Science et Génie Informatiques

Corpus – André-Guy Bruneau M.Sc. IT – Octobre 2025

Volume I : Fondations Théoriques de l'Informatique

Le parcours débute par l'établissement du socle intellectuel nécessaire à toute réflexion rigoureuse en informatique. Ce premier volume explore les **Mathématiques Discrètes et la Logique**, outils essentiels pour modéliser et raisonner sur les structures computationnelles. Il approfondit la **Théorie de la Computation**, abordant des questions fondamentales : Qu'est-ce qui est calculable ? Quelles sont les limites inhérentes aux machines ? À travers l'étude des automates, des langages formels et de la machine de Turing, le lecteur appréhende les concepts de calculabilité et d'indécidabilité. Enfin, la **Théorie de la Complexité** examine l'efficacité des calculs, introduisant les classes de complexité (comme P et NP) qui définissent la frontière entre ce qui est pratiquement réalisable et ce qui reste théoriquement intraitable.

Tables des matières

Chapitre 1 : Logiques et Méthodes de Preuve 2

Chapitre 1a : Logique Déductive à la Logique Inductive - Formalisation de la Théorie des Probabilités 34

Chapitre 2 : Structures Discrètes et Combinatoire 49

Chapitre 3 : Théorie des Graphes 78

Chapitre 4 : Langages Formels et Automates 98

Chapitre 5 : Calculabilité et Décidabilité 117

Chapitre 6 : Théorie de la Complexité 145

Chapitre 1 : Logiques et Méthodes de Preuve

Introduction

La logique, dans son essence, est la science du raisonnement valide. Ses origines remontent aux syllogismes d'Aristote, mais son développement formel, initié par des pionniers tels que Frege, Russell et Hilbert, a jeté les bases non seulement des mathématiques modernes, mais aussi, de manière inextricable, de l'informatique théorique.[1] Loin d'être une simple branche des mathématiques, la logique s'est révélée être le véritable calcul de la computation, le langage universel qui permet de décrire, d'analyser et de construire les systèmes informatiques, des plus simples aux plus complexes. Alan Turing et Alonzo Church, en formalisant la notion même de calcul, ont démontré que les processus algorithmiques pouvaient être entièrement capturés par des systèmes logiques formels, établissant ainsi un lien indissoluble entre la preuve et le programme.[2]

Ce chapitre a pour vocation de poser les fondations de ce lien. Il explore la dualité fondamentale qui est au cœur de toute logique formelle : la distinction entre la **syntaxe** et la **sémantique**. La syntaxe concerne la structure des énoncés, les règles qui gouvernent la formation de formules bien écrites à partir d'un alphabet de symboles. C'est le domaine de la manipulation formelle, purement symbolique, indépendante de toute signification.[4] La sémantique, quant à elle, s'intéresse au "sens" de ces formules, à leur interprétation et à leur valeur de vérité.[4] C'est elle qui nous permet de dire si un énoncé est vrai ou faux dans un certain contexte. L'informatique, dans sa globalité, navigue constamment entre ces deux pôles : un programme est un objet syntaxique, une suite de caractères respectant la grammaire d'un langage, mais son exécution et son utilité dépendent de sa sémantique, de ce qu'il accomplit dans le monde réel.

La maîtrise de ce formalisme est indispensable pour l'ingénieur et le chercheur en informatique. La conception de circuits logiques, qui sont à la base de tous les processeurs modernes, est une application directe de la logique propositionnelle.[6] La sémantique des langages de programmation, la vérification de la correction des logiciels et des systèmes matériels, la conception des langages de requête pour les bases de données, et le développement de l'intelligence artificielle reposent tous sur les formalismes plus expressifs de la logique des prédicats.[3] La logique fournit le cadre rigoureux nécessaire pour spécifier le comportement d'un système, pour raisonner sur ses propriétés et pour prouver qu'il se comportera comme attendu.

Ce chapitre se déroulera en quatre parties. La **Partie I** est consacrée à la **logique propositionnelle**, l'algèbre du raisonnement binaire. Nous y définirons rigoureusement sa syntaxe et sa sémantique, explorerons les notions de tautologie, de contradiction et de satisfiabilité, et introduirons les formes normales qui sont essentielles pour le traitement algorithmique. La **Partie II** étendra notre champ d'action à la **logique des prédicats**, un langage beaucoup plus riche permettant de parler d'objets, de leurs propriétés et des relations qui les unissent, grâce à l'introduction des quantificateurs. La **Partie III** abordera les **systèmes de preuve formelle**, des mécanismes syntaxiques permettant de dériver des conclusions valides à partir de prémisses. Nous y étudierons deux systèmes paradigmatiques : la déduction naturelle, qui modélise le raisonnement humain, et le principe de résolution, optimisé pour l'automatisation. Enfin, la **Partie IV** fera le pont entre ces formalismes et les **méthodes de raisonnement** fondamentales utilisées en mathématiques et en informatique, telles que la preuve par contraposée, le raisonnement par l'absurde, le principe des tiroirs et, surtout, le raisonnement par induction, qui est la clé de voûte de la preuve de correction des algorithmes et des structures de données récursives.

Partie I : La Logique Propositionnelle – L'Algèbre du Raisonnement

La logique propositionnelle, également connue sous le nom de calcul des propositions, constitue le premier échelon de la formalisation du raisonnement. Son pouvoir expressif est limité, car elle traite les propositions comme des entités atomiques indivisibles, sans analyser leur structure interne.[11] Cependant, sa simplicité est précisément ce qui en fait un outil d'une puissance et d'une importance fondamentales en informatique. Elle modélise parfaitement le monde binaire des circuits électroniques, où les signaux sont soit hauts (vrai), soit bas (faux), et constitue la base de l'algèbre de Boole qui régit le fonctionnement de chaque porte logique au sein d'un processeur.[7] De plus, elle sert de fondement à des logiques plus complexes et introduit les concepts essentiels de syntaxe, de sémantique, de validité et de prouvabilité qui seront généralisés par la suite.

1.1. Syntaxe et Sémantique : Le Langage des Propositions

L'étude de tout langage formel, y compris la logique propositionnelle, commence par la distinction cruciale entre sa syntaxe et sa sémantique.[4] Cette dualité est le principe organisateur central de la logique.

- La **syntaxe** définit l'ensemble des règles qui permettent de construire des énoncés correctement formés, appelés formules. Elle s'intéresse à la structure et à la forme des expressions, indépendamment de leur signification.[4] C'est la grammaire du langage logique.
- La **sémantique** attribue un sens, une signification, aux formules syntaxiquement correctes. En logique propositionnelle, ce sens est une valeur de vérité : vrai ou faux.[4] C'est la théorie de l'interprétation du langage.

1.1.1. Syntaxe Formelle

La syntaxe de la logique propositionnelle définit comment construire des formules valides à partir d'un ensemble de symboles de base.

Alphabet

Le langage est construit sur un alphabet Σ, qui est l'union de trois ensembles de symboles disjoints [4] :

1. Un ensemble infini dénombrable P={p,q,r,p1,p2,…} de **variables propositionnelles** (ou atomes). Chaque variable représente une proposition élémentaire, un énoncé simple qui peut être vrai ou faux, comme "il pleut" ou "n>2".[13]
2. Un ensemble de **connecteurs logiques** :

- ¬ (négation, "non")
- ∧ (conjonction, "et")
- ∨ (disjonction, "ou")
- → (implication, "si... alors...")
- ↔ (équivalence ou bi-implication, "si et seulement si")

3. Des symboles de ponctuation : les **parenthèses** (et).

Formules Bien Formées (EBF)

Toutes les chaînes de caractères construites avec l'alphabet Σ ne sont pas des formules valides. L'ensemble des **Expressions Bien Formées** (EBF), noté F, est le plus petit ensemble de mots sur Σ qui satisfait les règles de construction suivantes, définies de manière inductive [4] :

- **(Règle de base)** Toute variable propositionnelle p∈P est une EBF.
- **(Règles inductives)** Si F et G sont des EBF, alors les expressions suivantes sont aussi des EBF :
  - (¬F)
  - (F∧G)
  - (F∨G)
  - (F→G)
  - (F↔G)

Cette définition récursive est fondamentale. Elle garantit que toute formule est construite en un nombre fini d'étapes à partir des atomes. Par exemple, la chaîne (p→(¬q)) est une EBF car p et q sont des EBF (règle de base), donc (¬q) est une EBF (règle inductive), et enfin (p→(¬q)) est une EBF (règle inductive). En revanche, la chaîne p∧q) n'est pas une EBF car la parenthèse ouvrante est mal placée.

Arbres de Décomposition

La nature inductive de la définition des EBF garantit une propriété essentielle : la **lecture unique**. Chaque EBF peut être décomposée de manière unique en ses sous-formules immédiates. Cette structure de décomposition peut être visualisée sous la forme d'un **arbre de décomposition** (ou arbre syntaxique).[4] La racine de l'arbre est le connecteur principal de la formule, et ses enfants sont les racines des arbres de décomposition de ses sous-formules immédiates. Les feuilles de l'arbre sont les variables propositionnelles.

Par exemple, l'arbre de décomposition de la formule ((p∧(¬q))→r) est :

→

```
  /\
 ∧  r
 /\
 p ¬
|
  q
```

Cet arbre montre que le connecteur principal est →, qui relie la sous-formule (p∧(¬q)) à la sous-formule r. Cette structure arborescente est non seulement une aide visuelle, mais elle est aussi au cœur de nombreux algorithmes qui opèrent sur les formules logiques, comme les algorithmes d'évaluation sémantique.

Conventions et Abréviations

Pour améliorer la lisibilité et éviter une prolifération de parenthèses, on adopte des conventions de précédence entre les connecteurs. L'ordre de priorité décroissante est généralement le suivant [13] :

1. ¬
2. ∧
3. ∨
4. →
5. ↔

Ainsi, la formule ¬p∧q→r est interprétée comme (((¬p)∧q)→r). Les parenthèses restent nécessaires pour outrepasser ces priorités, comme dans ¬(p∧q).

De plus, il est courant de définir certains symboles comme des abréviations. Par exemple, les constantes logiques "vrai" (⊤) et "faux" (⊥) peuvent être introduites. On peut également définir l'équivalence comme une abréviation de la double implication [4] :

- (F↔G) est une abréviation pour ((F→G)∧(G→F)).

Ces conventions permettent de simplifier l'écriture sans altérer la rigueur formelle du langage.

1.1.2. Sémantique Formelle

La sémantique donne un sens aux formules bien formées. En logique propositionnelle, ce sens est une valeur de vérité, déterminée par l'interprétation des variables propositionnelles de base.

Domaine de Vérité et Valuation

Le domaine sémantique est l'ensemble des valeurs de vérité, B={Vrai,Faux}, que l'on représente souvent par l'ensemble binaire {1,0} pour des raisons de commodité algébrique.[4]

Le point de départ de l'interprétation sémantique est la **valuation** (parfois appelée interprétation ou assignation). Une valuation est une fonction v:P→{1,0} qui assigne une valeur de vérité à chaque variable propositionnelle.[4] Une valuation représente un "état du monde" possible, où chaque proposition atomique est soit vraie, soit fausse.

Extension aux Formules Complexes et Tables de Vérité

Le cœur de la sémantique propositionnelle est la définition inductive de la valeur de vérité d'une formule complexe, qui étend la fonction de valuation v de P à l'ensemble de toutes les EBF, F. Cette extension, notée v^{-}:F→{1,0}, est définie comme suit [4] :

- **Base :** Si F est une variable propositionnelle p, alors $v^{-}(p)=v(p)$.
- **Induction :** Si F et G sont des EBF, alors :
  - $v^{-}((\neg F))=1$ si et seulement si $v^{-}(F)=0$.
  - $v^{-}((F \wedge G))=1$ si et seulement si $v^{-}(F)=1$ et $v^{-}(G)=1$.
  - $v^{-}((F \vee G))=1$ si et seulement si $v^{-}(F)=1$ ou $v^{-}(G)=1$.
  - $v^{-}((F \rightarrow G))=1$ si et seulement si $v^{-}(F)=0$ ou $v^{-}(G)=1$. (Équivalent à : $v^{-}((F \rightarrow G))=0$ ssi $v^{-}(F)=1$ et $v^{-}(G)=0$).
  - $v^{-}((F \leftrightarrow G))=1$ si et seulement si $v^{-}(F)=v^{-}(G)$.

Ces règles définissent formellement la signification de chaque connecteur. Elles peuvent être visualisées et calculées systématiquement à l'aide de **tables de vérité**. Une table de vérité énumère toutes les valuations possibles pour les variables d'une formule et donne la valeur de vérité de la formule pour chaque valuation.[22] S'il y a

n variables propositionnelles, il y a 2n valuations possibles.

La table suivante présente les définitions sémantiques formelles des connecteurs logiques. Elle n'est pas seulement un outil de calcul, mais l'incarnation même de la sémantique en logique propositionnelle.

Table 1.1 : Tables de Vérité des Connecteurs Logiques [13]

p	q	¬p	p∧q	p∨q	p→q	p↔q
1	1	0	1	1	1	1
1	0	0	0	1	0	0

0	1	1	0	1	1	0
0	0	1	0	0	1	1

Une attention particulière doit être portée à la sémantique de l'implication ($p \rightarrow q$). Elle est fausse uniquement dans le cas où l'antécédent (p) est vrai et le conséquent (q) est faux. Elle est donc vraie chaque fois que l'antécédent est faux, un principe connu sous le nom de *ex falso sequitur quodlibet* ("du faux découle n'importe quoi").[14] Cette définition, bien que parfois contre-intuitive par rapport à l'usage causal du "si... alors..." dans le langage naturel, est absolument cruciale pour la rigueur du raisonnement mathématique. Elle capture l'idée que d'une prémisse fausse, on peut déduire n'importe quelle conclusion sans invalider le raisonnement lui-même.

1.2. Analyse Sémantique des Formules

Une fois la sémantique établie, nous pouvons classer les formules en fonction de leurs propriétés de vérité et définir les relations fondamentales qui existent entre elles. Ces concepts sont au cœur du raisonnement automatisé.

1.2.1. Classification des Formules

En fonction de leur comportement sur l'ensemble de toutes les valuations possibles, les formules se répartissent en trois catégories mutuellement exclusives.[19]

- **Tautologie :** Une formule F est une **tautologie** (ou est **valide**) si elle est vraie pour *toute* valuation v. On note ce fait $\models F$. Une tautologie représente une vérité logique universelle, indépendante de l'état du monde. L'exemple paradigmatique est la *loi du tiers exclu*, $p \vee \neg p$, qui est toujours vraie, que p soit vrai ou faux.[24]
- **Contradiction :** Une formule F est une **contradiction** (ou une **antilogie**, ou est **insatisfaisable**) si elle est fausse pour *toute* valuation v. Une contradiction représente une fausseté logique universelle. L'exemple le plus simple est la *loi de non-contradiction*, $p \wedge \neg p$.[24]
- **Formule Contingente :** Une formule F est **contingente** si elle n'est ni une tautologie ni une contradiction. Sa valeur de vérité dépend de la valuation choisie. Il existe au moins une valuation qui la rend vraie et au moins une qui la rend fausse.[18] Les variables propositionnelles elles-mêmes sont les formules contingentes les plus simples.

1.2.2. Concepts Centraux pour le Raisonnement Automatique

Ces classifications mènent à des notions plus fines, essentielles pour les applications informatiques.

- **Modèle :** Une valuation v est un **modèle** pour une formule F si v rend F vraie, c'est-à-dire $v^{-}(F)=1$. On note alors $v \vDash F$ et on dit que "v satisfait F".[19]
- **Satisfiabilité (SAT) :** Une formule F est **satisfiable** si elle admet au moins un modèle, c'est-à-dire s'il existe au moins une valuation qui la rend vraie.[17] Une formule est donc satisfiable si et seulement si elle n'est pas une contradiction. Le problème de déterminer si une formule donnée est satisfiable, connu sous le nom de problème **SAT**, est l'un des problèmes les plus étudiés en informatique théorique.

1.2.3. Relations entre Formules

La sémantique nous permet également de formaliser les relations de déduction et d'équivalence entre différentes formules.

- **Conséquence Logique :** Une formule G est une **conséquence logique** d'un ensemble de formules $\Gamma=\{F1,\ldots,Fn\}$ (noté $\Gamma \vDash G$) si tout modèle qui satisfait simultanément toutes les formules de Γ est aussi un modèle de G.[6] Cela formalise l'idée d'un argument valide : si les prémisses (
  Fi) sont vraies, alors la conclusion (G) doit nécessairement être vraie.
- **Équivalence Logique :** Deux formules F et G sont **logiquement équivalentes** (noté $F \equiv G$) si elles ont la même table de vérité, c'est-à-dire si pour toute valuation v, $v^{-}(F)=v^{-}(G)$.[4] Des formules équivalentes sont interchangeables dans n'importe quel contexte logique sans en altérer la valeur de vérité.

1.2.4. L'Interconnexion des Concepts Sémantiques

Ces différentes notions sémantiques ne sont pas isolées ; elles forment un réseau de concepts étroitement liés, où la résolution d'un type de problème peut être ramenée à la résolution d'un autre. Cette interdépendance est fondamentale, car elle démontre qu'un algorithme capable de résoudre un seul de ces problèmes fondamentaux peut, en principe, les résoudre tous.

1. **Équivalence et Tautologie :** Deux formules F et G sont logiquement équivalentes si et seulement si la formule $(F \leftrightarrow G)$ est une tautologie.[4] Vérifier l'équivalence de deux formules revient donc à vérifier la validité d'une seule formule.
2. **Conséquence et Tautologie :** Une formule G est une conséquence logique d'un ensemble de prémisses {F1,…,Fn} si et seulement si la formule $(F1 \wedge \cdots \wedge Fn) \rightarrow G$ est une tautologie.[19] La notion de déduction valide est ainsi ramenée à la vérification d'une tautologie.
3. **Tautologie et Contradiction :** Une formule F est une tautologie si et seulement si sa négation, ¬F, est une contradiction (insatisfaisable).[19]

Ces réductions convergent vers un point central : le problème de la **satisfiabilité (SAT)**. Si nous disposons d'un algorithme capable de déterminer si une formule est satisfiable, nous pouvons résoudre tous les autres problèmes

sémantiques :

- Pour vérifier si F est une **tautologie**, on vérifie si ¬F est insatisfiable.
- Pour vérifier si F est une **contradiction**, on vérifie si F est insatisfiable.
- Pour vérifier si F≡G, on vérifie si ¬(F↔G) est insatisfiable.
- Pour vérifier si F⊨G, on vérifie si F∧¬G est insatisfiable.

Cette convergence explique pourquoi le problème SAT est considéré comme le problème prototypique de la NP-complétude et pourquoi des décennies de recherche ont été consacrées au développement de **solveurs SAT** efficaces.[30] Un solveur SAT n'est pas seulement un outil pour un problème de niche ; c'est un moteur de raisonnement universel pour toute la logique propositionnelle.

1.3. Équivalences Notables et Formes Normales

Si les tables de vérité fournissent une méthode sémantique pour analyser les formules, elles deviennent rapidement impraticables à mesure que le nombre de variables augmente (2n lignes). Une approche alternative, plus algébrique et syntaxique, consiste à transformer les formules en utilisant des règles de réécriture qui préservent l'équivalence logique.

1.3.1. L'Algèbre des Propositions

Les lois d'équivalence logique forment une véritable algèbre sur l'ensemble des formules. Elles permettent de manipuler, simplifier et canoniser les expressions logiques. Le tableau suivant recense les lois les plus fondamentales.

Table 1.2 : Principales Lois d'Équivalence Logique

Nom de la Loi	Forme avec ∧	Forme avec ∨	Source(s)
Commutativité	F∧G≡G∧F	F∨G≡G∨F	[17]
Associativité	(F∧G)∧H≡F∧(G∧H)	(F∨G)∨H≡F∨(G∨H)	[19]
Idempotence	F∧F≡F	F∨F≡F	[17]
Distributivité	F∧(G∨H)≡(F∧G)∨(F∧H	F∨(G∧H)≡(F∨G)∧(F∨H	[21]

	)	)	
Lois de De Morgan	¬(F∧G)≡¬F∨¬G	¬(F∨G)≡¬F∧¬G	[21]
Absorption	F∧(F∨G)≡F	F∨(F∧G)≡F	[17]
Éléments Neutres	F∧⊤≡F	F∨⊥≡F	[17]
Éléments Absorbants	F∧⊥≡⊥	F∨⊤≡⊤	[17]
Double Négation	¬¬F≡F	[21]
Tiers Exclu	F∨¬F≡⊤	[17]
Non-Contradiction	F∧¬F≡⊥	[17]
Définition Implication	F→G≡¬F∨G	[17]
Définition Équivalence	F↔G≡(F→G)∧(G→F)	[21]

Ces lois peuvent être prouvées formellement, soit en construisant les tables de vérité pour les deux côtés de l'équivalence et en vérifiant qu'elles sont identiques [23], soit par des preuves algébriques qui utilisent des lois déjà établies pour en dériver de nouvelles.[25]

1.3.2. Formes Normales

Pour de nombreuses applications algorithmiques, il est avantageux de travailler avec des formules qui ont une structure syntaxique standardisée, ou canonique. Les formes normales les plus importantes sont la Forme Normale Conjonctive (FNC) et la Forme Normale Disjonctive (FND).

- **Définitions Préliminaires** [15] :
  - Un **littéral** est une variable propositionnelle (p) ou sa négation (¬p).
  - Une **clause** (ou somme élémentaire) est une disjonction de littéraux (ex: p∨¬q∨r).
  - Un **terme** (ou produit élémentaire) est une conjonction de littéraux (ex: p∧¬q∧r).

- **Forme Normale Conjonctive (FNC)** : Une formule est en FNC si elle est une conjonction de clauses. Exemple : $(p \vee \neg q) \wedge (\neg p \vee r) \wedge q$.[34]
- **Forme Normale Disjonctive (FND)** : Une formule est en FND si elle est une disjonction de termes. Exemple : $(p \wedge \neg q) \vee (\neg p \wedge r) \vee q$.[34]

Un théorème fondamental de la logique propositionnelle stipule que **toute formule possède une FNC et une FND qui lui sont logiquement équivalentes**.[35] Cette propriété garantit que nous pouvons toujours transformer une formule quelconque en une de ces formes standards.

1.3.3. Algorithmes de Conversion en Formes Normales

Il existe deux approches principales pour convertir une formule en forme normale.

Table 1.3: Algorithmes de Conversion en FNC/FND

Méthode	Description	Procédure pour la FND	Procédure pour la FNC	Source(s)
Sémantique	Basée sur la table de vérité de la formule.	Pour chaque ligne de la table où la formule est **vraie** (valeur 1), construire un terme (conjonction) qui correspond à cette valuation (le minterme). La FND est la disjonction de tous ces termes.	Pour chaque ligne de la table où la formule est **fausse** (valeur 0), construire une clause (disjonction) qui est la négation de cette valuation (le maxterme). La FNC est la conjonction de toutes ces clauses.	35
Syntaxique	Basée sur l'application successive des lois	1. **Éliminer ↔ et →** en utilisant leurs définitions ($A \leftrightarrow B \equiv \ldots$,	1. **Éliminer ↔ et →**. 2. **Pousser les négations** vers l'intérieur.	21

	d'équivalence.	A→B≡¬A∨B). 2. **Pousser les négations** vers l'intérieur en utilisant les lois de De Morgan et la double négation, jusqu'à ce qu'elles ne portent que sur des variables (mise en forme normale négative). 3. **Appliquer la distributivité** de ∧ sur ∨ pour "sortir" les conjonctions et obtenir une disjonction de conjonctions.	3. **Appliquer la distributivité** de ∨ sur ∧ pour "sortir" les disjonctions et obtenir une conjonction de disjonctions.	

1.3.4. Le Compromis entre Équivalence et Satisfiabilité

Bien que les méthodes ci-dessus garantissent la conversion en une forme normale *équivalente*, elles présentent un inconvénient majeur : la taille de la formule résultante peut croître de manière exponentielle par rapport à la taille de la formule originale.[34] Par exemple, la conversion de la formule

(p1∧q1)∨(p2∧q2)∨⋯∨(pn∧qn) en FNC par distributivité génère 2n clauses. Cette explosion combinatoire rend l'approche syntaxique directe impraticable pour des applications réelles comme la vérification de circuits ou la résolution de problèmes de planification.

C'est ici qu'intervient une distinction subtile mais cruciale : la différence entre **l'équivalence logique** et **l'équisatisfiabilité**. Deux formules sont équisatisfiables si elles sont soit toutes les deux satisfiables, soit toutes les deux insatisfiables. L'équisatisfiabilité est une relation plus faible que l'équivalence (deux formules équivalentes sont toujours équisatisfiables, mais la réciproque est fausse).

Pour de nombreuses applications, notamment la résolution SAT, il n'est pas nécessaire de préserver l'équivalence exacte de la formule originale. Il suffit de générer une nouvelle formule en FNC qui est satisfiable si et seulement si l'originale l'est. La **transformation de Tseitin** est une technique algorithmique ingénieuse qui accomplit précisément cela, en temps

linéaire.[34]

Le principe de la transformation de Tseitin est d'introduire de nouvelles variables propositionnelles pour représenter les sous-formules. Pour chaque sous-formule S de la formule originale F, on introduit une nouvelle variable xS et on ajoute un ensemble de clauses qui forcent l'équivalence xS↔S. Par exemple, pour une sous-formule A∧B, on introduit une nouvelle variable xA∧B et on ajoute les clauses équivalentes à xA∧B↔(xA∧xB), qui sont (¬xA∧B∨xA), (¬xA∧B∨xB), et (¬xA∨¬xB∨xA∧B). En appliquant ce processus à toutes les sous-formules, on obtient une grande conjonction de petites clauses, à laquelle on ajoute la clause finale [xF] (pour forcer la formule entière à être vraie). La FNC résultante est équisatisfiable à la formule originale et sa taille est linéaire par rapport à celle-ci. Ce compromis est au cœur de l'efficacité des solveurs SAT modernes.

1.3.5. Minimisation des Formes Normales

Dans des domaines comme la conception de circuits logiques, l'objectif n'est pas seulement de trouver une forme normale, mais de trouver la forme normale *la plus simple* (avec le moins de littéraux ou de termes/clauses), car cela se traduit directement par un circuit moins coûteux et plus rapide. Le problème de la minimisation de fonctions booléennes est un problème d'optimisation complexe.

L'**algorithme de Quine-McCluskey** est une méthode tabulaire systématique pour trouver une FND minimale pour une fonction booléenne donnée.[43] Il est plus systématique que les tableaux de Karnaugh et peut être programmé pour un plus grand nombre de variables. L'algorithme se déroule en deux étapes majeures.[45]

1. **Identification des Implicants Premiers :**
    - On liste tous les mintermes (les valuations qui rendent la fonction vraie) de la fonction.
    - On groupe les mintermes par nombre de '1' dans leur représentation binaire.
    - On compare itérativement les termes de groupes adjacents. Si deux termes ne diffèrent que par un seul bit, ils sont fusionnés en un nouveau terme où le bit différent est remplacé par un tiret - (qui signifie "indifférent"). Les deux termes originaux sont marqués comme "couverts".
    - On répète ce processus de fusion avec les nouveaux termes jusqu'à ce qu'aucune fusion ne soit plus possible.
    - Les termes qui n'ont pas été couverts à l'issue de ce processus sont les **implicants premiers** de la fonction. Un implicant premier est un terme qui ne peut être simplifié davantage tout en continuant à n'impliquer que des mintermes de la fonction originale.[46]
2. **Construction de la Table des Implicants Premiers :**
    - On construit une table avec les implicants premiers en lignes et les mintermes originaux en colonnes.
    - On coche une case (i,j) si l'implicant premier i "couvre" (implique) le minterme j.
    - On identifie les **implicants premiers essentiels** : ce sont ceux qui sont les seuls à couvrir un certain minterme. Ces implicants doivent obligatoirement faire partie de la solution minimale.
    - On sélectionne les implicants premiers essentiels et on retire de la table tous les mintermes qu'ils couvrent.
    - S'il reste des mintermes non couverts, on choisit un sous-ensemble minimal d'implicants premiers restants pour couvrir les mintermes restants. Cette dernière étape est un problème de couverture d'ensemble (set cover), qui est NP-difficile, mais souvent gérable en pratique pour des tailles raisonnables.

La disjonction des implicants premiers essentiels et des implicants choisis à la dernière étape forme la FND minimale de

la fonction.

Partie II : La Logique des Prédicats – Un Langage pour les Objets et leurs Relations

Si la logique propositionnelle fournit l'algèbre fondamentale du raisonnement, son pouvoir expressif reste sévèrement limité. Elle ne peut pas pénétrer la structure interne des propositions. Pour elle, "Tous les hommes sont mortels" et "Socrate est un homme" sont simplement deux atomes distincts, p et q, sans aucun lien logique apparent. Il est donc impossible, dans ce cadre, de déduire formellement que "Socrate est mortel".[12] Pour capturer la richesse du raisonnement mathématique et du langage naturel, il nous faut un langage plus puissant, capable de parler d'objets, de leurs propriétés, des relations qui les unissent, et de faire des affirmations générales sur eux. C'est le rôle de la logique du premier ordre, ou calcul des prédicats.

2.1. Des Propositions aux Prédicats : Limites et Extensions

La transition vers la logique des prédicats est motivée par la nécessité de surmonter les limitations inhérentes au calcul propositionnel.

2.1.1. Limites de la Logique Propositionnelle

La principale limitation est l'incapacité à analyser la structure sujet-prédicat des énoncés. Des affirmations comme :

1. Tous les informaticiens aiment la logique.
2. Alice est une informaticienne.

Ne peuvent être formalisées qu'en tant que deux variables, p et q. La conclusion évidente, "Alice aime la logique", ne peut être dérivée. La logique propositionnelle manque des outils pour exprimer la notion de "tous", pour appliquer une propriété générale ("aimer la logique") à un individu spécifique ("Alice"), et pour relier les concepts ("informaticien") entre les phrases.[12]

2.1.2. Introduction Intuitive des Nouveaux Concepts

La logique des prédicats enrichit notre langage formel avec trois nouveaux types d'éléments [13] :

- **Termes :** Des expressions qui désignent des **objets** ou des **individus** dans un univers de discours. Les termes peuvent être des **constantes** (comme Socrate, 0) ou des **variables** (x, y,...) qui représentent des objets non spécifiés. Ils peuvent aussi être construits à l'aide de **fonctions** (comme père_de(x) ou successeur(n)).
- **Prédicats :** Des expressions qui dénotent des **propriétés** d'objets ou des **relations** entre objets. Un prédicat appliqué à des termes forme une proposition atomique qui est soit vraie, soit fausse. Par exemple, Homme(x) est un prédicat à un argument (unaire) qui exprime la propriété "être un homme". Père(x, y) est un prédicat à deux arguments (binaire) qui exprime la relation "x est le père de y".[13] L'arité d'un prédicat est le nombre d'arguments qu'il requiert.[27]
- **Quantificateurs :** Des symboles qui permettent de faire des affirmations générales sur les variables.
    - Le **quantificateur universel** ∀ ("pour tout") permet d'affirmer qu'une propriété est vraie pour tous les objets du domaine. ∀x,Homme(x)→Mortel(x) se lit "Pour tout x, si x est un homme, alors x est mortel".[25]
    - Le **quantificateur existentiel** ∃ ("il existe") permet d'affirmer qu'il existe au moins un objet dans le domaine qui possède une certaine propriété. ∃x,Informaticien(x)∧AimeLaLogique(x) se lit "Il existe au moins un x tel que x est un informaticien et x aime la logique".[25]

Table 2.1 : Comparaison Logique Propositionnelle vs. Logique des Prédicats

Caractéristique	Logique Propositionnelle	Logique des Prédicats (Premier Ordre)
Unité de base	Proposition atomique (p,q)	Termes (objets) et prédicats (propriétés/relations)
Structure interne	Aucune	Les propositions sont structurées (ex: P(f(c),x))
Sémantique	Valeurs de vérité assignées aux atomes	Interprétation dans un domaine d'objets
Généralisation	Impossible	Possible via les quantificateurs (∀,∃)
Expressivité	Limitée à des combinaisons booléennes de faits	Riche, capable de formaliser les mathématiques et le langage naturel

2.2. Syntaxe de la Logique du Premier Ordre

La syntaxe de la logique du premier ordre est une extension de celle de la logique propositionnelle, définissant un vocabulaire plus riche et des règles de formation plus complexes.

2.2.1. Alphabet Étendu

Un langage du premier ordre est défini par sa **signature**, qui spécifie les symboles non logiques. L'alphabet complet comprend [47] :

- **Symboles logiques (communs à tous les langages du premier ordre) :**
  - Connecteurs logiques : $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
  - Quantificateurs : $\forall, \exists$
  - Parenthèses : (,)
  - Un ensemble infini de variables : x,y,z,x1,…
- **Symboles non logiques (spécifiques à une signature) :**
  - Un ensemble de symboles de **constantes** : c1,c2,…
  - Un ensemble de symboles de **fonctions**, chacun avec une arité $n \geq 1$: f1,f2,…
  - Un ensemble de symboles de **prédicats** (ou relations), chacun avec une arité $n \geq 0$: P1,P2,… (les prédicats d'arité 0 sont équivalents aux variables propositionnelles).

2.2.2. Définition Inductive des Termes et Formules

À partir de cet alphabet, on construit deux types d'expressions syntaxiques : les termes et les formules.

Termes

Un **terme** est une expression qui a pour but de nommer un objet du domaine. L'ensemble des termes est défini inductivement [13] :

- **(Base)** Toute variable est un terme.
- **(Base)** Toute constante est un terme.
- **(Induction)** Si f est un symbole de fonction d'arité n et si t1,…,tn sont des termes, alors f(t1,…,tn) est un terme.

Par exemple, si c est une constante, x une variable, et f une fonction binaire, alors c, x, f(x,c), et f(c,f(x,x)) sont des termes.

Formules

Une **formule** est une expression qui a pour but de faire une affirmation (vraie ou fausse) sur les objets. L'ensemble des formules est également défini inductivement [13] :

1. **Formules Atomiques :**
   - Si P est un symbole de prédicat d'arité n et si t1,…,tn sont des termes, alors P(t1,…,tn) est une formule (appelée formule atomique).
   - Si t1 et t2 sont des termes, t1=t2 est une formule atomique (si le langage inclut l'égalité).
2. **Formules Complexes :**
   - Si F et G sont des formules, alors (¬F), (F∧G), (F∨G), (F→G), et (F↔G) sont des formules.
   - Si F est une formule et x est une variable, alors (∀xF) et (∃xF) sont des formules.

2.2.3. Quantification et Variables

La distinction entre variables libres et liées est l'un des aspects les plus subtils et les plus importants de la syntaxe du premier ordre.

- **Portée (Scope) :** Dans une formule (∀xF) ou (∃xF), la formule F est appelée la **portée** du quantificateur.[13] Le quantificateur "agit" sur toutes les occurrences de
  x à l'intérieur de cette portée.
- **Variables Libres et Liées :** Une occurrence d'une variable x dans une formule est dite **liée** si elle se trouve dans la portée d'un quantificateur ∀x ou ∃x. Sinon, elle est dite **libre**.[6]
  - Dans la formule ∀x(P(x)→Q(y)), l'occurrence de x dans P(x) est liée. L'occurrence de y dans Q(y) est libre.
  - Dans la formule (∃xP(x))∧Q(x), la première occurrence de x est liée, mais la seconde est libre. Pour éviter cette ambiguïté, il est de bonne pratique de renommer les variables liées pour qu'elles soient distinctes des variables libres.
- **Formules Closes (Énoncés) :** Une formule qui ne contient aucune variable libre est appelée une **formule close** ou un **énoncé**. Seules les formules closes ont une valeur de vérité qui peut être déterminée indépendamment d'une assignation de valeurs aux variables.[54] Elles font des affirmations générales sur un domaine. Par exemple, ∀x∃y(y>x) est un énoncé sur les nombres, tandis que y>x n'est pas un énoncé car sa vérité dépend des valeurs assignées à x et y.

2.3. Sémantique de la Logique du Premier Ordre

La sémantique de la logique du premier ordre est considérablement plus riche que celle de la logique propositionnelle.

Une formule n'a plus une valeur de vérité absolue ; sa vérité est toujours relative à une **interprétation** dans un **modèle** donné. C'est le domaine d'étude de la **théorie des modèles**.[50] Une formule comme

$\forall x \exists y P(x,y)$ n'est ni vraie ni fausse en soi. Elle est vraie dans le modèle des entiers naturels où $P(x,y)$ signifie "$y>x$", mais elle est fausse dans le même modèle si $P(x,y)$ signifie "$y<x$". La sémantique formelle a donc pour but de définir précisément les conditions sous lesquelles une formule est vraie *dans une structure donnée*.

2.3.1. Structure d'Interprétation (Modèle)

Une **structure d'interprétation** (ou **modèle**) M pour un langage du premier ordre donné est un couple $M=(D,I)$ où [50] :

1. D est un ensemble non vide appelé le **domaine** ou l'univers du discours. C'est l'ensemble des objets sur lesquels les variables et les quantificateurs portent.
2. I est une **fonction d'interprétation** qui associe chaque symbole non logique du langage à un élément, une fonction ou une relation sur le domaine D :
    - Pour chaque symbole de constante c, $I(c)$ est un élément de D.
    - Pour chaque symbole de fonction f d'arité n, $I(f)$ est une fonction de D^n dans D.
    - Pour chaque symbole de prédicat P d'arité n, $I(P)$ est une relation sur D^n (c'est-à-dire un sous-ensemble de D^n).

2.3.2. Satisfaction d'une Formule

Pour évaluer la vérité d'une formule contenant des variables libres, nous avons besoin d'une **assignation** (ou valuation de variables) σ, qui est une fonction associant chaque variable à un élément du domaine D.

On définit alors la notion de **satisfaction**, notée $M,\sigma \models F$ ("le modèle M satisfait la formule F sous l'assignation σ"), par induction sur la structure de la formule F.[13]

D'abord, on étend l'interprétation I et l'assignation σ à tous les termes :

- $[c]\sigma M = I(c)$
- $[x]\sigma M = \sigma(x)$
- $[f(t1,\ldots,tn)]\sigma M = I(f)([t1]\sigma M,\ldots,[tn]\sigma M)$

Ensuite, on définit la satisfaction pour les formules :

- **Formules atomiques :**
    - $M,\sigma \models P(t1,\ldots,tn)$ si et seulement si le n-uplet $([t1]\sigma M,\ldots,[tn]\sigma M)$ appartient à la relation $I(P)$.
- **Connecteurs logiques :**
    - $M,\sigma \models \neg F$ ssi il n'est pas le cas que $M,\sigma \models F$.
    - $M,\sigma \models F \wedge G$ ssi $M,\sigma \models F$ et $M,\sigma \models G$.

    - (Les autres connecteurs suivent les règles propositionnelles).
- **Quantificateurs :**
    - $M,\sigma \models \forall x F$ si et seulement si pour **tout** élément $d \in D$, on a $M,\sigma[x \mapsto d] \models F$, où $\sigma[x \mapsto d]$ est l'assignation identique à σ sauf qu'elle assigne d à x.
    - $M,\sigma \models \exists x F$ si et seulement s'il **existe** un élément $d \in D$ tel que $M,\sigma[x \mapsto d] \models F$.

Si une formule F est close (sans variable libre), sa satisfaction ne dépend pas de l'assignation σ. On peut alors simplement écrire $M \models F$ et dire que "F est vraie dans le modèle M" ou que "M est un modèle de F".

2.3.3. Validité et Satisfiabilité

Les notions de validité et de satisfiabilité sont généralisées à partir de la logique propositionnelle [27] :

- Une formule F est **satisfiable** s'il existe au moins un modèle M et une assignation σ tels que $M,\sigma \models F$.
- Une formule F est **valide** (ou une **tautologie du premier ordre**) si elle est vraie dans **tous** les modèles possibles pour **toutes** les assignations possibles. On note alors $\models F$.

Des exemples de formules valides incluent $\forall x P(x) \rightarrow \exists x P(x)$ (à condition que le domaine soit non vide) ou encore $\neg(\exists x P(x)) \leftrightarrow \forall x \neg P(x)$ (loi de De Morgan pour les quantificateurs).

Partie III : Systèmes de Preuve Formelle – La Mécanique de la Déduction

La sémantique définit la notion de vérité et de conséquence logique ($\models$). Cependant, pour vérifier qu'une formule est une conséquence logique d'un ensemble de prémisses, il faudrait en théorie examiner tous les modèles possibles, ce qui est infini et donc infaisable. Les **systèmes de preuve formelle** (ou systèmes de déduction) offrent une approche alternative, purement syntaxique. Ils fournissent un ensemble de règles d'inférence qui permettent de dériver des formules (des théorèmes) à partir d'un ensemble d'axiomes ou d'hypothèses, par simple manipulation de symboles. L'objectif est de créer un système tel que les formules "prouvables" (dérivables syntaxiquement, noté $\vdash$) coïncident exactement avec les formules "valides" (vraies sémantiquement, noté $\models$). C'est l'objet des théorèmes de correction et de complétude.

3.1. La Déduction Naturelle

La déduction naturelle, développée par Gerhard Gentzen, est un système de preuve conçu pour refléter aussi fidèlement

que possible la structure du raisonnement logique humain. Son principe central est le raisonnement à partir d'**hypothèses** qui peuvent être introduites temporairement puis "déchargées" ou "annulées".[58] Par exemple, pour prouver une implication

A→B, la méthode naturelle consiste à supposer A, à en déduire B, puis à conclure que A→B est vrai, déchargeant ainsi l'hypothèse initiale A.

Le système est organisé autour de paires de règles pour chaque connecteur et quantificateur : une **règle d'introduction** qui explique comment construire une formule contenant ce symbole, et une **règle d'élimination** qui explique comment utiliser une formule contenant ce symbole pour en déduire autre chose.

3.1.1. Règles d'Introduction et d'Élimination

Le tableau suivant présente les règles de la déduction naturelle pour la logique classique. Une déduction est représentée par un arbre de preuve, où les feuilles sont des axiomes ou des hypothèses, les nœuds internes sont des applications de règles, et la racine est la conclusion. La notation [A] au-dessus d'une prémisse indique que l'hypothèse A est déchargée par l'application de la règle.

Table 3.1 : Système de Règles de la Déduction Naturelle [59]

Connecteur	Règle d'Introduction	Règle d'Élimination
Axiome	AI⊢A (ax)	
Conjonction (∧)	Γ,Δ⊢A∧BΓ⊢AΔ⊢B(∧I)	Γ⊢AΓ⊢A∧B(∧Eg)Γ⊢BΓ⊢A∧B (∧Ed)
Implication (→)	Γ⊢A→BΓ,[A]⊢B(→I)	Γ,Δ⊢BΓ⊢AΔ⊢A→B(→E,Modus Ponens)
Disjonction (∨)	Γ⊢A∨BΓ⊢A(∨Ig)Γ⊢A∨BΓ⊢B(∨Id)	Γ,Δ,Θ⊢CΓ⊢A∨BΔ,[A]⊢CΘ,⊢C (∨E,Raisonnement par cas)
Négation (¬)	Γ⊢¬AΓ,[A]⊢⊥(¬I)	Γ,Δ⊢⊥Γ⊢AΔ⊢¬A(¬E)
Absurde (⊥)		Γ⊢AΓ⊢⊥(⊥E,Ex falso quodlibet)

Raisonnement par l'absurde		$\Gamma \vdash A \Gamma, [\neg A] \vdash \bot$ (RAA)
Quant. Universel ($\forall$)	$\Gamma \vdash \forall x A \Gamma \vdash A[x \mapsto y]$ ($\forall$I)*	$\Gamma \vdash A[x \mapsto t] \Gamma \vdash \forall x A$ ($\forall$E)
Quant. Existentiel ($\exists$)	$\Gamma \vdash \exists x A \Gamma \vdash A[x \mapsto t]$ ($\exists$I)	$\Gamma, \Delta \vdash B \Gamma \vdash \exists x A \Delta, [A[x \mapsto y]] \vdash B$ ($\exists$E)**

* Condition pour ($\forall$I) : La variable y ne doit pas apparaître libre dans Γ ou dans $\forall x A$. Cela garantit que la preuve de $A[x \mapsto y]$ est générique et ne dépend pas d'une propriété spécifique de y.
** Condition pour ($\exists$E) : La variable y ne doit pas apparaître libre dans Γ, Δ, B ou $\exists x A$. Cela garantit que y agit comme un témoin arbitraire pour l'existence.

3.1.2. Construction de Preuves Formelles

Illustrons le système avec la preuve de la transitivité de l'implication : $(A \to B), (B \to C) \vdash (A \to C)$.

1. $(A \to B) \vdash (A \to B)$ (Axiome)
2. $(B \to C) \vdash (B \to C)$ (Axiome)
3. $A \vdash A$ (Axiome, hypothèse temporaire)
4. $(A \to B), A \vdash B$ (Modus Ponens sur 1 et 3)
5. $(A \to B), (B \to C), A \vdash C$ (Modus Ponens sur 2 et 4)
6. $(A \to B), (B \to C) \vdash (A \to C)$ (Intro $\to$ sur 5, décharge l'hypothèse A)

La dernière ligne est la conclusion souhaitée. Chaque étape est une application rigoureuse d'une règle d'inférence.

3.1.3. Logique Constructive et Correspondance de Curry-Howard

Le choix des règles n'est pas anodin. La règle du raisonnement par l'absurde (RAA), ou de manière équivalente l'axiome du tiers exclu ($A \vee \neg A$), est la marque de la **logique classique**. Si l'on retire cette règle, on obtient un système plus faible mais aux propriétés remarquables : la **logique intuitionniste** ou **constructive**.[60]

Dans une preuve constructive, pour démontrer $A \vee B$, il faut soit fournir une preuve de A, soit fournir une preuve de B. Pour démontrer $\exists x P(x)$, il faut exhiber un témoin t et fournir une preuve de $P(t)$. Les preuves d'existence non constructives (par exemple, "supposons qu'un tel x n'existe pas et dérivons une contradiction") ne sont pas valides.[62]

Cette distinction a des implications profondes en informatique. La **correspondance de Curry-Howard** établit un isomorphisme entre les preuves en logique intuitionniste et les programmes dans certains langages de programmation

typés (comme le lambda-calcul simplement typé). Dans cette correspondance :

- Une proposition est un type.
- Une preuve d'une proposition est un programme (un terme) de ce type.
- La simplification d'une preuve (élimination des détours) correspond à l'exécution (réduction) du programme.

Ainsi, prouver une formule de manière constructive, c'est écrire un programme qui la réalise. Cette connexion profonde entre logique et calcul est à la base des assistants de preuve comme Coq et Agda, où l'on peut écrire des programmes et prouver leur correction dans un cadre unifié.

3.2. Le Principe de Résolution

Contrairement à la déduction naturelle avec sa panoplie de règles, le **principe de résolution** est un système de preuve qui n'utilise qu'une seule règle d'inférence. Cette simplicité en fait un candidat idéal pour l'automatisation du raisonnement, et il est au cœur de la plupart des démonstrateurs de théorèmes modernes et des solveurs logiques.[63]

La méthode de résolution opère sur des formules qui ont été préalablement converties en une forme standard appelée **forme clausale**. La stratégie de preuve est une preuve par réfutation : pour démontrer qu'une conclusion F découle d'un ensemble de prémisses Γ, on montre que l'ensemble Γ∪{¬F} est insatisfaisable (contradictoire).[64]

3.2.1. Mise en Forme Clausale

La forme clausale est une généralisation de la FNC. Un ensemble de clauses est interprété comme la conjonction de ses clauses, et chaque clause est une disjonction de littéraux.

- **Logique Propositionnelle :** La conversion se fait en appliquant l'algorithme de mise en FNC vu précédemment. Il est crucial, pour l'efficacité, d'utiliser une transformation qui préserve la satisfiabilité (comme la transformation de Tseitin) plutôt que l'équivalence, afin d'éviter l'explosion de la taille de la formule.[65]
- **Logique du Premier Ordre :** La conversion est plus complexe et requiert plusieurs étapes :
  1. **Mise en Forme Normale Prénexe :** La formule est transformée en une formule équivalente où tous les quantificateurs apparaissent au début. Ceci est réalisé en utilisant des équivalences logiques pour "remonter" les quantificateurs (ex: $(\forall x A) \wedge B \equiv \forall x (A \wedge B)$ si x n'est pas libre dans B).[34]
  2. **Skolémisation :** Les quantificateurs existentiels sont éliminés. Chaque variable quantifiée existentiellement est remplacée par un **terme de Skolem**. Si le $\exists x$ est dans la portée de quantificateurs universels $\forall y1,\ldots,\forall yn$, alors x est remplacé par f(y1,…,yn), où f est un nouveau symbole de fonction (une **fonction de Skolem**). Si $\exists x$ n'est dans la portée d'aucun quantificateur universel, x est remplacé par une nouvelle constante (une **constante de Skolem**).[66] La skolémisation ne préserve pas l'équivalence, mais elle préserve la satisfiabilité.
  3. **Mise en FNC de la Matrice :** La partie sans quantificateur de la formule (la matrice) est convertie en FNC en utilisant les méthodes propositionnelles.
  4. **Suppression des Quantificateurs Universels :** Les $\forall$ restants sont implicites ; toutes les variables sont

considérées comme universellement quantifiées.

Le résultat est un ensemble de clauses du premier ordre.

3.2.2. La Règle de Résolution

Table 3.2: La Règle de Résolution et ses Applications

Cas	Règle d'Inférence	Description	Source(s)
Propositionnel	C1∨C2C1∨pC2∨¬p	À partir de deux clauses contenant un littéral et sa négation, on déduit une nouvelle clause (le **résolvant**) contenant tous les autres littéraux des deux clauses parentes.	63
Premier Ordre	(C1∨C2)σC1∨L1C2 ∨¬L2	L1 et L2 sont des littéraux atomiques. S'il existe une substitution σ (l'**unificateur le plus général**, ou MGU) telle que L1σ=L2σ, alors on peut déduire le résolvant. L'unification est l'algorithme qui trouve une telle substitution.	34

L'**unification** est le processus qui consiste à trouver une substitution de termes pour des variables qui rend deux expressions identiques. Par exemple, pour unifier P(x,f(a)) et P(g(y),f(z)), l'unificateur le plus général est la substitution σ={x↦g(y),z↦a}. L'unification est l'opération clé qui permet d'appliquer la résolution en logique du premier ordre.

3.2.3. La Preuve par Réfutation : Exemple Détaillé

Montrons que l'argument "Tous les hommes sont mortels. Socrate est un homme. Donc, Socrate est mortel." est valide.

1. **Formalisation :**
   - Prémisse 1 : $\forall x(H(x) \rightarrow M(x))$
   - Prémisse 2 : $H(s)$
   - Conclusion à prouver : $M(s)$
2. **Négation de la Conclusion :** $\neg M(s)$
3. **Mise en Forme Clausale :**
   - Prémisse 1 : $\forall x(\neg H(x) \vee M(x))$. La forme clausale est $\{\neg H(x) \vee M(x)\}$.
   - Prémisse 2 : $H(s)$. La forme clausale est $\{H(s)\}$.
   - Négation de la conclusion : $\neg M(s)$. La forme clausale est $\{\neg M(s)\}$.
   - Ensemble de clauses initial $S = \{\{\neg H(x) \vee M(x)\}, \{H(s)\}, \{\neg M(s)\}\}$.
4. **Dérivation par Résolution :**
   - (1) $\{\neg H(x) \vee M(x)\}$ (de S)
   - (2) $\{H(s)\}$ (de S)
   - (3) $\{\neg M(s)\}$ (de S)
   - (4) $\{M(s)\}$ (Résolvant de (1) et (2) avec l'unificateur $\sigma = \{x \mapsto s\}$)
   - (5) $\{\}$ (la clause vide, $\perp$) (Résolvant de (3) et (4))

Puisque nous avons dérivé la clause vide, l'ensemble de clauses initial est insatisfaisable. Par conséquent, la conclusion originale est une conséquence logique des prémisses.

Partie IV : Méthodes de Raisonnement et Applications Fondamentales

Les systèmes de preuve formels comme la déduction naturelle et la résolution fournissent les fondations rigoureuses de la logique. Cependant, dans la pratique quotidienne des mathématiques et de l'informatique, les preuves sont rarement écrites avec ce niveau de détail formel. Elles suivent plutôt des schémas de raisonnement de plus haut niveau. Cette dernière partie a pour but de connecter ces méthodes de preuve pratiques aux formalismes logiques que nous avons établis, montrant qu'elles ne sont pas des techniques ad-hoc mais des applications rigoureuses des principes logiques.

4.1. Techniques de Démonstration en Mathématiques et Informatique

Les méthodes de preuve les plus courantes peuvent être comprises comme des stratégies pour construire une déduction formelle.

Table 4.1: Structure des Méthodes de Preuve

Méthode	Objectif	Structure Logique	Justification Formelle	Source(s)
Preuve Directe	Prouver P→Q	Supposer P. En déduire Q par une suite d'étapes logiques.	Application directe de la règle d'introduction de l'implication (→I) en déduction naturelle.	[68]
Preuve par Contraposée	Prouver P→Q	Supposer ¬Q. En déduire ¬P.	Basée sur l'équivalence logique (P→Q)≡(¬Q→¬P). On construit une preuve directe de la contraposée.	[68]
Preuve par l'Absurde	Prouver P	Supposer ¬P. En déduire une contradiction (⊥).	Application directe de la règle du raisonnement par l'absurde (RAA) en déduction naturelle.	[68]

4.1.1. Le Principe des Tiroirs (Pigeonhole Principle)

Le principe des tiroirs est une technique de comptage simple mais d'une puissance surprenante pour prouver l'existence de certaines propriétés dans des ensembles finis.

- **Forme Simple :** Si n+1 objets (pigeons) sont placés dans n boîtes (tiroirs), alors au moins une boîte doit contenir plus d'un objet.[73]

- **Forme Généralisée :** Si n objets sont placés dans m boîtes, alors au moins une boîte doit contenir au moins ⌈n/m⌉ objets, où ⌈·⌉ est la fonction plafond.[74]

La preuve de ce principe est un exemple classique de raisonnement par l'absurde. Pour la forme simple : supposons que chaque boîte contienne au plus un objet. Comme il y a n boîtes, il y a au total au plus n×1=n objets. Ceci contredit le fait qu'il y a n+1 objets. Donc, l'hypothèse initiale est fausse, et au moins une boîte doit contenir plus d'un objet.[74]

La caractéristique la plus importante du principe des tiroirs est qu'il s'agit d'une **preuve d'existence non constructive**. Il garantit l'existence d'une boîte surpeuplée, mais ne donne aucune information sur laquelle c'est, ni comment la trouver.[74] Cette propriété en fait un outil puissant pour établir des bornes et des limitations fondamentales.

Une application classique et fondamentale en informatique concerne les **tables de hachage**. Une table de hachage est une structure de données qui mappe des clés d'un grand univers U à des indices dans un tableau de taille m, où m est généralement beaucoup plus petit que la taille de U. Ce mappage est réalisé par une fonction de hachage $h:U \to \{0,\dots,m-1\}$. Puisque l'ensemble des clés possibles (les "pigeons") est plus grand que le nombre d'indices de tableau (les "tiroirs"), le principe des tiroirs garantit que les **collisions** — deux clés distinctes $k1 \neq k2$ telles que $h(k1)=h(k2)$ — sont inévitables.[81] Toute la théorie des tables de hachage est donc construite non pas sur l'évitement des collisions, mais sur leur gestion efficace. Ce principe est également utilisé en cryptographie pour prouver l'existence de collisions dans les fonctions de hachage cryptographiques.[81]

4.2. Le Principe d'Induction : Raisonner sur les Structures Récursives

Le fil conducteur de ce chapitre a été la notion de définition inductive : nous avons défini les formules bien formées de manière inductive, puis les fonctions sur ces formules de manière récursive. Il est donc naturel qu'il existe une méthode de preuve spécifiquement adaptée à ces structures : le **raisonnement par induction**. L'induction et la récursion sont les deux faces d'une même médaille : la récursion est une méthode pour construire ou calculer, tandis que l'induction est une méthode pour raisonner sur ces constructions.[86] C'est sans doute le principe de preuve le plus important en informatique théorique, car il permet de prouver des propriétés sur des objets de taille potentiellement infinie (comme tous les entiers naturels) ou des structures de données récursives (comme les listes et les arbres).

4.2.1. Induction Simple et Forte sur **N**

- **Induction Simple (ou Faible) :** Pour prouver qu'une propriété P(n) est vraie pour tous les entiers naturels $n \geq n0$, on procède en deux étapes [89] :
  1. **Cas de Base :** On prouve que P(n0) est vraie.
  2. **Étape Inductive :** On suppose que P(k) est vraie pour un entier arbitraire $k \geq n0$ (c'est l'**hypothèse de récurrence**), et on utilise cette hypothèse pour prouver que P(k+1) est vraie.
- **Induction Forte (ou Complète) :** L'induction forte est une variante où l'hypothèse de récurrence est plus puissante [88] :

1. **Cas de Base :** On prouve P(n0).
2. **Étape Inductive :** On suppose que P(i) est vraie pour **tous** les entiers i tels que n0≤i≤k, et on utilise cette hypothèse pour prouver que P(k+1) est vraie.

Bien que l'induction forte semble plus puissante, les deux principes sont logiquement équivalents. Cependant, l'induction forte est souvent plus naturelle pour les problèmes où la propriété de k+1 dépend de plusieurs prédécesseurs, comme dans la preuve du théorème fondamental de l'arithmétique (tout entier supérieur à 1 est un produit de nombres premiers).

4.2.2. Induction Structurelle

L'induction structurelle généralise le principe d'induction à tout ensemble défini inductivement, comme l'ensemble des formules, des listes, ou des arbres binaires. La structure de la preuve par induction structurelle suit directement la structure de la définition inductive de l'ensemble.[87] Pour prouver qu'une propriété

P(x) est vraie pour tous les éléments x d'un ensemble inductif X :

1. **Cas de Base :** On prouve que P(b) est vraie pour chaque élément de base b de X.
2. **Étape Inductive :** Pour chaque règle de construction inductive r de X (par exemple, si x1,…,xn sont dans X, alors r(x1 ,…,xn) est dans X), on suppose que la propriété est vraie pour les composants (P(x1),…,P(xn) sont vraies) et on prouve qu'elle est alors vraie pour l'élément construit P(r(x1,…,xn)).

Par exemple, pour prouver une propriété sur tous les arbres binaires, on la prouverait pour l'arbre vide (cas de base), puis on supposerait qu'elle est vraie pour un sous-arbre gauche G et un sous-arbre droit D pour prouver qu'elle est vraie pour un nœud ayant G et D comme enfants.

