MPI
MPI*

Asli Grimaud
Gilles Grimaud

Informatique

Cours complets, exercices corrigés et projets guidés pour les classes préparatoires

RESSOURCES TÉLÉCHARGEABLES



ellipses

Références sciences

MPI
MPI*

Informatique

Cours complets, exercices corrigés et projets guidés pour les classes préparatoires

Aslı Grimaud
Gilles Grimaud

ellipses

Collection Références sciences

dirigée par Paul de Laboulaye
paul.delaboulaye@editions-ellipses.fr

Retrouvez tous les livres de la collection et des extraits sur www.editions-ellipses.fr

ISBN 9782340-118843
Dépôt légal : septembre 2026

©Ellipses Édition Marketing S.A.
8/10 rue la Quintinie 75015 Paris

DANGER LE PHOTOCOPILLAGE TUE LE LIVRE

Le Code de la propriété intellectuelle et artistique n'autorisant, aux termes des alinéas 2 et 3 de l'article L. 122-5, d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause, est illicite » (alinéa 1er de l'article L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

www.editions-ellipses.fr

À notre Luce,
que le souvenir du temps confié à cet ouvrage te rappelle la force de la constance.

Avant-propos

Cet ouvrage est destiné aux étudiants, étudiantes, enseignants et enseignantes des cours d'informatique des Classes Préparatoires aux Grandes Écoles (CPGE) en filière Mathématiques, Physique, et Informatique (MPI). Il s'adresse également aux classes MPI*, les développements proposés permettant un approfondissement rigoureux et complet des notions au programme. Les enseignements abordés dans ce document se conforment aux programmes de l'Éducation nationale relatifs à ce nouveau parcours [27]. Les notions en sciences informatiques présentées dans cet ouvrage s'adressent également aux étudiants et étudiantes d'écoles d'ingénieurs, de licence, ainsi qu'aux candidats et candidates au CAPES NSI (Numérique et Sciences Informatiques) et aux agrégatifs et agrégatives d'informatique.

Ce livre constitue la continuité naturelle de l'ouvrage *Informatique MP2I*, rédigé par les mêmes auteurs. Il en prolonge les contenus et en approfondit les notions dans le cadre spécifique de la filière MPI. La maîtrise des chapitres du volume MP2I est un préalable utile à la compréhension des développements proposés dans le présent ouvrage. Les lecteurs et lectrices souhaitant consulter le premier volume peuvent se référer au site dédié : `www.informatiquemp2i.fr`.

Contenu de cet ouvrage Les langages de programmation de référence du programme de MPI sont `C` et `OCaml`. Fidèle à l'esprit du premier volume, cet ouvrage accorde une place équilibrée à ces deux langages, qui sont utilisés de manière complémentaire et équitable tout au long des chapitres. Lorsque cela est utile, cet ouvrage repose sur l'exploitation des systèmes Unix et de la norme POSIX. Les implémentations utilisent l'anglais, langue internationale de l'informatique et compétence essentielle pour tout informaticien et toute informaticienne.

L'organisation pédagogique suggérée inclut des séances de cours, de travaux dirigés, de travaux pratiques, ainsi que la réalisation de projets. Chaque exercice, théorique ou pratique, est accompagné de sa solution. Les résultats théoriques sont également illustrés et démontrés de manière rigoureuse. Dans chaque chapitre, certains exercices sont marqués par ★, qui indique que l'exercice en question est un exercice de niveau avancé.

Site du livre Le site du livre est `www.informatiquempi.fr`. L'ensemble des implémentations proposées dans cet ouvrage, ainsi que la solution des exercices et des projets sont téléchargeables depuis le dépôt :

`https://github.com/InformatiqueMPI/`

Description des chapitres Le contenu est organisé par chapitres et couvre l'intégralité du programme d'informatique de la filière MPI. L'arbre des dépendances entre les chapitres, accompagné d'une proposition de positionnement temporel pour les projets, est présenté dans la figure 0.1. Cette organisation met en évidence la continuité pédagogique entre le volume MP2I et le présent ouvrage.

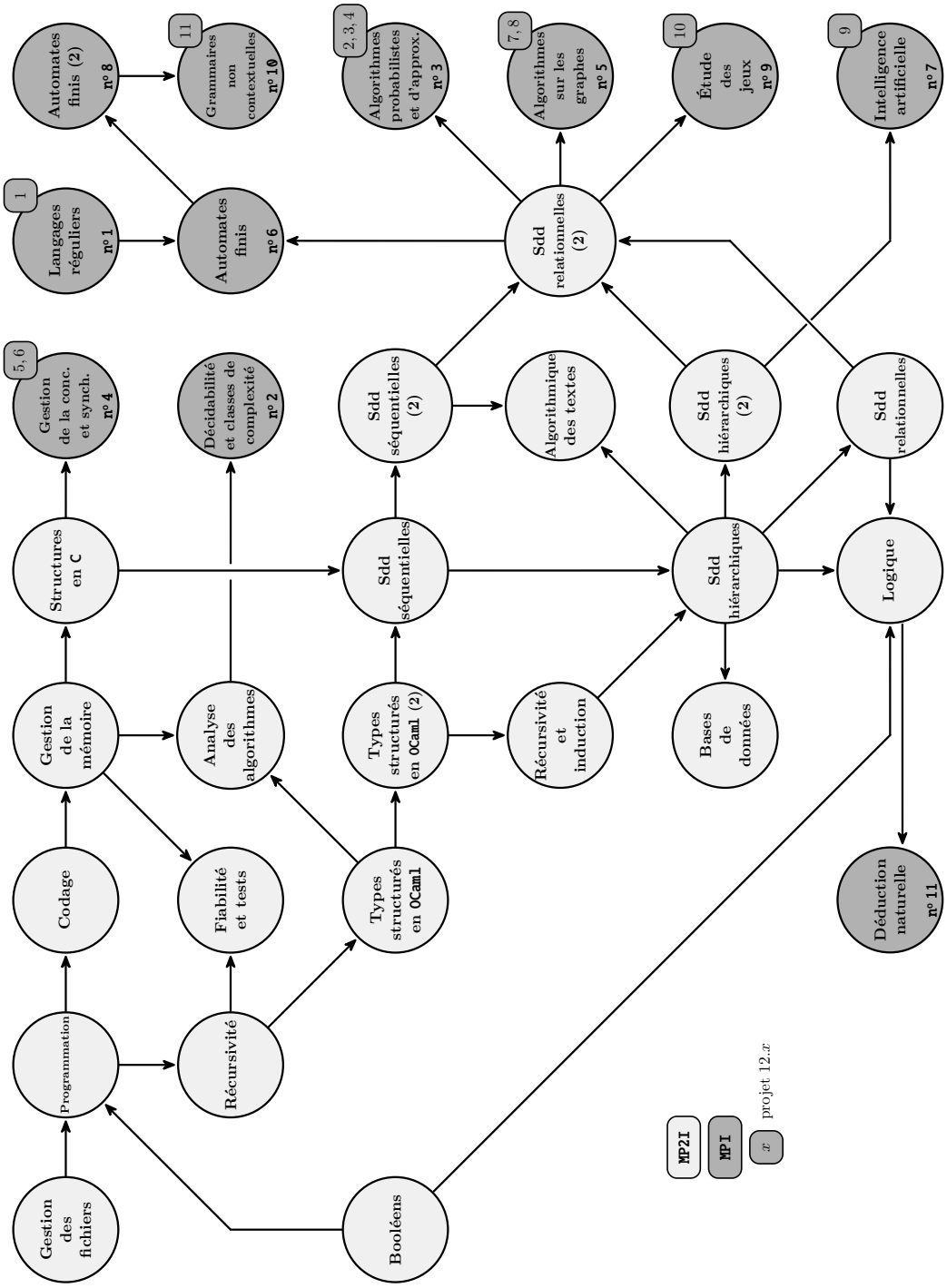


FIGURE 0.1 – Dépendances entre les chapitres

Table des matières

1 Langages réguliers **1**
- 1.1 Alphabets, mots . . . 2
- 1.2 Langages . . . 4
- 1.3 Langages réguliers, expressions régulières . . . 7
  - 1.3.1 Langages réguliers . . . 7
  - 1.3.2 Expressions régulières . . . 8
  - 1.3.3 Expressions régulières étendues . . . 11
- 1.4 TD/TP - Langages réguliers . . . 13

2 Décidabilité et classes de complexité **17**
- 2.1 Décidabilité . . . 18
  - 2.1.1 Notion de problème . . . 18
  - 2.1.2 Modèles de calcul . . . 21
  - 2.1.3 Problèmes décidables . . . 23
  - 2.1.4 Problème de l'arrêt . . . 25
- 2.2 Classes de complexité . . . 27
  - 2.2.1 Classes **P** et **NP** . . . 27
  - 2.2.2 Réduction polynomiale . . . 32
  - 2.2.3 **NP**-complétude . . . 37
  - 2.2.4 Problème SAT . . . 38
- 2.3 TD/TP - Décidabilité et classes de complexité . . . 44

3 Algorithmes probabilistes et d'approximation **51**
- 3.1 Différents types d'algorithmes . . . 52
- 3.2 Exemples d'algorithmes . . . 55
  - 3.2.1 Problème des N reines . . . 56
  - 3.2.2 Test de primalité de Fermat . . . 57
  - 3.2.3 MAX-SAT . . . 61
- 3.3 TD/TP - Algorithmes probabilistes et d'approximation . . . 73

4 Gestion de la concurrence et synchronisation **81**
- 4.1 Processus . . . 82
  - 4.1.1 Programme et processus . . . 82
  - 4.1.2 Ordonnanceur . . . 83
  - 4.1.3 Ressources et état d'un fil d'exécution . . . 86
- 4.2 Threads (processus légers) . . . 88
  - 4.2.1 Principe . . . 88
  - 4.2.2 Utilisation des threads en `C` . . . 90

4.3 Concurrence . . . 95
4.3.1 Problèmes de synchronisation . . . 95
4.3.2 Atomicité . . . 97
4.3.3 Cohérence séquentielle . . . 98
4.3.4 Utilisation d'opérations élémentaires atomiques . . . 99
4.4 Mutex . . . 100
4.4.1 Notion de section critique . . . 100
4.4.2 Utilisation des mutex en `C` . . . 101
4.4.3 Interblocage . . . 103
4.5 Implémentation d'un mutex . . . 105
4.5.1 Implémentation d'un verrouillage simple . . . 106
4.5.2 Algorithme de Peterson . . . 108
4.5.3 Algorithme de la boulangerie de Lamport . . . 111
4.5.4 Attente active . . . 113
4.6 Sémaphore . . . 114
4.6.1 Principe . . . 114
4.6.2 Utilisation des sémaphores en `C` . . . 115
4.7 TD/TP - Gestion de la concurrence et synchronisation . . . 117

5 Algorithmes sur les graphes **131**
5.1 Algorithme de Kruskal . . . 132
5.1.1 Tri par fusion . . . 132
5.1.2 Structure *union-find* . . . 134
5.1.3 Arbres couvrants de poids minimal . . . 141
5.1.4 Algorithme de Kruskal . . . 142
5.2 Algorithme de Kosaraju . . . 147
5.2.1 Graphes orientés . . . 147
5.2.2 Algorithme de Kosaraju . . . 152
5.2.3 2-SAT . . . 154
5.3 Algorithme de couplage dans un graphe biparti . . . 158
5.3.1 Graphes bipartis . . . 159
5.3.2 Couplages . . . 159
5.3.3 Chemins augmentants . . . 162
5.3.4 Graphes augmentants . . . 165
5.3.5 Algorithme de recherche d'un couplage maximum . . . 166
5.4 TD/TP - Algorithmes sur les graphes . . . 169

6 Automates finis **175**
6.1 Automates finis déterministes . . . 176
6.1.1 Définitions . . . 176
6.1.2 Émondage . . . 179
6.1.3 Complétion . . . 182
6.2 Automates finis non déterministes . . . 184
6.2.1 Définitions . . . 184
6.2.2 Élimination des ε-transitions . . . 187
6.2.3 Déterminisation . . . 190
6.3 Implémentation . . . 193
6.4 TD/TP - Automates finis . . . 194

7 Intelligence artificielle **199**
7.1 Apprentissage supervisé . . . 200
7.1.1 Algorithme des k-plus proches voisins . . . 201
7.1.2 Arbres k-dimensionnels . . . 207
7.1.3 Arbres de décision . . . 211
7.1.4 Algorithme ID3 . . . 212
7.1.5 Surapprentissage . . . 215
7.1.6 Matrice de confusion . . . 216
7.2 Apprentissage non supervisé . . . 217
7.2.1 Classification hiérarchique ascendante . . . 218
7.2.2 Algorithme des k-moyennes . . . 220
7.3 TD/TP - Intelligence artificielle . . . 223

8 Automates finis (2) **227**
8.1 Théorème de Kleene . . . 228
8.1.1 Automates à la Thompson . . . 228
8.1.2 Langages locaux . . . 230
8.1.3 Automate de Glushkov par l'algorithme de Berry-Sethi . . . 236
8.1.4 Procédé d'élimination des états . . . 241
8.2 Classe des langages reconnaissables . . . 244
8.2.1 Stabilité . . . 244
8.2.2 Lemme de l'étoile . . . 246
8.3 TD/TP - Automates finis (2) . . . 249

9 Étude des jeux **253**
9.1 Graphes d'états . . . 254
9.2 Jeux d'accessibilité à deux joueurs . . . 254
9.2.1 Représentation d'un jeu . . . 254
9.2.2 Attracteurs . . . 257
9.3 Algorithme min-max . . . 259
9.3.1 Arbres de décision . . . 259
9.3.2 Algorithme min-max sans heuristique . . . 259
9.3.3 Heuristiques . . . 263
9.3.4 Algorithme min-max avec heuristique . . . 263
9.3.5 Élagage alpha-bêta . . . 266
9.4 Algorithme A^* . . . 267
9.5 TD/TP - Étude des jeux . . . 275

10 Grammaires non contextuelles **283**
10.1 Positionnement . . . 284
10.2 Grammaires et langages non contextuelles . . . 285
10.3 Arbres d'analyse . . . 289
10.4 Application : mini-calculatrice . . . 298
10.4.1 Analyse lexicale . . . 299
10.4.2 Analyse syntaxique . . . 300
10.4.3 Analyse sémantique . . . 302
10.5 TD/TP - Grammaires non contextuelles . . . 303

