|  |  |  |
| --- | --- | --- |
| **Ministère de l’Enseignement Supérieur****et de la Recherche Scientifique****UNIVERSITE DE GUELMA** **Faculté de Mathématiques et de l’Informatique et des Sciences de la Matière****Département de l’Informatique** |  | **وزارة التـعليم العالي والبحث العلمي****جـــــــامـعـة قــــالمــــــة****كليـة الرياضيات و الإعلام الآلي و علو م المادة****قسم الإعلام الآلي** |

2016

Polycopié des exercices

Master : Logique Mathématique

Et Fondements de l’Informatique

**Responsables du module Dr. KOUAHLA Zineddine**

Université de Guelma

**Avertissement**

Ce document en est à sa première présentation. Malgré des soins particuliers consacrés à sa conception, l’auteur ne doute pas de la présence d’erreurs. Il prie le lecteur d’être indulgent, et le remercie de lui signaler les erreurs potentielles et des améliorations éventuelles. L’adresse électronique de l’auteur est kouahla.zineddine@univ-guelma.dz

**Remerciements**

 L’auteur n’ayant pas inventé la logique, il s’est contenté d’exposer ce qu’il pense devoir être les notions fondamentales de la logique, en s’inspirant des travaux et œuvres de ses aînés. Sans qu’il en soit toujours fait mention explicitement, l’auteur a été guidé par les ouvrages cités en bibliographie, en particulier ceux de **Lalement**, **Gochet** et **Gribomont**, et **Quine**. L’auteur s’est aussi inspiré d’un cours de **Denis** **Lugiez**, chercheur au CRIN (Centre de Recherche en Informatique de Nancy).

Enfin, l’auteur remercie **Séridi Hamid** pour de nombreuses et fructueuses discussions autour de ce cours.

**Logique est la science de la raison**

**SYLLABUS**

Unit2 d’enseignement : UE5 (Systèmes intelligents2)

Matière : logique et fondements de l’informatique

Domaine/ Filière : 1er année Master Informatique académique.

Semestre : 1 et 2

Crédit : 06

Volume Horaire Hebdomadaire Total : 4h 30 mn

Cours Magistral : 3h

Travaux dirigés : 1,5h

Enseignant responsable : Dr KOUAHLA Zin-eddine

E-mail : zineddine.kouahla@univ-guelma.dz

**Objectif du module**

Pour ce module nous avons fixé deux objectifs :

1. Introduire les étudiants à la logique mathématique et en particulier a la méthode de la démonstration
2. Fournir aux étudiants les bases nécessaires afin de pouvoir comprendre le fonctionnement de la plupart des outils de démonstration automatique développés en particulier dans le monde académique, et éventuellement de coder eux-mêmes un tel outil

Naturellement, ce cours est nécessaire pour les étudiants qui poursuivent ensuite des travaux de recherche dans le domaine de la vérification, de la démonstration automatique ou de la réécriture, mais aussi dans les domaines connexes comme la sécurité de systèmes informatiques, les systèmes embarqués, les preuves assistées et plus généralement l’utilisation de systèmes formels.

Afin d’atteindre ces objectifs nous proposons d’utiliser au moins une des logiques suivantes :

* La logique du premier ordre (très utilisée pour formalisation et preuve dans des domaines comme l’IA et les bases de données)
* La logique équationnelle (exp : spécification et la validation de programmes séquentiels)
* La logique intuitionniste (qui donne des preuves constructives et les techniques de base de la démonstration automatique).

**Semestre 1**

**Chapitre1 : Rappel mathématique**

* Notion de terme, formule, connecteurs… etc

**Chapitre2 : Logique de premier ordre**

* Syntaxe (Construction de terme et de formules, variables libres et liées …)
* Sémantique (Modèle d’une formule, notion de structures, satisfiabilité, validité, substitution, équivalence, …)
* Complétude de la logique du premier ordre
* Théorie logique
* Formes normale (prenex, skolem, clausale)
* Résolution et programmation logique

**Chapitre3 : Modèles de calcul**

* Machines de Turing (non déterministes, à plusieurs rubans, alternantes, ..)
* Les automates finis
* Les RAM (Random Access Machine)

**Semestre 2**

**Chapitre 1 : Calculabilité**

* Fonctions récursives
* Calculabilité récursive
* Systèmes récursifs
* Décidabilité

**Chapitre 2 : Décidabilité et incomplétudes**

* Arithmétique et fonctions représentables
* Codage et preuves
* Problèmes indécidables

**Chapitre 3 : Lambda Calcul**

* Notion de terme du lambda calcul
* Fonctions normales
* Fonction récursives
* Lambda calcul typé

**Chapitre 4 : Systèmes de types**

* Système de type simple
* Déduction naturelle et systèmes de types

**Chapitre 1 Introduction**

**(Logique propositionnelle & Logique des prédicats du premier ordre)**

**--Bilan--**

1. **Objectifs du chapitre**

Ce chapitre est consacré à la partie logique propositionnelle et prédicats du premier ordre, qui correspond à la pratique de la déduction sur des énoncés ne pouvant prendre que les valeurs « vrai » ou « faux ».

1. **Ce qu’il faut retenir**
* Pourquoi a-t-on besoin d’un langage artificiel pour étudier le raisonnement? (Langage formel)
* Garantir que deux raisonnements de même forme soit simultanément correctes ou simultanément incorrectes

**ON VOUDRAIT QUE L’EVALUATION DES RAISONNEMENTS SE FASSE D’UNE MANIERE MECANIQUE**

* Une logique formelle consiste en un ensemble de formules, généralement des mots, avec une notion syntaxique de preuves, généralement des suites de formules obéissant à des règles de déduction, et une notion sémantique de valeur, déterminée à l’aide de réalisations convenables.
* Les formules de la logique propositionnelle *L•* sont des assemblages de variables *Xi* à l’aide de connecteurs ¬, ∧, ∨, ⇒, ⇔.
* L’évaluation d’une formule de *L•* se fait inductivement à partir de l’affectation de valeurs 0/1 aux variables.
* Une formule est dite valide (*resp*. satisfaisable) si sa valeur est 1 pour toute affectation (*resp*. pour au moins une affectation).
* On dit que *H* se déduit par coupure à partir de *F* et *G* si *G* est la formule *F⇒H*. Une preuve par coupure est une suite de formules dont chacune est soit un axiome pris dans une liste fixée de 14 types de formules, soit obtenue par coupure à partir de formules antérieures de la liste.
* Le théorème de complétude affirme qu’une formule de *L•* est valide si et seulement si elle est prouvable.
* Le théorème de compacité affirme qu’un ensemble *T* de formules de *L•* est satisfaisable (c’est-`a-dire qu’il existe une affectation de valeurs 0/1 qui rende vraie chaque formule de *T*) si et seulement si tout sous-ensemble fini de *T* l’est.
* De nombreux problèmes peuvent être codés en un problème de satisfaisabilité (ou de validité) pour un ensemble de formules propositionnelles. Le théorème de Cook et Levin affirme le caractère NP-complet de l’ensemble SAT des formules satisfaisables.

**Définitions de base**

* Alphabet de la logique des prédicats est constitué de
* Ensemble dénombrable de symboles de prédicats à 0, 1, ou plusieurs arguments,
notés *p, q, r, ...,* homme, mortel, père, ...
* Ensemble dénombrable de variables d'objets (ou variables d'individu),
notées *x, y, z, x1, x2*, ...
* Ensemble dénombrable de fonctions à 0, 1, ou plusieurs arguments,
notées *f, g, ... ,* père-de, ...
* Quantificateurs *∀, ∃* (notations alternatives)
* Connecteurs False, ~, ∧, ∨, -> ainsi que les parenthèses de la logique propositionnelle

**Notation**.

* *Q* dénote un quantificateur quelconque, c-à-d ∀ ou ∃.
* Les fonctions à 0 arguments sont appellées constantes et sont notés sans parenthèses ( *a, b*, ..., Socrate, ...). Même chose pour les prédicats à 0 arguments, qui ne sont rien d'autre que des [variables propositionnelles](https://www.irit.fr/~Andreas.Herzig/C/prop.html#def:alphabet).

**Définition. *Terme***

L'ensemble des termes est le plus petit ensemble de mots construits sur l'alphabet de la logique des prédicats tel que

* Toute variable est un terme
* *f (t1,...,tn)* est un terme si f est une fonction à n arguments et *t1,...,tn* sont des termes

**Définition. *Formule*.**

* Si *p* est un prédicat à n arguments et *t1,...,tn* sont des termes alors *p(t1,...,tn)* est une formule atomique.
* L'ensemble FOR des formules (ou formules bien formées) de la logique des prédicats est alors défini de la même manière qu'en logique propositionnelle, en rajoutant une clause pour les quantificateurs :
* *(Q x A)* est une formule si *Q* est un quantificateur, x une variable et *A* une formule
* Une expression est un terme ou une formule.

**Définition (Validité, satisfiabilité, modèle)**

Soit*A*, une formule du calcul des prédicats sur une signature*( F, R)*.

* La formule est dite***valide***si sa valeur de vérité est vraie pour toute interprétation de la signature et tout environnement.
* La formule est dite***satisfiable***si sa valeur de vérité est vraie pour au moins une interprétation de la signature et un environnement. Une interprétation et un environnement qui rendent vraie la formule forment un***modèle***de la formule.
* La formule est dite***insatisfiable***si sa valeur de vérité est fausse pour toute interprétation de la signature et tout environnement.
* Ces définitions se généralisent à un ensemble de formules comme dans le cas propositionnel (interprétation comme une conjonction).

### Résolution

La résolution traite un ensemble de clauses et déduit à partir de cet ensemble de nouvelles clauses en utilisant la règle de résolution :

|  |  |  |  |
| --- | --- | --- | --- |
|

|  |
| --- |
| *C* ∨ *D*      *C*′ ∨ ¬ *D* |
|  |
| *C*∨ *C*′ |

 |

Dans le cas du premier ordre, les clauses peuvent contenir des variables et donc chaque symbole de prédicat peut apparaître plusieurs fois, en concernant des termes différents.

Par ailleurs ces termes peuvent contenir des variables et dans ce cas la clause représente l’ensemble de toutes les instances possibles de la formule.

On part d’un ensemble de clauses (disjonction de littéraux vue comme un ensemble de littéraux). Chaque clause *C* dont les variables libres sont *x*1,…,*xn* est interprétée comme la formule universellement quantifiée ∀ *x*1 … *xn*, *C* que l’on notera aussi ∀(*C*).

