RPA REPÈRES POUR AGIR
disciplines & compétences

Introduction à la science informatique

pour les enseignants de la discipline en lycée

dirigé par Gilles DOWEK

scérén
CNDP-CRDP

Une introduction à la science informatique

pour les enseignants de la discipline en lycée

Collection Repères pour agir
série Disciplines et Compétences

Coordination : Centre régional de documentation pédagogique de l'académie de Créteil
Directrice de collection : Christine Moulin, IA-IPR d'allemand
Responsable éditorial : Gilles Gony
Couverture : Dominique Florentin

Une introduction à la science informatique
pour les enseignants de la discipline en lycée

Édité par le Centre régional de documentation pédagogique de l'académie de Paris
Directrice : Marie-Christine Ferrandon
Chargé de mission Édition-TICE : Michel Bézard
Révision : Laure Técher
Édité avec le soutien de l'Association des sciences et techniques de l'information, et celui d'Enseignement public et informatique

Impression : Jouve
ISBN : 978-2-86631-188-9
ISNN : 1625-3000

Une introduction à la science informatique

pour les enseignants de la discipline en lycée

dirigé par Gilles Dowek

préface de Gérard Berry,
professeur au Collège de France

Auteurs

Préface : Gérard Berry
Professeur au Collège de France
Direction de l'ouvrage : Gilles Dowek
Directeur de recherche à l'Institut national de recherche en informatique et en automatique

Jean-Pierre Archambault
Chargé de mission au CNDP-CRDP Paris
Emmanuel Baccelli
Chargé de recherche à l'Institut national de recherche en informatique et en automatique
Sylvie Boldo
Chargé de recherche à l'Institut national de recherche en informatique et en automatique
Denis Bouhineau
Maître de conférences à l'université Joseph-Fourier, Grenoble
Patrick Cégielski
Professeur à l'université Paris-Est Créteil
Thomas Clausen
Maître de conférences à l'École polytechnique
Irène Guessarian
Professeur émérite à l'université Pierre-et-Marie-Curie, chercheur au Laboratoire d'informatique algorithmique : Fondements et Applications
Stéphane Lopès
Maître de conférences à l'université de Versailles Saint-Quentin
Laurent Mounier
Maître de conférences à l'université Joseph-Fourier, Grenoble
Benjamin Nguyen
Maître de conférences à l'université de Versailles Saint-Quentin
Franck Quessette
Maître de conférences à l'université de Versailles Saint-Quentin
Anne Rasse
Maître de conférences à l'université Joseph-Fourier, Grenoble
Brigitte Rozoy
Professeur à l'université de Paris-Sud
Claude Timsit
Professeur à l'université de Versailles Saint-Quentin
Thierry Viéville
Directeur de recherche à l'Institut national de recherche en informatique et en automatique
Jean-Marc Vincent
Maître de conférences à l'université Joseph-Fourier, Grenoble

Les auteurs remercient Nicolas Boullis, Julien Cervelle, Jean-Paul Delahaye, Ahmed Djebbar, Lena Domröse, Philippe Jacquet, Dominique Lacroix, Bernard Lang, Jérémie Lumbroso, Maryse Pelletier-Koskas et Tuong Vinh pour leur aide pendant la rédaction de cet ouvrage.

Ils souhaitent remercier également Gérard Berry, Robert Cabane, Pascal Guitton et Maurice Nivat sans qui cette entreprise n'aurait jamais été possible.

Sommaire

Préface

Gérard Berry
Professeur au Collège de France

C'est un grand plaisir que de préfacer cet ouvrage collectif, qui marque l'entrée de la science informatique en tant que discipline autonome dans l'enseignement secondaire. Depuis sa naissance dans les années 1950, et grâce à sa très grande flexibilité, l'informatique a successivement transformé tous les pans de la société : industrie, communication, commerce, culture, etc. Son essor s'accentue encore avec l'explosion d'Internet et la généralisation de l'informatisation des objets. Si elle reste souvent mystérieuse et quelquefois hostile pour les adultes, elle fait partie intégrante du monde des enfants d'aujourd'hui, au même titre que l'eau courante ou l'électricité. Il faut donc cesser de l'appeler une « nouvelle technologie » : si cette expression a encore un sens pour les adultes du XX^e^ siècle, elle n'en aura jamais pour les enfants du XXI^e^ siècle, ceux à qui l'enseignement secondaire s'adresse. Ils ne pourront comprendre l'avant-informatique que par les cours d'histoire.

Dans l'éducation secondaire française, l'informatique a longtemps été réduite à une technique auxiliaire dont il fallait simplement s'approprier l'usage. C'était négliger deux points fondamentaux. D'abord, son rôle de ferment d'une créativité extraordinaire, qui s'exprime dans de multiples applications non prévues encore récemment mais à l'essor foudroyant : diffusion à grande échelle de musique ou de films, moteurs de recherche permettant l'exploitation de masses gigantesques de données, réseaux sociaux créant des formes de communication qui s'affranchissent de l'espace et du temps, etc. Les résultats de cette créativité permanente modifient nos façons de faire les plus courantes à un point que nous ne réalisons peut-être pas encore complètement. Ils sont aussi associés à un essor non moins grand en nombre d'emplois qualifiés et intéressants dans le monde entier. Ensuite, le fait que cette créativité n'est pas seulement permise par l'avancée technologique. Elle a comme fondement les avancées constantes de la science informatique, qui est au cœur des machines, réseaux et logiciels modernes autant la science physique est au cœur des machines mécaniques, électriques ou électroniques. Après plus de soixante ans d'existence, cette science originale et féconde ne peut plus être appelée récente. Elle occupe des centaines de milliers

de chercheurs et d'enseignants dans le monde. Elle est assise sur un corpus théorique considérable mais aussi de nature profondément expérimentale. Elle conduit à de nouvelles connaissances et à de nouvelles façons de voir et de faire. Contrairement aux sciences de la nature, elle n'a pas comme préalable l'étude nécessairement lente et délicate de phénomènes complexes indépendants de notre volonté. Au contraire, elle construit ce qu'elle étudie avec très peu de barrières, ce qui explique sa vitesse d'expansion. De plus, les concepts et méthodes informatiques pénètrent maintenant toutes les sciences de la nature, même celles qui étaient traditionnellement peu touchées par les formalismes et méthodes mathématiques comme la biologie ou certaines sciences sociales, en leur fournissant des façons radicalement nouvelles de modéliser et de comprendre les phénomènes.

Il faut malheureusement constater que, malgré la qualité de sa recherche, notre pays n'est toujours pas un grand acteur de ce bouillonnement créateur, et se pose surtout en utilisateur et simple consommateur de technologies informatiques made in USA or Asia. *Pour ne pas continuer à accumuler le retard sur ces évolutions, il est indispensable que la science informatique et ses conséquences soient davantage connues dans tous les pans du tissu social, ce qui impose de lui donner sa place dans les cursus scolaires. C'est ce qui va heureusement recommencer en 2012 avec l'introduction de l'informatique en tant qu'enseignement de spécialité en terminale. Mais, avant de former les élèves, il faut former les professeurs. C'est l'objectif de ce livre, qui prend d'emblée un point de vue résolument moderne en abordant directement le cœur du sujet : l'informatique est fondée sur l'interaction et l'équilibre de quatre notions fondamentales, le codage numérique de l'information, les algorithmes ou procédés de calcul automatiques, les langages permettant de définir formellement les algorithmes, et les machines exécutant les programmes écrits dans ces langages. Ces notions sont bien sûr interdépendantes, mais elles sont suffisamment autonomes pour être décrites en chapitres bien articulés. Le livre s'intéresse aussi à de grands domaines d'applications conceptuellement formateurs et essentiels pour les applications en pratiques : l'architecture de machines, celle de réseaux et les bases de données. Les différents chapitres ne font pas que présenter brutalement les notions. Ils proposent des exercices, discutent des questions pédagogiques, et suggèrent des voies d'approfondissement.*

Ce travail sera certainement destiné à évoluer en fonction des retours des professeurs et des élèves, de l'évolution à venir des objectifs et programmes d'enseignement, et de celle du sujet lui-même. Et il sera bien sûr complété par d'autres contributions internes ou externes dans la myriade des formes proposées par les outils de documentation et communication modernes. L'informatique n'est-elle elle-même pas un merveilleux sujet pour développer des contenus créatifs sous toutes les formes, textuelles, audiovisuelles et interactives ?

Introduction

Depuis plus de quatre mille ans, nous connaissons des algorithmes qui permettent d'effectuer des opérations arithmétiques, calculer des intérêts composés, déterminer l'aire de surfaces agricoles, dériver des expressions fonctionnelles, résoudre des systèmes d'équations linéaires, fabriquer des tissus, préparer des aliments… Le mot « algorithme » lui-même dérive du nom du mathématicien al-Khwarizmi, auteur, au IXe siècle, d'un *Livre de l'addition et de la soustraction d'après le calcul indien*, qui systématise les opérations arithmétiques en numération décimale à position. Exécuter ces algorithmes ne demandant que d'effectuer, systématiquement et sans réfléchir, une suite d'opérations, l'idée est vite apparue de s'aider d'outils pour le faire. Ainsi, les baguettes à calculer, les bouliers, les échiquiers, les métiers à tisser, les machines mécanographiques… nous apparaissent rétrospectivement comme les précurseurs de nos ordinateurs.

Algorithmes et machines

Une révolution s'est cependant produite au milieu du XXe siècle, quand la technique des tubes à vide a permis de réaliser les premiers ordinateurs, c'est-à-dire les premières machines à calculer universelles : depuis longtemps, nous avions construit des machines capables d'exécuter plusieurs algorithmes, comme la machine de Pascal, qui effectuait des additions et des soustractions, la nouveauté avec les ordinateurs était qu'ils étaient capables d'exécuter, non seulement plusieurs algorithmes, mais tous les algorithmes possibles, opérant sur des données symboliques. C'est cette universalité des ordinateurs qui explique leur omniprésence dans nos vies professionnelles et personnelles : un comptable, un architecte et un médecin n'utilisent pas les mêmes algorithmes, mais ils utilisent tous un ordinateur pour les exécuter.

L'histoire de la naissance de l'informatique est donc celle de la rencontre de ces deux concepts d'algorithme et de machine. Un *algorithme* est une méthode opérationnelle qui permet de résoudre systématiquement toutes les instances d'un problème donné. Une *machine* est un système physique, avec lequel nous avons défini un protocole d'échange d'informations. Ces deux concepts sont très vastes : par exemple, une bille que nous laissons tomber dans le vide pen-

dant un temps donné et dont nous mesurons la distance parcourue est une machine qui calcule le carré d'un nombre – en choisissant judicieusement les unités de temps et de distance. Les machines ne se limitent donc pas aux ordinateurs, tels que celui qui se trouve dans notre cartable.

Langages et informations

Dans les toutes premières années de l'informatique, deux nouveaux concepts sont venus s'ajouter à ces deux concepts structurants d'algorithme et de machine. Tout d'abord est apparue la nécessité de décrire les algorithmes que nous souhaitons voir exécutés dans un *langage* compréhensible à la fois par la machine qui exécute l'algorithme et par l'être humain qui le décrit. C'est ainsi que sont apparus les premiers langages de programmation et, avec eux, les premiers programmes, incarnations d'un algorithme dans un langage particulier. Les langages de programmation ont une certaine parenté avec les langues naturelles que nous utilisons tous les jours ; on y distingue, par exemple, des *expressions* qui sont les homologues de nos groupes nominaux et des *instructions*, exprimées par un verbe à l'impératif, qui sont les homologues de nos phrases. Mais ils ont surtout de nombreux points communs avec d'autres langages formels, utilisés avant eux : la notation musicale, la notation algébrique, la nomenclature chimique, etc. Avec la naissance de l'informatique, nous avons commencé à nous poser quotidiennement des questions, rares jusqu'alors : quel langage inventer pour décrire tels ou tels objets ? Cette interrogation sur la notion de langage place l'informatique dans le prolongement de la linguistique, mais surtout de la logique.

Le second concept est celui d'*information*. Un ordinateur, nous l'avons vu, ne peut exécuter que des algorithmes opérant sur des données symboliques, c'est-à-dire des données qui, tels des nombres entiers ou des textes, peuvent s'exprimer par des suites finies de symboles, chiffres ou lettres, pris dans un ensemble fini. De telles données symboliques s'appellent des *informations*. Comme nous le verrons, l'ensemble de symboles choisis importe peu et toutes les informations peuvent s'exprimer avec deux symboles seulement, conventionnellement écrits 0 et 1.

Les images et les sons ne sont pas des données symboliques. Toutefois, ils peuvent être numérisés, c'est-à-dire représentés sous la forme de telles données. Bien entendu, nous perdons un peu de qualité en numérisant une image ou un son, mais parfois suffisamment peu pour que notre œil ou notre oreille ne s'en aperçoivent pas, *grosso modo* parce que ces organes, eux aussi, numérisent les images et les sons. Représenter les nombres, les textes, les images et les sons de manière uniforme, avec des 0 et des 1, permet de les transmettre

sur un réseau, de les conserver sur un disque, etc. en utilisant des méthodes uniformes, alors que, par le passé, nous utilisions des méthodes différentes – vinyles, pellicules, etc. – pour conserver, par exemple, les sons et les images. Avec la notion d'information apparaissent aussi une notion de quantité d'information et différentes méthodes de quantification – la quantité d'information au sens de Kolmogorov ou de Shannon –, selon que l'on s'intéresse à l'information contenue dans un message ou à l'information apportée par un message.

Quatre concepts indissociables

L'informatique nous apparaît donc comme structurée par les quatre concepts d'algorithme, de machine, de langage et d'information. Chacun de ces concepts est bien entendu antérieur à l'informatique : la nouveauté est dans l'articulation de ces concepts issus d'univers scientifiques et techniques si différents. Au début du XX^e^ siècle, les spécialistes de la notion d'algorithme étaient les mathématiciens, les comptables, etc., les spécialistes de la notion de machine étaient les ingénieurs qui les fabriquaient et les physiciens qui en étudiaient les principes – au sens où la thermodynamique étudie les principes de la machine à vapeur –, les spécialistes de la notion de langage étaient les linguistes, les logiciens, les traducteurs, etc. et les spécialistes de la notion d'information, les bibliothécaires, les imprimeurs, les agents du chiffre, etc. On aurait été alors très surpris d'apprendre que, quelques années plus tard, ces professionnels issus d'horizons si variés se mettraient à coopérer pour créer une science nouvelle et, surtout, que cette science atteindrait un tel degré de cohérence : pour concevoir un langage qui permet de décrire des algorithmes opérant sur des informations, afin qu'ils soient exécutés par une machine, il ne suffit pas d'être expert dans l'une de ces notions, il faut avoir une connaissance relativement approfondie de chacune d'elles.

Les évolutions récentes de l'informatique illustrent l'indissociabilité de ces quatre concepts. Les machines, nous l'avons vu, sont d'une grande diversité : un réseau qui relie des ordinateurs entre eux est une machine et une mémoire de masse qui permet de stocker de grandes quantités d'informations également. Ces deux types de machines ont permis l'apparition de nouveaux usages, qui constituent une évolution importante de l'informatique : pour nombre de leurs utilisateurs, les ordinateurs ne servent en effet pas à transformer des informations en leur appliquant un algorithme, mais plus simplement à transmettre ces informations d'un bout à l'autre d'un réseau et à les stocker pour les retrouver plus tard, c'est-à-dire à faire voyager des informations dans l'espace et dans le temps.

Toutefois, cette évolution ne signe pas pour autant la fin de la notion d'algorithme, car, pour transmettre de l'information sur les réseaux ou retrouver

de l'information dans une base de données, il faut inventer de nouveaux algorithmes. Ainsi, les algorithmes de routage, qui permettent d'orienter les informations qui voyagent sur les réseaux, et les algorithmes de notation de pages, à l'œuvre dans les moteurs de recherche, sont aujourd'hui parmi les algorithmes les plus étudiés. De même, pour interroger les bases de données, nous avons développé de nouveaux langages : les langages de requêtes. Ce sont donc encore les quatre mêmes concepts qui sont au centre de ces nouvelles questions de transmission et de stockage des informations.

On peut regretter qu'aucune langue n'ait réussi à exprimer ce caractère synthétique de l'informatique. Certaines langues, comme l'anglais, en font la science des machines, d'autres langues, comme le français, la science de l'information. Les unes comme les autres désignent le tout par la partie.

L'informatique et les autres sciences

L'apparition de l'informatique a quelque peu bouleversé notre vision de l'organisation des sciences. La classification des sciences qui a prévalu jusqu'au début du XX^{e} siècle opposait les mathématiques aux sciences de la nature. Sur le plan ontologique, les mathématiques cherchent à découvrir des vérités nécessaires, alors que les sciences de la nature cherchent à découvrir des vérités contingentes. Sur le plan méthodologique, les mathématiques, qui ne demandent aucune interaction avec la nature, sont une science *a priori*, alors que les sciences de la nature, fondées sur l'observation et l'expérimentation, sont des sciences *a posteriori*. L'informatique n'appartient à aucune de ces deux catégories, car elle est du côté des mathématiques sur le plan ontologique, mais du côté des sciences de la nature sur un plan méthodologique : le fait qu'un algorithme de tri transforme la suite 2, 8, 1, 5 en la suite 1, 2, 5, 8 est une vérité nécessaire, qui ne doit rien aux propriétés de la nature ; en revanche, pour l'établir, nous interagissons avec un ordinateur, qui est un objet de la nature.

Mais c'est surtout parce que les quatre concepts d'algorithme, de machine, de langage et d'information se sont propagés dans l'ensemble des sciences, que l'informatique a transformé le paysage scientifique. La notion d'information, par exemple, a permis aux physiciens de rétrospectivement mieux comprendre la notion d'entropie, qui se définit désormais comme l'information sur un système qui manque dans sa description macroscopique. Comme nous le verrons, c'est cette même notion d'information qui permet de définir la notion d'aléa.

Les notions d'algorithme et de langage, par ailleurs, se sont révélées de formidables outils de description d'objets concrets et abstraits, en particulier dans des situations, comme celle de la description des systèmes biologiques, où

le traditionnel langage des équations différentielles se révélait inopérant. On peut, par exemple, décrire la manière dont une cellule fabrique une protéine à partir d'un brin d'ARN par un algorithme qui associe un texte écrit dans un alphabet de vingt lettres – la protéine – à tout texte écrit dans un alphabet de quatre lettres – le brin d'ARN. De même, la logique définit aujourd'hui les démonstrations comme des algorithmes, par exemple une démonstration de la proposition $A \Rightarrow B$ comme un algorithme qui transforme les démonstrations de A en des démonstrations de B. Et si les scientifiques et les ingénieurs, jusqu'au début du XX^e^ siècle, ont décrit les grammaires ou les réseaux de chemin de fer en langue naturelle, ils inventent désormais chaque jour de nouveaux langages formels pour le faire de manière plus précise.

Enfin, s'il est devenu banal de s'émerveiller de la manière dont l'informatique a transformé l'instrumentation scientifique, il est peut-être moins fréquent de remarquer que les mathématiques étaient, jusqu'aux années 70, la seule science à ne pas utiliser d'instruments – à quelques rares exceptions près, comme la règle et le compas – jusqu'à ce que les ordinateurs les fassent entrer dans la phase instrumentée de leur histoire.

La complexité

Quand on démonte un réveil, une locomotive à vapeur ou un poste de radio, on se retrouve, en général, avec des engrenages, des tubes à fumée ou des composants électroniques, qui se comptent en dizaines ou en centaines.

Un programme informatique, en revanche, est souvent composé de plusieurs centaines de milliers d'instructions, voire de plusieurs millions pour certains. Le développement de programmes informatiques a donc constitué un saut radical dans la complexité des objets que nous sommes capables d'étudier ou de construire. Ces programmes ont permis en retour un saut similaire dans la complexité d'autres objets : grâce aux ordinateurs, nous sommes aujourd'hui capables de construire des circuits électroniques formés de plusieurs centaines de millions de transistors, des démonstrations mathématiques formées de plusieurs millions de pages ou des catalogues de plusieurs millions de molécules.

Une science et une technique

L'apparition de l'informatique nous mène aussi à revoir les relations qui existent entre les sciences et les techniques. Nous étions accoutumés à voir les techniques comme des applications des sciences, ce que traduit notre habitude de les appeler « sciences appliquées ». Cette notion d'application suppose que les sciences déterminent les techniques, lesquelles, en revanche, ne les influencent nullement. Depuis le XVII^e^ siècle, cette vision des relations entre

sciences et techniques est en contradiction avec les faits, ne serait-ce qu'à cause de l'importance des instruments dans la démarche expérimentale. Mais nous évacuons souvent cette objection en minimisant l'importance des instruments, que nous qualifions d'« outils », ce qui nous permet de rester cohérents avec nos valeurs qui, depuis l'Antiquité, placent le savoir au-dessus du faire.

Il est probable que l'apparition de l'informatique nous contraigne à abandonner cette hiérarchie. Ignorer que l'informatique étudie des objets – machines, programmes, etc. – mais en fabrique également donne une image complètement déformée de la discipline. Il est certes possible de distinguer une activité scientifique, qui vise à connaître, d'une activité technique, qui vise à fabriquer, mais il n'est plus possible pour le savant d'ignorer l'activité de l'ingénieur.

Comprendre cela est essentiel pour qui enseigne l'informatique. Enseigner l'informatique demande de faire sienne cette phrase attribuée à Confucius : « J'entends et j'oublie, je vois et je me souviens, je fais et je comprends. » Enseigner l'algorithmique sans enseigner aussi à écrire un programme ou comment on construit un ordinateur avec des transistors, c'est transmettre un savoir amputé.

Enseigner l'informatique

Comme celui de toutes les disciplines, l'enseignement de l'informatique a profondément évolué au cours de ces vingt dernières années, du fait de l'apparition de gigantesques ressources en ligne. Au XX^e^ siècle, on enseignait la représentation numérique des caractères en expliquant que le code ASCII de la lettre « A » était 65, que celui de la lettre « B » était 66, etc. On enseignait la programmation en expliquant que l'on testait la parité d'un nombre entier avec la fonction `odd` et que l'on calculait l'arc tangente d'un nombre réel avec la fonction `atn`.

Aujourd'hui, toutes ces informations factuelles se retrouvent sans peine sur le Web. Enseigner ne consiste plus à transmettre des atomes de connaissances, mais à les mettre en perspective et à montrer comment ils s'assemblent.

Un parcours

Notre parcours commence avec la notion d'information : nous montrons comment les objets les plus divers se numérisent et comment l'information se quantifie. Nous poursuivons au deuxième chapitre avec la notion de langage de programmation en insistant sur la difficulté, mais l'importance, de trouver des mots pour expliquer avec précision ce qu'il se passe quand on exécute un programme. Dans le troisième chapitre, nous présentons un certain nombre

d'algorithmes classiques qui peuvent s'exprimer dans un tel langage et nous insistons sur la multiplicité des algorithmes qui permettent de résoudre un problème donné. Au quatrième chapitre, nous expliquons enfin la manière dont un ordinateur se construit étape par étape à partir du transistor et la manière dont un programme écrit dans un langage évolué se traduit en langage machine, le langage natif de l'ordinateur, déterminé par son architecture. Nous poursuivons au cinquième chapitre avec la notion de réseau, en particulier le réseau Internet, et la manière dont des algorithmes permettent de transmettre de l'information d'un bout à l'autre de ces machines. Enfin, dans les sixième et septième chapitres, nous revenons sur la notion d'information, abordée du point de vue de sa protection et de sa structuration, puis des bases de données. Parce que le Web est une base de données et parce qu'il est l'interface de nombreuses autres bases de données, c'est aussi dans ce chapitre que le Web est abordé.

Bien entendu, de nombreux sujets ne sont pas abordés dans ce bref parcours. Si celui-ci donne envie au lecteur de poursuivre le voyage et de lire quelques-unes des références que nous donnons à la fin de chaque chapitre, nous aurons atteint notre but.

Représentation numérique de l'information

Notre parcours commence avec la notion d'information. L'information est la matière première de l'informatique : les algorithmes, les machines, les langages sont nés de notre désir de transformer, transmettre et stocker des informations. Nous commençons par détailler la manière dont les textes, les nombres, les sons, les images, etc. s'expriment comme des informations, c'est-à-dire comme des suites de bits *(*binary digits, *chiffres binaires), 0 ou 1, qui constituent les atomes d'information. Peu à peu apparaîtra l'idée essentielle de ce chapitre : comme beaucoup d'autres grandeurs, la quantité d'information se mesure quantitativement.*

Cours

Le codage numérique de l'information

Information, hasard et complexité Observons cette suite de chiffres :

14159265358979323846264338327950288419 71

Il y a sûrement plus d'informations dans cette suite-là que dans celle-ci :

00000000000000000000000000000000000000

qui n'est qu'une suite de zéros. Ah oui, mais en quoi la première suite de chiffres est-elle plus complexe ?

* Est-elle aléatoire, chaotique, bref sans ordre ni régularité ?
* Ou bien organisée, fortement structurée, riche en information « utile » ?

Nous pouvons porter peu ou beaucoup d'intérêt à ces symboles, leur attribuer une valeur ou pas, valeur marchande ou d'un autre ordre, trouver du sens ou non à leur enchaînement. Indépendamment de ces éléments externes, qu'entendons-nous lorsque nous parlons de son contenu en information ou de sa valeur en information ?

En fait, si nous regardons bien le début de cette suite, nous reconnaissons les premières décimales de π, peut-être le nombre le plus célèbre de toute l'histoire des mathématiques : autant dire que son information est très utile… du moins pour les géomètres et les mathématiciens !

En tout cas, il est bien compliqué de retenir les chiffres de la première suite. Il est sans doute impossible de faire autrement que de les apprendre par cœur. Mais pour la deuxième suite, il suffit de retenir : « voilà 40 zéros ».

Cet exemple est instructif, car il nous montre que nous devons bien distinguer :

* le contenu brut en information, qui ne dépend pas du sens ;
* et la valeur d'une information, qui elle dépend du but fixé.

Ce que les différents éléments de formalisation de l'information nous offrent, c'est une vue précise du contenu brut de cette notion quotidienne : nous supposons qu'il y a un contenu brut d'information pour chacun de ces objets, indépendamment des considérations précédentes.

Le bit, atome d'information Est proposé ici que ce contenu brut soit donné en bits. Pourquoi utiliser un codage binaire ?

Tout d'abord… parce qu'il faut au moins deux symboles ! Pourrions-nous utiliser moins de deux symboles ? Il semble que non, car si nous avons un seul symbole, par exemple 0, pour une longueur donnée, par exemple 100000, il n'y aurait qu'une seule suite : 000000000000000000000000… qui ne peut donc décrire qu'un seul objet ! Et non servir à les décrire tous… L'information commence donc avec la dualité – opposition et complémentarité – du 0 et du 1, du chaud et du froid, du bas et du haut… Quand on a un symbole unique, tout est uniforme. Et il n'y a pas d'information.

Ensuite, il s'avère qu'en utiliser deux est bien pratique ! Bien sûr, nous pourrions choisir d'utiliser plus de deux symboles, mais n'en utiliser que deux a plusieurs avantages. Le premier est que ce choix est indépendant de la nature des informations décrites. Il aurait été dommage de concevoir, par exemple, des ordinateurs grecs qui utilisent 24 symboles et des ordinateurs latins qui en utilisent 26.

Par ailleurs, la description de l'information avec deux symboles est plus simple à mettre en œuvre dans un ordinateur. Cela se réalise électriquement par un interrupteur ouvert ou fermé, un courant électrique présent ou absent, etc. Ce choix rend aussi les ordinateurs plus robustes : s'il y avait plus de deux valeurs, le système physique pourrait plus facilement les mélanger. Dès que les plages d'incertitude des valeurs se recouvriraient, par exemple avec la décharge de la batterie, des erreurs se créeraient. Cela minimise donc les erreurs d'interprétation en cas de perturbation du signal. C'est sans doute bien la qualité essentielle du code binaire que sa capacité à résister au bruit.

De plus, pour utiliser le moins d'énergie ou d'espace, il s'avère – mais ce n'est pas un résultat évident – que le codage binaire est le plus robuste.

Quand les informations s'ajoutent Il y a une idée encore plus profonde ici. Savoir de quelqu'un si c'est un homme ou une femme, un jeune ou un vieux, quelqu'un de grand ou petit, c'est très schématique, mais cela nous donne déjà trois atomes d'informations sur lui, trois bits. Les tailles en information de deux informations indépendantes s'ajoutent, mais pas celles de deux informations redondantes : par exemple, si nous ajoutons que cette personne est un être humain, nous ne gagnons rien, car, si c'est un homme ou une femme, c'est un être humain.

Ainsi, pour des informations indépendantes, l'information est additive. Il en découle que le nombre minimal de bits à utiliser pour coder une information est exactement la quantité brute d'information qu'elle contient.

Par exemple, la quantité d'information contenue dans une suite de symboles binaires peut se définir comme le nombre de cases mémoire que la chaîne occupe dans la mémoire d'un ordinateur, quand on ne lui fait subir aucun traitement particulier autre que la mise sous un format compatible avec le système numérique.

Autre exemple, la taille en information du choix entre plusieurs options correspond au nombre de bits pour le coder. Par exemple, choisir entre quatre couleurs : noir, rouge, vert, bleu nécessite deux bits. Choisir entre les sept couleurs que nous voyons aujourd'hui dans l'arc-en-ciel nécessite trois bits :

violet	indigo	bleu	vert	jaune	orange	rouge
'111'	'110'	'101'	'100'	'011'	'010'	'001'

tandis que le code « 000 » n'est pas utilisé ici, ce qui importe peu.

Deux théories complémentaires, la théorie *probabiliste* de l'information, due à Claude Shannon et en lien avec les concepts de la thermodynamique, et la théorie *algorithmique de l'information*, construite d'Andreï Kolmogorov – le même Andreï Nikolaïevitch Kolmogorov qui fonda la théorie moderne des probabilités – à Charles Bennett, permettent aujourd'hui de capturer et d'analyser formellement ces idées. Avec des conséquences que nous expliciterons.

L'information : une quantité universelle Mais l'idée fondamentale est que *toute information a un reflet numérique*. Avant l'ère numérique, la musique et le son étaient enregistrés sur des disques vinyles ou des bandes magnétiques, les séquences d'images sur des photos ou des films argentiques, les textes et les informations symboliques ou numériques imprimées sur papier, donc étaient sujets à des traitements physiquement différents, tandis que leur stockage, leur duplication, leur transmission étaient le fait d'actions distinctes.

Avec l'ère numérique, tout ce que nous considérons être une ou « de l'information », au sens « humain » du terme, a un *codage numérique*. Nous allons le détailler dans ce chapitre.

Cela concerne la quasi-totalité de nos informations sensorielles : la vue, avec les caméras et les écrans comme capteurs et effecteurs, le son, avec les microphones et écouteurs et haut-parleurs comme capteurs et effecteurs, le sens de l'équilibre, que nous oublions souvent de compter parmi les sens, mais que les systèmes de réalité virtuelle actifs intègrent en simulant des mouvements perçus par notre système vestibulaire, et partiellement le toucher, à travers des capteurs de pression qui permettent par exemple de « toucher » une image, et aussi l'odorat et le goût – des capteurs d'odeurs ou de goût, dits « nez électroniques » sont expérimentés –, même si ces sens restent encore moins bien compris.

Cela concerne aussi ce qui est au-delà des sens : on qualifie d'information humaine toute donnée pertinente que le système nerveux central est capable d'interpréter pour se construire une représentation de notre environnement et pour interagir correctement avec lui. On rejoint ici le sens étymologique de l'information : *ce qui prend du sens*, c'est-à-dire donne une forme à l'esprit. Le mot vient du verbe latin *informare*, qui signifie « se former une idée de ». L'information est donc immatérielle.

Mais, pour « exister », elle doit être consignée, directement ou pas, sur un support matériel qui prend alors la valeur de document. Le *codage numérique* est cette opération fondamentale par laquelle on consigne l'information pour pouvoir la manipuler.

Nous verrons alors que cette numérisation de l'information, qui entraîne sa discrétisation et sa représentation par des suites de bits, 0 ou 1, a pour conséquence de permettre de la traiter, de la transformer, de l'analyser à travers des algorithmes génériques insensibles à la nature et au contenu de l'information, car ne voyant que les bits 0 ou 1 qui la composent, ce qui rend leur usage universel. D'autres algorithmes spécifiques du contenu en information seront aussi détaillés.

Puis, selon le sens que nous donnons au mot *information*, il faut formaliser le concept différemment. Pour fixer ce concept, nous adopterons la procédure suivante : fixons un but, puis déterminons la valeur de l'information, relativement à ce but. Nous allons découvrir que, en précisant le but, on obtient différentes théories, dont la théorie probabiliste de l'information et la théorie algorithmique de l'information détaillées ici.

Le codage de l'information : un travail de standardisation et de pertinence

Deux dernières idées clés méritent d'être soulignées ici. Dans l'exemple ci-avant du codage des sept couleurs, le choix des bits est tel que l'ordre de l'arc-

en-ciel a été respecté. Ce serait pratique, si nous pensions que cet ordre a de l'importance. Mais ce n'était pas obligé.

Quel est le premier point clé ? *Convenir d'un standard pour toutes et tous.* Bien entendu, il faut que nous soyons tous d'accord pour coder les couleurs de la même façon ! Décider, comme nous l'avons fait dans l'exemple précédent, de choisir une correspondance « universelle » entre l'objet et son codage numérique. Le codage de l'information est avant tout une affaire de… convention.

Quel est le second point clé ? *Selon le codage choisi, l'information se manipule, selon les cas, plus ou moins facilement.* Si nous nous référons à l'arc-en-ciel, c'est parfait. Mais si nous sommes dans le cas où les couleurs primaires sont le rouge, vert et bleu, il était plus parlant d'attribuer un bit R, V, B pour chaque couleur primaire – rouge eût été 100, et jaune 011 comme combinaison de vert et bleu –, tandis que ce sont d'autres couleurs que nous eussions codées ici.

Le codage de l'information est en lien profond et étroit avec la manipulation de cette information.

C'est de ce travail de codage standard et, de fait, pertinent que nous allons rendre compte ici.

Codage numérique du texte

Un domaine où les conventions sont la base du décodage est le codage des lettres. Les premiers codages standardisés sont les codes du « Télégraphe de Chappe », au XVIII^e^ siècle, du « Morse » et du « Baudot », au XIX^e^ siècle. Pour ce qui est de l'informatique, c'est l'ASCII (*American Standard Code for Information Interchange*, code américain normalisé pour l'échange d'information) dont les premières versions datent du début des années 60. Le but est d'échanger du texte de façon portable entre deux machines. Le principe est assez simple et consiste à associer à chaque lettre ou chiffre un entier entre 0 et 127, donc représentable sur 7 bits. Ainsi, le chiffre « 0 » correspond à 48, la lettre « A » à 65 et la lettre « a » à 97. On peut représenter les chiffres, les lettres latines majuscules et minuscules et les principaux symboles de ponctuation.

De plus, ce codage doit être compris par tous : il doit être normalisé et connu afin qu'il n'y ait pas un codage des lettres pour chaque pays, qui nécessiterait une traduction lettre à lettre. Il existe des organismes de standardisation : l'ISO (*International Organization for Standardization*) est international et l'AFNOR (Association française de normalisation) est l'organisme français. Les standards sont utiles dans les domaines industriels et économiques, mais également pour l'interopérabilité dans le domaine informatique.

Pour représenter un texte anglais ou un programme informatique, l'ASCII suffit, mais, pour les autres langues, cela ne suffit plus. Bien plus tard ont été

définis des codages pour les autres langues. On peut citer l'ISO 8859-1 dans les années 1980, aussi appelé « *latin-1* », qui permet de représenter nos caractères accentués. La lettre est associée à une valeur entre 0 et 255, donc représentable sur 8 bits, et est compatible avec l'ASCII. Cela signifie que les nombres entre 0 et 127 correspondent aux mêmes lettres en latin-1 qu'en ASCII. D'autres codages ont été définis pour les autres langues : le chinois, le japonais… Ces codages sont compatibles avec l'ASCII mais ne le sont pas entre eux. En conséquence, comment écrire un texte multilingue, comme un dictionnaire ?

Une idée naturelle est de créer un unique codage pour tous les caractères existants. C'est l'idée d'*Unicode*. Initialement, les valeurs associées étaient entre 0 et 65535, mais cette plage, que l'on croyait suffisante, s'est révélée trop petite pour représenter l'ensemble des caractères de l'ensemble des langues. Ce codage est compatible avec le latin-1, et donc l'ASCII, mais souffre de défauts. Par exemple, le é peut se représenter soit par le caractère « é » du latin-1 soit par la suite de caractères « ’e » et cette redondance rend la comparaison difficile. D'autre part, le symbole physique pour l'ohm Ω n'est pas le même que le ω majuscule, même s'ils sont indiscernables à l'œil.

Pour représenter les caractères unicode se sont construits des codages plus ou moins compliqués tel que UTF-8 avec l'idée suivante : caractère → codage unicode → codage UTF-8. L'idée d'UTF-8 est d'être compatible avec l'ASCII, mais sans stocker exactement la valeur de codage en unicode. Les détails techniques se trouvent sur `http://fr.wikipedia.org/wiki/UTF-8` et sont assez élégants. Ainsi, si l'on récupère un flux d'octets correspondant à des caractères codés en UTF-8, on peut s'y repérer car on reconnaît facilement le premier octet correspondant à un caractère, même si ce caractère est composé de plusieurs octets, quatre au maximum.

Un texte n'est pas toujours suffisant. On veut parfois représenter une lettre, un article, un livre qui contiennent à la fois le texte et sa structure. Cela peut être de façon basique uniquement la mise en pages du texte. Tels mots sont centrés, en gras et de taille 14 comme dans les suites OpenOffice ou Microsoft Office. Ce livre a été écrit en LATEX où la structure du texte est l'élément important. Ensuite, chaque élément de structure – les items d'une liste, les titres de paragraphes – est affiché de la même façon, modifiable par l'utilisateur.

Codage numérique des nombres

Au contraire du codage des caractères, le codage des valeurs numériques n'est pas un code arbitraire standard, mais est issu de l'arithmétique. En effet, 2741 pourrait être la succession des caractères « 2 », « 7 », « 4 » et « 1 ». Mais ce codage ne permet pas d'effectuer facilement des calculs arithmétiques tels que

l'addition ou la multiplication. Pour obtenir une représentation utilisable dans des calculs, il suffit cependant de passer de la numération à base 10 classique à la numération à base 2 de manière à n'avoir que des bits.

Nous pouvons représenter des nombres entiers positifs par la suite de bits de leur représentation binaire. Ce codage est très précieux, car il permet de faire directement des opérations sur ces nombres. Par exemple, ajouter deux entiers positifs revient à ajouter leur représentation binaire : on obtient directement le codage binaire du résultat. De même pour toutes les autres opérations numériques, pour comparer ces entiers entre eux, etc. Souvenons-nous que toutes les valeurs numériques sont non seulement stockées en binaire, mais aussi manipulées en binaire dans les ordinateurs. À une valeur abstraite de nombre entier, on associe donc un ensemble de bits stockés dans la mémoire qui correspond à la présence ou l'absence d'une tension, comme nous le verrons au quatrième chapitre. On peut ensuite raisonner sur ces nombres, mais en interne il n'y a que des portes logiques réalisées par des circuits électroniques.

Nous allons utiliser une notation binaire, ce qui consiste ici à utiliser les chiffres 0 et 1 et à utiliser la base 2 comme base de numération. Cela signifie qu'un nombre binaire sera noté $a_n a_{n-1} \ldots a_0$ comme par exemple 100010101_2, l'indice 2 indiquant la base dans laquelle le nombre est représenté. Ce nombre a pour valeur l'entier $a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \ldots + a_1 \cdot 2 + a_0 = \sum^n a_i \cdot 2^i$.

Pour l'exemple $100010101_2 = 1 \cdot 2^8 + 0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 = 256 + 16 + 4 + 1 = 277$.

Il est difficile de considérer n arbitrairement grand. En général, on a $n = 32$, ce qui permet de représenter les entiers jusqu'à 4 milliards environ, ou $n = 64$, ce qui permet d'aller jusqu'à $1{,}8 \times 10^{20}$, qui sont les valeurs par défaut de stockage et de calcul des ordinateurs.

Comment passer du codage usuel décimal au codage binaire ? Pour écrire le nombre décimal n en binaire, on utilise l'algorithme suivant :

1. Si $n \,\%\, 2 = 0$ (reste de la division entière), écrire 0, sinon écrire 1, à gauche de ce qui est déjà écrit.
2. Remplacer n par $n \div 2$ (division entière).
3. Si $n = 0$, on a fini, sinon on retourne au point 1.

Considérons le déroulement de cet algorithme sur $n = 11$

n	11	5	2	1	0
$n \,\%\, 2$	1	1	0	1	
	1	11	011	1011	

Pour écrire le nombre binaire n_2 en décimal, on utilise l'algorithme suivant :

1. On pose $x = 1$ (cette variable contiendra, à chaque étape, la valeur 2^i) et $r = 0$ (cette variable accumulera le résultat).

2. Si $n_2 = 0$, rendre le résultat r.
3. Si n_2 finit par un 1, remplacer r par $r + x$.
4. Remplacer n_2 par $n_2 \div 2$ (ce qui revient à effacer son dernier chiffre).
5. Remplacer x par $2 \times x$.
6. Revenir au point 2.

Considérons le déroulement de cet algorithme sur $n_2 = 100011$.

n	100011	100011	10001	1000	100	10	1	0
r	0	1	3	3	3	3	35	35
x	1	2	4	8	16	32	64	

Comment calculer sur ces nombres ? De façon naturelle, comme nous en avons l'habitude, mais en base 2. Nous posons les opérations de façon identique.

Ainsi

	1	0	0	0	1	0	1	0	1	02	=	554
+				1	1	1	1	0	0	12	=	121
	1	0	1	0	1	0	0	0	1	12	=	675

Une opération est devenue triviale : la multiplication par 2. Elle consiste à ajouter un 0 à la droite de la représentation binaire du nombre : ainsi $277 \times 2 = 100010101_2 \times 2 = 1000101010_2 = 554$.

Entiers signés Considérons maintenant également des entiers négatifs. Une solution est que le premier bit soit considéré comme un bit de signe et le reste des bits comme la valeur absolue exprimée comme ci-avant. Mais cela a deux inconvénients : le premier est l'existence de deux zéros, l'un positif et l'autre négatif, dont il faudra gérer l'égalité. Le second inconvénient est le fait que l'on perde une valeur représentée : avec n bits, on ne peut représenter les valeurs que de $-2^{n-1} + 1$ à $2^{n-1} - 1$.

La solution choisie est proche et s'appelle le *complément à 2*. Il est nécessaire de fixer la valeur de n : la représentation de -1 dépend du nombre de bits choisi. Pour un nombre binaire en complément à 2 sur n bits, on sépare le premier bit s et les $n - 1$ bits suivants. Ces $n - 1$ bits représentent, en binaire usuel, une valeur positive v. La valeur de ce nombre est alors v si $s = 0$ et la valeur est $-2^{n-1} + v$ si $s = 1$.

Pour $n = 4$, cela donne les valeurs suivantes :

0000	0001	0010	...	0101	0110	0111	1000	1001	...	1101	1110	1111
0	1	2	...	5	6	7	- 8	- 7	...	- 3	- 2	- 1

Les valeurs possibles stockées vont de -2^{n-1} à $2^{n-1} - 1$: il n'y a pas de valeur perdue, ni de représentations multiples de 0. On garde également un discriminant facile du signe : les valeurs strictement négatives ont le premier bit à 1 et les valeurs positives ou nulles leur premier bit à 0, ce qui est très utile pour savoir si une opération sera une addition ou une soustraction entre les valeurs absolues des nombres.

Nombres réels Après les nombres entiers viennent naturellement les nombres réels ou du moins leur approximation dans les machines. L'ordinateur n'ayant qu'une mémoire finie, il ne peut stocker tous les chiffres du résultat exact d'un calcul comme $\sqrt{2}$ ou 1/3. Il a une représentation interne d'une partie des réels en « notation scientifique », ce qui signifie que l'on garde un exposant et un nombre fixé de chiffres. On appelle cela un nombre à virgule flottante. Ces nombres, ainsi que les calculs, sont régis par un standard, l'IEEE-754, qui définit la répartition et l'usage des bits d'un nombre. Les formats usuels sont la simple précision (`float` en Java ou en C) sur 32 bits et la double précision (`double` en Java ou en C) sur 64 bits. Des formats étendus, avec davantage de bits pour l'exposant et/ou la fraction, existent souvent, mais ne sont pas toujours accessibles au programmeur.

Exemple d'un nombre à virgule flottante simple précision :

11100011010010011110000111000000

1		11000110		10010011110000111000000
s		e		f
signe		exposant		fraction
↓		↓		↓
$(-1)^s$	×	2^{e-B}	×	$1 \bullet f$
$(-1)^1$	×	$2^{198-127}$	×	$1.10010011110000111000000_2$

$$-2^{54} \times 206727 \approx -3{,}7 \times 10^{21}$$

Pour les valeurs spéciales de l'exposant, minimale et maximale, on a des valeurs exceptionnelles : ±∞, nombres plus petits appelés dénormalisés, ±0 et NaN (*Not-a-Number*). Nous n'entrerons pas dans les détails ici, voir les explications et liens depuis `http://fr.wikipedia.org/wiki/IEEE-754` pour aller plus loin.

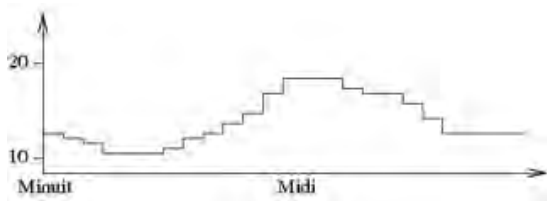
Avec ces nombres et ces valeurs, on peut effectuer des opérations. Chaque opération, addition, soustraction, multiplication, division, racine carrée, est parfaite : on a le même résultat que si l'on avait calculé avec une précision infinie et arrondi ensuite au format choisi. C'est une propriété très puissante qui permet de raisonner sur chaque étape de calcul et garantit une bonne précision

sur un calcul unique. En pratique, cependant, on fait de très nombreux calculs et il devient difficile de garantir la correction du résultat final. On peut résumer cela en disant que les ordinateurs calculent vite, mais faux.

Codage numérique des objets

Maintenant que nous disposons de la possibilité de coder des nombres, nous allons voir comment coder des objets numériques, c'est-à-dire créer un reflet numérique d'objets physiques, d'images, de sons, etc. dont l'information va être approximée par une suite de nombres. Tout le détail du codage des données usuelles de l'informatique est très bien décrit dans Wikipédia. Nous allons nous concentrer ici sur les points clés.

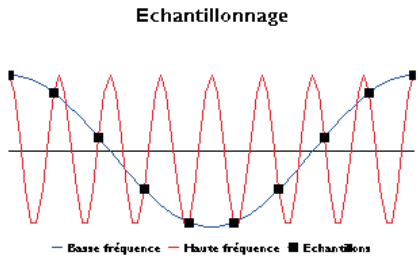
Coder une courbe ou un son Par exemple, comment coder la courbe de température de sa chambre au fil du jour ? On peut coder le nombre qui correspond à la mesure de température à chaque heure et les mettre bout à bout.



Ces 24 nombres vont donner une bonne approximation du chaud et froid, et du fait que… le chauffage est visiblement coupé la nuit… brrrr. Ou alors, c'est que la température extérieure est beaucoup descendue, et que l'isolation doit être revue.

Cette courbe n'est qu'une suite de codages binaires de nombres. Et le codage de chaque nombre a été discuté. Qu'en est-il du codage de la courbe elle-même ? Combien faut-il d'échantillons par seconde ? On comprend bien qu'une valeur à chaque heure suffit si le signal varie peu, mais non s'il varie beaucoup. Dans ce cas, quelle doit être la fréquence des relevés ? Le théorème de Nyquist-Shannon donne une solution définitive et simplissime : *il faut au moins deux points par oscillation du signal pour reproduire fidèlement sa dynamique lors du codage numérique*. Nous savons que chaque signal temporel contient des variations, lentes et rapides, de différentes fréquences que sa transformée de Fourier permet de calculer. Pour qu'un signal soit reproduit fidèlement lors de son échantillonnage – on dit aussi sa conversion analogique-numérique –, la fréquence d'échantillonnage doit être supérieure au double de la plus haute fréquence contenue dans le signal.

Si le codage ne respecte pas cette règle, il va y avoir un effet de crénelage, aussi appelé repli de spectre. Cela indique que l'on prend une sinusoïde pour une autre.



Il faut donc filtrer les hautes fréquences avant d'échantillonner un signal.

Par exemple, un son peut être représenté par une courbe qui correspond aux vibrations de la pression de l'air. C'est en effet le phénomène physique qui correspond au son de la voix, des bruits ou des instruments. Coder un son revient à coder une suite de valeurs numériques qui correspondent à cette courbe. Bien sûr, il faut beaucoup de valeurs : environ 44000 par seconde, pour obtenir une bonne approximation de toutes les vibrations sonores. L'oreille humaine entend les sons, au mieux, jusqu'à une fréquence aiguë de 22000 Hz (hertz, c'est-à-dire oscillations par seconde). Les notes de musique s'étalent d'environ 32 Hz pour le *do* le plus grave à 8000 Hz pour les notes les plus aiguës, tandis que des percussions génèrent des notes à des fréquences plus graves.

Il faut 16 bits pour coder correctement la valeur de l'intensité du son à chaque instant, donc à 0.0015 % près. Et, pour coder une minute de son, il faut : 16 bits × 40000 valeurs/seconde × 60 secondes = presque 40 millions de bits. C'est ce qui se passe dans les lecteurs MP3 par exemple, à une différence importante près. En effet, le codage de son avec le standard MP3 est « astucieux » : il met en œuvre des méthodes de compression qui tiennent compte de la perception humaine : elles ne reproduisent pas fidèlement le signal physique, mais les défauts sont inaudibles. Cette approximation supplémentaire économise de la place mémoire et du temps lors de la communication. Nous en parlerons plus loin.

Dans la mémoire d'un ordinateur, ce signal est stocké de manière générique par un tableau monodimensionnel de nombres.

Toutes les autres mesures physiques effectuées au cours du temps partagent la même problématique.

Coder une image Il est bien connu que l'on découpe une image en points élémentaires, ou « pixels » (*picture elements*), et attribue une couleur à chaque pixel, pour en faire un objet numérique. La couleur est représentée par un index. Une table de valeurs permet d'associer cet index aux valeurs numériques correspondant aux intensités lumineuses rouge, vert et bleu choisies : on parle de « pa-

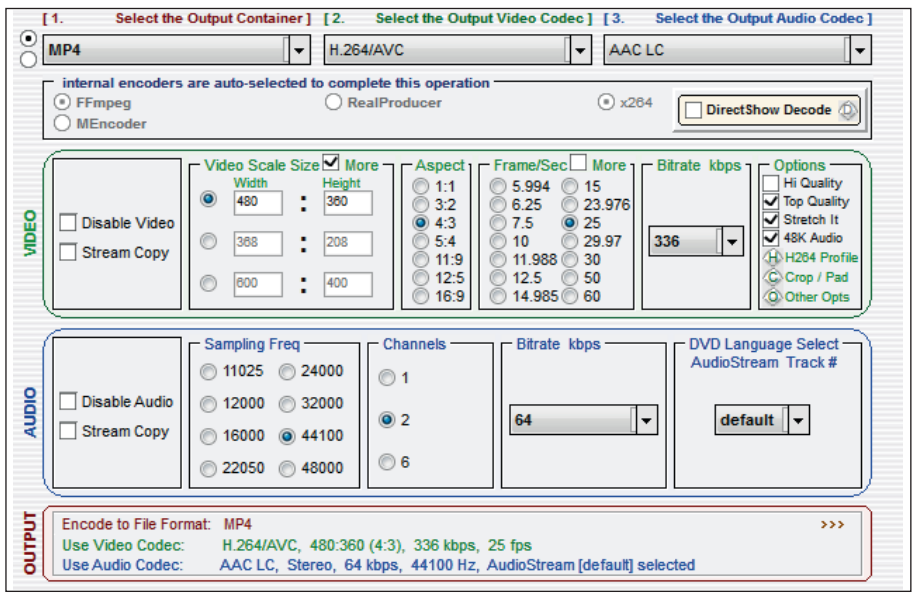
lette ». C'est une activité pédagogique fondamentale que nous proposons dans la partie didactique, que de travailler en détail sur ce codage qui reste bien intuitif.

Le codage le plus simple des couleurs est le codage « RGB » (Rouge-Vert-Bleu), sur trois canaux, la valeur codée entre 0 et 255, donc sur 8 bits, correspondant au taux de saturation en rouge, en vert, en bleu. Cela donne 3 × 8 bits par pixel. On peut stocker directement ce nombre, ou une « palette » de 256 couleurs qui permet de choisir parmi les 256^3 couleurs. Un quatrième nombre, dit *alpha*, peut aussi coder le pourcentage d'opacité, une valeur 0 rend le pixel invisible puisque complètement transparent et une valeur 255 le rend complètement opaque. On consultera les éléments de Wikipédia pour plus de détails et pour les détails de mise en œuvre.

Au-delà du format de stockage dans un fichier, les données de l'image, une fois décompressées, sont codées dans la mémoire de l'ordinateur par un tableau bidimensionnel de nombres correspondants aux pixels.

Coder une vidéo Une vidéo n'est jamais qu'une séquence d'images. Schématiquement, il suffit donc de coder les images et de mettre ces suites de bits bout à bout. Sans oublier de coder aussi le son. Là encore, il y a des astuces pour regrouper les parties qui se ressemblent et gagner de la place et du temps. Mais le principe reste le même.

La complexité vient du fait que le nombre de paramètres devient assez important, comme le montre ce panneau de paramètres du logiciel `Super:`



On doit choisir :
– le format conteneur qui contient des flux audio et vidéo respectant une quelconque norme, ici MP4 ;
– les codecs vidéo et audio qui décrivent les procédés capables de compresser ou décompresser les données dans un format normalisé ;
– l'encodeur qui est le nom de la bibliothèque logicielle qui peut effectuer ces opérations.
– Pour la vidéo, on doit encore choisir parmi les tailles d'images disponibles en pixels ; le ratio d'aspect de l'image réelle, qui est différent car les pixels ne sont pas forcément carrés, le nombre d'images par seconde, et le débit en nombre de kilo-bits par seconde, ce qui détermine le taux de compression, un débit faible conduisant à des images compressées de qualité réduite, tandis que d'autres options sont regroupées pour être sélectionnées globalement.
– Pour l'audio, on doit encore choisir la fréquence d'échantillonnage, le nombre de canaux, le son monophonique ou stéréophonique… et le débit.

Dans la mémoire de l'ordinateur les flux audio et vidéo sont séparés, ce dernier stocké de manière générique sous forme de liste d'images.

Qu'avons-nous découvert finalement ici ? La polyvalence des ordinateurs et des autres machines numériques : ils traitent des données très diverses mais toujours décrites de la même manière par des suites binaires, utilisant deux symboles.

Codage symbolique des valeurs

Au paragraphe précédent, nous avons vu comment approximer un objet numérique, coder un son ou une image. Coder un son, certes, mais pas une partition de musique. Une image, mais pas un dessin. Discutons la différence : une partition de musique ne représente pas tous les sons possibles – il est, par exemple, impossible de représenter le bruit d'un camion en duo avec le barrissement d'un éléphant. Elle se situe à un autre niveau de représentation et ne code que certains sons : les airs de musique. La notation musicale permet de définir une suite de symboles qui code la hauteur, aiguë ou grave, la durée, l'intensité, le timbre – le choix de l'instrument… Ensuite, en mettant bout à bout ces codes, nous obtenons la suite des notes. On code ainsi la partition musicale et son interprétation. Un tel codage existe, il est standard et se nomme MIDI (*Musical Instrument Digital Interface*). C'est une représentation *symbolique* de l'information.

Au-delà de cet exemple, détaillons cette façon de coder l'information.

Chaînes de caractères et noms de paramètres Si nous voulons "ah oui, relire la phrase lentement va aider à piger" coder une chaîne de caractères (`String`)

quelconque avec des mots quelconques, nous avons besoin de délimiter le début et la fin de la chaîne de caractère. Dans l'exemple qui s'est glissé dans la phrase précédente : `"ah oui, relire la phrase lentement va aider à piger"` est la chaîne de caractères à délimiter. Le premier guillemet délimite le début de la chaîne et le second en délimite la fin, tandis que chaque caractère est à coder en UTF-8, comme discuté précédemment.

On pourrait noter cette spécification de la manière suivante :

```
String:"[^"]*"
```

que nous ne manquerons pas d'expliquer dans quelques lignes.

Et si quelqu'un souhaite utiliser des guillemets, les caractères « » sont à sa disposition.

Définir un identificateur S'il s'agit maintenant de nommer un paramètre, d'identifier une variable, comme « *x* » en mathématiques, ou de référencer la propriété d'un objet, il ne faut utiliser qu'un sous-ensemble précis de caractères. Typiquement, un identificateur est une chaîne de caractères qui doit commencer par une lettre, ne contenir que des lettres ou des chiffres ou un caractère de séparation, par exemple « _ », tel que `mon_fichier, ou question_2,` etc. sans aucun caractère bizarre de ponctuation.

Selon le contexte, il vaut mieux en plus (1) ne pas utiliser d'accent, par exemple, dans les programmes, (2) ne pas mettre d'espace, pour éviter de laisser croire qu'il y a deux identificateurs `mon` et `fichier` au lieu de `mon_fichier`

Les majuscules sont à utiliser comme en français : nous les avons négligées ici pour ne pas alourdir le propos.

Lorsque, sur un site web, nous entrons notre « login » – le nom qui nous identifie –, selon les systèmes nous n'avons droit qu'à certains caractères, un peu comme ici. Pour choisir notre adresse électronique qui nous permet de recevoir et envoyer des courriels, nous avons le droit aux lettres non accentuées et aux chiffres, et au « . », « - » et « _ » comme séparateur – en fait, il nous est recommandé de toujours la définir explicitement sous la forme *prenom.nom@..*, comme par exemple `jean-sebastien.bach@somewhere.de` –, tandis que minuscule/majuscule ne sont pas prises en compte : tout est mis en minuscule. Il y a bien sûr une standardisation officielle, la RFC3696, mais de nombreuses applications n'acceptent pas l'ensemble des adresses valides ; il vaut donc mieux s'en tenir à un usage minimal.

Préciser de quel type est une valeur *La notion d'expression régulière*

On pourrait noter cette spécification de la manière suivante :

```
identifier: [a-z][a-z0-9_]*
```

qui se lit de la manière suivante :

`[a-z]` est une construction qui signifie : un caractère de `a` à `z`.

`[a-z0-9_]` est une construction qui signifie : un caractère de `a` à `z` ou de `0` à `9` ou le trait d'union _
`S*`, où `S` est une construction, signifie : `S` répétée 0 fois ou plus.

Donc, ici, `[a-z0-9_]*` signifie : une lettre, chiffre ou séparateur, répété 0 fois ou plus.

Nous découvrons là une façon formelle de décrire de quel type de suite de caractères il s'agit.

Par exemple, sachant que `[^"]` signifie : tous les caractères sauf `"`, nous voyons alors que `"[^"]*"` signifie : « le caractère ", suivi de tous les caractères sauf " 0 fois ou plus, c'est le sens de `[^"]*`, suivi du caractère " » : nous décrivons bien la `String` introduite précédemment.

Quel est l'intérêt de décrire *formellement* ce qu'est une chaîne de caractères ou un identificateur ? Pour être sûr que nous nous comprenons, jusqu'au moindre détail. Être précis est une bonne chose. Mais surtout, pour permettre à un *mécanisme algorithmique* de vérifier automatiquement si ce que nous donnons comme suite de caractères est une chaîne ou un identificateur *bien formé*. Ce mécanisme symbolique vérifie qu'il n'y a pas d'incohérence syntaxique. Cela ne veut pas dire que ce que nous allons spécifier est *valide*, c'est-à-dire constituera des données cohérentes, qui pourront être traitées, mais que les confusions syntaxiques ont été évitées.

Cette notion d'expression régulière est puissante en pratique et importante pour la formalisation des langages informatiques.

Définir un booléen. Une valeur binaire vraie ou fausse à deux valeurs se spécifie, en général :

```
boolean : false | true
```

où
`S | S'`, où `S` et `S'` sont deux constructions, signifie : soit `S`, soit `S'`.

Utiliser un `boolean` n'est qu'une façon plus « lisible » de dire « 0 » ou « 1 » en rendant explicite la sémantique vrai/faux de ce codage.

Définir une énumération. S'agit-il de définir par exemple une note de musique, sans altération, de la gamme occidentale ou une des sept couleurs de l'arc-en-ciel telles que nous les énumérons en Occident, pour éviter que la valeur soit mal définie, nous voilà conduits à définir d'autres types, comme par exemple

```
note : (do | re | mi | fa | sol | la | si)
```

ou

```
couleur : (rouge | orange | jaune | vert | bleu | indigo
| violet)
```

pour énumérer explicitement les valeurs possibles que prend une variable de ce type.

Là encore, ce n'est qu'une façon plus « lisible » d'attribuer un numéro à chaque item de cette énumération : tout se code en binaire, finalement. Mais ce serait juste indépétrable pour nous de ne pas substituer aux codes binaires bruts les corpus de mots qui désignent les objets réels dont nous codons un reflet numérique ici.

En pratique, deux types d'énumérations seront considérés :

– les énumérations *figées*, par exemple les noms des notes de la gamme, il y en a sept et cela n'aurait pas de sens dans la musique actuelle de créer un huitième nom. De même, si quelqu'un crée en plus de `false|true` une troisième valeur `bof`, nous sortons de la logique usuelle,

– les énumérations *adaptatives*, où en revanche cela a du sens de permettre d'ajouter de nouvelles valeurs, par exemple pour les couleurs, quelqu'un pourra ajouter `mordoré` ou `bleu_vert` pour désigner des couleurs intermédiaires de l'arc-en-ciel. Le système permettra alors d'ajouter *ponctuellement* une autre valeur – du coup, les valeurs de l'énumération ne servent que de valeurs exemples ou de valeurs par défaut – ou de *thésauriser* la nouvelle valeur en la gardant en mémoire pour servir d'exemple ultérieurement. C'est typiquement le cas d'un mot-clé qui sert à indexer un document : on donne une liste initiale qui s'enrichit au fil du temps.

Définir un nombre. De la même manière un nombre entier se spécifie :

```
integer : [-+]?[0-9]+
```

où

`S ?`, où `S` est une construction, signifie : `S` répétée 0 fois ou 1 fois, et
`S+`, où `S` est une construction, signifie : `S` répétée une fois ou plus.

La définition se lit « un signe - ou un + optionnel, c'est le sens de `[-+]?`, suivi d'un chiffre une fois ou plus ». Un nombre décimal se spécifiera :

```
float : $integer([.][0-9]+)?(e$integer)?
```

où `([.][0-9]+)?` correspond à l'ajout optionnel de décimales et `(e$integer)?` correspond à l'ajout optionnel d'un exposant, par exemple `314e-2 = 3.14`. Nous sommes ici en base 10 puisque nous spécifions comment un utilisateur va définir un nombre. Ici il n'y a pas de notion de précision machine, puisque le nombre est défini sous forme de chaîne de caractères : il n'y a pas un nombre limite de chiffres ou de décimales. C'est au moment où un calcul va s'effectuer que les limites entreraient en jeu.

