Arbres binaires

1) Une feuille est un nœud dont les deux fils sont vides. Écrire une fonction qui compte le nombre de feuilles d'un arbre : `nb_feuilles : 'a arbre -> int`

```
let rec nombre_feuilles a = match a with
  | Vide -> 0
  | Noeud (Vide,_,Vide) -> 1
  | Noeud (g,_,d) -> (nombre_feuilles g) + (nombre_feuilles d)
;;
```

La fonction `nombre_feuilles` a un complexité $\mathcal{O}(n)$ où n est la taille de l'arbre car la fonction va parcourir la totalité de l'arbre.

2) Rappeler la définition de la profondeur (ou hauteur) d'un nœud, ainsi que la profondeur d'un arbre binaire. Écrire les fonctions : `profondeur : 'a arbre -> int` et `taille : 'a arbre -> int`.

Réponse : La profondeur d'un nœud peut être définie par récurrence : la profondeur de la racine est 0, la profondeur des fils du nœud n est $1 + profondeur(n)$.
La profondeur d'un arbre est le maximum des profondeurs de ses nœuds non vides.
La taille d'un arbre est le nombre de nœuds non vides.
On rappelle qu'il peut arriver que la profondeur de la racine soit définie comme valant -1 ou même 1 ; lisez soigneusement l'énoncé.

```
let rec profondeur a = match a with
  | Vide -> -1
  | Noeud (g,_,d) -> 1 + max (profondeur g) (profondeur d)
;;

let rec taille a = match a with
  | Vide -> 0
  | Noeud (g,_,d) -> 1 + (taille g) + (taille d)
;;
```

Les deux fonctions ont une complexité $\mathcal{O}(n)$ où n est la taille de l'arbre car elles doivent parcourir la totalité de l'arbre.

3) Quelle relation d'égalité y a–t–il entre le nombre de nœuds vides d'un arbre binaire et le nombre de nœuds non vides ?

Réponse : $nb\_noeuds\_vides = nb\_noeuds\_non\_vides + 1$. Cela se démontre par induction structurelle ; pour a un arbre on note $nv(a)$ le nombre de noeuds vides et $n(a)$ le nombre de noeuds non vides :

- *Si a est l'arbre vide, il a un noeud vide et pas de noeuds non vides.*
- *Si a est de la forme $N(g, x, d)$, on a $n(a) = 1 + n(g) + n(d)$ et $nv(a) = nv(g) + nv(d)$. Or, par induction structurelle, $nv(g) = n(g) + 1$ et $nv(d) = n(d) + 1$.Finalement,*

$$nv(a) = nv(g) + nv(d) = n(g) + n(d) + 2 = n(a) + 1$$

Quelle relation d'inégalité permet de majorer la taille en fonction de la profondeur ?

Réponse : Soit a un arbre binaire, on note $t(a)$ sa taille et $p(a)$ sa profondeur. On sait qu'il y a, au plus, 1 noeud de profondeur 0 (la racine), 2 noeuds de profondeur 1 et de manière générale 2^k noeuds de profondeur k. On en déduit que

$$t(a) \leq \sum_{k=0}^{p(a)} 2^k = 2^{p(a)+1} - 1$$

4) Rappeler la définition d'un tas–max.

Réponse : Un tas–max est un arbre binaire étiqueté tel que l'étiquette de chaque nœud est supérieure ou égale à l'étiquette de chacun de ses fils, et quasi–complet c'est–à–dire que si la profondeur est p alors tous les niveaux de profondeur strictement inférieur à p sont complets ; et le dernier niveau se remplit de gauche à droite.

Écrire en pseudo–code l'algorithme de suppression du maximum d'un tas–max (le tas étant implémenté dans un arbre binaire).

Réponse :
Retirer la racine.
Remonter le dernier nœud à la racine (le dernier nœud est le nœud le plus à droite du nœud de profondeur maximum).
nœud courant $\leftarrow$ *racine*
Tant que *le nœud courant a un fils avec une étiquette strictement supérieure à la sienne*
Faire
  Échanger le nœud courant avec son fils d'étiquette maximum
Fin tant que

Quelle est la complexité de cet algorithme, en fonction du nombre d'éléments présents dans le tas ?