**L’unification** sur les termes s’étend de manière naturelle à l’unification sur les littéraux. Deux littéraux *L*1 et *L*2 s’unifient s’ils sont de la forme *P*(*t*1,…,*tn*) et *P*(*u*1,…,*un*) ou bien ¬ *P*(*t*1,…,*tn*) et ¬ *P*(*u*1,…,*un*) et que le problème (*ti*=? *ui*) admet une solution. Dans ce cas on note unif(*L*1,*L*2)l’unificateur principal.

#### Règles pour la résolution

Les transformations de clauses en calcul des prédicats se font en utilisant trois règles :

1. *Factorisation* : si σ est l’unificateur principal de *L*1 et *L*2 alors :

|  |  |  |  |
| --- | --- | --- | --- |
|

|  |
| --- |
| *L*1 ∨ *L*2 ∨ *C* |
|  |
| *L*1[σ]∨ *C*[σ] |

 |

1. *Renommage* : si σ est une bijection sur les variables alors :

|  |  |  |  |
| --- | --- | --- | --- |
|

|  |
| --- |
| *C* |
|  |
| *C*[σ] |

 |

1. *Résolution binaire* : si les clauses *L*1 ∨ *C* et ¬ *L*2 ∨ *C*′ n’ont pas de variables communes et si σ est l’unificateur principal de *L*1 et *L*2 alors :

|  |  |  |  |
| --- | --- | --- | --- |
|

|  |
| --- |
| *L*1 ∨ *C*      ¬ *L*2 ∨ *C*′ |
|  |
| *C*[σ]∨ *C*′[σ] |

 |

##### **Preuve par résolution**

Les définitions de preuves par résolution sont une simple généralisation du cas propositionnel. Soit *E* un ensemble de clauses:

* une déduction par résolution à partir de *E* est une suite de clauses *C*1,…,*Cn* telle que pour toute clause *Ci* dans cette suite
	+ soit *Ci*∈ *E*;
	+ soit il existe une clause *Cj* telle que *j*< *i* et telle que *Ci* soit le résultat de la règle de factorisation ou de renommage appliquée à *Cj*;
	+ soit il existe deux clauses *Cj* et *Ck* telles que *j*,*k*< *i* et telle que *Ci* soit le résultat de la règle de résolution appliquée à *Cj* et *Ck*.
* une réfutation de *E* est une déduction par résolution à partir de *E* qui contient la clause vide ⊥;
* une preuve par résolution (on dit aussi une preuve par réfutation) d’une formule quelconque *A* à partir d’un ensemble de clauses *E* est une preuve par réfutation de l’ensemble de clauses *E* ∪ *C*¬*A* avec *C*¬*A* une mise en forme clausale de la formule ¬ *A*.

La règle de renommage sert juste à préparer les clauses pour appliquer la règle de résolution qui nécessite que les deux clauses ne partagent pas de variables. La règle de factorisation permet de faire disparaitre plusieurs littéraux lors d’une étape de résolution binaire.

#### Correction de la résolution

La correction de la méthode de résolution établit que si l’on peut déduire une clause *C* à partir d’un ensemble de clauses *E* alors la formule ∀(*C*) est conséquence logique de l’ensemble des formules {∀ (*D*)|*D* ∈ *E*}

Une propriété clé est que si σ est une substitution alors ∀ (*C*) ⊨ ∀ (*C*[σ]) qui se déduit du lien entre substitution et vérité :

|  |
| --- |
| val(*e*,*C*[σ])=val({*x*↦ val(*e*,σ(*x*))},*C*) |

. Cette règle permet de ramener la correction de la résolution dans le cas du premier ordre à celle du cas propositionnel.

1. **Exercice avec solution (Examen proposé en 2013)**

*L : est un langage de prédicats définit comme suit :*

* *Alphabet x, y*
* *Constante 1*
* *Symbole b0 et b1*

On introduit les formules suivantes

*S0 : S(l, b0(l))*

*S1 : ∀x, S(b0(x), b1(x))*

*S2 : ∀x, S(x, y)⇒S(b1(x), b0(y))*

* 1. Soit une interprétation *I* dont le domaine *D* est formée des entiers naturels non nuls et dans laquelle les symboles sont interprétés de la manière suivante

 *II = 1*

*(b0)I (x) = 2 × x*

*(b1)I (x) = 2 × x + 1.*

Donner deux interprétations différentes de la relation S qui satisfont les trois formules S0, S1 et S2.

* 1. Pour chacun des couples de formules suivantes dire si elles sont unifiables et si oui donner l’unificateur principal :
		+ *1 : S(b1(x), y) ?= S(z, b0(x))*
		+ *2 : S(b1(x), y) ?= S(x, b0(z))*
		+ *3 : S(b1(x), l) ?= ¬S(b1(l), z)*
		+ *4 : S(b1(x), l) ?= S(b1(b0(y)), x)*
	2. En utilisant le système G, montrer le séquent

*S0, S1, S2├∀y, ∃x, S(x, b1(b0(y))).*

4. Mettre les formules S0, S1 et S2 en forme clausale.

5. En utilisant la méthode de résolution, montrer que la formule *∃x, S(x, b0(b1(l)))* est une conséquence logique des formules *S0, S1 et S2.*

6. Toto se demande s’il peut prouver la formule ∃x, S(x, l) par résolution `a partir de S0, S1 et S2. Il essaie d’utiliser la même méthode, que constate-t-il ?

7. La formule *(S0 ∧ S1 ∧ S2) ⇒ ∃x, S(x, l)* est-elle valide ? Est-elle satisfiable ?

**Correction**

* 1. On peut prendre *SI = {(x, y) ∈ N ∗ × N ∗ |y = x + 1}* en effet, dans cette interprétation les formules deviennent

*S0 : 2 × 1 = 1 + 1,*

*S1 : ∀x, 2x + 1 = 2x + 1,*

*S2 : ∀x y, y = x + 1⇒ (2 × y) = 2x + 1 + 1*

qui sont toutes vraies. Un autre modèle évident est si *S = N ∗ × N ∗* , c’est-à-dire que *S(x, y)* est vrai pour tous les entiers non nuls.

2.

 (a) unifiable *{y ← b0(x); z ← b1(x)}*

 (b) pas unifiable car il faudrait *x ?=* *b1(x)* qui n’a pas de solution

 (c) pas unifiable (formule et sa négation)

 (d) pas unifiable car il faudrait *x ?= l et x ?= b0(y)*

3. (∀D) (∃D(x = b0(b0(y)))) (∀G(S1, x = b1(y))) hyp S0, S1, S(b0(b0(y)), b1(b0(y))), S2 ` S(b0(b0(y)), b1(b0(y))) S0, S1, S2 ` S(b0(b0(y)), b1(b0(y))) S0, S1, S2 ` ∃x, S(x, b1(b0(y))) S0, S1, S2 ` ∀y, ∃x, S(x, b1(b0(y))) 4. C0 : S(l, b0(l)) C1 : S(b0(x), b1(x)) C2 : ¬S(x, y) ∨ S(b1(x), b0(y)) 5. soit la formule F def = ∃x, S(x, b0(b1(l))) – On met ¬F en forme clausale on a ¬F ≡ ∀x, ¬S(x, b0(b1(l))) et donc C3 : ¬S(z, b0(b1(l))) – On unifie C2 et C3, avec {y ← b1(l); z ← b1(x); x ← x 0} on obtient C4 : ¬S(x 0 , b1(l)) on unifie C4 avec C1 avec {x ← l; x 0 ← b0(l)} et on obtient la clause vide. 6. Lorsque l’on met la formule ¬∃x, S(x, l) en forme clausale, on obtient ¬S(x, l), cette clause ne s’unifie `a aucune des clauses positives de C0, C1, C2, il n’y a donc pas moyen de d´eriver la clause vide en utilisant C3. Comme on a vu que les formules S0, S1, S2 avaient un mod`ele, il n’y a pas non plus de mani`ere de ´eriver la clause vide `a partir de C0, C1 et C2. 7. La formule (S0 ∧S1 ∧S2)⇒ ∃x, S(x, l) n’est pas valide (sinon on aurait une contradiction dans l’´etape pr´ec´edente ou encore il y aurait un mod`ele de (S0 ∧ S1 ∧ S2) (celui du successeur) qui ne valide pas ∃x, S(x, l), par contre cette formule est satisfiable si on prend le mod`ele dans lequel S est vrai partout.

1. **Série des exercices proposés sans solution**

**Exercice 1.**

*P = {b, p}*

*b = « faire beau le matin »*

*p = « aller à la plage »*

1. Ecrivez les formules correspondant aux énoncés suivants :

a. S’il fait beau demain matin j’irai à la plage demain matin.

b. Je suis allé à la plage, donc il faisait beau ce matin

c. Je ne suis pas allé à la plage, donc il ne faisait pas beau ce matin

d. Il a plu, donc je ne suis pas allé à la plage

2. Quel(s) énoncé(s) *(b, c, d)* peut-on déduire de l’énoncé a.

**Exercices 2.**

Extrait simplifié du « livre qui rend fou » de R. Smullyan.

Dans une île, on trouve deux sortes de personnes les menteurs et les sincères.

Les menteurs mentent toujours ; et les sincères disent toujours la vérité.

P = {b, a, c, d, e}

b = « Bob est sincère »,

d = « Denis est sincère »

a = « Alice est sincère »,

c = « Chloé est sincère »

e = « Elise est sincère »

* 1. Ecrivez les formules correspondant aux trois énoncés suivants
		1. Alice dit « Bob dit toujours la vérité ».
		2. Bob dit « Alice est sincère ».
		3. Alice dit « Bob et moi nous sommes tous les deux sincères ou tous les deux menteurs ».
	2. Pouvez-vous déduire qu’Alice est sincère ?
	3. Ecrivez les formules correspondant à l’énoncé suivant
		1. Chloé dit « Je suis une menteuse ou Denis est sincère».
		2. Pouvez-vous déduite que Chloé est-elle une menteuse ? Que dire de Denis ?
	4. Ecrivez les formules correspondant à l’énoncé suivant
		1. Élise dit « Je suis une menteuse ».
		2. Qu’en pensez-vous ?
1. P = {f, g, h, i} f = « Franck est sincère », h = « Hugo est sincère » g = « Gabrielle est sincère » i = « Inès est sincère »
	1. Ecrivez les formules correspondant aux énoncés suivants
		1. Franck dit « Gabrielle et Hugo sont tous les deux sincères ou menteurs ».
		2. Gabrielle dit « Franck et Inès ne sont pas du même types : ils ne sont pas tous les deux menteurs ou tous les deux sincères ».
		3. Hugo dit « je suis un menteur ou Inès est une menteuse »
	2. 4. Qu’en pensez-vous ?

