# RAISONNEMENTS

## Récurrence sur $\mathbb{N}$

— Expliciter la propriété $\mathcal{P}_n$ pour bien comprendre ce que l'on veut démontrer.

— Annoncer le type de récurrence utilisé : simple, double ou forte.

**(Initialisation)** par exemple, montrer que $\mathcal{P}_0$ est vraie.

**(Hérédité)** :

— Expliciter l'hypothèse de récurrence pour $n$ (simple), pour $n$ et $n-1$ (double) ou jusqu'à $n$ (forte).

— Montrer que $\mathcal{P}_{n+1}$ est vraie en utilisant l'hypothèse de récurrence.

**(Conclusion)** Appliquer le principe de récurrence pour montrer que $\mathcal{P}_n$ est vraie pour tout entier naturel $n$.

(R) Attention à bien montrer que $\mathcal{P}_{n+1}$ est vraie. En effet, $n+1$ est le successeur de $n$ dans $\mathbb{N}$ : l'opérateur successeur est le constructeur de l'ensemble $\mathbb{N}$ et 0 le case de base.

**Application à l'étude d'une suite.** La suite de Thue-Morse, notée $(t_n)$ est définie par :

$$t_0 = 0 \tag{1}$$
$$t_{2n} = t_n \tag{2}$$
$$t_{2n+1} = 1 - t_n \tag{3}$$

Montrer que les termes de la suite de Thue-Morse ne peuvent valoir que 0 ou de 1.

*Démonstration.* Par récurrence forte sur l'entier $n$.

La propriété à démontrée est $\mathcal{P}_n$ : « $t_n$ est un élément de $\{0,1\}$. ».

**(Initialisation)** $\mathcal{P}_0$ est vraie car $t_0$ vaut 0 par définition de la suite.

**(Hérédité)** Supposons que $\mathcal{P}_n$ est vraie pour tout entier naturel inférieur ou égal à $n$. On considère $t_{n+1}$ et on procède alors par disjonction des cas :

— Si $n+1$ est pair, alors il existe un entier $k$ plus petit que $n$ tel que $t_{n+1} = t_{2k} = t_k$ et $t_k \in \{0,1\}$ par hypothèse de réccurence.

— Si $n+1$ est impair, alors il existe un entier $k$ plus petit que $n$ tel que $t_{n+1} = t_{2k+1} = 1 - t_k$ et $t_k \in \{0,1\}$ par hypothèse de réccurence.

Dans les deux cas, la propriété est héréditaire.

**(Conclusion)** Comme $\mathcal{P}_0$ est vraie et que la propriété est héréditaire, $\mathcal{P}_n$ est vraie pour tout entier naturel.

■

## Induction structurelle

L'induction structurelle est l'**extension du principe de récurrence à tout ensemble construit de manière inductive**. Pour utiliser cette méthode, il est **nécessaire** d'avoir défini une structure inductivement.

**Définition inductive d'un arbre binaire.** Soit $E$ un ensemble d'étiquettes. L'ensemble $\mathcal{A}_E$ des arbres binaires étiquetés par $E$ est défini inductivement par :

a. VIDE est un arbre binaire appelé arbre vide (parfois noté $\emptyset$),

b. Constructeur NŒUD : si $e \in E$, $f_g \in \mathcal{A}_E$ et $f_d \in \mathcal{A}_E$ sont deux arbres binaires, alors $\text{NŒUD}(f_g, e, f_d) \in \mathcal{A}_E$, c'est à dire que $\text{NŒUD}(f_g, e, f_d)$ est un arbre binaire étiqueté par $E$.

$f_g$ et $f_d$ sont respectivement appelés fils gauche et fils droit.

**Démonstration par induction structurelle sur un arbre binaire.** Soit $\mathcal{P}(a)$ un prédicat exprimant une propriété sur un arbre $a$ de $\mathcal{A}_E$. On souhaite montrer que cette propriété est vraie pour tous les arbres de $\mathcal{A}_E$.

La démonstration par induction structurelle procède comme suit :

a. (**CAS DE BASE**) Montrer que $\mathcal{P}(\text{VIDE})$ est vraie, c'est-à-dire que la propriété est vraie pour l'arbre vide,

b. (**PAS D'INDUCTION (Constructeur `Noeud`)**) Soit $e \in E$ une étiquette et $f_g \in \mathcal{A}_E$ et $f_d \in \mathcal{A}_E$ deux arbres binaires pour lesquels $\mathcal{P}(f_g)$ et $\mathcal{P}(f_d)$ sont vraies. Montrer que $\mathcal{P}\big(\text{NŒUD}(f_g, e, f_d)\big)$ est vraie.

c. (**CONCLUSION**) Conclure que quelque soit $a \in \mathcal{A}_E$, comme la propriété est vérifiée pour le cas de base et que le constructeur conserve propriété, $\mathcal{P}(a)$ est vraie.

## Par contraposition

## Par l'absurde

## Par double inclusion

## Par double implication

# ANALYSE D'ALGORITHMES

## Terminaison

Montrer la terminaison d'une fonction :

**Pour une fonction récursive :** montrer que les paramètres effectifs des appels récursifs forment une suite strictement décroissante au sens d'un **ordre bien fondé**.
Par exemple :

- un entier positif strictement plus petit,
- le fils gauche ou droit d'un arbre binaire,
- la queue d'une liste inductive,
- un couple plus petit selon l'ordre lexicographique.

**Pour une fonction impérative itérative :** on distingue la boucle `while` et `for`.

- pour une boucle `while`, on exhibe un **variant de boucle**, c'est-à-dire une quantité qui diminue strictement, au sens d'un ordre bien fondé. La plupart du temps, le variant peut être un entier.
- une boucle `for i = 0 to n`, dont la plage d'indices est définie explicitement, se termine toujours si le corps de la boucle termine. Un variant naturel est `n - i`.

## Correction

Pour montrer qu'une fonction est correcte :

**Pour une fonction récursive :** prouver la correction par induction structurelle ou récurrence.

**Pour une fonction impérative et itérative :** exhiber un **invariant de boucle**, c'est-à-dire une propriété qui est vraie avant la boucle et qui n'est pas modifiée par les instructions du corps de la boucle et qui donc sera vraie après la boucle.

On peut démontrer la correction partielle d'un algorithme en supposant qu'il termine.

## Complexité

On distingue :

**la complexité temporelle** c'est-à-dire comment évolue le temps de calcul en fonction des parmètres de l'algorithme.

**la complextié spatiale** c'est-à-dire comment évolue l'espace mémoire nécessaire au calcul en fonction des paramètres de l'algorithme. On ne compte généralement pas les paramètres d'une fonction dans le calcul de la complexité spatiale.

**Notation de Landau :**

- $f(n) = \mathcal{O}(g(n))$ signifie que $f$ est asymptotiquement majorée par une constante fois $g$.
- $f(n) = \Omega(g(n))$ signifie que $f$ est asymptotiquement minorée par une constante fois $g$.
- $f(n) = \Theta(g(n))$ signifie que $f$ et $g$ sont asymptotiquement du même ordre de grandeur.

On distingue la complexité :

- dans le pire des cas,
- dans le meilleur des cas,
- moyenne, la moyenne sur tous les paramètres d'entrée possibles,
- amortie, une complextié calculée sur un grand nombre d'opérations successives.

**Méthode 1 — Complexité d'un algorithme** Pour trouver la complexité d'un agorithme :

a. Trouver le(s) paramètre(s) d'entrée qui influe(nt) sur la complexité.
b. Déterminer si, une fois ce(s) paramètre(s) fixé(s), il existe un pire ou un meilleur des cas.
c. Compter le nombre d'opérations élémentaires effectuées par l'algorithme :
   - en calculant éventuellement une somme (fonction itérative),
   - en posant une formule récurrente sur la complexité (fonction récursive).

Par exemple, pour une fonction itérative, `for i=0 to n − 1 do for j=i+1 to n−1 do` on peut écrire :

$$\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \mathrm{C}_{\text{boucle}}$$

Il faut alors tenir compte de la complexité des instructions du corps de la boucle $\mathrm{C}_{\text{boucle}}$. Si celle-ci est constante $O(1)$, alors on aura :

$$\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} n-i-1 = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} = O(n^2)$$

Le tableau 1 récapitule les complexités des algorithmes récursifs à connaître.

| **Récurrence** | **Complexité** | **Algorithmes** |
|---|---|---|
| $T(n) = 1 + T(n-1)$ | $\to O(n)$ | factorielle |
| $T(n) = n + T(n-1)$ | $\to O(n)$ | fusion de deux listes triées |
| $T(n) = 1 + T(n/2)$ | $\to O(\log n)$ | dichotomie, exponentiation rapide |
| $T(n) = n + 2T(n/2)$ | $\to O(n \log n)$ | tri fusion |

TABLE 1 – Récurrences et complexités associées utiles et à connaître

# REPRÉSENTATION DES NOMBRES

La décomposition d'un **entier** sur une base est **unique**.

$$198_{10} = 1\times10^2 + 9\times10^1 + 8\times10^0 = 100+90+8$$
$$198_{10} = 11000110_2 = 1\times2^7 + 1\times2^6 + 1\times2^2 + 1\times2^1 = 128+64+4+2$$
$$198_{16} = C6_{16} = C\times16^1 + 6\times16^0 = 12\times16+6$$

En base $b$, on peut représenter $b^n$ nombres avec $n$ chiffres. Par exemple, avec 3 chiffres en base 10, on peut compter de 0 à 999, c'est-à-dire $10^3$ nombres.

