(G)$.

On notera $\Gamma(X)$ l'ensemble des sommets adjacents à $X \subset S$.

Exercice 10 — Théorème de Hall
Montrer qu'un graphe biparti G admet un couplage parfait si et seulement si $|\Gamma(X)| \geqslant |X|$ pour tout $X \subset U$.

2.2 Implémentation (Mines 2012)

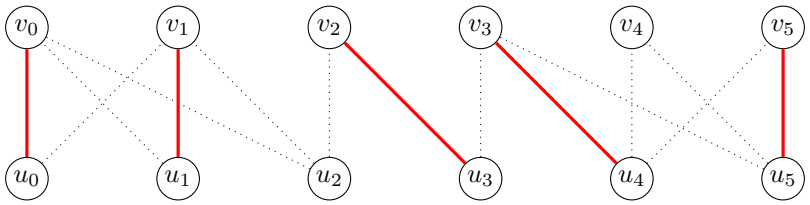
$G = (S, A)$ est un graphe biparti équilibré de cardinal $2n$: $|U| = |V| = n$.
On suppose que les éléments de U et V sont indicées par les entiers de 0 à $n-1$: $U = \{u_0, u_1, \ldots, u_{n-1}\}$ et $V = \{v_0, v_1, \ldots, v_{n-1}\}$.
G est représenté par une matrice de booléens `g` de dimension $n \times n$ dont les lignes correspondent aux éléments de U et les colonnes aux éléments de V. `g.(i).(j)` vaut `true` si et seulement si (u_i, v_j) est une arête.



Le graphe ci-dessus est donc représenté par la matrice :

```
let g = [|[|true;  true;  false; false; false; false|];
          [|true;  true;  false; false; false; false|];
          [|true;  true;  true;  false; false; false|];
          [|false; false; true;  true;  false; false|];
          [|false; false; false; true;  true;  true |];
          [|false; false; false; true;  true;  true |]|];;
```

Un couplage C est codé par un tableau d'entiers de taille n, `c` ; `c.(i)` vaut `j` si et seulement si (u_i, v_j) est une arête du couplage ou, si u_i n'est pas couplé, `c.(i)` vaut `-1`.
`[|0; 1; -1; 2; 3; 5|]` code le couplage représenté ci-dessous.



Exercice 11 —
Écrire une fonction `verifie` *telle que si* `g` *est une matrice codant le graphe G et* `c` *un tableau codant le couplage C alors* `verifie g c` *renvoie* `true` *si le tableau C représente un couplage dans G et* `false` *sinon.*

Exercice 12 —
Écrire une fonction `inverse c` *telle que si* `c` *est un tableau codant un couplage C alors* `inverse c` *renvoie tableau* `inv` *codant C depuis V :* `inv.(j)` *vaut* `i` *si (i, j) est une arête du couplage et vaut* `-1` *s'il n'en existe pas.*

2.3 Recherche exhaustive (Mines 2012)

Exercice 13 —
Écrire une fonction `une_arete g` *qui renvoie une arête quelconque du graphe représenté par* `g` *sous la forme* `Some (u, v)` *s'il existe au moins une arête (u, v), sinon la fonction renvoie* `None`.

On cherche un algorithme qui permette de déterminer un couplage de cardinal maximal dans un graphe biparti équilibré ; le principe est le suivant.
Si le graphe courant ne contient aucune arête, le cardinal maximum d'un couplage est 0 et aucun sommet n'est couplé.
Dans le cas contraire, l'algorithme considère une arête quelconque a du graphe courant et recherche successivement :

- un couplage de cardinal maximal parmi les couplages du graphe courant ne contenant pas a ;
- un couplage de cardinal maximal parmi les couplages du graphes contenant a.

L'algorithme déduit alors un couplage de cardinal maximal.

Exercice 14 —
Écrire une fonction `copie_matrice` *qui copie une matrice.*
On pourra utiliser la fonction `Array.copy` *qui copie un tableau mais ne permet pas de copier une matrice en assurant que les valeurs de chaque ligne soient indépendantes de la matrice originale.*

Exercice 15 —
Écrire une fonction `meilleur_couplage g` *qui renvoie un tableau codant un couplage de cardinal maximal dans G.*

