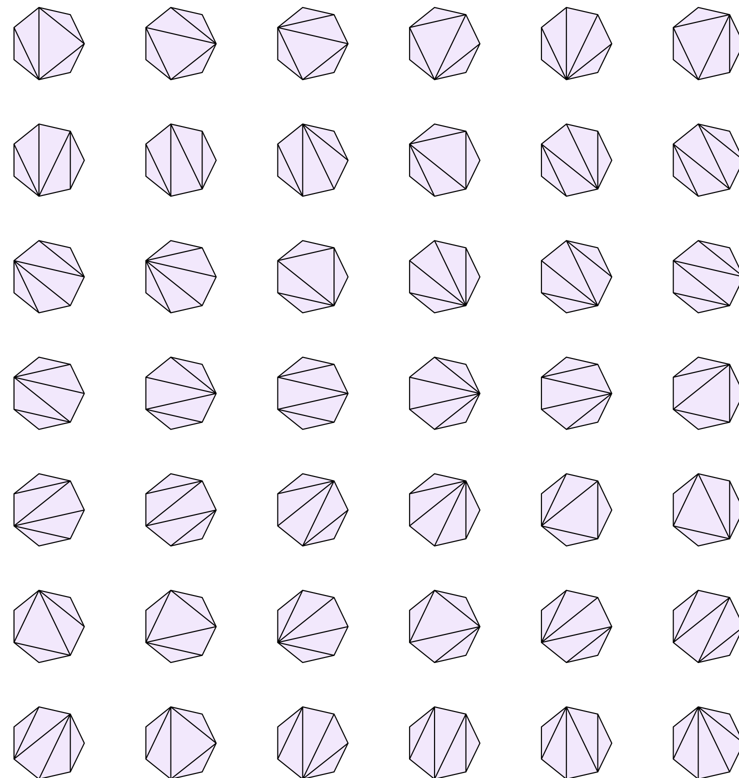
Exercices d'informatique



Computers are like Old Testament gods; lots of rules and no mercy.

Chapitre 1

Exercices

1.1 Révisions de programmation élémentaire

Exercice 1

Écrire les fonctions suivantes

1. `longueur` *qui calcule la longueur d'une liste.*
   `longueur : 'a list -> int`
2. `delete` *qui supprime un élément (toutes les occurrences) dans une liste.*
   `'a -> 'a list -> 'a list`
3. `rev` *qui renvoie le miroir d'une liste.*
   `'a list -> 'a list`
4. `merge` *qui fusionne deux listes triées en conservant le caractère trié.*
   `'a list -> 'a list -> 'a list`
5. `append` *qui concaténe deux listes.*
   `'a list -> 'a list -> 'a list`
6. `split` *qui sépare une liste de couples en un couple de listes.*
   `('a * 'b) list -> ('a list * 'b list)`
7. `mem` *qui recherche un élément dans un tableau.*
   `'a -> 'a vect -> bool`
8. `rev_vect` *qui renvoie le miroir d'un tableau.*
   `'a vect -> 'a vect`
9. `depth` *qui calcule la profondeur d'un nœud dans un arbre de type*

   type arbre
   ```
   type 'a tree =
       | F of 'a
       | N of 'a * 'a tree * 'a tree;;
   ```
   `'a -> 'a tree -> int`

Exercice 2

Corriger les propositions de réponses suivantes.

1. *pour* `longueur` *qui calcule la longueur d'une liste :*

   ```
   let rec longueur l = match l with
       | [] ->  failwith "liste vide"
       | _::q -> 1 + longueur q;;
   ```

2. *pour* `delete` *qui supprime un élément (toutes les occurrences) dans une liste :*

   ```
   let rec delete a l = match l with
       | [] -> []
       | t::q when t = a -> delete a q
       | t::q -> delete a t::q;;
   ```

3. *pour* `merge` *qui fusionne deux listes triées :*

   ```
   let rec merge l1 l2 = match (l1,l2) with
       | [], [] -> []
       | l, [] -> l
       | [], l -> l
       | t1::q1, t2::q2 -> if t1 <t2 then
                               t1::(t2::merge q1 q2)
                           else
                               t2::(t1::merge q1 q2);;

   let rec merge l1 l2 = match (l1,l2) with
       | l, [] -> l
       | [], l -> l
       | t1::q1, t2::q_2 ->
               (min t1 t2)::((max t1 t2)::merge q1 q2);;
   ```

4. *pour* `mem` *qui recherche un élément dans un tableau :*

   ```
   let mem t x =
       let n = vect_length t in
   ```

```
        let rec aux k = match k with
            | n -> false
            | _ -> (t.(k)=x) || aux (k+1)
        in aux 1;;

    let mem t x =
        let n = vect_length t in
        let b = ref false in
        for k = 0 to (n-1) do
            b:= (t.(k) = x)
        done;
        !b;;
```

Exercice 3

Écrire une fonction qui renvoie la réciproque d'une bijection donnée par son tableau de valeurs.

Exercice 4

Écrire une fonction qui compare deux entiers donnés par les listes de leurs écritures binaires. On précisera le sens choisi pour la représentation en binaire c'est-à-dire si la liste [1;1;0;1] représente 13 ou 11.
`int list -> int list -> bool`

Exercice 5

En remarquant que $(n+1)! = n!+n!+\ldots+n!$, proposer une fonction calculant la factorielle d'un entier donné en argument n'utilisant aucune multiplication. On pourra donner une version impérative et une version récursive de cette fonction.

Exercice 6

On représente un entier strictement positif par la liste croissante des exposants des puissances de 2 qui apparaissent dans son écriture binaire ; par exemple, l'entier $25 = 2^4 + 2^3 + 2^0$ est représenté par [0;3;4].

1. *Écrire une fonction qui prend un entier représenté ainsi et qui renvoie son double.*
2. *Écrire une fonction qui prend un entier représenté ainsi et qui renvoie son quotient par 2.*
3. *Écrire une fonction qui prend un entier représenté ainsi et qui renvoie son reste modulo 3.*
4. *Écrire une fonction qui prend un entier représenté ainsi et qui renvoie son successeur.*
5. *Écrire une fonction qui prend deux entiers représentés ainsi et qui renvoie leur somme.*

Exercice 7

Considérons la fonction suivante.

```
1  let aux t=
2      let n = vect_length t in
3      let v = make_vect (n+1) false in
4      v.(n) <- true;
5      for k = 0 to (n-1) do
6          while not v.(k) do
7              let i = ref k in
8              let x = ref t.(k) in
9              let j = ref (k + 1) in
10             while not v.(!j) do
11                 if t.(!j) < !x then begin
12                     t.(!i) <- t.(!j);
13                     incr i;
14                     t.(!j) <- t.(!i);
15                     end;
16                 incr j;
17             done;
18             v.(!i) <- true;
19             t.(!i) <- !x;
20         done;
21     done;
22     t;;
```

1. *Préciser le type de cette fonction.*
2. *Donner le résultat de l'instruction* `aux [|1;3;2;8;6;5;4;7|];;`
3. *Expliquer ce que fait globalement cette fonction.*
4. *Expliquer le rôle des lignes 8 à 17.*
5. *Que peut-on dire de* `t.(k)` *lorsque* `v.(k)` *(pour un entier* k≠n*) ?*
6. *Quel est l'avantage de cette implémentation sur l'implémentation d'une fonction récursive de même rôle ?*

1.2 Révisions algorithmique

Exercice 8

Dans cet exercice, on considère des permutations de n éléments représentées par le tableau de leurs images, c'est-à-dire qu'un tableau `t` *représente la permutation σ, si* `t.[k]` *contient la valeur $\sigma(k)$ pour tout indice k.*
Considérons l'algorithme suivant :

```
f(t)
    Tant que t.[0]<> 0 faire
        échanger les cases d'indices t.[0] et 0
    Renvoyer t
```

1. *La fonction* `f` *termine-t-elle pour tout tableau représentant une permutation ?*
2. *Que renvoie* `f([|5;4;2;1;0;3|])` *?*
3. *Écrire cette fonction en Caml.*

Exercice 9 (*Chvàtal*)

Dans cet exercice, on considère des permutations de $[\![1, n]\!]$ représentées par la liste de leurs images.
Considérons l'algorithme suivant

```
g(l)
    Tant que hd(l)<>1 faire
        retourner le préfixe de longueur hd(l) dans l
    Renvoyer l
```

1. *À une liste $l = [l_1; \dots; l_n]$ représentant une permutation, on associe l'entier*
$$\tilde{l} = \sum_{k=1}^{n} 2^k \mathbb{1}_{l_k = k}.$$
Montrer que si l' est la liste déduite de l en retournant le préfixe de longueur $l_1 > 1$, alors $\tilde{l'} > \tilde{l}$.
2. *En déduire la terminaison de la fonction* `g`.
3. *Écrire la fonction en Caml.*

Exercice 10

1. *Écrire une fonction qui renvoie la différence symétrique de deux listes. Quelle est sa complexité ?*
2. *Même question mais en supposant les listes triées par ordre croissant.*

Exercice 11

Écrire une fonction `maximum` *selon la méthode diviser pour régner pour obtenir le maximum d'un tableau. Préciser sa complexité temporelle.*

Exercice 12

Les nombres de Hamming sont les entiers naturels non nuls dont les diviseurs premiers appartiennent à $\{2, 3, 5\}$; par exemple, 1, 2, 12, 160 sont des nombres de Hamming tandis que 14 et 275 ne le sont pas.

1. *Préciser les dix premiers nombres de Hamming.*
2. *Donner un algorithme le plus efficace possible pour déterminer si un entier n donné en argument est un nombre de Hamming. Quelle est sa complexité ?*
3. *Donner un algorithme qui détermine le plus petit nombre de Hamming supérieur ou égal à un nombre n donné en argument. La complexité doit être $O(r \log r)$ où $r \geqslant 1$ est l'entier tel que $h_{r-1} \leq n < h_r$ où h_k désigne le k-ième nombre de Hamming.*
4. *Les pseudo-nombres de Hamming sont les entiers naturels non nuls dont les diviseurs premiers appartiennent à $\{2, 3\}$. Donner un algorithme qui détermine le plus petit pseudo-nombre de Hamming supérieur ou égal à un nombre n donné en argument avec une complexité en $O(\log n)$.*

Exercice 13

Soit E un ensemble fini, $f : E \to E$ une fonction et $u \in E^{\mathbb{N}}$ une suite telle que $u_{n+1} = f(u_n)$ pour tout $n \in \mathbb{N}$.