Ce que nous gagnons ici, c'est que si la chaîne de caractères est bien formée, c'est-à-dire respecte la syntaxe donnée, *il y a bien un nombre qui correspond à cette chaîne de caractères* : ce n'est pas n'importe quoi.

La notion de type. Un concept important émerge ici. Les valeurs des variables ne sont pas n'importe quoi, elles ont un *type*. Le paramètre `age_du_capitaine` sera du type entier positif – voire, pour raffiner et détecter des incohérences sur sa valeur, un entier positif inférieur à 200. La définition d'une `gamme` musicale sera sûrement quelque chose comme `(do | re | mi | fa | sol | la | si) (mineur | majeur)` si nous oublions dièses et bémols. En écrivant :

```
age_capitaine : positive_integer
```

nous déclarons que la variable de nom `age_capitaine` est de type `positive_integer` et quelles que soient les variantes syntaxiques. Certains langages préfixeront le type, en écrivant plutôt `positive_integer age_capitaine;` ou même `positive_integer age_capitaine = 59;` pour donner un type et une valeur par défaut à la variable.

Nous devons bien garder à l'esprit que toute valeur se stocke dans une variable qui est référencée par une étiquette et qui est *définie par son type* – comme les « ensembles de définition » en mathématiques, qui permettent de bien définir les fonctions.

La bonne nouvelle est que les chaînes de caractères `String`, les nombres entiers `integer` ou flottants `float` et les booléens `boolean` ou les énumérations `enumeration` sont les cinq types de base des spécifications symboliques que nous rencontrons en pratique. Le reste correspond à des données structurées que nous présentons dans le sixième chapitre.

Quantifier l'information

Notion algorithmique d'information incompressible

Est riche en information un message… complexe à calculer.

Complexité du contenu en information Si le but est de pouvoir fabriquer/reconstituer une suite binaire *s* correspondant au codage d'un objet numérique – nous savons désormais que *tous* les objets numériques se codent sous la forme de suite binaire – et si nous supposons que nous disposons pour cela d'une machine *M* qui produit une suite binaire correspondant à un objet numérique, alors une vision naturelle de la valeur de l'information de *s* est *la longueur du plus petit programme écrit en binaire qui, lorsqu'on le donne à* M, *lui permet de reconstituer la chaîne* s.

Expliquons cette notion. Si le contenu est pauvre en information, par exemple 0000… avec un milliard de 0, alors nous nous attendons à ce qu'un tout petit programme, dans cet exemple « `écrire` 10^9 `fois "0"` », génère *s*. En revanche, s'il existait un contenu *s* tellement aléatoire qu'aucun programme

au monde ne puisse le prédire, alors le programme naturel pour le prédire est « `écrire "s"` », c'est-à-dire un programme où *s* est mémorisé explicitement avec l'instruction pour le produire sur la machine *M*.

On positionne donc la notion d'information par rapport à la *taille* du plus petit programme qui peut la produire. Cette mesure est bornée par l'ordre de grandeur de la taille de la suite elle-même, c'est, au pire, « `écrire "s"` ». La valeur en information est en quelque sorte le *contenu incompressible* de *s*. On parle de compression sans perte, puisque l'on remplace la chaîne *s* par quelque chose de plus court, le plus petit programme qui la génère.

Cette notion semble anecdotique, puisque liée à la machine *M*. C'est là qu'un résultat important intervient. Certes, les machines *M*, *M'* peuvent aller plus ou moins vite, mais, dès que l'on a affaire à une machine d'une certaine puissance, dès qu'elle exécute les mécanismes algorithmiques de base, alors elle peut exécuter « tous les algorithmes du monde ». Donc, tous les algorithmes qu'une *autre* machine peut calculer. Et réciproquement, évidemment. Donc, ce qu'elles peuvent calculer est le maximum de ce que n'importe quelle machine puissante peut calculer. Un téléphone portable, pourvu que son processeur soit un processeur standard, a la même potentialité de calcul que le plus grand super-calculateur du monde – en beaucoup moins rapide. C'est là la découverte fondamentale de Turing en 1936 : il y a des mécanismes universels de calcul et n'importe quel processeur de système numérique – ordinateur, smartphone, téléviseur numérique… – constitue un tel mécanisme universel.

En conséquence, pourvu que l'on se donne un mécanisme de calcul universel, l'ordre de grandeur du plus petit programme pouvant engendrer *s* ne dépend pas de la machine universelle que l'on utilise.

En fait, il ne dépend de cette machine que de la manière suivante. Prenons une machine *M* et écrivons le programme *P* qui simule une autre machine *M'*. Ce programme *P* prend chaque instruction de la machine *M'* et le traduit en l'instruction de la machine *M* pour l'y exécuter. Un tel programme existe de par l'équivalence des machines de calcul universel. On voit donc qu'engendrer la suite *s* sur *M* ou *M'* demande des programmes du même ordre de grandeur. Plus précisément, la taille de ce programme ne dépend de cette machine que par une constante additive, la longueur de l'émulateur *P* qui simule *M'* sur *M*. C'est une constante que l'on peut négliger en première approximation si on traite des suites suffisamment longues. C'est la découverte de ce résultat d'indépendance, ou théorème d'invariance, qui fonde la théorie algorithmique de l'information.

Interprétation du contenu en information Cette mesure est liée à la notion d'aléa, au sens de « non prédictible ». Nous savons qu'il est facile de générer des

nombres entiers qui ont l'air aléatoires. Par exemple, la suite de nombres $a_0 = 0$, $a_n = (a_{n-1} + K) \% 2^n$ où l'on ajoute un grand nombre premier K au nombre précédent avant de ne garder que les n premiers bits – c'est ce que fait l'opération $p \% 2^n$, qui est le reste de la division euclidienne de p par 2^n – aura une distribution pseudo-aléatoire bien uniforme. Mais cette suite $s = \{a_0, a_1, \ldots, a_N\}$ de nombres a une bien faible complexité algorithmique, puisque la voilà générée par un tout petit programme. Une suite vraiment aléatoire est au contraire une suite s qui ne peut être prédite par aucun programme sauf « `écrire "s"` ». Plus formellement : c'est une suite binaire qui n'est produite que par des programmes plus longs qu'elle. L'étude de ces suites est un domaine de recherche ardu et très intéressant, et qui n'a pu émerger que grâce à l'informatique : c'est un des exemples de nouvelles mathématiques issues de l'informatique.

Un des premiers résultats de cette théorie est un résultat d'existence de suites aléatoires. En effet, il y a $N = 2^{1001} - 1$ suites binaires de longueur inférieure ou égale à 1000. Les programmes étant eux-mêmes des suites binaires, il y a au plus N programmes, de longueur inférieure ou égale à 1000, qui produisent une suite binaire. Or, parmi ces programmes, certains, comme le programme « `écrire 2000 fois "0"` », produisent une suite binaire de longueur strictement supérieure à 1000. Il y a donc strictement moins de N programmes, de longueur inférieure ou égale à 1000, qui produisent des suites binaires de longueur inférieure ou égale à 1000. Il existe donc une suite s, parmi les N suites binaires de longueur inférieure ou égale à 1000, qui n'est produite par aucun programme de longueur inférieure ou égale à 1000. Tous les programmes, par exemple le programme « `écrire "s"` », qui produisent la suite s sont donc strictement plus longs que s.

Voici la notion de complexité de Kolmogorov ou de contenu en information de Kolmogorov. Elle capture la complexité aléatoire de l'information et on peut la relier aux notions usuelles d'incompressibilité, d'*imprévisibilité*, donc d'absence de structure. Andreï Kolmogorov a introduit l'idée – Ray Solomonoff sans être aussi précis l'avait eue avant lui, et Leonid Levin, Per Martin-Löf et Gregory Chaitin ont fait le travail qui a permis de faire aboutir ce concept scientifique collectif.

Un cran plus loin, des mécanismes de calcul plus limités peuvent être considérés. Par exemple, des mécanismes ne servant pas à coder toutes les suites finies possibles mais seulement une certaine famille de suites, ou ne disposant que d'une quantité limitée de mémoire, ou devant travailler en temps proportionnel à la taille de s, etc. Nous parlons alors de complexité *descriptionnelle* ou de contenu descriptionnel en information d'une chaîne binaire s, lorsque nous ne supposerons pas que M est une machine universelle, mais un autre système

numérique. Un exemple très simple est la « table de référence » : considérons un nombre fini, disons L, de chaînes binaires $s_\ell, \ell \in \{1, 2, \ldots, L\}$ de longueur quelconque, par exemple les L films numériques d'une médiathèque. Dans ce cas fini, chaque film se décrit par son numéro $\ell \in \{1, \ldots, L\}$ donc un nombre de $\log_2(L)$ bits, dans ce cas très spécifique. Au contraire, la notion d'information de Kolmogorov est sans doute la moins arbitraire de toutes les définitions générales de contenu en information, du fait du résultat d'invariance évoqué ici.

Il y a un autre lien entre cette notion et l'informatique théorique. Cette notion de complexité est très bien définie mais… elle n'est en général pas calculable ! De même qu'il n'existe aucun procédé algorithmique pour garantir qu'un programme va ou non boucler à l'infini – on ne peut pas, sauf information additionnelle, exclure qu'il finisse par s'arrêter –, de même il est *prouvé* qu'aucun procédé de calcul, aucun programme, ne peut calculer la complexité en information de toutes les suites binaires finies. Pour calculer en pratique ce qui est donc incalculable – nous serions dans la même situation pratique si la quantité était calculable, mais de manière exponentiellement compliquée –, la démarche naturelle est d'en *approximer* la valeur. Les techniques consistent à compresser le message par les algorithmes que nous avons décrits précédemment et étudier dans quelle mesure la longueur du programme de décompression et de la chaîne compressée constitue une bonne approximation de la complexité de Kolmogorov.

Notion algorithmique d'information organisée

Complexité organisée et profondeur de l'information Avançons dans notre étude de l'information.

On dit souvent dans le langage courant qu'une pensée est profonde si elle est l'aboutissement d'un long temps de réflexion. La notion de profondeur logique proposée par Charles Bennett en 1988 capture de manière magistrale cette idée.

Ici, on ne cherche plus à définir une notion d'information aléatoire ou incompressible, mais une notion d'information liée à une complexité organisée. Regardons quelques exemples :

– simple et peu profond — un cristal
structure périodique d'organisation simple est sans aléa.

– simple et profond — π à 10^{-100} près
calculable par un petit programme, mais gros effort de calcul.

– aléatoire et peu profond — gaz parfait
pas prédictible donc pas reproductible, mais organisation triviale.

– aléatoire et profond — être vivant
pas prédictible et résultat de millions d'années d'évolution.

Nous voulons donc capturer le fait que cette information est « précieuse » par le fait qu'il a fallu un « effort de calcul » pour y arriver. Bennett propose à cette fin de prendre en compte *le temps de calcul du plus petit programme écrit pour une machine universelle qui génère le contenu binaire* en question. Le programme le plus court est celui qui a servi à calculer la complexité de Kolmogorov.

Notons tout de suite que ce n'est sûrement pas le temps de calcul du programme le plus rapide, qui est très probablement « `écrire "s"` », c'est-à-dire quelque chose de trivialement relié à la longueur du message et non à la complexité de son organisation.

Ce qui est remarquable, c'est que cette notion « tient bon », c'est-à-dire qu'elle commence à être reconnue comme une « bonne notion » de complexité organisée. Elle n'est pas additive si les données peuvent être reliées : deux moutons ne sont pas deux fois plus complexes que un, la croissance est lente : la profondeur augmente avec du calcul, mais lentement et il y a apparition spontanée de complexité organisée par le calcul.

Cette notion est robuste, c'est-à-dire qu'elle dépend relativement peu de la machine universelle que l'on se donne.

C'est bien sûr une quantité non calculable puisque basée sur la complexité de Kolmogorov.

Interprétation de la profondeur de l'information La proposition de Bennett n'est peut-être pas une proposition ultime, et il se peut que la notion de complexité organisée puisse encore être formalisée en se fondant sur d'autres bases ou en enrichissant l'idée de Bennett. Cette possibilité d'une évolution ne semble pas envisageable pour la complexité de Kolmogorov qui, elle, est une proposition définitive. Mais nous capturons là de manière formelle l'idée essentielle d'émergence, au cœur de l'idée évolutionniste darwinienne : un objet richement structuré et organisé ne peut pas sortir de rien, instantanément, mais demande un long processus d'interaction entre ses divers éléments, y compris son environnement, c'est-à-dire une sorte de calcul prolongé et cumulatif. La profondeur logique de Bennett mesure ainsi la quantité de calculs fixée dans un objet, c'est une mesure du contenu computationnel de l'objet, c'est une mesure de la longueur de la dynamique qui lui a donné naissance. L'émergence d'un objet profond ne peut être brusque, alors que l'apparition d'un objet complexe, au sens de la complexité de Kolmogorov, dans certaines situations se produit instantanément. C'est probablement ce qui explique de la manière la plus formelle pourquoi le leurre de « l'intelligence artificielle » – pouvoir programmer rapidement un être virtuellement intelligent – ne tient pas : ce qui fait l'intelligence a dû émerger de millions d'années de « calcul » biologique.

Lorsque le cerveau d'un bébé se construit en quelques mois, c'est en grande partie en « recopiant » ou « décompressant » le résultat de ces « calculs » biologiques millénaires.

Voici une des pointes de la recherche en informatique aujourd'hui. Que le lecteur ne s'y trompe pas : la présentation succincte de ces notions ne doit pas masquer (1) la très forte technicité de ces notions, car les « mots » cachent des formalismes difficiles, et (2) le chemin scientifique qui a pu les produire. Avant d'en « arriver » à la profondeur logique de Bennett, les cimetières académiques se sont remplis de « mauvaises » notions.

Notion probabiliste d'information

Quand la notion d'information est abordée avec une vision probabiliste, on considère que l'on accède aux événements observés par la mesure de leur probabilité d'occurrence. Et donc savoir quelque chose sur un événement, avoir de l'information, est lié à la probabilité que cet événement se produise. Un événement n'est pas « vrai » ou « faux », mais plus ou moins *probable*. Un rappel des bases des probabilités est proposé plus loin, au paragraphe « La vision probabiliste des choses : un rappel ».

Contenu probabiliste en information Comment définir une notion d'information dans ce contexte ? Assez naturellement, nous allons présupposer que le contenu probabiliste en information n'est fonction que de la probabilité des événements, puisque c'est ce qui donne l'information sur les événements.

Pour la conjonction d'événements, posons que les informations de deux événements indépendants s'ajoutent : si E_1 et E_2 sont indépendants, alors $h(E_1 \cap E_2) = h(E_1) + h(E_2)$, comme nous l'expliquions informellement en introduction.

Pour la disjonction d'événements incompatibles, posons que l'information se moyenne : si E_1 et E_2 sont incompatibles alors $h(E_1 \cup E_2) = p(E_1)\ h(E_1) + p(E_2)\ h(E_2)$. En effet, si l'événement $E_1 \cup E_2$ se produit, alors soit E_1 soit E_2 se produit, ce qui apporte une information soit $h(E_1)$ soit $h(E_2)$, et l'espérance de cette information combinée est donc $p(E_1)\ h(E_1) + p(E_2)\ h(E_2)$.

Finalement, il serait cohérent qu'un bit d'information corresponde à ce que nous gagnons en information de savoir qu'un élément binaire vaut 0 ou 1.

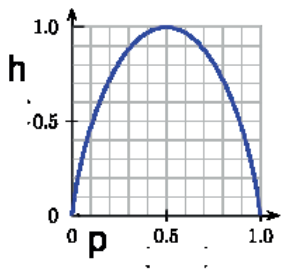
Avec ces trois leviers, il se trouve que la notion d'information est complètement et précisément définie. C'est l'*entropie* au sens de Shannon et sur un ensemble fini ou dénombrable. Elle vaut, en supposant que c'est une fonction continue :

$$h(E = .\{e_1, \quad e_2, \ldots, \}) = -\sum_{e_n \in E} p(e_n) \log_2(p(e_n))$$

Elle donne le *contenu moyen en information d'un événement d'une probabilité*. Il faut comprendre que c'est l'information que nous gagnons à connaître lors de la réalisation d'un événement aléatoire.

Regardons ce qui se passe si nous lançons une pièce de monnaie qui va tomber avec une probabilité p sur pile et $1 - p$ sur face. Son information vaut $h(pile, face) = -p\ log_2(p) - (1 - p)\ log_2(1 - p)$.
Si $p = 1/2$, c'est-à-dire qu'elle a autant de chance de tomber sur pile ou face, alors $h(pile, face) = 2(-1/2\ \log_2(1/2)) = 1$: nous gagnons exactement un bit d'information à savoir que la pièce tombe sur pile ou face.

Si, au contraire, $p = 0$, c'est-à-dire qu'elle va forcément tomber sur pile, alors puisque $0\log_2(0) = 0$ et que $\log_2(1) = 0$, $h(pile, face) = 0$: nous ne gagnons aucune information à savoir que la pièce tombe sur pile ou face, puisque ce sera pile. Si nous traçons l'entropie $h(pile, face)$ en fonction de la probabilité p



nous observons les éléments suivants :
– l'entropie est une quantité positive ou nulle ;
– l'entropie est minimale, égale à 0, si une des probabilités $p(e_n) = 1$, toutes les autres valant alors 0, donc si la situation est *sans aléa*, c'est-à-dire le résultat est certain ;
– l'entropie est maximale si l'aléa est maximal, c'est-à-dire que la distribution de probabilité est uniforme, sans privilégier aucune valeur ; ici, elle est égale à 1 pour un bit d'information. Elle est égale à $\log_2(N)$ si l'événement E prend N valeurs ;
– l'entropie est une fonction concave ; elle est symétrique : on peut inverser pile et face sans changer sa valeur, ou plus généralement inverser les valeurs de l'événement sans changer l'entropie.

Ce que nous voyons sur cet exemple est tout à fait général. La réalisation d'un événement apportera un maximum d'information si tous les événements sont équiprobables, donc complètement au hasard : s'il y a N réponses possibles et équiprobables, connaître cette information doit correspondre à N bits,

c'est-à-dire N « atomes » d'informations et vaudra $\log_2(N)$. C'est exactement le nombre de bits pour coder la suite binaire.

Cette notion d'information se relie à l'entropie dont il est question en thermodynamique, car elle propose une mesure du « désordre » au sein des données. Plus elle est importante, plus les données sont imprévisibles, donc susceptibles de contenir de l'information. Dans ce paradigme, on suppose que le récepteur est susceptible de faire certains calculs pour reconstituer *s* à partir de ce qu'on lui transmet. La machine *M* qui fait le décodage peut être prise comme référence, et on découvre alors que la théorie de Shannon doit être vue comme une version probabiliste de la théorie algorithmique de l'information de Kolmogorov et compatible avec elle dans le sens suivant : *le contenu algorithmique moyen d'information de Kolmogorov des chaînes binaires de longueur* n – *pondérées par les probabilités résultant des fréquences supposées – est du même ordre de grandeur que le contenu en information au sens de Shannon.*

De plus, cette notion d'information est une *véritable mesure physique, avec une unité, le bit.* C'est une quantité abstraite que l'on ne voit ni n'entend, comme l'énergie.

Cette notion a de nombreuses applications : plus des données sont prévisibles, plus il est facile de les compresser. De même, plus un message contient d'information au sens défini ici, plus le canal pour transmettre ce message devra avoir une grande capacité.

Si cette notion ne dit rien du sens qui peut être attaché à ces données, ni de la complexité nécessaire à leur création, elle est le pilier de plusieurs domaines de l'informatique et des sciences du numérique. Nous en verrons deux ici.

Transmission d'information L'entropie de Shannon est utilisée pour numériser une source avec le minimum possible de bits sans perte d'information. Elle permet aussi de quantifier le nombre minimum de bits sur lesquels on peut coder un fichier, mesurant ainsi les limites que peuvent espérer atteindre les algorithmes de compression sans perte, que nous discuterons dans le sixième chapitre. Plus précisément, cette formulation donne le contenu moyen d'information de l'ensemble des chaînes binaires *s* quand on tient compte des probabilités des blocs de bits utilisés. Le « théorème de la voie sans bruit » indique que l'on ne peut pas en moyenne compresser plus *s*.

Le point clé est donc :

Une variable d'entropie *h(x)* sera « presque sûrement » codée avec *h(x)* bits.

La théorie de l'information de Shannon est donc aussi une théorie de l'information par compression qui, au lieu de considérer des suites quelconques, suppose que les suites que l'on transmet vérifient certaines propriétés statistiques. Finalement, la théorie de Shannon est une théorie du contenu en infor-

mation relativement à un but de compression et à une certaine distribution statistique des suites.

La compression de données ou codage de source est l'opération informatique qui consiste à transformer une suite binaire en une suite plus courte, contenant les mêmes informations, en utilisant un algorithme particulier. Les « fichiers zip » sont des exemples de données compressées. Il n'existe pas de technique de compression de données sans perte universelle, qui pourrait compresser n'importe quel fichier : on démontre que si une technique sans perte compresse au moins un fichier, alors elle en grossit également au moins un.

Estimation d'information à partir de l'entropie L'entropie de Shannon est également utilisée dans d'autres domaines comme, par exemple, pour la sélection du meilleur point de vue d'un objet en trois dimensions, recaler deux images différentes l'une sur l'autre en minimisant l'entropie des deux images, etc.

Discutons maintenant cette facette de la théorie de l'information. Que veut dire estimer une valeur incertaine ? L'idée est que si je ne sais rien de rien, je ne peux qu'agir au hasard. Si je ne sais rien de la pièce de monnaie, il n'est sûrement pas raisonnable de penser qu'elle va tomber plus sur pile que sur face. L'hypothèse minimale est, compte tenu de mon manque d'information, de supposer qu'elle va tomber aléatoirement sur pile ou face, donc qu'il y a une probabilité $p = 1/2$ de tomber sur pile, ou face. En d'autres termes, faute d'information, choisissons la solution *d'entropie maximale*. Celle où l'on confie au hasard ce que nous ne pouvons pas expliquer.

Ce choix méthodologique est un principe de simplicité ou parcimonie que l'on nomme parfois « rasoir d'Occam ». C'est un principe qui refuse de présupposer une cause « par-delà ce qui est nécessaire ». À l'instar de ce dialogue entre Napoléon « ne trouvant pas mention de Dieu dans le système » de Monsieur de Laplace qui se doit alors de préciser : « Je n'ai pas eu besoin de cette hypothèse [qui] explique en effet tout, mais ne permet de prédire rien. » Bref, ce que nous n'expliquons pas est de l'*information non encore obtenue*, de l'entropie. Et son modèle minimal est le hasard.

Ce qui est remarquable ici est que ce principe général s'incarne alors dans un formalisme précis et nous offre un moyen de *calculer*, et même d'estimer au mieux en fonction des mesures que nous avons.

Les grandes distributions de probabilité que nous manipulons usuellement ne sont rien d'autre que des lois d'entropie maximale.

– Supposons que nous connaissions la moyenne et la variance d'une variable aléatoire continue, mais rien d'autre. Quelle est la « bonne » distribution ? C'est-à-dire celle qui va rendre compte de la moyenne et variance, mais sans

rien ajouter de fallacieux ? C'est la loi normale, dite gaussienne, avec sa courbe de Gauss en forme de cloche.
– Supposons que des événements se produisent aléatoirement et que nous connaissions le rythme moyen de ces événements, mais rien d'autre. C'est la loi de Poisson, qui aura alors les vertus demandées. Et ainsi de suite. En fait, nous pouvons suivre la même démarche *dans tous les cas*.

Cherchons la « vérité » sur un événement E, c'est-à-dire les valeurs $p(e_n)$ de sa distribution de probabilité. Pour cela, nous disposons de quelques mesures, par exemple une mesure de moyenne ou de dispersion. Nous savons donc que la distribution de probabilité n'est pas au hasard, puisqu'elle vérifie ces équations de mesure. Pour le reste, le hasard demeure. Alors, parmi toutes les distributions de probabilité, nous allons prendre celle d'entropie maximale. Nous passons d'une situation où il y a des équations de mesure que nous ne savons pas très bien résoudre, à une situation où nous définissons parfaitement la solution. Nous en détaillons les équations en complément, plus loin, au paragraphe « La vision probabiliste des choses : un rappel ».

Il s'est passé quelque chose d'essentiel dans le calcul ébauché ici : nous sommes passés d'un souhait louable – « n'expliquer que l'information obtenue par nos mesures, le reste étant laissé au hasard » – à un *calcul effectif*, qui (1) nous donne la forme de la distribution de probabilité, (2) permet de poser les équations du calcul numérique pour déterminer les paramètres de cette probabilité. Nous sommes passés d'une spécification, le « quoi » qui formalise ce que nous voulons obtenir, à une implémentation, le « comment » qui fournit un algorithme pour le calculer.

La classe d'algorithmes considérée ici est celle d'estimation de valeurs numériques par minimisation. Nous donnons plus de détails au paragraphe « Estimation d'information par optimisation ». La notion est scientifiquement belle, car elle rend simple quelque chose de compliqué. Dont acte. Seuls les Shadoks, double privilège de l'artiste et de l'humoriste, gardent alors encore le droit de rétorquer à cela : « Pourquoi faire simple quand on peut faire compliqué ? ».

Que faire après avoir estimé au mieux des valeurs numériques ? Agir, c'est-à-dire prendre des décisions, le plus souvent. Donc, comparer deux solutions pour mesurer s'il y a une différence significative – par exemple, comparer les scanners d'un patient quelconque avec un patient de référence. La théorie de l'information nous donne une nouvelle clé pour cela : nous pouvons mesurer quelle part d'aléatoire il reste dans un événement aléatoire en question après avoir été expliqué ou prédit par un événement aléatoire de référence : c'est l'*entropie relative*. On parle de *divergence*, qui est une mesure de dissimilarité

entre deux distributions de probabilité. Dotés de cet outil, il est alors possible de comparer des modèles, de statuer sur le fait que ces informations sont indépendantes ou non, de regarder le poids d'une conséquence par rapport à une cause, etc. D'autres outils probabilistes comme l'inférence baysienne viennent compléter ces méthodes qui s'adossent à ces concepts centraux.

Voici notre arsenal moderne de base pour traiter et manipuler l'information.

Conclusion : manipulation de l'information

La première idée est celle de l'utilisation d'algorithmes génériques, c'est-à-dire de méthodes de calcul et de transport insensibles à la nature et au contenu de l'information, car ne voyant que les bits 0 ou 1 qui la composent. On trouve ces algorithmes dans toutes les opérations de stockage, de cryptage et de transmission de données. C'est grâce à eux que l'on peut rendre l'information pérenne et indépendante des supports. En numérique, on peut effectivement transporter, répliquer, sauvegarder et diffuser arbitrairement l'information sans aucune perte de qualité, ce qui est fondamentalement impossible avec les supports analogiques traditionnels. C'est évidemment essentiel pour les télécommunications, qui ont pour charge principale d'assurer le déplacement de l'information dans le temps et dans l'espace sans la modifier. De plus, transport et sauvegarde se font avec un gain de place et une sécurité majeure par rapport aux méthodes analogiques : une bibliothèque numérique tient sur quelques grammes et, si on en garde des copies multiples, il n'est pas grave que l'une d'entre elles brûle.

La deuxième idée est l'utilisation d'algorithmes spécifiques liés à un type et à un volume d'informations donnés. On cherche les fautes d'orthographe dans les textes, pas dans les images. On ne compresse pas de la même façon les textes, les photos, la vidéo et les sons ; on zoome sur une image, pas sur un texte ; il est bien plus simple de chercher de l'information dans un ensemble de textes que dans un ensemble d'images ; les informations liées au pilotage d'un avion sont fondamentalement volatiles, alors que celles liées à sa maintenance doivent être gardées toute sa vie, etc. On assiste à l'heure actuelle à un développement considérable d'algorithmes spécifiques, très divers mais tous fondés sur un petit nombre de principes scientifiques que nous décrirons plus tard.

La troisième idée est celle de la multiplicité des représentations. Une même chose peut être représentée de façon différente selon les cas et les utilisations. Par exemple, considérons l'exemple de la représentation d'une matrice. La façon la plus simple et la plus évidente est la représentation matricielle, comme tableau bidimensionnel. Cela permet que certains algorithmes, comme l'addi-

tion ou la multiplication de matrices, soient simples. Imaginons maintenant que, pour des raisons physiques, les matrices que l'on considère soient de taille importante mais creuses – avec beaucoup de valeurs nulles. Dans ce cas, il peut être rentable en termes d'occupation mémoire d'utiliser une autre représentation, par exemple une liste de tous les endroits où la matrice est non nulle avec sa valeur en ce point. Il faut alors utiliser un algorithme d'addition de matrices adapté à cette représentation. Il sera un peu plus complexe, mais beaucoup plus rapide si les matrices sont très creuses. La représentation de l'information et l'algorithme la manipulant sont donc intimement liés.

Exercices corrigés et commentés

Exercice 1. Jouons au jeu du portrait.

Devinez le nom d'une personne en ne posant que des questions auxquelles on répond par oui ou par non dans la liste suivante

Qui ?	Genre ?	Looké Emo ?	Moins de 15 ans ?	
Pierre	garçon	oh non ! 0	oui	'001'
Nadia	fille	good-good 1	oui	'111'
David	garçon	énormément 1	non	'010'
Hamdi	garçon	certes 1	oui	'011'
Marie	fille	surtout pas 0	non	'100'
Adèle	fille	évidemment 1	non	'110'
MingYu	fille	beurk 0	oui	'101'
Igor	garçon	horreur! 0	non	'000'

Pour que les choix restent binaires, la réponse à propos du « looké Emo » ne peut être que oui ou non, sans introduire les nuances explicitées ici.

Y a-t-il deux personnes qui ont les mêmes attributs dans la liste proposée ?

Combien faut-il au minimum de questions oui/non pour deviner une personne de ce tableau ? Dans ce cas, est-ce suffisant ?

Que se serait-il passé si la proportion entre garçons et filles n'était pas égale ? Ou si la répartition des « looké Emo » avait été différente ?

Si la réponse à propos du « looké Emo » avait été explicitée comme dans le tableau et non binaire, que se serait-il passé ?

Peut-on interpréter chacun des bits de la colonne de droite ?

Pour deviner une personne parmi seize, combien faudrait-il de questions binaires au minimum ? Quelles propriétés devraient avoir ces questions pour être en nombre minimum ?

Refaire le jeu avec quatre personnes de la classe en leur trouvant des bits d'information pertinents.

Correction. En trois questions, « Est-ce une fille ? » « Aime-t-elle le look Emo ? » « A-t-elle moins de 15 ans ? », nous allons forcément deviner parmi les huit personnes celle correspondant au portrait, car toute l'information est utile ici. D'ailleurs, nous avons symbolisé ce fait en utilisant un bit pour chacune des questions binaires, reporté dans la colonne de droite.

Deux questions binaires, c'est-à-dire deux bits d'information, ne suffisent pas ici. Par exemple, si on ne sait pas si la personne recherchée a moins de 15 ans, alors Adèle et Nadia ou Igor et Pierre, par exemple, ne peuvent pas être distingués, puisqu'ils pensent la même chose du look Emo. On constate bien ici que le nombre de bits correspond au nombre de questions binaires à poser pour deviner toute l'information. Cela correspond donc à la taille en information.

On voit ici que la réciproque est fausse : ce n'est pas parce qu'il y a trois bits d'information que nous pouvons distinguer huit éléments : si tous ou simplement plus ou moins de la moitié des garçons avaient eu moins de quinze ans, par exemple, les deux informations auraient été redondantes et le codage de l'information n'aurait pas été suffisant.

On mesure aussi qu'il faut bien s'entendre sur le codage choisi : dans le cas du « looké Emo » les réponses contiennent bien plus d'information que oui ou non, mais des appréciations de valeur ou des nuances de ton qui peuvent, ou non, être elles-mêmes codées lors de la numérisation de l'information.

Exercice 2. Combien de temps peut durer ce jeu ?

Deux personnes A et B jouent à un jeu. A choisit un nombre entre 1 et 1000 que B doit deviner.

Avec la première règle, A ne répond que « oui » ou « non ». Si B est très intelligent mais très malchanceux, de combien d'essais au maximum a-t-il besoin ?

Avec la seconde règle, A répond « oui », « plus petit », ou « plus grand ». De combien d'essais au maximum B a-t-il besoin ?

Correction. Avec la première règle, il faut évidemment 1000 essais au maximum. B n'a aucune stratégie possible et se contente d'énumérer les nombres possibles, de 1 à 1000 par exemple. S'il est très malchanceux, le nombre choisi par A est 1000 et n'est trouvé qu'au millième essai.

Avec la seconde règle, B peut utiliser l'information supplémentaire et se permettre d'être plus intelligent. L'idée est de couper l'ensemble des nombres possibles en 2 à chaque question. On ne commencera donc pas par 1, mais par 500. L'ensemble des possibilités {1, …, 1000} sera alors réduit, si 500 n'est pas le nombre recherché, soit en {1, …, 499 }, soit en {501, …, 1000 }. On appelle cela une dichotomie. Un des pires cas correspond alors à
$\{1, \ldots, 1000\} \to^{500} \{1, \ldots, 499\} \to^{250} \{1, \ldots, 249\} \to^{125} \{1, \ldots, 124\} \to^{112} \{1, \ldots, 111\} \to^{56} \{1, \ldots, 55\} \to^{28} \{1, \ldots, 27\} \to^{14} \{1, \ldots, 13\} \to^{7} \{1, \ldots, 6\} \to^{3} \{1, \ldots, 2\} \to^{2} 1$.

Il faut poser au pire 10 questions pour obtenir avec certitude le résultat. Ce 10 est en fait la partie entière supérieure du logarithme en base 2 de 1000. En effet, on scinde l'intervalle en deux à chaque étape. Pour obtenir le nombre d'étapes, il suffit donc de calculer $\lceil \frac{\ln(1000)}{\ln(2)} \rceil = 10$.

Exercice 3. Binaire/décimal.

Écrire en décimal les nombres binaires 10010, 111010, 1111111.

Écrire en binaire les nombres décimaux 9, 42, 2049 et 10010.

Correction. En appliquant l'algorithme, on obtient $10010_2 = 18$, $111010_2 = 58$ et $1111111_2 = 127$.

De même, on obtient : $9 = 1001_2$, $42 = 101010_2$, $2049 = 100000000001_2$ et $10010 = 10011100011010_2$.

Exercice 4. Binaire/décimal signés.

Écrire en décimal les nombres binaires sur 8 bits : 10010, 10111010, 11111111.

Écrire en binaire sur 8 bits les nombres décimaux -9, -42, -2049 et -10010.

Correction. On a $10010_2 = 18$, sur 8 bits, son premier bit est 0. Puis, on a $10111010_2 = -27 + 58 = -70$ et $11111111_2 = -2^7 + 127 = -1$.

Ensuite, sur 8 bits, on a : $-9 = -2^7 + 119 = -2^7 + 1110111_2 = 11110111$ et $-42 = -2^7 + 86 = -2^7 + 1010110_2 = 11010110$. Enfin, 10010 nécessite 14 bits. Il en faut au moins 15 pour représenter −10010 qui ne peut donc pas être représenté sur 8 bits.

Exercices non corrigés

Exercice 1

Définir un codage binaire pour coder une information courante, par exemple les dates de naissance des personnes de la classe. Rechercher un code qui utilise un minimum de bits, *compte tenu* des données de la classe. Puis rechercher comment coder la date de naissance de n'importe quel élève du lycée.

Exercice 2

Chercher dans Wikipédia combien il y a de caractères codés en UTF-8 pour toutes les langues du monde. Expliquer en 2 lignes le lien entre le codage ASCII et UTF-8.

Exercice 3

Expérimenter les calculs en binaire : ajouter, soustraire et multiplier des nombres en binaire.

Exercice 4

Pour incrémenter un nombre binaire, il faut : 1. commencer par le bit de poids faible ; 2. inverser le bit ; 3. tant que ce bit est à zéro, recommencer 2. avec le bit situé à gauche.

Vérifier que cet algorithme est bien correct et décrire celui qui décrémente un nombre binaire. Et celui qui le multiplie par deux.

Exercice 5

Compter le nombre de bits pour coder une image, par exemple 1024 × 1024, en noir ou blanc, puis avec des niveaux de gris de 0 à 255.

Exercice 6

Ouvrir un fichier audio quelconque avec un logiciel comme `audacity` ou tout autre logiciel qui puisse lire les paramètres d'un fichier audio. Regarder la durée D du son en seconde, la fréquence F d'échantillonnage – 44 100 Hz ? Davantage ? –, le nombre C de canaux – mono ? stéréo ? –, et le nombre O d'octets pour coder un échantillon – 16 bits donc 2 octets ? Davantage ? Comparer la taille du fichier en octet au nombre $D \times F \times C \times O$ et en déduire le taux de compression.

Exercice 7

Concevoir le même exercice que le précédent mais pour une image.

Exercice 8

Sachant que le cerveau humain a environ 100 milliards de neurones qui ont chacun en moyenne 10 000 synapses qui transmettent environ 100 impulsions binaires, des potentiels d'action, par seconde, calculer la taille en information maximale de l'activité d'un cerveau.

Exercice 9

Ouvrir une image au choix avec un logiciel d'édition élémentaire d'images tel que `gimp` ou tout autre logiciel similaire et essayer de modifier la lumière et le contraste : laquelle de ces opérations correspond à ajouter/retirer une valeur constante à la valeur d'un pixel et laquelle correspond à multiplier le pixel par un nombre plus grand/petit que un ?

Exercice 10

Deviner et décrire avec des mots la transformation d'image qui correspond pour chaque pixel à faire la moyenne avec les pixels voisins. Indice : penser soit au zoom, soit à la mise au point d'une lentille optique.

Exercice 11

Prenons deux photos de suite d'une scène avec un objet fixe et un objet mobile et soustrayons les deux images, c'est-à-dire faisons la soustraction de chaque valeur du pixel de la deuxième image avec le pixel correspondant dans la première. Que s'est-il passé pour l'objet fixe ou mobile ? À quoi peut servir la soustraction d'images alors ?

Exercice 12

Exercice rédactionnel : expliquer en moins de 5 lignes le lien entre la complexité de Kolmogorov et la profondeur logique de Bennett.

Exercice 13

Que dire d'un poème qui aurait une très grande complexité de Kolmogorov ? et une profondeur logique de Bennett ?

Exercice 14

Combien a-t-on de chance de pirater mon code de carte de paiement – avec un code à 4 chiffres et le droit de faire 3 essais ?

Exercice 15

En combien de temps un moteur Internet qui peut essayer 1 000 mots de passe par seconde pourrait craquer un mot de passe de 6 lettres ? et de 6 caractères ASCII de 7 bits ? Est-ce bien utile de ne pas utiliser que des lettres pour son mot de passe ? Et si, pour se protéger, le site web multipliait par deux le délai à chaque essai – 1/1000 de seconde pour le 1er essai, 2/1000 de seconde pour le 2e essai… – que se passerait-il ?

Exercice 16

Calculer l'entropie d'un jet de dé à six faces non pipé. Et d'un dé avec « 1, 1, 2, 2, 2, 3 » sur les faces.

Exercice 17

Shannon propose une façon simple de déterminer l'entropie d'un texte donné pour un récepteur donné : A dispose du texte et demande à B de le deviner lettre par lettre, espaces compris. Si B devine correctement la lettre, on compte 1, car A, en lui répondant « oui », lui a transmis 1 bit d'information. Si B se trompe, on lui indique la bonne lettre et on compte 4.75, car un caractère parmi 26 lettres plus 1 caractère d'espace représente environ $ln_2(27) = 4.75$ bits d'information. Appliquer cette méthode à un texte courant. Constater qu'environ une lettre sur deux peut être ainsi devinée, la redondance du langage

courant était en conséquence d'un facteur 2, et le contenu informatif d'une lettre dans ce contexte de 2,35 bits seulement.

Exercice 18

En utilisant une calculette programmable ou tout autre outil de calcul, programmer la suite $a_0 = 0$, $a_n = (a_{n-1} + 7919)\%16$ et tracer ou faire compter les occurrences des valeurs : qu'en conclure ?

Exercice 19

Mon banquier m'a proposé cet investissement :

– on me donne $e = \exp(1)$ €,
– l'année suivante, je prends 1 € de frais et je multiplie par 1,
– l'année suivante, je prends 1 € de frais et je multiplie par 2,
– l'année suivante, je prends 1 € de frais et je multiplie par 3,
– …
– après n ans, je prends 1 € de frais et je multiplie par n,
– Pour récupérer mon argent, il y a 1 € de frais.

Dans 50 ans, pour ma retraite, combien d'argent aurai-je ?

L'intérêt de cet exercice repose dans l'utilisation de plusieurs calculatrices, logiciels de calculs… de façon à faire varier *grandement* la réponse obtenue.

Exercice 20

Codage décodage d'une phrase en arabe. Parmi les 28 symboles de l'alphabet arabe, avec ses symboles complémentaires, que l'on devine sur cette ancienne machine à écrire :

nous allons en considérer 16 :

ء	ا	ا	ؤ	إ	ئ	ا	ت	د	ش	ف	ك	ل	ن	و	ي
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Donner alors le codage binaire de la phrase, qui se lit de droite à gauche :

« كل شيء في ثنائي الكود»

Pour qui ne sait pas lire l'arabe : aller sur Google et essayer de traduire en arabe « tout se code en binaire ». Que remarquons-nous ? Aurait-il été facile de faire l'inverse ?

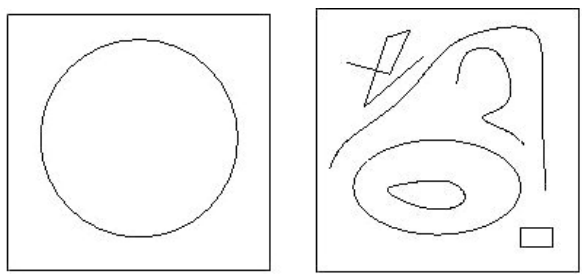
Questions d'enseignement

Découvrir le codage d'un dessin avec les élèves

Préambule : Plutôt que de se lancer dans des considérations générales de didactique, commençons par un exemple concret qui « fonctionne » bien avec les élèves. Il s'avère, en pratique, que la notion de tableau de pixels paraît naturelle aux jeunes élèves, qui sont entourés d'images au quotidien. En revanche, le « codage de chaque pixel » est en général un peu plus mystérieux. Dévoilons-le ici. Ce sera aussi l'occasion de montrer sur cet exemple concret la différence entre description numérique et symbolique d'un objet et l'approximation inhérente à la numérisation.

Ce paragraphe, pour tenter d'avoir valeur d'exemple, cible des élèves et non des enseignants.

Présentation de l'exemple : Regardons en détail comment coder un dessin au tracé simple. Comment faire pour décrire un dessin ? Par exemple, comment décrire le dessin de gauche ?



Une solution est de dire : « Cette image est formée d'un cercle. » Nous pouvons même être plus précis et indiquer les coordonnées du centre du cercle, son rayon, sa couleur, l'épaisseur du trait… À partir de cette description, n'importe qui pourrait reconstituer le dessin. On donne alors une description *symbolique* du dessin : « un cercle, dont le centre est le milieu de la figure, le diamètre les trois quarts de la figure, la couleur du trait est noire, son épaisseur est de 1 mm, tandis que le fond de la figure est blanc ». Il semble que nous n'ayons rien oublié : c'est-à-dire que *si nous donnons tous ces éléments à quelqu'un, il va pouvoir*

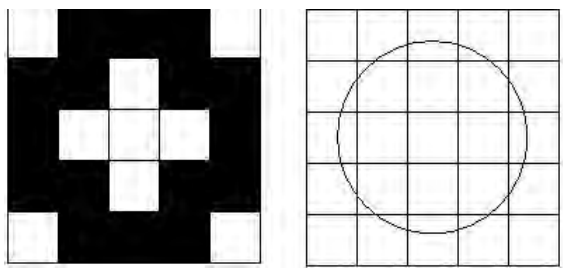
reproduire le même dessin. Nous pourrions, du reste, exprimer ces données de manière structurée :

```
{
 forme = {
   type = "cercle"
   centre = (size/2, size/2)
   rayon = 3/4 * size/2
  }
  trait = {
   couleur = "noir"
   largeur = "1 mm"
 }
 fond = {
   couleur = "blanc"
  }
 }
```

en supposant ici que `size` est la taille de la figure carrée. Le dessin est donc défini par sa forme, son trait et le fond. La forme a un type, ici un cercle, qui est entièrement défini par son centre et son rayon. Il faut, bien entendu, que la sémantique de chaque variable, `forme, type, trait, ...`, soit convenue et que les valeurs de ces variables soient comprises aussi.

Cette méthode marche bien pour le dessin de gauche. Mais tout décrire avec des mots serait bien moins pratique pour le dessin de droite ! En effet, comment facilement décrire en détail ces traits qui semblent tracés au hasard ? Qu'en serait-il avec un dessin qui représente un objet réel, ou un visage ?

Une autre méthode, qui a l'avantage de pouvoir être utilisée pour n'importe quel dessin, consiste à superposer un quadrillage au tracé, comme proposé à gauche :



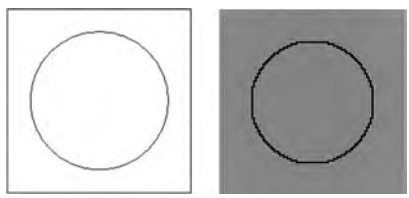
Chacune des cases de ce quadrillage s'appelle un pixel. On noircit ensuite les pixels qui contiennent une portion de trait, comme ci-dessus à droite. Puis, il nous suffit d'indiquer la couleur de chacun des pixels, en les lisant de gauche à droite et de haut en bas, comme un texte – dans notre culture occidentale. Ce dessin se

décrit donc par une suite de mots « blanc » ou « noir ». Comme seuls les mots « noir » ou « blanc » sont utilisés, nous pouvons être plus économes et remplacer chacun de ces mots par un seul symbole, par exemple le mot « noir » par la lettre « n » ou le chiffre « 0 » et le mot « blanc » par la lettre « b » ou le chiffre « 1 ».

Le dessin ci-avant, avec une grille de 5 × 5, se décrit alors par la suite de 25 chiffres : 1000100100011100010010001

Ces différentes descriptions sont équivalentes : le dessin se décrit toujours par une suite de symboles choisis parmi deux. Le nom de ces deux symboles importe peu, et nous pourrions les appeler noir et blanc, a et b, 0 et 1, pile et face, - et +, Dupond et Dupont… que cela ne changerait pas grand-chose. En général, on choisit 0 et 1.

De fait, cette description est assez approximative, et on constate une grande différence entre ces deux images, mais nous pouvons rendre la description plus précise en utilisant un quadrillage, non plus de 5 × 5, mais de 100 × 100 pixels.



À partir de quelques millions de pixels, nous ne serions plus capables de faire la différence entre les deux images. Cette méthode est donc approximative, mais universelle : n'importe quel dessin, même très compliqué, se décrit exactement comme un dessin simple.

Cette suite de 0 et de 1 s'appelle une suite binaire, ou parfois un mot binaire. Notre dessin se décrit ainsi avec 25 chiffres binaires, ou encore 25 bits ; sa longueur est 25.

Quel est le premier point clé ? *Convenir d'un standard pour toutes et tous.* Bien entendu, il faut que nous soyons tous d'accord pour décrire tous les dessins de la même façon ! Décider, comme nous l'avons fait dans l'exemple précédent, que le noir est représenté par 0 et le blanc par 1 et que les pixels se lisent de gauche à droite et de haut en bas. Le codage de l'information est avant tout une affaire de… convention.

Quel est le second point clé ? *Selon le codage choisi, l'information se manipule, selon les cas, plus ou moins facilement.* Bien entendu, nous aurions pu coder le dessin, par exemple, en commençant par le pixel du milieu, puis en énumérant les pixels en spirale. C'est une très bonne idée pour une machine mécanographique qui doit tracer ce dessin : au fil des bits, le stylo a toujours à tracer des

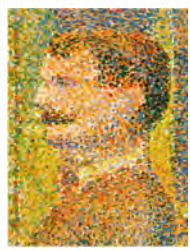
pixels consécutifs, donc évite de grands déplacements, d'une ligne à l'autre comme le codage proposé ci-avant l'implique. Mais c'est une idée un peu tordue, et si une personne doit tracer le dessin pixel à pixel à la main, elle sera plus embarrassée avec un codage en spirale. De même, si nous devons analyser ce dessin avec un logiciel, le codage facilitera ou non certains traitements, comme la détection de traits horizontaux.

Le codage de l'information est donc en lien profond et étroit avec la manipulation de cette information.

Décryptage de l'exemple. Que ce soit sous forme magistrale ou sous forme d'activité, il semble que la description d'une image soit le bon levier pour faire découvrir la notion de codage numérique. Le codage des nombres réclame plus de compétences calculatoires, et celui des sons et de la musique, bien qu'au cœur du quotidien des jeunes, se révèle finalement moins intuitif.

On peut facilement imaginer, pour le codage symbolique, un jeu à deux joueurs, où l'un doit donner, par écrit, des consignes précises sous forme rédigée puis sous forme symbolique, afin que l'autre reproduise tel carré, losange, rectangle avec un texte dedans, petit personnage stylisé, etc. L'intérêt est de montrer la nécessité de la rigueur et, surtout, de la standardisation : il faut se mettre d'accord sur le sens des mots. C'est à la fois un bel exercice de français et de structuration de la pensée.

Il est assez intéressant de relier le travail de pixélisation d'une image avec la démarche pointilliste du mouvement impressionniste, qui porte cette fulgurante intuition de la relation entre la couleur et la forme comme l'illustre ce dessin de Georges Seurat :



car, au-delà de l'aspect interdisciplinaire, cela montre que la démarche n'est pas exclusivement technologique, et qu'elle implique une certaine représentation du monde. Le lien entre codage numérique et représentation artistique est loin d'être anecdotique, et constitue un levier pour permettre d'accéder aux deux champs culturels.

Pour en savoir plus sur ce point précis : *L'art assisté par ordinateur (2011), Anne-Charlotte Philippe, http://interstices.info*

Manipuler une image

Une image est un tableau à deux dimensions de points lumineux, c'est-à-dire de pixels. Ici, chaque pixel a une valeur d'intensité entre 0, pour le « noir », et 255, pour le « blanc ».

Regardons, sous forme de trame d'exercice, quelles opérations nous aident à mieux comprendre les manipulations que nous pouvons faire sur ces images.

Changer chaque pixel Par exemple, pour chaque valeur d'intensité $I_{i,j}$ d'un pixel de coordonnée (i, j), où i est l'abscisse et j l'ordonnée, calculons :

$$I_{i,j} \leftarrow 255 - I_{i,j}$$

donc remplaçons les valeurs claires par des valeurs foncées et inversement. Nous venons tout simplement de faire une « image négative », les valeurs de luminance sont inversées.

De même :

– Augmenter ou baisser la *luminosité* d'une image revient à ajouter ou soustraire une valeur constante à la valeur de chaque pixel.

– Augmenter ou baisser le *contraste* d'une image revient à multiplier ou diviser par un gain constant la valeur de chaque pixel.

Ce sont des opérations très faciles à expérimenter lors d'un petit exercice de programmation.

Combiner deux images pixel à pixel Que se passe-t-il si on « additionne » deux images ? C'est-à-dire si l'on construit une nouvelle image dont la valeur de chaque pixel est la somme des valeurs des pixels de chaque image ? Si chaque image est dessinée sur une feuille de papier-calque, et si les couleurs sombres s'interprètent comme de la transparence, cela revient en quelque sorte à superposer les deux calques. C'est assez amusant à expérimenter numériquement.

Si l'on « soustrait » les deux images de manière similaire à ce qui a été fait pour les ajouter, une autre fonctionnalité émerge. Prenons deux images d'une scène où un objet bouge sur un fond fixe et soustrayons pixel à pixel. Pour les parties immobiles de l'image, les valeurs sont similaires. Pour les parties de l'image correspondant à l'objet mobile, les intensités des pixels vont varier. On a donc commencé à détecter ce qui bouge.

Ici les zones jaunes et bleutées correspondent aux variations d'intensité liées aux mouvements. Cet opérateur est à la base de tous les détecteurs visuels de mouvement, biologiques ou artificiels.

Modifier les pixels en fonction des voisins Considérons une image et modifions un pixel en recalculant sa valeur avec celles des pixels voisins. Par exemple, faisons la moyenne avec les quatre voisins de gauche, de droite, de dessus et de dessous :

$$I_{i,j} \leftarrow \frac{1}{4}(I_{i+1,j} + I_{i-1,j} + I_{i,j+1} + I_{i,j-1})$$

comme illustré ici. Nous sommes en train de faire un *lissage* de cette image en faisant cette moyenne, c'est-à-dire de gommer les petites variations parasites de luminance d'un pixel à l'autre, donc de rendre l'image un peu floue en ne gardant que les variations de lumière à plus grande échelle.

Si nous répétons cette opération sur toute l'image plusieurs fois de suite, l'image deviendra de plus en plus floue, les détails étant gommés progressivement du plus petit au plus grand. Il y a des mécanismes non linéaires plus sophistiqués mais basés sur le même principe de filtrage qui permettent de réduire le bruit de l'image, associé aux petites variations de l'intensité, tout en préservant les formes de cette image, associées aux contours, c'est-à-dire aux grandes variations de l'intensité.

Si, au lieu de moyenner l'information de luminance, on calcule plutôt ses variations d'un pixel à l'autre, ce sont les éléments de contraste, donc les contours de l'image, qui vont émerger :

$$I_{i,j} \leftarrow |I_{i+1,j} - I_{i-1,j}| + |I_{i,j+1} - I_{i,j-1}|$$

comme illustré ici, où nous donnons un exemple simple de tel opérateur.

D'autres opérateurs, dits de « morphologie mathématique », sont des calculs locaux non linéaires particulièrement utiles pour filtrer, segmenter et quantifier des images.

Ces opérateurs, à la manière de briques logicielles, se combinent et s'enchaînent. Ils sont la base des systèmes de vision industrielle, qui permettent de concevoir des applications qui détectent la présence d'un objet, calculent les caractéristiques d'un ou de plusieurs éléments de l'image, etc.

Le principe de la dichotomie

S'il est un mécanisme algorithmique général accessible aux lycéens qui illustre ce que peut être une notion abstraite d'informatique, c'est bien le principe de recherche dichotomique. Revoyons-le ici, dans le cadre d'une activité scolaire.

Exemple : devinons l'âge d'un élève

Devinons l'âge d'un élève du secondaire, entre 11 et 18 ans inclus.

Avec une première règle, nous ne répondons que « oui » ou « non ». De combien d'essais au maximum avons-nous besoin ?

Avec une seconde règle, nous répondons « oui », « plus jeune », ou « moins jeune ». De combien d'essais au maximum avons-nous besoin ?

Nous pouvons alors proposer un codage binaire des âges entre 11 et 18 ans, avec un nombre minimal de bits.

Décryptage de l'exemple Nous constatons que, dans ce cas facile à mentaliser, il est très simple de réaliser que, dans le cas oui/non, nous avons à poser 8 questions : « A-t-il 11 ans ? A-t-il 12 ans ? A-t-il 13 ans ? A-t-il 14 ans ? A-t-il 15 ans ? A-t-il 16 ans ? A-t-il 17 ans ? A-t-il 18 ans ? » avec le risque, si nous manquons de chance, de devoir poser les huit questions avant de connaître la solution, tandis que, dans le cas oui/plus/moins, une meilleure méthode peut être intuitée en demandant d'abord si l'élève a moins de 15 ans. Nous pourrons expliquer en détail que si la réponse est oui, alors nous savons que l'élève a entre 11 et 14 ans ; si la réponse est non, alors nous savons que l'élève a entre 15 et 18 ans. Dans les deux cas, nous sommes passés d'une fourchette de 8 ans à une fourchette de 4 ans.

On parle de dichotomie (couper en deux) pour ce processus de recherche où, à chaque étape, on coupe en deux parties l'espace de recherche.

Il est assez facile de montrer par un petit calcul de probabilité, ou de faire intuiter par le langage que, dans le cas oui/non, si l'âge peut être quelconque entre 11 et 18 – que la probabilité est uniformément répartie, donc –, peu importe l'ordre dans lequel nous posons les questions. Nous pouvons le faire par

exemple dans l'ordre croissant de 11 à 18. Avec de la chance, nous poserons une seule question, au pire sept questions, puisqu'au septième non… la huitième réponse sera forcément oui. Nous en déduirons ou intuiterons qu'en moyenne nous poserons quatre questions, nous en poserons : 1, 2, 3, 4, 5, 6 ou 7 avec une probabilité de 1/7, soit la moitié. Ce résultat étant évidemment général.

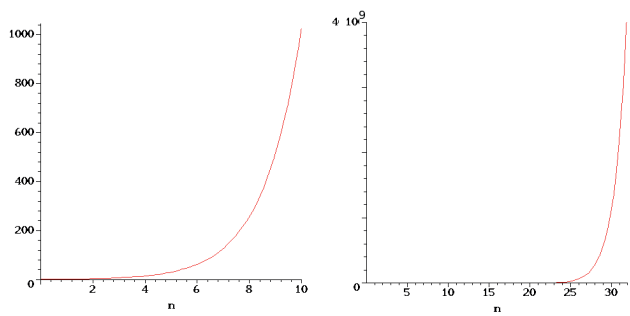
Si nous continuons à poser une question qui divise l'intervalle de recherche par deux, nous arrivons à une fourchette de deux ans, puis, à la troisième question, à une fourchette d'un an… et nous avons trouvé la solution, l'âge de l'élève. Cela en exactement trois questions au lieu de huit, et il faut alors faire calculer à l'élève que :

– s'il y avait le choix parmi 1, alors 0 question aurait suffi !
– s'il y avait le choix parmi 2, alors 1 question aurait suffi.

etc. jusqu'à 16, puis lui faire intuiter qu'une question de plus permet de multiplier par deux le nombre de choix à discriminer et lui proposer quelques ordres de grandeur.

– S'il y avait le choix parmi 256, alors 8 questions auraient suffi.
– S'il y avait le choix parmi environ mille, jusqu'à 1024 précisément, mais qu'importe, alors 10 questions auraient suffi.
– S'il y avait le choix parmi environ un million, jusqu'à 1048576 précisément, alors 20 questions auraient suffi.
– S'il y avait le choix parmi environ quatre milliards, jusqu'à 4294967296 précisément, alors 32 questions auraient suffi.

Nous sommes évidemment en train de faire découvrir le plus concrètement du monde la notion d'exponentielle et de logarithme :



Ce calcul est du reste directement lié au codage des entiers positifs, et nous pouvons facilement calculer combien de bits sont nécessaires pour les coder :

Entiers	0 à 1	0 à 3	0 à 7	0 à 15	0 à 255	0 à 1023	0 à environ 4 milliards	0 à plus de 1.8×10^{20}
Codés sur	1 bit	2 bits	3 bits	4 bits	8 bits	10 bits	32 bits	64 bits

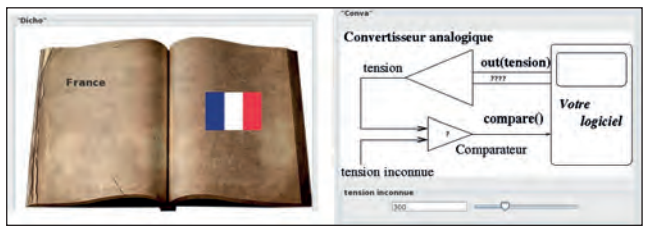
Nos ordinateurs utilisent aujourd'hui souvent 32 ou 64 bits pour coder les nombres entiers, ils peuvent donc coder directement des nombres très grands et, si besoin était, en utilisant plus de bits… des nombres vertigineusement grands !

Pour chercher quelqu'un parmi les 60-70 millions de Français-se-s, dans le cas oui/non, nous aurons en moyenne à poser 30-35 millions de questions et seulement 26 dans le cas oui/plus/moins. Il faut faire prendre conscience de la différence.

Ce qu'il faut faire voir ici est que la différence entre le cas oui/non et le cas oui/plus/moins est liée à la structure de donnée. Si les Français-se-s sont « en vrac » sans aucun ordre, alors impossible de définir un plus/moins et il faut poser les questions une à une. Mais dans le cas où il y a un ordre total, par exemple à la suite d'un tri par ordre alphabétique, alors le mécanisme de dichotomie est possible. Le lien entre algorithme et codage de l'information est bien illustré ici.

Ce qu'il faut rendre explicite ici, c'est que nous avons divisé l'intervalle de recherche par deux, donc que nous avons gagné un atome d'information, un bit d'information. Le fait que les informations soient triées par ordre croissant permet de les associer à un codage numérique binaire, « 000 » pour 11, « 001 » pour 12, etc., *isomorphe* aux 8 âges et que nous cherchons finalement un code binaire parmi 8. Du coup, chaque question dichotomique correspond à deviner un bit en commençant par celui de poids fort.

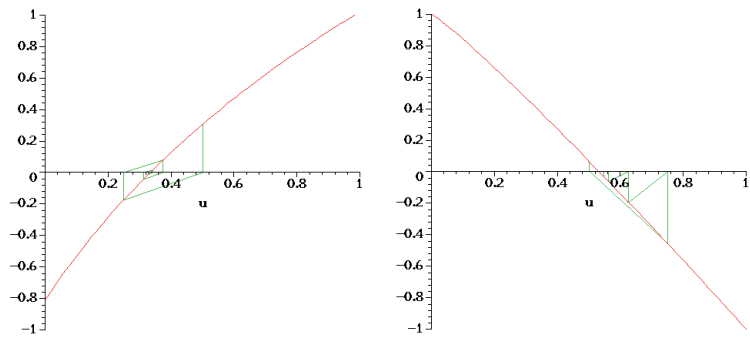
Généricité du principe algorithmique Qu'il s'agisse de trouver un mot dans un dictionnaire ou de considérer un tout autre problème, tel que convertir une tension électrique en valeur numérique, le même principe va s'appliquer. Pour « deviner » la valeur d'une tension électrique continue, un ordinateur numérique doit en général comparer cette valeur à une valeur de référence qu'il va produire en sortie, pour de proche en proche cerner cette valeur, comme le montre le diagramme ci-après :



Là encore, la façon de fonctionner de ces convertisseurs à approximations successives est de procéder de manière dichotomique en divisant l'espace de

recherche par deux à chaque étape. Cela permet d'atteindre rapidement les précisions requises, le millième en 10 étapes, le millionième en 20, comme vu précédemment.

Tous ces problèmes sont finalement reliés au problème mathématique suivant : résoudre une équation de la forme $f(x) = 0$, $x_{min} < x < x_{max}$, où f est une fonction continue monotone dans l'intervalle $]x_{min}, x_{max}[$, donc a au plus une solution, puisque bijection vers un intervalle réel. Si elle change de signe dans cet intervalle, c'est-à-dire $f_{(xmin)}f_{(xmax)} < 0$, elle a une solution unique. Et le même mécanisme de dichotomie permet de résoudre le problème comme illustré sur l'exemple suivant en coupant l'intervalle de recherche en deux, soit en gagnant un bit, à chaque étape :



Les courbes vertes correspondent au test des valeurs positives/négatives des fonctions rouges jusqu'à se rapprocher des zéros recherchés.