11 Déduction naturelle **307**
11.1 Principes . . . 308
11.2 Règles d'inférence et dérivations . . . 309
11.3 Déduction naturelle en logique propositionnelle . . . 312
11.3.1 Hypothèses . . . 312
11.3.2 Règles implicites . . . 313
11.3.3 Règles associées aux connecteurs logiques . . . 313
11.3.4 Règles dérivées . . . 317
11.3.5 Règles clés en logique classique . . . 319
11.4 Arbres de preuve . . . 321
11.5 Syllogismes . . . 323
11.6 Déduction naturelle en logique du premier ordre . . . 324
11.6.1 Substitutions . . . 327
11.6.2 Règles du quantificateur universel . . . 329
11.6.3 Règles du quantificateur existentiel . . . 330
11.7 Correction et complétude de la déduction naturelle . . . 332
11.8 Preuves de type . . . 335
11.8.1 Jugements de type . . . 335
11.8.2 Typages des constantes . . . 336
11.8.3 Types . . . 337
11.8.4 Fonctions . . . 338
11.8.5 Unification . . . 339
11.8.6 Déclarations locales . . . 341
11.9 TD/TP - Déduction naturelle . . . 342
12 Projets et problèmes **345**
12.1 Plus longue sous-séquence commune . . . 345
12.2 N reines . . . 356
12.3 MAX-2-SAT . . . 359
12.4 Sac à dos . . . 363
12.5 Ensembles de Julia . . . 372
12.6 Ordonnancement des tâches . . . 384
12.7 Algorithme de Prim . . . 389
12.8 Couverture de sommets . . . 395
12.9 Classification hiérarchique ascendante . . . 399
12.10 Jeu de Nim . . . 405
12.11 Calculatrice . . . 410
13 Solutions des exercices **415**
13.1 TD/TP - Langages réguliers . . . 415
13.2 TD/TP - Décidabilité et classes de complexité . . . 418
13.3 TD/TP - Algorithmes probabilistes et d'approximation . . . 428
13.4 TD/TP - Gestion de la concurrence et synchronisation . . . 437
13.5 TD/TP - Algorithmes sur les graphes . . . 451
13.6 TD/TP - Automates finis . . . 459
13.7 TD/TP - Intelligence artificielle . . . 469
13.8 TD/TP - Automates finis (2) . . . 473
13.9 TD/TP - Étude des jeux . . . 480
13.10 TD/TP - Grammaires non contextuelles . . . 487
13.11 TD/TP - Déduction naturelle . . . 494

14 Solutions des projets et problèmes **505**
14.1 Plus longue sous-séquence commune 505
14.2 N reines 515
14.3 MAX-2-SAT 519
14.4 Sac à dos 523
14.5 Ensembles de Julia 531
14.6 Ordonnancement des tâches 538
14.7 Algorithme de Prim 543
14.8 Couverture de sommets 548
14.9 Classification hiérarchique ascendante 552
14.10 Jeu de Nim 559
14.11 Calculatrice 563

Bibliographie **569**

Liste des figures **573**

Liste des tableaux **575**

Liste des codes **577**

Index **579**

Chapitre 1
Langages réguliers

Stephen C. Kleene pose les fondations de la notion de langage régulier [1] dans un article scientifique intitulé *Representation of Events in Nerve Nets and Finite Automata* [41], publié en 1956 dans *Automata Studies* sous la direction éditoriale de C. Shannon et J. McCarthy. Cependant, les résultats qui y sont présentés sont issus de travaux remontant à l'été 1951. La RAND Corporation, un centre de recherche stratégique étroitement lié à l'armée de l'air américaine, avait chargé Kleene de clarifier et de formaliser les résultats de W. McCulloch et W. Pitts publiés en 1943 [53].

McCulloch et Pitts proposent un modèle formel du fonctionnement neuronal fondé sur la logique booléenne, montrant que des réseaux de neurones idéalisés peuvent, en principe, effectuer tout raisonnement logique déductif. Cette approche est alors perçue comme une piste prometteuse pour explorer les potentialités de la cybernétique et établir des liens entre calcul logique, activité cérébrale et architectures de circuits : des problématiques alors perçues comme centrales pour les développements à venir en intelligence artificielle.

Stephen C. Kleene, qui est en 1951 un logicien reconnu, ancien doctorant d'Alonzo Church (1936), entreprend de reconstruire, selon les formalismes rigoureux de la logique, le fonctionnement des réseaux proposés par McCulloch et Pitts. Plutôt que de travailler directement sur le modèle neuronal, il définit les séquences d'activations symboliques observables dans ces réseaux. Il propose une définition inductive de ces événements réguliers, construits à partir d'événements élémentaires en utilisant trois opérations fondamentales : l'union, la concaténation et l'itération. Cette dernière, associé à l'opérateur $*$, a depuis été nommée étoile de Kleene, en son hommage [57].

> *« Nous allons maintenant décrire une classe d'événements que nous appellerons "événements réguliers". (Nous serons reconnaissants pour toute suggestion d'un terme plus explicite.)* [...] *Notre classe d'événements réguliers sera définie inductivement, en partant des événements définis, et en utilisant trois opérations : $E \vee F$, EF, E^*F ; c'est-à-dire qu'il s'agira de la plus petite classe contenant les événements définis, et fermée par ces opérations. »*
>
> Stephen C. Kleene, *Representation of events in nerve nets and finite automata*, [42] pages 46-48

L'impact de ce travail est considérable. Il jette les bases de la théorie des langages réguliers qui, sur le plan théorique, joue un rôle important dans le champ des langages formels [44], mais qui s'avère surtout essentiel pour le traitement des langages de programmation (compilateurs, interpréteurs, *etc.*) et qui est au cœur de la reconnaissance syntaxique et du traitement des expressions régulières. L'article de Kleene, écrit dans le contexte militaro-scientifique de la guerre froide, inaugure ainsi une forme de recherche qui a structuré en profondeur l'informatique contemporaine.

1. Appelé aussi *langage rationnel* dans la tradition mathématique française ; Kleene parlait quant à lui d'événements réguliers.

1.1 Alphabets, mots

Définition 1.1 - Alphabet, symbole

Un *alphabet*, noté Σ, est un ensemble fini et non vide de *symboles*, appelés aussi des *lettres*.

Exemple 1.1 Les ensembles suivants sont des exemples d'alphabets :

- ensemble des lettres minuscules de la langue française $\{a, b, c, \ldots, x, y, z\}$;
- ensemble des chiffres binaires $\{0, 1\}$;
- ensemble des caractères `ASCII` ;
- ensemble des bases nucléotidiques $\{\texttt{A}, \texttt{T}, \texttt{C}, \texttt{G}\}$ utilisés en biologie moléculaire ;
- ensemble des opérateurs arithmétiques de base $\{+, -, \times, \div\}$.

La notion d'alphabet est indépendante de la nature des symboles considérés, pourvu qu'ils forment un ensemble fini d'éléments distincts. □

Définition 1.2 - Mot

Un *mot* sur un alphabet Σ est une suite finie de $n \in \mathbb{N}$ symboles de Σ. La *longueur* d'un tel mot u est le nombre de symboles qui le composent, comptés avec multiplicité, notée $|u| = n$.

- Lorsque $n = 0$, on obtient le mot vide, noté ε.
- Lorsque $n > 0$, on note $u = u_0 \ldots u_{n-1}$ où $\forall i \in [\![0, n-1]\!]$, $u_i \in \Sigma$.

L'ensemble des mots non vides sur Σ est noté Σ^+. L'ensemble de tous les mots formés à partir de Σ est noté $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. L'ensemble de tous les mots de longueur n est noté Σ^n.

Dans un mot, les symboles sont ordonnés et peuvent apparaître plusieurs fois. Contrairement à un ensemble, la multiplicité et la position importent.

Remarque 1.1 Un mot vide n'appartient pas à un alphabet, c'est-à-dire quelque soit l'alphabet Σ choisi, $\varepsilon \notin \Sigma$. □

Remarque 1.2 $\forall n \in \mathbb{N}^*$, on a $\Sigma^n \subseteq \Sigma^+ \subseteq \Sigma^*$. □

Exemple 1.2

- La représentation binaire `01101000` de la lettre « h » en code `ASCII` est un mot sur l'alphabet binaire.
- La séquence `C T G T C T G A` est un mot sur l'alphabet des bases nucléotidiques.
- La suite de notes `do`, `ré`, `mi` est un mot sur l'alphabet des notes de la gamme.
- Le mot ♣♣♡ appartient à l'alphabet $\{♣, ♢, ♡, ♠\}$.
- L'expression logique $(p \vee q) \wedge r$ peut être vue comme un mot sur l'alphabet $\{p, q, r, \vee, \wedge, (,)\}$.

□

Définition 1.3 - Concaténation des mots

Soient $u, v \in \Sigma^*$ deux mots avec $|u| = n$ et $|v| = m$. On appelle *concaténation* de u et de v le mot noté $u \cdot v$ (ou uv) de longueur $n + m$, obtenu en écrivant u puis v à la suite.

- L'opération $\cdot$ est une loi de composition interne : $\forall u, v \in \Sigma^*, u \cdot v \in \Sigma^*$.
- Le mot vide ε est l'élément neutre de l'opération $\cdot$: $\forall u \in \Sigma^*, u \cdot \varepsilon = \varepsilon \cdot u = u$.
- L'opération $\cdot$ est associative : $\forall u, v, w \in \Sigma^*, u \cdot (v \cdot w) = (u \cdot v) \cdot w$.

On dit que $(\Sigma^*, \cdot)$ est un monoïde libre.

Remarque 1.3 L'opération de concaténation n'est pas commutative, donc en général, $u \cdot v \neq v \cdot u$. □

Remarque 1.4 L'opération de concaténation appliquée aux chaînes de caractères, coïncide avec la fonction de concaténation de `OCaml` qui est `^`. Cette fonction permet de combiner deux chaînes de caractères en une seule, en plaçant la seconde chaîne à la suite de la première. □

Grâce à l'opération de la concaténation $\cdot$, on peut définir la puissance d'un mot qui résulte de la concaténation d'un mot avec lui-même.

Définition 1.4 - Puissance d'un mot

Soit $u \in \Sigma^*$ et $p \in \mathbb{N}$. La *puissance* p de u est définie récursivement par :

$$\begin{cases} u^0 &= \varepsilon \\ u^p &= u \cdot u^{p-1} = u^{p-1} \cdot u, \quad \forall p \geqslant 1 \end{cases}$$

Définition 1.5 - Préfixe, suffixe, facteur

Soient $u, v \in \Sigma^*$ deux mots.

- u est *préfixe* de v s'il existe $w \in \Sigma^*$ tel que $v = u \cdot w$.
- u est *suffixe* de v s'il existe $w \in \Sigma^*$ tel que $v = w \cdot u$.
- u est *facteur* de v s'il existe $w, w' \in \Sigma^*$ tels que $v = w \cdot u \cdot w'$.

Le mot w utilisé dans la définition d'un préfixe et d'un suffixe est unique. En revanche, les mots w et w' utilisés dans la définition d'un facteur ne sont pas forcément uniques. On peut retrouver également plusieurs fois le même facteur à des emplacements différents dans un mot donné. Le mot vide ε est un préfixe, un suffixe et un facteur de tous les mots. De même, un mot est toujours un préfixe, un suffixe et un facteur de lui-même. On peut aussi dire qu'un préfixe ou un suffixe d'un mot est aussi un facteur de ce même mot.

Définition 1.6 - Sous-mot

Soit $u \in \Sigma^*$ un mot de longueur n. Le mot v est un *sous-mot* de u de longueur $m \leqslant n$, si et seulement si, il existe une fonction $\sigma : [\![0, m-1]\!] \to [\![0, n-1]\!]$ strictement croissante telle que $v = u_{\sigma(0)} \dots u_{\sigma(m-1)}$.

On peut remarquer que ε est le sous-mot de tous les mots et que chaque mot est un sous-mot de lui-même. De plus, si u est un facteur de v alors il est aussi un sous-mot de v, mais la réciproque est fausse.

Exemple 1.3 Soit $\Sigma = \{a, b, c\}$ un alphabet avec trois symboles. On pose $u = bcac$ et $v = ab$.

- $u \cdot v = bcacab$ et $v \cdot u = abbcac$.
- $u^0 = \varepsilon, v^3 = ababab$.
- bc est un préfixe de u.
- cac est un suffixe de u.
- ca est un facteur de u.
- L'ensemble des préfixes de u est $\{\varepsilon, b, bc, bca, bcac\}$.
- ba est un sous-mot de u mais il n'est pas un facteur de u.

□

1.2 Langages

Définition 1.7 - Langage

Un *langage* $\mathcal{L}$ sur un alphabet Σ est un sous-ensemble de ses mots : $\mathcal{L} \subseteq \Sigma^*$.

Remarque 1.5 Un langage étant un ensemble, un langage vide, noté $\varnothing$, est un langage qui ne contient aucun mot. En revanche, le langage $\{\varepsilon\}$ est un langage singleton, c'est-à-dire il contient un seul mot qui est le mot vide. □

Exemple 1.4

- $\mathcal{L}_1 = \{bcac, ab\}$ sur $\Sigma = \{a, b, c\}$.
- $\mathcal{L}_2$ est l'ensemble des mots qui se terminent par 0 sur l'alphabet binaire.
- $\mathcal{L}_3 = \{a^p \mid p \text{ est impair}\}$ sur $\Sigma = \{a\}$.
- $\mathcal{L}_4$ est le langage des palindromes sur $\Sigma = \{a, b\}$.
- $\mathcal{L}_5 = \Sigma^*$ sur un alphabet Σ.

Un langage peut donc avoir une cardinalité infinie. □

Il y a deux types d'opérations possibles sur les langages : les opérations ensemblistes usuelles comme l'union, l'intersection, le complémentaire à Σ^*, *etc.* et l'extension des opérations définies sur les mots comme la concaténation.

Définition 1.8 - Union, intersection des langages

Soient $\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages. L'*union* $\mathcal{L}_1 \cup \mathcal{L}_2$ et l'*intersection* $\mathcal{L}_1 \cap \mathcal{L}_2$ sont deux ensembles définis par :

$$\begin{aligned} \mathcal{L}_1 \cup \mathcal{L}_2 &= \{u \mid u \in \mathcal{L}_1 \text{ ou } u \in \mathcal{L}_2\} \\ \mathcal{L}_1 \cap \mathcal{L}_2 &= \{u \mid u \in \mathcal{L}_1 \text{ et } u \in \mathcal{L}_2\} \end{aligned}$$

Les opérations de l'union et de l'intersection de deux langages sont associatives et commutatives. L'élément neutre de l'union est le langage vide $\varnothing$ et l'élément absorbant est Σ^*. En revanche, pour l'intersection c'est l'inverse, l'élément neutre de l'intersection est Σ^* et l'élément absorbant de l'intersection est le langage vide $\varnothing$.

Remarque 1.6 Si deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$ sont définis sur des alphabets distincts Σ_1 et Σ_2, on peut les considérer comme des langages sur l'alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$ afin d'y appliquer les opérations ensemblistes. □

Définition 1.9 - Complémentaire d'un langage

Soit $\mathcal{L}$ un langage défini sur Σ. Le *complémentaire* de $\mathcal{L}$ est l'ensemble $\overline{\mathcal{L}}$ défini par :

$$\overline{\mathcal{L}} = \Sigma^* \backslash \mathcal{L}$$

Définition 1.10 - Concaténation de deux langages

Soient $\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages définis sur Σ. La concaténation de $\mathcal{L}_1$ et $\mathcal{L}_2$ est l'ensemble défini par :

$$\mathcal{L}_1\mathcal{L}_2 = \{u \cdot v \mid u \in \mathcal{L}_1 \text{ et } v \in \mathcal{L}_2\}$$

L'opération de concaténation de deux langages est associative. L'élément neutre de cette opération est l'ensemble $\{\varepsilon\}$ et l'élément absorbant est le langage vide $\varnothing$. La concaténation est également distributive par rapport à l'union :

$$\begin{aligned} \forall \mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3 \in \mathcal{P}(\Sigma^*), \quad \mathcal{L}_1(\mathcal{L}_2 \cup \mathcal{L}_3) &= (\mathcal{L}_1\mathcal{L}_2) \cup (\mathcal{L}_1\mathcal{L}_3) \\ (\mathcal{L}_1 \cup \mathcal{L}_2)\mathcal{L}_3 &= (\mathcal{L}_1\mathcal{L}_3) \cup (\mathcal{L}_2\mathcal{L}_3) \end{aligned}$$

Remarque 1.7 La concaténation n'est pas distributive par rapport à l'intersection. En effet, on peut donner un contre-exemple avec $\mathcal{L}_1 = \{\varepsilon, a\}$, $\mathcal{L}_2 = \{ab, ba\}$ et $\mathcal{L}_3 = \{aab\}$ définis sur $\Sigma = \{a, b\}$. D'une part on a $\mathcal{L}_1(\mathcal{L}_2 \cap \mathcal{L}_3) = \varnothing$, d'autre part $\mathcal{L}_1\mathcal{L}_2 \cap \mathcal{L}_1\mathcal{L}_3 = \mathcal{L}_3$ et le langage $\mathcal{L}_3$ n'est pas vide. □

Grâce à l'opération de concaténation de deux langages, on peut définir la puissance d'un langage qui résulte de la concaténation d'un langage avec lui-même.