1. *Montrer qu'il existe un entier $i \leq |E|$ tel que $(u_n)_{n \geq i}$ est périodique.*
Soit r le plus petit entier vérifiant cette propriété et T la plus petite période de $(u_n)_{n \geq r}$.
2. *Montrer que, si deux entiers $i < j$ vérifient $u_i = u_j$, alors $i \geq r$ et $j - i$ est multiple de T.*
3. *Soit v la suite définie par $v_n = u_{2n}$ pour tout n. Caractériser l'ensemble A des entiers n tels que $u_n = v_n$.*
4. *En déduire un algorithme pour calculer T de complexité temporelle $O(r + T)$ (mais de complexité spatiale constante). On suppose qu'un appel à f correspond à une complexité constante.*

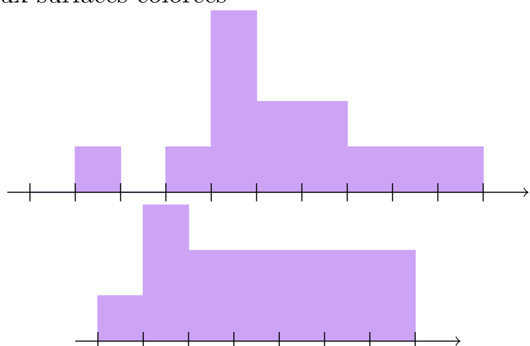
5. Notons i le plus petit élément non nul de A. En comparant la suite u avec la suite w définie par $w_n = u_{i+n}$ pour tout n, montrer qu'on peut également déterminer l'entier r avec cette même complexité .

Exercice 14

Définissons un profil d'horizon par une liste de points à coordonnées entières triée par ordre des abscisses croissantes. Un profil définit une surface polygonale du plan délimité par l'axe des abscisses et des segments horizontaux et verticaux associés aux points de la liste. Par exemple, les profils

```
[(0,0);(1,1);(2,0);(3,1);(4,4);(5,2); (7,1);(10,0)]
[(0,1);(1,3);(2,2);(7,0)]
```

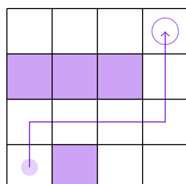
correspondent aux surfaces colorées



Déterminer une fonction qui prend un profil et renvoie l'aire maximale d'un rectangle entièrement inclus dans la surface correspondante (dans les exemples précédents, les résultats sont 7 et 12).
On pourra considérer une pile des points correspondants aux « montées » jusqu'au point actuel dans le parcours du profil.

Exercice 15

On considère un labyrinthe de taille $n \times n$ donné par une matrice de booléens `l` *: la case* `(i,j)` *est bloquée si, et seulement si,* `l.(i).(j)` *vaut* `true`*. Écrire une fonction qui détermine s'il existe un chemin joignant la case $(0, 0)$ à la case (n, n) n'utilisant que des déplacements vers le haut ou vers la droite et qui renvoie un tel chemin.*



1.3 Arbres

Exercice 16

Écrire la fonction `depth_min` *qui calcule la profondeur minimale d'une feuille dans un arbre binaire.*

Exercice 17

Écrire une fonction qui renvoie la feuille la plus profonde d'un arbre binaire.

Exercice 18

1. *Écrire une fonction qui remplit un tableau avec la taille du fils du nœud k dans un arbre de type*
   ```
   type arbre = N of int * arbre list;;
   ```
   en supposant que les nœuds sont indexés par des entiers distincts entre 0 et N (nombre fixé au préalable).

2. *Reprendre la question pour un arbre binaire de type*
   ```
   type arbre = Vide | N of int * arbre * arbre;;
   ```

Exercice 19

Considérons le type arbre suivant

```
type 'a arbre = N of 'a * 'a arbre list;;
```

Écrire une fonction `chemin` *qui prend un arbre a et un nœud x et qui renvoie la liste des nœuds de a parcourus de la racine jusqu'au nœud x.*

Exercice 20

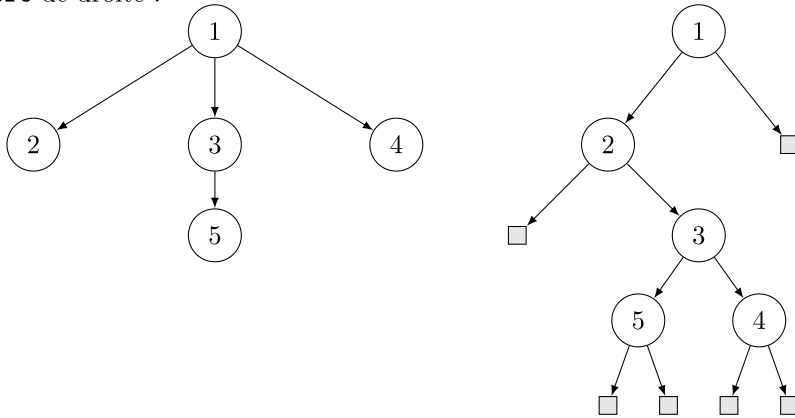
Considérons les deux types d'arbres suivants :

```
type 'a arbre= N of 'a * 'a arbre list;;

type 'a barbre =
    | Vide
    | Nb of 'a *'a barbre *'a barbre;;
```

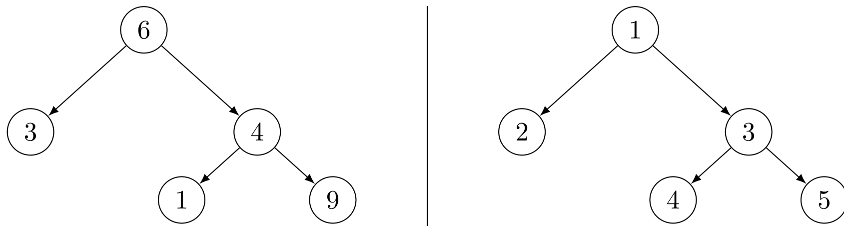
Écrire une fonction qui, à un arbre de type `arbre`*, associe un arbre de type* `barbre` *dans lequel chaque nœud admet pour fils gauche l'arbre associé à son fils aîné et pour fils droit l'arbre associé à son frère suivant.*
Par exemple, on passe de l'arbre de type `arbre` *de gauche à l'arbre de type*

`barbre` *de droite :*



Exercice 21

Écrire une fonction `test_squelette` *qui teste l'égalité du « squelette » de deux arbres binaires.*
Par exemple, les deux arbres suivants ont le même squelette

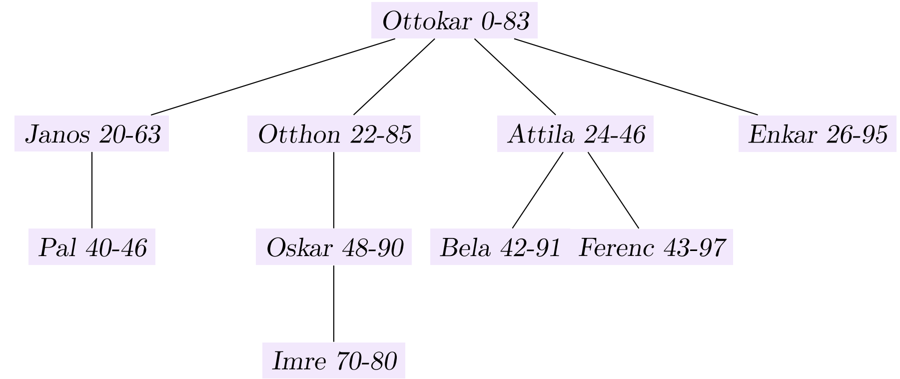


Exercice 22

Considérons un arbre avec tous les représentants mâles de la descendance directe du roi de Syldavie (chaque étiquette est un triplet composé d'un prénom de type `string` *et de deux entiers qui correspondent aux dates de naissance et de décès). Le type correspondant à cet arbre est*

```
type  arbre =  N of  string*int*int*(arbre list);;
```

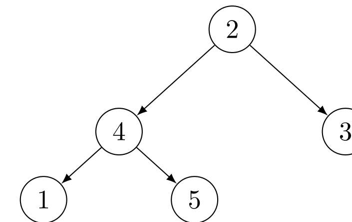
Écrire une fonction `rois` *qui prend un tel arbre généalogique (avec les listes d'enfants supposées triées de l'aîné vers le benjamin) et donne les différents dépositaires du titre. Par exemple, avec l'arbre suivant*




```
Ottokhar
Otthon
Oskar
Bela
Ferenc
```

Exercice 23

Montrer qu'un arbre binaire est uniquement déterminée par la donnée de la liste de ses nœuds dans un parcours en profondeur préfixe et la liste de ses nœuds dans un parcours en profondeur infixe.
Par exemple, avec les données préfixe `[2;4;1;5;3]` *et infixe* `[1;4;5;2;3]`*, on doit retrouver l'arbre*



1.4 Expressions

Exercice 24

On reprend le type d'expressions logiques du cours et la fonction `negation`*. Comparer, pour chaque expression* `f`*, la hauteur des expressions* `f` *et* `negation (negation f)`*.*

Exercice 25

On reprend le type d'expressions logiques du cours. Écrire une fonction `sans_et` *qui enlève toutes les occurrences du connecteur* `et`*.*

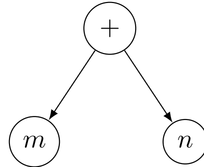
Exercice 26

On reprend le type d'expressions logiques du cours.

1. *Écrire une fonction qui prend une expression logique et qui renvoie une expression équivalente sous forme disjonctive (c'est-à-dire une disjonction d'une conjonction de littéraux).*
2. *Reprendre la première question avec la forme conjonctive.*

Exercice 27

Écrire une fonction de simplification des expressions algébriques qui effectue les opérations entre entiers.
Par exemple, on remplace par le nœud $m+n$ l'arbre suivant



Exercice 28

1. *Écrire une fonction* `expand` *qui distribue les produits sur toutes les sommes dans une expression algébrique donnée en argument.*
2. *Commenter la fonction suivante*

```
let rec expand f = match f with
    | prod(add(a,b),c)  -> add (prod(expand a,expand c),
                                prod(expand b,expand c))
    | prod(a, add(b,c)) -> add (prod(expand a,expand b),
                                prod(expand a,expand c))
    | moins(a)          -> moins (expand a)
    | add(a,b)          -> add(expand a,expand b)
    | prod(a,b)         -> prod(expand a,expand b)
    | quot(a,b)         -> quot(expand a,expand b)
    | _                 -> f;;
```

Exercice 29

On reprend le type d'expressions algébriques du cours.