**Exercice 3.**

Les définitions suivantes portent sur les entiers naturels (positifs ou nuls) :

* l’ensemble des fonctions : *G = {zéro, succ, plus} avec rang(zéro) = 0, rang(succ) = 1 et rang(plus) = 2.*
* l’ensemble des prédicats : *P = {pair, premier, inf\_strict} avec rang(pair) = rang(premier) = 1 et rang(inf\_strict) = 2.*

• l’ensemble des variables *: X = { x, y, z }.*

1. Les expressions ci-dessous sont ou non des formules syntaxiquement correctes, des termes syntaxiquement corrects, ou non ? Justifiez en détail
2. Pour chaque formule syntaxiquement correcte, déterminez si elle est close et la liste de ses variables libres.

|  |  |
| --- | --- |
| *1* | *plus (x, plus(zéro, succ(zéro)))* |
| *2* | *succ(zéro)* |
| *3* | *((premier (x) ∨ inf\_strict(zéro,y)) ∧ ¬pair(plus(y,z)))* |
| *4* | *∀x ∀z (premier(zéro) ⇒ (inf\_strict(x, z) ∨ pair(plus(x, z))))* |
| *5* | *plus (plus ( plus(plus(zéro, x), y), plus(x, plus(y, z)))* |
| *6* | *pair(x, 0)* |
| *7* | *x* |
| *8* | *(∀x premier (y) ⇒ ∃y premier (x))* |
| *9* | *∃x ∀y ∃x ∀z inf\_strict (zéro, plus(x, plus(y, z)))* |

**Exercice 4.**

**Soit** *G = {vide, Ajouter, Retirer} ∪ {a, …, z}*

**avec** *rang(vide) = 0 ; rang(Ajouter) = 2 ; rang(Retirer) = 2 ; rang(a) = 0 = … = rang(z).*

**Soit** *P = {Estvide, EstLettre, Contient}*

**avec** *rang(EstVide) = 1 = rang(EstLettre), rang(Contient) = 2*

Soit *X = {x1, x2, y1, y2, z1}.*

1. Les expressions suivantes sont-elles des termes syntaxiquement corrects ou non ?
2. Les expressions suivantes sont-elles des formules syntaxiquement correctes ou non ? Justifiez en détail.
3. Pour chaque formule syntaxiquement correcte, déterminez si elle est close et la liste de ses variables libres.

*1. Ajouter(x1)*

*2. EstLettre(Ajouter(x1,y1))*

*3. ∀x1 Ajouter(x2,d)*

*4. ∀x1 EstVide(Ajouter(x2,d))*

*5. ∀x2 EstVide(Ajouter(x2,d))*

*6. Contient(z1)*

*7. EstLettre(Ajouter(Retirer(x1,a),y1))*

*8. ∃x1 Contient(Retirer(x1,d),d)*

*9. ∃x1 Ajouter(x1,d)*

*10. Ajouter(Ajouter(Ajouter(a,b),c),d)*

*11. Ajouter(Retirer(Ajouter(a,b),c),d) ∧ Retirer(Ajouter(v,e),l)*

**Exercice 5.**

***Soit*** *G2= {g0, g1, g2} avec rang(g0) = 0 ; rang(g1) = 1 ; rang(g2) = 2.*

***Soit*** *P2 = {p1, p2} avec rang(p1) = 2 = rang(p2).*

***Soit*** *X = {x1, x2, y1, y2, z1}.*

* 1. Définissez sur L2= P2 ∪ G2 une interprétation I2 dont la sémantique correspond aux entiers relatifs (positifs, nuls ou négatifs) qui permet d’ordonner et de comparer les entiers. Remarque : l'interprétation des prédicats dans I1 et I2 doivent être différents.
	2. Ecrivez un terme dont la valeur dans I2 est 1 et un autre dont la valeur dans I2 est -1.
	3. Ecrivez un terme dont la valeur dans I2 est 2 et un autre dont la valeur dans I2 est -2.

**Chapitre 2 Modèle des preuves**

* 1. **Résumé du cours**

**Définition (axiomatique à la Hilbert).**

Les schémas d'axiome de la logique des prédicats sont ceux de la logique propositionnelle, plus

*∀x A -> ((A){x\t})*

*(A){x\t} -> ∃x A*

(où *{x\t}* est une substitution quelconque), et les règles d'inférence sont le Modus Ponens :

*A A -> B*

*\_\_\_\_\_\_\_\_\_*

*B*

Plus les deux règles pour les quantificateurs

*A -> B*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*A -> (∀x B)*

S'il n'y a pas d'occurrence libre de *x* dans *A* et

*A -> B*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*(∃x A) -> B*

S'il n'y a pas d'occurrence libre de *x* dans *B*

**Note.**

Il est possible de définir également la déduction. Nous l'omettons ici, car elle est plus complexe que celle de la logique propositionnelle (les deux règles d'inférence pour les quantificateurs peuvent seulement être appliquées à des axiomes logiques, et non à des axiomes non-logiques et des formules déduites à partir de ceux-ci). De plus, cette notion peut être réduite à la validité, par un théorème de la déduction similaire à celui de la logique propositionnelle.

**Équivalences relatives aux quantificateurs**

Ce sont les principes de base pour pouvoir mettre en forme normale.

**Fermeture universelle :**

Soit *A* sans occurrence libre de *x*. Alors

*|- A ssi |- ∀x A*

**Fermeture existentielle :**

Soit A sans occurrence libre de x. Alors

*A est satisfiable ssi ∃x A est satisfiable.*

**Quantification répetée :**

*|- (Q1 x Q2 x A) <-> Q2 x A*

**Quantification sans variable libre :**

soit A sans occurrence libre de x. Alors

*|- (Q x A) <-> A*

*|- (A){x\t} <-> A*

**Renommage :**

*|- (Q x A) <-> Q y (A){x\y}*

**Les deux cas où la distribution est possible :**

*|- (∀x A) ∧ (∀x B) <-> ∀x (A ∧ B)*

*|- (∃x A) ∨ (∃x B) <-> ∃x (A ∨ B)*

**Elargissement de la portée des quantificateurs :**

Soit B sans occurrence libre de x. Alors

*|- (Q x A) ∧ B <-> Q x (A ∧ B)*

*|- (Q x A) ∨ B <-> Q x (A ∨ B)*

**Note**. Ces équivalences ne sont pas prouvables si x a des occurrences libres dans *B* :

*p.ex. (∃x p(x)) ∧ ~p(x) n'est pas équivalent à ∃x (p(x) ∧ ~p(x))*

**Définition (forme normale prénexe).**

Une formule *A* est en forme normale prénexe si elle est de la forme *Qx1 ... Qxn B , où B* est une formule sans quantificateurs.

La suite des quantificateurs est appelée préfixe, et *B* est appelée matrice.

**Exemple 1 : mise en forme normale prénexe**

Soit la formule *∃x ( p(x) -> ∀x p(x) ) :*

***Éliminer****-> :*

*∃x ( ~p(x) ∨ ∀x p(x) )*

***Renommer****∀x p(x) en ∀y p(y) :*

*∃x ( ~p(x) ∨ ∀y p(y) )*

***Sortir****∀y :*

*∃x ∀y ( ~p(x) ∨ p(y) )*

**Note**. Le renommage est essentiel ici ; p.ex. ∃x ( ~p(x) ∨ ∀x p(x) ) n'est pas équivalent à ∃x *∀x ( ~p(x) ∨ p(x) )*

**Exemple 2. mise en forme normale prénexe**

Soit la formule *~( p(x) -> ((∃y q(x,y)) ∧ ∃y r(y)) ) :*

**Éliminer** -> :

*~( ~p(x) ∨ ((∃y q(x,y)) ∧ ∃y r(y)) )*

**Rentrer** ~ :

*~~p(x) ∧ ~((∃y q(x,y)) ∧ ∃y r(y))*

*p(x) ∧ ~((∃y q(x,y)) ∧ ∃y r(y))*

*p(x) ∧ (~(∃y q(x,y)) ∨ ~∃y r(y))*

*p(x) ∧ ((∀y ~q(x,y)) ∨ ~∃y r(y))*

*p(x) ∧ ((∀y ~q(x,y)) ∨ ∀y ~r(y))*

**Renommer** *∀y ~r(y) en ∀z ~r(z) :*

*p(x) ∧ ((∀y ~q(x,y)) ∨ ∀z ~r(z))*

**Sortir** *∀y et ∀z :*

*p(x) ∧ ∀y ( ~q(x,y) ∨ ∀z ~r(z))*

*∀y ( p(x) ∧ ( ~q(x,y) ∨ ∀z ~r(z)) )*

*∀y ( p(x) ∧ ∀z ( ~q(x,y) ∨ ~r(z)) )*

*∀y ∀z ( p(x) ∧ ( ~q(x,y) ∨ ~r(z)) )*

**Remarque.**

Il est parfois possible d'être plus économique :

1. On peut économiser des quantificateurs en utilisant qu'[on peut éliminer les quantificateurs qui « ne servent à rien »](https://www.irit.fr/~Andreas.Herzig/C/pred.html#fact:elim) :
La formule *∀x ∃x (p(x) ∨ q(x))*
peut alors être simplifiée en : *∃x (p(x) ∨ q(x)).*
2. On peut économiser des variables (c-à-d éviter l'introduction de nouvelles variables) en utilisant [la distribution de ∀x sur ∧ et de ∃x sur ∨](https://www.irit.fr/~Andreas.Herzig/C/pred.html#fact:elim), (après avoir utilisé le renommage pour engendrer le même x partout) :
La formule *∀x ( p(x) ∧ ∀y p(y) )*
est alors d'abord renommée en *∀x ( p(x) ∧ ∀x p(x) )*
et ensuite mis en forme normale prénexe : *∀x ( p(x) ∧ ∀x p(x) ) .*

**Définition (Forme normale de Skolem)**

Une formule est en *forme normale de Skolem* si elle est en forme normale prénexe et ne contient pas de quantificateur existentiel.