4.2.3. Applications à la Preuve de Correction d'Algorithmes

L'induction est l'outil principal pour prouver formellement qu'un algorithme est correct. La correction totale d'un algorithme se compose de deux propriétés distinctes [97] :

1. **Correction Partielle :** *Si* l'algorithme termine, alors il produit le résultat attendu.
   - Pour les **algorithmes récursifs**, la correction partielle se prouve naturellement par induction (simple, forte ou structurelle) sur la structure des entrées.[98] On suppose que les appels récursifs sur des entrées "plus petites" retournent le bon résultat (hypothèse de récurrence) et on montre que la combinaison de ces résultats produit le bon résultat pour l'entrée actuelle.
   - Pour les **algorithmes itératifs** (avec des boucles), la correction partielle est prouvée à l'aide d'un **invariant de boucle**. Un invariant de boucle est une propriété qui est vraie avant la première itération et qui, si elle est vraie avant une itération, reste vraie après cette itération. On prouve par induction sur le nombre d'itérations que l'invariant est vrai à chaque étape. En combinant l'invariant avec la condition d'arrêt de la boucle, on peut

alors démontrer que le résultat final est correct.[97]

2. **Terminaison :** L'algorithme s'arrête en un temps fini pour toute entrée valide.
    - La terminaison est également prouvée par induction. On doit trouver une quantité (un "variant") qui est associée à l'état de l'algorithme, qui prend ses valeurs dans un ensemble bien fondé (comme les entiers naturels avec l'ordre ≥), et qui **décroît strictement** à chaque appel récursif ou à chaque itération de boucle.[97] Comme il ne peut y avoir de séquence infinie strictement décroissante dans un ensemble bien fondé, l'algorithme doit nécessairement terminer.

Conclusion

Ce chapitre a entrepris un voyage au cœur des fondements de l'informatique, en partant des briques élémentaires de la logique propositionnelle pour atteindre les structures expressives de la logique des prédicats, et en reliant les notions sémantiques de vérité aux mécanismes syntaxiques de la preuve. Plusieurs thèmes centraux ont émergé.

Premièrement, la **dualité syntaxe-sémantique** est le principe organisateur de tout langage formel. La capacité de manipuler des symboles selon des règles strictes (syntaxe) tout en garantissant que ces manipulations préservent la vérité (sémantique) est ce qui rend le raisonnement formel et la computation possibles. Deuxièmement, le pouvoir de l'**abstraction** nous a permis de passer d'énoncés atomiques à des prédicats sur des mondes d'objets, montrant comment des langages de plus en plus riches peuvent être construits sur des bases plus simples. Troisièmement, le lien intime entre les **définitions inductives et le raisonnement par induction** s'est révélé être la clé de voûte de la preuve en informatique, nous permettant de raisonner sur des structures et des processus potentiellement infinis. Enfin, nous avons vu l'importance des **compromis** fondamentaux, comme celui entre l'expressivité d'un langage et la complexité calculatoire de son analyse, ou celui entre l'équivalence logique et l'équisatisfiabilité, qui est au cœur de l'efficacité des outils de raisonnement automatique.

Les concepts introduits ici — satisfiabilité, modèles, déduction, unification, induction — ne sont pas de simples curiosités théoriques. Ils constituent le socle sur lequel reposent des domaines avancés qui seront explorés dans les chapitres suivants de cet ouvrage. La **vérification formelle** des systèmes critiques, qui garantit la sûreté des logiciels et du matériel, utilise des techniques de *model checking* et des démonstrateurs de théorèmes basés sur la logique des prédicats. La **sémantique des langages de programmation** s'appuie sur des systèmes logiques pour définir précisément le comportement des programmes. Le **raisonnement automatique en intelligence artificielle**, de la planification à la représentation des connaissances, utilise la résolution et d'autres calculs logiques comme moteurs d'inférence.[103] La

théorie des bases de données utilise la logique du premier ordre pour définir les langages de requête et pour raisonner sur l'intégrité des données.[103]

En somme, la logique n'est pas seulement un outil pour l'informaticien ; elle est l'échafaudage même sur lequel la discipline est construite. Une compréhension profonde de ses fondements est la condition sine qua non pour passer de la simple programmation à la véritable ingénierie de systèmes complexes, fiables et intelligents.

Ouvrages cités

1. Logique et fondements des mathématiques - Éditions Payot & Rivages, dernier accès : septembre 21,

2025, https://www.payot-rivages.fr/payot/livre/logique-et-fondements-des-math%C3%A9matiques-9782228884839
2. La LOGIQUE PROPOSITIONNELLE en 6 minutes - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=yR7tGmamCpo
3. Fondements Logiques pour l'Informatique - USTO-MB, dernier accès : septembre 21, 2025, https://www.univ-usto.dz/images/coursenligne/FLI_HY.pdf
4. Calcul propositionnel - LOGIQUE MATHÉMATIQUE - l'IRIF, dernier accès : septembre 21, 2025, https://www.irif.fr/~roziere/logiqueL3MI/calculPropPred.pdf
5. Syntaxe, sémantique et arguments philosophiques - Eduscol, dernier accès : septembre 21, 2025, https://eduscol.education.fr/document/19888/download
6. Introduction à la logique Support de cours - Mines Saint-Etienne, dernier accès : septembre 21, 2025, https://www.emse.fr/~zimmermann/Teaching/Logique/Livret/
7. Fondements de l'informatique Logique, modèles, et calculs, dernier accès : septembre 21, 2025, https://www.enseignement.polytechnique.fr/informatique/INF412/uploads/Main/chap3-goodINF412.pdf
8. Logique pour l'informatique - Mathieu Jaume , Matthieu Journault - Librairie Eyrolles, dernier accès : septembre 21, 2025, https://www.eyrolles.com/Informatique/Livre/logique-pour-l-informatique-9782340042612/
9. Fiche_Exposé_Logique_Proposi, dernier accès : septembre 21, 2025, https://www.scribd.com/document/841058557/Fiche-Expose-Logique-Propositionnelle
10. Logique propositionnelle: Introduction, Concepts | StudySmarter, dernier accès : septembre 21, 2025, https://www.studysmarter.fr/resumes/mathematiques/logique-et-fondements/logique-propositionnelle/
11. LOGIQUE L3 Informatique Salim Lardjane Université Bretagne Sud, dernier accès : septembre 21, 2025, http://web.univ-ubs.fr/lmba/lardjane/logique/c3.pdf
12. Partie I- La logique propositionnelle (ORDRE 0), dernier accès : septembre 21, 2025, https://elearning-facsci.univ-annaba.dz/mod/resource/view.php?id=1908
13. Logique des propositions, dernier accès : septembre 21, 2025, http://www.pageperso.lif.univ-mrs.fr/~alexis.nasr/Ens/IntroLing/logique.pdf
14. Logique propositionnelle =1Version préliminaire du cours. Tout retour sur la forme comme sur le fond est le bienvenu. - CRIL, dernier accès : septembre 21, 2025, https://www.cril.univ-artois.fr/~benferhat/logique.pdf
15. Logique mathématique - Syntaxe (Formalisation), dernier accès : septembre 21, 2025, https://www.ens-oran.dz/images/cours-en-ligne/sciences-exactes/Cours_web_gen/co/module_Logique_mathematique_7.html
16. Formule propositionnelle - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Formule_propositionnelle
17. Systèmes formels logique propositionnelle (partie 1) - Cedric-Cnam, dernier accès : septembre 21, 2025, https://cedric.cnam.fr/~pons/NFP108/PROPOSITION1.pdf
18. TD no 1 Calcul propositionnel — syntaxe et sémantique, dernier accès : septembre 21, 2025, https://pageperso.lis-lab.fr/~luigi.santocanale/teaching/1516teaching/LC/DOCS/tds1-8.pdf
19. 1. Logique propositionnelle - [Verimag], dernier accès : septembre 21, 2025, http://www-verimag.imag.fr/~wack/cours_inf242/node4.html
20. Sémantique de la logique propositionnelle (valuation, valeur ..., dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=KNDouBw7NYI
21. La logique propositionnelle, dernier accès : septembre 21, 2025, https://webusers.imj-prg.fr/~richard.lassaigne/coursLogique/logique-prop.pdf

22. La méthode des tables de vérité en logique propositionnelle - LACL, dernier accès : septembre 21, 2025, http://lacl.u-pec.fr/cegielski/logique/ch2.pdf
23. Tables de vérité, dernier accès : septembre 21, 2025, http://deptinfo.cnam.fr/Enseignement/CycleA/Anciens%3F/APA0/MVA/node37.html
24. Logique propositionnelle, dernier accès : septembre 21, 2025, http://l.roussarie.free.fr/IMG/pdf/ho-logique-2-5-2.pdf
25. 4.2. Tableau de vérité. Nous présentons ces définitions en forme de tableaux de vérité, où V:=vrai, et F:=faux. Pour la n - Département de mathématiques et statistique, dernier accès : septembre 21, 2025, https://dms.umontreal.ca/~broera/MAT1500Notes_3.pdf
26. Tautologie (logique) - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Tautologie_(logique)
27. La sémantique propositionnelle, dernier accès : septembre 21, 2025, https://www.univ-orleans.fr/lifo/Members/Isabelle.Tellier/poly_info_ling/linguistique008.html
28. Méthodes formelles Support de cours - Modèle Pages Personnelles, dernier accès : septembre 21, 2025, https://perso.atilf.fr/wsayer/wp-content/uploads/sites/20/2015/09/1supportMF.pdf
29. Chapitre 3 Lois de la logique propositionnelle Cours 4 Conséquence logique, équivalences, substitutions - l'IRIF, dernier accès : septembre 21, 2025, https://www.irif.fr/~kesner/enseignement/ol3/chap3-1.pdf
30. Combinations of Boolean Gröbner Bases and SAT Solvers, dernier accès : septembre 21, 2025, https://d-nb.info/1064306241/34
31. Problème SAT - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Probl%C3%A8me_SAT
32. Lois de distributivité [Connecteurs] - Unisciel, dernier accès : septembre 21, 2025, https://uel.unisciel.fr/mathematiques/logique1/logique1_ch01/co/apprendre_ch1_05.html
33. Lois ou théorème de De Morgan - Positron-libre, dernier accès : septembre 21, 2025, https://www.positron-libre.com/cours/electronique/logique-combinatoire/theoreme-morgan.php
34. Chapitre 3 Manipuler les formules de la logique - Laboratoire de Recherche en Informatique, dernier accès : septembre 21, 2025, https://www.lri.fr/~paulin/Logique/html/cours005.html
35. Formes normales, dernier accès : septembre 21, 2025, http://lecomte.al.free.fr/ressources/PARIS8_LSL/Formes-normales.pdf
36. PROJET TUTOR´E : Formes normales en logique propositionnelle - IDMC, dernier accès : septembre 21, 2025, https://idmc.univ-lorraine.fr/wp-content/uploads/2018/05/IDMC-2014_2015-ProjetTutore-RAPPORT-_lentschat_mercuriali_tiv.pdf
37. Formes Normales et leurs Propriétés - toposuranos.com/material, dernier accès : septembre 21, 2025, https://toposuranos.com/material/fr/formes-normales-et-leurs-proprietes/
38. Logique propositionnelle Plan Interprétation - définition Vocabulaire, dernier accès : septembre 21, 2025, https://perso.eleves.ens-rennes.fr/~afalq494/docs_mp2i/annexes/logique_slides_a_imp
39. Chapitre 4 Formes Normales - l'IRIF, dernier accès : septembre 21, 2025, https://www.irif.fr/~kesner/enseignement/ol3/chap4.pdf
40. Algorithme de Forme Normale et Applications - toposuranos.com/material, dernier accès : septembre 21, 2025, https://toposuranos.com/material/fr/algorithme-de-forme-normale-et-applications/
41. Modélisation et logique propositionnelle classique - CRIL, dernier accès : septembre 21, 2025, http://www.cril.univ-artois.fr/~salhi/HDR-YS.pdf
42. Forme normale disjonctive - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Forme_normale_disjonctive
43. Méthode de Quine-Mc Cluskey — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/M%C3%A9thode_de_Quine-Mc_Cluskey
44. Cm2algo QMC | PDF | Informatique théorique - Scribd, dernier accès : septembre 21, 2025,

https://fr.scribd.com/document/839870924/Cm2algo-Qmc
45. Quine–McCluskey algorithm - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Quine%E2%80%93McCluskey_algorithm
46. Quine McCluskey Method - GeeksforGeeks, dernier accès : septembre 21, 2025, https://www.geeksforgeeks.org/digital-logic/quine-mccluskey-method/
47. Chapitre III. Introduction Générale à la logique des Prédicats - ops.univ-batna2.dz, dernier accès : septembre 21, 2025, https://staff.univ-batna2.dz/sites/default/files/djaroudib-khamsa/files/chapitre3et4_01.pdf
48. Logique propositionnelle et Logique des Prédicats, dernier accès : septembre 21, 2025, https://perso.u-cergy.fr/~ibriquel/documents/l1Logique.poly.cours.pdf
49. LOGIQUE PROPOSITIONNELLE ET PRÉDICATS Les objectifs visés sont de, dernier accès : septembre 21, 2025, http://www.iro.umontreal.ca/~boyer/Archives/Automne07/Aut07/cours/semaine1.pdf
50. Calcul des prédicats - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Calcul_des_pr%C3%A9dicats
51. 4. Logique du premier ordre - [Verimag], dernier accès : septembre 21, 2025, http://www-verimag.imag.fr/~wack/cours_inf242/node8.html
52. logique des prédicats - CNRS, dernier accès : septembre 21, 2025, https://perso.liris.cnrs.fr/alain.mille/enseignements/Master_PRO/BIA/logique_predicats.pdf
53. La Logique Des Prédicats: Notes de Cours | PDF - Scribd, dernier accès : septembre 21, 2025, https://fr.scribd.com/document/535316612/1545688639120
54. Logique du premier ordre Logique temporelle et vérification, dernier accès : septembre 21, 2025, https://webusers.imj-prg.fr/~richard.lassaigne/coursLogique/logique-1erordre.pdf
55. Mathématiques pour l'informatique - CHAPITRE 2. Logique des prédicats, théorie des ensembles, dernier accès : septembre 21, 2025, https://zanotti.univ-tln.fr/MD/MD-Ensembles.html
56. logique du premier ordre: langages, modèles, preuves - LIRMM, dernier accès : septembre 21, 2025, https://www.lirmm.fr/~retore/GeoLog/mementoFOL.pdf
57. Logique des propositions et logique des prédicats, dernier accès : septembre 21, 2025, https://www.iro.umontreal.ca/~nie/IFT3335/logique.pdf
58. 3. Déduction Naturelle, dernier accès : septembre 21, 2025, http://www-verimag.imag.fr/~wack/cours_inf242/node6.html
59. Déduction naturelle - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/D%C3%A9duction_naturelle
60. Déduction naturelle - CELENE, dernier accès : septembre 21, 2025, https://celene.univ-orleans.fr/pluginfile.php/1565524/mod_folder/content/0/memo_d%C3%A9duction_naturelle.pdf?forcedownload=1
61. Systèmes de déduction, dernier accès : septembre 21, 2025, https://webusers.imj-prg.fr/~richard.lassaigne/coursLogique/syst-deduction.pdf
62. Logique intuitionniste - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Logique_intuitionniste
63. 2. Résolution propositionnelle - [Verimag], dernier accès : septembre 21, 2025, http://www-verimag.imag.fr/~wack/cours_inf242/node5.html
64. Principe de résolution en logique des propostions - CNRS, dernier accès : septembre 21, 2025, https://perso.liris.cnrs.fr/alain.mille/enseignements/emiage/supports-IA/logique/principe_resolution_logique_prop.pdf
65. 7-Logique propositionnelle - III la méthode de résolution, dernier accès : septembre 21, 2025, http://lecomte.al.free.fr/ressources/M1_IHS/res1.pdf
66. Fondements logiques pour l'informatique - 4MMFLI - Grenoble INP - Ensimag, UGA, dernier accès :

septembre 21, 2025, https://ensimag.grenoble-inp.fr/fr/formation/fondements-logiques-pour-l-informatique-4mmfli
67. Chapitre 3 Manipuler les formules de la logique, dernier accès : septembre 21, 2025, https://www.lri.fr/~paulin/Logique/html2020/cours005.html
68. Démontrer une implication ou une équivalence, dernier accès : septembre 21, 2025, https://www.editions-ellipses.fr/PDF/9782340026933_extrait.pdf
69. Chapitre 4 Quelques types de raisonnement, dernier accès : septembre 21, 2025, https://perso.univ-rennes1.fr/marie-pierre.lebaud/a04/pdf/raisonnement.pdf
70. Aujourd'hui nous allons discuter : • Exemple de preuve par contradiction., dernier accès : septembre 21, 2025, https://dms.umontreal.ca/~broera/MAT1500Slides_190925.pdf
71. Proposition contraposée - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Proposition_contrapos%C3%A9e
72. La démonstration par l'absurde : définition et applications - Progresser-en-maths, dernier accès : septembre 21, 2025, https://progresser-en-maths.com/la-demonstration-par-labsurde-definition-et-applications/
73. 10. Principe des tiroirs de Dirichlet, ou le principe des nids de pigeon - Département de mathématiques et statistique - Université de Montréal, dernier accès : septembre 21, 2025, https://dms.umontreal.ca/~broera/MAT1500_pp93-137
74. Principe des tiroirs - Mathraining, dernier accès : septembre 21, 2025, https://www.mathraining.be/chapters/6/all
75. Pigeonhole principle - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Pigeonhole_principle
76. fr.scribd.com, dernier accès : septembre 21, 2025, https://fr.scribd.com/document/855886065/Principe-Des-Tirroirs#:~:text=Principe%20Des%20Tirroirs-,Le%20principe%20des%20tiroirs%2C%20%C3%A9galement%20connu%20sous%20le%20nom%20de,%C3%A9galement%20dans%20des%20contextes%20infinis.
77. The Pigeonhole Principle | Baeldung on Computer Science, dernier accès : septembre 21, 2025, https://www.baeldung.com/cs/pigeonhole-principle
78. Principe Des Tirroirs | PDF | Concepts mathématiques - Scribd, dernier accès : septembre 21, 2025, https://fr.scribd.com/document/855886065/Principe-Des-Tirroirs
79. Principe des tiroirs, dernier accès : septembre 21, 2025, https://maths-olympiques.fr/wp-content/uploads/2025/02/Principe_des_tiroirs___groupe_A___visio.pdf
80. Principe des tiroirs: Maths, Logique - StudySmarter, dernier accès : septembre 21, 2025, https://www.studysmarter.fr/resumes/mathematiques/mathematiques-discretes/principe-des-tiroirs/
81. Principe des tiroirs - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Principe_des_tiroirs
82. Cours n° 9 : les tables de hachage, dernier accès : septembre 21, 2025, https://pageperso.lis-lab.fr/christophe.gonzales/teaching/algo/polys/cours9_poly.pdf
83. Tables d'association, tables de hachage - IGM, dernier accès : septembre 21, 2025, http://igm.univ-mlv.fr/~pivoteau/STRUCT/COURS/struct_cours8_1x4.pdf
84. Le hachage - Interstices.info, dernier accès : septembre 21, 2025, https://interstices.info/le-hachage/
85. Pigeonhole Principle - GeeksforGeeks, dernier accès : septembre 21, 2025, https://www.geeksforgeeks.org/engineering-mathematics/discrete-mathematics-the-pigeonhole-principle/
86. defeo.lu, dernier accès : septembre 21, 2025, http://defeo.lu/in310/poly/induction/#:~:text=Une%20preuve%20par%20induction%20(on,aussi%20pour%20le%20cas%20suivant.

87. Définitions récursives et induction structurelle - Introduction à la théorie de l'informatique, dernier accès : septembre 21, 2025, https://people.montefiore.uliege.be/geurts/Cours/iti/2012/03-inductionstructurelle-2012-2013.pdf
88. Induction structurelle - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Induction_structurelle
89. 3.6: Mathematical Induction - An Introduction, dernier accès : septembre 21, 2025, https://math.libretexts.org/Courses/Monroe_Community_College/MTH_220_Discrete_Math/3%3A_Proof_Techniques/3.6%3A_Mathematical_Induction_-_An_Introduction
90. Induction et récursion – IN310 - Luca De Feo, dernier accès : septembre 21, 2025, http://defeo.lu/in310/poly/induction/
91. Induction mathématique (GP) - L'encylopédie philosophique, dernier accès : septembre 21, 2025, https://encyclo-philo.fr/item/63
92. 4. Induction mathématique Les propriétés considérées essentielles de l'ensemble des nombres naturels N = {0 ,1,2,3,...,n,, dernier accès : septembre 21, 2025, https://dms.umontreal.ca/~broera/MAT1500H16_Notes3.pdf
93. Induction, dernier accès : septembre 21, 2025, https://people.montefiore.uliege.be/geurts/Cours/iti/2011/chap2-2011-2012.pdf
94. Preuves par induction structurelle - Anthony Lick, dernier accès : septembre 21, 2025, https://anthonylick.com/wp-content/uploads/mp2i_chap18.pdf
95. Fondements de l'informatique Logique, modèles, et calculs, dernier accès : septembre 21, 2025, https://www.enseignement.polytechnique.fr/informatique/INF412/uploads/Main/chap2-goodINF412.pdf
96. Chapitre 2 Ensembles définis inductivement et Induction structurelle - Informatique, dernier accès : septembre 21, 2025, http://informatique.cnam.fr/fr/spip.php?pdoc1052
97. L3 Info Cours 3 : preuve d'algorithmes - [Verimag], dernier accès : septembre 21, 2025, http://www-verimag.imag.fr/~wack/ALGO5/Cours03.pdf
98. Introduction à l'algorithmique et la complexité (et un peu de CAML) Prouvons que nos algorithmes sont corrects, dernier accès : septembre 21, 2025, http://www-sop.inria.fr/members/Nicolas.Nisse/lectures/prepa/2_correction.pdf
99. Outils d'analyse, dernier accès : septembre 21, 2025, https://people.montefiore.uliege.be/geurts/Cours/PA/2014/02-outils-2014-2015.pdf
100. Algorithmique IV - Preuve & complexité [JP. Zanotti], dernier accès : septembre 21, 2025, https://zanotti.univ-tln.fr/ALGO/II/Complexite.html
101. Utiliser les invariants pour corriger un algorithme - myMaxicours, dernier accès : septembre 21, 2025, https://www.maxicours.com/se/cours/utiliser-les-invariants-pour-corriger-un-algorithme/
102. Algorithme récursif - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Algorithme_r%C3%A9cursif
103. Tout apprendre sur les systèmes de gestion de bases de données, dernier accès : septembre 21, 2025, https://sgbd.developpez.com/tutoriels/cours-complet-bases-de-donnees/?page=logique-et-bases-de-donnees
104. (PDF) La logique et l'Intelligence Artificielle : Raisonnements et Algorithmes - ResearchGate, dernier accès : septembre 21, 2025, https://www.researchgate.net/publication/301796310_La_logique_et_l'Intelligence_Artificielle_Raisonnements_et_Algorithmes
105. Communication et connaissance - Intelligence artificielle et bases de données - CNRS Éditions, dernier accès : septembre 21, 2025, https://books.openedition.org/editionscnrs/30793

Chapitre 1a : Logique Déductive à la Logique Inductive - Formalisation de la Théorie des Probabilités

Abstract

Le Chapitre 1 a établi les fondements de la logique déductive, le calcul du raisonnement certain. Cependant, de nombreux domaines de l'informatique, notamment l'intelligence artificielle et la science des données, reposent sur le raisonnement inductif, ou raisonnement plausible en présence d'incertitude. Ce chapitre introduit la **logique inductive formelle**, un système rigoureux qui généralise la logique déductive et unifie la logique formelle avec la théorie moderne des probabilités (théorie de la msure). Nous explorons la construction de cette logique à travers sa dualité syntaxe-sémantique. Du côté syntaxique, nous définissons un **calcul inductif** basé sur des règles d'inférence permettant de manipuler des énoncés inductifs de la forme (X,ϕ,p). Du côté sémantique, nous introduisons les **modèles inductifs**, définis comme des espaces de probabilité sur des ensembles de structures logiques. Ce chapitre aborde le théorème de complétude inductive, qui garantit l'lignement entre le calcul et la sémantique, et présente une formalisation rigoureuse du **Principe d'Indifférence**, démontrant la puissance expressive de ce cadre unifié.

1A.1. Introduction au Raisonnement Inductif

1A.1.1. Dualité : Raisonnement Déductif vs Inductif (Plausibilité)

La logique, dans son acception la plus classique, est l'art du raisonnement certain. Le raisonnement déductif, tel qu'il est formalisé en mathématiques et en logique formelle, opère sur un principe de garantie : si les prémisses d'un argument sont vraies, alors la conclusion est nécessairement vraie. Une preuve mathématique est l'archétype de ce processus ; elle établit une certitude absolue ou elle échoue complètement.[1] Cependant, la grande majorité des raisonnements humains et scientifiques se déploient en dehors de cette sphère de certitude. Les lois de la physique, les diagnostics médicaux, les verdicts juridiques ou les prévisions économiques ne sont pas des certitudes, mais des conjectures dotées d'un certain degré de plausibilité.[1] Chaque expérience qui confirme la théorie de la relativité la rend plus plausible, mais ne la prouve jamais de manière absolue. Ce type de raisonnement, qui vise à évaluer la plausibilité d'une conclusion à la lumière d'un ensemble de preuves, est le raisonnement inductif.[1]

L'objectif de la logique inductive est de formaliser les règles de ce "raisonnement plausible" avec la même rigueur que celle appliquée à la logique déductive. Il ne s'agit pas de concevoir une logique "plus faible", mais au contraire une logique *plus générale*. Dans le cadre que nous allons développer, la logique déductive apparaît comme un cas particulier de la logique inductive. Les certitudes de la déduction (vérité et fausseté absolues) correspondent aux cas limites de la plausibilité inductive, où les probabilités prennent les valeurs extrêmes de 1 et 0. Les règles de l'inférence déductive, comme nous le verrons, sont parfaitement encapsulées dans les règles de l'inférence inductive pour ces cas extrêmes. Le

passage de la déduction à l'induction ne constitue donc pas une perte de rigueur, mais une extension du domaine d'application du raisonnement formel à l'immense champ de l'incertitude.[1]

1A.1.2. La Nature de la Probabilité : Une Perspective Logique

Quelle est la nature fondamentale de la probabilité? La réponse conventionnelle, issue des travaux fondateurs d'Andrei Kolmogorov, identifie la probabilité à sa formalisation mathématique : la théorie de la mesure. Dans cette perspective, un probabiliste est un mathématicien qui étudie les espaces de probabilité, c'est-à-dire les structures satisfaisant les axiomes de Kolmogorov.[1] Cependant, cette vision, bien que puissante, est restrictive. Les espaces de probabilité ne sont pas les concepts probabilistes eux-mêmes, mais des modèles mathématiques utilisés pour les représenter.[1]

Ce chapitre adopte une perspective différente, d'inspiration logique, qui remonte à Leibniz et Boole : la probabilité *est* la logique du raisonnement inductif. Elle est un mode de raisonnement abstrait qui se manifeste à travers deux systèmes parallèles et équivalents : un calcul syntaxique d'inférence et un système sémantique d'interprétation. La théorie de la mesure de Kolmogorov trouve sa place naturelle au sein de ce système sémantique, mais ne l'épuise pas.[1]

Cette perspective logique permet de recadrer le débat philosophique sur les "interprétations" de la probabilité. Le cadre formel présenté ici est agnostique quant à la source ultime de la plausibilité (qu'elle soit interprétée comme un degré de croyance subjective, une relation logique objective, ou une mesure de l'évidence). Cependant, il pose une contrainte fondamentale : les probabilités sont des relations entre des ensembles d'énoncés (les prémisses ou l'évidence) et un énoncé cible (la conclusion ou l'hypothèse). Cette nature relationnelle et propositionnelle rend le cadre incompatible avec les interprétations purement physiques de la probabilité, telles que les approches fréquentistes ou propensionnistes, qui voient la probabilité comme une propriété intrinsèque du monde physique. En revanche, il offre un méta-cadre unificateur pour toutes les interprétations qui conçoivent la probabilité comme une mesure de l'implication partielle ou du soutien évidentiel.[1]

1A.1.3. Structure de la Logique Inductive : Syntaxe et Sémantique

À l'instar de la logique du premier ordre, la logique inductive est construite sur une dualité fondamentale entre la syntaxe et la sémantique.[1] Cette structure duale garantit que le système n'est pas un simple ensemble de conventions arbitraires, mais qu'il capture fidèlement une réalité conceptuelle.

1. La Syntaxe : Le Calcul et la Dérivabilité ($\vdash$)
   Le versant syntaxique est un calcul formel. Il se compose d'un langage permettant de formuler des énoncés et d'un ensemble de règles d'inférence qui dictent comment manipuler ces énoncés pour en dériver de nouveaux. L'objet central de ce calcul est la "preuve" ou la "dérivation". L'expression $Q \vdash (X,\phi,p)$ signifie qu'il existe une dérivation formelle de l'énoncé inductif (X,ϕ,p) à partir de l'ensemble d'hypothèses Q, en utilisant uniquement les règles du calcul.1
2. La Sémantique : Les Modèles et la Conséquence ($\models$)

Le versant sémantique concerne l'interprétation et la signification. Il définit ce que les énoncés sont censés représenter. En logique déductive, le concept sémantique central est la "structure" (ou "modèle"), un univers mathématique dans lequel les énoncés sont soit vrais, soit faux. La logique inductive généralise cette notion : un "modèle inductif" est un espace de probabilité défini sur un ensemble de structures déductives. La relation de conséquence, notée $Q \models (X, \phi, p)$, signifie que dans tout modèle inductif où les hypothèses de Q sont satisfaites, l'énoncé (X, ϕ, p) l'est également.1

L'alignement parfait entre ces deux versants est garanti par un **théorème de complétude**, qui stipule que $Q \vdash (X, \phi, p)$ si et seulement si $Q \models (X, \phi, p)$. Ce théorème est la pierre angulaire de la logique inductive, car il prouve que le calcul syntaxique que nous allons définir capture de manière exacte et complète la sémantique de la théorie des probabilités moderne.[1]

1A.2. Préliminaires Logiques et Mathématiques

1A.2.1. Introduction aux Langages Infinitaires ($L_{\omega_1,\omega}$)

La logique propositionnelle et la logique des prédicats classiques sont des langages *finitaires* : toute formule est de longueur finie et ne peut contenir que des conjonctions ($\wedge$) et des disjonctions ($\vee$) portant sur un nombre fini de sous-formules. Cette restriction constitue un obstacle fondamental à l'unification de la logique et de la théorie des probabilités. La théorie des probabilités, dans sa formulation moderne par Kolmogorov, repose sur l'axiome de σ-additivité, qui concerne des unions *dénombrables* d'événements. Pour capturer cette propriété au niveau syntaxique, il est nécessaire d'étendre le langage logique.

Nous utilisons pour cela le langage infinitaire $L_{\omega_1,\omega}$. Ce langage est une extension de la logique du premier ordre qui autorise la formation de conjonctions et de disjonctions sur des ensembles dénombrables (finis ou infinis) de formules.[1] Formellement, si

$\Phi = \{\phi_n\}_{n \in N}$ est un ensemble dénombrable de formules bien formées, alors $\bigwedge_{n \in N} \phi_n$ (la conjonction de toutes les ϕ_n) et $\bigvee_{n \in N} \phi_n$ (leur disjonction) sont également des formules bien formées.[1] L'indice

ω_1 fait référence à la cardinalité des conjonctions/disjonctions autorisées (inférieures à ω_1, c'est-à-dire au plus dénombrables), tandis que l'indice ω indique que les séquences de quantificateurs restent finies.[2]

Ce choix n'est pas une simple commodité technique ; il est la clé de voûte de l'édifice. Le langage $L_{\omega_1,\omega}$ fournit l'isomorphe syntaxique de la σ-algèbre en théorie de la mesure. De la même manière qu'une σ-algèbre est close sous les unions dénombrables, l'ensemble des formules de $L_{\omega_1,\omega}$ est clos sous les disjonctions dénombrables. C'est cette correspondance structurelle qui permet d'établir une sémantique probabiliste complète pour la logique et de prouver le théorème de complétude.[1]

1A.2.2. Rappels sur la Théorie de la Mesure et les σ-Algèbres

La sémantique de la logique inductive repose sur les fondements de la théorie moderne des probabilités. Il est donc essentiel de rappeler les définitions de base de la théorie de la mesure.[1]

Un **espace mesurable** est un couple (Ω,Σ) où Ω est un ensemble non vide (appelé l'univers ou l'espace des échantillons) et Σ est une **σ-algèbre** (ou tribu) sur Ω. Une σ-algèbre est une collection de sous-ensembles de Ω qui contient l'ensemble vide, est close par complémentation et est close par union dénombrable. Les éléments de Σ sont appelés les ensembles mesurables (ou événements).[1]

Une **mesure** sur (Ω,Σ) est une fonction $\mu:\Sigma\rightarrow[0,\infty]$ qui assigne une "taille" non négative à chaque ensemble mesurable, telle que $\mu(\emptyset)=0$ et qui satisfait la propriété de σ-additivité : pour toute suite $\{A_n\}_{n=1}^{\infty}$ d'ensembles mesurables deux à deux disjoints, $\mu(\bigcup_{n=1}^{\infty}A_n)=\sum_{n=1}^{\infty}\mu(A_n)$. Un **espace de mesure** est un triplet (Ω,Σ,μ).[1]

Un **espace de probabilité** est un espace de mesure (Ω,Σ,P) où la mesure P est une mesure de probabilité, c'est-à-dire qu'elle satisfait $P(\Omega)=1$. Dans ce contexte, les éléments de Ω sont appelés les issues, et les éléments de Σ sont les événements.[1] Enfin, un espace de mesure est dit **complet** si tout sous-ensemble d'un ensemble de mesure nulle est lui-même mesurable. Tout espace de mesure peut être étendu en un espace complet via une procédure de **complétion**.[1]

1A.2.3. Structures et Théorie des Modèles

La sémantique de la logique déductive classique est définie par rapport à des objets mathématiques appelés structures. Une **signature extra-logique** L est un ensemble de symboles de constantes, de fonctions et de relations, chacun doté d'une arité spécifique.[1]

Une **L-structure** ω est un couple (A,L^{ω}) où A est un ensemble non vide appelé le **domaine** (ou l'univers) de la structure, et L^{ω} est un ensemble d'interprétations pour les symboles de L sur ce domaine. Chaque symbole de constante $c\in L$ est associé à un élément $c^{\omega}\in A$, chaque symbole de fonction n-aire $f\in L$ est associé à une fonction $f^{\omega}:A^n\rightarrow A$, et chaque symbole de relation n-aire $r\in L$ est associé à une relation $r^{\omega}\subseteq A^n$.[1]

Par exemple, l'arithmétique standard peut être vue comme une structure $\mathcal{N}=(\mathbb{N}_0,\{0,S,+,\cdot,<\})$ où le domaine est l'ensemble des entiers naturels et les symboles sont interprétés de manière usuelle.[1] Les structures sont les mondes possibles dans lesquels les formules logiques peuvent être évaluées comme vraies ou fausses. Le concept central de la sémantique inductive, que nous aborderons en section A.4, sera de ne plus considérer une unique structure, mais un espace de probabilité défini sur un ensemble de telles structures. Ce passage d'une structure unique à une distribution de probabilité sur les structures est la généralisation fondamentale qui connecte la logique à la probabilité.

1A.3. Le Calcul Inductif : Syntaxe et Inférence

1A.3.1. Énoncés Inductifs : Le Triplet **(X,ϕ,p)**

Le calcul déductif manipule des formules (ou énoncés). Le calcul inductif, quant à lui, manipule des objets plus complexes qui capturent la notion de plausibilité conditionnelle. L'objet fondamental de notre calcul est l'**énoncé inductif**, un triplet ordonné (X,ϕ,p).[1]

- X est un ensemble de formules du langage Lω1,ω, appelé l'**antécédent**. Il représente l'ensemble des prémisses, des connaissances de base ou de l'évidence.
- ϕ est une formule unique du langage Lω1,ω, appelée le **conséquent**. Elle représente l'hypothèse ou la conclusion dont on évalue la plausibilité.
- p est un nombre réel dans l'intervalle $$, appelé la **probabilité**. Il représente le degré de soutien que l'antécédent X confère au conséquent ϕ.

Intuitivement, l'énoncé (X,ϕ,p) asserte que "la probabilité de ϕ, étant donné X, est p". Pour alléger la notation, lorsque nous manipulons des ensembles d'énoncés inductifs, nous adoptons une notation fonctionnelle. Un ensemble P d'énoncés inductifs est traité comme une fonction partielle, et nous écrivons P(ϕ|X)=p pour signifier que le triplet (X,ϕ,p) appartient à P.[1]

1A.3.2. Les Règles de l'Inférence Inductive (R1-R9)

Le cœur du calcul inductif est un ensemble de neuf règles d'inférence, notées de (R1) à (R9). Ces règles définissent comment de nouveaux énoncés inductifs peuvent être légitimement dérivés à partir d'un ensemble existant. Elles constituent l'équivalent, pour le raisonnement plausible, des règles comme le *modus ponens* pour le raisonnement déductif. Ces règles peuvent être regroupées en trois catégories fonctionnelles, comme résumé dans le tableau A.1.[1]

Règle	Nom	Formulation Concise (Informelle)	Rôle
R1	Équivalence Logique	Si X≡X’ et ϕ≡Xϕ’, alors $P(\varphi	X) = P(\varphi'
R2	Implication Logique	Si X⊢ϕ, alors	X) = 1$.

		$P(\varphi	
R3	Implication Matérielle	Si $P(\psi	X, \varphi) = 1$, alors $P(\varphi \to \psi
R4	Transitivité Déductive	Si $P(\varphi	X) = 1$ et ϕ⊢ψ, alors $P(\psi
R5	Règle d'Addition	Si X⊢¬(ϕ∧ψ), alors $P(\varphi \vee \psi	X) = P(\varphi
R6	Règle de Multiplication	$P(\varphi \wedge \psi	X) = P(\varphi
R7	Règle de Continuité	Pour une suite croissante d'événements ϕn, $P(\bigvee_n \varphi_n	X) = \lim_n P(\varphi_n
R8	Extension Inductive	Si une valeur de probabilité est déterminée de manière unique par les règles R1-R7, elle est inférée.	Règle de clôture qui empêche de laisser indéterminées des probabilités qui sont logiquement contraintes.
R9	Extension Déductive	On peut ajouter des énoncés de probabilité 1 à un antécédent sans changer les autres probabilités.	Permet de simplifier les contextes en intégrant les certitudes acquises.
Tableau A.1 : Les Règles de l'Inférence Inductive (R1-R9).			

1A.3.2.1. Connexion avec l'Inférence Déductive (R1-R4)

Les quatre premières règles garantissent que la logique inductive est une extension propre et cohérente de la logique déductive.[1] La règle

(R1), la règle d'équivalence logique, stipule que si deux antécédents X et X’ sont logiquement équivalents, et si deux conséquents ϕ et ϕ’ sont logiquement équivalents *dans le contexte de* X, alors la probabilité doit être la même. Cela assure que la probabilité est attachée au contenu sémantique des propositions, pas à leur formulation accidentelle. La règle **(R2)**, la règle de l'implication logique, est le pont le plus direct entre les deux logiques : tout ce qui est déductivement prouvable à partir de X doit se voir assigner une probabilité de 1 étant donné X. Les règles **(R3)** et **(R4)** renforcent cette connexion en assurant que la certitude (probabilité 1) se comporte de manière attendue par rapport à l'implication et à la transitivité déductive.[1]

1A.3.2.2. Règles Probabilistes Fondamentales : Addition, Multiplication, Continuité (R5-R7)

Ces trois règles sont les analogues syntaxiques des axiomes fondamentaux de la probabilité de Kolmogorov.[1] La règle

(R5), la règle d'addition, s'applique à des conséquents mutuellement exclusifs (prouvablement incompatibles étant donné l'antécédent) et stipule que la probabilité de leur disjonction est la somme de leurs probabilités. La règle **(R6)**, la règle de multiplication, est la définition formelle de la probabilité conditionnelle, reliant la probabilité d'une conjonction aux probabilités conditionnelles et marginales. Enfin, la règle **(R7)**, la règle de continuité, est l'innovation la plus significative, rendue possible par le langage infinitaire Lω1,ω. Elle stipule que pour une suite croissante d'énoncés (où chaque énoncé implique le précédent), la probabilité de leur disjonction infinie est la limite de leurs probabilités. C'est l'expression syntaxique directe de la σ-additivité des mesures de probabilité.[1]

1A.3.2.3. Règles d'Extension (R8-R9)

Les règles (R8) et (R9) sont des règles de clôture plus subtiles, qui garantissent que le système de déduction est complet et bien élevé.[1] La règle

(R8), la règle d'extension inductive, est une sorte de principe de raison suffisante. Si, à partir d'un ensemble d'énoncés, il n'existe qu'une seule valeur possible pour une probabilité non encore spécifiée qui soit compatible avec les règles (R1)-(R7), alors cette valeur doit être inférée. Cela empêche le système de rester silencieux sur des probabilités qui sont en fait logiquement déterminées. La règle **(R9)**, la règle d'extension déductive, stipule que si certains énoncés ont une probabilité de 1 étant donné X, on peut les ajouter à l'antécédent X sans affecter les autres probabilités. Cela permet de

simplifier le raisonnement en absorbant les certitudes acquises dans le contexte.[1]

1A.3.3. Théories Inductives, Connectivité et Cohérence

En logique déductive, une théorie est simplement un ensemble d'énoncés clos sous la relation de dérivabilité (⊢). En logique inductive, cette définition doit être affinée. Une **théorie inductive** est un ensemble d'énoncés inductifs qui est non seulement *clos* sous les neuf règles d'inférence, mais qui est également *connecté*.[1]

La notion de **connectivité** est une caractéristique essentielle et nouvelle de la logique inductive. Elle est introduite pour garantir la cohérence globale d'un cadre de raisonnement. L'ensemble des énoncés inductifs est si vaste qu'il est possible de construire des ensembles d'énoncés qui sont clos sous les règles (R1)-(R9) mais qui contiennent des "îlots" de raisonnement sans aucun lien logique entre eux. Par exemple, un tel ensemble pourrait contenir des énoncés sur la physique des particules et d'autres sur les résultats d'élections, sans qu'aucun chemin d'inférence ne relie les deux domaines. La connectivité exige qu'il existe un antécédent de base (la "racine" de la théorie) à partir duquel tous les autres antécédents de la théorie peuvent être construits par l'ajout d'un nombre dénombrable d'axiomes. Cette condition garantit que tous les énoncés au sein d'une même théorie inductive appartiennent à un seul et même "fil de discussion" logique et cohérent.[1]

1A.3.4. Dérivabilité Inductive (⊢)

Avec la définition d'une théorie inductive en place, nous pouvons maintenant définir formellement la **dérivabilité inductive**. Soit Q un ensemble d'énoncés inductifs servant d'hypothèses. Nous disons que l'énoncé (X,ϕ,p) est **dérivable** de Q, et nous notons Q⊢(X,ϕ,p), si et seulement si (X,ϕ,p) appartient à la plus petite théorie inductive qui contient Q.[1]

Cette définition est l'analogue direct de la définition de la dérivabilité en logique déductive, où X⊢ϕ signifie que ϕ appartient à la plus petite théorie déductive contenant X. Le processus de dérivation inductive consiste donc à générer la clôture d'un ensemble d'hypothèses initiales sous les neuf règles d'inférence, tout en s'assurant que le résultat final forme un tout connecté et cohérent.

1A.4. Sémantique de la Logique Inductive : Les Modèles

1A.4.1. Définition d'un Modèle Inductif (Espace Probabilisé de Structures)

La sémantique fournit une interprétation, une signification, aux formules syntaxiques. En logique déductive, un modèle est une unique structure mathématique qui représente un "état du monde" possible. Dans ce monde, chaque énoncé est soit vrai, soit faux. Pour donner un sens à des énoncés de probabilité, nous devons généraliser cette notion pour représenter l'incertitude sur l'état du monde.

Un **modèle inductif** P est formellement défini comme un espace de probabilité (Ω,Σ,P) où l'espace des issues Ω n'est pas un ensemble abstrait, mais un ensemble de L-structures, c'est-à-dire un ensemble de mondes possibles.[1]

- Ω est un ensemble de L-structures. Chaque structure $\omega\in\Omega$ est un modèle déductif classique, avec un domaine et une interprétation pour chaque symbole de la signature L.
- Σ est une σ-algèbre sur Ω.
- P est une mesure de probabilité sur (Ω,Σ).

Un modèle inductif peut donc être vu comme une collection de mondes possibles, pondérée par une distribution de probabilité qui représente notre degré de croyance ou la vraisemblance relative de chaque monde.

1A.4.2. Relation de Conséquence Inductive ($\models$)

À partir de la définition d'un modèle inductif, nous pouvons définir la notion de satisfaction, qui relie la sémantique à la syntaxe. Pour un énoncé déductif ϕ, nous disons qu'une structure ω satisfait ϕ, noté $\omega\models\phi$, si ϕ est vrai dans ω. Pour un énoncé inductif (X,ϕ,p), la satisfaction par un modèle inductif $P=(\Omega,\Sigma,P)$, notée $P\models(X,\phi,p)$, est définie de manière plus complexe.

Intuitivement, $P\models(X,\phi,p)$ signifie que la probabilité conditionnelle de l'ensemble des mondes où ϕ est vrai, étant donné l'ensemble des mondes où X est vrai, est égale à p. Formellement, cela est défini en utilisant une version de la formule de la probabilité conditionnelle sur les sous-ensembles de Ω définis par les formules. Soit $\phi\Omega=\{\omega\in\Omega|\omega\models\phi\}$ l'ensemble des structures dans Ω qui satisfont ϕ. Alors $P\models(X,\phi,p)$ s'il existe une décomposition de l'antécédent X en un ensemble de certitudes Y (telles que $P(Y\Omega)=1$) et une hypothèse contingente ψ, de sorte que $X\equiv Y\cup\{\psi\}$ et $P(\phi\Omega\cap\psi\Omega)/P(\psi\Omega)=p$.[1]

La **relation de conséquence inductive** est alors définie naturellement : $Q\models(X,\phi,p)$ signifie que pour tout modèle inductif P, si P satisfait tous les énoncés de l'ensemble d'hypothèses Q, alors P doit également satisfaire (X,ϕ,p).[1]

1A.4.3. Théorème de Complétude : Équivalence entre $\vdash$ et $\models$

Le résultat central de la logique inductive, qui en assure la cohérence et la puissance, est le **théorème de complétude**. Il établit une équivalence parfaite entre le calcul syntaxique et la sémantique probabiliste.[1]

Théorème (Complétude Inductive) : Soit Q un ensemble d'énoncés inductifs et (X,ϕ,p) un énoncé inductif. Alors :

Q⊢(X,ϕ,p)si et seulement siQ⊨(X,ϕ,p)
Ce théorème est d'une importance capitale. Il démontre que les neuf règles d'inférence (le système ⊢) ne sont pas un ensemble arbitraire de règles, mais qu'elles capturent de manière *exacte et complète* la notion de conséquence logique dans les espaces de probabilité de structures (le système ⊨). Cela signifie que tout ce qui est sémantiquement valide est syntaxiquement prouvable, et vice-versa. Le raisonnement probabiliste peut donc être entièrement formalisé comme un système de déduction logique, unifiant ainsi les deux piliers du raisonnement formel et réalisant la vision de pionniers comme Leibniz et Boole.[1]

1A.4.4. Extension à la Logique des Prédicats

L'extension du cadre de la logique propositionnelle à la logique des prédicats (qui inclut les quantificateurs ∀,∃) enrichit considérablement son pouvoir expressif, notamment en permettant une modélisation naturelle des variables aléatoires et en révélant une propriété profonde de l'aléatoire.

1A.4.4.1. Variables Aléatoires comme Symboles Extra-logiques

En théorie des probabilités, une variable aléatoire est une fonction qui associe une valeur numérique à chaque issue d'un espace d'échantillonnage. Dans le cadre de la logique inductive, ce concept est capturé de manière élégante et directe. Un symbole extra-logique s (une constante, une fonction ou une relation) n'a pas une interprétation fixe et unique, mais une interprétation sω qui dépend de la structure (le "monde") ω∈Ω considérée.

L'application ω↦sω est donc une fonction définie sur l'espace des structures Ω. C'est l'analogue formel exact d'une variable aléatoire. Par exemple, si c est un symbole de constante et que le domaine des structures est l'ensemble des nombres réels, l'application ω↦cω est une variable aléatoire à valeur réelle.[1]

1A.4.4.2. La Relativité de l'Aléatoire et les Cadres de Référence

L'extension à la logique des prédicats révèle une subtilité conceptuelle majeure. Considérons un énoncé probabiliste comme P{X>0}. En théorie des probabilités classique, il est implicite que X est une variable aléatoire, tandis que la relation > et la constante 0 sont des entités mathématiques fixes et non aléatoires. Dans le formalisme de la logique inductive, l'énoncé correspondant s'interprète dans un modèle inductif comme la probabilité de l'ensemble des structures ω où Xω>ω0ω.

Dans ce cadre, il n'y a *a priori* aucune raison pour que l'interprétation de la relation > ou de la constante 0 soit la même

dans toutes les structures ω. Elles peuvent elles-mêmes être "aléatoires". Ce qui est considéré comme fixe (déterministe) et ce qui est considéré comme variable (aléatoire) n'est pas une propriété intrinsèque du monde, mais dépend d'un **cadre de référence** choisi par le modélisateur.[1] Un cadre de référence est un ensemble de contraintes qui forcent certains symboles à avoir la même interprétation dans toutes les structures considérées.

Cette "relativité de l'aléatoire" a des implications profondes. Elle suggère que l'aléatoire n'est pas une propriété ontologique, mais une propriété épistémique relative à un cadre de connaissance. Pour l'intelligence artificielle, cela signifie qu'un agent rationnel doit explicitement définir son cadre de référence — quelles sont les constantes de son univers et quelles sont les variables? — pour pouvoir raisonner de manière cohérente sur l'incertitude.[1]

1A.5. Théories Avancées et Principe d'Indifférence

1A.5.1. Théories Inductives Réelles et Intégration dans ZFC

La puissance expressive de la logique inductive est telle qu'elle peut englober l'ensemble des mathématiques modernes. En adoptant une théorie axiomatique des ensembles, telle que la théorie de Zermelo-Fraenkel avec l'axiome du choix (ZFC), comme partie de l'antécédent d'une théorie inductive, il devient possible de construire formellement les nombres réels, les espaces fonctionnels, et toute autre structure mathématique. Le cadre permet alors de formuler des énoncés inductifs (probabilistes) rigoureux sur n'importe quel objet mathématique, des nombres réels aux fonctions et aux espaces topologiques.[1]

1A.5.2. Reformulation des Théorèmes Limites (LGN, TCL)

Pour illustrer cette puissance, les théorèmes fondamentaux de la théorie des probabilités peuvent être reformulés et prouvés comme des théorèmes au sein de la logique inductive. La **Loi des Grands Nombres (LGN)**, qui stipule que la moyenne d'un grand nombre de variables aléatoires i.i.d. converge vers leur espérance, et le **Théorème Central Limite (TCL)**, qui décrit la convergence de cette moyenne vers une distribution normale, peuvent être énoncés comme des énoncés inductifs dérivés des axiomes de ZFC et des définitions appropriées de l'indépendance et de l'espérance.[1] Cela démontre que le cadre logique ne se contente pas de reproduire les axiomes de base, mais qu'il est suffisamment riche pour recapturer les résultats les plus profonds de la discipline.

1A.5.3. Le Principe d'Indifférence : Historique et Problématique

Le **Principe d'Indifférence**, dont l'origine remonte à Pierre-Simon Laplace, est une heuristique intuitivement très puissante pour l'assignation des probabilités a priori. Il stipule que si, en l'absence de toute information pertinente, nous n'avons aucune raison de privilégier une possibilité par rapport à une autre, nous devrions leur assigner des probabilités égales.[1] C'est le principe qui sous-tend l'assignation d'une probabilité de 1/2 à chaque face d'une pièce de monnaie présumée non biaisée.

Cependant, l'application naïve de ce principe a une histoire notoire de production de paradoxes. Comme l'a souligné John Maynard Keynes, des manières apparemment équivalentes de décrire un même problème peuvent conduire à des partitions différentes des possibilités, et l'application du principe à chaque partition peut donner des résultats contradictoires.[1] Le plus célèbre de ces paradoxes est celui de Bertrand, concernant la probabilité qu'une corde "choisie au hasard" dans un cercle soit plus longue que le côté du triangle équilatéral inscrit.

1A.5.4. Formulation Rigoureuse en Logique Inductive

La logique inductive offre, pour la première fois, une formulation mathématiquement rigoureuse du Principe d'Indifférence, résolvant ainsi ses ambiguïtés historiques. Cette formalisation est impossible dans le cadre standard de Kolmogorov car elle dépend de la structure syntaxique du langage logique.[1]

1A.5.4.1. Permutations de Signature et Invariance

L'idée clé est de capturer la notion d'"ignorance symétrique" par une invariance syntaxique. Une **permutation de signature** π est un renommage bijectif et cohérent des symboles extra-logiques de la signature L qui préserve leur type et leur arité (par exemple, un symbole de relation binaire est remplacé par un autre symbole de relation binaire).[1] Une telle permutation peut être appliquée à n'importe quelle formule

ϕ pour obtenir une nouvelle formule $\phi\pi$. Un ensemble de formules X est dit **invariant** sous π si X et $X\pi$ sont logiquement équivalents.[1]

1A.5.4.2. Indifférence Déductive et Inductive

Le raisonnement déductif obéit déjà à une forme d'indifférence : si $X \vdash \phi$, alors il est trivial de montrer que $X\pi \vdash \phi\pi$. La preuve est insensible au nom des symboles. Le Principe d'Indifférence est l'extension naturelle de cette propriété à

l'inférence inductive. Une théorie inductive P satisfait le principe si, pour toute permutation de signature π, on a :

$P(\phi|X)=P(\phi\pi|X\pi)$

En particulier, si la connaissance de base X est invariante sous π ($X\equiv X\pi$), alors le principe exige que $P(\phi|X)=P(\phi\pi|X)$. La "symétrie de l'ignorance" est ainsi formellement identifiée à l'invariance de la théorie de base sous une transformation syntaxique.1 Cette formulation est une avancée conceptuelle majeure, car elle démontre que la logique inductive est strictement plus expressive que la théorie des probabilités de Kolmogorov, qui ne dispose pas des outils syntaxiques nécessaires pour exprimer une telle condition d'invariance.1

1A.5.5. Application : Résolution du Paradoxe de Bertrand

La puissance de cette formulation rigoureuse est spectaculairement démontrée par son application au **paradoxe de Bertrand**. Le paradoxe naît de l'ambiguïté de l'expression "choisir une corde au hasard". Différentes méthodes de sélection (par les extrémités, par le milieu, etc.) conduisent à des probabilités différentes (1/3, 1/2, 1/4).[5]

La logique inductive résout le paradoxe en forçant le modélisateur à être explicite. Pour poser le problème, il faut définir une signature logique et une théorie de base T0 décrivant la géométrie du cercle. Chaque méthode de "choix au hasard" correspond à une hypothèse différente sur les symétries (les permutations de signature) sous lesquelles la théorie T0 est supposée être invariante. La logique inductive montre que les trois solutions classiques du paradoxe ne sont pas contradictoires ; elles sont les réponses correctes à trois problèmes distincts, chacun caractérisé par un groupe de symétrie différent. Le paradoxe se dissout : il n'y a pas de contradiction, seulement une spécification initiale incomplète du problème.[1]

1A.6. Implications pour l'Informatique et l'IA

1A.6.1. Fondations Logiques pour le Raisonnement Probabiliste en IA

L'intelligence artificielle est de plus en plus confrontée à la nécessité de raisonner et d'agir dans des environnements incertains. Pour ce faire, une multitude de formalismes probabilistes ont été développés, tels que les réseaux bayésiens, les modèles de Markov cachés, et les logiques probabilistes.[7] Bien qu'efficaces, ces outils sont souvent perçus comme des constructions ad-hoc, chacune avec sa propre sémantique et ses propres algorithmes.

La logique inductive, telle que présentée ici, offre une fondation unificatrice pour l'ensemble de ces approches.[1] Elle fournit un langage commun et une sémantique rigoureuse dans lesquels ces différents modèles peuvent être exprimés

et compris comme des théories inductives spécifiques. Un réseau bayésien, par exemple, peut être vu comme une spécification compacte d'une théorie inductive qui postule un ensemble d'indépendances conditionnelles. Cette perspective unifiée permet non seulement de clarifier les hypothèses sous-jacentes à chaque modèle, mais aussi de comparer leur pouvoir expressif et d'étudier leurs relations sur une base formelle solide.[10]

1A.6.2. Analyse des Méthodes Probabilistes

En plaçant le raisonnement probabiliste sur des fondations logiques, ce cadre ouvre la voie à l'analyse formelle et à la vérification des systèmes d'IA probabilistes. De la même manière que la logique formelle est utilisée pour vérifier la correction des logiciels et des circuits matériels, la logique inductive peut être utilisée pour analyser la cohérence et la robustesse des algorithmes d'apprentissage et de raisonnement.[12]

Par exemple, la **Programmation Logique Inductive (ILP)**, un sous-domaine de l'IA qui vise à apprendre des programmes logiques à partir d'exemples, peut être naturellement intégrée dans ce cadre.[15] L'apprentissage d'une règle en ILP peut être vu comme la recherche d'une hypothèse

ϕ qui, ajoutée à la connaissance de base B, maximise la probabilité des exemples positifs tout en minimisant celle des exemples négatifs. Le formalisme de la logique inductive permet d'analyser la validité de ces inférences et de garantir qu'un système d'IA ne tirera jamais de conclusions probabilistes incohérentes avec ses connaissances de base.

1A.6.3. Gestion de l'Indécidabilité et de l'Incertitude

Un des défis les plus profonds en IA est de construire des agents capables de raisonner avec des connaissances incomplètes, dans des mondes ouverts où tout n'est pas connu à l'avance. La logique déductive classique est fragile face à cette réalité : un énoncé indécidable (qui ne peut être ni prouvé ni réfuté à partir de la base de connaissances) paralyse le raisonnement. Le système ne peut rien conclure.

La logique inductive offre une solution élégante et puissante à ce problème.[1] Face à un énoncé indécidable

ψ, un agent n'est plus contraint au silence. Il peut utiliser la logique inductive pour postuler une probabilité pour cet énoncé, par exemple en posant $P(\psi|X)=0.5$ via le Principe d'Indifférence si aucune information ne favorise ψ ou $\neg\psi$. À partir de cette hypothèse, l'agent peut explorer de manière cohérente les conséquences probabilistes qui en découlent, raisonner sur ses actions possibles, et mettre à jour sa croyance en ψ à la lumière de nouvelles preuves via la règle de Bayes (qui est un théorème de la logique inductive).