1. *Écrire une fonction* `derivation` *qui calcule la dérivée d'une expression algébrique.*
2. *Déterminer la hauteur de la dérivation de l'expression correspondant à $\frac{2x+1}{2x+3}$ avec la fonction précédemment codée.*
3. *Donner un majorant de la hauteur de la dérivée d'une expression de hauteur n (on pourra discuter selon le nombre de certains connecteurs).*

Exercice 30

Une expression régulière sur un alphabet Σ est

- ▷ *soit $\varnothing$;*
- ▷ *soit ε ;*
- ▷ *soit un caractère de Σ ;*
- ▷ *soit la concaténation de deux expressions régulières que l'on note $e.f$ ou, plus simplement, ef ;*
- ▷ *soit la réunion (aussi appelée le choix) de deux expressions régulières que l'on note $e|f$;*
- ▷ *soit l'itération (aussi appelée l'étoile) d'une expression régulière que l'on note $e\star$.*

On les implémente avec le type suivant

```
type expreg=
    | Vide
    | MotVide
    | Const of string
    | Concatenation of expreg * expreg
    | Choix of expreg * expreg
    | Etoile of expreg;;
```

L'expression régulière $e = a|bb\star a$ est ainsi codée par

```
let e = Choix (Const "a", Concatenation (Concatenation (Const "b",
        Etoile (Const "b")), Const "a"));;
```

La hauteur d'étoile d'une expression régulière e est l'entier $h(e)$ défini récursivement à partir des règles suivantes

- ▷ $h(\varnothing) = 0$
- ▷ $h(\varepsilon) = 0$
- ▷ *pour tout $x \in \Sigma$, $h(x) = 0$*
- ▷ *pour toutes expressions régulières e et f, $h(e|f) = \max\{h(e), h(f)\}$*
- ▷ *pour toutes expressions régulières e et f, $h(e.f) = \max\{h(e), h(f)\}$*
- ▷ *pour toute expression régulière e, $h(e\star) = h(e) + 1$*

Écrire une fonction Caml `hauteur` *qui renvoie la hauteur d'étoile d'une expression régulière.*

Exercice 31

L'exposant maximal d'un entier $n \in \mathbb{N}^*$ en base b est l'unique entier p tel que

$$b^p \leq n < b^{p+1}$$

La décomposition complète d'un entier n en base b est définie récursivement par

- ▷ 0 *si* $n = 0$
- ▷ $b^d + d'$ *où d désigne la décomposition complète en base b de l'exposant maximal de n en base b et d' désigne la décomposition complète en base b de $n - b^d$.*

Une décomposition complète en base b ne s'écrit qu'avec les « chiffres » b et 0 et les opérations d'exponentiation et d'addition. Par exemple, la décomposition complète en base 2 de 9 est

$$9 = 2^{2^{2^0}+2^0} + 2^0$$

1. *Déterminer les décompositions complètes en base 3 des entiers 81 et 83.*
   On considère dorénavant des arbres binaires de type

```
type arbrebin =
    | Vide
    | N of arbrebin*arbrebin ;;
```

   De plus, à un entier naturel n, on associe `T(b,n)` *l'arbre binaire de décomposition complète en base b défini par*

```
T(b,0)= Vide
T(b,n)= N(T(b,d), T(b,d'))
```

   où d désigne l'exposant maximal de n en base b et $d' = n - b^d$.
2. *Écrire une fonction* `maxexp` *qui prend en argument deux entiers naturels non nuls* `b` *et* `n` *et renvoie* `d` *le plus grand exposant de* `n` *en base* `b` *et l'entier $d' = n - b^d$.*
3. *Écrire une fonction* `arbre` *qui prend en argument un entier naturel non nul* `b` *et un entier naturel* `n` *et renvoie l'arbre de décomposition complète* `T(b,n)`.
   Justifier la terminaison de cette fonction.
4. *Écrire une fonction* `erbra` *qui prend en argument un entier naturel non nul* `b` *et un arbre binaire* `T` *et renvoie l'entier naturel* `n` *dont* `T` *est l'arbre de décomposition complète. Ainsi, l'instruction* `erbra(b,arbre(b,n))` *renvoie l'entier* `n`.
5. *Écrire, avec les fonctions précédentes, une fonction* `add_arbre` *qui associe à deux arbres de décomposition complète* `arbre(b,n)` *et* `arbre(b,m)`, *l'arbre de décomposition complète* `arbre(b,n+m)`.

1.5 ABR/Tas

Exercice 32

Écrire la fonction `predecesseur` *qui prend un arbre binaire de recherche et l'étiquette d'un de ses nœuds et renvoie le prédécesseur de ce nœud dans le parcours infixe de l'arbre.*

Exercice 33

1. *Écrire la fonction* `suppression_droite` *qui prend un arbre binaire et renvoie le couple formé de son plus grand élément et le reste de l'arbre.*
2. *En déduire une nouvelle fonction* `suppression` *qui retire un élément d'un arbre binaire de recherche.*

Exercice 34

Écrire une fonction `vecteur_arbre` *qui passe d'un arbre binaire codé par un vecteur à sa représentation par le type suivant*

```
type arbre =
    | Vide
    | N of 'a * arbre * arbre;;
```

Exercice 35

Écrire une fonction `verification_tas` *qui vérifie si l'arbre codé par un vecteur correspond à un tas.*

Exercice 36

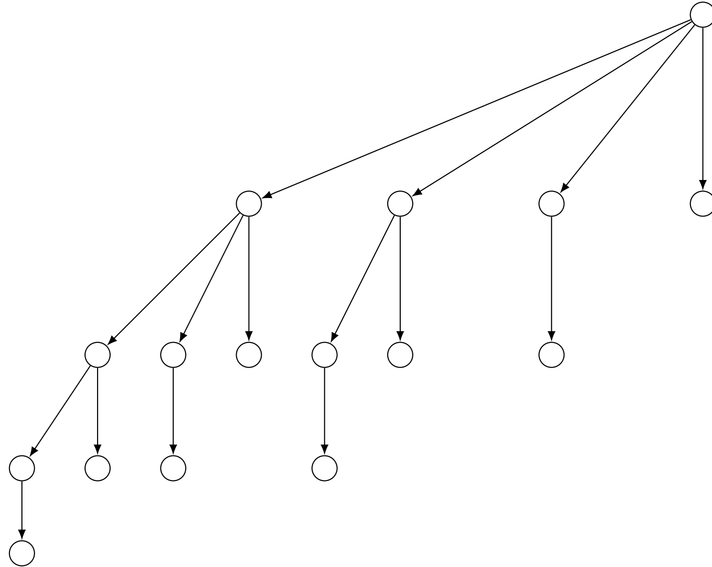
Soit b un entier supérieur ou égal à 2. Proposer un codage pour un tas b-aire puis écrire une fonction qui insère un élément dans un tel tas.

Exercice 37

Un arbre binomial d'ordre k est

- ▷ *un arbre réduit à un nœud si $k = 0$*
- ▷ *une racine qui admet k fils qui sont respectivement des arbres binomiaux d'ordre $k-1$, $k-2$, ..., 0 respectivement.*

Voici par exemple le squelette d'un arbre binomial d'ordre 4



1. *Déterminer la taille (nombre de nœuds) d'un arbre binomial d'ordre k.*
2. *Déterminer le nombre de nœuds de profondeur p dans un arbre binomial d'ordre k.*
   Un tas binomial est un ensemble fini d'arbres binomiaux vérifiant la propriété de file de priorité et d'ordre deux à deux distincts.
3. *Expliquer comment on ajoute un nouvel élément dans un tas binomial.*

1.6 Combinatoire des graphes

Exercice 38

Soit (S, A) un graphe non orienté non connexe. Montrer que le complémentaire $(S, \overline{A})$ est connexe.

Exercice 39

Montrer qu'un graphe non orienté à n sommets et k composantes connexes admet au moins $n - k$ arêtes.

Exercice 40

Montrer qu'un graphe non orienté à n sommets et strictement plus de $\binom{n-1}{2}$ arêtes est connexe.
En déduire que, pour tous entiers $a_1, \ldots, a_N$ de somme n,

$$\sum_{k=1}^{N} \binom{a_k}{2} \leq \binom{n-1}{2}$$

Exercice 41

Soit $G = (S, A)$ un graphe connexe et $x \in S$. Montrer qu'il existe un sommet $y \neq x$ tel que le graphe G privé de y reste connexe.

Exercice 42

Soit $G = (S, A)$ un graphe non orienté. Une clique de G est un ensemble de sommets voisins deux à deux dans G tandis qu'une anti-clique est un ensemble de sommets deux à deux non voisins dans G.

1. *Soit S_1 une clique de G et S_2 une anti-clique de G. Montrer que*
   $$|S_1 \cap S_2| \leq 1.$$
2. *Montrer que si l'ensemble des sommets de G se partitionne en une clique et une anti-clique, alors il en est de même pour $G^* = (S, \overline{A})$, le graphe complémentaire de G.*
3. *Supposons $|S| = 6$. Montrer que G contient une clique de taille 3 ou une anti-clique de taille 3.*

Exercice 43

Un circuit eulérien d'un graphe orienté G fortement connexe, est un « parcours » qui part d'un sommet quelconque et finit par ce même sommet en empruntant chaque arc une et une seule fois.
Montrer que G admet un circuit eulérien, si, et seulement si, chaque sommet admet autant d'arcs entrants que d'arcs sortants.

Exercice 44

Soit $G = (S, A)$ un graphe connexe avec $A = \{a_1, \ldots, a_m\}$.
Un ensemble d'arêtes $B \subset A$ est pair si tous les sommets du graphe (S, B) sont de degré pair. En particulier, l'ensemble des arêtes d'un cycle est un ensemble pair.
On assimile dorénavant un cycle à son ensemble d'arêtes.

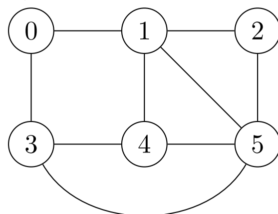
1. *Montrer que tout ensemble pair est la réunion disjointe de cycles. En déduire les ensembles pairs d'un arbre.*

   Le vecteur caractéristique de l'arête a_i est le m-uplet $\mathbb{I}_{a_i}$ d'éléments de $\mathbb{Z}/2\mathbb{Z}$ dont toutes les coordonnées sont nulles sauf la i-ième qui vaut 1. Le vecteur caractéristique $\mathbb{I}_B$ d'une partie $B \subset A$ est la somme des vecteurs caractéristiques des éléments de B.
2. *Montrer que l'ensemble E des vecteurs caractéristiques des parties paires de A est un $\mathbb{Z}/2\mathbb{Z}$-espace vectoriel.*

   Considérons (S, B) avec $B \subset A$ un arbre couvrant de G. Pour tout arête $a \in A \setminus B$, le graphe $(S, B \cup \{a\})$ contient un cycle C_a (passant par a) de vecteur caractéristique noté $\mathbb{I}_{C_a}$.
3. *Justifier que la famille des $\mathbb{I}_{C_a}$ avec $a \in A \setminus B$ est libre.*
4. *Montrer que la famille des $\mathbb{I}_{C_a}$ avec $a \in A \setminus B$ est une base de E.*
5. *En déduire le nombre de parties paires de G en fonction de $|S|$ et $|A|$.*

Exercice 45

Un graphe non orienté (avec au moins 3 sommets) est 2-connexe si, privé d'un sommet quelconque et des arêtes associées, il demeure connexe. Voici un exemple de graphe 2-connexe



Montrer qu'un graphe est 2-connexe si, et seulement si, pour tous sommets i et j, il existe un cycle contenant i et j.