Définition 1.11 - Puissance d'un langage

Soit $\mathcal{L}$ un langage et $p \in \mathbb{N}$. La *puissance* p de $\mathcal{L}$ est l'ensemble défini récursivement par :

$$\begin{cases} \mathcal{L}^0 &= \{\varepsilon\} \\ \mathcal{L}^p &= \mathcal{L}\mathcal{L}^{p-1} = \mathcal{L}^{p-1}\mathcal{L}, \quad \forall p \geqslant 1 \end{cases}$$

Définition 1.12 - Étoile de Kleene

Soit $\mathcal{L}$ un langage. L'*étoile de Kleene* de $\mathcal{L}$ est l'ensemble défini par :

$$\mathcal{L}^* = \bigcup_{p \geqslant 0} \mathcal{L}^p$$

L'étoile de Kleene est un opérateur qui permet de générer l'ensemble des mots possibles sur un alphabet donné, incluant toujours le mot vide ε. C'est l'ensemble des combinaisons finies de symboles de Σ.

Exemple 1.5 Soit $\Sigma = \{a, b\}$, l'alphabet sur lequel sont définis les deux langages $\mathcal{L}_1 = \{a, ab\}$ et $\mathcal{L}_2 = \{b, ba\}$.

- Union. $\mathcal{L}_1 \cup \mathcal{L}_2 = \{a, ab, b, ba\}$
- Concaténation.
  - $\mathcal{L}_1\mathcal{L}_2 = \{a \cdot b, a \cdot ba, ab \cdot b, ab \cdot ba\} = \{ab, aba, abb, abba\}$
  - $\mathcal{L}_2\mathcal{L}_1 = \{b \cdot a, b \cdot ab, ba \cdot a, ba \cdot ab\} = \{ba, bab, baa, baab\}$
- Étoile de Kleene.
  - $\mathcal{L}_1^0 = \{\varepsilon\}, \quad \mathcal{L}_1^1 = \mathcal{L}_1, \quad \mathcal{L}_1^2 = \{aa, aab, aba, abab\}$,
    $\mathcal{L}_1^3 = \{aaa, aaab, aaba, aabab, abaa, abaab, ababa, ababab\}$, *etc.* donc on a :

$$\mathcal{L}_1^* = \{\varepsilon, a, ab, aa, aab, aba, abab, aaa, aaab, aaba, \dots\}$$

  - $\mathcal{L}_2^0 = \{\varepsilon\}, \quad \mathcal{L}_2^1 = \mathcal{L}_2, \quad \mathcal{L}_2^2 = \{bb, bba, bab, baba\}$,
    $\mathcal{L}_2^3 = \{bbb, bbba, bbab, bbaba, babb, babba, babab, bababa\}$, *etc.* donc on a :

$$\mathcal{L}_2^* = \{\varepsilon, b, ba, bb, bba, bab, baba, bbb, bbba, bbab, \dots\}$$

  $\mathcal{L}_1^*$ inclut tous les mots formés par zéro ou plusieurs répétitions des mots de $\mathcal{L}_1$, de même pour $\mathcal{L}_2^*$. □

Remarque 1.8

- $\varnothing^* = \{\varepsilon\}$
- Pour $\mathcal{L} = \{u\}$, $\mathcal{L}^* = \{u^n \mid n \in \mathbb{N}\}$.
- Si $\mathcal{L}$ contient un mot non vide alors $|\mathcal{L}^*|$ est infini.

□

1.3 Langages réguliers, expressions régulières

1.3.1 Langages réguliers

Définition 1.13 - Langage régulier

L'ensemble des *langages réguliers* (ou *rationnels*) sur Σ, noté REG_Σ, est défini inductivement par les règles suivantes :

- le langage vide $\varnothing$ est un langage régulier ;
- $\forall \sigma \in \Sigma$, $\{\sigma\}$ est un langage régulier ;
- si $\mathcal{L}_1$ et $\mathcal{L}_2$ sont des langages réguliers alors $\mathcal{L}_1 \cup \mathcal{L}_2$, $\mathcal{L}_1\mathcal{L}_2$ et $\mathcal{L}_1^*$ sont des langages réguliers.

Définition 1.14 - Opération régulière

Les opérations d'union, de concaténation et d'étoile de Kleene sont appelées *opérations régulières*.

Un langage régulier est donc construit à partir des langages de base (l'ensemble vide $\varnothing$ et les singletons $\{\sigma\}$ pour $\sigma \in \Sigma$) en appliquant un nombre fini d'opérations régulières. Autrement dit, les langages réguliers constituent le plus petit ensemble de langages contenant ces cas de base et fermé sous ces opérations.

Exemple 1.6 Le langage $\mathcal{L} = \{a^n b^m \mid n, m \in \mathbb{N}\}$ est un langage régulier. Les langages $\mathcal{L}_1 = \{a^n \mid n \in \mathbb{N}\} = \{a\}^*$ et $\mathcal{L}_2 = \{b^m \mid m \in \mathbb{N}\} = \{b\}^*$ sont réguliers : les singletons sont des langages réguliers et l'étoile de Kleene appliquée à ces singletons fournit encore des langages réguliers. La concaténation des langages précédents, $\mathcal{L} = \mathcal{L}_1\mathcal{L}_2$, reste également un langage régulier. □

Proposition 1.1

Tout langage fini est régulier.

Démonstration Un langage fini sur Σ est défini par un ensemble fini de mots : soit $\mathcal{L} = \varnothing$, donc régulier, soit $\mathcal{L} = \{u_0, \ldots, u_{k-1}\}$ avec $k \in \mathbb{N}^*$. Or, dans ce cas, on peut écrire $\mathcal{L} = \{u_0\} \cup \cdots \cup \{u_{k-1}\}$ où $\forall i \in [\![0, k-1]\!]$, $u_i = \sigma_{i0} \ldots \sigma_{in_i}$ avec $\sigma_{ij} \in \Sigma$. Chaque $\{u_i\}$ est régulier, étant la concaténation des langages singletons. Ainsi, le langage $\mathcal{L}$ est régulier comme union des langages réguliers. □

Exemple 1.7 Le langage $\mathcal{L} = \{a^n b^n \mid n \in \mathbb{N}\}$ n'est pas régulier (voir la proposition 8.12, page 247). Ce langage est défini par tous les mots qui contiennent n symboles a suivis de n symboles b, pour un n quelconque. Les mots ε, ab, $aabb$ et $aaaabbbb$ appartiennent à ce langage. En revanche, les mots a, b, abb n'y appartiennent pas. Reconnaître ce langage nécessite de se souvenir du nombre de a pour le comparer au nombre de b. Le langage $\mathcal{L}$ fait partie des *langages non contextuels* ; en

cela que, ses mots peuvent être reconnus par des *grammaires non contextuelles* (voir le chapitre 10 Grammaires non contextuelles, page 283) ainsi que par des automates à pile (hors programme de MPI). □

Exemple 1.8 Soient $\Sigma = \{(,)\}$ et $\mathcal{L}$ le langage des parenthèses bien formées défini sur Σ par :

$$\mathcal{L} = \{u \in \Sigma^* \mid u \text{ contient des parenthèses correctement appariées.}\}$$

$\mathcal{L}$ n'est pas régulier (voir l'exercice 69, page 250 pour la démonstration). Les mots « () », « (()) » et « ()() » appartiennent à ce langage mais les mots « (», «) » ou encore « (() » n'y appartiennent pas. La propriété de ce langage exige de maintenir un comptage équilibré des parenthèses ouvrantes et fermantes ainsi que leur position relative, ce qui nécessite une mémoire non bornée. □

Remarque 1.9 Les langages réguliers sont énumérables, c'est-à-dire qu'il existe un algorithme capable de générer chacun des mots du langage un par un, sans en omettre aucun. □

1.3.2 Expressions régulières

Les expressions régulières sont des notations compactes qui permettent de spécifier des ensembles de mots. Le formalisme utilisé décrit précisément les motifs qu'il représente. En informatique, les motifs sont utilisés pour rechercher, correspondre et manipuler des séquences de symboles dans diverses applications, telles que les recherches textuelles, le traitement de texte et l'analyse des données.

Définition 1.15 - Expression régulière

L'ensemble des *expressions régulières* sur Σ, noté $\mathcal{E}_\Sigma$, est défini inductivement par les règles suivantes :

- $\varnothing$ est une expression régulière qui représente le langage vide ;
- ε est une expression régulière qui représente le langage contenant uniquement le mot vide ;
- $\forall \sigma \in \Sigma$, σ est une expression régulière qui représente le langage $\{\sigma\}$;
- si e, e_1 et e_2 sont des expressions régulières alors $e_1|e_2$, e_1e_2 et e^* sont des expressions régulières, représentant respectivement l'union, la concaténation et l'étoile de Kleene.

L'étoile de Kleene est prioritaire par rapport à la concaténation, qui est elle-même prioritaire par rapport à l'union. Les parenthèses sont utilisés pour modifier l'ordre de priorité des opérations et donc pour regrouper des parties d'une expression régulière afin de préciser l'ordre dans lequel les opérations doivent être évaluées.

Remarque 1.10 En informatique, le terme *regexp* est une abréviation de *regular expression* qui veut dire « expression régulière » en anglais. □

Définition 1.16 - Langage d'une expression régulière

On note $\mathfrak{L}$, la fonction qui associe à une expression régulière un langage régulier. Cette fonction est définie par :

$$\begin{array}{rcccccl}
\mathfrak{L} & : & \mathcal{E}_\Sigma & \to & \textsc{Reg}_\Sigma & & \\
 & & \varnothing & \mapsto & \mathfrak{L}(\varnothing) & = & \varnothing \\
 & & \varepsilon & \mapsto & \mathfrak{L}(\varepsilon) & = & \{\varepsilon\} \\
 & \forall \sigma \in \Sigma, & \sigma & \mapsto & \mathfrak{L}(\sigma) & = & \{\sigma\} \\
 & & e_1|e_2 & \mapsto & \mathfrak{L}(e_1|e_2) & = & \mathfrak{L}(e_1) \cup \mathfrak{L}(e_2) \\
 & & e_1e_2 & \mapsto & \mathfrak{L}(e_1e_2) & = & \mathfrak{L}(e_1)\mathfrak{L}(e_2) \\
 & & e^* & \mapsto & \mathfrak{L}(e^*) & = & (\mathfrak{L}(e))^*
\end{array}$$

Si $\mathcal{L} = \mathfrak{L}(e)$, alors on dit que e *dénote* le langage régulier $\mathcal{L}$.

Les expressions régulières permettent donc de décrire de manière concise des ensembles de mots possédant une structure spécifique.

Exemple 1.9 Dans l'alphabet binaire, l'expression régulière (0|1)*1 décrit la représentation binaire des nombres impairs et l'expression régulière (0|1)*00 celle des nombres divisibles par 4. □

Exemple 1.10 Soit $\Sigma = \{a, b, c\}$.

expression régulière	langage régulier
a	$\{a\}$
$b\|c$	$\{b, c\}$
ac	$\{ac\}$
$a\|bc$	$\{a, bc\}$
$(ab)^*$	$\{ab\}^* = \{\varepsilon, ab, abab, ababab, \dots\}$
$ab\|c^*$	$\{ab\} \cup \{c\}^* = \{ab, \varepsilon, c, cc, ccc, \dots\}$

Le tableau ci-dessus récapitule la correspondance entre certaines expressions régulières et les langages réguliers qu'elles dénotent. □

Définition 1.17 - Équivalence des expressions régulières

Deux expressions régulières sont dites *équivalentes* si elles dénotent le même langage.

Il existe des règles d'équivalences concernant les expressions régulières dont certaines sont énumérées dans le tableau 1.1, page 10, avec e, e_1, e_2 et e_3 quatre expressions régulières définies dans le même alphabet. Ces règles d'équivalences permettent de démontrer des équivalences entre des expressions régulières.

Équivalence	**Règle**
$e_1\|(e_2\|e_3) \equiv (e_1\|e_2)\|e_3$	associativité de l'union
$e_1\|e_2 \equiv e_2\|e_1$	commutativité de l'union
$\varnothing\|e \equiv e\|\varnothing \equiv e$	élément neutre de l'union
$e\|e \equiv e$	idempotence de l'union
$e_1(e_2e_3) \equiv (e_1e_2)e_3$	associativité de la concaténation
$e\varepsilon \equiv \varepsilon e \equiv e$	élément neutre de la concaténation
$\varnothing e \equiv e\varnothing \equiv \varnothing$	élément absorbant de la concaténation
$e_1(e_2\|e_3) \equiv e_1e_2\|e_1e_3$	distributivité de la concaténation sur l'union
$(e_1\|e_2)e_3 \equiv e_1e_3\|e_2e_3$	distributivité de la concaténation sur l'union
$\varepsilon\|ee^* \equiv \varepsilon\|e^*e \equiv e^*$	étoile de Kleene
$\varnothing^* \equiv \varepsilon$	étoile de Kleene

Tableau 1.1 – Règles d'équivalence des expressions régulières

Exemple 1.11 Soit $\sigma \in \Sigma$.

$$\begin{array}{rcll} (\varepsilon|\sigma)(\varepsilon|\sigma) & \equiv & \varepsilon(\varepsilon|\sigma)|\sigma(\varepsilon|\sigma) & \text{distributivité de la concaténation sur l'union} \\ & \equiv & \varepsilon\varepsilon|\varepsilon\sigma|\sigma\varepsilon|\sigma\sigma & \text{distributivité de la concaténation sur l'union} \\ & \equiv & \varepsilon|\sigma|\sigma|\sigma\sigma & \text{élément neutre de la concaténation} \\ & \equiv & \varepsilon|\sigma|\sigma\sigma & \text{idempotence de l'union} \end{array}$$

L'équivalence $(\varepsilon|\sigma)(\varepsilon|\sigma) \equiv \varepsilon|\sigma|\sigma\sigma$ peut donc être démontrée par cette suite d'équivalences. □

En `OCaml`, on peut utiliser les types inductifs pour représenter les expressions régulières conformément à la structure inductive de ces dernières. Par exemple, si on souhaite manipuler des séquences nucléotides `A`, `T`, `C` et `G`, on pourrait définir les types `alphabet`, `word` et `regexp` comme dans le code 1.1.

```
type symbol = A | T | C | G
type word = symbol list
type regexp =
  | Epsilon
  | Symbol of symbol
  | Union of regexp * regexp
  | Concat of regexp * regexp
  | Kleene of regexp
```

Code 1.1 – `[OCaml]` Implémentation des expressions régulières

Le type `symbol` représente les symboles de base (ici les nucléotides), tandis qu'un mot de type `word` est simplement une liste de ces symboles. Enfin, le type `regexp` permet de décrire les expressions régulières en combinant ces symboles au moyen des opérations.

1.3.3 Expressions régulières étendues

Une expression régulière étendue est une version plus flexible des expressions régulières classiques. Elles sont utilisées pour décrire des motifs de texte avec une syntaxe et des fonctionnalités supplémentaires qui rendent leur utilisation plus expressive et concise.