En base 2, on peut donc représenter $2^n$ **nombres avec $n$ bits**. Pour encoder 256 nombres entiers de 0 à 255, 8 bits suffisent.

En **binaire**, en base 2, les chiffres sont 0 ou 1. Un **octet** est composé de 8 bits.

Un entier **non signé** est représenté en machine en binaire sur un certain nombre de bits. Si $n = \sum_{i=0}^{m} b_i 2^i$ avec les $b_i \in \{0,1\}$, alors $n$ sera représenté en $b_m b_{m-1} \dots b_1 b_0$ sur $m$ bits. Cette représentation peut entraîner un **dépassement de capacité** : par exemple si choisit de coder les entiers non signés sur 8 bits et qu'on cherche à mémoriser un nombre plus grand que 255 ou qu'on opère une addition telle que $200+134$.

Un entier **signé** $p \in [\![-2^{n-1}, 2^{n-1}-1]\!]$ est représenté en complément à $2^n$ sur $n$ bits . Cela revient à représenter $p$ par une entier non signé sur $n-1$ bits si $p \geqslant 0$ et par $2^n+p$ si $p<0$. Par exemple, pour écrire le nombre $-67_{10}$ sur 8 bits en complément à $2^8$, on calcule $2^8-67 = 189$ qui s'écrit $10111101_2$. Avec ce système, 67 s'écrit $01000011_2$.

Un nombre est dit dyadique s'il peut s'écrire comme un entier divisé par une puissance de deux. Les **flottants** représentables en machine avec les normes usuelles sont tous **dyadiques**. La représentation machine d'un flottant $x$ :

- d'un bit de signe $s$,
- d'un exposant biaisé $E$,
- et d'une pseudo-mantisse $M$ : $\pm 1, M.2^e$.

C'est pourquoi il est codé en machine par `s E M`.

- En simple précision (32 bits), 6 chiffres significatifs en base 10.
- En double précision (64 bits), 15 chiffres significatifs en base 10.

Pour faire les opérations sur les flottants (addition, multiplication), une mise à sur la même l'échelle des puissances est opérée ce qui peut dégrader la précision du calcul. Ce mécanisme est nommé **mécanisme d'absorption**.

# MÉMOIRE

## Références et portée des variables

Une **variable** lie :

— un identifiant, c'est-à-dire son nom,

— un objet, c'est-à-dire sa valeur,

— et un type.

La **portée d'une variable** commence là où le nom est défini et s'arrête :

— à la fin du programme pour une variable globale,

— ou à la fin du bloc (expression, structure de contrôle, structure alternative, fonction) pour une variable locale.

En Python, il faut considéré les variables comme des références vers des objets, des adresses mémoires vers un objet.

En OCaml, on distingue les variables qui sont immuables et des références qui sont muables, c'est-à-dire dont on peut faire évoluer la valeur.

Une copie superficielle d'une variable consiste à recopier l'adresse mémoire, ce qui crée une liaison entre les variables. Par exemple, `L = L1` en Python.

Une **copie en profondeur** d'une variable recrée un autre objet en mémoire, identique à celui pointé par la variable copiée et l'attribue à la nouvelle variable. Par exemple, `L = [e for e in L1]` en Python.

## Type abstrait et implémentation

On distingue le type abstrait et la structure de données (son implémentation) :

a. le type abstrait est une **description théorique** de ce que doit contenir la structure de données, ainsi que les opérations qu'on souhaite effectuer dessus.

b. la structure de données correspond à la **réalisation effective** de la structure de données, c'est-à-dire la manière de la représenter en machine, ainsi que la description des algorithmes associés à chacune des opérations, et leurs complexités.

## Modèle mémoire lors de l'exécution d'un programme

On distingue trois zones mémoire dans l'exécution d'un programme. La durée de vie d'un objet dépend de la manière dont il a été alloué en mémoire.

a. la zone statique, où sont écrites les variables globales,

b. la pile, où sont écrites les variables locales,

c. le tas (qui n'est pas un tas au sens arborescent), où est allouée dynamiquement la mémoire.

La zone statique existe pendant toute la durée de vie du programme.

La taille de la pile est limitée. Il peut donc se produire un **débordement de pile** (stack overflow) si la taille limite est dépassée, par exemple lors d'un trop grand nombre d'appels récursifs imbriqués.

En OCaml et en Python, le GC (Garbage Collector) gère l'allocation mémoire automatiquement.

# ARBRES ENRACINÉS

Un arbre enraciné est un ensemble muni d'une opération de parenté vérifiant :

- qu'il existe un unique nœud sans parent, appelé racine,
- tout nœud non racine possède un unique parent,
- pour tout nœud, il existe une unique suite d'aïeux menant à la racine, chaque nœud étant le parent du nœud précédent sur le chemin,
- les enfants d'un nœud peuvent être ordonnés ou non.

## Arbre binaire

**Définition inductive d'un arbre binaire.** Soit $E$ un ensemble d'étiquettes. L'ensemble $\mathcal{A}_E$ des arbres binaires étiquetés par $E$ est défini inductivement par :

- VIDE est un arbre binaire appelé arbre vide (parfois noté $\emptyset$),
- Si $e \in E$, $f_g \in \mathcal{A}$ et $f_d \in \mathcal{A}$ sont deux arbres binaires, alors $(f_g, e, f_d) \in \mathcal{A}_E$, c'est à dire que le triplet $(f_g, e, f_d)$ est un arbre binaire étiqueté par $E$.

$f_g$ et $f_d$ sont respectivement appelés fils gauche et fils droit.

**Par convention, l'arbre vide est de taille 0 et de hauteur - 1.** Pour un arbre **binaire** $a = (g, e, d)$, on définit alors la taille et la hauteur inductivement :

- $|a| = |g| + |d| + 1$
- $h(a) = \max(h(g), h(d)) + 1$.

Un arbre **binaire** $a$ vérifie $h(a) + 1 \leqslant |a| \leqslant 2^{h(a)+1} - 1$.
Le parcours en profondeur parcourt inductivement de gauche à droite les enfants d'un nœud :

**parcours préfixe** : la racine est placée avant les parcours des enfants,

**parcours infixe** (cas binaire uniquement) : la racine est placée entre les parcours des deux enfants,

**parcours postfixe** : la racine est placée après les parcours des enfants.

Le parcours en largeur parcourt les nœuds de gauche à droite, par profondeur croissante.

## Arbres binaires de recherche

**Arbre binaire de recherche.** Soit un ensemble d'étiquettes $\mathcal{E}$ muni d'un ordre total. Un arbre binaire de recherche est un arbre binaire étiqueté par $\mathcal{E}$ dont les nœuds $N(e, g, d)$ vérifient la propriété suivante :

- l'élément $e$ est plus grand que toutes les étiquettes du sous-arbre gauche $g$,
- l'élément $e$ est plus petit que toutes les étiquettes du sous-arbre droit $d$.

Les inégalités peuvent être strictes selon les définitions.

**Un arbre est ABR si et seulement si son parcours infixe est croissant.**

Les principales **opérations** sur les arbres de recherche sont :

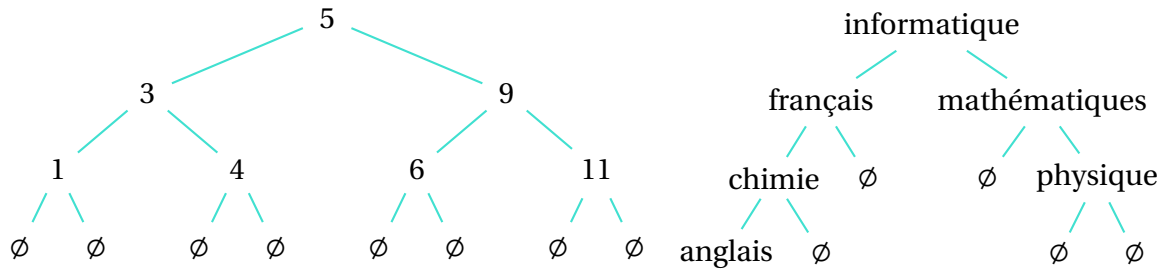



FIGURE 1 – Arbres binaires de recherche équilibré (à gauche) et non équilibré (à droite)

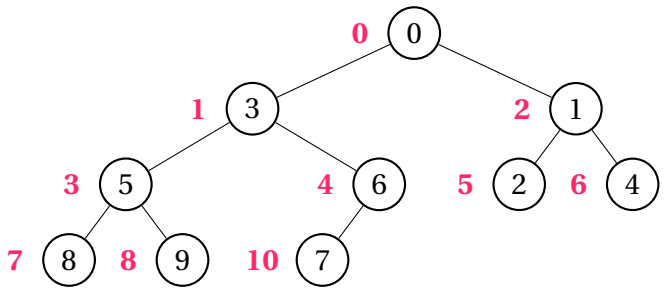
— la recherche d'un élément, opération de complexité $\mathcal{O}(h)$,
— l'insertion d'un élément, opération de complexité $\mathcal{O}(h)$,
— la suppression d'un élément $\mathcal{O}(h)$.