Réponse : De l'ordre de la profondeur du tas, c'est–à–dire $\mathcal{O}(\log n)$

5) Écrire en Caml une fonction `ajoute : int array -> int -> int -> unit` qui permet d'ajouter un élément dans un tas–max implémenté dans un tableau.

Le premier paramètre est le tableau qui implémente le tas, le deuxième est l'élément à insérer et le troisième est le nombre d'éléments insérés jusqu'à présent dans le tas.

Réponse :

```
let ajoute t cle =
  t.(0) <- t.(0) + 1;
  let pere i = i/2 in
  t.(t.(0)) <- cle;
  let noeud = ref n in
  while !noeud>1 && t.(pere !noeud) < t.(!noeud) do
    let valeur = t.(!noeud) in
    t.(!noeud) <- t.(pere !noeud);
    t.(pere !noeud) <- valeur;
    noeud := pere !noeud
  done
;;
```

Quelle est la complexité de votre fonction, en fonction du nombre d'éléments présents dans le tas ?

Réponse : De l'ordre de la profondeur du tas, c'est–à–dire $\mathcal{O}(\log n)$

6) Rappeler la définition d'un arbre binaire de recherche.

Réponse : C'est un arbre binaire étiqueté tel que pour le tout nœud n*, pour tout nœud* g *descendant de* n *par son fils gauche on a* $etiquette(g) \leqslant etiquette(n)$ *et pour tout nœud* d *descendant de* n *par son fils droit on a* $etiquette(n) \leqslant etiquette(d)$*. Attention : la condition sur les étiquettes stipule que TOUS les noeuds du sous-arbre de gauche sont inférieur à l'étiquette de* n *et pas juste le fils gauche ; de même à droite.*

7) Écrire une fonction de type `int -> int arbre -> bool` qui permet de tester si un élément appartient ou non à un arbre binaire de recherche.

Réponse :

```
let rec recherche cle a = match a with
  | Vide -> false
  | Noeud (g,etiquette,d) ->
      if cle = etiquette then true
      else if cle < etiquette then recherche cle g
      else recherche cle d
;;
```

La complexité est majorée par la profondeur de l'arbre (et donc par le logarithme de la taille de l'arbre si ce dernier est "équilibré".)

8) Rappeler le principe de l'algorithme d'insertion d'un élément dans un arbre binaire de recherche. Écrire la fonction `insert : int -> int arbre -> int arbre`.

Réponse : On part de la racine et on descend en comparant l'élément à insérer avec l'étiquette du nœud courant. L'insertion se fait lorsqu'on arrive à un nœud vide.

```
let rec insertion cle a = match a with
  | Vide -> Noeud (Vide,cle,Vide)
  | Noeud (g,etiquette,d) ->
      if cle < etiquette then Noeud (insertion cle g,etiquette,d)
      else Noeud (g,etiquette,insertion cle d)
;;
```

9) Rappeler l'algorithme de suppression de la racine dans un arbre binaire de recherche. Rappeler le principe de suppression d'un élément dans un arbre binaire de recherche. S'il reste du temps, écrire ces fonctions.

Réponse : Suppression de la racine r : si son fils gauche est vide, on remonte le fils droit à la racine. Si le fils gauche g de r est non vide, on cherche dans g l'élément le plus à droite (d'étiquette maximum) s. Notons alors s_g le fils gauche de s (le fils droit de s est vide), on remonte s à la racine et dans l'arbre on remplace s par s_g.
Pour supprimer un élément dans un arbre binaire de recherche, on descend le long de l'arbre jusqu'à trouver cet élément, et à ce niveau on supprime la racine.

```
let suppression_racine a = match a with
  | Vide -> failwith ("Suppression impossible")
  | Noeud (Vide,_,d) -> d
  | Noeud (g,_,d) ->
      begin
        let rec recMax aa = match aa with
        (* renvoie le couple  (arbre modifie, valeur max) *)
          | Vide -> failwith ("Normalement pas possible...")
          | Noeud (g,etiquette,Vide) -> (g, etiquette)
          | Noeud (g,etiquette,d) -> let (ars,ys) = recMax d in
                                       (Noeud (g,etiquette,ars), ys)
        in
        let (gm, yg) = recMax g in Noeud (gm,yg,d)
      end
;;

let rec suppression cle a = match a with
  | Vide -> failwith ("Suppression impossible")
  | Noeud (g,etiquette,d)  ->
      if cle = etiquette then suppression_racine a
      else if cle < etiquette then Noeud (suppression cle g,etiquette,d)
      else Noeud (g,etiquette,suppression cle d)
;;
```