**Algorithme de mise en forme normale de Skolem**

***Entrée* :** une formule *A*

***Sortie* :** une formule en forme normale de Skolem

***Début***

mettre *A* en forme normale prénexe ;

***pour tout*** quantificateur existentiel ∃x apparaissant dans A ***faire***

 - Appliquer la substitution *{x\f(x1,...,xn)}* à la matrice de A , où

 *x1,...,xn* sont les quantificateurs *universels* précédant *∃x* dans

 le préfixe de *A*

 - *f* est une nouvelle fonction qui n'a pas encore été utilisée ;

 supprimer ∃x du préfixe de *A*

 ***fin pour tout***

***fin***

*Remarque.* Si *n = 0* on substitue par une constante.

**Théorème.** Pour toute formule d'entrée *A*, l'algorithme de mise en forme normale de Skolem s'arrête. Il retourne une formule *A'* en forme normale de Skolem telle que *A* est satisfiable ssi *A'* est satisfiable.
(la démonstration utilise que toutes les étapes préservent la satisfiabilité ; notons que la mise en forme normale prénexe préserve même l'équivalence logique)

**Note.** Dualement, on obtient l'équivalence de validité si on remplace les variables quantifiées universellement par une fonction des variables quantifiées existentiellement.

**Exemple****s. mise en forme normale de Skolem**

* Soit la formule *∃x ∀y p(x,y) :*

remplacer x par la constante a : *∀y p(a,y)*

* Soit la formule *∀x ∃y p(x,y) :*

remplacer y par la fonction *f(x) :∀x p(x,f(x))*

* Soit la formule *∃u ∀x ∃y ∀z ∃t ( p(x) ∧ q(y) ∧ r(x,z,t) ∧ s(y) ∧ k(u) ) :*

remplacer u par la constante *a* : *∀x ∃y ∀z ∃t ( p(x) ∧ q(y) ∧ r(x,z,t) ∧ s(y) ∧ k(a) )*

remplacer y par la fonction *f(x) : ∀x ∀z ∃t ( p(f(x)) ∧ q(f(x)) ∧ r(f(x),z,t) ∧ s(y) ∧ k(a) )*

remplacer t par la fonction *g(x,z) : ∀x ∀z ( p(f(x)) ∧ q(f(x)) ∧ r(f(x),z,g(x,z)) ∧ s(y) ∧ k(a) )*

**Définition (forme normale clausale).**

Une formule est en *forme normale clausale* si elle est en forme normale de Skolem, fermée et sa matrice est en forme normale conjonctive propositionnelle.

**Algorithme de mise en forme normale clausale**

***Entrée* :** une formule *A*

***Sortie* :** une formule en forme normale clausale

***Début***

***Pour tout*** variable *x*apparaissant libre dans *A* ***faire***

 fermer *A* existentiellement : remplacer *A par ∃x A*

***fin pour tout****;*

 mettre *A* en forme normale de Skolem ;

 mettre la matrice de *A* en forme normale conjonctive

***fin***

**Notation***.*

* Comme toute variable est quantifiée universellement, nous pouvons éliminer le préfixe.
* Avec les mêmes définitions de littéral et clause qu'en logique propositionnelle nous pouvons appliquer la même convention notationnelle : une formule est représentée par un ensemble d'ensembles de littéraux.

**Exemples**

1. Soit la formule *∃x ( p(x) -> ∀x p(x) ) :*

mise en forme normale prénexe : *∃x ∀y ( ~p(x) ∨ p(y) )*

mise en forme normale de Skolem : *∀y ( ~p(a) ∨ p(y) )*

... qui est aussi sa forme normale clausale ; l'ensemble de clauses associé est :

*{ {~p(a)} , {p(y)} }*

1. Soit la formule *∀x ∃y ( (p(x) ∧ q(x,y)) ∨ r(z) ) :*

fermer existentiellement : *∃z ∀x ∃y ( (p(x) ∧ q(x,y)) ∨ r(z) )*

mettre en forme normale prénexe : la formule l'est déjà

mettre en forme normale de Skolem : *∀x ( (p(x) ∧ q(x,f(x))) ∨ r(a) )*

mettre la matrice en forme normale conjonctive : *∀x (p(x) ∨ r(a)) ∧ (q(x,f(x)) ∨ r(a))*

L'ensemble de clauses associé est *{ {p(x) , r(a)}, {q(x,f(x)) , r(a)} }*

1. **Série des exercices proposés**

 **Exercice avec solution (Examen 2014)**

On dira qu’une formule est en forme minimale si elle n’utilise que le connecteur ⇒, et comme formule atomique ⊥ et les variables propositionnelles.

1. Montrer que les formules *p⇒⊥ et ¬p* sont logiquement équivalentes.

2. Sachant que les formules *p ⇒ q et ¬p ∨ q* sont logiquement équivalentes, donner une

 formule équivalente à *p ∨ q* en forme minimale.

3. (a) Donner la table de vérité *de ((p⇒(q⇒⊥))⇒⊥*.

(b) Comparer cette table et celle de *p ∧ q.*

(c) En déduire une formule équivalente à *¬p ∧ q* qui soit en forme minimale.

 4. Déduire des questions précédentes une fonction *min* qui transforme toute formule

 propositionnelle *P* (sans oublier ┬) en une formule équivalente en forme minimale

 **Corrigé**

* 1. Si *p* est vrai alors *p ⇒ ⊥ et ¬p* sont faux, si *p* est faux alors *p ⇒ ⊥ et ¬p* sont vrais, les formules sont équivalentes.
	2. Une forme minimale pour *p ∨ q est (p⇒⊥)⇒q* (qui est équivalent à *¬p⇒q*)
	3. (a)

|  |  |  |  |
| --- | --- | --- | --- |
| *p* | *q* | *p⇒(q⇒⊥)* | *(p⇒(q⇒⊥))⇒⊥* |
| *F* | *-* | *V* | *F* |
| *V* | *F* | *V* | *F* |
| *V* | *V* | *F* | *V* |

 (b) c’est la même table que *p ∧ q*

 (c) *((p⇒⊥)⇒q⇒⊥)⇒⊥*, une autre forme possible est *(q⇒p)⇒⊥*

4.

*min(⊥) = ⊥*

*min(>) = ⊥⇒⊥*

*min(p) = p si p var. prop.*

*min(¬P) = min(P)⇒⊥*

*min(P ∨ Q) = (min(P)⇒⊥)⇒min(Q)*

*min(P ∧ Q) = (min(P)⇒(min(Q)⇒⊥))⇒⊥*

*min(P ⇒Q) = min(P)⇒min(Q)*

**Exercice 1 : Preuve par construction**

De nombreux théorèmes stipulent qu'un objet particulier existe.

Une technique pour prouver un tel théorème est de montrer comment construire cet objet.

Cette technique est appelée *preuve par construction*.

**Exemple**

Un graphe non orienté *G* est défini par le couple *(S, A), S* étant un ensemble de sommets,

*A ⊂ S2* un ensemble d'arrêtes. On appelle le degré d'un sommet le nombre d'arrêtes reliées à ce sommet. Un graphe est dit k-régulier si tous ses sommets sont de degré *k*.

**Théorème** : Pour tout entier n pair, *n > 2*, il existe un graphe 3-régulier composé de *n* sommets.

**Preuve** : Exercice.

**Exercice 2. Preuve par contradiction**

Cette technique consiste à supposer que le théorème est faux, puis à montrer que cette hypothèse conduit systématiquement à une conséquence fausse, appelée contradiction.

**Exemple**

Un nombre *x* est rationnel s'il existe deux entiers m et n tels que *x = m / n*. Par exemple,

*3 / 4* est un nombre rationnel.

Un nombre non rationnel est appelé un nombre irrationnel.

**Théorème** : *√2* est irrationnel.

**Preuve** : Exercice.

**Théorème** : Soit *R* une relation binaire définie sur un ensemble fini *A*, et soient *a* et *b* deux éléments de *A*. S’il existe un chemin de a vers b dans *R*, alors il existe un chemin de longueur au plus *|A|*.

**Preuve** : exercice.

**Exercice 3. Preuve par récurrence, ou induction**

Cette technique de preuve permet de montrer que tous les éléments d'une suite infinie vérifient une certaine propriété *P*. Elle comporte deux étapes :

- étape de base : on montre que *P(k)* est vraie ;

- pas d'induction : pour *i ≥ k*, on montre que si *P(i)* est vraie, alors *P(i+1)* est également vraie.

*P(n)* est alors vraie, pour tout *n ≥ k*, parce que *P(k)* est vraie (étape de base), donc *P(k+1)* est vraie (pas d'induction pour *i = k*), donc *P(k+2)* est vraie, etc.

**Exemple**

**Théorème** *: n4 – 4n2* est divisible par *3*.

**Preuve** : exercice.

**Exercice** **4**. Montrer que, dans la liste des axiomes pour *LΣ*, on peut remplacer les axiomes pour l’égalité par les formules *x = x et x = y⇒F(x)⇔F(y)* avec *x* et *y* sont libres pour *z* dans *F(z).*

**Chapitre 3 Modèle de calcul**

* 1. **Machine de Turing**

Alan Turing inventa cette machine abstraite en 1936 pour définir la notion de "fonction calculable".

* Toute tâche exécutée par une machine de Turing MT peut l’être sur un ordinateur ... et vice-versa !
* Peut simuler n’importe quel automate fini, automates à pile et même n’importe quel programme exécutable sur un ordinateur
* Il existe une MT universelle capable de simuler toutes les autres MT : programme comme donnée

**Définition.**

Une MT est composée d’une unité de contrôle, d’un ruban infini divisé en cases et d’une tête de lecture/ écriture.

****

En fonction de ce qu’elle lit sur la case courante et de son état courant (conformément à sa relation de transition δ), elle :

1. Ecrit un symbole sur sa case courante
2. Déplace la tête de lecture/ écriture d’une case (à droite, à gauche)
3. Change d’état.

Une Machine de Turing M est donnée par

*(Q, ∑, Γ, δ, q0, F) où :*

• *Q* est un ensemble fini d’états.

• *∑* est l’alphabet (fini) d’entrée

• *Γ* est l’alphabet (fini) du ruban, tel que *∑*  ⊆ *Γ*

• δ : Q × *Γ* → Q × *Γ* × {G,D, I} est la fonction de transition.

• *q0* est l’état initial.

• *F ⊆ Q* est l’ensemble des états accepteurs.

• On suppose qu’aucune transition n’est définie dans les états accepteurs.