Les arbres rouge et noir (ou bicolores) et les arbres AVL permettent d'améliorer la complexité dans le pire cas de ces opérations, en garantissant l'équilibre de l'arbre et donc que $h = \mathcal{O}(\log n)$.

## Tas

**Tas max et tas min.** On appelle tas max (resp. tas min) un arbre binaire parfait étiqueté par un ensemble ordonné $E$ tel que l'étiquette de chaque nœud soit inférieure (resp. supérieure) ou égale à l'étiquette de son père. La racine est ainsi la valeur maximale (resp. minimale) du tas.

On peut implémenter un tas binaire de taille bornée dans un tableau :




| **Indices** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **Valeurs** | 0 | 3 | 1 | 5 | 6 | 2 | 4 | 8 | 9 | 7 |

FIGURE 2 – Implémentation d'un tas min par un tableau. **On vérifie que les fils du nœud à l'indice $k$ se trouvent à l'indice $2k+1$ et $2k+2$**

Dans un tas de taille $n$, **la première feuille se situe à l'indice $\lfloor n/2 \rfloor$.**
La hauteur d'un tas est $h = \mathcal{O}(\log n)$ car c'est un arbre binaire parfait.
On définit des opérations **descendre** et **faire monter** un élément dans un tas qui **préservent** la structure du tas. Ce sont des opérations dont la complexité est $\mathcal{O}(h(a)) = \mathcal{O}(\log n)$.
**Faire monter** un nœud dans le tas consiste à l'échanger avec son parent tant que l'ordre n'est pas respecté.
**Faire descendre** un nœud dans le tas consiste à l'échanger avec son plus grand enfant tant que l'ordre n'est pas respecté.
Pour **insérer un élément**, on l'ajoute à la première position de feuille disponible (celle d'indice $n$ pour un tas de taille $n$ ), puis on le fait remonter. La complexité est $\mathcal{O}(\log n)$.
Pour **extraire le plus grand élément (tas max)**, qui se trouve à la racine, on l'échange avec la dernière feuille (celle d'indice $n-1$ ), puis on fait descendre la nouvelle racine. La complexité est $\mathcal{O}(\log n)$.
**Pour construire un tas**, il faut transformer un tableau de taille $n$ en tas. On fait descendre les $\lfloor n/2 \rfloor$ premiers éléments à partir du $\lfloor n/2 \rfloor$ dans les feuilles et on réunit les tas en un seul tas au fur et à mesure. La complexité est $\mathcal{O}(n)$.

**Le tri par tas** trie un tableau de taille $n$ en temps $\mathcal{O}(n \log n)$ : on transforme le tableau en tas, puis on effectue $n$ extractions successives.

## Unir et trouver

**Unir-trouver.** Unir-trouver est une structure de données qui représente une partition d'un ensemble fini ou de manière équivalente une relation d'équivalence.

Un ensemble est composé de $n$ éléments représentés par des entiers de 0 à $n-1$. Une partie de l'ensemble est identifiée par un des éléments qui la compose. Au départ, une structure unir-trouver est initialisée de telle manière que chaque élément de l'ensemble appartient à sa propre partie.

La structure unir-trouver possède deux opérations :

**Trouver (find)** détermine à quelle partie appartient un élément. Cette opération permet de déterminer si deux éléments appartiennent à la partie.

**Unir (union)** réunit deux parties de la partition en une seule.

Pour être efficace, cette structure est composite et composée :

- **d'un tableau `parent`** dans laquelle on maintient l'identité de la partie à laquelle appartient l'élément : `parent[i]` est un entier qui représente la partie à laquelle appartient `i`.
- d'un **tableau `rank`** dans laquelle on maintient un ordre entre les parties : leur rang. Ce rang représente d'une certaine manière la taille de la partie. `rank[i]` est un entier qui vaut 0 à l'initialisation de la structure et qui croit au fur et à mesure des unions de parties.

**Exemple de tructure unir-trouver.** Par exemple, pour un ensemble à 5 éléments, `parent = [1,1,4,1,4]` signifie que les éléments d'indice 0,1 et 3 sont dans une même partie représentée par 1 et que 2 et 4 sont dans une même partie représentée par 4. `rang=[1,1,2,1,2]` signifie que le rang de la partie 1 est 1 et celui de la partie 4 est 2.

R Pour que la complexité soit optimale, il est important d'implémenter la compression de chemin dans les opérations de la structure unir-trouver. Ce concept est illustré sur la figure 3. Au fur que l'on interroge (trouver) la structure, on relie directement un élément à l'entier qui représente sa partie.

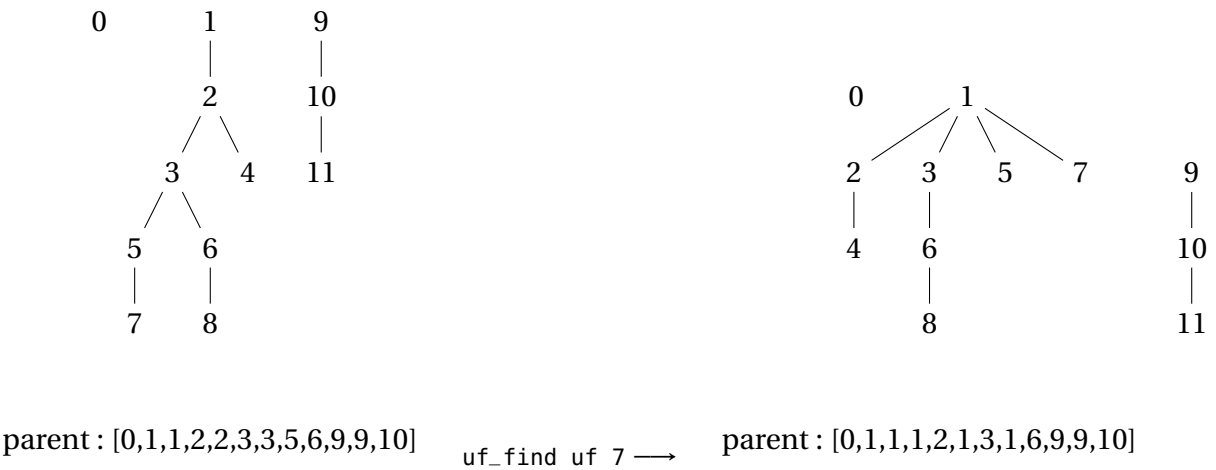



FIGURE 3 – Illustration de la compression de chemin lorsqu'on cherche le parent de l'élément 7. Cet ensemble est divisé en trois parties identifiées par les racines 0, 1 et 9.

# GRAPHES

**Graphe.** Un graphe $G$ est un couple $G = (S, A)$ où $S$ est un ensemble fini et non vide d'éléments appelés **sommets** et $A$ un ensemble de paires d'éléments de $S$ appelées **arêtes**.

**Graphe pondéré.** Un graphe $G = (S, A)$ est pondéré s'il existe une application $w : A \longrightarrow \mathbb{R}$. Le poids de l'arête $ab$ vaut $w(ab)$.

**Graphe orienté.** Un graphe $G = (S, A)$ est orienté si ses arêtes sont orientées selon une direction. Les arêtes sont alors désignées par le mot arc.

**Graphe complet.** Un graphe $G = (S, A)$ est complet si et seulement si une arête existe entre chaque sommet, c'est-à-dire si tous les sommets sont voisins.

**Graphe planaire.** Une graphe planaire est un graphe que l'on peut représenter sur un plan sans qu'aucune arête ne se croise.

**Graphe biparti.** un graphe $G = (S, A)$ est biparti si l'ensemble $S$ de ses sommets peut être divisé en deux sous-ensembles disjoints $U$ et $W$ tels que chaque arête de $A$ ait une extrémité dans $U$ et l'autre dans $W$.

**Adjacent ou voisin.** Deux sommets $a$ et $b$ sont adjacents ou voisins si le graphe contient une arête $ab$. Deux arêtes sont adjacentes ou voisines s'il existe un sommet commun à ces deux arêtes.

## Caractérisation structurelle des graphes

**Ordre d'un graphe.** L'ordre d'un graphe est le nombre de ses sommets. Pour $G = (S, A)$, l'ordre du graphe vaut donc le cardinal de l'ensemble $S$ que l'on note généralement $|S|$. On note parfois l'ordre d'un graphe $|G|$.