La commande `grep` est un utilitaire de ligne de commande Unix utilisée pour rechercher des motifs spécifiques dans des fichiers texte, *via* des expressions régulières pour effectuer des correspondances complexes. La norme POSIX est un ensemble de standards qui garantit la compatibilité et l'interopérabilité des systèmes d'exploitation, incluant des spécifications pour les différents utilitaires. En respectant les directives POSIX, `grep` et ses variantes, comme `egrep` (ou `grep -E`) pour les expressions régulières étendues, offrent une interface portable pour la manipulation de texte à travers différents systèmes Unix.

Dans la norme POSIX, le mot vide ε n'existe pas en tant qu'entité explicite mais peut être représenté implicitement dans des expressions régulières en utilisant des constructions qui permettent des correspondances optionnelles ou des alternatives vides. Par exemple `a|` correspond au symbol a ou au mot vide.

Il existe aussi deux autres opérateurs qui gèrent les mots vides de façon indirecte. Si e est une expression régulière alors ee^* correspond à une ou plusieurs occurrences de cette expression, tandis que $\varepsilon|e$ correspond à zéro ou une occurrence. Dans les expressions régulières étendues, les opérateurs `+` et `?` permettent de représenter ces motifs de manière plus concise. Par exemple, a`+` définit le langage $\{a, aa, aaa, \dots\}$ et a`?` définit le langage $\{\varepsilon, a\}$.

De la même manière, si $e_1, e_2, \dots, e_k$ sont k expressions régulières, plutôt que d'écrire $e_1|e_2|\dots|e_k$, dans les expressions régulières étendues, on peut utiliser la notation `[`$e_1e_2\dots e_k$`]`. Cette notation permet de définir également `[^`$e_1e_2\dots e_k$`]` qui correspond à tout symbole sauf ceux spécifiés. Par exemple `[`abc`]` correspond à a, b ou c ; tandis que `[^`abc`]` correspond à tous symboles sauf a, b et c.

Les expressions régulières étendues utilisent également deux ancrages : `^` pour le début d'une ligne et `$` pour la fin d'une ligne.

Exemple 1.12 Une commande qui permet de rechercher les adresses email dans un fichier fourni est donnée ci-dessous.

```
1 $ grep -E -o '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9-]+\.[a-zA-Z]{2,}' text.
    txt > emails.txt
```

L'option `-E` de `grep` indique d'utiliser les expressions régulières étendues. L'option `-o` indique d'afficher uniquement les parties des lignes qui correspondent à l'expression régulière, plutôt que d'afficher les lignes complètes contenant les correspondances. On cherche l'expression régulière, donnée entre guillemets simples, dans le fichier `text.txt` et on redirige la sortie vers un autre fichier `emails.txt`.

L'expression régulière étendue recherchée contient plusieurs parties.

- `[a-zA-Z0-9._%+-]+` correspond à une ou plusieurs occurrences de lettres majuscules ou minuscules, chiffres, points, underscores, pourcentages, plus ou tirets. C'est le nom d'utilisateur.
- `@` correspond au symbole obligatoire arobase des adresses emails.
- `[a-zA-Z0-9-]+` correspond à une ou plusieurs occurrences de lettres majuscules ou minuscules, chiffres, ou tirets. C'est le nom de domaine.
- `\.` correspond au symbole point litéral, obligatoire dans une adresse mail. Comme le point seul représente un caractère unique quelconque, on ajoute le \ pour qu'il corresponde au caractère point en tant que tel.
- Enfin, `[a-zA-Z]{2,}` correspond à deux ou plus lettres majuscules ou minuscules. C'est l'extension.

Cette commande permet donc d'extraire toutes les adresses email d'un fichier texte et de les stocker dans un autre fichier. □

1.4 TD/TP - Langages réguliers

Exercice 1 : Construction de langages ⇝ *p.415*

Soit $\Sigma = \{x\}$. On pose $\forall k \in \mathbb{N}, \mathcal{L}_k = \left\{x^{5n+k}, n \in \mathbb{N}\right\}$.

1) Déterminer $\mathcal{L}_k \cap \mathcal{L}_j$.

2) Déterminer $\mathcal{L}_k\mathcal{L}_j$.

Exercice 2 : Équivalence des expressions régulières ⇝ *p.415*

Soit $\Sigma = \{a, b\}$.

1) Trouver une expression régulière plus simple et équivalente à $(\varepsilon|a)^*$.

2) Démontrer que $(ab)^*a \equiv a(ba)^*$.

3) Démontrer que $(a^*b)^*a^* \equiv a^*(ba^*)^*$. Donner une expression régulière équivalente plus simple.

4) Démontrer que $(b|\varepsilon)(ab)^*(a|\varepsilon) \equiv b(ab)^*a|b(ab)^*|(ab)^*a|(ab)^*$. Décrire le langage dénoté par ces expressions régulières.

Exercice 3 : Langage décrit par une expression régulière ⇝ *p.416*

Soit $\Sigma = \{0, 1\}$. On s'intéresse aux langages suivants qui sont dénotés par les expressions régulières indiquées :

- $\mathcal{L}_1 = \mathfrak{L}((0|10)^*)$
- $\mathcal{L}_2 = \mathfrak{L}(0(00|1(01)^*1)^*1)$
- $\mathcal{L}_3 = \mathfrak{L}((1|01^*0)^*)$
- $\mathcal{L}_4 = \mathfrak{L}(1^*(0(11)^*)^*1^*)$
- $\mathcal{L}_5 = \mathfrak{L}((0|1)^*10)$

1) Donner tous les mots de longueurs 0, 1, 2, 3 et 4 des langages $\mathcal{L}_1$ et $\mathcal{L}_2$.

2) Soit $\mathcal{L}$ un langage dont tous les mots sont de longueur paire. Montrer que tous les mots de $\mathcal{L}^*$ sont aussi de longueur paire.

3) En déduire que tous les mots de $\mathcal{L}_2$ sont de longueur paire.

4) Décrire les langages $\mathcal{L}_3$, $\mathcal{L}_4$ et $\mathcal{L}_5$.

Exercice 4 : Expression régulière d'un langage ⇝ *p.416*

Donner une expression régulière qui dénote chacun des langages suivants définis sur $\Sigma = \{0, 1\}$.

- $\mathcal{L}_1$: Ensemble des mots dont le suffixe est 000.
- $\mathcal{L}_2$: Ensemble des mots dont le préfixe est 01 et le suffixe est 1.
- $\mathcal{L}_3$: Ensemble des mots comportant le facteur 11.
- $\mathcal{L}_4$: Ensemble des mots ne comportant pas le facteur 11.

Exercice 5 : ★ Code ⇝ *p.416*

Un *code* sur un alphabet Σ est un langage $\mathcal{L}$ qui vérifie $u_1u_2\dots u_p = v_1v_2\dots v_q$ avec :

$$\forall i \in [\![1,p]\!]\,, \forall j \in [\![1,q]\!]\,,\ u_i, v_j \in \mathcal{L} \implies p = q \text{ et } \forall i \in [\![1,p]\!]\,,\ u_i = v_i$$

Toute factorisation d'un mot de $\mathcal{L}^*$ est unique.

Soient $\Sigma = \{0,1,2\}$ et $\mathcal{L}_1 = \{102, 02, 111, 11\}$, $\mathcal{L}_2 = \{01, 102, 111, 202\}$.

1) Montrer que $\mathcal{L}_1$ n'est pas un code.

2) Justifier que $\mathcal{L}_2$ est un code.

3) Construire un code $\mathcal{L}_3$ de 5 mots sur Σ.

4) Démontrer qu'un langage $\mathcal{L}$ où aucun mot n'est le préfixe propre d'un autre mot est un code. Rappeler un code déjà vu qui vérifie cette propriété.

5) Démontrer que si tous les mots de $\mathcal{L}$ ont la même longueur alors $\mathcal{L}$ est un code. Rappeler un code déjà vu qui vérifie cette propriété.

6) Soient $u, v \in \mathcal{L}$ deux mots distincts. Démontrer que $\{u, v\}$ est un code si et seulement si $uv \neq vu$.

Exercice 6 : Lemme de Levi ⇝ *p.417*

Lemme de Levi. Soit Σ un alphabet et $u_1, u_2, v_1, v_2 \in \Sigma^*$ tels que $u_1v_1 = u_2v_2$. Alors il existe un mot $x \in \Sigma^*$ tel qu'une des deux affirmations suivantes est vraie :

- $u_1 = u_2\,x \quad$ et $\quad x\,v_1 = v_2$
- $u_2 = u_1\,x \quad$ et $\quad x\,v_2 = v_1$

Le lemme de Levi dit qu'il est toujours possible d'ajuster deux découpages possibles d'un mot en insérant une partie intermédiaire qui permet de relier l'un à l'autre. Il existe un chevauchement qui relie les deux découpages.

Corollaire 1. Si u_1 et u_2 sont deux préfixes de w, alors u_1 est préfixe de u_2 ou u_2 est préfixe de u_1.

Corollaire 2. Si v_1 et v_2 sont deux suffixes de w, alors v_1 est suffixe de v_2 ou v_2 est suffixe de v_1.

1) Démontrer le lemme de Levi.

2) Démontrer le corollaire 1.

3) Démontrer le corollaire 2.

On s'intéresse à la propriété suivante : pour deux mots $u, v \in \Sigma^*$ tels que $uv = vu$, u et v sont des puissances d'un même mot w. On note $P(n)$ le prédicat pour $|uv| = n$:

$$(uv = vu) \iff (\exists w \in \Sigma^*, \exists p, q \in \mathbb{N} \text{ tels que } u = w^p \text{ et } v = w^q)$$

4) Démontrer le sens direct ($\implies$) de $P(n)$.

5) Démontrer le sens indirect ($\impliedby$) de $P(n)$.

Exercice 7 : Évaluation d'une expression régulière ⇝ *p.417*

Dans cet exercice, on travaille avec $\Sigma = \{\texttt{A}, \texttt{T}, \texttt{C}, \texttt{G}\}$ et les expressions régulières qui dénotent un langage sur Σ^*. Le langage de programmation à utiliser est `OCaml`. Les types inductifs `symbol`, `word` et `regexp` sont définis dans le code 1.1, page 10.

1) Créer une variable contenant l'expression régulière (`A`|`T`)*`G`.

2) Écrire une fonction de signature

```
val split : word -> (word * word) list
```

qui prend en paramètre un mot et qui renvoie une liste de paires. Chaque paire contient deux sous-listes obtenues en scindant la liste à chaque position possible.

Un exemple de sortie dans l'interpréteur est donné ci-dessous.

```
# let (w:word) = [C;A;T;G;G];;
val w : word = [C; A; T; G; G]
# split w;;
- : (word * word) list =
[([C; A; T; G; G], []); ([C; A; T; G], [G]); ([C; A; T], [G; G]);
 ([C; A], [T; G; G]); ([C], [A; T; G; G]); ([], [C; A; T; G; G])]
```

3) Écrire une fonction de signature

```
val match_regexp : regexp -> word -> bool
```

qui prend en paramètres une expression régulière, un mot et qui renvoie un booléen indiquant si le mot appartient au langage décrit par cette expression régulière.

4) Dans une expression principale en `OCaml`, décrire une expression régulière et quatre mots, dont deux qui appartiennent au langage associé et dont deux qui n'y appartiennent pas. Un exemple de sortie attendu pour l'expression régulière (`A`|`T`)*`G` est le suivant.

```
$ ./is_in_language
Word TTG matches: true
Word ATAG matches: true
Word CG matches: false
Word ATTC matches: false
```

Chapitre 2
Décidabilité et classes de complexité

Au début du XX^{e} siècle, David Hilbert formule l'*Entscheidungsproblem* : « Existe-t-il une procédure mécanique permettant de déterminer, pour toute formule du calcul des prédicats, si elle est valide ? ». Cette interrogation conduit à la définition de la notion de *décidabilité*. Un problème de décision est dit décidable lorsqu'il existe un procédé effectif qui, en un nombre fini d'étapes déterministes, détermine systématiquement si une instance donnée doit être acceptée ou rejetée.

En 1936, Alan Turing établit que l'*Entscheidungsproblem* n'admet pas de solution générale. Pour cela il introduit un modèle formel de calcul : des machines depuis renommées *machines de Turing*. Ce faisant il formalise l'intuition « être calculable » [72]. La même année, Alonzo Church obtient un résultat analogue à partir du $\lambda-$calcul. Ces travaux fournissent une caractérisation mathématique de la *calculabilité*, entendue comme l'existence d'une procédure effective qui, pour toute instance d'un problème, termine après un nombre fini d'opérations déterministes et produit la solution associée. Ainsi, un problème de décision est décidable si et seulement si sa fonction caractéristique, qui renvoie par définition un booléen, est calculable.

Dans les années 1960, alors que les machines théoriques imaginées par Alan Turing étaient devenues des objets concrets, la question de l'efficience des calculs devint de plus en plus prépondérante. C'est à cette époque que Jack Edmonds explicite l'idée selon laquelle tout algorithme raisonnable produit une réponse en un temps polynomial de la taille de l'instance et utilise, au pire, une mémoire dont la taille est également polynomialement bornée.

> *« Pour des raisons pratiques, les détails computationnels sont essentiels. Néanmoins, mon objectif consiste à démontrer de la façon la plus rigoureuse possible l'existence d'un algorithme efficace. Selon le dictionnaire, « efficace » signifie « adéquat dans l'opération ou la performance ». C'est précisément ce sens que je souhaite invoquer ici, en admettant qu'il soit concevable que [mon problème] ne possède pas d'algorithme efficace.* [...] *Un algorithme fini existe bien sûr, mais sa complexité croît de manière exponentielle avec la taille de [l'instance du problème]. Il n'est pas aisé d'établir de façon décisive s'il existe un algorithme dont la complexité croîtrait uniquement de manière algébrique en fonction de cette taille. »*
>
> Jack Edmonds, *Paths, Trees, and Flowers*, [26] page 450.

Cette réflexion donne naissance à deux sous-classes de problèmes calculables. Les problèmes de décision appartenant à la classe **P** sont ceux qui peuvent être résolus en temps (et en espace) polynomial, tandis que la classe **NP** rassemble ceux qui ne connaissent pas (à ce jour) de solution en temps polynomial. Parmi ces derniers on identifie également les problèmes dits **NP**-difficiles, pour lesquels il est établi que « produire une solution en temps polynomial » impliquerait l'existance d'une solution polynomiale pour l'ensemble des problèmes de la classe **NP**.

2.1 Décidabilité

2.1.1 Notion de problème

Définition 2.1 - Problème

Soient $\mathbb{X}$ un ensemble d'entrées et $\mathbb{Y}$ un ensemble de solutions possibles. Un *problème* $\mathcal{P} \subseteq \mathbb{X} \times \mathbb{Y}$ est une relation vérifiant :

$$\forall x \in \mathbb{X},\ \exists y \in \mathbb{Y} \mid (x, y) \in \mathcal{P}$$

Un *algorithme* est une suite d'instructions structurées et finies, conçues pour résoudre un problème. Résoudre un problème consiste alors à proposer un algorithme qui, pour chaque entrée de l'ensemble $\mathbb{X}$, produit un résultat correct dans $\mathbb{Y}$. Les ensembles $\mathbb{X}$ et $\mathbb{Y}$ sont supposés dénombrables et munis d'une représentation finie permettant leur encodage effectif dans la machine (et donc, en dernière analyse, réductibles à une séquence finie de bits).