Exercice 46

Un graphe est biparti s'il existe une partition (X, Y) de ses sommets telle qu'aucune arête ne relie deux sommets de X ou deux sommets de Y.
Montrer qu'un graphe est biparti si, et seulement si, il n'admet pas de cycle de longueur impaire.

Exercice 47

Soit G un graphe non orienté avec au moins trois sommets. Un cycle est un chemin fermé ne passant qu'une seule fois par un sommet. On définit, si elle existe, la maille de G comme la plus petite longueur d'un cycle.

1. *Soit d la distance maximale entre deux sommets de G. Montrer que la maille, si elle existe, est majorée par $2d + 1$.*
2. *Montrer qu'un graphe est biparti (c'est-à-dire tel que l'ensemble des sommets se partitionne en deux parties non vides S_1 et S_2 telles que toute arête joigne un sommet de S_1 et un sommet de S_2) si, et seulement si, il ne contient pas de cycle de longueur impaire. Qu'en déduire sur la maille d'un tel graphe ?*
3. *Montrer qu'un graphe à n sommets qui contient au plus un cycle, contient au plus n arêtes.*
4. *Montrer que la maille d'un graphe à n sommets et au moins $n+1$ arêtes est majorée par*
$$\left\lfloor \frac{2(n+1)}{3} \right\rfloor .$$
5. *Soit G un graphe tel que le plus petit degré δ d'un sommet et la maille m soient supérieurs ou égaux à 3. Montrer que, si m est pair, le nombre de sommets est alors minoré par*
$$\frac{2}{\delta - 2}((\delta - 1)^{m/2} - 1).$$

1.7 Manipulations de graphes

Exercice 48

Écrire des fonctions qui prennent un entier n et renvoient

1. *le graphe complet à n sommets,*
2. *le graphe linéaire à n sommets,*
3. *le cycle à n sommets,*
4. *le graphe divisoriel à n sommets.*

Exercice 49

Écrire les fonctions d'ajout et de suppression d'un arc pour un graphe orienté décrit par le type suivant

```
type graphe = (int * int list) list;;
```

Exercice 50

On considère le type suivant

```
type graphe = {mutable sommets : int list;
    mutable aretes : (int*int) list};;
```

1. *Expliquer les deux fonctions suivantes*

```
let rec filtre_sommets x l = match l with
    |[]   -> []
    |t::q -> if (t==x) then q
             else t::(filtre_sommets x q);;

let rec filtre_aretes x l = match l with
    |[]        -> []
    |(u,v)::q -> if (x==u || x==v) then (filtre_aretes x q)
                 else (u,v)::(filtre_aretes x q);;
```

2. *En déduire une fonction qui enlève un sommet (et les arêtes adjacentes à ce sommet) à un graphe de type* `graphe`.

Exercice 51

Écrire une fonction `retourne` *qui prend un graphe orienté et renvoie le graphe où chaque arc a été retourné. On précisera le type utilisé et on précisera sa complexité.*

Exercice 52

Écrire une fonction `puits` *qui détermine si un graphe admet un sommet puits total (c'est-à-dire un sommet $x \in S$ de degré entrant $|S|-1$ et de degré sortant 0).*

Exercice 53

Écrire une fonction qui prend un graphe donné par ses listes d'adjacence et renvoie le tableau des degrés entrant des sommets.

Exercice 54

Écrire une fonction qui teste si un graphe donné par sa matrice d'adjacence admet un cycle de longueur 3.

Exercice 55

Soit $G = (S, A)$ un graphe et $f : S \to [\![1, N]\!]$. La fonction f est un coloriage du graphe si, pour tout $(i, j) \in A$, $f(i) \neq f(j)$. Écrire une fonction `coloriage` *qui détermine si une fonction (donnée par un tableau) est un coloriage du graphe G, donné par listes d'adjacence.*

Exercice 56

Écrire une fonction qui donne la distance d'un sommet s à un sommet t dans un graphe non orienté non pondéré donné par sa matrice d'adjacence.

Exercice 57

Écrire une fonction qui prend un graphe donné par ses listes d'adjacence et renvoie le tableau des rangs d'apparition des sommets dans un parcours en profondeur depuis le sommet 0. On indique le rang -1 pour les sommets éventuellement non atteints par ce parcours.

Exercice 58

Soit $G = (S, A)$; pour tout $x \in S$, on définit l'excentricité de x comme $R(x) = \max\{d(x, y),\ y \in S\}$ où $d(x, y)$ est la distance entre les sommets x et $y \in S$. Le rayon de G est défini comme $R(G) = \min\{R(x),\ x \in S\}$. Le centre de S est alors l'ensemble de sommets $C(G) = \{x \in V,\ R(x) = R(G)\}$.

1. *Préciser le rayon des graphes complet, linéaire, cyclique et divisoriel.*
2. *Écrire une fonction qui prend un graphe connexe et un sommet puis qui fournit l'excentricité de ce sommet.*
3. *Écrire une fonction qui prend un graphe et renvoie son rayon.*

1.8 Plus courts chemins dans un graphe

Exercice 59

Soit G un graphe non orienté connexe, i et j deux sommets.

1. *Donner un exemple où la suppression d'un arc appartenant à un plus court chemin de i à j ne change pas la distance de i à j.*
2. *Un arc est vital pour i et j si la suppression de cet arc augmente la distance de i à j. Est-ce qu'un arc vital pour i et j appartient nécessairement à tout plus court chemin de i à j ?*

Exercice 60

Que renvoie l'algorithme de Dijkstra à partir d'un sommet i pour un graphe sans cycle absorbant et tel que tous les arcs issus d'un sommet autre que i aient une poids positif ?

Exercice 61

Adapter l'algorithme de Floyd-Warshall pour tester l"existence d'un cycle de poids strictement négatif.

Exercice 62

Comparer l'algorithme de Floyd-Warshall et l'algorithme de Bellman-Ford appliqué en chaque sommet.

1.9 Arbres couvrants

Exercice 63

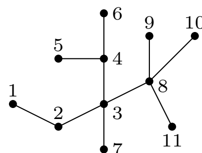
Déterminer le nombre d'arbres couvrants du graphe cyclique C_n.

Exercice 64

Le codage de Prüfer d'un arbre T de sommets $\{1, \dots, n\}$ est la liste L définie comme résultat de l'algorithme suivant

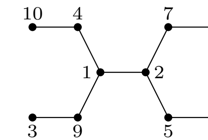
- ▷ *Tant que l'arbre T contient strictement plus de deux sommets*
  - — *identifier la feuille x de T (c'est-à-dire un sommet de degré 1) de plus petit numéro ;*
  - — *déterminer le voisin y de x dans T ;*
  - — *ajouter y à L, retirer x des sommets de T et $\{x, y\}$ des arêtes de T.*
- ▷ *Renvoyer L.*

Par exemple, l'arbre suivant



admet le codage de Prüfer $< 2; 3; 4; 4; 3; 3; 8; 8; 8 >$.

1. *Déterminer le codage de Prüfer de l'arbre suivant*



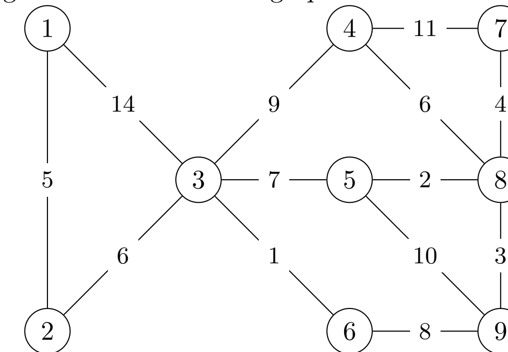
2. *Montrer qu'un sommet s est de degré k dans l'arbre T si, et seulement si, s apparaît $k - 1$ fois dans le codage de Prüfer de l'arbre T.*
3. *Proposer une méthode pour reconstruire l'arbre T à partir de son codage de Prüfer et de son tableau des degrés.*
4. *Déterminer le nombre d'arbres différents obtenus avec l'ensemble de sommets $\{1, \dots, n\}$.*
5. *En déduire le nombre d'arbres couvrants du graphe complet K_n.*

Exercice 65

Proposer un algorithme pour déterminer un arbre couvrant de poids maximal d'un graphe non orienté pondéré connexe.

Exercice 66

Appliquer les algorithmes du cours au graphe suivant



Exercice 67

Considérons Q_3 le graphe du cube en dimension 3 et munissons le d'une fonction de poids f telle que le poids d'une arête est le numéro de la coordonnée qui diffère entre les deux extrémités de l'arête.
Déterminer un arbre couvrant de poids minimal.

Exercice 68

Soit $G = (V, E)$ un graphe non orienté pondéré connexe. Pour tout sommet x, considérons a_x une arête d'extrémité x et de poids minimal.
Est-ce que $\{e_x,\ x \in V\}$ est un arbre couvrant de poids minimal ?

1.10 Mots

Exercice 69

Déterminer les mots sur $\Sigma = \{a, b\}$ dont tous les facteurs sont des préfixes.

Exercice 70

Soit $a,\ b \in \Sigma$ et $x \in \Sigma^*$ tels que $ax = xb$. Montrer que $a = b$ et que $x \in a^*$.

Exercice 71

Les mots de Fibonacci sur l'alphabet $\Sigma = \{a, b\}$ sont définis par récurrence par $f_0 = \varepsilon$, $f_1 = a$, $f_2 = b$ et, pour tout $n \geq 3$, $f_n = f_{n-1}f_{n-2}$.

1. *Montrer que, pour tout $n \geq 3$, le suffixe de longueur 2 de f_n est ab si n est pair, ba si n est impair.*
2. *Notons, pour tout $n \geq 3$, g_n le mot déduit de f_n en enlevant le suffixe de longueur 2 précisé à la question précédente. Montrer que g_n est un palindrome.*

Exercice 72

1. *Montrer qu'un mot m de Dyck s'écrit de manière unique sous la forme am_1bm_2 où m_1 et m_2 sont des mots de Dyck.*
   Que valent m_1 et m_2 pour $m = abab$ et $m = aababb$? Considérons, pour la suite de l'exercice, les types

```
type lettre = a | b;;

type arbre = Vide | N of arbre*arbre;;
```

2. *Écrire une fonction `decompose` qui prend un mot de Dyck m et renvoie le couple (m_1, m_2) de mots définis à la question précédente ; elle correspond au typage*

```
decompose : lettre list -> lettre list * lettre list
```

   L'arbre associé à un mot de Dyck m est défini récursivement par
   - ▷ *une feuille pour le mot vide,*
   - ▷ *un noeud ayant l'arbre de m_1 comme fils gauche et l'arbre de m_2 comme fils droit.*
3. *Donner l'arbre associé au mot $aaabbabbaababb$.*
4. *Écrire une fonction `mot_arbre` qui prend un mot de Dyck de type `lettre list` et renvoie l'arbre correspondant.*
5. *Écrire une fonction `arbre_mot` qui prend un arbre binaire et renvoie le mot de Dyck (de type `lettre list`) correspondant.*

Exercice 73 (*Mots de Lyndon*)

Une rotation propre d'un mot u non vide est un mot v tel qu'il existe deux mots x et y non vides vérifiant $u = xy$ et $v = yx$.
Un facteur propre de u est un facteur de u différent de u et de ε.
Soit Σ un alphabet totalement ordonné. Un mot $u \in \Sigma^*$ non vide est de Lyndon s'il est strictement plus petit au sens de l'ordre lexicographique, noté $<$, que toutes ses rotations propres.