**Taille d'un graphe.** La taille d'un graphe désigne le nombre de ses arêtes. On le note $|A|$ et parfois $||G||$

**Voisinage d'un sommet.** L'ensemble de voisins d'un sommet $a$ d'un graphe $G = (S, A)$ est le voisinage de ce sommet. On le note généralement $V_G(a)$.

**Incidence.** Une arête est dite incidente à un sommet si ce sommet est une des extrémités de cette arête

**Degré d'un sommet.** Le degré $d(a)$ d'un sommet $a$ d'un graphe $G$ est le nombre d'arêtes incidentes au sommet $a$. C'est aussi $|V_G(a)|$.

**Degrés d'un graphe orienté.** Dans un graphe orienté $G$ et pour un sommet $s$ de ce graphe, on distingue :

- le degré entrant $d_+(s)$ : le nombre d'arêtes incidentes à $s$ et dirigées sur $s$,
- le degré sortant $d_-(s)$ : le nombre d'arêtes incidentes qui sortent de $s$ et qui sont dirigées vers un autre sommet.

# Représentations d'un graphe

**Listes d'adjacence** : c'est le tableau des listes des voisins de chaque sommet du graphe : pour $s \in S, G[s]$ est la liste des voisins de $s$.

— Avantage : gain en espace mémoire, accès en temps constant aux voisins d'un sommet.

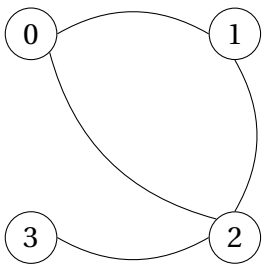
— Inconvénient : test d'existence d'une arête en temps linéaire.

**Matrice d'adjacence** : pour $(s,t) \in S^2, G[s][t]$ est un booléen vrai si et seulement si $(s,t)$ est dans $A$.

— Avantage : test d'existence d'une arête en temps constant.

— Inconvénient : espace mémoire toujours en $\Theta\left(|S|^2\right)$, accès aux voisins en temps linéaire.

```
g = [[1,2], [0,2], [0,1,3], [2]]
```




$$M = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

FIGURE 4 – Graphe simple : liste d'adjacence à gauche, matrice d'adjacence à droite

R On peut implémenter un graphe pondéré par :

— une listes d'adjacence : les éléments d'une liste sont alors des couples (voisin, poids)

— une matrice d'adjacence : $G[s][t]$ est égal au poids de l'arête de $s$ à $t$ ou 0 si $s = t$ ou $+\infty$ si l'arête n'existe pas.

# Parcours de graphes

**Parcours d'un graphe.** Un parcours d'un graphe $G$ est un ordre pour visiter chaque sommet d'un graphe.

**Parcours en largeur.** Parcourir en largeur un graphe signifie qu'on cherche à visiter tous les voisins d'un sommet avant de parcourir les autres sommets du graphe. On explore des voisins les plus près aux voisins les plus éloignés.

**Parcours en profondeur.** Parcourir en profondeur un graphe signifie qu'on cherche à visiter tous les voisins descendants d'un sommet qu'on découvre avant de parcourir les autres sommets découverts en même temps.

**Complexités des parcours en largeur et profondeur** : $\mathcal{O}\left(|S| + \sum_{s \in S} \deg(s)\right) = \mathcal{O}(|S| + |A|)$ avec des listes d'adjacence.

L'implémentation d'un parcours nécessite une structure de données qui **mémorise les sommets déjà visités**. Cette structure souvent nommée `decouverts` ou **`deja_vus`** est un tableau de booléens ou un dictionnaire.

**Le parcours en largeur nécessite une file d'attente (FIFO).**

**Le parcours en profondeur nécessite une pile (LIFO).**

**Utilisations du parcours en largeur :**

— calcul des composantes connexes,

— bicoloration d'un graphe (vérification de la bipartition d'un graphe),

**Algorithme 1** Parcours en largeur d'un graphe

```
1 : Fonction PARCOURS_EN_LARGEUR(G, s)                    ▷ s est un sommet de G
2 :     F ← une file d'attente vide                       ▷ F comme file
3 :     D ← ∅                                             ▷ D ensemble des sommets découverts
4 :     P ← une liste vide                                ▷ P comme parcours
5 :     ENFILER(F,s)
6 :     AJOUTER(D,s)          ▷ Pour la preuve de la correction, on précise que d[s] = 0
7 :     tant que F n'est pas vide répéter
8 :         v ← DÉFILER(F)
9 :         AJOUTER(P, v)                                 ▷ ou bien traiter le sommet en place
10 :        pour chaque voisin x de v dans G répéter
11 :            si x ∉ D alors                            ▷ x n'a pas encore été découvert
12 :                AJOUTER(D, x)
13 :                ENFILER(F, x)   ▷ Pour la preuve de la correction, on ajoute ici d[x] = d[u] + 1
14 :    renvoyer P                  ▷ Facultatif, on pourrait traiter chaque sommet v en place
```

**Algorithme 2** Parcours en profondeur d'un graphe (version récursive)

```
1 : Fonction PARCOURS_EN_PROFONDEUR(G, s, D, C)           ▷ s est un sommet de G
2 :     AJOUTER(C, s)                                     ▷ s est ajouté au parcours du graphe
3 :     AJOUTER(D, s)                                     ▷ s est marqué découvert
4 :     pour chaque voisin x de s dans G répéter
5 :         si x ∉ D alors                                ▷ x n'a pas encore été découvert
6 :             PARCOURS_EN_PROFONDEUR(G, x, D, C)
```

**Algorithme 3** Parcours en profondeur d'un graphe (version itérative)

```
1 : Fonction PARCOURS_EN_PROFONDEUR(G, s)                 ▷ s est un sommet de G
2 :     P ← une pile vide                                 ▷ P comme pile
3 :     D ← un ensemble vide                              ▷ D comme découverts
4 :     C ← un liste vide                                 ▷ C pour le parcours
5 :     EMPILER(P,s)
6 :     tant que P n'est pas vide répéter
7 :         v ← DÉPILER(P)
8 :         AJOUTER(C, v)
9 :         pour chaque voisin x de v dans G répéter
10 :            si x ∉ E alors
11 :                EMPILER(P, x)
12 :                AJOUTER(D, x)
13 :    renvoyer C                  ▷ Facultatif, on pourrait traiter chaque sommet v en place
```

— recherche des plus courts chemins dans un graphe non pondéré,

— exploration d'un arbre de jeu en largeur.

**Utilisations du parcours en profondeur :**

— détecter des boucles dans un graphe orienté,

— trouver un ordre topologique dans un graphe orienté acyclique : **Tri topologique**. Pour produire un ordre dans l'ensemble des sommets du graphe, ce qui est possible si le graphe orienté est acyclique, car il existe alors au moins un sommet qui ne possède pas d'arc entrant. Application typique : programmer l'exécution de tâches dépendantes les unes des autres. par exemple via un parcours en profondeur réitéré sur chaque sommet.

— calcul des composantes fortement connexes.

# Plus courts chemins pondérés

La distance entre deux sommets est la somme des pondérations des arêtes traversées pour relier les deux sommets. On maintient donc **un tableau des distances**.

D'une manière générale, chercher des chemins peut se faire en parcourant en largeur un graphe et en utilisant conjointement **un tableau de parents sur le chemin**.

## a Algorithme de Dijsktra

— trouve les distances depuis un sommet de **départ** à tous les autres,

— applicable uniquement dans un graphe pondéré à **poids positifs**,

— à chaque itération, **on extrait le sommet le plus proche du sommet de départ**, et on met à jour la distance pour ses voisins,

— complexité : $\mathcal{O}((|S|+|A|)\log|S|)$ avec des **listes d'adjacence**.

**Algorithme 4** Algorithme de Dijkstra, plus courts chemins à partir d'un sommet donné

```
1 : Fonction DIJKSTRA(G = (S, A, w), s_0)        ▷ Trouver les plus courts chemins à partir de a ∈ S
2 :     Δ ← s_0                                  ▷ Δ est l'ensemble des sommets dont on connaît la distance à s_0
3 :     n ← |S|                                  ▷ L'ordre du graphe
4 :     Π ← tableau vide (taille n)              ▷ Π[s] est le parent de s dans le plus court chemin de s_0 à s
5 :     d ← tableau vide (taille n)              ▷ l'ensemble des distances au sommet s_0
6 :     ∀ s ∈ S, d[s] ← w(s_0, s)                ▷ w(s_0, s) = +∞ si s n'est pas voisin de s_0, 0 si s = s_0
7 :     tant que Δ̄ n'est pas vide répéter        ▷ Δ̄ : sommets dont la distance n'est pas connue
8 :         Choisir u dans Δ̄ tel que d[u] = min(d[v], v ∈ Δ̄)   ▷ Choix glouton!
9 :         Δ = Δ ∪ {u}                          ▷ On prend la plus courte distance à s_0 dans Δ̄
10 :        pour chaque voisin x de u répéter
11 :            si d[x] > d[u] + w(u, x) alors
12 :                d[x] ← d[u] + w(u, x)        ▷ Mises à jour des distances des voisins
13 :                Π[x] ← u                     ▷ Pour garder la tracer du chemin le plus court
14 :    renvoyer d, Π
```

## b Algorithme $A^*$

— trouve la distance entre un sommet de **départ** et un sommet d'**arrivée**,

— à chaque itération, **on extrait le sommet qui minimise la somme entre la distance depuis le sommet de départ et l'estimation donnée par une heuristique au sommet d'arrivée**,

— si l'heuristique est **admissible** (c'est-à-dire optimiste dans son estimation), l'algorithme est exact,

— complexité : comme Dijkstra si l'heuristique est monotone (respecte une inégalité triangulaire), peut-être exponentielle sinon.