• On considère un symbole B ∈ *Γ* \ *∑*  particulier dit blanc.

* 1. **Machine de Turing à plusieurs rubans**
* Une MT à *k* rubans, avec *k ≥ 1*, est composée d’une unité de contrôle, de *k* rubans infinis divisé en cases, de *k* têtes de lecture/ écriture.
* Chaque ruban a sa propre tête de lecture.
* En fonction de ce qu’elle lit sur les *k* cases courantes et de son état courant (conformément à sa relation de transition *δ*), elle :
1. écrit un symbole sur la case courante de chaque ruban
2. déplace chacune des têtes de lecture/ écriture d’une case (à droite, à gauche)
3. change d’état

*(Q, ∑, Γ, δ, q0, F) où :*

**Théorème de Rice**

Soit *∑ = {0, 1}.*

Une propriété non-triviale sur les machines de Turing est un sous-ensemble

*P ⊆ ∑∗* tel que :

1. Pout toutes machines *M* et *N* :

*L(M) = L(N) ⇒ [M ∈ P ⇔ N ∈ P].*

1. Il existe *M0, M1 : M0 ∈ P et M1* ∉ *P.*
2. **Haut niveau (Machines RAM)**

 Le modèle de la machine de Turing peut paraître extrêmement rudimentaire.

Il n’en demeure pas extrêmement puissant, et capable de capturer la notion de calculable en informatique.

 **Définition & Fonctionnement**

Un modèle de calcul qui ressemble beaucoup plus aux langages machine actuels, et à la façon dont fonctionnent les processeurs actuels.

Une machine RAM possède des registres qui contiennent des entiers naturels (nuls si pas encore initialisés).

Les instructions autorisées dépendent du processeur que l’on veut modéliser, mais elles incluent en général la possibilité de :

1. Copier le contenu d’un registre dans un autre ;
2. L’adressage indirect : récupérer/écrire le contenu d’un registre dont le numéro est donné par la valeur d’un autre registre ;
3. Effectuer des opérations élémentaires sur un ou des registres, par exemple additionner 1, soustraire 1 ou tester l’égalité à 0 ;
4. Effectuer d’autres opérations sur un ou des registres, par exemple l’addition, la soustraction, la multiplication, division, les décalages binaires, les opérations binaires bit à bit.
5. **Haut niveau (Machines RISC (Reduced Instruction Set))**

Une machine RISC est une machine RAM dont les instructions sont uniquement de la forme :

1. *x0  ← 0 ;*
2. *x0 ← x0 + 1 ;*
3. *x0 ← x0 1 ;*
4. ***if***  *x0 = 0* ***then aller à l’instruction numéro*** *j ;*
5. *x0 ← xi ;*
6. *xi ← x0 ;*
7. *x0 ← xxi ;*
8. *xx0 ← xi .*

**Définition**Théorème. Toute machine RISC peut être simulée par une machine de Turing.

(et réciproquement).

**Exercice (Démonstration du Théorème)**

1. **Exercice proposés**

**Exercice avec solution**

1. Construire une machine de Turing acceptant le langage *{ucu- | u ∈ {a, b}∗}*.

2. Construire une machine de Turing acceptant le langage *{u ∈ {a, b}∗ | |u|a = |u|b}.*

3. construire une machine de Turing calculant *n + 1* pour *n* donné en binaire sur le

 ruban d’entrée.

4. En utilisant ce qui précède, décrire une méthode qui permettrait de calculer

 *n+m* en binaire (*n et m donnés sur le ruban*).

**Solution**

****

Figure . Un mot, un séparateur (c), un mot miroir



Figure . Autant de a que de b

****

Figure 3. Successeur binaire

**Exercice 1.** On considère l’automate défini par le schéma suivant :

****

1. Construire une machine de Turing qui « mime » le fonctionnement de cet automate.
2. Généralisez votre réponse en indiquant un moyen de transformer tout automate déterministe fini en une machine de Turing.
3. Qu’en est-il des automates indéterministes ? Des automates à pile ?

**Exercice 2.**

1. Construisez une machine de Turing qui accepte *L = {anbncn | n ∈ N}.*

2. Quel est le langage reconnu par la machine suivante ?

****

**Exercice 3.**

1. Construisez une machine de Turing qui calcule la fonction *add* : ***N****2 →* ***N***

 telle que *add* (*m,n*) = *m + n*, *m* et *n*

Étant représentés en unaire.

2. Construisez une machine de Turing qui calcule la fonction *f :* ***N*** *→* ***N***

telle que *f(x) = x + 1,*

 *x* étant représenté en binaire.

3. Soit *Σ = {a, b}.* Construisez une machine de Turing calculant la fonction f : *Σ\* → Σ\** qui décale tout mot w d'une case vers la droite.

4. Construisez une machine de Turing calculant la fonction *f : Σ\* → Σ\** qui à tout mot w associe le mot miroir *wR.*

**Exercice 4.**

1. Construisez la machine de Turing *C* sous forme de combinaison de machines de Turing élémentaires qui effectue une copie :

*(q0, # σ1…σn) ├ C\* (qf, #σ1…σn # σ1…σn)*

1. Déroulez *C* sur l’exemple *(q0, #aba)*

**Exercice 5.**

1. Soit Σ = {a, b}.

a) Construire une machine de Turing qui prend un mot *u* sur son ruban d'entrée et écrit *uu* sur un ruban de sortie.

b) Construire une machine de Turing qui décide le langage L = {w ∈ Σ\* | w = wR}.

c) Construire une machine de Turing qui décide le langage L = {w ∈ Σ\* | |w|a = |w|b}.

2. Soit Σ = {0, 1}.

a) Construire une machine de Turing qui

- ne s'arrête pas si le mot en entrée contient le symbole 1,

- écrit en binaire, sur son ruban de sortie, le nombre de 0 du mot en entrée

avant de s'arrêter.

b) Construire une machine de Turing qui convertit en unaire un entier entré en binaire.

c) Construire une machine de Turing qui convertit en binaire un entier entré en unaire.

**Chapitre 4 Calculabilité & Décidabilité**

* 1. **Motivation**

Cette partie présente les principaux résultats de la calculabilité. En d’autres termes, l’étudiant doit chercher à comprendre la puissance des programmes informatiques.

On y prouve que certains problèmes peuvent se résoudre informatiquement et que certains problèmes ne peuvent pas l’être.

Tout d’abord il convient de comprendre que les problèmes auxquels nous allons nous intéresser ne sont pas vraiment des problèmes pour lesquels on ne connaît pas de solution, mais, et c’est en encore beaucoup plus fort, des problèmes pour lesquels on sait qu’il est impossible de produire une solution algorithmique.

Pourquoi s’intéresser à comprendre les problèmes qui ne peuvent pas être résolus ? Premièrement, parce que comprendre qu’un problème ne peut pas être résolu est utile. Cela signifie que le problème doit être simplifié ou modifié pour pouvoir être résolu. Deuxièmement, parce que tous ces résultats sont culturellement très intéressants et permettent de bien mettre en perspective la programmation, et les limites des dispositifs de calculs, ou de l’automatisation de certaines tâches, comme par exemple la vérification.

* 1. **Résumé du cours**
1. **Langages et problèmes décidables**
2. **Problèmes de décision**

**Définition**

Un problème de décision *P* est la donnée d’un ensemble *E*, que l’on appelle l’ensemble des instances, et d’un sous-ensemble *E+ de E*, que l’on appelle l’ensemble des instances positives

1. **Langages décidables**

On rappelle la notion de langage décidé, qui invite à introduire la notion de langage décidable.

**Définition****(Langage décidable)**

Un langage *L ⊂ M∗* est dit décidable s’il est décidé par une certaine machine de Turing.

Un langage ou un problème décidable est aussi dit récursif.

Un langage qui n’est pas décidable est dit indécidable.

1. **Problèmes semi-décidables**

Le problème HALTING − PROBLEM est toutefois semi-décidable dans ce sens

**Définition (Langage semi-décidable)**

Un langage *L ⊂ M∗* est dit semi-décidable s’il correspond à l’ensemble des mots acceptés par une machine de Turing *M*.

**Corollaire.** Le langage universel HALTING − PROBLEM est semi-décidable.

**Démonstration**

Pour savoir si on doit accepter une entrée qui correspond au codage <*M*> d’une machine de Turing *M* et au mot w, il suffit de simuler la machine de Turing *M* sur l’entrée *w*.

On arrête la simulation et on accepte si l’on détecte dans cette simulation que la machine de Turing *M* atteint un état accepteur. Sinon, on simule *M* pour toujours.

Un langage semi-décidable est aussi dit récursivement énumérable.

**Corollaire.** R ⊄ RE.

**Démonstration**

L’inclusion est par définition. Puisque HALTING − PROBLEM est dans *RE* et n’est pas dans *R*, l’inclusion est stricte.

1. **Exercices proposées**

**Exercice (Calculabilité)**

Monter que la division euclidienne est calculable.

**Exercice (Calculabilité)**

Montrer que la fonction *x → x2* est calculable.

**Exercice (Calculabilité)**

Montrer que si

*f : Nn → N*

*g : N2 → N*

sont calculables alors

*h : Nn+1 → N* avec *h(x1, ・ ・ ・ , xn+1) = g(f(x1, ・ ・ ・ , xn), xn+1)*

est calculable.

**Exercice (Décidabilité des classes de langages)**

1. Les langages rationnels sont-ils décidables ? Justifiez.

2. Les langages algébriques sont-ils décidables ? Justifiez.

**Exercice (Problèmes indécidables à propos des grammaires algébriques)**

1. Montrez que le problème suivant est indécidable :

 Soit G une grammaire algébrique. Est-ce que L(G) = Σ\* ?

2. Montrez que le problème suivant est indécidable :

 Soient G1 et G2 deux grammaires algébriques. Est-ce que L(G1) ∩L(G2) = ∅ ?

**Exercice (Problèmes indécidables à propos des machines de Turing)**

1. Soient M1 et M2 deux machines de Turing. Montrez que déterminer si M1 et M2 acceptent le même langage est un problème indécidable.
2. Soit L un langage accepté par une machine de Turing, et soit LR le langage constitué des mots de L vus dans un miroir.

Par exemple, L = {a, ab, abb, abbb}, LR = {a, ba, bba, bbba}.

Montrez que le problème qui consiste à montrer que L = LR est un problème indécidable.