1. *Écrire une fonction qui teste si un mot donné est de Lyndon.*
2. *Soit u un mot non vide qui admet un facteur propre qui est à la fois son préfixe et son suffixe. Montrer que u n'est pas un mot de Lyndon.*
3. *Montrer que u est un mot de Lyndon si, et seulement si, u est strictement plus petit que tous ses suffixes propres.*
4. *Montrer que si u et v sont des mots de Lyndon et $u < v$ alors le mot uv est aussi de Lyndon.*
5. *Montrer que tout mot non vide s'écrit de manière unique comme une concaténation décroissante (toujours pour l'ordre lexicographique) de mots de Lyndon.*
6. *Proposer un algorithme permettant d'obtenir cette décomposition d'un mot quelconque non vide.*
7. *Écrire une fonction qui donne la décomposition d'un mot quelconque.*

Exercice 74 (*Mots de Lukasiewicz*)

On considère un alphabet infini $\Sigma = \{a_n,\ n \in \mathbb{N}\}$ et δ le morphisme défini par $\delta(a_n) = n - 1$ pour tout $n \in \mathbb{N}$. Un mot $u \in \Sigma^*$ est de Lukasiewicz s'il vérifie les propriétés
- ▷ $\delta(u) = -1$,
- ▷ *pour tout préfixe propre v de u, $\delta(v) \geq 0$.*

1. *Les mots suivants sont-ils de Lukasiewicz ?*

$$a_2a_1a_0a_2a_0a_1a_0, \qquad a_3a_2a_0a_2a_0a_1a_1a_0.$$

   Notons L l'ensemble des mots de Lukasiewicz.
2. *Caractériser $L \cap \{a_0, a_1\}^*$.*
3. *Caractériser $L \cap \{a_0, a_2\}^*$.*
   On considère maintenant un système de réécriture. On se donne des

variables X et des règles de transformation, pour tout $n \in \mathbb{N}$, $X \to a_N X^N$.

Un mot est engendré par ce système de réécriture s'il peut s'obtenir à partir de X par applications successives des règles de réécriture. Par exemple, le mot $a_2a_0a_1a_0$ est obtenu par

$$X \to a_2XX \to a_2Xa_1X \to a_2a_0a_1X \to a_2a_0a_1a_0.$$

On note $X \xrightarrow{*} u$ pour signifier que le mot u est obtenu à partir de X en un nombre fini d'étapes.

Plus généralement, pour des mots u, $v \in (\Sigma \cup \{X\})^*$ on peut appliquer une règle qui transforme u en v si $u = u_1Xu_2$ et $v = u_1a_nX^nu_2$ et on note $u \to v$. On définit de même $u \xrightarrow{*} v$ et on remarque que l'on a toujours $u \xrightarrow{*} u$.

4. *Montrer que $a_2a_1a_0a_2a_0a_1a_0$ est engendré par ce système de réécriture.*
5. *Soit $u \in (\Sigma \cup \{X\})^*$ tel que $X^2 \xrightarrow{*} u$. Montrer qu'il existe des mots u_1 et u_2 tel que $u = u_1u_2$, $X \xrightarrow{*} u_1$ et $X \xrightarrow{*} u_2$.*
6. *Soit $u \in \Sigma^*$ engendré par ce système de réécriture. Montrer que $u \in L$.*
7. *Soit $u \in \Sigma^*$. Montrer que si $u \in L$, alors u est engendré par ce système de réécriture.*
8. *Soit $n \in \mathbb{N} \setminus \{0\}$. En se ramenant aux arbres, dénombrer $L \cap \Sigma^n$.*

Exercice 75

Considérons l'alphabet $\Sigma = \{a, b\}$ et la relation $\to$ de réécriture sur Σ^* définie, pour tous u, $v \in \Sigma^*$ par $uabv \to uv$. Définissons enfin la clôture réflexive et transitive $\xrightarrow{*}$ de $\to$.

1. *Montrer qu'il n'existe pas de suite infinie de mots $(u_n)_n$ telle que*
$$\forall n \in \mathbb{N},\ u_n \to u_{n+1}.$$
2. *Déterminer les mots $u \in \Sigma^*$ irréductibles, c'est-à-dire tels qu'il n'existe pas de mot $v \in \Sigma^*$ vérifiant $u \to v$.*
3. *Décrire le langage $L = \{u \in \Sigma^*,\ u \xrightarrow{*} \varepsilon\}$.*
4. *Soit $u \in \Sigma^*$. Montrer qu'il existe un unique couple d'entiers (m, n) tel que $u \xrightarrow{*} b^ma^n$. Préciser les valeurs de m et n en fonction de u.*
5. *Soit u_1, $u_2 \in \Sigma^*$. Montrer que si $u_1 \xrightarrow{*} b^{m_1}a^{n_1}$ et $u_2 \xrightarrow{*} b^{m_2}a^{n_2}$, alors $u_1u_2 \xrightarrow{*} b^ma^n$ avec m et n des entiers à déterminer en fonction de m_1, n_1, m_2 et n_2.*
6. *En déduire un calcul (de complexité linéaire en la longueur du mot) des entiers m et n associés à un mot $u \in \Sigma^*$.*

1.11 Langages

Exercice 76

Soit $\Sigma = \{a, b\}$. Décrire en une phrase les langages suivants

$$a^*ba^*,\ \Sigma^*b\Sigma^*,\ (\Sigma\Sigma)^*,\ \Sigma^*a^*,\ \Sigma^*a\Sigma^*a\Sigma^*b\Sigma^*,\ \varepsilon^*,\ (a^*b^*)^*.$$

Exercice 77

Soit $\Sigma = \{a, b, c\}$. Calculer $L_1 \cap L_2$ pour $L_1 = \{a^nb^nc^m,\ n, m \in \mathbb{N}\}$ et $L_2 = \{a^nb^mc^m,\ n, m \in \mathbb{N}\}$.

Exercice 78

Montrer que $(a\Sigma^*b \cup a)^*a\Sigma^* = a\Sigma^*$.

Exercice 79

Soit L un langage non vide. Montrer que $\varepsilon \in L$ si, et seulement si, $L \subset L^2$.

Exercice 80

Calculer $\sqrt{ab^*a}$.

Exercice 81

Calculer $\frac{1}{2}L$ pour $L = \{a^nb^na^n,\ n \in \mathbb{N}\}$.

Exercice 82

Un langage L est préfixe s'il n'existe pas deux mots u et $v \in L$ distincts tels que u soit un préfixe de v. Montrer que la concaténation de deux langages préfixes est préfixe.

Exercice 83

Notons, pour tout langage L, $\mathrm{pref}(L)$ le langage des préfixes des mots de L. Déterminer, pour tous langages L_1, L_2, le langage $\mathrm{pref}(L_1L_2)$.

Exercice 84

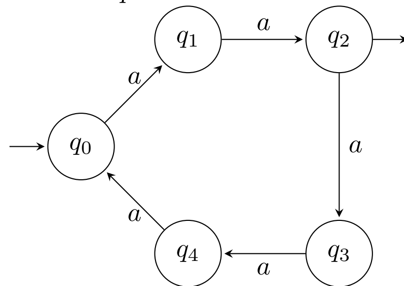
Un code sur un alphabet Σ est un langage L tel que l'égalité $x_1x_2\dots x_p = y_1y_2\dots y_q$ avec $x_1, \dots, x_p, y_1, \dots, y_q \in L$ entraîne $p = q$ et $x_i = y_i$ pour tout $i \in [\![1, p]\!]$.

1. *Déterminer les codes parmi les langages finis suivants*
   - $L_1 = \{ab, baa, abba, aabaa\}$
   - $L_2 = \{b, ab, baa, abaa, aaaa\}$
   - $L_3 = \{aa, ab, aab, bba\}$
   - $L_4 = \{a, ba, bba, baab\}$
2. *Soit* $u \in \Sigma^*$, *montrer que* $\{u\}$ *est un code si, et seulement si,* $u \neq \epsilon$.
3. *Soit* u *et* $v \in \Sigma^*$ *distincts ; montrer que* $\{u, v\}$ *est un code si, et seulement si,* $uv \neq vu$.
4. *Soit* $L \subset \Sigma^*$ *un langage ne contenant pas* ϵ *et tel qu'aucun mot de* L *ne soit préfixe d'un autre mot de* L. *Montrer que* L *est un code.*

1.12 Automates

Exercice 85

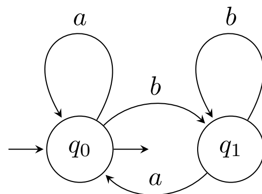
Déterminer le langage reconnu par l'automate suivant



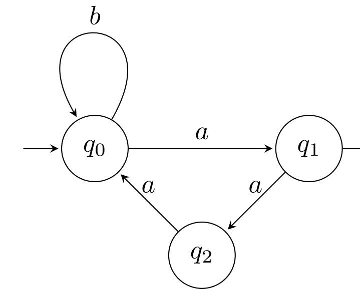
Exercice 86

Déterminer le langage reconnu par chacun des automates suivants :

1.

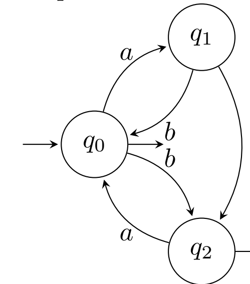


2.



Exercice 87

Déterminer le langage reconnu par l'automate suivant



Exercice 88

Déterminer un automate (déterministe) simple reconnaissant chacun des langages suivants sur l'alphabet $\Sigma = \{a, b\}$

1. $ab\Sigma^*$
2. $\Sigma^* ab \Sigma^*$
3. $\Sigma^* ab$
4. $\{m \in \Sigma^*,\ |m|_a \leq 1\}$
5. $\{m \in \Sigma^*,\ |m|_a = |m|_b[2]\}$
6. $\{m \in \Sigma^*,\ |m|_a = |m|_b\}$

Exercice 89

Déterminer un automate reconnaissant le langage des mots sur l'alphabet $\{a, b\}$ *ne comportant pas le facteur* aa.

Exercice 90

1. *Déterminer un automate reconnaissant le langage des mots sur l'alphabet* $\{a, b\}$ *comptant un nombre pair de* a.
2. *Déterminer un automate reconnaissant le langage des mots sur l'alpha-*

bet $\{a, b\}$ comptant un nombre pair de a et un nombre multiple de 3 de b.

Exercice 91

Déterminer un automate reconnaissant tous les mots sur l'alphabet $\{a, b\}$ comptant au moins deux occurrences de son dernier caractère.