Cette capacité à assigner des probabilités à des hypothèses indécidables et à raisonner rigoureusement à partir d'elles est une avancée fondamentale. Elle connecte la logique formelle à la théorie de la décision bayésienne et fournit une base normative pour la construction d'agents rationnels capables d'agir de manière intelligente dans des conditions d'incertitude radicale, ce qui est l'un des objectifs ultimes de l'intelligence artificielle.[9]

Ouvrages cités

1. Logique formelle et probabilité inductive.pdf
2. Infinitary logic - Wikipedia, dernier accès : septembre 22, 2025, https://en.wikipedia.org/wiki/Infinitary_logic

3. A Primer on Infinitary Logic, dernier accès : septembre 22, 2025, http://homepages.math.uic.edu/~marker/inf.pdf
4. [1304.5208] Omitting types for infinitary [0, 1]-valued logic - arXiv, dernier accès : septembre 22, 2025, https://arxiv.org/abs/1304.5208
5. Bertrand's Paradox and the Principle of Indifference | Philosophy of Science, dernier accès : septembre 22, 2025, https://www.cambridge.org/core/journals/philosophy-of-science/article/bertrands-paradox-and-the-principle-of-indifference/DC735A7B90AD19EB0572A5EA9C5B07BB
6. Problems for Probability: the Principle of Indifference - University of Pittsburgh, dernier accès : septembre 22, 2025, https://sites.pitt.edu/~jdnorton/teaching/paradox/chapters/probability_for_indifference/probability_for_indifference.html
7. Probabilistic logic - Wikipedia, dernier accès : septembre 22, 2025, https://en.wikipedia.org/wiki/Probabilistic_logic
8. Probabilistic Logics - (Formal Logic II) - Vocab, Definition, Explanations | Fiveable, dernier accès : septembre 22, 2025, https://fiveable.me/key-terms/formal-logic-ii/probabilistic-logics
9. Probabilistic Reasoning in Artificial Intelligence - GeeksforGeeks, dernier accès : septembre 22, 2025, https://www.geeksforgeeks.org/artificial-intelligence/probabilistic-reasoning-in-artificial-intelligence/
10. A Logic for Inductive Probabilistic Reasoning - Department of Computer Science - Aalborg University, dernier accès : septembre 22, 2025, https://homes.cs.aau.dk/~jaeger/publications/Synthese-inductive.pdf
11. Logic and Probability - Stanford Encyclopedia of Philosophy, dernier accès : septembre 22, 2025, https://plato.stanford.edu/entries/logic-probability/
12. medium.com, dernier accès : septembre 22, 2025, https://medium.com/@vaishnaviyada/understanding-deductive-vs-inductive-reasoning-in-artificial-intelligence-ebf8b65b8346#:~:text=Applications%20of%20Inductive%20Reasoning%20in,analyzing%20vast%20amounts%20of%20text.
13. Inductive Reasoning in AI - GeeksforGeeks, dernier accès : septembre 22, 2025, https://www.geeksforgeeks.org/artificial-intelligence/inductive-reasoning-in-ai/
14. Inductive Learning Algorithm - Applied AI Course, dernier accès : septembre 22, 2025, https://www.appliedaicourse.com/blog/inductive-learning-algorithm/
15. Inductive Logic Programming - Lirias, dernier accès : septembre 22, 2025, https://lirias.kuleuven.be/retrieve/145739
16. Inductive logic programming - Wikipedia, dernier accès : septembre 22, 2025, https://en.wikipedia.org/wiki/Inductive_logic_programming
17. What is Inductive Logic Programming | AI Basics - Ai Online Course, dernier accès : septembre 22, 2025, https://www.aionlinecourse.com/ai-basics/inductive-logic-programming
18. What is Probabilistic Reasoning in AI? - Artificial Intelligence Masterclass, dernier accès : septembre 22, 2025, https://www.aimasterclass.com/glossary/probabilistic-reasoning-in-ai
19. Reasoning with Uncertainty: Approaches to Probabilistic Knowledge Representation | by Neelam Mahraj | Artificial Intelligence in Plain English, dernier accès : septembre 22, 2025, https://ai.plainenglish.io/reasoning-with-uncertainty-approaches-to-probabilistic-knowledge-representation-e7d2190fd340

Chapitre 2 : Structures Discrètes et Combinatoire

Introduction

Les sciences et le génie informatiques reposent sur la manipulation d'informations et l'exécution de processus qui sont, par leur nature même, fondamentalement discrets. Contrairement aux mathématiques du continu, telles que l'analyse et le calcul infinitésimal qui décrivent des phénomènes fluides et ininterrompus, les mathématiques discrètes s'attachent à l'étude d'objets distincts et dénombrables. Un programme informatique est une séquence finie d'instructions ; les données sont stockées dans des structures composées d'un nombre fini d'éléments (bits, octets, enregistrements) ; les réseaux de communication sont des ensembles de nœuds interconnectés par des liens. Chacun de ces exemples illustre un système dont les composantes peuvent être comptées, énumérées et traitées comme des unités individuelles.

Ce chapitre se consacre à l'exploration des structures discrètes et de la combinatoire, qui constituent le langage formel et l'arsenal d'outils indispensables à la modélisation et à l'analyse des systèmes informatiques complexes.[1] Les structures discrètes, telles que les ensembles, les relations et les fonctions, fournissent le cadre conceptuel pour organiser l'information, tandis que la combinatoire, l'art du dénombrement, nous permet de quantifier les configurations possibles, d'évaluer la complexité des algorithmes et de fonder la théorie des probabilités sur des bases solides.

Notre parcours débutera par les fondements les plus élémentaires : la théorie des ensembles. Nous établirons comment ce langage, avec ses opérations et ses propriétés, correspond de manière profonde à la logique propositionnelle qui sous-tend tout calcul. Nous explorerons ensuite comment les relations et les fonctions permettent d'imposer une structure à ces ensembles, créant des notions aussi fondamentales que l'équivalence et l'ordre, essentielles à la classification et à l'optimisation.

Armés de ces outils structurels, nous nous tournerons vers l'analyse combinatoire pour apprendre à compter de manière systématique et efficace. Ces techniques de dénombrement nous mèneront naturellement à l'arithmétique modulaire et à la théorie des nombres, domaines dont l'élégance théorique se révèle d'une importance pratique capitale, notamment en cryptographie.

Enfin, le chapitre culminera avec l'étude de concepts avancés qui illustrent la puissance de l'approche discrète. Nous aborderons la notion de cardinalité pour comparer la taille des infinis, ce qui nous conduira à l'une des limitations les plus profondes de l'informatique : l'existence de problèmes non calculables. Nous introduirons les relations de récurrence et les fonctions génératrices, des outils puissants pour l'analyse d'algorithmes. Une introduction aux probabilités discrètes nous permettra de formaliser l'analyse en cas moyen et le raisonnement en présence d'incertitude. Pour synthétiser l'ensemble de ces connaissances, nous conclurons par une étude de cas détaillée de l'algorithme de cryptographie RSA, un système complexe dont la conception, la validité et la sécurité reposent entièrement sur les principes des structures discrètes et de la théorie des nombres explorés dans ce chapitre.

Section 1 : Les Fondements – La Théorie des Ensembles

La théorie des ensembles est le socle sur lequel reposent les mathématiques modernes et, par extension, une grande partie de l'informatique théorique. Elle fournit un langage formel pour décrire des collections d'objets et des opérations logiques sur ces collections. La maîtrise de ce langage est un prérequis essentiel, car il permet de définir avec précision les structures de données, les espaces d'états d'un système ou encore les domaines de validité d'un algorithme.

1.1 Ensembles et Éléments

Définitions formelles

Un **ensemble** est une collection d'objets bien définis et distincts, appelés ses **éléments**.[3] La notion d'ensemble est primitive, c'est-à-dire qu'on ne la définit pas à partir de concepts plus simples, mais on l'appréhende par ses propriétés. Le caractère "bien défini" signifie que pour n'importe quel objet, on doit pouvoir déterminer sans ambiguïté s'il appartient ou non à l'ensemble. Le caractère "distinct" signifie qu'un élément ne peut pas apparaître plus d'une fois dans un ensemble.

Il existe deux manières principales de décrire un ensemble [4] :

1. **En extension :** en énumérant tous ses éléments entre accolades. Par exemple, l'ensemble A des voyelles de l'alphabet français s'écrit A={a,e,i,o,u,y}. L'ordre des éléments n'a pas d'importance : {a,e,i}={i,a,e}.
2. **En compréhension :** en décrivant une propriété commune à tous ses éléments. On utilise la notation {x|P(x)}, qui se lit "l'ensemble des x tels que la propriété P(x) est vraie". Par exemple, l'ensemble B des entiers naturels pairs peut s'écrire B={x∈N|∃k∈N,x=2k}.

Notations et concepts de base

- **Appartenance :** Si un objet x est un élément de l'ensemble E, on note x∈E. Dans le cas contraire, on note x∈/E.[6]
- **Inclusion :** Un ensemble A est **inclus** dans un ensemble B, ou est un **sous-ensemble** de B, si tous les éléments de A sont aussi des éléments de B. On note A⊆B. Formellement : A⊆B⟺(∀x,x∈A⟹x∈B).[7] La notation A⊂B est souvent utilisée pour signifier la même chose, mais peut parfois désigner une inclusion stricte (A⊆B et A≠B).
- **Égalité :** Deux ensembles A et B sont égaux s'ils ont exactement les mêmes éléments. Cela se démontre par le principe de **double inclusion** : A=B⟺(A⊆B∧B⊆A).[7]

Il est crucial de distinguer l'appartenance de l'inclusion. Par exemple, si E={1,2,3}, alors 2∈E, mais {2}⊆E. L'élément 2 n'est pas un ensemble, tandis que {2} est un ensemble (un singleton).[3]

Ensembles spécifiques

- **L'ensemble vide :** C'est l'ensemble qui ne contient aucun élément. Il est noté ∅ ou {}.[5] Pour tout ensemble A, on a toujours ∅⊆A.
- **Le singleton :** C'est un ensemble qui ne contient qu'un seul élément, comme {x}.[3]
- **L'ensemble des parties :** Pour tout ensemble E, l'ensemble de tous les sous-ensembles de E est appelé **l'ensemble des parties** de E, et est noté P(E).[4] Formellement,
  P(E)={A|A⊆E}. Par exemple, si E={1,2}, alors P(E)={∅,{1},{2},{1,2}}.[11] Il est important de noter que si x∈E, alors {x}∈P(E).[3] Si un ensemble
  E est fini et contient n éléments, son ensemble des parties P(E) contient 2n éléments.[11]

1.2 Opérations sur les Ensembles

À partir d'ensembles existants, on peut en construire de nouveaux à l'aide d'opérations fondamentales, souvent visualisées à l'aide de diagrammes de Venn.

- **Union (ou Réunion) :** L'union de deux ensembles A et B, notée A∪B, est l'ensemble de tous les éléments qui appartiennent à A, ou à B, ou aux deux.[4]
  A∪B={x|x∈A∨x∈B}
- **Intersection :** L'intersection de deux ensembles A et B, notée A∩B, est l'ensemble de tous les éléments qui appartiennent à la fois à A et à B.[4]
  A∩B={x|x∈A∧x∈B}

  Si A∩B=∅, on dit que les ensembles A et B sont disjoints.5
- **Complémentaire :** Si A est un sous-ensemble d'un ensemble universel U, le complémentaire de A dans U, noté Ac ou ∁UA, est l'ensemble de tous les éléments de U qui n'appartiennent pas à A.[8]
  Ac={x∈U|x∈/A}
- **Différence :** La différence de A et B, notée A\B, est l'ensemble des éléments qui sont dans A mais pas dans B.[5] On a A\B=A∩Bc.
- **Différence symétrique :** La différence symétrique de A et B, notée AΔB, est l'ensemble des éléments qui appartiennent à A ou à B, mais pas aux deux.[3] On a
  AΔB=(A\B)∪(B\A)=(A∪B)\(A∩B).
- **Produit cartésien :** Le produit cartésien de deux ensembles A et B, noté A×B, est l'ensemble de tous les **couples ordonnés** (a,b) où a∈A et b∈B.[5]
  A×B={(a,b)|a∈A∧b∈B}

  L'ordre dans un couple est primordial : si a□=b, alors (a,b)□=(b,a), contrairement à la paire {a,b}={b,a}.3 Le plan cartésien

R×R=R2 est l'exemple le plus connu de produit cartésien.

1.3 Propriétés des Opérations Ensemblistes

Les opérations ensemblistes obéissent à un ensemble de règles algébriques qui structurent leur manipulation et sont essentielles pour les démonstrations formelles. Ces règles sont analogues à celles de l'algèbre des nombres, mais avec des spécificités propres.

- **Commutativité :** L'ordre des opérandes ne change pas le résultat.
  - A∪B=B∪A
  - A∩B=B∩A
- **Associativité :** Le groupement des opérations ne change pas le résultat, ce qui permet d'omettre les parenthèses dans des expressions comme A∪B∪C.[4]
  - (A∪B)∪C=A∪(B∪C)
  - (A∩B)∩C=A∩(B∩C)
- **Idempotence :** Opérer un ensemble avec lui-même ne le change pas.
  - A∪A=A
  - A∩A=A
- **Absorption :** Une opération peut "absorber" l'autre dans certaines conditions.
  - A∪(A∩B)=A
  - A∩(A∪B)=A
- **Éléments neutres :** L'ensemble vide est l'élément neutre de l'union, et l'ensemble universel U est celui de l'intersection.
  - A∪∅=A
  - A∩U=A

Deux propriétés particulièrement importantes sont la distributivité et les lois de De Morgan. Elles établissent un pont direct entre la théorie des ensembles et la logique propositionnelle, révélant que ces deux formalismes sont deux facettes d'une même structure sous-jacente : une algèbre de Boole. Chaque opération ensembliste (∪, ∩, ^c) correspond à un connecteur logique (∨ pour OU, ∧ pour ET, ¬ pour NON). Ainsi, une preuve d'identité ensembliste peut être directement traduite en une preuve d'équivalence logique, et vice-versa. Cette correspondance est fondamentale en informatique, où la manipulation de données (ensembles) et l'exécution de conditions logiques sont omniprésentes.

Distributivité

L'intersection est distributive par rapport à l'union, et l'union est distributive par rapport à l'intersection.[4]

1. **Distributivité de l'intersection sur l'union :** A∩(B∪C)=(A∩B)∪(A∩C)
2. **Distributivité de l'union sur l'intersection :** A∪(B∩C)=(A∪B)∩(A∪C)

Démonstration de la première loi de distributivité :
Pour prouver l'égalité $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$, nous utilisons le principe de double inclusion.9

- Première inclusion : Montrons que $A \cap (B \cup C) \subseteq (A \cap B) \cup (A \cap C)$.
  Soit $x \in A \cap (B \cup C)$. Par définition de l'intersection, cela signifie que $(x \in A) \land (x \in B \cup C)$.
  Par définition de l'union, $x \in B \cup C$ signifie $(x \in B) \lor (x \in C)$.
  Donc, nous avons $x \in A \land (x \in B \lor x \in C)$.
  En logique, l'opérateur ET ($\land$) est distributif sur l'opérateur OU ($\lor$), donc cette proposition est équivalente à $(x \in A \land x \in B) \lor (x \in A \land x \in C)$.
  La première partie, $(x \in A \land x \in B)$, signifie $x \in A \cap B$.
  La seconde partie, $(x \in A \land x \in C)$, signifie $x \in A \cap C$.
  La proposition complète devient donc $(x \in A \cap B) \lor (x \in A \cap C)$, ce qui, par définition de l'union, signifie $x \in (A \cap B) \cup (A \cap C)$.
  Ainsi, tout élément du premier ensemble est aussi dans le second, ce qui prouve l'inclusion.15
- Seconde inclusion : Montrons que $(A \cap B) \cup (A \cap C) \subseteq A \cap (B \cup C)$.
  Soit $x \in (A \cap B) \cup (A \cap C)$. Par définition de l'union, cela signifie que $(x \in A \cap B) \lor (x \in A \cap C)$.
  - **Cas 1 :** $x \in A \cap B$. Alors $x \in A$ et $x \in B$. Si $x \in B$, alors il est aussi dans $B \cup C$. Comme x est aussi dans A, on a $x \in A \cap (B \cup C)$.
  - Cas 2 : $x \in A \cap C$. Alors $x \in A$ et $x \in C$. Si $x \in C$, alors il est aussi dans $B \cup C$. Comme x est aussi dans A, on a $x \in A \cap (B \cup C)$.
    Dans les deux cas, x appartient à $A \cap (B \cup C)$. L'inclusion est donc prouvée.15

Les deux inclusions étant vérifiées, l'égalité est démontrée. La preuve de la distributivité de l'union sur l'intersection suit un raisonnement analogue.[14]

Lois de De Morgan

Les lois de De Morgan décrivent comment le complémentaire interagit avec l'union et l'intersection.[8]

1. $\complement(A \cup B) = \complement A \cap \complement B$
2. $\complement(A \cap B) = \complement A \cup \complement B$

Démonstration de la première loi de De Morgan :
Nous allons prouver l'égalité par équivalences logiques successives, ce qui est plus direct que la double inclusion.17

Un élément x appartient au complémentaire de $A \cup B$ si et seulement s'il n'appartient pas à $A \cup B$.
$$\begin{aligned} x \in \complement(A \cup B) &\iff x \notin (A \cup B) \\ &\iff \neg(x \in A \cup B) \\ &\iff \neg(x \in A \lor x \in B) \quad \text{(par définition de l'union)} \\ &\iff (\neg(x \in A)) \land (\neg(x \in B)) \quad \text{(par la loi de De Morgan en logique)} \\ &\iff (x \notin A) \land (x \notin B) \\ &\iff (x \in \complement A) \land (x \in \complement B) \quad \text{(par définition du complémentaire)} \\ &\iff x \in (\complement A \cap \complement B) \quad \text{(par définition de l'intersection)} \end{aligned}$$
Comme l'appartenance à l'ensemble de gauche est logiquement équivalente à l'appartenance à l'ensemble de droite pour n'importe quel élément x, les deux ensembles sont égaux.17 La démonstration de la seconde loi est similaire.

Section 2 : Relations et Fonctions – Structurer l'Information

Si les ensembles permettent de regrouper des objets, les **relations** et les **fonctions** permettent d'établir des liens et des correspondances entre ces objets. Elles sont le fondement de la modélisation de presque toutes les structures en informatique, des bases de données relationnelles aux graphes de dépendances, en passant par les hiérarchies de classes en programmation orientée objet.

2.1 Relations Binaires

Une **relation binaire** R d'un ensemble A vers un ensemble B est formellement définie comme un sous-ensemble du produit cartésien $A \times B$. Si $(a,b) \in R$, on dit que "a est en relation avec b" et on note aRb.[19] Lorsque

A=B, on parle de relation binaire sur l'ensemble A.

Propriétés des relations binaires

Les relations binaires peuvent posséder plusieurs propriétés fondamentales qui déterminent leur nature et leurs applications. Soit R une relation sur un ensemble E.

- Réflexivité : R est réflexive si chaque élément est en relation avec lui-même.

  $\forall x \in E, xRx$
  - **Exemple :** La relation "inférieur ou égal à" ($\leq$) sur R est réflexive car $x \leq x$ pour tout réel x.[21] L'inclusion ($\subseteq$) sur P(E) est aussi réflexive car $A \subseteq A$ pour toute partie A.[19]
  - **Contre-exemple :** La relation "strictement inférieur à" (<) sur R n'est pas réflexive car x<x est faux.[21]
- Symétrie : R est symétrique si, lorsque x est en relation avec y, alors y est aussi en relation avec x.

  $\forall x,y \in E, (xRy \Longrightarrow yRx)$
  - **Exemple :** La relation d'égalité (=) sur N est symétrique car si m=n, alors n=m.[19] La relation "est marié(e) à" sur un ensemble de personnes est symétrique.
  - **Contre-exemple :** La relation ≤ sur R n'est pas symétrique. Par exemple, 3≤5 mais 5□≤3.[21]
- Antisymétrie : R est antisymétrique si les seuls éléments qui sont mutuellement en relation sont les éléments égaux à eux-mêmes.

  $\forall x,y \in E, ((xRy \wedge yRx) \Longrightarrow x=y)$
  - **Exemple :** La relation de divisibilité sur $\mathbb{N}*$ est antisymétrique. Si a divise b et b divise a, alors a=b. La relation ≤ sur R est également antisymétrique, car si $x \leq y$ et $y \leq x$, alors x=y.[22]

- **Contre-exemple :** La relation "différent de" (□=) sur N n'est pas antisymétrique. On a 0□=1 et 1□=0, mais 0□=1.[22]

- Transitivité : R est transitive si un lien entre x et y et un lien entre y et z impliquent un lien direct entre x et z.

  ∀x,y,z∈E,((xRy∧yRz)⟹xRz)

  - **Exemple :** La relation de divisibilité sur N est transitive. Si a divise b et b divise c, alors a divise c.[23] L'inclusion est aussi transitive.[19]
  - **Contre-exemple :** La relation "est l'ami(e) de" n'est pas transitive. La relation □= sur N n'est pas transitive : 0□=1 et 1□=0, mais 0□=0 est faux.[22]

Ces propriétés ne sont pas de simples classifications ; elles agissent comme des axiomes qui, une fois combinés, définissent des structures mathématiques de première importance. La symétrie et l'antisymétrie, par exemple, sont des propriétés presque mutuellement exclusives (une relation peut être les deux seulement si son graphe est un sous-ensemble de la diagonale [24]). Cette distinction crée une bifurcation fondamentale dans la nature des relations : celles qui modélisent une forme de "similitude" ou d' "interchangeabilité" (les relations d'équivalence) et celles qui modélisent une "précedence" ou une "hiérarchie" (les relations d'ordre). Cette dualité est au cœur de nombreux paradigmes algorithmiques : les algorithmes de regroupement (clustering) cherchent à identifier des classes d'équivalence, tandis que les algorithmes de tri ou de planification topologique cherchent à établir une relation d'ordre.

Relation	Ensemble	Réflexive	Symétrique	Antisymétrique	Transitive
Égalité (=)	Z	Oui	Oui	Oui	Oui
Inférieur ou égal (≤)	R	Oui	Non (3≤5 mais 5□≤3)	Oui	Oui
Divisibilité ($	$)	N∗	Oui	Non ($2	4$ mais 4∤2)
Congruence (≡(modn))	Z	Oui	Oui	Non (2≡5(mod3) et 5≡2(mod3) mais 2□=5)	Oui
Inclusion (⊆)	P(E)	Oui	Non	Oui	Oui
Perpendicularité (⊥)	Droites du plan	Non	Oui	Non	Non (d1⊥d2 et d2⊥d3

					⟹d1∥d3)

2.2 Relations d'Équivalence

Une **relation d'équivalence** est une relation binaire qui est à la fois **réflexive, symétrique et transitive** (RST).[19] Elle généralise la notion d'égalité et permet de regrouper des éléments qui partagent une propriété commune.

- **Exemples :**
  - L'égalité sur n'importe quel ensemble.
  - La congruence modulo n sur Z : a≡b(modn) si n divise a−b.[27]
  - La relation "avoir la même parité" sur N.
  - La relation "être parallèle à" sur l'ensemble des droites du plan.[26]

Classes d'équivalence et Partition

Le concept central associé à une relation d'équivalence ∼ sur un ensemble E est celui de **classe d'équivalence**. Pour tout élément x∈E, sa classe d'équivalence, notée [x] ou x^{-}, est l'ensemble de tous les éléments de E qui sont équivalents à x.[26]

[x]={y∈E|x∼y}

Les classes d'équivalence possèdent une propriété fondamentale : pour deux éléments quelconques x et y de E, leurs classes [x] et [y] sont soit strictement identiques, soit totalement disjointes.26
∀x,y∈E,([x]=[y])∨([x]∩[y]=∅)

Cette propriété découle directement des axiomes RST. Si deux classes [x] et [y] ont un élément commun z, alors x∼z et y∼z. Par symétrie, z∼y. Par transitivité, x∼y. On peut alors montrer que tout élément de [x] est dans [y] et vice-versa, prouvant que [x]=[y].29
Il en résulte qu'une relation d'équivalence induit une **partition** de l'ensemble E, c'est-à-dire une décomposition de E en une collection de sous-ensembles non vides, deux à deux disjoints, dont l'union est E tout entier.[30] Chaque sous-ensemble de la partition est une classe d'équivalence.

L'ensemble de toutes les classes d'équivalence est appelé l'**ensemble quotient** de E par ∼, noté E/∼. Par exemple, pour la congruence modulo 3 sur Z, il y a trois classes d'équivalence : $ = {\dots, -3, 0, 3, 6, \dots}$, $ = \{\dots, -2, 1, 4, 7, \dots\}$, et $ = {\dots, -1, 2, 5, 8, \dots}$. L'ensemble quotient est Z/3Z={,,}.[27]

2.3 Relations d'Ordre

Une **relation d'ordre** est une relation binaire qui est **réflexive, antisymétrique et transitive** (RAT).[32] Elle formalise la notion intuitive de comparaison ou de hiérarchie. Un ensemble muni d'une relation d'ordre est appelé un

ensemble partiellement ordonné (en anglais, *partially ordered set* ou **poset**).

- **Ordre partiel et ordre total :**
  - Un ordre est dit **partiel** si certains éléments peuvent être incomparables. C'est-à-dire, il peut exister x,y∈E tels que x⋢y et y⋢x.[35]
    - **Exemple :** La relation de divisibilité sur N∗. Les entiers 2 et 3 sont incomparables car 2 ne divise pas 3 et 3 ne divise pas 2.[37] L'inclusion sur P({a,b,c}) est un ordre partiel : {a} et {b} sont incomparables.[34]
  - Un ordre est dit **total** (ou linéaire) si tous les éléments sont comparables. Un ensemble muni d'un ordre total est appelé une **chaîne**.[33]
    - **Exemple :** La relation ≤ sur R est un ordre total.[39]
- **Éléments remarquables dans un poset (E,≤) :**
  - Un élément m∈E est **minimal** si aucun autre élément n'est strictement plus petit que lui (∀x∈E,x≤m⟹x=m).
  - Un élément M∈E est **maximal** si aucun autre élément n'est strictement plus grand que lui (∀x∈E,m≤x⟹x=m).[40]
  - Un élément m0∈E est le **plus petit élément** (ou minimum) s'il est plus petit que tous les autres éléments (∀x∈E,m0≤x). S'il existe, il est unique et est le seul élément minimal.
  - Un élément M0∈E est le **plus grand élément** (ou maximum) s'il est plus grand que tous les autres éléments (∀x∈E,x≤M0). S'il existe, il est unique et est le seul élément maximal.[38]

Diagrammes de Hasse

Pour visualiser la structure d'un ensemble ordonné **fini**, on utilise un **diagramme de Hasse**. C'est une représentation graphique simplifiée de la relation d'ordre qui élimine les informations redondantes.[43] La construction suit trois règles [35] :

1. Les éléments de l'ensemble sont représentés par des points (ou sommets).
2. Si x<y, le point représentant y est placé plus haut que celui représentant x.
3. Un segment de droite relie x à y si et seulement si y **recouvre** x, c'est-à-dire x<y et il n'existe aucun z tel que x<z<y.

Grâce à ces conventions, les arêtes dues à la réflexivité (boucles sur chaque sommet) et à la transitivité (raccourcis) sont omises, et l'orientation des arêtes est implicite (toujours de bas en haut).[33]

Exemple : Le diagramme de Hasse pour la relation de divisibilité sur l'ensemble des diviseurs de 12, D12={1,2,3,4,6,12}, est le suivant [41] :

Extrait de code

```
graph TD
  1 --> 2;
  1 --> 3;
  2 --> 4;
  2 --> 6;
  3 --> 6;
  4 --> 12;
  6 --> 12;
```

Ce diagramme montre que 1 est le plus petit élément, 12 est le plus grand. Les éléments 4 et 6 sont maximaux dans le sous-ensemble {1,2,3,4,6}, mais ne sont pas comparables entre eux.

2.4 Fonctions

Une **fonction** (ou application) f d'un ensemble de départ E (le **domaine**) vers un ensemble d'arrivée F (le **codomaine**) est une relation binaire de E vers F telle que chaque élément de E est associé à **un et un seul** élément de F.[13] On note

$f:E \to F$.

Injection, Surjection, Bijection

- **Injection :** Une fonction $f:E \to F$ est **injective** (ou une injection) si des éléments distincts du domaine ont toujours des images distinctes dans le codomaine.[16]
  $\forall x1,x2 \in E,(f(x1)=f(x2) \Longrightarrow x1=x2)$

  Cela signifie que tout élément de F a au plus un antécédent.16 Si
  E et F sont finis, une injection n'est possible que si $|E| \leq |F|$.[46]
- **Surjection :** Une fonction $f:E \to F$ est **surjective** (ou une surjection) si chaque élément du codomaine est l'image d'au moins un élément du domaine.[16]
  $\forall y \in F,\exists x \in E,f(x)=y$

  Cela signifie que l'image de la fonction est égale à son codomaine, f(E)=F.16 Si
  E et F sont finis, une surjection n'est possible que si $|E| \geq |F|$.[46]
- **Bijection :** Une fonction $f:E \to F$ est **bijective** (ou une bijection) si elle est à la fois injective et surjective.[45]

$\forall y \in F, \exists! x \in E, f(x)=y$

Cela signifie que chaque élément de F a un unique antécédent. Une bijection établit une correspondance un-à-un entre les éléments de E et de F. Si E et F sont finis, une bijection n'existe que si $|E|=|F|$.45

Composition et Fonction Inverse

- **Composition :** Étant données deux fonctions $f:E \to F$ et $g:F \to G$, leur **composée** est la fonction $g \circ f:E \to G$ définie par $(g \circ f)(x)=g(f(x))$ pour tout $x \in E$.[13]
  - **Propriété :** La composition préserve l'injectivité et la surjectivité.
    - Si f et g sont injectives, alors $g \circ f$ est injective.
    - Si f et g sont surjectives, alors $g \circ f$ est surjective.
    - Par conséquent, si f et g sont bijectives, $g \circ f$ est bijective.[47]

  Démonstration (Injectivité) :Supposons f et g injectives. Soient $x1, x2 \in E$ tels que $(g \circ f)(x1)=(g \circ f)(x2)$.Cela signifie $g(f(x1))=g(f(x2))$.Comme g est injective, on en déduit $f(x1)=f(x2)$.Comme f est injective, on en déduit $x1=x2$.Donc, $g \circ f$ est injective.48
- **Fonction Inverse :** Une fonction $f:E \to F$ est dite **inversible** s'il existe une fonction $g:F \to E$ telle que $g \circ f=id_E$ et $f \circ g=id_F$, où id_E est la fonction identité sur E.[16]
  - **Théorème :** Une fonction f est inversible si et seulement si elle est bijective.
  - Si elle existe, cette fonction g est unique, est appelée la **fonction inverse** (ou réciproque) de f, et est notée f^{-1}.[16]

Démonstration (esquisse) :($\Longrightarrow$) Supposons f bijective. Pour tout $y \in F$, il existe un unique $x \in E$ tel que $f(x)=y$. On peut donc définir une fonction $g:F \to E$ par $g(y)=x$. Par construction, $g(f(x))=x$ et $f(g(y))=y$, ce qui prouve que g est l'inverse de f.($\Longleftarrow$) Supposons qu'il existe un inverse g. Pour montrer que f est injective, si $f(x1)=f(x2)$, alors $g(f(x1))=g(f(x2))$, ce qui donne $x1=x2$. Pour montrer que f est surjective, pour tout $y \in F$, posons $x=g(y)$. Alors $f(x)=f(g(y))=y$, donc y a un antécédent.16

Section 3 : L'Art du Dénombrement – Analyse Combinatoire

L'analyse combinatoire est la branche des mathématiques qui étudie les configurations d'objets. Elle cherche à répondre à des questions du type "Combien y a-t-il de...?". En informatique, ces questions sont omniprésentes : combien de mots de passe de 8 caractères peut-on former? Combien de chemins existent entre deux nœuds dans un réseau? Combien de comparaisons un algorithme de tri effectue-t-il dans le pire des cas? Le dénombrement est donc essentiel pour l'analyse de la complexité, la conception de structures de données et le calcul des probabilités.

3.1 Principes Fondamentaux

Toutes les techniques de dénombrement, même les plus complexes, reposent sur deux principes de base qui relient les opérations sur les ensembles à des opérations arithmétiques.

- **Principe de la Somme (ou de l'Addition) :** Si une tâche peut être accomplie de n1 manières et une seconde tâche de n2 manières, et que ces deux tâches ne peuvent pas être accomplies simultanément (elles sont mutuellement exclusives), alors il y a n1+n2 manières d'accomplir l'une ou l'autre de ces tâches. En termes d'ensembles, si A et B sont deux ensembles finis **disjoints**, alors $|A \cup B| = |A| + |B|$.[52]
- **Principe du Produit (ou de la Multiplication) :** Si une procédure peut être décomposée en une séquence de deux tâches, où il y a n1 manières d'accomplir la première tâche et, pour chacune de ces manières, n2 manières d'accomplir la seconde, alors il y a n1×n2 manières d'accomplir la procédure complète. En termes d'ensembles, pour deux ensembles finis A et B, le cardinal de leur produit cartésien est $|A \times B| = |A| \times |B|$.[10]

3.2 Permutations et Arrangements

Les arrangements et permutations concernent les problèmes de dénombrement où **l'ordre des éléments est important**. La distinction cruciale entre les différentes formules de dénombrement réside dans deux questions : l'ordre importe-t-il et les répétitions sont-elles permises?

- Arrangements avec répétition (k-uplets) :
  Il s'agit du nombre de manières de choisir k éléments parmi n éléments distincts, avec remise et en tenant compte de l'ordre. Pour chacun des k choix, il y a n possibilités. Par le principe du produit, le nombre total d'arrangements est $n \times n \times \cdots \times n$ (k fois).

  $AR(n,k) = n^k$
  - **Exemple :** Le nombre de mots de 3 lettres que l'on peut former avec l'alphabet {A, B} est $2^3=8$ (AAA, AAB, ABA, BAA, ABB, BAB, BBA, BBB).[12]
- Arrangements sans répétition (k-permutations) :
  Il s'agit du nombre de manières de choisir et d'ordonner k éléments parmi n éléments distincts, sans remise. Le premier élément peut être choisi de n manières, le deuxième de $n-1$ manières,..., et le k-ième de $n-k+1$ manières. Par le principe du produit, on obtient 54 :
  $P(n,k) = n \times (n-1) \times \cdots \times (n-k+1)$

  En utilisant la notation factorielle ($n! = n \times (n-1) \times \cdots \times 1$), cette formule se réécrit de manière compacte :

  $P(n,k) = \frac{n!}{(n-k)!}$
  - **Exemple :** Le nombre de podiums possibles (or, argent, bronze) dans une course de 8 athlètes est $P(8,3) = 8 \times 7 \times 6 = 336$.[12]
- Permutations :
  Une permutation est un arrangement de tous les éléments d'un ensemble. C'est un cas particulier d'arrangement sans répétition où k=n. Le nombre de permutations d'un ensemble de n éléments est 12 :
  $P(n,n) = n!$
  - **Exemple :** Le nombre de manières d'ordonner les lettres du mot "MATH" est $4! = 24$.

3.3 Combinaisons

Les combinaisons concernent les problèmes de dénombrement où **l'ordre des éléments n'est pas important**. Il s'agit de choisir un sous-ensemble d'éléments.

- Combinaisons sans répétition :
  Il s'agit du nombre de manières de choisir k éléments parmi n éléments distincts, sans remise et sans tenir compte de l'ordre. On note ce nombre C(n,k) ou (kn), et on le lit "k parmi n".55
  Pour dériver la formule, on peut partir d'un raisonnement unificateur qui relie arrangements et combinaisons. Un arrangement de k éléments peut être formé en deux étapes :
  1. Choisir un sous-ensemble de k éléments (une combinaison). Il y a (kn) manières de le faire.
  2. Ordonner ces k éléments. Il y a k! manières de le faire (une permutation).

  Par le principe du produit, le nombre d'arrangements est le produit du nombre de combinaisons par le nombre de permutations de ces dernières : P(n,k)=(kn)×k!. En isolant le terme qui nous intéresse, on obtient la formule fondamentale des combinaisons [56] :(kn)=k!P(n,k)=k!(n−k)!n!Cette approche révèle que la distinction entre arrangements et combinaisons est une question de "division par les symétries" : on compte d'abord les objets ordonnés, puis on divise par le nombre de redondances créées par l'ordre (k!) pour obtenir le nombre d'objets non ordonnés.
  - **Exemple :** Le nombre de manières de choisir un comité de 3 personnes dans un groupe de 10 est (310)=3!7!10! =3×2×110×9×8=120.

	Ordre important	**Ordre non important**
Sans répétition	Arrangement (k-permutation) P(n,k)=(n−k)!n!	Combinaison (kn)=k!(n−k)!n!
Avec répétition	k-uplet nk	Combinaison avec répétition (kn+k−1)

3.4 Le Binôme de Newton

La formule du binôme de Newton est une généralisation des identités remarquables comme (a+b)2=a2+2ab+b2. Elle fournit une expression pour (a+b)n en utilisant les coefficients binomiaux.[58]

Identité et Triangle de Pascal

Les coefficients binomiaux peuvent être calculés de manière récursive grâce à l'**identité de Pascal** [59] :

(kn+1)=(k−1n)+(kn)

Démonstration combinatoire :
Considérons un ensemble E de n+1 éléments. On souhaite compter le nombre de sous-ensembles de E de taille k. On distingue un élément particulier, disons x0∈E. Un sous-ensemble de E de taille k peut soit contenir x0, soit ne pas le contenir.

- **Cas 1 : Le sous-ensemble contient x0.** Il faut alors choisir les k−1 autres éléments parmi les n éléments restants de E\{x0}. Il y a (k−1n) manières de le faire.
- **Cas 2 : Le sous-ensemble ne contient pas x0.** Il faut alors choisir les k éléments parmi les n éléments restants de E\{x0}. Il y a (kn) manières de le faire.

Comme ces deux cas sont disjoints, par le principe de la somme, le nombre total de sous-ensembles est (k−1n)+(kn), ce qui prouve l'identité.[59]

Cette relation est à la base de la construction du **triangle de Pascal**, où chaque nombre est la somme des deux nombres situés juste au-dessus de lui.[60]

Formule du binôme

Pour tous nombres a,b (réels ou complexes) et tout entier naturel n, on a :

(a+b)n=k=0∑n(kn)akbn−k

Le coefficient (kn) correspond au nombre de fois où le terme akbn−k apparaît lorsqu'on développe le produit (a+b)(a+b)…(a+b) (n fois). Choisir le terme akbn−k revient à choisir k parenthèses parmi les n desquelles on prendra le terme a (et donc on prendra b dans les n−k autres).61
Démonstration par récurrence :
Soit P(n) la proposition (a+b)n=∑k=0n(kn)akbn−k.

- **Initialisation (n=0) :** (a+b)0=1. La somme se réduit à (00)a0b0=1. P(0) est vraie.[62]
- Hérédité : Supposons P(n) vraie pour un certain n≥0. Montrons P(n+1).
  \begin{align*} (a+b)^{n+1} &= (a+b)(a+b)^n \ &= (a+b) \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k} \quad \text{(par hypothèse de récurrence)} \ &= a \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k} + b \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k} \ &= \sum_{k=0}^{n} \binom{n}{k} a^{k+1} b^{n-k} + \sum_{k=0}^{n} \binom{n}{k} a^k b^{n+1-k} \end{align*}
  On effectue un changement d'indice j=k+1 dans la première somme :
  $$ (a+b)^{n+1} = \sum_{j=1}^{n+1} \binom{n}{j-1} a^j b^{n-(j-1)} + \sum_{k=0}^{n} \binom{n}{k} a^k b^{n+1-k} $$
  En renommant j par k et en isolant les termes extrêmes (k=n+1 dans la première somme et k=0 dans la seconde),

on obtient :
$$ (a+b)^{n+1} = \binom{n}{n}a^{n+1}b^0 + \sum_{k=1}^{n} \binom{n}{k-1} a^k b^{n+1-k} + \binom{n}{0}a^0b^{n+1} + \sum_{k=1}^{n} \binom{n}{k} a^k b^{n+1-k} $$
En regroupant les sommes et en utilisant (nn)=1=(n+1n+1) et (0n)=1=(0n+1) :
$$ (a+b)^{n+1} = \binom{n+1}{n+1}a^{n+1} + \binom{n+1}{0}b^{n+1} + \sum_{k=1}^{n} \left(\binom{n}{k-1} + \binom{n}{k} \right) a^k b^{n+1-k} $$
Grâce à l'identité de Pascal, le terme entre crochets devient (kn+1). On peut alors réintégrer les termes extrêmes dans la somme :

(a+b)n+1=k=0∑n+1(kn+1)akbn+1−k

La proposition P(n+1) est donc vraie. Par le principe de récurrence, la formule est vraie pour tout n∈N.58

Section 4 : Arithmétique et Algorithmique – Les Nombres au Cœur du Calcul

L'arithmétique, l'étude des propriétés des nombres entiers, est l'une des plus anciennes branches des mathématiques. En informatique, elle transcende son statut de discipline théorique pour devenir un outil algorithmique fondamental. Les propriétés de la divisibilité, des nombres premiers et des congruences ne sont pas seulement des curiosités abstraites ; elles sont au cœur d'algorithmes efficaces pour des tâches allant de la génération de nombres pseudo-aléatoires à la sécurisation des communications mondiales.

4.1 Divisibilité et Nombres Premiers

- **Division Euclidienne :** Pour deux entiers a (dividende) et b□=0 (diviseur), il existe un unique couple d'entiers (q,r) (quotient et reste) tel que a=bq+r et 0≤r<|b|.[23] Cet algorithme simple est la base de nombreuses opérations en arithmétique.
- **Plus Grand Commun Diviseur (PGCD) :** Le PGCD de deux entiers non nuls a et b, noté pgcd(a,b), est le plus grand entier positif qui divise à la fois a et b. Deux nombres sont dits **premiers entre eux** si leur PGCD est 1.[65]
- **Nombres Premiers :** Un entier p>1 est dit **premier** s'il n'admet que deux diviseurs positifs : 1 et lui-même. Les nombres premiers sont les "atomes" de l'arithmétique.
- **Théorème Fondamental de l'Arithmétique :** Tout entier n>1 peut être écrit de manière **unique** (à l'ordre des facteurs près) comme un produit de nombres premiers.[66]
  n=p1a1p2a2…pkak

  Démonstration (par induction forte) :
  - **Cas de base :** n=2 est premier, sa décomposition est unique.
  - **Hypothèse d'induction :** Supposons que tout entier m tel que 2≤m<n admette une décomposition unique en

facteurs premiers.

- **Étape d'induction :** Considérons l'entier n.
  - **Existence :** Si n est premier, il est sa propre décomposition. S'il est composé, n=ab avec 1<a,b<n. Par hypothèse d'induction, a et b ont des décompositions en facteurs premiers. Le produit de ces décompositions donne une décomposition pour n.
  - **Unicité :** Supposons que n ait deux décompositions : n=p1…pr=q1…qs. Le nombre premier p1 divise le produit q1…qs. Par le lemme d'Euclide (si un premier divise un produit, il divise l'un des facteurs), p1 doit diviser l'un des qj. Comme les qj sont premiers, p1 doit être égal à ce qj. En réordonnant les facteurs, on peut supposer p1=q1. On peut alors simplifier par p1 pour obtenir p2…pr=q2…qs. Ce nombre est strictement inférieur à n. Par hypothèse d'induction, sa décomposition est unique. Donc, les listes (p2,…,pr) et (q2,…,qs) sont les mêmes à permutation près. Il en va de même pour les décompositions initiales de n.[67]

4.2 Arithmétique Modulaire

L'arithmétique modulaire est un système d'arithmétique pour les entiers où les nombres "s'enroulent" après avoir atteint une certaine valeur, le **module**.

- **Relation de Congruence :** Deux entiers a et b sont dits **congrus modulo n** si leur différence a−b est un multiple de n. On note a≡b(modn).[65] Cela revient à dire que
  a et b ont le même reste dans la division euclidienne par n.
  Il s'agit d'une relation d'équivalence : elle est réflexive (a≡a), symétrique (a≡b⟹b≡a) et transitive (a≡b et b≡c⟹a≡c).69 Les classes d'équivalence sont les ensembles de nombres ayant le même reste, formant l'ensemble quotient
  Z/nZ.
- **Opérations Modulaires :** La congruence est compatible avec l'addition et la multiplication. Si a≡b(modn) et c≡d(modn), alors :
  - a+c≡b+d(modn)
  - ac≡bd(modn)
    Ces propriétés permettent de définir une arithmétique cohérente sur l'ensemble des restes {0,1,…,n−1}.70
- **Équations de Congruence Linéaire :** Il s'agit d'équations de la forme ax≡b(modn).
  - **Théorème :** Une solution existe si et seulement si pgcd(a,n) divise b.[71]
  - **Cas particulier important :** Si pgcd(a,n)=1, alors a admet un **inverse modulaire** unique modulo n, noté a−1. C'est un entier tel que a·a−1≡1(modn). L'existence de cet inverse transforme la résolution de l'équation en une simple multiplication : x≡a−1b(modn).[70]

4.3 L'Algorithme d'Euclide Étendu

La question cruciale pour la résolution des congruences linéaires et pour la cryptographie est de savoir comment calculer efficacement le PGCD et l'inverse modulaire. La réponse est fournie par l'algorithme d'Euclide et sa version étendue. C'est ici que la théorie des nombres passe d'une collection de théorèmes d'existence à une boîte à outils algorithmique puissante.

- **Algorithme d'Euclide :** Pour calculer pgcd(a,b), on utilise la propriété pgcd(a,b)=pgcd(b,a(modb)).[74] On applique cette réduction de manière itérative jusqu'à obtenir un reste nul. Le dernier reste non nul est le PGCD.[75] Cet algorithme est extrêmement efficace, son temps d'exécution est logarithmique en la taille des entrées.
- **Identité de Bézout :** Pour tous entiers a et b, il existe des entiers u et v (appelés coefficients de Bézout) tels que au+bv=pgcd(a,b).[65] Ce théorème garantit, par exemple, l'existence d'un inverse modulaire pour a modulo n si pgcd(a,n)=1, car l'identité devient au+nv=1, ce qui implique au≡1(modn).
- **Algorithme d'Euclide Étendu :** Il s'agit d'une modification de l'algorithme d'Euclide qui, en plus de calculer le PGCD, calcule également les coefficients de Bézout u et v.[76] La méthode la plus courante consiste à maintenir à chaque étape de la division euclidienne une expression du reste comme une combinaison linéaire de a et b.[77] En partant des égalités triviales a=1·a+0·b et b=0·a+1·b, on peut calculer les coefficients pour chaque reste successif.

Exemple : Calculer l'inverse de 8 modulo 35. On cherche u tel que 8u≡1(mod35). On applique l'algorithme d'Euclide étendu à 35 et 8.

1. $35 = 4 \cdot 8 + 3 \Longrightarrow 3 = 35 - 4 \cdot 8$
2. $8 = 2 \cdot 3 + 2 \Longrightarrow 2 = 8 - 2 \cdot 3 = 8 - 2(35 - 4 \cdot 8) = 9 \cdot 8 - 2 \cdot 35$
3. $3 = 1 \cdot 2 + 1 \Longrightarrow 1 = 3 - 1 \cdot 2 = (35 - 4 \cdot 8) - (9 \cdot 8 - 2 \cdot 35) = 3 \cdot 35 - 13 \cdot 8$
   L'identité de Bézout est $1 = 3 \cdot 35 - 13 \cdot 8$. En regardant cette équation modulo 35, on obtient $-13 \cdot 8 \equiv 1 \pmod{35}$. L'inverse est donc −13, qui est congruent à 22(mod35).71

4.4 Théorèmes de Fermat et d'Euler

Ces théorèmes sont des résultats fondamentaux de l'arithmétique modulaire concernant l'exponentiation.

- Petit Théorème de Fermat : Si p est un nombre premier, alors pour tout entier a non divisible par p, on a :

  $a^{p-1} \equiv 1 \pmod{p}$

  Une forme équivalente, valide pour tout entier a, est $a^p \equiv a \pmod{p}$.78
- **Indicatrice d'Euler :** La fonction indicatrice d'Euler $\phi(n)$ (ou totient) compte le nombre d'entiers positifs inférieurs ou égaux à n qui sont premiers avec n.
  - Si p est premier, $\phi(p) = p - 1$.
  - Si p et q sont deux premiers distincts, $\phi(pq) = (p-1)(q-1)$.[80]
- Théorème d'Euler : Ce théorème généralise celui de Fermat à un module non premier. Si a et n sont des entiers premiers entre eux, alors :

  $a^{\phi(n)} \equiv 1 \pmod{n}$

Démonstration :
Soit $Un=(Z/nZ)*$ l'ensemble des classes d'équivalence modulo n qui sont inversibles. Cet ensemble forme un groupe multiplicatif d'ordre $\phi(n)$. Soit a^- la classe de a dans Un. D'après le théorème de Lagrange, l'ordre d'un élément d'un groupe fini divise l'ordre du groupe. Donc, l'ordre de a^- divise $\phi(n)$. Cela signifie que $a^-\phi(n)=1^-$, ce qui est équivalent à $a\phi(n)\equiv 1(mod n)$.80

Ce théorème est la clé de voûte de l'algorithme RSA, car il garantit que l'opération de déchiffrement annule bien celle du chiffrement.

Section 5 : Concepts Avancés et Applications aux Systèmes Complexes

Cette dernière section explore des concepts qui illustrent la puissance et la portée des mathématiques discrètes, en montrant comment les outils fondamentaux développés précédemment s'assemblent pour analyser des questions profondes sur la nature du calcul, modéliser des processus dynamiques et construire des systèmes sécurisés. Les idées présentées ici, de la hiérarchie des infinis à l'analyse probabiliste des algorithmes, forment un pont entre la théorie pure et les défis concrets du génie informatique.

5.1 Cardinalité et Calculabilité

Cardinalité des ensembles

La notion de **cardinalité** généralise la notion de "nombre d'éléments" aux ensembles infinis. Deux ensembles A et B ont la même cardinalité, noté $|A|=|B|$, s'il existe une bijection entre eux.[82]

- **Ensembles finis et dénombrables :** Un ensemble est **fini** s'il est en bijection avec $\{1,2,\ldots,n\}$ pour un certain $n\in N$. Un ensemble est **dénombrable** s'il est en bijection avec l'ensemble des entiers naturels N.[84] Les ensembles Z (entiers relatifs) et Q (nombres rationnels) sont dénombrables.[82] Le cardinal de N est noté $\aleph_0$ (aleph-zéro).[84]
- **Ensembles indénombrables :** Un ensemble infini qui n'est pas dénombrable est dit **indénombrable**.

L'argument de la diagonale de Cantor

En 1891, Georg Cantor a démontré de manière révolutionnaire qu'il existe différentes "tailles" d'infini. Plus précisément, il a prouvé que l'ensemble des nombres réels R n'est pas dénombrable. Sa preuve, connue sous le nom d'**argument de la diagonale**, est une magnifique illustration du raisonnement par l'absurde.

Théorème : L'intervalle $
2. Construction : Écrivons chaque nombre de cette liste avec son développement décimal (propre, c'est-à-dire sans suite infinie de 9) :
\begin{align*} r_1 &= 0,d_{11}d_{12}d_{13}\dots \ r_2 &= 0,d_{21}d_{22}d_{23}\dots \ r_3 &= 0,d_{31}d_{32}d_{33}\dots \ & \vdots \end{align*}
Nous allons construire un nouveau nombre x∈[qui ne peut pas figurer dans cette liste. Pour chaque i≥1, définissons la i-ème décimale de x, notée xi, de la manière suivante :

xi={12si dii=1si dii=1

Ce nombre x=0,x1x2x3… est bien un réel dans [.
3. Contradiction : Par construction, le nombre x diffère de chaque nombre rn de la liste au moins à la n-ième décimale (xn=dnn). Par conséquent, x ne peut être égal à aucun rn de la liste.
4. Conclusion : Nous avons construit un nombre x∈[qui n'est pas dans la liste supposée exhaustive de tous les nombres de cet intervalle. C'est une contradiction. L'hypothèse initiale est donc fausse : [n'est pas dénombrable.86
Un argument similaire montre que pour tout ensemble S, son ensemble des parties P(S) a une cardinalité strictement supérieure à celle de S (|S|<|P(S)|).[89] Cela implique une hiérarchie infinie de cardinalités :

|N|<|P(N)|<|P(P(N))|<….

Implications pour la calculabilité

Cette distinction entre le dénombrable et l'indénombrable a une conséquence profonde en informatique.

- L'ensemble de tous les programmes informatiques possibles (dans un langage donné) est **dénombrable**. On peut en effet lister tous les programmes par taille croissante, puis par ordre lexicographique.
- Cependant, l'ensemble de toutes les fonctions possibles de N dans N est **indénombrable**, car il est en bijection avec P(N×N), qui a la même cardinalité que R.

Puisqu'il y a "plus" de fonctions que de programmes, il doit nécessairement exister des fonctions qui ne peuvent être calculées par aucun programme. Ces fonctions sont dites **non-calculables**.[91]

L'exemple le plus célèbre de problème non-calculable (ou indécidable) est le **problème de l'arrêt** : il n'existe aucun programme universel capable de déterminer, pour n'importe quel autre programme et n'importe quelle entrée, si ce programme finira par s'arrêter ou s'il bouclera indéfiniment.[93] La preuve de ce résultat utilise également un argument diagonal.

5.2 Suites Récurrentes et Fonctions Génératrices

Les relations de récurrence sont un outil puissant pour modéliser des problèmes dont la solution pour une taille n dépend des solutions pour des tailles plus petites. Elles sont omniprésentes dans l'analyse d'algorithmes récursifs.

- Relations de Récurrence Linéaires Homogènes à Coefficients Constants :
  Une suite (un) est définie par une telle relation si chaque terme est une combinaison linéaire des termes précédents :

  $u_n = c_1 u_{n-1} + c_2 u_{n-2} + \cdots + c_k u_{n-k}$

  La méthode de résolution consiste à former l'équation caractéristique associée :

  $r^k - c_1 r^{k-1} - c_2 r^{k-2} - \cdots - c_k = 0$

  La solution générale de la récurrence est une combinaison linéaire de termes basés sur les racines de cette équation.94
  - **Racines réelles distinctes r1,…,rk :** La solution est de la forme $u_n = \lambda_1 r_1^n + \cdots + \lambda_k r_k^n$.
  - **Racine réelle r1 de multiplicité m :** La solution contient les termes $(\lambda_0 + \lambda_1 n + \cdots + \lambda_{m-1} n^{m-1}) r_1^n$.
  - **Racines complexes conjuguées $\rho e^{\pm i\theta}$:** La solution contient des termes de la forme $\rho^n(\lambda_1 \cos(n\theta) + \lambda_2 \sin(n\theta))$.[94]

  Les constantes λ_i sont déterminées par les conditions initiales de la suite.
- Fonctions Génératrices :
  Une fonction génératrice est un moyen d'encoder une suite infinie de nombres $(a_n)_{n \geq 0}$ comme les coefficients d'une série formelle.96 La fonction génératrice ordinaire est définie par :
  $G(x) = \sum_{n=0}^{\infty} a_n x^n$

  Cette approche transforme un problème discret (une relation de récurrence sur une suite) en un problème d'analyse ou d'algèbre (une équation sur une fonction). Pour résoudre une récurrence, on suit généralement ces étapes 97 :
  1. Traduire la relation de récurrence en une équation algébrique pour sa fonction génératrice G(x).
  2. Résoudre cette équation pour obtenir une forme close pour G(x).
  3. Développer G(x) en série de Taylor pour extraire le coefficient de x^n, qui est le terme a_n recherché.
     Cette méthode est particulièrement puissante pour des récurrences plus complexes, y compris non homogènes.98

5.3 Introduction aux Probabilités Discrètes

La théorie des probabilités fournit un cadre formel pour raisonner sur l'incertitude. Dans le contexte discret, elle est

essentielle pour l'analyse en cas moyen des algorithmes, la modélisation de systèmes stochastiques et les fondements de l'apprentissage automatique.

- **Espace Probabilisé Discret :** Un espace probabilisé est un triplet (Ω,F,P) où :
  - Ω est l'**univers**, l'ensemble (fini ou dénombrable) de tous les résultats possibles d'une expérience.[100]
  - F est la **tribu** des événements, qui dans le cas discret est généralement l'ensemble des parties P(Ω).[102]
  - P est une **mesure de probabilité**, une fonction $P: \mathcal{F} \to $ qui assigne une probabilité à chaque événement, telle que P(Ω)=1 et pour toute collection d'événements disjoints (Ai), P(∪Ai)=∑P(Ai).[101]
- **Probabilité Conditionnelle et Indépendance :**
  - La probabilité conditionnelle de l'événement A sachant que l'événement B s'est produit (avec P(B)>0) est définie par :

    P(A|B)=P(B)P(A∩B)
  - Deux événements A et B sont **indépendants** si la réalisation de l'un n'influence pas la probabilité de l'autre, c'est-à-dire P(A∩B)=P(A)P(B). Si P(B)>0, cela équivaut à P(A|B)=P(A).[104]
- Théorème de Bayes : Ce théorème fondamental permet d' "inverser" les probabilités conditionnelles et de mettre à jour nos croyances à la lumière de nouvelles données.

  P(A|B)=P(B)P(B|A)P(A)

  En utilisant la formule des probabilités totales pour décomposer P(B), on obtient la forme étendue :

  P(A|B)=P(B|A)P(A)+P(B|¬A)P(¬A)P(B|A)P(A)

  Ce théorème est la pierre angulaire de l'inférence bayésienne.106
- Variables Aléatoires Discrètes :
  Une variable aléatoire X est une fonction de Ω vers R. Elle est discrète si son image X(Ω) est un ensemble fini ou dénombrable.109
  - Espérance : L'espérance E[X] est la valeur moyenne de X, pondérée par les probabilités.

    E[X]=x∈X(Ω)∑x·P(X=x)

    L'espérance est linéaire : E=aE[X]+bE.110
  - Variance : La variance V(X) mesure la dispersion de X autour de son espérance.

    V(X)=E[(X−E[X])2]

    La formule de Koenig-Huygens fournit un moyen de calcul pratique : V(X)=E[X2]−(E[X])2.110
- **Analyse en Cas Moyen :** L'espérance est l'outil central pour l'analyse en cas moyen d'un algorithme. Si C(i) est le coût de l'algorithme sur une entrée i et que les entrées sont tirées selon une certaine distribution de probabilité, alors le coût moyen est l'espérance de la variable aléatoire C.[52]

5.4 Étude de Cas – Cryptographie à Clé Publique (RSA)

L'algorithme de chiffrement RSA est une application spectaculaire qui synthétise de nombreux concepts abordés dans ce chapitre. Il illustre comment des propriétés profondes de la théorie des nombres peuvent être exploitées pour construire un système informatique complexe dont la sécurité est cruciale dans notre monde numérique.

- **Contexte :** La cryptographie asymétrique (ou à clé publique) utilise une paire de clés : une **clé publique** pour chiffrer les messages, et une **clé privée** pour les déchiffrer. La clé publique peut être partagée librement, tandis que la clé privée doit rester secrète. La sécurité repose sur le fait qu'il est "calculatoirement infaisable" de déduire la clé privée à partir de la clé publique.[115]
- **Génération des Clés :**
  1. Choisir deux très grands nombres premiers distincts, p et q.
  2. Calculer le module $n=pq$.
  3. Calculer l'indicatrice d'Euler $\phi(n)=(p-1)(q-1)$.
  4. Choisir un exposant de chiffrement e tel que $1<e<\phi(n)$ et $pgcd(e,\phi(n))=1$.
  5. Calculer l'exposant de déchiffrement d comme l'inverse modulaire de e modulo $\phi(n)$, c'est-à-dire $d \equiv e^{-1} (mod \phi(n))$. Ce calcul est réalisé efficacement grâce à l'**algorithme d'Euclide étendu**.[116]
  - **Clé publique :** (n,e).
  - **Clé privée :** (n,d).
- Chiffrement et Déchiffrement :
  Un message, représenté par un entier $M<n$, est chiffré en calculant :

  $C \equiv M^e (mod n)$

  Le message chiffré C est déchiffré en calculant :

  $M' \equiv C^d (mod n)$
- Justification de la Correction : Nous devons montrer que $M'=M$.