Le mécanisme de dichotomie, vu du point de vue du codage, est un bon levier didactique pour faire comprendre les notions essentielles et complémentaires revues ici.

Compléments

La vision probabiliste des choses : un rappel

Si la notion d'information est abordée avec une vision probabiliste, on considère alors que l'on accède aux événements observés par la mesure de leur probabilité d'occurrence. Un événement n'est pas « vrai » ou « faux », mais plus ou moins *probable*.

Pour définir correctement une « tribu » d'événements et leur probabilité nous avons besoin que :
– l'ensemble vide ∅ donc impossible soit un événement, il sera de probabilité $p(\emptyset) = 0$,
– l'ensemble de tout ce qui est possible Ω soit un événement, il sera de probabilité $p(\Omega) = 1$, tandis que les événements E sont des sous-ensembles de tout ce qui est possible $E \subseteq \Omega$.

On ne dit plus ce qui est vrai ou faux sur le monde, on donne une valeur numérique de probabilité qu'un événement se produise. Deux visions différentes du monde se distingueront par ce qui est considéré comme un événement et par la probabilité d'un tel événement.

Il faut pouvoir combiner de manière logique les événements. C'est-à-dire que :
– pour un événement E, son contraire *non* E, c'est-à-dire son complémentaire parmi tout ce qui est possible $\neg E = \Omega - E$ est un événement de probabilité $p(\neg E) = 1 - p(E)$;
– pour un ensemble dénombrable d'événements $\{\ldots E_i, \ldots\}$, leur disjonction, c'est-à-dire leur réunion $E\cup = \cup_i E_i$ soit un événement, qui correspond au fait que E_1 ou E_2 *ou* … se produise
et si les événements sont *incompatibles* – ne peuvent se produire ensemble –, c'est-à-dire si $E_i \cap E_j = \emptyset, \forall i, j$, alors la probabilité de la disjonction est une somme $p(E\cup) = \Sigma_i p(E_i)$;
– pour un ensemble d'événements $\{\ldots E_i, \ldots\}$, leur conjonction, c'est-à-dire leur intersection $E\cap = \cap_i E_i$ soit un événement, qui correspond au fait que E_1 et E_2 *et* … se produisent,
et on posera la définition que les événements sont *indépendants*, si et seulement si la probabilité de la conjonction est un produit $p(E\cap) = \Pi_i p(E_i)$;
– tandis qu'un événement E_1 *implique* E_2 si $E_2 \subseteq E_1$; ce qui implique $p(E_1) \geq p(E_2)$. La réciproque est bien sûr fausse.

À partir de ces briques de base, toutes les propriétés logiques s'expriment en termes d'opérations ensemblistes sur les événements, avec les probabilités qui en découlent.

Ce que nous mettons en avant ici en faisant ce bref rappel, c'est qu'il y a substitution des opérations logiques de base *et*, *ou*, *non* par des calculs sur les probabilités, tandis que l'on fait un lien profond entre une vision ensembliste des événements et une vision logique, ce qui fonde la théorie moderne des probabilités.

Bref, la vérité est relative et ne se déduit plus : elle se mesure et se calcule.

Il faut noter que ces définitions, qui semblent naturelles, sont en fait pleines de subtilités techniques : la notion d'indépendance est une définition sur les probabilités, celle d'incompatibilité une notion ensembliste avec une contrainte sur les probabilités, certains éléments sont des axiomes tandis que d'autres s'en déduisent. Tous les sous-ensembles ne sont pas nécessairement des événements, mais cette subtilité intervient dans des espaces infinis continus, les définitions étant pertinentes et valides y compris dans ce cas.

Dans le cas fini ou dénombrable développé ici, il faut juste considérer les valeurs de probabilités $p(e_n)$ sur chaque événement « atomique » ou valeur e_1, … e_n , … tandis que les probabilités sur chaque événement E_i s'en déduisent additivement $P(E_i) = \Sigma_{e_n \varepsilon E_i} p(e_n)$. On notera N le nombre de valeurs.

De la maximisation d'entropie

Pour mettre l'estimation par maximum d'entropie en équation, nous posons que nous disposons de quelques fonctions $\mu_1(e_n), \ldots, \mu_M(e_n)$ qui associent aux probabilités $p(e_1), \ldots, p(e_N)$, des valeurs moyennes de la forme $\Sigma_n p(e_n)\mu_1(e_n) = v_1, \ldots, \Sigma_n p(e_n)\mu_M(e_n) = v_M$. Ce sont des valeurs pondérées par les probabilités $p(e_n)$ puisque nous n'accédons pas à une valeur aléatoire, mais uniquement à sa valeur moyenne. Prendre celle d'entropie maximale, en équation, cela s'écrit :

$$\max_{p(e1),\ldots,p(e_N)} -\sum_n p(e_n)\log_2(p(e_n)) \; ; \forall m \sum_n p(e_n)\mu m(e_n) = v_m$$

et un peu de calcul nous permet d'écrire la solution sous la forme :

$$p(e_n) = \frac{1}{Z} e^{\sum_m \lambda_m \mu_m(e_n)}$$

où la constante de normalisation $Z = \sum_n e^{\sum_m \lambda_m \mu_m(e_n)}$ permet que la somme des probabilités $\Sigma_n p(e_n) = 1$ soit bien normalisée, tandis que les paramètres $\lambda_1, \ldots \lambda_M$ se calculent numériquement à partir des équations de mesure. *e*

Estimation d'information par optimisation

Rendons explicite une méthodologie profonde utilisée pour trouver la solution d'entropie maximale : nous avions des équations de mesure et nous ne savions pas si ces équations avaient une seule solution, plusieurs solutions ou n'avaient aucune solution exacte, mais uniquement approchées. Et cela ne nous a pas empêchés de trouver… la « bonne solution » ! Que s'est-il passé ? Nous avons tout simplement changé de point de vue.

Prenons des équations $z_1(x_1, x_2, \ldots, x_N) = 0$, $z_2(x_1, x_2, \ldots, x_N) = 0$, … $z_M(x_1, x_2, \ldots x_N) = 0$. Nous savons que s'il y a autant d'équations que d'incon-

nues et si les équations sont linéaires et que le système d'équation n'est pas dégénéré, il y a une unique solution, les lycéens en calculent dans le cas de une ou deux équations. S'il y a une équation, mais de degré deux, alors nous pouvons avoir une, deux ou zéro solutions. Les choses se gâtent. Et elles se gâtent vite. Par exemple, N équations non linéaires avec des polynômes ou des fractions rationnelles de degré D peuvent avoir D^N solutions – ce qui est vite énorme – réelles ou toutes complexes, voire à l'infini, ou encore numériquement instables, etc. Et bien entendu il n'y a aucune formule exacte sauf dans quelques cas d'école.

Le changement de point de vue est alors le suivant : (1) au lieu de chercher « la » solution, nous *cherchons une solution proche d'une solution raisonnable* $(\overline{x}_1, \ldots \overline{x}_N)$ et (2) au lieu de résoudre ces équations nous *minimisons l'erreur de ces équations.* Cela se met en équation de la manière suivante : chercher à minimiser un critère de coût

$$\min_{x_1,\ldots x_N} \underbrace{\sum_n \frac{|x_i - \overline{x}_i|}{V_n}}_{\text{plus proche solution}} + \underbrace{\sum_m \frac{|z_m(x_1, \ldots x_N)|}{\varepsilon_m}}_{\text{qui minimise l'erreur}}$$

Détaillons.

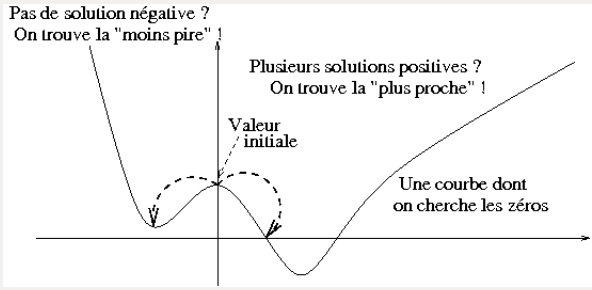
1. Le premier terme est d'autant plus petit que la solution $(x_1, \ldots x_N)$ est proche de notre solution initiale raisonnable $(\overline{x}_1, \ldots \overline{x}_N)$, puisque c'est la somme des valeurs absolues des erreurs, il vaut zéro si nous sommes à la solution raisonnable, plus si nous nous en éloignons. Ici, V_n permet de pondérer la précision.

2. Chaque terme de droite $\frac{|z_m(x1, \ldots x_N)|}{\varepsilon_m}$, avec $\varepsilon_m > 0$ vaut zéro.

si l'équation est vérifiée, tandis qu'une pénalité positive apparaît sinon ; en ajustant ε_m à une plus ou moins grande valeur, ce terme de pénalité est ajusté.

Nous prendrons des valeurs minimales de ε_m si nous voulons forcer l'équation à être vérifiée – un théorème du multiplicateur de Lagrange garantit que chercher les valeurs minimales de ε_m permet, sous de bonnes conditions, d'annuler l'équation $z_m(x_1, \ldots x_N) = 0$ – ou une valeur de ε_m correspondant à la précision de la mesure.

Nous illustrons l'idée ci-dessous, avec une courbe qui n'a pas de racine négative, et plusieurs racines positives. Si nous cherchons une solution négative, la moins mauvaise est celle proche de zéro, et si nous cherchons une solution positive, la meilleure est celle la plus proche de notre valeur initiale.



Qu'avons-nous gagné ici ? En terme de *spécification* nous avons bien défini ce que nous cherchons dans tous les cas. Nous voyons que c'est en ajoutant la notion de solution initiale raisonnable que nous éludons le problème des solutions multiples et que c'est en remplaçant « résoudre » $z_m(x_1,\ldots x_N) = 0$ par minimiser $\frac{|z_m(x_1, \ldots x_N)|}{\varepsilon_m}$ que nous contournons le problème d'équations inexactes ou redondantes.

Qu'avons-nous gagné d'autre ici ? En terme d'*implémentation*, il est beaucoup plus facile de *minimiser* un critère de coût que de résoudre une équation.

Implémentation du paradigme Il suffit de chercher à chaque étape à améliorer un peu la solution actuelle, en essayant quelles solutions voisines sont meilleures, et de réitérer jusqu'à ce que l'information à gagner devienne négligeable. Des résultats mathématiques permettent de savoir quand un tel mécanisme marche et avec quelle performance.

Nous avons donc non seulement un problème bien posé, mais des classes d'algorithmes numériques généraux pour le résoudre efficacement et on peut dire que la grande majorité des algorithmes non linéaires de traitement des signaux de parole ou image, ou d'autres mesures physiques, de calcul de boucles sensori-motrices dans les systèmes artificiels, par exemple dans les robots, sont construits sur ces principes de base. Ne nous trompons pas : les techniques numériques qui marchent à grande échelle sont bigrement complexes, mais les principes généraux sont là.

De plus, à chaque étape, si nous avons gagné quelque chose : trouvé une valeur « moins mauvaise » que la précédente, alors nous avançons vers la solution. En termes de système interactif, les conséquences sont énormes : l'utilisateur ou le robot n'a pas à attendre que… le système trouve une solution, ou non ! À chaque instant, il peut accéder à la moins mauvaise des

solutions, donc prendre une décision sans attendre. Par ailleurs, l'utilisateur peut aider le mécanisme en lui fournissant une solution initiale, ou plusieurs solutions initiales en concurrence, que le calcul va raffiner. Nous sommes au cœur d'une réalité générale des mathématiques du numérique : il est souvent difficile de résoudre un problème global, par exemple trouver une trajectoire, mais beaucoup plus facile de raffiner par le calcul une solution approximative. C'est un des piliers de la conception des systèmes numériques et nous en avons donné les clés ici.

Pour quelques illustrations sur ces points précis : *Pourquoi ne pas confier au hasard ce qui est trop compliqué à estimer ? (2007) Thierry Viéville, http://interstices.info/hasard-estimation* ou *Une solution au problème de la génération de trajectoires (2006) Thierry Viéville, http://interstices.info/generation-trajectoires.*

Contenu en information et neuroscience computationnelle

La complexité d'un cerveau, en termes de nombre d'états, semble être de l'ordre de grandeur du milliard d'ordinateurs que nous avons actuellement dans le monde. Cette complexité du système nerveux, que nous voyons aujourd'hui comme une « machine à traiter de l'information », conduit à deux visions.

Selon la première, le cerveau ne serait rien d'autre qu'une « machine programmable » mais d'une complexité énorme. La seconde met en avant ses propriétés complètement opposées à un processeur d'ordinateur. Le traitement dans le cerveau est complètement distribué, chaque groupe de neurones mettant localement à jour son état, avec émergence d'une fonctionnalité globale de cette population de traitement locaux. *A contrario*, le parallélisme au niveau des processeurs reste aujourd'hui limité et les modèles de machines discutés ici sont intrinsèquement séquentiels et centralisés. De plus, tandis que le processeur effectue ses opérations à une durée inférieure au milliardième de seconde, chaque neurone ne met à jour son état qu'à l'échelle de la milliseconde. Par ailleurs, tandis que le processeur calcule sans erreur – et surtout sans erreur qui ne soit pas fatale ! –, le système nerveux a un comportement intrinsèquement stochastique : les valeurs ne sont pas exactes, mais fluctuent sans que cette fluctuation ne prenne de sens déterministe, ce qui ne gêne pas les traitements, voire aide à explorer de nouvelles solutions lors de certains traitements. Cette profonde fracture entre ces propriétés conduit à penser que c'est dans la confrontation, et non l'assimilation, du cerveau à un processus calculable que se situe la vision scientifique la plus fructueuse.

Pour aller plus loin

Une présentation générale des grands concepts et enjeux des sciences numériques : Gérard Berry, *Pourquoi et comment le monde devient numérique*, Leçon inaugurale au Collège de France (2007). http://www.college-de-france.fr/default/EN/all/inn_te

Pour en savoir plus sur les théories de l'information : Jean-Paul Delahaye, *Théories et théorie de l'information*,)i(nterstices, (2009) http://interstices.info/information, Jean-Paul Delahaye, *Information, complexité et hasard*, Hermès (1999), Jean-Paul Delahaye, *Émergence et complexité de Kolmogorov*, et Jean-Paul Delahaye, *Émergence : de la fascination à la compréhension*, http://www.science-inter.com/Pages%20AEIS/programme.htm auxquels ce chapitre emprunte largement.

Plus d'informations techniques sur les formats de données dans Wikipédia : *Format_de_données*

http://fr.wikipedia.org/wiki/Format_de_données

sur fr.wikipedia.org donc *Image_numérique*

http://fr.wikipedia.org/wiki/Image_numérique

sur Wikipédia pour la description détaillée du codage des couleurs. Pour la description des fichiers *PNG*

http://fr.wikipedia.org/wiki/Portable_Network_Graphics

avec *GIF*

http://fr.wikipedia.org/wiki/Graphics_Interchange_Format

et *JPEG*

http://fr.wikipedia.org/wiki/JPEG

En savoir plus sur les expressions régulières *Expression Régulière*

http://fr.wikipedia.org/wiki/Expression_régulière

dans *Wikipédia*

En savoir plus sur les liens entre cerveau et ordinateur *Le cerveau, un ordinateur ?*

http://interstices.info/cerveau-ordinateur

(2008) Aurélien Liarte, Yves Geffroy,)i(nterstices.

En savoir plus sur les calculs sur ordinateur *Pourquoi mon ordinateur calcule-t-il faux ?*

http://interstices.info/a-propos-calcul-ordinateurs

(2008) Sylvie Boldo,)i(nterstices

En savoir plus sur l'histoire des sciences de l'information

http://www-sop.inria.fr/science-participative.film

Langages et programmation

Notre parcours se poursuit avec la notion de langage de programmation. Apprendre un premier langage de programmation est une étape importante dans la découverte de l'informatique : c'est le moment où l'on devient autonome, en étant capable de construire soi-même quelque chose. Apprendre un langage de programmation demande de comprendre les constructions de ce langage, et, surtout, de savoir les utiliser. Et, en programmation, c'est beaucoup en forgeant que l'on devient forgeron. Ce chapitre propose donc une importante série d'exercices de programmation, qui permettent de mettre en pratique les notions apprises. Le langage choisi comme exemple ici est Java, mais les notions introduites sont universelles. Elles se transposent sans trop de difficultés à presque tous les langages. À la fin du chapitre, nous étendons la problématique des langages de programmation aux langages formels en général, en contrastant le langage Java avec le langage XHTML, qui est un langage formel, mais pas un langage de programmation.

Cours

Le noyau impératif

Affectation, déclaration, séquence, test et boucle

La plupart des langages de programmation comportent, parmi d'autres, cinq constructions : l'affectation, la déclaration de variable, la séquence, le test et la boucle. Ces constructions forment le *noyau impératif* de ces langages.

L'*affectation* est une construction qui permet de former une *instruction* avec une variable `x` et une expression `t`. En Java, cette instruction s'écrit `x = t;`, par exemple, `x = y + 3;`. Les *variables* sont des identificateurs, c'est-à-dire des mots formés de une ou plusieurs lettres. Les *expressions* sont formées à

partir des variables et des constantes avec des opérations, comme, par exemple, `+`, `-`, `*`, `/` et `%`, ces deux dernières opérations étant respectivement les quotient et reste de la division euclidienne.

Pour expliquer ce qui se passe quand on exécute l'instruction `x = t;` on doit supposer qu'il y a, dans un recoin de la mémoire d'un ordinateur, une case appelée `x`. Exécuter l'instruction `x = t;` consiste alors à remplir cette case avec la *valeur* de l'expression `t`. La valeur contenue antérieurement dans la case `x` est effacée. Si l'expression `t` est une constante, par exemple `3`, sa valeur est cette même constante. Si c'est une expression sans variables, comme `3 + 4`, sa valeur s'obtient en effectuant quelques opérations arithmétiques, ici une addition. Si l'expression `t` contient des variables, alors il faut aller chercher les valeurs correspondant à ces variables dans les cases de la mémoire de l'ordinateur. L'ensemble des contenus des cases de la mémoire de l'ordinateur s'appelle un *état*.

Il est important de distinguer les expressions, comme `x + 2`, des instructions, comme `y = x + 2;`. Une expression *se calcule*, une instruction *s'exécute*.

Dans ces exemples, les valeurs des expressions sont des nombres entiers relatifs. En fait, dans les langages de programmation, la valeur d'une expression ne peut pas être un entier arbitraire et une telle valeur appartient toujours à un intervalle fini, en Java à l'intervalle $[-2^{31}, 2^{31} - 1]$ – voir le premier chapitre.

Avant de pouvoir affecter une variable `x`, il faut la déclarer, c'est-à-dire associer le nom `x` à une case de la mémoire de l'ordinateur. La *déclaration de variable* est une construction qui permet de former une instruction à partir d'un type, d'une variable, d'une expression et d'une instruction. En Java, cette instruction s'écrit `{int x = t; p}` où `p` est une instruction. Par exemple, `{int x = 4; x = x + 1;}` déclare une variable `x` de type `int` – entier –, de valeur initiale `4`, utilisable dans l'instruction `x = x + 1;`. Plus généralement, la variable `x` peut ensuite être utilisée dans l'instruction `p`, qui s'appelle la *portée* de la variable `x`.

Outre le type `int`, il y a, en Java, trois autres types d'entiers : `byte`, `short` et `long` qui correspondent aux intervalles $[-2^7, 2^7 - 1]$, $[-2^{15}, 2^{15} - 1]$ et $[-2^{63}, 2^{63} - 1]$. Il y a aussi, en Java, d'autres *types de données scalaires* : deux types, `float` et `double`, pour les nombres à virgule, qui s'écrivent en notation « scientifique », par exemple `3.14159`, `666` ou `6.02E23`, un type `boolean` pour les booléens qui s'écrivent `false` et `true` et un type `char` pour les caractères qui s'écrivent entre apostrophes, par exemple `'b'`. Enfin, il y a des *types composites*, comme les tableaux et les chaînes de caractères, sur lesquels nous reviendrons ci-après. Chaque type est muni d'opérations qui servent à construire des expressions de ce type.

Pour déclarer une variable d'un type `T`, on doit remplacer le type `int` par `T`. La forme générale d'une déclaration est donc `{T x = t; p}`. Nous avons choisi dans ce chapitre d'initialiser systématiquement les variables au moment de leur déclaration.

La *séquence* est une construction qui permet de former une instruction à partir de deux instructions p_1 et p_2. En Java, cette instruction s'écrit $\{p_1\ p_2\}$, par exemple `{x = 1; y = 2;}`. Dans cette instruction, le premier point-virgule appartient à la première affectation `x = 1;` et non à la séquence elle-même. Par convention, l'instruction $\{p_1\ \{p_2\ \{\ \ldots\ p_n\}\ \ldots\}\}$ s'écrit aussi $\{p_1\ p_2\ \ldots\ p_n\}$. Pour exécuter l'instruction $\{p_1\ p_2\}$ dans un état `s`, on exécute d'abord p_1 dans l'état `s`, ce qui produit un état `s'`, puis on exécute p_2 dans l'état `s'`.

Le *test* est une construction qui permet de former une instruction à partir d'une expression booléenne `b` et de deux instructions p_1 et p_2. En Java, cette instruction s'écrit `if(b)`p_1 `else` p_2, par exemple `if(x == 1)y = 2; else y = 5;`. Pour exécuter l'instruction `if (b)` p_1 `else` p_2 dans un état `s`, on commence par calculer la valeur de l'expression `b` dans l'état `s`, puis, selon que cette valeur est `true` ou `false`, on exécute p_1 ou p_2 dans l'état `s`.

La *boucle* est une construction qui permet de former une instruction à partir d'une expression booléenne `b` et d'une instruction `p`. En Java, cette instruction s'écrit `while (b) p`, par exemple `while (x <= 1000) x = x * 2;`. Pour exécuter l'instruction `while (b) p` dans un état `s`, on commence par calculer la valeur de `b` dans l'état `s`. Si cette valeur est `false`, on a terminé. Sinon, on exécute `p`, puis on recalcule `b` dans le nouvel état. Si cette valeur est `false`, on a terminé. Sinon, on exécute `p`, puis on recalcule `b` dans le nouvel état. Si cette valeur est `false`, on a terminé. Sinon, on exécute `p`, puis on recalcule `b` dans le nouvel état… Et ce processus se poursuit jusqu'à ce que la valeur de `b` soit `false`.

Avec cette construction apparaît une possibilité de *non-terminaison*. En effet, si l'expression booléenne `b` vaut éternellement `true`, alors l'instruction `p` s'exécute à l'infini et l'exécution de l'instruction `while (b) p` ne termine pas. C'est par exemple le cas de l'instruction

```
int x = 1;
while (x >= 0) {x = 3;}
```

En introduisant une instruction fictive `skip;` dont l'exécution ne fait rigoureusement rien, on peut définir l'instruction `while (b) p` comme une abréviation pour l'instruction infinie

```
if (b) {p if (b) {p if (b) {p if (b) …
                    else skip;}
               else skip;}
```

```
        else skip;}
    else skip;
```

La boucle est donc l'une des multiples notations qui permettent d'exprimer un objet infini avec une expression finie. Et le fait qu'une boucle puisse ne pas terminer est une conséquence du fait que c'est un objet infini.

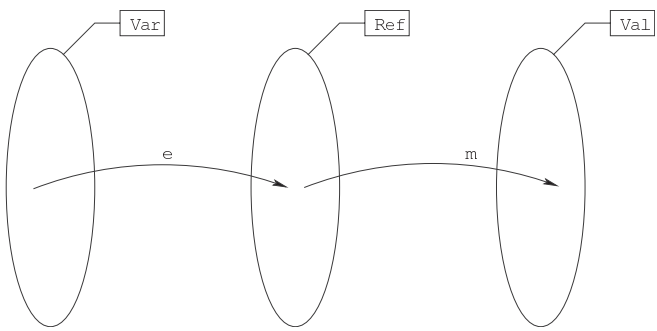
La boucle `for` est juste une notation plus pratique pour un cas particulier de la boucle `while`. L'instruction `for (`p_1`; b; `p_2`)` p_3 est une abréviation pour l'instruction p_1`; while (b) {`p_3 p_2`;}`. Par exemple, l'instruction
`for (i = 1; i <= 10; i = i + 1) s = s + i;`
ajoute successivement à `s` les entiers de `1` à `10`.

La sémantique du noyau impératif

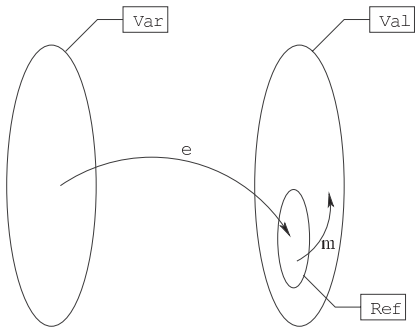
On peut, comme nous l'avons fait ci-avant, expliquer par une phrase en français la *sémantique* de chaque instruction, c'est-à-dire ce qui se passe quand on exécute cette instruction. Toutefois, ces explications deviendront vite compliquées et imprécises, quand nous introduirons d'autres constructions des langages de programmation, en particulier les fonctions. Nous allons donc définir plus précisément la sémantique de ces instructions, et tout d'abord une notion d'état, puis la manière dont l'exécution d'une instruction transforme cet état.

Pour définir la notion d'état, on pose un ensemble infini `Var` dont les éléments sont appelés *variables*. Et on définit un ensemble `Val` de *valeurs* qui contient les nombres entiers, les booléens, etc. Un *état* est une fonction d'une partie finie de l'ensemble `Var` dans l'ensemble `Val`.

Il sera utile, dans la suite de ce chapitre, de considérer cette fonction comme la composée de deux fonctions de domaine fini : la première, l'*environnement,* d'une partie finie de l'ensemble `Var` dans un ensemble intermédiaire `Ref`, dont les éléments sont appelés *références*, et la seconde, la *mémoire*, d'une partie finie de l'ensemble `Ref` dans l'ensemble `Val`.



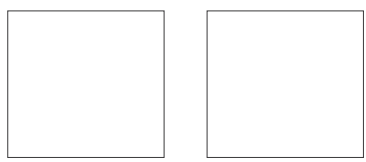
Il sera également utile de définir l'ensemble `Ref` comme un sous-ensemble de `Val`.



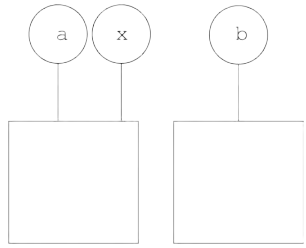
Cela mène à poser deux ensembles infinis `Var` et `Ref` puis à définir un ensemble `Val` qui contient les nombres entiers, les booléens, etc. et les éléments de l'ensemble `Ref`. L'ensemble des *environnements* se définit alors comme l'ensemble des fonctions d'une partie finie de l'ensemble `Var` dans l'ensemble `Ref` et l'ensemble des *mémoires* comme l'ensemble des fonctions d'une partie finie de l'ensemble `Ref` dans l'ensemble `Val`. On écrit $[x = r_1, y = r_2]$ l'environnement qui associe la référence r_1 à la variable `x` et la référence r_2 à la variable `y`. De même, on écrit $[r_1 = 5, r_2 = 7]$ la mémoire qui associe la valeur `5` à la référence r_1 et la valeur `7` à la référence r_2.

On définit une fonction de mise à jour d'un environnement notée `+` telle que l'environnement `e + [x = r]` soit la fonction coïncidant partout avec `e` sauf en `x` où elle vaut `r`. De même, on définit une fonction de mise à jour d'une mémoire telle que la mémoire `m + [r = V]` soit la fonction coïncidant partout avec `m` sauf en `r` où elle vaut `V`.

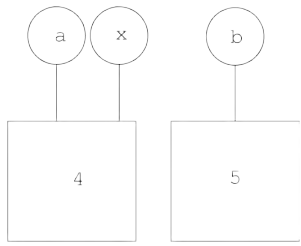
On est souvent amené à représenter les états graphiquement. Chaque référence est représentée par une case dessinée sur le plan. Deux cases situées à des endroits différents du plan représentent toujours des références différentes.



Puis, on représente l'environnement en ajoutant à certaines cases une ou plusieurs étiquettes avec un nom.



Si chaque étiquette est associée à une référence unique, rien n'empêche, en principe, deux étiquettes d'être associées à la même référence. Enfin, on représente la mémoire en remplissant chaque case avec une valeur.



À nouveau, rien n'empêche deux cases de contenir la même valeur.

Pour définir ce qui se passe quand on calcule une expression, on définit alors la fonction Θ qui associe une valeur à chaque triplet formé d'une expression, d'un environnement et d'une mémoire. Par exemple, `Θ(x + 3,[x = r1, y = r2],[r1 = 5,r2 = 6])= 8.`

– `Θ(x,e,m) = m(e(x))`, par exemple si `e = [x = r]` et `m = [r = 5]`, alors

`e(x) = r` et `m(e(x)) = 5`,

– `Θ(c,e,m) = c`, si `c` est une constante, comme `4`, `true`, etc.

– `Θ(t + u,e,m) = Θ(t,e,m) + Θ(u,e,m)`,

– et de même pour les autres opérations.

Enfin, pour définir ce qui se passe quand on exécute une instruction, on définit la fonction Σ qui associe une mémoire à chaque triplet formé d'une instruction, d'un environnement et d'une mémoire. Par exemple, `Σ(x = x + 1;,[x = r],[r = 5]) = [r = 6].` Cette fonction est partielle et elle n'est pas définie sur le triplet `(p, e, m)` si l'instruction `p` ne termine pas quand on l'exécute dans l'environnement `e` et la mémoire `m`, ou si cette exécution produit une erreur.

– Quand l'instruction `p` est une déclaration de variable de la forme `{T x = t; q}`, la fonction Σ se définit ainsi

Σ({T x = t; q},e,m) = Σ(q,e + [x = r],m + [r = Θ(t,e,m)])
où r est une référence quelconque qui n'apparaît pas dans e et m.
– Quand l'instruction p est une affectation de la forme x = t;, cette fonction se définit ainsi

Σ(x = t;,e,m) = m + [e(x) = Θ(t,e,m)]

ce qui traduit la phrase « Exécuter l'instruction x = t; consiste à remplir la case x avec la valeur de l'expression t ».
– Quand l'instruction p est une séquence de la forme {p1 p2}, la fonction Σ se définit ainsi

Σ({p1 p2 },e,m) = Σ(p2 ,e,Σ(p1 ,e,m))

– Quand l'instruction p est un test de la forme if (b) p1 else p2, la fonction Σ se définit ainsi. Si Θ(b,e,m) = true alors

Σ(if(b) p_1 else p_2,e,m) = Σ(p_1,e,m)

Si Θ(b,e,m) = false alors

Σ(if(b) p_1 else p_2,e,m)= Σ(p_2,e,m)

– Venons-en maintenant au cas où l'instruction p est une boucle de la forme while (b) q. Nous avons vu qu'en introduisant une instruction fictive skip; telle que Σ(skip;,e,m) = m, on peut définir l'instruction while (b) q comme une abréviation pour une instruction infinie. Pour traiter de tels objets infinis, on cherche souvent à les approcher par des objets finis. Ainsi, en introduisant une instruction fictive giveup; telle que la fonction Σ ne soit jamais définie en (giveup;,e,m), on peut définir une suite d'approximations finies de l'instruction while (b) q.

p_0 = if (b) giveup; else skip;
p_1 = if (b) {q if (b) giveup; else skip;} else skip;
…
p_{n+1} = if (b) {q p_n} else skip;

L'instruction p_n tente d'exécuter l'instruction while (b) q en faisant au maximum n tours de boucle. Si, après n tours, elle n'a pas terminé, alors elle abandonne. Si la fonction Σ n'est définie en aucun triplet (p_n,e,m), cela signifie que, quel que soit le nombre de tours de boucle que l'on s'accorde, le programme ne termine pas. Dans ce cas, on pose que Σ n'est pas définie en (while (b) q,e,m). Sinon, soit n le plus petit entier tel que Σ soit définie en (p_n,e,m), on pose Σ(while (b) q,e,m) = Σ(p_n,e,m).

Les constructions d'entrée/sortie

Les constructions d'entrée d'un langage permettent de lire des données sur un périphérique, par exemple un clavier, et les constructions de sortie permettent d'afficher des données sur un périphérique, par exemple un écran.

Les constructions d'entrée en Java sont assez complexes, nous utiliserons donc une extension de Java, la classe `Isn` – le ficher `Isn.java` est disponible sur le site de ressources pédagogiques SILO, il suffit de le placer dans le répertoire courant pour pouvoir utiliser les instructions décrites ici. L'évaluation de l'expression `Isn.readInt()` attend que l'utilisateur tape un nombre entier au clavier et produise ce nombre comme valeur. Une utilisation typique est `n = Isn.readInt();`. De manière similaire, la construction `Isn.readDouble` permet de lire un nombre à virgule et la construction `Isn.readChar` un caractère.

L'exécution de l'instruction `System.out.print(t);` affiche à l'écran la valeur de l'expression `t`. L'exécution de l'instruction `System.out.println();` passe à la ligne. L'exécution de l'instruction `System.out.println(t);` affiche à l'écran la valeur de l'expression `t`, puis passe à la ligne.

La notion de fonction

Isoler une instruction

La manière la plus simple d'introduire la notion de fonction est de voir cette construction comme un moyen d'éviter les redondances dans un programme. Si une même instruction revient plusieurs fois dans un programme, on peut l'isoler dans une *fonction* et la remplacer dans le programme par un *appel* à cette fonction. L'instruction que l'on isole dans une fonction s'appelle le *corps* de cette fonction. Par exemple, au lieu de recopier plusieurs fois dans un programme l'instruction

```
System.out.println();
System.out.println();
System.out.println();
```

qui permet de sauter trois lignes, on définit une fonction `sauterTroisLignes`

```
static void sauterTroisLignes () {
 System.out.println();
 System.out.println();
 System.out.println();}
```

puis on appelle cette fonction depuis le programme principal

```
sauterTroisLignes();
```

autant de fois que nécessaire.

Organiser un programme en fonctions permet d'éviter les redondances. En outre, cela rend les programmes plus clairs et plus faciles à lire : pour comprendre un programme qui utilise la fonction `sauterTroisLignes`, il n'est pas nécessaire de savoir comment cette fonction est programmée, il suffit de savoir ce qu'elle fait. Enfin, cela permet d'organiser l'écriture du programme. On peut décider d'écrire la fonction `sauterTroisLignes` un jour et le programme principal le lendemain ou organiser une équipe de manière qu'un programmeur écrive la fonction `sauterTroisLignes` et un autre le programme principal.

Dans certains langages de programmation comme les langages d'assemblage ou Basic, la notion de fonction se réduit à ce procédé sommaire d'abréviations.

Le passage d'arguments

Dans de nombreux cas, l'instruction que l'on cherche à isoler présente quelques variations d'une occurrence à l'autre et on souhaite donner un paramètre, un *argument*, à la fonction. Par exemple, écrire une fonction `sauterDesLignes` qui prend en argument un nombre entier `n` et saute `n` lignes

```
static void sauterDesLignes (int n) {
int i = 0;
for (i = 0; i < n; i = i + 1)
 System.out.println();}
```

puis l'utiliser dans le programme principal sous la forme `sauterDesLignes(3);` ou `sauterDesLignes(10);`. La variable `n` qui figure comme argument dans la définition de la fonction s'appelle un *argument formel* de la fonction. Quand on appelle une fonction `sauterDesLignes(3);` l'expression `3` que l'on donne en argument s'appelle un *argument réel* de l'appel.

La notion de valeur

Dans de nombreux cas également, on veut que la fonction puisse non seulement recevoir de l'information depuis le programme principal, mais également en transmettre en retour en direction de ce programme. Par exemple, on veut pouvoir écrire une fonction `hypotenuse` qui prend en argument les longueurs des deux petits côtés d'un triangle rectangle, calcule la longueur de l'hypoténuse de ce rectangle et la renvoie au programme principal

```
static double hypotenuse (double x, double y) {
 return Math.sqrt(x * x + y * y);}
```

puis l'utiliser dans le programme principal

```
u = hypotenuse(3,4);
v = hypotenuse(5,12);
```

Une fonction peut d'une part effectuer une action, par exemple afficher quelque chose ou modifier la mémoire, et d'autre part renvoyer une valeur.

En Java, la déclaration d'une fonction est formée du mot-clé `static` suivi du type de la valeur retournée, du nom de la fonction, de la liste des arguments formels, chacun précédé de son type, et du corps de la fonction. Quand la fonction ne renvoie pas de valeur, le type de retour est remplacé par le mot-clé `void`.

Une fonction qui ne renvoie pas de valeur s'appelle une *procédure*. Dans certains langages, comme Pascal ou Ada, les procédures sont distinguées et déclarées par un mot-clé spécial. À l'inverse, dans d'autres langages, comme Caml, une procédure est simplement une fonction qui renvoie une valeur de type `unit`. Comme son nom l'indique, `unit` est un type singleton qui ne contient qu'une valeur, écrite `()`. Une procédure renvoie donc invariablement la valeur `()`, ce qui ne transmet aucune information. Le cas de Java est intermédiaire, puisque l'on y déclare une procédure en remplaçant le type de la valeur retournée par le mot-clé `void`. Contrairement au `unit` de Caml, le `void` de Java n'est pas un type, mais seulement un mot-clé qui indique l'absence de valeur renvoyée : il n'est, par exemple, pas possible de déclarer une variable de type `void`.

L'appel d'une fonction, comme `hypotenuse(3,4)`, est une expression et celui d'une procédure, comme `sauterDesLignes(10);`, est une instruction.

Les variables globales

Imaginons que nous voulions isoler l'instruction `x = 0;` dans le programme

```
int x = 0;
x = 3;
x = 0;
```

Cela nous amènerait à écrire la fonction

```
static void reset () {x = 0;}
```

puis le programme principal

```
int x = 0;
x = 3;
reset();
```

Mais ce programme est incorrect, car l'instruction `x = 0;` qui apparaît dans le corps de la fonction `reset` n'est plus dans la portée de la variable `x`. Pour que la fonction `reset` puisse avoir accès à la variable `x`, il faut déclarer la variable `x` comme une variable *globale*, c'est-à-dire commune à toutes les fonctions du programme et au programme principal

```
static int x = 0;

static void reset () {x = 0;}
```

puis le programme principal

```
x = 3;
reset();
```

En Java, toutes les fonctions peuvent utiliser toutes les variables globales, que celles-ci soient déclarées avant ou après la fonction.

Un *programme* est donc constitué de la déclaration de variables globales x_1, …, x_n, de la définition de fonctions f_1, …, f_n, puis du *programme principal* `p`. Il a été choisi, en Java, de faire du programme principal une fonction qui porte un nom spécial : `main`. La fonction `main` ne doit pas renvoyer de valeur et doit toujours avoir un argument de type `String []`. Outre le mot-clé `static`, la définition de la fonction `main` doit aussi être précédée du mot-clé `public`.

Par ailleurs, il faut donner un nom au programme et introduire ce nom par le mot-clé `class`. La forme générale d'un programme est donc

```
class Prog {
static T1 x1 = t1;
…
static Tn xn = tn;
static … f1 (…) …
…
```

```
static … f_n'    (…) …

public static void main (String [] args) {p}}
```

Par exemple

```
class Hypotenuse {

static double hypotenuse (double x, double y) {
return Math.sqrt(x * x + y * y);}

public static void main (String [] args) {
System.out.println(hypotenuse(3,4));}}
```

La sémantique des fonctions

Étendre la sémantique du noyau impératif que nous avons définie précédemment demande de prendre en compte plusieurs nouveautés.

Tout d'abord, les fonctions Θ et Σ doivent prendre en arguments, outre une instruction, un environnement et une mémoire, un *environnement global* `G`. Cet environnement global est constitué, d'une part, d'un environnement `e` qui contient la déclaration des variables globales et, d'autre part, des définitions de fonctions. Ensuite, nous devons prendre en compte le fait que, puisqu'une expression peut désormais être un appel de fonction et que l'exécution du corps de cette fonction peut modifier la mémoire, l'évaluation d'une expression peut modifier la mémoire. De ce fait, le résultat Θ`(t,e,m,G)` de l'évaluation d'une expression `t` ne sera plus simplement une valeur, mais un couple formé d'une valeur et d'une mémoire. Il faut donc modifier les définitions ci-avant, afin de prendre en compte ces nouveautés.

Ensuite, il faut ajouter dans la définition des fonctions Θ et Σ le cas des instructions formées d'un appel de fonction.

Pour définir Σ`(f(t`$_1$`, …, t`$_n$`);,e,m,G)`, on doit, dans un premier temps, évaluer les arguments réels de la fonction, ce qui produit des valeurs `V`$_1$`, … V`$_n$ et une nouvelle mémoire `m`$_n$ `((V`$_1$`, m`$_1$`) =` Θ`(t`$_1$`,e,m,G), (V`$_2$`, m`$_2$`) =` Θ`(t`$_2$`,e,m`$_1$`,G), …, (V`$_n$`, m`$_n$`) =` Θ`(t`$_n$`,e,m`$_{n-1}$`,G))`. Puis, on étend l'environnement `e'` des variables globales, qui fait partie de l'environnement global `G`, en ajoutant des liaisons entre les arguments formels `x`$_1$`, …, x`$_n$ de la fonction et de nouvelles références `r`$_1$`, …, r`$_n$, ce qui donne l'environnement `e" = e' + [x`$_1$ `= r`$_1$`] + … + [x`$_n$ `= r`$_n$`]`. De même, on étend la mémoire `m`$_n$ en ajoutant des liaisons entre les références `r`$_1$`, …, r`$_n$ et les valeurs `V`$_1$`, …, V`$_n$, ce qui donne la mémoire `m" = m`$_n$ `+ [r`$_1$ `= V`$_1$`] + … + [r`$_n$ `= V`$_n$`]`. On

exécute alors le corps de la fonction dans cet environnement et cette mémoire étendus et le résultat $\Sigma(p,e'',m'',G)$ est une mémoire m'''. On pose $\Sigma(f(t_1, \ldots, t_n),e,m,G) = m'''$. Le point essentiel ici est que l'environnement dans lequel on exécute le corps de la fonction contient les déclarations des variables globales et des arguments formels de la fonction, mais pas les variables déclarées dans le programme principal.

Dans la définition des fonctions Σ et Θ que nous avons données ci-avant, les arguments d'une instruction ou expression de la forme $f(t_1, \ldots, t_n)$ sont évalués de la gauche vers la droite : t_1 est évalué dans la mémoire m, puis t_2 est évalué dans la mémoire m_1 produite par l'évaluation de t_1…

Une alternative aurait été d'évaluer les arguments de la droite vers la gauche : évaluer t_n dans la mémoire m, puis t_{n-1} dans la mémoire m_{n-1} produite par l'évaluation de t_n… Et ces deux manières de faire ne sont pas équivalentes.

Par exemple, si la fonction `f` renvoie son argument après avoir augmenté de `1` la valeur d'une variable globale `n`

```
static int f (int x) {
n = n + 1;
return x;}
```

la fonction `g` renvoie son argument après avoir multiplié par `2` la valeur de la variable `n`

```
static int g (int x) {
n = 2 * n;
return x;}
```

et la fonction `h` renvoie la somme de ses deux arguments et de la variable `n`, alors l'évaluation de l'expression `h(f(4),g(5))` dans un état où la valeur de la variable `n` est `0`, donne la valeur `11` dans le premier cas et `10` dans le second. En effet, dans le premier cas, la valeur de `n` est d'abord augmentée de `1` avant d'être multipliée par `2`, ce qui donne `2`, alors que, dans le second, elle est d'abord multipliée par `2` avant d'être augmentée de `1`, ce qui donne `1`.

Le passage d'arguments par valeur et par référence

Si le contenu initial de la variable `x` est `4` et celui de la variable `y` est `7`, après avoir exécuté l'instruction `z = x; x = y; y = z;`, la variable `x` contient la valeur `7` et la variable `y` la valeur `4`. On peut vouloir isoler cette instruction, qui intervertit le contenu des variables `x` et `y`, dans une fonction.

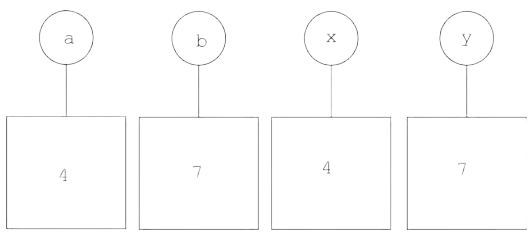
```
class Prog {

static int a = 0;
static int b = 0;
static void echange (int x, int y) {
int z = x; x = y; y = z;}

static public void main (String [] args) {
a = 4;
b = 7;
echange(a,b);
System.out.println(a);
System.out.println(b);}}
```

De manière surprenante, alors que l'on s'attendrait naïvement à ce que les contenus de `a` et `b` aient été intervertis et que les nombres `7` puis `4` s'affichent, c'est le nombre `4` qui s'affiche en premier, suivi du nombre `7`.

En fait, ce résultat est tout à fait cohérent avec la sémantique définie ci-avant. On part avec un environnement $e = [a = r_1, b = r_2]$ et une mémoire $m = [r_1 = 4, r_2 = 7]$. L'appel de la fonction `echange(a,b);` demande de calculer les valeurs des expressions `a` et `b` dans l'environnement `e` et la mémoire `m`. On obtient `4` et `7` respectivement. Puis on construit l'environnement $[a = r_1, b = r_2, x = r_3, y = r_4]$ et la mémoire $[r_1 = 4, r_2 = 7, r_3 = 4, r_4 = 7]$



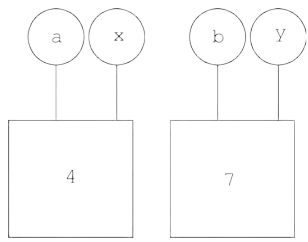
On intervertit ensuite le contenu des variables `x` et `y`, en utilisant une variable auxiliaire, ce qui donne la mémoire $[r_1 = 4, r_2 = 7, r_3 = 7, r_4 = 4, r_5 = 4]$ et on revient au programme principal.

L'environnement est alors $e = [a = r_1, b = r_2]$ et la mémoire $[r_1 = 4, r_2 = 7, r_3 = 7, r_4 = 4, r_5 = 4]$. Le contenu des variables `a` et `b` n'a pas changé.

Autrement dit, la fonction `echange` ignore tout des variables `a` et `b`, elle ne peut utiliser que leur valeur au moment de l'appel et, en aucun cas, ne peut modifier leur contenu : exécuter l'instruction `echange(a,b);` donne le même résultat qu'exécuter l'instruction `echange(4,7);`, c'est-à-dire cela ne fait rien.

Le mécanisme de passage des arguments, tel que nous l'avons décrit, et qui s'appelle le *passage par valeur*, ne permet donc pas d'écrire une fonction `echange` qui intervertit le contenu de deux variables.

Afin que l'interversion des contenus des variables `x` et `y` intervertisse celui des variables `a` et `b`, on souhaiterait plutôt que le corps de la fonction s'exécute dans l'environnement $[a = r_1, b = r_2, x = r_1, y = r_2]$ et la mémoire $[r_1 = 4, r_2 = 7]$.



Distinguer ces deux états est la principale motivation de la décomposition de l'état en un environnement et une mémoire en introduisant un ensemble intermédiaire de références, comme nous l'avons fait ci-avant.

La plupart des langages de programmation comportent une construction qui permet d'écrire une telle fonction. Mais cette construction est un peu différente dans chaque langage. Certains langages, comme Pascal, comportent un mécanisme primitif appelé le passage d'arguments *par référence*, ou *par variable*. Ainsi, dans la définition de la procédure `echange`, on peut faire précéder chaque argument du mot-clé `var`. Quand un argument d'une fonction est ainsi déclaré en passage par référence, on ne peut appliquer cette fonction qu'à une variable. Ainsi, on peut écrire `echange(a,b)` mais pas `echange(4,7)`, ni `echange(2 * a,b)`. D'autres langages, comme Caml et C, utilisent le fait que les références sont des valeurs, c'est-à-dire que l'ensemble `Ref` est un sous-ensemble de l'ensemble `Val`, et on écrit une fonction `echange` qui prend en argument, non deux entiers, mais deux références. En C, la référence associée à une variable `x` dans l'environnement s'écrit `&x` (`Θ(&x,e,m,G) = e(x)`). En Caml, la référence associée à une variable `x` dans l'environnement s'écrit `x`, – en Caml `Θ(x,e,m,G) = e(x)` et non `m(e(x))` – alors que la valeur associée à cette référence dans la mémoire s'écrit `!x` – `Θ(!t,e,m,G) =`

m(Θ(t,e,m,G)). Le mécanisme utilisé en Java est encore différent, il consiste à utiliser des *types enveloppés*. Nous le présenterons au paragraphe « Les enregistrements », car il utilise des constructions du langage que nous n'avons pas encore introduites.

La récursivité

Précédemment, pour définir la fonction Σ sur une instruction de la forme $f(t_1, \ldots, t_n);$, nous avons utilisé la fonction Σ sur l'instruction p, qui est le corps de la fonction f. Cette définition est-elle bien formée ou peut-elle être circulaire ?

Cette définition est clairement bien formée quand l'instruction p ne contient pas elle-même d'appels de fonctions, c'est-à-dire quand le programme principal – la fonction main – appelle des fonctions qui n'appellent pas, elles-mêmes, de fonctions.

Cette définition est également bien formée quand le programme contient k définitions de fonctions $f_1, \ldots, f_k$ telles que le corps de la fonction f_i ne contienne que des appels à des fonctions antérieurement définies, c'est-à-dire à des fonctions f_j pour $j < i$. Certains langages, comme Fortran, ne permettent ainsi d'appeler une fonction f dans le corps d'une fonction g que si f a été définie avant g. Remarquons qu'un tel ordre sur les fonctions existe toujours si ces fonctions sont introduites à partir du programme principal en isolant des instructions l'une après l'autre : on isole une instruction p_k du programme principal, puis une instruction p_{k-1} du programme principal ou de la fonction p_k …

Cependant, la plupart des langages de programmation permettent d'écrire des fonctions qui s'appellent elles-mêmes, ou qui appellent des fonctions qui appellent d'autres fonctions… qui, *in fine*, appellent la fonction initiale. On les appelle *définitions récursives*. Un exemple de définition récursive est celle de la fonction factorielle

```
static int fact (int x) {
if (x == 0) return 1;
else return x * fact(x - 1);}
```

Pour calculer la factorielle du nombre 3, on doit calculer la factorielle de 2, ce qui demande de calculer la factorielle de 1, ce qui demande de calculer la factorielle de 0. Cette valeur est 1. La factorielle de 1 est donc obtenue en multipliant 1 par cette valeur, ce qui donne 1. La factorielle de 2 est obtenue en

multipliant `2` par cette valeur, ce qui donne `2`. Et la factorielle de `3` est obtenue en multipliant `3` par cette valeur, ce qui donne `6`.

Pour les définitions récursives, la définition de la fonction Σ ci-avant peut être circulaire. Par exemple, si `f` est une fonction définie ainsi

```
static void f (int x) {f(x);}
```

alors la définition de `Σ(f(x);,e,m,G)` utilise la valeur de `Σ(f(x);,e,m,G)`. Nous devons donc trouver une autre manière de définir cette fonction Σ.

On dit parfois qu'une définition récursive est celle qui utilise l'objet qu'elle définit. Cette idée est absurde : les définitions circulaires sont incorrectes, dans les langages de programmation, comme ailleurs. D'ailleurs, s'il était possible d'utiliser une fonction dans sa propre définition, la fonction factorielle aurait une définition beaucoup plus simple

```
static int f (int x) {return f(x);}
```

et la définition de la fonction qui multiplie son argument par `4` ou qui l'élève au carré serait absolument identique. Une autre tentative de comprendre les définitions récursives est d'y voir des définitions par récurrence. Mais, si cette idée marche dans le cas de la fonction factorielle, elle est insuffisante dans le cas général, car rien n'empêche une fonction récursive de s'appeler elle-même sur un argument plus grand que son argument.

Une idée plus intéressante est de transformer une définition récursive, par exemple celle de la fonction `fact`, en une autre, non récursive, en remplaçant les appels à la fonction `fact` dans le corps de la fonction `fact` par des appels à une autre fonction `fact1`, identique à `fact`, mais définie avant elle

```
static int fact1 (int x) {
if (x == 0) return 1;
else return x * fact1(x - 1);}

static int fact (int x) {
if (x == 0) return 1;
else return x * fact1(x - 1);}
```

La définition de la fonction `fact` n'est désormais plus récursive, mais celle de la fonction `fact1` l'est. On peut, de même, remplacer les appels à la fonction `fact1` dans le corps de la fonction `fact1` par des appels à une fonction `fact2`,

et ainsi de suite. On aboutit à un programme qui n'est plus récursif, mais qui est infini. Les définitions récursives sont donc, comme la boucle `while`, un moyen d'exprimer des programmes infinis et, comme la boucle `while`, elles introduisent une potentialité de non-terminaison.

Comme pour le cas de la boucle `while`, on peut introduire une expression fictive `giveup` et approcher ce programme `p` infini par des approximations finies p_n obtenues en remplaçant la définition de la $n^{ème}$ copie de la fonction `fact` par la fonction `giveup` et en supprimant les suivantes qui ne sont plus utiles. Calculer la valeur de la $n^{ème}$ approximation du programme `p` consiste à tenter de calculer la valeur du programme `p` en faisant au maximum `n` appels récursifs imbriqués. Si, au bout de ces `n` appels, le calcul n'est pas terminé, on l'abandonne.

Si la fonction Σ n'est définie en aucun quadruplet (p_n,e,m,G), cela signifie que, quel que soit le nombre d'appels récursifs imbriqués que l'on s'accorde, le programme ne termine pas. Dans ce cas, on pose que Σ n'est pas définie en `(p,e,m,G)`. Sinon, soit `n` le plus petit entier tel que Σ soit définie en (p_n,e,m,G), on pose $\Sigma(p,e,m,G) = \Sigma(p_n,e,m,G)$.

Une autre manière de définir la sémantique des fonctions récursivement définies est de les voir comme des équations. La fonction `fact` est définie comme une solution de l'équation `fact = if (x == 0) return 1; else return x * fact(x - 1);`. Cette démarche aboutit exactement au même résultat, car pour démontrer l'existence d'une solution de cette équation, on procède à nouveau par approximations successives.

Programmer sans affectation

En comparant la fonction factorielle écrite avec une boucle

```
static int fact (int x) {
int i = 0;
int r = 1;
for (i = 1; i <= x; i = i + 1) {r = r * i;}
return r;}
```

et récursivement

```
static int fact (int x) {
if (x == 0) return 1;
else return x * fact(x - 1);}
```

on constate que la première utilise des affectations : `i = 1;`, `i = i + 1;` et `r = r * i;`, mais pas la seconde. Il est donc possible de programmer la fonction factorielle sans utiliser d'affectation.

Plus généralement, on peut considérer un sous-langage de Java dans lequel on supprime l'affectation. Dans ce cas, la séquence et la boucle deviennent inutiles. Il reste un noyau de Java formé de la déclaration de variables, de la définition récursive de fonctions, de l'appel de fonctions, des opérations arithmétiques et logiques et du test. Ce sous-langage s'appelle le *noyau fonctionnel* de Java. Après son noyau impératif, nous voyons donc apparaître un nouveau sous-langage de Java. Et on peut, de même, définir le noyau fonctionnel de beaucoup de langages de programmation. De manière surprenante, ces deux sous-ensembles de Java ont la même puissance que Java tout entier. L'ensemble des fonctions que l'on peut programmer, en Java, dans le noyau impératif de Java ou dans son noyau fonctionnel est le même. Bien entendu, ce résultat n'est vrai que parce que les définitions de fonctions peuvent être récursives. La boucle et la récursivité sont donc deux moyens essentiellement redondants de construire des programmes infinis, et chaque fois que l'on a besoin de construire un programme infini, on peut choisir d'utiliser ou bien une boucle ou bien une définition récursive.

Les enregistrements

Dans les programmes que nous avons décrits ci-avant, chaque variable contient un nombre entier, un nombre à virgule, un booléen ou un caractère. Une telle variable ne peut pas contenir un objet formé de plusieurs nombres, booléens ou caractères, comme, par exemple, un nombre complexe formé de deux nombres à virgule, un vecteur formé de plusieurs coordonnées ou une chaîne formée de plusieurs caractères. Nous allons maintenant présenter une construction, les *enregistrements*, qui permet de construire des types *composites*, c'est-à-dire des types construits comme produits cartésiens d'autres types.

Les n-uplets à champs nommés

Mathématiquement, un n-uplet est une fonction dont le domaine est un segment initial de $\mathbb{N}$. Dans les langages de programmation, les n-uplets sont la plupart du temps *à champs nommés*, c'est-à-dire que ce sont des fonctions dont le domaine est, non un segment initial de $\mathbb{N}$, mais un ensemble fini quelconque, dont les éléments sont appelés *étiquettes*. Un tel n-uplet à champs nommés s'appelle un *enregistrement*.

Par exemple, si l'on se donne un ensemble d'étiquettes `latitude`, `longitude` et `altitude`, on peut construire l'enregistrement `{latitude = 45.83, longitude = 6.86, altitude = 4810.0}`.

En Java, on définit un nouveau type d'enregistrements en indiquant l'étiquette et le type de chacun de ses champs. Par exemple, le type `Point` se définit ainsi

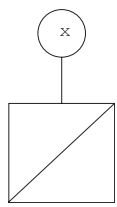
```
class Point {
double latitude;
double longitude;
double altitude;}
```

Une telle définition s'écrit avant l'introduction du nom du programme par le mot-clé `class`.

Une fois un tel type défini, on peut donner le type `Point` à une variable.

```
Point x = null;
```

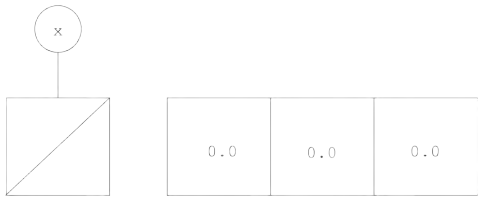
Comme pour toute déclaration de variable, cela ajoute un couple à l'environnement associant une référence `r` à cette variable et un couple à la mémoire associant une valeur à la référence `r`. Dans cet exemple, on déclare cette variable en lui donnant comme valeur initiale une valeur spéciale appelée `null`. On représente ainsi un état, dans lequel la variable `x` est associée dans l'environnement à une référence `r`, elle-même associée dans la mémoire à la valeur `null`.



En Java, la référence `r` n'est jamais associée directement à un enregistrement dans la mémoire. La case `r` est toujours une « petite » case qui ne peut contenir que `null` ou une autre référence. Pour associer un enregistrement à la variable `x`, il faut donc commencer par créer une case suffisamment grande pour contenir trois nombres à virgule. Cela se fait avec une nouvelle construction : `new`

```
new Point()
```

L'évaluation de l'expression `new Point()` crée une nouvelle référence `r'` et associe cette référence à un enregistrement, par défaut `{latitude = 0.0, longitude = 0.0, altitude = 0.0}`. La valeur de cette expression est la référence `r'`.

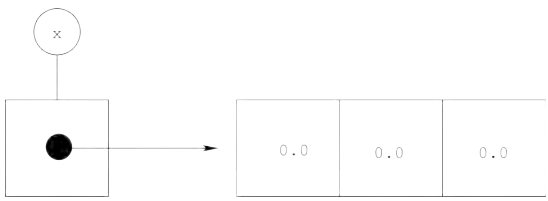


Une référence qui a été ajoutée à la mémoire par la construction `new` s'appelle une *cellule*. L'ensemble des cellules de la mémoire s'appelle le *tas*. L'opération consistant à ajouter une nouvelle cellule à la mémoire s'appelle l'*allocation* d'une cellule.

Quand on exécute l'instruction

```
x = new Point();
```

on associe la référence `r'` – à droite sur le dessin – à la référence `r` – à gauche sur le dessin – dans la mémoire. L'environnement est alors `[x = r]` et la mémoire `[r = r', r' = {latitude = 0.0, longitude = 0.0, altitude = 0.0}]`.



La valeur de l'expression `x` dans cet environnement et cette mémoire est alors `m(e(x))`, c'est-à-dire la référence `r'`.

Construire de telles mémoires dans lesquelles une référence `r'` est associée à une référence `r` est la principale motivation pour considérer les références comme des valeurs, c'est-à-dire pour prendre `Ref` comme un sous-ensemble de `Val`.

Il est, bien entendu, possible de déclarer la variable `x` en lui donnant une cellule comme valeur initiale

```
Point x = new Point();
```

Si la valeur de l'expression `t` est une référence `r'` associée dans la mémoire à un enregistrement et `l` est une étiquette, la valeur de l'expression `t.l` est le champ `l` de cet enregistrement. Ainsi, l'instruction

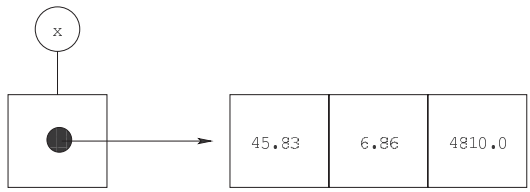
```
System.out.println(x.longitude);
```

affiche `0.0`. En particulier, quand `t` est une variable `x`, sa valeur est `m(e(x))` et donc la valeur de l'expression `x.latitude` est le champ `latitude` de l'enregistrement `m(m(e(x)))`.

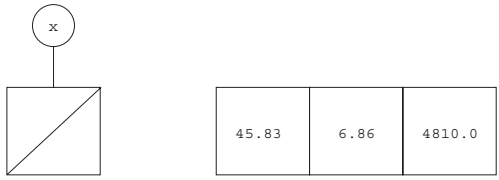
Pour affecter un champ d'un enregistrement, on utilise une nouvelle instruction `t.l = u;`, où `l` est une étiquette et `t` et `u` sont des expressions. Si la valeur de l'expression `t` est une référence `r` associée dans la mémoire à un enregistrement qui a un champ `l`, alors, quand on exécute l'instruction `t.l = u;`, le champ `l` de cet enregistrement reçoit la valeur de l'expression `u`. Ainsi, en exécutant les instructions

```
x.latitude = 45.83;
x.longitude = 6.86;
x.altitude = 4810.0;
```

on construit l'état



Enfin, quand une cellule n'est plus utilisée par un programme, elle peut être supprimée de la mémoire. Cette opération s'appelle la *désallocation* de la cellule. Ainsi, si on exécute l'instruction `x = null;` dans l'environnement `[x = r]` et la mémoire `[r = r', r' = {latitude = 45.83, longitude = 6.86, altitude = 4810.0}]`, on obtient la mémoire `[r = null, r' = {latitude = 45.83, longitude = 6.86, altitude = 4810.0}]`,



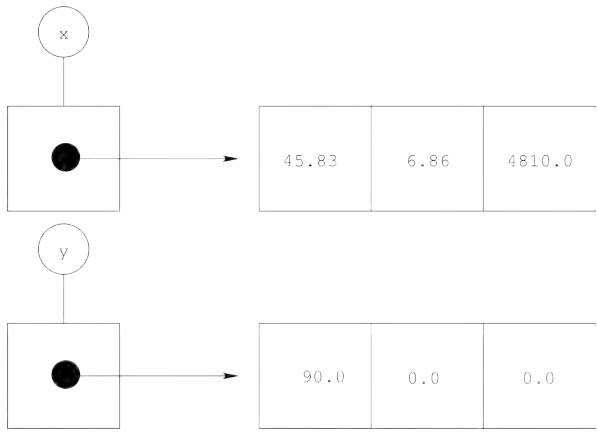
dans laquelle la cellule `r'` est devenue inutile. On peut la désallouer, afin d'obtenir la mémoire `[r = null]`. En Java, cette opération est automatique ; dans d'autres langages, c'est au programmeur de décider quelles cellules désallouer.