2.4 Algorithme Hongrois (Mines 2012)

L'**algorithme hongrois** s'appuie sur le théorème de Berge et détermine un couplage de cardinal maximal dans G. L'algorithme débute avec un couplage C de cardinal nul ; tant qu'il existe une chaîne augmentante relativement à C, l'algorithme modifie C pour incrémenter de 1 le cardinal du couplage en utilisant une telle chaîne augmentante.
Pour trouver une chaîne alternée augmentante, on recherche les sommets qui sont extrémités de chaînes alternées en procédant de proche en proche à partir des sommets de A non couplés.
Un sommet y est dit **atteint** si une chaîne alternée relativement à C et d'extrémité y est mise en évidence.
Au départ tous les sommets non couplés de U (un tel sommet est extrémité d'une chaîne alternée réduite à ce sommet) sont considérés comme atteints ; aucun autre sommet n'est considéré comme atteint. Un sommet y non encore atteint peut être atteint à partir d'un de ses voisins x **déjà atteint** si on a :

- soit y est dans V (et donc x est dans A) et l'arête (x, y) n'est pas dans le couplage C ;
- soit y est dans U (et donc x est dans V) et l'arête (y, x) est dans le couplage C.

On utilise des marques attribuées aux sommets. Ces marques sont des entiers initialisés à -1 pour tous les sommets. Lorsqu'un sommet y est atteint à partir d'un sommet x la marque de y devient égale au numéro de x (on rappelle que le numéro d'un sommet u_i ou v_i vaut i) ; x est alors l'avant-dernier sommet dans la chaîne alternée $\mathrm{Ch}(y)$ d'extrémité y mise en évidence. La chaîne $\mathrm{Ch}(y)$ peut être retrouvée à l'envers, de proche en proche, grâce aux marques ; dans cette chaîne, seule l'origine porte une marque de valeur -1. Si un sommet non couplé y de V est atteint, $\mathrm{Ch}(y)$ est une chaîne alternée augmentante relativement à C.
On pourra utiliser définir deux fonctions `chercherU` et `chercherV` utilisant une récursivité croisée (chacune pouvant appeler l'autre) telles que telle que `chercherX s` détermine des sommets non encore atteints qui peuvent être atteints, de voisin en voisin, récursivement, à partir de $s \in X$ en modifiant les marques de ces sommets nouvellement atteints et s'arrête dans les cas suivants :

- un sommet $v_i \in V$ non couplé a été atteint ; elle renvoie alors i
- tous les sommets qui peuvent être atteints de voisin en voisin à partir de s l'ont été et aucun sommet non couplé de V n'est atteint ; elle renvoie alors -1.

Exercice 16 —
Écrire une fonction `chaine_augmentante g c` *qui renvoie un triplet formé de deux tableaux codant les marques d'une chaîne alternée augmentante du graphe G pour le couplage C et le numéro de l'extrémité (dans V) de ce couplage.*
S'il n'existe pas de chaîne alternée augmentante le numéro sera `-1`.

Exercice 17 —
Écrire une fonction `actualiser c mU mV fin` *qui modifie le couplage codé par c en utilisant la chaîne alternée augmentante codée par les tableaux* `mU` *et* `mV` *et le sommet final* `fin`.

Exercice 18 —
Écrire une fonction `algorithme_hongrois g` *qui renvoie un tableau codant un couplage de cardinal maximal dans G en utilisant cette méthode.*

3 Solutions

Solution de l'exercice 1 - On note $\overline{S'} = S \setminus S'$ et $n = |S|$.

1. $\big|\sigma(a) \cap S'\big| + \big|\sigma(a) \cap \overline{S'}\big|\big|\sigma(a) \cap S\big| = \big|\sigma(a)\big| = 2$ donc
   $\big[\forall a \in A\ \big|\sigma(a) \cap S'\big| \leqslant 1\big] \iff \big[\forall a \in A\big|\sigma(a) \cap \overline{S'}\big| \geqslant 1\big]$
2. Si S' est un stable de cardinal maximal, $\alpha(G)$, alors $\overline{S'}$ est un recouvrement donc $\beta(G) \leqslant |\overline{S'}| = n - |S'| = n - \alpha(G)$: $\alpha(G) + \beta(G) \leqslant n$.
   Si T est un recouvrement de cardinal minimal, $\beta(G)$, alors $\overline{T}$ est un stable donc $\alpha(G) \geqslant |\overline{T}| = n - |T| = n - \beta(G)$: $\alpha(G) + \beta(G) \geqslant n$.
   On a bien $\alpha(G) + \beta(G) \geqslant n$.

Solution de l'exercice 2 -
Pour un couplage C les extrémités des arêtes sont distinctes donc $2|C| \leqslant n$.
Pour un recouvrement R tout sommet est l'une des deux extrémités des arêtes de R donc $n \leqslant |R|$.