  $M' \equiv C^d \equiv (M^e)^d \equiv M^{ed} (mod n)$

  Par construction de d, on a $ed \equiv 1 (mod \phi(n))$, donc il existe un entier k tel que $ed=1+k\phi(n)$.

  $M^{ed}=M^{1+k\phi(n)}=M \cdot (M^{\phi(n)})^k$

  D'après le théorème d'Euler, si $pgcd(M,n)=1$, alors $M^{\phi(n)} \equiv 1 (mod n)$.
  Donc, $M^{ed} \equiv M \cdot 1^k \equiv M (mod n)$.
  Le cas où pgcd(M,n)=1 se traite séparément mais mène à la même conclusion.116 Le déchiffrement fonctionne donc correctement.
- **Sécurité :** La sécurité de RSA repose sur l'hypothèse que la **factorisation** de grands nombres est un problème calculatoirement difficile. Un attaquant qui connaît la clé publique (n,e) pourrait trouver la clé privée d s'il pouvait calculer $\phi(n)=(p-1)(q-1)$. Pour cela, il lui faudrait connaître p et q, c'est-à-dire factoriser n. À ce jour, aucun algorithme classique efficace n'est connu pour factoriser de très grands nombres.[116]

RSA est ainsi un exemple paradigmatique de la manière dont les structures discrètes (l'arithmétique modulaire sur Z/nZ), la théorie des nombres (théorème d'Euler, nombres premiers) et l'algorithmique (algorithme d'Euclide étendu, exponentiation modulaire) s'unissent pour créer un système complexe dont les propriétés (correction, sécurité) peuvent être analysées et prouvées formellement.

Conclusion

Ce chapitre a parcouru les paysages fondamentaux des mathématiques discrètes, depuis les notions élémentaires d'ensembles jusqu'aux applications sophistiquées en cryptographie. Nous avons établi que les structures discrètes ne sont pas une simple collection de sujets disparates, mais un système conceptuel cohérent et profondément interconnecté, formant le socle théorique sur lequel repose l'informatique moderne.

La théorie des ensembles nous a fourni un langage universel pour décrire les collections d'objets, dont les opérations se sont révélées être le reflet direct des opérateurs de la logique booléenne. En structurant ces ensembles avec des relations, nous avons vu émerger des concepts puissants comme l'équivalence, qui permet de classifier, et l'ordre, qui permet de hiérarchiser. L'analyse combinatoire, fondée sur les principes simples de la somme et du produit, nous a donné les moyens de dénombrer des configurations complexes, une compétence essentielle pour évaluer l'efficacité des algorithmes.

L'exploration de l'arithmétique a démontré que les propriétés des entiers, loin d'être purement abstraites, sont au cœur d'algorithmes concrets et efficaces, comme l'algorithme d'Euclide étendu, qui transforme des théorèmes d'existence en outils de calcul pratiques. Enfin, l'introduction à la cardinalité, à la calculabilité, aux récurrences et aux probabilités discrètes a ouvert la voie à l'analyse de systèmes complexes, où il ne s'agit plus seulement de construire des algorithmes, mais de prouver leur correction, d'analyser leur performance moyenne et de garantir leur sécurité face à des adversaires. L'étude de cas de l'algorithme RSA a cristallisé cette synergie, montrant comment un système de sécurité mondial repose sur l'interaction subtile entre ces différentes briques fondamentales.

Les concepts développés dans ce chapitre sont des prérequis essentiels pour la suite de cet ouvrage. La théorie des graphes, qui sera abordée prochainement, peut être vue comme l'étude de relations binaires sur des ensembles finis. La logique formelle et la théorie de la calculabilité approfondiront les liens entre les ensembles, la logique et les limites intrinsèques du calcul. La théorie de l'information et l'apprentissage automatique s'appuieront massivement sur les fondements des probabilités discrètes ici esquissés. Ainsi, les structures discrètes et la combinatoire ne sont pas seulement un chapitre des mathématiques pour l'informatique ; elles en sont le système d'exploitation.

Ouvrages cités

1. MAT-1310 Mathématiques discrètes - Cours - Université Laval, dernier accès : septembre 21, 2025, https://www.ulaval.ca/etudes/cours/mat-1310-mathematiques-discretes
2. Mathématiques discrètes - Admission - Université de Montréal, dernier accès : septembre 21, 2025, https://admission.umontreal.ca/cours-et-horaires/cours/mat-1500/
3. ensemble.pdf, dernier accès : septembre 21, 2025, https://perso.univ-rennes1.fr/marie-pierre.lebaud/ma2/pdf/ensemble.pdf
4. Chapitre 1 Ensembles et sous-ensembles - Université de Rennes, dernier accès : septembre 21, 2025,

https://perso.univ-rennes1.fr/laurent.moret-bailly/docpedag/polys/MA2.pdf
5. Chapitre 2 : ensembles - Ceremade, dernier accès : septembre 21, 2025, https://www.ceremade.dauphine.fr/~viossat/PDFs/algebre1/2009-10/ensembles09-10.pdf
6. 2 Théorie naive des ensembles, dernier accès : septembre 21, 2025, https://www.lri.fr/~paulin/MathInfo/html/cours004.html
7. Notions de base et notations courantes en mathématiques - EPFL, dernier accès : septembre 21, 2025, https://www.epfl.ch/labs/anchp/wp-content/uploads/2018/05/NotionsDeBases-1.pdf
8. Calcul ensembliste, fonctions, applications 1 Ensembles - St-Etienne, dernier accès : septembre 21, 2025, https://perso.univ-st-etienne.fr/rool6510/L3-info-cours.pdf
9. Démonstrations [Langage des ensembles], dernier accès : septembre 21, 2025, https://uel.unisciel.fr/mathematiques/logique1/logique1_ch02/co/apprendre_ch2_03.html
10. Concepts et notations de la théorie des ensembles Le cours va commencer de façon bien abstraite, par une énumér, dernier accès : septembre 21, 2025, https://math.univ-lyon1.fr/~caldero/cours.pdf
11. Ensemble des parties - BibM@th, dernier accès : septembre 21, 2025, https://www.bibmath.net/dico/index.php?action=affiche&quoi=./e/ensdesparties.html
12. Nombre de parties d'un ensemble fini et permutations - myMaxicours, dernier accès : septembre 21, 2025, https://www.maxicours.com/se/cours/nombre-de-parties-d-un-ensemble-fini-et-permutations/
13. Résumé de cours : ensembles, applications, relations - BibM@th, dernier accès : septembre 21, 2025, https://www.bibmath.net/ressources/index.php?action=affiche&quoi=mathsup/cours/ensembleapplicationrelation.html
14. Ensembles de mots : langages, dernier accès : septembre 21, 2025, https://pages.lip6.fr/Jean-Francois.Perrot/inalco/Automates/Cours2.html
15. Propriétés de distributivité [Langage des ensembles], dernier accès : septembre 21, 2025, https://uel.unisciel.fr/mathematiques/logique1/logique1_ch02/co/apprendre_ch2_07.html
16. Ensembles et applications - Exo7 - Cours de mathématiques, dernier accès : septembre 21, 2025, http://exo7.emath.fr/cours/ch_ensembles.pdf
17. DEMONSTRATION DU THEOREME DE MORGAN - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=s3W8vPwdf20
18. Morgan's Law. BTS SIO Demonstration - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=j90-FYIn8Is
19. Cours d'Algèbre 1 : Chapitre 3, Relations binaires sur un ensemble, dernier accès : septembre 21, 2025, https://dmath.univ-tlemcen.dz/assets/uploads/Documents/2024-2025/Licence/1er_annee_Info/Alg%C3%A8bre1_L1-Info/Chapitre3_Alg%C3%A8bre1_24_25_Fr_Bis.pdf
20. MAT 1101 Mathématiques fondamentales hiver 2017, dernier accès : septembre 21, 2025, https://dms.umontreal.ca/~mat1101/notes/H18/chap1.pdf
21. Relations binaires : définition et propriétés. - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=ptAvL1V00M8
22. Relations binaires antisymétriques - IGM, dernier accès : septembre 21, 2025, https://igm.univ-mlv.fr/~giraudo/Old/Enseignements/Enseignements2020-2022/MD/Ancien/2020-2021/CM_2020-11-09.pdf
23. Concepts de base en arithmétique, dernier accès : septembre 21, 2025, https://maths-olympiques.fr/wp-content/uploads/2017/09/arith_base.pdf
24. Relation antisymétrique - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Relation_antisym%C3%A9trique
25. Relation binaire : définition et explications - Techno-Science, dernier accès : septembre 21, 2025, https://www.techno-science.net/definition/6463.html
26. Relation d'équivalence, dernier accès : septembre 21, 2025,

https://uel.unisciel.fr/mathematiques/logique1/logique1_ch06/co/apprendre_ch6_02.html
27. ENSEMBLES, APPLICATIONS ET RELATIONS BINAIRES, dernier accès : septembre 21, 2025, https://sc-mi.univ-batna2.dz/sites/default/files/mi/files/ensembles_application_et_relations_binaires.pdf
28. 6.3: Equivalence Relations and Partitions - Mathematics LibreTexts, dernier accès : septembre 21, 2025, https://math.libretexts.org/Courses/Monroe_Community_College/MTH_220_Discrete_Math/6%3A_Relations/6.3%3A_Equivalence_Relations_and_Partitions
29. Equivalence Relations, dernier accès : septembre 21, 2025, https://www.cs.sfu.ca/~ggbaker/zju/math/equiv-rel.html
30. Equivalence relation - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Equivalence_relation
31. Equivalence Relations and Partitions, dernier accès : septembre 21, 2025, https://courses.grainger.illinois.edu/cs173/sp2010/Lectures/lect_35.pdf
32. 5 Équivalence et Ordres, dernier accès : septembre 21, 2025, https://www.lri.fr/~paulin/MathInfo/html/cours007.html
33. 7.4: Partial and Total Ordering - Mathematics LibreTexts, dernier accès : septembre 21, 2025, https://math.libretexts.org/Bookshelves/Combinatorics_and_Discrete_Mathematics/A_Spiral_Workbook_for_Discrete_Mathematics_(Kwong)/07%3A_Relations/7.04%3A_Partial_and_Total_Ordering
34. Ensemble partiellement ordonné - EPFL Graph Search, dernier accès : septembre 21, 2025, https://graphsearch.epfl.ch/fr/concept/23572
35. RELATIONS - DePaul University, dernier accès : septembre 21, 2025, https://condor.depaul.edu/ntomuro/courses/400/bookslides/EppDm4_08_05.pdf
36. Partially ordered set, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Partially_ordered_set
37. Relations d'ordre - frama.io, dernier accès : septembre 21, 2025, https://alainbusser.frama.io/NSI-IREMI-974/hasse.html
38. Ensemble partiellement ordonné - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Ensemble_partiellement_ordonn%C3%A9
39. Introduction aux ensembles ordonnés et à leurs applications - BSSM, dernier accès : septembre 21, 2025, https://bssm.ulb.ac.be/data/past_conference/2013_rexhep_slides.pdf
40. Hasse Diagrams for Partially Ordered Sets | Discrete Math - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=0UqXYpAscy8
41. Hasse Diagrams - GeeksforGeeks, dernier accès : septembre 21, 2025, https://www.geeksforgeeks.org/engineering-mathematics/discrete-mathematics-hasse-diagrams/
42. Cours 2 : Relations binaires/d'équivalence/d'ordre, Diagramme de Hasse, Treillis, dernier accès : septembre 21, 2025, https://younesse.net/Maths-discretes/Cours2/
43. Diagramme de Hasse - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Diagramme_de_Hasse
44. Relation binaire - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Relation_binaire
45. Injection - Surjection - Bijection - BibM@th, dernier accès : septembre 21, 2025, https://www.bibmath.net/dico/index.php?action=affiche&quoi=./i/injection.html
46. CPGEI - P2 Correction DM 2 - Injectivité et surjectivité pour des applications quelconques - de la licence Math, dernier accès : septembre 21, 2025, https://licence-math.univ-lyon1.fr/lib/exe/fetch.php?media=pmi:dm2-correction.pdf
47. Bijection, injection and surjection - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Bijection,_injection_and_surjection
48. 4.2Injective, surjective and bijective functions - SIUE, dernier accès : septembre 21, 2025,

https://www.siue.edu/~jloreau/courses/math-223/notes/sec-injective-surjective.html
49. Injectivité, surjectivité et bijectivité - Département de mathématiques et applications, dernier accès : septembre 21, 2025, https://www.math.ens.psl.eu/~pgervais/Documents/2019/MPInt1/TD3.5.pdf
50. Inverse function - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Inverse_function
51. Composition and Inverse Functions, dernier accès : septembre 21, 2025, https://saylordotorg.github.io/text_intermediate-algebra/s10-01-composition-and-inverse-functi.html
52. Probabilités, Statistiques, Combinatoire - LaBRI, dernier accès : septembre 21, 2025, https://www.labri.fr/perso/duchon/Enseignements/Probas/cours1-2.pdf
53. Combinations and Permutations - Math is Fun, dernier accès : septembre 21, 2025, https://www.mathsisfun.com/combinatorics/combinations-permutations.html
54. Permutation and Combination Formula - BYJU'S, dernier accès : septembre 21, 2025, https://byjus.com/permutations-and-combinations-formulas/
55. Coefficients binomiaux et loi de Pascal - myMaxicours, dernier accès : septembre 21, 2025, https://www.maxicours.com/se/cours/coefficients-binomiaux-et-loi-de-pascal/
56. How to Derive the Formula for Combinations - ThoughtCo, dernier accès : septembre 21, 2025, https://www.thoughtco.com/derive-the-formula-for-combinations-3126262
57. Permutation and Combination - Definition, Formulas, Derivation ..., dernier accès : septembre 21, 2025, https://www.cuemath.com/data/permutations-and-combinations/
58. Formule du binôme de Newton et démonstration – Démos Maths MPSI - Share, dernier accès : septembre 21, 2025, https://share.miple.co/content/mvaCWx45g3XzR
59. Coefficients binomiaux - Triangle de Pascal - Mathraining, dernier accès : septembre 21, 2025, https://www.mathraining.be/chapters/38/theories/126
60. Coefficient binomial - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Coefficient_binomial
61. Formule de binôme de Newton, démonstration 1 - Les Bons Profs, dernier accès : septembre 21, 2025, https://www.lesbonsprofs.com/nos-cours/terminale/maths-expertes/formule-de-binome-de-newton-demonstration-1-159
62. Théorème (formule du binôme de Newton) : Démonstration :, dernier accès : septembre 21, 2025, http://ece.couperin.free.fr/ece1/cours/formule_binome.pdf
63. Formule du Binôme de Newton : la démonstration par récurrence expliquée ! - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=bDReglXusAI
64. Démonstration par récurrence du binôme de Newton - Statistiques, dernier accès : septembre 21, 2025, http://www.jybaudot.fr/Maths/binomedemo.html
65. Arithmétique des entiers - BibM@th, dernier accès : septembre 21, 2025, https://www.bibmath.net/ressources/index.php?action=affiche&quoi=capes/cours/arithm.html
66. Théorème fondamental de l'arithmétique - Gerard Villemin, dernier accès : septembre 21, 2025, http://villemin.gerard.free.fr/ThNbCo01/ThFoDemo.htm
67. Arithmétique, dernier accès : septembre 21, 2025, https://www.dms.umontreal.ca/~mat1101/notes/H18/chap2.pdf
68. Le théorème fondamental de l'arithmétique - Terminale - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=mCQWYTLwYZc
69. Démonstrations capes - Arithmétique - Bibmath.net, dernier accès : septembre 21, 2025, https://www.bibmath.net/ressources/index.php?action=affiche&quoi=capes/demos/arithmetique.html
70. 3 Congruences and Congruence Equations, dernier accès : septembre 21, 2025, https://www.math.uci.edu/~ndonalds/math180a/3congruence.pdf
71. Dr. Z.'s Number Theory Lecture 10 Handout: Linear Congruences and Modular Inverse By Doron

Zeilberger, dernier accès : septembre 21, 2025, https://sites.math.rutgers.edu/~zeilberg/numtheory/L10.pdf
72. 4.5: Linear Congruences - Mathematics LibreTexts, dernier accès : septembre 21, 2025, https://math.libretexts.org/Courses/Mount_Royal_University/Higher_Arithmetic/4%3A_Greatest_Common_Divisor_least_common_multiple_and_Euclidean_Algorithm/4.5%3A_Linear_Congruences
73. Solving Congruences, dernier accès : septembre 21, 2025, https://www.math.uh.edu/~pwalker/3336Sp21Sec4.4Slides.pdf
74. Algorithme d'Euclide étendu, Théorème de Bézout – IN310 - Luca De Feo, dernier accès : septembre 21, 2025, http://defeo.lu/in310/poly/euclide-bezout/
75. L'algorithme d'Euclide étendu - Bibm@th, dernier accès : septembre 21, 2025, https://www.bibmath.net/crypto/index.php?action=affiche&quoi=complements/algoeuclid
76. fr.wikipedia.org, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Algorithme_d%27Euclide_%C3%A9tendu#:~:text=En%20math%C3%A9matiques%2C%20l'algorithme%20d,PGCD(a%2C%20b).
77. Algorithme d'Euclide étendu - Maths-cours.fr, dernier accès : septembre 21, 2025, https://www.maths-cours.fr/methode/algorithme-euclide-etendu
78. Petit théorème de Fermat - BibM@th, dernier accès : septembre 21, 2025, https://www.bibmath.net/dico/index.php?action=affiche&quoi=./f/fermatpetit.html
79. Petit théorème de Fermat - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Petit_th%C3%A9or%C3%A8me_de_Fermat
80. Théorème d'Euler-Fermat | Mathraining, dernier accès : septembre 21, 2025, https://www.mathraining.be/chapters/5/all
81. Théorème d'Euler (arithmétique) - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_d%27Euler_(arithm%C3%A9tique)
82. Cardinalité - Luca De Feo, dernier accès : septembre 21, 2025, http://defeo.lu/in310-2013/Cardinalit%C3%A9
83. Ensembles dénombrables, dernier accès : septembre 21, 2025, https://www.math.univ-toulouse.fr/~jroyer/TD/2018-19-L2PS/ChA-Denombrabilite.pdf
84. Ensemble dénombrable - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Ensemble_d%C3%A9nombrable
85. Comment savoir si un ensemble est fini, infini, infini dénombrable ou infiniment indénombrable? - Quora, dernier accès : septembre 21, 2025, https://fr.quora.com/Comment-savoir-si-un-ensemble-est-fini-infini-infini-d%C3%A9nombrable-ou-infiniment-ind%C3%A9nombrable
86. Procédé diagonal de Cantor - Bibm@th, dernier accès : septembre 21, 2025, https://www.bibmath.net/dico/index.php?action=affiche&quoi=./p/procdiag.html
87. Argument de la diagonale de Cantor : définition et explications - Techno-Science, dernier accès : septembre 21, 2025, https://www.techno-science.net/definition/6441.html
88. Argument de la diagonale de Cantor — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Argument_de_la_diagonale_de_Cantor
89. Cantor's Diagonal Argument - Jeremy L. Martin, dernier accès : septembre 21, 2025, https://jlmartin.ku.edu/courses/math410-S09/cantor.pdf
90. Cantor's diagonal argument - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Cantor%27s_diagonal_argument
91. Cours Logique et Calculabilité, dernier accès : septembre 21, 2025, https://pageperso.lis-lab.fr/kevin.perrot/documents/2016/calculabilite/Cours_16.pdf
92. Théorie de la calculabilité - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Th%C3%A9orie_de_la_calculabilit%C3%A9

93. Calculabilité, décidabilité - Informatique au lycée, dernier accès : septembre 21, 2025, http://nsinfo.yo.fr/nsi_term_programmation_calculabilite.html
94. Suite récurrente linéaire — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Suite_r%C3%A9currente_lin%C3%A9aire
95. Formulaire - Suites récurrentes linéaires - BibM@th, dernier accès : septembre 21, 2025, https://www.bibmath.net/formulaire/index.php?action=affiche&quoi=suitereclin
96. Fonctions génératrices: Séries, Calcul - StudySmarter, dernier accès : septembre 21, 2025, https://www.studysmarter.fr/resumes/mathematiques/mathematiques-discretes/fonctions-generatrices/
97. Complexité : Récurrence, dernier accès : septembre 21, 2025, https://imss-www.upmf-grenoble.fr/prevert/Prog/Complexite/recurrence.html
98. www.studysmarter.fr, dernier accès : septembre 21, 2025, https://www.studysmarter.fr/resumes/mathematiques/mathematiques-discretes/fonctions-generatrices/#:~:text=Comment%20les%20fonctions%20g%C3%A9n%C3%A9ratrices%20r%C3%A9solvent,lin%C3%A9aires%20ou%20d'ordre%20sup%C3%A9rieur.
99. Plan - Introduction à la théorie de l'informatique, dernier accès : septembre 21, 2025, https://people.montefiore.uliege.be/geurts/Cours/iti/2012/08-f-generatrices-part2-2012-2013.pdf
100. Espaces probabilisés 1 Définition 2 Propriétés élémentaires des probabilités - Mathieu Mansuy, dernier accès : septembre 21, 2025, http://www.mathieu-mansuy.fr/pdf/MM0804-cours1.pdf
101. Probabilités discrètes, dernier accès : septembre 21, 2025, https://helios2.mi.parisdescartes.fr/~jdelon/enseignement/cogmaster/cogmaster_probas_discretes.pdf
102. Probabilités discrètes - math, dernier accès : septembre 21, 2025, https://maths.ac-noumea.nc/sites/maths.ac-noumea.nc/IMG/pdf/cours-2.pdf
103. Espaces probabilisés discrets, dernier accès : septembre 21, 2025, https://cahier-de-prepa.fr/mp-corneille/download?id=1395
104. Indépendance en probabilités - BibM@th, dernier accès : septembre 21, 2025, https://www.bibmath.net/dico/index.php?action=affiche&quoi=./i/independantproba.html
105. 3. Probabilité conditionelle et indépendence - GitHub Pages, dernier accès : septembre 21, 2025, https://matthwilhelm.github.io/ProbaStat/Probabilit%C3%A9/probabilite_conditionnelle_independance.html
106. Théorème de Bayes [Introduction à la statistique bayésienne pour les ingénieurs et les médecins] - Université de Strasbourg, dernier accès : septembre 21, 2025, https://dun.unistra.fr/ipm/unit/bayesien/co/1_2_3.html
107. Théorème de Bayes - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_de_Bayes
108. Formule de Bayes - Formule des probabilités totales. - IREM, dernier accès : septembre 21, 2025, http://docs.irem.univ-paris-diderot.fr/up/B101/Bayes_Maths_comp.pdf
109. Résumé de cours : variables aléatoires discrètes - BibM@th, dernier accès : septembre 21, 2025, https://www.bibmath.net/ressources/index.php?action=affiche&quoi=mathspe/cours/vadiscrete.html
110. Variables aléatoires discrètes - Mathieu Mansuy, dernier accès : septembre 21, 2025, http://www.mathieu-mansuy.fr/pdf/ECG2-Chapitre5.pdf
111. Chapitre 20 : Variables aléatoires discrètes, dernier accès : septembre 21, 2025, https://cahier-de-prepa.fr/ecg1-joffre/download?id=331
112. Espérance et variance, dernier accès : septembre 21, 2025, https://perso.univ-rennes1.fr/eric.jourdain/AP2/Chapitre5.pdf
113. Espérance et variance, dernier accès : septembre 21, 2025, https://mathematiques.elodiebouchet.fr/wp-content/uploads/Cours-Esperance-et-variance.pdf

114. Cours 9: Introduction aux algorithmes probabilistes, dernier accès : septembre 21, 2025, https://www.di.ens.fr/~busic/cours/LI325/slidesCAAC9_1213.pdf
115. Chiffrement et algorithme RSA : tout comprendre à Rivest Shamir Adleman - Splunk, dernier accès : septembre 21, 2025, https://www.splunk.com/fr_fr/blog/learn/rsa-algorithm-cryptography.html
116. Chiffrement RSA — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Chiffrement_RSA
117. Grands nombres premiers Cryptographie RSA - Laboratoire de Mathématiques d'Orsay - Université Paris-Saclay, dernier accès : septembre 21, 2025, https://www.imo.universite-paris-saclay.fr/~joel.merker/Enseignement/Algebre-effective/rsa.pdf
118. Cryptosystème RSA, dernier accès : septembre 21, 2025, https://www.di.ens.fr/~nitulesc/files/crypto3.pdf
119. Démystifier RSA : Comprendre l'algorithme de chiffrement RSA - simeononsecurity, dernier accès : septembre 21, 2025, https://fr.simeononsecurity.com/articles/understanding-the-rsa-cipher-algorithm/

Chapitre 3 : Théorie des Graphes

3.1 Introduction : D'un Problème de Ponts à une Théorie Universelle

La théorie des graphes, aujourd'hui un pilier des mathématiques discrètes et de l'informatique théorique, trouve son origine dans une énigme populaire du 18ème siècle. En 1736, la ville de Königsberg en Prusse (aujourd'hui Kaliningrad, Russie) était divisée en quatre zones terrestres — deux rives et deux îles — reliées par sept ponts.[1] Les habitants se demandaient s'il était possible d'effectuer une promenade qui traverse chaque pont une et une seule fois.[3] Ce problème, en apparence anecdotique, a résisté à toutes les tentatives de résolution jusqu'à ce qu'il attire l'attention du mathématicien suisse Leonhard Euler.

L'approche d'Euler fut révolutionnaire, non pas tant par la réponse qu'il apporta, mais par la méthode qu'il employa pour l'obtenir. Il comprit que les détails géographiques — la taille des îles, la longueur des ponts, la configuration exacte des rues — étaient superflus. La seule information pertinente était la manière dont les zones étaient connectées les unes aux autres.[1] Il réalisa alors une abstraction radicale : il représenta chaque zone terrestre par un point (un **sommet**) et chaque pont par une ligne reliant deux points (une **arête**).[3] Ce faisant, il transforma un problème de topographie en un problème de structure pure, donnant naissance au premier graphe de l'histoire.[4] Cet acte d'abstraction, qui consiste à modéliser un système complexe en ne conservant que ses relations structurelles, est aujourd'hui au cœur de la démarche en sciences et en génie informatiques.[1]

Une fois le problème modélisé, la solution d'Euler devint d'une clarté limpide. Il remarqua que pour qu'un tel parcours soit possible, tout sommet servant d'étape intermédiaire (c'est-à-dire qui n'est ni le point de départ ni le point d'arrivée de la promenade) doit avoir un nombre pair d'arêtes. En effet, chaque fois que l'on entre dans une zone par un pont, on doit pouvoir en ressortir par un autre pont.[5] Or, dans le graphe de Königsberg, les quatre sommets avaient un nombre impair de ponts (degrés 5, 3, 3 et 3).[1] Euler put ainsi prouver mathématiquement, sans avoir à tester toutes les promenades possibles, qu'un tel parcours était impossible.[3] Cette preuve est remarquable car elle s'applique directement à la réalité physique des ponts et fournit une explication sur la raison de cette impossibilité.[3]

La contribution d'Euler ne s'est pas limitée à une simple réponse négative. Il a généralisé son raisonnement pour établir une condition nécessaire et suffisante qui caractérise l'existence de tels chemins, connus aujourd'hui sous le nom de chemins eulériens.[1] Il a ainsi transformé la solution d'un cas particulier en un théorème universel, illustrant une démarche scientifique fondamentale. Ce qui a débuté comme une énigme récréative est devenu un cadre mathématique d'une portée immense, capable de modéliser des relations et des connexions dans une multitude de systèmes complexes.[4] Aujourd'hui, la théorie des graphes est omniprésente : elle est le langage des réseaux de communication et des réseaux sociaux, l'outil de la logistique et de la planification urbaine, le fondement d'algorithmes en bio-informatique pour le séquençage du génome, en conception de circuits électroniques, et même dans les neurosciences pour modéliser la connectivité du cerveau.[3]

3.2 Fondements et Vocabulaire Formel des Graphes

Pour étudier les graphes de manière rigoureuse, il est indispensable d'établir un vocabulaire formel et précis. Cette section introduit les définitions fondamentales qui constituent le langage de la théorie des graphes.

3.2.1 Définitions Formelles

Un graphe est une structure abstraite composée de deux types d'éléments : des sommets et des liaisons entre ces sommets.

- **Graphe Non Orienté** : Un graphe non orienté G est formellement défini comme un couple G=(S,A), où S est un ensemble fini et non vide dont les éléments sont appelés **sommets** (ou nœuds), et A est un ensemble de paires de sommets {u,v} avec u,v∈S, appelées **arêtes**.[11] L'
  ordre d'un graphe est le cardinal de son ensemble de sommets, noté |S|, et sa **taille** est le cardinal de son ensemble d'arêtes, noté |A|.[11]
- **Graphe Orienté (ou Digraphe)** : Un graphe orienté G est un couple G=(S,A), où S est un ensemble fini de sommets et A est un ensemble de couples ordonnés de sommets (u,v), appelés **arcs**.[4] Pour un arc
  (u,v), le sommet u est appelé l'**origine** (ou la queue) et le sommet v est l'**extrémité** (ou la tête) de l'arc.[11]

3.2.2 Taxonomie des Graphes

La nature des arêtes et la structure globale permettent de classer les graphes en différentes familles.

- **Graphe Simple** : Un graphe est dit simple s'il ne contient ni **boucle** (une arête ou un arc reliant un sommet à lui-même), ni arêtes multiples entre une même paire de sommets.[6] Sauf indication contraire, les graphes considérés dans ce chapitre seront simples.
- **Multigraphe et Pseudo-graphe** : Un multigraphe autorise l'existence de plusieurs arêtes (ou arcs) entre deux sommets. Un pseudo-graphe est un multigraphe qui autorise également les boucles.[11] Le graphe modélisant le problème des ponts de Königsberg est un multigraphe.
- **Sous-graphe et Graphe Partiel** : Un graphe H=(S’,A’) est un **sous-graphe** de G=(S,A) si S’⊆S et A’⊆A.[11] Si
  S’=S, H est un **graphe partiel** de G.[11]
- **Graphe Pondéré (ou Valué)** : Un graphe pondéré est un triplet G=(S,A,w), où (S,A) est un graphe et w:A→R est une fonction qui associe un **poids** (ou coût, ou valeur) à chaque arête ou arc.[15]

3.2.3 Propriétés des Sommets et Arêtes

- **Adjacence et Incidence** : Dans un graphe (orienté ou non), deux sommets u et v sont dits **adjacents** s'il existe une arête ou un arc les reliant. Une arête a={u,v} est dite **incidente** aux sommets u et v.[11]
- **Degré d'un Sommet** :
  - Dans un graphe non orienté, le **degré** d'un sommet v, noté d(v) ou deg(v), est le nombre d'arêtes qui lui sont incidentes. Une boucle est conventionnellement comptée pour 2 dans le calcul du degré.[4] Un sommet de degré 0 est dit
    isolé ; un sommet de degré 1 est dit **pendant**.[11]
  - Dans un graphe orienté, on distingue le **degré sortant** d+(v), qui est le nombre d'arcs ayant v pour origine, et le **degré entrant** d−(v), qui est le nombre d'arcs ayant v pour extrémité.[17]
- Lemme des Poignées de Main : Dans tout graphe non orienté G=(S,A), la somme des degrés des sommets est égale au double du nombre d'arêtes :

  $\sum_{v \in S} d(v) = 2|A|$

  Ce lemme, également connu sous le nom de formule de la somme des degrés, est un premier exemple d'un invariant structurel simple mais puissant. Il impose une contrainte fondamentale sur la structure de n'importe quel graphe. La preuve repose sur un argument de double dénombrement : chaque arête {u,v} contribue exactement pour une unité au degré de u et pour une unité au degré de v. Ainsi, en sommant les degrés de tous les sommets, chaque arête est comptée précisément deux fois.4
  - **Corollaire** : Dans tout graphe non orienté, le nombre de sommets de degré impair est nécessairement pair.[4] En effet, la somme des degrés étant un nombre pair (
    $2|A|$), la somme des degrés des sommets de degré impair doit elle-même être paire, ce qui n'est possible que si le nombre de termes de cette somme est pair.

3.2.4 Familles de Graphes Remarquables

La classification des graphes en familles aux propriétés bien définies n'est pas un simple exercice de catalogage. Elle constitue une grille d'analyse essentielle pour l'étude de la complexité algorithmique. De nombreux problèmes, intraitables dans le cas général, deviennent résolubles en temps polynomial lorsqu'ils sont restreints à l'une de ces familles.

- **Graphe Complet (Kn)** : Un graphe simple non orienté d'ordre n est complet si chaque paire de sommets distincts est reliée par une arête. Un graphe Kn possède $n(n-1)/2$ arêtes et est $(n-1)$-régulier.[6]
- **Graphe Biparti** : Un graphe est biparti si son ensemble de sommets S peut être partitionné en deux sous-ensembles disjoints U et V (c'est-à-dire $S=U \cup V$ et $U \cap V=\emptyset$) de telle sorte que chaque arête du graphe relie un sommet de U à un sommet de V.[6]
- **Graphe Régulier** : Un graphe est k-régulier si tous ses sommets ont le même degré k.[6]
- **Arbre** : Un graphe connexe et acyclique. Cette famille fondamentale sera étudiée en détail dans la section 3.5.[19]

- **Graphe Planaire** : Un graphe qui peut être dessiné sur un plan sans qu'aucune de ses arêtes ne se croise. Cette famille sera étudiée en détail dans la section 3.7.[6]

3.3 Représentation Informatique des Graphes

Pour qu'un graphe puisse être traité par un algorithme, sa structure topologique doit être encodée dans une structure de données informatique. Le choix de cette représentation est une décision de conception fondamentale qui a des implications profondes sur l'efficacité en temps et en espace des algorithmes. Ce choix dépend principalement des propriétés du graphe, notamment sa densité (le rapport entre son nombre d'arêtes et le nombre maximal d'arêtes possibles), et des opérations qui seront effectuées le plus fréquemment.[22]

3.3.1 La Matrice d'Adjacence

La matrice d'adjacence est une représentation directe et intuitive des relations d'adjacence dans un graphe.

- **Définition** : Pour un graphe G=(S,A) d'ordre n, dont les sommets sont numérotés de 0 à n−1, la matrice d'adjacence est une matrice carrée M de taille n×n.
  - Pour un **graphe non valué**, le coefficient M[i][j] est égal à 1 s'il existe une arête (ou un arc) entre le sommet i et le sommet j, et 0 sinon.[12]
  - Pour un **graphe pondéré**, M[i][j] contient le poids de l'arête (i,j). Si l'arête n'existe pas, une valeur sentinelle, telle que l'infini (∞) ou NIL, est utilisée.[25]
- **Propriétés** : Dans le cas d'un graphe non orienté, la matrice d'adjacence est symétrique par rapport à sa diagonale principale (M[i][j]=M[j][i]).[13] Une propriété combinatoire remarquable est que le coefficient (i,j) de la matrice Mk (la k-ième puissance de M) correspond au nombre de chemins de longueur exactement k allant du sommet i au sommet j.[11]

3.3.2 Les Listes d'Adjacence

Les listes d'adjacence offrent une représentation plus compacte, particulièrement adaptée aux graphes peu denses.

- **Définition** : Cette structure consiste en un tableau (ou une structure associative comme un dictionnaire) de n listes. La liste associée au sommet i contient l'ensemble de ses voisins (ou successeurs dans un graphe orienté).[19]
  - Pour un **graphe pondéré**, chaque élément de la liste peut être une paire (voisin, poids) pour stocker le coût de l'arête correspondante.[25]
- **Implémentation** : En pratique, un dictionnaire (ou une table de hachage) est souvent utilisé, où les clés sont les

identifiants des sommets et les valeurs sont les listes de leurs voisins. Cette approche est flexible car elle permet de représenter des sommets par des objets arbitraires (hachables) et pas seulement par des entiers.[17]

3.3.3 Analyse Comparative

Le choix entre une matrice et des listes d'adjacence illustre un compromis fondamental en informatique : celui entre l'utilisation de la mémoire et la vitesse d'exécution de certaines opérations. La matrice d'adjacence investit massivement en mémoire (O(|S|2)) pour pré-calculer et stocker l'information de connectivité de toutes les paires de sommets possibles, ce qui garantit un accès en temps constant (O(1)) à cette information. À l'inverse, les listes d'adjacence sont économes en mémoire (O(|S|+|A|)) mais requièrent un parcours de liste, et donc un temps proportionnel au degré du sommet, pour vérifier l'existence d'une arête.

Ce choix structurel n'est pas une simple question d'implémentation ; il est intrinsèquement lié à l'efficacité asymptotique des algorithmes qui manipuleront le graphe. Par exemple, la complexité canonique des algorithmes de parcours comme BFS et DFS, O(|S|+|A|), n'est atteignable qu'avec une représentation par listes d'adjacence. Avec une matrice, l'itération des voisins de chaque sommet prendrait un temps O(|S|), menant à une complexité globale de O(|S|2), indépendamment de la densité du graphe. Le tableau suivant synthétise ce compromis.

Tableau 3.1 : Comparaison des Représentations de Graphes

Opération	Matrice d'Adjacence	Listes d'Adjacence	Analyse et Cas d'Usage
Complexité Spatiale	$O(	S	^2)$
Ajouter une arête {u,v}	O(1)	O(1)	Efficace dans les deux cas, en supposant qu'on n'a pas besoin de vérifier au préalable l'existence de l'arête.
Vérifier si {u,v} est une arête	O(1)	O(min(d(u),d(v)))	Avantage net pour la matrice. Avec les listes, il faut parcourir la liste des voisins de u (ou de v).[19]

Lister les voisins de u	$O(	S	)$
Supprimer une arête {u,v}	O(1)	O(d(u))	Similaire à la vérification d'arête.

3.4 Parcours, Chemins et Connectivité

L'exploration systématique des sommets et des arêtes d'un graphe est une opération fondamentale qui sous-tend de nombreux algorithmes. Cette section définit formellement les notions de parcours et de connectivité, et présente les deux stratégies d'exploration canoniques : le parcours en largeur et le parcours en profondeur.

3.4.1 Terminologie des Parcours

- **Chaînes et Chemins** : Dans un graphe non orienté, une **chaîne** est une séquence de sommets (s0,s1,…,sk) telle que {si−1,si} est une arête pour tout i∈{1,…,k}.[11] Dans un graphe orienté, un **chemin** est une séquence similaire où (si−1,si) est un arc.[11] La **longueur** du parcours est le nombre d'arêtes ou d'arcs, k. Un parcours est dit **simple** s'il n'emprunte jamais deux fois la même arête/arc, et **élémentaire** s'il ne visite jamais deux fois le même sommet (à l'exception possible des extrémités pour un cycle).[11]
- **Cycles et Circuits** : Une chaîne fermée (s0=sk) et simple dans un graphe non orienté est un **cycle**.[11] Un chemin fermé et simple dans un graphe orienté est un **circuit**.[13] Un graphe ne contenant aucun cycle (ou circuit) est dit **acyclique**.

3.4.2 Algorithmes de Parcours Fondamentaux

Les algorithmes de parcours en largeur (BFS) et en profondeur (DFS) représentent deux stratégies d'exploration duales, dont les propriétés structurelles distinctes les rendent adaptés à différentes classes de problèmes.

- **Parcours en Largeur (BFS - Breadth-First Search)**
  - **Principe** : Le BFS explore un graphe en "front d'onde" à partir d'un sommet source. Il visite d'abord la source, puis tous ses voisins directs (niveau 1), puis les voisins de ces voisins (niveau 2), et ainsi de suite. Cette stratégie est naturellement implémentée à l'aide d'une structure de données de type **file (FIFO - First-In, First-Out)**.[31]

- **Pseudo-code** :

```
BFS(Graphe G, Sommet s_depart):
 file = CreerFile()
 marques = CreerEnsemble()

 file.enfiler(s_depart)
 marques.ajouter(s_depart)

 tant que la file n'est pas vide:
  s_actuel = file.defiler()
  // Traiter s_actuel (ex: l'afficher)

  pour chaque voisin v de s_actuel:
   si v n'est pas dans marques:
    marques.ajouter(v)
    file.enfiler(v)
```

[31]

- **Analyse et Propriétés** : La complexité en temps est de O(|S|+|A|) avec une représentation par listes d'adjacence, car chaque sommet est enfilé et défilé une seule fois, et chaque arête est examinée une seule fois.[31] La complexité en espace est de O(|S|) dans le pire des cas (pour stocker la file).[31] La propriété la plus importante du BFS est qu'il découvre les sommets par ordre de distance croissante (en nombre d'arêtes) depuis la source. Il est donc l'algorithme de choix pour calculer les **plus courts chemins dans les graphes non pondérés**.[31]

- **Parcours en Profondeur (DFS - Depth-First Search)**
  - **Principe** : Le DFS explore une branche du graphe aussi loin que possible avant de revenir sur ses pas (backtracking) pour explorer une autre branche. Cette stratégie "plongeante" est naturellement implémentée à l'aide d'une **pile (LIFO - Last-In, First-Out)**, qui est souvent gérée implicitement par la pile d'appels d'une fonction récursive.[32]
  - **Pseudo-code (version récursive)** :

```
DFS(Graphe G):
 marques = CreerEnsemble()
 pour chaque sommet s de G:
  si s n'est pas dans marques:
   explorer(G, s, marques)

explorer(Graphe G, Sommet s, Ensemble marques):
 marques.ajouter(s)
 // Traiter s (ex: l'afficher)

 pour chaque voisin v de s:
  si v n'est pas dans marques:
   explorer(G, v, marques)
```

[36]

- **Analyse et Applications** : La complexité du DFS est également de O(|S|+|A|) en temps et O(|S|) en espace (pour la pile de récursion dans le pire cas).[35] Le DFS est un outil puissant pour la **détection de cycles**, le **tri topologique** des graphes orientés acycliques (DAGs), et sert de brique de base à des algorithmes plus sophistiqués, notamment pour l'analyse de la connectivité.[36]

3.4.3 Connectivité

La notion de connectivité décrit si et comment les sommets d'un graphe sont reliés entre eux.

- **Graphes Non Orientés** : Un graphe non orienté est dit **connexe** si, pour toute paire de sommets {u,v}, il existe une chaîne les reliant.[11] Un graphe qui n'est pas connexe est composé de plusieurs **composantes connexes**, qui sont des sous-graphes connexes maximaux.[20]
- **Graphes Orientés** : La notion de connectivité est plus nuancée en raison de la directionnalité des arcs.
  - Un graphe orienté est **faiblement connexe** si son graphe non orienté sous-jacent (obtenu en ignorant la direction des arcs) est connexe.[40]
  - Un graphe orienté est **fortement connexe** si, pour toute paire ordonnée de sommets (u,v), il existe un chemin de u à v *et* un chemin de v à u.[13]
  - Les **composantes fortement connexes (CFC)** d'un graphe orienté sont ses sous-graphes fortement connexes maximaux. Tout graphe orienté admet une partition unique de ses sommets en CFC.[41] La structure d'un graphe peut être comprise comme une "factorisation" en ses "atomes cycliques" (les CFC) et la structure acyclique qui les relie. En effet, si l'on contracte chaque CFC en un unique super-sommet, le graphe résultant, appelé graphe des composantes, est toujours un graphe orienté acyclique (DAG).[41]
- **Algorithmes de Décomposition en CFC** : Deux algorithmes classiques, tous deux en temps linéaire O(|S|+|A|), permettent de trouver les CFC.
  - **Algorithme de Kosaraju** : Conceptuellement simple, il s'effectue en deux passes de DFS. La première passe sur le graphe G calcule les temps de fin de visite de chaque sommet. La seconde passe s'effectue sur le graphe transposé GT (où tous les arcs sont inversés), en explorant les sommets dans l'ordre décroissant de leur temps de fin. Chaque arbre de la forêt DFS générée lors de cette seconde passe correspond à une CFC.[42]
  - **Algorithme de Tarjan** : Plus subtil mais plus efficace en pratique, il n'effectue qu'une seule passe de DFS sur le graphe G. Il maintient une pile des sommets en cours d'exploration et associe à chaque sommet v un indice de découverte (v.num) et une valeur "low-link" (v.numAccessible). Cette dernière valeur correspond au plus petit indice de découverte accessible depuis v (y compris par un seul arc de retour). Un sommet v est la "racine" d'une CFC si et seulement si v.num == v.numAccessible. Lorsque la visite de la sous-arborescence d'une telle racine se termine, tous les sommets de sa CFC se trouvent au sommet de la pile et peuvent être extraits.[44] Cette approche illustre comment une logique plus complexe et un état plus riche peuvent permettre de réduire le nombre de passes sur les données.

3.5 Les Arbres : Structure Acyclique Fondamentale

Au sein de la théorie des graphes, les arbres occupent une place centrale. Leur simplicité structurelle — l'absence de cycles — leur confère des propriétés combinatoires et algorithmiques remarquables. Ils représentent la forme la plus épurée de la connectivité.

3.5.1 Définition et Caractérisations Équivalentes

- **Définition** : Un **arbre** est un graphe non orienté qui est à la fois **connexe** et **acyclique**.[24] Une **forêt** est un graphe acyclique, dont chaque composante connexe est un arbre.[19]
- **Théorème des Caractérisations Équivalentes** : Les multiples facettes de la structure d'un arbre sont capturées par une série de propriétés équivalentes. Pour un graphe G d'ordre n, les affirmations suivantes sont équivalentes :
  1. G est un arbre (connexe et acyclique).
  2. G est connexe et possède exactement n−1 arêtes.
  3. G est acyclique et possède exactement n−1 arêtes.
  4. Pour toute paire de sommets {u,v}, il existe une et une seule chaîne élémentaire les reliant.
  5. G est connexe, et toute arête est un **pont** (c'est-à-dire que sa suppression déconnecte le graphe).
  6. G est acyclique, et l'ajout de n'importe quelle arête entre deux sommets non adjacents crée un unique cycle.

     [46]

Ces équivalences révèlent la nature profonde de l'arbre comme "squelette" minimal de la connectivité. Un graphe connexe à n sommets doit avoir au moins n−1 arêtes, tandis qu'un graphe acyclique à n sommets peut en avoir au plus n−1.[46] L'arbre se situe précisément à cette intersection critique : il contient le nombre exact d'arêtes nécessaires pour assurer la connectivité, sans aucune redondance qui se manifesterait par un cycle.

- **Preuve de l'équivalence (esquisse)** : La preuve complète établit un cycle d'implications. Par exemple, pour montrer (1)⇔(2)⇔(3) :
  - (1)⇒(2) : On montre par récurrence sur n qu'un arbre à n sommets a n−1 arêtes. Un arbre a toujours au moins une feuille (sommet de degré 1). En retirant une feuille et son arête incidente, on obtient un arbre plus petit, ce qui permet d'appliquer l'hypothèse de récurrence.[47]
  - (2)⇒(3) : Si G est connexe avec n−1 arêtes, il doit être acyclique. Sinon, la suppression d'une arête d'un cycle préserverait la connexité, et on obtiendrait un graphe connexe avec n sommets et n−2 arêtes, ce qui est impossible car un graphe connexe a au moins n−1 arêtes.[51]
  - (3)⇒(1) : Si G est acyclique avec n−1 arêtes, il doit être connexe. S'il ne l'était pas, il serait une forêt de k>1 arbres. Chaque composante i (un arbre) aurait ni sommets et ni−1 arêtes. Le nombre total d'arêtes serait ∑(ni −1)=(∑ni)−k=n−k. Or, on a n−1 arêtes, donc n−1=n−k, ce qui implique k=1. Le graphe est donc connexe.[47]

3.5.2 Arbres Couvrants de Poids Minimum (MST)

- **Problématique** : Étant donné un graphe G connexe et pondéré, un **arbre couvrant de poids minimum** (ou MST, de l'anglais *Minimum Spanning Tree*) est un sous-graphe partiel de G qui est un arbre, qui connecte tous les sommets de G, et dont la somme des poids des arêtes est la plus faible possible.[52] Ce problème est fondamental dans la conception de réseaux (télécommunications, électricité, transport) où l'objectif est de connecter un ensemble de points avec un coût minimal.[10]
- **Propriétés d'Optimalité** : La raison pour laquelle des algorithmes gloutons peuvent résoudre ce problème de manière optimale repose sur deux propriétés fondamentales :
  - **Propriété des Coupes** : Soit une partition quelconque des sommets de G en deux ensembles disjoints. Si une arête a un poids strictement inférieur à celui de toutes les autres arêtes reliant ces deux ensembles, alors cette arête appartient à *tout* MST de G.[55]
  - **Propriété des Cycles** : Pour tout cycle dans G, si une arête a un poids strictement supérieur à celui de toutes les autres arêtes du cycle, alors cette arête n'appartient à *aucun* MST de G.[55]
- **Algorithme de Prim**
  - **Principe** : Cet algorithme glouton construit l'MST en faisant croître un arbre à partir d'un sommet de départ arbitraire. À chaque étape, il ajoute à l'arbre l'arête de poids minimum qui relie un sommet déjà dans l'arbre à un sommet qui n'y est pas encore.[56] Cette stratégie est une application directe de la propriété des coupes.
  - **Implémentation et Complexité** : L'efficacité de l'algorithme de Prim dépend de la manière dont on trouve l'arête de poids minimum à chaque étape. Une implémentation utilisant une **file de priorité** (comme un tas binaire ou un tas de Fibonacci) pour stocker les sommets non encore dans l'arbre, avec pour clé leur distance minimale à l'arbre, est la plus performante. Avec un tas binaire, la complexité est de $O(|A|\log|S|)$. Avec un tas de Fibonacci, elle atteint $O(|A|+|S|\log|S|)$.[57]
  - **Preuve de Correction** : La preuve formelle montre par récurrence que l'arbre construit à chaque étape est un sous-arbre d'au moins un MST.[57]
- **Algorithme de Kruskal**
  - **Principe** : Cet autre algorithme glouton adopte une approche différente. Il examine toutes les arêtes du graphe par ordre de poids croissant. Une arête est ajoutée à la solution si et seulement si elle ne crée pas de cycle avec les arêtes déjà sélectionnées.[59] Cette stratégie est une application directe de la propriété des cycles.
  - **Implémentation et Complexité** : L'algorithme requiert deux composantes clés : un tri initial des arêtes par poids, et une structure de données **Union-Find** (ou structure d'ensembles disjoints) pour détecter efficacement si l'ajout d'une arête connecterait deux sommets déjà dans la même composante connexe, et donc créerait un cycle.[59] La complexité totale est dominée par l'étape de tri, soit $O(|A|\log|A|)$ ou $O(|A|\log|S|)$.[59]
  - **Preuve de Correction** : La preuve montre que l'algorithme maintient l'invariant selon lequel l'ensemble d'arêtes sélectionné est toujours un sous-ensemble d'un certain MST.[59]

Le fait que ces stratégies gloutonnes, qui font des choix optimaux locaux, aboutissent à un optimum global n'est pas anodin. Cela est dû à la structure mathématique sous-jacente du problème (celle d'un matroïde), qui garantit que les solutions partielles peuvent toujours être étendues en une solution globale optimale.

3.6 Cycles et Chemins Spécifiques : Problèmes Eulériens et Hamiltoniens

Cette section explore deux problèmes fondamentaux de parcours de graphes qui, bien que formulés de manière similaire — "visiter chaque élément une et une seule fois" —, révèlent une dichotomie spectaculaire en termes de complexité computationnelle. Cette différence illustre de manière frappante la distinction entre les problèmes traitables (classe P) et les problèmes vraisemblablement intraitables (classe NP-complet).

3.6.1 Graphes Eulériens

Revenant au problème fondateur des ponts de Königsberg, nous formalisons ici les concepts introduits par Euler.

- **Définitions** : Un **parcours eulérien** est une chaîne qui passe par chaque arête du graphe exactement une fois. Si ce parcours est un cycle, il est appelé **cycle eulérien**.[11] Un graphe qui admet un cycle eulérien est dit **eulérien**, et s'il admet un parcours eulérien (non fermé), il est dit **semi-eulérien**.[64]
- **Théorème d'Euler (1736)** : Ce théorème fournit une caractérisation complète et simple des graphes eulériens, basée uniquement sur une propriété locale des sommets.
  - Un graphe connexe est **eulérien si et seulement si tous ses sommets sont de degré pair**.[8]
  - Un graphe connexe est **semi-eulérien si et seulement si il possède exactement deux sommets de degré impair**. Le parcours doit alors commencer à l'un de ces sommets et se terminer à l'autre.[1]
- **Preuve du Théorème** :
  - **Condition Nécessaire** : La preuve est intuitive. Pour tout cycle eulérien, chaque fois qu'on entre dans un sommet intermédiaire par une arête, on doit en sortir par une autre arête non encore utilisée. Chaque passage consomme donc deux arêtes incidentes à ce sommet. Le sommet de départ est également le sommet d'arrivée, donc chaque départ et chaque arrivée consomment également les arêtes par paires. Par conséquent, chaque sommet doit avoir un degré pair.[11]
  - **Condition Suffisante** : La preuve est constructive. On part d'un sommet arbitraire et on suit un chemin d'arêtes non utilisées jusqu'à revenir au point de départ, ce qui est toujours possible car chaque sommet a un degré pair. On obtient un premier cycle. S'il reste des arêtes non utilisées dans le graphe, la connexité garantit qu'au moins un sommet du cycle a une arête incidente non utilisée. On repart de ce sommet pour former un nouveau cycle, que l'on "fusionne" ensuite avec le premier. On répète ce processus jusqu'à ce que toutes les arêtes soient épuisées.[11]
- **Algorithme de Hierholzer** : Cet algorithme formalise la preuve constructive de la condition suffisante. Il trouve un cycle eulérien en temps linéaire, O(|A|), en construisant et en fusionnant des cycles.[66] La facilité de ce problème, décidable par une simple vérification des degrés, le place fermement dans la classe de complexité **P**.

3.6.2 Graphes Hamiltoniens

En changeant simplement l'objet à visiter — les sommets au lieu des arêtes — on bascule dans un tout autre univers de complexité.

- **Définitions** : Un **chemin hamiltonien** est un chemin simple qui visite chaque sommet du graphe exactement une fois. Un **cycle hamiltonien** est un cycle qui visite chaque sommet exactement une fois (sauf le sommet de départ/arrivée).[11]
- **Complexité du Problème** : Déterminer si un graphe donné possède un cycle hamiltonien est l'un des problèmes les plus célèbres de la classe **NP-complet**. Il figure parmi les 21 problèmes originaux de Karp, dont la NP-complétude a été démontrée en 1972.[69] Contrairement au problème eulérien, il n'existe aucune caractérisation simple connue.[71] La difficulté réside dans le fait que la propriété est globale et combinatoire, sans indicateur local évident.
- **Conditions Suffisantes d'Existence** : Bien qu'il n'existe pas de condition nécessaire et suffisante simple, plusieurs théorèmes fournissent des conditions suffisantes qui garantissent l'existence d'un cycle hamiltonien, généralement en imposant une forte densité d'arêtes.
    - **Théorème de Dirac (1952)** : Un graphe simple à $n \geq 3$ sommets est hamiltonien si le degré de chaque sommet est au moins $n/2$.[68]
    - **Théorème d'Ore (1960)** : Un graphe simple à $n \geq 3$ sommets est hamiltonien si, pour toute paire de sommets non adjacents $\{u,v\}$, la somme de leurs degrés vérifie $d(u)+d(v) \geq n$.[68] Le théorème de Dirac est un corollaire direct de celui d'Ore. Ces théorèmes montrent qu'une densité d'arêtes suffisamment élevée "force" l'existence d'un cycle hamiltonien.

3.6.3 Le Problème du Voyageur de Commerce (TSP)

Le problème du voyageur de commerce (TSP) est une généralisation du problème du cycle hamiltonien et constitue l'archétype des problèmes d'optimisation combinatoire NP-difficiles.

- **Définition** : Étant donné un graphe complet pondéré (où les poids représentent des distances, des coûts ou des temps), le TSP consiste à trouver le cycle hamiltonien de poids total minimum.[73]
- **Complexité et lien avec P vs. NP** : Le problème de décision associé ("existe-t-il un tour de poids inférieur ou égal à k?") est **NP-complet**. Le problème du cycle hamiltonien se réduit trivialement au TSP : un graphe G a un cycle hamiltonien si et seulement si le graphe complet construit sur les mêmes sommets, avec un poids de 1 pour les arêtes de G et un poids infini (ou très grand) pour les autres, a un tour de poids minimal égal à n.[75] Cela établit que le TSP est
  NP-difficile.

L'absence d'algorithme connu résolvant ces problèmes en temps polynomial (c'est-à-dire en un temps borné par un polynôme en la taille de l'entrée) est au cœur de la conjecture **P vs. NP**, l'un des sept problèmes du prix du millénaire. Si un tel algorithme était découvert pour le TSP ou le cycle hamiltonien, cela prouverait que P = NP, avec des conséquences révolutionnaires pour l'informatique, la cryptographie et de nombreux autres domaines scientifiques et industriels.[77]

3.7 Graphes Planaires et Coloration

Les graphes planaires sont ceux qui peuvent être représentés sur une surface bidimensionnelle sans croisement d'arêtes. Cette contrainte topologique simple a des conséquences combinatoires profondes, notamment en ce qui concerne la coloration des sommets.

3.7.1 Planarité et Formule d'Euler

- **Définition** : Un graphe est dit **planaire** s'il admet une représentation dans le plan où les sommets sont des points et les arêtes des courbes qui ne se croisent qu'à leurs extrémités communes.[6] Un tel dessin est appelé une **représentation planaire** (ou plongement).
- Formule d'Euler (1758) : Pour tout graphe planaire connexe, la relation entre le nombre de sommets (s), le nombre d'arêtes (a) et le nombre de faces (f) (y compris la face infinie extérieure) est un invariant topologique donné par :

  $$s-a+f=2$$
  [6]

  Cette formule est un pont remarquable entre la topologie et la combinatoire, car elle relie une propriété géométrique (le dessin dans le plan) à des quantités numériques.
  - **Preuve par récurrence sur le nombre d'arêtes (a)** :
    1. **Initialisation** : Si le graphe est un arbre, il est planaire. Il a $a=s-1$ arêtes et ne délimite qu'une seule face ($f=1$). La formule est vérifiée : $s-(s-1)+1=2$.
    2. **Hérédité** : Supposons la formule vraie pour tout graphe planaire connexe avec $a-1$ arêtes. Soit G un graphe avec a arêtes. Si G n'est pas un arbre, il contient un cycle. Choisissons une arête e appartenant à un cycle. Le graphe $G'=G-e$ est toujours connexe, a s sommets et $a-1$ arêtes. En retirant e, deux faces de G fusionnent en une seule dans G', donc G' a $f-1$ faces. Par hypothèse de récurrence, pour G', on a $s-(a-1)+(f-1)=2$, ce qui se simplifie en $s-a+f=2$. La formule est donc vraie pour G.[83]
- **Corollaires et Graphes Non Planaires** : La formule d'Euler impose de fortes contraintes sur le nombre d'arêtes qu'un graphe planaire peut avoir. Pour un graphe planaire simple et connexe avec $s \geq 3$ sommets, on peut déduire que $a \leq 3s-6$. Si de plus le graphe n'a pas de cycle de longueur 3 (comme un graphe biparti), alors $a \leq 2s-4$.[80] Ces inégalités fournissent un moyen simple de prouver que certains graphes ne sont pas planaires. Par exemple, pour le graphe complet
  K_5, on a $s=5$ et $a=10$. L'inégalité $a \leq 3s-6$ donne $10 \leq 3(5)-6=9$, ce qui est faux. Donc, K_5 n'est pas planaire. De même, pour le graphe biparti complet $K_{3,3}$, on a $s=6$, $a=9$, et pas de cycle de longueur 3. L'inégalité $a \leq 2s-4$ donne $9 \leq 2(6)-4=8$, ce qui est faux. Donc, $K_{3,3}$ n'est pas planaire.[80]
- **Théorème de Kuratowski (1930)** : Ce théorème fondamental offre une caractérisation complète de la planarité. Un graphe est planaire si et seulement s'il ne contient pas de sous-graphe qui soit une **subdivision** de K_5 ou de $K_{3,3}$.[81] Une subdivision d'un graphe est obtenue en insérant des sommets de degré 2 sur ses arêtes.