Exemple 2.1 On considère le problème de recherche d'un chemin entre deux sommets dans un graphe non orienté pondéré connexe.

PATH
In : $G = (V, E, W)$ un graphe non orienté pondéré connexe ; $v, w \in V$
Out : Un chemin entre les sommets v et w.

Ce problème peut être résolu par un parcours en largeur du graphe. □

Exemple 2.2 On considère le problème de décision portant sur la primalité d'un entier n donné.

PRIME
In : $n \in \mathbb{N}$
Out : n est-il un nombre premier ?

Ce problème peut être représenté par : $\mathbb{X} = \mathbb{N}$, $\mathbb{Y} = \mathbb{B} = \{0, 1\}$ ainsi que par l'ensemble des paires qui le définissent : $\{(0, 0), (1, 0), (2, 1), (3, 1), (4, 0), ...\} \subseteq \mathbb{N} \times \mathbb{B}$. Pour le résoudre, il est possible d'utiliser un algorithme naïf consistant à tester tous les entiers entre 2 et $\sqrt{n}$, l'algorithme déterministe AKS (Agrawal, Kayal, Saxena) ou encore un algorithme probabiliste tel que celui de Miller-Rabin. □

Les problèmes de décision seront particulièrement étudiés dans ce chapitre. Leur étude illustre de façon pertinente les limites de la calculabilité, c'est-à-dire ce qui peut être résolu par un algorithme. Elle permet également de classer la difficulté des tâches selon les ressources nécessaires à leur résolution, conduisant ainsi à l'analyse des classes de complexité telles que **P**, **NP** ou encore **NP**-complet.

Définition 2.2 - Problème de décision

Soient $\mathbb{X}$ un ensemble d'entrées et $\mathbb{B} = \{0, 1\}$. Un *problème de décision* est une relation $\mathcal{P} \subseteq \mathbb{X} \times \mathbb{B}$ vérifiant :

$$\forall x \in \mathbb{X},\ \exists! b \in \mathbb{B} \mid (x, b) \in \mathcal{P}$$

On appelle chacun des éléments $x \in \mathbb{X}$, une *instance*.
On note $\mathcal{P}^+$ l'ensemble des *instances positives* de $\mathcal{P}$: $\forall x \in \mathcal{P}^+,\ (x, 1) \in \mathcal{P}$.
On note $\mathcal{P}^-$ l'ensemble des *instances négatives* de $\mathcal{P}$: $\forall x \in \mathcal{P}^-,\ (x, 0) \in \mathcal{P}$.
De plus, $\mathcal{P}^+$ et $\mathcal{P}^-$ forment une partition de $\mathbb{X}$.

Un problème de décision revient à poser une question formelle pour laquelle la réponse attendue est soit « oui », soit « non ». Ces réponses peuvent être représentées par l'ensemble des booléens $\mathbb{B} = \{0, 1\}$. Un tel problème est défini par un ensemble d'entrées et une condition à vérifier pour chacune de ces entrées. La résolution d'un problème de décision consiste alors à déterminer, pour chaque entrée, si elle satisfait ou non cette condition.

Exemple 2.3 On considère le problème de décision Amicable suivant. Deux entiers a et b sont dits *amicaux* si chacun est égal à la somme des diviseurs propres de l'autre (les diviseurs strictement inférieurs à l'entier lui-même).

Amicable
In : $a, b \in \mathbb{N}$
Out : Les entiers a et b sont-ils amicaux ?

On note $s(n)$ la somme des diviseurs propres de n. Pour 220 et 284, on peut écrire :

$$\begin{aligned} s(220) &= 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284 \\ s(284) &= 1 + 2 + 4 + 71 + 142 = 220 \end{aligned}$$

Ce qui montre que $(220, 284)$ est une instance positive du problème Amicable, c'est-à-dire $(220, 284) \in \text{Amicable}^+$. □

Exemple 2.4 On considère le problème FunctionC suivant.

FunctionC
In : Une fonction écrite en C
Out : La fonction C est-elle syntaxiquement correcte ?

Ce problème est un problème de décision qui nécessite l'analyse syntaxique du code de la fonction fournie en entrée.

```c
int sum_of_squares(int n) {
  int sum = 0;
  for (int i = 1; i <= n; i++)
    sum += i * i;
  return sum;
}
```

Par exemple, une instance positive de ce problème est la fonction `sum_of_squares`. Le problème consiste à déterminer si elle est syntaxiquement correcte. Puisque la réponse est positive, on peut dire que `sum_of_squares` $\in$ FunctionC$^+$. □

Définition 2.3 - Problème d'optimisation

Soient $\mathbb{X}$ un ensemble d'entrées et $\mathbb{Y}$ un ensemble de solutions possibles. Un *problème d'optimisation* $\mathcal{P} \subseteq \mathbb{X} \times \mathbb{Y}$ est un problème qui est défini par une fonction objectif $f : \bar{\mathbb{Y}} \to \mathbb{R}$ où $\bar{\mathbb{Y}} \subseteq \mathbb{Y}$ est l'ensemble des solutions admissibles. Le but est de trouver une solution qui

- soit minimise la fonction objectif :

  Problème de minimisation $\mathcal{P}_{min}$
  In : $x \in \mathbb{X}$
  Out : $y_{min} \in \arg\min_{y \in \bar{\mathbb{Y}}} f(y)$

- soit maximise la fonction objectif :

  Problème de maximisation $\mathcal{P}_{max}$
  In : $x \in \mathbb{X}$
  Out : $y_{max} \in \arg\max_{y \in \bar{\mathbb{Y}}} f(y)$

Un problème de *minimisation* ou de *maximisation* est donc un problème d'optimisation visant à fournir une *solution optimale* y telle que $f(y)$ soit la *valeur optimale* sous les contraintes données.

Exemple 2.5 On considère le problème d'optimisation ShortestPath du plus court chemin dans un graphe non orienté pondéré connexe.

ShortestPath
In : $G = (V, E, W)$ un graphe non orienté pondéré connexe ; $v, w \in V$
Out : Le plus court chemin entre les sommets v et w.

Ce problème peut être résolu, entre autres, par l'algorithme de Dijkstra ou encore par l'algorithme de Floyd-Warshall vus en MP2I [34].

L'ensemble $\mathbb{Y}$ contient tous les chemins possibles dans le graphe G. L'ensemble $\bar{\mathbb{Y}}$ contient tous les chemins possibles entre les sommets v et w. La fonction objectif $f : \bar{\mathbb{Y}} \to \mathbb{R}$ associe à chaque chemin $y \in \bar{\mathbb{Y}}$ entre v et w, son coût. Si un chemin y est représenté par $v = v_0, \ldots, v_n = w$, alors on a :

$$f(y) = \sum_{i=0}^{n-1} W(v_i, v_{i+1})$$

Il faut donc chercher à minimiser cette fonction objectif. □

Il est possible de construire un problème de décision à partir d'un problème d'optimisation en introduisant un seuil sur la fonction objectif f. Il peut s'agir soit de minimiser, soit de maximiser la fonction f sur un ensemble de solutions admissibles $\bar{\mathbb{Y}}$. Pour obtenir un problème de décision, une valeur seuil $\tau \in \mathbb{R}$ est fixée et la question est reformulée de manière binaire : « Existe-t-il une solution $y \in \bar{\mathbb{Y}}$ telle que $f(y) \leqslant \tau$? » dans le cas d'un problème de minimisation, ou « Existe-t-il une solution $y \in \bar{\mathbb{Y}}$ telle que $f(y) \geqslant \tau$? » dans le cas d'un problème de maximisation. La réponse à ces problèmes de décision est alors « oui » ou « non », selon l'existence d'une solution satisfaisant la condition par rapport au seuil τ défini.

τ-$\mathcal{P}_{min}$
In : $x \in \mathbb{X}$, $\tau \in \mathbb{R}$
Out : Existe-t-il $y \in \bar{\mathbb{Y}}$ telle que $f(y) \leqslant \tau$?

τ-$\mathcal{P}_{max}$
In : $x \in \mathbb{X}$, $\tau \in \mathbb{R}$
Out : Existe-t-il $y \in \bar{\mathbb{Y}}$ telle que $f(y) \geqslant \tau$?

Les ensembles $\mathbb{X}$ et $\mathbb{Y}$ sont des ensembles au sens mathématique. En pratique, lors de la résolution des problèmes en informatique (sur un ordinateur ou une machine de calcul théorique), les entrées d'un algorithme et sa sortie, chacune d'un type fixé, sont des mots appartenant au langage $\{0,1\}^*$. En effet, la sérialisation [34] permet de représenter n'importe quelle structure de données sous forme d'une séquence de symboles, et donc, *in fine*, d'une série de bits.

Remarque 2.1 Si la fonction objectif a des valeurs dans $\mathbb{N}$ alors résoudre un problème d'optimisation revient à travailler plusieurs fois avec le problème de décision qui lui est associé. En revanche, si la fonction objectif a des valeurs dans $\mathbb{R}$, alors on a obligatoirement $\tau \in \mathbb{R}$ et donc la solution au problème d'optimisation peut être fournie sous la forme d'un intervalle de précision fixée. □

2.1.2 Modèles de calcul

Un modèle de calcul est la représentation formelle d'un système capable d'exécuter des algorithmes, qui ont initialement été conçus comme des procédures de calcul. Il définit comment les calculs peuvent être effectués, quelles opérations sont permises et comment les entrées sont transformées en sorties.

Deux modèles de calcul historiques servent de référence à tous les autres :

- la machine de Turing [72], dispositif théorique qui suit une table d'opérations prédéfinies pour traiter et modifier des symboles inscrits sur un ruban, de longueur infinie (alors que la table et l'ensemble des symboles sont finis) ;
- le λ-calcul [16, 17], formalisme purement syntaxique basé sur des règles de réduction de fonctions anonymes (applications de fonctions à des arguments) dont l'opération primitive est la substitution de variables.

L'existence de ces deux modèles, définis indépendamment mais dotés du même pouvoir de calcul, a conduit Church et Turing à formuler en 1936 la *thèse de Church-Turing* : toute fonction qui peut être calculée de manière effective par un procédé mécanique peut l'être par une machine de Turing ou, de façon équivalente, dans le λ-calcul.

Une machine de calcul est un modèle, abstrait ou physique, conçu pour effectuer des opérations. Elle se définit généralement par la présence des éléments fondamentaux suivants.

- *Un ensemble d'états :* la machine possède des états internes qui évoluent au fur et à mesure du calcul.
- *Un mécanisme de transition :* des règles ou instructions précisent de façon déterministe comment l'état courant se transforme en un nouvel état selon les données manipulées.
- *Une condition d'arrêt :* le calcul prend fin si la machine atteint un état terminal, elle produit alors un résultat. Cependant elle peut aussi ne jamais atteindre d'état terminal, rester active indéfiniment et en conséquence ne jamais présenter de résultat.

Définition 2.4 - Calculabilité d'un problème

Un problème est *calculable* (ou *effectivement calculable*) s'il existe une machine qui, pour toute donnée d'entrée, exprimée sous la forme d'une séquence finie de symboles, termine son exécution et produit en sortie une solution correcte de ce problème sous la forme d'une nouvelle séquence de symboles.

La calculabilité définit si un problème donné peut être calculé de manière systématique en un nombre fini d'étapes à l'aide d'un modèle de calcul. Ce dernier est une abstraction qui permet de formaliser et d'étudier ce qui est calculable ou non.

Exemple 2.6 Les automates finis déterministes (voir le chapitre 6 Automates finis, page 175) forment un modèle de calcul. Chaque machine est caractérisée par un quintuplet $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$. Les entrées sont des mots dans Σ^*. L'automate est exécuté sur $u \in \Sigma^*$ à partir de l'état initial q_0 avec les transitions définies par la fonction de transition δ et le calcul s'arrête dans un état $q \in Q$. Si $q \in F$, c'est-à-dire que l'automate s'est arrêté dans un état final, alors la machine renvoie 1 et sinon elle renvoie 0.

Ce sont des machines de calcul simples qui ne disposent que d'un nombre limité d'états et qui n'ont pas de mémoire supplémentaire pour stocker des données. Elles sont utilisées pour modéliser des processus simples et reconnaître des langages réguliers, mais leur pouvoir de calcul est limité par rapport à des modèles plus complexes comme la machine de Turing. Elles ne peuvent pas couvrir les problèmes de décision. Par exemple, dans la proposition 8.12, page 247, il est démontré que le langage $\mathcal{L} = \{a^n b^n \mid n \in \mathbb{N}\}$ n'est pas régulier, donc il ne peut pas être reconnu par un automate fini. □

Dans le cadre théorique, les machines de calcul sont utilisées pour étudier la *décidabilité* (ce qui peut être calculé) et la *complexité* (les ressources nécessaires pour effectuer ces calculs). Le modèle le plus puissant et connu est la machine de Turing, qui formalise l'idée de calculabilité dans un cadre général. En revanche, au regard du programme de MPI [27], cet ouvrage se limite à l'étude de modèles de calculs déterministes, conformément à la définition 2.5.

Définition 2.5 - Modèle de calcul

Un *modèle de calcul* est une machine à mémoire infinie qui exécute un programme, par exemple rédigé en `OCaml` ou en `C`. Toute entrée et toute sortie d'une machine peuvent, par sérialisation, se ramener à une séquence de bits, c'est-à-dire à un mot dans $\{0,1\}^*$. Une machine $\mathcal{M}$ peut alors être définie comme une fonction sur ce langage.

$$\begin{array}{rccc} \mathcal{M} & : & \{0,1\}^* & \to & \{0,1\}^* \\ & & u & \mapsto & \begin{cases} \mathcal{M}(u) \text{ si } \mathcal{M} \text{ s'arrête sur } u \\ \text{sinon comportement indéfini} \end{cases} \end{array}$$

Dans ce modèle de calcul, les transitions de $\mathcal{M}$ correspondent aux étapes d'exécution du programme en `OCaml` ou en `C`.

Exemple 2.7 La fonction $f : \mathbb{N} \to \mathbb{N}$ définie par $n \mapsto \sum_{i=1}^{n} i^2$ est une fonction calculable. On peut définir une machine composée de la suite de bits représentant une fonction `C` qui calcule cette somme, exécute cette fonction sur la suite de bits associée à une entrée n, puis renvoie la suite de bits correspondant à cette somme (voir l'exemple 2.4, page 19 pour le code). □

Pour les problèmes de décision, une machine est donc définie par $\mathcal{M} : \{0,1\}^* \to \mathbb{B}$. D'un point de vue informatique appliquée, on peut voir ces machines comme une fonction en `OCaml` ou en `C` de signature :

```
val m_OCaml : char array -> bool
bool m_C(unsigned char *array);
```

Remarque 2.2 En informatique théorique, on peut se permettre d'imaginer que la mémoire de la machine est infinie, bien que cette hypothèse ne soit pas applicable en pratique. □

2.1.3 Problèmes décidables

Un problème est dit *décidable* s'il existe un algorithme capable de fournir une réponse correcte pour toutes les entrées possibles en un temps fini. En revanche, un problème est *indécidable* s'il n'existe aucun algorithme capable de le résoudre pour toutes les instances, quel que soit le temps disponible. Dans la suite, on se limite aux problèmes de décision.