Solution de l'exercice 3 -

1. Soit C un couplage. Si $\overline{\sigma(C)}$ n'est pas un stable, il existe une arête a dont les deux extrémités sont dans $\overline{\sigma(C)}$ d'où $C \cup \{a\}$ est encore un couplage : C n'est pas de cardinal maximal d'où le résultat par contraposition.
   On a démontré en fait que si C est un couplage maximal pour l'inclusion (ce qui est plus général que la maximalité du cardinal
2. Soit R un recouvrement. S'il existe $a \in R$ dont les deux sommets sont adjacents chacun à une autre arête de R alors $R \setminus \{a\}$ est encore un recouvrement donc R n'est pas de cardinal minimal. Le résultat s'en déduit par contraposition.
3. Soit C un couplage de cardinal maximal, les sommet du stable $S' = \overline{\sigma(C)}$ ne sont pas reliées par des arêtes mais ont chacun une arête adjacente car le graphe n'a pas de sommet isolé.
   Si on ajoute ces arêtes à C on obtient un recouvrement R avec $|R| = |C| + |S'|$.
   Or les arêtes de C sont non adjacentes donc $|\sigma(C)| = 2|C|$ puis $|S'| = n - 2|\sigma(C)| = 2 - 2|C|$; ainsi $|\beta'(G) \leqslant |R| = n - |C| = n - \alpha'(G)$. On a $\alpha'(G) + \beta'(G) \leqslant |S|$.
   Soit R un recouvrement de cardinal minimal. Pour chaque arête de R on choisit un sommet qui n'est adjacent qu'à cette arête. Il reste $n - |C|$ sommets. On obtient un couplage en choisissant une arête de R (il peut y en avoir plusieurs) pour chacun de ces sommets restants.
   On a ainsi $\alpha'(G) \geqslant n - |C| = n - \beta'(G)$ d'où $\alpha'(G) + \beta'(G) \geqslant |S|$ puis l'égalité.

Solution de l'exercice 4 -
Si G admet un couplage parfait C alors $|C| \leqslant \alpha'(G)$ car C est un couplage et $|C| \geqslant \beta'(G)$ car C est un recouvrement. On a donc $2\alpha'(G) \geqslant 2\beta'(G)$, comme l'inégalité inverse est toujours vraie, on en d"duit $\alpha'(G) = \beta'(G)$. Comme la somme est égale à n la valeur commune est $\frac{n}{2}$.
Inversement si $\alpha'(G) = \frac{n}{2}$ alors, pour un couplage C de cardinal maximal, les sommets couverts forment un ensemble de cardinal $2|C| = n$ donc C est aussi un recouvrement, c'est un couplage parfait.

Solution de l'exercice 5 -
Si p était pair, $p \geqslant 2$ alors p serait une extrémité de l'arête $(x_{p-1}, x_p) \in C$. On pose $p = 2q + 1$.
On remplace les q arêtes (x_1, x_2), (x_3, x_4), …, (x_{2q-1}, x_{2q}) par les $q + 1$ arêtes (x_0, x_1), (x_2, x_3), …, (x_{2q}, x_{2q+1}) dans C. Le résultat est toujours un couplage, dont le cardinal est supérieur de 1 à celui de C.

Solution de l'exercice 6 - Dans C et dans C' les arêtes n'ont pas de sommet en commun ; ainsi, dans $C \Delta C'$ un sommet ne peut être l'extrémité que de deux arêtes au plus (chaque sommet est de degré 2 au plus) et, dans le cas où deux arêtes ont un sommet commun, l'une appartient à C et l'autre à C'.
On en déduit que les composantes connexes de $C \Delta C'$ sont

- soit des chemins simples alternant les arêtes de C et les arêtes de C',
- soit des cycles de longueur paire.

Les composantes connexes contiennent donc
- soit autant d'arêtes de C et d'arêtes de C', c'est le cas des cycles car ils sont d'ordre pair,
- soit une arête de C de plus que le nombre d'arêtes de C', c'est donc un chemin qui commence et finit par une arête de C : une chaîne alternée augmentante pour C',
- soit une arête de C' de plus que le nombre d'arêtes de C, c'est donc un chemin qui commence et finit alors par une arête de C' : une chaîne alternée augmentante pour C.

Si on note $n = |C|$, $n' = |C'|$ et $d = |C \cap C'|$, on a $n < n'$ et le nombre d'arêtes de C (resp. de C') dans $C\Delta C'$ est $n - d$ (resp. $n' - d$).
Il y a donc plus d'arêtes de C' que d'arêtes de C' dans $C\Delta C'$; d'après le décompte ci-dessus, on en déduit qu'il existe au moins $n' - n$ chaînes alternées augmentantes pour C dans $C\Delta C'$.