Le mécanisme des enregistrements dans un langage de programmation, comme Java, est donc constitué de cinq constructions qui permettent :
– de définir un type, `class` en Java,
– d'allouer une cellule, `new` en Java,
– d'accéder à un champ, `t.l` en Java,
– d'affecter un champ, `t.l = u;` en Java,
– de désallouer une cellule, automatique en Java.

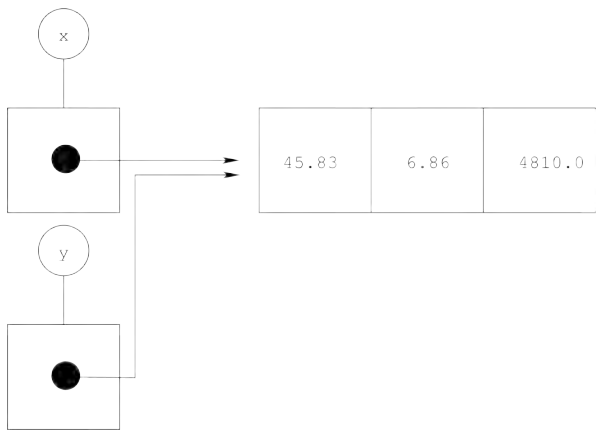
Comprendre le mécanisme des enregistrements dans un nouveau langage de programmation consiste à comprendre les constructions qui servent à définir un type, à allouer et désallouer une cellule, à accéder à un champ et à l'affecter.

Le partage

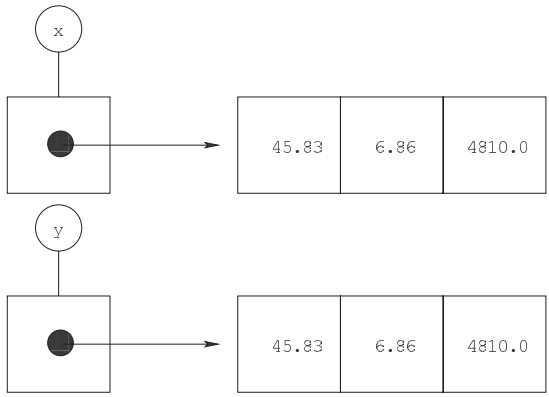
Supposons que `x` et `y` soient deux variables de type `Point`, associées à deux références r_1 et r_2 dans l'environnement. Supposons, en outre, que, dans la mémoire, r_1 soit associée à une référence r_3, elle-même associée à un enregistrement {`latitude = 45.83, longitude = 6.86, altitude = 4810.0`} et r_2 à une référence r_4, elle-même associée à l'enregistrement {`latitude = 90.0, longitude = 0.0, altitude = 0.0`}. Ainsi, `e = [x =` r_1`, y =` r_2`]`, `m = [`r_1 `=` r_3`,` r_2 `=` r_4`,` r_3 `= {latitude = 45.83, longitude = 6.86, altitude = 4810.0}, `r_4 `= {latitude = 90.0, longitude = 0.0, altitude = 0.0}]`.



Quand on exécute l'instruction `y = x;`, on calcule la valeur de `x`, qui est la référence r_3 et on associe cette valeur à la référence r_2. La mémoire devient [r_1 = r_3, r_2 = r_3, r_3 = {latitude = 45.83, longitude = 6.86, altitude = 4810.0}].



En revanche, si l'on exécute les instructions `y.latitude = x.latitude; y.longitude = x.longitude; y.altitude = x.altitude;`, on obtient la mémoire [r_1 = r_3, r_2 = r_4, r_3 = {latitude = 45.83, longitude = 6.86, altitude = 4810.0}, r_4 = {latitude = 45.83, longitude = 6.86, altitude = 4810.0}].



Si l'on affecte ensuite le champ `latitude` de l'enregistrement `x` : `x.latitude = 23.45;` et que l'on affiche le champ `latitude` de `y`, on obtient `23.45` dans le premier cas, et `45.83` dans le second. On dit, dans le premier cas, que les variables `x` et `y` *partagent* la cellule r_3. Toute modification de la cellule associée à `x` entraîne automatiquement une modification de celle associée à `y` et réciproquement.

Si `a` et `b` sont deux expressions de type `Point`, leur valeur est une référence et l'expression `a == b` vaut `true` uniquement quand ces deux références sont identiques. C'est-à-dire quand `a` et `b` partagent une même cellule. On appelle cette relation l'égalité *physique*.

Il est cependant possible d'écrire une fonction qui teste l'égalité *structurelle* de deux enregistrements, c'est-à-dire l'égalité champ à champ

```
static boolean equal (Point x, Point y) {
return (x.latitude == y.latitude)
&& (x.longitude == y.longitude)
&& (x.altitude == y.altitude);}
```

Les types enveloppés

Un type *enveloppé* est un type enregistrement à un seul champ. Un exemple est le type `Integer`

```
class Integer {
int c;}
```

À première vue, le type `Integer` peut sembler redondant avec le type `int`, puisque l'enregistrement, c'est-à-dire le mono-uplet, `{c = 4}` n'est pas très différent de l'entier `4`. Et il est vrai que le programme

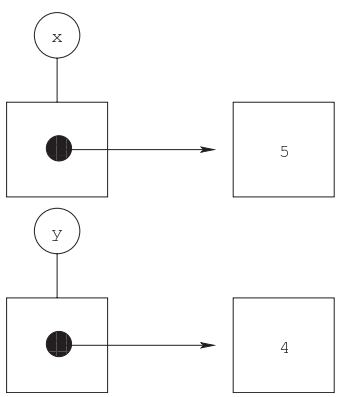
```
Integer x = new Integer();
x.c = 4;
Integer y = new Integer();
y.c = x.c;
x.c = 5;
System.out.println(y.c);
```

peut se réécrire en

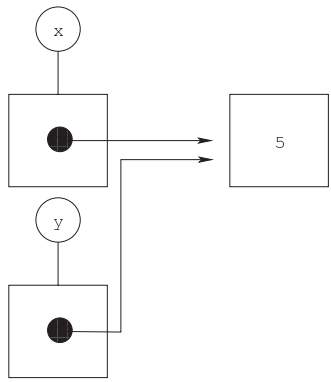
```
int x = 4;
int y = x;
x = 5;
System.out.println(y);
```

qui produit le même résultat : l'un comme l'autre affichent le nombre `4`.

Cependant, si on remplace l'instruction `Integer y = new Integer(); y.c = x.c;` par `Integer y = x;` au lieu d'obtenir l'état

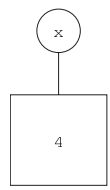


on obtient l'état

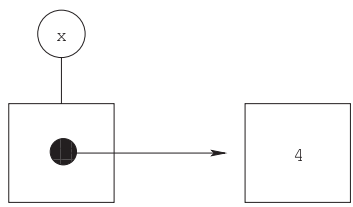


et le programme affiche `5`, et non `4`.

Plus généralement, au lieu d'avoir un état dans lequel une variable `x` est associée à une référence `r` associée à une valeur, par exemple, `4`



les types enveloppés permettent d'avoir un état dans lequel une variable `x` est associée à une référence `r`, associée à une référence `r'`, elle-même associée à une valeur `4`.



Cela permet, en particulier, à plusieurs variables de partager une valeur.

Comme nous l'avons vu précédemment, en Java, il n'est pas possible d'écrire une fonction qui intervertit le contenu de deux arguments de type `int`. En revanche, cela est possible pour des arguments de type `Integer`.

```
static void echange (Integer x, Integer y) {
int z = x.c;
x.c = y.c;
y.c = z;}
```

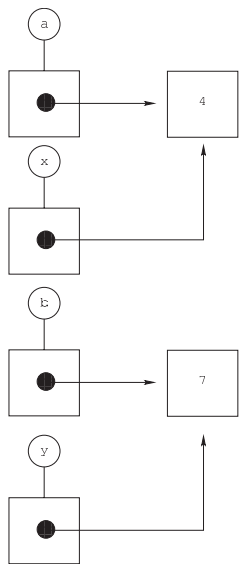
et le programme

```
public static void main (String [] args) {
a = new Integer();
a.c = 4;
b = new Integer();
b.c = 7;
echange(a,b);
System.out.println(a.c);}
System.out.println(b.c);}
```

affiche `7` et `4`.

Les types enveloppés permettent donc de réaliser le passage d'arguments par référence en Java.

En effet, quand on appelle la fonction `echange`, on construit l'état



et intervertir les contenus de `x` et `y` intervertit bien ceux de `a` et `b`.

Les tableaux

Les langages de programmation permettent aussi de construire des n-uplets dont les champs ne sont pas nommés, mais indicés par des entiers. On les appelle des *tableaux*.

Les champs d'un tableau, contrairement à ceux d'un enregistrement, sont tous du même type.

Le nombre de champs d'un tableau est déterminé au moment de l'allocation du tableau, et non au moment de la déclaration de son type, comme c'est le cas pour un enregistrement. Cela permet de calculer, au cours de l'exécution d'un programme, un entier `n` et d'allouer un tableau de taille `n`. Il est également possible de calculer, au cours de l'exécution d'un programme, un entier `k` et d'accéder au $k^{ème}$ champ d'un tableau, ou de l'affecter. En revanche, une fois le tableau alloué, il n'est plus possible de changer sa taille. La seule possibilité est d'allouer un nouveau tableau et de recopier l'ancien dans le nouveau.

En Java, un tableau dont les éléments sont de type `T` est de type `T []`.

On alloue un tableau avec l'opération `new`

```
new int [10]
```

L'évaluation de l'expression `new int [u]`, où `u` est une expression dont la valeur est un entier `n`, crée une nouvelle référence `r'` et associe cette référence à un n-uplet ne contenant que des valeurs par défaut : `0` dans ce cas. Les champs sont numérotés de `0` à `n - 1`.

Si la valeur de l'expression `t` est une référence associée dans la mémoire à un tableau et la valeur de l'expression `u` est un entier `k`, la valeur de l'expression `t[u]` est la valeur contenue dans le $k^{ème}$ champ de ce tableau.

Pour affecter le $k^{ème}$ champ d'un tableau, on utilise une nouvelle instruction `t[u] = v`; où `t` est une expression dont la valeur est une référence associée dans la mémoire à un tableau, `u` est une expression dont la valeur est un entier `k` et `v` une expression de même type que les éléments du tableau. Quand on exécute cette instruction, le $k^{ème}$ champ du tableau reçoit la valeur de l'expression `v`.

Ainsi, le programme

```
int [] t = new int [10];
int k = 5;
t[k] = 4;
System.out.println(t[k]);
```

affiche `4`.

Pour indicer des n-uplets par des couples ou des triplets d'entiers, typiquement pour représenter des matrices ou des images, une possibilité est de construire un tableau dont les éléments sont eux-mêmes des tableaux. Ainsi, l'élément d'indice `(i,j)` du tableau `t` s'écrit `t[i][j]`. Un tel tableau a le type `T [][]`. L'allocation d'un tel tableau à entrées multiples se fait cependant en une seule opération

```
int [][] t = new int [20][20];
```

Les types de données dynamiques

Les enregistrements permettent de construire des données composées de plusieurs nombres ou caractères. Mais toutes les valeurs d'un tel type de données sont formées du même nombre de champs. Il est impossible de définir un type enregistrement qui contienne plusieurs nombres ou caractères sans que l'on sache *a priori* combien. On peut représenter de tels objets par des tableaux, mais la taille d'un tableau est déterminée une fois pour toutes au moment de son allocation. Nous allons à présent voir comment utiliser des données dont la taille n'est pas bornée, si ce n'est par la taille de la mémoire de l'ordinateur utilisé, qui est nécessairement finie. On appelle *données dynamiques* de telles données composites dont la taille n'est pas connue *a priori* et peut évoluer au cours de l'exécution du programme.

Précédemment, nous avons construit un type enregistrement dont les champs étaient d'un type scalaire `double`. Il est également possible de définir un type enregistrement `T` dont les champs sont eux-mêmes d'un type enregistrement, en particulier le type `T` lui-même. Par exemple, un triplet d'entiers `(a,b,c)` peut se définir comme le couple `(a,(b,c))`, et plus généralement une *liste* non vide d'entiers peut se définir comme un couple formé d'un entier, la *tête* de la liste, et d'une liste plus courte, la *queue* de la liste. Cela amène à définir le type `IntList` ainsi

```
class IntList {
int hd;
IntList tl;}
```

La tête de la liste `1, 2, 3, 4`, par exemple, est l'entier `1`. La queue de cette liste est la liste `2, 3, 4` – et non l'entier `4`.

Le type `IntList` est donc un type enregistrement récursif, c'est-à-dire dont l'un des champs est lui-même de type `IntList`. Parmi les types enregistrements récursifs, certains ont un unique champ récursif et d'autres plusieurs.

On appelle *types de listes* les différents types qui ont un unique champ récursif, quel que soit le nombre de champs non récursifs, et *types d'arbres* les types qui en ont plusieurs.

Les éléments du type `IntList` sont soit :

– la valeur `null`, qui représente, par convention, la liste vide,

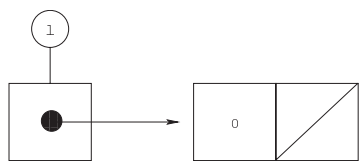
– un élément de `int × IntList`, qui représente une liste non vide.

Autrement dit, `IntList` vérifie la propriété `IntList = {null} ∪ (int × IntList)`.

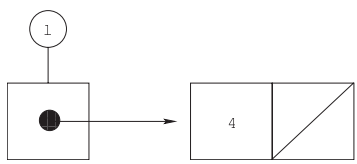
Observons ce qui se passe quand on exécute le programme

```
IntList l = new IntList();
l.hd = 4;
l.tl = new IntList();
l.tl.hd = 5;
l.tl.tl = null;
```

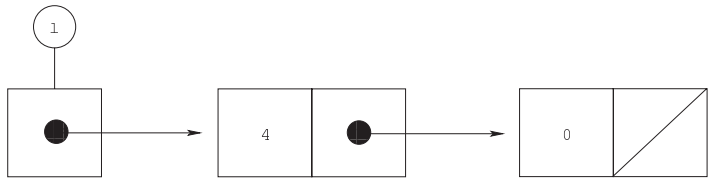
La déclaration `IntList l = new IntList();` alloue une cellule `r'`, remplit cette cellule avec les valeurs par défaut `0` et `null`, associe la variable `x` à une cellule `r` dans l'environnement et la référence `r` à la référence `r'` dans la mémoire



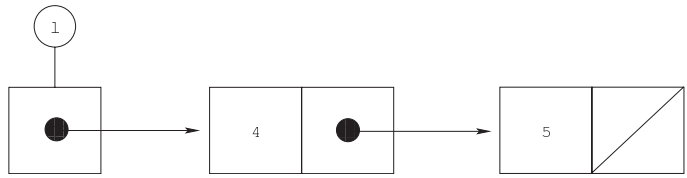
L'instruction `l.hd = 4;` affecte le champ `hd` de la cellule avec la valeur `4`.



L'instruction `l.tl = new IntList();` alloue une nouvelle cellule



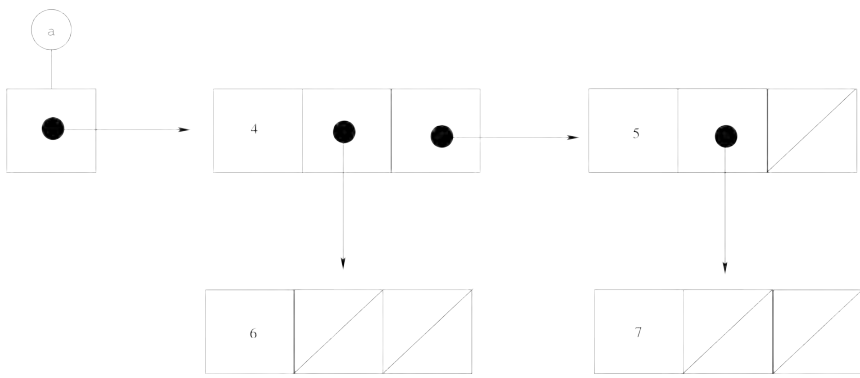
Enfin, les deux dernières instructions affectent les champs de cette cellule



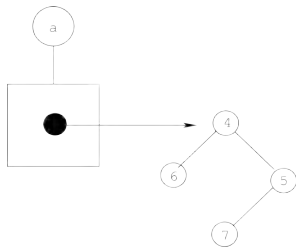
De manière similaire, nous pouvons définir le type des arbres binaires

```
class Arbre {
int val;
Arbre gauche;
Arbre droit;}
```

Un *arbre* est une valeur de ce type, c'est ou bien la valeur `null` appelée l'*arbre vide* ou bien une référence associée dans la mémoire à un enregistrement formé d'un nombre entier et de deux arbres, appelés le sous-arbre gauche et le sous-arbre droit de l'arbre `a`.



Nous utilisons souvent une notation plus pratique pour les arbres. Chaque cellule est représentée simplement par un cercle et les flèches vers les autres cellules seront remplacées par de simples segments. Ainsi, ce même état sera représenté ainsi



Nous avons « défini » ci-avant le type `IntList`, comme `IntList = {null} ∪ (int × IntList)`. Le fait que `IntList` apparaisse à droite du signe = est dû à ce que la définition est récursive. Encore une fois, les définitions récursives nous apparaissent comme des définitions circulaires. Et, encore une fois, une manière de briser cette circularité est de définir un ensemble `L` qui est solution de l'équation `X = {null} ∪ (int × X)` et de procéder par approximations successives, en définissant l'ensemble L_i des valeurs de type `IntList` que l'on peut construire en `i` étapes au plus

$L_0 = \emptyset$
$L_1 = \{null\} \cup (int \times L_0) = \{null\}$
$L_2 = \{null\} \cup (int \times L_1)$
$L_3 = \{null\} \cup (int \times L_2)$
…

puis de définir l'ensemble `L` comme la réunion de ces ensembles : $L = \bigcup_i L_i$

La valeur `null` est essentielle dans cette construction. La même construction avec l'équation `X = int × X` donnerait un type vide.

L'ensemble `L` ainsi construit n'est pas l'unique solution de l'équation `X = {null} ∪ (int × X)`, l'ensemble de toutes les suites finies ou infinies en est également une, mais l'ensemble construit est la plus petite de ces solutions pour la relation d'inclusion.

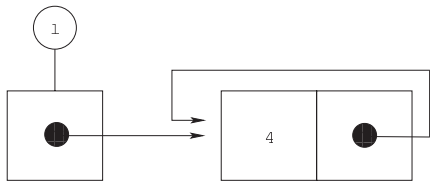
Les valeurs infinies

Le type `IntList` est également solution de l'équation `X = {null} ∪ (int × X)`, mais ce n'est pas exactement la plus petite. En effet, ce type

contient des valeurs qui ne peuvent pas se construire en un nombre fini d'étapes. Par exemple, le programme

```
IntList l = new IntList();
l.hd = 4;
l.tl = l;
```

construit la liste



c'est-à-dire la liste `4, 4, 4, 4, 4`... qui est infinie.

Le type `IntList` contient en fait toutes les listes *rationnelles*, c'est-à-dire les listes finies ou infinies qui n'ont qu'un nombre fini de sous-listes distinctes. Toutes les listes finies sont rationnelles, mais il y a des listes rationnelles infinies, comme celle-ci.

Abstraire un type de données

Lorsqu'on écrit un programme, il est souvent utile d'abstraire la définition des types de données, en cachant la définition effective de ce type et en utilisant un nombre limité de fonctions qui permettent de manipuler les éléments de ce type. On distingue ainsi souvent les fonctions de construction, ou *constructeurs*, qui servent à définir de nouvelles valeurs du type considéré, et les primitives d'accès, ou *sélecteurs*, qui servent à connaître ces valeurs. Cette abstraction permet de construire des programmes plus modulaires, plus faciles à modifier et à réutiliser. Par exemple un type `Suite`, des suites d'entiers, se spécifie ainsi.

– Constructeurs :

• `Suite vide ();` est la suite vide, qui ne contient aucun élément,

• `Suite ajoutTete (Suite s, int e);` est la suite dont le premier élément est `e` et le reste `s`.

– Sélecteurs :

• `boolean estVide (Suite s);` vaut `true` si et seulement si `s` est la suite vide,

• `int premier (Suite s);` est le premier élément de la suite `s`,

• `Suite fin (Suite s);` est la suite `s` privée de son premier élément.

Bien sûr, `premier` et fin ne sont pas définis pour une suite vide.

Différentes solutions sont alors possibles pour implémenter ce type `Suite`, par exemple en utilisant un tableau d'entiers ou une liste. Le point essentiel est que, pour utiliser ce type `Suite`, nous n'avons pas besoin de savoir s'il est défini comme un type de listes ou un type de tableaux, mais uniquement de connaître ces cinq fonctions.

La notion générale de langage

Un langage de programmation, nous l'avons vu, a cette caractéristique d'être compréhensible à la fois par une machine et par un être humain. De nombreux autres langages partagent cette caractéristique. Pour définir un tel langage, on commence toujours par définir sa syntaxe, c'est-à-dire l'ensemble des chaînes de caractères qui appartiennent à ce langage. Pour commencer par un exemple simple, nous pouvons définir la syntaxe du langage qui contient toutes les chaînes de caractères formées d'un nombre quelconque de fois la lettre `a` suivie de la lettre `b`. Ce langage contient les chaînes `b, ab, aab`… mais pas la chaîne `aba`. L'ensemble L de ces chaînes de caractères peut se définir par les deux règles suivantes.

– La chaîne de caractères `b` appartient à L.

– Si la chaîne `S` appartient à L, alors la chaîne `a S` aussi.

Ces règles peuvent s'écrire dans une notation plus compacte

$$L = \texttt{b} \mid \texttt{a}\ L$$

Nous pouvons, de même, décrire la syntaxe des instructions du noyau impératif de Java. La description que nous proposons n'est qu'indicative, la véritable syntaxe de Java est bien entendu plus complexe.

I = `T V = E ;` I | `V = E ;` | `{` $I\ I$ `}` | `if (E)` I `else` I | `while (E)` I

T = `byte | short | int | long | float | double | boolean | char`

$\mathcal{E}$ = *une expression permettant de calculer une valeur d'un type donné*

V = *un identificateur en Java*

Nous retrouvons l'idée qu'une instruction est une déclaration (`T V = E ;`), une affectation (`V = E ;`), une séquence (`{` $I\ I$ `}`), un test (`if (E)` I `else` I) ou une boucle (`while (E)` I). Dans cette définition, nous utilisons trois catégories grammaticales annexes : celle des types (T) et celles des expressions ($\mathcal{E}$) et des identificateurs (V) qui restent à définir.

La syntaxe, telle que nous l'avons définie, ne nous contraint pas à affecter une variable avec une expression du même type, ainsi l'instruction `int x; x = "toto";` est syntaxiquement correcte. Elle ne déclenchera pas une erreur

de syntaxe, mais une erreur de type. De même, en français, le verbe « dormir » s'emploie avec un sujet qui désigne une personne, mais la phrase « Une idée dort » est syntaxiquement correcte, contrairement à la phrase « Une malgré dort ».

Cette manière de décrire la syntaxe d'un langage peut s'utiliser avec d'autres langages que les langages de programmation. Un exemple est le langage HTML (*Hypertext Markup Language*) qui sert à décrire des pages web. Dans la suite de ce paragraphe, nous considérerons une version de HTML, appelée XHTML 1.0, dont la syntaxe est plus simple que celle du langage HTML proprement dit. En effet, le langage HTML permet de nombreuses libertés, ce qui complique l'écriture de sa syntaxe et des programmes l'interprétant, les navigateurs web.

La manière la plus simple de décrire une page web est d'indiquer la chaîne de caractères qui doit s'afficher sur cette page, par exemple la chaîne `Bonjour tout le monde`. Cela mène à une syntaxe simple

B = ε | *C B*

C = `a` | `b` | `c` | `d` | … | `z`

qui spécifie qu'un élément de *B* est ou bien la chaîne vide ε ou bien une chaîne non vide, formée d'un caractère, *C*, et d'une chaîne *B*.

Mais, rapidement, nous nous rendons compte que nous voulons, dans notre page web, mettre une certaine partie du texte en gras et une autre en italique. Pour mettre, par exemple, le mot `tout` en gras, il faut insérer la balise `<b>` au début du mot et la balise `</b>` à la fin de ce mot, et de même avec les balises `<i>` et `</i>` : `Bonjour <b>tout</b> le <i>monde</i>`.Ce qui nous mène à la syntaxe suivante

B = ε| *CB* | *EB*

C = `a` | `b` | `c` | `d` | …

E = `<b>` *B* `</b>` | `<i>` *B* `</i>`

qui exprime qu'un élément du langage *B* est une suite composée de caractères ou d'éléments du langage entourés de balises, comme `<b>tout</b>`. L'expression `<b><i>tout</i> le monde</b>` appartient au langage, mais pas l'expression `<i>tout <b>le</i> monde</b>`, car la seule imbrication autorisée est qu'une balise soit ouverte et fermée entre l'ouverture et la fermeture d'une autre balise, même si les êtres humains que nous sommes ont souvent une bonne idée de la sémantique qu'il faudrait attacher à cette seconde expression. Pour représenter, dans la syntaxe définie ci-avant, un document dont la sémantique serait d'écrire le mot `tout` en italique, `le` en gras et italique, puis `le monde` en gras, il faut par exemple écrire l'expression `<i>tout <b>le</b></i> <b>monde</b>`.

Nous pouvons ensuite ajouter d'autres types d'éléments, par exemple les ancres pour les liens hypertextes et les titres de paragraphes

E = `<b>` *B* `</b>` | `<i>` *B* `</i>` | `<a` *A*`>` *B* `</a>` | `<h1>` *B* `</h1>`

La balise `<h1>...</h1>` (*Heading 1*) signifie que le texte qu'elle contient est un titre de paragraphe. En XHTML, il existe 6 niveaux de titres, de `h1` à `h6`. Concernant la balise `<a>` (*Anchor*), il faut ajouter un attribut, *A*, qui sert à indiquer l'adresse de la page web vers laquelle le navigateur devra s'orienter si nous cliquons sur ce lien

A = `href =` "*U*"
U = *une adresse web*

où *U*, qui reste à décrire, est la syntaxe des adresses sur le Web. Cette syntaxe est disponible à l'adresse: `http://www.w3.org/Addressing/URL/5_BNF.html`. Elle fait environ 120 lignes. Ainsi l'expression `Bonjour <a href = "http://science-info">tout</a> le <b>monde</b>` appartient au langage.

Enfin, nous voulons pouvoir inclure dans la description d'une page web des informations, par exemple un titre, qui n'apparaîtront pas sur la page. Pour cela la page contient un en-tête. Et comme le langage XHTML peut évoluer, il faut aussi indiquer, au début du fichier, la version du langage utilisée, chaque version correspondant à une grammaire bien spécifique.

Cela nous amène à définir, en utilisant le langage *B* défini ci-avant, un langage *L*

L = *V* `<html> <head>` *H* `</head> <body>` *B* `</body> </html>`

Pour simplifier, nous pouvons supposer que *V* est le langage qui ne contient que la chaîne de caractères

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

et que *H* se limite à des expressions de la forme `<title>` *C* `</title>`. Ainsi nous arrivons à la conclusion que l'expression

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head><title>Ma page web</title></head>
<body>
Bonjour <a href = "http://science-info-lyca.fr">tout</a>
le <b>monde</b>
```

```
</body>
</html>
```

est une expression bien formée en XHTML.

La sémantique d'une telle expression est d'afficher la page

Bonjour tout le **monde**

dont le titre est `Ma page web` et où un clic sur le lien tout mène à la page `http://science-info-lyca.fr`. Le fait que le texte contenu entre les balises `<a>...</a>` soit cliquable et mène vers une autre page web fait partie de la sémantique du langage XHTML, de même que le texte entre les balises `<title>...</title>` correspond au titre du document.

Toutefois, le *rendu visuel* de ces éléments sémantiques est paramétrable : il existe un langage formel, appelé CSS (*Cascading Style Sheets*) qui permet de définir le rendu de la plupart des balises du langage XHTML et, par exemple, de décider que les liens cliquables sont en bleu au lieu d'être soulignés, ou que les titres de paragraphes doivent être d'une certaine police et taille.

Le langage XHTML, comme le langage Java, est compréhensible par une machine. Un programme qui doit traiter un texte écrit en XHTML ou en Java, comme un navigateur ou un compilateur, commence en général par transformer ce texte en un arbre – cette transformation est l'*analyse syntaxique* du texte –, car un arbre est une structure beaucoup plus facile à traiter qu'un texte linéaire.

Ces langages sont également compréhensibles par les êtres humains, même si nous avons souvent du mal à ne pas oublier une parenthèse ou une balise quand nous cherchons à nous exprimer dans un langage aussi contraint. C'est pour cela que, après avoir débuté comme langage produit par les humains dans les années 1990, (X)HTML est devenu un langage essentiellement produit par des programmes, comme, par exemple, les éditeurs de sites web, ou encore les traitements de texte. En sauvegardant un document édité avec un traitement de texte en format (X)HTML, le document sauvegardé sur le disque est un document texte possédant comme annotations de mise en forme des instructions HTML et CSS. Un tel document peut ainsi être lu par un navigateur, ce qui lui confère une bien meilleure portabilité que l'utilisation d'un format propriétaire.

Exercice corrigé et commenté

Le but de cet exercice est d'écrire un programme qui permet d'extraire le contour d'une image. Par exemple, extraire le contour de l'image

 produit l'image 

Le type des images

Comme nous l'avons vu dans le premier chapitre, une image est un ensemble de pixels. Nous représentons donc une image comme un tableau `tab` à double entrée, l'élément du tableau `tab[x][y]` étant la valeur du pixel de coordonnées `x, y`. Dans cet exercice, nous utilisons des images en noir et blanc, à niveau de gris. Chaque pixel est représenté par un nombre entier compris entre `0` – noir – et une certaine valeur `max` – blanc – qui, en pratique, sera souvent égale à `255`. Une situation plus simple serait d'utiliser des images en noir et blanc mais sans gris, où chaque pixel est représenté par un booléen. Une situation plus complexe serait d'utiliser des images en couleur, où chaque pixel est représenté par un triplet de nombres entiers.

Une image se définit donc comme un enregistrement composé de deux nombres entiers : la largeur et la hauteur de l'image ; un troisième nombre entier : la valeur maximale que peut prendre un pixel ; et enfin le principal : le tableau des pixels.

```
class Image {
 int l;
 int h;
 int max;
 int [][] tab;}
```

Pour fabriquer, transformer et lire une image, nous écrivons trois fonctions. La première prend en arguments trois nombres entiers `l`, `h` et `max`, crée et renvoie une image de largeur `l`, hauteur `h` et valeur maximale `max`. La valeur des pixels de cette image n'est pas spécifiée. En pratique, il est probable que le tableau soit rempli par défaut de `0` et donc que l'image soit entièrement noire.

```
static Image creeImage (int l, int h, int max) {
 Image p = new Image ();
 p.l = l;
 p.h = h;
```

```
 p.max = max;
 p.tab = new int [l][h];
 return p;}
```

La deuxième fonction permet de transformer une image `p` en affectant le pixel de coordonnées `x, y` avec la valeur `k`.

```
static void affectePixel (Image p, int x, int y, int k){
 p.tab[x][y] = k;}
```

La dernière permet d'accéder à la valeur d'un pixel de coordonnées `x, y` dans une image `p`. Si jamais les valeurs `x` et `y` correspondent à un point hors de l'image, la fonction n'échoue pas, mais renvoie la valeur du pixel le plus proche.

```
static int valPixel (Image p, int x, int y) {
 return p.tab[Math.min(Math.max(x,0),p.l-1)]
 [Math.min(Math.max(y,0),p.h-1)];}
```

Dans la suite de l'exercice, nous nous astreindrons à créer, transformer et accéder aux images en utilisant ces trois fonctions uniquement. Ainsi, si nous souhaitons, un jour, transformer la manière dont les images sont représentées, il nous suffira de transformer ces trois fonctions et le reste du programme ne sera pas affecté.

Un exemple d'image

Première question de l'exercice : créer une image, par exemple, l'image ci-avant qui est la réunion du carré dont les points extrêmes sont de coordonnées `50, 50` et `200, 200` et du rectangle dont les points extrêmes sont de coordonnées `150, 100` et `300, 300`. Il faut pour cela créer une image avec la fonction `creeImage`, puis affecter chacun de ses pixels avec la valeur `0` ou `255` selon que le pixel est dans l'un de ces deux rectangles ou non.

```
static Image exempleImage () {
 Image p = creeImage(400,400,255);
 int x = 0; int y = 0;
 for (x = 0; x < 400; x = x + 1) {
 for (y = 0; y < 400; y = y + 1) {
 if (((50 <= x) && (x <= 200) && (50 <= y) && (y <= 200))
    ||
    ((150 <= x) && (x <= 300) && (100 <= y) && (y <= 300)))
 affectePixel(p,x,y,0);
```

```
 else affectePixel(p,x,y,255);}}
 return p;}
```

Pour afficher cette image, nous écrivons une fonction `afficheImage` qui, à nouveau, affiche l'image pixel par pixel.

```
static void afficheImage(Image p) {
 Isn.initDrawing(«Contour»,10,10,p.l,p.h);
 int x = 0;
 int y = 0;
 for (x = 0; x < p.l; x = x + 1) {
 for (y = 0; y < p.h; y = y + 1) {
 Isn.drawPixel(x,y,valPixel(p,x,y));}}}
```

Nous pouvons enfin écrire le programme principal, qui consiste simplement à créer cette image et à l'afficher.

```
public static void main (String [] args) {
 afficheImage(exempleImage());}
```

Extraire le contour

Venons-en maintenant à l'algorithme d'extraction de contour proprement dit. Il s'agit de créer une nouvelle image `q` à partir d'une image `p`. Un point `x, y` est noir dans l'image `q` s'il appartient au contour de l'image `p`, c'est-à-dire s'il est très différent de l'un des deux points situés à sa droite et en dessous de lui. Très différent signifie, par exemple, que la différence de valeur entre ces deux pixels est supérieure à `20`.

```
static Image contour (Image p) {
 int x = 0;
 int y = 0;
 int a = 0;
 int b = 0;
 int c = 0;
 Image q = creeImage(p.l,p.h,255);
 for (x = 0; x < q.l; x = x + 1) {
  for (y = 0; y < q.h; y = y + 1) {
  a = valPixel(p,x,y);
  b = valPixel(p,x+1,y);
  c = valPixel(p,x,y+1);
```

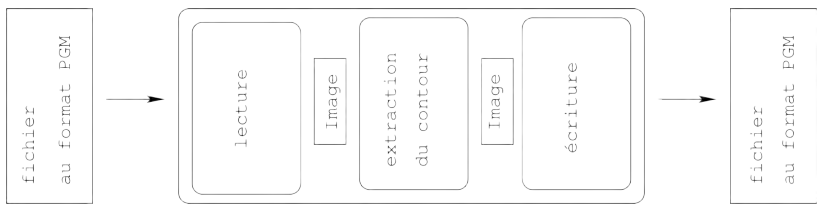
```
  if ((Math.abs(a - b) > 20) || (Math.abs(a - c) > 20))
  affectePixel(q,x,y,0);
  else affectePixel(q,x,y,255);}}
 return q;}
```

Nous pouvons alors transformer notre programme principal pour extraire le contour de notre image.

```
public static void main (String [] args) {
 afficheImage(contour(exempleImage()));}
```

Le format PGM

Avec ce programme, nous pouvons extraire le contour de l'image que nous avons créée, mais pas extraire le contour d'une image créée par d'autres. Pour ce faire, nous devons remplacer la fonction `exempleImage` par une fonction qui crée une valeur de type `Image` à partir d'un fichier qui décrit une image dans un format standard et, de même, remplacer la fonction `afficheImage` par une fonction qui écrit une valeur de type `Image` dans un tel fichier. Ce qui nous donne un programme organisé de la manière suivante



Nous choisissons d'utiliser le format PGM (*portable graymap*). Il existe deux variantes du format PGM : la variante ASCII et la variante brute. Nous choisissons la première. Il est possible de transformer au format PGM des images de formats très variés en utilisant un logiciel de traitement d'images, par exemple le logiciel `gimp`.

Un fichier au format PGM, dans sa variante ASCII, est constitué de

– sur la première ligne, la chaîne « P2 »,

– une ligne de commentaire, entre les caractères # et retour à la ligne,

– une ligne qui contient deux entiers, la largeur `l` et la hauteur `h` de l'image,

– une ligne qui contient un entier, la valeur maximale `max` utilisée pour coder les niveaux de gris,

– `l × h` entiers compris entre `0` et `max` pour chacun des pixels, séparés par des espaces ou des retours à la ligne. L'ordre des pixels est ligne par ligne de haut en bas et de gauche à droite.

En fait, le format PGM est un peu plus général que cela, car il est possible de mettre une ligne de commentaire non seulement à la deuxième ligne, mais n'importe où dans le fichier. En pratique cependant, la plupart des fichiers PGM suivent cette grammaire plus restreinte.

Nous écrivons donc une fonction qui lit une image à ce format

```
static void lisLigne () {
 char c = Isn.readChar();
 while (c != '\n') c = Isn.readChar();}

static Image lisImage () {
 int x = 0;
 int y = 0;
 lisLigne();
 lisLigne();
 int l = Isn.readInt();
 int h = Isn.readInt();
 int max = Isn.readInt();
 Image p = creeImage(l,h,max);
 for (y = 0; y < h; y = y + 1) {
  for (x = 0; x < l; x = x + 1) {
  affectePixel(p,x,y,Isn.readInt());}}
 return p;}
```

Et une autre qui écrit une image à ce format.

```
static void ecrisImage (Image p) {
 int x = 0;
 int y = 0;
 System.out.println("P2");
 System.out.println("#");
 System.out.print(p.l);
 System.out.print(" ");
 System.out.println(p.h);
 System.out.println(p.max);
 for (y = 0; y < p.h; y = y + 1) {
  for (x = 0; x < p.l; x = x + 1) {
 System.out.println(valPixel(p, x,y));}}}
```

Il nous suffit alors de transformer le programme principal.

```
public static void main (String [] args) {
 ecrisImage(contour(lisImage()));}
```

Pour utiliser notre programme avec une image décrite dans un fichier `f1.pgm` et en récupérant l'image produite dans un fichier `f2.pgm`, il suffit de rediriger l'entrée et la sortie du programme, par exemple avec la commande

```
java Contour < f1.pgm > f2.pgm
```

Nous pouvons ensuite visualiser ces images en utilisant n'importe quel logiciel de traitement d'image.

Exercices non corrigés

Compréhension pas à pas du cours. Compréhension du noyau impératif – exercices « papier »

Exercice 1. La valeur des expressions : la fonction Θ

Si `t` et `u` sont deux expressions, on a `Θ(t & u,e,m) = Θ(t,e,m) & Θ(u,e,m)`. Donner de façon similaire :

1. `Θ(t | u,e,m)`,
2. `Θ(t && u,e,m)`,
3. `Θ(t || u,e,m)`.

Exercice 2. L'exécution des instructions : la fonction Σ

Donner, pour chacun des triplets ci-après, la mémoire qui lui est associée par la fonction Σ en appliquant pas à pas les définitions :

1. `Σ(x = 7;,[x = r,y = r'],[r = 5,r'= 6])`,
2. `Σ({x = 7; y = x + 1; y = y + x + 2;},[x = r,y = r',z = r"],[r = 5, r'= 6,r"= 4])`,
3. `Σ(if (x == 5) {y = x + 1; x = 4;} else x = 5;, [x = r, y = r'], [r = 5, r'= 7])`.

Exercice 3. Le test

Donner, pour chacun des triplets ci-après, la mémoire qui lui est associée par la fonction Σ :

1. `Σ(if (x == 1) x = x + 1;,[x = r],[r = 1])`,
2. `Σ(if (x == 1) if (y == 7) x = 2; else x = 3;,[x = r, y = r'], [r = 1, r'= 5])`,
3. `Σ(if (x == 1) {if (y == 7) x = 2;} else x = 3, [x = r, y = r'], [r = 1, r'= 5])`.

Exercice 4. Boucles

1. Soit l'instruction `while (x > 0) x = x -1;`. Écrire explicitement les approximations p_0, p_1, p_2, p_3 et p_4. Calculer `Σ(while (x > 0) x = x-1;,[x = r],[r = 3])`.
2. Calculer
   - `Σ(do x = x-1; while (x > 0),[x = r],[r = 3])`,
   - `Σ(do x = x -1; while (x > 0),[x = r],[r =-2])`,
   - `Σ(while (x > 0) x = x - 1;,[x = r],[r = -2])`.
3. Soit l'instruction `while (x > 0) {while (y < 5) {x = x - 1; y = y + 1;} x = x + 2;}` exécutée dans l'environnement `[x = r, y = r']` et la mémoire `[r = 4, r'= 0]`. Décrire le déroulement de son exécution.

4. Calculer : – Σ({y = 1; for(x = 1;;x < 10; x = x + 1;) y = y*x;}, [x=r, y=r'], [r=3,r'=7]),

– Σ({x = 0; for(i = 1;i <11; i = i + 1;) for(j = 1;j < 6; j = j + 1) x = x + 1;}, [x=r, i=r', j=r''],[r=0, r'=0, r''=0]).

Exercice 5. Programmes équivalents

On considère les deux programmes p1 et p2 suivants dans lesquels b est une expression et q1 et q2 sont des affectations ou des séquences d'affectations. On suppose que les seules variables utilisées dans b, q1 et q2 sont x et y.

```
static public void main (String [] args){ /* p1 */
int x=0;
int y=0;
if (b) q1 else q2}

static public void main (String [] args) { /* p2 */
int x=0;
int y=0;
if (b) q1;
if (!b) q2}
```

Donner des exemples possibles de b, q1 et q2 pour que, après exécution de p1 et p2 dans des environnements identiques :

1. les mémoires obtenues soient identiques,
2. les mémoires obtenues soient différentes.

Fonctions, procédures, récursivité

Exercice 6. Passage de paramètres

Donner les affichages produits lors de l'exécution des programmes suivants – on supposera que l'évaluation des arguments se fait toujours de la gauche vers la droite :

1.
```
class essaiPassage{
static int y=3;

static int g(int u, int v) {
int x=2;
return (x+u+v*v);}

public static void main(string[]args) {
system.out.println (g(1, y));
system.out.println (g(1, 3));}}
```

2.
```
class essaiMasquage {
static int y=1;

static int f (int x) {
 return (x+x);}

static int g(int z) {
 int x=7;
 return (z+x);}

static int h(int z) {
 int x=7;
 return (z+x);}

public static void main (String [] args) {
 System.out.println(f(x));
 System.out.println(g(x) + « « + x);
 System.out.println(h(x) + « « + x);}}
```

3.
```
class Essai-OrdreEvaluation {
static int n = 0;

static void reset () {
 n = 0;}

static int f(int x) {
 n = n+1 ;
 return (x + n);}

static int g (int y) {
 n = 2*n;
 return y + n);}

public static void main (String [] args) {
 reset () ;
 System.out.println (f(4) + g(4));
 reset () ;
 System.out.println (g(4) + f(4));}}
```

Exercice 7. Fonctions récursives

1. On considère la fonction `biz` suivante. Évaluer `biz(25)`, `biz(24)`, `biz(23)`.

L'évaluation termine-t-elle toujours ? Exprimer le détail de la fonction Θ d'évaluation sur l'appel `biz(15)`.

```
static int biz (int x) {
 if (x < 2) return 0;
 else if (x % 5 = 0) return biz (x / 5) ;
 else return (3 + biz(x + 1));}
```

2. Écrire en Java la fonction `L` suivante, qui prend en argument un entier et rend un entier :

```
si x <2
alors L(x) = 0
sinon si x est pair
alors L(x) = 1 + L(x /2)
sinon L(x) = 1 + L((x-1) /2)
```

Décrire l'évolution des mémoires au cours de l'évaluation de `L(37)`.

3. On considère la fonction `Pr` ci-après. Donner l'évolution des mémoires lors de l'évaluation de `Pr(45)` et de `Pr(17)`. Que fait en général cette fonction ?

```
static boolean pr (int n) {
 if (n < 2) return false;
 else return nd(n,2);}

static boolean nd (int n, int k) {
 if (n % k = 0) return false;
 else if ((k+1)*(k+1) <= n) return nd(n, k+1);
 else return true;}
```

Écriture de programmes

Exercice 8. Parcours d'une suite

Dans cet exercice, `l` est une suite d'entiers pour laquelle on considérera deux implémentations possibles : tableau et liste dynamique.

1. Trouver l'élément maximal de `l`.
2. Calculer la somme des éléments de `l`.

3. Calculer la « moyenne olympique » des éléments de `l` – la moyenne de `l` privée de ses éléments de valeur minimale et maximale.

4. Trouver le nombre d'éléments de `l` supérieur à une valeur `k`.

5. Trouver la valeur du « second maximum » de `l` – le plus grand élément de `l` strictement inférieur à son élément maximal.

6. Calculer la longueur de la plus longue sous-suite croissante de `l`.

Exercice 9. Compter les mots

On considère une suite de caractères dans laquelle chaque élément est soit une *lettre*, soit un *séparateur* – espace, tabulation, retour à la ligne – soit une *ponctuation* – virgule, point, etc. On définit un *mot* comme une suite de lettres située entre deux caractères séparateurs ou de ponctuation, exception faite du premier mot du texte, lequel n'est pas nécessairement précédé d'un tel caractère. Une telle suite de caractères peut être obtenue par la lecture, caractère par caractère, d'un fichier texte.

1. Compter les mots du texte – attention, il peut y avoir plusieurs séparateurs entre chaque mot.

2. Calculer la fréquence de chaque mot du texte.

Exercice 10. Interclassement

On considère deux tableaux d'entiers `T1[n1]` et `T2[n2]` dont les valeurs sont triées par ordre croissant : les suites `T1[0], T1[1] … T1[n1-1]` et `T2[0], T2[1] … T2[n2-1]` sont croissantes. En parcourant simultanément ces deux suites, construire dans un tableau `T[n1+n2-1]` la suite croissante obtenue par interclassement des deux suites initiales.

Exercice 11. Schéma de Hörner, changement de base

Un nombre entier naturel `N` est représenté en base `10` par une chaîne de caractères, dont chaque élément est compris entre `'0'` et `'9'`, et terminé par une marque de fin.

1. Écrire une fonction qui convertit une telle chaîne en entier.

2. Afficher la valeur du carré de `N`.

3. Afficher la chaîne de caractères représentant `N` en base `b`, pour `b` donné.

Exercice 12. Chaînes de caractères

On considère deux chaînes de caractères S_1 et S_2, de longueurs respectives l_1 et l_2, représentées dans des tableaux `T1` et `T2`.

1. Déterminer si S_1 est un palindrome.

2. Déterminer si S_1 est une projection de S_2, c'est-à-dire si tous les éléments de S_1 sont présents dans S_2 dans le même ordre.

3. Déterminer si S_1 et S_2 sont des anagrammes.

4. Calculer le nombre minimal de caractères à ajouter à S_1, à n'importe quelle position, pour que S_1 soit un palindrome.

Exercice 13. À propos d'ensembles

On considère le type ensemble d'entiers sur {0, 1, 2, …, N-1} et deux représentations possibles pour ce type :

– Chaque ensemble E est représenté par un tableau de k entiers, où k est le cardinal de E : E est l'ensemble des k éléments du tableau.

– Chaque ensemble E est représenté par un tableau de booléens T[s0..N-1] tel que T[x] est true si et seulement si x ∈ E.

1. Écrire une fonction qui transforme une représentation en l'autre.

2. Écrire dans ces deux cas les fonctions suivantes : initialisation d'un ensemble vide, ajout et suppression d'un élément, test de l'ensemble vide, union et intersection d'ensembles.

Exercice 14. Point fixe d'une suite

Étant donné un entier x, soit d(x) et c(x) les entiers obtenus en ordonnant les chiffres de l'écriture en base 10 de x par ordre décroissant, resp. croissant. On considère la suite récurrente (X_i) suivante :

X_0 est un nombre de N chiffres en base 10.

$X_{i+1} = d(X_i) - c(X_i)$.

On suppose que les nombres s'écrivent toujours avec N chiffres, avec éventuellement des 0 non significatifs en tête. Écrire un programme qui affiche oui ou non selon que la suite X_i comporte ou non un point fixe – X_i comporte un point fixe s'il existe un rang n tel que $X_n = X_{n+1}$.

Questions :

1. L'existence d'un point fixe dépend de la valeur de N – le nombre de chiffres – et de celle de X_0. S'il y a un point fixe, le trouve-t-on au bout d'un temps fini ? S'il n'y a pas de point fixe, comment éviter que le programme boucle et savoir répondre non ?

2. À partir de quelle valeur de N le programme n'aura-t-il pas le comportement attendu ? Pourquoi ?

Exercice 15. Expressions arithmétiques

On considère un arbre binaire a représentant une expression arithmétique :

1. les feuilles sont étiquetées par des nombres entiers,

2. les nœuds internes sont étiquetés par des opérateurs arithmétiques binaires (+, -, *, …).

Écrire une fonction qui calcule la valeur de cette expression arithmétique. Écrire une action qui affiche à l'écran l'expression représentée par a sous différentes formes : préfixée, infixée et postfixée.

Dans un second temps, on pourra chercher à minimiser le nombre de parenthèses en supprimant les parenthèses inutiles.

Questions d'enseignement

La programmation

Un programme est un texte qui exprime un *algorithme* transformant de l'*information* et qui est écrit dans un *langage* de programmation afin d'être exécuté par une *machine*. Clé de voûte où les quatre arcs qui structurent l'informatique se rejoignent, la programmation a naturellement une place privilégiée dans un cursus de découverte de l'informatique.

La programmation est, en outre, un élément de ce cursus souvent apprécié des élèves, car elle les place dans une situation active et créative, dans laquelle ils peuvent eux-mêmes fabriquer un objet. Même si les programmes écrits par les élèves sont de bien plus petite taille que ceux qu'ils utilisent quotidiennement – navigateurs, messageries, jeux vidéo… –, écrire un programme soi-même permet d'extrapoler à partir de sa propre expérience et d'imaginer comment ont été écrits les programmes que l'on utilise. Les élèves acquièrent alors un sentiment d'autonomie, en prenant conscience qu'ils pourraient avoir écrit ces programmes eux-mêmes. Cependant, la programmation est réputée difficile à enseigner.

Tout d'abord, enseigner la programmation demande de choisir un langage. Surviennent alors parfois des querelles de chapelle et de longs débats sur les avantages et inconvénients de tel ou tel langage. Le choix d'un langage mène souvent à prendre conscience du nombre de langages existants et de leur fugacité. La question de l'intérêt d'apprendre un langage particulier et éphémère se pose alors naturellement. Apprendre un langage demande aussi d'apprendre un certain nombre de détails dépourvus d'intérêt, par exemple, que l'affectation s'écrit = et la comparaison ==, à moins que – tout dépend du langage choisi – l'affectation ne s'écrive := et la comparaison =.

Ensuite, comme tout savoir technique, c'est-à-dire qui consiste à mobiliser des connaissances dans le but de fabriquer un objet, la programmation demande du temps pour être apprise.

Enfin, enseigner la programmation demande de trouver un vocabulaire conceptuel adéquat pour verbaliser ce qu'il se passe quand un programme est exécuté. Il faut expliquer aux élèves pourquoi leurs programmes font ce qu'ils font sans personnifier la machine, qui « voudrait » ou « ne voudrait pas » fonctionner comme on s'y attend, et en évitant l'écueil d'un trop grand niveau de détail, en expliquant, par exemple, ce qu'il se passe quand on exécute telle ou telle instruction, en termes de signaux circulant sur un bus ou de transistors basculant d'un état à un autre.

Cependant, avec le temps, des réponses à ces légitimes questions ont été apportées.

Enseigner la programmation v.s. enseigner un langage

La première réponse est sans doute que choisir un langage ou un autre n'est pas si important. À l'exception de langages spécialisés – utilisés par exemple pour écrire des programmes parallèles –, tous les langages sont organisés autour d'un petit nombre de fonctionnalités qui sont les mêmes de l'un à l'autre et qui sont relativement stables dans le temps.

Enseigner la programmation demande de se concentrer sur l'essentiel, c'est-à-dire sur les notions universelles de déclaration, d'affectation, de séquence, de test, de boucle, de fonction, de récursivité, puis, après le lycée, d'enregistrement, d'exception, de module, d'objet… et non sur les bizarreries de tel ou tel langage. C'est seulement ainsi que peut apparaître la portée des notions enseignées.

Décrire la sémantique

La question de la conceptualisation de la sémantique des programmes mène, comme nous l'avons fait dans ce chapitre, à donner un rôle central à la notion d'état et aux idées qu'une instruction est toujours exécutée dans un certain état et que cette exécution produit un autre état. L'exécution d'une instruction transforme un état en un autre état. Autrement dit, il faut passer de l'idée qu'un programme fait quelque chose, à celle qu'il fait quelque chose *à quelque chose*.

La manière dont on décrit les états doit alors être adaptée aux instructions dont on veut décrire la sémantique : une simple structure associant une valeur à chaque variable est suffisante pour expliquer la sémantique de la déclaration, de l'affectation, de la séquence, du test et de la boucle. Décomposer l'état en un environnement et une mémoire est en revanche nécessaire pour expliquer la notion de fonction et le passage d'arguments. Cette notion d'état pourra être introduite informellement ou précisément définie, elle pourra être écrite, dessinée, figurée par des boîtes de carton… ces choix dépendent bien entendu des élèves auxquels on s'adresse, mais, quels qu'ils soient, introduire cette notion d'état et de transformation d'état donne le cadre conceptuel dans lequel exprimer clairement et confortablement ce qu'il se passe quand un programme est exécuté.

Exprimer ainsi la sémantique des instructions mènera naturellement à distinguer les instructions des expressions et les expressions de leur valeur.

Cela mènera aussi naturellement à prendre conscience que les programmes font ce qu'ils font, mais qu'ils pourraient aussi faire autre chose. Par exemple, quand on exécute la séquence `{x = 4; y = x + 1; x = 10;}`, on construit un état dans lequel la variable `y` contient la valeur `5` et non la valeur `11` – l'affectation `x = 10;` ne modifie pas rétroactivement la valeur de `y`. Mais une autre sémantique serait de mettre dans la case `y` non la valeur de l'expression `x + 1` mais l'expression `x + 1` elle-même, si bien que l'affectation `x = 10;` changerait rétroactivement la valeur de `y`. Cette seconde sémantique est différente de la première, mais elle n'est pas absurde, à tel point que c'est la sémantique de quelques langages de programmation, notamment ceux inclus dans les systèmes de calcul formel.

De même, l'évaluation de l'expression booléenne `t & u` consiste à évaluer les expressions `t` et `u` puis à effectuer la conjonction des deux valeurs booléennes obtenues. Une solution alternative serait de commencer par évaluer l'expression `t`. Si cette valeur est `true`, on doit évaluer `u`, mais si c'est `false`, on peut éviter de le faire et renvoyer la valeur `false`. Cette seconde sémantique est différente, car l'évaluation de `u` peut ne pas terminer ou produire une erreur. Si c'est le cas, la première sémantique ne donnera pas de valeur pour l'expression `t & u` alors que la seconde donnera la valeur `false` si l'évaluation de `t` produit la valeur `false`. Cette seconde sémantique n'est cependant pas absurde, à tel point que de nombreux langages de programmation proposent les deux opérateurs – `&` et `&&` en Java.

Enfin, une question difficile est souvent posée par les élèves : un état est-il un état de la machine physique et la sémantique du langage décrit-elle la réalité du processus physique qui se déroule à l'intérieur de la machine ?

La seule réponse honnête à cette question est négative. L'architecture des machines est devenue si complexe que l'explication que l'on donne est de plus en plus éloignée de la réalité. Par exemple, on imagine souvent que lorsque l'environnement associe une référence `r` à une variable `x` et la mémoire la valeur `3` à la référence `r`, cette référence peut être assimilée à une adresse physique dans la mémoire de l'ordinateur. Mais cela est de moins en moins vrai : afin d'optimiser la gestion de la mémoire, il se peut très bien que la valeur `3` soit en fait stockée dans une mémoire située à l'intérieur du processeur ou au contraire dans une mémoire externe ou encore qu'elle migre d'un lieu à un autre au cours de l'exécution du programme. Plutôt que de mentir aux élèves en leur disant que l'on décrit la réalité ou de leur avouer le mensonge, il est plus intéressant de les amener à réfléchir sur la notion d'adéquation d'un modèle à la réalité. Le modèle que constitue la sémantique du langage est adéquat *observationnellement*, c'est-à-dire que *vu*

de l'extérieur tout se passe comme si les choses se passaient comme on les a décrites.

Loin de spécifier la manière dont les choses doivent se dérouler à l'intérieur de la machine, la sémantique d'un langage ne fait que définir un contrat que doivent remplir les concepteurs des compilateurs et des machines. Pour remplir ce contrat, ils ont au contraire toute liberté de s'organiser comme ils le souhaitent.

La classe terminale

La compréhension des constructions des langages de programmation peut grossièrement se décomposer en quatre étapes :

1. le noyau impératif – déclaration, affectation, séquence, test et boucle – les tableaux,
2. les notions de fonction et de récursivité,
3. les enregistrements et les types dynamiques,
4. les modules et les objets.

En arrivant en terminale, les lycéens devraient avoir déjà franchi la première étape, au cours de l'enseignement d'algorithmique inclus dans le cours de mathématiques de seconde et par l'usage de calculatrices programmables – même s'il n'est pas exclu que quelques révisions soient nécessaires.

En ce qui concerne la programmation, le programme de terminale est donc essentiellement centré sur la deuxième étape : les notions de fonctions et de récursivité – les troisième et quatrième étapes étant, quant à elles, hors programme.

La notion de fonction est une notion très riche, mais elle peut s'introduire étape par étape, chacune pouvant être illustrée par un exemple simple

– isolation d'une instruction – fonction `sauterTroisLignes` ci-avant

– passage d'argument – fonction `sauterDesLignes`

– le retour de valeur – fonction `hypotenuse`

– la notion de variable globale et de portée des variables – fonction `reset`

– la différence entre passage d'arguments par valeur et par référence – fonction `echange`

– la récursivité – fonction `fact`.

Les trois premières étapes sont en général rapidement assimilées par les élèves et les trois dernières demandent davantage de temps. La notion de variable globale est d'autant mieux comprise que les notions de déclaration et de portée d'une variable et la distinction entre l'ordre des instructions dans le texte du programme de l'ordre dans lequel elles sont exécutées, sont assimilées – mais précisément elles le sont rarement, car elles ne deviennent

réellement indispensables que lorsque cette notion de fonction est introduite. La différence entre passage d'arguments par valeur et par référence est difficile pour tout le monde, car c'est le premier contact avec la notion de partage. La récursivité est bizarrement vite assimilée par certains et reste longtemps difficile pour d'autres – ce qui reflète sans doute une difficulté pour les seconds à penser localement, sans essayer de comprendre la globalité d'un processus.

Les langages de programmation

Dans un cursus de découverte de l'informatique, les notions relatives aux langages de programmation sont le plus souvent abordées à travers l'activité de programmation elle-même. Il est cependant possible d'aller un tout petit peu plus loin et d'inciter les élèves à réfléchir à un certain nombre de particularités des langages de programmation.

La première particularité de ces langages est qu'ils obéissent à une double contrainte : être compréhensibles par l'être humain qui écrit le programme et par la machine qui l'exécute. Tant que nous n'utilisions pas de machines pour exécuter des algorithmes, nous exprimions ces algorithmes – l'algorithme de la multiplication, le triangle de Pascal, le pivot de Gauss… – en langue naturelle, et les langages de programmation ne sont apparus que lorsque nous avons commencé à utiliser des machines pour exécuter ces algorithmes.

Les premiers langages de programmation, que l'on appelle *langages machine*, étaient surtout compréhensibles par les machines et les êtres humains devaient faire beaucoup d'efforts pour s'y exprimer. Une grande partie de l'histoire de la théorie des langages de programmation peut se raconter comme l'effort de concevoir des langages toujours de plus haut niveau, c'est-à-dire toujours plus faciles à utiliser pour les humains. Certains ont imaginé que concevoir des langages de programmation de plus en plus faciles à utiliser mènerait, *in fine*, à programmer les ordinateurs en langue naturelle. Mais, finalement, il semble que les langues naturelles ne sont peut-être pas si bien adaptées à l'expression des algorithmes.

Par exemple, l'histoire montre qu'au cours des siècles le langage mathématique s'est progressivement éloigné des langues naturelles – la grande rupture dans cette évolution étant l'introduction de la notion de variable, au XVI^{e} siècle. De même, les musiciens ne cherchent pas spécialement à rapprocher la notation musicale d'une langue naturelle. De ce fait, l'usage d'un langage formel pour exprimer des algorithmes est peut-être une bonne chose, indépendamment du fait que ces algorithmes doivent être exécutés par des machines.

Découvrir l'informatique est l'occasion pour les lycéens de prendre conscience de l'importance des langages artificiels – le langage mathématique, la notation musicale, la nomenclature chimique… Cette incursion vers le langage mathématique sera aussi l'occasion de remarquer que la notion de variable d'un langage de programmation a peu à voir avec la notion mathématique de variable.

Dès le lycée, on peut apprendre à distinguer la notion de *syntaxe*, qui décrit la manière dont une instruction s'écrit, de sa *sémantique* qui décrit ce qu'il se passe quand on l'exécute. La syntaxe se divisant elle-même en *syntaxe abstraite,* qui décrit les ingrédients dont une instruction est composée – par exemple un test est toujours composé d'une expression `b` et de deux instructions `p` et `q` –, et *syntaxe concrète* qui décrit comment cette instruction s'écrit littéralement – le test s'écrivant alors `if (b) p else q` ou `if b then p else q` selon les langages.

Dès que l'on a introduit les fonctions et la récursivité, il devient possible d'identifier deux fragments dans le langage que l'on utilise : le fragment impératif – formé de la déclaration, de l'affectation, de la séquence, du test et de la boucle – et le fragment fonctionnel – formé de la déclaration, du test, des fonctions et de la récursivité. Chacun de ces fragments est complet, c'est-à-dire que tous les programmes peuvent s'y exprimer. On peut inciter les élèves à écrire les mêmes petits programmes dans chacun de ces fragments.

Questions d'organisation

Lors de la préparation d'une séance de travaux pratiques, plusieurs questions d'organisation se posent. Premièrement, faut-il donner une ébauche de programme aux élèves, qu'ils doivent alors compléter, ou faut-il ne rien leur donner ? Ici encore, la réponse dépend des élèves auxquels on s'adresse, mais l'expérience semble montrer que, s'il est difficile de ne partir de rien pour écrire un programme, il est plus difficile encore d'insérer son travail dans un programme existant. Il vaut donc mieux, au début, laisser les élèves écrire leurs programmes entièrement, quitte à leur proposer des ébauches quand ils commencent à acquérir un peu de maturité.

Une question liée est celle de l'opportunité de faire travailler les élèves individuellement, deux par deux ou en groupe. Ici encore, il n'y a pas de réponse universelle, mais l'expérience semble montrer qu'écrire un programme en groupe – c'est-à-dire définir des modules que chacun doit écrire et parvenir à assembler ces modules – est difficile pour les débutants. C'est donc un mode d'organisation qu'il vaut mieux n'utiliser qu'avec des élèves qui ont acquis un peu de maturité. En revanche, faire travailler les élèves, y compris débutants, deux par deux présente de nombreux avantages, en particulier celui de les forcer à verbaliser leurs idées avant de les mettre en œuvre.

Enfin, il est important de donner une exigence de résultat élevée aux élèves qui se contentent souvent de tester leurs programmes sur un seul exemple. Leur proposer un jeu de test assez complet, comme les inciter à lire et comprendre les messages d'erreur, à instrumenter leurs programmes avec des impressions à chaque étape, voire à utiliser des outils de mise au point, est un moyen de les mener progressivement à réfléchir sur les questions de qualité du logiciel. De même, rendre les programmes lisibles par une indentation soignée et quelques commentaires doit être valorisé : un programme est écrit une fois, mais lu et modifié de nombreuses fois.

Compléments

Le partage

Comprendre le passage d'argument des fonctions, mais aussi la notion d'enregistrement, nous a menés à introduire une notion de *partage* qui est une notion très générale. On peut la comprendre de manière simple en constatant qu'il se peut qu'un jour il fasse 22 °C à Paris et 22 °C à Rome. Dans ce cas, on pourra dire que la phrase « la température à Rome est identique à la température à Paris » est vraie. Ce même jour, la phrase « la température à Rome est identique à la température dans la capitale de l'Italie » sera vraie également. Mais, bien entendu, ces deux phrases sont vraies pour des raisons tout à fait différentes. Les logiciens expriment cette différence en disant que les expressions « la température à Rome » et « la température à Paris » ont la même dénotation, mais pas le même sens. Alors que les expressions « la température à Rome » et « la température dans la capitale de l'Italie » ont le même sens, et donc la même dénotation – opposition qui est très proche de l'opposition référent/signifié des linguistes. De même, on pourrait dire que deux variables associées, dans l'environnement, à deux références elles-mêmes associées, dans la mémoire, à une unique valeur ont la même dénotation, mais qu'elles n'ont pas le même sens. En revanche, deux variables associées, dans l'environnement, à la même référence ont le même sens, et donc la même dénotation.

Le dilemme partager/recopier se pose de manière constante quand on réalise, par exemple, une page web. Si on veut mettre sur le site web de son lycée les horaires des bus qui permettent d'accéder à l'établissement, on peut mettre un lien vers la page de la compagnie de bus de la ville, ou alors recopier les informations qui se trouvent sur cette page sur une page du site du lycée. Dans le premier cas, les informations ne seront plus accessibles si le

site de la compagnie des bus est réorganisé, en revanche, elles seront automatiquement mises à jour avec celle du site de la compagnie de bus.

États et transitions

Notre description de la sémantique des instructions des langages de programmation nous a menés à introduire une notion d'état et une notion de transition entre états. Par exemple, la sémantique de l'instruction `x = 2 ;` est un ensemble de transitions de l'état `([x = r],[r = 1])` à l'état `([x = r],[r = 2])`, de l'état `([x = r],[r = 4])` à `([x = r],[r = 2])`,...

Ces notions d'état et de transition permettent de décrire formellement de nombreux objets, par exemple les règles de beaucoup de jeux. Au jeu d'échecs par exemple, un état décrit essentiellement la position de chaque pièce sur l'échiquier – c'est, formellement, une relation entre les pièces et les cases de l'échiquier – et les règles définissent des transitions possibles entre états. Dans ce cas, cependant, il faut ajouter quelques informations à la description d'un état : à quel joueur est-ce le tour de jouer ? combien de coups ont-ils déjà été joués depuis le début de la partie ? quels sont les joueurs qui ont déjà roqué ? afin que l'ensemble des transitions possibles à partir d'un état dépende uniquement de cet état, et non de l'histoire qui a mené à cet état.

De même, une tortue graphique se définit par un état – sa position et son azimut – et des transitions – avancer, tourner. Une machine également se définit par un état – la valeur de toutes les mémoires, registres, compteur ordinal – et des transitions qui sont exécutées à chaque cycle de l'horloge.

Un tel système défini par un ensemble d'états et un ensemble de transitions s'appelle un *automate*. Cette notion est fondamentale pour l'étude des langages en général, et de l'interprétation et de la compilation des langages de programmation.

La notion de licence

Avec le développement de logiciels, c'est-à-dire de programmes de grande taille qui demandent donc beaucoup de travail et d'ingéniosité, et l'émergence d'un marché de ces logiciels, est apparue la nécessité de préciser les conditions de leur exploitation par des licences.

L'exploitation des logiciels repose sur un dilemme, qui a son origine dans le fait que les logiciels, biens immatériels, sont des biens *non rivaux* : leur utilisation par une personne n'empêche pas leur utilisation par une autre. De ce fait, si on donne à l'auteur d'un logiciel le monopole de son exploitation et la possibilité d'exploiter ce droit, on limite la diffusion du logiciel

qui pourrait, du fait de sa non-rivalité, être utilisé par tous. À l'inverse, si on lui refuse ce droit, on limite les revenus qu'il peut tirer de son travail. La société doit donc contradictoirement encourager les créateurs et favoriser la diffusion de leur création. Le droit des brevets, dont relèvent les inventions et les activités industrielles, et le droit d'auteur, dont relèvent les arts et les lettres, sont pleinement conscients de ce dilemme puisqu'ils donnent aux inventeurs et aux auteurs le monopole de l'exploitation de leur création et leur permettent de vendre ce droit, mais pour une durée limitée seulement – même si l'on constate une tendance à l'allongement de cette durée.

Second dilemme : la publication du texte du logiciel – du programme source. L'auteur d'un logiciel peut en effet donner à d'autres le droit d'utiliser ce logiciel, sans pour autant en diffuser le source. Cela lui évite de voir ses idées reprises par d'autres. Mais cela limite la possibilité pour d'autres d'améliorer ce logiciel, ou plus simplement de l'examiner pour s'assurer qu'il est conforme à certaines exigences de sûreté et de sécurité – par exemple, qu'il ne laisse pas fuir d'informations confidentielles. Cette question de la publication ne se pose pas dans les domaines des arts et des lettres, où les œuvres ne peuvent pas être exploitées sans être rendues publiques. Dans le domaine des brevets, en revanche, la publication de l'invention est la condition nécessaire à l'acquisition du droit d'exploitation de son invention.

Pour les logiciels, la législation retenue est une forme dérivée du droit d'auteur. En effet, le droit des brevets ne constitue pas un cadre adéquat pour une activité cumulative qui consiste à agencer de nombreuses « briques de base ». Comme il est impossible d'écrire un programme sans reprendre de nombreuses idées déjà formulées par d'autres, le droit des brevets aurait contraint les programmeurs à payer des redevances trop nombreuses. Cela étant, dans certains pays, les logiciels sont brevetables, ce qui pose parfois de sérieux problèmes.

L'idée de propriété intellectuelle remonte au XVI^e^ siècle. Les droits d'auteur sont nés pour répondre à des intérêts de réglementation de la concurrence dans l'édition et l'impression. Dans sa forme juridique moderne, le droit d'auteur a été créé à la veille de la Révolution française par les auteurs de théâtre qui se considéraient spoliés par le monopole d'exploitation de la Comédie-Française. Il n'existe en fait de problème de propriété intellectuelle que là où il y a un marché : Shakespeare, Molière plagiaient sans entraves et la musique d'un de leurs contemporains pouvait librement inclure une appropriation de la musique d'un autre compositeur, ce qui les mènerait aujourd'hui directement au tribunal.

Ces questions du monopole de l'exploitation d'un logiciel et de la diffusion de son source sont celles qui distinguent les deux principales formes de licence utilisées aujourd'hui : les licences *propriétaires* et les licences *libres*. Avec une licence propriétaire, l'auteur du logiciel vend un droit d'utilisation. Il garde le source du logiciel secret afin d'éviter de voir ses idées reprises par d'autres – ce qui peut être considéré comme une régression par rapport au droit des brevets qui impose la publication des inventions. Avec une licence libre, en revanche, l'auteur d'un logiciel donne à ses utilisateurs le droit d'utiliser ce logiciel pour quelque usage que ce soit, d'en étudier le fonctionnement et de l'adapter à ses propres besoins, d'en redistribuer des copies sans limitation, de le modifier et diffuser la version modifiée. L'auteur doit donc donner accès au source de son logiciel.

À première vue, l'auteur d'un logiciel libre semble abandonner la possibilité de voir son travail rémunéré et, de fait, certains logiciels libres ne rapportent aucun revenu à leurs auteurs – c'est par exemple le cas de nombreux logiciels développés par des chercheurs dans les universités. Cependant, il est aussi fréquent pour une entreprise de diffuser un logiciel librement, tout en développant un modèle commercial autour de cette libre diffusion, qui consiste souvent en la vente de biens complémentaires au logiciel. Ainsi, certaines entreprises diffusent librement et gratuitement un logiciel, tout en faisant payer une activité de conseil ou de formation à l'utilisation de ce logiciel. Dans d'autres cas, enfin, une entreprise, ou plus souvent un groupement d'entreprises, paie une autre entreprise pour développer un logiciel libre, sans espoir d'autre retour que celui de l'utilisation du logiciel.

Parmi les licences libres, on en distingue de deux types. Les licences *copyleft*, comme la *General Public Licence* (GPL) ou certaines versions de la licence *CEA CNRS INRIA Logiciel Libre* (CeCILL), imposent que toutes les versions diffusées du logiciel, modifiées ou non, soient distribuées sous la même licence, afin de faire bénéficier les autres des mêmes libertés que celles dont on a soi-même bénéficié – la licence CeCILL résout les problèmes posés en droit français par la licence GPL. Cela empêche que le cycle de la vertu ne soit interrompu par une appropriation privée du logiciel. D'autres licences, comme *Berkeley Software Distribution* (BSD), donnent au contraire la liberté aux utilisateurs du logiciel de le modifier et de diffuser la version modifiée sous une licence propriétaire.

Les licences libres ont permis un nouveau mode de développement des logiciels, par une démarche coopérative. Le système d'exploitation Linux, par exemple, a été développé par des centaines de programmeurs aux quatre coins du monde, qui ont pu travailler ensemble, à cette échelle, grâce au

réseau. Cette démarche, qui présente un certain nombre d'analogies avec la recherche scientifique – coopération internationale, publication, validation par les pairs, liberté de critiquer et d'amender… alors que, dans les temps anciens, Pythagore interdisait à ses disciples de divulguer théorèmes et démonstrations –, participe de l'émergence d'une nouvelle forme d'intelligence collective, qui répond à un besoin créé par l'augmentation de la complexité des objets industriels – développer un logiciel de plusieurs millions de lignes n'est plus à la portée d'une unique équipe de développeurs ou d'une unique entreprise.

Au-delà des logiciels, des licences libres existent aussi pour d'autres contenus immatériels : encyclopédies, ressources pédagogiques… Elles constituent des réponses en termes de droit d'auteur ayant l'objectif de fluidifier la circulation des documents et de faciliter le travail en commun. Elles sont des adaptations du droit des auteurs qui fournissent un cadre juridique au partage sur le Web des œuvres de l'esprit. À la manière de la GPL, les licences *Creative Commons* renversent le principe de l'autorisation obligatoire. Elles permettent à l'auteur d'autoriser par avance, et non au coup par coup, certains usages et d'en informer le public. Les droits donnés doivent correspondre à la nature de la ressource. Autant on peut enrichir collectivement un scénario de travaux pratiques au lycée, et donc permettre de le modifier, autant modifier un article d'opinion n'a pas de sens. Méta-licence, *Creative Commons* permet aux auteurs de se fabriquer des licences, dans une espèce de jeu de Lego simple constitué de seulement quatre briques. Première brique, Attribution : l'utilisateur, qui souhaite diffuser une œuvre, doit mentionner l'auteur. Deuxième brique, Commercialisation : l'auteur indique si son travail peut faire l'objet ou pas d'une utilisation commerciale. Troisième brique, Non-dérivation : un travail, s'il est diffusé, ne doit pas être modifié. Quatrième brique, Partage à l'identique : si l'auteur accepte que des modifications soient apportées à son travail, il impose que leur diffusion se fasse dans les mêmes termes que l'original, c'est-à-dire sous la même licence.

Cette nouvelle manière de produire et de diffuser des objets industriels préfigure peut-être une évolution plus globale de l'industrie et de la notion de propriété, dans un monde dans lequel de plus en plus de biens sont complexes et immatériels.

Pour aller plus loin

Poursuivre la description des fonctionnalités des langages de programmation donnée dans ce chapitre nous mènerait à aborder d'autres fonction-

nalités : en particulier, la notion d'exception et la notion d'objet. La notion d'objet est relativement facile à comprendre une fois les notions de fonction et d'enregistrement assimilées, car un objet est essentiellement un enregistrement dont certains champs sont des fonctions. L'exemple le plus simple d'objet est un enregistrement à quatre champs : un champ `val` qui est un entier, un champ `print` qui est une fonction qui prend en argument un enregistrement et affiche la valeur de son champ `val`, un champ `reset` qui prend en argument un enregistrement et affecte la valeur `0` à son champ `val` et un champ `add` qui prend en argument un enregistrement et ajoute `1` à son champ `val`.

Ainsi, si `t` est un tel objet, le programme