**Aller plus loin**

**Exercice**

Soit *E ⊂ N* un ensemble récursivement énumérable, énuméré par une fonction calculable *f* strictement croissante.

Montrer que *E* est décidable.

**Exercice**

En déduire que tout sous-ensemble récursivement énumérable infini de *N* contient un ensemble décidable infini.

**Exercice**

Soit *E ⊂ N* un ensemble décidable.

Montrer qu’il est énuméré par une fonction calculable *f* strictement croissante.

**Exercice**

Soit *A ⊂ N 2* un ensemble décidable de couples d’entiers.

On note *∃A* pour la (première) projection de *A*, à savoir le sous-ensemble de *N* défini par

*∃A = {x|∃y ∈ N tel que (x, y) ∈ A}.*

**Chapitre 5 Récursivité**

* 1. **Résumé du cours**

Les définitions récursives sont omniprésentes en informatique.

Elles sont présentes à la fois dans les langages de programmation, mais aussi présentes dans de nombreux concepts que l’on manipule

Les termes *récursif*, *récursion*, *récursivité* ont la même racine latine que *récurrence* : *currere* = *courir*, et par là : *recurrere* = *courir en arrière* qui a donné également *recourir*, *recours* = *action de faire appel*(en justice).

* Le terme *récurrence* apparaît avec Poincaré dans *La Science et L'hypothèse* au tout début du 20è siècle. On parle dès lors de formule de récurrence, et de raisonnement par récurrence pour signifier le raisonnement par induction de Pascal.
* La *récursivité*n'apparaît dans le Larousse qu'en 1968 avec les progrès de l'informatique et des algorithmes récursifs : la notion actuelle de fonction (ou de procédure) *récursive* rencontrée en programmation se rattache à celle définie par Skolem (1923) puis Gödel (1926) sans en avoir exactement le même sens.

|  |
| --- |
| Les fonctions récursives primitives, récursives élémentaires et récursives générales : |

On suppose ici que l'ensemble ***N*** des entiers naturels est construit ou accepté dans son sens intuitif. Selon Kleene, les fonctions de ***N****p* dans ***N****, p ≥ 1*, établies comme effectivement calculables depuis les premiers travaux de Skolem et Gödel sur le sujet, dites *récursives primitives* (appellation du à Skolem) sont les suivantes :

1. La fonction successeur (incrémentation) Incr qui à tout entier naturel *x* associe l'entier immédiatement supérieur *x+*. C'est donc *x + 1*, mais tant que l'addition n'est pas définie...
2. Les fonctions constantes *Fc(x l, ..., xp) = c (entier)* dont la fonction nulle.
3. Les fonctions projections *Pi(xl, ..., xp) = xi*.
Pour deux variables : *P2(x1,x2) = x2*, pour une variable : *P1(x1) = x1*, c'est *l'application identique*.
4. Les fonctions obtenues par récurrence simple sur une des variables entières, par exemple x1 : on suppose g récursive primitive, alors f définie par l'algorithme ci-dessous l'est aussi :
    a. initialisation : *f(0, x 2, ..., xp) = k, k* donné dans ***N*** Raisonnement par induction
    b. hérédité (ou *induction*) : *f(n+1, x 2, ..., xp) = g(f(n, x2, ..., xp), x2, ..., xp)*
5. Les fonctions obtenues par composition (on dit aussi par *superposition*) : si f et g sont récursives primitives, alors la fonction composée h définie par

 *h(x l, ..., xp) = f(x l, ...,g(x l, ...,  xp),...,  xp)* l'est aussi.

1. Les fonctions obtenues par l'opération de minimum (dû à Kleene) : si *R(x l, ...,  xp)* entre éléments de ***N***, on désigne par *(µx)R(x l, ...,  xp),* *mu* comme minimum, le plus petit entier naturel x réalisant *R(x l, ...,  xp),* ce qui permet de construire de nouvelles fonctions.

La nature récursive de ces fonctions (au sens usuel de fonction se définissant en faisant appel à elle-même) n'apparaît peut-être pas clairement. Étudions-les à travers des exemples plus concrets, *fonctions récursives élémentaires* :

**La fonction prédécesseur *Dec* (décrémentation, antécédent)**

Si *x* est un entier naturel, son antécédent est *0* s'il est nul, sinon, c'est *x - 1*. Or, *(x+1)-1=x*. On peut donc la définir très simplement au moyen de la fonction successeur par :

Ce qui peut s'écrire *Dec(0) = 0* et *Dec(n+1)=n*  Il vient ainsi *Decr(1)=Decr(0+1) =0, Decr(2)=Decr(1+1)=1,* et ainsi de suite. L'algorithme correspondant procède par itération (boucle).

**La fonction addition *Add* :**

Additionner *y* à *x*, c'est incrémenter *x* en décrémentant *y* jusqu'à obtenir *0*.

On peut la définir par récurrence :

*Add(x,0) = P1(x)  (i.e. = x, condition d'arrêt)
Add(x,y) = Incr(Add(x,Decr(y))*

* Par exemple : *Add(2,3) = Incr(Add(2,2)) = Add(2,2) + 1 = [Incr(Add(2,1)) + 1] + 1
                                     = [[Add(2,0) + 1] + 1] + 1
                                     = [[2 + 1] + 1] + 1 = [3 + 1] + 1 = 4 + 1 = 5.*

Et on pose plus simplement : *Add(x,y) = x + y.*

**La fonction soustraction Sous :**

La soustraction dans ***N*** définie par : *S(x,y) = x - y si x ≥ y* et *0* sinon, peut être définie par :

        *Sous(x, 0) = x et Sous(x,y) = Decr(Sous(x,Decr(y))*

* Par exemple : *Sous(5,3) = Decr(Sous(5,2)) =  Decr(Decr(Sous(5,1)))
                                     = Decr(Decr(Decr(Sous(5,0)))) = Decr(Decr(Decr(5)))
                                     = Decr(Decr(4)) = Decr(3) = 2*

**La fonction multiplication *M* :**

*M(x,y) = x*×*y* Elle sera définie par :

*M(x,0) = N(x)* ( fonction nulle : condition d'arrêt)
*M(x,y) = A(M(x,D(y)),P1(x,y))* , c'est à dire *x(y - 1) + x* : *appel récursif*

**Exemple (Arbres ordonnés)**

Une alternative naturelle serait de présenter les arbres ordonnés par une définition récursive : un arbre ordonné est soit vide, soit réduit à un sommet (une racine), soit constitué d’un sommet (une racine) et une liste (ordonnée) d’arbres ordonnés (ses fils).

Dans ce chapitre, nous nous attardons sur les définitions inductives d’ensembles et de fonctions, qui permettent de donner un sens à des définitions récursives. Nous discutons, par ailleurs comment il est possible de faire des preuves sur des structures définies inductivement, en introduisant les preuves par induction structurelle.

**Raisonnement par récurrence sur N**

**Théorème** Soit *P(n)* un prédicat (une propriété) dépendant de l’entier *n*.

Si les deux conditions suivantes sont vérifiées :

*(B) P(0)* est vrai ;

*(I) P(n)* implique *P(n + 1)* pour tout *n* ;

Alors pour tout entier *n*, *P(n)* est vrai.

**Démonstration**:

Le raisonnement se fait par l’absurde. Considérons *X = {k ∈ N|P(k) est faux}.*

Si *X* est non vide, il admet un plus petit élément *n*.

D’après la condition *(B)*, n ≠ 0, et donc *n−1* est un entier, et *P(n − 1)* est vrai par définition de *X*. On obtient une contradiction avec la propriété *(I)* appliquée pour l’entier *n−1*.

Pour faire une preuve par récurrence, on établit donc une propriété en 0 (cas de base), et on établit que la propriété est héréditaire, ou inductive : *P(n)* implique *P(n+ 1)* pour tout *n*.

Le concept de preuve inductive généralise cette idée à d’autres ensembles que les entiers, à savoir aux ensembles qui se définissent inductivement.

Formellement, tout cela se justifie par le théorème qui suit.

**Définition**. (Définition inductive) Soit E un ensemble. Une définition inductive d’une partie *X* de *E* consiste à se donner

* un sous ensemble non vide *B* de *E* (appelé ensemble de base)
* et d’un ensemble de règles *R* : chaque règle *ri* *∈ R* est une fonction (qui peut être partielle) *ri* de Eni → E pour un certain entier *ni ≥ 1*

**Théorème** (Théorème du point fixe) *A* une définition inductive correspond un plus petit ensemble qui vérifie les propriétés suivantes

(B) il contient *B : B ⊂ X ;*

(I) il est stable par les règles de *R* : pour chaque règle *ri ∈ R*, pour tout

*x1, · · · , xni ∈ X, on a ri(x1, · · · , xni) ∈ X.*

On dit que cet ensemble est alors défini inductivement.

**Démonstration** Soit *F* l’ensemble des parties de *E* vérifiant *(B) et (I)*.

L’ensemble *F* est non vide car il contient au moins un élément : en effet, l’ensemble *E* vérifie les conditions *(B) et (I) et donc E ∈ F.*

On peut alors considérer *X* défini comme l’intersection de tous les éléments de *F*.

**Exercice proposés (Exemples d’applications)**

**Exercice 1.**

Montrez que les fonctions suivantes sont primitives récursives :

a) la fonction *factorielle*(*n*)

b) la fonction *pgcd*(m, *n*)

c) le prédicat *prem*(*n*) qui vaut *1* si *n* est un nombre premier, *0* sinon

**Exercice 2.**

Montrer que les fonctions suivantes sont récursives primitives.

1. *x → 0*
2. les fonctions constantes égales à *3* d'arité *0, 1 et 2 ;*
3. la fonction addition *: (x; y) → x + y ;*

4. la fonction multiplication : *(x; y) → x \* y ;*

5. la fonction exposant : *(x; y) → xy = x " y ;*

6. la fonction prédécesseur *Pred : x → (x – 1) si x > 0*

 *0* sinon ;

 7. la fonction différence tronquée *Diff : (x; y) → (x-y) si x ≥ y*

*0* sinon ;

**Exercice 3.**

Montrer que le prédicat x > y est récursif primitif.

**Exercice 4.**

1. Montrer que les constructeurs suivants sont récursifs primitifs (c'est à dire que s'ils sont utilises sur des fonctions récursives primitives du bon type alors les fonctions qu'ils construisent sont bien récursives primitives).