3.7.2 Coloration de Graphes

- **Définition** : Une **k-coloration** d'un graphe est une attribution à chaque sommet d'une couleur parmi un ensemble de k couleurs, de telle sorte que deux sommets adjacents reçoivent toujours des couleurs différentes. Le **nombre chromatique** $\chi(G)$ d'un graphe G est le plus petit entier k pour lequel une k-coloration existe.[11]
- **Théorème des Cinq Couleurs** : Tout graphe planaire est 5-coloriable ($\chi(G) \leq 5$). La preuve est élégante et constructive. Elle procède par récurrence sur le nombre de sommets, en utilisant le fait qu'un graphe planaire a toujours un sommet de degré au plus 5. Si les voisins de ce sommet utilisent moins de 5 couleurs, on peut le colorier. Dans le cas contraire, un argument ingénieux utilisant les **chaînes de Kempe** (chemins bicolores) permet de modifier localement la coloration pour libérer une couleur.[88]
- **Théorème des Quatre Couleurs** : Tout graphe planaire est 4-coloriable ($\chi(G) \leq 4$).
  - **Histoire** : Conjecturé en 1852, ce problème a résisté aux mathématiciens pendant plus d'un siècle. Une célèbre tentative de preuve par Alfred Kempe en 1879 s'est avérée défectueuse, bien que ses idées (notamment les chaînes de Kempe) aient été cruciales pour la suite.[91]
  - **La Preuve d'Appel et Haken (1976)** : La première preuve acceptée a marqué un tournant dans l'histoire des mathématiques, car elle était assistée par ordinateur. La stratégie consistait à montrer qu'un ensemble fini de 1 936 **configurations réductibles** était **inévitable** (tout graphe planaire doit en contenir au moins une). Un programme informatique a ensuite vérifié que chaque configuration de cet ensemble était bien 4-coloriable.[92] Cette dépendance à l'ordinateur pour une partie exhaustive et non vérifiable par l'homme a déclenché un débat épistémologique sur la nature même d'une preuve mathématique.[94]
- **Algorithme de Coloration Glouton** : C'est une heuristique simple et rapide pour colorier un graphe. Les sommets sont parcourus dans un ordre prédéfini, et chacun se voit attribuer la plus petite couleur (le plus petit entier positif) non utilisée par ses voisins déjà colorés.[97] Bien qu'il ne garantisse pas de trouver le nombre chromatique optimal, il est efficace en pratique et garantit une coloration avec au plus $\Delta(G)+1$ couleurs, où $\Delta(G)$ est le degré maximum du graphe.

3.8 Isomorphisme et Complexité

Le concept d'isomorphisme permet de définir formellement quand deux graphes sont "les mêmes" d'un point de vue structurel, indépendamment de leur représentation graphique. Le problème algorithmique associé à la détection de l'isomorphisme occupe une place unique et fascinante dans le paysage de la théorie de la complexité.

3.8.1 Le Problème de l'Isomorphisme de Graphes (GI)

- **Définition Formelle** : Deux graphes $G1=(S1,A1)$ et $G2=(S2,A2)$ sont dits **isomorphes** s'il existe une bijection $f:S1 \rightarrow S2$

qui préserve la structure d'adjacence. Autrement dit, pour toute paire de sommets u,v∈S1, l'arête {u,v} existe dans A1 si et seulement si l'arête {f(u),f(v)} existe dans A2.[99] L'isomorphisme est une relation d'équivalence qui partitionne l'ensemble de tous les graphes en classes de graphes structurellement identiques.[99]

- **Applications** : La capacité à déterminer si deux graphes sont isomorphes est fondamentale dans de nombreux domaines. En chimie, elle permet d'identifier des composés moléculaires en comparant leurs graphes structurels. En conception de circuits, elle est utilisée pour vérifier que le schéma logique d'un circuit correspond à sa disposition physique (Layout Versus Schematic). En analyse de réseaux, elle permet de détecter des motifs structurels récurrents.[101]

3.8.2 Un Statut Unique en Théorie de la Complexité

Le problème de décision de l'isomorphisme de graphes (GI) est l'un des problèmes les plus étudiés en informatique théorique en raison de son statut ambigu par rapport aux grandes classes de complexité.

- **GI est dans NP** : Le problème appartient à la classe **NP**. Pour prouver que deux graphes sont isomorphes, il suffit de fournir la bijection f comme "certificat". Il est possible de vérifier en temps polynomial que f est bien une bijection et qu'elle préserve les arêtes.[100]
- **Ni en P, ni NP-complet (conjecture)** : GI est l'un des très rares problèmes naturels de la classe NP pour lequel on ne connaît ni d'algorithme en temps polynomial (ce qui le placerait dans **P**), ni de preuve qu'il est **NP-complet**.[99] Cette situation suggère que la vision binaire "P ou NP-complet" pourrait être une simplification de la structure réelle de la classe NP.
- **La Classe NP-Intermédiaire** : Si P ?= NP, alors il existe des problèmes dans NP qui ne sont ni dans P ni NP-complets. Cette classe hypothétique est appelée **NP-intermédiaire**, et GI est le principal candidat pour en faire partie, avec le problème de la factorisation des entiers.[107] L'existence de tels problèmes impliquerait un spectre de difficultés plus riche et plus fin au sein de NP.
- **Implications Théoriques** : Le statut de GI a des implications profondes. Il a été démontré que si GI était NP-complet, alors la hiérarchie polynomiale s'effondrerait à son deuxième niveau, un résultat considéré comme très improbable par la plupart des experts en complexité.[100]

3.8.3 L'Avancée de Babai

Pendant des décennies, la meilleure borne supérieure connue pour la complexité de GI était de nature exponentielle. En 2015, László Babai a annoncé une avancée spectaculaire en présentant un algorithme résolvant le problème GI en temps **quasi-polynomial**, c'est-à-dire en O(2(logn)c) pour une constante c.[6] Un temps quasi-polynomial est une fonction qui croît plus vite que n'importe quel polynôme, mais de manière significativement plus lente que n'importe quelle fonction exponentielle. Ce résultat, salué comme une percée majeure, place GI beaucoup plus près de P que ce que l'on pensait auparavant, le distinguant encore davantage des problèmes NP-complets et renforçant la conjecture qu'il n'est pas NP-complet. Cela suggère que la structure de symétrie des graphes, bien que complexe, pourrait posséder une régularité

exploitable qui la rendrait plus facile à analyser que les problèmes combinatoires "chaotiques" typiques de la NP-complétude.

Ouvrages cités

1. Les ponts de Königsberg - Université de Moncton, dernier accès : septembre 21, 2025, https://www.umoncton.ca/umcm-sciences-mathstat/files/umcm-sciences-mathstat/wf/wf/pdf/les_ponts_de_konigsberg_1.pdf
2. Les 7 ponts de Königsberg - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=IZHIalIdzg0
3. Seven Bridges of Königsberg - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg
4. Introduction à la Théorie des Graphes Study Guide - Quizlet, dernier accès : septembre 21, 2025, https://quizlet.com/study-guides/introduction-a-la-theorie-des-graphes-5d2d7fe1-772c-46d1-9f22-154dfd5303ba
5. Les ponts de Königsberg - Mathrais, dernier accès : septembre 21, 2025, https://mathrais.fr/les-ponts-de-konigsberg/
6. Théorie des graphes: Concepts, Exemples - StudySmarter, dernier accès : septembre 21, 2025, https://www.studysmarter.fr/resumes/mathematiques/mathematiques-decisionnelles/theorie-des-graphes/
7. Problème des sept ponts de Königsberg — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Probl%C3%A8me_des_sept_ponts_de_K%C3%B6nigsberg
8. Ponts de Königsberg et cycle eulérien - BibM@th, dernier accès : septembre 21, 2025, https://www.bibmath.net/dico/index.php?action=affiche&quoi=./p/pont.html
9. Optimisation du problème du voyageur de commerce par du Machine Learning, dernier accès : septembre 21, 2025, https://blog.octo.com/optimisation-du-probleme-du-voyageur-de-commerce-par-du-machine-learning
10. Théorie des Graphes/ Arbre Couvrant Poids Minimum (Kruskal, Prim, Sollin); part 1, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=E8uEu-LoQ6k
11. Introduction à la théorie des graphes, dernier accès : septembre 21, 2025, https://www.imo.universite-paris-saclay.fr/~ruette/mathsdiscretes/polygraph-Sigward.pdf
12. Théorie des graphes - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Th%C3%A9orie_des_graphes
13. Quelques rappels sur la théorie des graphes - CNRS, dernier accès : septembre 21, 2025, https://perso.liris.cnrs.fr/samba-ndojh.ndiaye/fichiers/App_Graphes.pdf
14. Chapitre 1-Concepts de Base | PDF | Théorie des graphes | Matrice (Mathématiques), dernier accès : septembre 21, 2025, https://fr.scribd.com/document/715520918/Chapitre-1-Concepts-de-base
15. Les graphes, dernier accès : septembre 21, 2025, https://cahier-de-prepa.fr/mp1-janson/download?id=2905
16. Université Claude Bernard Lyon 1 Chapitre 7 Graphes, dernier accès : septembre 21, 2025, https://licence-math.univ-lyon1.fr/lib/exe/fetch.php?media=enseignements:poly-adm3.pdf
17. Structure de données : Graphes - NSI - GitLab, dernier accès : septembre 21, 2025, https://stephane_ramstein.gitlab.io/nsi/nsi_terminale/14_graphes/cours/graph/
18. Généralités sur les graphes - Eduscol - Ministère de l'Éducation nationale, dernier accès : septembre 21, 2025, https://eduscol.education.fr/document/7304/download
19. GRAPHES ET COMPLEXITE - Laboratoire de Recherche en Informatique, dernier accès : septembre 21, 2025, https://www.lri.fr/~jcohen/documents/enseignement/Cours-glouton.pdf
20. Graphes orientés, dernier accès : septembre 21, 2025, https://www.math.univ-

paris13.fr/~bonino/archives/graphes.pdf
21. Théorie des graphes: Ordre, taille et théorème des poignées de mains - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=LMNQcYRTBrA
22. Représentation des graphes et Programmation, dernier accès : septembre 21, 2025, https://helios2.mi.parisdescartes.fr/~lomn/Cours/AV/Complements/representation_graphe.pdf
23. Lemme des poignées de main — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Lemme_des_poign%C3%A9es_de_main
24. Arbre (théorie des graphes) - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Arbre_(th%C3%A9orie_des_graphes)
25. Numérique et sciences informatiques, dernier accès : septembre 21, 2025, https://eduscol.education.fr/document/7316/download
26. Liste d'adjacence - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Liste_d%27adjacence
27. Passer d'une représentation d'un graphe à une autre - myMaxicours, dernier accès : septembre 21, 2025, https://www.maxicours.com/se/cours/passer-d-une-representation-d-un-graphe-a-une-autre/
28. représentation d'un graphe (matrice d'adjacence et liste d'adjacence) , graph representation - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=u_PtF0C1sFI
29. Définitions et concepts de base - GERAD, dernier accès : septembre 21, 2025, https://www.gerad.ca/~alainh/Chapitre1.pdf
30. Algorithme de parcours en largeur — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur
31. Recherche en largeur: Algorithmes, Graphes - StudySmarter, dernier accès : septembre 21, 2025, https://www.studysmarter.fr/resumes/informatique/algorithmes-en-informatique/recherche-en-largeur/
32. Parcours de graphe: DFS, BFS - StudySmarter, dernier accès : septembre 21, 2025, https://www.studysmarter.fr/resumes/informatique/algorithmes-en-informatique/parcours-de-graphe/
33. Chapitre 19 : Algorithme sur les graphes - duranton.net, dernier accès : septembre 21, 2025, https://duranton.net/terminale_nsi/graphealgo/
34. Recherche en profondeur (Depth-First Search) en Python : Traverser des graphes et des arbres | DataCamp, dernier accès : septembre 21, 2025, https://www.datacamp.com/fr/tutorial/depth-first-search-in-python
35. Algorithme de parcours en profondeur — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_profondeur
36. DFS - Infobrol, dernier accès : septembre 21, 2025, https://www.gaudry.be/graphes-dfs.html
37. Algorithmique des graphes - Cours 4 – Parcours en profondeur - LaBRI, dernier accès : septembre 21, 2025, https://www.labri.fr/perso/fkardos/cours4.pdf
38. Graphe connexe - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Graphe_connexe
39. Chapitre 4: Graphes connexes, dernier accès : septembre 21, 2025, http://www.gymomath.ch/javmath/polycopie/th_graphe4.pdf
40. Composante fortement connexe - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Composante_fortement_connexe
41. Algorithme de Kosaraju — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Algorithme_de_Kosaraju
42. Algorithmique des graphes - 3 — Graphes orientés suite - IGM, dernier accès : septembre 21, 2025, https://igm.univ-mlv.fr/~beal/Teaching/CoursAlgoGraphes/algographes-03-graphes-orientes.pdf
43. Algorithme de Tarjan - Wikipédia, dernier accès : septembre 21, 2025,

https://fr.wikipedia.org/wiki/Algorithme_de_Tarjan
44. Une preuve formelle de l'algorithme de Tarjan-1972 pour trouver les composantes fortement connexes dans un graphe - Inria, dernier accès : septembre 21, 2025, http://pauillac.inria.fr/~levy/pubs/16jfla.pdf
45. Arbres 1 Arbres et forêts, dernier accès : septembre 21, 2025, https://www.math.univ-toulouse.fr/~msablik/Cours/Graphes/2021-2022-Graphe-TD5-Arbre.pdf
46. Chapitre 3 - Graphes connexes, arbres A - Caractérisation des arbres - Nombre cyclomatique d'un graphe - Université Paris Cité, dernier accès : septembre 21, 2025, https://helios2.mi.parisdescartes.fr/~lomn/Cours/AV/Complements/Arbre/Chapitre3GraphesConnexesEtarbres.pdf
47. Arbres couvrants aléatoires uniformes - Département de mathématiques et applications, dernier accès : septembre 21, 2025, https://www.math.ens.psl.eu/shared-files/10422/?bureaux.pdf
48. Arbres - Laboratoire G-SCOP, dernier accès : septembre 21, 2025, https://pagesperso.g-scop.grenoble-inp.fr/~stehlikm/teaching/inf303/4-arbres.pdf
49. Arbres v. 1.1 20.01.2024 Lemme 1 Soit G un graphe fini acyclique. Alors δ(G) ≤ 1, et si G est connexe, alors δ(G)=1. Preuve, dernier accès : septembre 21, 2025, http://www-labs.iro.umontreal.ca/~hahn/IFT3545/trees.pdf
50. TD1 - Arbres, dernier accès : septembre 21, 2025, https://webusers.i3s.unice.fr/~ccrespelle/enseignements/22-23-aut_M1Graphes/TD/TD1_correc.pdf
51. Minimum spanning tree, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Minimum_spanning_tree
52. www.nexa.fr, dernier accès : septembre 21, 2025, https://www.nexa.fr/post/tout-savoir-sur-lalgorithme-de-kruskal#:~:text=Un%20arbre%20couvrant%20minimal%20ou,l'ayant%20d%C3%A9velopp%C3%A9%20en%201956.
53. Arbres couvrant & MST - Université Grenoble Alpes, dernier accès : septembre 21, 2025, https://imss-www.upmf-grenoble.fr/prevert/MathSHS/MASS3/TD/TD10.htm
54. Arbre couvrant de poids minimal — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Arbre_couvrant_de_poids_minimal
55. Théorie des graphes et algorithmes - Algorithme de Prim, dernier accès : septembre 21, 2025, http://ressources.unit.eu/cours/EnsROtice/module_avance_thg_voo6/co/algoprim.html
56. Algorithme de Prim — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Algorithme_de_Prim
57. Proof of Correctness for Prim's Algorithm - UNC Greensboro, dernier accès : septembre 21, 2025, https://home.uncg.edu/cmp/faculty/srtate/330.f16/primsproof.pdf
58. Kruskal's algorithm - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Kruskal%27s_algorithm
59. Kruskal's Minimum Spanning Tree (MST) Algorithm - GeeksforGeeks, dernier accès : septembre 21, 2025, https://www.geeksforgeeks.org/dsa/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/
60. How to write pseudo code for Kruskal's algorithm - Quora, dernier accès : septembre 21, 2025, https://www.quora.com/How-can-I-write-pseudo-code-for-Kruskal-s-algorithm
61. Algorithme de Kruskal — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Algorithme_de_Kruskal
62. Chapitre 6: Graphes eulériens et hamiltoniens 6.1 Introduction et les ..., dernier accès : septembre 21, 2025, http://www.gymomath.ch/javmath/polycopie/th_graphe6.pdf
63. Graphe eulérien - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Graphe_eul%C3%A9rien
64. Graphes eulériens - Théorie des graphes | Mathraining, dernier accès : septembre 21, 2025,

https://www.mathraining.be/chapters/42/theories/138
65. Hierholzer's algorithm - Visualizations of Graph Algorithms, dernier accès : septembre 21, 2025, https://algorithms.discrete.ma.tum.de/routing/hierholzer/
66. Hierholzer's Algorithm for directed graph - GeeksforGeeks, dernier accès : septembre 21, 2025, https://www.geeksforgeeks.org/dsa/hierholzers-algorithm-directed-graph/
67. Graphe hamiltonien — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Graphe_hamiltonien
68. Théorie des graphes (3), dernier accès : septembre 21, 2025, http://www.discmath.ulg.ac.be/cours/slides03-gr.pdf
69. An introduction to NP-complete problems, dernier accès : septembre 21, 2025, http://math.uchicago.edu/~may/REU2017/REUPapers/ZhouJames.pdf
70. Parcours et graphes Hamiltoniens, dernier accès : septembre 21, 2025, http://mathafou.free.fr/themes/kgham.html
71. Ore's theorem - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Ore%27s_theorem
72. Application #2 Problème du voyageur de commerce (TSP) - GERAD, dernier accès : septembre 21, 2025, https://www.gerad.ca/Sebastien.Le.Digabel/MTH6311/8_applications_2.pdf
73. Problème du voyageur de commerce - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_voyageur_de_commerce
74. Le problème du voyageur de commerce - Départements d'enseignement et de recherche, dernier accès : septembre 21, 2025, https://www.enseignement.polytechnique.fr/informatique/INF412/uploads/Main/pc-tsp-2014-solution.pdf
75. 2-approximation du problème du voyageur de commerce euclidien - Léo Gayral, dernier accès : septembre 21, 2025, https://lgayral.pages.math.cnrs.fr/agreg/voyageur.pdf
76. If we solve the Hamiltonian cycle problem in P time, does that really show P=NP? - Quora, dernier accès : septembre 21, 2025, https://www.quora.com/If-we-solve-the-Hamiltonian-cycle-problem-in-P-time-does-that-really-show-P-NP
77. CH.4 PROBLEMES NP - COMPLETS - IGM, dernier accès : septembre 21, 2025, http://www-igm.univ-mlv.fr/~desar/cours/M1-1_Optimisation_Combinatoire/chap4.pdf
78. Problème P $\stackrel{?}{=}$ NP - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Probl%C3%A8me_P_%E2%89%9F_NP
79. Planarit´e 1 Définition et formule d'Euler, dernier accès : septembre 21, 2025, https://www.math.univ-toulouse.fr/~msablik/Cours/Graphes/2021-2022-Graphe-TD8-GraphePlanaire
80. Graphes planaires - Laboratoire G-SCOP, dernier accès : septembre 21, 2025, https://pagesperso.g-scop.grenoble-inp.fr/~stehlikm/teaching/inf303/7-planarite.pdf
81. Les graphes planaires - GERAD, dernier accès : septembre 21, 2025, https://www.gerad.ca/~alainh/Chapitre2.pdf
82. Graphes et Surfaces, dernier accès : septembre 21, 2025, https://www-sop.inria.fr/mascotte/seminaires/JCALM/lecture_notes/surf.pdf
83. Graphes Planaires, dernier accès : septembre 21, 2025, https://www.graphes.fr/planaires.html
84. Math 228: Kuratowski's Theorem, dernier accès : septembre 21, 2025, https://www.math.cmu.edu/~mradclif/teaching/228F16/Kuratowski.pdf
85. Kuratowski's theorem - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Kuratowski%27s_theorem
86. Graphe planaire — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_de_Kuratowski

87. Colorations - GERAD, dernier accès : septembre 21, 2025, https://www.gerad.ca/~alainh/Colorations.pdf
88. Théorème des cinq couleurs — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_des_cinq_couleurs
89. Deux (deux ?) minutes pour... le théorème des 4 couleurs - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=WbAgfT79t6w
90. Le théorème des quatre couleurs Projet Informatique - INF431 2014 - LIX, dernier accès : septembre 21, 2025, https://www.lix.polytechnique.fr/~werner/PI-4C/four.pdf
91. Le théorème des quatre couleurs - Accromath - UQAM, dernier accès : septembre 21, 2025, https://accromath.uqam.ca/2019/01/le-theoreme-des-quatre-couleurs/
92. Théorème des 4 couleurs: preuve et démonstration(s) par ordinateur - Indico, dernier accès : septembre 21, 2025, https://indico.math.cnrs.fr/event/3369/attachments/2198/2568/pres_4CT.pdf
93. Théorème des 4 couleurs - BibM@th, dernier accès : septembre 21, 2025, https://www.bibmath.net/dico/index.php?action=affiche&quoi=./q/quatcouleurs.html
94. The Legend of the 4 Color Theorem (MINI DOCU) - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=6y3grKkDd_c
95. Théorème des quatre couleurs — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_des_quatre_couleurs
96. Coloration gloutonne — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Coloration_gloutonne
97. Coloration d'un graphe - info-llg.fr, dernier accès : septembre 21, 2025, https://info-llg.fr/option-mp/pdf/TP_coloration.pdf
98. Isomorphisme de graphes - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Isomorphisme_de_graphes
99. Problème de l'isomorphisme de graphes — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Probl%C3%A8me_de_l%27isomorphisme_de_graphes
100. en.wikipedia.org, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Graph_isomorphism#:~:text=Its%20practical%20applications%20include%20primarily,design%20of%20an%20electronic%20circuit).
101. Isomorphisme de graphes: Théorie, Applications | StudySmarter, dernier accès : septembre 21, 2025, https://www.studysmarter.fr/resumes/mathematiques/mathematiques-discretes/isomorphisme-de-graphes/
102. Graph Theory Applications. today I would like to give you… | by Ferdi Eraslan - Medium, dernier accès : septembre 21, 2025, https://medium.com/@frdiersln/graph-theory-applications-eb8a4a1f59c8
103. Complexité - TD 2.1 Problème NP-complet sur les graphes - Laboratoire Spécification et Vérification, dernier accès : septembre 21, 2025, https://lsv.ens-paris-saclay.fr/~bordais/Complexite/2021/TD02/Solution2_1.pdf
104. Graph isomorphism problem - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Graph_isomorphism_problem
105. Complexité - CNU 27 Marseille, dernier accès : septembre 21, 2025, https://pageperso.lis-lab.fr/~benjamin.monmege/diu_eil_semaine5/cours/cours10_adhoc.pdf
106. P vs NP | What are NP-Complete and NP-Hard Problems? - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=qnl1x4GrxS8

Chapitre 4 : Langages Formels et Automates

4.1 Introduction : La Structure du Langage et de la Computation

La théorie des langages formels et des automates constitue le socle mathématique sur lequel repose une grande partie de l'informatique, de la conception des compilateurs à l'intelligence artificielle, en passant par la vérification de systèmes complexes. Elle offre un cadre rigoureux pour étudier les notions fondamentales de description, de reconnaissance et de calcul. Ce chapitre explore la relation intime entre les langages, définis par des règles de génération (les grammaires), et les machines abstraites conçues pour les reconnaître (les automates). Cette dualité est au cœur de la hiérarchie de Chomsky, une classification élégante qui organise les langages selon leur complexité structurelle et la puissance de calcul nécessaire pour les manipuler.

4.1.1 Définitions Préliminaires : Alphabets, Mots, Langages

Avant de construire des structures complexes, il est essentiel de définir leurs composants les plus élémentaires.

- **Alphabet** : Un alphabet, noté Σ, est un ensemble fini et non vide de symboles, aussi appelés lettres.[1] Par exemple, l'alphabet binaire est $\Sigma=\{0,1\}$, et l'alphabet des lettres latines minuscules est $\Sigma=\{a,b,c,...,z\}$.
- **Mot** : Un mot (ou une chaîne de caractères) sur un alphabet Σ est une suite finie de symboles de Σ.[1] La **longueur** d'un mot w, notée $|w|$, est le nombre de symboles qui le composent. Le **mot vide**, noté ε, est l'unique mot de longueur zéro.[1] L'ensemble de tous les mots possibles sur un alphabet Σ est noté $\Sigma*$.
- **Concaténation** : L'opération fondamentale sur les mots est la concaténation, qui consiste à juxtaposer deux mots. Si $u=a1...an$ et $v=b1...bm$, alors leur concaténation est $uv=a1...anb1...bm$. Cette opération est associative et possède le mot vide ε comme élément neutre ($w\varepsilon=\varepsilon w=w$). L'ensemble $\Sigma*$ muni de la concaténation forme une structure algébrique appelée **monoïde libre**.[1]
- **Langage Formel** : Un langage formel L sur un alphabet Σ est simplement un sous-ensemble de $\Sigma*$, c'est-à-dire un ensemble de mots.[3] Ce peut être un ensemble fini, comme $L=\{chat,chien,souris\}$, ou un ensemble infini, comme l'ensemble de tous les programmes C++ valides.

4.1.2 Le concept de Grammaire Formelle : Un Mécanisme Génératif

Comment définir rigoureusement un langage, surtout s'il est infini? Une approche consiste à utiliser un mécanisme génératif : une grammaire formelle. Intuitivement, une grammaire est un ensemble de règles de réécriture qui permettent de construire toutes les "phrases" correctes d'un langage, et uniquement celles-ci.[5]

Formellement, une grammaire est un quadruplet $G=(V,T,P,S)$.[5]

- V (ou VN) est un ensemble fini de **symboles non terminaux** ou **variables**. Ces symboles représentent des concepts syntaxiques abstraits, comme <phrase>, <nom> ou <verbe>.[5]
- T (ou Σ, VT) est un alphabet fini de **symboles terminaux**, disjoint de V. Ce sont les symboles qui composent les mots finaux du langage.[5]
- P est un ensemble fini de **règles de production** (ou de réécriture). Chaque règle est de la forme $u \to v$, où u et v sont des chaînes de symboles de $(V \cup T)*$, et u doit contenir au moins un symbole non terminal.[5]
- $S \in V$ est un non-terminal particulier appelé l'**axiome** ou le **symbole de départ**. C'est le point de départ de toute construction.[5]

4.1.3 Le Processus de Dérivation

Une grammaire engendre un langage par un processus appelé dérivation.

- **Dérivation directe** : On dit qu'une chaîne x dérive directement en une chaîne y, noté $x \Rightarrow y$, s'il existe une règle $u \to v$ dans P et des chaînes α,β telles que $x=\alpha u\beta$ et $y=\alpha v\beta$.[5] L'application d'une règle consiste à remplacer une occurrence de sa partie gauche par sa partie droite. Il est crucial de distinguer le symbole $\to$, qui définit une règle, du symbole $\Rightarrow$, qui dénote son application.[5]
- **Dérivation** : La relation $\Rightarrow*$ est la fermeture réflexive et transitive de $\Rightarrow$. On écrit $x \Rightarrow* y$ si y peut être obtenue à partir de x en zéro ou plusieurs étapes de dérivation directe.
- **Langage engendré** : Le langage engendré par une grammaire G, noté L(G), est l'ensemble de tous les mots composés uniquement de symboles terminaux qui peuvent être dérivés à partir de l'axiome S. Formellement : $L(G)=\{w \in T* \mid S \Rightarrow* w\}$.[5]

4.1.4 Présentation de la Hiérarchie de Chomsky : Une feuille de route pour la complexité

En 1956, le linguiste Noam Chomsky a proposé une classification des grammaires formelles qui est devenue un pilier de l'informatique théorique.[6] Cette classification, connue sous le nom de

hiérarchie de Chomsky, organise les grammaires (et donc les langages qu'elles engendrent) en quatre types, numérotés de 0 à 3, en fonction de restrictions croissantes sur la forme de leurs règles de production.[6]

Le principe fondamental est celui d'une **hiérarchie de confinement** : tout langage de type i est aussi un langage de type i−1. Ainsi, les langages de Type-3 sont un sous-ensemble des langages de Type-2, qui sont eux-mêmes un sous-ensemble des langages de Type-1, et ainsi de suite.[8]

La beauté de cette hiérarchie réside dans sa dualité : à chaque type de grammaire, qui est un modèle *génératif*, correspond une classe de machines abstraites, les **automates**, qui agissent comme des modèles de *reconnaissance* de puissance équivalente.[6]

Cette structure n'est pas une simple taxonomie. Elle incarne un principe fondamental de l'informatique : un compromis constant entre la **puissance expressive** d'un formalisme et la **décidabilité** des problèmes qui lui sont associés.[5] Plus une classe de grammaires est restrictive (plus on descend dans la hiérarchie, de 0 vers 3), moins elle peut décrire de langages complexes. En contrepartie, les questions fondamentales comme "ce mot appartient-il au langage?" deviennent algorithmiquement plus simples à résoudre. Le Type-0, au sommet, peut décrire tout ce qui est calculable, mais au prix de l'indécidabilité de presque toutes les questions intéressantes, comme le fameux problème de l'arrêt.[3] À l'inverse, le Type-3 est très limité dans son expressivité, mais le problème de l'appartenance y est résoluble de manière très efficace. La hiérarchie de Chomsky est donc une cartographie de ce compromis fondamental.

Le tableau suivant synthétise cette organisation, qui servira de fil conducteur pour le reste de ce chapitre.

Tableau 4.1 : La Hiérarchie de Chomsky - Synthèse

Type	Nom du Langage	Forme des Règles de Grammaire (A,B∈V, a∈T, w∈(V∪T)∗, u,v∈(V∪T)+)	Automate de Reconnaissance	Décidabilité de l'Appartenance	Exemple Canonique
Type-3	Régulier	A→aB ou A→a (linéaire à droite)	Automate Fini (AF)	Décidable (temps linéaire)	a∗b
Type-2	Hors-Contexte (Algébrique)	A→w	Automate à Pile (AP)	Décidable (temps polynomial, ex: O(n3))	$\{a^nb^n | n \ge 0\$
Type-1	Contextuel	$	u	\le	v

	(Sensible au contexte)				
Type-0	Récursivement Énumérable	u→v (aucune restriction)	Machine de Turing (MT)	Semi-décidable	Langage de l'arrêt

Sources pour le tableau : [5]

4.2 Type-3 : Langages Réguliers et Automates à États Finis

La classe la plus simple et la plus fondamentale de la hiérarchie est celle des langages réguliers. Bien que limités en expressivité, ils sont omniprésents en informatique pratique, notamment pour la reconnaissance de motifs et l'analyse lexicale. Ils sont caractérisés par des machines à mémoire finie : les automates finis.

4.2.1 Le Modèle de l'Automate Fini Déterministe (AFD)

Un automate fini déterministe (AFD) est une machine abstraite qui lit un mot d'entrée, symbole par symbole, et change d'état en fonction du symbole lu. Il ne dispose d'aucune mémoire supplémentaire au-delà de son état courant.[17]

Un AFD est formellement défini comme un quintuplet $A=(Q,\Sigma,\delta,q_0,F)$ où [1] :

- Q est un ensemble fini d'**états**.
- Σ est l'**alphabet** d'entrée.
- $\delta: Q\times\Sigma\to Q$ est la **fonction de transition**. Le caractère "déterministe" vient du fait que pour chaque couple (état, symbole), il existe une et une seule transition possible.[1]
- $q_0\in Q$ est l'**état initial** unique.
- $F\subseteq Q$ est l'ensemble des **états finaux** ou acceptants.

Le fonctionnement de l'automate est formalisé par la **fonction de transition étendue**, $\delta^*: Q\times\Sigma^*\to Q$, qui calcule l'état final après la lecture d'un mot entier. Le **langage reconnu** par l'AFD A est l'ensemble des mots qui mènent l'automate d'un état initial à un état final : $L(A)=\{w\in\Sigma^* \mid \delta^*(q_0,w)\in F\}$.[1] Un AFD peut être représenté visuellement par un

diagramme d'états (un graphe orienté) ou par une **table de transitions**.[17]

4.2.2 Le Non-déterminisme : L'Automate Fini Non-déterministe (AFN)

Un automate fini non-déterministe (AFN) est une généralisation de l'AFD qui autorise l'ambiguïté dans ses transitions. À une étape donnée, plusieurs chemins de calcul peuvent être possibles.[23]

Un AFN est formellement défini comme un quintuplet $A=(Q,\Sigma,\delta,I,F)$.[1] Les différences clés avec l'AFD sont :

- La fonction de transition δ est de la forme $Q\times(\Sigma\cup\{\varepsilon\})\to P(Q)$ (où $P(Q)$ est l'ensemble des parties de Q). Cela signifie que pour un état et un symbole donnés, l'automate peut transiter vers un *ensemble* d'états. De plus, il peut effectuer des **ε-transitions**, c'est-à-dire des changements d'état spontanés sans consommer de symbole d'entrée.[2]
- $I\subseteq Q$ est un *ensemble* d'états initiaux (bien que souvent, par convention, on utilise un seul état initial).

Un mot w est accepté par un AFN s'il existe **au moins un** chemin de calcul possible, partant d'un état initial, qui consomme w et se termine dans un état final.[18]

4.2.3 Théorème d'Équivalence : La Construction par Sous-Ensembles

Le non-déterminisme semble intuitivement plus puissant, car il permet à la machine d'"explorer" plusieurs possibilités en parallèle. Cependant, pour les automates finis, cette puissance apparente est une illusion. Le **théorème de Rabin-Scott** stipule que pour tout AFN, il existe un AFD qui reconnaît exactement le même langage.[19]

La preuve de ce théorème est constructive et repose sur l'**algorithme de construction par sous-ensembles** (ou déterminisation).[18] L'idée centrale est de construire un AFD où chaque état correspond à un

ensemble d'états de l'AFN. Un état $\{q1,...,qk\}$ dans l'AFD signifie que l'AFN, après avoir lu une certaine partie du mot, pourrait se trouver dans n'importe lequel des états $q1,...,qk$. L'état initial de l'AFD est l'ensemble des états de l'AFN atteignables depuis les états initiaux par des ε-transitions (la **ε-clôture**).[26] Les états finaux de l'AFD sont tous les ensembles d'états qui contiennent au moins un état final de l'AFN.[21]

Puisque l'ensemble des états de l'AFN est fini (Q), le nombre de sous-ensembles possibles est également fini ($2^{|Q|}$), bien que potentiellement beaucoup plus grand.[18] L'AFD simule donc l'exploration parallèle de tous les chemins de l'AFN en maintenant à chaque étape l'ensemble des "fronts d'onde" du calcul. Cette simulation prouve que la puissance de reconnaissance est identique.

Enfin, pour un langage régulier donné, il existe un AFD unique ayant un nombre minimal d'états. Cet automate minimal peut être obtenu à partir de n'importe quel AFD reconnaissant le langage en fusionnant les états "indiscernables" (qui se comportent de la même manière pour le reste de n'importe quel mot).[1]

4.2.4 Expressions Régulières : Un Formalisme Déclaratif

Les expressions régulières (ER), ou expressions rationnelles, offrent une manière algébrique et déclarative de décrire les langages réguliers. Elles sont à la base des outils de recherche de motifs (comme grep).[8]

Une expression régulière sur un alphabet Σ est définie inductivement [30] :

- **Cas de base** : ∅ (dénote le langage vide), ε (dénote le langage {ε}), et a pour tout a∈Σ (dénote le langage {a}) sont des expressions régulières.
- **Cas inductifs** : Si R et S sont des expressions régulières, alors :
  - (R|S) ou (R+S) (l'**union**) est une ER dénotant L(R)∪L(S).
  - (RS) (la **concaténation**) est une ER dénotant L(R)L(S).
  - (R∗) (l'**étoile de Kleene**) est une ER dénotant L(R)∗.

4.2.5 Le Théorème de Kleene : L'Unification des Langages Réguliers

Le **théorème de Kleene** est un résultat majeur qui établit l'unité de la classe des langages réguliers. Il affirme qu'un langage est régulier (reconnaissable par un automate fini) si et seulement s'il peut être décrit par une expression régulière.[33]

La preuve est constructive et fournit des algorithmes pour passer d'une représentation à l'autre :

1. **ER → AFN (Algorithme de Thompson)** : Cet algorithme construit un AFN avec ε-transitions à partir d'une ER, en suivant la structure inductive de l'expression. Il existe des constructions simples pour les cas de base (∅,ε,a) et pour combiner des automates existants pour les opérations d'union, de concaténation et d'étoile.[36]
2. **AF → ER (Méthode d'élimination d'états)** : Cet algorithme transforme un automate en une expression régulière en éliminant progressivement ses états. À chaque élimination, les transitions sont ré-étiquetées avec des expressions régulières qui décrivent les chemins qui passaient par l'état supprimé. Le processus se termine lorsqu'il ne reste que les états initial et final, reliés par une transition dont l'étiquette est l'ER recherchée.[35]

4.2.6 Grammaires Régulières

Les grammaires de Type-3, dites **régulières**, sont les grammaires les plus restrictives de la hiérarchie. Leurs règles de production doivent être de la forme A→aB ou A→a (grammaire **linéaire à droite**), ou bien de la forme A→Ba ou A→a (grammaire **linéaire à gauche**).[5] Il est crucial de ne pas mélanger les deux formes dans une même grammaire.

Il existe une équivalence directe entre ces grammaires et les automates finis : les langages qu'elles engendrent sont précisément les langages réguliers. On peut construire un AFN à partir d'une grammaire régulière (les non-terminaux

deviennent des états) et inversement (les états deviennent des non-terminaux).[17]

4.2.7 Propriétés des Langages Réguliers

- **Propriétés de fermeture** : La classe des langages réguliers est fermée pour un grand nombre d'opérations : union, concaténation, étoile (par définition), mais aussi **intersection**, **complémentation** et **image miroir**.[13] La fermeture par intersection et complémentation se démontre élégamment en utilisant les AFD (pour le complément, il suffit d'inverser les états finaux et non-finaux d'un AFD complet ; pour l'intersection, on utilise une construction appelée produit d'automates).[13]
- **Le Lemme de l'Étoile (Pumping Lemma)** : C'est un outil puissant pour prouver qu'un langage n'est *pas* régulier. Il formalise la limitation fondamentale des automates finis : leur mémoire finie.
  Théorème : Pour tout langage régulier L, il existe une constante $p \geq 1$ (la longueur de pompage) telle que tout mot $w \in L$ de longueur $|w| \geq p$ peut être décomposé en trois parties, $w=xyz$, satisfaisant les conditions suivantes :
  1. $|y|>0$ (la partie à "pomper" n'est pas vide).
  2. $|xy| \leq p$ (la boucle apparaît au début du mot).
  3. Pour tout entier $i \geq 0$, le mot $xyiz$ est aussi dans L.

     [13]

La preuve de ce lemme repose sur le **principe des tiroirs** : si un AFD à p états lit un mot de longueur p ou plus, il doit nécessairement visiter au moins un état deux fois. Le chemin entre les deux occurrences de cet état forme une boucle, dont l'étiquette est le mot y. Cette boucle peut alors être parcourue zéro fois (i=0), une fois (i=1), ou plusieurs fois, générant une famille infinie de mots qui doivent tous être dans le langage.[13] Le lemme capture donc l'idée que la mémoire finie de l'automate l'empêche de "compter" au-delà dep.Pour montrer qu'un langage n'est pas régulier, on suppose par l'absurde qu'il l'est, on applique le lemme, et on montre qu'en "pompant" un mot bien choisi, on génère un mot qui n'est pas dans le langage, ce qui est une contradiction. L'exemple canonique est de prouver que $L=\{a^n b^n | n \geq 0\}$ n'est pas régulier.[13]

4.2.8 Applications Pratiques

Les langages réguliers et leurs formalismes sont fondamentaux dans de nombreux domaines pratiques :

- **Analyse lexicale** : La première étape d'un compilateur, la "tokenization", consiste à découper le code source en unités lexicales (identifiants, mots-clés, opérateurs) à l'aide d'expressions régulières.[8]
- **Recherche de motifs** : Les outils comme grep et les fonctions de recherche dans les éditeurs de texte utilisent des moteurs d'expressions régulières pour localiser des motifs dans de grands volumes de texte.[8]
- **Validation de données** : Vérification que des entrées utilisateur respectent un format précis (adresses e-mail, numéros de téléphone, URLs).

4.3 Type-2 : Langages Hors-Contexte et Automates à Pile

Les langages réguliers, malgré leur utilité, ne peuvent pas modéliser des structures imbriquées ou récursives, comme les paires de parenthèses bien formées ou la structure des blocs dans les langages de programmation. Pour cela, il faut monter d'un cran dans la hiérarchie de Chomsky vers les langages de Type-2.

4.3.1 Au-delà de la Régularité : Introduction aux Grammaires Hors-Contexte

La limitation des langages réguliers est illustrée par le langage L={anbn|n≥0}, qui n'est pas régulier car un automate fini ne peut pas "se souvenir" du nombre de a lus pour le comparer au nombre de b.[8] Les grammaires hors-contexte (GHC), aussi appelées grammaires algébriques, surmontent cette limitation.

Une grammaire est dite **hors-contexte** (ou de Type-2) si toutes ses règles de production sont de la forme A→w, où A est un unique symbole non terminal et w est une chaîne quelconque de terminaux et/ou de non-terminaux (w∈(V∪T)∗).[5] Le terme "hors-contexte" signifie que la règle de remplacement pour

A peut être appliquée indépendamment des symboles qui l'entourent (son contexte).[48]

Par exemple, la grammaire G=({S},{a,b},{S→aSb,S→ε},S) est hors-contexte et engendre précisément le langage {anbn|n≥0}.

4.3.2 Arbres de Dérivation et le Problème de l'Ambiguïté

Une dérivation dans une GHC peut être représentée visuellement par un **arbre de dérivation** (ou arbre d'analyse syntaxique), dont les nœuds internes sont des non-terminaux, les feuilles sont des terminaux, et la descendance d'un nœud correspond à l'application d'une règle de production.[52] Cet arbre révèle la structure syntaxique sous-jacente du mot engendré.

Une grammaire est dite **ambiguë** si au moins un mot de son langage peut être engendré par plusieurs arbres de dérivation distincts.[54] L'ambiguïté est un problème majeur en compilation, car elle implique qu'un même programme pourrait avoir plusieurs interprétations sémantiques. Des exemples classiques incluent les grammaires pour les expressions arithmétiques (l'expression

3+4*5 peut être interprétée comme (3+4)*5 ou 3+(4*5)) [52] et le problème du "dangling else" dans les instructions conditionnelles.[56]

Certains langages, dits **inhéremment ambigus**, ne peuvent être décrits par aucune grammaire non ambiguë. C'est le cas,

par exemple, du langage {aibjck|i=j ou j=k}.[48]

4.3.3 Simplification et Formes Normales : La Forme Normale de Chomsky (FNC)

Pour faciliter l'analyse algorithmique des GHC, il est souvent utile de les convertir en une forme standardisée. La **Forme Normale de Chomsky (FNC)** est l'une des plus importantes.[14] Une grammaire est en FNC si toutes ses règles sont de l'une des deux formes suivantes :

- A→BC (un non-terminal se réécrit en deux non-terminaux)
- A→a (un non-terminal se réécrit en un terminal)
  où A,B,C sont des non-terminaux et a est un terminal.14 La règle
  S→ε est autorisée si l'axiome S n'apparaît jamais en partie droite d'une règle.

Il est prouvé que tout langage hors-contexte ne contenant pas le mot vide peut être engendré par une grammaire en FNC. Un algorithme systématique permet de convertir n'importe quelle GHC en FNC en éliminant d'abord les ε-productions et les règles unitaires (A→B), puis en restructurant les règles restantes.[14]

4.3.4 Le Modèle de l'Automate à Pile (AP) : Une Mémoire LIFO

L'automate capable de reconnaître les langages hors-contexte est l'**automate à pile** (AP). Il s'agit d'un automate fini non-déterministe auquel on a ajouté une mémoire auxiliaire : une **pile**, qui fonctionne selon le principe LIFO (Last-In, First-Out).[57]

Un automate à pile est formellement un 7-uplet $A=(Q,\Sigma,\Gamma,\delta,q0,Z0,F)$ [58] :

- Q,Σ,q0,F sont définis comme pour un AFN.
- Γ est l'**alphabet de pile**, l'ensemble des symboles pouvant être stockés dans la pile.
- $Z0 \in \Gamma$ est le **symbole initial de pile**.
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to P(Q \times \Gamma*)$ est la fonction de transition. Une transition dépend non seulement de l'état courant et du symbole d'entrée, mais aussi du symbole au sommet de la pile. L'action consiste à changer d'état et à remplacer le symbole au sommet de la pile par une chaîne de symboles de pile (empiler, dépiler ou laisser inchangé).

La pile est la matérialisation de la capacité récursive des GHC. Pour reconnaître {anbn}, l'automate peut empiler un symbole pour chaque a lu, puis dépiler un symbole pour chaque b lu. La pile sert de compteur non borné, ce qui était impossible pour un automate fini.[60]

4.3.5 Équivalence des Grammaires Hors-Contexte et des Automates à Pile

Un théorème fondamental établit que les GHC et les AP ont la même puissance expressive : un langage est hors-contexte si et seulement s'il est reconnu par un automate à pile.[59]

- **GHC → AP** : On peut construire un AP qui simule une dérivation (gauche, par exemple) d'une grammaire. L'automate utilise sa pile pour mémoriser la séquence de terminaux et de non-terminaux à engendrer. Si le sommet de la pile est un non-terminal, l'automate choisit (de manière non déterministe) une règle pour le remplacer. Si c'est un terminal, il le compare avec le symbole d'entrée courant.[59]
- **AP → GHC** : La construction inverse est plus complexe. L'idée est de créer des non-terminaux de la forme [p,X,q] qui représentent la capacité de l'automate à passer de l'état p à l'état q en consommant une partie de l'entrée et en dépilant le symbole X. Les règles de la grammaire simulent alors les transitions de l'automate.

4.3.6 Le Déterminisme dans les Automates à Pile et ses Limites

Un automate à pile est **déterministe** (APD) si, pour toute configuration (état, symbole d'entrée, symbole de pile), il existe au plus une transition possible.[58]

Ici, une différence fondamentale apparaît avec les automates finis : les automates à pile déterministes sont **strictement moins puissants** que leurs homologues non déterministes.[63] Le non-déterminisme devient une source de puissance expressive. Par exemple, le langage des palindromes sur

{a,b} n'est pas reconnaissable par un APD. L'automate doit "deviner" où se trouve le milieu du mot pour inverser son comportement (passer de l'empilement au dépilement). Un APN peut faire cette "divination" en explorant les deux possibilités (continuer à empiler ou commencer à dépiler) via une ε-transition. Un APD, ne pouvant faire de choix, est incapable de reconnaître ce langage.[65]

4.3.7 Propriétés des Langages Hors-Contexte

- **Propriétés de fermeture** : Les langages hors-contexte sont fermés par union, concaténation et étoile de Kleene. Cependant, ils ne sont **pas fermés par intersection ni par complémentation**.
- **Lemme d'itération pour les langages hors-contexte** (parfois appelé lemme de Bar-Hillel, Perles et Shamir, ou lemme d'Ogden) : C'est une généralisation du lemme de l'étoile. Il stipule que pour tout mot suffisamment long w d'un langage hors-contexte, on peut le décomposer en w=uvxyz de telle sorte que les deux sous-chaînes v et y (non toutes deux vides) peuvent être "pompées" simultanément : uvixyiz appartient au langage pour tout i≥0.[49] Ce lemme est utilisé pour prouver que des langages comme
  {anbncn|n≥0} ne sont pas hors-contexte.[65]

4.3.8 Applications Pratiques

- **Analyse syntaxique (Parsing)** : Les GHC sont le fondement de l'analyse syntaxique dans les compilateurs. Un analyseur syntaxique (parser) prend une séquence de tokens (produite par l'analyseur lexical) et tente de construire un arbre de dérivation selon les règles de la grammaire du langage de programmation.[8]
- **Définition de formats structurés** : Des formats comme XML ou JSON peuvent être décrits par des grammaires hors-contexte.
- **Linguistique computationnelle** : Modélisation de la structure syntaxique des langues naturelles.[48]

4.4 Type-1 : Langages Contextuels et Automates Linéairement Bornés

Les langages de Type-1, ou contextuels, ajoutent une couche de complexité par rapport aux langages hors-contexte. Ils permettent de modéliser des dépendances qui ne sont pas purement imbriquées, comme celles requises par le langage de référence {anbncn|n≥1}, qui n'est pas hors-contexte.[11]

4.4.1 La Notion de Contexte : Grammaires Contextuelles et Croissantes

Une grammaire est dite **contextuelle** (ou sensible au contexte, Type-1) si ses règles de production sont de la forme αAβ→αγβ, où A est un non-terminal, α et β sont des chaînes quelconques (le "contexte"), et γ est une chaîne non vide.[8] Cette forme stipule que

A ne peut être remplacé par γ que s'il est entouré par α et β.

Une définition équivalente et plus simple à manipuler est celle de **grammaire croissante** (ou non contractante). Dans une telle grammaire, pour chaque règle u→v, la longueur de la partie gauche est inférieure ou égale à celle de la partie droite : |u|≤|v|.[5] Ces grammaires engendrent la même classe de langages que les grammaires contextuelles (à l'exception possible du mot vide).[16]

4.4.2 Le Modèle de l'Automate Linéairement Borné (ALB)

L'automate correspondant aux langages contextuels est l'**automate linéairement borné** (ALB, ou LBA en anglais). Il s'agit d'une machine de Turing non déterministe dont la capacité de mémoire est restreinte : la tête de lecture/écriture ne peut pas se déplacer au-delà des cases du ruban qui contenaient initialement le mot d'entrée.[8] La mémoire de travail est donc bornée par une fonction linéaire de la taille de l'entrée.

4.4.3 Équivalence des Grammaires Contextuelles et des ALB

Le **théorème de Kuroda (1964)** établit l'équivalence entre ces deux formalismes : un langage est contextuel si et seulement s'il est reconnu par un ALB.[70]

- **GCS → ALB** : La preuve de cette direction est intuitive. Pour vérifier si un mot w appartient à L(G), un ALB peut simuler la dérivation à l'envers, en partant de w et en essayant de le réduire à l'axiome S. Comme les règles de la grammaire sont non contractantes, chaque étape de cette réduction inverse ne peut pas allonger la chaîne. Par conséquent, tout le processus de calcul peut s'effectuer dans l'espace initialement occupé par w.[68]
- **ALB → GCS** : La construction inverse est plus technique. Elle consiste à construire une grammaire dont les règles simulent les transitions de l'ALB. Les non-terminaux encodent les configurations complètes de l'ALB (état, position de la tête, contenu du ruban).[68]

4.4.4 Propriétés et Complexité de la Reconnaissance

La contrainte de non-contraction a une conséquence majeure : elle rend le problème de l'appartenance décidable. Pour un mot w donné, une dérivation ne peut pas passer par des chaînes intermédiaires de longueur infinie. Le nombre de chaînes de longueur au plus |w| est fini. On peut donc, en théorie, énumérer toutes les dérivations possibles sans entrer dans une boucle infinie de chaînes de plus en plus courtes, ce qui garantit la terminaison de l'algorithme de reconnaissance.[11]

Cette décidabilité est la conséquence directe de la borne sur la mémoire de l'automate. Un ALB travaillant sur un mot de longueur n possède un nombre de configurations total qui, bien qu'exponentiel en n ($|Q|\times n\times|\Gamma|^n$), est fini. Un calcul qui dure plus longtemps que ce nombre doit nécessairement répéter une configuration, entrant dans une boucle détectable. C'est la suppression de cette borne qui mène à l'indécidabilité pour les machines de Turing.

Cependant, "décidable" ne signifie pas "efficace". Le problème de l'appartenance pour les langages contextuels est **PSPACE-complet**, ce qui le place parmi les problèmes considérés comme très difficiles et intraitables en pratique.[16] De plus, contrairement au problème de l'appartenance, le

problème du vide (savoir si L(G) est vide) est indécidable pour les grammaires contextuelles.[16]

Une question longtemps ouverte était de savoir si les langages contextuels étaient fermés par complémentation. La réponse est affirmative, un résultat profond connu sous le nom de **théorème d'Immerman-Szelepcsényi** (1987).[70] Une

autre question majeure reste ouverte : les ALB déterministes sont-ils aussi puissants que les non-déterministes? En termes de classes de complexité, cela revient à se demander si NSPACE(O(n)) = DSPACE(O(n)).[10]

4.5 Type-0 : Langages Récursivement Énumérables et la Machine de Turing

Au sommet de la hiérarchie se trouvent les grammaires de Type-0 et leur automate équivalent, la machine de Turing. Cette classe représente la limite de ce qui peut être décrit par un formalisme de réécriture et de ce qui peut être calculé par un algorithme.

4.5.1 Grammaires Générales : Le Pouvoir de Réécriture Illimité

Les grammaires de Type-0, aussi appelées **non contraintes** ou **générales**, sont les plus permissives. Leurs règles de production sont de la forme u→v, avec pour seule contrainte que u ne soit pas le mot vide et contienne au moins un non-terminal.[5] La possibilité d'avoir des règles "raccourcissantes" (

|u|>|v|) est ce qui leur confère leur pleine puissance et est la source de l'indécidabilité des problèmes les concernant.[15]

4.5.2 La Machine de Turing : Le Modèle Universel de Calcul

La **machine de Turing (MT)** est le modèle d'automate équivalent aux grammaires de Type-0.[8] Conçue par Alan Turing, elle est constituée d'un automate fini (l'unité de contrôle) et d'un

ruban infini qui sert à la fois d'entrée, de sortie et de mémoire de travail.[58] À chaque étape, en fonction de son état et du symbole lu sur le ruban, la machine peut changer d'état, écrire un nouveau symbole sur le ruban et déplacer sa tête de lecture/écriture d'une case vers la gauche ou la droite.

Le passage de l'ALB à la MT consiste à lever la restriction sur la mémoire. Ce simple changement fait passer d'une machine puissante mais décidable à un modèle de calcul universel, capable de simuler n'importe quel autre processus algorithmique, y compris d'autres machines de Turing (via le concept de machine de Turing universelle).[75]

Les langages reconnus par les machines de Turing sont appelés **langages récursivement énumérables**.[3]

4.5.3 La Thèse de Church-Turing

La machine de Turing n'est pas juste un modèle parmi d'autres. La **thèse de Church-Turing** (une thèse et non un théorème, car elle relie un concept formel à une notion intuitive) postule que toute fonction calculable par un algorithme peut être calculée par une machine de Turing. Autrement dit, la MT capture formellement la notion même de "calculabilité effective".

4.5.4 Les Limites du Calcul : Décidabilité et le Problème de l'Arrêt

La puissance universelle de la machine de Turing a un prix : l'existence de problèmes fondamentalement insolubles par un algorithme.

- **Langages récursifs vs. récursivement énumérables** :
  - Un langage est **récursivement énumérable** si une MT existe qui s'arrête et accepte tout mot du langage. Si un mot n'est pas dans le langage, la machine peut soit s'arrêter et le rejeter, soit boucler indéfiniment. On parle de **semi-décideur**.[3]
  - Un langage est **récursif** (ou décidable) si une MT existe qui s'arrête *pour toute entrée*, en acceptant ou en rejetant le mot. On parle de **décideur**.[8] La classe des langages récursifs est un sous-ensemble strict de celle des langages récursivement énumérables.
- **Le problème de l'arrêt** : C'est le problème indécidable le plus célèbre. Il n'existe aucun algorithme (aucune MT) capable de déterminer, pour une MT M et une entrée w arbitraires, si le calcul de M sur w finira par s'arrêter. Le langage correspondant à ce problème est récursivement énumérable (on peut simuler M sur w et s'arrêter si la simulation s'arrête), mais il n'est pas récursif.
- **Théorème de Rice** : Ce théorème généralise l'indécidabilité du problème de l'arrêt. Il affirme que toute propriété *non triviale* (c'est-à-dire qui n'est pas vraie pour tous les langages RE ou pour aucun) des langages récursivement énumérables est indécidable.[11] Par exemple, il est indécidable de savoir si le langage reconnu par une MT est vide, fini, régulier, ou contextuel.

4.6 Synthèse : La Hiérarchie en Perspective

4.6.1 Récapitulatif des Classes de Langages et de leurs Propriétés

La hiérarchie de Chomsky offre une vision structurée et profonde de la relation entre la description des langages et la

complexité de leur reconnaissance. Chaque niveau, du Type-3 au Type-0, représente une classe de langages aux propriétés bien définies, caractérisée par une forme de grammaire et un modèle d'automate équivalent. Les inclusions entre ces classes sont strictes, signifiant qu'à chaque niveau, on gagne en puissance expressive, capable de décrire des structures linguistiques que le niveau inférieur ne pouvait pas capturer.[8]

4.6.2 L'Interaction entre Expressivité, Puissance de Reconnaissance et Décidabilité

Le fil conducteur de ce chapitre a été le compromis fondamental entre la capacité d'un formalisme à décrire des motifs complexes et la possibilité de répondre algorithmiquement à des questions simples à son sujet.[5]

- Les **langages réguliers (Type-3)**, avec leur mémoire finie, sont simples, efficaces à analyser et leurs propriétés sont toutes décidables.
- Les **langages hors-contexte (Type-2)**, en ajoutant une pile (mémoire LIFO), permettent de modéliser la récursivité et les structures imbriquées, au prix d'une complexité d'analyse plus élevée et de la perte de certaines propriétés de fermeture.
- Les **langages contextuels (Type-1)**, avec une mémoire linéairement bornée, capturent des dépendances croisées plus complexes tout en conservant la décidabilité du problème de l'appartenance, bien que ce dernier devienne intraitable en pratique.
- Les **langages récursivement énumérables (Type-0)**, avec une mémoire infinie, atteignent la limite de la calculabilité, capables de décrire tout ce qu'un algorithme peut générer, mais au détriment de la décidabilité de la plupart des problèmes, y compris le plus fondamental : celui de l'arrêt.