```
t.reset(t);
t.add(t);
t.add(t);
t.print(t);
```

affiche la valeur `2`. Cet objet est un compteur, mais contrairement à un enregistrement qui aurait un unique champ `val`, il « sait » « lui-même » comment se réinitialiser, s'ajouter `1` ou s'afficher, puisque ces fonctions font partie de sa propre définition.

Bien souvent, les langages de programmation proposent une syntaxe spéciale pour appliquer un champ fonctionnel d'un objet à cet objet même et, dans la définition d'une telle fonction, une syntaxe spéciale pour désigner l'objet lui-même – un pronom réfléchi.

Les exceptions, les objets sont, par exemple, décrits, dans un style proche de celui adopté dans ce chapitre, dans

Gilles Dowek, Les principes des langages de programmation, Les éditions de l'École polytechnique, 2008.

Poursuivre l'étude des langages de programmation nous mènerait à aborder des langages plus spécialisés qui permettent par exemple de programmer des machines parallèles – des ordinateurs constitués de plusieurs petits ordinateurs qui effectuent des calculs en même temps. Un cas particulier est celui où chaque machine n'a qu'un nombre fini d'états possibles et, à chaque instant, évolue d'un état à un autre. Ces « systèmes dynamiques discrets » peuvent être *asynchrones*, ce qui signifie que chacun calcule à son rythme, ou *synchrones*, ce qui signifie qu'ils partagent une horloge commune. Entrent dans ce cadre les automates cellulaires, les réseaux de neurones, les pro-

grammes écrits dans des langages synchrones… Ces questions sont abordées dans

`http://fr.wikipedia.org/wiki/Automate_cellulaire` et `http://en.wikipedia.org/wiki/Esterel`.

Un dernier aspect de cette étude concerne la manière dont on réalise des interpréteurs et des compilateurs pour des langages que l'on conçoit soi-même. Ces questions sont abordées dans

Gilles Dowek, *Introduction à la théorie des langages de programmation,* Les éditions de l'École polytechnique, 2006.

Algorithmique

Troisième étape de notre parcours : la notion d'algorithme. Cette notion peut être vue comme une abstraction de celle de programme : la même manière de trier une liste, d'ajouter deux nombres ou de compresser un fichier peut s'exprimer dans différents langages de programmation. Il existe donc un objet abstrait, qui s'incarne dans ces programmes écrits dans différents langages. C'est cet objet que l'on appelle un algorithme. Un algorithme peut se définir comme une manière particulière d'enchaîner des actions élémentaires pour résoudre toutes les instances d'un problème donné. L'autonomie de cette notion est renforcée par l'observation que, dans l'histoire de l'humanité, comme dans notre histoire personnelle, il y a de nombreux algorithmes que nous avons su exécuter bien avant de savoir les exprimer dans un langage de programmation, et même avant de savoir les verbaliser. Il y a aussi de nombreux algorithmes qui s'exécutent dans notre cerveau, sans que nous sachions complètement les expliquer. Il y en a d'autres, comme la recette de la ratatouille, que nous avons fini par réussir à verbaliser.

Cours

Considérons la recette de cuisine *Ratatouille niçoise*

« Ingrédients (pour deux personnes) : 1 oignon, 1 à 2 poivrons, 1 aubergine, 2 courgettes, 1 tomate, 1 gousse d'ail, 2 cuillères d'huile d'olive.

« Éplucher l'oignon. Le couper en petits dés.

« Chauffer l'huile d'olive dans un faitout.

« Ajouter les oignons.

« (Pendant que les oignons blondissent) couper les poivrons en petits dés. Les ajouter dans le faitout et remuer.

« Éplucher l'aubergine, la couper en petits dés, ajouter et mélanger.

« Éplucher et couper les courgettes, couper en rondelles, ajouter et mélanger.

« Laisser cuire quelques minutes.
« Couper la tomate en petits dés. Écraser l'ail. Les ajouter et remuer.
« Laisser cuire quelques minutes et régalez-vous. »

Cette recette semble à première vue une description bien précise de la préparation d'une ratatouille niçoise. Mais il y a beaucoup de non-dits.

– Bien entendu, il faut savoir ce qu'est un oignon – de quelle variété ? est-ce que cela a une importance ? et de même pour les autres ingrédients. Nous verrons qu'il en sera toujours ainsi : il y a des notions primitives et on ne peut pas faire sans.

– Un algorithme doit être bien défini. Faut-il un ou deux poivrons ? Cela dépend évidemment de leur taille. Toute ménagère apprendra ce qu'il en est. N'empêche qu'on laisse ici une latitude à l'exécutant : cela pourra aussi se produire parfois dans les algorithmes – voir les algorithmes 13 et 14. Un algorithme qui à certains moments laisse le choix entre plusieurs actions est dit *non déterministe*.

– On parle de « remuer » et « mélanger ». On emploie deux mots ayant ici la même signification pour éviter de se répéter.

– Les temps de cuisson de chacun des cinq légumes sont différents, ce qui explique qu'on les ajoute les uns après les autres. À part cela, la façon de les traiter est la même. Nous avons vu au chapitre « Langages et programmation » comment la notion de *fonction* permet de ne pas répéter cinq fois la même chose.

Nombre de théorèmes et formules mathématiques sont en fait des algorithmes, par exemple :

– les formules qui permettent de calculer une aire ou un périmètre ;
– la méthode d'addition des entiers en numération positionnelle décimale ;
– la résolution d'une équation du second degré – qui introduit un *test* ;
– le calcul du plus grand diviseur commun de deux nombres – on ne se contente en général pas de l'algorithme naïf ;
– l'établissement d'une table de logarithmes – algorithme où l'on voit apparaître la possibilité d'erreurs dues à des causes diverses lors de l'exécution à la main.

Ce dernier exemple nous rappelle aussi que, avant l'apparition des ordinateurs, les besoins en calcul avaient mené à la création de *bureaux de calcul*.

Algorithmes de tri

Parmi les nombreux problèmes qui peuvent se poser à la sagacité humaine, certains sont des *problèmes algorithmiques* : il s'agit de décrire très précisément une suite d'actions à effectuer pour obtenir la solution au problème.

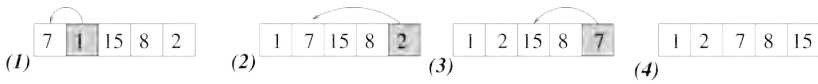
Considérons un premier problème algorithmique, celui du *tri* : il s'agit, étant donné une suite de n nombres, de les ranger par ordre croissant. La suite

de n nombres est représentée par un tableau de n nombres et on suppose que ce tableau est entièrement en machine.

Si nous considérons une petite suite, par exemple 7, 1, 15, 8, 2, il est évidemment facile de l'ordonner pour obtenir la suite 1, 2, 7, 8, 15. Mais, quand il s'agit d'une suite de plusieurs centaines, voire millions d'éléments, cela est nettement moins évident. Nous avons besoin d'une stratégie et, à partir d'une certaine taille, d'un outil pour réaliser cette stratégie à notre place, car nous ne pouvons plus le faire à la main. Cette stratégie s'appelle un *algorithme*. L'outil utilisé pour réaliser l'algorithme est de nos jours souvent un ordinateur.

Le tri par sélection

Le premier algorithme auquel on pense généralement pour trier un tableau s'appelle le *tri par sélection* : on recherche le plus petit élément parmi les n éléments et on l'échange avec le premier élément ; puis on recherche le plus petit élément parmi les $n - 1$ derniers éléments de ce nouveau tableau et on l'échange avec le deuxième élément ; plus généralement, à la k-ième étape, on recherche le plus petit élément parmi les $n - k + 1$ derniers éléments du tableau en cours et on l'échange avec le k-ième élément.



Comment l'algorithme TRI-SÉLECTION trie le tableau 7, 1, 15, 8, 2.

On peut formaliser cet algorithme ainsi.

Algorithme 1: Le tri par sélection

```
TRI-SÉLECTION(tableau T, entier N)
  Données : Un tableau T de N éléments comparables
  Résultat : Le tableau T contient les mêmes éléments mais rangés par
             ordre croissant
  Indice entier i, j, min
1 for i = 1 to N − 1
2 |   min = i   // On initialise l'indice du minimum des N − i + 1
  |  derniers éléments
3 |  for j = i + 1 to N
4 |  |   if (T[j] < T[min])
  |  |   |  min = j  // On actualise l'indice du minimum en cours
5 |  échange (T, i, min)          // min est l'indice du plus petit
```

```
    élément compris entre les indices i et N inclus et on
    permute les éléments d'incices i et min
    /* Le tableau T contient les i plus petits éléments du
       tableau initial rangés aux places 1...i                */
```

Cette manière de décrire les algorithmes rappelle les langages de programmation, tels ceux que nous avons rencontrés au deuxième chapitre. Toutefois, cette notation est un peu moins contraignante que celle d'un langage de programmation, car elle n'est pas destinée à être exploitée par des machines. C'est avant tout un moyen de communication entre des êtres humains. On appelle une telle notation un *pseudo-code*. Dans le pseudo-code que nous utilisons ici, nous notons **for** *i* = *t* **to** *u* **p** la boucle qui se noterait `for (i = t; i <= u; i = i + 1) p` en Java.

Pour montrer en quoi cet algorithme naïf n'est pas suffisant pour les applications courantes, en particulier en gestion, intéressons-nous à la « complexité » de notre algorithme. On peut évaluer la complexité d'un algorithme de plusieurs manières :

1. complexité en espace, par exemple nombre de variables, taille des valeurs, place mémoire…

2. complexité en temps : le temps d'exécution $t(n)$ du programme dépendra en général de la taille n des données d'entrée ; on distinguera :

– la complexité moyenne, c'est-à-dire la valeur moyenne des $t(n)$: elle est en général difficile à évaluer, car il faut commencer par décider quelle donnée est « moyenne », ce qui demande un recours aux probabilités ;

– la complexité pour le pire des cas, c'est-à-dire pour la donnée d'entrée donnant le calcul le plus long, soit $t(n)$ maximal ;

– la complexité pour le meilleur des cas, c'est-à-dire pour la donnée d'entrée correspondant au calcul le plus court, soit $t(n)$ minimal.

L'algorithme TRI-SÉLECTION trie les éléments du tableau dans le tableau lui-même et n'a besoin d'aucun espace supplémentaire. Il est donc très économe en espace car il fonctionne en *espace constant* : on trie le tableau sur place, sans avoir besoin de stocker une partie du tableau dans un tableau auxiliaire.

En revanche, il n'est pas économe en temps : en effet, les lignes 1 et 3 sont deux boucles **for** imbriquées,

– la boucle de la ligne 1 doit être exécutée $N - 1$ fois, et elle comprend l'instruction 2 et la boucle **for** de la ligne 3 ;

– la boucle de la ligne 3 effectue $N - 1$ comparaisons, à la ligne 4, à sa première exécution, puis $N - 2$, $N - 3$, etc. Le nombre d'opérations effectuées par l'algorithme est donc de $2(N-1) + (N - 1) + (N - 2) + \ldots + 1 = N - 2 + N(N$

+ 1)/2, qui est de l'ordre de N^2. Nous verrons plus loin que l'on peut trier un tableau en faisant moins d'opérations.

Ce premier exemple d'algorithme nous a permis de voir comment formaliser un algorithme. Nous pouvons penser avoir ainsi résolu le problème algorithmique posé. Prenons cependant un cas concret dans lequel nous avons besoin de trier de grands tableaux. On effectue en France de l'ordre de cent millions d'opérations bancaires par jour – ce qui ne fait jamais que 1.5 par habitant mais les entreprises en effectuent beaucoup plus que les particuliers. Un organisme national, la Banque de France, a besoin d'effectuer les débits et les crédits chaque jour entre les différentes banques. Cet organisme commence par trier ces opérations, par numéro de compte. Avec le tri par sélection, cela prendrait environ 1 000 jours – voir l'exercice 1. Il faut donc concevoir des algorithmes plus efficaces. Par exemple, le *tri fusion*, que nous présenterons ci-après, traitera le même nombre de transactions en quelques dizaines de secondes.

Le tri fusion

Le *tri fusion* utilise une stratégie différente : on divise le tableau à trier en deux parties de tailles à peu près égales, que l'on trie, puis on interclasse les deux tableaux triés ainsi obtenus. La stratégie sous-jacente est du type *Diviser pour régner* : on divise un problème sur une donnée de « grande taille » en sous-problèmes de même nature sur des données de plus petite taille, et on applique *récursivement* cette division jusqu'à arriver à un problème de très petite taille et facile à traiter ; ensuite on recombine les solutions des sous-problèmes pour obtenir la solution au problème initial. Pour le tri fusion, la très petite taille est 1, dans le cas d'un tableau de taille 1, il n'y a rien à trier.

L'algorithme 2 TRI-FUSION fait appel, à la ligne 6, à l'algorithme 3. Il se présente ainsi.

Algorithme 2 : Le tri fusion

```
TRI-FUSION (tableau T, entier N, l, r)
  Données : Un tableau T de N entiers indicés de l à r
  Résultat : Le tableau T contient les mêmes éléments mais rangés par
             ordre croissant
1 Indice entier m
2 if l < r
3 |  m = ⌊(l + r)/2⌋   // On calcule l'indice du milieu du tableau
4 |  TRI-FUSION(T, l, m)
5 |  TRI-FUSION(T, m + 1, r)
6 |  fusion(T, l, r, m)
```

Algorithme 3: Fusion de deux tableaux

```
FUSION (tableau T, entier l, r, m)
  Données : Un tableau T d'entiers indicés de l à r, et tel que l ≤ m < r et
            que les sous-tableaux T[l...m] et T[m+1...r] soient ordonnés
  Résultat : Le tableau T contenant les mêmes éléments mais rangés par
             ordre croissant
1 Indice entier i, j, k, n1, n2
2 Var tableau d'entiers L, R
3 n1 = m − l + 1
4 n2 = r − m
  /* On utilise des tableaux temporaires L, R et une allocation
     par blocs L = T[l...m] et R = T[m+1...r]                    */
5 for i = 1 to n1
6 |  L[i] = T[l + i − 1]
7 for j = 1 to n2
8 |  R[j] = T[m + j]
9 i = 1
10 j = 1
11 L[n1 + 1] = ∞                  // On marque la fin du tableau gauche
12 R[n2 + 1] = ∞                  // On marque la fin du tableau droit
13 for k = l to r
14 |  if (L[i] ≤ R[j])
15 |  |  T[k] = L[i]
16 |  |  i = i + 1
17 |  else
18 |  |  T[k] = R[j]
19 |  |  j = j + 1
```

L'algorithme TRI-FUSION présente plusieurs caractères intéressants.

– C'est un algorithme *récursif*: cela signifie que pour exécuter TRI-FUSION(*T*, *l*, *r*), il faut appeler la même fonction, avec d'autres paramètres, TRI-FUSION(*T*, *l*, *m*) et TRI-FUSION(*T*, *m* + 1, *r*). Nous verrons plus loin d'autres algorithmes récursifs, le tri rapide par exemple; l'avantage des algorithmes récursifs est en général la grande facilité avec laquelle on peut les écrire, car, souvent, l'algorithme récursif correspond de très près à l'énoncé du problème à résoudre. La difficulté est le choix des paramètres des appels récursifs qui doit à la fois accélérer le temps de calcul et ne pas introduire une non-terminaison de l'algorithme – voir l'exercice non corrigé 1.

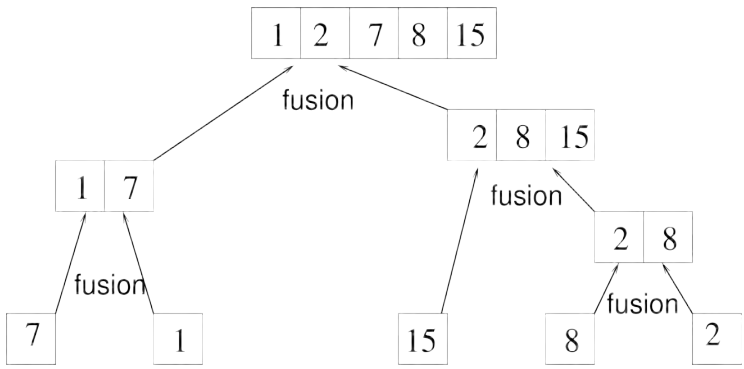
– C'est un algorithme qui emploie la stratégie *Diviser pour régner*, c'est-à-dire:

1. on *divise* le problème en sous-problèmes identiques mais sur des données de plus petite taille; ici on divise le tableau en deux;

2. on *résout* les sous-problèmes récursivement ; si leur taille est suffisamment petite, la solution est immédiate ; ici, ordonner un tableau à un élément ;

3. on *combine* les solutions des sous-problèmes pour obtenir la solution du problème initial ; ici, la fusion de deux tableaux déjà ordonnés.

La figure ci-après illustre l'algorithme 2.



Comment TRI-FUSION trie le tableau 7, 1, 15, 8, 2.

La fonction FUSION prend un tableau de nombres indicés de l à r, qui est divisé en deux parties triées – de l à m, et de $m + 1$ à r. Pour faire cette fusion, on copie dans deux tableaux annexes L et R les deux parties à trier ; puis on interclasse L et R : on les parcourt et on choisit à chaque étape le plus petit des deux nombres pour le mettre dans T. Lorsque l'une des parties L ou R est vide, on recopie l'autre dans T. Pour simplifier l'écriture de l'algorithme 3, on a placé à la fin des tableaux L, R une *sentinelle* ∞, aux lignes 11 et 12, plus grande que tous les nombres présents, et qui évite de tester si L ou R est vide.

L'algorithme TRI-FUSION s'exécute en *espace linéaire* : il faut recopier tout le tableau puisque l'on doit avoir une copie pour pouvoir interclasser. Cela est moins bon que l'espace constant. En revanche, sa complexité en temps est excellente. Soit $t(n)$ le nombre d'opérations pour trier un tableau de taille n : la fusion des deux moitiés de tableau prend un temps n, et le tri des deux moitiés de tableau prendra un temps $2t(n/2)$. Donc $t(n)$ satisfait l'équation de récurrence

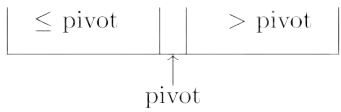
$$t(n) = 2t(n/2) + n$$

comme $t(1) = 1$, on en déduit $t(n) = O(n \log n)$.

Pour le problème des transactions bancaires, on passe d'un temps $O(100000000^2)$ à un temps $O(100000000 \times \log 100000000) = O(100000000 \times 8 \log 10) = O(100000000 \times 10)$, on a donc divisé le temps par un facteur 10^7, ce qui fait passer de 385 jours – voir l'exercice 1 – à environ 10 secondes.

Le tri rapide

Donnons l'idée de l'algorithme TRI-RAPIDE : pour trier un tableau de longueur n, on choisit un élément p, appelé *pivot*, dans ce tableau et on permute 2 à 2 les éléments du tableau de façon à regrouper en début de tableau tous les éléments inférieurs ou égaux au pivot, et en fin de tableau tous les éléments strictement supérieurs au pivot. On met le pivot à sa place, c'est-à-dire entre les éléments qui lui sont inférieurs et ceux qui lui sont supérieurs, et on recommence l'opération avec la partie de tableau contenant les éléments inférieurs au pivot et avec la partie de tableau contenant les éléments supérieurs.



La place du pivot.

Il s'agit ici encore d'une stratégie de type *Diviser pour régner*.

Soit $T[i\ldots j]$ le tableau des éléments que l'on veut trier, l'algorithme que l'on vient de décrire – et qui fait appel à une fonction `SEGMENTE` décrite dans l'algorithme 5 ci-après – s'écrit ainsi.

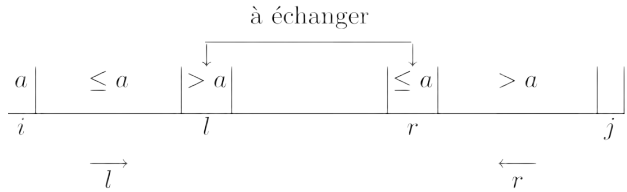
Algorithme 4: Le tri rapide

```
TRI-RAPIDE (tableau T, entier i, j)
  Données : Un tableau T d'éléments indicés de i à j
  Résultat : Le tableau T contenant les mêmes éléments mais rangés par
             ordre croissant
1 Indice entier j
2 if (i < j)
3 |  k =SEGMENTE(T, i, j)            // T(k) est à sa place finale
4 |  TRI-RAPIDE(T, i, k − 1)      // trie les éléments de i à k − 1
  |  inclus
5 |  TRI-RAPIDE(T, k + 1, j)      // trie les éléments de k + 1 à j
  |_ inclus
```

L'algorithme TRI-RAPIDE est aussi un algorithme récursif et il emploie également une stratégie de type *Diviser pour régner*; on remarquera l'élégance et la concision de l'algorithme 4 : élégance et concision sont en général le propre des algorithmes récursifs.

L'algorithme SEGMENTE choisit pour pivot le premier élément du tableau $T[i]$ et permute les éléments de $T[i\ldots j]$ jusqu'à ce qu'à la fin $T[i]$ soit à

sa place définitive dont l'indice est k, et tous les $T[j] \leq T[i]$ soient à gauche de $T[i]$ et tous les $T[j] > T[i]$ soient à la droite de $T[i]$. L'algorithme SEGMENTE utilise deux compteurs courants pour les $T[j]$, l'un croissant, l, l'autre décroissant, r : l croît jusqu'à ce que $T[l]$ soit plus grand que le pivot, r décroît jusqu'à ce que $T[r]$ soit plus petit que le pivot et à ce moment on permute $T[l]$ et $T[r]$ et on continue ; ensuite, au moment où r et l se croisent, c'est-à-dire quand l devient supérieur ou égal à r, on a terminé et on peut placer le pivot en k qui est sa place finale, en échangeant $T[i]$ et $T[k]$ – par l'appel de fonction `échange`(T, i, k) – et $T[k]$ se retrouve à sa place définitive.



Un échange `échange`*(T, l, r)*

Algorithme 5: Segmentation d'un tableau

SEGMENTE (tableau T, entier i, j)
Données : Un tableau T d'éléments indicés de i à j
Résultat : Le tableau T et l'indice k : T contient les mêmes éléments mais $T[i]$ est placé à sa place finale k, et tous les nombres à gauche de l'indice k sont plus petits que $T[i]$, et tous les nombres à droite de l'indice k sont plus grands que $T[i]$

```
1  Indice entier k, l, r
2  Var élément p
3  l = i + 1
4  r = j
5  p = T[i]
6  while (l ≤ r)
7  |  while (T[r] > p)
8  |  |  r = r − 1
9  |  while (T[l] ≤ p)
10 |  |  l = l + 1
11 |  if (l < r)
12 |  |  échange (T, l, r)
   |  |                    // On échange les valeurs de (T[l] et T[r])
13 |  |  r = r − 1
14 |  |  l = l + 1
15 k = r
```

```
échange (T, l, k)
retourner k
```

On peut étudier l'exemple du tri du tableau 100, 202, 22, 143, 53, 78, 177. Les pivots sont soulignés et les sous-tableaux des éléments ≤ k et des éléments > k qui restent à trier sont en caractères gras. La partie de tableau qui sera traitée à la ligne suivante est entre crochets […]. La première colonne donne l'appel de fonction dont le résultat est écrit sur la même ligne.

résultat de							
	[<u>100</u>	**202**	**22**	**143**	**53**	**78**	**177]**
SEGMENTE(T,1,7)	**[<u>53</u>**	**78**	**22]**	100	**143**	**202**	**177**
SEGMENTE(T,1,3)	**22**	53	**78**	100	**[<u>143</u>**	**202**	**177]**
SEGMENTE(T,5,7)	22	53	78	100	143	**[<u>202</u>**	**177]**
SEGMENTE(T,6,7)	22	53	78	100	143	**[177]**	202

L'algorithme TRI-RAPIDE s'exécute en *espace constant* : on n'a pas besoin de recopier le tableau à trier, en revanche il faut stocker les appels récursifs de TRI-RAPIDE, et il peut y avoir $O(n)$ appels récursifs à TRI-RAPIDE.

Pour la complexité en temps, comptée en nombre de comparaisons d'éléments :

– Le *pire cas* est obtenu si le tableau de départ est déjà trié et chaque appel de SEGMENTE ne fait que constater que le premier élément est le plus petit du tableau. Soit $p(n)$ la complexité la pire pour trier un tableau de longueur n, on a donc

$$p(n) = n - 1 + p(n - 1) \qquad (3.1)$$

avec $p(2) = 3$. On écrit les égalités 3.1 pour $i = 2,\ldots, n$, on somme ces égalités, et on en déduit

$$\forall n \geq 2, \quad p(n) = \frac{n(n-1)}{2} - 3$$

c'est-à-dire une complexité quadratique, la même que pour le tri par sélection.

– La *meilleure complexité* en temps pour TRI-RAPIDE est obtenue lorsque chaque appel récursif de TRI-RAPIDE se fait sur un tableau de taille la moitié du tableau précédent, ce qui minimise le nombre d'appels récursifs ; soit $m(n)$ la complexité la meilleure pour trier un tableau de longueur $n = 2^p$, on a donc

$$m(n) = n + 2m(n/2)$$

avec $m(2) = 2$, d'où $m(n) = n \log(n)$.

– La *complexité moyenne* de TRI-RAPIDE est plus difficile à calculer ; on montre que la moyenne de la complexité en temps sur toutes les permutations possibles est de l'ordre de $O(n \log n)$, la même que pour le tri fusion. La complexité en moyenne est donc égale à la complexité dans le meilleur cas ; de plus, c'est la meilleure complexité possible pour un algorithme de tri, ce qui nous permet de dire que TRI-RAPIDE est un bon algorithme.

Le tableau ci-après résume les complexités des algorithmes de tri que nous avons vus.

complexité	espace	temps		
		pire	moyenne	meilleure
tri sélection	constant	n^2	n^2	n^2
tri fusion	linéaire	$n \log n$	$n \log n$	$n \log n$
tri rapide	constant	n^2	$n \log n$	$n \log n$

Il y a aussi d'autres méthodes de tri : nous verrons un peu plus loin, au paragraphe « Arbres binaires », qu'en organisant les données sous la forme d'un arbre binaire de recherche, on peut construire l'arbre au fur à mesure qu'on lit les données, et ce de telle sorte qu'un parcours judicieusement choisi dans l'arbre donne très rapidement le tableau de données trié.

Algorithmes de recherche

Le problème est de rechercher des informations dans une table – qui peut être une liste, un tableau, un arbre, etc. Dans la suite, nous supposerons que :

1. les tables sont des tableaux,

2. les tableaux contiennent des nombres – on pourrait traiter de la même façon des enregistrements quelconques, par exemple formés d'un numéro de Sécurité sociale, d'un nom et d'une adresse, ou alors d'un nom et d'un numéro de téléphone.

Recherche séquentielle

Cette méthode simple consiste à parcourir le tableau à partir du premier élément et à s'arrêter dès que l'on trouve l'élément cherché – on ne cherche que la première occurrence d'un élément. Soit T un tableau de n éléments et k l'élément que l'on recherche.

Algorithme 6: Recherche séquentielle

```
RECHERCHE (tableau T, élément k)
  Données : Un tableau T de n éléments et un élément k
  Résultat : Le premier indice i où se trouve l'élément k si k est dans T,
             et sinon la réponse« k n'est pas dans T »
1 Indice entier i
2 i = 1
3 while ((i ≤ n) ∧ (T[i] ≠ k))
  |  // voir le paragraphe 3.4, questions d'implémentations
4 |  i = i + 1
5 if (i ≤ n)
6 |  afficher T[i] = k
7 else
8 |  afficher k n'est pas dans T
```

La complexité en temps de RECHERCHE est linéaire, de l'ordre de n, puisqu'il faudra au pire parcourir tout le tableau.

Recherche dichotomique

Cette méthode s'applique si le tableau est déjà trié et s'apparente alors à la technique *Diviser pour régner*. Elle suppose donc

1. que les éléments du tableau sont comparables,

2. un *prétraitement* éventuel du tableau où s'effectue la recherche, qui consiste à trier ce tableau.

Il faut certes ajouter le temps du précalcul au temps de l'algorithme de recherche proprement dit, mais si l'on fait plusieurs recherches dans un même tableau, le précalcul est effectué une unique fois.

Soit T un tableau déjà trié de n nombres et k le nombre que l'on recherche. On compare le nombre k au nombre qui se trouve au milieu du tableau T. Si c'est le même, on a trouvé, sinon on recommence sur la première moitié – ou la seconde – selon que k est plus petit – ou plus grand – que le nombre du milieu du tableau.

Algorithme 7: Recherche dichotomique

```
RECHERCHEDICHO (tableau T, entier k)
  Données : Un tableau T[1...N] d'entiers déjà ordonné et un entier k
  Résultat : Un indice i où se trouve l'élément k, ou bien -1, par
             convention si k n'est pas dans T
```

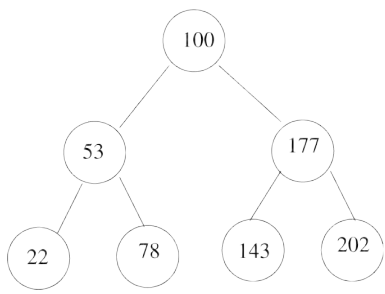
```
Résultat : Un indice i où se trouve l'élément k, ou bien -1, par
           convention si k n'est pas dans T
1  Indice entier i, l, r
2  l = 1
3  r = N
4  i = ⌊(l + r)/2⌋
5  while ((k ≠ T[i]) ∧ (l ≤ r))
6  |  if (k < T[i])
7  |  |  r = i − 1
8  |  else
9  |  |  l = i + 1
10 |  i = ⌊(l + r)/2⌋
11 if (k == T[i])
12 |  retourner [i]
13 else
14 |  retourner [−1]
```

Cet algorithme applique aussi la stratégie *Diviser pour régner*, bien qu'il ne soit pas récursif.

Soit $t(n)$ le nombre d'opérations effectuées par l'algorithme 7 sur un tableau de taille n : $t(n)$ satisfait l'équation de récurrence $t(n) = t(n/2) + 1$, comme $t(1) = 1$, on en déduit $t(n) = O(\log n)$. On remarquera que la complexité en temps est réduite de linéaire, $O(n)$, à logarithmique, $O(\log n)$, entre la recherche séquentielle et la recherche dichotomique.

Remarquons que la recherche dichotomique dans un tableau déjà ordonné revient à organiser les données du tableau sous la forme d'un arbre

22	53	78	100	143	177	202



Un arbre représentant T = *22, 53, 78, 100, 143, 177, 202 ; une recherche dichotomique dans* T *revient à suivre un chemin dans cet arbre.*

puis à descendre dans l'arbre jusqu'à soit trouver le nombre cherché, soit arriver à un échec. Par exemple, rechercher 177 dans le tableau

$$T = 22, 53, 78, 100, 143, 177, 202$$

par cet algorithme revient à suivre le chemin indiqué en gras dans la figure **A** ci-après et rechercher 180 dans T revient à suivre le chemin indiqué en gras dans la figure **B**.

Dans la figure A recherche dichotomique de 177 dans T = *22, 53, 78, 100, 143, 177, 202 ; dans la figure B recherche dichotomique de 180 dans* T.

Nous formalisons cette utilisation des arbres au paragraphe suivant.

Arbres binaires

Généralités et définitions

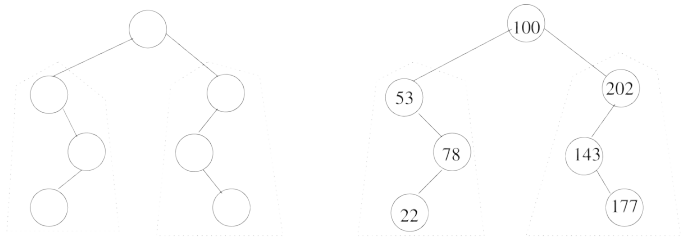
Nous montrons maintenant comment tri et recherche en table peuvent être traités en utilisant des structures d'arbres binaires.

L'ensemble *Arbre* des arbres binaires dont les nœuds sont étiquetés par les éléments d'un alphabet A a été défini inductivement au deuxième chapitre par :

– l'ensemble vide est un arbre binaire. Pour abréger l'écriture, l'arbre vide – noté `null` au deuxième chapitre – sera noté ∅,

– si l et r sont des arbres binaires, et a un élément de l'alphabet A, alors le triplet (a, l, r) est un arbre binaire dont la racine est étiquetée par a et dont l – resp. r – est le sous-arbre gauche – resp. droit.

On peut aussi considérer l'ensemble AB des *arbres binaires non étiquetés*, – c'est-à-dire sans le champ valeur défini au deuxième chapitre.



a. un arbre binaire non étiqueté b. un arbre binaire étiqueté

3

Un arbre non étiqueté et un arbre étiqueté.
Les sous-arbres gauches et droits sont entourés de traits en pointillé.

Dans la suite, tous nos arbres sont étiquetés et l'ensemble A des étiquettes est l'ensemble $\mathbb{N}$ des entiers naturels. On définit l'ensemble des nœuds d'un arbre binaire T, la racine de T, le fils gauche – resp. droit – d'un nœud, le sous-arbre gauche – resp. droit – d'un nœud :

– $noeuds(\emptyset) = \emptyset$,

– $noeuds(a, l, r) = \{(a, l, r)\} \cup noeuds(\ell) \cup noeuds(r)$,

– les fils droits et gauches, de même que les sous-arbres droits et gauches, ne sont pas définis pour l'arbre vide $\emptyset$,

– par convention, on définit $racine(\emptyset) = NIL$,

– si $T = (a, l, r)$, avec $l \neq \emptyset \neq r$, alors

1. $racine(T) = \{(a, l, r)\}$ – on identifie la racine à l'arbre et on la note $\{T\}$,
2. $filsgauche(T) = filsgauche(\{(a, l, r)\}) = racine(\ell)$,
3. $sous\text{-}arbre\text{-}gauche(T) = sous\text{-}arbre\text{-}gauche(\{(a, l, r)\}) = l$, et symétriquement,
4. $filsdroit(T) = filsdroit(\{(a, \ell, r)\}) = racine(r)$,
5. $sous\text{-}arbre\text{-}droit(T) = sous\text{-}arbre\text{-}droit\{(a, l, r)\}) = r$,

– si l est l'arbre vide $\emptyset$, c'est-à-dire $T = (a, \emptyset, r)$, alors

1. $racine(T) = \{(a, \emptyset, r)\}$,
2. $filsgauche(T) = filsgauche(\{(a, \emptyset, r)\}) = NIL$,
3. $sous\text{-}arbre\text{-}gauche(T) = sous\text{-}arbre\text{-}gauche(\{(a, \emptyset, r)\}) = \emptyset$,
4. $filsdroit(T) = filsdroit(\{(a, \emptyset, r)\}) = racine(r)$,
5. $sous\text{-}arbre\text{-}droit(T) = sous\text{-}arbre\text{-}droit(\{(a, \emptyset, r)\}) = r$,

– symétriquement si $r = \emptyset$;

– si $l = r = \emptyset$, alors $\{(a, \emptyset, \emptyset)\}$ n'a ni fils gauche ni fils droit – tous deux sont NIL.

Un nœud n d'un arbre T est identifié par le sous-arbre complet de T qui a pour racine n et tous les nœuds de ce sous-arbre – ils sont en dessous de n – sont appelés les *descendants* de n dans T.

Pour pouvoir désigner les nœuds d'un arbre, on les numérote par des suites de 0 et de 1 comme suit : la racine est numérotée ε – la suite vide –, son fils

gauche est numéroté 0 et son fils droit est numéroté 1 ; de manière générale, le fils gauche d'un nœud de numéro n est numéroté $n \cdot 0$ et son fils droit est numéroté $n \cdot 1$ – où $\cdot$ désigne la concaténation, par exemple $01 \cdot 11 = 0111$. Pour simplifier, on notera $n0$ au lieu de $n \cdot 0$, et $n1$ au lieu de $n \cdot 1$. Dans la suite, on désignera un nœud par son numéro. Cette numérotation nous permet de définir facilement les relations *père* et *fils* sur les nœuds d'un arbre : $pere(\varepsilon)$ n'est pas défini – la racine n'a pas de père –, et $pere(n0) = pere(n1) = n$. Les fils du nœud n sont les nœuds $n0$ – fils gauche – et $n1$ – fils droit –; par exemple le fils gauche de 01 sera 010, et le père de 01 sera 0.

On définit une application $label_T$ qui donne l'étiquette d'un nœud de T :

– si $T = \emptyset$, $label_T$ est l'application vide – $\emptyset$ n'a pas de nœud

– si $T = (a,l,r)$, alors

– $label_T(racine(T)) = a$, et

– si le nœud v est un nœud de l, $label_T(v) = label_l(v)$,

– si le nœud v est un nœud de r, $label_T(v) = label_l(v)$.

Dans la suite, par abus de notation, nous écrirons $T(v)$ pour $label_T(v)$.

Le seul nœud sans père est la racine de T. Un nœud sans fils – c'est-à-dire dont les sous-arbres droit et gauche sont l'arbre vide $\emptyset$ – est appelé une *feuille*. Par convention, on dira que l'étiquette d'un nœud de l'arbre vide est *NIL*.

La figure ci-après montre deux arbres binaires différents dont les nœuds sont étiquetés par le même ensemble d'étiquettes.

a. un arbre binaire de recherche

b. un arbre binaire

Deux arbres binaires qui diffèrent par l'ordre de leurs sous-arbres. L'arbre a *est un arbre binaire de recherche.*

Parcours dans les arbres

Un parcours d'arbre énumère toutes les étiquettes des nœuds d'un arbre selon un ordre choisi par avance. On distingue :

– *le parcours en largeur*: on énumère les étiquettes des nœuds niveau par niveau, on commence par la racine, puis on énumère les nœuds qui sont à la distance 1 de la racine, puis les nœuds qui sont à la distance 2 de la racine, etc.
– la *distance* entre deux nœuds est le nombre d'arêtes entre ces deux nœuds;

– les *parcours en profondeur*: ces parcours descendent « en profondeur » dans l'arbre tant que c'est possible. Pour traiter un nœud *n*, on traite d'abord tous ses descendants et on « remonte » ensuite pour traiter le père de *n* et son autre fils. Ces parcours en profondeur sont de trois types, et on les définit récursivement de manière fort simple.

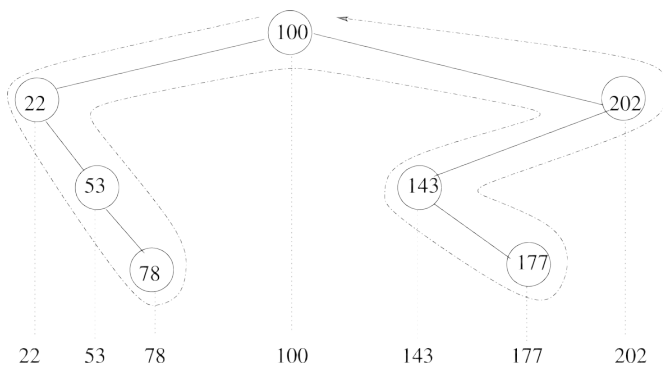
1. Le *parcours infixe* traite d'abord le sous-arbre gauche, puis le nœud courant puis son sous-arbre droit: l'étiquette du nœud courant est *entre* les listes d'étiquettes des sous-arbres gauches et droits.

2. Le *parcours préfixe* traite d'abord le nœud courant, puis son sous-arbre gauche, puis son sous-arbre droit: l'étiquette du nœud courant est *avant* les listes d'étiquettes des sous-arbres gauches et droits.

3. Le *parcours suffixe* – ou *postfixe* – traite d'abord le sous-arbre gauche, puis le sous-arbre droit, puis le nœud courant: l'étiquette du nœud courant est *après* les listes d'étiquettes des sous-arbres gauches et droits.

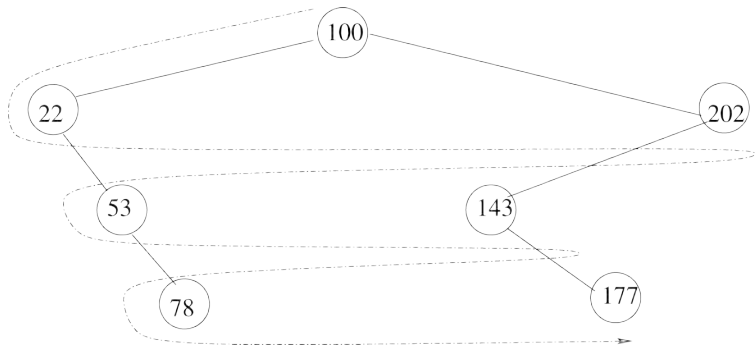
Par exemple, pour l'arbre a de la figure ci-avant,
– le parcours en largeur donne la liste: 100, 22, 202, 53, 143, 78, 177,
– le parcours infixe donne la liste: 22, 53, 78, 100, 143, 177, 202,
– le parcours préfixe donne la liste: 100, 22, 53, 78, 202, 143, 177,
– le parcours suffixe donne la liste: 78, 53, 22, 177, 143, 202, 100.

On remarquera que la liste obtenue par le parcours infixe est ordonnée. Cela est dû au fait que l'arbre est un arbre binaire de recherche.



Un arbre binaire et son parcours infixe.

L'algorithme 8 réalise un parcours en largeur d'arbre. T est un arbre, L la liste des étiquettes des nœuds de T quand on le parcourt, et F une file qui contient les nœuds en cours de traitement et dont les éléments sont traités dans l'ordre d'arrivée.



Un arbre binaire et son parcours en largeur.

Les opérations sur une file sont : *enfiler*(*F*,*n*), qui ajoute le nœud *n* à la fin de la file *F*, et premier(*F*) – resp. *defiler*(*F*) – qui prend le premier élément de la file – resp. la file privée de son premier élément. En programmation, on note *n* = defiler(*F*) pour désigner les *deux* opérations consistant à affecter le premier élément de *F* à *n* et à enlever ce premier élément de *F*.

Algorithme 8: Parcours en largeur d'un arbre T

PARCOURS-L (arbre T)
Données : Un arbre T dont les nœuds sont étiquetés par des entiers
Résultat : Le tableau L des étiquettes des nœuds de T quand on le parcourt en largeur

```
1  Var nœud n
2  Var file F
3  Var tableau L
4  i = 1
5  F = ∅
6  n = racine(T)
7  enfiler(F, n)
8  while (F ≠ ∅)
9  |   n = defiler(F)
10 |   if (filsgauche(n) ≠ ∅)
11 |   |   enfiler(F, filsgauche(n))
```

```
12    if (filsdroit(n) ≠ ∅)
13      enfiler(F, filsdroit(n))
14    L[i] = T(n)
15    i = i + 1
16  retourner L
```

Les algorithmes réalisant les parcours en profondeur s'écrivent de manière récursive, beaucoup plus simplement, car ils sont définis par récurrence sur la structure des arbres. Par exemple, l'algorithme réalisant le parcours infixe s'écrit :

Algorithme 9: Parcours infixe d'un arbre T

```
PARCOURS-IN (arbre T )
  Données : Un arbre T dont les nœuds sont étiquetés par des entiers
  Résultat : La liste L des étiquettes des nœuds de T quand on le
             parcourt en infixe
1 Var nœud n
2 Var liste d'entiers L, L1, L2
3 if (T == ∅)
4   L = ∅                    // Si T est vide, L est la liste vide
5 else
6   L1 =PARCOURS-IN(sous-arbre-gauche(T))
7   L2 =PARCOURS-IN(sous-arbre-droit(T))
8   L = L1 · racine(T) · L2  // On concatène L1, la racine de T et
    L2
9 retourner L
```

Voir le deuxième chapitre – paragraphe « Les types de données dynamiques » – pour une définition des listes. On remarquera l'élégance et la concision de l'algorithme récursif 9.

La complexité en temps de l'algorithme PARCOURS-IN sur un arbre T de taille n, c'est-à-dire ayant n nœuds, est linéaire ; en effet, soit $t(T)$ le nombre d'opérations nécessaires pour engendrer le parcours infixe de T. Pour calculer PARCOURS-IN(T), on effectue deux fois l'opération $\cdot$: $L \cdot L'$ met la liste L' à la suite de la liste L. On suppose que $\cdot$ est une opération primitive qui prend un temps constant c, alors on peut montrer que $t(T) = O(n)$ par récurrence sur le nombre de nœuds de T. On doit d'abord définir le nombre n de nœuds d'un arbre T par n = *nnoeuds*(T), où *nnoeuds* est la fonction définie par : *nnoeuds*(∅) = 0 et, si $T \neq \emptyset$, n = *nnoeuds*(T) = 1 + *nnoeuds*(*sous-arbre-gauche*(T)) + *nnoeuds*(*sous-arbre-droit*(T)). L'hypothèse de récurrence est : si T a n nœuds,

alors $t(T) \leq 2c \times n$; on constate en regardant la définition de PARCOURS-IN(T) que $t(T) = 2c + t(\textit{sous-arbre-gauche}(T)) + t(\textit{sous-arbre-droit}(T))$; en utilisant l'hypothèse de récurrence, $t(T) \leq 2c + 2c \times \textit{nnoeuds}(\textit{sous-arbre-gauche}(T)) + 2c \times \textit{nnoeuds}(\textit{sous-arbre-droitT})) = 2c \times n$, en remarquant que $\textit{nnoeuds}(T) = 1 + \textit{nnoeuds}(\text{sous-arbre-gauche}(T)) + \textit{nnoeuds}(\textit{sous-arbre-droit}(T))$.

Puisque le parcours infixe d'un arbre binaire de recherche prend un temps linéaire et fournit une liste ordonnée, on peut penser à utiliser les arbres binaires de recherche comme outil de tri. C'est une possibilité, que nous explorons maintenant, en commençant par donner un algorithme de construction d'arbre binaire de recherche.

Arbres binaires de recherche

Un arbre binaire de recherche est un arbre binaire dont les nœuds sont étiquetés par les éléments d'un ensemble *A totalement ordonné* de sorte que la propriété *P* suivante soit toujours vraie : si *a* est l'étiquette d'un nœud *n*, alors tous les nœuds du sous-arbre gauche de *n* sont étiquetés par des éléments *b* tels que $b \leq a$, et tous les nœuds du sous-arbre droit de *n* sont étiquetés par des éléments *b* tels que $b > a$.

L'arbre a de la figure p. 150 est un arbre binaire de recherche, mais non l'arbre b, car le nœud étiqueté 53 ne satisfait pas la propriété *P* : en effet, 22 apparaît dans le sous-arbre droit du nœud étiqueté 53, bien qu'il soit inférieur à 53.

Les arbres binaires de recherche ont la grande vertu de faciliter le tri d'un ensemble d'éléments : si les éléments sont stockés dans un arbre binaire de recherche, il suffit de parcourir cet arbre dans l'ordre infixe pour ranger les éléments par ordre croissant.

Construction des arbres binaires de recherche

Un arbre binaire de recherche est une *structure de données dynamique* qui doit pouvoir changer par l'ajout ou la suppression d'éléments.

Pour ajouter un nombre v dans un arbre binaire T, on descend depuis la racine de T jusqu'à arriver à un nœud de T où l'on pourra ajouter une nouvelle feuille z étiquetée v à sa bonne place. Si l'on cherche à insérer un nombre qui est déjà dans l'arbre, ce dernier n'est pas modifié.

Algorithme 10: Insertion d'un nombre à une feuille dans un arbre T

INSERER (arbre T, entier v)
Données : Un arbre binaire de recherche T dont les nœuds sont étiquetés par des entiers et un entier v à insérer dans T

```
   Résultat : L'arbre T avec une nouvelle feuille étiquetée v
 1 Var nœud x
 2 Var arbre T'
 3 T' = T
 4 while (T' ≠ ∅)
 5 |   x = racine(T')
 6 |   if ((v < T(x)) ∧ (sous-arbre-gauche(T') ≠ ∅))
 7 |   |   T' = sous-arbre-gauche(T')
 8 |   if ((v > T(x)) ∧ (sous-arbre-droit(T') ≠ ∅))
 9 |   |   T' = sous-arbre-droit(T')
10 |   if (v == T(x))
11 |   |   EXIT
12 if (T == ∅)
13 |   T = (v, ∅, ∅)
14 else
15 |   if v(< T(x))
16 |   |   y = (T(y), (v, ∅, ∅), sous-arbre-droit(T'))
   |   /* le sous-arbre gauche vide de y est remplacé par (v, ∅, ∅)
   |      */
17 |   if v > T(x)
18 |   |   y = (T(y), sous-arbre-droit(T'), (v, ∅, ∅))
   |   /* le sous-arbre droit vide de y est remplacé par (v, ∅, ∅)
   |      */
```

Pour ajouter 155 dans l'arbre a, on descend le long de la branche dont les nœuds sont étiquetés 100, 202, 143, on ajoute un nœud z de numéro 100 et d'étiquette 155. On voit que la complexité du programme INSERER est linéaire en la hauteur de l'arbre binaire de recherche. On conçoit donc qu'un arbre binaire le moins haut possible optimisera le temps.

On peut montrer que pour générer à partir de ∅ un arbre binaire de recherche contenant tous les nombres d'une liste de n nombres, il faut *en moyenne* un temps $O(n \log n)$, et que la construction de cet arbre effectue des comparaisons similaires à celui du tri rapide. En reprenant maintenant l'idée de trier une liste en l'implantant comme un arbre binaire de recherche, que l'on parcourt ensuite de manière infixe, on voit que le temps de tri est le temps de parcours, $O(n)$, auquel il faut ajouter le temps de création de l'arbre binaire qui représente la liste donnée, soit $O(n \log n)$, et donc en tout $O(n \log n) + O(n) = O(n \log n)$; ce temps est comparable à celui du tri rapide. Si l'on s'intéresse à trier et aussi à faire d'autres opérations sur une liste de nombres, il est judicieux d'implanter cette liste sous la forme d'un arbre binaire de recherche.

Opérations sur les arbres binaires de recherche

Les opérations usuelles sur les arbres binaires de recherche sont : rechercher un nombre, supprimer un nombre, rechercher le nombre maximum, etc. Toutes ces opérations se feront en temps $O(h)$ si h est la hauteur de l'arbre binaire de recherche. Par exemple, l'algorithme de recherche d'un nombre k dans un arbre binaire de recherche s'écrit :

Algorithme 11: Recherche dans un arbre binaire de recherche T