2. Pour *k ∈ N*, si *P* est un prédicat d'arite *k*, *f et g* sont des fonctions d'arite *k*, alors Si*(P; f; g)* est la fonction h d'arité *k* telle que lorsque *P* est vrai *h = f* et lorsque *P* est faux *h = g*.

3. Pour *k ∈ N*, si f est une fonction d'arite *k + 1* alors Somme(f) est la fonction h d'arité

*k + 1* telle que



4. Pour *k ∈ N*, si f est une fonction d'arité *k + 1* alors *Produit(f)* est la fonction h d'arité

*k + 1* telle que



**Exercice 5. (Fonctions μ-récursives)**

1. Montrez que la fonction logarithme est une fonction μ-récursive.
2. Utilisez l’opérateur de minimisation pour définir les fonctions :

****

1. On suppose connu le prédicat *prem*(*n*) qui vaut *i* si *n* est un nombre premier, *0* sinon.

Montrez que la fonction *nprem(n)* qui renvoie le *nème* nombre premier est une fonction μ-récursive.

**Chapitre 6 Lambda Calcul**

* 1. **Résumé du cours**

Pour l’informaticien, l’étude du *λ-*calcul (prononcer lambda-calcul) permet de comprendre sur un langage minimal des concepts théoriques qui pourront par la suite se reporter sur des langages de programmation plus riches. Par exemple, nous pourrons raisonner sur la terminaison des programmes, sur l’équivalence entre deux programmes, sur les relations entre la programmation et la logique

**Syntaxe**

On donne habituellement la syntaxe du *λ*-calcul en disant que c’est l’ensemble des expressions e formées de la façon suivante :

*e ::= λx. e|e1e2|x*

Pour une présentation un peu plus précise, on suppose l’existence d’un ensemble *V* infini de variables (que nous noterons *x, y, . . ., xi, yi, f, g)* et l’ensemble des *λ*-termes est constitué de la façon suivante :

* + Si *x* est une variable et *e* est un *λ*-terme déjà construit, alors *λx. e* est un *λ*-terme, nous appellerons un tel *λ*-terme une abstraction.
	+ Si *e1* et *e2* sont des *λ*-termes déjà construits alors *e1 e2* (la juxtaposition des termes) est un *λ*-terme, nous appellerons un tel *λ*-terme une application.
	+ Si *x* est une variable, alors c’est également un *λ*-terme.

Quelques exemples fréquemment rencontrés de *λ*-termes :

1. *λx. x (on l’appelle I),*
2. *λx. λ y. x (K),*
3. *λx. x x (∆),*
4. *λx. λy. λz.x z (y z)) (S),*
5. *λx. λy. x y*

**α-équivalence, variables libres et liées**

La relation d’*α*-équivalence est une relation d’équivalence et c’est aussi une relation de congruence : si *e* et *e’* sont α-équivalents alors *λx. e et λx. e’* le sont aussi, *e e1* et *e’* *e1* le sont aussi, et *e1 e* et *e1 e’* le sont aussi. Quelques exemples et contre-exemples d’*α*-*équivalence* :

1. *λx. λy. y, λy. λx. x,* et *λx. λx. x* sont *α*-équivalents.
2. *λx. x y* et *λy. y y* ne sont pas *α*-équivalents.

**Substitution**

On peut remplacer toutes les occurrences d’une variable liée par un *λ*-terme, mais il faut faire attention que cette opération soit stable par *α*-équivalence.

Plus précisément, nous noterons *e[e’/x]* le terme obtenu en remplaçant toutes les occurrences libres de *x* par *e’* dans *e*.

L’opération doit en fait se faire en deux temps :

1. d’abord construire un terme *α*-équivalent à *e* où aucune des abstractions n’utilise *x* ou l’une des variables libres de *e’*
2. ensuite remplacer toutes les occurrences de x par *e’*

Les variables libres de *e[e’/x]* sont alors les variables libres de e et les variables libres de *e’* .

On peut également décrire récursivement l’opération de substitution par les équations suivantes :

1. *x[e’/x] = e’,*
2. *y[e’/x] = y, si y ≠ x,*
3. *(e1 e2)[e’/x] = e1[e’/x] e2[e’/x],*
4. *(λx. e)[e’/x] = λx. e,*
5. *(λy. e)[e’/x] = λy.(e[e 0/x]), si y n’est pas libre dans e’,*
6. *(λy. e)[e’/x] = λz.((e[z/y])[e’/x]), si z n’apparait pas libre dans e et e’.*

La dernière équation s’applique aussi quand les deux précédentes s’appliquent, mais on appliquera celles-ci de préférence quand c’est possible.

Si l’on applique la dernière alors que les précédentes s’appliquent, on obtient simplement un terme *α*-équivalent

**Conversions** *α−conversion* :

*(λx.M) →α (λy.M[y/x]) y non libre dans M*

**Convention**

Nous ne distinguerons pas les termes qui sont identiques après (éventuellement plusieurs) *α* −conversion.

**Conversions *β−conversion*** :

*((λx.M)N) →β M[N/x]) si la substitution est sûre*

**Réduction**

Ces deux conversions sont étendues aux sous-termes :

*(λx.M) → (λx.M’) Si M →M’*

*(MN) → (M’N) Si M →M’*

*(MN) → (MN’) Si M →M’*

**Β−réduction**

*→\*B ∩n∈N →nB représente le calcul sur un λ−terme*

**Exemple**

* *((λx . λy. (xy))b)c*
* *((λx .(a( λy. (xy)))b)c*

**Forme normale Définition (Rédex)**

* Un rédex d’un terme est une sous-expression de la forme

*((λx.X)Y)*

* Un terme est en forme normale s’il ne contient aucun rédex.
* Un terme en forme normale ne peut plus être β−réduit (le calcul est terminé).

**Définition (Terme normalisable)**

* Un terme *E* est normalisable s’il existe un terme *E’* en forme normale telle que

E*→\*B E’*

1. *(λx. (x x)) (λx. (x x))* est-il normalisable ?
2. Donner un exemple de terme normalisable pour lequel il existe une séquence infinie de β−réduction.

**Stratégie de réduction** Il existe plusieurs façons de réduire un terme.

**Théorème (Church-Rosser)**

*Si E→\*B E1  et E→\*B E2, alors il existe E’ tel que E1→\*B E’ et E2→\*B E’.*

Ce théorème assure l’unicité de la forme normale (exercice).

Il existe une stratégie gagnante :

**Définition (Réduction normale)**

La réduction normale d’un terme est la stratégie qui applique la β-conversion aux rédex le plus à gauche.

**Théorème (Curry)**

Si *E* est normalisable vers une forme normale *N* alors la réduction normale de *E* mène aussi à *N*. Autrement dit, la réduction normale d’un terme normalisable termine toujours.

**Nombres naturels**

On peut représenter les nombres entiers positifs par les expressions de la forme

*λf x.f(f · · ·(f x)· · ·)*

Par exemple, *0* est le terme *λf x.x, 1* est le terme *λf x. f x, 2* est le terme *λf x. f(f x)*, et ainsi de suite.

On peut alors représenter les fonctions arithmétiques les plus basiques de la façon suivante :

* **Incrémentation**

*λx f z.x f(f z)*

* **Addition**

*λx y f z. x f(y f z)*

* **Multiplication**

*λx y f z.x (y f) z*

Dans la suite nous noterons *succ*, plus et *mult* ces fonctions. Dans la littérature ces nombres sont appelés entiers de Church.

**Exemple d’application**

1. Donner une représentation alternative de *incr*.
2. Ecrire les nombres *2* et *3*, construire la dérivation correspondant à *2 + 3*, reconnait-on bien le nombre *5* ?
3. Ecrire la dérivation correspondant à *2 × 3*.
4. Ecrire la fonction d’exponentiation.

**Série d’exercices proposés**

**Exercice avec solution**

1. réduire les termes suivants à des termes bêta-normaux

****

**Solution**

****

****

****

****

**Exercice 1.**

Le terme suivant est il syntaxiquement correct ? Si oui, décomposez-le en sous-termes en délimitant toutes les abstractions et applications.

* 𝑥𝑦 .𝜆𝑦.𝑥
* 𝜆𝑥.𝑥𝑦𝑦
* 𝜆𝑥.𝑦 𝑥𝑥
* 𝜆𝑧𝑥. 𝑥.𝜆𝑦 𝑧
* 𝜆𝑥.𝑥𝑦𝜆𝑡.𝑡𝑥
* 𝜆𝑥.𝑡𝑥

**Exercice 2.**

Soit *M* un terme du \_-calcul ; réduire autant que possible les expressions suivantes :

**add** *M* **0**, **add 0** *M*, **mul** *M* **0**, **mul 0** *M*, **mul** *M* **1**, **mul 1** *M*, et **exp** *M* **0**.

**Exercice 3.**

1. réduire (**true** *M* ) et ( **false** *M* ), où *M* désigne un terme quelconque ;

2. donner des définitions pour les opérateurs logiques **and**, **or** et **not**. *Conseil* : ces définitions

commencent par **if** .

**Exercice 4.**

Ecrire un terme qui code le test **iszero**, de telle sorte que **iszero 0 true** et **iszero n false** si **n** est un entier de Church différent de **0**

**Exercice 5.**

On code un *couple* par le terme **pair** = \_ *x y f . f x y*

Donner les définitions de **first** et **second** de telle sorte que, pour tous termes *M*, *N* on ait les réductions :

**first** ( **pair** *M N* ) *M* , **second** ( **pair** *M N* ) *N*.

**Exercice 6. (Examen 2015)**

Réduire et donner la forme normale des termes suivants :

1. ((λx.(x x)) (λz.z))

2. ((λxy.y) f g)

3. ((λxy.(x (x (x y)))) (λx.z))

**Chapitre 7 Complexité & NP Complet**

* 1. **Résumé du cours**

La complexité en espace correspond à l’espace mémoire (nombre d’octets) nécessaire à l’exécution sur une donnée de taille N.

La complexité en temps correspond au temps d’exécution (nombre d’opérations élémentaires) de l’algorithme sur une donnée de taille N.

* Le principe de la complexité est d’avoir une mesure pour évaluer les performances d’un algorithme (temps d’exécution ou espace mémoire)
* Le but est de comparer les algorithmes résolvant un même problème

## Ce qu’il faut retenir

A un problème donné, il existe plusieurs solutions algorithmiques.

Il faut les comparer par rapport à leurs complexités.

Il faut s’assurer que l’algorithme proposé est le meilleur possible pour le problème considéré.