Logique

10) Le type suivant permet de coder des formules logiques :

```
type formule = Variable of char
             | Constante of bool
             | Non of formule
             | Ou of formule * formule
             | Et of formule * formule
```

Pour les instances de type `formule`, expliquer la différence entre les notions d'égalité suivantes :

- égalité syntaxique ;

  Réponse : Deux formules sont syntaxiquement égales si les arbres syntaxiques sont les mêmes, quitte à dupliquer les nœuds. On rappelle qu'en Caml le test `=` *est un test d'égalité syntaxique (récursif).*

- égalité sémantique ;

  Réponse : Deux formules sont sémantiquement égales si elles ont la même évaluation dans tous les contextes.

11) Qu'est–ce qu'une tautologie ? Une antilogie ? Une formule satisfiable ?

Réponse : Une tautologie est une formule logiquement équivalente à `true`*, c'est–à–dire qu'elle s'évalue toujours en* `true`*.*
Une antilogie est une formule logiquement équivalente à `false`*.*
Une formule est satisfiable si ce n'est pas une antilogie, c'est–à–dire s'il existe un contexte dans lequel elle s'évaluera en `true`*.*

Graphes

12) Définir les notions suivantes : degré d'un sommet dans un graphe orienté ;

Réponse : Le degré entrant d'un sommet s est le nombre de sommets t tels que (t, s) soit un arc.
Le degré sortant d'un sommet s est le nombre de sommets t tels que (s, t) soit un arc.
Le degré est la somme du degré entrant et du degré sortant.

Composante connexe dans un graphe non orienté.

Réponse : Une composant connexe est une classe d'équivalence par la relation entre les sommets : "deux sommets s et t sont en relation si et seulement s'il existe un chemin de s à t".

13) Programmer en Caml une fonction permettant grâce à un parcours en profondeur de connaître les sommets qui sont dans la même composante connexe qu'un sommet donné, dans un graphe non orienté codé par listes d'adjacence.

Réponse : Voici une solution qui n'utilise pas de pile mais qui utilise la structure de l'arbre des appels récursifs (pour une version avec une pile, il suffit de prendre la solution de la question suivante sur le parcours en largeur, en remplaçant la file par une pile). On utilise un tableau auxiliaire de booléen `visit` *permettant de mémoriser les états déjà explorés ; ainsi qu'une liste* `composante` *qui mémorise les sommets de la composante déjà rencontrés. La fonction auxiliaire récursive* `parcours : int -> unit` *effectue le parcours en profondeur.*

```
let profondeur graphe sommet =
  let n = Array.length graphe in
  let visit = Array.make n false in
  visit.(sommet) <- true;
  let composante = ref [sommet] in
  let rec parcours l = match l with
    | [] -> ()
    | s::queue -> if not (visit.(s))
                  then begin
                      visite.(s) <- true;
                      composante := s::(!composante);
                      parcours graphe.(s)
                    end;
                  parcours queue
  in parcours graphe.(sommet);
  !composante
;;
```

Quelle est la complexité de votre fonction ?

Réponse : Ici, chaque arc est emprunté au plus une fois. De plus la création du tableau `visit` *a une complexité de l'ordre du nombre de sommets.*
Au total la complexité est $\mathcal{O}(|V| + |E|)$.
Dans le programme ci-dessus, le tableau ***`visit`*** *et la référence de listes* ***`composante`*** *contiennent, sous une forme différente les mêmes données. En fonction de la taille des données, il peut être judicieux de se passer de l'un ou l'autre. Par exemple, si la taille de la composante est très petite par rapport au nombre de sommets, il serait intéressant de ne pas créer le tableau de booléens* ***`visit`*** *pour savoir si un sommet est déjà exploré mais simplement de tester s'il est déjà dans* `composante`.