```
RECHERCHER (arbre T, entier k)
   Données : Un arbre binaire de recherche T dont les nœuds sont
             étiquetés par des entiers et une valeur k à rechercher dans T
   Résultat : Un nœud de T étiqueté k s'il en existe
 1 Var nœud n
 2 n = racine(T)
 3 while ((T(n) ≠ NIL) ∧ (T(n) ≠ k))
 4 |   if (k < T(n))
 5 |   |   n = filsgauche(n)
 6 |   else
 7 |   |   n = filsdroit(n)
 8 if (T(n) ≠ NIL)
 9 |   afficher T(n) = k
10 else
11 |   afficher k n'est pas dans T
```

L'algorithme de recherche du maximum d'un arbre binaire de recherche est encore plus simple, puisqu'il suffit de descendre toujours à droite :

Algorithme 12: Recherche du maximum d'un arbre binaire de recherche T

```
MAX (arbre T)
   Données : Un arbre binaire de recherche non vide T dont les nœuds sont
             étiquetés par des entiers
   Résultat : Le maximum des étiquettes des nœuds de T
1 Var nœud n
2 n = racine(T)
3 while (T(filsdroit(n)) ≠ NIL)
4 |   n = filsdroit(n)
5 afficher le maximum de T est T(n) = k
6 retourner [k]
```

La plupart des opérations se font en temps $O(h)$ sur un arbre binaire de recherche de hauteur h. Comme, en moyenne, la hauteur h d'un arbre binaire de recherche ayant n nœuds est en $O(\log n)$, on peut donc implanter une liste de longueur n par un arbre binaire de recherche en temps $O(n \log n)$: on fait n insertions dont chacune prend un temps $O(n \log n)$ à partir de l'arbre vide. On peut en conclure que les arbres binaires de recherche donnent un bon outil de travail sur les listes.

Quelques algorithmes classiques sur les graphes

Les arbres que nous avons étudiés au paragraphe précédent, « Arbres binaires » sont un cas particulier des graphes auxquels nous allons nous intéresser maintenant.

Un *graphe non orienté* G est un triplet (S, A, δ), où :

– S est un ensemble de *sommets*,

– A est un ensemble d'*arêtes*, disjoint de S,

– $\delta : A \longrightarrow S \times S$ associe à chaque arête deux sommets non nécessairement distincts : les extrémités de l'arête.

Voir les figures ci-après pour des exemples de graphes.

Un graphe non orienté est *connexe* si, pour toute paire (s, s') de sommets distincts, il existe un chemin joignant s et s'. Dans un graphe non orienté connexe, la *distance* $d(s,s')$ entre deux sommets s et s' est égale à la longueur du plus court chemin joignant ces deux sommets – en particulier $d(s, s) = 0$.

Tous nos graphes sont non orientés et connexes.

La différence essentielle entre graphes et arbres est qu'un arbre n'a aucun cycle : si on part d'un nœud, on ne pourra *jamais* y revenir en suivant les arêtes de l'arbre et « sans faire marche arrière », cela parce qu'entre deux nœuds d'un arbre il y a toujours *un et un seul* chemin ; en revanche, entre deux nœuds, que l'on appelle sommets, d'un graphe, il peut y avoir plusieurs chemins. Tout arbre est connexe, alors qu'un graphe peut ne pas l'être, et, dans ce cas, entre deux sommets du graphe, il peut n'y avoir aucun chemin. Toutefois, comme nous nous restreindrons aux graphes connexes, les problèmes de parcours se posent pour les graphes comme pour les arbres. Il faudra cependant un peu plus de travail pour les résoudre dans le cas des graphes, du fait de la pluralité de chemins entre deux sommets.

Les graphes sont une très riche source de problèmes algorithmiques. Les premiers algorithmes sur les graphes furent motivés par des problèmes pratiques importants :

– la conception du réseau d'électrification de la Moravie conduisit à concevoir le premier algorithme glouton de construction d'un arbre couvrant minimal ;

– le problème de savoir comment un voyageur de commerce doit organiser un circuit pour visiter un certain nombre de villes avec un coût le plus petit possible est la source de nombreux problèmes.

Cela a conduit à généraliser aux graphes les problèmes de parcours que nous avons vus dans les arbres ; de plus, des problèmes d'optimisation – recherche de plus court chemin, de coût minimum, etc. – se posent pour les graphes. Nous illustrerons la technique dite d'algorithme glouton qui est assez fréquente pour les graphes.

Les arêtes peuvent être munies d'un *poids*, qui est donné par une application $poids : A \longrightarrow \mathbb{N}$; par exemple, si le graphe représente un réseau ferré, les sommets sont les villes desservies, une arête relie deux villes entre lesquelles un train peut circuler et son poids est la distance entre ces deux villes. Pour simplifier les notations, on suppose que les poids sont des entiers naturels.

Parcours dans les graphes

Comme pour les arbres, on souhaite définir des parcours dans les graphes, de telle sorte que chaque sommet soit traité une fois et une seule. Le problème est plus complexe car, dans un graphe, il peut y avoir plusieurs chemins pour aller d'un sommet à un autre et il n'y a pas un ordre naturel sur les successeurs d'un sommet. Il faut donc annoter les sommets déjà visités pour ne pas les traiter une seconde fois : pour ce faire, S est partitionné, à tout instant de l'algorithme, en trois ensembles disjoints *blanc*, *gris*, *noir*. L'idée est que

– *blanc* est l'ensemble des sommets non traités,

– *gris* est l'ensemble des sommets visités et en cours de traitement : dès qu'un sommet est coloré en gris, on le traite, et on le met dans la file des sommets dont les sommets adjacents doivent être visités,

– *noir* est l'ensemble des sommets traités et dont tous les sommets adjacents ont été visités, marqués en gris et traités.

Au départ, tous les sommets sont blancs, sauf celui d'où l'on commence le parcours, qui est gris ; à la fin de l'algorithme, tous les sommets sont noirs.

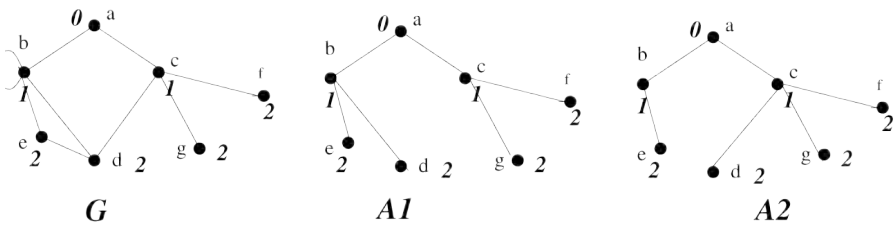
L'algorithme ci-après généralise l'algorithme de parcours en largeur PARCOURS-L dans les arbres au cas des graphes. On souhaite calculer un *arbre couvrant* de G, c'est-à-dire un arbre qui contienne tous les sommets de G et dont les arêtes sont aussi des arêtes de G, voir la figure ci-après ; l'arbre couvrant engendré ici préserve les distances au sommet s, c'est-à-dire le nombre d'arêtes entre un sommet n et s dans l'arbre est le nombre minimal d'arêtes pour aller de s à n dans G. Le traitement d'un sommet n consiste à trouver en quel nœud de l'arbre couvrant on doit le placer : pour cela, il suffit de préciser quel est son père dans l'arbre, on calculera donc un tableau $pred[n]$ où $pred[n]$

est le sommet de G qui sera le père de n dans l'arbre couvrant – voir plus loin le paragraphe « Question de programmation », pour la manière de programmer le calcul du tableau *pred*. Remarquons qu'un arbre couvrant n'est pas nécessairement binaire – voir figure ci-après où l'arbre A_2 n'est pas binaire.

L'ensemble des sommets en cours de traitement, ici les sommets gris, est géré par une file, comme dans PARCOURS-L. Pour chaque sommet s de G, on dispose de la liste $ADJ(s)$ des sommets reliés à s par une arête ; on utilise aussi dans l'algorithme un tableau C donnant la couleur courante associée aux sommets, et le tableau *pred* donnant le prédécesseur de chaque sommet dans le parcours.

Algorithme 13: Parcours en largeur d'un graphe G préservant les distances à un sommet s

```
PARCOURS-LG (graphe G, sommet s)
  Données : Un graphe G et un sommet s de G
  Résultat : Un arbre couvrant de G de racine s tel que la distance à s de
             chaque sommet de G soit préservée dans l'arbre
1  Var sommet v, u
2  Var file F
3  Var tableau de couleurs C
4  Var tableau de sommets pred
5  for (v ∈ S \ {s})
6      C[v] = blanc
7  C[s] = gris
8  pred[s] = NIL          // On donne la racine s à l'arbre couvrant
9  F = ∅
10 enfiler(F, s)
11 while (F ≠ ∅)
12     u = defiler(F)
13     for ( v ∈ ADJ(u))
14         if (C[v] == blanc)
               // On vérifie que v n'a pas déjà été visité
15             C[v] = gris
16             pred[v] = u          // Actualise l'arbre couvrant en y
               ajoutant v et son père
17             enfiler(F, v)  // Actualise la file F en y ajoutant v
18     C[u] = noir          // u et tous ses voisins ont été traités
```

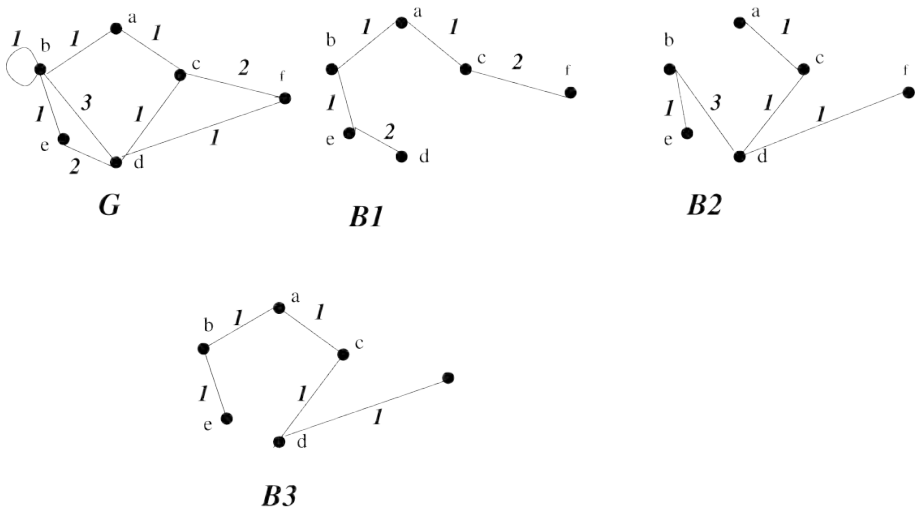


Un graphe G *et deux arbres couvrants* A_1 *et* A_2 *de* G *préservant les distances au sommet étiqueté* a*: à côté de chaque sommet, on voit une lettre qui est son étiquette et un nombre qui est sa distance à* s*. Les deux arbres* A_1 *et* A_2 *sont obtenus par l'algorithme 13.*

L'algorithme 13 PARCOURS-LG est de type *glouton*, c'est-à-dire à chaque étape il fait le meilleur choix instantané possible, même si l'on n'est pas certain que ce choix soit le meilleur dans l'absolu ; il parcourt *G* en calculant les distances de chaque nœud *n* au nœud *s* choisi comme point de départ ; le parcours de *G* engendre en même temps un arbre couvrant préservant les distances au sommet *s* ; voir la figure ci-avant pour un exemple de graphe et deux arbres couvrants préservant les distances au sommet *a* obtenus par PARCOURS-LG : cette figure montre que cet algorithme est *non déterministe* : il peut faire plusieurs choix – ligne 13 – et produire plusieurs résultats, par exemple les arbres couvrants A_1 et A_2.

Le problème des arbres couvrants est utile lorsqu'on veut relier par un ensemble de câbles un ensemble de points, par exemple des téléphones ou des postes de télévision. Ce problème admet des variantes : si les arêtes de *G* sont étiquetées par des *poids* ou *coûts*, on cherchera souvent un arbre couvrant de coût minimal. Dans l'exemple de couverture d'un réseau de points par des câbles, certaines sections de câble peuvent coûter plus cher, car le câble doit être enterré plus profondément, ou doit être mieux protégé, ou est plus long, etc. On souhaitera alors trouver un arbre couvrant de *coût*, ou *poids*, *minimum*.

Cette figure montre un exemple de graphe avec poids sur les arêtes, deux arbres couvrants de poids non minimaux et un arbre couvrant de poids minimum obtenu par l'algorithme de Prim – algorithme 14 – qui calcule efficacement un arbre couvrant de poids minimum.



Un graphe G *: à côté de chaque sommet on voit une lettre qui est son étiquette et à côté de chaque arête un nombre qui est son coût. Trois arbres couvrants de* G *:* B_1 *– de poids non minimum 7 – et* B_2 *– de poids non minimum 6 –, et un arbre couvrant* B_3 *de poids minimum 5 – obtenu par l'algorithme 14.*

L'algorithme de Prim suit aussi une stratégie gloutonne pour calculer un arbre couvrant de poids minimum. L'arbre couvrant commence d'un sommet s choisi comme racine, auquel on ajoute à chaque étape une arête de poids minimal reliant un des sommets de l'arbre à un sommet non encore connecté à l'arbre. On utilise :

– le tableau *poids* où *poids*(x, y) donne le poids de l'arête reliant le sommet x au sommet y, si elle existe ;

– pour chaque sommet x la liste *ADJ*(x) des sommets *différents* de x et reliés à x par une arête ;

– une liste P de sommets non encore connectés à l'arbre, initialisée à s ;

– un tableau *pred* de paires de sommets (v, n) donnant le prédécesseur – ou père – de chaque sommet v de G dans l'arbre couvrant, initialisée à (s, *NIL*) ;

– un tableau *cle* qui donnera pour chaque sommet x de G, le poids minimum d'une arête reliant x à l'arbre couvrant en cours de construction.

Algorithme 14: Algorithme de Prim : calcul d'un arbre couvrant de poids minimum d'un graphe G

```
PRIM (graphe G sommet s)
  Données : Un graphe G et un sommet s de G
  Résultat : Un arbre couvrant de G de racine s obtenu avec un parcours
             en largeur de G
1  Var liste d'entiers L
2  Var sommet v, u
3  Var ensemble de sommets P
4  Var tableau d'entiers cle , poids
5  Var tableau de sommets pred
6  for (v ∈ S \ {s})
7  |  cle[v] = ∞                                   // Initialisations
8  cle[s] = 0
9  P = S
10 pred[s] = NIL
11 while (P ≠ ∅)
12 |  u =un sommet de cle minimale dans P          // Appel à la file de
   |  priorité P
13 |  P = P \ {u}
14 |  for (v ∈ ADJ(u))
15 |  |  if ((v ∈ P) ∧ (poids(u, v) < cle[v]))
16 |  |  |  cle[v] = poids(u, v)
17 |  |  |  pred[v] = u
```

On remarquera que l'arbre couvrant ainsi construit n'est pas unique, car plusieurs choix sont possibles à la ligne 12.

Comme pour les arbres, on peut parcourir les graphes « en profondeur », c'est-à-dire visiter tous les descendants du sommet courant s avant de retourner en arrière pour explorer les sommets qui sont au même niveau que s. Donnons un algorithme PARCOURS-GP de parcours en profondeur de graphe. Comme pour le parcours en largeur, PARCOURS-GP utilise le tableau de couleurs blanc – non encore visité –, gris – en cours de traitement – et noir – traité – pour les sommets, et un tableau *pred* qui donne le prédécesseur, ou père, d'un sommet dans le parcours. On utilise en outre une variable *temps* qui permet d'estampiller chaque sommet x par deux dates, la date $d(x)$ de la première visite – où il est coloré en gris – et la date $f(x)$ de la dernière visite – où il a été traité ainsi que tous ses voisins et où il est coloré en noir. P est la liste des sommets non encore visités – colorés en blanc.

Algorithme 15: Parcours en profondeur de G à partir du sommet s

PARCOURS-GP (graphe G, sommet s)
Données : Un graphe G et un sommet s de G
Résultat : Parcours en profondeur de G à partir du sommet s

```
1  Var entier temps
2  Var tableau d'entiers d, f
3  Var tableau de couleurs C
4  Var tableau de  sommets pred
5  Var sommet v
6  Var tableau de  sommets pred
7  for (v ∈ S)
8  |  C[v] = blanc
9  |  d[v] = f[v] = ∞
10 |  pred[v] = NIL
11 temps = 0
12 VISITEP(G, s, temps, f, C, pred)
```

Algorithme 16: Parcours en profondeur de G à partir d'un sommet n

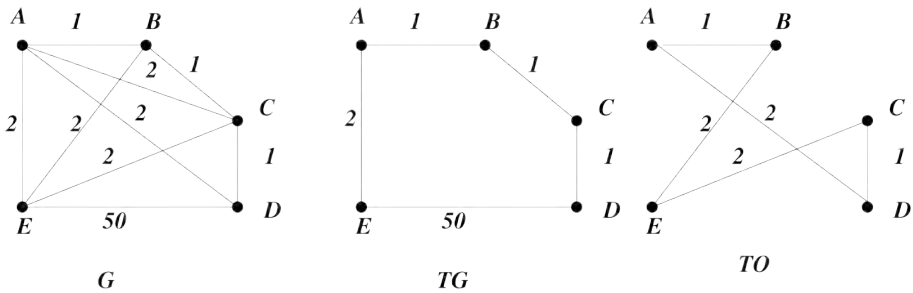
VISITEP (graphe G, sommet u, entier $temps$, tableau entier f, tableau couleur C, tableau sommets $pred$)
Données : Un graphe G avec un marquage indiquant les sommets déjà visités et leurs prédécesseurs, un sommet n de G
Résultat : Parcours en profondeur de G à partir du sommet n

```
1  Var tableau d'entiers d
2  Var sommet v
3  temps = temps + 1
4  C[u] = gris
5  d[u] = temps                          // Début traitement du sommet u
6  for v ∈ ADJ(u)
7  |  if (C[v] == blanc)
8  |  |  C[v] = gris
9  |  |  pred[v] = u
10 |  |  VISITEP(G, v, temps, f, C, pred)
11 |  C[u] = noir
12 |  temps = temps + 1                  // Fin traitement du sommet u
13 |  f[u] = temps
```

Cet algorithme de parcours en profondeur des graphes est un algorithme récursif, comme PARCOURS-IN pour les arbres.

Remarque : un algorithme glouton fait à chaque étape le choix le meilleur possible à cet instant ; si ce choix est aussi le meilleur possible dans l'absolu,

l'algorithme glouton sera alors optimal ; toutefois, il y a des problèmes pour lesquels le choix le meilleur à chaque étape ne donne pas le meilleur choix global possible. Le problème du voyageur de commerce consiste, étant donné un graphe *G* dont les arêtes sont munies de poids et un sommet *s* de ce graphe, à trouver un cycle de poids total minimum, partant de *s* et passant une fois et une seule par chaque sommet. Pour ce problème, l'algorithme glouton choisira à chaque étape la ville « la plus proche » non encore visitée, et il se peut que ce choix soit fort mauvais globalement. Par exemple, dans la figure ci-après, en partant du sommet *A*, la stratégie gloutonne qui consiste à choisir à chaque instant l'arête de poids minimum menant à un sommet non encore visité donne le parcours *TG* – parcours glouton *ABCDEA* – de poids 55, alors que le parcours optimal *TO* – *ABECDA* – a un poids de 8.



Un graphe G *et deux parcours de voyageur de commerce : parcours glouton* TG *et parcours optimal* TO.

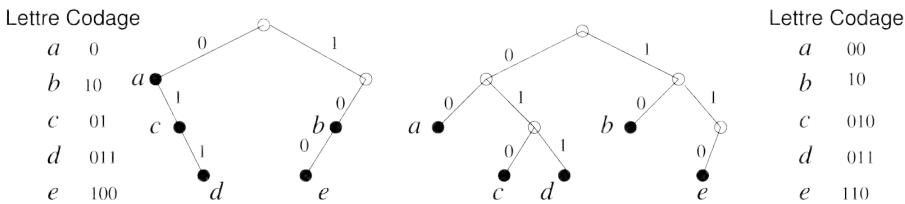
Algorithme de codage de Huffman

Comme nous l'avons vu au premier chapitre, coder un message en binaire consiste à représenter chaque lettre de ce message par une suite de 0 et de 1, par exemple en représentant chaque lettre par son code ASCII. Un code *C* est donc une bijection entre un ensemble de symboles, un alphabet, $A = \{a_1, \ldots, a_K\}$ et un ensemble de mots binaires $m_1, m_2, \ldots, m_K$.

On dit qu'un code est *préfixe* si aucune des suites m_i n'est un préfixe, c'est-à-dire le début, d'une autre. Avec un code préfixe, on peut coder un message sans utiliser de séparateurs entre les codes des différents symboles, et le message codé reste non ambigu. Par exemple, si le code 00 est utilisé pour coder un symbole *a*, alors aucun autre symbole n'est codé par une suite qui commence par 00, et tous les messages codés par une suite qui commence par 00 commencent par la lettre *a*.

On peut représenter tout code par un arbre binaire : certains nœuds de l'arbre sont étiquetés par des lettres de *A* et le codage de la lettre qui étiquette

n est le numéro du nœud n, avec la sémantique déjà donnée pour numéroter les nœuds : fils gauche codé par 0 et fils droit par 1. Un code est préfixe si seules les feuilles de l'arbre binaire sont étiquetées par les lettres de l'alphabet A. La longueur du codage d'une lettre dans un code préfixe est la distance entre la racine de l'arbre et la feuille correspondante. Voici deux exemples de codes. Le premier n'est pas préfixe, le second est préfixe.



A: Code non préfixe **B:** Code préfixe

L'intérêt d'un code de longueur variable est d'adapter la longueur du codage de chaque lettre à sa fréquence d'apparition dans le texte à coder : plus une lettre est fréquente, plus son code doit être court. On notera l_i la longueur du codage m_i et p_i la proportion, ou fréquence ou probabilité, de la lettre S_i dans le texte à coder.

Le problème à résoudre est de trouver un code binaire préfixe qui minimise la longueur moyenne du code,

$$L(C) = \sum_{i=1}^{K} p_i l_i.$$

Cela revient à construire un arbre binaire pour lequel la moyenne pondérée des feuilles soit minimale. Pour réaliser cet arbre, on construit une solution en partant des feuilles les plus profondes, puis on construit l'arbre progressivement en les combinant deux à deux.

Algorithme 17: Algorithme de Huffman (1951)

ALGORITHME_HUFFMAN
Données : Un ensemble $\mathcal{S}$ de K symboles avec leurs pondérations p
Résultat : Un arbre optimal de codage (nœud racine)
Indice i
Indice x,y,z
File de Priorité F
`/* F est une file de priorité dans laquelle on insère des`

```
   couples (nœud, priorité) et on extrait les couples selon
   leur priorité                                              */
/* pour cet algorithme, l'extraction se fera par poids de
   nœud croissant                                             */
for (s ∈ 𝒮)                          // Initialisation de la forêt
    z=nouveau_nœud() ; z.symbole=s ; z.poids=p(s)
    Insérer (F,z)
for i = 1 to K − 1                          // Itération principale
    // Il existe un arbre de codage optimal dont les arbres
       contenus dans F sont des sous-arbres
    // F contient K − i + 1 nœuds
    x=Extraire (F) ; y=Extraire (F) ;
    z=nouveau_nœud() ; z.gauche=x ; z.droit=y
    z.poids=x.poids+y.poids
    Insérer (F,z)
Renvoie Extraire (F)
```

L'algorithme de Huffman produit un code préfixe de longueur moyenne optimale. C'est un autre exemple d'algorithme glouton. L'algorithme nécessite donc de l'ordre de K opérations d'insertion et d'extraction de F et, si la file de priorité est implantée dans une structure de données adéquate, par exemple un tas, le coût d'insertion et d'extraction est majoré par $\log K$. Ce qui donne à cet algorithme une complexité de l'ordre de $O(K \log K)$.

Exercices corrigés et commentés

Exercice 1

Justifiez le temps de 1 000 jours pour le tri sélection d'un tableau de 100 millions d'éléments.

Correction. En utilisant un ordinateur actuel cadencé à 3GHz et en supposant qu'une comparaison s'effectue en dix cycles machines – en général c'est beaucoup plus – on a besoin de

$$\frac{(100000000)^2}{3000000000/10} = \frac{10^6}{3\times10^8}\ \text{s} = \frac{10^8}{3}\ \text{s}$$

soit $10^8/180$ mn, ou

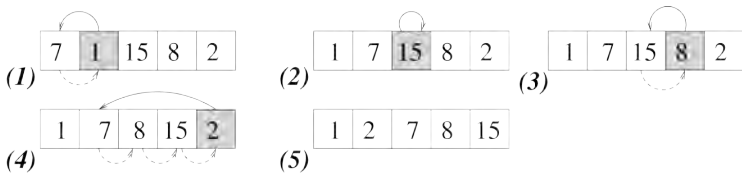
$$\frac{10^8}{540} \sim 10^6/5 \sim 2 \times 10^5 \text{ heures}$$

soit environ 10^3 = 385 jours.

Exercice 2

Un autre algorithme naïf de tri, celui que l'on fait lorsque par exemple on a en main des cartes à jouer que l'on veut ranger dans un certain ordre, est le tri par insertion.

Le tri par insertion réordonne les nombres du tableau, en commençant par le premier, et de telle sorte que, lorsque l'on traite le j-ème nombre, les $j-1$ premiers nombres sont ordonnés ; pour traiter le j-ème nombre, il suffit alors de l'insérer à sa place parmi les $j-1$ premiers nombres.



Comment INSERTION trie la liste 7, 1, 15, 8, 2.

Soit T une liste de n nombres. Donner un algorithme qui réalise le tri par insertion et calculer sa complexité.

Correction

Algorithme 18: Le tri par insertion

```
INSERTION (tableau T, entier n)
  Données : Un tableau T de n entiers
  Résultat : Le tableau T contient les mêmes éléments mais rangés par
             ordre croissant
1 Var entier k
2 Indice entier i, j
3 for j = 1 to n
4 |   k = T[j]
  |   /* Les j − 1 premiers éléments de T sont ordonnés et on va
  |      insérer T[j] à sa place dans le tableau T[1..(j − 1)]    */
5 |   i = j − 1
6 |   while ((i > 0) ∧ (T[i] > k))
7 |   |   T[i + 1] = T[i]                  // On décale le i-ème élément
8 |   |   i = i − 1
9 |   T[i + 1] = k
  |   /* Les j premiers éléments de T sont ordonnés              */
```

Le tri par insertion fonctionne en *espace constant* : on n'utilise qu'un nombre constant de nombres hors du tableau à trier.

Soit $t(n)$ la *complexité en temps*, comptée en nombre d'opérations :

– le meilleur cas est celui où le tableau est déjà trié, et la complexité est *linéaire* car $t(n)$ est une fonction linéaire de n ;

– le pire cas est celui où le tableau est en ordre décroissant, alors pour chaque j entre 2 et n, il faut faire dans la boucle **while** $C \times j$ opérations, et la complexité en temps est de l'ordre de $\sum_2^n C \times j = C \times n(n-1)/2$, c'est une fonction quadratique de n ;

– la complexité en moyenne, en supposant qu'en moyenne dans la boucle **while** la moitié des nombres seront inférieurs à $T[j]$, sera aussi quadratique car il faudra faire $C \times j/2$ opérations pour chaque itération de la boucle **while**.

Exercice 3. On suppose que la taille N du tableau T est une puissance de 2, $N = 2^n$.

1. Modifier l'algorithme 7 pour trouver le premier indice i où se trouve l'élément k.

2. Écrire un algorithme récursif pour trouver le premier indice i où se trouve l'élément k.

Correction

Algorithme 19: Recherche dichotomique de la première occurrence de k dans un tableau à 2^n éléments

```
RECHERCHE-DIC-PUIS2 (tableau T, entier k)
  Données : Un tableau T[1...2^n] d'entiers déjà ordonné et un entier k
  Résultat : Le premier indice i où se trouve l'élément k, ou bien -1, par
             convention si k n'est pas dans T
1  Indice entier i, l, r
2  l = 1
3  r = 2^n
4  i = 2^(n-1)
5  while ((k ≠ T[i]) ∧ (l ≤ r))
6  |  if (k < T[i])
7  |  |  r = i
8  |  else
9  |  |  l = i + 1
10 |  n = n − 1
11 |  i = l + 2^(n-1)
12 if (k == T[i])
13 |  repeat
14 |  |  i = i − 1
15 |  until (k == T[i])
16 |  retourner [i]
```

```
17 else
18 |_ retourner [−1]
```

On remarquera l'utilisation d'une boucle **repeat**.

Algorithme 20: Recherche dichotomique récursive de la première occurrence de k dans un tableau à 2^n éléments

```
RECH-DIC-REC-PUIS2 (tableau T, entier k)
  Données : Un tableau T[l + 1...l + 2^n] d'entiers déjà ordonné et un entier
            k
  Résultat : Le premier indice i où se trouve l'élément k, ou bien -1, par
             convention si k n'est pas dans T
1 Indice entier i
2 i = l + 2^(n−1)
3 if (k ≤ T[i])
4 |  RECH-DIC-REC-PUIS2(T[1...2^(n−1)], k)
5 else
6 |_ RECH-DIC-REC-PUIS2(T[2^(n−1) + 1...2^n], k)
7 if ((i == l) ∧ (k == T[i]))
8 |  retourner i
9 else
10|_ retourner −1
```

On remarquera que l'algorithme récursif 20 est beaucoup plus concis. Toutefois, cet algorithme récursif naïf prendra toujours un temps log n, alors que l'algorithme itératif peut être beaucoup plus rapide – si, par exemple, k se trouve au milieu du tableau, l'algorithme 19 terminera en temps constant –; bien sûr, la complexité dans le pire cas des deux algorithmes est $O(\log n)$.

Exercice 4

Donner les numéros des nœuds de l'arbre a de la figure p. 150 et leurs étiquettes.

Correction. La racine est numérotée ε et son étiquette est $T(\varepsilon) = 100$, les autres nœuds : $T(0) = 22$, $T(1) = 202$, $T(01) = 53$, $T(10) = 143$, $T(011) = 78$, $T(101) = 177$ et il y a deux feuilles étiquetées 78, 177.

Exercice 5

1. La preuve donnée au paragraphe « Arbres binaires » pour montrer que la complexité en temps du parcours infixe est linéaire est fausse : pourquoi ? 2. Corriger cette preuve : il faut prendre en compte le cas de base de la récurrence, à savoir le cas où $n = 0$.

Correction. 1. On a compté pour 0 le temps de parcours d'un arbre vide – base de la récurrence –, or il faut au moins tester si l'arbre est vide. Cet exercice

montre la difficulté d'évaluer la complexité d'un algorithme, même simple, *et* le soin qu'il faut apporter aux preuves par récurrence.

2. Soit c' le temps d'exécution de la ligne 4 de PARCOURS-IN, alors, pour le cas de base de la récurrence, $t(0) = c'$, et ensuite, en posant l'hypothèse de récurrence : $t(T) \leq (c' + 2c)k$ si k est le nombre de nœuds de T, on a $t(T) \leq c'+2c+2c\ nnoeud(sous\text{-}arbre\text{-}gauche(T))+2c\ nnoeud(sous\text{-}arbre\text{-}droit(T)) = (c' + 2c)n$, ce qui donne encore un temps linéaire.

Exercice 6

1. Quel type de liste donne la complexité la pire pour INSERER ? 2. Quel est l'ordre de grandeur de cette complexité la pire ?

Correction. Le pire cas se produit lorsque la liste est triée, alors l'arbre binaire de recherche correspondant est un arbre filiforme, qui se réduit à sa branche la plus à droite, et pour ajouter le $(i + 1)$-ème élément de la liste, il faudra parcourir les i nœuds déjà créés, d'où le temps de calcul pour une liste de longueur n : $t(n) = C \times \sum_{i=1}^{n-1} i = C \times n\,(n - 1)/2 = O(n^2)$, $t(n)$ est donc quadratique. On remarquera que, pour le tri rapide aussi, la pire complexité est obtenue pour une liste triée et elle est quadratique aussi.

Exercice 7

1. Écrire un algorithme MIN qui trouve le minimum d'un arbre binaire de recherche. 2. Écrire un algorithme qui trouve le successeur d'un nœud d'un arbre binaire de recherche dans l'ordre donné par le parcours infixe de l'arbre.

Correction. 1. Il suffit de remplacer « droit » par « gauche » dans l'algorithme MAX. 2. Si le sous-arbre droit de n est non vide, le successeur de n est le minimum de ce sous-arbre ; sinon,

– soit n est le plus grand nombre de l'arbre T,

– soit n n'est pas le plus grand nombre de T et son successeur est l'ancêtre x de n le plus proche de n tel que le fils gauche de x soit aussi un ancêtre de n – la relation ancêtre est réflexive, c'est-à-dire que n est un ancêtre de lui-même. Par exemple, dans l'arbre a de la figure p. 150, le successeur du nœud étiqueté 78 est le nœud étiqueté 100. L'algorithme s'écrit ainsi :

Algorithme 21: Recherche du successeur d'un nœud d'un arbre binaire de recherche T

SUCCESSEUR (arbre T nœud n)
Données : Un arbre binaire de recherche T dont les nœuds sont étiquetés par des entiers et un nœud n
Résultat : Le nœud de T dont l'étiquette est le successeur de l'étiquette de n

```
1  Var nœud n, x, y
2  if (T(filsdroit(n)) ≠ NIL)
3  |  MIN(sous-arbre-droit(n))
4  else
5  |  x = pere(n)
6  |  y = n
7  while ((T(x) ≠ NIL) ∧ (y == filsdroit(x)))
8  |  y = x
9  |  x = pere(x)
10 afficher le successeur de n est x
11 retourner [x]
```

Exercice 8

1. Calculer la borne inférieure m des poids des chemins parcourant un graphe G en partant d'un sommet et en revenant à ce sommet s après avoir parcouru tous les sommets de G. 2. Donner une heuristique qui calcule, en temps polynomial en le nombre de sommets de G, un chemin de poids au plus 2 m.

Correction. Voir Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein, *Introduction to Algorithms*, paragraphe 2.1.1.

Exercice 9

Preuve de l'algorithme de Huffman – algorithme 17. La preuve de cet algorithme repose sur 2 lemmes préliminaires dont la preuve est basée sur un argument d'échange.

Lemme 1: Probabilités les plus faibles. Soit S un alphabet de K lettres. Soit s et s' deux symboles de plus petite probabilité. Alors il existe un code préfixe optimal pour S tel que les codages de s et de s' ne diffèrent que par le dernier bit.

Indication : prendre un arbre optimal et le transformer de manière à vérifier la propriété.

Lemme 2: Propagation de l'optimalité. Soit T un arbre de codage optimal complet de S. Alors la fusion z de 2 feuilles sœurs x et y affectée de la somme des poids des feuilles *z.poids = x.poids + y.poids* produit un arbre optimal pour l'alphabet S' dans lequel tous les caractères *x.symbole* et *y.symbole* ont été remplacés par un même nouveau symbole.

Indication : raisonner par l'absurde.

Prouver l'algorithme en utilisant les lemmes 1 et 2.

Correction. On va montrer que l'assertion *la file de priorité contient une forêt, c'est-à-dire un ensemble d'arbres, incluse dans un arbre de codage optimal de l'alphabet S* est vraie à chaque fin de l'itération principale.

D'abord, la question de l'existence d'un arbre optimal ne se pose pas. En effet, l'ensemble des arbres binaires complets avec K feuilles étiquetées – ensemble des codes préfixes complets – est fini, donc admet au moins un élément de longueur moyenne minimale.
Initialisation : L'assertion est donc vraie au début de l'itération principale – une feuille est un sous-arbre de tout sous-arbre optimal.
Cas inductif: Si l'assertion est vraie à l'entrée de l'itération, il existe un arbre optimal contenant la forêt incluse dans la file de priorité. Soit x et y les nœuds extraits de la file de priorité, d'après le lemme 1 il existe un arbre optimal tel que x et y soient 2 feuilles sœurs. Par le lemme 2, l'arbre optimal, lorsque l'on réalise la fusion de x et y, reste optimal.
Terminaison: L'algorithme, faisant un nombre fini d'itérations, se termine. Chaque itération diminue de 1 le nombre de nœuds dans la file de priorité.
Conclusion: À la fin des itérations, il ne reste qu'un nœud qui est la racine d'un arbre optimal.

Exercices non corrigés

Exercice 1

Soit $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ définie par : $f(0,n) = n + 1$,
$f(m, 0) = f(m - 1,1)$,
$f(m, n) = f(m - 1, f(m, n - 1))$.
1) Montrer que $f(m, n)$ est définie pour tout couple $(m, n) \in \mathbb{N} \times \mathbb{N}$.
2) Soit g : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ définie par : $g(0,n) = n + 1$,
$g(m, 0) = g(m - 1, 1)$,
$g(m, n) = g(m - 1, g(m, n + 1))$.
Pour quels couples $(m, n) \in \mathbb{N} \times \mathbb{N}$ la fonction $g(m, n)$ est-elle définie ?

Exercice 2

Modifier le programme PARCOURS-IN pour obtenir le parcours préfixe – resp. suffixe ou postfixe.

Exercice 3

Écrire un algorithme récursif qui cherche le maximum d'un arbre binaire de recherche – voir l'algorithme 12.

Exercice 4

Écrire un algorithme récursif qui cherche un élément dans un arbre binaire de recherche – voir l'algorithme 11.

Exercice 5

Modifier PARCOURS-LG pour calculer la distance à s de tous les sommets de G, c'est-à-dire que le traitement d'un sommet n consiste à calculer la distance entre n et s.

Exercice 6

Pour un alphabet de $K = 4$ symboles, donner un code optimal en fonction de $p_1, \ldots, p_4$. Indication : on peut supposer sans perte de généralité que les p_i sont ordonnés, on peut également supposer que l'arbre est complet – argument d'optimalité. Il y a 2 types d'arbres complets ayant 4 feuilles et le choix de l'arbre se fait sur la comparaison de p_1 à $p_3 + p_4$.

Exercice 7

Soit A un arbre binaire ayant K feuilles dont les distances à la racine sont respectivement $l_1, \ldots, l_K$. Montrer l'inégalité de Kraft

$$\sum_i 2^{-l_i} \leq 1.$$

Exercice 8

Montrer qu'un code ayant la propriété du préfixe n'est pas ambigu.

Questions d'enseignement

On pourra utiliser les exemples proposés dans l'introduction de ce chapitre pour donner une première intuition sur les algorithmes, par exemple :

– la résolution d'une équation du second degré permet d'introduire les *tests* ;

– le calcul du pgcd permet de montrer la nécessité d'aller au-delà de l'*algorithme naïf* ;

– l'établissement d'une table de logarithmes permet de montrer la possibilité d'erreurs dues à des causes diverses lors de l'exécution à la main et l'utilité des *bureaux de calcul*.

Il est aussi important d'expliquer qu'un algorithme *ne se réduit pas* à un programme : il est la description précise d'une méthode pour résoudre un problème, alors qu'un programme est l'incarnation d'un algorithme dans un langage de programmation, destinée à être exécutée sur une machine. De même que « *to work* » et « travailler » expriment une même action selon que l'on parle anglais ou français, de même « $x = 1$ » et « $x := 1$ » expriment le même algorithme selon que l'on « parle » Java ou Pascal.

Conception et analyse d'algorithmes

Pour l'élève, la conception d'algorithmes repose sur un ensemble de compétences qu'il est nécessaire d'acquérir simultanément et progressivement.

En effet, pour concevoir un algorithme, il est indispensable de *spécifier* clairement le problème posé, en examinant quelles seront les données en entrée et quel est le résultat attendu. Même s'il existe des langages formels de spécification, au niveau de la Terminale, un énoncé en français décrivant les données du problème et leur format ainsi que le résultat est suffisant. Un ensemble d'exemples peut compléter la description du problème et aider l'élève à construire un jeu de tests pour analyser son algorithme.

Le problème posé sous-entend également l'ensemble des opérateurs qui sont à disposition pour le résoudre, c'est-à-dire que l'algorithme repose sur un ensemble d'opérateurs, une « machine abstraite » sur laquelle va s'exécuter l'algorithme. Par exemple, on peut disposer d'un opérateur de tri, de structures de données telles que les piles, files, arbres avec les opérateurs associés, ou encore d'opérateurs plus complexes. *A contrario*, on pourrait également se contraindre à ne disposer que des opérations arithmétiques, de l'affectation, de la séquence, du test et de la boucle. La solution algorithmique proposée serait évidemment bien différente. De manière générale, un bon choix de niveau d'abstraction de la machine permet de clarifier l'écriture d'un algorithme et de mettre l'accent sur ses propriétés. À titre d'exercice, il est particulièrement intéressant de comparer l'écriture d'un même algorithme dans différents ouvrages d'algorithmique, quel est le niveau de détail choisi, quels sont les types abstraits utilisés.

Le niveau d'abstraction étant choisi, la difficulté pour l'élève est de *décomposer le problème* initial en sous-problèmes « plus simples ». Par exemple, le tri rapide – l'algorithme 4 – fait appel à une fonction qui réalise une segmentation d'une partie du tableau – l'algorithme 5 – et cette fonction de segmentation est réutilisée récursivement. Enfin, pour être efficace en conception, il est important de savoir reconnaître dans un problème posé des schémas génériques d'algorithmes – une recherche, un tri, un parcours de graphe ou d'arbre, sous-mots d'une séquence… – et les structures de données qui sont associées – piles, files de priorité, arbres binaires, tables de hachage… C'est la raison de la présentation d'algorithmes classiques dans ce chapitre dont l'objectif est de donner un ensemble de schémas de référence qui se retrouveront dans de nombreux contextes de l'informatique.

En général, un algorithme ne se déduit pas directement de la spécification du problème. Lors du travail de conception, de multiples solutions algorithmiques sont écrites, de la plus naïve à la plus efficace. Lorsqu'une version d'un algorithme est écrite, il est nécessaire de *vérifier* que l'algorithme répond bien aux spécifications du problème. Si quelques exemples permettent rapidement de se convaincre du bien-fondé de l'algorithme, cela reste insuffisant pour garantir que l'algorithme est correct. Ainsi, à chaque algorithme est associée une *preuve* qui exprime, à partir de l'état des variables au cours de l'exécution de l'algorithme, les

propriétés qui doivent être vérifiées par ses variables. Le niveau de formalisme des preuves varie selon les exigences de la spécification et le problème posé.

Cependant, un algorithme correct n'est pas nécessairement efficace, car sa *complexité en temps*, le nombre d'opérations nécessaires pour obtenir la solution – abstraction du temps d'exécution du programme –, est trop grande pour que le programme soit pratiquement utile. Le cas échéant, on utilise la complexité en espace, c'est-à-dire la taille de la mémoire utilisée lors de l'exécution de l'algorithme. L'évaluation de la complexité d'un programme repose en général sur des approches combinatoires, on compte le nombre d'opérations à effectuer sur chaque entrée de l'algorithme et on raisonne souvent par encadrements – bornes au pire et au mieux – de la complexité. La démarche de calcul de la complexité d'un algorithme est très proche de la démarche de preuve, toutes deux entraînent l'élève à analyser le comportement de l'algorithme et à établir ses propriétés.

Algorithmique et programmation

L'activité algorithmique est indissociable du développement de programmes qui implantent les algorithmes. Cette concrétisation permet à l'élève de mieux comprendre, par observation de la trace d'exécution, le déroulement de l'algorithme, l'évolution des valeurs de variables, les propriétés vérifiées par ces valeurs, etc. Des tests quantitatifs ou l'usage d'outils de *profiling* lui indiqueront le coût du programme en temps et en mémoire. Cette observation soulèvera des questions de correction ou de complexité, permettant la construction d'un algorithme moins naïf et plus efficace.

Le choix du langage de programmation dépend de plusieurs facteurs. De manière générale, on pourrait implanter un algorithme dans n'importe quel langage classique. Cependant, certains langages permettent de mieux exprimer la récursion – les langages fonctionnels – ou l'usage de la mémoire – les langages impératifs – ou encore des propriétés de généricité – les langages objets. De fait, le choix du langage dépend fortement du contexte dans lequel se situe le projet de développement de programme. On choisira donc son langage en fonction du problème posé, du cadre dans lequel le programme doit s'insérer – projet logiciel de plus grande ampleur –, de l'environnement de développement associé au langage – outils logiciels, éditeurs, débogueurs, profileurs… –, ou encore de ses goûts personnels. Un excellent exercice consiste à écrire le même algorithme dans différents langages de programmation, de remarquer les différences de constructions du programme, les différents modes d'expression de l'affectation, des itérations, des conditionnelles, de la manière dont est gérée la mémoire…

Propriétés des algorithmes

Après avoir donné quelques exemples d'algorithmes, nous conseillons vivement à l'enseignant de dire explicitement quelles sont les propriétés qu'un algorithme doit avoir ; nous reprenons ici la description donnée par Knuth, qui caractérise un algorithme comme étant une « recette de cuisine », ou encore un nombre fini de règles que l'on applique dans l'ordre donné dans la « recette » pour obtenir la solution d'un problème. Toutefois, cette « recette » doit satisfaire aux contraintes suivantes :

– Définition non ambiguë : chaque étape d'un algorithme doit pouvoir être effectuée par un ordinateur ou un robot et ce d'une seule manière possible. Par exemple, l'algorithme de Prim donné plus haut – algorithme 14 – ne satisfait pas à cette condition de non-ambiguïté puisque, à la ligne 12, plusieurs choix sont possibles. Notons toutefois que la notion d'algorithme a depuis été généralisée de manière à inclure les algorithmes non déterministes et probabilistes.

– Définition effective : chaque étape d'un algorithme doit pouvoir être effectuée par un humain de manière exacte et en un temps fini. Par exemple, il n'y a pas d'algorithme pour diviser deux réels donnés par une représentation décimale infinie.

– Données : l'algorithme prend en entrée des données qui sont précisées par le problème à résoudre.

– Résultats : on peut montrer que l'algorithme fournit des résultats qui sont ceux demandés par le problème à résoudre.

Preuves des algorithmes

Nous avons illustré la *complexité des algorithmes* – voir les algorithmes 1, 18, etc. Il est aussi possible de considérer l'aspect *preuve de correction* des algorithmes, à savoir, le résultat de l'algorithme est-il bien le résultat attendu ? Cela peut se faire au moyen d'assertions que l'on met à des lignes stratégiques de l'algorithme et que l'on doit vérifier – voir par exemple les assertions mises entre /* */ dans les algorithmes 1, 18. Toutefois, il faut ensuite *démontrer* ces assertions formellement : on pourra se reporter au chapitre 14.3 de André Arnold, Irène Guessarian, *Mathématiques pour l'informatique*, 4e ed., EdiScience, Dunod (Paris), 2005, pour des exemples détaillés de preuves d'algorithmes simples.

Questions d'implantation

Remarquons que les $\wedge$ (et) qui figurent dans les conditions d'arrêt des boucles **while** – ligne 3 de l'algorithme 6, ligne 5 de l'algorithme 7, ligne 3 de l'algorithme 11, etc. sont implantés comme des opérateurs *paresseux* ou *séquentiels*, c'est-à-dire : on évalue d'abord le premier booléen de la conjonction,

ensuite et *uniquement* si ce premier booléen est vrai, on évalue le booléen suivant de la conjonction. Si l'on évaluait en parallèle, ou dans un autre ordre, ces booléens, on pourrait avoir des erreurs : par exemple ligne 6 de l'algorithme 18, si $i = 0$, $T[i]$ n'est pas défini : en Java le *et logique* `&&` – voir le deuxième chapitre – est un opérateur paresseux qui permet d'éviter ces erreurs.

Questions de programmation

Dans l'algorithme 13, par exemple, on doit calculer plusieurs tableaux indexés par des sommets, or les indices d'un tableau sont normalement des entiers : comment faire ? Par exemple, pour calculer *pred*, on calculera un tableau de couples $(n, pere(n))$ où $pere(n)$ est le sommet de G qui sera le père de n dans l'arbre couvrant : on déclarera un tableau *pred* de paires de sommets – Var tableau de couples de sommets *pred* – indexé par un entier – indice entier i, initialisé à $i = 1$ – ; la ligne 8 $pred(s) = NIL$ de l'algorithme 13 sera remplacée par $pred(1) = (s,\text{NIL})$, et la ligne 17 $pred(v) = n$ sera remplacée par les deux lignes

i = i + 1
pred(i) = (v, n)

Compléments

Recherche de motifs

Nous étudierons la recherche d'un motif dans un texte, c'est-à-dire, étant donné un motif p et un texte t qui sont deux chaînes de caractères, quelles sont les occurrences de p dans t. Supposons donnés un texte t sous forme d'un tableau de n lettres prises dans un alphabet A, et un motif p, qui est un tableau de m lettres prises dans A.

Algorithme naïf de recherche de motif

L'algorithme naïf fait glisser le motif le long du texte, en commençant à la première lettre t [1], puis en comparant lettre à lettre, et ce jusqu'à la position $n - m$ du texte.

Algorithme 22: Recherche des occurrences du motif p dans le texte t

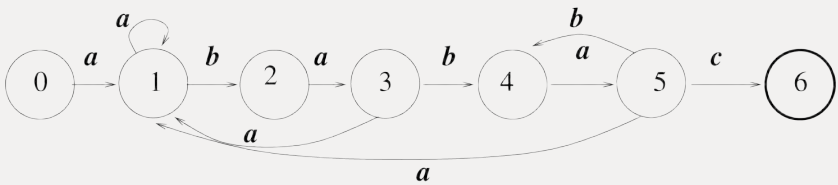
```
PATTERN-MATCH (chaîne de caractères t, p)
  Données : Une chaîne de caractères t et une chaîne de caractères p
  Résultat : Les occurrences de p dans t
1 Var entier n, m, i
```

```
2 n = length(t)
3 m = length(p)
4 for i = 0 to n − m
5 |  if (p[1...m] == t[i + 1...i + m])
6 |  |  afficher p apparaît à la position i dans t
```

Cet algorithme fait $m \times (n - m + 1)$ comparaisons, dont un certain nombre sont inutiles : en effet, après chaque échec, on recommence à comparer à partir de l'indice $i + 1$ sans tenir compte des comparaisons de i à $i + m - 1$ qui ont déjà été faites. Certaines de ces comparaisons inutiles peuvent être évitées si l'on prétraite le motif p pour construire un *automate fini* – voir le deuxième chapitre – avec lequel on peut ensuite traiter le texte t avec une lecture directe sans retour en arrière.

Algorithme amélioré utilisant un automate fini

Au motif p on associe un automate fini $A(p)$ qui reconnaît toutes les suites de lettres ayant p pour suffixe. Si p = *ababac*, on peut construire $A(p)$ par l'algorithme 23



L'automate du motif p = ababac*: toutes les transitions non dessinées vont à l'état initial 0; l'état final est l'état 6. L'état 1 reconnaît les mots se terminant par* a*, l'état 2 reconnaît les mots se terminant par* ab*, l'état 3 reconnaît les mots se terminant par* aba*, l'état 4 reconnaît les mots se terminant par* abab*, l'état 5 reconnaît les mots se terminant par* ababa*, et l'état 6 reconnaît les mots se terminant par* ababac*, c'est-à-dire le motif recherché.*

L'algorithme 24 se décompose en deux parties. AUTOM prétraite p et calcule l'automate $A(p)$: ce précalcul de l'algorithme 23 prend un temps $O(m^3 \times S)$, où S est le nombre de lettres de l'alphabet A : en effet les boucles *for* (lignes 4 et 5) sont exécutées $m \times S$ fois, la boucle *repeat* peut s'exécuter $m + 1$ fois et le test de la ligne 9 peut effectuer m comparaisons. Ensuite, l'algorithme PATTERN parcourt le texte t pour trouver toutes les occurrences de p en temps linéaire en la taille de t en utilisant $A(p)$.

Algorithme 23: Calcul de l'automate des suffixes $A(p)$ associé au motif p

```
AUTOM (chaîne de caractères, p)
  Données : Une chaîne de caractères p
  Résultat : La table des transitions de l'automate A(p)
1 Var entier k j
2 Var tableau δ: {0,...,m} × A ⟶ {0,...,m}
  /* L'automate a m+1 états 0,...,m et ses transitions sont
     représentées par la fonction δ                          */
3 m = length(p)
4 for j = 0 to m
5 |   for (a ∈ A)
6 |   |   k = min(m+1, j+2)
7 |   |   repeat
8 |   |   |   k = k − 1
9 |   |   until (p[1]...p[j]a est un suffixe de p[1]...p[k])
10|   |   δ(j, a) = k
11 return δ
```

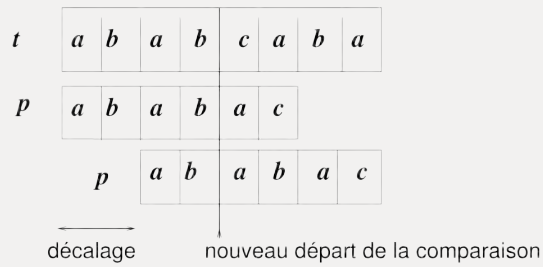
Algorithme 24: L'algorithme de recherche de motif utilisant l'automate des suffixes calculé par l'algorithme 23

```
PATTERN(chaîne de caractères t, p)
  Données : Une chaîne de caractères t et une chaîne de caractères p
  Résultat : Les occurrences de p dans t
1 Var entier i , j
2 n = length(t)
3 m = length(p)
4 δ=AUTOM(p)
5 j = 0                          // On initialise l'état de l'automate
6 for i = 1 to n
7 |   j = δ(j, t[i])
8 |   if (j == m)   // L'état final est atteint, une occurrence du
  |   motif est trouvée
9 |
10|   |   afficher "p apparaît à la position i + 1 − m dans t"
```

Algorithme KMP (Knuth-Morris-Pratt)

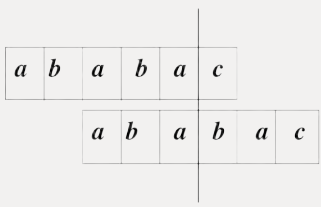
Cet algorithme améliore encore l'algorithme 24 en évitant même de pré-calculer la fonction de transition δ de l'automate des suffixes – algorithme

23 –: au lieu de précalculer $A(p)$, on précalcule un « tableau des préfixes » *pref*(p) du motif p ; ensuite, lorsqu'on lit le texte t, on a recours à *pref*(p) pour calculer *efficacement* et « à la volée » les transitions δ de $A(p)$, uniquement lorsqu'on en aura besoin – et non plus pour chaque lettre $t[i]$ du texte. Illustrons sur un exemple l'idée de KMP. Soit le motif $p[1…6]$ = *ababac* et le texte $t[1…9]$ = *ababcaba*. La comparaison entre le texte et le motif échoue à la cinquième lettre du motif ; toutefois, une observation du motif montre qu'il est inutile de recommencer la comparaison à partir de la deuxième lettre du texte – un *b*, puisqu'elle correspond à la deuxième lettre du motif et pas à la première. Il faut donc décaler le motif de 2 lettres et repartir de la troisième lettre du texte, dont on sait qu'elle est identique à la première lettre du motif. De plus, il est inutile de comparer les deux premières lettres du motif avec les troisième et quatrième lettres du texte, car on sait déjà qu'elles sont identiques, il suffit donc de repartir de la cinquième lettre du texte, c'est-à-dire du point où l'on avait échoué : au final, on ne parcourra le texte qu'une seule fois, soit un nombre n de comparaisons.



Décalage du motif p = ababac *sur le texte* t = ababcaba *après le premier échec.*

Pour implanter cet algorithme, il faut un prétraitement du motif et calculer un tableau qui donne pour chaque i, $1 \leq i \leq m$, le décalage à faire si la comparaison vient d'échouer à la position i du motif. Dans ce tableau, *pref*(p)[i] est la longueur du motif initial le plus long se terminant en $p[i]$, c'est-à-dire la longueur du plus long préfixe de p qui soit un suffixe de $p[1… i]$.



i	1	2	3	4	5	6
p	*a*	*b*	*a*	*b*	*a*	*c*
pref	0	0	1	2	3	0

Le tableau pref(p) *du motif* p = ababac.

Algorithme 25: Algorithme de Knuth-Morris-Pratt : recherche des occurrences du motif p dans le texte t

```
KMP (chaîne de caractères t, p)
  Données : Une chaîne de caractères t et une chaîne de caractères p
  Résultat : Les occurrences de p dans t
1 Var entier n, m, i j
2 Var tableau d'entiers pref
3 n = length(t)
4 m = length(p)
5 i = 0
6 pref =CALCULPREF(p)
7 for j = 0 to n
8     while ((i > 0) ∧ (p[i + 1] ≠ t[j]))
9         i = pref[i]
10    if (p[i + 1] == t[j])
11        i = i + 1
12    if (i == m)
13        afficher p apparaît à la position j − m dans t
14        i = pref[i]
```

Algorithme 26: Précalcul du tableau des préfixes de p pour l'algorithme de Knuth-Morris-Pratt