En réalité, chaque problème a sa propre complexité (connue ou non). Dès qu’un algorithme à la même complexité, il est optimal.

Plus formellement : soient *A* un algorithme, d la donnée de cet algorithme, *Dn* l’ensemble de toutes les données possibles, de taille *n* :

*Coût espace  (A, d) (en octets)* et *coût temps (A, d)* (en nombre d’opérations élémentaires) de l’exécution de *A* sur *d*

Dans ce chapitre, On s’intéresse à trois types de coût :

* *Le coût minimum* : il correspond au meilleur des cas : *Minn(A) = mindЄDn {coût (A, d)}*
* *Le coût maximum* : il correspond au pire des cas : *Maxn(A)= maxdЄDn {coût (A, d)}*

*Le coût moyen* : *Moyn(A)=ΣdЄDn P(d)\*{coût (A, d)}* Ou *P(d)* est la probabilité d’apparition de la donnée *d*. Si les données sont équiprobables alors *P(d)=1/(Card[Dn])*

**Notations**

Soient deux fonctions *f* et *g N🡺N*

On dit que *f=O(g)* « *f* est dominée par *g* » ⬄ A partir d’un rang *n0 f(n) ≤ Cste.g(n)*

On dit que *f=Θ(g)* « f et g sont asymptotiquement équivalentes » ⬄ *f=O(g)* et *g=O(f)*

En pratique, on exprime donc les complexités sous la forme *Θ(g)* avec g très simple.

## Exemples

* *Θ(1)* : complexité constante (exemple : la somme des entiers de *1* à *n*)
* *Θ* *(log(n))*: logarithmique (exemple : recherche dichotomique)
* *Θ(n)* : complexité linéaire (exemple : recherche séquentielle)
* *Θ(n\*log(n))* : complexité sub-quadratique (exemple : tri rapide)
* *Θ(n²)* : complexité quadratique (exemple : tris simples)
* *Θ(nk)* : complexité polynomiale
* *Θ(2n)* : *c* (exemple : Tours de Hanoi)

On fera donc par exemple : *3n3+6n²+17n+3 🡺 Θ(n3)*

Influence de la complexité sur l’exécution : Prenons l’exemple d’une machine qui fait une opération élémentaire en 1µs (soit *106* opérations par seconde)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Taille de n \ Complexité* | *1* | *log(n)* | *n* | *n\*log(n)* | *n²* | *n3* | *2n* |
| *100* | *1µs* | *7 µs* | *100 µs* | *0.7 ms* | *10 ms* | *1 s* | *4.1016 années* |
| *10000* | *1 µs* | *14 µs* | *10 ms* | *0.14 s* | *1,5 min* | *11.5 jours* | *infini* |
| *106* | *1 µs* | *20 µs* | *1s* | *20s* | *11.5 jours* | *32.106 années* | *infini* |

# Détermination de la complexité

## Dans l’espace

C’est en général facile, surtout pour les données statiques : la taille est connue en fonction du type.

Pour les ordres de grandeurs, seuls les tableaux comptent.

La seule difficulté apparaît dans les allocations dynamiques (mémoire demandée en cours d’exécution) ou dans les algorithmes récursifs (il faut compter le nombre d’appels pour avoir une idée de la taille.

## Dans le temps

* Enchaînements séquentiels : L’ordre de grandeur de la séquence correspond à la valeur maximale.
* Séquences d’actions élémentaires : *Θ(1)*
* S’il y a un appel de fonction, il faut inclure la complexité de cette fonction dans la complexité de la séquence.
* Alternative (Si…alors…sinon) : On prend toujours la complexité maximale. Sinon on peut aussi faire une complexité moyenne mais il faut avoir des informations sur le résultat moyen de la condition.
* Boucles : Complexité = (Complexité du corps de boucle) *x* (Nombre de tours).
* Algorithme récursif : Complexité = (Complexité de la fonction) *x* (nombre d’appels).

**Structures de données et algorithmes en pratique**

La résolution de problèmes algorithmiques requiert presque toujours la combinaison de structures de données et d’algorithmes sophistiqués pour la gestion et la recherche dans ces structures.

D’autant plus vrai qu’on à traiter des volumes de données importants.

Quelques exemples de problèmes réels :

1. Routage dans les réseaux informatiques
2. Moteurs de recherche
3. Alignement de séquences ADN en bio-informatique

**Exemples : Les algorithmes de tris**

Les algorithmes de tris ont deux types d’opérations fondamentales : les comparaisons et les transferts.

On peut distinguer deux types de tris :

* Les tris internes : en mémoire centrale :
	+ Tri d’un tableau
	+ Tri d’un tableau sur lui-même
* Les tris externes : tri de fichiers

**Exercice proposé sur la Classes de complexité P et NP**

**Exercice 1**.

La méthode naïve pour chercher le plus petit diviseur *d* non trivial d'un entier *p* impair consiste à tester tous les entiers *d* impairs inférieurs à *p*1/2, jusqu'à trouver un diviseur (en cas d'échec, on en déduit que *p* est premier)

Exprimer la complexité de cet algorithme en utilisant la notation "grand-O", d'abord en fonction de *p*, ensuite en fonction de la *taille* de

**Exercice 2.**

Montrer qu'une expression *O ( ln n )* est aussi *O ( log2 n ),* et vice-versa.

Les notations *O ( en )* et *O ( 2n )* sont-elles équivalentes ?

**Exercice 3.**

1. Calculer la table de vérité de la formule *: Γ = ( x ∧ ¬y ) ∨ ( y∧ ¬z ) ∨ z*

2. Trouver une formule *£* de la forme *x*'*∧* *y*' *∧ z*', avec *x*' = *x* ou ¬*x*, etc. telle que *£ et Γ* soit une tautologie

3. Transformer ( *x∧* ¬*y* ) *∨* ( *y ∧*¬*z* ) en forme normale conjonctive

**Exercice 4.**

Transformer la formule ( x1 ∨ y1 ) ∧  ( x2 ∨ y2 ) ..∧..  ( xn ∨ yn ) en forme normale disjonctive

**Exercice 5.**

1. Exprimer l'égalité *y* = ¬*x* sous forme de conjonction de clauses

2. Même question pour *z* = *x* ∧ *y*

3. Même question pour *z* = *x* ∨ *y*

**Exercice 6.**

1. Exprimer l'implication *x ⇒ ( ¬y ∧ ¬z )* sous forme de conjonction de clauses.

2. En déduire une formule en forme normale conjonctive, qui exprime que l'une des trois variables *x*, *y*, *z* vaut *1*, et une seulement.

3. Généraliser au cas de *n* variables *x*1 , *x*2 , … , *xn* .

4. Prouver directement, c'est-à-dire sans utiliser la question 1, que la formule obtenue est correcte

**Exercice** 7.

Prouver que la classe P est close par union, concaténation et complément

**Exercice 8.**

Prouver que la classe NP est close par union et concaténation

**Exercice 9.**

Prouver que si NP n’est pas égal à son complément, alors P≠NP

**Exercice 10.**

Prouver que si P = NP alors tout langage *A ∈ P* sauf *A = ɸ* et *A =∑\** sont NP-complets

**Référence :**

1. [*Logique mathématique, tome 2 : Fonctions récursives, théorème de Gödel, théorie des ensembles, théorie des modèles*](http://www.amazon.fr/gp/product/2100054538/ref%3Das_li_qf_sp_asin_tl?ie=UTF8&tag=chronomath-21&linkCode=as2&camp=1642&creative=6746&creativeASIN=2100054538)*, René Cori, Daniel Lascar. Éd. Dunod, Paris, 2003.*
2. *Fonctions récursives primitives (définitions et résultats sous forme d'exercices corrigés) par  Bruno Salvy (Ecole Polytechnique) :*[*http://www.enseignement.polytechnique.fr/informatique/INF423/uploads/Main/pc4.pdf*](http://www.enseignement.polytechnique.fr/informatique/INF423/uploads/Main/pc4.pdf) *Corrigé :*[*http://www.enseignement.polytechnique.fr/informatique/INF423/uploads/Main/pc4-correction.pdf*](http://www.enseignement.polytechnique.fr/informatique/INF423/uploads/Main/pc4-correction.pdf)
3. *LTF Gamut [1992] : Logic, Language and Meaning, vol 1 : Introduction to logic, University of Chicago Press, Chicago, en particulier les chapitres 1 et 2, pp. 1-64. [Ouvrage remarquable par sa clarté didactique et par la richesse de ses exemples.*

*Les méthodes de preuves introduites (déduction naturelle) sont cependant différentes de celles présentées ici.]*

1. *R. Smullyan [1968] : First Order Logic, Dover Publications, New York, en particulier la première partie, pp. 3-40*
2. *R. Lassaigne et M. de Rougemont. Logique et fondements de l’informatique. Hermes, 1993.*
3. *M.-D. Popelard et D. Vernant. Éléments de logique. Mémo Seuil, 1998. [3] S. Russel et P. Norvig. Intelligence artificielle (2e édition). Pearson Education, 2006.*

**Index**

|  |  |
| --- | --- |
| *(Ag, r, Ad),* voir arbre binaire | *(V, E),* voir graphe |
| (q, u, v), voir configuration d’une machine de Turing | union de deux ensembles notation, voir ∪ |
| ., voir concaténation | Ac |
| , voir complémentaire | F(G/p), voir substitution |
| L(M), voir langage accepté par une machine | de Turing |
| ⇔, voir double implication | ⇒, voir définition inductive / différentes |
| notations d’une, voir implication | Σ, voir alphabet |
| Σ, voir ensemble des mots sur un alphabet | ∗ |
| R, voir décidable | ∩, voir intersection de deux ensembles |
| ∪, voir union de deux ensembles | , voir mot vide |
| ≡, voir équivalence entre formules, | voir équivalence entre problèmes |
| ∃, voir quantificateur | ∀, voir quantificateur |
| ≤m, voir réduction | |w|, voir longueur d’un mot |
| ≤, voir réduction | P(E), voir parties d’un ensemble |
| |=, voir conséquence sémantique | ¬, voir négation |
| |=, voir conséquence sémantique | ⊂, voir partie d’un ensemble |
| ×, voir produit cartésien de deux ensembles | **├**, voir démonstration, voir relation successeur entre configurations d’une machines de Turing |
| notation , voir HALTING − PROBLEM | ∧, voir conjonction |
| RE-complétude, voir RE-complet | ∨, voir disjonction |