Exercice 92

Déterminer un automate reconnaissant le langage local associé à $P = \{a, b\}$, $F = \{ab, bc, ca, aa, cc\}$ et $S = \{b, c\}$.

Exercice 93

Considérons l'alphabet $\{a, b\}$.

1. *Déterminer un automate reconnaissant le langage des mots ayant aba pour préfixe.*
2. *Déterminer un automate reconnaissant le langage des mots ayant aba pour suffixe.*
3. *Déterminer un automate reconnaissant le langage des mots ayant aba pour sous-mot.*
4. *Déterminer un automate reconnaissant le langage des mots ayant aba pour facteur.*

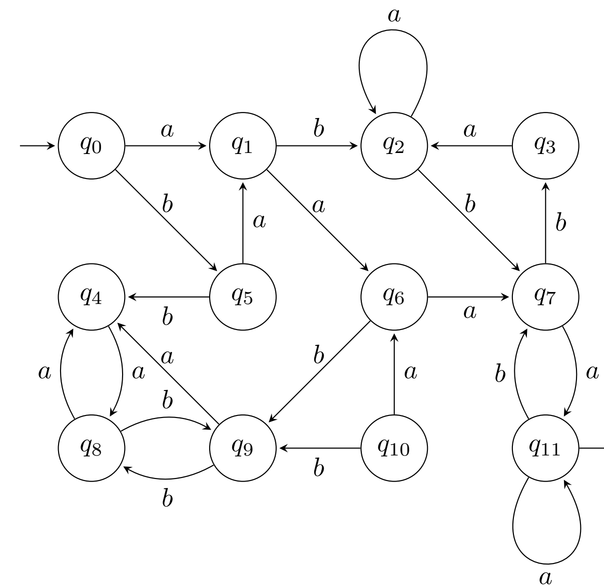
Exercice 94

Soit $m \in \Sigma^*$ un mot fixé et $\mathrm{Pref}(m)$ le langage des préfixes de m.

1. *Déterminer le langage reconnu par l'automate $(\mathrm{Pref}(m), \varepsilon, \{m\}, \delta)$ avec, pour tout $x \in \mathrm{Pref}(m)$ et tout $a \in \Sigma$, $\delta(x, a)$ est le plus long suffixe de xa qui appartient à $\mathrm{Pref}(m)$.*
2. *Adapter l'automate précédent pour qu'il reconnaisse les mots dont m est facteur.*

Exercice 95

Émonder l'automate suivant



Exercice 96

Soit $\mathcal{A}$ un automate à n états et L le langage reconnu par cet automate. Montrer que si L est non vide, alors il contient au moins un mot de longueur au plus $n - 1$.

Exercice 97

Soit A, B, C et D des langages sur un alphabet fini Σ avec D rationnel. L'objectif de l'exercice est d'étudier l'équation (E) : $AXB \cup C = D$ où l'inconnue X est un langage sur Σ.

1. *Soit X une solution de l'équation (E). Montrer que le langage*

$$\Delta(X) = \{u \in \Sigma^*,\ X \cup \{u\} \text{ est solution de } (E)\}$$

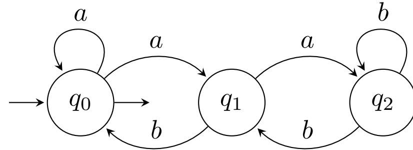
est rationnel.

2. *Montrer que si l'équation (E) admet une solution, alors (E) admet une solution rationnelle.*
3. *Dans cette question, on suppose également les langages A, B et C rationnels et décrits par leurs automates respectifs. Décrire un algorithme qui détermine si (E) admet une solution.*

1.13 Automates non déterministes

Exercice 98

Déterminer le langage reconnu par l'automate suivant puis proposer un automate déterministe reconnaissant le même langage.

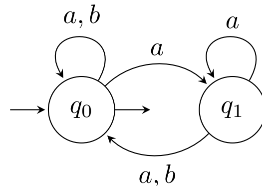


Exercice 99

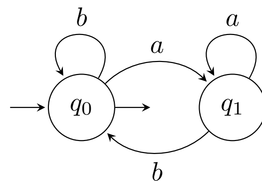
Déterminer un automate non déterministe et un automate déterministe reconnaissant le langage $a\Sigma^*a$.

Exercice 100

Déterminer le langage reconnu par l'automate

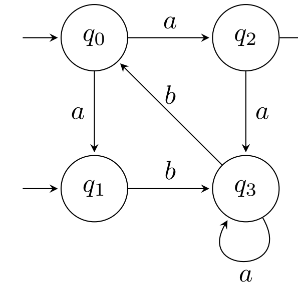


Est-ce le même langage reconnu que l'automate suivant ?



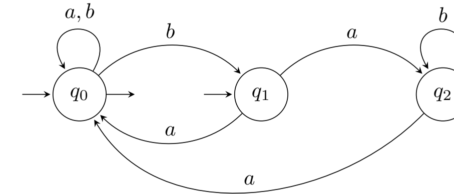
Exercice 101

Déterminiser l'automate suivant



Exercice 102

Déterminiser l'automate suivant :



Exercice 103

Trouver un automate non déterministe qui reconnaît le langage des suffixes de baba puis le déterminiser.

Exercice 104

Un automate non déterministe avec transitions instantanées est un quintuplet $(Q, I, F, \delta, \varphi)$ composé de

- *Q, un ensemble fini. Les éléments de Q sont les états de l'automates.*
- *$I \subset Q$ l'ensemble des états initiaux.*
- *$F \subset Q$ l'ensemble des états acceptants.*
- *δ une application de $Q \times \Sigma$ dans $\mathcal{P}(Q)$. Cette application est la fonction de transition.*
- *φ une application de Q dans $\mathcal{P}(Q)$. Cette application est la fonction de transition instantanée.*

La clôture instantanée d'une partie $A \subset Q$ est la plus petite partie, notée $\kappa(A)$, de Q contenant A et stable par φ.

La fonction de transition δ^* étendue aux mots est la fonction définie récursivement par

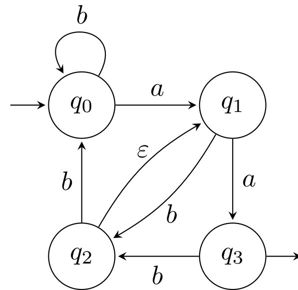
- *pour tout $q \in Q$, $\delta^*(q, \varepsilon) = \kappa(\{q\})$,*

▷ *pour tous $q \in Q$, $m \in \Sigma^*$ et $x \in \Sigma$,*

$$\delta^*(q, m.x) = \kappa \left(\bigcup_{q' \in \delta^*(q,m)} \delta(q', x) \right).$$

Un mot $m \in \Sigma^*$ est reconnu s'il existe $q_0 \in I$ tel que $\delta^*(q_0, m) \cap F \not\emptyset$. Le langage reconnu par cet automate est l'ensemble des mots reconnus.

1. *Montrer qu'un langage reconnu par un automate non déterministe avec transitions instantanées est reconnu par un automate déterministe.*
2. *Déterminiser l'automate suivant*



Exercice 105

Soit $L_n = \{uav \in \Sigma^*,\ |u| = n\}$ Montrer qu'il n'existe pas d'automate reconnaissant L_n avec strictement moins de $n + 3$ états.

Exercice 106

1. *Montrer que tout langage reconnu par un automate non déterministe est reconnu par un automate non déterministe avec un seul état acceptant.*
2. *Montrer qu'il existe des langages reconnus par un automate déterministe qui ne peuvent être reconnus par un automate déterministe avec un seul état acceptant.*

1.14 Langages reconnaissables

Exercice 107

Soit L le langage sur l'alphabet $\Sigma = \{a, b\}$ formé des mots où toute occurrence de a est suivie d'une occurrence de b. Le Langage L est-il reconnaissable ?

Exercice 108

Soit L le langage sur l'alphabet $\Sigma = \{a, b\}$ formé des mots m n'admettant pas le facteur a^2 et tel que $|m|_a = 0[2]$. Montrer que L est reconnaissable.

Exercice 109

Déterminer l'automate de Glushkov associé aux expressions régulières

▷ $a(a|b) \star a$

▷ $(a|b) \star (abb|\varepsilon)$

▷ $ab \star (ba) \star b$

Exercice 110

Soit (Q, q_0, F, δ) un automate. Montrer que le langage des mots bloquant cet automate est reconnaissable.

Exercice 111

Soit L un langage reconnaissable. Montrer que le langage des sous-mots des mots de L est reconnaissable.

Exercice 112

Soit L un langage reconnaissable. Montrer que $\sqrt{L}$ et $\frac{1}{2}L$ sont reconnaissables.

Exercice 113

Soit L, M des langages reconnaissables. Montrer que $L : M = \{m \in L,\ \exists u \in M,\ |m| = |u|\}$ est aussi reconnaissable.
Retrouver ainsi que, si L est reconnaissable, alors $\frac{1}{2}L$ est reconnaissable

Exercice 114

Soit L un langage reconnaissable. Montrer que $\{uv,\ vu \in L\}$ est aussi reconnaissable.

Exercice 115

Soit L un langage reconnaissable. Montrer que $\{m \in \Sigma^*,\ m\overline{m} \in L\}$ (où $\overline{m}$ est le miroir du mot m) est aussi reconnaissable.

Exercice 116

Soit L un langage reconnaissable sur l'alphabet $\Sigma = \{a, b\}$ et L_1 l'ensemble des mots qui ne diffèrent d'un mot de L (de même longueur) que d'une lettre. Montrer que L_1 est reconnaissable.

Exercice 117

Soit L un langage reconnaissable et M un langage quelconque. Montrer que le langage

$$M^{-1}L = \{m \in \Sigma^*,\ \exists u \in M,\ um \in L\}$$

est reconnaissable.

Exercice 118

Montrer que les langages suivants ne sont pas reconnaissables

- $L_1 = \{u \in \{a,b\}^*,\ |u|_a = |u|_b\}$
- $L_2 = \{a^n b^n c^n,\ n \in \mathbb{N}\}$
- $L_3 = \{a^i b a^j b a^{i+j},\ i,j \in \mathbb{N}\}$
- $L_4 = \{a^{n^2},\ n \in \mathbb{N}\}$
- $L_5 = \{uu,\ u \in \Sigma^*\}$
- L_6 *le langage des mots de Dyck*

Exercice 119

Un mot sur un alphabet Σ est double s'il contient au moins deux occurrences de chaque lettre présente dans son écriture.

1. *Montrer qu'un mot de longueur $2^{|\Sigma|}$ contient un facteur double.*
2. *Montrer que cette borne est optimale, c'est-à-dire qu'il existe un mot de longueur $2^{|\Sigma|} - 1$ sans facteur double.*
3. *Le langage des mots doubles est-il reconnaissable ?*

Exercice 120

On fixe un alphabet Σ. Pour tout langage $L \subset \Sigma^*$, la densité de L est la fonction ρ_L donnée par $\rho_L(n) = |L \cap \Sigma^n|$ pour tout $n \in \mathbb{N}$.