14) Même question mais avec un parcours en largeur. On pourra supposer qu'on dispose d'une strucure de file dans laquelle les opérations élémentaires se font en temps constant (création

d'une file vide, tester si la file est vide, enfiler un nouvel élément, défiler l'élément le plus ancien).

Réponse : On suppose disposer d'un type `file` *et des fonctions suivantes :*
`defile : file -> int` *qui défile l'élément le plus ancien ;*
`enfile : file -> int -> unit` *qui enfile un élément ;*
`est_vide : file -> bool` *qui teste si une file est vide ;*
`nouvelle_file : unit -> file` *qui crée une nouvelle file (vide).*
On crée une fonction auxiliaire (externe) `enfile_liste -> file -> int list -> unit` *qui enfile plusieurs éléments passés en paralètres dans une liste.*

```
let rec enfile_liste f l = match l with
  | [] -> ()
  | x::q -> enfile f x; enfile_liste f q
;;

let largeur graphe sommet =
  let n = Array.length graphe in
  let visit = Array.make n false in
  let frontiere = nouvelle_file () in
  let composante = ref [] in
  enfile frontiere sommet;
  while not (est_vide frontiere) do
    let s = defile frontiere in
    if not (visit.(s)) then begin
                            composante := s::(!composante);
                            visit.(s) <- true;
                            enfile_liste frontiere graphe.(s) end
  done;
  !composante
;;
```

15) Programmer en pseudo–langage l'algorithme de Floyd–Warshall dans un graphe orienté pondéré codé par matrice d'adjacence.

Réponse : en notant n le nombre de sommets,

Pour $k = 0$ **à** $n - 1$ **Faire**
  Pour $i = 0$ **à** $n - 1$ **Faire**
    Pour $j = 0$ **à** $n - 1$ **Faire**
      $p_{i,j} \leftarrow \min(p_{i,j}, p_{i,k} + p_{k,j})$
    Fin Pour j
  Fin Pour i

Fin Pour k

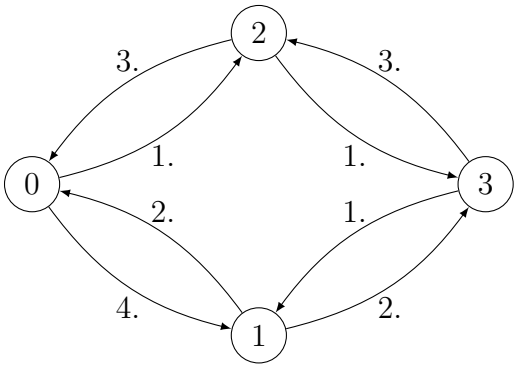
Quelle est la complexité de votre fonction ?

Réponse : $\mathcal{O}(|V|^3)$

Est–il possible d'exécuter l'algorithme de Floyd–Warshall lorsque certains poids sont négatifs ?

Réponse : Oui, à condition qu'il n'y ait pas de cycles de poids total strictement négatif.

16) On considère la graphe orienté pondéré suivant :



Détailler à la main les différentes étapes du parcours de ce graphe en partant du sommet 0 et en appliquant l'algorithme de Dijkstra.

Réponse : On stocke dans la file de priorité les couples (sommet, coût pour y arriver).

Sommet en cours d'exploration	Sommets déjà explorés	File de priorité	Conclusion
0	0	$(2, 1.), (1, 4.)$	$0 \to 0$ coût : 0.
2	$0, 2$	$(3, 2.), (1, 4.)$	$0 \to 2$ coût : 1.
3	$0, 2, 3$	$(1, 3.)$	$0 \to 3$ coût : 2.
1	$0, 2, 3, 1$		$0 \to 1$ coût : 3.

Langages et automates

17) Rappeler la définition d'un automate. Rappeler la définiton d'un automate déterministe, d'un automate complet.