### c Algorithme de Floyd-Warshall

— trouve **toutes les distances** entre deux sommets quelconques,

— applicable uniquement dans un **graphe orienté sans cycle négatif** ou dans un **graphe non orienté à poids positifs**,

— algorithme de **programmation dynamique**

— complexité : $\mathcal{O}\left(|S|^3\right)$ avec une **matrice d'adjacence**.

L'équation de la programmation dynamique exprime une suite de matrices :

$$\forall p \in [\![ 1, n ]\!], \forall i, j \in [\![ 0, n-1 ]\!], M_p(i,j) = \min\left(M_{p-1}(i,j), M_{p-1}(i,p-1) + M_{p-1}(p-1,j)\right) \tag{4}$$

Pour $p = 0$, on pose $M_0$, la matrice d'adjacence du graphe.

## Arbres couvrants

On ne considère dans cette section **que** des graphes non orientés, pondérés ou non.

— Un arbre est un graphe qui satisfait les conditions suivantes : **connexe, sans cycle et** $|A| = |S| - 1$. Deux suffisent et entraînent la troisième.

— Un arbre **couvrant** est un sous-graphe qui est un arbre et contient tous les sommets.

### d Algorithme de Kruskal

Il permet de trouver une **forêt d'arbres couvrants de poids minimal**, et fonctionne donc même si le graphe n'est pas connexe. Il est glouton et optimal.

— on part d'un graphe vide puis on parcourt les arêtes par poids croissant et on garde celles qui ne créent pas de cycle,

— on peut utiliser une structure Unir-Trouver pour garder en mémoire les composantes connexes et tester la création d'un cycle,

— complexité : $\mathcal{O}(|A| \log |S|)$.

### e Algorithme de Prim

Il permet de trouver **un arbre couvrant de poids minimal**, si le graphe est connexe. Il est glouton et optimal.

— on part d'un graphe vide et d'un sommet,

— on fait croitre l'arbre en choisissant une arête dont l'extrémité de départ appartient à l'arbre et mais pas l'extrémité d'arrivée **et** dont le poids est le plus faible, garantissant ainsi l'absence de cycle et la minimalité.

— complexité : $\mathcal{O}((|S| + |A|) \log |S|)$.

## Couplages

**Couplage.** Un couplage $\Gamma$ dans un graphe non orienté $G = (S, A)$ est **un ensemble d'arêtes deux à deux non adjacentes**, c'est-à-dire que deux arêtes de $\Gamma$ n'ont jamais de sommet en commun.

**Sommets couplés, sommets exposés.** Un sommet est couplé s'il fait parti d'une arête de $\Gamma$. Un sommet est exposé s'il ne fait pas parti des arêtes de $\Gamma$, c'est-à-dire qu'il n'est pas couplé.

**Couplage maximal.** Un couplage maximal est tel que si on lui ajoute une arête, ce n'est plus un couplage. Il est donc maximal au sens de l'inclusion.

**Couplage de cardinal maximum.** Un couplage de cardinal maximum est un couplage contenant le plus grand nombre d'arêtes possible.

**Chemin alternant.** Une chemin alternant dans un graphe non orienté $G$ et pour un couplage $\Gamma$ est tel que les arêtes appartiennent successivement à $\Gamma$ et $E \setminus \Gamma$.

**Chemin augmentant.** Un chemin augmentant est un chemin alternant dont les extrémités sont des sommets exposés, c'est-à-dire qui n'appartiennent pas au couplage $\Gamma$.

**Théorème 1 — Lemme de Berge.** Soit un couplage $\Gamma$ dans un graphe. $\Gamma$ est de cardinal maximum si et seulement s'il ne possède aucun un chemin augmentant.

- Recherche d'un couplage maximum : tant qu'il existe un chemin augmentant $\sigma$, on calcule un nouveau couplage $\Gamma' = \Gamma \Delta \sigma$, où $\Delta$ représente la différence symétrique.
- Dans le cas particulier des graphes bipartis, un chemin augmentant peut être trouvé par un parcours de graphe particulier, qui part et termine d'un sommet non couvert et alterne les arêtes.

# PARADIGMES ALGORITHMIQUES

## Algorithmes gloutons

Un algorithme glouton est généralement une manière intuitive et simple de résoudre un problème :
- on construit une solution systématiquement sans jamais revenir sur des choix faits précédemment,
- on choisit le meilleur localement à chaque itération.

**Exemples :**
- algorithme de Kruskal et Prim (optimal),
- construction de l'arbre de Huffman (optimal),
- rendu de monnaie (optimal si le système monétaire est canonique),
- voyageur de commerce (non optimal).

## Diviser pour régner

Un algorithme de type diviser pour régner consiste à **diviser** le problème en sous-problèmes **indépendants** plus petits jusqu'à atteindre une taille **élémentaire** de problème pour laquelle on sait le **résoudre**. Après résolution des problèmes élémentaire, on **combine** les solutions pour construire la solution au problème de départ.

La complexité vérifie généralement $C(n) = aC(n/b) + f(n)$ où $a$ est le nombre de sous-problèmes, $n/b$ leurs tailles et $f(n)$ la complexité de l'ellaboration des sous-problèmes et de la construction de solution.

**Exemples :**
- recherche dichotomique dans un tableau trié $\mathcal{O}(\log n)$,
- tri fusion $\mathcal{O}(n \log n)$,
- tri rapide.

## Programmation dynamique

Un algorithme de programmation dynamique consiste à résoudre un problème à sous-structure optimale, c'est-à-dire qu'**il existe une formulation récursive de la solution optimale du problème en fonction des solutions optimales des sous-problèmes.** Les sous-problèmes se **chevauchent**, ne sont pas indépendants.

On peut programmer :
- par approche descendante : **récursivement** en utilisant la **mémoïsation** pour ne pas recalculer plusieurs fois les solutions aux mêmes sous-problèmes.
- par approche ascendante : **impérativement**, **en complétant un tableau de résolution** dans le bon ordre (à déterminer en fonction de la récurrence)

**Exemples :**
- algorithme de Floyd-Warshall,
- rendu de monnaie,
- sac à dos,
- distance d'édition,
- plus longue sous-chaîne commune.

# Algorithmes probabilistes

— Algorithme Monte Carlo : temps de calcul toujours identique, mais résultat variable. Exemples : test de primalité (Fermat, Miller-Rabin), recherche d'une coupe minimale (Karger).

— Algorithme Las Vegas : résultat correct, mais temps de calcul variable. Exemple : tri rapide randomisé.

# Exploration

Trouver toutes les solutions à un problème peut se faire de manière exhaustive, c'est-à-dire en essayant toutes les solutions possibles. On parle alors d'**approche par exhaustion** ou par **force brute**. C'est une approche desespérée[1], pas efficace mais simple et qui peut fonctionner lorsque la dimension du problème est faible.

Un algorithme de **backtracking** (ou retour sur trace) construit une solution partielle potentielle **satisfaisant des contraintes** petit à petit en explorant en profondeur l'arbre des construction possibles, **en revenant en arrière** lorsqu'on réalise qu'une solution ne peut pas être continuée. Lorsqu'il a construit une solution au problème, il peut soit la renvoyer si l'objectif était de trouver une solution, soit la mémoriser et continuer l'exploration si l'objectif est de trouver toutes les solutions.

**Exemples :**

— $N$-reines,

— résolution de sudoku.

1. souvent utilisée quand on ne sait pas faire autrement

## Automates finis

**Automate fini déterministe (AFD).** Un automate fini déterministe est un quintuplet $(Q, \Sigma, q_i, \delta, F)$ tel que :

a. $Q$ est un ensemble non vide et fini dont les éléments sont les états,
b. $\Sigma$ est l'alphabet,
c. $q_i \in Q$ est l'état initial,
d. $\delta : Q \times \Sigma \longrightarrow Q$ est la **fonction** de transition de l'automate,
e. $F \subseteq Q$ est l'ensemble des états accepteurs ou terminaux.