1. *Majorer et minorer $\rho_L(n)$ en fonction de $|\Sigma|$ et n. Ces bornes sont-elles atteintes ?*
2. *Donner une fonction de $\mathbb{N}$ dans $\mathbb{N}$ qui n'est la densité d'aucun langage rationnel.*

   Un langage rationnel L est fin s'il existe $K \in \mathbb{N}$ tel que, pour tout $n \in \mathbb{N}$ $\rho_L(n) \leq K$. Le but de ce problème est d'établir le théorème suivant

   Un langage rationnel L est fin si, et seulement si, il existe $l \in \mathbb{N}$ et des mots u_1 ,..., u_l , v_1,..., v_l, w_1,..., $w_l \in \Sigma^*$ tels que

   $$L = \bigcup_{k=1}^{l} u_k v_k^* w_k.$$

3. *Montrer le sens retour du théorème.*

   Un automate fini déterministe est monoboucle si, tout état q pouvant apparaître plusieurs fois dans un chemin de l'état initial à un état final vérifie les trois propriétés

   (i) *il existe une unique boucle allant de q vers q n'ayant pas q comme état intermédiaire ;*

   (ii) *il existe un nombre fini de chemins de l'état initial vers q qui n'ont pas q comme état intermédiaire ;*

   (iii) *il existe un nombre fini de chemins de q vers un état final qui n'ont pas q comme état intermédiaire.*
4. *Soit L un langage fin et $\mathcal{A}$ un automate fini déterministe acceptant L. Montrer que $\mathcal{A}$ est monoboucle.*
5. *Terminer la preuve du théorème.*
6. *Donner un algorithme qui reçoit en entrée un automate fini déterministe $\mathcal{A}$ et qui renvoie vrai si le langage reconnu par $\mathcal{A}$ est fin et faux sinon.*

1.15 Satisfiabilité

Exercice 121

1. *Écrire le type* `expression` *correspondant aux expressions construites à partir de variables implémentées par des chaînes de caractères.*

   On implémente dorénavant une distribution de vérité avec une liste de couples `string*bool`.
2. *Donner une fonction* `evaluation` *qui prend une expression et une distribution de vérité, puis qui renvoie l'évaluation correspondante.*
3. *Écrire une fonction* `variables` *qui renvoie la liste des variables d'une expression.*
4. *Écrire une fonction* `verite` *qui associe à une liste de variables (donc de type* `string list`*) la liste des expressions correspondantes.*
5. *Tester si une expression donnée est une tautologie. Quelle est la complexité de cette fonction ?*

Exercice 122

Soit v_1, v_2 et v_3 des variables. Donner les formes normales conjonctive et disjonctive de

$$f = \neg(v_1 \wedge \neg(v_2 \vee v_3)).$$

Exercice 123

Une clause est une disjonction de littéraux (variables ou négation de variables) portant sur des variables différentes ; dans cet exercice, une forme normale conjonctive (FNC) est un ensemble de clauses.
Pour toute FNC S et toute variable x, on définit $S[\mathbf{V}/x]$ (respectivement $S[\mathbf{F}/x]$) la FNC déduite de S en remplaçant chaque clause $C \vee x$ par la constante $\mathbf{V}$ (respectivement C) et $C \vee \neg x$ par C (respectivement la constante $\mathbf{F}$).

1. *Soit S une FNC et x une variable. Montrer que la variable x n'apparaît ni dans $S[\mathbf{V}/x]$ ni dans $S[\mathbf{F}/x]$.*
2. *Soit S une FNC, x une variable et δ une distribution de vérité. Montrer que*
   (a) *si $\delta(x) = 1$, alors $\delta(S) = \delta(S[\mathbf{V}/x])$;*
   (b) *si $\delta(x) = 0$, alors $\delta(S) = \delta(S[\mathbf{F}/x])$.*
3. *Soit S une FNC contenant la clause réduite à la variable x. Montrer que S est satisfiable si, et seulement si, $S[\mathbf{V}/x]$ est satisfiable. Énoncer un résultat analogue pour une FNC S contenant la clause $\neg x$.*
   Un littéral est pur dans une FNC S si sa négation n'apparaît pas dans S. Une clause est pure dans la FNC S lorsqu'elle fait contient un littéral pur dans S.
4. *Soit S une FNC et C une clause pure dans S. Montrer que S est satisfiable si, et seulement si, $S \setminus \{C\}$ est satisfiable.*
5. *Soit x un littéral qui n'est pas pur dans S. Construisons S' en remplaçant pour tout couple formé de clauses $C_1 \vee x$ et $C_2 \vee \neg x$ de S par*
   (a) *$\mathbf{V}$ si C_1 et C_2 contiennent un littéral et sa négation,*
   (b) *$C_1 \vee C_2$ sinon.*
   Montrer que S est satisfiable si, et seulement si, S' l'est.
6. *En déduire un algorithme pour tester la satisfiabilité d'une FNC.*

Exercice 124 (*interpolation*)

1. *Soit F, G et H des formules logiques. Montrer que*
   (a) *$F \Rightarrow (\neg H \Rightarrow \neg G)$ est une tautologie si, et seulement si, $G \Rightarrow (F \Rightarrow H)$ en est une.*
   (b) *$(F \Rightarrow G) \wedge (F \Rightarrow H)$ est une tautologie si, et seulement si, $F \Rightarrow (G \wedge H)$ en est une.*
   Soit F, G des formules et y une variable n'apparaissant pas dans G. La formule $F[G/y]$ est déduite de F en remplaçant chaque occurrence de y par la formule G.
2. *Soit δ une distribution de vérité. Définissons δ' par*

$$\forall x \in \mathcal{V}, \qquad \delta'(x) = \begin{cases} \delta(G) & \text{si } x = y \\ \delta(x) & \text{sinon} \end{cases}$$

   Montrer que $\delta(F[G/y]) = \delta'(F)$.
   Soit F, G_1, ..., G_p des formules et y_1, ..., y_p des variables n'apparaissant pas dans G_1, ..., G_p. La formule $F[G_1/y_1, \dots, G_p/y_p]$ est déduite de F en remplaçant, pour tout i, chaque occurrence de y_i par la formule G_i.
3. *Montrer que si F est une tautologie, alors $F[G_1/y_1, \dots, G_p/y_p]$ est une tautologie.*
4. *Soit deux formules logiques F et G d'ensembles de variables disjoints. Montrer que $F \Rightarrow G$ est une tautologie si, et seulement si, l'une (au moins) des formules $\neg F$ ou G est une tautologie.*
   On se propose dans la suite de l'exercice d'établir le théorème suivant PropositionSoit F, G deux formules logiques et x_1, ..., x_n les variables intervenant dans F et G. Les deux propositions suivantes sont équivalentes :
   (a) *$F \Rightarrow G$ est une tautologie ;*
   (b) *il existe une formule H d'ensemble de variables inclus dans $\{x_1, \dots, x_n\}$ telle que $F \Rightarrow H$ et $H \Rightarrow G$ sont des tautologies.*
5. *Justifier le sens retour.*
6. *Montrer le sens direct. On pourra commencer par le cas où l'ensemble des variables de F est inclus dans l'ensemble des variables de G.*

Exercice 125

On se propose de démontrer le théorème suivant (de compacité du calcul des propositions). Soit $\mathfrak{I}$ un ensemble de formules propositionnelles sur un ensemble dénombrable de variables. Les deux propositions suivantes sont équivalentes :

i. *$\mathfrak{I}$ est satisfiable (c'est-à-dire, il existe une valuation qui satisfait toutes les formules de $\mathfrak{I}$) ;*
ii. *toute partie finie de $\mathfrak{I}$ est satisfiable.*

1. *Justifier l'implication $i \Rightarrow ii$.*
   Soit $(x_i)_i$ une énumération des variables propositionnelles et σ une valuation sur la partie finie $(x_i)_{i \leq n}$. La valuation σ est une valuation finie adaptée à $\mathfrak{I}$ si toute partie finie $\mathfrak{F} \subset \mathfrak{I}$ est satisfiable par un prolongement de σ à l'ensemble des variables propositionnelles apparaissant dans $\mathfrak{F}$.
2. *Montrer que si toute partie finie de $\mathfrak{I}$ est satisfiable, alors toute valua-*

tion finie adaptée à $\mathfrak{I}$ définie sur n variables propositionnelles peut se prolonger en une valuation finie adaptée à $\mathfrak{I}$ définie sur $n+1$ variables propositionnelles.

3. *Démontrer le théorème de compacité.*

Exercice 126

Un circuit logique est un graphe orienté sans cycle dont

- ▷ *les sommets de degré entrant nul sont appelés entrées,*
- ▷ *les sommets de degré sortant nul sont de degré entrant 1 et sont appelés sorties,*
- ▷ *les autres sommets sont des portes, qui peuvent être de type `NON`, `ET` ou `OU` ; pour ces deux dernières, on suppose que le degré entrant est au moins 2.*

Montrer qu'il existe un circuit avec seulement deux sommets `NON` qui prend 3 entrées booléennes a, b et c et dont les sorties sont leurs négations $\neg a$, $\neg b$ et $\neg c$. Est-il possible de faire un tel circuit avec une seul sommet `NON` ?