Définition 2.6 - Décidabilité d'un problème de décision

Un problème de décision $\mathcal{P}$ est dit *décidable* s'il existe une machine $\mathcal{M}$ le décidant. Soit $u_x \in \{0,1\}^*$ la représentation binaire de $x \in \mathbb{X}$ après sa sérialisation *via* une fonction totale $ser : \mathbb{X} \to \{0,1\}^*$. On a alors $\forall x \in \mathbb{X}$, $u_x = ser(x) \in \{0,1\}^*$ et

$$\big(x \in \mathcal{P}^+ \iff \mathcal{M}(u_x) = 1\big) \text{ et } \big(x \in \mathcal{P}^- \iff \mathcal{M}(u_x) = 0\big)$$

Un problème de décision est donc décidable s'il existe un algorithme qui, pour toute entrée, termine avec une réponse binaire. La machine $\mathcal{M}$ fonctionne donc sans comportement indéfini et on a : $\mathbb{X} = \mathcal{P}^+ \oplus \mathcal{P}^-$.

Exemple 2.8 Les problèmes mentionnés précédemment, Prime, Path, Amicable, FunctionC, ShortestPath sont des problèmes décidables. □

Exemple 2.9 Tout problème de décision défini sur un ensemble $\mathbb{X}$ fini est décidable. En effet, il suffit d'écrire un programme correct qui énumère tous les éléments de $\mathbb{X}$ et qui renvoie le booléen associé selon un tableau d'association. □

Lorsqu'un problème de décision est indécidable, cela veut dire qu'il n'existe aucun algorithme capable de résoudre le problème pour toutes les entrées possibles en un nombre fini d'étapes. La machine peut conduire à un comportement indéfini en rentrant dans une boucle infinie. Dans ce cas, par définition, elle ne produira pas de résultat terminal.

Théorème 2.1

Il existe (infiniment) plus de problèmes indécidables que de problèmes décidables.

Démonstration Ce théorème peut être démontré en raison du déséquilibre fondamental entre le nombre d'algorithmes possibles (qui est dénombrable) et le nombre de problèmes potentiels (qui est non dénombrable).

Chaque machine représente un algorithme sous forme d'un programme écrit dans un langage de programmation. En conséquence, tout algorithme peut être encodé par un mot fini dans $\{0,1\}^*$. Or, cet ensemble est dénombrable car on peut énumérer ses éléments - infinis - de manière séquentielle, en bijection avec $\mathbb{N}$. Il y a donc une quantité infinie mais dénombrable de machines (et donc d'algorithmes).

Un problème de décision est un langage défini dans $\{0,1\}^*$. L'ensemble des problèmes de décision est donc l'ensemble des sous-ensembles de $\{0,1\}^*$, c'est-à-dire l'ensemble des parties de $\{0,1\}^*$. Or, le nombre de ces parties est non dénombrable.

D'après le théorème de Cantor [14], il n'existe pas de bijection entre un ensemble et l'ensemble des parties de cet ensemble. Cela signifie qu'il existe bien plus de problèmes théoriques qu'il n'existe d'algorithmes pour les résoudre. Il existe infiniment plus de problèmes pour lesquels aucun algorithme ne peut donner une réponse en un nombre fini d'étapes et donc infiniment plus de problèmes indécidables. □

Un exemple classique d'un problème indécidable est le problème de l'arrêt (*Halting problem*) qui est étudié dans la section 2.1.4. Un exemple classique d'un problème décidable est le problème SAT qui a été déjà étudié en MP2I *via* l'algorithme de Quine [34] mais qui sera retraité pour ses propriétés de **NP**-complétude dans la section 2.2.4 Problème SAT, page 38.

2.1.4 Problème de l'arrêt

On peut remarquer qu'une machine $\mathcal{M}$ est elle-même une suite binaire qui peut être représentée comme un mot dans $\{0,1\}^*$. Contrairement à une machine spécifique qui est conçue pour résoudre un seul problème ou exécuter un seul algorithme, il est possible de concevoir une machine qui simule toute machine. Une telle machine est dite universelle.

Définition 2.7 - Machine universelle

Soit $\mathcal{M}_{univ}$ une machine qui prend en paramètres la représentation binaire d'une machine $\mathcal{M}$ et de son paramètre u et qui renvoie la sortie de $\mathcal{M}$ sur u.

$$\begin{array}{rcccl} \mathcal{M}_{univ} & : & \{0,1\}^* \times \{0,1\}^* & \to & \{0,1\}^* \\ & & (\mathcal{M}, u) & \mapsto & \begin{cases} \mathcal{M}(u) \text{ si } \mathcal{M} \text{ s'arrête sur } u \\ \text{sinon comportement indéfini} \end{cases} \end{array}$$

La machine $\mathcal{M}_{univ}$ est appelée une *machine universelle*.

Remarque 2.3 Si on écarte l'hypothèse d'une mémoire infinie, une machine universelle correspond à un interpréteur de langage de programmation. □

Le problème de l'arrêt consiste à déterminer, pour une machine $\mathcal{M}$ et une entrée u données, si $\mathcal{M}$ s'arrête ou s'exécute indéfiniment lorsqu'on lui fournit u en entrée. Dans la démonstration de l'indécidabilité de ce problème, on utilise l'idée d'une machine universelle, c'est-à-dire une machine qui est capable de simuler n'importe quelle autre machine sur une entrée donnée. L'intuition de cette démonstration repose sur l'application d'un algorithme à lui-même. On suppose qu'on dispose d'un moyen algorithmique pour savoir si un algorithme se termine. « Je » peux alors lui demander ce que « je » vais faire, puis faire le contraire, ce qui conduit à un paradoxe.

Théorème 2.2

On considère le problème de l'arrêt suivant.

HaltingProblem
In : $\mathcal{M}$ une machine et u son argument
Out : $\mathcal{M}$ s'arrête-t-elle sur u ?

Le problème de décision HaltingProblem est indécidable [72].

Démonstration On raisonne par l'absurde et on suppose que HaltingProblem est décidable. Par définition, il existe une machine $\mathcal{M}_{\text{HP}}$ qui le décide. La machine $\mathcal{M}_{\text{HP}}$ se termine donc toujours, elle ne peut pas avoir de comportement indéfini.

$$\begin{array}{rccl} \mathcal{M}_{\text{HP}} & : & \{0,1\}^* & \to & \mathbb{B} \\ & & (\mathcal{M}, u) & \mapsto & \begin{cases} 1 \text{ si } \mathcal{M} \text{ s'arrête sur } u \\ 0 \text{ si } \mathcal{M} \text{ ne s'arrête pas sur } u \end{cases} \end{array}$$

On définit ensuite une machine $\mathcal{M}_{paradox}$ qui utilise la machine $\mathcal{M}_{\text{HP}}$ lors de son exécution. On peut exprimer les instructions de $\mathcal{M}_{paradox}$ en pseudo-code.

$\mathcal{M}_{paradox}$

```
Input : (M, u)                                  Output : 0 ou 1
1 : if M_HP((M, u)) then                        ▷ si M s'arrête sur u
2 :     while 1 do                              ▷ boucle infinie
3 :         nothing
4 :     end while
5 :     return 1
6 : else                                        ▷ si M ne s'arrête pas sur u
7 :     return 0
8 : end if
```

Puisque $\mathcal{M}_{\text{HP}}$ se termine toujours, $\mathcal{M}_{paradox}$ exécute toujours une des deux branches de son expression conditionnelle. On s'intéresse maintenant au comportement de $\mathcal{M}_{paradox}$ lorsqu'elle s'exécute avec son propre code en argument, c'est-à-dire avec l'entrée $(\mathcal{M}_{paradox}, u)$. Il y a deux cas possibles.

- Si $\mathcal{M}_{paradox}$ se termine alors elle a nécessairement exécuté la branche `else` de son expression conditionnelle. Cela veut dire que $\mathcal{M}_{\text{HP}}$ a renvoyé 0, indiquant que $\mathcal{M}_{paradox}$ ne s'est pas arrêtée. Cela est contradictoire puisqu'on a supposé que l'exécution se termine.
- Si $\mathcal{M}_{paradox}$ ne se termine pas alors elle a nécessairement exécuté la branche `if` de son expression conditionnelle. Cela veut dire que $\mathcal{M}_{\text{HP}}$ a renvoyé 1, indiquant que $\mathcal{M}_{paradox}$ s'est arrêtée. Cela est contradictoire puisqu'on a supposé que l'exécution ne se termine pas.

Ces contradictions montrent que la machine $\mathcal{M}_{\text{HP}}$ ne peut pas exister, et donc que le problème de décision HaltingProblem est indécidable. □

Remarque 2.4 Alan Turing a démontré le théorème de l'arrêt qui stipule donc qu'il n'existe aucun algorithme universel capable de déterminer, pour toute machine de Turing et toute entrée donnée, si la machine s'arrêtera ou s'exécutera indéfiniment. En d'autres termes, certains problèmes ne peuvent pas être résolus par un algorithme, quel que soit le temps ou les ressources alloués. Cependant, bien que le théorème de l'arrêt soit indécidable de manière générale, il est possible d'étudier et de résoudre la terminaison des algorithmes particuliers [34]. Ainsi, la démonstration de Turing ouvre la voie à la compréhension des limites de ce qui est calculable. □

2.2 Classes de complexité

Certains problèmes de décision, bien que décidables, sont difficiles à résoudre en pratique en raison de la complexité excessive des calculs nécessaires. Il est donc essentiel de distinguer la décidabilité d'un problème de la faisabilité de sa résolution dans un temps raisonnable.

Une opération élémentaire dans un algorithme désigne une action de base qui peut être effectuée en une seule étape, comme l'addition de deux nombres, la comparaison de deux valeurs ou le déplacement d'une donnée en mémoire. Bien que les machines de Turing soient un modèle théorique abstrait, elles simulent ce que l'on considère comme une opération élémentaire dans un algorithme : chaque étape de la machine consiste en une action unitaire (comme lire un symbole et changer d'état). Ce modèle permet de formaliser les calculs sous forme de séquences d'opérations élémentaires. En effet, les classes de complexité usuelles sont définies en utilisant les machines de Turing comme modèles de calcul. Dans le programme de MP2I, on a étudié la complexité d'un algorithme. Par la suite, on s'intéresse à la complexité en temps des problèmes de décision, en fonction de la taille de leur instance, souvent notée n.

Remarque 2.5 En mathématiques, les entiers sur lesquels peuvent être appliquées des opérations arithmétiques « élémentaires » (addition, multiplication, *etc.*) n'ont pas de limite de taille. En informatique, au contraire, un entier manipulé par la machine doit tenir dans un nombre de bits fixé à l'avance : par exemple, un entier non signé stocké en mémoire sur 4 octets ne peut prendre que des valeurs comprises entre 0 et $2^{32}-1$. Cet ensemble de nombres « informatiques » peut être pensé comme celui des entiers modulo 2^{32}. Ainsi, en informatique, par exemple une opération d'addition dite élémentaire est limitée par ce que la machine peut réaliser directement. Un entier non signé représenté sur 4 octets n'appartient pas à $\mathbb{N}$, mais à la classe d'équivalence modulo 2^{32}. De plus, lorsqu'un programme travaille avec des entiers plus grands que ceux que le processeur peut gérer, une opération arithmétique mathématiquement élémentaire peut ne plus l'être informatiquement parlant et, au contraire, nécessiter un nombre arbitrairement long d'opérations machine. □

2.2.1 Classes P et NP

Définition 2.8 - Classe P

La *classe* ***P*** est l'ensemble des problèmes de décision qui peuvent être résolus par un algorithme de complexité temporelle polynomiale, c'est-à-dire en $\mathcal{O}(n^p)$ où n est la taille de l'instance et $p \in \mathbb{N}$.

Remarque 2.6 La classe **P**, parfois notée **PTIME**, regroupe les problèmes de décision qui peuvent être résolus en temps polynomial. En revanche, la classe **PSPACE** concerne les problèmes de décision qui peuvent être résolus en utilisant un espace polynomial, même si le temps requis pour les résoudre peut être supérieur, voire exponentiel. □

Exemple 2.10 En 2002, il a été prouvé que le problème de décision PRIME, présenté dans l'exemple 2.2, page 18, appartient à la classe **P**. Soit n un nombre dont la primalité est cherchée. Ce nombre n peut être représenté avec $\log n$ bits. La complexité de l'algorithme déterministe AKS [1] est en $\mathcal{O}\Big((\log n)^6\Big)$ donc polynomiale en la taille de l'instance. □

Exemple 2.11 Le problème de décision DUPLICATE, qui détermine si oui ou non un tableau contient un doublon, appartient à la classe **P**.

DUPLICATE
In : $n \in \mathbb{N}$, un tableau t de n entiers
Out : Le tableau t contient-il un doublon ?

La taille d'une instance est n. On peut savoir s'il y a un doublon en examinant ces n éléments un par un après avoir réalisé un tri. Ce problème peut donc être résolu grâce à un algorithme de complexité temporelle polynomiale en $\mathcal{O}(n \log n + n)$. □

Exemple 2.12 Le problème de décision CONNECTEDGRAPH sur la connexité des graphes appartient à la classe **P**.

CONNECTEDGRAPH
In : $G = (V, E)$ un graphe non orienté
Out : G est-il connexe ?

Afin de décider si un graphe non orienté est connexe, il suffit de réaliser un parcours en profondeur dont la complexité temporelle est en $\mathcal{O}(|V| + |E|)$. □

Un problème appartenant à la classe **P** peut être résolu par un algorithme dont la complexité est polynomiale, c'est-à-dire par un algorithme dont le temps de calcul est majoré par $\mathcal{O}(n^p)$, avec $p \in \mathbb{N}$. Toutefois, cette définition ne précise pas la valeur du degré p du polynôme. Cela signifie que, bien qu'un problème appartienne à **P**, cela n'assure pas nécessairement une résolution efficace en pratique pour des instances de grande taille. Par exemple, un algorithme avec une complexité en $\mathcal{O}\big(n^{10}\big)$ pourrait, dans certains cas, nécessiter plus de temps qu'un algorithme de complexité exponentielle comme $\mathcal{O}(2^n)$, et ce même pour des valeurs modérées de n. Ainsi, bien qu'un problème soit théoriquement résoluble en temps polynomial, un polynôme de degré très élevé peut rendre l'algorithme impraticable dans les faits. Cette observation souligne l'importance de considérer non seulement la classe de complexité, mais aussi l'efficacité pratique des algorithmes pour traiter des problèmes réels.

Exemple 2.13 Le problème de décision k-COLORGRAPH consiste à déterminer, pour un graphe non orienté donné, s'il est possible d'attribuer l'une de k couleurs à chaque sommet de manière que deux sommets adjacents n'aient jamais la même couleur.

k-COLORGRAPH
In : $G = (V, E)$ un graphe non orienté, $k \in \mathbb{N}$
Out : G est-il *coloriable* avec k couleurs ?

Le problème 3-ColorGraph est un problème qui ne peut pas être résolu, à ce jour, avec un algorithme en complexité temporelle polynomiale. Pour le moment, il n'appartient donc pas à la classe **P**. En revanche, il est possible de vérifier qu'un graphe non orienté est coloriable avec trois couleurs en temps polynomiale si une proposition de coloriage correcte est présentée (voir l'exercice 16 page 49). □

Comme le problème du coloriage à 3 couleurs, il existe de nombreux problèmes qui ne sont pas dans **P**, à ce jour. Au-delà de **P**, il existe une hiérarchie de classes de complexité de plus en plus étendues. Parmi elles, la classe **NP** regroupe les problèmes de décision pour lesquels il est possible de vérifier une solution correcte en temps polynomial. Toutefois, bien que la vérification soit efficace, le processus de recherche de la solution reste « difficile » et n'est pas garanti d'être réalisable en temps polynomial.