```
CALCULPREF(chaîne de caractères p)
  Données : Une chaîne de caractères p
  Résultat : Le tableau des préfixes de p
1 Var entier n, m, i j
2 Var tableau d'entiers pref
3 m = length(p)
4 i = 0
5 pref[1] = 0
6 for j = 2 to m
7     while ((i > 0) ∧ (p[i + 1] ≠ p[j]))
8         i = pref[i]
9     if (p[i + 1] == t[j])
10        i = i + 1
11        pref[j] = i
```

L'algorithme KMP se généralise à des recherches de motifs « avec des trous », par exemple le motif *vie* apparaît dans le mot *ville* si l'on permet des trous, pour reprendre le slogan publicitaire « dans *ville* il y a *vie* ». Cette problématique est particulièrement importante dans les études portant sur la génomique ou la fouille de données.

Pour aller plus loin

André Arnold, Irène Guessarian, *Mathématiques pour l'informatique*, 4e ed., Edi-Science, Dunod (Paris), 2005.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, 3e ed., MIT Press, London, 2009.

David Harel, *Algorithmics: The Spirit of Computing*, Addison-Wesley, Reading, Massachusetts, (1992).

Donald E. Knuth, *The Art of Computer Programming, Vol. 3, Sorting and Searching*, 2e ed., Addison-Wesley, Reading, Massachusetts, (1998).

Wikipédia `http://en.wikipedia.org/wiki/Boruvkas_algorithm`

Architecture

Concevoir des algorithmes qui transforment de l'information et les décrire dans des langages de programmation a comme objectif premier de permettre leur exécution par un ordinateur. Un ordinateur peut se décrire à plusieurs échelles. À l'échelle du nanomètre, c'est un assemblage de quelques millions de transistors, alternativement conducteurs et non conducteurs, dont l'évolution aboutit ultimement à l'apparition sur un écran de quelques pixels, auxquels nous accordons une signification. À une échelle un peu plus grande, c'est un processeur entouré de nombreuses cases mémoire, qui lit le contenu de certaines de ces cases et, en fonction du contenu lu, lit le contenu d'autres cases, le modifie en effectuant quelques opérations arithmétiques et logiques, avant de le réécrire en mémoire. À une échelle plus grande encore, c'est une machine capable d'exécuter des programmes écrits dans un langage fruste, le langage machine, mais vers lequel nous pouvons traduire tous les programmes écrits dans des langages évolués. Dans ce chapitre, un zoom arrière, du transistor à l'ordinateur, nous fera comprendre comment chacune de ces structures se construit à partir de structures de taille immédiatement inférieure.

Cours

Objectifs

L'ordinateur est fondé sur plusieurs idées simples : c'est une structure physique utilisant des circuits électroniques câblés une fois pour toutes, capable d'interpréter électriquement, on dit d'« exécuter », un programme constitué d'un ensemble d'instructions codées en binaire et traitées séquentiellement.

Ce jeu d'instructions binaires, qui constituent un *langage machine*, diffère d'une machine à l'autre. Mais tous ces langages reposent sur les mêmes principes et sont, en fait, très proches. C'est, à nouveau, les principes universels,

et non les détails propres à un langage ou un autre, qu'il importe de comprendre ici.

Un langage machine est conçu pour être facilement représentable et exécutable par un ordinateur. Mais les programmes exprimés dans ce langage, qui sont des suites de 0 et de 1, sont difficilement intelligibles pour les êtres humains. Aussi a-t-on introduit deux niveaux d'outils qui permettent d'exprimer les programmes dans des langages plus intelligibles : les langages d'assemblage et les langages de haut niveau.

Le langage d'assemblage reprend la structure du langage machine, mais il est plus facile à lire et à comprendre grâce à l'utilisation de mnémoniques, de symboles et d'étiquettes. Les programmes écrits dans ce langage sont traduits, quasiment ligne à ligne, en langage machine, par un outil appelé un *assembleur*.

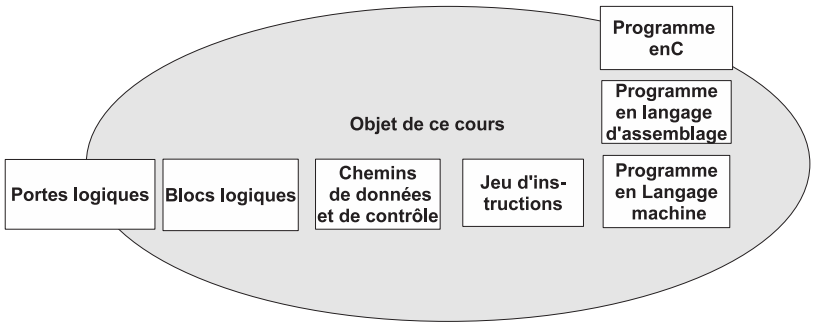
Cependant, un langage d'assemblage reste particulier à un type d'ordinateurs et nécessite donc, de la part du programmeur, une certaine spécialisation. On a donc introduit un second niveau d'outils : les langages de programmation de haut niveau, comme Java ou C, qui sont traduits par un compilateur en langage d'assemblage, puis en langage machine.

Les bases de ce chapitre se trouvent en électronique, en logique, en théorie des langages, en compilation et en algorithmique. Les compétences acquises seront, en retour, utiles en algorithmique, en programmation, en compilation, mais aussi en système et même en réseau.

Dans ce chapitre, nous verrons tout d'abord comment réaliser, avec des transistors, des circuits électroniques simples, en particulier des portes logiques. Nous utiliserons ensuite ces portes pour construire une architecture minimale, capable de traiter toutes sortes de programmes. Nous étudierons les instructions types et les principaux modes d'adressage de la mémoire utilisés dans les programmes écrits en langage machine. Nous terminerons par la description du fonctionnement d'un assembleur et par les rudiments de la traduction en langage d'assemblage de programmes écrits dans un langage de haut niveau. En conclusion, nous présenterons des extensions de ce modèle simplifié de machine, qui permettent de le rendre plus efficace et plus rapide.

Comme dans les autres chapitres, la méthode proposée ici laisse une large part à la création personnelle et à l'assimilation par des travaux pratiques dans lesquels les outils sont complètement ouverts et compréhensibles par tous : c'est pourquoi, pour comprendre ces notions, l'utilisation de simulateurs est préconisée à tous les niveaux. Les simulateurs permettent de réali-

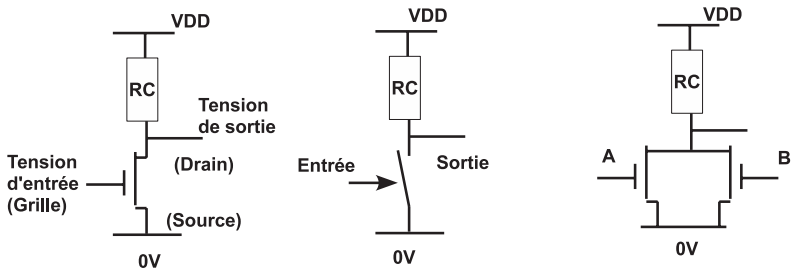
ser à tout moment des modifications personnelles et donc de s'approprier sa propre version de ces outils.



Portes logiques

Les transistors, qui constituent la plus grande partie des circuits d'un ordinateur, sont des dipôles à impédance contrôlée par une tension externe. En les simplifiant à l'extrême, on peut considérer que cette impédance entre les électrodes de drain et de source est soit nulle soit infinie, en fonction de la tension appliquée à l'électrode d'entrée, appelée grille. Ces transistors sont alors équivalents à un court-circuit ou à un circuit ouvert.

Cette vue d'un transistor NMOS, par exemple, est très simplifiée, mais elle permet cependant de comprendre le fonctionnement d'une porte logique. Le premier exemple est un inverseur : il comporte un transistor en série avec une résistance de charge fixe. L'électrode de sortie de ce montage est le drain VDD, qui est connecté à une résistance de charge, elle-même connectée à la tension d'alimentation du circuit. Si la tension de grille est trop faible pour que le transistor soit dans un état « passant », la tension de sortie se rapproche de la tension d'alimentation. Si, en revanche, la tension de grille dépasse un certain seuil, le transistor devient passant et court-circuite la sortie à la masse, ce qui conduit à une sortie de tension nulle. Si l'on définit un état logique 0 comme une tension proche de 0V et un état 1 comme une tension proche de la tension d'alimentation, ce circuit électronique est donc appelé inverseur, car si l'on considère que la tension d'entrée – de grille – est au niveau logique 0 – inférieure à la tension de seuil – la sortie est à un niveau logique 1, et vice versa.

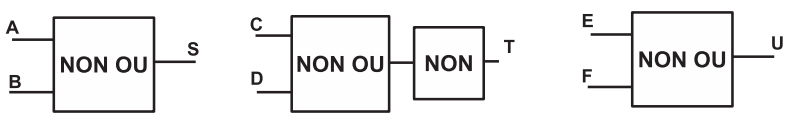


a) porte inverseuse, b) schéma simplifié de a,
c) porte NON OU à 2 entrées, C = NON (A OU B)

Le second circuit est encore plus intéressant, car il comporte deux transistors montés en parallèle et une seule résistance de charge. Pour que le circuit donne une sortie logique à 0, il suffit que l'un ou l'autre des signaux d'entrée A et B soit à un potentiel élevé – 1 logique. Autrement, la sortie sera un potentiel élevé – 1 logique. Ces portes logiques sont appelées NON OU ou NOR. On peut résumer cela à l'aide d'un tableau appelé table de vérité.

A	B	C = NON (A OU B)
0	0	1
0	1	0
1	0	0
1	1	0

Il est possible à partir de ces types de portes de réaliser des portes OU, en connectant la sortie d'une porte NON OU sur un inverseur – porte NON – et des portes ET, en connectant les entrées sur des inverseurs.



a) porte NON OU, b) Porte OU, c) porte ET

Cette partie, montrant le passage d'un circuit électronique à un circuit logique, relève plus d'un cours d'électronique que d'informatique. Cependant, ce survol pose quelques éléments d'évaluation de la complexité, au sens informatique, d'un circuit : la surface physique occupée par un circuit est proportionnelle à la surface d'un transistor multipliée par le nombre d'entrées des portes logiques

employées ; le temps de réponse d'un circuit est proportionnel au temps de basculement d'un transistor multiplié par le nombre de portes logiques se trouvant sur le plus long chemin menant d'une entrée à une sortie du circuit.

Il existe deux grands types de fonctions logiques : les fonctions combinatoires, qui n'ont pas de mémoire et sont indépendantes du temps, et les fonctions séquentielles, qui tiennent compte du passé et dont les sorties dépendent donc non seulement des entrées actuelles mais aussi de l'état précédent du circuit.

Une fonction logique combinatoire est une fonction de $\{0,1\}^n$ dans $\{0,1\}^p$. Elle transforme un ensemble de n bits d'entrée en un ensemble de p bits de sortie et est indépendante du temps. Toute fonction logique combinatoire peut être représentée par sa table de vérité qui définit la sortie d'un circuit, à partir de ses entrées. Cette table indique aussi comment créer un circuit réalisant cette fonction avec des portes ET, OU, et NON. Une méthode élémentaire consiste à faire la somme – c'est-à-dire le OU, la disjonction – des différents cas où la fonction est vraie, chaque cas s'exprimant comme le produit – c'est-à-dire le ET, la conjonction – de l'ensemble des entrées. Les entrées positives 1 se retrouvent dans le produit telles quelles, les entrées négatives 0 sont complémentées dans le produit, avec une porte NON en série. Les formules obtenues sont dites en forme normale disjonctive. Il existe de nombreuses manières d'optimiser cette construction, afin de réduire le nombre de portes : la méthode de Karnaugh, la méthode de Quine-McCluskey, etc., qui ne sont pas présentées ici.

Dans ce qui suit, les notations utilisées seront $\overline{X}$ pour représenter NON X, ET et OU pour représenter le ET et le OU logiques.

Tables de vérité, formules booléennes et circuits sont trois façons de représenter de manière concrète les mêmes objets abstraits : les fonctions booléennes. Des exercices classiques demandent de passer d'un formalisme à l'autre. En particulier, les exercices portant sur le dessin de circuits sont très ouverts, car il y a de nombreuses façons de dessiner des circuits. Il y a même un aspect assez ludique à chercher à obtenir des dessins clairs, réguliers ou compacts…

Création de blocs logiques combinatoires à partir des portes logiques

Voyons maintenant comment utiliser ces portes logiques pour réaliser des circuits combinatoires plus complexes.

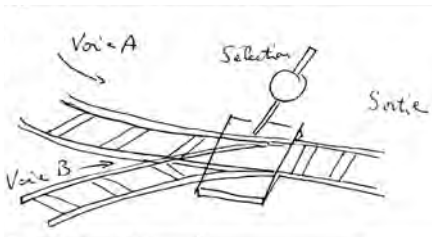
La notion de bus

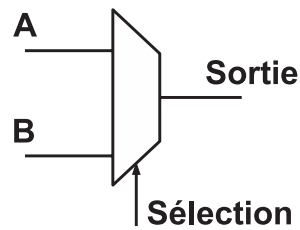
Les signaux transitant sur un fil sont des signaux binaires : chaque signal représente un bit, 0 ou 1. Si l'on désire représenter, par exemple, des nombres

entiers, il est nécessaire de les représenter en base 2, comme nous l'avons vu au premier chapitre, et d'utiliser un fil par chiffre binaire. Avec n fils, on peut représenter les nombres compris entre 0 et $2^n - 1$. Ces n fils, commutés simultanément, constituent un *bus*. Un bus est connecté à un émetteur qui impose une tension sur chaque fil, et à un ou plusieurs récepteurs, qui utilisent les valeurs véhiculées. Un bus ne peut avoir, à un moment donné, qu'un seul émetteur.

Réalisation d'un multiplexeur à deux voies

Un multiplexeur est une structure combinatoire qui sélectionne une valeur parmi plusieurs, en fonction d'une adresse. On peut le comparer à un aiguillage de chemin de fer qui, de même, transfère vers la sortie, une voie ou une autre.





Multiplexeur à 2 voies

Le numéro de la voie choisie est lui-même codé en binaire, et donc représenté par 1 bit pour sélectionner une voie parmi 2, 2 bits pour sélectionner une voie parmi 4, etc. Par exemple, la table de vérité d'un multiplexeur à deux voies est

A	B	Selection	Sortie
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

Et la formule logique d'un tel multiplexeur est :

$$\text{Sortie} = (A \text{ ET } \overline{\text{Selection}}) \text{ OU } (B \text{ ET Selection})$$

On remarquera que le nombre de fils de sortie est égal au nombre de fils des entrées A ou B.

Réalisation d'un additionneur en nombres binaires naturels

Quand on réalise une addition entre deux bits, le résultat peut être 0, 1 ou 2, ce qui se représente en binaire sous la forme 00, 01 ou 10. Le bit de poids faible est appelé « résultat » et le bit de poids fort est appelé « retenue sortante ». Si l'on ajoute deux nombres de plus de 1 bit, on commence par la somme des deux bits de poids faible, qui donne un résultat sur un bit et une retenue. Pour le calcul du bit suivant, la retenue obtenue par la somme des bits de poids faible doit être ajoutée aux bits à ajouter. En appelant ce bit de retenue *Cin* (*Carry in*, retenue entrante), l'addition se fait alors sur A, B et *Cin*, et donc le résultat peut être maintenant 0, 1, 2 ou 3 qui se code toujours sur 2 bits : le bit de résultat et le bit de retenue sortante que nous appellerons *Cout* (*Carry out*, retenue sortante). Donc, pour chacun des bits autres que le tout premier, pour lequel *Cin* vaut 0, la somme est réalisée en additionnant 3 bits de même poids : le bit issu de la première entrée A, celui de la deuxième entrée B et le bit de retenue provenant de l'étage antérieur *Cin*.

Cela se traduit donc très simplement par la table de vérité suivante :

A	B	Cin	C out	Somme
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

D'où l'on déduit immédiatement les équations logiques suivantes :

Somme = ($\overline{A}$ ET $\overline{B}$ ET *Cin*) OU ($\overline{A}$ ET B ET $\overline{Cin}$) OU (A ET $\overline{B}$ ET $\overline{Cin}$) OU (A ET B ET *Cin*)

Cout = ($\overline{A}$ ET B ET *Cin*) OU (A ET $\overline{B}$ ET *Cin*) OU (A ET B ET $\overline{Cin}$) OU (A ET B ET *Cin*)

Le travail sur les tables de vérité et sur les circuits que l'on peut dessiner à la main a des limites – en particulier, celles de la feuille utilisée. Le multiplexeur, l'additionneur donnés précédemment pour une largeur de 1 bit s'étendent à des circuits de 32 bits, ou plus, sous réserve de savoir combiner algorithmiquement les circuits de base, sur un bit, obtenus lors de ces premières étapes. Pour obtenir un multiplexeur à deux entrées sur 32 bits donnant une sortie sur 32 bits, il suffit de dupliquer le multiplexeur précédent 32 fois, une fois pour chaque fil du bus ; mais pour obtenir un multiplexeur à 32 entrées, sur 1 bit, avec une sélection sur 5 bits, donnant le numéro de l'entrée sélectionnée en

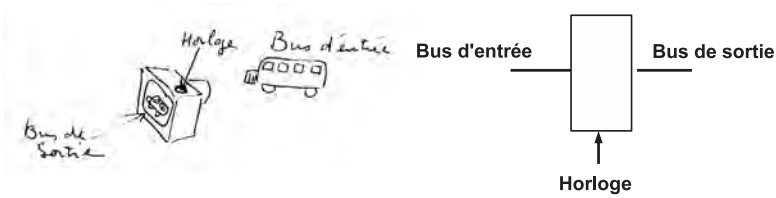
sortie, l'assemblage est plus ardu et demande un peu d'invention. Nous renvoyons le lecteur intéressé par cet aspect de l'architecture, l'algorithmique des circuits, au livre *Arithmétique des ordinateurs* de Jean-Michel Muller.

Blocs de logique séquentielle

La logique séquentielle calcule ses sorties non seulement à partir de ses entrées, mais aussi à partir des états précédents. Tous les circuits séquentiels présentés ici utilisent un signal externe appelé horloge. Ils ne changent d'état que lorsque ce signal d'horloge a une valeur bien précise, on parle alors de synchronisation sur niveau, ou change de valeur, s'il fonctionne avec un front.

La bascule D est la fonction de mémorisation la plus simple : elle est considérée ici comme un composant électronique possédant une entrée pour les données appelée D, une sortie Q et naturellement une horloge H. Son rôle est d'enregistrer la valeur de l'entrée D en fonction du signal d'horloge. Il existe différents types de bascules qui réagissent soit sur un niveau, soit sur un front de l'horloge. Dans une bascule sensible au niveau, la sortie Q prend la valeur de D présente quand, par exemple, H est à 1, et conserve la valeur prise quand H est à 0.

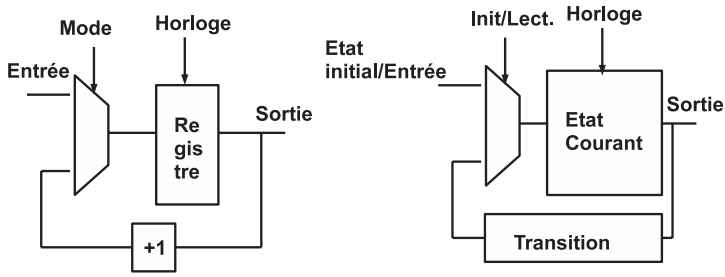
Un *registre* est un ensemble de bascules D connectées chacune en entrée sur l'un des fils d'un bus d'entrée et en sortie sur le fil de même indice du bus de sortie. L'horloge H est commune à toutes les bascules constituant le registre. Bascules D et registres se comportent donc comme des appareils de photo qui enregistrent une image d'entrée lorsque l'on appuie sur le déclencheur H et, pour compléter cette analogie, qui affichent alors l'image acquise sur l'écran LCD de l'appareil.



Un registre.

Le *compteur* est un outil qui permet de réaliser sur action du signal d'horloge, soit le chargement, soit l'incrémentation d'une donnée enregistrée dans un registre, et ce en fonction d'un bit de mode – par exemple, si mode = 0, on effectue un chargement et, si mode = 1, la valeur est incrémentée d'une unité.

Une réalisation possible de ce compteur comporte un registre associé à un additionneur et à un multiplexeur.



a) Compteur, b) Forme générale d'un circuit séquentiel.

La forme de ce circuit est particulièrement représentative des circuits séquentiels avec initialisation, avec un registre conservant l'état d'une variable, et une boucle de retour prenant comme entrée cette valeur pour fournir au prochain top d'horloge une nouvelle valeur à la variable : c'est la fonction de transition. En particulier, les automates à états finis – forme élémentaire de modélisation du calcul algorithmique, voir le deuxième chapitre, paragraphe « États et transitions » – peuvent être représentés à l'aide de circuits de cette forme.

Pour arriver à une forme de circuits séquentiels significativement plus complexes, une méthode consiste à séparer dans la réalisation d'un circuit souhaité la partie algorithmique, modélisée à l'aide d'un automate d'états finis, et dépendant du comportement du circuit souhaité, et la partie calculatoire, qui effectue les calculs sur les données à l'aide d'une unité arithmétique et logique (UAL) associée à quelques registres via des bus. On notera que cette partie est similaire d'un circuit à l'autre. Partie algorithmique – on dit parfois partie contrôle ou PC – et partie calculatoire – on dit parfois partie opérative ou PO – communiquent : la partie algorithmique commande les calculs effectués par la partie calculatoire, les résultats de ces calculs servent à la partie algorithmique en retour. Cette décomposition est très proche de la structure d'un ordinateur – il ne manque que la mémoire et les entrées/sorties –, on la nomme parfois décomposition PC/PO.

La mémoire vive – ou RAM (*Random Access Memory*) – est un système capable d'écrire une valeur d'entrée ou de lire une valeur stockée dans l'une des N cases mémoires qui la constituent. Pour sélectionner la case mémoire désirée en lecture comme en écriture l'utilisateur donne une adresse. Seul le cas où la lecture de la mémoire est combinatoire et non destructive est considéré ici.

La mémoire vive peut dès lors être vue comme un ensemble de registres dont les entrées sont connectées sur le même bus. En écriture, on envoie le signal d'horloge uniquement vers le registre où l'on désire écrire la donnée d'entrée. En lecture, on sélectionne le registre désiré, grâce à un multiplexeur, auquel on communique l'adresse de ce registre.

La machine de Von Neumann

La machine de Von Neumann reprend la décomposition Partie Opérative/ Partie Contrôle des circuits séquentiels proposée précédemment et lui ajoute une mémoire vive (RAM), c'est pourquoi l'on dit parfois Machine RAM. La partie opérative implémente un algorithme d'analyse et d'interprétation du langage machine reconnu par l'ordinateur. Ce modèle de machine informatique, base de tous les processeurs actuels, comporte essentiellement, de façon visible au programmeur :

– une mémoire (RAM), qui contient les programmes et les données ;

– un compteur ordinal (CO) qui indique à chaque instant l'instruction en cours d'exécution. En général, ce compteur est incrémenté à la fin de chaque instruction de façon que la machine exécute l'instruction suivante. Pour effectuer un branchement, ou rupture de séquence, il suffit de modifier la valeur de ce compteur ;

– un ou plusieurs registres accumulateurs destinés à stocker des valeurs temporaires lors d'un calcul ;

– une unité arithmétique et logique (UAL), qui permet d'effectuer les calculs entre les données contenues en mémoire et l'accumulateur, ou entre les registres généraux si l'on utilise une version de ces machines appelée RISC (*Reduced Instruction Set Computer*).

De façon un peu moins visible, la machine comporte aussi pour la partie contrôle :

– un registre adresse mémoire qui permet de sauvegarder l'adresse d'une donnée pendant la durée d'exécution d'une instruction,

– tous les bus d'interconnexion qui permettent le traitement des données,

– un registre instruction qui conserve l'instruction en cours de traitement, et se trouve à l'interface entre la partie opérative et la partie contrôle,

– un système de décodage lié à l'automate de contrôle dont le rôle est de gérer le séquencement de chaque instruction,

– l'ensemble de la machine reçoit une horloge de période habituellement fixe et dont le signal peut être à deux ou plusieurs phases.

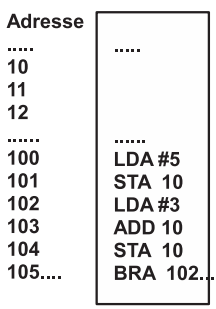
Pour simplifier, nous nous limitons au cas d'une machine à un seul registre accumulateur. Un programme informatique vu par une machine de Von Neu-

mann est de type impératif, c'est-à-dire que l'ordre d'exécution est imposé. Dans la grande majorité des cas, le passage d'une instruction à la suivante est séquentiel, sauf dans les cas où un branchement est demandé.

Il y a une très nette différence entre mémoire RAM et registres accumulateurs, même si l'un et l'autre sont capables de contenir des valeurs binaires : pour limiter le nombre d'instructions à une trentaine ou une soixantaine, par exemple, on ne permet qu'un seul accès à la mémoire de données par instruction. S'il faut réaliser des calculs entre plusieurs cases mémoire, la solution consiste à passer par un ou plusieurs registres qui serviront de stockage temporaire et dont l'accès est implicite. Par exemple, pour réaliser TOTO + TITI ⇒ TATA, où TOTO, TITI et TATA sont des cases mémoire, on programmera LDA TOTO – charger TOTO dans le registre A ou accumulateur –, ADD TITI – lui ajouter TITI –, STA TATA – sauvegarder le résultat dans TATA. Dans les années 70, certaines machines – dites machines CISC (*Complex Instruction Set Computer*) comme le VAX de DEC – ont utilisé des types d'instructions à trois adresses. Cela conduit à des nombres gigantesques d'instructions, plus de 300, avec des modes d'adressage très délicats, jusqu'à 7 pour chacun des trois opérandes, et donc à des matériels en définitive beaucoup plus lents et difficiles à maintenir.

Premières instructions, premiers modes d'adressage

Les données traitées par un programme sont de deux types : les constantes, dont la valeur est explicite dans le programme, et les variables, qui sont mises en mémoire à une adresse explicite dans le programme. Cette figure



CO	RI	A	RADM	MEM(10)
100	LDA #5	XX	XX	XX
		5	XX	XX
101	STA 10	5	XX	XX
		5	10	5
102	LDA #3	5	10	5
		3	10	5
103	ADD 10	3	10	5
		8	10	5
104	STA 10	8	10	5
		8	10	8
105	BRA 102	8	10	8
102	.....			

Un programme et son exécution.

représente la mémoire, le compteur ordinal, le registre instruction, l'accumulateur, le registre adresse mémoire et la valeur à l'adresse mémoire 10 avec leur évolution cycle par cycle. Chaque instruction dans ce schéma dure deux cycles :

lecture de l'instruction, décodage et exécution. Ce programme commence à l'adresse 100 où il charge la constante de valeur 5 dans l'accumulateur A. Le compteur ordinal passe à 101, et la valeur de A est recopiée à l'adresse 10. Puis, à l'instruction suivante, la constante 3 est mise dans A. Le compteur ordinal passe à 103 où l'on ajoute à A le contenu de la case 10. Puis, à l'instruction 104, le contenu de A est recopié dans la case mémoire 10.

Les instructions LDA #3 et ADD 10, par exemple, utilisent deux modes d'adressage différents : le premier est le mode IMMEDIAT, qui permet d'utiliser une constante, dont la valeur 3 est explicite dans l'instruction. Le second est l'adressage DIRECT où c'est l'adresse 10 de la donnée qui est explicite dans l'instruction.

La dernière instruction est une opération de branchement (*Branch Always*) qui permet de se brancher à l'instruction dont l'adresse est donnée en paramètre, ici 102. Donc, à chaque tour de cette boucle, comprise ici entre les instructions 102 et 105, la valeur immédiate 3 sera ajoutée à la case mémoire 10. Cette instruction fait partie des instructions de rupture de séquence. Ici, le mode d'adressage direct est utilisé. Le mode immédiat n'aurait aucun sens, puisque, par nature, un branchement demande de spécifier une adresse.

Chemins de données et unité de contrôle

Une analyse rapide du programme précédent montre que, du point de vue des données manipulées, le circuit électronique de l'ordinateur gère deux flux venant de la mémoire : celui lié au programme, en particulier lors du cycle de lecture de l'instruction, et celui lié aux données du programme. Ce circuit ne doit plus seulement effectuer un calcul à chaque étape, mais d'abord lire un morceau du programme qui indique les calculs à effectuer. Un ordinateur n'est pas une calculatrice : une calculatrice calcule, un ordinateur exécute un programme qui calcule. De plus, pour ajouter à la possible confusion de cette partie, les deux niveaux, programme et calcul, sont mis en œuvre au même endroit, au sein du même circuit.

L'ordinateur doit donc être capable :

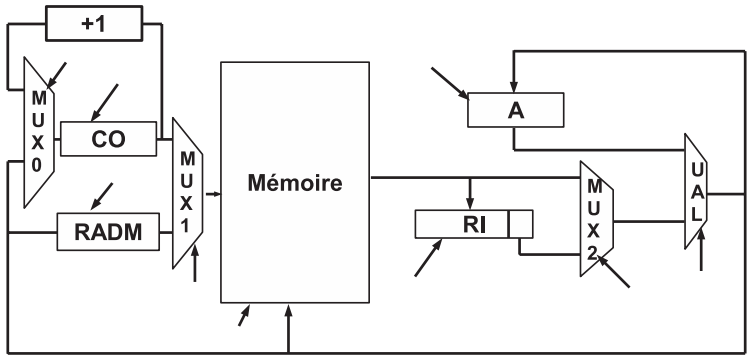
– de lire une case mémoire dont l'adresse est donnée par le compteur ordinal CO pour obtenir l'instruction de code à exécuter ;

– de lire ou d'écrire une case mémoire dont l'adresse est donnée par le programme pour la manipulation des valeurs du programme ;

– de transférer la sortie mémoire vers le registre instruction ou vers l'unité arithmétique et logique ;

– d'écrire une valeur que l'on peut avoir calculée dans A (LDA, ADD), dans la mémoire (STA) dans le CO (BRA) ou dans RADM.

Cette partie de l'ordinateur est qualifiée de « chemin de données ». De nombreuses solutions sont possibles pour la construire, par exemple



Le chemin de données initial.

Pour arriver à réaliser l'ensemble des opérations voulues, il est nécessaire d'ajouter les éléments combinatoires suivants : le multiplexeur MUX1 piloté par un bit de contrôle SELMUX1, pour sélectionner l'adresse mémoire choisie – instruction ou donnée, respectivement SELMUX1 = 0 et 1 –, le MUX2 – commandé par SELMUX2 – pour choisir entre une sortie mémoire ou le paramètre de l'instruction – respectivement SELMUX2 = 0 et 1 –, et l'unité arithmétique et logique. Celle-ci est une fonction combinatoire pour réaliser une fonction FUAL – pour l'instant, soit un choix entre ses deux entrées : A pour entrée accumulateur, B pour entrée issue de MUX2, soit une addition entre ses deux entrées, mais qui sera étendue plus tard à d'autres fonctions.

Le compteur ordinal CO est du même type que le compteur examiné ci-avant avec la possibilité de passer de CO à CO + 1, pour les exécutions en séquence, ou d'instancier CO avec une nouvelle valeur, pour les ruptures de séquence. Le registre instruction RI sera virtuellement découpé en deux champs distincts, le numéro d'instruction et le paramètre.

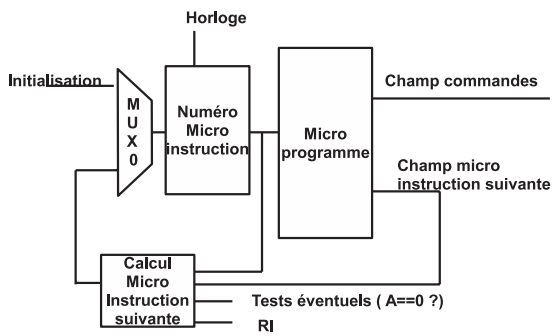
Les différents éléments du chemin de données reçoivent des commandes permettant soit une modification combinatoire, soit une modification liée au signal d'horloge. Des exemples de modifications combinatoires sont la commande FUAL, qui indique la fonction de l'unité arithmétique et logique (UAL) à exécuter, par exemple ici FUAL = A pour obtenir A, FUAL = B pour obtenir B ou FUAL = ADD pour obtenir A + B ; la commande CO + 1 qui permet de choisir d'incrémenter le CO ou de le charger à partir du bus de données, les valeurs de ces deux commandes étant imposées par l'unité arithmé-

tique et logique – on remarque que le bit de commande CO + 1 n'est autre que le bit SELMUX0, il est plus aisé de comprendre que le CO est incrémenté si ce bit est à 1 et que HCO = 1 – ; et les commandes SELMUX1 et SELMUX2. Dans le cas d'une modification liée au signal d'horloge, l'unité de contrôle fournit des bits permettant de valider le signal d'horloge au niveau du CO, de RADM, de la mémoire RAM, du registre instruction et de l'accumulateur – respectivement HCO, HRADM, HMEM, HRI, HACCU.

En contrepartie, le chemin de données – par l'intermédiaire de RI et A – fournit à l'unité de contrôle des informations sur le code de l'instruction courante et la valeur de l'accumulateur.

Le chemin de données vu ici est un ensemble totalement inerte, que l'on peut comparer à une marionnette, qui ne peut rien faire si l'on n'actionne pas ses fils. L'unité de contrôle est la partie qui permet de donner vie à cette marionnette. « Faire un pas en avant » est une instruction, que le marionnettiste décompose en plusieurs micro-instructions : lever le pied droit, l'avancer, le reposer, lever le pied gauche, l'avancer et le reposer, qui correspondent chacune au mouvement d'un fil de la marionnette. Quand le marionnettiste lit le programme « exécuter une valse », instruction par instruction, comme dans les livres initiant à l'art de la danse, il les réalise, l'une après l'autre, en décomposant chacune en plusieurs micro-instructions.

De même, chaque instruction du langage machine est décomposée par l'unité de contrôle en plusieurs micro-instructions. L'unité de contrôle est un circuit séquentiel comme celui présenté ci-avant. Plus en détail, ce circuit a la forme suivante



L'unité de contrôle.

Elle a pour entrées l'horloge, le numéro de l'instruction à réaliser, le numéro de l'état en cours et parfois des valeurs extérieures. Une des solutions les plus

compréhensibles pour réaliser un tel système est la machine dite microprogrammée, dont l'ensemble des micro-instructions à réaliser est enregistré dans une mémoire annexe, en général non modifiable, en fonction du numéro de l'état actuel, qualifié par la suite de « numéro de micro-instruction ». Le passage à l'état suivant est en général connu, sauf dans le cas du décodage d'une nouvelle instruction, où il faut utiliser le contenu du registre instruction RI pour décider de la micro-instruction à lancer.

Les instructions de base et leurs micro-instructions

Dans tout ce qui suit, les bits de validation d'horloge – HRI, HCO, HMEM, HA – sont considérés comme actifs s'ils sont à une valeur 1 logique. Tous les circuits reçoivent implicitement une horloge commune CK. De ce fait, l'horloge sur l'une des parties synchrones ne sera active que si l'horloge est active et que le signal de validation correspondant est à 1.

Au démarrage de la machine, nous supposerons que tous les registres sont mis à zéro. Il faut alors lancer l'opération de lecture du registre instruction, qui est commune à l'exécution de toutes les instructions. Cette opération nécessite une micro-instruction, appelée *décodage*, qui transfère le contenu de la mémoire d'adresse CO dans le registre d'instruction RI – Mem (CO) ⇒ RI. Les autres micro-instructions dépendent de l'instruction qui se trouve désormais dans le registre RI. Ces micro-instructions sont stockées en ROM (*Read-Only Memory*) – mémoire morte, c'est-à-dire non modifiable, dont la valeur est définie à la construction de l'ordinateur –, dans la partie contrôle de l'ordinateur, à une adresse $f(RI)$ qui dépend de l'instruction. Cette adresse peut, par exemple, être trois fois le numéro de l'instruction plus une constante. Cette opération s'écrit simplement SELMUX1 = 0, HRI = 1, micro-instruction = $f(RI)$. Si, par exemple, la première instruction à traiter est l'instruction LDA #5 et si le code du LDA# est n, la micro-instruction à traiter après la micro-instruction de décodage aura pour adresse $3 * n$ + constante.

Après la micro-instruction de décodage, il peut y avoir une ou plusieurs micro-instructions désignées par le numéro de l'instruction : une seule si l'instruction contient l'ensemble des informations nécessaires, plusieurs si l'instruction contient aussi l'adresse en mémoire des données à charger.

Détaillons ces micro-instructions pour quelques instructions classiques :

LDA # paramètre – chargement immédiat – la première micro-instruction correspondant à cette instruction réalise : Paramètre ⇒ A, CO + 1 ⇒ CO, aller à la micro-instruction de décodage

Ce qui se traduit immédiatement par

SELMUX2 = 1, FUAL = B, HA = 1, CO + 1 = 1, HCO = 1, aller à la micro-instruction de décodage

LDA paramètre – chargement de A en mode direct –, la première micro-instruction doit transférer le paramètre dans le registre RADM :

Paramètre ⇒ RADM, micro-instruction suivante, soit

SELMUX2 = 1, FUAL = B, HRADM = 1, micro-instruction suivante

la seconde micro-instruction réalise le stockage du résultat sur le bus dans l'accumulateur : M(RADM) ⇒ A, CO + 1 ⇒ CO, aller à la micro-instruction de décodage, soit

SELMUX1 = 1, SELMUX2 = 0, FUAL = B, HA = 1, CO + 1 = 1, HCO, aller à la micro-instruction de décodage

ADD #paramètre – addition en mode immédiat –, la première micro-instruction ajoute le paramètre au registre A

Paramètre + A ⇒ A, CO + 1 ⇒ CO, aller à la micro-instruction de décodage, soit

SELMUX2 = 1, FUAL = ADD,HA = 1,CO + 1 = 1, HCO, aller à la micro-instruction de décodage

ADD paramètre – chargement de A en mode direct –, la première micro-instruction doit transférer le paramètre dans le registre RADM : Paramètre ⇒ RADM, micro-instruction suivante, soit

SELMUX2 = 1, FUAL = B, HRADM = 1, micro-instruction suivante

la seconde micro-instruction réalise l'addition et le stockage du résultat dans l'accumulateur : M(RADM) + A ⇒ A, CO + 1 ⇒ CO, HCO, aller à la micro-instruction de décodage, soit

SELMUX1 = 1, SELMUX2 = 0, FUAL = ADD, HA = 1, CO + 1 = 1, HCO, aller à la micro-instruction de décodage

STA paramètre – stockage de A en mode direct – la première micro-instruction doit transférer le paramètre dans le registre RADM :

Paramètre ⇒ RADM, micro-instruction suivante, soit

SELMUX2 = 1, FUAL = B, HRADM = 1, micro-instruction suivante

la seconde micro-instruction réalise le stockage du résultat en mémoire : A ⇒ M(RADM), CO + 1 ⇒ CO, aller à la micro-instruction de décodage, soit

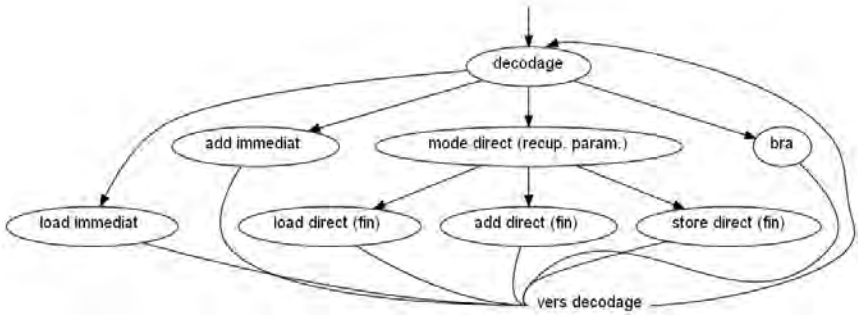
SELMUX1 = 1, SELMUX2 = X (0 ou 1, c'est indifférent), FUAL = A, HMem = 1, CO + 1 = 1, HCO, aller à la micro-instruction de décodage

BRA paramètre – branchement inconditionnel direct –, se traduit par

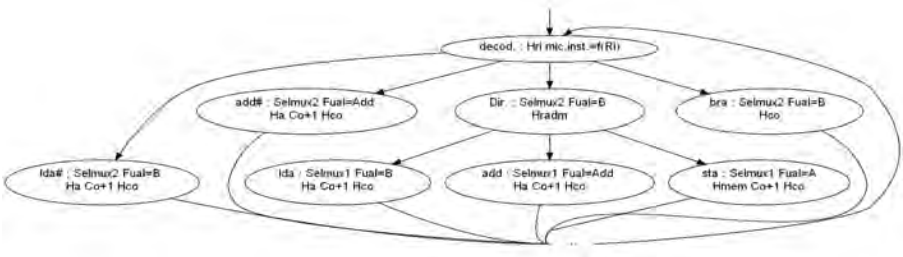
Paramètre ⇒ CO, aller à la micro-instruction de décodage

SELMUX2 = 1, FUAL = B, CO + 1 = 0, HCO, aller à la micro-instruction de décodage

Les microprogrammes d'interprétation peuvent être représentés par un graphe sous la forme d'un automate appelé graphe de contrôle.



Automate de contrôle (version symbolique).



Automate de contrôle (version avec les micro-instructions).

Réalisation de tests

Cette première version de machine n'a pas la possibilité de tester une valeur. Pour y parvenir, il est nécessaire de créer des instructions de branchement conditionnel. Plusieurs solutions existent pour cela, beaucoup de processeurs utilisent des *indicateurs* connectés à l'UAL qui sauvegardent des résultats de tests réalisés dans les instructions précédentes et les éventuels débordements de capacité observés. La solution présentée ici est beaucoup plus intuitive et consiste à tester le contenu du registre A. On peut ainsi créer une instruction :

BRZ paramètre, qui effectue un branchement à l'adresse indiquée si A vaut 0, et incrémente le CO dans les autres cas.

Si A = 0, paramètre ⇒ CO

Sinon CO + 1 ⇒ CO

Les micro-instructions correspondent à très peu près à celles du BRA, sinon que la valeur du bit CO + 1 doit être modifiée en fonction du test de A à 0.

Pour cette instruction seulement, si $A = 0$, CO + 1 = 0, en revanche, si $A \neq 0$, CO + 1 = 1.

On peut naturellement créer des instructions BRN branchement si A est négatif et même, pourquoi pas, BRP branchement si A est strictement positif.

Extension des modes d'adressage

Les modes d'adressage présentés jusqu'ici, le mode immédiat et le mode direct, ne permettent pas au programme de calculer l'adresse d'une variable ou d'un branchement. Pour écrire, par exemple, un programme qui effectue la somme des éléments d'un tableau, il est nécessaire d'accéder à des cases mémoire dont l'adresse est calculée par le programme. Deux types d'adressage sont possibles pour cela : l'adressage *indirect* et l'adressage *indexé*.

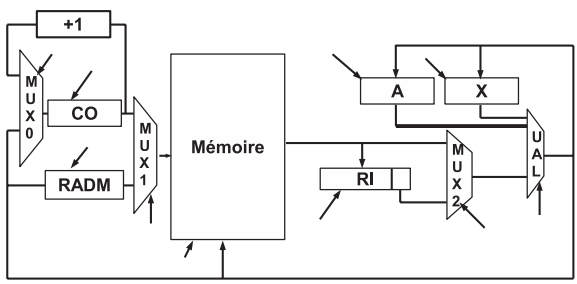
Dans le cas de l'adressage indirect, l'adresse réelle est calculée et mise dans une case mémoire d'adresse connue, c'est donc un pointeur. L'adressage indirect se note avec des crochets [ ]. Détaillons deux exemples dans lesquels on donne un nom à la valeur du paramètre.

LDA [pointeur] met dans A la valeur dont l'adresse est contenue dans la case mémoire pointeur.

BRA [retour] branche à l'adresse contenue dans la case mémoire réservée à l'adresse retour.

Dans le cas du LDA, cet adressage effectue deux appels mémoire successifs, ce qui peut ralentir l'exécution du programme.

Dans le cas de l'adressage indexé, qui est souvent préféré, une opération d'indirection est réalisée non plus avec une case mémoire, mais avec un registre accumulateur supplémentaire : le registre X, ou registre d'index.



Le chemin de données avec registre d'index.

Il est nécessaire pour cela d'ajouter ce registre au chemin de données ci-avant, et d'ajouter aussi des instructions complémentaires pour charger et sau-

vegarder ce registre – LDX en modes direct et immédiat, STX en mode direct –, ainsi que des instructions pour modifier X – ADDX direct et immédiat, par exemple.

Cela nécessite naturellement d'autre part une modification de l'unité arithmétique et logique UAL qui accepte maintenant 3 entrées, et dont les fonctions vont être A, X, B, A + B, X + B et, pour la soustraction, A – B et X – B.

Une instruction utilisant l'adressage indexé aura encore deux champs, l'instruction et le paramètre. Le paramètre pourra être utilisé pour ajouter un déplacement connu et constant au contenu du registre X. LDA paramètre, X réalisera alors comme fonction principale Mem([X] + paramètre) ⇒ A. Souvent, le paramètre vaut 0, c'est directement X qui donne l'adresse de la donnée à charger ; les cas où le paramètre est différent de 0 peuvent correspondre à des données structurées lorsque le champ visé n'est pas le premier.

Introduction au langage d'assemblage

Le langage d'assemblage est un langage plus intelligible que le langage machine, bien qu'il n'en diffère que par l'utilisation de mnémoniques, de symboles et d'étiquettes. Les programmes écrits dans ce langage sont traduits, quasiment ligne à ligne, en langage machine, par un outil appelé un *assembleur*.

Structure d'une ligne de code, instructions et directives, commentaires

Un programme en langage d'assemblage comporte des lignes très structurées constituées des éléments suivants.

Un premier champ qui commence en début de ligne. Il est appelé étiquette et représente l'adresse symbolique de la ligne. La ligne en cours définit cette étiquette si le champ n'est pas blanc. Ce champ est un symbole – commençant par une lettre – qui ne peut être défini au plus qu'une fois dans un module de programme donné.

Le deuxième champ comporte une instruction ou une directive d'assemblage. S'il n'est pas reconnu, il doit envoyer à l'utilisateur une erreur d'assemblage. S'il s'agit d'une instruction, elle pourra être exécutée – si elle a un sens – par la machine. Ce champ peut aussi être une directive d'assemblage. Auquel cas, la ligne sera gérée non pas par la machine mais par l'assembleur lui-même. Quelques directives fondamentales ont été choisies ici : ORG (*Origin*) définit l'adresse en mémoire à laquelle on veut implanter un programme ou des données ; RMW (*Reserve Memory Word*) réserve un certain nombre de mots en mémoire ; FCW (*Form Constant Word*) réserve et initialise un mot mémoire ; EQU (*Equate*) attribue une valeur à un symbole ; END (fin du texte source) définit la fin du texte source écrit en langage d'assemblage.

Le troisième champ est destiné au paramètre de l'instruction, ou de la directive, et au mode d'adressage. Ce champ peut représenter une adresse de donnée, dans le cas de l'adressage direct, une valeur constante, donnée en décimal ou en hexadécimal si elle est précédée par un $, le nom d'un symbole, qui sera remplacé par sa valeur dans la table des symboles, ou * qui représente alors le compteur d'implantation, c'est-à-dire l'adresse mémoire à laquelle le programme d'assemblage est en train d'écrire la valeur d'une instruction. Il est possible d'ajouter ou de soustraire des termes de ce type, avec la syntaxe : TERMEopTERME où op représente + ou –, sans espaces ni parenthèses, par exemple : Nom EQU*-TAB.

Le quatrième champ, optionnel, est réservé aux commentaires, qui commencent en général par le caractère *.

Par habitude, le mode d'adressage qui devrait se trouver lié à l'instruction, deuxième champ, est en réalité mis dans le troisième champ.

Les instructions du langage d'assemblage et directives de l'assembleur dépendent de la machine et de l'assembleur. Pour ce chapitre, les instructions et directives, qui sont celles de la Machine 2 présentée dans le livre *Du transistor à l'ordinateur* de Claude Timsit, et qui sont cohérentes avec les outils fournis dans `http://www.e-campus.uvsq.fr/dutransistoralordinateur/`, sont les suivantes

Instructions	Code	Exemple	Action
LDA immédiat	0	LDA #Param	Param=>A, CO++
LDA direct	1	LDA Param	Mem(Param)=>A, CO++
LDA indexé	9	LDA Param,X	Mem([X]+Param)=>A,CO++
STA direct	3	STA Param	[A]=>Mem(Param),CO++
STA indexé	10	STA Param,X	[A]=>Mem([X]+Param),CO++
LDX immédiat	7	LDX #Param	Param=>X, CO++
LDX direct	8	LDX Param	Mem(Param)=>X, CO++
ADD immédiat	4	ADD #Param	[A]+Param=>A, CO++
ADD direct	6	ADD Param	[A]+Mem(Param)=>A, CO++
ADD indexé	15	ADD Param,X	[A]+Mem([X]+Param)=>A,CO++
ADDX immédiat	12	ADDX #Param	[X]+Param=>X, CO++
ADDX direct	19	ADDX Param	[X]+Mem(Param)=>X, CO++
SUB immédiat	13	SUB #Param	[A]-Param=>A, CO++
SUB direct	14	SUB Param	[A]-Mem(Param)=>A, CO++
SUB indexé	16	SUB Param,X	[A]-Mem([X]+Param)=>A,CO++
SUBX immédiat	17	SUBX #Param	[X]-Param=>X, CO++
SUBX direct	20	SUBX Param	[X]-Mem(Param)=>X, CO++
STX direct	18	STX Param	[X]=>Mem(Param),CO++
BRA direct	2	BRA Param	Param=>CO
BRA indexé	11	BRA Param,X	[X]-Param=>CO
BRA indirect	24	BRA [Param]	Mem(Param)=>CO
BRZ direct	5	BRZ Param	SiA==0 Param=>CO, sinon CO++
BAL direct	21	BAL Param	Param=>CO, CO++=>SVCO
STSV direct	22	STSV Param	[SVCO]=>Mem(Param),CO++
STSV indexé	24	STSV Param,X	[SVCO]=>Mem([X]+Param),CO++

Directive		Action	
ORG	définit une origine		
exemple	debut	ORG	$100
définit une étiquette exemple et l'associe à l'adresse hexa 100			
RMW	réserver un ou plusieurs mots mémoire		
exemple	T	RMW	10
Réserve 10 mots en mémoire à partir de l'adresse actuelle			
FCW	réserve et initialise un mot mémoire		
exemple	N	FCW	5
Réserve une case mémoire et l'initialise à 5			
END	fin d'assemblage		
Ce n'est que la fin du texte source. Le paramètre doit être 0.			
A ne pas confondre avec la dernière instruction exécutable			
EQU	définit un symbole (#define)		
exemples	TOTO	EQU	5
	TOTO ne réserve pas de mémoire mais		
	sera remplacé par la valeur 5 à chaque apparition		
	TOTO	EQU	*
	dans les calculs d'adresses * a un sens particulier		
	(compteur d'implantation		

Instructions et directives reconnues par l'assembleur M2 de « Du transistor à l'ordinateur »

On retrouve différents types d'instruction, communs à la plupart des langages machine : des instructions de transfert de données – LDA, STA, LDX, STX, STSV –, des instructions de calcul – ADD, SUB, ADDX, SUBX – et des instructions de rupture de séquence – BRA, BRZ, BAL.

Supposons que nous voulions faire la somme des N premières valeurs du tableau d'entiers T, considéré comme déjà chargé. Le code de ce programme peut s'écrire de façon symbolique, en langage d'assemblage, comme suit :

```
DONNEES ORG $100 *Origine des données a l'adresse hexadécimale $100
T     RMW  10 *Réservation de 10 mots (10 en décimal)
N     EQU  3 *N est une constante valant 3
I     RMW  1 *I est par exemple l'indice de boucle
SOMME RMW 1 *SOMME est le résultat attendu
      * On considère que quelqu'un a chargé le tableau T.
PROG  ORG $0 *Adresse de début d'exécution
      LDA #0 * for (I=0;I<N;I=I+1) // on commence par mettre I
à 0
      STA I * I=0
      LDA #0
      STA SOMME * SOMME=0  //et SOMME à 0
      LDX #T *  T est l'adresse du tableau
TEST  LDA I * Boucle  //Puis on effectue le test de boucle
      SUB #N * comparaison de I à N
      BRN SUITE
      BRA FIN
SUITE LDA SOMME *    //et le calcul du corps de boucle
      ADD 0,X * { SOMME=SOMME+T[I];}
      STA SOMME
      ADDX #1 * //incrémentation de l'indice de boucle et de
l'adresse
      LDA I
      ADD #1
      STA I
      BRA TEST * Retour en début de boucle
FIN   ...................
```

Dans ce programme, T est une constante représentant l'adresse du tableau. Si l'on réalise LDA T, on met donc, dans A, le contenu du premier élément de tableau (Mem(T) ⇒ A). L'instruction LDX #T effectue, en revanche, le transfert de la constante, soit ici l'adresse du tableau dans X (Adresse T ⇒ X), ce qui permet de réaliser simplement des opérations indexées : X peut être incrémenté (ADDX #1) et LDA 0,X réalise alors Mem(0 + contenu de X) ⇒ A.

On peut d'autre part remarquer que LDX #T suivi de LDA 0,X est équivalent à LDA T. En revanche, dans ce cas, aucun calcul ne peut être fait sur T.

Une autre façon de faire consiste à réaliser une indirection par rapport à la mémoire : LDA #T; STA pointeur; LDA[pointeur]. Pour des raisons de performance, la solution utilisant l'adressage indexé est très souvent préférée, car elle économise un appel mémoire.

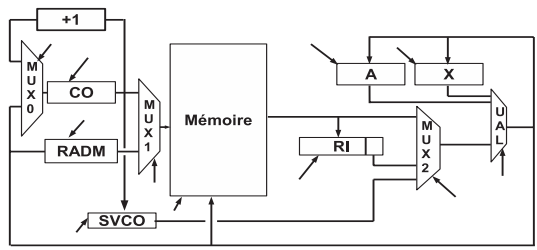
La programmation en langage d'assemblage est utile au programmeur, car elle permet d'utiliser des symboles de mnémoniques d'instructions et d'étiquettes, mais le programme ne peut être interprété tel qu'il est par le matériel qui s'attend à des instructions codées en valeurs binaires. Le paragraphe « Le fonctionnement d'un assembleur » montre le fonctionnement d'un programme appelé « assembleur » dont le rôle est, après quelques vérifications de syntaxe et d'adresses mémoire, de traduire ce programme en langage machine exécutable par l'ordinateur.

Appel de sous-programmes

Un sous-programme, ou fonction, peut être appelé de plusieurs endroits différents par un programme. L'appel du sous-programme est un simple branchement, mais, pour revenir au bon endroit dans le programme appelant, il est nécessaire de conserver l'adresse à laquelle ce retour doit s'effectuer. C'est ce que fait l'instruction *Branch and Link* (BAL).

Pour mettre en évidence toutes les opérations nécessaires dans un appel de sous-programme, nous avons fait le choix de sauvegarder l'adresse de retour dans un registre spécialisé, appelé SVCO – sauvegarde du compteur ordinal CO –, qui doit être ensuite entièrement géré par logiciel : ce registre est transféré en mémoire, par une instruction STSV.

Cela conduit à une légère modification du chemin de données, du fait de l'introduction du registre SVCO, avec son signal de validation d'horloge HSVCO et d'une voie supplémentaire sur MUX2, avec l'extension de SELMUX2 à 2 bits.



Le chemin de données permettant le BAL.

L'exécution de l'instruction BAL paramètre réalise donc : CO + 1 ⇒ SVCO ; paramètre ⇒ CO et l'exécution de STSV paramètre : SVCO ⇒ M(paramètre) ;

CO + 1 ⇒ CO. D'autres modes d'adressage sont naturellement concevables pour ces deux instructions.

Un exemple de fonction simple: la fonction suivante réalise le calcul d'une multiplication par trois. On charge A avec une valeur, puis on appelle cette fonction et on récupère, à l'issue de son exécution, le résultat dans A.

```
Donnees ORG 2 *Origine des données
Resultat RMW 1
RET   RMW 1 *Adresse de retour
Tmp   RMW 1 *Temporaire pour le calcul

Prog ORG 12 *Origine du programme
.....
      LDA #5 *chargement de A
      BAL FOISTROIS *Appel de fonction
      STA Resultat *Utilisation du résultat
.....
Fin   BRA Fin

FOISTROIS STSV RET *Sauvegarde de SVCO
      STA Tmp *Calcul
      ADD Tmp
      ADD Tmp * A vaut maintenant trois fois sa valeur initiale
      BRA [RET] *Retour à l'adresse contenue dans RET, résultat dans A
```

Dans cet exemple, un seul paramètre d'entrée a été utilisé. On peut alors le mettre dans A. Le résultat de la fonction est renvoyé dans A. Si plusieurs paramètres d'entrée étaient nécessaires, on les aurait rangés dans une pile logicielle avant l'appel de la fonction. De même, l'appel à la fonction FOISTROIS étant unique, le registre SVCO a suffi à conserver l'adresse de retour. Dans des programmes plus complexes, possédant plusieurs appels de fonctions imbriqués, en particulier pour les programmes récursifs, le registre SVCO doit être sauvegardé. Une solution habituelle consiste à le mettre dans une pile logicielle.

On remarque la dernière instruction du programme principal Fin BRA Fin dont le seul rôle est d'empêcher le programme de continuer son exécution. L'horloge continuant toujours à fonctionner, il faut en effet limiter strictement l'exécution en « bloquant » le compteur ordinal.

La fonction FOISTROIS peut être appelée de n'importe où et même plusieurs fois dans le programme principal. C'est pourquoi le BAL sauvegarde

l'adresse de retour. Celle-ci est mise dans un pointeur appelé ici RET. Pour revenir à l'adresse contenue dans ce pointeur, il est donc nécessaire de réaliser l'adressage indirect BRA[RET].

Réalisation et utilisation d'une pile logicielle

La machine définie ci-dessus permet de traiter de très nombreux problèmes, en particulier d'implanter très simplement une pile logicielle. Rappelons qu'une pile est une zone mémoire de taille fixe dans laquelle il est possible d'empiler une valeur, si la pile n'est pas pleine, ou de dépiler une valeur, si la pile n'est pas vide. L'analogie avec une pile d'assiettes dans une mémoire est immédiate. Une variable de type pointeur, appelée « sommet de pile », permet très simplement de réaliser ces opérations. Si l'on simplifie le problème en supprimant les tests de débordement, ces fonctions sont très facilement réalisées.

```
N     EQU 10 *taille de pile
Pile ORG *
      RMW N *réservation mémoire
Spile RMW 1 *pointeur de sommet de pile

PROG EQU * *Le programme commence ici
……
INIT EQU * *INIT est implanté ici
      LDA #Pile
      STA Spile

EmpileA EQU * *empiler A
      LDX Spile
      STA 0,X
      ADDX #1
      STX Spile

DepileA EQU * *dépiler A
      LDX Spile
      SUBX #1
      STX Spile
      LDA 0,X
…
```

Entrées/sorties simples

Un ordinateur sans entrée ni sortie n'a aucun intérêt, car ce n'est qu'un système fermé qui consomme de l'énergie ! Pour que ce système effectue des opérations utiles, il faut le doter de périphériques. Ceux-ci sont généralement répartis entre périphériques d'entrée – par exemple, clavier, souris, scanner, etc. – et périphériques de sortie – écrans, voyants lumineux, imprimantes. Deux grands modes de fonctionnement permettent de gérer les périphériques : le mode *programmé*, dans lequel un transfert entre le processeur et le périphérique est à l'initiative du processeur, et un mode plus complexe, *par interruption*, dans lequel le périphérique envoie un signal au processeur pour signaler une opération d'entrée/sortie. Pour l'analyse du processus d'interruption, nous renvoyons aux livres *Du transistor à l'ordinateur*, cité ci-avant, et *Systèmes d'exploitation* d'Andrew Tanenbaum.

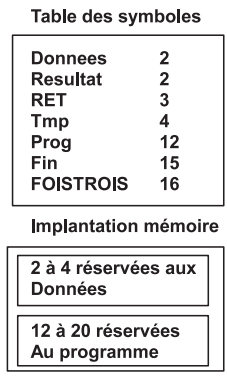
Dans le mode programmé, le processeur lit et écrit directement dans des « registres » d'entrées/sorties. Les deux méthodes habituelles utilisées pour cela sont soit de prévoir des registres supplémentaires, dits registres d'entrées/sorties, et des instructions supplémentaires servant à effectuer des transferts entre la mémoire et ces registres d'entrées/sorties, soit de donner à ces registres une adresse dans l'espace mémoire, les instructions servant à effectuer ces communications étant alors des LDA et des STA classiques. Ces deux méthodes sont employées, par exemple par les processeurs Intel pour la première et par les processeurs Motorola pour la seconde. La première conduit à un ensemble d'instructions supplémentaires. La seconde demande de neutraliser quelques cases mémoire pour les remplacer par des registres, mais la programmation en devient beaucoup plus simple. Les simulateurs Machine 3 et Machine 4 proposés dans *Du transistor à l'ordinateur*, utilisent cette seconde méthode.

Si, par exemple, l'affichage décimal est installé à l'adresse 100, un affichage décimal du contenu du registre A se fera simplement par STA 100. De même, si une entrée de valeur entière est assignée à l'adresse 101 – appelée clavier décimal dans les simulateurs Machine 3 et Machine 4 –, LDA 101 permettra de la lire.

Le fonctionnement d'un assembleur

Les rôles de l'assembleur sont de vérifier que le programme écrit en langage d'assemblage est correct et, si possible, de le traduire en langage machine interprétable directement par la machine. Il faut donc que l'assembleur crée une photographie de ce que sera la mémoire juste avant l'exécution du programme. Cette image de la mémoire, l'implantation mémoire, sera ensuite chargée dans le processeur, avec – si l'on désire tracer les variables en mémoire – une table

des symboles, qui associe aux étiquettes et autres symboles une valeur – cette valeur sera une adresse pour les étiquettes. Un assembleur fonctionne généralement en deux passes : lors de la première passe, on remplit au fur et à mesure la table des symboles et l'on essaye de créer l'implantation mémoire, ou, si l'on n'y arrive pas, tout au moins de réserver de la place dans cette implantation mémoire. Ce cas se produit, par exemple, si l'on effectue une référence en avant, en particulier quand on traduit une instruction BRA Fin, dans laquelle Fin est une étiquette située, dans le texte source, après l'instruction en cours de traduction. Ne connaissant pas l'adresse de l'instruction dont l'étiquette est Fin, il faut attendre la fin de la lecture totale du fichier source pour pouvoir terminer la traduction de l'instruction.



Les tables de l'assembleur dans l'exemple traité.

```
1 Donnees  ORG 2 *Origine des données
2 Resultat RMW 1
3 RET      RMW 1*Adresse de retour
4 TmpRMW 1 *Temporaire pour le calcul
5
6
7 Prog ORG 12 *Origine du programme
......
8     LDA #5 *chargement de A
9     BAL FOISTROIS *Appel du sous-programme FOISTROIS
10    STA Resultat *Utilisation du résultat
......
11 Fin     BRA Fin
```

```
12 FOISTROIS STSV RET *Sauvegarde de SVCO
13    STA Tmp *Calcul
14    ADD Tmp
15    ADD Tmp
16    BRA [RET] *Retour à l'adresse contenue dans RET
17    END  0
```

La première directive ORG, ligne 1, donne une valeur au compteur d'implantation, qui définit l'adresse à partir de laquelle il faut modifier l'implantation mémoire, ici c'est 2. Une case mémoire d'adresse 2 est donc réservée, et l'on attribue la valeur 2 au symbole `Resultat`. Ce 2 représente ici une adresse, la valeur de `Resultat` n'étant pas déterminée ici. Avant de faire cette opération, l'assembleur vérifie que l'étiquette `Resultat` n'a pas été déjà définie. Si c'était le cas, il y aurait un essai de double définition qui serait considéré comme une erreur. Le compteur d'implantation est incrémenté du nombre de cases désiré, ici 1, la mémoire est réservée et l'on passe aux lignes suivantes qui permettent de définir les emplacements de RET et de `Tmp` et d'effectuer la réservation nécessaire.

À la ligne 7, on assigne à la valeur `Prog` la valeur 12, mais, avant de continuer à implanter des données ou des instructions dans la mémoire, il faut vérifier que celle-ci est encore libre. À la ligne 8, on traduit l'instruction LDA #5 : c'est ici possible, le code de LDA# et la valeur 5 étant connus. La case mémoire n'étant pas déjà utilisée, elle est réservée et chargée à la bonne valeur – par exemple, pour Machine 2, à 0*1024 + 5. La ligne 9 représente un cas typique de référence en avant : si l'on sait traduire BAL, son paramètre fait appel à une étiquette TROISFOIS que la table des symboles ne connaît pas. Une case mémoire est cependant réservée pour le cas où, lors de la deuxième passe, celle-ci sera connue. En continuant ainsi jusqu'à la ligne 12, l'étiquette TROISFOIS va être définie, etc. La ligne 17 représente la fin du fichier source. Quand l'assembleur y arrive, il lance la seconde passe qui permet de remplir les cases de la mémoire d'implantation qui n'ont pas pu l'être auparavant – ici, celle qui correspond à la ligne 9 – en utilisant la table des symboles qui s'est remplie au cours de la première phase. Si aucune erreur de syntaxe, de non-définition ou de double définition d'étiquette, d'implantation, etc. n'est trouvée, le résultat de l'assemblage peut être transféré dans la mémoire de la machine destinée à exécuter le code – opération de chargement réalisée par un programme souvent nommé Chargeur ou *Loader* – pour être exécuté.

Rudiments de compilation manuelle

La compilation consiste à transformer un code écrit dans un langage de haut niveau, ici Java, en un code en langage d'assemblage. Nous montrons ici comment traduire un programme simple, à la main, afin de définir une méthodologie stricte, pour créer des programmes en langage d'assemblage. Les idées principales sont de conserver strictement le nom des variables et de respecter strictement les algorithmes écrits, en les traduisant directement ligne à ligne, sans chercher à optimiser. La machine présentée ici a volontairement un seul registre accumulateur et un seul registre d'index, qui sont considérés comme des temporaires. Toute variable doit donc être implantée en mémoire. Les directives d'assemblage permettent de réaliser la réservation de la mémoire, de définir des constantes, etc.

Un exemple de programme avec une boucle

Un exemple, valant un long discours, est présenté ici. Pour faciliter la lecture, les constantes et variables ne faisant pas partie explicitement du code source sont en majuscules. On donne ainsi l'adresse de l'afficheur décimal, les adresses correspondant aux données et au programme, mais aussi SAVX variable qui va permettre une utilisation simple de l'adresse de l'élément de tableau et de son incrémentation. Il y a de très nombreuses façons de traduire cette boucle en langage d'assemblage, celle présentée ici est simple et ne fait aucune optimisation, ce qui facilite la maintenance. L'utilisation de SAVX dans ce programme n'est pas nécessaire, car X n'est utilisé que comme pointeur sur l'élément de tableau en cours de traitement. Elle devient nécessaire quand on utilise plusieurs tableaux ou plusieurs boucles imbriquées. Dans ce cas, on créera autant de variables de sauvegarde de X que nécessaire. Dans le cas de programmes récursifs, il faudra utiliser la pile pour sauvegarder les adresses de retour des appels récursifs ainsi que la valeur des autres registres, voire les variables locales intermédiaires.