Solution de l'exercice 7 - On a déjà vu la contraposée du sens direct : si un couplage admet une chaîne alternée augmentante, il n'est pas de cardinal maximal.
Inversement, si C est un couplage non maximal, il existe un couplage C' tel que $|C'| > |C|$ et la question précédente montre qu'il existe une chaîne alternée augmentante pour C dans $C\Delta C'$.

Solution de l'exercice 8 - Si s appartient à Z alors il y a un chemin de x_0 à s vérifiant

$$x_0 \to x_1 \to x_2 \to \cdots \to x_{p-1} \to x_p = s$$

- $x_0 \in U_0$
- $x_i \in U$ pour i pair, $x_i \in V$ pour i impair,
- $(x_i, x_{i+1}) \notin C$ pour i pair, $(x_i, x_{i+1}) \in C$ pour i impair.

Si $a = (u, v)$ est une arête du graphe on a
- soit $u \in U_0$ donc (u, v) n'est pas une arête du couplage et $v \in Z \cap V \subset T$,
- soit $u \in Z \cap U$ donc $u = x_p$ avec p pair et $(x_{p-1}, u) \in C$; on en déduit qu'on a
  $v = x_{p-1}$ donc $v \in Z \cap V \subset T$
  ou $v \neq x_{p-1}$ donc $(u, v) \notin C$ et $v \in Z \cap V \subset T$
- Soit $u \in U \setminus (U_0 \cup Z) \subset T$.

Dans tous les cas une extrémité de l'arête est dans T : T est un transversal.

Solution de l'exercice 9 - Si T est un transversal, chaque arête d'un couplage a (au moins) une extrémité dans T et les extrémités des arêtes du couplage sont distinctes donc on peut définir une injection de puis l'ensemble des arêtes d'un couplage vers les sommet d'un transversal (en choisissant éventuellement une des deux extrémités). On a donc $|C| \leqslant |T|$ d'où $\alpha'(G) \leqslant \beta(G)$.

Pour un couplage maximal C, le lemme a construit un transversal T.
Tout sommet $u \in T \cap U$ est l'extrémité d'une arête de C car $n \notin U_0$.
Tout sommet $v \in T \cap V$ est l'extrémité d'un chemin alternant commençant par un sommet de U_0. S'il n'existait pas d'arête de C passant par v, le chemin alternant serait augmentant, ce qui n'est pas possible car C est maximal. v est l'extrémité d'une arête de C
Comme il ne peut exister qu'une arête de C passant par un sommet le raisonnement ci-dessus définit une application de T vers C.
Si deux sommets distincts étaient associés à une même arête de C on aurait alors $(u, v) \in C$ avec $u \in T$ et $v \in T$. Comme $v \in T \cap V = Z \cap V$, il serait l'extrémité d'un chemin alternant commençant par un sommet de U_0 donc u appartiendrait aussi à Z, ce qui est impossible car $u \in T \cap U = U \setminus (U_0 \cup Z)$.
On a donc une injection de T dans C d'où $|T| \leqslant |C|$ puis $\beta(G) \leqslant \alpha'(G)$.
On en déduit l'égalité souhaitée.

Solution de l'exercice 10 - $G = (U \sqcup V, A)$ avec $|U| = |V| = n$.
Si G admet un couplage de cardinal n, celui ci définit une bijection φ de U vers V donc, pour tout $X \subset U$, $\Gamma(X) \supset \varphi(X)$ et $|\Gamma(X)| \geqslant |\varphi(X)| = |X|$.

On suppose que les inégalités sont vérifiées.
Soit p le cardinal maximal d'un couplage. On a $p \leqslant n$.
D'après le théorème de König, p est le cardinal (minimal) d'un transversal T. On note $T_1 = T \cap U$ et $T_2 = T \cap V$.

Comme T est un transversal, toute arête du graphe a une extrémité dans T_1 ou dans T_2 donc il n'y a aucune arête allant de $U \setminus T_1$ vers $V \setminus T_2$; on a donc $\Gamma(U \setminus T_1) \subset T_2$ et $\Gamma(V \setminus T_2) \subset T_1$. Les inégalités donnent donc
$|V| - |T_2| = |V \setminus T_2| \leqslant |\Gamma(V \setminus T_2)| \leqslant |T_1|$ d'où $n \geqslant |T_1| + |T_2| = p$.
Ainsi $p = n$ et le couplage maximal est parfait.