4.6.3 L'Héritage de la Hiérarchie de Chomsky en Informatique Fondamentale et Appliquée

Loin d'être une simple curiosité théorique, la hiérarchie de Chomsky est un cadre conceptuel qui a profondément influencé l'informatique.

- Elle fournit les fondements théoriques de la **théorie de la compilation**, où l'analyse lexicale est basée sur les automates finis (Type-3) et l'analyse syntaxique sur les automates à pile et les grammaires hors-contexte (Type-2).[6]
- Elle a guidé la **conception des langages de programmation**, en définissant ce qui peut être analysé efficacement.
- Elle établit un lien direct avec la **théorie de la complexité**, où les classes de complexité basées sur les ressources (temps, espace) trouvent des équivalents dans les modèles d'automates (par exemple, les langages contextuels correspondent à la classe d'espace non déterministe NSPACE(O(n))).[70]
- Ses concepts ont été adaptés à de nombreux autres domaines, de la **linguistique computationnelle** à la **bio-informatique**, pour modéliser et analyser des séquences et des structures complexes.[3]

En somme, la hiérarchie de Chomsky n'est pas seulement une classification des langages, mais une carte des frontières de la computation elle-même, délimitant ce qui est simple, ce qui est complexe, ce qui est calculable, et ce qui est hors

de portée de tout algorithme.

Ouvrages cités

1. Automate fini déterministe - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Automate_fini_d%C3%A9terministe
2. Automate fini non déterministe - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Automate_fini_non_d%C3%A9terministe
3. Formal language theory: refining the Chomsky hierarchy - PMC, dernier accès : septembre 21, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC3367686/
4. Théorie des langages - CNRS, dernier accès : septembre 21, 2025, https://perso.liris.cnrs.fr/christine.solnon/langages.pdf
5. Grammaires et Hiérarchie de Chomsky Grammaires : Séance 1 - Chamilo Grenoble INP, dernier accès : septembre 21, 2025, https://chamilo.grenoble-inp.fr/courses/ENSIMAG3MMTL1/document/Notes-de-cours-2page/Notes-cours1-grammaire.pdf
6. Hiérarchie de Chomsky: 'Types', 'Niveaux' | StudySmarter, dernier accès : septembre 21, 2025, https://www.studysmarter.fr/resumes/informatique/theorie-de-la-computation/hierarchie-de-chomsky/
7. 7 Théorie des Langages et Compilation - Grammaire et Classification de Chomsky - Partie 2, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=Y6Y6ktmlAuI
8. Chomsky hierarchy - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Chomsky_hierarchy
9. Comment classifie-t-on un langage ? (la hiérarchie de Chomsky) - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=qzhPecHmcZ4
10. Automata Theory - Chomsky Hierarchy - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=9HBF9StGpjg
11. CH.9 La hiérarchie de Chomsky - IGM, dernier accès : septembre 21, 2025, https://igm.univ-mlv.fr/~desar/cours/automates/ch9.pdf
12. Grammaires - Théorie des langages, dernier accès : septembre 21, 2025, https://pageperso.lis-lab.fr/~alexis.nasr/Ens/THL/slides_grammaires.pdf
13. CH.3 Propriétés des langages réguliers - IGM, dernier accès : septembre 21, 2025, https://igm.univ-mlv.fr/~desar/cours/automates/ch3.pdf
14. Grammaires formelles : Forme normale de Chomsky - LORIA, dernier accès : septembre 21, 2025, https://members.loria.fr/KFort/files/fichiers_cours/FNC.pdf
15. Grammaires formelles : Grammaires de type 1 et de type 0 - LORIA, dernier accès : septembre 21, 2025, https://members.loria.fr/KFort/files/fichiers_cours/Grammaires1Et0.pdf
16. Grammaire contextuelle - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Grammaire_contextuelle
17. Chapitre IV : Les automates d'états finis et les ... - DepInfoSkikda, dernier accès : septembre 21, 2025, https://www.depinfoskikda.com/upload/5185.pdf
18. Automates - info-llg.fr, dernier accès : septembre 21, 2025, https://info-llg.fr/option-mp/pdf/04.automates.pdf
19. Chapitre 7 Les automates - Gallium, dernier accès : septembre 21, 2025, http://gallium.inria.fr/~maranget/X/421/poly/automate.html
20. Chapter 3 DFA's, NFA's, Regular Languages, dernier accès : septembre 21, 2025, https://www.seas.upenn.edu/~cis2620/notes/cis262sl1-aut.pdf
21. Théorie des Langages Formels Chapitre 4 : Automates ... - MIS, dernier accès : septembre 21, 2025, https://home.mis.u-picardie.fr/~leve/Enseign/LF1718/chapitre4_LF.pdf

22. Automates finis déterministes - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=FZD0o_uk7AI
23. Automate fini non déterministe: Théorie, Applications | StudySmarter, dernier accès : septembre 21, 2025, https://www.studysmarter.fr/resumes/informatique/theorie-de-la-computation/automate-fini-non-deterministe/
24. Automate Fini Non-déterministe Théorème de Kleene - LISIC, dernier accès : septembre 21, 2025, https://www-lisic.univ-littoral.fr/~verel/TEACHING/17-18/ATI-L3/cm03.pdf
25. Equivalence of DFA and NFA • NFA's are usually easier to “program” in. • Surprisingly, for any NFA N there is a DFA D,, dernier accès : septembre 21, 2025, https://www.cs.ucr.edu/~jiang/cs150/slides4week2_FA+REX.pdf
26. Conversion from NFA to DFA - GeeksforGeeks, dernier accès : septembre 21, 2025, https://www.geeksforgeeks.org/theory-of-computation/conversion-from-nfa-to-dfa/
27. Conversion of NFA to DFA (Powerset/Subset Construction Example) - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=jMxuL4Xzi_A
28. Expressions régulières (ou RegExs) : définition, cas d'usages et exemples - Empirik, dernier accès : septembre 21, 2025, https://www.empirik.fr/nos-ressources/article/expressions-regulieres-ou-regex-definition-cas-dusages-et-exemples/
29. Expression régulière - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Expression_r%C3%A9guli%C3%A8re
30. Cours 4 Les Expressions Régulières | PDF - Scribd, dernier accès : septembre 21, 2025, https://fr.scribd.com/document/687795343/Cours-4-Les-expressions-regulieres
31. Regular expression - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Regular_expression
32. kleene.pdf - Théorie des langages, dernier accès : septembre 21, 2025, https://pageperso.lis-lab.fr/~alexis.nasr/Ens/THL/kleene.pdf
33. Théorie des automates et langages formels - Mathématiques Discrètes, dernier accès : septembre 21, 2025, http://www.discmath.ulg.ac.be/cours/main_autom.pdf
34. 2. Le théorème de Kleene - E. Desmontils, dernier accès : septembre 21, 2025, https://www.desmontils.net/emiage/Module209EMiage/c6/Ch6_2.htm
35. Théorème de Kleene - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_de_Kleene
36. Algorithme de Thompson — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Algorithme_de_Thompson
37. WikiMath » TP/Construction Des Automates Finis? - cours-info, dernier accès : septembre 21, 2025, https://cours-info.iut-bm.univ-fcomte.fr/pmwiki/pmwiki.php/TP/ConstructionDesAutomatesFinis
38. Thompson's construction - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Thompson%27s_construction
39. Théorie des Langages Formels Chapitre 3 : Théorème de ... - MIS, dernier accès : septembre 21, 2025, https://home.mis.u-picardie.fr/~leve/Enseign/LF1718/chapitre3_LF.pdf
40. Les grammaires "Context-Free" et la hiérarchie de Chomsky, dernier accès : septembre 21, 2025, https://pages.lip6.fr/Jean-Francois.Perrot/inalco/Automates/Cours18.html
41. Automates finis non-déterministes Tout langage régulier est reconnaissable par un automate fini, dernier accès : septembre 21, 2025, https://pages.lip6.fr/Jean-Francois.Perrot/inalco/Automates/Cours5.html
42. Langage rationnel - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Langage_rationnel
43. 5. Lemme de l'étoile - Arnaud Casteigts, dernier accès : septembre 21, 2025,

https://arnaudcasteigts.net/files/langages-5-cours.pdf
44. Notes de cours Informatique théorique - I (version 2.3.5, 14 février 2024) s.v.p. me signaler toute faute de frappe, grammaire, dernier accès : septembre 21, 2025, http://www.iro.umontreal.ca/~hahn/ift2105/Notes/cours.pdf
45. Comprendre le lemme de l'étoile (dit aussi lemme de pompage) - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=QkZ7J1JNpj4
46. Chapitre 2, dernier accès : septembre 21, 2025, https://v-assets.cdnsw.com/fs/Root/5m0tb-Chap_2.pdf
47. Langages non réguliers, dernier accès : septembre 21, 2025, https://www.site.uottawa.ca/~zaguia/csi3504/Chapitre10-3pages-2012.pdf
48. Grammaire non contextuelle - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Grammaire_non_contextuelle
49. Chapitre 4 Motivation Grammaire hors-contexte, dernier accès : septembre 21, 2025, http://www.iro.umontreal.ca/~mckenzie/ift2105/E10/P4horsnoir4.pdf
50. 6. Grammaires formelles - Arnaud Casteigts, dernier accès : septembre 21, 2025, https://arnaudcasteigts.net/files/langages-6-cours.pdf
51. Grammaires et Analyse Syntaxique - Cours 3 Grammaires algébriques - l'IRIF, dernier accès : septembre 21, 2025, https://www.irif.fr/~treinen/teaching/gas6/cours/cours03.handout1.pdf
52. Arbres associés aux grammaires CF, ambiguïté, dernier accès : septembre 21, 2025, https://pages.lip6.fr/Jean-Francois.Perrot/inalco/Automates/Cours19.html
53. Arbre de dérivation pour les grammaires algébriques (hors contexte) - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=Jh7y8SrpiX4
54. Grammaire et dérivations, dernier accès : septembre 21, 2025, https://perso.eleves.ens-rennes.fr/~afalq494/docs_info/fiches_info/Grammaire_et_derivations.pdf
55. Motivation Langages et grammaires hors-contextes Forme normale ..., dernier accès : septembre 21, 2025, https://pageperso.lis-lab.fr/~alexis.nasr/Ens/M17b/pcfg1.pdf
56. TD 5 - Chomsky et ambigüité 6. {w#w0 - LIRMM, dernier accès : septembre 21, 2025, https://www.lirmm.fr/~pvalicov/Cours/archives/Lyon/FDI/TD5.pdf
57. Chapitre 5 : Automates à pile Introduction, dernier accès : septembre 21, 2025, https://v-assets.cdnsw.com/fs/Root/e4s1j-CH5_AP.pdf
58. Automate à pile - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Automate_%C3%A0_pile
59. Théorie des langages, dernier accès : septembre 21, 2025, https://pageperso.lis-lab.fr/~alexis.nasr/Ens/THL/slides_ap.pdf
60. 2.3 Automates `a pile, dernier accès : septembre 21, 2025, http://www.linguist.univ-paris-diderot.fr/~amsili/Ens06/poly-li324-2.pdf
61. Automates à pile déterministes - l'IRIF, dernier accès : septembre 21, 2025, https://www.irif.fr/~carton/Enseignement/Complexite/ENS/Redaction/2007-2008/mathieu.barbin.pdf
62. Automates à pile déterministes - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=-326r91wLQ4
63. Grammaires hors-contexte - LORIA, dernier accès : septembre 21, 2025, https://members.loria.fr/KFort/files/fichiers_cours/GrammairesHC.pdf
64. Pourquoi les automates à pile sont-ils importants ? : r/compsci - Reddit, dernier accès : septembre 21, 2025, https://www.reddit.com/r/compsci/comments/e54n40/why_are_pushdown_automata_important/?tl=fr
65. Automate à pile - Définition et Explications - Techno-Science, dernier accès : septembre 21, 2025, https://www.techno-science.net/glossaire-definition/Automate-a-pile.html

66. CH.6 Propriétés des langages non contextuels - • 6.1 Le lemme de pompage - IGM, dernier accès : septembre 21, 2025, https://igm.univ-mlv.fr/~desar/cours/automates/ch6.pdf
67. Arbre de dérivation v/s arbre abstrait, dernier accès : septembre 21, 2025, https://pageperso.lis-lab.fr/~alexis.nasr/Ens/Compilation/arbre_abstrait.pdf
68. Machines linéairement bornées - l'IRIF, dernier accès : septembre 21, 2025, https://www.irif.fr/~carton/Enseignement/Complexite/ENS/Cours/lbm.html
69. Grammaire contextuelle - EPFL Graph Search, dernier accès : septembre 21, 2025, https://graphsearch.epfl.ch/fr/concept/6211
70. Automate linéairement borné — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Automate_lin%C3%A9airement_born%C3%A9
71. Linear bounded automaton - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Linear_bounded_automaton
72. NPTEL IIT Guwahati, dernier accès : septembre 21, 2025, https://archive.nptel.ac.in/content/storage2/courses/106103070/modules/module-14/lecture-1/topic-1/s19.htm
73. Context-Sensitive Grammars and Linear-Bounded Automata - SciSpace, dernier accès : septembre 21, 2025, https://scispace.com/pdf/context-sensitive-grammars-and-linear-bounded-automata-20nex5s215.pdf
74. Context Sensitive Grammars and Linear Bounded Automata - CSA – IISc Bangalore, dernier accès : septembre 21, 2025, https://www.csa.iisc.ac.in/~deepakd/atc-2021/ATC_Seminar_CSL.pdf
75. Linear Bounded Automata LBAS, dernier accès : septembre 21, 2025, https://ggn.dronacharya.info/Mtech_CSE/Downloads/QuestionBank/ISem/Mathematical_Foundation_Computer_Science/PPT/lecture21_Linear_bound_automata.pdf
76. LOGIQUE, THÉORIE DES MODÈLES, COMPLEXITÉ - » Tous les membres, dernier accès : septembre 21, 2025, https://webusers.imj-prg.fr/~antoine.chambert-loir/enseignement/2008-09/ltmc/ltmc.pdf
77. Grammaire d'arbres adjoints - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Grammaire_d%27arbres_adjoints

Chapitre 5 : Calculabilité et Décidabilité

Introduction : La Quête d'une Définition Formelle de l'Algorithme

Au cœur des sciences et du génie informatiques se trouve une notion à la fois intuitive et profondément énigmatique : celle d'**algorithme**. Depuis l'Antiquité, avec les procédures d'Euclide pour trouver le plus grand commun diviseur, jusqu'aux complexes algorithmes d'apprentissage machine qui façonnent aujourd'hui notre monde numérique, l'humanité a toujours cherché à formaliser des « procédures effectives » — des séquences d'étapes finies, non ambiguës et mécaniques permettant de résoudre une classe de problèmes.[1] Pendant des siècles, cette notion est demeurée informelle, reposant sur une compréhension partagée de ce que signifie « calculer ».

Cependant, au début du XXe siècle, cette fondation intuitive s'est avérée insuffisante. Des mathématiciens comme David Hilbert ont posé des questions fondamentales sur la nature même des mathématiques. L'une de ces questions, connue sous le nom d'***Entscheidungsproblem*** (le problème de la décision), demandait s'il existait une procédure mécanique capable de déterminer, pour n'importe quel énoncé logique, si cet énoncé est universellement valide.[2] Pour répondre à une telle question, il ne suffisait plus de savoir

comment calculer ; il devenait impératif de définir rigoureusement ce qu'un calcul *est*, et par extension, de tracer les frontières de ce qui est, en principe, *calculable*.

Cette quête d'une formalisation a conduit, dans les années 1930, à une explosion d'activité intellectuelle qui allait jeter les bases de l'informatique théorique. Des esprits brillants, travaillant indépendamment et depuis des perspectives radicalement différentes, ont proposé leurs propres modèles mathématiques du calcul. Alan Turing, un mathématicien britannique, a imaginé une machine abstraite, un automate d'une simplicité désarmante doté d'un ruban et d'une tête de lecture/écriture, capable d'exécuter des instructions élémentaires.[2] Alonzo Church, un logicien américain, a développé le calcul lambda, un système formel de réécriture de fonctions anonymes.[5] Kurt Gödel, en collaboration avec Jacques Herbrand et Stephen Kleene, a exploré la classe des fonctions récursives, définies sur les entiers naturels par des schémas de composition et de récursion.[6]

Ce chapitre se consacre à l'exploration de ces idées fondatrices. Dans un premier temps, nous nous attacherons à définir avec une rigueur mathématique la notion de calculabilité. Nous débuterons par le modèle le plus influent et le plus intuitif, la **machine de Turing**, en disséquant sa structure formelle et en démontrant sa puissance à travers la conception d'automates pour des problèmes non triviaux. Nous explorerons ensuite ses variantes, comme les machines multibandes et non déterministes, pour établir un résultat remarquable : ces extensions, bien que pratiques, n'ajoutent aucune puissance de calcul fondamentale. Nous nous tournerons ensuite vers les modèles alternatifs des **fonctions récursives** et du **calcul lambda**, pour aboutir à la **thèse de Church-Turing**, une pierre angulaire de notre discipline, qui postule que tous ces modèles, et de fait tout modèle de calcul « raisonnable », sont équivalents.[7]

Dans un second temps, armés d'une définition robuste de ce qui est calculable, nous nous aventurerons dans le territoire opposé : celui de l'**incalculable**. Nous poserons la question qui a motivé Turing : existe-t-il des problèmes bien définis pour lesquels aucun algorithme ne pourra jamais exister? La réponse, un « oui » retentissant, constitue l'une des plus grandes découvertes intellectuelles du XXe siècle. Nous fournirons une preuve complète et détaillée de l'indécidabilité du **problème de l'arrêt**, le résultat emblématique qui démontre une limite intrinsèque et infranchissable à la computation. À partir de ce point d'ancrage, nous introduirons la technique de la **réduction** pour propager cette indécidabilité à une vaste classe d'autres problèmes. Finalement, nous couronnerons cette exploration avec le **théorème de Rice**, un méta-théorème d'une puissance stupéfiante qui affirme que toute propriété non triviale du *comportement* d'un programme est indécidable.

Ce chapitre navigue donc entre la définition du pouvoir et la démonstration des limites. Il établit le cadre formel qui permet non seulement de concevoir des systèmes complexes, mais aussi de comprendre, avec une certitude mathématique, ce qu'ils ne pourront jamais accomplir.

5.1 Modèles de Calcul

L'objectif de cette section est de répondre de manière formelle et exhaustive à la question : « Qu'est-ce qu'une fonction calculable? » ou, de manière équivalente, « Qu'est-ce qu'un langage décidable? ». Pour ce faire, nous allons construire, brique par brique, les fondations de la théorie de la calculabilité. Nous présenterons trois formalismes majeurs qui, bien qu'issus de traditions intellectuelles distinctes — le modèle mécanique des machines de Turing, le modèle arithmétique des fonctions récursives et le modèle fonctionnel du calcul lambda —, convergent de manière spectaculaire vers une unique et robuste notion de calculabilité. Cette convergence n'est pas une coïncidence ; elle est l'argument le plus puissant en faveur de la thèse de Church-Turing, qui postule que nous avons bien capturé l'essence même de ce qu'est un algorithme.

5.1.1 Machines de Turing : Le Paradigme Impératif

Le modèle de la machine de Turing, proposé par Alan Turing en 1936, reste à ce jour le modèle de calcul le plus central et le plus intuitif. Sa force réside dans sa simplicité conceptuelle : il s'agit d'une idéalisation mathématique d'un calculateur humain qui manipule des symboles sur une feuille de papier.[10] En dépit de son minimalisme, ce modèle est suffisamment puissant pour simuler n'importe quel algorithme exécutable sur un ordinateur moderne.[11]

La Machine de Turing Standard : Définition Formelle et Mécanique du Calcul

Une machine de Turing est un automate qui opère sur un ruban infini. Ce ruban sert à la fois de mémoire, de support

pour l'entrée et de support pour la sortie. Une tête de lecture/écriture peut se déplacer le long de ce ruban, case par case, pour lire et écrire des symboles. Le comportement de la machine à chaque instant est déterminé par son état interne, tiré d'un ensemble fini d'états, et le symbole qu'elle lit actuellement.

Définition Formelle

La machine de Turing standard est un automate déterministe doté d'un unique ruban, infini dans les deux directions. Formellement, une machine de Turing (MT) est définie par un 7-uplet $M=(Q,\Sigma,\Gamma,\delta,q0,qaccept,qreject)$, où chaque composant a un rôle précis, comme détaillé dans la Table 5.1.

Table 5.1: Définition Formelle d'une Machine de Turing Déterministe Standard

Symbole	Description Formelle	Rôle et Explications
Q	Un ensemble fini d'**états**.	Représente la mémoire interne finie de la machine. À tout instant, la machine est dans un et un seul de ces états.
Σ	Un alphabet fini appelé **alphabet d'entrée**.	L'ensemble des symboles qui peuvent constituer le mot d'entrée initial. Cet alphabet ne contient pas le symbole blanc.
Γ	Un alphabet fini appelé **alphabet de ruban**.	L'ensemble de tous les symboles que la machine peut écrire sur le ruban. On a toujours $\Sigma \subseteq \Gamma$. Il contient un symbole spécial, le **symbole blanc** (noté $\sqcup$ ou B), qui remplit initialement toutes les cases du ruban à l'exception de celles contenant l'entrée.
δ	La **fonction de transition**. $\delta: Q \times \Gamma \to Q \times \Gamma \times \{L,R\}$	C'est le "programme" de la machine. Pour un état courant $q \in Q$ et un symbole lu $a \in \Gamma$, $\delta(q,a)=(q',b,D)$ signifie que la machine passe à l'état q', écrit le symbole b à la place de a, et

		déplace sa tête d'une case dans la direction D (L pour gauche, R pour droite).
q0	L'**état initial**. q0∈Q.	L'état dans lequel la machine commence son calcul.
qaccept	L'**état d'acceptation**. qaccept ∈Q.	Un état terminal unique. Si la machine entre dans cet état, le calcul s'arrête immédiatement et l'entrée est considérée comme **acceptée**.
qreject	L'**état de rejet**. qreject∈Q, avec qreject□=qaccept.	Un état terminal unique. Si la machine entre dans cet état, le calcul s'arrête immédiatement et l'entrée est considérée comme **rejetée**.

Sources: [13]

Mécanique du Calcul : Configurations et Transitions

Pour décrire formellement l'état global d'un calcul à un instant donné, nous utilisons la notion de **configuration** (parfois appelée description instantanée). Une configuration capture tout ce qui est nécessaire pour poursuivre le calcul : l'état courant de l'automate, le contenu complet du ruban, et la position de la tête de lecture/écriture.

Une configuration est typiquement représentée par une chaîne de la forme uqv, où :

- q∈Q est l'état courant de la machine.
- La chaîne uv représente la portion non blanche du ruban.
- La tête de lecture est positionnée sur le premier symbole de v. La chaîne u représente la partie du ruban à gauche de la tête.

Par exemple, la configuration a1a2q3a4a5 signifie que la machine est dans l'état q3, que le ruban contient la chaîne a1a2 a4a5 (entourée de symboles blancs), et que la tête de lecture pointe sur le symbole a4.[10]

Le calcul d'une machine de Turing est une séquence de configurations. Le passage d'une configuration à la suivante est dicté par la fonction de transition δ. Nous disons qu'une configuration C1 **conduit en une étape** à une configuration C2, noté C1⊢C2, si C2 peut être obtenue à partir de C1 par une seule application de la fonction δ.

Par exemple, si δ(q,b)=(p,c,R) et que la configuration actuelle est uaqbu', alors la configuration suivante sera uacpu'.[16] Des règles spécifiques gèrent les cas où la tête se déplace sur une case blanche, étendant potentiellement la partie non

blanche du ruban.

Calcul et Terminaison

Un **calcul** sur une entrée w est une séquence de configurations C0,C1,C2,… où C0 est la configuration initiale q0w (avec la tête sur le premier symbole de w), et Ci⊢Ci+1 pour tout i≥0.

Il y a trois issues possibles pour un calcul :

1. **Acceptation :** Le calcul atteint une configuration contenant l'état qaccept. La machine s'arrête et le mot d'entrée w est accepté.
2. **Rejet :** Le calcul atteint une configuration contenant l'état qreject. La machine s'arrête et le mot d'entrée w est rejeté.
3. **Boucle :** Le calcul ne s'arrête jamais. La machine n'atteint ni qaccept ni qreject. Dans ce cas, le mot d'entrée w est également considéré comme rejeté.

Cette distinction entre le rejet par arrêt et le rejet par boucle est fondamentale et sera au cœur de la distinction entre les langages récursifs et récursivement énumérables.

Conception de Machines de Turing : Un Exemple Exhaustif pour le Langage **{anbncn|n≥1}**

Pour illustrer la puissance et le fonctionnement concret des machines de Turing, nous allons concevoir une machine qui décide le langage L={anbncn|n≥1}. Ce langage est un exemple canonique de langage qui n'est pas hors-contexte (non "context-free"), ce qui signifie qu'il ne peut pas être reconnu par un automate à pile.[18] La nécessité de faire correspondre trois ensembles de symboles en nombre égal dépasse les capacités d'une pile unique. Une machine de Turing, avec sa capacité à se déplacer librement sur le ruban et à modifier son contenu, est nécessaire pour résoudre ce problème.

Stratégie de Conception

L'algorithme que notre machine mettra en œuvre est une stratégie de balayage et de marquage, souvent qualifiée de "zigzag".[21] L'idée est la suivante :

1. Balayer le ruban de gauche à droite pour vérifier que l'entrée est de la forme a...ab...bc...c.
2. Retourner au début du ruban.
3. Dans une boucle :
   a. Trouver le premier 'a' non marqué et le remplacer par un symbole de marquage (par exemple, X).
   b. Avancer pour trouver le premier 'b' non marqué et le remplacer par un autre marqueur (par exemple, Y).
   c. Avancer encore pour trouver le premier 'c' non marqué et le remplacer par un troisième marqueur (par exemple, Z).
   d. Retourner au début du ruban (juste après le dernier X marqué) pour recommencer le processus.
4. Si, à un moment, un 'b' ou un 'c' correspondant ne peut être trouvé, le mot est mal formé, et la machine doit rejeter.
5. Lorsque tous les 'a' ont été marqués, la machine doit effectuer une dernière vérification : s'assurer qu'il ne reste

plus de 'b' ou de 'c' non marqués sur le ruban. Si c'est le cas, la machine accepte. Sinon, elle rejette.

Implémentation Détaillée

Construisons formellement la machine Manbncn qui met en œuvre cette stratégie.

- Q={q0,q1,q2,q3,q4,qaccept,qreject}
  - q0: État initial. Cherche le premier 'a' à gauche.
  - q1: A marqué un 'a'. Cherche un 'b' correspondant en se déplaçant vers la droite.
  - q2: A marqué un 'b'. Cherche un 'c' correspondant en se déplaçant vers la droite.
  - q3: A marqué un 'c'. Revient au début pour le prochain cycle.
  - q4: Tous les 'a' ont été marqués. Vérifie qu'il ne reste plus de 'b' ni de 'c'.
- Σ={a,b,c}
- Γ={a,b,c,X,Y,Z,␣}
- q0 est l'état initial.
- qaccept et qreject sont les états d'acceptation et de rejet.

La fonction de transition δ est le cœur de la machine. Elle peut être représentée par un graphe d'états ou une table.

Graphe d'états de Manbncn

(Une description textuelle du graphe est fournie ici, car la génération d'images n'est pas possible. Chaque ligne représente une ou plusieurs transitions.)

- **De q0 :**
  - Sur un 'a' : Écrit X, va à droite, passe à q1. (Début d'un cycle de marquage).
  - Sur un 'Y' : Écrit Y, va à droite, passe à q4. (Tous les 'a' ont été traités, passe à la vérification finale).
  - Sur tout autre symbole (␣,b,c,X,Z): Va à qreject.
- **De q1 :**
  - Sur un 'a' : Écrit 'a', va à droite, reste en q1. (Saute les 'a' déjà vus).
  - Sur un 'Y' : Écrit 'Y', va à droite, reste en q1. (Saute les 'b' déjà marqués).
  - Sur un 'b' : Écrit Y, va à droite, passe à q2. (Marque le 'b' correspondant).
  - Sur tout autre symbole : Va à qreject.
- **De q2 :**
  - Sur un 'b' : Écrit 'b', va à droite, reste en q2. (Saute les 'b').
  - Sur un 'Z' : Écrit 'Z', va à droite, reste en q2. (Saute les 'c' déjà marqués).
  - Sur un 'c' : Écrit Z, va à gauche, passe à q3. (Marque le 'c' correspondant et commence le retour).
  - Sur tout autre symbole : Va à qreject.
- **De q3 :**
  - Sur 'a', 'b', 'Y', 'Z' : Écrit le même symbole, va à gauche, reste en q3. (Revient vers la gauche).
  - Sur un 'X' : Écrit X, va à droite, passe à q0. (A trouvé le début du cycle, se repositionne pour le suivant).
- **De q4 :**
  - Sur 'Y', 'Z' : Écrit le même symbole, va à droite, reste en q4. (Vérifie qu'il n'y a que des Y et des Z).
  - Sur un ␣ : Écrit ␣, va à droite, passe à qaccept. (A atteint la fin du ruban sans trouver de 'b' ou 'c' non marqués, accepte).
  - Sur tout autre symbole ('a', 'b', 'c') : Va à qreject.

Exemple d'Exécution sur l'Entrée "aabbcc"

Traçons les configurations clés du calcul de Manbncn sur l'entrée w=aabbcc.

1. q0aabbcc⊢Xq1abbcc (Marque le premier 'a').
2. ⊢Xaq1bbcc⊢XaYq2bcc (Trouve et marque le premier 'b').
3. ⊢XaYbq2cc⊢XaYbZq3c (Trouve et marque le premier 'c', commence le retour).
4. ⊢XaYq3bZc⊢⋯⊢Xq0aYbZc (Retourne jusqu'au X et se repositionne).
5. ⊢XXq1YbZc (Commence le deuxième cycle, marque le deuxième 'a').
6. ⊢XXYq1bZc⊢XXYYq2Zc (Trouve et marque le deuxième 'b').
7. ⊢XXYYZq2c⊢XXYYZZq3 (Trouve et marque le deuxième 'c', retourne).
8. ⊢⋯⊢XXq0YYZZ (Retourne et se repositionne).
9. ⊢XXYq4YZZ (Ne trouve plus de 'a', passe à l'état de vérification q4).
10. ⊢XXYYq4ZZ⊢XXYYZq4Z⊢XXYYZZq4⊔ (Traverse les Y et les Z).
11. ⊢XXYYZZ⊔qaccept (Trouve un blanc, le mot est accepté).

Ce processus détaillé illustre comment la machine utilise ses états et les symboles du ruban pour mémoriser sa progression dans l'algorithme de vérification. Tout écart par rapport à la structure anbncn (par exemple, "abcc") conduirait la machine à une configuration où aucune transition n'est définie depuis un état autre que qaccept, menant implicitement à qreject.

Variantes et Équivalence de Puissance : Machines Multibandes et Non Déterministes

Le modèle standard de la machine de Turing, bien que suffisant pour définir la calculabilité, peut être fastidieux à programmer. Plusieurs variantes ont été proposées pour faciliter la conception d'algorithmes. Il est crucial de se demander si ces variantes sont plus "puissantes", c'est-à-dire si elles peuvent résoudre des problèmes que la machine standard ne peut pas résoudre. La réponse, étonnamment, est non. Toutes les variantes "raisonnables" se sont avérées équivalentes en puissance de calcul au modèle standard. Cette robustesse est un argument majeur en faveur de la thèse de Church-Turing.

Nous allons prouver cette équivalence pour deux des variantes les plus importantes : les machines multibandes et les machines non déterministes. Ces preuves ne sont pas de simples affirmations d'existence ; ce sont des **algorithmes de simulation** constructifs. Elles montrent explicitement comment un modèle apparemment plus simple peut simuler un modèle plus complexe, jetant ainsi les bases de concepts comme la compilation et la virtualisation.

Machines de Turing Multibandes

Une machine de Turing à k rubans (ou MT multibande) est une généralisation naturelle du modèle standard. Elle possède k rubans infinis, chacun avec sa propre tête de lecture/écriture qui se déplace indépendamment.[14]

Formellement, la seule différence réside dans la fonction de transition, qui prend en compte les symboles lus sur les k rubans pour décider de l'action à effectuer sur chacun d'eux :

$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$

où S signifie "rester sur place" (Stationary). Une transition $\delta(q,(a1,\ldots,ak))=(p,(b1,\ldots,bk),(D1,\ldots,Dk))$ signifie que si la machine est dans l'état q et que la tête i lit le symbole ai pour chaque i, alors elle passe à l'état p, écrit le symbole bi sur le ruban i, et déplace la tête i dans la direction Di.
Théorème : Toute machine de Turing multibande a une machine de Turing à un seul ruban équivalente.

Preuve (par simulation constructive) :

Nous allons montrer comment une MT standard à un seul ruban, que nous appellerons S, peut simuler une MT à k rubans, que nous appellerons M.[23]

1. Encodage du contenu des rubans : Le ruban unique de S va stocker le contenu des k rubans de M. Pour ce faire, nous utilisons un symbole spécial, #, qui n'appartient pas à l'alphabet de ruban de M, comme séparateur. Le contenu du ruban de S aura la forme :

   #contenu1#contenu2#…#contenuk#

   où contenui est le contenu du i-ème ruban de M.23
2. **Marquage des positions des têtes :** Pour simuler les k têtes de M, S doit savoir où chaque tête virtuelle est positionnée. Pour cela, nous étendons l'alphabet de ruban de S. Pour chaque symbole $a\in\Gamma$, nous ajoutons un symbole "pointé" $a\dot{}$. Ce symbole $a\dot{}$ sur le ruban de S signifiera que la tête virtuelle correspondante de M pointe sur une case contenant le symbole a.[24] Il n'y aura donc à tout moment que
   k symboles pointés sur le ruban de S.
3. Simulation d'une étape de calcul de M : Pour simuler une seule transition de M, la machine S effectue la séquence d'opérations suivante :
   a. Lecture des symboles : S balaye son ruban de gauche à droite, du premier # au dernier, pour trouver les k symboles pointés. Elle mémorise ces k symboles dans ses états finis.
   b. Détermination de la transition : Une fois les k symboles lus, S connaît l'état courant de M (qu'elle mémorise aussi) et les symboles sous chaque tête. Elle peut donc consulter la fonction de transition δ de M pour déterminer le nouvel état, les k symboles à écrire, et les k directions de déplacement.
   c. Mise à jour du ruban : S effectue un second balayage de son ruban. Pour chaque symbole pointé $a\dot{}$ qu'elle rencontre, elle le remplace par le nouveau symbole b et déplace le "point" vers la gauche ou la droite, comme dicté par la transition de M. Par exemple, si la tête doit se déplacer à droite et que la case voisine contient un symbole c, S remplace $a\dot{}$ par b et c par $c\dot{}$.
   d. Gestion du dépassement de ruban : Si un déplacement de tête virtuelle nécessite de dépasser un séparateur # (ce qui signifie que le ruban de M s'agrandit), S doit décaler tout le contenu à droite du # d'une case pour faire de la place, en y insérant un symbole blanc pointé. C'est une opération fastidieuse mais purement mécanique.

Après ces étapes, le ruban de S représente la configuration des k rubans de M après la transition. S a ainsi simulé une étape de M. Elle peut alors répéter ce processus. Si M entre dans un état d'acceptation ou de rejet, S fait de même. Ce processus montre que S accepte un mot si et seulement si M l'accepte. La puissance de calcul est donc la même.

Machines de Turing Non Déterministes

Le non-déterminisme est un concept puissant en informatique théorique. Une machine de Turing non déterministe (MTN) est une MT qui, à une étape donnée, peut avoir plusieurs choix de transitions possibles.[28]

La fonction de transition d'une MTN n'est plus une fonction mais une relation. Pour un état et un symbole donnés, elle retourne un ensemble de triplets (nouvel état, symbole à écrire, direction) :

$\delta : Q \times \Gamma \to P(Q \times \Gamma \times \{L,R\})$

Le calcul d'une MTN n'est plus une séquence unique de configurations, mais un arbre de calcul. La racine de l'arbre est la configuration initiale. Chaque nœud a pour enfants les configurations accessibles en une étape.30
Une MTN **accepte** une entrée w s'il existe **au moins une** branche dans l'arbre de calcul qui mène à une configuration acceptante. Si toutes les branches s'arrêtent dans des états de rejet ou bouclent, l'entrée est rejetée. Le non-déterminisme peut être vu comme une forme de "devinette chanceuse" : la machine explore en parallèle toutes les possibilités et si l'une d'elles réussit, la réponse est "oui".[28]

Théorème : Toute machine de Turing non déterministe a une machine de Turing déterministe équivalente.

Preuve (par simulation constructive) :

Nous allons montrer qu'une MTD D peut simuler une MTN N.[20] La stratégie consiste pour

D à explorer l'arbre de calcul de N de manière systématique jusqu'à trouver une branche acceptante. Une recherche en largeur (Breadth-First Search) est idéale car elle garantit de trouver la branche acceptante la plus courte, si elle existe.

Pour implémenter cette recherche, nous utilisons une MTD à 3 rubans, D, qui, comme nous l'avons vu, est équivalente à une MTD standard.

1. **Ruban 1 (Ruban d'entrée) :** Ce ruban contient le mot d'entrée initial w et n'est jamais modifié. Il sert de référence.
2. **Ruban 2 (Ruban de simulation) :** Ce ruban est utilisé pour simuler l'exécution de N sur une branche particulière de son arbre de calcul.
3. **Ruban 3 (Ruban d'adresse) :** Ce ruban guide la simulation. Il contient une chaîne de caractères qui représente un chemin depuis la racine de l'arbre de calcul de N. Soit b le nombre maximal de choix pour n'importe quelle transition de N. Le ruban d'adresse contiendra une chaîne de l'alphabet $\{1,2,\ldots,b\}*$. Par exemple, la chaîne "213" signifie : "au premier pas, prendre le deuxième choix ; au deuxième pas, prendre le premier choix ; au troisième pas, prendre le troisième choix".[24]

L'algorithme de simulation de D est le suivant :

1. Initialement, le ruban 1 contient w. Les rubans 2 et 3 sont vides.
2. Copier w sur le ruban 2. Placer la chaîne "1" sur le ruban 3.
3. Boucle de simulation :
   a. Simuler N sur le ruban 2, en utilisant la chaîne sur le ruban 3 pour résoudre les choix non déterministes. À chaque étape de la simulation de N, D consulte le prochain symbole sur le ruban 3 pour savoir quelle transition choisir.
   b. Si la simulation atteint un état d'acceptation de N, alors D s'arrête et accepte.
   c. Si la chaîne d'adresse sur le ruban 3 est invalide (par exemple, elle demande un choix qui n'existe pas) ou si la branche de calcul correspondante s'arrête dans un état de rejet, ou si la chaîne d'adresse a été entièrement consommée, la simulation de cette branche est terminée.
   d. Remplacer la chaîne sur le ruban 3 par la chaîne lexicographiquement suivante (par exemple, "1", "2",..., "b", "11", "12",...).

e. Réinitialiser le ruban 2 avec le contenu du ruban 1 (le mot d'entrée w).
f. Retourner à l'étape 3a pour simuler la nouvelle branche.

Si N accepte w, il existe une branche acceptante dans son arbre de calcul. La recherche en largeur de D finira par explorer cette branche, et D acceptera. Si N n'accepte pas w, aucune branche n'est acceptante, et D continuera sa recherche indéfiniment (elle bouclera). Ainsi, D accepte un mot si et seulement si N l'accepte.

Cette preuve a une implication profonde qui anticipe la théorie de la complexité. La simulation d'une MTN par une MTD peut entraîner une explosion exponentielle du temps de calcul. Si une MTN résout un problème en temps polynomial, la MTD simulatrice pourrait nécessiter un temps exponentiel. La question de savoir si cette explosion est inévitable est au cœur du célèbre problème P = NP.[20]

Langages Récursifs et Récursivement Énumérables : La Hiérarchie de la Calculabilité

L'existence de machines de Turing qui peuvent boucler sur certaines entrées nous amène à définir deux classes de langages distinctes, qui forment la première strate de la hiérarchie de la calculabilité.

Définition Formelle

- Un langage L est dit **récursivement énumérable (RE)**, ou **Turing-reconnaissable**, s'il existe une machine de Turing M telle que L(M)=L. Cela signifie que pour tout mot w∈L, la machine M s'arrête et accepte. Pour un mot w∈/L, la machine M peut soit s'arrêter et rejeter, soit boucler indéfiniment.[24] Une telle machine est appelée un **reconnaisseur**.
- Un langage L est dit **récursif**, ou **Turing-décidable** (ou simplement **décidable**), s'il existe une machine de Turing M qui s'arrête sur *toutes* les entrées et telle que L(M)=L. Pour tout mot w, le calcul de M sur w s'arrête, soit en acceptant (si w∈L), soit en rejetant (si w∈/L).[10] Une telle machine est appelée un **décideur**.

Relation et Propriétés

De par ces définitions, il est clair que la classe des langages récursifs est un sous-ensemble de la classe des langages récursivement énumérables. Tout décideur est un reconnaisseur, mais la réciproque n'est pas vraie, comme nous le prouverons de manière spectaculaire dans la section 5.2.

Une propriété fondamentale relie ces deux classes au concept de complémentation de langage.

Théorème : Un langage L est récursif si et seulement si L et son complément L sont tous deux récursivement énumérables.

Preuve :

(⇒) Si L est récursif, il est décidé par une MT M qui s'arrête toujours. L est donc trivialement RE. Pour décider L, il suffit de construire une machine M’ qui simule M et inverse ses états d'acceptation et de rejet. Puisque M s'arrête toujours, M’ s'arrête aussi toujours, donc L est récursif, et par conséquent RE.[10]

(⇐) Supposons que L soit reconnu par une MT M1 et que L soit reconnu par une MT M2. Nous pouvons construire un décideur M pour L comme suit :
M, sur une entrée w, utilise deux rubans pour simuler M1 sur w et M2 sur w en parallèle. À chaque étape, M exécute une transition de M1 puis une transition de M2.
Puisque tout mot w appartient soit à L soit à L, l'une des deux machines M1 ou M2 doit nécessairement s'arrêter et accepter.

- Si M1 accepte, M s'arrête et accepte.
- Si M2 accepte, M s'arrête et rejette.
  Comme l'une de ces deux situations doit se produire, la machine M s'arrête sur toutes les entrées. Elle décide donc le langage L, qui est par conséquent récursif.13

5.1.2 Modèles Alternatifs : Une Convergence Fondamentale

La machine de Turing, avec son approche mécanique et impérative, n'est qu'une des manières de formaliser le calcul. D'autres modèles, issus de traditions mathématiques différentes, offrent des perspectives complémentaires. Leur étude révèle un fait remarquable : malgré leurs différences apparentes, les modèles de calcul les plus puissants convergent tous vers la même classe de fonctions calculables.

Les Fonctions μ-Récursives : Le Calcul comme Arithmétique

Cette approche, développée par Gödel, Herbrand et Kleene, définit la calculabilité non pas en termes de machine, mais en termes de fonctions sur les nombres naturels (N). L'idée est de partir d'un ensemble de fonctions de base, incontestablement calculables, et de définir des opérateurs qui permettent de construire des fonctions plus complexes.

Fonctions Primitives Récursives

La classe des fonctions primitives récursives constitue une première étape importante. Elle capture une vaste classe de fonctions calculables qui terminent toujours.

- **Fonctions de base :**
  1. **Fonction Zéro :** Z(x)=0.
  2. **Fonction Successeur :** S(x)=x+1.
  3. **Fonctions de Projection :** Pik(x1,…,xk)=xi (permet de sélectionner un argument).[35]
- **Opérateurs de construction :**
  1. **Composition :** Si g1,…,gm sont des fonctions à k variables et h est une fonction à m variables, alors leur composition f(x1,…,xk)=h(g1(x1,…,xk),…,gm(x1,…,xk)) est également une fonction constructible.[35]
  2. Récursion Primitive : Cet opérateur formalise la récurrence. Si g est une fonction à k variables et h une fonction à k+2 variables, on peut définir une nouvelle fonction f à k+1 variables comme suit :

     f(x1,…,xk,0)=g(x1,…,xk)

f(x1,…,xk,y+1)=h(x1,…,xk,y,f(x1,…,xk,y))

Le cas de base est donné par g, et le cas récursif par h, qui utilise la valeur de la fonction à l'étape précédente.35

À l'aide de ces outils, on peut construire la plupart des fonctions arithmétiques usuelles. Par exemple, l'addition peut être définie par récursion primitive sur le second argument :

- add(x,0)=x=P11(x)
- add(x,y+1)=S(add(x,y))=S(P33(x,y,add(x,y)))

De même, la multiplication se définit par récursion sur l'addition, la factorielle sur la multiplication, etc..[36] Les fonctions primitives récursives sont toutes des fonctions

totales, c'est-à-dire qu'elles sont définies pour tous les entiers en entrée.

Limites de la Récursion Primitive : La Fonction d'Ackermann

Pendant un temps, on a pu croire que les fonctions primitives récursives capturaient toutes les fonctions calculables. Cependant, en 1928, Wilhelm Ackermann a construit une fonction qui, bien qu'intuitivement calculable, ne peut pas être définie par récursion primitive. La version la plus courante est la fonction d'Ackermann-Péter :
$$ A(m, n) = \begin{cases} n+1 & \text{si } m=0 \ A(m-1, 1) & \text{si } m > 0 \text{ et } n=0 \ A(m-1, A(m, n-1)) & \text{si } m > 0 \text{ et } n > 0 \end{cases} $$
Cette fonction croît à une vitesse phénoménale. Par exemple, A(4,2) est un nombre de 19 729 chiffres. Il peut être prouvé que la fonction d'Ackermann croît plus vite que n'importe quelle fonction primitive récursive. Par conséquent, elle ne peut pas être elle-même primitive récursive.35 Cela démontre que la classe des fonctions primitives récursives est un sous-ensemble strict de l'ensemble des fonctions calculables.
Fonctions μ-Récursives

Pour capturer toutes les fonctions calculables, y compris la fonction d'Ackermann, il faut ajouter un opérateur plus puissant, capable de simuler des boucles non bornées (l'équivalent d'une boucle while). C'est le rôle de l'opérateur de **minimisation non bornée (μ)**.

- **Opérateur de Minimisation (μ) :** Soit g(x1,…,xk,z) une fonction totale. La fonction f(x1,…,xk)=μz[g(x1,…,xk,z)=0] est définie comme "le plus petit entier z≥0 tel que g(x1,…,xk,z)=0".

Pour trouver ce z, un algorithme calcule successivement g(…,0), g(…,1), g(…,2), etc., jusqu'à trouver une valeur qui s'annule. Si une telle valeur n'existe pas, le calcul ne s'arrête jamais. Cet opérateur introduit donc la possibilité de définir des **fonctions partielles**, c'est-à-dire des fonctions qui ne sont pas définies pour toutes les entrées.[35]

La classe des **fonctions μ-récursives** (ou simplement **fonctions récursives**) est le plus petit ensemble de fonctions contenant les fonctions de base et fermé sous la composition, la récursion primitive et la minimisation non bornée. Il s'avère que cette classe correspond exactement à la classe des fonctions calculables par une machine de Turing.[17]

Le calcul lambda (ou λ-calcul), introduit par Alonzo Church, offre une perspective radicalement différente. Il s'agit d'un formalisme minimaliste qui ne contient qu'une seule entité : la fonction. Tout — les nombres, les booléens, les structures de données — est encodé comme une fonction. Le calcul n'est pas une exécution sur une machine, mais un processus de réécriture symbolique d'expressions.[5]

Fondements

Le langage du λ-calcul est constitué de **λ-termes**, définis inductivement :

- **Variable :** Une variable, comme x, est un λ-terme.
- **Abstraction :** Si M est un λ-terme et x est une variable, alors (λx.M) est un λ-terme. Il représente la fonction anonyme qui prend un argument x et retourne M.
- **Application :** Si M et N sont des λ-termes, alors (MN) est un λ-terme. Il représente l'application de la fonction M à l'argument N.

Dans une abstraction λx.M, la variable x est dite **liée**. Une variable qui n'est pas liée est dite **libre**.[5] Cette distinction est cruciale pour définir correctement le mécanisme de calcul.

Règles de Conversion

Le calcul dans le λ-calcul est gouverné par un ensemble de règles de réécriture, appelées conversions.

- **α-conversion (renommage) :** Cette règle stipule que le nom d'une variable liée n'a pas d'importance. On peut le changer, à condition de ne pas créer de conflit avec d'autres variables. Par exemple, la fonction identité λx.x est équivalente à λy.y. Formellement : $\lambda x.M \to \alpha \lambda y.M[x:=y]$, où M[x:=y] est la substitution de x par y dans M.[43]
- **β-réduction (application) :** C'est le moteur du calcul. Elle formalise l'application d'une fonction à un argument. L'expression (λx.M)N se réduit en remplaçant toutes les occurrences libres de x dans le corps M par l'argument N. Formellement : $(\lambda x.M)N \to \beta M[x:=N]$.[43] Par exemple,
  $(\lambda x.x+1)3 \to \beta 3+1$.
- **η-conversion (extensionnalité) :** Cette règle exprime le principe d'extensionnalité : deux fonctions sont égales si elles produisent le même résultat pour tous les arguments. Elle permet de simplifier une expression de la forme λx.(Mx) en M, à condition que x ne soit pas une variable libre de M.[43]

Encodage de Données (Entiers de Church)

La puissance du λ-calcul réside dans sa capacité à tout représenter en tant que fonction. L'encodage des nombres naturels, connu sous le nom d'**entiers de Church**, en est l'exemple le plus célèbre.[43] L'idée est de représenter un nombre

n par une fonction qui prend deux arguments, une fonction f et une valeur x, et qui applique n fois la fonction f à la valeur x.

- 0:=λf.λx.x (applique f zéro fois)
- 1:=λf.λx.fx (applique f une fois)
- 2:=λf.λx.f(fx) (applique f deux fois)

- En général, $n^{-} := \lambda f.\lambda x.fn(x)$.[49]

Avec cet encodage, les opérations arithmétiques deviennent des opérations sur des fonctions de haut niveau :

- Successeur : La fonction SUCC prend un entier de Church n^{-} et retourne n+1. Elle le fait en ajoutant une application de f :

  SUCC:=λn.λf.λx.f(nfx)
- Addition : L'addition de m et n consiste à appliquer m fois le successeur à n.

  PLUS:=λm.λn.m SUCC n
- Multiplication : La multiplication de m et n consiste à composer les fonctions "appliquer f m fois" et "appliquer f n fois".

  MULT:=λm.λn.λf.m(nf)

Le fait que des concepts aussi fondamentaux que les nombres et l'arithmétique puissent être construits à partir d'un système aussi simple démontre l'extraordinaire expressivité du λ-calcul. Church a prouvé que les fonctions calculables dans le λ-calcul sont précisément les fonctions μ-récursives, et donc équivalentes aux fonctions calculables par machine de Turing.

5.1.3 La Thèse de Church-Turing : Un Fondement pour les Sciences Informatiques

Nous avons exploré trois formalismes de calcul distincts :

1. Les **machines de Turing**, un modèle mécanique.
2. Les **fonctions μ-récursives**, un modèle arithmétique.
3. Le **calcul lambda**, un modèle de réécriture fonctionnelle.

Le résultat le plus remarquable de cette exploration est que ces trois approches, développées indépendamment pour capturer la notion de "procédure effective", définissent *exactement la même classe de fonctions calculables*. Cette convergence n'est pas une simple coïncidence ; elle est l'évidence la plus forte en faveur d'une hypothèse fondamentale qui sous-tend toute l'informatique : la thèse de Church-Turing.

Énoncé de la Thèse

La **thèse de Church-Turing** peut être formulée comme suit :

> La notion intuitive de fonction calculable par un algorithme (ou une procédure effective) coïncide avec la notion formelle de fonction calculable par une machine de Turing.

Puisque les machines de Turing, les fonctions μ-récursives et le λ-calcul sont formellement équivalents, la thèse affirme que n'importe lequel de ces modèles capture précisément et complètement notre concept intuitif de calcul.[6]

Arguments en Faveur de la Thèse

1. **Convergence et Robustesse :** Comme mentionné, le fait que des modèles si différents aboutissent au même résultat est un argument puissant. Si un nouveau modèle de calcul est proposé, il s'avère presque toujours être soit moins puissant (comme les automates finis), soit exactement équivalent aux machines de Turing.[9] Cette robustesse suggère qu'une classe naturelle et fondamentale de problèmes a été identifiée.
2. **Analyse de Turing du Calcul Humain :** L'argument de Turing en faveur de son modèle n'était pas seulement sa simplicité, mais une analyse philosophique des étapes élémentaires qu'un être humain (un "computer", à l'époque) effectue lors d'un calcul avec un crayon et du papier. Turing a argumenté que toute procédure mécanique peut être décomposée en un ensemble fini d'opérations atomiques (lire un symbole, écrire un symbole, changer d'état de concentration, déplacer son attention) que sa machine peut simuler.[54]

Statut de "Thèse" et Non de "Théorème"

Il est crucial de comprendre pourquoi il s'agit d'une thèse et non d'un théorème mathématique. Un théorème relie des objets formels au sein d'un système axiomatique. La thèse de Church-Turing, elle, cherche à établir un pont entre un concept intuitif et informel (la "procédure effective") et un objet mathématique formel (la "machine de Turing"). On ne peut pas prouver mathématiquement qu'une définition formelle capture parfaitement une idée intuitive. La thèse est donc une affirmation que l'on accepte sur la base de l'évidence accumulée, un peu comme un axiome qui fonde une théorie physique en reliant les mathématiques au monde observable.[56]

Implications et Défis Modernes

L'acceptation de la thèse de Church-Turing est ce qui nous donne le droit de faire des déclarations universelles sur la calculabilité. Lorsque nous prouverons dans la section suivante que le problème de l'arrêt est "indécidable", nous ne voudrons pas dire seulement "indécidable par une machine de Turing", mais bien "indécidable par *n'importe quel algorithme imaginable*". La thèse est le fondement qui justifie cette généralisation.[12]

Cependant, il est important de noter que la thèse concerne la **calculabilité** (ce qui peut être calculé en principe, avec des ressources de temps et d'espace illimitées) et non l'**efficacité**. L'avènement de modèles comme l'**ordinateur quantique** a remis en question la *version efficace* de la thèse de Church-Turing (parfois appelée principe de Church-Turing-Deutsch). Des algorithmes quantiques comme celui de Shor peuvent résoudre certains problèmes, comme la factorisation d'entiers, de manière exponentiellement plus rapide qu'on ne le pense possible sur un ordinateur classique. Cela ne viole pas la thèse originale — une machine de Turing classique peut toujours simuler un ordinateur quantique, bien que de manière extrêmement lente — mais cela suggère que notre compréhension de ce qui est "calculable efficacement" pourrait dépendre des lois de la physique.[7]

5.2 Indécidabilité : Les Limites Infranchissables du Calcul

Après avoir établi une définition formelle et robuste de ce qui est calculable, nous abordons maintenant une question plus profonde et, à bien des égards, plus surprenante : tout est-il calculable? Existe-t-il des problèmes formulés avec une parfaite précision mathématique, pour lesquels aucun algorithme, aussi ingénieux soit-il, ne pourra jamais fournir de solution?

La réponse, découverte par Church et Turing, est un oui catégorique. Il existe des abîmes d'incalculabilité, des problèmes dont l'insolubilité n'est pas une question de technologie ou d'efficacité, mais une limitation fondamentale et permanente de la logique et de la computation. Cette section est consacrée à la cartographie de ce territoire de l'impossible. Nous commencerons par prouver l'existence d'un premier problème indécidable, le célèbre **problème de l'arrêt**. Ce résultat servira de pierre de Rosette, un point de référence de l'indécidabilité à partir duquel nous utiliserons la puissante technique de la **réduction** pour démontrer l'insolubilité de nombreux autres problèmes. Enfin, nous atteindrons le sommet de cette exploration avec le **théorème de Rice**, un résultat d'une généralité saisissante qui trace une ligne de démarcation claire entre les questions syntaxiques (faciles) et les questions sémantiques (impossibles) que l'on peut poser sur les programmes.

5.2.1 Le problème de l'arrêt (Halting Problem)

Le problème de l'arrêt est sans doute le résultat le plus célèbre de l'informatique théorique. Intuitivement, il pose la question suivante : peut-on écrire un programme qui, étant donné un autre programme et une entrée, peut déterminer si ce programme finira par s'arrêter ou s'il bouclera à l'infini? Turing a prouvé que la réponse est non. Un tel analyseur de programmes universel et infaillible ne peut exister.

Définition Formelle du Langage de l'Acceptation **ATM**

Pour prouver ce résultat formellement, il est plus commode de travailler avec une variante légèrement différente mais équivalente, appelée le **problème de l'acceptation**. Au lieu de demander si une machine s'arrête, nous demandons si elle accepte son entrée.

Nous formalisons ce problème à travers la notion de langage. Pour cela, nous avons besoin d'un moyen de représenter une machine de Turing M et une entrée w comme une seule chaîne de caractères. Nous utilisons la notation $\langle M,w \rangle$ pour désigner un encodage raisonnable de la paire (M,w) en une chaîne unique sur un alphabet fixe (par exemple, binaire).

Définition : Le langage de l'acceptation, noté ATM, est l'ensemble des paires $\langle M,w \rangle$ où M est une machine de Turing et w est une chaîne que M accepte.

$ATM=\{\langle M,w \rangle | M \text{ est une MT qui accepte } w\}$

Sources: 13
Avant de prouver que ATM est indécidable, montrons d'abord qu'il est au moins Turing-reconnaissable.

Théorème : Le langage ATM est récursivement énumérable (Turing-reconnaissable).

Preuve :
Pour reconnaître ATM, nous pouvons construire une machine de Turing spéciale, souvent appelée Machine de Turing

Universelle (UTM), notée U. Cette machine U prend en entrée l'encodage ⟨M,w⟩. Son fonctionnement est le suivant :

1. U vérifie que l'entrée est un encodage valide d'une MT M et d'une chaîne w.
2. U simule l'exécution de M sur l'entrée w. Elle utilise une partie de son ruban pour stocker la configuration courante de M (son état, le contenu de son ruban, la position de sa tête) et une autre partie pour stocker la description de M (sa fonction de transition).
3. À chaque étape, U consulte la description de M pour déterminer la prochaine transition à effectuer et met à jour la configuration simulée.
4. Si la simulation de M atteint l'état d'acceptation, U s'arrête et accepte.
5. Si la simulation de M atteint l'état de rejet, U s'arrête et rejette.

Si M accepte w, la simulation sur U finira par atteindre l'état d'acceptation, et U acceptera ⟨M,w⟩. Si M rejette w en s'arrêtant, U rejettera. Cependant, si M boucle sur w, la simulation sur U bouclera également. Par conséquent, U accepte précisément les chaînes de ATM mais peut boucler sur les autres. U est donc un reconnaisseur pour ATM, ce qui prouve que ATM est récursivement énumérable.[13]

Preuve Détaillée de l'Indécidabilité par l'Argument de la Diagonalisation

Nous allons maintenant prouver le résultat principal : ATM n'est pas décidable. La preuve est un chef-d'œuvre de logique, une preuve par l'absurde qui utilise l'**argument de la diagonalisation** de Cantor. L'idée est de supposer qu'un décideur pour ATM existe, puis d'utiliser ce décideur pour construire une nouvelle machine "paradoxale" dont l'existence même mène à une contradiction logique inéluctable.