Réponse : Un automate est un quintuplet (Q, A, E, I, T) *où* Q *est l'ensemble des états,* A *est un alphabet,* I *et* T *sont des parties de* Q *(*I *est l'ensemble des états initiaux et* T *l'ensemble des états terminaux), et* $E \subset Q \times A \times Q$ *est l'ensemble des transitions.*
Un tel automate est dit déterministe s'il possède un seul état initial, et si pour toute origine $o \in Q$ *et toute lettre* $l \in A$*, il existe au plus une extrémité* $e \in Q$ *tel que* $(o, l, e) \in E$*.*
Un tel automate est dit complet si pour toute origine $o \in Q$ *et toute lettre* $l \in A$*, il existe au moins une extrémité* $e \in Q$ *tel que* $(o, l, e) \in E$*.*

18) Comment rendre complet un automate ?

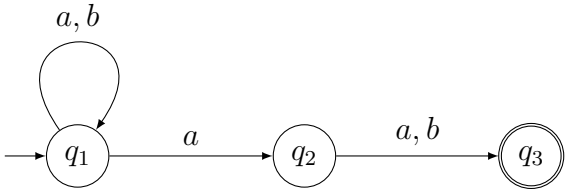
Réponse : Pour obtenir un automate complet équivalent, on rajoute un état–puits p (en supposant que $p \notin Q$) qui ne sera ni initial ni terminal et pour chaque état $o \in Q \cup \{p\}$, pour chaque lettre $l \in A$, s'il n'existe pas de transition d'origine o étiquetée par l, on rajoute une transition (o, l, p).

19) Démontrer que le langage $L = \{a^n b^n \mid n \in \mathbb{N}\}$ n'est pas rationnel.

1^e^ rédaction : supposons qu'il existe un automate (fini) qui reconnaisse L. On peut, quitte à déterminiser l'automate, supposer qu'il est déterministe complet. Notons e_n l'état où on arrive en lisant le préfixe a^n. Si $i < j$ alors $e_i \neq e_j$ car en lisant le suffixe b^i depuis e_i on arrive à un état terminal, mais pas depuis e_j. Ainsi l'automate posséderait une infinité d'états.

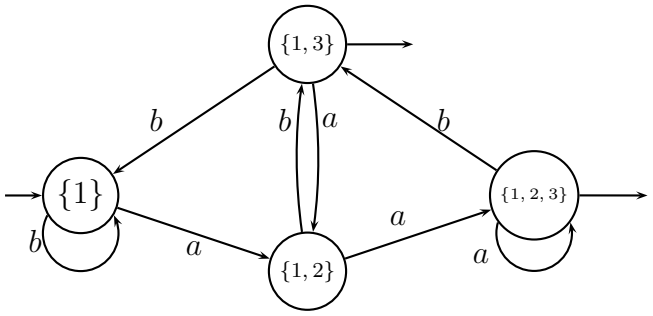
2^e^ rédaction : supposons qu'il existe un automate (déterministe complet) fini qui reconnaisse le langage L. Si on considère les états $e_p = \delta^*(q_0, a^p)$ atteints en lisant le mot a^p à partir de l'état initial. Le nombre d'états étant fini il existe p, q avec $p < q$ tels que $e_p = e_q$. Comme $a^p b^p$ appartient à L, $\delta^*(q_0, a^p b^p)$ est un état terminal mais $\delta^*(q_0, a^p b^p) = \delta^*(e_p, b^p) = \delta^*(e_q, b^p) = \delta^*(q_0, a^q b^p)$. Cela implique que $a^q b^p$ appartient au langage ce qui est absurde.

20) Rappeler l'algorithme de déterminisation d'un automate. Cet algorithme donne–t–il un automate complet ? Déterminiser l'automate :

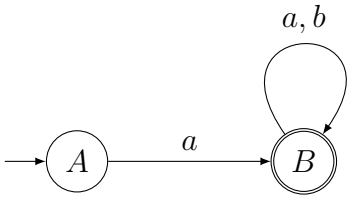


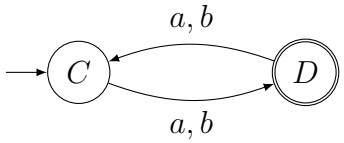
Réponse : pour déterminiser l'automate (Q, A, E, I, T) on construit l'automate $(\mathcal{P}(Q), A, E', \{I\}, T')$, où T' est l'ensemble des parties de Q qui contiennent au moins un élément de T, et E' est l'ensemble des transitions $(U, l, V) \in \mathcal{P}(Q) \times A \times \mathcal{P}(Q)$ où $V = \{e \in Q \mid \exists u \in U, (u, l, e) \in E\}$
L'automate ainsi construit est équivalent à l'automate de départ, déterministe et complet (l'ensemble vide joue le rôle d'état–puits).
Voici l'automate qu'on obtient en déterminisant l'automate donné en exemple :