Solution de l'exercice 11 - Pour chaque nouveau couplage rencontré il faut vérifier si l'arête correspondante existe dans le graphe et si elle n'est pas incidente à une arête déjà rencontrée. Pour cela on utilise un tableau `vu` qui indique si un sommet de B a déjà été rencontré.

```
let verifie g c =
   let n = Array.length c in
   let reponse = ref true in
   let vu = Array.make n false in
   for i = 0 to (n-1) do
      let j = c.(i) in
      if j <> -1 && not vu.(j)
      then begin reponse := !reponse && g.(i).(j);
           vu.(j) <- true end done;
  !reponse;;
```

La complexité est linéaire en n, le nombre de sommets.

Solution de l'exercice 12 -

```
let inverse c =
   let n = Array.length c in
   let r = Array.make n (-1) in
   for i = 0 to (n-1) do
      if c.(i) <> -1
      then r.(c.(i)) <- i done;
   r;;
```

Solution de l'exercice 13 -

```
let une_arete g =
   let n = Array.length g in
   let out = ref None in
   for u = 0 to (n-1) do
      for v = 0 to (n-1) do
         if g.(u).(v) then out := Some (u, v) done done;
   !out;;
```

Solution de l'exercice 14 -

```
let copie_matrice m =
   let n = Array.length m in
   let cop = Array.make n [||] in
   for i = 0 to (n-1) do
      cop.(i) <- Array.copy m.(i) done;
   cop;;
```

Solution de l'exercice 15 - Pour ne pas avoir à compter la taille du couplage, on renvoie celle-ci dans une fonction auxiliaire en plus du couplage. Pour déterminer un couplage auquel on pourra ajouter une arête $a = (u, v)$, on part du graphe qui interdit toute arête depuis u ou vers v.

```
let rec meilleur_couplage g =
   let n = Array.length g in
   let rec aux_mc g =
     match une_arete g with
     |None -> Array.make n (-1), 0
     |Some (u, v) ->  let g1 = copie_matrice g in
                      g1.(u).(v) <- false;
                      let g2 = copie_matrice g in
                      for k = 0 to (n-1) do
                         g2.(u).(k) <- false;
                         g2.(k).(v) <- false done;
                      let c1, n1 = aux_mc g1 in
                      let c2, n2 = aux_mc g2 in
                      c2.(u) <- v;
                      if n1 > n2 then c1, n1 else c2, (n2+1)
   in aux_mc g;;
```

Les complexités temporelles et spatiales sont monstrueuses : de l'ordre de $n^2.2^{n^2}$.

Solution de l'exercice 16 - On parcourt dans l'ordre les sommets de A jusqu'à trouver un éventuel sommet non couplé origine d'une chaîne alternée augmentante.

```
let chaine_augmentante g c =
   let n = Array.length c in
   let r = inverse c in
   let fin  = ref (-1) in
   let mU = Array.make n (-1) in
   let mV = Array.make n (-1) in
   let rec chercherU u =
   (* on cherche les sommets atteignables non vus et non coupl
      és à u *)
      let v = ref 0 in
      while !v < n && !fin = -1 do
         if g.(u).(!v) && c.(u) <> !v && mV.(u) = -1
         then begin mV.(!v) <- u;
              chercherV !v;
              if !fin = -1 then mV.(!v) <- -1 end;
         v := !v + 1 done;
   and chercherV v =
      match r.(v) with
      |(-1) -> fin := v
      |u -> mU.(u) <- v;
            chercherU u in
   let u = ref 0 in
   while !u < n && !fin = -1 do
      for k = 0 to (n-1) do mU.(k) <- -1; mV.(k) <- -1 done;
      chercherU !u;
      u := !u + 1 done;
   mU, mV, !fin;;
```

Solution de l'exercice 17 -
On remonte la chaîne augmentante grâce aux marques en partant de `fin` ;
pour un sommet v de V, on lit `u = mB.(v)` et on crée l'arête (u, v) dans le couplage
puis on recommence à partir de `mU.(u)` s'il est distinct de -1.

```
let actualiser c mU mV fin =
   let rec actu v =
       let u = mV.(v) in
       c.(u) <- v;
       let v1 = mU.(u) in
       if v1 <> -1 then actu v1 in
   actu fin;;
```

Solution de l'exercice 18 -

```
let algorithme_hongrois g =
   let n = Array.length g in
   let c = Array.make n (-1) in
   let rec aux_hon () =
      let mU, mV, fin  = chaine_augmentante g c in
      if fin = -1
      then c
      else begin actualiser c mU mV fin;
                 aux_hon () end in
   aux_hon ();;
```