Étape 1 : Hypothèse par l'Absurde

Supposons, pour les besoins de la contradiction, que le langage ATM est décidable. Cela signifie qu'il existe une machine de Turing H (pour "Hypothétique") qui est un décideur pour ATM. Par définition d'un décideur, H s'arrête sur toutes les entrées. Son comportement est le suivant :
$$ H(\langle M, w \rangle) = \begin{cases} \text{accepte} & \text{si } M \text{ accepte } w \\ \text{rejette} & \text{si } M \text{ n'accepte pas } w \text{ (c.-à-d. rejette ou boucle)} \end{cases} $$
Sources: 32
Étape 2 : Construction de la Machine Contradictoire D

En utilisant notre décideur hypothétique H comme une "boîte noire" ou une sous-routine, nous construisons une nouvelle machine de Turing, que nous nommerons D (pour "Diagonale" ou "Démoniaque"). La machine D est conçue pour se comporter de manière perverse.

- **Entrée de D :** D prend en entrée l'encodage d'une seule machine de Turing, ⟨M⟩.
- **Fonctionnement de D :**
  1. Sur l'entrée ⟨M⟩, D construit la chaîne ⟨M,⟨M⟩⟩. Elle utilise donc la description de M comme sa propre entrée.
  2. D exécute ensuite le décideur H sur cette entrée construite, ⟨M,⟨M⟩⟩.
  3. D observe le résultat de H et fait délibérément le contraire :
     - Si H accepte ⟨M,⟨M⟩⟩ (ce qui signifie que M accepterait sa propre description en entrée), alors D **rejette** ⟨M⟩.

- Si H rejette ⟨M,⟨M⟩⟩ (ce qui signifie que M n'accepterait pas sa propre description), alors D **accepte** ⟨M⟩.

Puisque H est un décideur et s'arrête toujours, et que les autres étapes de D sont de simples manipulations de chaînes, D est également un décideur. Elle s'arrête sur toutes les entrées ⟨M⟩ et retourne soit "accepte", soit "rejette".[32]

Étape 3 : L'Argument de la Diagonalisation – Le Paradoxe

Le piège se referme lorsque nous posons la question fatidique : que se passe-t-il si nous donnons à la machine D sa propre description, ⟨D⟩, comme entrée? Puisque D est une machine de Turing et un décideur, elle doit avoir un comportement bien défini sur cette entrée. Examinons les deux seules possibilités logiques.

- **Cas 1 : Supposons que D accepte son entrée ⟨D⟩.**
  - Regardons la définition de D. Pour que D accepte son entrée ⟨M⟩=⟨D⟩, il faut que la simulation de H sur l'entrée ⟨D,⟨D⟩⟩ ait retourné "rejette".
  - Mais que signifie le fait que H rejette ⟨D,⟨D⟩⟩? Par définition de H, cela signifie que la machine D n'accepte *pas* l'entrée ⟨D⟩.
  - Nous avons donc atteint une contradiction : D accepte ⟨D⟩⟹D n'accepte pas ⟨D⟩.
- **Cas 2 : Supposons que D rejette son entrée ⟨D⟩.**
  - Regardons à nouveau la définition de D. Pour que D rejette son entrée ⟨M⟩=⟨D⟩, il faut que la simulation de H sur l'entrée ⟨D,⟨D⟩⟩ ait retourné "accepte".
  - Mais que signifie le fait que H accepte ⟨D,⟨D⟩⟩? Par définition de H, cela signifie que la machine D *accepte* l'entrée ⟨D⟩.
  - Nous avons de nouveau atteint une contradiction : D rejette ⟨D⟩⟹D accepte ⟨D⟩.

Sources: [62]

Pour mieux visualiser, imaginons un tableau infini où les lignes sont indexées par toutes les machines de Turing (M1,M2 ,…) et les colonnes par tous les encodages de machines de Turing (⟨M1⟩,⟨M2⟩,…). La case (i,j) contient "accepte" si Mi accepte ⟨Mj⟩ et "rejette" sinon. Notre machine hypothétique H peut calculer n'importe quelle case de ce tableau. La machine D est construite pour regarder la **diagonale** de ce tableau (les cases (i,i)) et pour avoir le comportement inverse. Le paradoxe survient lorsque nous essayons de situer D elle-même dans ce tableau. Si D est la k-ième machine, Mk, alors son comportement sur l'entrée ⟨D⟩=⟨Mk⟩ doit être, par construction, l'inverse de ce qui se trouve dans la case (k,k), ce qui est logiquement impossible.[66]

Étape 4 : Conclusion

Les deux cas possibles mènent à une absurdité logique. La seule conclusion possible est que notre prémisse initiale était fausse. La machine D ne peut pas exister. Mais la construction de D était parfaitement légitime, à une seule condition : l'existence de la sous-routine H. Par conséquent, c'est l'hypothèse de l'existence du décideur H qui doit être rejetée.

Conclusion : Le langage ATM n'est pas décidable. Il n'existe aucun algorithme capable de résoudre le problème de l'acceptation pour toutes les machines de Turing et toutes les entrées.[62]

Ce résultat est d'une portée immense. Il ne s'agit pas d'un échec temporaire de notre ingéniosité, mais d'une barrière fondamentale. Il existera toujours des programmes dont le comportement est impossible à prédire de manière algorithmique.

5.2.2 Réductions et preuves d'indécidabilité

L'argument de la diagonalisation est puissant mais complexe à manier directement pour chaque nouveau problème. Heureusement, une fois que nous avons établi un premier problème indécidable comme ATM, nous pouvons l'utiliser comme un "levier" pour prouver l'indécidabilité d'une multitude d'autres problèmes. La technique qui permet cette propagation de l'indécidabilité est la **réduction**.

Le Concept de Réductibilité par Mappage (≤m)

L'idée intuitive d'une réduction est de montrer qu'un problème A peut être "transformé" en un problème B de telle sorte qu'une solution pour B nous donnerait directement une solution pour A. Si une telle transformation existe, cela implique que le problème A n'est "pas plus difficile" que le problème B. Par contraposée, si A est connu pour être difficile (c'est-à-dire indécidable), alors B doit l'être aussi.[69]

Nous formalisons cette idée avec la **réductibilité par mappage** (ou réduction "many-one").

Définition : Un langage A est réductible par mappage au langage B, noté $A \leq_m B$, s'il existe une fonction calculable f (c'est-à-dire, calculable par une machine de Turing qui s'arrête toujours) qui transforme une chaîne w en une chaîne f(w), telle que :

$w \in A \Leftrightarrow f(w) \in B$

La fonction f est appelée la réduction de A à B.69
Le théorème suivant est l'outil principal que nous utiliserons.

Théorème : Si $A \leq_m B$ et A est indécidable, alors B est indécidable.

Preuve :
Nous raisonnons par l'absurde. Supposons que $A \leq_m B$, que A est indécidable, mais que B est décidable.
Puisque B est décidable, il existe un décideur pour B, appelons-le MB. Nous pouvons alors construire un décideur pour A, que nous nommerons MA, comme suit :
MA = "Sur une entrée w :

1. Calculer f(w) en utilisant la machine de Turing qui calcule la fonction de réduction f. Puisque f est calculable, cette étape s'arrête toujours.
2. Exécuter le décideur MB sur le résultat f(w). Puisque MB est un décideur, cette étape s'arrête toujours.
3. Si MB accepte f(w), alors MA accepte w.
4. Si MB rejette f(w), alors MA rejette w."

Cette machine MA s'arrête sur toutes les entrées. De plus, par la définition de la réduction, $w \in A \Leftrightarrow f(w) \in B$. Et par la

définition de MB, MB accepte f(w)⇔f(w)∈B. Donc, MA accepte w⇔w∈A.
Nous avons construit un décideur pour A. Mais ceci contredit notre hypothèse que A est indécidable. Par conséquent, notre supposition que B est décidable doit être fausse. B est donc indécidable.69

Preuves d'Indécidabilité par Réduction : **HALTTM***,* **ETM***, et* **EQTM**

Armés de la réductibilité, nous pouvons maintenant établir une "carte de l'indécidabilité", montrant comment l'insolubilité de ATM se propage à d'autres problèmes fondamentaux.

Le Problème de l'Arrêt (HALTTM)

Définissons formellement le langage du problème de l'arrêt :

HALTTM={⟨M,w⟩|M est une MT qui s’arreˆte sur l’entre´e w}
Théorème : HALTTM est indécidable.

Preuve : Nous allons montrer que ATM≤mHALTTM. Pour ce faire, nous devons construire une fonction calculable f qui transforme une instance ⟨M,w⟩ de ATM en une instance ⟨M’,w’⟩ de HALTTM de telle sorte que M accepte w si et seulement si M’ s'arrête sur w’.

Voici la construction de la fonction de réduction f(⟨M,w⟩)=⟨M’,w⟩ :

1. f prend en entrée la description ⟨M,w⟩.
2. f produit en sortie la description d'une nouvelle machine M’.
3. La machine M’ est conçue comme suit :
   M’ = "Sur une entrée x (que M’ ignorera) :
   a. Simuler l'exécution de M sur l'entrée w.
   b. Si M entre dans son état d'acceptation, alors M’ s'arrête et accepte.
   c. Si M entre dans son état de rejet, alors M’ entre dans une boucle infinie."

Analysons le comportement de M’ :

- Si M accepte w, la simulation à l'étape 3a se terminera, et M’ s'arrêtera (étape 3b).
- Si M rejette w, la simulation se terminera, mais M’ entrera dans une boucle infinie (étape 3c).
- Si M boucle sur w, la simulation à l'étape 3a ne se terminera jamais, et donc M’ bouclera aussi.

Par conséquent, M’ s'arrête si et seulement si M accepte w. Nous avons donc bien :

⟨M,w⟩∈ATM⇔⟨M’,w⟩∈HALTTM

La fonction f qui transforme ⟨M,w⟩ en ⟨M’,w⟩ est calculable (elle consiste à prendre le code de M et à y ajouter quelques états et transitions pour implémenter la boucle). Nous avons donc établi que ATM≤mHALTTM. Puisque ATM est indécidable, HALTTM doit également être indécidable.71
Le Problème du Langage Vide (ETM)

Ce problème demande si le langage accepté par une machine de Turing est vide.

ETM={⟨M⟩|M est une MT et L(M)=∅}
Théorème : ETM est indécidable.

Preuve : Nous allons réduire ATM au complément de ETM, noté ETM={⟨M⟩|L(M)=∅}. Prouver que ETM est indécidable implique que ETM l'est aussi. La réduction f transforme une instance ⟨M,w⟩ de ATM en une instance ⟨M'⟩ de ETM.

Construction de la réduction f(⟨M,w⟩)=⟨M'⟩ :

1. f prend en entrée la description ⟨M,w⟩.
2. f produit en sortie la description d'une nouvelle machine M'.
3. La machine M' est conçue comme suit :
   M' = "Sur une entrée x :
   a. Ignorer l'entrée x.
   b. Simuler l'exécution de M sur l'entrée w.
   c. Si la simulation de M accepte w, alors M' accepte x."

Analysons le langage de M' :

- Si M accepte w, alors pour n'importe quelle entrée x, la simulation à l'étape 3b réussira, et M' acceptera x. Dans ce cas, L(M')=Σ∗, qui est non vide.
- Si M n'accepte pas w (rejette ou boucle), la simulation à l'étape 3b ne mènera jamais à l'acceptation. Par conséquent, M' n'acceptera jamais aucune entrée x. Dans ce cas, L(M')=∅.

Nous avons donc la correspondance parfaite :
$$ \langle M, w \rangle \in A_{TM} \iff L(M') \neq \emptyset \iff \langle M' \rangle \in \overline{E_{TM}} $$
La fonction f est calculable. Nous avons donc montré que ATM≤mETM. Puisque ATM est indécidable, ETM est indécidable, et par conséquent, ETM est également indécidable.71
Le Problème de l'Équivalence (EQTM)

Ce problème demande si deux machines de Turing acceptent le même langage.

EQTM={⟨M1,M2⟩|M1,M2 sont des MT et L(M1)=L(M2)}
Théorème : EQTM est indécidable.

Preuve : Nous allons montrer que ETM≤mEQTM. La réduction est particulièrement élégante.

Construction de la réduction f(⟨M⟩)=⟨M,MØ⟩ :

1. f prend en entrée la description ⟨M⟩ d'une machine de Turing.
2. f construit une machine MØ qui rejette systématiquement toutes ses entrées. Le langage de cette machine est L(MØ)=∅. Cette machine est fixe et simple à décrire.
3. f produit en sortie la paire ⟨M,MØ⟩.

Analysons la condition d'équivalence :

- La condition L(M)=L(MØ) est vraie si et seulement si L(M)=∅.

Nous avons donc la correspondance :
$$ \langle M \rangle \in E_{TM} \iff L(M) = \emptyset \iff L(M) = L(M_{\emptyset}) \iff \langle M, M_{\emptyset} \rangle \in EQ_{TM} $$
La fonction f est clairement calculable. Nous avons donc montré que ETM≤mEQTM. Puisque nous venons de prouver que ETM est indécidable, il s'ensuit que EQTM est également indécidable.70

5.2.3 Théorème de Rice : Un Méta-théorème d'Indécidabilité

Les preuves précédentes montrent que de nombreuses questions spécifiques sur le comportement des machines de Turing sont indécidables. Le théorème de Rice généralise ces résultats de manière spectaculaire. Il affirme que *toute* question non triviale sur le *langage* accepté par une machine de Turing est indécidable. C'est un "méta-théorème" car il ne prouve pas l'indécidabilité d'un seul problème, mais d'une classe infinie de problèmes en une seule fois.[75]

Énoncé Formel du Théorème

Pour énoncer le théorème rigoureusement, nous devons définir ce que nous entendons par "propriété d'un langage".

- Une **propriété des langages récursivement énumérables** est un ensemble de langages récursivement énumérables. Par exemple, la propriété "être un langage fini" est l'ensemble de tous les langages RE qui sont finis.
- Une propriété P est dite **non triviale** si elle n'est ni toujours vraie, ni toujours fausse. C'est-à-dire qu'il existe au moins un langage RE qui possède la propriété P, et au moins un langage RE qui ne la possède pas.[75]

Le théorème de Rice concerne la décidabilité du problème suivant : étant donné une machine de Turing M, le langage qu'elle accepte, L(M), possède-t-il une certaine propriété P?

Théorème de Rice : Soit P une propriété non triviale des langages récursivement énumérables. Le langage LP ={⟨M⟩|L(M) posse`de la proprie´te´ P} est indécidable.

Il est crucial de noter que la propriété doit porter sur le langage L(M) (la sémantique, le comportement) et non sur la machine M elle-même (la syntaxe, le code). Par exemple, "la machine M a plus de 10 états" est une propriété syntaxique et est parfaitement décidable. En revanche, "le langage L(M) est régulier" est une propriété sémantique, et comme nous le verrons, elle est indécidable.

Preuve Complète et Commentée du Théorème

La preuve du théorème de Rice est une généralisation élégante de la preuve d'indécidabilité de ETM. Elle utilise une réduction de ATM au problème de la décision de LP pour une propriété P non triviale quelconque.[75]

Preuve :

Soit P une propriété non triviale des langages RE. Nous voulons montrer que LP est indécidable.

1. **Mise en place :**
   - Sans perte de généralité, supposons que le langage vide, $\emptyset$, ne possède pas la propriété P (c'est-à-dire $\emptyset \in /P$). Si ce n'est pas le cas, nous pouvons simplement mener la preuve avec la propriété complémentaire P, car si P est non triviale, P l'est aussi, et l'indécidabilité de LP implique celle de LP.
   - Puisque P est non triviale, il doit exister au moins un langage qui possède cette propriété. Soit Loui un tel langage, avec $Loui \in P$. Puisque Loui est RE, il existe une machine de Turing, appelons-la Moui, qui le reconnaît ($L(Moui)=Loui$).
2. Construction de la Réduction : Nous allons réduire ATM à LP. Nous construisons une fonction calculable f qui prend une instance $\langle M,w \rangle$ de ATM et produit une instance $\langle M' \rangle$ de LP.
   $f(\langle M,w \rangle)=\langle M' \rangle$, où M' est la machine de Turing suivante :
   M' = "Sur une entrée x :
   a. Simuler l'exécution de la machine M sur l'entrée w.
   b. Si la simulation de M sur w accepte, alors exécuter la machine Moui sur l'entrée x. Accepter x si et seulement si Moui accepte x.
   c. Si la simulation de M sur w rejette ou boucle, alors rejeter immédiatement x."
3. **Analyse de la Correction de la Réduction :** Nous devons vérifier que $\langle M,w \rangle \in ATM \Leftrightarrow \langle M' \rangle \in LP$.
   - Cas 1 : Supposons que $\langle M,w \rangle \in ATM$ (c'est-à-dire, M accepte w).
     Dans ce cas, lorsque M' est exécutée sur une entrée x, la simulation de l'étape (a) se termine avec succès. L'exécution passe à l'étape (b). M' se comporte alors exactement comme Moui sur l'entrée x. Par conséquent, le langage accepté par M' est le même que celui accepté par Moui.

     $L(M')=L(Moui)=Loui$

     Or, nous avons choisi Loui spécifiquement parce qu'il possède la propriété P. Donc, L(M') possède la propriété P, ce qui signifie que $\langle M' \rangle \in LP$.
   - Cas 2 : Supposons que $\langle M,w \rangle \in /ATM$ (c'est-à-dire, M n'accepte pas w).
     Dans ce cas, lorsque M' est exécutée sur une entrée x, la simulation de l'étape (a) ne se termine jamais par une acceptation (elle rejette ou boucle). L'exécution passe donc toujours à l'étape (c), où M' rejette x. M' ne peut donc accepter aucune chaîne.

     $L(M')=\emptyset$

     Par notre supposition initiale, le langage vide $\emptyset$ ne possède pas la propriété P. Donc, L(M') ne possède pas la propriété P, ce qui signifie que $\langle M' \rangle \in /LP$.
4. Conclusion :
   Nous avons établi une équivalence parfaite : M accepte w si et seulement si L(M') possède la propriété P. La fonction f qui construit $\langle M' \rangle$ à partir de $\langle M,w \rangle$ est calculable. Nous avons donc une réduction valide : $ATM \leq_m LP$. Puisque ATM est indécidable, nous concluons que LP doit être indécidable. Ceci achève la preuve du théorème de Rice.

Le théorème de Rice est un outil d'une puissance extraordinaire pour classer rapidement des problèmes comme étant indécidables. Pour prouver qu'un problème de décision sur les programmes est indécidable, il suffit de vérifier trois conditions :

1. Le problème concerne une propriété du **langage** accepté par le programme, et non les détails de son implémentation.
2. La propriété est **non triviale** (il existe au moins un programme qui la satisfait et un qui ne la satisfait pas).
3. Le modèle de programmation est **Turing-complet**.

Appliquons ce théorème pour démontrer instantanément l'indécidabilité de plusieurs problèmes d'intérêt pratique en génie logiciel et en informatique théorique [76] :

- **Problème : Le langage d'une MT est-il vide?**
  - Propriété du langage? Oui.
  - Non triviale? Oui. Il existe des MT qui acceptent le langage vide (par exemple, une machine qui rejette tout), et des MT qui acceptent des langages non vides (par exemple, une machine qui accepte tout).
  - **Conclusion (par le théorème de Rice) :** Indécidable. (Ceci confirme notre preuve par réduction pour ETM).
- **Problème : Le langage d'une MT est-il régulier?**
  - Propriété du langage? Oui. La régularité est une propriété d'un ensemble de chaînes.
  - Non triviale? Oui. Le langage $\{0n1n|n\geq 0\}$ n'est pas régulier mais est décidable par une MT. Le langage $\{0,1\}*$ est régulier et décidable par une MT.
  - **Conclusion :** Indécidable. Il est impossible d'écrire un programme qui détermine si un autre programme donné reconnaît un langage régulier.
- **Problème : Le langage d'une MT contient-il la chaîne "abc"?**
  - Propriété du langage? Oui.
  - Non triviale? Oui. La MT qui accepte uniquement "abc" a cette propriété. La MT qui n'accepte rien ne l'a pas.
  - **Conclusion :** Indécidable.
- **Problème : Le langage d'une MT est-il fini?**
  - Propriété du langage? Oui.
  - Non triviale? Oui. Le langage {"abc"} est fini. Le langage $\{0,1\}*$ est infini. Les deux sont reconnaissables par des MT.
  - **Conclusion :** Indécidable.

Le théorème de Rice formalise une intuition profonde sur la difficulté de l'analyse de programmes. Il établit une barrière fondamentale : alors que nous pouvons facilement analyser les propriétés **statiques** ou **syntaxiques** d'un programme (son nombre de lignes, les variables qu'il utilise), il est en général impossible de prédire de manière algorithmique ses propriétés **dynamiques** ou **sémantiques** (ce qu'il fera à l'exécution). Toute tentative de créer un outil d'analyse de code universel et complet pour de telles propriétés est vouée à l'échec. Cette limitation n'est pas une faiblesse de nos outils actuels, mais une caractéristique intrinsèque de la nature même du calcul.

Ouvrages cités

1. Séance 1 | PDF | Théorie de la calculabilité | Entier naturel - Scribd, dernier accès : septembre 21, 2025, https://fr.scribd.com/document/465701029/Se-ance-1

2. Alan Turing : pionnier visionnaire de l'informatique et architecte de l'IA - Babbar.tech Blog, dernier accès : septembre 21, 2025, https://blog.babbar.tech/alan-turing-pionnier-visionnaire-de-linformatique-et-architecte-de-lia/
3. Alan Turing - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Alan_Turing
4. La Machine de Turing - Odyssud, dernier accès : septembre 21, 2025, https://www.odyssud.com/sites/default/files/dossier-pedagogique-la-machine-de-turing_0.pdf
5. The Lambda Calculus, dernier accès : septembre 21, 2025, https://cs.lmu.edu/~ray/notes/lambdacalculus/
6. Cours 5: Thèse de Church. Indécidabilité., dernier accès : septembre 21, 2025, https://www.enseignement.polytechnique.fr/informatique/INF412/AMPHIS/cours5-handout2x2.pdf
7. Théorie de la calculabilité: Concepts, Exercices | StudySmarter, dernier accès : septembre 21, 2025, https://www.studysmarter.fr/resumes/mathematiques/logique-et-fondements/theorie-de-la-calculabilite/
8. Introduction à la calculabilité: cours et exercices corrigés - Pierre Wolper - Google Books, dernier accès : septembre 21, 2025, https://books.google.com/books/about/Introduction_%C3%A0_la_calculabilit%C3%A9.html?id=klMcGQAACAAJ
9. Théorie de la calculabilité - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Th%C3%A9orie_de_la_calculabilit%C3%A9
10. Calculabilité Cours 1 : machines de Turing, dernier accès : septembre 21, 2025, https://pageperso.lis-lab.fr/~kevin.perrot/documents/2018/calculabilite/Cours01_18.pdf
11. Alan Turing, père de l'informatique, naissait il y a 100 ans… - Cigref, dernier accès : septembre 21, 2025, https://www.cigref.fr/archives/histoire-cigref/blog/alan-turing-pere-de-l-informatique-naissait-il-y-a-100-ans/
12. Alan Turing : du concept à la machine - Interstices.info, dernier accès : septembre 21, 2025, https://interstices.info/alan-turing-du-concept-a-la-machine/
13. FORMAL LANGUAGES, AUTOMATA AND COMPUTABILITY, dernier accès : septembre 21, 2025, https://www.cs.cmu.edu/~lblum/flac/Handouts_pdf/Lecture9-handouts.pdf
14. M1 informatique – Lyon 1 2017 – 2018 M1if09 – Calculabilité et complexité Support de cours 1 / 22 Calculabilité et com - CNRS, dernier accès : septembre 21, 2025, https://perso.liris.cnrs.fr/sylvain.brandel/wiki/lib/exe/fetch.php%3Fmedia%3Dens:m1if09:m1if09-support-de-cours.pdf
15. Machine de Turing - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Machine_de_Turing
16. Cours Logique et Calculabilité, dernier accès : septembre 21, 2025, https://pageperso.lis-lab.fr/kevin.perrot/documents/2016/calculabilite/Cours_16.pdf
17. Leçon 913 : Machines de Turing. Applications., dernier accès : septembre 21, 2025, https://perso.eleves.ens-rennes.fr/people/julie.parreaux/fichiers_agreg/Memoire_913.pdf
18. Langages non-réguliers et machines de Turing, dernier accès : septembre 21, 2025, https://perso.univ-rennes1.fr/dimitri.petritis/enseignement/lcm1/lcm1-chap4.pdf
19. 11. Machines de Turing (divers) - Arnaud Casteigts, dernier accès : septembre 21, 2025, https://arnaudcasteigts.net/files/langages-11-cours-2023.pdf
20. 11. Machines de Turing non déterministes (et autres) - Arnaud Casteigts, dernier accès : septembre 21, 2025, https://arnaudcasteigts.net/files/langages-B-cours.pdf
21. Turing Machine for a^n b^n c^n - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=l_yoELubOv8&pp=0gcJCfwAo7VqN5tD
22. Turing Machine Example: a^n b^n c^n - YouTube, dernier accès : septembre 21, 2025,

https://www.youtube.com/watch?v=wBYG_F9I78E
23. Simulation - Lawrence University, dernier accès : septembre 21, 2025, http://www2.lawrence.edu/fast/GREGGJ/CMSC515/chapt03/Simulation.html
24. ITEC 420 - Section 3.2 - Variants of Turing Machines, dernier accès : septembre 21, 2025, https://sites.radford.edu/~nokie/classes/420/Chap3-2.html
25. Chaque machine de Turing multi-bande possède-t-elle une machine de Turing mono-bande équivalente ? - Académie EITCA - EITCA Academy, dernier accès : septembre 21, 2025, https://fr.eitca.org/la-cyber-s%C3%A9curit%C3%A9/eitc-est-cctf-fondamentaux-de-la-th%C3%A9orie-de-la-complexit%C3%A9-computationnelle/machines-de-turing/machines-de-turing-multitape/Est-ce-que-chaque-machine-de-tournage-multi-bandes-a-une-machine-de-tournage-%C3%A0-bande-unique-%C3%A9quivalente-a/
26. How to map the tapes of a "k-tape" Turing Machine into the single tape of a "1-tape" Turing Machine - Computer Science Stack Exchange, dernier accès : septembre 21, 2025, https://cs.stackexchange.com/questions/14619/how-to-map-the-tapes-of-a-k-tape-turing-machine-into-the-single-tape-of-a-1-t
27. Simulating multi-tape turing machines by single-tape TM - Computer Science Stack Exchange, dernier accès : septembre 21, 2025, https://cs.stackexchange.com/questions/112453/simulating-multi-tape-turing-machines-by-single-tape-tm
28. Machine de Turing non déterministe - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Machine_de_Turing_non_d%C3%A9terministe
29. Comprendre les machines de Turing non déterministes - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=5VRjCk7R-Xg
30. 1 Non-deterministic Turing Machine A nondeterministic Turing ..., dernier accès : septembre 21, 2025, https://www.cs.rpi.edu/~goldberg/14-CC/02-ndt.pdf
31. Nondeterministic Turing machine - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Nondeterministic_Turing_machine
32. Decidability The Acceptance Problem for TMs - Kent State University, dernier accès : septembre 21, 2025, https://www.cs.kent.edu/~dragan/ThComp/lect13-2.pdf
33. Définition des langages récursifs et récursivement énumérables - l'IRIF, dernier accès : septembre 21, 2025, https://www.irif.fr/~carton/Enseignement/Complexite/MasterInfo/Cours/recursif.html
34. Récursivement énumérable - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/R%C3%A9cursivement_%C3%A9num%C3%A9rable
35. TD – Fonctions μ-récursives, λ-calcul, pavages, dernier accès : septembre 21, 2025, https://pageperso.lis-lab.fr/~kevin.perrot/documents/2020/calculabiliteavancee/TD_tct_20.pdf
36. Séance 5 : Fonctions récursives et machine de Turing, dernier accès : septembre 21, 2025, https://www.i3s.unice.fr/~nlt/cours/licence/it/s5_itdut_poly.pdf
37. μ-Recursive Functions, dernier accès : septembre 21, 2025, https://homepage.cs.uri.edu/faculty/hamel/courses/2015/spring2015/csc544/lecture-notes/11-recursive-functions.pdf
38. Éléments de Théorie de la Calculabilité - Mathematics, dernier accès : septembre 21, 2025, https://www.math.ru.nl/~terwijn/teaching/syllabus_fr.pdf
39. TD 2 – Fonctions récursives, dernier accès : septembre 21, 2025, https://flemaitre.perso.math.cnrs.fr/enseignements/1819/logiqueetcomplexite/TD-2-corrige.pdf
40. General recursive function - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/General_recursive_function
41. Les machines de Turing sont équivalentes que les fonctions μ-récursives, dernier accès : septembre 21, 2025, https://perso.eleves.ens-rennes.fr/people/julie.parreaux/fichiers_agreg/info_dev/MachinesTuring-Recursive.pdf

42. Cours de Lambda-calcul - IA, dernier accès : septembre 21, 2025, https://webia.lip6.fr/~hardin/DEA/poly.pdf
43. Lambda calculus - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Lambda_calculus
44. Lambda-calcul - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Lambda-calcul
45. Lambda Calculus | Good Times Paradigms, dernier accès : septembre 21, 2025, https://tgdwyer.github.io/lambdacalculus/
46. Lambda (λ) calculus evaluation rules (δ, β, α, η conversion/reduction) - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=VS_GK-9xUO4
47. The Lambda Calculus (Stanford Encyclopedia of Philosophy), dernier accès : septembre 21, 2025, https://plato.stanford.edu/entries/lambda-calculus/
48. Lambda Calculus, dernier accès : septembre 21, 2025, http://web.stanford.edu/class/cs242/materials/lectures/lecture04.pdf
49. Recursion Lecture 8 Thursday, February 17, 2016 1 Lambda calculus encodings, dernier accès : septembre 21, 2025, https://groups.seas.harvard.edu/courses/cs152/2016sp/lectures/lec08-encodings.pdf
50. Encoding Data in Lambda Calculus: An Introduction, dernier accès : septembre 21, 2025, https://cse.sc.edu/~pfu/document/talks/lam.pdf
51. Lambda Calculus and Church Encoded Integers - Daniel's Blog, dernier accès : septembre 21, 2025, https://danilafe.com/blog/lambda_calculus_integers/
52. Lecture 28: Introduction to the λ-Calculus - Cornell: Computer Science, dernier accès : septembre 21, 2025, https://www.cs.cornell.edu/courses/cs3110/2014sp/recitations/25/lambda-calculus.html
53. The Physical Church–Turing Thesis: Modest or Bold? | The British Journal for the Philosophy of Science: Vol 62, No 4, dernier accès : septembre 21, 2025, https://www.journals.uchicago.edu/doi/10.1093/bjps/axr016
54. The Church-Turing Thesis (Stanford Encyclopedia of Philosophy), dernier accès : septembre 21, 2025, https://plato.stanford.edu/entries/church-turing/
55. Initiation à la calculabilité - [Verimag], dernier accès : septembre 21, 2025, http://www-verimag.imag.fr/~monniaux/biblio/Monniaux_Quadrature2012.pdf
56. Why do we believe the Church-Turing Thesis? - Mathematics Stack Exchange, dernier accès : septembre 21, 2025, https://math.stackexchange.com/questions/402934/why-do-we-believe-the-church-turing-thesis
57. What is the Church-Turing Thesis?, dernier accès : septembre 21, 2025, https://faculty.runi.ac.il/udiboker/files/CTT.pdf
58. Pourquoi n'y a-t-il pas de preuve de la thèse de Church-Turing ? : r/math - Reddit, dernier accès : septembre 21, 2025, https://www.reddit.com/r/math/comments/1hbumro/why_is_there_no_proof_of_the_churchturing_thesis/?tl=fr
59. Thèse de Church-Turing: Concepts clés, Applications | StudySmarter, dernier accès : septembre 21, 2025, https://www.studysmarter.fr/resumes/informatique/theorie-de-la-computation/these-de-church-turing/
60. Les deux formes de la thèse de Church-Turing et l'épistémologie du calcul, dernier accès : septembre 21, 2025, https://journals.openedition.org/philosophiascientiae/769
61. Algorithmes quantiques : quand la physique quantique défie la ..., dernier accès : septembre 21, 2025, https://www.ins2i.cnrs.fr/fr/cnrsinfo/algorithmes-quantiques-quand-la-physique-quantique-defie-la-these-de-church-turing

62. Acceptance for Turing Machines is Undecidable, but Recognizable - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=ASgv7SmUCzo
63. The Acceptance Problem, dernier accès : septembre 21, 2025, https://people.computing.clemson.edu/~goddard/handouts/cpsc3500/MATERIAL/14b9-11.pdf
64. Turing Machines - Washington, dernier accès : septembre 21, 2025, https://courses.cs.washington.edu/courses/cse322/10au/slides/11tm.pdf
65. Calculabilité : le problème de l'arrêt - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=13O1qhX4Bqo
66. Argument de la diagonale de Cantor - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Argument_de_la_diagonale_de_Cantor
67. Argument de la diagonale de Cantor : définition et explications - Techno-Science, dernier accès : septembre 21, 2025, https://www.techno-science.net/definition/6441.html
68. Problème de l'arrêt : démonstration ?, dernier accès : septembre 21, 2025, https://quentin.pradet.me/blog/probleme-de-larret-demonstration.html
69. Mapping Reductions, dernier accès : septembre 21, 2025, https://web.stanford.edu/class/archive/cs/cs103/cs103.1134/lectures/22/Small22.pdf
70. Lecture 15: Undecidability by Reduction 1 Some Closure, dernier accès : septembre 21, 2025, https://faculty.cc.gatech.edu/~ladha/toc/L15.pdf
71. Réductions - l'IRIF, dernier accès : septembre 21, 2025, https://www.irif.fr/~carton/Enseignement/Complexite/MasterInfo/Cours/reduction.html
72. 10 Reducibility, dernier accès : septembre 21, 2025, https://www.cs.wm.edu/~wm/CS423/ln10.pdf
73. Lecture 41/65: Halting Problem: A Proof by Reduction - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=dP7fTn0pLvw
74. 1 Reducability, dernier accès : septembre 21, 2025, https://www.utsc.utoronto.ca/~bretscher/c63/worksheets/reducibility.pdf
75. Théorème de Rice - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_de_Rice
76. Théorème de Rice : définition et explications - Techno-Science, dernier accès : septembre 21, 2025, https://www.techno-science.net/definition/6238.html
77. Théorème de Rice - Minerve de l'ENS Rennes, dernier accès : septembre 21, 2025, https://minerve.ens-rennes.fr/images/Dvt_rice.pdf
78. Calculabilité, dernier accès : septembre 21, 2025, https://www.enseignement.polytechnique.fr/informatique/INF412/uploads/Main/chap8-good.pdf
79. Rice's Theorem - Department of Computer Science, University of Toronto, dernier accès : septembre 21, 2025, https://www.cs.toronto.edu/~vassos/teaching/c63/handouts/RicesThm.pdf
80. Théorème de Rice - Digiscope, dernier accès : septembre 21, 2025, http://www.digiscope.fr/~hellouin/Agreg/Th%C3%A9or%C3%A8me-de-Rice.pdf
81. TD MC3 Théorème de Rice - LaBRI, dernier accès : septembre 21, 2025, https://www.labri.fr/perso/betrema/MC/TD3s.html
82. TD : Théorème de Rice - LaBRI, dernier accès : septembre 21, 2025, https://www.labri.fr/perso/betrema/MC/TD3.html

Chapitre 6 : Théorie de la Complexité

Introduction : Le Grand Schisme – De la Calculabilité à la Complexité

Les chapitres précédents de cet ouvrage ont jeté les fondations de l'informatique théorique en s'attaquant à une question d'une portée philosophique et mathématique immense : quels sont les problèmes qu'une machine peut, en principe, résoudre? Cette quête, menée par des pionniers tels qu'Alan Turing, Alonzo Church et Kurt Gödel, a donné naissance à la théorie de la calculabilité.[1] Elle a tracé une ligne de démarcation nette et absolue dans l'univers des problèmes formels : d'un côté, les problèmes

calculables (ou décidables), pour lesquels il existe un algorithme garanti de s'arrêter et de fournir une réponse correcte ; de l'autre, les problèmes *incalculables* (ou indécidables), comme le célèbre problème de l'arrêt, pour lesquels aucun algorithme de ce type ne peut exister.[3] La théorie de la calculabilité opère dans un monde d'abstraction pure, où les modèles de calcul, telle la machine de Turing, disposent de ressources illimitées : un ruban infini et un temps infini.

Cependant, l'avènement des premiers ordinateurs physiques dans les années 1940 et 1950 a confronté cette vision idéalisée à une réalité beaucoup plus contraignante. Ces machines, bien que réalisations concrètes des modèles théoriques, étaient dotées de mémoires finies et de vitesses de calcul limitées.[5] Un problème pouvait être théoriquement calculable, mais son exécution sur une machine réelle pouvait exiger des siècles, voire des millénaires, le rendant insoluble en pratique. Cette prise de conscience a provoqué un schisme conceptuel, déplaçant le centre de gravité de la recherche. La question n'était plus seulement « Peut-on le résoudre? », mais bien « Peut-on le résoudre

efficacement? ».[5]

C'est de cette interrogation pragmatique qu'est née la théorie de la complexité computationnelle dans les années 1960. Elle se propose de quantifier la difficulté inhérente des problèmes calculables en mesurant les ressources nécessaires à leur résolution. Les deux ressources fondamentales sont le **temps**, qui correspond au nombre d'opérations élémentaires effectuées par un algorithme, et l'**espace**, qui représente la quantité de mémoire requise.[8] La théorie de la complexité est donc la science formelle des contraintes du monde réel, le pont entre ce qui est abstraitement possible et ce qui est concrètement faisable.

Ce chapitre se consacre à l'exploration de ce domaine fascinant. Nous commencerons par définir formellement les mesures de complexité en temps et en espace, en nous appuyant sur le modèle robuste de la machine de Turing. Nous introduirons ensuite l'analyse asymptotique, le langage mathématique qui nous permet de raisonner sur l'efficacité des algorithmes de manière indépendante de la technologie. Armés de ces outils, nous explorerons le "zoo" des classes de complexité, en nous concentrant sur les classes fondamentales **P** et **NP**. La classe **P** regroupe les problèmes considérés

comme "faciles" ou "traitables", tandis que la classe **NP** contient des problèmes dont la solution, une fois trouvée, est facile à vérifier.

Cette exploration nous mènera inévitablement au cœur de l'informatique théorique moderne : le problème **P versus NP**. Cette question, dotée d'un prix d'un million de dollars par l'Institut Clay, demande si ces deux classes sont en réalité une seule et même chose. Nous analyserons les implications profondes qu'une réponse, qu'elle soit positive ou négative, aurait sur la science, la technologie, l'économie et même notre compréhension de la créativité. Enfin, nous étudierons le concept de **NP-complétude**, qui identifie les problèmes les plus difficiles de la classe **NP**, et nous conclurons par un aperçu des classes de complexité plus avancées, au-delà de **NP**, pour donner une idée de l'étendue et de la richesse de ce champ de recherche toujours en pleine effervescence.

6.1 Mesures de la Complexité (Temps et Espace)

Pour quantifier rigoureusement la consommation de ressources d'un algorithme, il est indispensable de s'appuyer sur un modèle de calcul formel et universel. Conformément à la thèse de Church-Turing, qui postule que tout ce qui est intuitivement calculable peut l'être par une machine de Turing, ce modèle constitue le fondement de la théorie de la complexité.[1]

Le Modèle de Calcul de Référence : La Machine de Turing

Nous rappelons brièvement les définitions des deux variantes principales de la machine de Turing, qui sont essentielles pour distinguer les classes de complexité fondamentales.

La Machine de Turing Déterministe (MTD)

Une machine de Turing déterministe est un modèle de calcul abstrait qui manipule des symboles sur un ruban infini. Sa caractéristique principale est que, pour toute configuration donnée, l'action suivante est unique et entièrement déterminée.

Définition Formelle : Une machine de Turing déterministe (MTD) est formellement définie par un 7-uplet $M=(Q,\Sigma,\Gamma,\delta,q_0,B,F)$, où :

- Q est un ensemble fini d'**états**.
- Σ est un ensemble fini de symboles appelé l'**alphabet d'entrée**.
- Γ est un ensemble fini de symboles appelé l'**alphabet de ruban**, tel que $\Sigma \subset \Gamma$.
- $\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L,R\}$ est la **fonction de transition**. L et R représentent respectivement un déplacement de la tête

de lecture/écriture vers la gauche et vers la droite.

- $q_0 \in Q$ est l'**état initial**.
- $B \in \Gamma \setminus \Sigma$ est le **symbole blanc**, qui remplit initialement le ruban à l'exception de l'entrée.
- $F \subseteq Q$ est l'ensemble des **états finaux** ou **d'acceptation**.

Une **configuration** de la machine est une description instantanée de son état, du contenu de son ruban et de la position de sa tête. Elle peut être représentée par un triplet (u,q,v), où $q \in Q$ est l'état courant, et uv est le contenu du ruban, la tête se trouvant sur le premier symbole de v.[14] Un

calcul est une séquence de configurations $C_0, C_1, C_2, \ldots$, où C_0 est la configuration initiale et chaque C_{i+1} est obtenue à partir de C_i en appliquant la fonction de transition δ.[14] Le calcul s'arrête si la machine atteint un état pour lequel aucune transition n'est définie ou, par convention pour les problèmes de décision, si elle entre dans un état final.

La Machine de Turing Non Déterministe (MTND)

La machine de Turing non déterministe se distingue par sa capacité à explorer plusieurs chemins de calcul simultanément. À chaque étape, la machine peut avoir plusieurs choix d'actions possibles.

Définition Formelle : Une machine de Turing non déterministe (MTND) est définie par un 7-uplet similaire à celui de la MTD, $M=(Q,\Sigma,\Gamma,\delta,q_0,B,F)$, mais avec une fonction de transition différente :

- $\delta : (Q \setminus F) \times \Gamma \to P(Q \times \Gamma \times \{L,R\})$ est la **relation de transition**, où $P(\cdot)$ désigne l'ensemble des parties. Pour un état et un symbole donnés, δ retourne un *ensemble* de transitions possibles.[16]

Le calcul d'une MTND sur une entrée w n'est plus une séquence linéaire de configurations, mais un **arbre de calcul**. Chaque nœud de l'arbre est une configuration, et les enfants d'un nœud représentent les configurations atteignables en une seule étape. L'acceptation est définie de manière existentielle : une MTND accepte une entrée w s'il existe *au moins un* chemin dans l'arbre de calcul, partant de la configuration initiale, qui mène à un état d'acceptation.[19] Si aucun chemin ne mène à un état d'acceptation (soit parce que tous les chemins s'arrêtent dans un état de rejet, soit parce que certains bouclent indéfiniment sans accepter), l'entrée est rejetée.

Définitions Formelles des Mesures de Complexité

Avec ces modèles en place, nous pouvons maintenant définir formellement les ressources de temps et d'espace.

Complexité en Temps (Time Complexity)

La complexité en temps mesure le nombre d'étapes de calcul nécessaires pour qu'une machine de Turing résolve un problème.

- **Pour une MTD :** Soit M une MTD qui s'arrête sur toutes les entrées. Pour une entrée w, le **temps de calcul** de M sur w, noté timeM(w), est le nombre de transitions (ou d'étapes) que M effectue avant de s'arrêter.[11] C'est la longueur de la séquence de calcul.
- **Pour une MTND :** Soit M une MTND qui décide un langage L (c'est-à-dire que pour toute entrée, tous les chemins de calcul s'arrêtent). Le **temps de calcul** de M sur w, noté timeM(w), est défini comme suit :
  - Si w∈L, timeM(w) est la longueur du **plus court** chemin de calcul acceptant.[21]
  - Si w∈/L, timeM(w) est la longueur du **plus long** chemin de calcul (tous les chemins menant à un rejet).

Cette définition asymétrique pour la MTND est un choix conceptuel d'une importance capitale. En définissant le coût d'une réponse positive par le chemin le plus court, le modèle capture l'idée d'une "divination parfaite" ou d'une "preuve". La machine non déterministe n'a pas besoin d'explorer tout l'arbre de recherche ; sa complexité est définie par le coût de trouver et de vérifier une seule solution correcte. Une MTD qui simulerait cette MTND devrait potentiellement explorer l'arbre entier. Cette subtile définition est le germe mathématique d'où émergera naturellement la définition de la classe NP en termes de vérificateurs.

Complexité en Espace (Space Complexity)

La complexité en espace mesure la quantité de mémoire (le nombre de cases du ruban) utilisée pendant un calcul.

- **Pour une MTD ou une MTND :** Soit M une machine de Turing qui s'arrête sur toutes les entrées. Pour une entrée w, l'**espace de calcul** de M sur w, noté spaceM(w), est le nombre de cases distinctes du (ou des) ruban(s) de travail qui sont visitées par la tête de lecture/écriture au cours du calcul.[20]
  - Pour les machines à plusieurs rubans, on ne compte généralement que l'espace utilisé sur les rubans de travail, en excluant le ruban d'entrée (lecture seule) et de sortie (écriture seule), afin de permettre l'étude de classes de complexité en espace sous-linéaire.[24]
  - Pour une MTND, spaceM(w) est l'espace maximal utilisé sur n'importe quel chemin de l'arbre de calcul pour l'entrée w.[22]

Analyse de la Complexité : Pire Cas, Cas Moyen et Meilleur Cas

Une fois le coût (en temps ou en espace) défini pour une instance unique w, il est nécessaire d'agréger cette mesure pour toutes les entrées d'une taille donnée n=|w|. Cela donne lieu à trois types d'analyses distinctes, chacune ayant sa propre pertinence.

Complexité dans le Pire des Cas (Worst-Case)

La complexité dans le pire des cas est la mesure la plus fondamentale et la plus utilisée en théorie de la complexité. Elle représente une borne supérieure sur les ressources requises par un algorithme pour toute entrée d'une taille donnée.

Définition Formelle : La complexité en temps dans le pire des cas d'une machine M est la fonction $T_M: \mathbb{N} \to \mathbb{N}$ définie par :

$$T_M(n) = \max\{time_M(w) \mid w \in \Sigma^*, |w| = n\}$$

De même, la complexité en espace dans le pire des cas est la fonction $S_M: \mathbb{N} \to \mathbb{N}$ définie par :

$$S_M(n) = \max\{space_M(w) \mid w \in \Sigma^*, |w| = n\}$$

Cette analyse est cruciale car elle fournit une garantie de performance. Un algorithme avec une complexité dans le pire des cas $T_M(n)$ ne dépassera jamais cette limite, quelles que soient les spécificités de l'entrée. C'est essentiel pour les applications critiques où la fiabilité et la prévisibilité sont primordiales, comme dans les systèmes de contrôle aérien ou les dispositifs médicaux.25

Complexité en Cas Moyen (Average-Case)

L'analyse en cas moyen cherche à caractériser la performance "typique" d'un algorithme, en moyennant les coûts sur toutes les entrées possibles d'une taille donnée.

Définition Formelle : La complexité en temps en cas moyen d'une machine M est la fonction $A_M: \mathbb{N} \to \mathbb{R}^+$ définie par :

$$A_M(n) = \sum_{w \in \Sigma^n} P(w) \cdot time_M(w)$$

où Σ^n est l'ensemble des mots de longueur n et $P(w)$ est la probabilité de l'instance w. Une définition analogue existe pour la complexité en espace.27
Cette analyse est souvent plus représentative de la performance en pratique, car les pires cas peuvent être rares ou artificiels. Cependant, sa principale difficulté réside dans la définition d'une distribution de probabilité $P(w)$ qui modélise fidèlement les données du monde réel. Une hypothèse courante mais parfois irréaliste est la distribution uniforme, où toutes les entrées de taille n sont équiprobables.[29]

Complexité dans le Meilleur des Cas (Best-Case)

La complexité dans le meilleur des cas identifie le scénario le plus favorable pour un algorithme.

Définition Formelle : La complexité en temps dans le meilleur des cas d'une machine M est la fonction $B_M: \mathbb{N} \to \mathbb{N}$ définie

par :

$BM(n)=\min\{timeM(w)|w\in\Sigma*,|w|=n\}$

Bien qu'elle puisse être utile pour comprendre les limites inférieures de la performance d'un algorithme, cette mesure est rarement utilisée seule car elle est souvent peu informative sur le comportement général. Par exemple, un algorithme de recherche peut avoir une complexité dans le meilleur des cas constante (s'il trouve l'élément au premier essai), même si sa performance dans le pire des cas est linéaire ou pire.25
Par convention, sauf mention contraire, le terme "complexité" dans ce chapitre et dans la littérature se réfère à la **complexité dans le pire des cas**.

6.2 Analyse asymptotique (Notations O, Ω, Θ, o, ω)

L'analyse de la complexité vise à comprendre le comportement d'un algorithme lorsque la taille de l'entrée devient très grande. Le nombre exact d'opérations, par exemple $3n2+10n+5$, est moins important que son ordre de croissance, ici quadratique. L'analyse asymptotique, à l'aide des notations de Bachmann-Landau, est l'outil mathématique qui formalise cette abstraction en se concentrant sur le comportement à la limite, ignorant les constantes multiplicatives et les termes d'ordre inférieur qui dépendent de l'implémentation et de la machine.[28]

Définitions Formelles des Notations Asymptotiques

Soient $f,g:N\rightarrow R+$ deux fonctions représentant des complexités.

Notation **O** *(Grand O) : Borne Supérieure Asymptotique*

La notation "Grand O" est utilisée pour donner une borne supérieure au taux de croissance d'une fonction. Elle caractérise la complexité dans le pire des cas d'un algorithme.

Définition : On dit que $f(n)$ est dans $O(g(n))$, et on note $f(n)\in O(g(n))$ ou abusivement $f(n)=O(g(n))$, s'il existe des constantes positives $c\in R+$ et $n0\in N$ telles que pour tout $n\geq n0$:

$0\leq f(n)\leq c\cdot g(n)$

Intuitivement, cela signifie que pour n suffisamment grand, la fonction $f(n)$ est bornée supérieurement par un multiple constant de $g(n)$. La fonction f ne croît pas asymptotiquement plus vite que g.32

Figure 6.1 : Représentation graphique de f(n)=O(g(n)). La fonction f(n) est bornée supérieurement par c·g(n) pour tout n≥n0.

Notation Ω (Grand Oméga) : Borne Inférieure Asymptotique

La notation "Grand Oméga" fournit une borne inférieure au taux de croissance d'une fonction. Elle est souvent utilisée pour caractériser la complexité dans le meilleur des cas ou pour établir des bornes inférieures sur la complexité d'un problème.

Définition : On dit que f(n) est dans Ω(g(n)), noté f(n)∈Ω(g(n)), s'il existe des constantes positives c∈R+ et n0∈N telles que pour tout n≥n0 :

0≤c·g(n)≤f(n)

Intuitivement, f(n) croît au moins aussi vite que g(n).32

Figure 6.2 : Représentation graphique de f(n)=Ω(g(n)). La fonction f(n) est bornée inférieurement par c·g(n) pour tout n≥n0.

Notation Θ (Grand Thêta) : Borne Étanche Asymptotique

La notation "Grand Thêta" est la plus précise. Elle indique que le taux de croissance d'une fonction est encadré à la fois par une borne supérieure et une borne inférieure, basées sur la même fonction de référence. Elle décrit un ordre de croissance exact.

Définition : On dit que f(n) est dans Θ(g(n)), noté f(n)∈Θ(g(n)), si f(n)∈O(g(n)) et f(n)∈Ω(g(n)). De manière équivalente, s'il existe des constantes positives c1,c2∈R+ et n0∈N telles que pour tout n≥n0 :

0≤c1·g(n)≤f(n)≤c2·g(n)

Intuitivement, f(n) et g(n) ont le même taux de croissance asymptotique.32
!(https://i.imgur.com/5wQ4u3s.png)
Figure 6.3 : Représentation graphique de f(n)=Θ(g(n)). La fonction f(n) est "prise en sandwich" entre c1·g(n) et c2·g(n) pour tout n≥n0.

Notations o (Petit o) et ω (Petit oméga) : Bornes Strictes

Ces notations sont utilisées pour exprimer des relations de croissance strictes, analogues aux inégalités strictes < et >.

Définition (Petit o) : On dit que f(n) est dans o(g(n)), noté f(n)∈o(g(n)), si pour toute constante c>0, il existe une constante n0∈N telle que pour tout n≥n0 :

$$0 \le f(n) < c \cdot g(n)$$Une définition équivalente utilisant les limites est :$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Intuitivement, f(n) devient insignifiante par rapport à g(n) lorsque n grandit ; f croît strictement moins vite que g.33
Définition (Petit oméga) : On dit que f(n) est dans ω(g(n)), noté f(n)∈ω(g(n)), si g(n)∈o(f(n)). De manière équivalente :

n→∞limg(n)f(n)=∞

Intuitivement, f croît strictement plus vite que g.33

Propriétés des Notations Asymptotiques

Ces notations définissent des relations sur les fonctions qui possèdent des propriétés mathématiques fondamentales, analogues à celles des relations d'ordre et d'équivalence sur les nombres. La démonstration rigoureuse de ces propriétés est ce qui confère sa solidité à l'analyse de complexité.

Réflexivité

Toute fonction est une borne asymptotique pour elle-même.

- f(n)∈O(f(n))
- f(n)∈Ω(f(n))
- f(n)∈Θ(f(n))

Preuve pour O(f(n)) :
Nous devons trouver des constantes c>0 et n0≥1 telles que f(n)≤c·f(n) pour tout n≥n0. En choisissant c=1 et n0=1, l'inégalité f(n)≤1·f(n) est trivialement vraie pour toute fonction positive. Les preuves pour Ω et Θ sont similaires (en choisissant c=1 pour Ω, et c1=c2=1 pour Θ).38

Transitivité

Cette propriété est la plus cruciale pour l'analyse de complexité, car elle permet de chaîner les relations.

- Si f(n)∈O(g(n)) et g(n)∈O(h(n)), alors f(n)∈O(h(n)).
- La propriété est également valable pour Ω,Θ,o,ω.

Preuve pour O :

1. Par hypothèse, $f(n) \in O(g(n))$, donc il existe $c1>0$ et $n1$ tels que $f(n) \leq c1 \cdot g(n)$ pour tout $n \geq n1$.
2. Par hypothèse, $g(n) \in O(h(n))$, donc il existe $c2>0$ et $n2$ tels que $g(n) \leq c2 \cdot h(n)$ pour tout $n \geq n2$.
3. Posons $n0=\max(n1,n2)$. Pour tout $n \geq n0$, les deux inégalités sont valides.
4. Nous pouvons donc écrire : $f(n) \leq c1 \cdot g(n) \leq c1 \cdot (c2 \cdot h(n)) = (c1c2) \cdot h(n)$.
5. En posant $c=c1c2$, nous avons $f(n) \leq c \cdot h(n)$ pour tout $n \geq n0$.
6. Par définition, $f(n) \in O(h(n))$.[32]

La transitivité n'est pas un simple exercice mathématique ; elle est le fondement logique de la théorie de la NP-complétude. C'est cette propriété qui nous autorisera plus tard à construire des chaînes de réductions ($A \leq_p B$ et $B \leq_p C$ implique $A \leq_p C$), permettant ainsi de propager la "difficulté" d'un problème à un autre et de bâtir toute la structure hiérarchique des problèmes NP-complets.

Symétrie et Symétrie Transposée

- **Symétrie (pour Θ) :** $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$. Cette propriété fait de Θ une relation d'équivalence.
- **Symétrie Transposée :** $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$. Cette dualité est une conséquence directe des définitions.

Le tableau suivant résume ces propriétés :

Propriété	O	Ω	Θ	o	ω
Réflexivité	Oui	Oui	Oui	Non	Non
Symétrie	Non	Non	Oui	Non	Non
Transitivité	Oui	Oui	Oui	Oui	Oui
Relation Analogue	≤	≥	=	<	>

6.3 Classes de Complexité Fondamentales

Armés des outils de l'analyse asymptotique, nous pouvons maintenant commencer à classifier les problèmes en fonction de leurs besoins en ressources. Cette classification donne naissance à un "bestiaire" de classes de complexité, dont les

deux plus fondamentales sont **P** et **NP**.

6.3.1 La classe P

La classe **P** est la première et la plus intuitive des classes de complexité. Elle représente l'ensemble des problèmes de décision qui peuvent être résolus "efficacement".

Définition Formelle

La classe **P** (pour Polynomial time) est l'ensemble de tous les langages (ou problèmes de décision) qui peuvent être décidés par une machine de Turing **déterministe** en un temps polynomial par rapport à la taille de l'entrée.

$P=k\in N\cup DTIME(nk)$
Où DTIME(f(n)) est l'ensemble des langages décidables par une MTD en temps O(f(n)).[11] Un algorithme dont la complexité en temps est bornée par un polynôme en la taille de l'entrée est appelé un

algorithme polynomial.

La Thèse de Cobham-Edmonds : L'Équation "Polynomial = Traitable"

Pourquoi le temps polynomial est-il le critère de l'efficacité? La **thèse de Cobham-Edmonds**, formulée indépendamment par Alan Cobham et Jack Edmonds au milieu des années 1960, postule que la classe **P** correspond à l'ensemble des problèmes qui sont "pratiquement solubles" ou "traitables" (en anglais, *computationally tractable*).[46]

Cette thèse repose sur plusieurs arguments solides :

1. **Robustesse du modèle :** La classe **P** est remarquablement insensible au modèle de calcul déterministe choisi. Un problème qui est polynomial sur une machine de Turing à un ruban le sera aussi sur une machine à plusieurs rubans ou sur une machine à accès aléatoire (RAM), le modèle qui se rapproche le plus d'un ordinateur réel. Les surcoûts de simulation entre ces modèles sont polynomiaux.[48]
2. **Propriétés de clôture :** La classe des algorithmes polynomiaux est fermée par composition. Un algorithme polynomial qui appelle en sous-routine un autre algorithme polynomial reste globalement polynomial. Cela correspond à notre pratique de la programmation modulaire.[45]
3. **Distinction pratique :** Il existe une différence spectaculaire entre la croissance des fonctions polynomiales (ex: n2,n3) et celle des fonctions exponentielles (ex: 2n,n!). Pour des entrées de taille modeste (par exemple, n=50), un algorithme en n3 est réalisable, tandis qu'un algorithme en 2n exigerait plus de temps que l'âge de l'univers.[50]

Il est important de noter que cette thèse est une abstraction. Un algorithme en O(n100) est polynomial mais totalement impraticable, tandis qu'un algorithme en O(2n/1000) est exponentiel mais peut être très efficace pour des valeurs de n rencontrées en pratique. Néanmoins, la thèse de Cobham-Edmonds s'est avérée être une ligne de démarcation extraordinairement utile et prédictive pour séparer les problèmes traitables des problèmes intrinsèquement difficiles.[46]

Exemples Détaillés de Problèmes dans P

De nombreux problèmes fondamentaux en informatique appartiennent à la classe **P**.