Vrai ou Faux de révision

1. La recherche dichotomique dans une liste triée est de complexité logarithmique.
2. Le recherche du plus petit et du plus grand élément d'une liste est de complexité linéaire.
3. La complexité du tri rapide est quadratique.
4. La recherche du plus petit élément dans un tas-max est de complexité constante.
5. La recherche du plus grand élément dans un tas-max est de complexité constante.
6. L'insertion d'un élément dans un tas-max est de complexité constante.
7. La recherche d'un élément dans un arbre binaire de recherche est de complexité logarithmique en le nombre de nœuds.
8. Pour une suite telle que $u_{n+1} = 2u_n + n$, on a $u_n = \Theta(n^2)$.
9. Après une étape du tri par insertion, le plus petit élément est bien placé.
10. Un arbre binaire enraciné à n nœuds internes admet n feuilles.
11. Un arbre binaire enraciné à n feuilles admet $n+1$ nœuds internes.
12. Un arbre binaire enraciné à n nœuds est de profondeur $\lfloor n \rfloor$.
13. Le nombre d'arbres binaires à n nœuds internes est C_n (n-ième nombre de Catalan).
14. La hauteur d'un arbre enraciné est le nombre d'arêtes entre la racine et la feuille la plus profonde.
15. Soit x et y deux nœuds d'un arbre binaire. Si x est un ancêtre de y alors x est avant y dans le parcours en profondeur préfixe.
16. Dans un parcours en profondeur suffixe d'un arbre, la racine est le dernier nœud traité.
17. La racine d'un arbre binaire de recherche est la médiane de l'ensemble des étiquettes de l'arbre.
18. Un arbre à n sommets admet $n-1$ arêtes.
19. Un graphe à n sommets et $n-1$ arêtes est un arbre.
20. Un graphe à n sommets et $n-k$ arêtes admet au plus k composantes connexes.
21. Un graphe connexe sans cycle est un arbre.
22. Un graphe d'arité maximale 2 est un arbre binaire.
23. Un graphe connexe d'arité maximale 2 est un arbre binaire.
24. Un graphe non orienté admet un arbre couvrant.
25. Un graphe non orienté à s sommets et a arêtes tel que $s \geq \binom{a-1}{2}$ est connexe.
26. Un graphe orienté sans cycle admet au moins un sommet de degré entrant nul.
27. Un graphe orienté sans cycle admet au moins un sommet de degré sortant nul.
28. Un cycle est 2-coloriable.
29. Si A est la matrice d'adjacence d'un graphe, $[(I_n + A)^m]_{i,j}$ est le nombre de chemins de longueur m reliant le sommet i à j.
30. Si A est la matrice d'adjacence d'un graphe, $[A^m]_{i,j}$ est le nombre de chemins de longueur m reliant le sommet i à j.
31. Si A est la matrice d'adjacence d'un graphe, $\mathrm{tr}(A^3)$ est le nombre de chemins de longueur 3 dans le graphe.
32. Le parcours en largeur d'un graphe est basé sur la structure de pile.
33. Le parcours en profondeur d'un graphe orienté depuis un sommet s donne la composante fortement connexe de s.
34. Le dernier sommet obtenu par un parcours en largeur d'un graphe depuis un sommet s est le sommet le plus loin de s.
35. L'algorithme de Dijkstra permet de calculer toutes les distances entre sommets d'un graphe ordonné.
36. L'algorithme de Dijkstra à partir d'un sommet s pour un graphe sans cycle absorbant et tel que tous les arcs issus d'un sommet autre que s aient une poids positif renvoie un résultat correct.
37. L'algorithme de Floyd-Warshall sur les graphes relève de la programmation dynamique.
38. Un graphe admet un arbre couvrant si, et seulement si, il est connexe.
39. Un arbre couvrant comporte toujours l'arête de poids minimal.

A computer lets you make more mistakes faster than any invention in human history.

40. Si u est à la fois un préfixe et un suffixe de m, alors $|m| \geq 2|u|$.
41. Un mot m admet $|m|+1$ préfixes.
42. Si un mot u est plus petit qu'un mot v pour l'ordre lexicographique, alors u est un préfixe de v.
43. La réunion de deux expressions régulières est une expression régulière.
44. Si e est une expression régulière, $e\star$ en est aussi une.
45. Tout langage reconnaissable est reconnaissable par un automate standardisé.
46. Tout langage reconnaissable est reconnaissable par un automate émondé.
47. Tout langage reconnaissable est reconnaissable par un automate local.
48. Tout langage reconnaissable est reconnaissable par un automate déterministe.
49. Tout langage local est reconnaissable.
50. Le langage denoté par une expression rationnelle linéaire est local.
51. Le langage denoté par une expression rationnelle est local.
52. Le langage a^*bc^* est local.
53. Le langage a^*ba^* est local.
54. Le langage $a^*b \cup ba^*$ est local.
55. Un langage fini est local.
56. La réunion de deux langages locaux est locale.
57. L'intersection de deux langages locaux est locale.
58. Si L est reconnu par l'automate déterministe (Q, q_0, F, δ), alors le complémentaire de L est reconnu par $(Q, q_0, Q \setminus F, \delta)$.
59. Si L est reconnu par l'automate déterministe (Q, q_0, F, δ), alors le langage des préfixes de L est reconnu par (Q, q_0, Q, δ).
60. Si L est reconnu par l'automate déterministe (Q, q_0, F, δ), alors le langage des suffixes de L est reconnu par (Q, Q, F, δ).
61. La différence symétrique de deux langages reconnaissables est reconnaissable.
62. Le langage des palindromes est reconnaissable.
63. Le langage des mots de longueur pair est reconnaissable.
64. L'automate obtenu par déterminisation avec l'algorithme des parties est standard.
65. Un automate déterministe qui reconnaît un langage dont le mot non vide le plus court est de longueur k admet au moins k états.
66. Le parcours en profondeur suffixe de l'arbre d'une formule logique donne sont écriture polonaise inverse.
67. L'opérateur ou exclusif `XOR` est associatif.
68. Le nombre de distributions de vérité/valuations sur n variables est 2^n.
69. La conjonction de deux formules satisfiables est satisfiable.
70. Une disjonction de formules satisfiables est satisfiable.
71. La négation d'une formule satisfiable est satisfiable.
72. Le nombre de termes de la forme normale disjonctive de f est égal au nombre de distributions de vérité satisfaisant f.

Computers are like Old Testament gods; lots of rules and no mercy.

Aide-mémoire

Cet aide-mémoire non exhaustif indique les fonctions à connaître impérativement dès le début de l'année.

Affectation	
`let x = a`	affectation de `x` à la valeur `a`
`let x = a in`	affectation locale de `x` à la valeur `a`

Booléen	
`true`	valeur vrai
`false`	valeur faux
`&&`	et logique
`||`	ou logique
`not`	négation

Entier	
`min_int`	plus petit entier Caml
`max_int`	plus grand entier Caml
`a + b`	somme de `a` et `b`
`a - b`	différence de `a` et `b`
`a * b`	produit de `a` et `b`
`a / b`	quotient dans la division euclidienne de `a` par `b`
`a mod b`	reste dans la division euclidienne de `a` par `b`
`abs a`	valeur absolue de `a`
`min a b`	minimum de `a` et `b`
`max a b`	maximum de `a` et `b`
`a < b`	comparaison stricte de `a` et `b`
`a <= b`	comparaison large de `a` et `b`

Flottant	
`6.02e23`	écriture d'un flottant
`a +. b`	somme de `a` et `b`
`a -. b`	différence de `a` et `b`
`a *. b`	produit de `a` et `b`
`a /. b`	quotient de `a` par `b`
`a ** b`	exponentiation de `a` à la puissance `b`
`abs_float a`	valeur absolue de `a`
`min a b`	minimum de `a` et `b`
`max a b`	maximum de `a` et `b`
`a <. b`	comparaison stricte de `a` et `b`
`a <=. b`	comparaison large de `a` et `b`
`sqrt a`	racine de `a`
`log a`	logarithme népérien de `a`
`exp a`	exponentielle de `a`
`sin a`	sinus de `a`
`asin a`	arcsinus de `a`
`power a b`	`a` élevé à la puissance `b`

Liste	
`[ ]`	liste vide
`[x1; x2; x3 ]`	liste composée de `x1`, `x2` et `x3`
`t::q`	liste constituée de l'élément `t` puis de la liste `q`
`map f l`	liste constituée des éléments `f`(x) pour x dans `l`
`hd l`	tête de la liste `l`
`tl l`	queue de la liste `l`
`l @ l'`	concaténée de `l` et `l'`
`list_length l`	longueur de `l`
`rev l`	liste renversée de `l`
`mem x l`	test de l'appartenance de l'élément `x` à la liste `l`
`index x l`	premier indice d'apparition de l'élément `x` dans la liste `l`

Tableau, vecteur	
`make_vect n x`	tableau de longueur `n` avec tous les éléments égaux à `x`
`init_vect n f`	tableau de longueur `n` avec l'élément d'indice k égal à `f`(k)
`copy_vect t`	tableau copie de `t`
`t.(n)`	`n`-ième élément du tableau `t`
`t.(n) <- x`	affectation à `x` du `n`-ième élément du tableau `t`
`vect_length t`	longueur du tableau `t`

Chaîne	
`'a'`	caractère `a`
`"abc"`	chaîne composée des caractères `a`, `b` et `c`
`s.[n]`	`n`-ième caractère de la chaîne `s`
`s.[n] <- x`	affectation à `x` du `n`-ième caractère de la chaîne `s`
`string_length s`	longueur de la chaîne `s`

Enregistrement	
`type comp = {mutable a: t1; b: t2};;`	création d'un type enregistrement avec des champs `a` et `b` de type `t1` et `t2`, le premier étant modifiable
`let x = {a=v1; b=v2}`	création d'un enregistrement `x`
`x.a`	champs `a` de l'enregistrement `x`
`x.a <- v`	affectation de la valeur `v` dans le champs `a` de l'enregistrement `x`

Couple	
`a,b`	couple `(a,b)`
`fst (a,b)`	renvoie de la première coordonnée `a`
`snd (a,b)`	renvoie de la seconde coordonnée `b`

Type	
`type` *nom* `=` *def*	déclaration du type *nom*
`type` *nom* `==` *def*	abréviation de type pour une définition préexistente
`type` *nom* `= { x : string; y: int}`	création d'un enregistrement avec deux champs `x` et `y` de type `string` et `int`
`type` *nom* `= { mutable y: int}`	création d'un enregistrement avec un champ mutable `y` de type `int`

Fonction	
`let f = function x -> expr;;`	fonction `f` qui associe `expr` à `x`
`let f x = expr;;`	fonction `f` qui associe `expr` à `x`

Filtrage	
`   | x when x = y ->`	test dans filtrage

`match expr with`
`   | motif1 -> x1`
`   | motif2 -> x2`
	filtrage selon deux motifs pour `expr`

`function`
`   | motif1 -> x1`
`   | motif2 -> x2`
	filtrage selon deux motifs (argument non précisé)

`fun`
`   | a1 b1 -> x1`
`   | a2 b2 -> x2`
	filtrage selon deux motifs (arguments non précisés)

Exception	
`exception e`	création de l'exception `e`
`exception e of int`	création de l'exception `e` paramétrée par un entier
`raise e`	déclenchement de l'exception `e`
`failwith s`	déclenchement de l'exception commentée par la chaîne `s`
`try expr with e -> v`	rattrapage de l'exception `e` avec la valeur `v`

Effet de bord	
`print_string s`	affichage de la chaîne `s`
`print_int n`	affichage de l'entier `n`
`print_float x`	affichage du flottant `x`
`print_newline`	saut de ligne dans l'affichage

Référence	
`let x = ref n in`	création d'une référence `x` initialisée à `n`
`!x`	valeur de la référence `x`
`x := a`	affectation de la valeur `a` dans la référence `x`
`incr x`	incrémentation de la référence entière `x`
`decr x`	décrémentation de la référence entière `x`

Computers are like Old Testament gods; lots of rules and no mercy.

Boucle	

`for k = 1 to n do`
`    expr1; expr2`
`done;`
	boucle avec `k` croissant de `1` à `n`

`for k = n downto 1 do`
`    expr1; expr2`
`done;`
	boucle avec `k` décroissant de `n` à `1`

Structure conditionnelle	
`if bool then expr1` `else expr2`	structure conditionnelle entre deux expressions `expr1` et `expr2` (de même type) selon la valeur du booléen `bool`

`begin`
`    expr1; expr2`
`end`
	bloc d'instructions