$\to$	$\{1\}$	a	$\{1,2\}$	
		b	$\{1\}$	
	$\{1,2\}$	a	$\{1,2,3\}$	$\to$
		b	$\{1,3\}$	$\to$
	$\{1,2,3\}$	a	$\{1,2,3\}$	$\to$
		b	$\{1,3\}$	$\to$
	$\{1,3\}$	a	$\{1,2\}$	
		b	$\{1\}$	



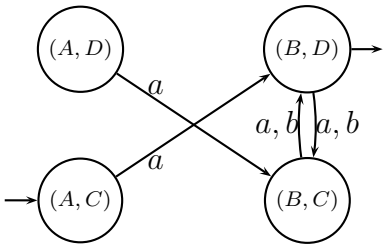
21) Rappeler la construction d'un automate produit de deux automates. Quel langage est reconnu par l'automate produit ? Faire cette construction dans le dans cas des automates :





Quels sont les langages reconnus par ces différents automates ?

Réponse : Soient deux automates sur un même alphabet $\mathscr{A}_1 = (Q_1, A, E_1, I_1, T_1)$ *et* $\mathscr{A}_2 = (Q_2, A, E_2, I_2, T_2)$. *On définit l'automate produit* $(Q_1 \times Q_2, A, E, I_1 \times I_2, T_1 \times T_2)$ *où :*
$E = \{((o_1, o_2), l, (e_1, e_2)) \in (Q_1 \times Q_2) \times A \times (Q_1 \times Q_2) \,|\, (o_1, l, e_1) \in E_1, (o_2, l, e_2) \in E_2\}$
Cet automate produit reconnaît l'intersection des langages reconnus par chacun des deux automates.
Dans les automates de l'exemple, le premier automate reconnaît le langage des mots dont a est préfixe, le second reconnaît le langage des mots de longueur impaire sur l'alphabet $\{a, b\}$*, et l'automate produit reconnaît donc le langage des mots de longueur impaire sur l'alphabet* $\{a, b\}$ *dont a est un préfixe.*



22) Glushkov

Soit e l'expression rationnelle $(a+b)^*a$. On cherche à construire un automate reconnaissant le langage défini par e avec l'algorithme de Glushkov. Effectuer successivement les opérations suivantes :

- linéarisation par marquage ;

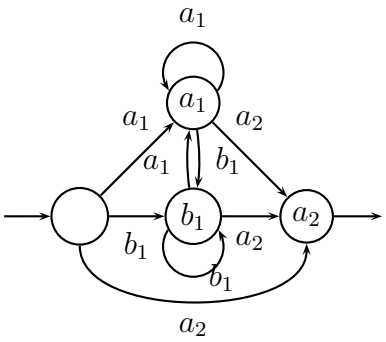
  Réponse : $(a_1 + b_1)^* a_2$

- description locale du langage ;

  Réponse : Notons P l'ensemble des premières lettres, S l'ensemble des dernières lettres, C l'ensemble des couples de lettres autorisés, et ε un marqueur permettant de savoir si le mot vide est reconnu ou non.

	P	S	C	ε
$a_1 + b_1$	$\{a_1, b_1\}$	$\{a_1, b_1\}$	$\emptyset$	$false$
$(a_1 + b_1)^*$	$\{a_1, b_1\}$	$\{a_1, b_1\}$	$\{a_1a_1, a_1b_1, b_1a_1, b_1b_1\}$	$true$
$(a_1 + b_1)^* a_2$	$\{a_1, b_1\}$	$\{a_2\}$	$\{a_1a_1, a_1b_1, b_1a_1, b_1b_1, a_1a_2, b_1a_2\}$	$false$

- construction de l'automate local ;



- suppression des marques.