- **Recherche du plus court chemin dans un graphe :** Étant donné un graphe orienté ou non, avec des poids positifs sur les arêtes, et deux sommets s (source) et t (destination), le problème est de trouver un chemin de s à t de poids total minimal. L'**algorithme de Dijkstra** résout ce problème. Avec une implémentation utilisant un tas binaire, sa complexité est de O((|E|+|V|)log|V|), où |V| est le nombre de sommets et |E| le nombre d'arêtes. C'est une complexité polynomiale en la taille de l'entrée (qui est de l'ordre de |V|+|E|), donc ce problème est dans **P**.[50]
- **Produit de deux matrices :** Le calcul du produit de deux matrices n×n par l'algorithme standard, basé sur la définition, requiert n3 multiplications et n2(n−1) additions, soit une complexité en Θ(n3). Des algorithmes plus sophistiqués, comme celui de Strassen, réduisent cette complexité à environ O(n2.807), et des algorithmes encore plus avancés existent. Tous sont polynomiaux, plaçant ce problème dans **P**.[52]
- **Test de primalité :** Étant donné un entier N, déterminer s'il est un nombre premier. Pendant longtemps, on ne connaissait pas d'algorithme polynomial déterministe pour ce problème. Il était dans **NP**, mais on ne savait pas s'il était dans **P**. Ce n'est qu'en 2002 que Manindra Agrawal, Neeraj Kayal et Nitin Saxena ont publié l'**algorithme AKS**, qui résout le problème en temps polynomial, prouvant ainsi que le test de primalité est dans **P**.
- **2-SAT :** Le problème de satisfiabilité pour des formules booléennes où chaque clause contient au plus deux littéraux. Contrairement à sa version généralisée (SAT) ou à 3-SAT, 2-SAT peut être résolu en temps linéaire en construisant un graphe d'implications et en cherchant des composantes fortement connexes. Si une variable et sa négation se trouvent dans la même composante, la formule est insatisfiable ; sinon, elle l'est.[53]

6.3.2 La classe NP et le problème P vs NP

La classe **NP** est sans doute la classe de complexité la plus célèbre et la plus étudiée. Elle capture un grand nombre de problèmes d'optimisation et de recherche qui apparaissent constamment en pratique, et dont la résolution efficace reste un défi majeur.

Définitions Équivalentes de NP

Il existe deux manières formelles et équivalentes de définir la classe **NP**, chacune offrant un éclairage différent sur sa

nature.

1. Via le Non-déterminisme (NTIME)

La première définition, qui donne son nom à la classe (**N**ondeterministic **P**olynomial time), est une extension directe de la définition de **P** au modèle non déterministe.

Définition 1 : La classe **NP** est l'ensemble de tous les langages qui peuvent être décidés par une **machine de Turing non déterministe** en un temps polynomial par rapport à la taille de l'entrée.[21]

$NP=k\in N\bigcup NTIME(nk)$
Cette définition modélise l'idée d'une recherche par force brute massivement parallèle. La machine "devine" une solution potentielle et la vérifie ensuite. Grâce au non-déterminisme, l'étape de "divination" est considérée comme une seule opération.

2. Via la Vérification (Certificat/Témoin)

La seconde définition est souvent plus intuitive et plus utile en pratique. Elle caractérise les problèmes de **NP** non pas par la difficulté de *trouver* une solution, mais par la facilité de *vérifier* une solution si elle est fournie.

Définition 2 : Un langage L est dans NP s'il existe un algorithme déterministe en temps polynomial V (appelé vérificateur) et un polynôme p tels que pour toute entrée x :

$x\in L\Leftrightarrow\exists c,|c|\leq p(|x|)$ tel que $V(x,c)=accepte$

La chaîne c est appelée un certificat ou un témoin. Elle sert de "preuve" que x appartient au langage L. Le vérificateur V prend l'instance du problème x et le certificat c et doit confirmer en temps polynomial si le certificat est valide.11
Par exemple, pour le problème du **Circuit Hamiltonien** (existe-t-il un cycle qui visite chaque sommet exactement une fois?), une instance x est un graphe G. Un certificat c serait une permutation des sommets. Le vérificateur V n'a qu'à vérifier si cette permutation forme bien un cycle dans G, ce qui est faisable en temps polynomial. Trouver le cycle peut être très difficile, mais le vérifier est facile.

Preuve de l'Équivalence des Définitions

La puissance de ces deux définitions réside dans leur équivalence.

Preuve (Définition 1 $\Longrightarrow$ Définition 2) :

Supposons qu'un langage L est dans NTIME(p(n)) pour un polynôme p. Il existe donc une MTND M qui décide L en temps p(n). Pour une entrée x∈L, il existe au moins un chemin de calcul acceptant dans l'arbre de calcul de M. La séquence des choix non déterministes le long de ce chemin constitue un certificat c. La longueur de ce chemin, et donc de c, est au plus p(|x|).
On peut construire un vérificateur déterministe V(x,c) qui simule M sur l'entrée x, en utilisant les choix spécifiés par c à chaque étape non déterministe. Cette simulation est déterministe et s'exécute en temps polynomial, car la longueur du chemin est polynomiale. Si la simulation aboutit à un état d'acceptation, V accepte ; sinon, il rejette. Ainsi, L possède un vérificateur polynomial.56
Preuve (Définition 2 ⟹ Définition 1) :
Supposons qu'un langage L possède un vérificateur déterministe V qui s'exécute en temps p(n), et que la taille du certificat est bornée par q(n). On peut construire une MTND M pour décider L comme suit :

1. **Phase de divination (non déterministe) :** Pour une entrée x de taille n, M écrit non déterministement une chaîne c de longueur q(n) sur un ruban de travail. Cela prend q(n) étapes.
2. Phase de vérification (déterministe) : M simule ensuite le vérificateur déterministe V sur l'entrée (x,c). Cette simulation prend un temps polynomial p(|x|+|c|).
   Si V accepte, M accepte. Si V rejette, cette branche de calcul de M rejette.
   Si x∈L, il existe un certificat c qui fera accepter V. La MTND M trouvera ce chemin de calcul (par définition de l'acceptation non déterministe). La complexité totale de M est la somme des complexités des deux phases, ce qui reste polynomial. Donc, L est dans NTIME.56

Le Problème P vs NP

L'inclusion P⊆NP est directe. Si un problème est dans **P**, il est résolu par une MTD polynomiale. Cette MTD peut être vue comme une MTND qui n'utilise jamais le non-déterminisme. De manière équivalente, si un problème est dans **P**, on peut construire un vérificateur qui ignore simplement le certificat et résout le problème directement en temps polynomial.

La question fondamentale, posée au début des années 1970, est de savoir si cette inclusion est stricte :

Est-ce que P=NP?

C'est le problème P versus NP. Il demande si tout problème pour lequel une solution peut être vérifiée rapidement peut également être résolu rapidement. Malgré des décennies d'efforts par les plus grands esprits de l'informatique et des mathématiques, ce problème reste non résolu. Il est l'un des sept Problèmes du Prix du Millénaire, chacun doté d'un prix d'un million de dollars par l'Institut de mathématiques Clay.7

Implications Profondes de la Résolution

La réponse à cette question aurait des conséquences cataclysmiques, bien au-delà de l'informatique théorique.

- **Si P = NP :**

- **Révolution en Optimisation et en Science :** Des milliers de problèmes NP-complets (les plus difficiles de NP, que nous verrons bientôt) deviendraient traitables. Cela inclut des problèmes au cœur de la logistique (problème du voyageur de commerce), de la bio-informatique (repliement des protéines, conception de médicaments), de l'intelligence artificielle (apprentissage automatique, planification), de l'économie (allocation optimale des ressources) et de l'ingénierie (conception de circuits, vérification de programmes).[62] La capacité de résoudre ces problèmes de manière optimale transformerait radicalement l'industrie et la recherche scientifique.
- **Effondrement de la Cryptographie Moderne :** La sécurité de la plupart des protocoles cryptographiques à clé publique (comme RSA) repose sur la difficulté supposée de problèmes comme la factorisation d'entiers ou le calcul du logarithme discret. Ces problèmes sont dans **NP**. Si P=NP, des algorithmes polynomiaux pour les résoudre existeraient probablement, rendant obsolètes la quasi-totalité de l'infrastructure de sécurité d'Internet, des transactions bancaires aux communications sécurisées.[62]
- **Automatisation de la Créativité :** D'un point de vue philosophique, P=NP signifierait que l'acte de "trouver" une solution (ou une preuve mathématique, une composition musicale, une stratégie économique) n'est pas fondamentalement plus difficile que de "vérifier" sa validité. La créativité, dans de nombreux domaines, pourrait être automatisée. Trouver une preuve mathématique deviendrait une tâche mécanique, changeant à jamais la nature de la recherche.[60]

- **Si P ≠ NP :**
  - **Confirmation des Limites Intrinsèques :** Cela confirmerait l'intuition de la plupart des chercheurs : il existe une classe de problèmes intrinsèquement "difficiles" que nous ne pourrons jamais résoudre de manière optimale et efficace.
  - **Justification des Approches Actuelles :** Cela validerait formellement la nécessité des domaines de recherche actuels qui visent à contourner cette difficulté : les algorithmes d'approximation (qui cherchent des solutions quasi-optimales), les algorithmes probabilistes, les heuristiques et les solveurs SAT/SMT qui, bien qu'exponentiels dans le pire des cas, sont remarquablement efficaces sur de nombreuses instances pratiques.[60]
  - **Fondement pour une Cryptographie Sûre :** La preuve que P≠NP renforcerait la confiance dans les fondements de la cryptographie moderne, en garantissant qu'il existe bien une asymétrie fondamentale entre la difficulté de casser un code et la facilité de l'utiliser avec la bonne clé.

Cette question est bien plus qu'une énigme technique ; elle interroge la nature même de la résolution de problèmes et les limites fondamentales de notre capacité à calculer et à créer. L'asymétrie apparente entre la découverte et la validation est-elle une caractéristique fondamentale de notre univers computationnel, ou simplement le reflet de notre ignorance algorithmique actuelle? C'est le cœur du problème P vs NP.

6.4 NP-Complétude

Face à la difficulté de résoudre la question P vs NP, les chercheurs ont développé un outil puissant pour classifier les problèmes au sein de NP : la théorie de la **NP-complétude**. Cette théorie permet d'identifier les problèmes qui sont, en un sens, les "plus difficiles" de la classe NP. Si un seul de ces problèmes pouvait être résolu en temps polynomial, alors tous les problèmes de NP le pourraient aussi, et on aurait P = NP.

6.4.1 Réductions Polynomiales

Le concept central pour comparer la difficulté relative des problèmes est la **réduction polynomiale**.

Définition Formelle

Intuitivement, un problème A se réduit à un problème B si l'on peut utiliser un algorithme pour B comme une sous-routine pour résoudre A. La réduction en temps polynomial, ou réduction de Karp, formalise cette idée en exigeant que la traduction d'une instance de A en une instance de B soit elle-même efficace.

Définition : Soient L1 et L2 deux langages (problèmes de décision). On dit que L1 est **réductible en temps polynomial** à L2, noté $L1 \leq_p L2$, s'il existe une fonction $f:\Sigma^* \to \Sigma^*$ qui satisfait les deux conditions suivantes :

1. f est calculable par une machine de Turing déterministe en temps polynomial.
2. Pour tout mot $w \in \Sigma^*$, on a l'équivalence : $w \in L1 \iff f(w) \in L2$.

La fonction f est appelée la **fonction de réduction**.[68]

Importance Cruciale de la Réduction

La relation $\leq_p$ est une relation de pré-ordre (réflexive et transitive) sur l'ensemble des problèmes de décision. Son importance découle du théorème suivant :

Théorème : Si $L1 \leq_p L2$ et $L2 \in P$, alors $L1 \in P$.

Preuve : Pour décider si un mot w est dans L1, on procède en deux étapes :

1. Calculer f(w) en utilisant l'algorithme polynomial pour la réduction. Soit p_f le polynôme qui borne ce temps.
2. Utiliser l'algorithme polynomial pour L2 pour décider si $f(w) \in L2$. Soit p_{L2} le polynôme qui borne ce temps. La taille de f(w) est elle-même polynomiale en la taille de w. Le temps total est donc une composition de polynômes, ce qui reste un polynôme. L'algorithme global est donc polynomial.71

Ce théorème a un corollaire puissant : si $L1 \leq_p L2$ et que l'on sait que $L1 \notin P$, alors on peut conclure que $L2 \notin P$. La réduction permet de propager la "difficulté".

NP-Difficulté et NP-Complétude

Ces définitions permettent de formaliser la notion de "problème le plus difficile" dans NP.

Définition (NP-Difficulté) : Un problème H est dit NP-difficile (NP-hard) si tout problème L de la classe NP se réduit polynomialement à H.

$\forall L \in NP, L \leq_p H$

Un problème NP-difficile est donc au moins aussi difficile que n'importe quel problème dans NP. Notons qu'un problème NP-difficile n'a pas besoin d'être lui-même dans NP. Par exemple, le problème de l'arrêt est NP-difficile (car il est indécidable, donc plus difficile que tout problème dans NP), mais il n'est pas dans NP.73
Définition (NP-Complétude) : Un problème C est dit **NP-complet** (NP-complete) s'il satisfait deux conditions :

1. $C \in NP$ (le problème est dans NP).
2. C est NP-difficile.

Les problèmes NP-complets sont donc les problèmes les plus difficiles *au sein* de la classe NP. Ils capturent l'essence de la difficulté de toute la classe. La découverte d'un algorithme polynomial pour un seul problème NP-complet entraînerait l'effondrement de la hiérarchie : P = NP.[68]

6.4.2 Théorème de Cook-Levin (Problème SAT)

La théorie de la NP-complétude serait une coquille vide s'il n'existait aucun problème NP-complet. Le **théorème de Cook-Levin** est le résultat fondateur qui a établi l'existence d'un tel problème, fournissant le point d'ancrage pour toutes les preuves de NP-complétude qui ont suivi.

Problème SAT (Satisfiabilité Booléenne) :

- **Instance :** Une formule booléenne ϕ en forme normale conjonctive (FNC), c'est-à-dire une conjonction (ET) de clauses, où chaque clause est une disjonction (OU) de littéraux (une variable ou sa négation).
- **Question :** Existe-t-il une assignation de valeurs de vérité (vrai/faux) aux variables de ϕ qui rend la formule entière vraie?

Théorème (Cook, 1971 ; Levin, 1973) : Le problème **SAT** est NP-complet.

Esquisse de Preuve Détaillée

La preuve se déroule en deux étapes : montrer que SAT est dans NP, puis montrer qu'il est NP-difficile.

1. SAT est dans NP

Cette partie est relativement simple. Pour une instance donnée (une formule ϕ), un certificat est une assignation de valeurs de vérité à toutes les variables. Le vérificateur est un algorithme qui :

1. Prend en entrée la formule ϕ et une assignation.
2. Substitue les valeurs de vérité dans la formule.
3. Évalue chaque clause, puis la conjonction de toutes les clauses.
   Ce processus peut être accompli en temps linéaire par rapport à la taille de la formule. Donc, SAT est dans NP.77

2. SAT est NP-difficile

C'est le cœur de la preuve. Nous devons montrer que tout langage L∈NP se réduit polynomialement à SAT. C'est-à-dire, pour un langage L quelconque dans NP et une entrée w, nous devons construire en temps polynomial une formule booléenne ϕw qui est satisfiable si et seulement si w∈L.

La preuve repose sur la simulation du calcul d'une machine de Turing non déterministe par une formule booléenne.

- Le Tableau de Calcul :
  Soit L∈NP. Par définition, il existe une MTND M qui décide L en temps polynomial p(n), où n=|w|. Un calcul de M sur w peut être visualisé comme un tableau (une "computation history") où chaque ligne représente une configuration de la machine à un instant t. Le tableau a une hauteur de p(n) (le nombre d'étapes) et une largeur de p(n) (le nombre maximal de cases de ruban utilisées).77
- Les Variables Booléennes :
  Nous créons un ensemble de variables booléennes pour décrire ce tableau. Ces variables vont encoder l'état de la machine, la position de la tête et le contenu du ruban à chaque instant.
  - Qt,q : Vrai si, à l'étape t, la machine est dans l'état q.
  - Ht,i : Vrai si, à l'étape t, la tête de lecture est à la position i.
  - St,i,σ : Vrai si, à l'étape t, la case i du ruban contient le symbole σ.
    Le nombre total de ces variables est polynomial en n, car t et i vont jusqu'à p(n), et le nombre d'états et de symboles est constant.78
- Les Clauses de la Formule ϕw :
  La formule ϕw est une conjonction de plusieurs sous-formules, chacune imposant une contrainte pour que le tableau représente un calcul valide et acceptant.
  1. **ϕcell :** Assure la cohérence du tableau. Pour chaque cellule (t,i) du tableau, elle doit contenir exactement un symbole. Cela se traduit par des clauses comme (⋁σ∈ΓSt,i,σ) et (¬St,i,σ1∨¬St,i,σ2) pour σ1≠σ2. Des clauses similaires garantissent un seul état et une seule position de tête à chaque instant.
  2. **ϕstart :** Encode la configuration initiale. À t=0, la machine est dans l'état q0, la tête est à la position 0, et le ruban contient l'entrée w suivie de symboles blancs.
  3. **ϕmove :** C'est la partie la plus complexe. Elle encode la logique de la fonction de transition δ de la MTND M. Pour chaque "fenêtre" locale de 2x3 cases dans le tableau (centrée sur la case (t,i)), cette sous-formule garantit que le contenu de la case (t+1,i) est une conséquence légale du contenu des trois cases (t,i−1),(t,i),(t,i+1) et de l'état de la machine. Si la tête n'est pas sur la case i à l'instant t, le symbole ne doit pas changer. Si la tête est sur la case i, le changement d'état, de symbole et de position de la tête doit correspondre à l'une des transitions possibles dans δ.
  4. **ϕaccept :** Assure que le calcul est acceptant. Elle stipule qu'à la dernière étape, t=p(n), la machine doit être dans un état d'acceptation : ⋁q∈FQp(n),q.

- Conclusion de la Preuve :
  La formule finale est $\phi w = \phi cell \wedge \phi start \wedge \phi move \wedge \phi accept$.
  - Si $w \in L$, il existe un calcul acceptant de M sur w. Les valeurs des variables booléennes correspondant à ce calcul (par exemple, Qt,q est vrai si la machine est en état q à l'instant t) satisferont toutes les clauses de ϕw. Donc, ϕw est satisfiable.
  - Réciproquement, si ϕw est satisfiable, toute assignation satisfaisante correspond à un tableau de calcul valide qui commence par la configuration initiale et se termine dans un état d'acceptation. Cela implique qu'il existe un calcul acceptant pour w, donc $w \in L$.
    La construction de la formule ϕw est purement mécanique et sa taille est polynomiale en $|w|$, donc la réduction est polynomiale. SAT est donc NP-difficile. Combiné au fait que SAT est dans NP, le théorème est prouvé.77

6.4.3 Exemples de Problèmes NP-Complets

Une fois que l'on dispose d'un premier problème NP-complet (SAT), on peut en prouver d'autres par transitivité. Il suffit de montrer que $LNPC \leq_p Lnouveau$ pour un problème LNPC déjà connu comme NP-complet, et de prouver que $Lnouveau \in NP$.

Réduction de SAT à 3-SAT

Problème 3-SAT : Une restriction de SAT où chaque clause de la formule FNC doit contenir **exactement** trois littéraux.

Théorème : 3-SAT est NP-complet.

Preuve :

1. **3-SAT ∈ NP :** Trivial, pour la même raison que SAT.
2. **SAT ≤p 3-SAT :** Nous devons transformer une formule SAT quelconque ϕ en une formule 3-SAT ϕ' équisatisfiable. Pour chaque clause C de ϕ :
   - **Si C a 1 littéral (l1) :** On la remplace par $(l1 \vee y1 \vee y2) \wedge (l1 \vee \neg y1 \vee y2) \wedge (l1 \vee y1 \vee \neg y2) \wedge (l1 \vee \neg y1 \vee \neg y2)$, où y1,y2 sont de nouvelles variables. Cette conjonction est vraie si et seulement si l1 est vrai.
   - **Si C a 2 littéraux (l1∨l2) :** On la remplace par $(l1 \vee l2 \vee y1) \wedge (l1 \vee l2 \vee \neg y1)$.
   - **Si C a 3 littéraux :** On la garde telle quelle.
   - Si C a k>3 littéraux $(l1 \vee \cdots \vee lk)$: On introduit $k-3$ nouvelles variables $y1,\ldots,yk-3$ et on remplace C par la conjonction de $k-2$ clauses :

     $(l1 \vee l2 \vee y1) \wedge (\neg y1 \vee l3 \vee y2) \wedge \cdots \wedge (\neg yk-3 \vee lk-1 \vee lk)$

     On peut prouver que la clause originale est satisfiable si et seulement si cette nouvelle conjonction de clauses l'est. La transformation est clairement polynomiale.54

Réduction de 3-SAT à CLIQUE

Problème CLIQUE :

- **Instance :** Un graphe non orienté G=(V,E) et un entier k.
- **Question :** Existe-t-il un sous-ensemble de sommets V’⊆V de taille |V’|=k tel que chaque paire de sommets dans V’ est connectée par une arête (formant une clique)?

Théorème : CLIQUE est NP-complet.

Preuve :

1. **CLIQUE ∈ NP :** Le certificat est un ensemble de k sommets. Le vérificateur vérifie en temps polynomial si toutes les paires de sommets de cet ensemble sont bien reliées par une arête.
2. **3-SAT ≤p CLIQUE :** Soit ϕ une formule 3-SAT avec m clauses C1,…,Cm. Nous construisons une instance (G,k) de CLIQUE.
    - **Construction de G :** On pose k=m. Pour chaque littéral li,j (le j-ème littéral de la i-ème clause), on crée un sommet vi,j dans le graphe G.
    - **Ajout des arêtes :** On ajoute une arête entre deux sommets vi,j et vi’,j’ si et seulement si deux conditions sont remplies :
        1. Ils proviennent de clauses différentes (i≠i’).
        2. Ils ne sont pas contradictoires (on n'a pas li,j=¬li’,j’).
    - **Preuve de la réduction :**
        - (⇒) Si ϕ est satisfiable, il existe une assignation qui rend au moins un littéral vrai dans chaque clause. Choisissons un de ces littéraux vrais dans chaque clause Ci. L'ensemble des m sommets correspondants forme une clique de taille m=k dans G. En effet, tous ces sommets proviennent de clauses différentes, et ils ne peuvent pas être contradictoires car ils sont tous vrais dans la même assignation.
        - (⇐) Si G a une clique de taille k=m, cette clique doit contenir exactement un sommet par "groupe" de clause (car les sommets d'un même groupe ne sont pas connectés entre eux). On peut alors construire une assignation satisfaisante : pour chaque sommet vi,j dans la clique, on assigne la valeur "vrai" au littéral correspondant li,j. Comme il n'y a pas d'arête entre des littéraux contradictoires, cette assignation est cohérente et satisfait donc ϕ.
          La construction du graphe est polynomiale en la taille de ϕ.85

Réduction de CLIQUE à VERTEX-COVER

Problème VERTEX-COVER (Couverture par sommets) :

- **Instance :** Un graphe G=(V,E) et un entier k.
- **Question :** Existe-t-il un sous-ensemble de sommets V’⊆V de taille |V’|≤k tel que chaque arête de E a au moins une de ses extrémités dans V’?

Théorème : VERTEX-COVER est NP-complet.

Preuve :

1. **VERTEX-COVER ∈ NP :** Le certificat est un ensemble de k sommets. Le vérificateur parcourt toutes les arêtes du graphe et s'assure que chacune est incidente à au moins un sommet du certificat. C'est polynomial.
2. **CLIQUE ≤p VERTEX-COVER :** Soit (G=(V,E),k) une instance de CLIQUE. On construit une instance (G',k') de VERTEX-COVER.
   - **Construction :** On pose $G'=G^{-}$, le graphe **complémentaire** de G (mêmes sommets, mais une arête existe dans G^{-} si et seulement si elle n'existe pas dans G). On pose $k'=|V|-k$.
   - **Preuve de la réduction :** On montre l'équivalence : G a une clique de taille $k \Leftrightarrow G^{-}$ a une couverture par sommets de taille $|V|-k$.
     - Soit C une clique de taille k dans G. Par définition, il n'y a aucune arête entre les sommets de C dans G^{-}. Donc, pour toute arête (u,v) de G^{-}, au moins un des sommets u ou v ne doit pas être dans C. Cela signifie que l'ensemble $V \setminus C$ est une couverture par sommets de G^{-}. Sa taille est $|V|-k$.
     - Réciproquement, soit S une couverture de taille $|V|-k$ dans G^{-}. Alors, pour toute arête (u,v) de G^{-}, on a $u \in S$ ou $v \in S$. Par contraposée, si $u \in /S$ et $v \in /S$, alors (u,v) n'est pas une arête de G^{-}, ce qui signifie que c'est une arête de G. L'ensemble $V \setminus S$, de taille k, est donc une clique dans G.
       Le calcul du graphe complémentaire est polynomial.89

Réduction de 3-SAT à HAMILTONIAN-PATH

Problème HAMILTONIAN-PATH (Chemin Hamiltonien) :

- **Instance :** Un graphe orienté G=(V,E).
- **Question :** Existe-t-il un chemin simple qui visite chaque sommet de V exactement une fois?

Théorème : HAMILTONIAN-PATH est NP-complet.

Preuve :
Cette réduction est plus complexe et repose sur la construction de "gadgets" dans le graphe pour simuler la logique de la formule 3-SAT.

1. **HAM-PATH ∈ NP :** Un certificat est une permutation des sommets. Le vérificateur vérifie si les arêtes correspondantes existent dans le graphe.
2. **3-SAT ≤p HAM-PATH :** Soit ϕ une formule 3-SAT avec n variables x1,…,xn et m clauses C1,…,Cm.
   - **Gadget de variable :** Pour chaque variable xi, on crée une structure en "diamant" ou une longue chaîne de sommets. Un chemin hamiltonien ne peut traverser cette structure que de deux manières : de "gauche à droite" (ce qui correspondra à xi=vrai) ou de "droite à gauche" (xi=faux).
   - **Gadget de clause :** Pour chaque clause Cj, on crée un unique sommet cj.
   - **Connexions :** On relie les gadgets de variables en série. Ensuite, on connecte les gadgets de variables aux gadgets de clauses. Si le littéral xi apparaît dans la clause Cj, on ajoute des arêtes permettant un détour du chemin "gauche-droite" du gadget de xi pour passer par le sommet cj. Si ¬xi apparaît dans Cj, on ajoute des arêtes pour un détour depuis le chemin "droite-gauche".
   - **Preuve de la réduction :** Un chemin hamiltonien doit visiter tous les sommets, y compris ceux des clauses.

Pour visiter un sommet de clause cj, le chemin doit emprunter l'un des détours prévus. Ce détour n'est possible que si le chemin traverse le gadget de variable correspondant dans la "bonne" direction (celle qui rend le littéral vrai). Donc, un chemin hamiltonien existe si et seulement si on peut choisir une direction pour chaque gadget de variable (une assignation de vérité) de telle sorte que chaque sommet de clause soit visité (chaque clause est satisfaite).[94]

Réduction de HAMILTONIAN-PATH à TSP (décision)

Problème TSP (Voyageur de Commerce, version décision) :

- **Instance :** Un ensemble de n villes, une matrice de distances d(vi,vj) entre chaque paire de villes, et un budget B.
- **Question :** Existe-t-il un tour (un cycle qui visite chaque ville exactement une fois) de longueur totale inférieure ou égale à B?

Théorème : TSP (décision) est NP-complet.

Preuve :

1. **TSP ∈ NP :** Le certificat est une permutation des villes. Le vérificateur calcule la longueur du tour et la compare à B.
2. **HAMILTONIAN-PATH ≤p TSP :** On peut réduire la version **HAMILTONIAN-CYCLE** (qui est aussi NP-complète) à TSP, ce qui est légèrement plus simple. Soit G=(V,E) une instance de HAM-CYCLE. On construit une instance de TSP.
   - Construction : L'ensemble des villes est V. On définit les distances comme suit pour chaque paire de villes (u,v) :

     d(u,v)={12si (u,v)∈Esinon

     On pose le budget B=|V|.
   - **Preuve de la réduction :**
     - (⇒) Si G a un cycle hamiltonien, ce cycle est un tour dans l'instance de TSP. Sa longueur est la somme des poids des arêtes, qui sont toutes à 1. La longueur totale est donc |V|, ce qui respecte le budget B.
     - (⇐) Si l'instance de TSP a un tour de longueur ≤|V|, ce tour doit nécessairement n'emprunter que des arêtes de poids 1 (car s'il utilisait ne serait-ce qu'une arête de poids 2, la longueur totale serait >|V|). Un tour de longueur |V| utilisant uniquement des arêtes de poids 1 correspond exactement à un cycle hamiltonien dans le graphe original G.
       Cette chaîne de réductions illustre la puissance du concept de NP-complétude. La difficulté inhérente d'un problème de logique booléenne (3-SAT) se propage à des problèmes de graphes (CLIQUE, VERTEX-COVER, HAM-PATH) et finalement à un problème d'optimisation numérique fondamental (TSP), révélant une connexion profonde et inattendue entre des domaines apparemment très éloignés.

6.5 Au-delà de NP

Si la distinction entre **P** et **NP** est au cœur de la théorie de la complexité, elle n'est que le point de départ d'une cartographie bien plus vaste et complexe du paysage computationnel. De nombreuses autres classes de complexité ont été définies pour capturer différentes nuances de difficulté, que ce soit en considérant des ressources autres que le temps déterministe, ou en explorant des modèles de calcul plus exotiques.

6.5.1 Classes co-NP et la Hiérarchie Polynomiale

La définition de **NP** est intrinsèquement asymétrique : elle concerne les problèmes pour lesquels une réponse "OUI" possède un certificat court et facile à vérifier. Qu'en est-il des problèmes pour lesquels une réponse "NON" est facile à vérifier?

La Classe co-NP

Définition : La classe co-NP est l'ensemble des langages L dont le complément $L^- = \Sigma^* \setminus L$ est dans NP.

$$co-NP = \{L \mid L^- \in NP\}$$

De manière équivalente, un problème est dans co-NP s'il existe un vérificateur polynomial V tel que pour toute instance x :

$$x \in L \iff \forall c, |c| \leq p(|x|), V(x,c) = \text{accepte}$$

Cela signifie qu'une réponse "NON" ($x \notin L$) peut être démontrée par un contre-exemple (un certificat c tel que V(x,c) rejette).45

- Exemple : TAUTOLOGY
  Le problème TAUTOLOGY demande si une formule booléenne donnée est une tautologie (vraie pour toutes les assignations de variables). Ce problème est dans co-NP. Son complément est le problème de savoir si une formule n'est pas une tautologie, ce qui est équivalent à demander si elle est satisfiable par au moins une assignation qui la rend fausse. Le complément de TAUTOLOGY est donc équivalent à SAT, qui est dans NP.

Le Problème NP vs co-NP

On sait que $P \subseteq NP$ et $P \subseteq co-NP$ (car **P** est fermée par complémentation). Cependant, la relation entre **NP** et **co-NP** est une question ouverte majeure.

- On ne sait pas si $NP = co-NP$.

- On conjecture fortement que NP≠co−NP.
- Si l'on pouvait prouver que NP=co−NP, cela impliquerait immédiatement que P=NP. En effet, si P=NP, alors comme **P** est fermée par complémentation (P=co−P), on aurait NP=P=co−P=co−NP.

La Hiérarchie Polynomiale (PH)

La hiérarchie polynomiale est une généralisation des classes **P**, **NP** et **co-NP** qui forme une tour infinie de classes de complexité de plus en plus puissantes. Elle est définie en utilisant des machines de Turing polynomiales avec accès à un **oracle**. Un oracle est une "boîte noire" capable de résoudre instantanément un problème d'une classe de complexité donnée.

Définition par Oracles :

- $\Delta_0^p = \Sigma_0^p = \Pi_0^p = P$
- Pour $k \geq 0$:
  - $\Delta_{k+1}^p = P^{\Sigma_k^p}$ (problèmes résolus en temps polynomial avec un oracle pour un problème de Σ_k^p)
  - $\Sigma_{k+1}^p = NP^{\Sigma_k^p}$ (problèmes résolus par une MTND polynomiale avec un oracle pour Σ_k^p)
  - $\Pi_{k+1}^p = co-NP^{\Sigma_k^p}$

On a ainsi :

- $\Sigma_1^p = NP$
- $\Pi_1^p = co-NP$
- $\Sigma_2^p = NP^{NP}$

Une définition équivalente et plus intuitive utilise des quantificateurs alternés. Un langage L est dans Σ_k^p s'il peut être défini par une formule avec k quantificateurs alternés commençant par un quantificateur existentiel, suivi d'un prédicat polynomial.

$$L = \{x \mid \exists y_1 \forall y_2 \ldots Q_k y_k, V(x, y_1, \ldots, y_k)\}$$

où V est un prédicat vérifiable en temps polynomial et la taille des y_i est polynomiale en $|x|$.
La hiérarchie polynomiale est l'union de toutes ces classes :

$$PH = \bigcup_{k \geq 0} \Sigma_k^p$$

Si à un certain niveau k, on a $\Sigma_k^p = \Pi_k^p$, alors la hiérarchie "s'effondre" à ce niveau, c'est-à-dire que $PH = \Sigma_k^p$. La conjecture P≠NP est équivalente à l'affirmation que la hiérarchie ne s'effondre pas au niveau 0. La conjecture NP≠co−NP est équivalente à l'affirmation qu'elle ne s'effondre pas au niveau 1.98

6.5.2 Complexité en Espace

La complexité en espace se concentre sur la quantité de mémoire requise pour résoudre un problème, plutôt que sur le temps de calcul.

Les Classes de Complexité en Espace

- **L (Logarithmic Space) :** La classe des problèmes de décision résolubles par une MTD en espace O(logn). L'espace logarithmique est très faible ; il ne permet même pas de stocker l'entrée en mémoire. On utilise un modèle de machine avec un ruban d'entrée en lecture seule et un ou plusieurs rubans de travail. Seul l'espace sur les rubans de travail est compté.
- **NL (Nondeterministic Logarithmic Space) :** La classe des problèmes résolubles par une MTND en espace O(logn).
- PSPACE (Polynomial Space) : La classe des problèmes résolubles par une MTD en espace polynomial, O(nk).

  PSPACE=k∈N⋃DSPACE(nk)
- **NPSPACE (Nondeterministic Polynomial Space) :** La classe des problèmes résolubles par une MTND en espace polynomial.[22]

Le Théorème de Savitch

Contrairement au temps, où le non-déterminisme est supposé apporter une puissance de calcul supplémentaire (P vs NP), en matière d'espace, ce n'est pas le cas pour les bornes polynomiales (et supérieures). C'est le résultat fondamental du théorème de Savitch.

Théorème (Savitch, 1970) : Pour toute fonction s(n)≥logn,

NSPACE(s(n))⊆DSPACE(s(n)2)

Ce théorème stipule que tout algorithme non déterministe utilisant un espace s(n) peut être simulé par un algorithme déterministe utilisant un espace quadratique, s(n)2.
Conséquence Majeure : Une conséquence directe et profonde de ce théorème est l'égalité des classes polynomiales en espace :

PSPACE=NPSPACE

La preuve du théorème de Savitch repose sur un algorithme récursif déterministe qui vérifie l'accessibilité entre deux configurations d'une MTND. En utilisant une approche de type "diviser pour régner", il vérifie s'il existe une configuration intermédiaire, réduisant ainsi la profondeur de la récursion au détriment d'un recalcul, ce qui permet de conserver un espace de pile limité.11

Relations entre les Classes de Temps et d'Espace

Nous avons maintenant une hiérarchie plus complète des classes de complexité. Les relations connues sont les suivantes :
$$ \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXPTIME} $$

- P⊆PSPACE : Un calcul en temps polynomial ne peut explorer qu'un nombre polynomial de cases mémoire.
- NP⊆PSPACE : On peut simuler une MTND polynomiale en temps en explorant tout son arbre de calcul. L'espace requis pour cela est polynomial (on peut réutiliser l'espace pour chaque branche).
- PSPACE⊆EXPTIME : Une machine utilisant un espace polynomial p(n) ne peut avoir qu'un nombre fini de configurations distinctes (de l'ordre de 2O(p(n))). Si le calcul dure plus longtemps, il doit boucler. Un algorithme peut donc détecter les boucles et s'arrêter en temps au plus exponentiel.[22]

Les théorèmes de hiérarchie en temps et en espace prouvent que certaines de ces inclusions sont strictes : L⊂PSPACE et P⊂EXPTIME. Cependant, toutes les autres inclusions (NL vs P, P vs NP, NP vs PSPACE) sont conjecturées strictes, mais non prouvées.

6.5.3 Classes de Complexité Probabilistes

Les algorithmes déterministes et non déterministes ne sont pas les seuls modèles de calcul. L'introduction de l'aléa dans les algorithmes a donné naissance à une famille riche et puissante de classes de complexité.

La Machine de Turing Probabiliste

Une machine de Turing probabiliste est une MTND où, à chaque étape, chaque choix possible est assorti d'une probabilité (typiquement, une machine qui peut "lancer une pièce" et prendre une transition ou une autre avec une probabilité de 1/2).

Définitions des Classes Probabilistes

Les classes probabilistes sont définies en fonction de la probabilité d'erreur de l'algorithme.

- RP (Randomized Polynomial Time) :
  Cette classe capture les problèmes avec des algorithmes probabilistes à erreur unilatérale.
  Un langage L est dans RP si il existe un algorithme probabiliste en temps polynomial A tel que :

- Si $x \in L$, alors $P(A(x) \text{ accepte}) \geq 1/2$.
- Si $x \notin L$, alors $P(A(x) \text{ accepte}) = 0$.
  L'algorithme ne peut se tromper que pour les instances positives (faux négatifs), mais jamais pour les instances négatives (pas de faux positifs). La probabilité de succès peut être amplifiée à une valeur arbitrairement proche de 1 en répétant l'algorithme.

- BPP (Bounded-error Probabilistic Polynomial Time) :
  Cette classe est considérée comme la véritable classe des problèmes "efficacement solubles" par des moyens probabilistes. Elle autorise une erreur bilatérale, mais bornée.
  Un langage L est dans BPP si il existe un algorithme probabiliste en temps polynomial A tel que :
  - Si $x \in L$, alors $P(A(x) \text{ accepte}) \geq 2/3$.
  - Si $x \notin L$, alors $P(A(x) \text{ accepte}) \leq 1/3$.
    La "marge" entre 2/3 et 1/3 garantit que la probabilité d'erreur est significativement inférieure à 1/2. Comme pour RP, cette probabilité peut être rendue arbitrairement faible par répétition.
- ZPP (Zero-error Probabilistic Polynomial Time) :
  Cette classe correspond aux algorithmes de type "Las Vegas". Ils ne se trompent jamais, mais leur temps d'exécution peut varier.
  Un langage L est dans ZPP si il existe un algorithme probabiliste A qui retourne toujours la bonne réponse ("OUI" ou "NON") et dont le temps d'exécution attendu (en moyenne) est polynomial.
  On peut montrer que $ZPP = RP \cap co-RP$.

Relations et Conjectures

Les relations suivantes sont connues :

$P \subseteq ZPP \subseteq RP \subseteq BPP$
$RP \subseteq NP$

La relation entre BPP et NP est inconnue. Cependant, une conjecture majeure en théorie de la complexité est que l'aléa n'ajoute pas fondamentalement de puissance de calcul. De nombreux résultats en "dérandomisation" suggèrent que les algorithmes probabilistes pourraient être simulés par des algorithmes déterministes sans surcoût exponentiel.
Conjecture : $P = BPP$.

Si cette conjecture était vraie, cela signifierait que tout ce qui peut être calculé efficacement avec des pièces de monnaie peut aussi l'être sans.

L'existence de ce riche "zoo" de classes de complexité, dont la plupart des questions de séparation restent ouvertes, met en lumière une réalité fondamentale de l'informatique théorique : nous sommes devenus experts dans la classification des problèmes et la compréhension de leurs relations relatives (via les réductions), mais nous restons profondément ignorants quant à la puissance de calcul absolue requise pour les résoudre. La théorie de la complexité nous a fourni une carte détaillée de notre propre ignorance.

Tableau Récapitulatif des Classes de Complexité Majeures

Classe	Définition Formelle	Définition Intuitive	Problème Complet Typique	Relations Connues
P	$\bigcup_k \text{DTIME}(n^k)$	Problèmes résolubles efficacement (déterministe).	- (Ex: Plus court chemin)	$P \subseteq NP$
NP	$\bigcup_k \text{NTIME}(n^k)$	Problèmes dont une solution est vérifiable efficacement.	SAT, 3-SAT, CLIQUE	$P \subseteq NP \subseteq PSPACE$
co-NP	$\{L \mid \bar{L} \in NP\}$	Problèmes dont un contre-exemple est vérifiable efficacement.	TAUTOLOGY	$P \subseteq co-NP$
PSPACE	$\bigcup_k \text{DSPACE}(n^k)$	Problèmes résolubles avec une mémoire polynomiale.	QSAT, Échecs généralisés	$NP \subseteq PSPACE = NPSPACE$
EXPTIME	$\bigcup_k \text{DTIME}(2^{n^k})$	Problèmes résolubles en temps exponentiel.	Go généralisé	$PSPACE \subseteq EXPTIME$
L	$\text{DSPACE}(\log n)$	Problèmes résolubles avec une mémoire logarithmique.	- (Ex: Palindrome)	$L \subseteq NL \subseteq P$

NL	NSPACE(logn)	Problèmes résolubles en mémoire log. (non déterministe).	PATH (Accessibilité)	NL=co−NL
BPP	Prob. bornée, temps poly.	Problèmes résolubles efficacement avec un algorithme probabiliste.	-	P⊆BPP (conjecturé P=BPP)
RP	Erreur unilatérale, temps poly.	Problèmes "OUI" détectables rapidement par hasard.	- (Ex: Test de primalité Miller-Rabin)	ZPP⊆RP⊆NP
ZPP	Erreur nulle, temps poly. attendu	Problèmes résolubles efficacement et sans erreur par hasard.	-	P⊆ZPP=RP∩co−RP

Ouvrages cités

1. Théorie de la calculabilité - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Th%C3%A9orie_de_la_calculabilit%C3%A9
2. Théorie de la calculabilité: Concepts, Exercices | StudySmarter, dernier accès : septembre 21, 2025, https://www.studysmarter.fr/resumes/mathematiques/logique-et-fondements/theorie-de-la-calculabilite/
3. Calculabilité et complexité - Paul Brunet's Home Page, dernier accès : septembre 21, 2025, https://paul.brunet-zamansky.fr/cours/cc.html
4. Alan Turing : du calculable à l'indécidable - Philosophie, Science et Société, dernier accès : septembre 21, 2025, https://philosciences.com/alan-turing-du-calculable-a-l-indecidable
5. COMPLEXITÉ ALGORITHMIQUE - l'IRIF, dernier accès : septembre 21, 2025, https://www.irif.fr/~sperifel/complexite.pdf
6. Complexité et calculabilité - LaBRI, dernier accès : septembre 21, 2025, https://www.labri.fr/perso/anca/MC/poly.pdf
7. La théorie de la complexité algorithmique pour calculer efficacement - Interstices.info, dernier accès : septembre 21, 2025, https://interstices.info/la-theorie-de-la-complexite-algorithmique/

8. Complexité | PDF | Théorie de la complexité (informatique théorique) - Scribd, dernier accès : septembre 21, 2025, https://fr.scribd.com/document/866663459/complexite
9. 1NSI : Cours Complexité d'un Algorithme - GitLab, dernier accès : septembre 21, 2025, https://eskool.gitlab.io/1nsi/algorithmique/complexite/
10. Théorie de la complexité: Algorithmes, Problèmes | StudySmarter, dernier accès : septembre 21, 2025, https://www.studysmarter.fr/resumes/informatique/theorie-de-la-computation/theorie-de-la-complexite/
11. La machine de Turing (2 / 2) - [Les nouvelles technologies pour l'enseignement des mathématiques] - MathémaTICE, dernier accès : septembre 21, 2025, http://revue.sesamath.net/spip.php?article1508
12. Thèse de Church-Turing: Concepts clés, Applications | StudySmarter, dernier accès : septembre 21, 2025, https://www.studysmarter.fr/resumes/informatique/theorie-de-la-computation/these-de-church-turing/
13. Church–Turing thesis - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Church%E2%80%93Turing_thesis
14. M1 informatique – Lyon 1 2017 – 2018 M1if09 – Calculabilité et complexité Support de cours 1 / 22 Calculabilité et com - CNRS, dernier accès : septembre 21, 2025, https://perso.liris.cnrs.fr/sylvain.brandel/wiki/lib/exe/fetch.php%3Fmedia%3Dens:m1if09:m1if09-support-de-cours.pdf
15. Leçon 913 : Machines de Turing. Applications., dernier accès : septembre 21, 2025, https://perso.eleves.ens-rennes.fr/people/julie.parreaux/fichiers_agreg/Memoire_913.pdf
16. Complexity class - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Complexity_class
17. Complexity - LMF, dernier accès : septembre 21, 2025, https://home.lmf.cnrs.fr/downloads/SergeHaddad/transparents-Complexite.pdf
18. Machines de Turing et complexité algorithmique Olivier Hudry, dernier accès : septembre 21, 2025, ftp://mse.univ-paris1.fr/pub/mse/cahiers2003/B03119.pdf
19. Décidabilité et complexité 1/4 : Machines de Turing - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=X610pII4_J8
20. Chapitre 3 : Les classes de complexité - Dimitri Watel, dernier accès : septembre 21, 2025, http://dimitri.watel.free.fr/teaching/s4_cal/courses/2018/03_ClassesComplexite.pdf
21. Complexité en temps des machines de Turing non déterministes, Classe NP - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=17gi9-nUxNo
22. Complexité en espace - l'IRIF, dernier accès : septembre 21, 2025, https://www.irif.fr/~carton/Enseignement/Complexite/ENS/Cours/MasterInfo/space.html
23. Séance 6 : Décidabilité et Complexité, dernier accès : septembre 21, 2025, https://www.i3s.unice.fr/~nlt/cours/licence/it/s6_itdut_poly.pdf
24. Complexité - LACL, dernier accès : septembre 21, 2025, https://www.lacl.fr/fmadelaine/Download/M1Droit/Automates/DetailsComplexite.pdf
25. Complexité d'un algorithme I Généralités, dernier accès : septembre 21, 2025, https://cahier-de-prepa.fr/psi-michelet/download?id=239
26. Algorithmique et Complexité, dernier accès : septembre 21, 2025, https://homepages.laas.fr/ehebrard/papers/handout2019.pdf
27. Analyse de la complexité des algorithmes — Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Analyse_de_la_complexit%C3%A9_des_algorithmes
28. ANALYSE D'ALGORITHMES, dernier accès : septembre 21, 2025, https://www.site.uottawa.ca/~stan/csi2514/trs/chap02
29. Chapitre 10 - Bases de l'analyse de complexité d'algorithmes - Départements d'enseignement et de

recherche, dernier accès : septembre 21, 2025, https://www.enseignement.polytechnique.fr/informatique/INF412/uploads/Main/chap10-good.pdf
30. Complexité dans le meilleur des cas - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Complexit%C3%A9_dans_le_meilleur_des_cas
31. Notations Asymptotiques - Learn X in Y Minutes, dernier accès : septembre 21, 2025, https://learnxinyminutes.com/fr/asymptotic-notation/
32. Asymptotic Notations for Analysis of Algorithms - GeeksforGeeks, dernier accès : septembre 21, 2025, https://www.geeksforgeeks.org/dsa/types-of-asymptotic-notations-in-complexity-analysis-of-algorithms/
33. Algorithmique IV - Notations asymptotiques [JP. Zanotti], dernier accès : septembre 21, 2025, https://zanotti.univ-tln.fr/ALGO/I31/Asymptotiques.html
34. Asymptotic Notation: O(), o(), Ω(), ω(), and Θ() The Idea The Definitions, dernier accès : septembre 21, 2025, https://www2.cs.arizona.edu/classes/cs345/summer14/files/bigO.pdf
35. Confus au sujet des notations Big-oh, Theta et Omega : r/algorithms - Reddit, dernier accès : septembre 21, 2025, https://www.reddit.com/r/algorithms/comments/13rcitf/confused_about_bigoh_theta_and_omega_notations/?tl=fr
36. Asymptotic Notation: Omega and Theta, dernier accès : septembre 21, 2025, https://people.iith.ac.in/aravind/Files-CS2443/asymptotics-six.pdf
37. What is Big-Oh, Big-Omega and Big-Theta Notation | Medium, dernier accès : septembre 21, 2025, https://medium.com/@alejandro.itoaramendia/a-guide-to-understanding-big-o-big-%CF%89-and-big-%CE%B8-notation-7c407a6824d9
38. Properties of Asymptotic Notations (Reflexive Property) - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=sIwtCChDZhM
39. 2 - Outils Mathématiques Pour La Complexité Algorithmique | PDF | Fonction exponentielle, dernier accès : septembre 21, 2025, https://fr.scribd.com/presentation/828633431/2-Outils-mathematiques-pour-la-complexite-algorithmique
40. Notation asymptotique - Université de Montréal, dernier accès : septembre 21, 2025, http://www.iro.umontreal.ca/~hamelsyl/grandO.pdf
41. real analysis - How to prove Big O, Omega and Theta asymptotic ..., dernier accès : septembre 21, 2025, https://math.stackexchange.com/questions/4067252/how-to-prove-big-o-omega-and-theta-asymptotic-notations
42. Properties of Asymptotic Notations (Transitive Property) - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=1qnXNyGQr7k
43. Analyse asymptotique - Mathieu Mansuy, dernier accès : septembre 21, 2025, http://www.mathieu-mansuy.fr/pdf/PCSI5-chapitre15.pdf
44. P versus NP - IGM, dernier accès : septembre 21, 2025, http://igm.univ-mlv.fr/~nicaud/M1/cours7.pdf
45. Chapitre 8 : Classes de complexité P et NP - LaBRI, dernier accès : septembre 21, 2025, https://www.labri.fr/perso/betrema/MC/MC8.html
46. Cobham's thesis - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/Cobham%27s_thesis
47. mamund.site44.com, dernier accès : septembre 21, 2025, https://mamund.site44.com/books/20150201-152212000.html
48. Thèse de Cobham - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Th%C3%A8se_de_Cobham
49. INFORMATION TO USERS - Iowa State University Digital Repository, dernier accès : septembre 21, 2025, https://dr.lib.iastate.edu/bitstreams/d43b2f63-5d9e-4cc2-a031-b37c16cb8741/download

50. Les problèmes NP-complets - Math93, dernier accès : septembre 21, 2025, https://www.math93.com/19-histoire-des-mathematiques/1167-les-problemes-np-complets.html
51. Algorithmes efficaces et Complexité - Du Page Rank au problème P=NP, dernier accès : septembre 21, 2025, https://webusers.imj-prg.fr/~richard.lassaigne/coursLogique/complexite-algos-slides.pdf
52. Algorithmes rapides pour les polynômes, séries formelles et matrices - MATHEXP, dernier accès : septembre 21, 2025, https://mathexp.eu/bostan/publications/Bostan10.pdf
53. COMPLEXITÉ CLASSES P ET NP - CNRS, dernier accès : septembre 21, 2025, https://perso.liris.cnrs.fr/sbrandel/wiki/lib/exe/fetch.php%3Fmedia%3Dens:m1if09:m1if09-cm07.pdf
54. Problème SAT - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Probl%C3%A8me_SAT
55. NP (complexité) - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/NP_(complexit%C3%A9)
56. NP : 2 définitions équivalentes, dernier accès : septembre 21, 2025, http://lim.univ-reunion.fr/staff/fred/Enseignement/AlgorithmiqueAvancee/doc/NP-2defns-equiv.pdf
57. Est-ce que P = NP - Science Étonnante, dernier accès : septembre 21, 2025, https://scienceetonnante.com/blog/2020/07/17/est-ce-que-p-np/
58. Cours 9: NP-complétude. Suite. Preuve Le gadget, dernier accès : septembre 21, 2025, https://www.enseignement.polytechnique.fr/informatique/INF412/AMPHIS/cours9-handout2x2.pdf
59. NP (complexity) - Wikipedia, dernier accès : septembre 21, 2025, https://en.wikipedia.org/wiki/NP_(complexity)
60. Problème P $\stackrel{?}{=}$ NP - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Probl%C3%A8me_P_%E2%89%9F_NP
61. Le problème P=NP - Guide à l'usage du promeneur égaré dans le paradis (ou l'enfer) de la complexité des algorithme, dernier accès : septembre 21, 2025, https://webusers.imj-prg.fr/~richard.lassaigne/exposes/P=NP.pdf
62. Le problème P = NP ? expliqué simplement - Ambient IT, dernier accès : septembre 21, 2025, https://www.ambient-it.net/p-np/
63. Pourquoi les gens (du domaine) croient-ils fermement que P != NP ? : r/math - Reddit, dernier accès : septembre 21, 2025, https://www.reddit.com/r/math/comments/1lji77l/why_do_people_in_the_field_strongly_believe_p_np/?tl=fr
64. Problème P vs NP: Informatique, Mathématiques - StudySmarter, dernier accès : septembre 21, 2025, https://www.studysmarter.fr/resumes/mathematiques/logique-et-fondements/probleme-p-vs-np/
65. What are the implications of solving the P versus NP problem? What practical eff, dernier accès : septembre 21, 2025, https://news.ycombinator.com/item?id=15010538
66. Explained: P vs. NP | MIT News | Massachusetts Institute of Technology, dernier accès : septembre 21, 2025, https://news.mit.edu/2009/explainer-pnp
67. If the P versus NP problem gets solved, somehow, what would be the biggest impact?, dernier accès : septembre 21, 2025, https://www.quora.com/If-the-P-versus-NP-problem-gets-solved-somehow-what-would-be-the-biggest-impact
68. TD5 - Réductions polynomiales 1 Rappel de cours - LIMOS, dernier accès : septembre 21, 2025, https://perso.limos.fr/ffoucaud/Teaching/Qualite_algo/2024-2025/TD5-Reductions.pdf
69. Leçon 4. Réductions polynomiales, dernier accès : septembre 21, 2025, https://datamove.imag.fr/denis.trystram/SupportsDeCours/lesson4reductions.pdf
70. Réduction polynomiale - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/R%C3%A9duction_polynomiale
71. What is polynomial time reduction? - Educative.io, dernier accès : septembre 21, 2025,

https://www.educative.io/answers/what-is-polynomial-time-reduction
72. Chapter 20 Polynomial Time Reductions, dernier accès : septembre 21, 2025, https://courses.grainger.illinois.edu/cs473/fa2011/lec/20_notes.pdf
73. NP-difficile - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/NP-difficile
74. Problème NP-complet - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Probl%C3%A8me_NP-complet
75. Qu'est-ce qu'un problème NP-difficile EXACTEMENT ? : r/computerscience - Reddit, dernier accès : septembre 21, 2025, https://www.reddit.com/r/computerscience/comments/op4bap/what_exactly_is_an_nphard_problem/?tl=fr
76. Complexité Il existe, grosso modo, deux types de problèmes. Les faciles et les probablement difficiles. Sans faire trop de for - GERAD, dernier accès : septembre 21, 2025, https://www.gerad.ca/Sebastien.Le.Digabel/MTH8415/8_Complexite.pdf
77. Développement : Théorème de Cook, dernier accès : septembre 21, 2025, https://perso.eleves.ens-rennes.fr/~tpier758/agreg/dvpt/info/cook.pdf
78. The Cook-Levin Theorem - UBC Computer Science, dernier accès : septembre 21, 2025, https://www.cs.ubc.ca/~condon/cpsc506/handouts/Cook-Levin.pdf
79. Théorème de Cook-Levin, dernier accès : septembre 21, 2025, https://www.lri.fr/~hellouin/Agreg/Cook-Levin.pdf
80. 9.1 Cook-Levin Theorem, dernier accès : septembre 21, 2025, https://pages.cs.wisc.edu/~shuchi/courses/787-F07/scribe-notes/lecture09.pdf
81. The Cook-Levin Theorem - Cornell: Computer Science, dernier accès : septembre 21, 2025, https://www.cs.cornell.edu/~msoloviev/slides/4820/lecture3.pdf
82. Reduction from SAT to 3SAT - CSE IIT KGP, dernier accès : septembre 21, 2025, https://cse.iitkgp.ac.in/~palash/2018AlgoDesignAnalysis/SAT-3SAT.pdf
83. 21.6.2 Reducing SAT to 3SAT, dernier accès : septembre 21, 2025, https://courses.grainger.illinois.edu/cs374/fa2020/lec_prerec/21/21_6_2_0.pdf
84. 3-SAT Is NP-Complete - UTEP CS, dernier accès : septembre 21, 2025, https://www.cs.utep.edu/vladik/cs5315.20/31.pdf
85. 28.15. Reduction of 3-SAT to Clique - OpenDSA, dernier accès : septembre 21, 2025, https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/threeSAT_to_clique.html
86. 3sat to clique reduction program - Computer Science Stack Exchange, dernier accès : septembre 21, 2025, https://cs.stackexchange.com/questions/162226/3sat-to-clique-reduction-program
87. Reduction from 3Sat problem to Decision Clique Problem | NP Complete Problem, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=vOTC42zADh4
88. NP-Complete Problems - Stanford CS Theory, dernier accès : septembre 21, 2025, https://theory.stanford.edu/~trevisan/cs154-12/np-reductions.pdf
89. 1 VERTEX-COVER → CLIQUE, dernier accès : septembre 21, 2025, https://www.clear.rice.edu/comp487/VC_Clique.pdf
90. Note_Oct14, dernier accès : septembre 21, 2025, https://users.cs.fiu.edu/~giri/teach/UoM/7713/f98/wyang/wyang.html
91. 20191122 - k-Clique to Vertex Cover.pdf, dernier accès : septembre 21, 2025, https://ciscwww.cs.queensu.ca/courses/cisc365/Record/20191122%20-%20k-Clique%20to%20Vertex%20Cover.pdf
92. Lecture 15: Independent Set, Clique, and Vertex Cover, dernier accès : septembre 21, 2025, https://faculty.cc.gatech.edu/~ladha/S24/3510/L15.pdf
93. Proof that vertex cover is NP complete - GeeksforGeeks, dernier accès : septembre 21, 2025,

https://www.geeksforgeeks.org/dsa/proof-that-vertex-cover-is-np-complete/
94. Polynomial-time linear-reduction from Directed Hamiltonian Path Problem to 3SAT, dernier accès : septembre 21, 2025, https://cs.stackexchange.com/questions/120264/polynomial-time-linear-reduction-from-directed-hamiltonian-path-problem-to-3sat
95. 28.18. Reduction of 3-SAT to Hamiltonian Cycle - OpenDSA, dernier accès : septembre 21, 2025, https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/threeSAT_to_hamiltonianCycle.html
96. 3SAT to Hamiltonian cycle reduction - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=rgMrzCWHiNo
97. Hamiltonian Path is NP-Complete (Directed, Reduction from 3SAT) - YouTube, dernier accès : septembre 21, 2025, https://www.youtube.com/watch?v=4r78NtOnWWA
98. Hiérarchie polynomiale - Wikipédia, dernier accès : septembre 21, 2025, https://fr.wikipedia.org/wiki/Hi%C3%A9rarchie_polynomiale