R Le déterminisme d'un AFD est dû aux faits que :
- l'état initial est un **singleton**,
- $\delta$ est une **fonction** : à un couple (état,lettre) $(q, a)$, $\delta$ associe au plus un état $q'$.

**Fonction de transition partielle.** On dit que la fonction de transition $\delta$ est partielle s'il existe au moins un couple (état,lettre) pour lequel elle n'est pas définie.

**Automate complet.** Un AFD $\mathcal{A} = (Q, \Sigma, q_i, \delta, F)$ est dit complet si $\delta$ est une **application**, c'est-à-dire $\delta$ n'est pas partielle, il existe une transition pour chaque lettre de $\Sigma$ pour tous les états.

**Automate normalisé.** Un automate est normalisé s'il ne possède pas de transition entrante sur son état initial et s'il possède un seul état final sans transition sortante.

**Fonction de transition étendue aux mots.** La fonction de transition peut être étendue aux mots par passages successifs d'un état à un autre en lisant les lettres d'un mot.
On définit inductivement cette fonction étendue noté $\delta^*$ :

$$\forall q \in Q, \delta^*(q, \epsilon) = q \tag{5}$$
$$\forall q \in Q, \forall w \in \Sigma^*, \forall a \in \Sigma, \delta^*(q, w.a) = \delta(\delta^*(q, w), a) \tag{6}$$

**Acceptation d'un mot par un automate.** Un mot $w \in \Sigma^*$ est **accepté** par un automate $\mathcal{A}$ si et seulement si $\delta^*(q_i, w) \in F$, c'est-à-dire la lecture du mot $w$ par l'automate conduit à un état accepteur.

**Langage reconnu par un AFD.** Le langage $\mathcal{L}_{\text{rec}}(\mathcal{A})$ **reconnu** par un automate fini déterministe $\mathcal{A}$ est l'ensemble des mots reconnus par $\mathcal{A}$ :

$$\mathcal{L}_{\text{rec}}(\mathcal{A}) = \{w \in \Sigma^*, w \text{ est accepté par } \mathcal{A}\} \tag{7}$$

**Langage reconnaissable.** Un langage $\mathcal{L}$ sur un alphabet $\Sigma$ est reconnaissable s'il existe un automate fini déterministe $\mathcal{A}$ d'alphabet $\Sigma$ tel que $\mathcal{L} = \mathcal{L}_{\text{rec}}(\mathcal{A})$.

R On peut implémenter un automate comme un graphe orienté dont les arêtes sont étiquetées.

**Accessibilité d'un état.** Un état $q$ d'un automate est dit accessible s'il existe un mot $w \in \Sigma^*$ tel que $\delta^*(q_i, u) = q$, c'est-à-dire s'il est possible de l'atteindre depuis l'état initial.

**Co-accessibilité d'un état.** Un état $q$ d'un automate est dit co-accessible s'il existe un mot $w \in \Sigma^*$ tel que $\delta^*(q, u) \in F$, c'est-à-dire si, à partir de cet état, il est possible d'atteindre un état accepteur.

**Automate émondé.** Un automate est dit émondé tous ses états sont à la fois accessibles et co-acessibles.

**Théorème 2 — Automate d'un langage reconnaissable.** Si un langage est reconnaissable alors il existe un automate fini déterministe :

- normalisé qui le reconnaît.
- émondé qui le reconnaît.
- complet qui le reconnaît.

**Automate fini non déterministe (AFND).** Un automate fini non déterministe est un quintuplet $(Q, \Sigma, Q_i, \Delta, F)$ tel que :

a. $Q$ est un ensemble non vide et fini dont les éléments sont les états,
b. $\Sigma$ est l'alphabet,
c. $Q_i \subseteq Q$ sont **les** états initiaux,
d. $\Delta \subseteq Q \times \Sigma \times Q$ est la **relation** de transition de l'automate,
e. $F \subseteq Q$ est l'ensemble des états accepteurs ou terminaux.

R Un AFND est par essence asynchrone. Son exécution nécessite l'exécution de toutes les transitions possibles au départ de chaque état traversé.

R Le non déterminisme d'un AFND est dû au fait que $\Delta$ n'est pas une fonction mais une relation : depuis un état $q$, une même lettre peut faire transiter l'AFND vers des états différents. Par exemple, $(q, a, q')$ et $(q, a, q'')$. Quelle transition choisir ? Là est le non déterminisme.

M **Méthode 2 — Déterminisé d'un AFND ou automate des parties** Le déterminisé d'un automate fini non déterministe $\mathcal{A} = (Q, \Sigma, Q_i, \Delta, F)$ est l'automate $\mathcal{A}_d = (\mathcal{P}(Q), \Sigma, q_i, \delta, \mathcal{F})$ défini par :

- $\mathcal{P}(Q)$ est l'ensemble des parties de $Q$,
- $q_i$ est **l'ensemble des états initiaux**,
- $\forall \pi \in \mathcal{P}(Q), \forall a \in \Sigma, \delta(\pi, a) = \bigcup_{q \in \pi} \{q' \in Q, (q, a, q') \in \Delta\}$,
- $\mathcal{F} = \{\pi \in \mathcal{P}(Q), \pi \cap F \neq \emptyset\}$.

Ce qui signifie que :

- l'état initial du déterminisé est l'état initial de l'AFND, ou bien, s'il possède plusieurs états initiaux, l'état initial constitué par la partie de tous les états initiaux de l'AFND.
- toute partie de $Q$ est susceptible d'être un état. En pratique dans les exercices, vous construirez les états au fur et à mesure à partir de l'état initial.
- les états accepteurs sont **les parties qui contiennent un état accepteur** de l'AFND.

**Théorème 3** Un AFND $\mathcal{A}$ et son déterminisé $\mathcal{A}_d$ reconnaissent le même langage.

$$\mathcal{L}_{rec}(\mathcal{A}) = \mathcal{L}_{rec}(\mathcal{A}_d) \tag{8}$$

**$\epsilon$-transition.** Une $\epsilon$-transition est une transition dans un automate non déterministe dont l'étiquette est le mot vide $\epsilon$. C'est une transition spontanée d'un état à un autre.

# Langages réguliers

**Ensemble des langages réguliers.** L'ensemble des langages réguliers $\mathcal{L}_r$ sur un alphabet $\Sigma$ est défini inductivement par :

**(Base (i))** $\emptyset \in \mathcal{L}_r$,

**(Base (ii))** $\{\epsilon\} \in \mathcal{L}_r$,

**(Base (iii))** $\forall a \in \Sigma, \{a\} \in \mathcal{L}_r$,

**(Règle de construction (i) union)** $\forall \mathcal{L}_1, \mathcal{L}_2 \in \mathcal{L}_r, \mathcal{L}_1 \cup \mathcal{L}_2 = \mathcal{L}_r$

**(Règle de construction (ii)) concaténation** $\forall \mathcal{L}_1, \mathcal{L}_2 \in \mathcal{L}_r, \mathcal{L}_1.\mathcal{L}_2 = \mathcal{L}_r$,

**(Règle de construction (iii)) fermeture de Kleene** $\forall \mathcal{L} \in \mathcal{L}_r, \mathcal{L}^* = \mathcal{L}_r$.

R L'union, la concaténation et l'étoile de Kleene d'un nombre fini de langages rationnels sont des langages rationnels. On dit que les langages réguliers sont **stables** par ces opérations.

**Syntaxe des expressions régulières.** L'ensemble des expressions régulières $\mathcal{E}_R$ sur un alphabet $\Sigma$ est défini inductivement par :

**(Base)** $\{\emptyset, \epsilon\} \cup \Sigma \in \mathcal{E}_R$,

**(Règle de construction (union))** $\forall e_1, e_2 \in \mathcal{E}_R, e_1 \mid e_2 \in \mathcal{E}_R$

**(Règle de construction (concaténation))** $\forall e_1, e_2 \in \mathcal{E}_R, e_1 e_2 \in \mathcal{E}_R$,

**(Règle de construction (fermeture de Kleene))** $\forall e \in \mathcal{E}_R, e^* \in \mathcal{E}_R$.

R À toute expression régulière, on fait correspondre un ensemble de mots, un langage dénoté par cette expression. Souvent, on confond cette expression et le langage dénoté.

**Théorème 4 — Un langage $\mathcal{L}$ est régulier si et seulement s'il existe une expression régulière $e$ telle que $\mathcal{L}_{ER}(e) = \mathcal{L}$.**

**Théorème 5 — Kleene.** Un langage $\mathcal{L}$ sur un alphabet $\Sigma$ est un langage régulier si et seulement s'il est reconnaissable.

**Algorithmes à connaître :**

**ER** $\rightarrow \mathcal{A}$ pour construire un automate à partir d'une expression régulière : Berry-Sethi, méthode compositionnelle (construction de Thompson)

$\mathcal{A}$ *to* **ER** pour trouver une ER à partir d'un automate : algorithme de Brzozowski-McCluskey ou élimination des états.