Définition 2.9 - Certificat

Un *certificat* c pour un problème $\mathcal{P}$ est une donnée supplémentaire qui représente une solution potentielle à une instance x de ce problème. L'ensemble des certificats est noté $\mathcal{C} \subseteq \{0,1\}^*$.

Étant donné un problème de décision, un certificat est une donnée auxiliaire qui, jointe à une instance du problème, permet de vérifier que cette instance est une instance positive.

Définition 2.10 - Problème de vérification

Soit $\mathcal{P}$ un problème de décision défini sur $\mathbb{X} \times \mathbb{Y}$ et soit $\mathcal{C}$ l'ensemble des certificats associés à ce problème. Un *problème de vérification* associé à $\mathcal{P}$ est un problème de décision défini par :

VERIFY$_\mathcal{P}$
In : $x \in \mathbb{X}$, $c \in \mathcal{C}$
Out : Le certificat c vérifie-t-il x ?

Vérifier qu'un certificat c valide une instance x d'un problème de décision $\mathcal{P}$ consiste à s'assurer que c atteste que x est une instance positive de $\mathcal{P}$.

Définition 2.11 - Classe NP

La classe **NP** est l'ensemble des problèmes de décisions $\mathcal{P}$ pour lesquels VERIFY$_\mathcal{P} \in$ **P** et $\forall x \in \mathbb{X}$:

$$x \in \mathcal{P}^+ \iff \left(\exists c \in \mathcal{C}, \exists k \in \mathbb{N} \mid |c| \leqslant |x|^k\right) \text{ et } \left((x,c) \in \text{VERIFY}_\mathcal{P}^+\right)$$

où $|x|$ et $|c|$ désignent respectivement la taille de l'instance et du certificat.

Un problème de décision est donc dans la classe **NP** s'il existe pour chaque instance positive, un certificat de taille polynomiale en la taille de l'instance, permettant de vérifier en temps polynomial que cette instance est bien positive.

Remarque 2.7 À l'origine, la classe **P** (*polynomial*) est définie comme l'ensemble des problèmes de décision pouvant être résolus en temps polynomial par une machine de Turing déterministe. La classe **NP** (*non-deterministic polynomial*) regroupe l'ensemble des problèmes de décision pour lesquels il existe une machine de Turing non déterministe capable de résoudre le problème en temps polynomial. Une machine de Turing non déterministe peut être interprétée comme un modèle de calcul qui « devine » un certificat en temps polynomial et en vérifie ensuite la validité en temps polynomial [1]. □

Proposition 2.1

Le problème de décision SAT est défini, pour une formule logique, par :

SAT
In : $F \in \mathcal{F}$
Out : Existe-t-il un modèle pour F ?

SAT $\in$ **NP**.

Démonstration Soit $F \in \mathcal{F}$ une formule logique. L'ensemble des certificats associé au problème est constitué des valuations appartenant à $\mathbb{B}^{\mathcal{F}}$. La taille d'un certificat $val \in \mathbb{B}^{\mathcal{F}}$ est majorée par la taille de la formule F puisqu'une seule valeur booléenne est requise pour chaque variable propositionnelle de F.

Le problème de vérification associé au problème SAT est alors défini par :

$\textsc{Verify}_{\text{SAT}}$
In : $F \in \mathcal{F}$, $val \in \mathbb{B}^{\mathcal{F}}$
Out : $val \overset{?}{\models} F$

La fonction `eval` donnée dans le code 2.1, et implémentée en MP2I [34], permet de répondre à ce problème de vérification. Le type inductif `formula` représente les formules logiques, et une valuation est implémentée sous la forme d'une liste d'associations de type `(string * bool) list` fournissant une valeur pour chaque variable.

```
type formula =
  | Var of string
  | Const of bool
  | Not of formula
  | And of formula * formula
  | Or of formula * formula
  | Imp of formula * formula
  | Eqv of formula * formula
```

1. Le concept de machine de Turing non déterministe est hors programme de MPI. Elle n'est pas formellement définie comme une machine capable de « deviner » une solution. Il s'agit plutôt d'un modèle théorique dans lequel, à partir d'un état donné, l'exécution peut se poursuivre vers un ensemble d'états et non un seul. Toutes ces branches d'exécutions sont explorées simultanément. Une telle machine de Turing n'a aucune réalisation matérielle (y compris en informatique quantique) et sert uniquement de modèle abstrait pour définir la classe **NP**.

```
let rec eval formula valuation =
  match formula with
  | Var v -> List.assoc v valuation
  | Const b -> b
  | Not f -> not (eval f valuation)
  | And (f1, f2) -> (eval f1 valuation) && (eval f2 valuation)
  | Or (f1, f2) -> (eval f1 valuation) || (eval f2 valuation)
  | Imp (f1, f2) -> not (eval f1 valuation) || (eval f2 valuation)
  | Eqv (f1, f2) -> (eval f1 valuation) = (eval f2 valuation)
```

CODE 2.1 – [OCaml] Évaluation d'une formule logique *via* une valuation

Cette fonction `eval` est de complexité temporelle polynomiale en la taille de l'entrée (F, val) donc $\text{VERIFY}_{\text{SAT}} \in \mathbf{P}$. De plus, pour toute formule $\forall F \in \mathcal{F}$:

$$F \in \text{SAT}^+ \iff \left(\exists val \in \mathbb{B}^{\mathcal{F}}, \exists k \in \mathbb{N} \mid |val| \leqslant |F|^k\right) \text{ et } (F, val) \in \text{VERIFY}^+_{\text{SAT}}$$

Finalement, $\text{SAT} \in \mathbf{NP}$. □

Exemple 2.14 Pour certifier qu'un graphe non orienté est 3-coloriable, un certificat consiste en une coloration, représentable par une structure de taille proportionnelle à celle du graphe. Il suffit alors de vérifier que deux sommets adjacents reçoivent des couleurs différentes. Cette vérification s'effectuant en temps polynomial, on peut conclure que le problème 3-COLORGRAPH appartient à la classe **NP**. □

Proposition 2.2

$\mathbf{P} \subseteq \mathbf{NP}$.

Démonstration Soit $\mathcal{P}$ un problème de décision dans **P**. Pour toute instance positive $x \in \mathcal{P}^+$ du problème, le certificat est l'instance elle-même : $c = x$. La taille de c est nécessairement polynomiale en la taille de x. Par définition, il existe un algorithme de complexité temporelle polynomiale qui résout ce problème. Le problème de vérification $\text{VERIFY}_{\mathcal{P}}$ est alors simplement le problème de décision lui-même (après projection sur son ensemble de définition $\mathbb{X}$). On peut donc affirmer que, pour chaque instance positive de $\mathcal{P}$, en prenant le certificat égal à l'instance, celui-ci peut être vérifié en temps polynomial. Cela revient à dire que $\mathcal{P} \in \mathbf{NP}$. Finalement il en résulte que : $\mathbf{P} \subseteq \mathbf{NP}$. □

Remarque 2.8 Le problème $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ est l'une des questions les plus célèbres et non résolues en informatique théorique. Si $\mathbf{P} = \mathbf{NP}$, cela impliquerait qu'il est possible de trouver rapidement (en temps polynomial) des solutions à tous les problèmes pour lesquels il est actuellement possible de vérifier rapidement une solution. En revanche, si $\mathbf{P} \neq \mathbf{NP}$, cela confirmerait que certains problèmes sont fondamentalement difficiles à résoudre, même si leur solution peut être facilement vérifiée. Depuis que Stephen Cook a posé cette question en 1971 [18], personne n'a encore réussi à démontrer l'une des deux possibilités (en 2025). La question fait partie des sept problèmes du prix du millénaire établis par l'Institut Clay de mathématiques en 2000, avec une récompense d'un million de dollars pour quiconque trouvera une solution. □

2.2.2 Réduction polynomiale

Pour établir qu'un problème est indécidable, on utilise fréquemment la technique de la *réduction polynomiale* : si un problème de décision $\mathcal{P}$ peut être réduit en temps polynomial à un autre problème $\mathcal{Q}$, et que $\mathcal{P}$ est déjà connu comme indécidable alors $\mathcal{Q}$ est aussi indécidable. Cette idée de transfert de difficulté par réduction se trouve également dans l'étude de la complexité algorithmique.

Pour démontrer qu'un problème appartient à la classe **P**, il « suffit » de concevoir un algorithme dont la complexité temporelle est polynomiale en fonction de la taille de l'instance. Cela prouve que le problème peut être résolu efficacement. En revanche, démontrer qu'un problème n'appartient pas à **P** est bien plus délicat. Par exemple, trouver un algorithme dont la complexité est exponentielle ne suffit pas, car cela n'exclut pas l'existence potentielle d'un algorithme plus performant en temps polynomial. Pour montrer qu'un problème ne fait pas partie de **P**, il faut prouver qu'il n'existe aucun algorithme polynomiale pour le résoudre. L'une des approches communes consiste à utiliser la *réduction polynomiale* : si un problème de décision $\mathcal{P}$ peut être réduit en temps polynomial à un autre problème $\mathcal{Q}$, et que $\mathcal{Q}$ est déjà connu pour appartenir à **NP**, alors cela indique que $\mathcal{P}$ est au moins aussi difficile que $\mathcal{Q}$, et donc dans **NP** (voir la proposition 2.8, page 36).

Définition 2.12 - Réduction polynomiale

Soient deux problèmes de décisions $\mathcal{P} \subseteq \mathbb{X}_\mathcal{P} \times \mathbb{B}$ et $\mathcal{Q} \subseteq \mathbb{X}_\mathcal{Q} \times \mathbb{B}$. On dit que $\mathcal{P}$ se réduit polynomialement à $\mathcal{Q}$, et on note $\mathcal{P} \preceq_{\text{poly}} \mathcal{Q}$, s'il existe une fonction totale calculable $f : \mathbb{X}_\mathcal{P} \to \mathbb{X}_\mathcal{Q}$ de complexité temporelle polynomiale en la taille de son entrée telle que :

$$\forall x \in \mathbb{X}_\mathcal{P}, \quad x \in \mathcal{P}^+ \iff f(x) \in \mathcal{Q}^+$$

Exemple 2.15 On considère les problèmes de décisions suivantes qui consistent à savoir si un tableau d'entier est trié en ordre croissant ou décroissant.

AscendingSort
In : $n \in \mathbb{N}$, un tableau t de n entiers
Out : Le tableau t est-il trié en ordre croissant ?

DescendingSort
In : $n \in \mathbb{N}$, un tableau t de n entiers
Out : Le tableau t est-il trié en ordre décroissant ?

Afin de réduire le problème AscendingSort à DescendingSort, il faut inverser tous les éléments du tableau t. Si le tableau inversé est trié en ordre décroissant, cela est équivalent à dire que le tableau original est trié en ordre croissant. On peut écrire une fonction en `C` pour cette inversion.

```c
int *inverse(int *t, int n) {
  int *inversed = (int *)malloc(n * sizeof(int));
  if (inversed == NULL) {
    fprintf(stderr, "Memory allocation failed\n");
    return NULL;
  }
  for (int i = 0; i < n; i++)
    inversed[i] = t[n - 1 - i];
  return inversed;
}
```

Cette fonction **inverse** est de complexité temporelle linéaire en $\mathcal{O}(n)$ et donc polynomiale. On peut donc conclure que AscendingSort $\leq_{\text{poly}}$ DescendingSort. De même, la même fonction **inverse** permet de conclure la réduction polynomiale réciproque : DescendingSort $\leq_{\text{poly}}$ AscendingSort. □

Proposition 2.3

La relation $\leq_{\text{poly}}$ est un préordre.

Démonstration Une relation préordre est une relation réflexive et transitive.

Réflexivité. Soit $\mathcal{P} \subseteq \mathbb{X}_{\mathcal{P}} \times \mathbb{B}$ un problème de décision. La fonction utilisée pour établir la réflexivité est la fonction identité $id : \mathbb{X}_{\mathcal{P}} \to \mathbb{X}_{\mathcal{P}}$ dont la complexité temporelle est linéaire (et donc polynomiale). Cette fonction satisfait :

$$\forall x \in \mathbb{X}_{\mathcal{P}},\ x \in \mathcal{P}^+ \iff id(x) \in \mathcal{P}^+$$

Transitivité. Soient $\mathcal{P} \subseteq \mathbb{X}_{\mathcal{P}} \times \mathbb{B}$, $\mathcal{Q} \subseteq \mathbb{X}_{\mathcal{Q}} \times \mathbb{B}$, $\mathcal{R} \subseteq \mathbb{X}_{\mathcal{R}} \times \mathbb{B}$ trois problèmes de décisions tels que $\mathcal{P} \leq_{\text{poly}} \mathcal{Q}$ et $\mathcal{Q} \leq_{\text{poly}} \mathcal{R}$. Par définition, il existe deux fonctions totales $f_1 : \mathbb{X}_{\mathcal{P}} \to \mathbb{X}_{\mathcal{Q}}$ et $f_2 : \mathbb{X}_{\mathcal{Q}} \to \mathbb{X}_{\mathcal{R}}$, toutes deux calculables en temps polynomial en la taille de leurs entrées respectives, telles que :

$$\forall x \in \mathbb{X}_{\mathcal{P}},\ x \in \mathcal{P}^+ \iff f_1(x) \in \mathcal{Q}^+ \quad \text{et} \quad \forall x \in \mathbb{X}_{\mathcal{Q}},\ x \in \mathcal{Q}^+ \iff f_2(x) \in \mathcal{R}^+$$

La fonction $f = f_2 \circ f_1 : \mathbb{X}_{\mathcal{P}} \to \mathbb{X}_{\mathcal{R}}$ résultant de la composition est elle-même calculable en temps polynomial, l'ensemble des fonctions totales calculables en temps polynomial étant stable par composition. On a alors :

$$\forall x \in \mathbb{X}_{\mathcal{P}},\ x \in \mathcal{P}^+ \iff f_1(x) \in \mathcal{Q}^+ \iff f_2(f_1(x)) \in \mathcal{R}^+ \iff f(x) \in \mathcal{R}^+$$

Ce qui conclut que $\mathcal{P} \leq_{\text{poly}} \mathcal{R}$. □

Remarque 2.9 La relation $\leq_{\text{poly}}$ n'est pas antisymétrique. La réduction polynomiale d'un problème de décision à un autre se concentre sur la complexité temporelle polynomiale de cette conversion. Si $\mathcal{P} \leq_{\text{poly}} \mathcal{Q}$ et $\mathcal{Q} \leq_{\text{poly}} \mathcal{P}$, alors les problèmes $\mathcal{P}$ et $\mathcal{Q}$ sont dans la même classe de complexité sans nécessairement être le même problème. Par exemple, dans l'exemple 2.15, page 32, il a été démontré les deux réductions polynomiales AscendingSort $\leq_{\text{poly}}$ DescendingSort et DescendingSort $\leq_{\text{poly}}$ AscendingSort. Cependant, ces deux problèmes sont bien distincts mais ils sont tous les deux dans **P**. □

Proposition 2.4

Soient $\mathcal{P}$ et $\mathcal{Q}$ deux problèmes de décision avec $\mathcal{P} \preceq_{\text{poly}} \mathcal{Q}$. Si $\mathcal{Q}$ est décidable alors $\mathcal{P}$ est décidable.