M **Méthode 3 — Linéarisation de l'expression régulière** À partir de l'expression régulière de départ, on numérote à partir de 1 chaque lettre de l'expression dans l'ordre de lecture.

Si une lettre apparaît plusieurs fois, elle est numérotée autant de fois qu'elle apparaît avec un numéro

différent.

Par exemple, la linéarisation de $ab|(ac)^*$ est $a_1b_1|(a_2c_1)$.

**M** **Méthode 4 — Algorithme de Berry-Sethi** Pour obtenir un automate fini local reconnaissant le langage $\mathcal{L}_{ER}(e)$ à partie d'une expressions régulière $e$ sur un alphabet $\Sigma$ :

a. Linéariser l'expression $e$ : cela consiste à numéroter toutes les lettres qui apparaissent afin de créer une expression rationnelle linéaire $e'$.

b. Déterminer les ensembles $P$ (premières lettres), $S$ (dernières lettres) et $F$ (facteurs de deux lettres) associés au langage local $\mathcal{L}(e')$.

c. Déterminer un automate **local** $\mathcal{A}$ reconnaissant $\mathcal{L}_{ER}(e')$ à partir de $P, S$ et $F$. On peut associer ses états aux les lettres de l'alphabet de $e'$. L'état initial est relatif au mot vide : s'il appartient au langage, on fait de cet état un état accepteur. Toutes les transitions qui partent de l'état initial conduisent à un état associé à une lettre de $P$. Les états accepteurs sont associés aux éléments de $S$. Les facteurs de deux lettres déterminent les autres transitions.

d. Supprimer les numéros sur les transitions et faire réapparaître l'alphabet initial $\Sigma$.

L'automate obtenu est nommé automate de Glushkov. C'est un automate **local et sans transitions spontanées**. Il n'est pas nécessairement déterministe mais on peut le déterminiser facilement en utilisant la procédure de déterminisation d'un AFND (cf méthode 2). Il possède $|\Sigma_e|$ états où $\Sigma_e$ est l'alphabet étendu obtenu en faisant le marquage. $|\Sigma_e|$ correspond donc au nombre total de lettres dans $e$ en comptant les répétitions. Dans le pire des cas, le nombre de transitions de l'automate est en $O(|\Sigma_e^2|)$.

**M** **Méthode 5 — Construire l'expression régulière équivalent à un automate normalisé** Deux grandes étapes sont nécessaires pour construire l'expression régulière équivalent à un automate. **Pour chaque état q à éliminer**, c'est-à-dire les états autres que l'état initial ou l'état final,

a. **fusionner** les expressions régulières des transitions au départ de $q_s$ et à destination du même état $q_n$. Formellement, si on a les transitions $(q_s, e_1, q_n)$ et $(q_s, e_2, q_n)$, alors on fusionne les deux expressions en faisant leur somme : $(q_s, e_1|e_2, q_n)$. On ne conserve ainsi qu'une seule expression par destination au départ de $q_s$.

b. **éliminer l'état** $q_s$ en mettant à jour les transitions au départ des états précédents. Considérons chaque transition de type $(q_p, e_1, q_s)$ et $(q_s, e_2, q_n)$, c'est-à-dire les transitions pour lesquelles $q_s$ intervient. Si on souhaite éliminer $q_s$, il faut considérer à chaque fois deux cas :

   a. une transition boucle $(q_s, e_b, q_s)$ existe : alors il est nécessaire d'ajouter la transition $(q_p, e_1 e_b^* e_2, q_n)$,

   b. dans le cas contraire, il suffit d'ajouter la transition $(q_p, e_1 e_2, q_n)$.

**R** Grâce au théorème de Kleene et aux propriétés des langages reconnaissables, les langages réguliers sont stables par intersection finie et passage au complémentaire.

**R** Le langage des puissances $\{a^n b^n \mid n \in \mathbb{N}\}$, $\{uu \mid u \in \Sigma^*\}$ ou le langage des palindromes sont des **langages non réguliers**.

**Théorème 6 — Lemme de l'étoile.** Pour tout langage **régulier** $\mathcal{L}$ sur une alphabet $\Sigma$, on a :

$$\exists n \geq 1, \forall w \in \mathcal{L}, |w| \geq n \Rightarrow \exists x, y, z \in \Sigma^*, w = xyz \wedge \left(y \neq \epsilon \wedge |xy| \leq n \wedge \mathcal{L}_{ER}(xy^*z) \subseteq \mathcal{L}\right) \tag{9}$$

**R** On peut utiliser le lemme de pompage pour prouver qu'un langage n'est pas rationnel en procédant par l'absurde.

## Logique propositionnelle

**Constante universelle $\top$.** La constante $\top$ désigne le vrai.

**Constante vide $\bot$.** La constante $\bot$ désigne la contradiction.

**Variable propositionnelle.** Une variable propositionnelle est une proposition atomique.

**Ensemble des formules propositionnelles $\mathcal{F}$ (défini inductivement).** L'ensemble $\mathcal{F}$ des formules propositionnelles sur $\mathcal{V}$ est défini inductivement comme suit :

$$\bot \in \mathcal{F} \quad \text{(Base)} \tag{10}$$
$$\top \in \mathcal{F} \quad \text{(Base)} \tag{11}$$
$$\forall x \in \mathcal{V}, x \in \mathcal{F} \quad \text{(Base)} \tag{12}$$
$$\forall \phi \in \mathcal{F}, \text{not}(\phi) \in \mathcal{F} \quad \text{(Constructeur négation)} \tag{13}$$
$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, \text{and}(\phi, \psi) \in \mathcal{F} \quad \text{(Constructeur conjonction)} \tag{14}$$
$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, \text{or}(\phi, \psi) \in \mathcal{F} \quad \text{(Constructeur disjonction)} \tag{15}$$

Cela signifie qu'une formule logique est soit :

- une constante universelle ou vide,
- une variable propositionnelle,
- une négation d'une formule logique,
- une conjonction ou une disjonction de formules logiques.

R Une formule logique de $\mathcal{F}$ peut être représentée par un arbre. On la désigne par le terme arbre syntaxique.

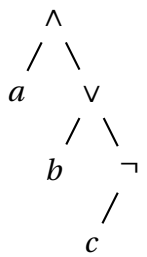



FIGURE 5 – Arbre représentant la formule logique $a \wedge (b \vee \neg c)$

**Ensemble des valeurs de vérité.** L'ensemble des valeurs de vérité est un ensemble à deux éléments que l'on peut noter de différentes manières :

$$\mathbb{B} = \{0,1\} = \{F,V\} = \{\text{Faux},\text{Vrai}\} = \{\text{F},\text{T}\} \tag{16}$$

**Valuation ou interprétation.** Une valuation de $\mathcal{V}$ est une distribution des valeurs de vérité sur l'ensemble des variables propositionnelles $\mathcal{V}$, soit une fonction $v : \mathcal{V} \longrightarrow \mathbb{B}$.

**Évaluation d'une formule logique (définie inductivement).** Soit $v$ une valuation de $\mathcal{V}$ : $v : \mathcal{V} \longrightarrow \mathbb{B}$. L'évaluation d'une formule logique d'après $v$ est notée $[\![\phi]\!]_v$. Elle est définie inductivement par :

$$[\![\bot]\!]_v = \text{F} \quad \text{(Base)} \tag{17}$$
$$[\![\top]\!]_v = \text{V} \quad \text{(Base)} \tag{18}$$
$$\forall x \in \mathcal{V}, [\![x]\!]_v = v(x) \quad \text{(Base)} \tag{19}$$
$$\forall \phi \in \mathcal{F}, [\![\neg\phi]\!]_v = \neg[\![\phi]\!]_v \quad \text{(Constructeur négation)} \tag{20}$$
$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, [\![\phi \wedge \psi]\!]_v = [\![\phi]\!]_v \wedge [\![\psi]\!]_v \quad \text{(Constructeur conjonction)} \tag{21}$$
$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, [\![\phi \vee \psi]\!]_v = [\![\phi]\!]_v \vee [\![\psi]\!]_v \quad \text{(Constructeur disjonction)} \tag{22}$$
$$\tag{23}$$

R La sémantique d'une formule $\varphi$ peut-être donnée par une table de vérité : une colonne par variable et pour $\varphi$, une ligne par valuation possible des variables.

| $a$ | $\neg a$ |
|---|---|
| F | V |
| V | F |

| $a$ | $b$ | $a \wedge b$ |
|---|---|---|
| F | F | F |
| F | V | F |
| V | F | F |
| V | V | V |

| $a$ | $b$ | $a \vee b$ |
|---|---|---|
| F | F | F |
| F | V | V |
| V | F | V |
| V | V | V |

| $a$ | $b$ | $a \Longrightarrow b$ |
|---|---|---|
| F | F | V |
| F | V | V |
| V | F | F |
| V | V | V |

TABLE 2 – Tables de vérité des opérateur négation, conjonction, disjonction et implication