Démonstration On a $\mathcal{P} \preceq_{\text{poly}} \mathcal{Q}$, par définition il existe une fonction totale et calculable $f : \mathbb{X}_{\mathcal{P}} \to \mathbb{X}_{\mathcal{Q}}$ de complexité temporelle polynomiale en la taille de son entrée telle que :

$$\forall x \in \mathbb{X}_{\mathcal{P}}, \quad x \in \mathcal{P}^+ \iff f(x) \in \mathcal{Q}^+$$

Par ailleurs, puisque $\mathcal{Q}$ est décidable, il existe une machine $\mathcal{M}$ le décidant. La machine définie par $\mathcal{M} \circ ser \circ f$ (totale et calculable en tant que composée de telles fonctions) décide alors le problème $\mathcal{P}$:

$$\forall x \in \mathbb{X}_{\mathcal{P}}, \quad (\mathcal{M} \circ ser \circ f)(x) = 1 \iff f(x) \in \mathcal{Q}^+ \iff x \in \mathcal{P}^+$$

Donc $(\mathcal{M} \circ ser \circ f)(x) \neq 1 \iff x \in \mathcal{P}^- \iff (\mathcal{M} \circ ser \circ f)(x) = 0$. On peut alors conclure que $\mathcal{P}$ est décidable. □

Proposition 2.5

Soient $\mathcal{P}$ et $\mathcal{Q}$ deux problèmes de décision avec $\mathcal{P} \preceq_{\text{poly}} \mathcal{Q}$. Si $\mathcal{P}$ est indécidable alors $\mathcal{Q}$ est indécidable.

Démonstration La contraposée de la proposition 2.4 indique que si $\mathcal{P}$ n'est pas décidable alors $\mathcal{Q}$ n'est pas décidable. Or, l'ensemble des problèmes décidables et l'ensemble des problèmes indécidables sont complémentaires. Donc lorsqu'on a la réduction polynomiale $\mathcal{P} \preceq_{\text{poly}} \mathcal{Q}$, si $\mathcal{P}$ est indécidable alors $\mathcal{Q}$ est indécidable. □

Remarque 2.10 En pratique, afin de démontrer qu'un problème $\mathcal{P}$ est indécidable, on réduit le problème de l'arrêt HaltingProblem (voir le théorème 2.2, page 25), ou un autre problème déjà démontré indécidable, à $\mathcal{P}$ [2]. □

Proposition 2.6

Soit le problème de décision PositiveInstanceCheck, où on se demande si une machine a des instances positives, défini par :

PositiveInstanceCheck
In : $\mathcal{M}$ une machine
Out : $\mathcal{M}$ a-t-elle une instance positive ?

PositiveInstanceCheck est indécidable.

2. Le fait qu'il existe une fonction totale et calculable $f : \mathbb{X}_{\mathcal{P}} \to \mathbb{X}_{\mathcal{Q}}$ telle que $\forall x \in \mathbb{X}_{\mathcal{P}}$, $x \in \mathcal{P}^+ \iff f(x) \in \mathcal{Q}^+$ suffit pour conclure que $\mathcal{Q}$ est indécidable dès que $\mathcal{P}$ l'est. La classe de complexité de f (**P**, **NP**, ou toute autre considération de complexité) est sans importance : seule la calculabilité de f est requise. Cependant, tout problème de **P** ou de **NP** est par définition calculable.

Démonstration On démontre que PositiveInstanceCheck est indécidable en réduisant polynomialement le problème de l'arrêt HaltingProblem à ce problème. Soit $(\mathcal{M}', u')$ une instance de HaltingProblem. Le but est de construire une machine $\mathcal{M}$ telle que :

$$\begin{array}{rccl} \mathcal{M} & : & \{0,1\}^* & \to & \mathbb{B} \\ & & u & \mapsto & \begin{cases} 1 \text{ si } \mathcal{M}' \text{ s'arrête sur } u' \\ 0 \text{ si } \mathcal{M}' \text{ ne s'arrête pas sur } u' \end{cases} \end{array}$$

La notion de la machine universelle assure qu'il est possible de définir une fonction f totale et calculable qui associe à $(\mathcal{M}', u')$ une machine $\mathcal{M}$ et elle est de complexité temporelle polynomiale en la taille de $(\mathcal{M}', u')$.

Si $\mathcal{M}'$ s'arrête sur u' alors $\mathcal{M}$ accepte toutes ses instances, c'est-à-dire $\forall u \in \{0,1\}^*$, $\mathcal{M}(u) = 1$. Réciproquement, si $\mathcal{M}$ accepte n'importe quelle entrée, cela signifie que la simulation de $\mathcal{M}'$ sur u' a terminé. En revanche, si $\mathcal{M}'$ ne s'arrête pas sur u' alors $\mathcal{M}$ a un comportement indéfini. En effet, comme $\mathcal{M}$ attend la terminaison de $\mathcal{M}'$ pour produire une sortie, elle ne peut pas constater que $\mathcal{M}'$ ne s'arrêtera jamais ; elle reste donc elle-même bloquée indéfiniment, et en conséquence sa sortie ne peut être définie dans ce cas. Ainsi, $\forall (\mathcal{M}', u') \in \{0,1\}^*$:

$$\begin{aligned} (\mathcal{M}', u') \in \text{HaltingProblem}^+ &\iff \forall u \in \{0,1\}^*, \mathcal{M}(u) = 1 \\ &\iff \mathcal{M} \in \text{PositiveInstanceCheck}^+ \end{aligned}$$

Finalement, HaltingProblem $\preceq_{\text{poly}}$ PositiveInstanceCheck. De plus, la proposition 2.5 permet de conclure que PositiveInstanceCheck est indécidable. □

Proposition 2.7

Soient $\mathcal{P}$ et $\mathcal{Q}$ deux problèmes de décision avec $\mathcal{P} \preceq_{\text{poly}} \mathcal{Q}$. Si $\mathcal{Q} \in \mathbf{P}$ alors $\mathcal{P} \in \mathbf{P}$.

Démonstration Puisque $\mathcal{P} \preceq_{\text{poly}} \mathcal{Q}$, par définition, il existe une fonction totale et calculable $f : \mathbb{X}_{\mathcal{P}} \to \mathbb{X}_{\mathcal{Q}}$ de complexité temporelle polynomiale en la taille de l'entrée qui traduit une instance $x_{\mathcal{P}}$ de $\mathcal{P}$ en une instance $x_{\mathcal{Q}}$ de $\mathcal{Q}$. Donc on peut dire que la taille de $x_{\mathcal{Q}}$ est majorée par un polynôme, supposé croissant sans perte de généralité, en la taille de $x_{\mathcal{P}}$. Puisqu'on a $\mathcal{Q} \in \mathbf{P}$, par définition, il existe aussi un algorithme, de complexité temporelle polynomiale en la taille de son entrée, qui le résout. On note q la fonction qui réalise cet algorithme. La complexité temporelle de q est donc aussi majorée par un polynôme, supposé croissant, en la taille de $x_{\mathcal{Q}}$ c'est-à-dire en la taille de $x_{\mathcal{P}}$.

On construit alors une fonction $g = q \circ f$ qui commence par traduire une entrée de $\mathcal{P}$ en une entrée de $\mathcal{Q}$, puis renvoie le résultat de q. La complexité temporelle de cette fonction est aussi majorée par la composition des complexités temporelles des fonctions f et q. Elle est donc polynomiale en la taille de $x_{\mathcal{P}}$. Cela prouve que $\mathcal{P} \in \mathbf{P}$. □

Proposition 2.8

Soient $\mathcal{P}$ et $\mathcal{Q}$ deux problèmes de décision avec $\mathcal{P} \preceq_{\text{poly}} \mathcal{Q}$. Si $\mathcal{Q} \in \mathbf{NP}$ alors $\mathcal{P} \in \mathbf{NP}$.

Démonstration Puisque $\mathcal{P} \preceq_{\text{poly}} \mathcal{Q}$, par définition, il existe une fonction totale et calculable $f : \mathbb{X}_\mathcal{P} \to \mathbb{X}_\mathcal{Q}$ de complexité temporelle polynomiale en la taille de l'entrée qui traduit une instance $x_\mathcal{P}$ de $\mathcal{P}$ en une instance $x_\mathcal{Q}$ de $\mathcal{Q}$. On peut donc dire que la taille de $x_\mathcal{Q}$ est majorée par $|x_\mathcal{P}|^{k_1}$ avec $k_1 \in \mathbb{N}$.

Par ailleurs, $\mathcal{Q} \in \mathbf{NP}$, donc le problème de vérification $\textsc{Verify}_\mathcal{Q} \in \mathbf{P}$ et $\forall x_\mathcal{Q} \in \mathbb{X}_\mathcal{Q}$:

$$x_\mathcal{Q} \in \mathcal{Q}^+ \iff \left(\exists c_\mathcal{Q} \in \mathcal{C}_\mathcal{Q}, \exists k_2 \in \mathbb{N} \mid |c_\mathcal{Q}| \leqslant |x_\mathcal{Q}|^{k_2}\right) \text{ et } (x_\mathcal{Q}, c_\mathcal{Q}) \in \textsc{Verify}^+_\mathcal{Q}$$

On montre que $\textsc{Verify}_\mathcal{P} \in \mathbf{P}$. On s'intéresse au problème de vérification défini par :

$\textsc{Verify}_\mathcal{P}$
In : $x_\mathcal{P} \in \mathbb{X}_\mathcal{P}$, $c_\mathcal{P} \in \mathcal{C}_\mathcal{P}$
Out : Le certificat $c_\mathcal{P}$ vérifie-t-il $x_\mathcal{P}$?

On crée un certificat $c_\mathcal{P}$ pour le problème $\mathcal{P}$ à partir d'un certificat $c_\mathcal{Q}$ en y concaténant une suite de données supplémentaires. On définit alors la fonction $\pi : \mathcal{C}_\mathcal{P} \to \mathcal{C}_\mathcal{Q}$ qui permet de projeter un certificat $c_\mathcal{P}$ sur le certificat $c_\mathcal{Q}$ dont il est issu. Cette projection est de complexité temporelle polynomiale. La taille de $c_\mathcal{P}$ est bornée par un polynôme en la taille de $x_\mathcal{Q}$ par construction. On a $|c_\mathcal{P}| \leqslant |x_\mathcal{Q}|^{k_3}$ avec $k_3 \in \mathbb{N}$ et puisque $\mathcal{P} \preceq_{\text{poly}} \mathcal{Q}$, $|c_\mathcal{P}| \leqslant |f(x_\mathcal{P})|^{k_3}$.

On remarque aussi que :

$$|c_\mathcal{Q}| = |\pi(c_\mathcal{P})| \leqslant |c_\mathcal{P}| \leqslant |f(x_\mathcal{P})|^{k_3}$$

Afin de décider le problème $\textsc{Verify}_\mathcal{P}$, il faut calculer $x_\mathcal{Q} = f(x_\mathcal{P})$, $c_\mathcal{Q} = \pi(c_\mathcal{P})$ à partir de $x_\mathcal{P}$ et $c_\mathcal{P}$. Ensuite il faut vérifier si $(x_\mathcal{Q}, c_\mathcal{Q}) \in \textsc{Verify}^+_\mathcal{Q}$. Or, par hypothèse, $\textsc{Verify}_\mathcal{Q} \in \mathbf{P}$ et les fonctions f et π sont toutes les deux de complexité temporelle polynomiale. Finalement on a $\textsc{Verify}_\mathcal{P} \in \mathbf{P}$.

On montre l'équivalence pour $\mathcal{P}$. $\forall x_\mathcal{P} \in \mathbb{X}_\mathcal{P}$, on a :

$$\begin{aligned}
& x_\mathcal{P} \in \mathcal{P}^+ \\
\iff \quad & f(x_\mathcal{P}) \in \mathcal{Q}^+ \\
\iff \quad & \left(\exists c_\mathcal{Q} \in \mathcal{C}_\mathcal{Q}, \exists k_2 \in \mathbb{N} \mid |c_\mathcal{Q}| \leqslant |f(x_\mathcal{P})|^{k_2}\right) \text{ et } (f(x_\mathcal{P}), c_\mathcal{Q}) \in \textsc{Verify}^+_\mathcal{Q} \\
\iff \quad & \left(\exists c_\mathcal{P} \in \mathcal{C}_\mathcal{P}, \exists k_4 \in \mathbb{N} \mid |c_\mathcal{P}| \leqslant |f(x_\mathcal{P})|^{k_4}\right) \text{ et } (f(x_\mathcal{P}), \pi(c_\mathcal{P})) \in \textsc{Verify}^+_\mathcal{Q} \\
\iff \quad & \left(\exists c_\mathcal{P} \in \mathcal{C}_\mathcal{P}, \exists k_5 \in \mathbb{N} \mid |c_\mathcal{P}| \leqslant |x_\mathcal{P}|^{k_5}\right) \text{ et } (x_\mathcal{P}, c_\mathcal{P}) \in \textsc{Verify}^+_\mathcal{P}
\end{aligned}$$

On vient donc de démontrer que $\mathcal{P} \in \mathbf{NP}$. □

2.2.3 NP-complétude

Certains problèmes de la classe **NP** reflètent la difficulté intrinsèque de tous les problèmes de cette classe. Ces problèmes dits « **NP**-difficile » sont au moins aussi difficile qu'un problème **NP**, dans le sens où tout problème **NP** peut être réduit à ce problème en temps polynomial [3]. En effet cela marque un seuil de difficulté clé.

Remarque 2.11 Il est important de noter que les problèmes **NP**-difficiles ne doivent pas nécessairement appartenir à **NP**, car ils ne sont pas forcément des problèmes de décision (ils peuvent relever, par exemple, de l'optimisation) ou n'ont pas nécessairement une solution vérifiable en temps polynomial. Néanmoins, dans la suite, on ne s'intéresse qu'aux problèmes de décision. □

Définition 2.13 - NP-difficile

Un problème **NP**-difficile est un problème auquel tout problème **NP** peut être réduit polynomialement. Si $\mathcal{P} \in$ **NP**-difficile alors $\forall \mathcal{Q} \in$ **NP**, on a $\mathcal{Q} \preceq_{\text{poly}} \mathcal{P}$.

Proposition 2.9

Soient $\mathcal{P}$ et $\mathcal{Q}$ deux problèmes de décision avec $\mathcal{P} \preceq_{\text{poly}} \mathcal{Q}$. Si $\mathcal{P} \in$ **NP**-difficile alors $\mathcal{Q} \in$ **NP**-difficile.

Démonstration Par définition, puisque $\mathcal{P}$ est **NP**-difficile, $\forall \mathcal{R} \in$ **NP**, $\mathcal{R} \preceq_{\text{poly}} \mathcal{P}$. Par la transitivité de la réduction polynomiale (voir la proposition 2.3, page 33), puisque $\mathcal{R} \preceq_{\text{poly}} \mathcal{P} \preceq_{\text{poly}} \mathcal{Q}$, on peut dire que $\forall \mathcal{R} \in$ **NP**, $\mathcal{R} \preceq_{\text{poly}} \mathcal{Q}$ et donc $\mathcal{Q}$ est un problème de décision **NP**-difficile. □

Définition 2.14 - NP-complet

Un problème **NP**-complet est un problème de décision **NP**-difficile qui appartient à **NP**.

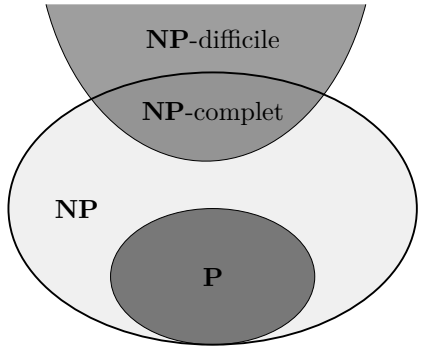


FIGURE 2.1 – Classes de complexité avec l'hypothèse **P** ≠ **NP**

3. On parle alors de la réduction polynomiale même entre deux problèmes qui ne sont pas forcément de décision.