**Modèle.** Un modèle pour une formule logique $\phi$ est une valuation $v$ telle que :

$$[\![\phi]\!]_v = \text{V} \tag{24}$$

**Conséquence sémantique $\vDash$.** Soit $\phi$ et $\psi$ deux formules de $\mathcal{F}$. On dit que $\psi$ est une conséquence sémantique de $\phi$ si tout modèle de $\phi$ est un modèle de $\psi$. On note alors : $\phi \vDash \psi$

**Équivalence sémantique $\equiv$.** Deux formules logiques $\phi$ et $\psi$ sont équivalentes sémantiquement si quelle que soit la valuation choisie, l'évaluation des deux formules produit le même résultat.

$$\phi \equiv \psi \Longleftrightarrow \forall v : \mathcal{V} \longrightarrow \mathbb{B}, [\![\phi]\!]_v = [\![\psi]\!]_v \tag{25}$$

**Tautologie.** Une formule $\phi$ toujours vraie quelle que soit la valuation est une tautologie. On la note $\top$.

**Antilogie.** Une formule $a$ toujours fausse quelle que soit la valuation est une antilogie. On la note $\bot$.

**Formule satisfaisable.** S'il existe une valuation $v$ de $\mathcal{V}$ qui satisfait $\phi$, alors $\phi$ est dite satisfaisable.

$$\phi \text{ est une formule satisfaisable} \Longleftrightarrow \exists v : \mathcal{V} \longrightarrow \mathbb{B}, [\![\phi]\!]_v = \text{V} \tag{26}$$

**Littéral.** Un littéral est une variable propositionnelle ou sa négation : $a$ ou $\neg a$.

**Clause conjonctive.** Une clause conjonctive est une conjonction de littéraux.

**Forme normale disjonctive(FND).** Une forme normale disjonctive d'une formule logique est une disjonction de clauses conjonctives.

**Clause disjonctive.** Une clause disjonctive est une disjonction de littéraux.

**Forme normale conjonctive (FNC).** Une forme normale conjonctive d'une formule logique est une conjonction de clauses disjonctives.

**Théorème 7 — Toute formule logique est équivalente à une forme normale conjonctive.**

**Théorème 8 — Toute formule logique est équivalente à une forme normale disjonctive.**

R Pour trouver la FND d'une formule logique, il suffit de prendre la disjonction de ses modèles.
Pour trouver la FNC d'une formule logique, il suffit de prendre la conjonction de ses anti-modèles puis on prend la négation de chaque littéral pour construire une clause disjonctive.

Exemple :

| $a$ | $b$ | $\phi$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

FND : $\phi = (\neg a \wedge b) \vee (a \wedge \neg b)$ FNC : $\phi = (a \vee b) \wedge (\neg a \vee \neg b)$

**Problème SAT.** Le problème de satisfaisabilité booléenne (SAT) est un problème de décision lié à une formule de logique propositionnelle et dont l'objectif est de déterminer s'il existe une valuation qui rend la formule vraie.

**Algorithme de Quine :**
L'algorithme de Quine explore l'ensemble des valeurs possibles pour chaque variable propositionnelle. Cette exploration se fait de manière arborescente avec simplification des formules logiques après remplacement des variables par $\top$ ou $\bot$ et retour sur trace lorsqu'on peut déterminer que la branche mène à une impasse, c'est-à-dire aboutit à une contradiction.

**Algorithme 5** Algorithme Quine (SAT)

```
1 : Fonction QUINE_SAT(f)                                   ▷ f est une formule logique
2 :     SIMPLIFIER(f)
3 :     si f ≡ ⊤ alors
4 :         renvoyer Vrai
5 :     sinon si f ≡ ⊥ alors
6 :         renvoyer Faux
7 :     sinon
8 :         Choisir une variable x parmi les variables propositionnelles restantes de f
9 :         renvoyer QUINE(f[x ← Vrai]) ou QUINE(f[x ← Faux])
```

# Déduction naturelle

**Séquent ou jugement.** Soit $\mathcal{F}$ l'ensemble des formules logiques, $\Gamma$ une partie de $\mathcal{F}$ (les hypothèses) et $a$ une formule logique (la conclusion). Un séquent est une relation binaire entre l'ensemble $\mathcal{P}(\mathcal{F})$ et $\mathcal{F}$. On la note ainsi :

$$\Gamma \vdash a \tag{27}$$

Elle signifie que l'on peut déduire $a$ en utilisant uniquement les hypothèses $\Gamma$ : de $\Gamma$ on peut conclure $a$.

**Axiome.** Un axiome est une règle d'inférence pour laquelle l'ensemble des hypothèses est vide.

$$\frac{}{\Gamma \vdash a}\ \text{ax}$$

**Arbres de preuve ou arbres de dérivation (définition inductive).** L'ensemble des arbres de preuve $\mathcal{A}$ d'un séquent $s$ par déduction naturelle est soit :

**(une feuille)** l'application d'un **axiome** dont la conclusion est $s$,

**(un nœud)** l'application d'une règle d'inférence (R) dont la conclusion est $s$ et dont les prémisses dérivent d'éléments de $\mathcal{A}$ par des règles d'inférence [a].

*a.* Ces prémisses sont les conclusions d'arbres de preuve

R Une règle d'inférence forme un constructeur de l'ensemble inductif des arbres de preuve. Les nœuds internes d'un arbre de preuve sont des séquents, les feuilles sont des axiomes.

R Pour passer d'une ligne à une autre dans une preuve, on applique uniquement les règles d'inférence de la déduction naturelle.

**Exemple de preuve :**

$$\dfrac{\dfrac{\dfrac{}{p\to q, q\to r, p\vdash q\to r}\,\text{ax} \quad \dfrac{\dfrac{}{p\to q, q\to r, p\vdash p\to q}\,\text{ax} \quad \dfrac{}{p\to q, q\to r, p\vdash p}\,\text{ax}}{p\to q, q\to r, p\vdash q}\,\to_e}{p\to q, q\to r, p\vdash r}\,\to_e}{p\to q, q\to r\vdash p\to r}\,\to_i$$

R Il existe plusieurs logiques qui se différencient principalement par la manière de gérer les déductions de l'absurde :

- La **logique minimale** n'utilise qu'un seul connecteur : l'implication. Elle a pour caractéristique de ne rien déduire de $\bot$. Cela signifie qu'elle n'inclut ni le principe du tiers exclu, ni le principe d'explosion.
- La **logique classique** utilise les opérateurs définis dans ce cours. Elle a pour caractéristique d'inclure les principes ci-dessus et permet donc de conduire des raisonnements par l'absurde. La critique faite à ce type de raisonnement, c'est qu'il permet d'accepter l'existence d'un concept sans pouvoir le construire explicitement.
- La **logique intuitionniste** est constructive : la notion de preuve constructive remplace la notion de vérité. Constuire un concept, c'est exhiber sa preuve d'existence. Si un concept ne peut pas être établi par une preuve constructive, cela signifie qu'il n'existe pas. La logique intuitionniste distingue le *être vrai* du *ne pas être faux*. C'est pourquoi elle n'inclut ni le principe du tiers exclu, ni le raisonnement par l'absurde, ni la double négation. Elle inclut par contre le principe d'explosion.

| **Formule** | **Introduction** | **Élimination** |
|---|---|---|
| $\top$ | $\dfrac{}{\Gamma \vdash \top}\ \top_i$ | |
| $\bot$ | | $\dfrac{\Gamma \vdash \bot}{\Gamma \vdash a}\ \bot_e$ (Principe d'explosion) |
| $a \in \Gamma$ (Axiome) | $\dfrac{}{\Gamma \vdash a}\ \text{ax}$ | |
| Conjonction | $\dfrac{\Gamma \vdash a \qquad \Gamma \vdash b}{\Gamma \vdash a \wedge b}\ \wedge_i$ | $\dfrac{\Gamma \vdash a \wedge b}{\Gamma \vdash a}\ \wedge_e$ |
| Disjonction | $\dfrac{\Gamma \vdash a}{\Gamma \vdash a \vee b}\ \vee_i$ et / ou $\dfrac{\Gamma \vdash b}{\Gamma \vdash a \vee b}\ \vee_i$ | $\dfrac{\Gamma \vdash a \vee b \qquad \Gamma, a \vdash c \qquad \Gamma, b \vdash c}{\Gamma \vdash c}\ \vee_e$ |
| Implication | $\dfrac{\Gamma, a \vdash b}{\Gamma \vdash a \to b}\ \to_i$ | $\dfrac{\Gamma \vdash a \to b \qquad \Gamma \vdash a}{\Gamma \vdash b}\ \to_e$ (Modus ponendo ponens) |
| Négation | $\dfrac{\Gamma, a \vdash \bot}{\Gamma \vdash \neg a}\ \neg_i$ | $\dfrac{\Gamma \vdash \neg a \qquad \Gamma \vdash a}{\Gamma \vdash \bot}\ \neg_e$ (Introduction de la contradiction $\bot_i$) |

TABLE 3 – Ensemble des règles de la déduction naturelle