```
                                ECRAN EQU 100 *adr. de l'aff. décimal
                                DEBDATA EQU 2 *adr. d'impl. des données
                                DEBPROG EQU 12 *adr. d'impl. du programme
static final int N=5;           N EQU 5 * traitement du #define
static int i;                   Data ORG DEBDATA *origine des données
static int resultat;            i RMW 1 *réservation des variables
static int [] Tab=new int[N]; resultat RMW 1
                                Tab RMW N *reservation du tableau
                                SAVX RMW 1 *sauvegarde de l'index
```

```
public static void main(String[] args){
 for (i=0;i<N;i=i+1) {
   Tab[i]=i;
   resultat=resultat+i;
 }
 System.out.println(resultat);
}
```

```
Prog ORG DEBPROG *début du programme
LDA #0 *initialisation de la boucle
STA i
LDX #Tab *adresse du tableau
STX SAVX *pointeur courant ds le tableau
TEST LDA #N *test de i
SUB i
BRZ FINBCL * traitement de i<N
BRN FINBCL
SUITEBCL LDX SAVX
LDA i
STA 0,X *Tab[i]=i;
LDA resultat
ADD i
STA resultat *resultat=resultat+i

LDA i *modifications indice et index
ADD #1
STA i
LDX SAVX
ADD #1
STA SAVX
BRA  TEST *branchement au test
FINBCL LDA resultat
STA ECRAN *impression
..
```

Exercices corrigés et commentés

Circuits combinatoires

Petits circuits classiques

Exercice 1

Donner les tables de vérité des circuits suivants :

Majorité à 3 entrées, une sortie : S = *Majorité*(E_2, E_1, E_0) le bit S est le gagnant du vote à la majorité.

Encodeur à 3 entrées, 2 sorties : la sortie donne l'indice de la première entrée à 1, ou 3 si aucune entrée n n'est à 1. Discuter du cas où l'on peut garantir qu'il y a toujours une et une seule entrée à 1.

Décodeur à 3 entrées. Le circuit a 8 sorties dont une seule est à 1 (vrai), celle dont l'indice correspond au nombre binaire donné en entrée sur trois bits. Autrement dit : $S_i = ((E_2E_1E_0)_{\text{base 2}} = i)$.

Correction

E2	E1	E0	Ma j(E2,E1,E0)	Enc(E2,E1,E0)	Dec(E2,E1,E0)
0	0	0	0	11	00000001
0	0	1	0	00	00000010
0	1	0	0	01	00000100
0	1	1	1	01	00001000
1	0	0	0	10	00010000
1	0	1	1	10	00100000
1	1	0	1	10	01000000
1	1	1	1	10	10000000

Pour l'encodeur, lorsqu'il est garanti qu'une et une seule entrée est à 1, la première ligne peut être quelconque, ce qui libère un code en sortie. Les lignes où apparaissent deux 1 peuvent aussi avoir une valeur quelconque. Pour la réalisation d'un circuit, cette hypothèse supplémentaire permet, en outre, de simplifier les formules réalisant l'encodage.

Exercice 2. Unité arithmétique et logique

1. Rappeler le circuit d'un additionneur 1 bit, et de l'additionneur *n* bits.

2. À partir du circuit précédent, construire un circuit faisant des soustractions sur *n* bits.

3. Donner le dessin d'une unité arithmétique et logique pouvant faire les 4 opérations suivantes : Addition/Soustraction/Et (bit à bit)/ Opérande Gauche, au choix selon une commande (F1 F0).

Correction (indications)

1. Se reporter au paragraphe sur les circuits booléens pour retrouver les formules pour l'additionneur 1 bit. Le circuit d'addition *n* bits s'obtient en cascadant la cellule d'addition 1 bit *n* fois, le *Cin* de la première cellule – pour le bit de poids faible – valant 0, le *Cout* de

cette cellule étant relié au *Cin* de la cellule suivante, et ainsi jusqu'à la dernière cellule, pour laquelle *Cout* sert de retenue globale de l'additionneur.

2. Comme $A - B = A + B + 1$, pour obtenir un soustracteur, il suffit de prendre un additionneur avec en entrée *A* et *B*. Pour le +1, la retenue entrante de la première cellule d'addition peut être positionnée à 1 au lieu de 0. C'est tout.

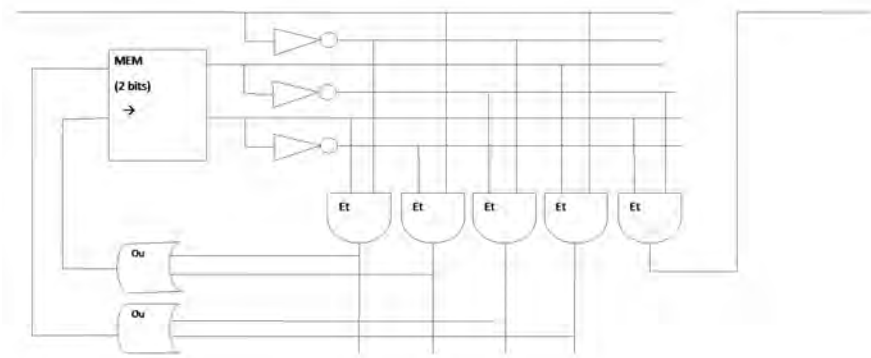
3. À partir de ce qui précède, pour réaliser une UAL faisant 4 opérations, on peut utiliser un additionneur *n* bits, un ET *n* bits, et deux multiplexeurs – un premier : 2 vers 1 sur *n* bits pour les entrées et un second 3 vers 1 sur *n* bits pour les sorties –: dans un premier étage du circuit, un multiplexeur 2 vers 1 sur *n* bits avec en entrée *B* et *B* sélectionne la valeur nécessaire à la réalisation du calcul demandé – si c'est une addition *B*, si c'est une soustraction *B*, sinon peu importe. Dans un second étage du circuit, le ET bit à bit de *A* et de *B* est effectué en parallèle de l'addition de *A* et de la sortie du multiplexeur précédent – avec en *Cin* = 0 si c'est une addition et *Cout* = 1 si c'est une soustraction. Dans un troisième étage du circuit, le second multiplexeur 3 vers 1, avec en entrée *B*, le résultat du ET précédent et de l'addition précédente, sélectionne en fonction de l'opération demandée le bon résultat.

Logique séquentielle

Circuit d'un petit automate

Exercice 3

1. En supposant que le circuit suivant



Circuit d'un petit automate.

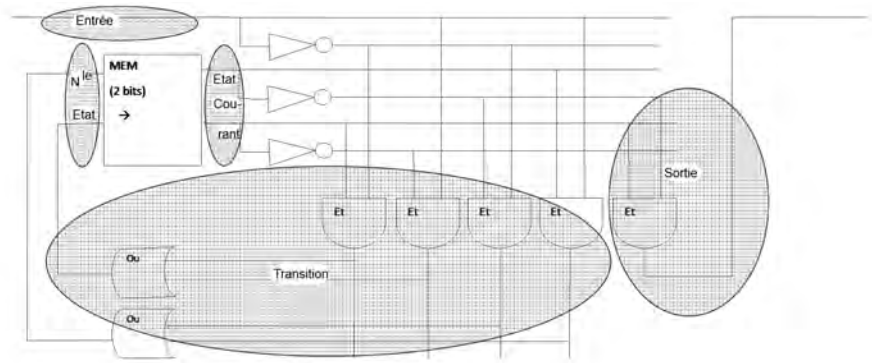
représente un automate, identifier sur le schéma les différents éléments : état courant, nouvel état, entrée, sortie, circuit de sortie et circuit de transition.

2. Donner les tables de vérité des deux circuits (sortie, transition).

3. Dessiner l'automate correspondant.

Correction

1.

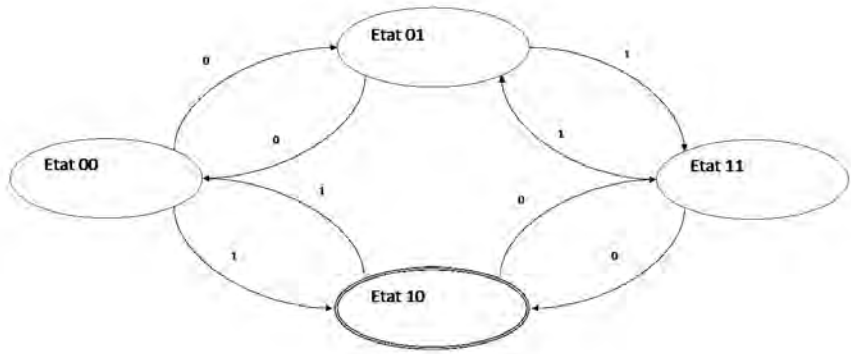


Circuit d'un petit automate

2. En nommant les fils correspondant à l'état courant de haut en bas ET0, ET1 – resp. NE0, NE1 pour le nouvel état –, les tables de vérité des circuits de transition et de sortie sont données par le tableau

ET1	ET0	Entrée	NE1	NE0	Sortie
0	0	0	0	1	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	0	0	1
1	1	0	1	0	0
1	1	1	0	1	0

3. L'automate défini a quatre états : 00 (E1=0, E0=0), 01 (E1=0, E0=1), 10 (E1=1, E0=0), 11 (E1=1, E0=1). L'état 10 est final (sortie à 1). Les transitions sont les suivantes



Modes d'adressage

Exercice 4

Le tableau ci-dessous représente l'état de certaines cases mémoire et des registres A et X avant et après les instructions données sur la colonne de gauche. Modifier les cases du tableau au fur et à mesure de l'exécution du programme en expliquant les modifications :

	X 3	A 5	M(1) 1	M(2) 4	M(3) 8	M(4) 3	M(5) 1	M(6) 2	M(8) 0
LDA3	–	–	–	–	–	–	–	–	–
STA2,X	–	–	–	–	–	–	–	–	–
LDA#2	–	–	–	–	–	–	–	–	–
ADD3	–	–	–	–	–	–	–	–	–
STA4	–	–	–	–	–	–	–	–	–
SUB1,X	–	–	–	–	–	–	–	–	–
LDX[6]	–	–	–	–	–	–	–	–	–
STX[3]	–	–	–	–	–	–	–	–	–

Correction

	X	A	M(1)	M(2)	M(3)	M(4)	M(5)	M(6)	M(8)
	3	5	1	4	8	3	1	2	0
LDA3	–	– 8	–	–	–	–	–	–	–
STA2,X	–	–	–	–	–	–	– 8	–	–
LDA#2	–	– 2	–	–	–	–	–	–	–
ADD3	–	– 10	–	–	–	–	–	–	–
STA4	–	–	–	–	–	– 10	–	–	–
SUB1,X	–	– 0	–	–	–	–	–	–	–
LDX[6]	– 4	–	–	–	–	–	–	–	–
STX[3]	–	–	–	–	–	–	–	–	– 4

Ligne 1 A= M(3)=8
Ligne 2 M(2+[X])= M(5)= A= 8
Ligne 3 A= 2
Ligne 4 A =A+ M(3)= 10
Ligne 5 M(4)=A=10
Ligne 6 A=A- M(1+[X])=A- M(1+3)= 10-10=0
Ligne 7 X=M(M(6))= M(2)=4
Ligne 8 M(M(3))= M(8)=4

Un exemple complet de fonction

Exercice 5. Traduire en langage d'assemblage le programme suivant qui réalise la multiplication non signée de deux mots de 10 bits. Contraintes : on considérera que les paramètres d'entrée de la fonction sont mis dans la pile, et que le résultat de la fonction est remis en A.

```
3 /************************************
4 * Multiplication non signée 10bits *
5 ************************************/
6 public static int mul(unsigned int Na,unsigned int Nb) {
7 int i;
8 intj = 1;
9 int Nc = 0; //résultat
10
11  for (i = 0; i < 10; i = i + 1) {
```

```
12   if ((Nb & j) != 0) {Nc = Nc + Na;}// Addition
13   Na = Na + Na;   //Décalage de Na à gauche
14   j = j + j; //Calcul du masque de Nb suivant
15  }
16  return (Nc);
17 }

18 public static void main (String [] args) { //test de mul
19 int a = 5;
20 int b = 6;
21 int c;
22      c = mul(a,b);
23      System.out.println(c); // à simuler par un STA 100
24 }
```

On considérera que les instructions AND, OR et XOR n'existent qu'en mode direct. Elles réalisent l'opération bit à bit entre les 10 bits de poids faibles du mot lu et les 10 bits de poids faibles de A, avec un résultat dans A.

Pour tester les différents bits de Nb, il suffit de créer un masque égal initialement à 1, puis, à chaque itération, de doubler le masque, lignes 12 et 14. On réalisera donc *test* = *masque* AND B suivi d'un branchement conditionnel, ligne 12.

La fonction de décalage n'existant pas à proprement parler dans la machine simulée, la méthode consiste, pour réaliser le décalage à gauche, à ajouter la valeur à décaler à elle-même, lignes 13 et 14.

Correction

```
*************************************************************
* Programme en langage d'assemblage            *
* de test de multiplication compatible Machine 3    *
* Les deux valeurs sont empilées par le programme appelant *
* Celui-ci appelle la fonction MUL par un BAL  *
* Le résultat se retrouve dans le registre A au retour de fonction *
* Les numéros de ligne correspondent au texte source     *
*************************************************************
*DONNEESPP ORG 2
a RMW 1
b RMW 1
c RMW 1
SPILE RMW 1 *pointeur de sommet de pile
PILE RMW 2 *une toute petite pile
```

```
PROG ORG 15
MAIN EQU * *Début du main
     LDA #5
     STA a */ligne19
     LDX #PILE *mise de l'adresse de la pile en SPILE
     STX SPILE
     STA 0,X *a est mis en pile (ligne 22)
     LDA #6
     STA b */ligne20
     STA 1,X *b est mis en pile (ligne 22)
     ADDX #2    *Déplacement du sommet de pile
     STX SPILE

     BAL mul *Appel de mul(ligne 22)

     STA c *rangement du résultat (ligne 22)
     STA ECRAND *simulation du printf
Fin BRA Fin     *arret du programme par blocage (ligne24)

DONNEESSP EQU * *on considère qu'elles sont dans le tas.
*On aurait pu les mettre dans la pile. Simplifions.
i     RMW 1
j     RMW 1
Na   RMW 1
Nb   RMW 1
Nc   RMW 1
RET   RMW 1 *relais d'adresse de retour
SAVX RMW 1 *pour sauvegarde du registre X (A est la valeur de retour)

mul     STX SAVX
     STSV RET *Tout est sauvegardé, on peut y aller

     LDX SPILE
     SUB #2 *Pour avoir un accès simple aux valeurs empilées
     LDA 0,X *Passage d'arguments a dans NA puis b dans Nb
     STA Na
     LDA 1,X
     STA Nb
     STX SPILE *On ne touche plus à la pile
```

```
      LDA #1 */ligne 8
      STA j
      LDA #0 */ligne 9
      STA Nc
Initbcl LDA #0 * bien qu'inutile, respect méthodologie de traduction
      STA i */ligne 11
TEST  LDA i
      SUB #10
      BRN SUITE *méthode sûre de traduction. D'autres sont possibles
      BRA FINBCLE
SUITELDA Nb
      AND j *on considère que cette instruction existe
      BRZ SUITE1 *aller en ligne 13
      LDA Nc
      ADD Na
      STA Nc *fin de la ligne 12
SUITE1 LDA      Na
      ADD Na
      STA Na *ligne 13
      LDA j
      ADD j
      STA j *ligne 14

INCI    LDA i *incrémentation de i et rebouclage
      ADD #1
      STA i
      BRA TEST

FINBCLE LDA Nc *Régénération des registres et retour appelant
      LDX SAVX
      BRA [RET]
```

Choix d'algorithme : calcul de la puissance

Exercice 6. Deux algorithmes sont proposés pour calculer R : la puissance n-ème d'un nombre X ($R \leftarrow X^n$).

```
Algorithme 1 :
R<-1
Pour i allant de 1 à n faire :
```

```
R<-R*X

Algorithme 2:
R<-1
Tant que n != 0 faire:
Si n est impair :
R<-R*X;N<-N-1
X<-X^2
N<-N/2
```

1. Choix de l'algorithme à implémenter. Choisir entre les deux algorithmes celui que vous comptez implémenter, donnez vos raisons.

2. Implémentation sous la forme d'un circuit à flot de données. Donner un circuit type « flot de données » réalisant l'exécution du calcul.

3. Implémentation sous la forme d'un circuit PC/PO (Partie Contrôle/Partie Opérative). En explicitant vos hypothèses sur la partie opérative PO, donner le dessin de l'automate de la partie contrôle PC d'un circuit PC/PO réalisant le calcul.

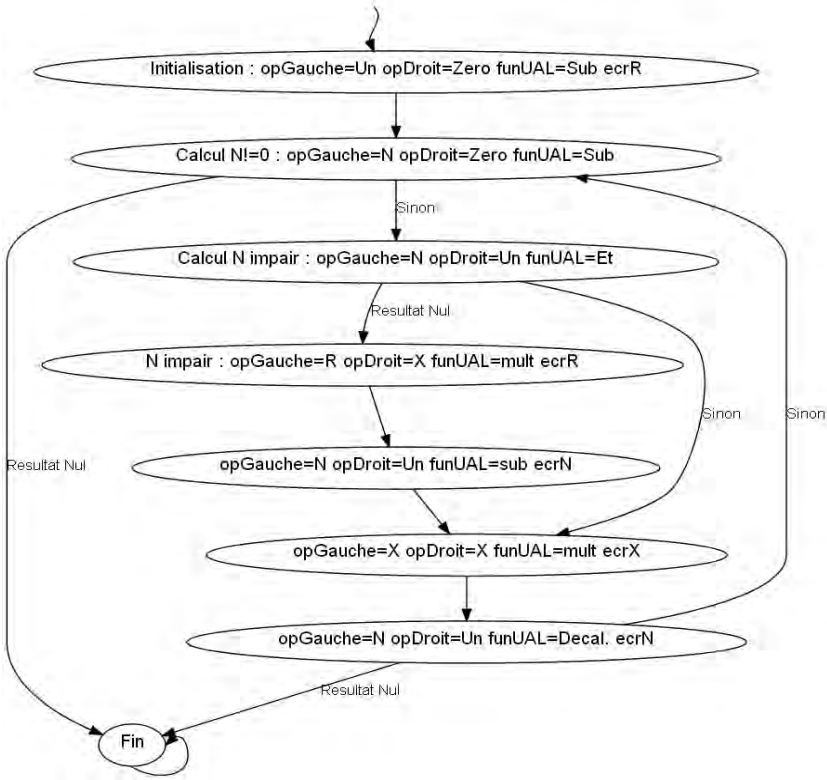
Correction (indications)

1. Pour comparer les deux algorithmes, observons leurs complexités. La complexité en place du premier algorithme sera liée à la complexité en place de la multiplication. Idem pour le second algorithme. Cependant, le second algorithme nécessite deux multiplications. La complexité en temps du premier algorithme sera proportionnelle à n et à la complexité en temps de la multiplication. Pour le second algorithme, la complexité en temps sera proportionnelle à $log(n)$ et à la complexité en temps de la multiplication. Pour la mise en œuvre, le premier algorithme semble plus simple.

2. Pour l'algorithme 2, on peut prendre le circuit donné au paragraphe « Blocs de logique séquentielle », avec comme état courant le triplet (N, R, X), comme entrée (N, 1, X), comme sortie (X, N==0) et comme fonction de transition (N, R, X) ← (si (N est pair) N-1/2 sinon N/2, si (N est pair) R*X sinon R, X/2).

3. On suppose une partie opérative PO possédant au moins 5 registres N, R, X, Zero et Un – Zero et Un sont des registres contenant les valeurs 0 et 1 –, une unité arithmétique et logique UAL permettant de faire des décalages à droite, des soustractions, des ET et des multiplications.

La partie contrôle PC est donnée par l'automate suivant.



Automate pour le calcul $R \leftarrow X^n$

Exercices non corrigés

Circuits logiques de base

Exercice 1

1. Dresser la table de vérité d'une porte OU, dresser la table de vérité du circuit NON (NON-OU(X,Y)) et comparer.

2. Dresser la table de vérité d'une porte ET, dresser la table de vérité du circuit NON-OU(NON(X),NON(Y)) et comparer.

Exercice 2

1. Concevoir un multiplexeur à 4 entrées de deux bits chacune. À partir d'un simulateur de portes, testez le circuit que vous avez conçu.

2. Réaliser un additionneur sur un bit, *A*, *B* et *Cin*, sur le simulateur proposé, dupliquez-le et testez-le sur des données de 3 bits.

Exercice 3

Réaliser et tester une mémoire vive de ce type sur simulateur à partir de bascules D et de portes.

Réalisation d'instructions de base

Exercice 4

Comment peut-on réaliser les instructions SUB #paramètre – soustraction immédiate – et SUB paramètre – soustraction en mode direct ?

Pile logicielle

Exercice 5

Ajouter au programme ci-avant les tests de débordement de pile pleine et pile vide.

Exercice 6

Le programme décrit au paragraphe « Introduction au langage d'assemblage » ne sauvegarde pas la valeur de X. Le modifier de telle sorte que cette valeur soit restituée après le retour de fonction.

Exercice 7

Créer et utiliser une fonction P1 PLUS P2 qui considère que P1 et P2 sont mis en pile et que le résultat renvoyé dans A est la somme de ces valeurs. Il est préférable de sauvegarder X !

Entrées/sorties simples

Exercice 8

Réaliser et tester un programme qui lit au clavier une valeur, appelle la fonction TROISFOIS vue ci-avant et renvoie le résultat sur l'afficheur décimal, puis reboucle sur la lecture.

Compilation manuelle

Exercice 9

Compilation d'une conditionnelle

Traduire en langage d'assemblage, en faisant ressortir l'aspect systématique, le programme suivant de calcul de la valeur absolue d'un nombre. Le résultat de la fonction sera mis dans A.

```
public static void main(String[] args)
  {
     int N,Resultat;
       // lire N au clavier;
```

```
        if (N>0) {Resultat=N;}
        else {Resultat=-N;}
        //Resultat est imprime;
}
```

Questions d'enseignement

Pour être accepté par les élèves, l'enseignement de l'architecture doit être le plus ludique possible. Cela nécessite, à notre sens, de fournir des simulateurs à tous les niveaux, de telle sorte que l'élève puisse s'affronter à la machine de simulation qui est, de ce fait, la meilleure alliée de l'enseignant.

Tout d'abord, si on en a le temps, il est utile de bien comprendre comment fonctionne un transistor : ce n'est qu'un tuyau souple qui est pincé, donc fermé, quand la tension de grille est inférieure à un certain seuil et devient conducteur, donc passant, quand la tension de seuil est dépassée. Puis il est utile de réaliser une porte avec ce transistor – un inverseur, par exemple – en montrant avec la loi d'Ohm la tension de sortie en fonction de la tension d'entrée. Ces notions permettent de montrer qu'il doit y avoir une relation entre la tension d'alimentation, la résistance de charge et la tension de pincement du transistor. Les portes réalisées sont très simples mais ont le mérite de bien montrer les problèmes technologiques liés à la réalisation de toutes les portes logiques.

La notion de porte étant acquise, on peut utiliser un simulateur et donner des challenges aux élèves, qu'ils doivent réaliser et tester sur le simulateur.

Puis l'on passe à la simulation de circuits plus complexes possédant des bus à plusieurs bits. Dans ce cas, il faut se méfier des simulateurs qui ne permettent pas l'usage de bus, car le fait d'avoir à dupliquer et à organiser les dessins devient vite rébarbatif. La simulation d'un compteur, par exemple, est une étape importante, qui permet de découvrir un tout premier circuit séquentiel.

L'utilisation d'une machine de Von Neumann simplifiée simulée permet de se familiariser avec les modes d'adressage de base, ce qui se révèle souvent difficile, et d'appréhender le fonctionnement intime de l'ordinateur grâce à la visualisation des bus et des registres. Là encore, de très nombreux défis sont possibles : si la machine simulée présente des sorties en ASCII et des entrées clavier simulées, un défi est d'écrire un programme qui affiche le prénom de l'élève, ce qui est une occasion de revoir le code ASCII, et si le simulateur possède une sortie décimale, ou même s'il n'offre que la possibilité d'analyser une case de la mémoire simulée, un défi est d'afficher la somme des n premiers entiers, ce qui demande de réaliser un programme comprenant une boucle.

Des programmes plus savants, comme la multiplication proposée en exercice corrigé, permettront de bien sentir le lien entre algorithme et architecture.

Nous ne saurions trop insister sur la nécessité de l'aspect ludique de cet enseignement, qui a fait ses preuves et permet, à la condition de laisser le temps de compréhension et d'appréhension, « vous terminerez à la maison », de faire passer ces notions qui semblent un peu complexes au premier abord.

Compléments

Quelques extensions du modèle d'architecture

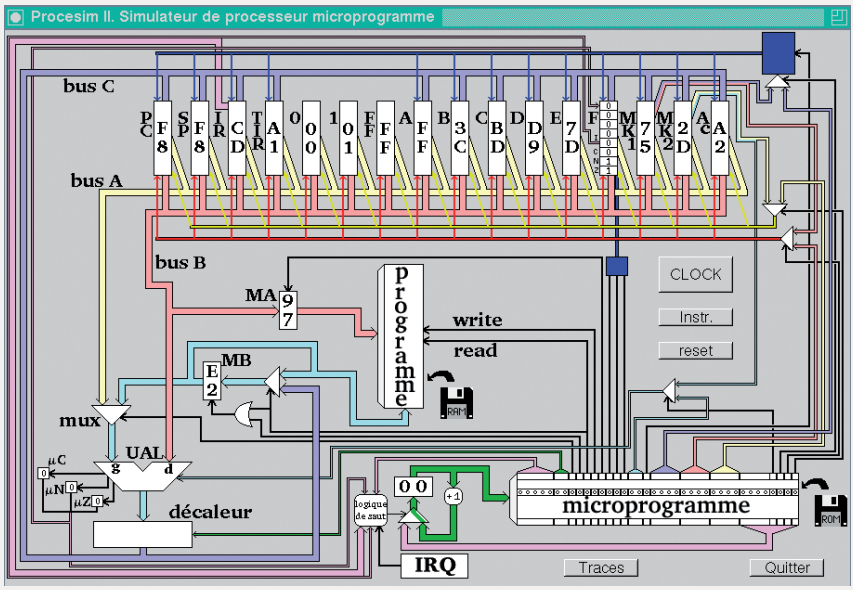
Pour gérer des entrées/sorties sans que le programme passe son temps à vérifier si elles ont été réalisées, on utilise un processus d'interruption qui, quand un signal physique arrive, réveille un « programme d'interruption ». La tâche est alors mise en sommeil pendant la durée du traitement de l'interruption. Cela nécessite naturellement une sauvegarde, au moins partielle, du contexte de la tâche interrompue, de telle sorte que celle-ci soit susceptible d'être reprise après la fin de l'interruption. Cela nécessite l'ajout de registres de sauvegarde du compteur ordinal CO spécifique aux interruptions, de même qu'un mécanisme permettant de masquer le signal d'interruption et de supprimer la demande d'interruption et les instructions spécifiques qui lui seront attachées. Une version de ce dispositif est présentée dans *Du transistor à l'ordinateur*, cité ci-avant, et les simulateurs associés à cette machine ainsi que l'assembleur se trouvent en `http://www.e-campus.uvsq.fr/dutransistoralordinateur/`.

Mis à part ce dispositif d'interruption, qui n'a pu être développé ici par manque de place, la machine décrite représente la base de toutes les machines réalisées à ce jour. Elle est capable de traiter toutes sortes de programmes. Cependant, pour obtenir des performances meilleures, il est possible d'ajouter un certain nombre d'améliorations :

– un nombre plus important de registres généraux de façon à permettre à l'utilisateur de conserver dans ses registres des données intéressantes sans être obligé d'effectuer des opérations incessantes de transferts de et vers la mémoire de données,

– un ou plusieurs registres de pointeurs de pile spécialisés, qui peuvent dans certains cas posséder des fonctions matérielles d'auto-incrémentation et d'auto-décrémentation.

Un exemple de simulateur de machine de ce type est le suivant.



Exemple de simulateur de machine RISC à banc de registres.

On y retrouve la séparation entre une partie opérative, avec un chemin de données comportant le banc de registres, la mémoire et l'UAL, et une partie contrôle, avec, dans le quart en bas à droite, le microprogramme commandant la partie opérative avec le numéro de la micro-instruction en cours d'exécution – 00 sur la copie d'écran. Une vidéo du fonctionnement de cette machine est présentée en `http://www.noe-kaleidoscope.org/public/people/DenisB/Enseignement/Architecture/Demo-Procesim.htm`. Même si ce fait est difficilement visible à la lecture des programmes en langage d'assemblage, les machines actuelles utilisent de plus une structure permettant un accès simultané à deux mémoires : une mémoire de programme et une mémoire de données, appelées *machines de Harvard*, ce qui divise par deux le nombre de cycles nécessaires pour un LDA par exemple. Cette transformation s'opère souvent de façon invisible à l'utilisateur grâce à l'utilisation d'un cache de données et d'un cache de programme qui permettent non seulement un accès simultané, mais aussi un temps d'accès simulé à la mémoire beaucoup plus court.

L'unité arithmétique et logique UAL décrite ici est extrêmement simple et ne réalise que des opérations d'addition, de soustraction et quelques fonc-

tions logiques, ET, OU, etc. Il est naturellement possible de l'étendre en ajoutant des opérations de décalage, mais aussi des opérations câblées de multiplication et de division, voire de calcul flottant. Pour plus d'information sur ce domaine, nous renvoyons au livre *Structured Computer Organization* d'Andrew Tanenbaum.

Enfin, les micro-instructions réalisées ici de façon séquentielle peuvent parfaitement être engendrées de façon câblée, et même être pipelinées, c'est-à-dire que plusieurs instructions peuvent se dérouler simultanément dans la machine : pendant qu'une instruction est en phase d'exécution, il y a de nombreux cas où l'instruction suivante peut être décodée par exemple. Le gain en temps est alors d'un facteur 2 ; en revanche, ce type de réalisation demande parfois un doublement d'une partie du chemin de données.

Pour aller encore plus vite : des machines parallèles

Les circuits d'un processeur ne peuvent aller plus vite que la vitesse permise par la technologie. Or, il est nécessaire pour certaines applications – comme le calcul scientifique ou le calcul embarqué, par exemple les applications radar – d'aller encore beaucoup plus vite, et donc de paralléliser les traitements : si l'on considère que, sur un chantier, un maçon monte un mur en 6 heures, quatre maçons peuvent soit aller quatre fois plus vite pour monter un mur, soit monter quatre murs dans le même temps. Naturellement, certains problèmes se laissent paralléliser mieux que d'autres.

De nombreux types d'architecture de machines utilisant cette idée ont été conçus. Cela nécessite que les processeurs travaillent sur une ou plusieurs tâches communes, partagent les données et le code, et surtout se synchronisent entre eux.

La complexité de telles machines dépasse de beaucoup le but de ce chapitre d'introduction, et la complexité des méthodes utilisées par les compilateurs est plus grande encore. Nous renvoyons le lecteur intéressé au livre *Computer Architecture : A Quantitative Approach*, de John L. Hennessy et David A. Patterson.

Qu'est-ce qu'un système d'exploitation ?

Un ordinateur tel que celui que nous avons décrit dans ce chapitre serait très difficile à utiliser s'il n'était pas équipé de programmes qui permettent

– d'exécuter « en même temps » plusieurs programmes, appartenant ou non à plusieurs utilisateurs,

– de gérer les entrées/sorties à un haut niveau, par exemple, non pas allumer un pixel de l'écran, mais écrire une lettre dans une fenêtre,

– d'organiser les informations présentes sur le disque en une arborescence de fichiers,

– d'utiliser une mémoire plus grande que la mémoire physique de l'ordinateur,

– d'utiliser un ensemble d'instructions plus grand que celui du processeur de l'ordinateur.

Ces différents programmes, qui présentent au programmeur une machine plus évoluée que la machine réelle, forment le *système d'exploitation* : Windows, Unix, Linux, MacOS…

La notion clé pour comprendre le fonctionnement d'un système d'exploitation est celle d'*interruption* : une interruption lance des tâches en fonction d'un signal externe ou interne à la machine.

Un système multitâches ou multi-utilisateurs

Les codes de plusieurs programmes ou tâches prêtes à être exécutées se trouvent dans la mémoire vive. Imaginons alors qu'un générateur externe au processeur envoie périodiquement des interruptions. À chaque interruption, le programme d'interruption examine, parmi l'ensemble des tâches à exécuter en attente du processeur, celle qui est prioritaire. Au lieu de revenir directement à la tâche interrompue, il range le contexte de celle-ci dans l'ensemble des contextes des tâches en attente. Cela permet donc à chaque interruption de choisir la tâche qui doit utiliser le processeur.

Tous les systèmes d'exploitation comme UNIX, Windows, etc. utilisent ce principe pour partager une ressource fondamentale, le processeur, entre les différentes tâches en cours d'exécution.

L'algorithme choisissant, entre les différentes tâches, celle qui est exécutée à un instant donné dépend de la priorité affectée à telle ou telle tâche et des critères d'optimisation choisis.

Avoir plusieurs tâches pour un même utilisateur n'est souvent utile que si elles peuvent communiquer entre elles ; ces communications supposent un partage des données, mais aussi des moyens de synchronisation. Par exemple, dans un programme de calculette, il peut y avoir une tâche dévolue à la lecture du clavier, une tâche de traitement et une tâche d'affichage qui fonctionnent simultanément. Tant que les données ne sont pas entrées au clavier, il n'est pas nécessaire de lancer la tâche de calcul qui se met donc en attente et ne consomme aucune ressource processeur pendant ce temps.

Les grosses machines peuvent être partagées entre plusieurs utilisateurs. Dans ce cas, le même principe s'applique aux différentes tâches de tous les utilisateurs simultanés. L'un des problèmes à résoudre est alors l'étanchéité

totale qu'il doit y avoir entre les données de deux utilisateurs simultanés. Ils doivent se sentir parfaitement indépendants les uns des autres et dans un environnement sûr – les autres utilisateurs n'apprécieraient pas que vous modifiiez leurs données sans leur consentement !

Apports du système au niveau des entrées/sorties

Nous avons vu qu'il est possible de lire et d'écrire dans des registres de périphériques. Cependant, si nous considérons un disque dur par exemple, celui-ci est constitué d'un système mécanique constitué du disque lui-même et d'un bras muni de têtes de lecture – assez analogue à celui d'un pick-up pour disques vinyles – auquel on donne l'adresse – position en `r` – d'un bloc de données à lire ou à écrire. Le temps de réponse d'un tel système est de l'ordre de la dizaine de millisecondes, ce qui est très lent par rapport à la vitesse de la mémoire de l'ordinateur – de l'ordre de la nanoseconde. Le processeur envoie alors souvent une requête au disque puis, au lieu d'attendre le résultat de cette requête, met en sommeil cette tâche, en attente de la réponse, et effectue une autre tâche plus prioritaire.

Pendant ce temps, le disque effectue le travail qui lui a été demandé et transfère à son rythme les données entre le support – magnétique ou optique – et la mémoire du processeur. Quand cela est terminé, il signale au processeur, grâce à une interruption, la fin du transfert, ce qui a pour effet de réveiller la tâche endormie.

Le système de gestion de fichiers

Une partie du système d'exploitation appelée « système de gestion de fichiers » effectue pour l'utilisateur

– la vérification de l'existence d'un fichier de nom donné,

– la transformation entre le nom du fichier et son adresse physique sur l'ensemble des périphériques connectés sur une machine – ce qui permet de masquer les propriétés physiques de tel ou tel périphérique,

– la gestion des droits d'accès — le système vérifie que le programme utilisateur a le droit de lire, d'écrire ou d'exécuter le fichier sélectionné,

– certaines optimisations de performance des accès demandés.

La mémoire virtuelle

Une des fonctions du système d'exploitation est de laisser croire à chacun des utilisateurs que la mémoire qui lui est attribuée est bien plus grosse que la mémoire physique de la machine. L'idée est simple : stocker le programme et les données de chaque utilisateur, par exemple sur les disques durs de la

machine qui sont de très grande taille, et aller rechercher ces données au moment où l'on en a besoin.

Mais, comme nous l'avons vu, les disques durs sont très lents par rapport à la vitesse de la mémoire interne du processeur. Le temps d'accès à la mémoire est de l'ordre de la nanoseconde, alors que celui d'un bon disque dur est de l'ordre de la milliseconde, d'où un rapport de vitesses de l'ordre du million. Et on ne peut naturellement pas se permettre une chute de performance d'un facteur un million.

Une solution est de copier en mémoire vive, non une case mémoire unique, mais des blocs de données contiguës, l'hypothèse étant que, lorsque l'on vient d'utiliser une donnée, il est probable que l'on utilise les données contiguës dans un avenir proche. Cette hypothèse s'appelle la *localité spatiale*. Une autre hypothèse est que, lorsqu'on utilise une case mémoire, par exemple une instruction, les programmes étant très souvent constitués de boucles, il est probable que l'on réutilise cette même case mémoire dans un avenir proche. Cette hypothèse s'appelle la *localité temporelle*.

Sachant que les disques durs transfèrent des blocs de données de taille fixe, appelés *pages*, on peut découper la mémoire, ou une partie de celle-ci, en « cadres » de la taille d'une page, et utiliser un mécanisme qui effectue la transformation de l'adresse logique, appelée le plus souvent *adresse virtuelle*, en une adresse physique en mémoire. Cela peut être réalisé par un dispositif matériel appelé MMU (*Memory Management Unit*) qui transforme les adresses logiques en adresses physiques et qui, en cas de défaut, lance un programme d'interruption spécifique qui gère les pages, entre les disques et les cadres.

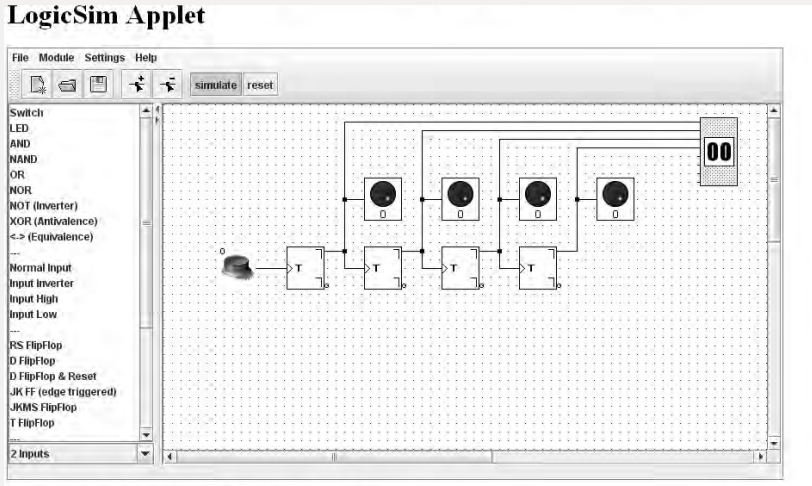
L'extension de l'ensemble des instructions

Une machine donnée détecte et exécute directement un jeu d'instructions assez réduit. En revanche, elle détecte aussi les instructions inexistantes éventuelles. Cette détection engendre un signal d'interruption qui lance un programme spécifique.

Cette méthode permet, par exemple, de traiter par logiciel des « instructions » trop complexes pour la machine câblée. C'est ainsi qu'un processeur ne possédant pas d'opérateur flottant, par exemple, peut détecter les instructions flottantes et les faire traiter par un programme spécifique qui se substitue alors au processeur de base. Comme pour toute interruption, cette opération s'effectue à l'insu, ou presque, de la tâche principale. Du point de vue de l'utilisateur, seul le temps d'exécution sera donc différent entre deux machines possédant tel type d'instruction câblée ou programmée.

Quelques simulateurs permettant de se familiariser avec la logique et les processeurs

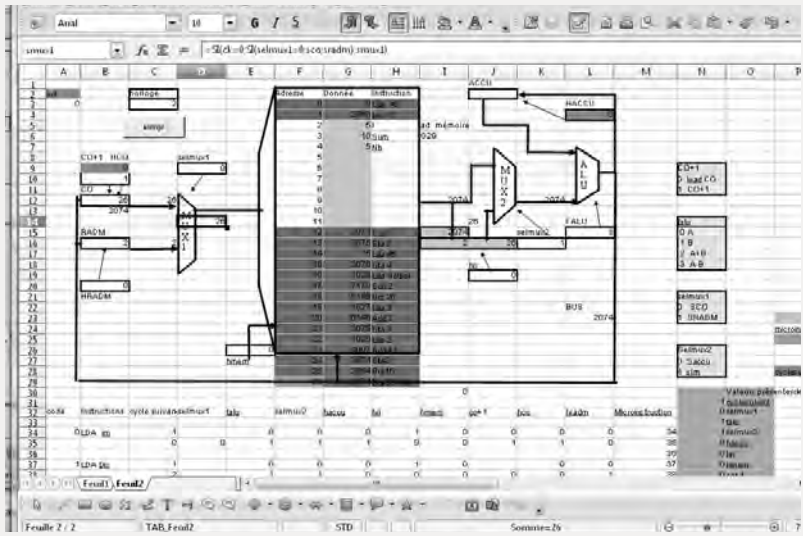
De nombreux simulateurs de logique simple, commerciaux et non commerciaux, existent sur le Web. On citera des simulateurs écrits en Java comme LogicSim `http://www.tetzl.de/java_logic_simulator.html`, facile d'emploi, mais limité en taille à de petits circuits de quelques dizaines de portes.



Exemple de simulateur de portes logiques écrit en Java (LogicSim de Tetzl).

Des simulateurs écrits avec des tableurs: `http://www.e-campus.uvsq.fr/dutransistoralordinateur/`.

Au niveau des simulateurs de machine on trouve des simulateurs écrits en C, en Java ou avec des fonctions de tableurs `http://www.e-campus.uvsq.fr/dutransistoralordinateur/`.



Exemple de simulateur de machine sur tableur.

Pour en savoir plus

Joffroy Beauquier et Béatrice Bérard, *Systèmes d'exploitation : concepts et algorithmes*, Ediscience international, 1994.

John L. Hennessy et David A. Patterson, *Computer Architecture : A Quantitative Approach*, Morgan Kaufmann, 1996.

Jean-Michel Muller, *Arithmétique des ordinateurs*, Masson, 1989.

Andrew Tanenbaum, *Structured Computer Organization*, Prentice-Hall, 2005.

Andrew Tanenbaum, *Systèmes d'exploitation*, Pearson Education, 2008.

Claude Timsit, *Du transistor à l'ordinateur*, Hermann, 2010.

Réseaux

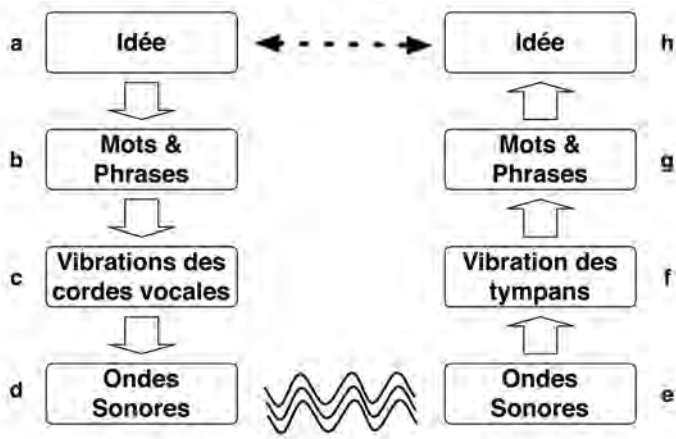
Les ordinateurs, nous l'avons vu, sont construits en assemblant des « briques de base » : les transistors. Ces ordinateurs constituent à leur tour des briques de base qui s'assemblent en réseaux. Ces réseaux s'étendent sur des dizaines de milliers de kilomètres et cette extension spatiale change leur finalité. Avec un ordinateur, le but était d'effectuer un calcul, et la transmission des informations par le bus, d'un bout à l'autre d'un ordinateur – par exemple du processeur à la mémoire –, était un moyen pour atteindre ce but. Transmettre des informations d'un bout à l'autre d'un réseau est, en revanche, un but en soi. C'est pour cela que les réseaux sont abordés du point de vue de leur fonction : la communication. La communication entre les ordinateurs utilise des techniques spécifiques. Cependant, ses principes élémentaires ne sont pas très différents de ceux de la communication entre les êtres humains. Cette analogie nous servira de guide tout au long de ce chapitre, consacré aux principes qui régissent la communication entre les ordinateurs, et notamment au réseau Internet.

Cours

Une des formes de communication les plus immédiates est la communication orale, qui peut prendre diverses formes, du chant au discours, dans des langues variées. Cependant, toutes ces formes ont en commun d'être un moyen pour véhiculer des idées, d'un émetteur à un récepteur : une mère chantant une berceuse à son enfant lui communique des idées combinant sécurité et calme, tandis qu'un homme politique prononçant un discours devant ses électeurs potentiels leur communique des idées alliant « je comprends la situation » et « je pourrais bien vous représenter ».

Communication entre êtres humains

Faute de pouvoir employer la télépathie, nous avons souvent recours à l'oral pour communiquer une idée. Nous ferons l'hypothèse simplificatrice que, lors d'une communication entre deux êtres humains, le locuteur a, dans un premier temps, une idée dans la tête, et qu'il formule cette idée en mots et en phrases, dans un second temps. Ces mots et ces phrases sont alors transmis au récepteur qui les comprend et peut ainsi se représenter l'idée dans sa propre tête. Pour être communiqués par la parole, les mots et les phrases doivent être traduits en ondes sonores produites par les cordes vocales de l'émetteur. Ces ondes sont le *support physique* de la communication et se propagent jusqu'aux oreilles du récepteur. Les vibrations induites des tympans de ce dernier sont retraduites en mots et en phrases. Cette décomposition en huit étapes peut se symboliser ainsi.

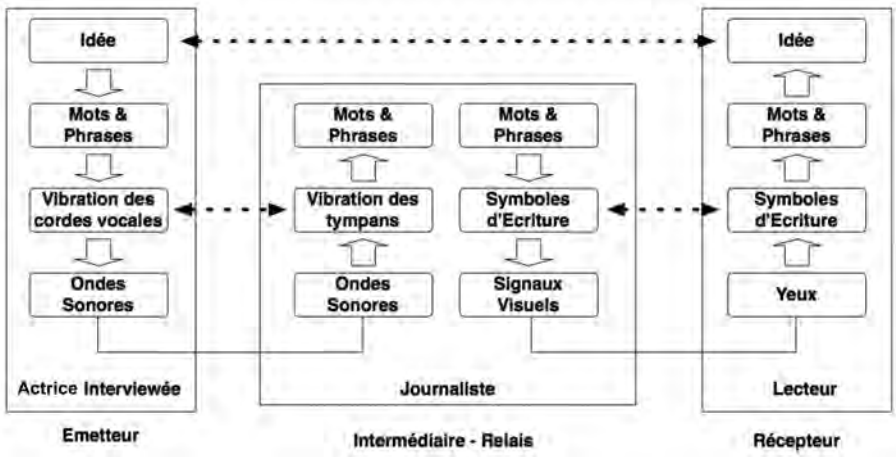


Communication orale entre êtres humains.

Bien sûr, pour que la communication fonctionne, il faut que les mots soient compréhensibles, énoncés dans une langue préétablie, et prononcés de manière suffisamment intelligible pour être entendus correctement.

Une autre forme de communication fréquemment utilisée est la communication écrite. Par rapport à l'oral, le support physique change – ce ne sont plus des ondes sonores, mais des taches d'encre sur une feuille de papier –, mais l'émetteur formule toujours ses idées en mots et en phrases qui sont transmis au récepteur qui les comprend, et saisit les idées. Ce mode de fonctionnement

commun permet par exemple à un journaliste de servir d'*intermédiaire*, et de retranscrire ce qu'a déclaré une actrice au cours d'une interview.



Lecture d'une retranscription écrite d'une interview orale.

Des lecteurs pourront ultérieurement en prendre connaissance, pourvu bien sûr que les taches d'encre soient lisibles et exprimées dans une langue connue.

Comme nous allons le voir par la suite, la communication entre ordinateurs fonctionne grâce à des mécanismes équivalents à ceux décrits jusqu'ici pour la communication entre les êtres humains.

À retenir :

– La communication est un moyen, et non un but.

– Les idées communiquées d'un esprit à un autre sont l'essentiel, les moyens de communication sont variables et secondaires.

– Des mécanismes communs à différentes formes de communication permettent de relayer une idée à travers des intermédiaires, au moyen de supports qui peuvent être hétérogènes.

Communication entre ordinateurs, réseaux d'ordinateurs

Un ordinateur peut manipuler des quantités d'informations diverses, pourvu que celles-ci soient représentées sous forme binaire, c'est-à-dire écrites au moyen d'un alphabet simplifié contenant seulement deux lettres, qu'il est coutume de représenter par les chiffres 0 et 1 – voir le premier chapitre. Dans cet alphabet, les mots et les phrases représentant l'information s'écrivent donc sous

forme de suites de bits, comme par exemple 0001011100010. Dans la suite de ce chapitre, on appellera ces informations binaires des *données*.

Comme les humains, les ordinateurs sont capables de communiquer entre eux quand ils sont reliés les uns aux autres, par exemple via Internet. Au travers de programmes appelés *applications*, les internautes peuvent transmettre d'un ordinateur à l'autre des amas de bits cohérents appelés *fichiers* représentant par exemple du texte, du son ou de l'image. Les applications sont la partie visible d'Internet, que tout le monde connaît et utilise tous les jours. Le courriel, la navigation web, ou le *chat* sont des exemples d'applications.

Les fichiers transmis sont reçus quasiment instantanément à des dizaines de milliers de kilomètres de là, comme par magie. Comment cela marche-t-il ? Le fonctionnement d'Internet est simple, il est fondé sur la décomposition d'un problème complexe en plusieurs sous-problèmes plus faciles à résoudre. Pour en saisir les principes, reprenons l'analogie avec la communication écrite entre humains, sous la forme épistolaire, étape par étape.

Étape 1 : L'équivalent de écrire et lire les lettres de l'alphabet pour communiquer par écrit entre humains. Pour les ordinateurs, sachant que les informations à transmettre sont représentées par des bits – les chiffres 0 et 1 –, le premier pas est d'être capable de transférer un 0 ou un 1 au moyen d'un support physique reliant deux ordinateurs – le plus souvent par câble ou par radio.

Étape 2 : L'équivalent de écrire et lire des mots et des phrases. Pour un ordinateur, il s'agit de coordonner le transfert groupé de suites de bits d'un bout à l'autre du câble ou du lien radio. On appelle de tels groupements des *paquets*, analogues aux mots et aux phrases des langages humains.

Étape 3 : L'équivalent de fournir un service postal pour acheminer des lettres à bon port. Pour un ordinateur qui veut joindre un autre ordinateur distant, il faut gérer le transfert de paquets à travers des ordinateurs intermédiaires, interconnectés par une série de câbles et/ou liens radio qui mènent le paquet jusqu'à destination. On appelle cette interconnexion un réseau d'ordinateurs, un réseau de données ou plus simplement : un *réseau*. Le plus vaste et le plus connu d'entre eux est Internet, qui est sans doute la construction la plus grande en taille jamais réalisée par l'homme.

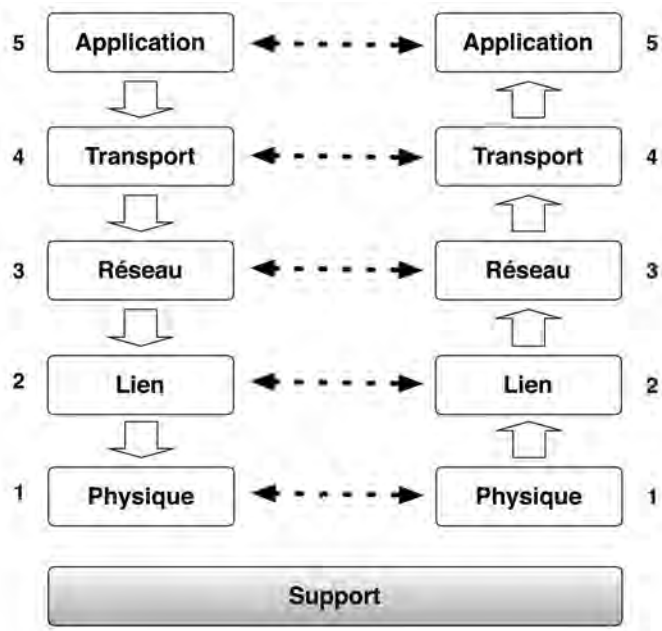
Étape 4 : L'équivalent d'adapter les envois au format lettre du service postal disponible. Par exemple, pour transférer des fichiers entiers d'un ordinateur à un autre à travers le réseau, il faut fragmenter l'information d'un fichier en plusieurs paquets – un fichier est souvent trop gros pour tenir dans un seul paquet. Il faut également assurer la fiabilité du transfert de chaque paquet jusqu'à destination, où le fichier sera alors réassemblé.

Étape 5 : L'équivalent de la communication par lettres. Pour un ordinateur, il s'agit de permettre aux applications d'utiliser Internet pour envoyer des données vers d'autres applications sur d'autres ordinateurs, à travers le réseau.

Organisation en pile

Les étapes listées précédemment correspondent à une décomposition type, appelée organisation en pile, qui régit les réseaux d'ordinateurs en général et Internet en particulier. Cette décomposition est similaire à l'empilement décrit précédemment pour la communication entre humains – voir le paragraphe « Communication entre êtres humains ».

La communication entre ordinateurs, quant à elle, utilise un empilement de couches, présentes sur chaque ordinateur du réseau. Ces différentes couches et leurs interactions peuvent se représenter ainsi.



Communication entre deux ordinateurs : organisation en pile présente sur chaque ordinateur. Chaque couche empilée communique avec son homologue – flèches horizontales – pour fournir des services à la couche immédiatement supérieure, en utilisant les services fournis par la couche immédiatement inférieure.

Chaque couche est constituée d'un ensemble de programmes dédiés à fournir les fonctionnalités listées dans l'une des 5 étapes décrites précédemment.

On appelle *protocoles* les programmes dédiés à faire fonctionner le réseau. Ce chapitre introduit les mécanismes fondamentaux à l'œuvre dans les protocoles, ainsi que l'architecture en *couche* qui gouverne l'ensemble.

Couche 1 : La *couche physique* est constituée des protocoles responsables du transfert individuel d'un bit, 0 ou 1, à travers un support physique – généralement un câble ou un lien radio. Nous verrons des exemples simples de mécanismes à l'œuvre au sein de cette couche au paragraphe « La couche physique ».

Couche 2. La *couche lien* est constituée des protocoles responsables de la coordination du transfert de paquets à travers un support physique, de l'identification des ordinateurs directement connectés à ce support et de la gestion du « temps de parole » de chacun sur ce support. Des exemples typiques de protocoles de cette couche sont le Wifi ou l'Ethernet.

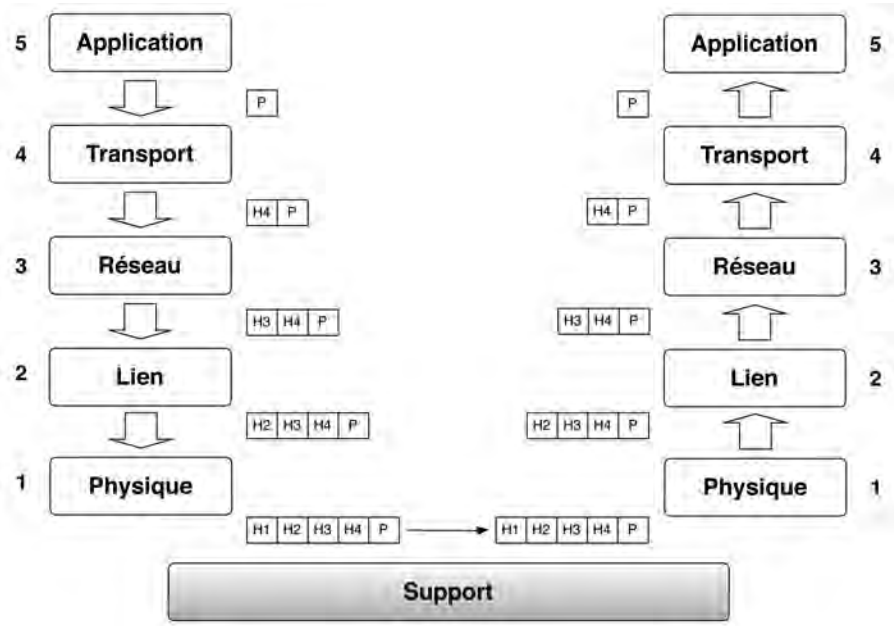
Couche 3. La *couche réseau* est constituée des protocoles responsables de l'aiguillage de chaque paquet vers sa destination, à chaque embranchement entre différents supports physiques rencontrés au cours du périple de ce paquet à travers le réseau. Ces protocoles doivent également identifier les ordinateurs connectés sur le réseau. Le protocole le plus connu de cette couche est le protocole IP (*Internet Protocol*).

Couche 4. La *couche transport* est constituée des protocoles responsables de l'empaquetage des données à transmettre, ainsi que de la coordination entre l'envoyeur et le destinataire des paquets, pour assurer la fiabilité de leur transport de bout en bout. Cette couche doit également identifier les applications en cours d'exécution qui utilisent le réseau. Le protocole le plus connu de cette couche est TCP (*Transmission Control Protocol*).

Couche 5. La *couche application* est constituée des programmes appelés *applications* qui utilisent le réseau. Tout comme la transmission d'idées est le but de la communication entre humains – voir ci-avant –, le but de la communication entre ordinateurs est la transmission de données entre applications. Les programmes de cette couche ne sont pas dédiés à faire fonctionner le réseau, et ne sont donc pas abordés en détail dans ce chapitre.

Chaque couche peut correspondre avec son homologue – flèches horizontales – située sur un autre ordinateur, à travers une « boîte aux lettres » fournie par la couche du dessous, schématisée par des fonctions interfaces simples, comme *recevoir*() et *envoyer*(). Quand, sur un ordinateur, la couche n reçoit un paquet P à transmettre de la part de la couche du dessus $n + 1$ à son homologue sur un autre ordinateur, elle encapsule tel quel ce paquet, dans un paquet plus grand P' avec en en-tête (*header*) les informations nécessaires, spécifiques au bon fonctionnement de la couche n, pour l'acheminement du paquet vers la couche n de l'ordinateur destinataire, soit : $P' = [\text{header}(n) : P]$.

Autrement dit : les paquets à transmettre pour le compte de la couche $n + 1$ sont *tels quels* le contenu des paquets transmis par la couche n. Cette dernière ajoute un en-tête contenant des informations de contrôle qui sont utilisées par la couche n, comme par exemple l'adresse de l'ordinateur destinataire, et certaines autres à l'aide desquelles des services élaborés peuvent être fournis par la couche n. Ce principe s'illustre ainsi, pour $n = 5$.



Interactions verticales entre les couches. En-têtes imbriqués.

Le paquet poursuivant son parcours, la couche n sollicite ensuite la couche $n - 1$ pour envoyer P'. Cette dernière encapsule P' selon le même principe, dans un paquet plus grand $P'' = [header(n - 1) : header(n) : P]$. Et ainsi de suite jusqu'à la couche 1, la couche physique, qui transmet le paquet final $[header(1) : \ldots : header(n - 1) : header(n) : P]$ sous forme de suite de 0 et de 1 à travers un support physique.

Le paquet est réceptionné par la couche 1 de l'ordinateur destinataire, qui consulte l'en-tête la concernant, à savoir *header*(1), et si celui-ci est estimé correct, transmet le contenu du paquet de son point de vue, soit $[header(2) : \ldots : header(n - 1) : header(n) : P]$, à la couche 2 de l'ordinateur destinataire. Cette dernière consulte l'en-tête la concernant, et si celui-ci est estimé correct, transmet le contenu du paquet de son point de vue, omettant donc *header*(2), à la

couche 3 et ainsi de suite jusqu'à la couche $n + 1$ qui reçoit donc finalement le paquet P qui lui a été envoyé.

Les informations de contrôle contenues dans l'en-tête utilisé par la couche n contiennent en général de nombreux renseignements utiles pour les couches n homologues d'autres ordinateurs. Cet en-tête contient entre autres choses l'identité de la destination du paquet, du point de vue de la couche n. Chaque couche utilise un système d'identification spécifique : un certain format d'adresse adapté aux tâches particulières que doit accomplir cette couche. En conséquence, quand la couche $n - 1$ reçoit un paquet [*header*(n) : P] à transmettre pour le compte de la couche n, l'identité de la destination du paquet du point de vue de la couche $n - 1$ doit être déduite de l'adresse contenue dans l'en-tête de la couche supérieure *header*(n). L'identité déduite sera alors renseignée dans *header*($n - 1$). Ce procédé s'appelle la *résolution d'adresse*.

Les informations de contrôle contenues dans l'en-tête utilisé par la couche n ainsi que leur traitement par la couche n homologue sur l'ordinateur destinataire, interactions horizontales dans la figure ci-avant, sont en général complexes, au contraire de l'interaction entre la couche n et les couches $n + 1$ et $n - 1$, interactions verticales, qui se résume simplement aux fonctions *recevoir*() et *envoyer*(). Cela permet notamment de totalement changer les mécanismes d'une couche sur un ordinateur sans avoir à changer les mécanismes des autres couches, tant que les interfaces *recevoir*() et *envoyer*() avec les couches immédiatement supérieures et inférieures sont conservées. L'organisation en pile permet essentiellement aux mécanismes internes d'une couche d'être « agnostique » concernant les mécanismes internes des autres couches.

La suite de ce chapitre entre plus en détail dans le fonctionnement de chaque couche, ainsi que de quelques protocoles clés. Il existe d'autres modèles d'organisation en couche que celui décrit ci-avant. Le plus connu d'entre eux est le modèle OSI (*Open Systems Interconnection*) qui définit sept couches au lieu des cinq présentées ci-avant. Malgré cette différence, le modèle OSI fonctionne selon le même principe générique d'organisation en pile.

À retenir :

– Les ordinateurs communiquent entre eux de manière comparable aux êtres humains entre eux, au moyen d'un alphabet simplifié et de langues que l'on nomme protocoles.

– Les protocoles sont organisés en couches empilées les unes sur les autres, présentes sur chaque ordinateur du réseau.

– Sur un ordinateur, chaque couche interagit de manière complexe avec la couche de même niveau, son homologue, sur l'ordinateur avec lequel on communique.

– Sur un ordinateur, chaque couche interagit de manière simple avec les couches qui lui sont directement inférieures et directement supérieures.

La couche physique

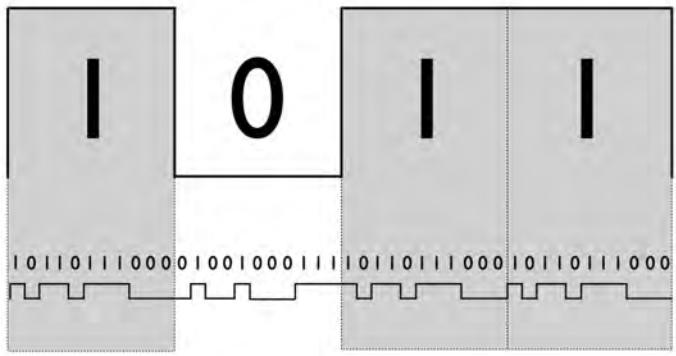
La tâche accomplie par les protocoles de cette couche est le transfert individuel d'un 0 ou d'un 1, d'un bout à l'autre d'un support physique. Plusieurs types de supports physiques sont utilisés pour connecter les ordinateurs entre eux : des câbles métalliques véhiculant des électrons, des câbles optiques véhiculant des photons, le vide véhiculant des ondes radio sont des exemples de supports physiques utilisés de nos jours. Pour chacun de ces supports, il existe des protocoles spécialisés dans le transfert individuel d'un 0 ou d'un 1 d'un bout à l'autre du support.

Pour saisir le principe de base de ces protocoles, on peut se rappeler les systèmes de communication très anciens, comme par exemple celui utilisé lors de l'élection du pape, qui date du Moyen Âge. L'élection se fait à huis clos dans une chapelle où sont enfermés les électeurs, qui n'ont le droit de communiquer avec le reste du monde qu'à travers la cheminée de la chapelle jusqu'à ce qu'un nouveau pape soit élu : après chaque scrutin, les cardinaux communiquent les résultats par une fumée noire – vote non concluant, l'équivalent d'un 0 – ou par une fumée blanche – vote concluant, l'équivalent d'un 1. En voyant la couleur de la fumée, les observateurs extérieurs comprennent le message élémentaire envoyé par les cardinaux : un nouveau pape est-il élu, oui ou non, 0 ou 1.

Les protocoles modernes à l'œuvre dans la couche physique fonctionnent sur une base similaire. À la place de signaux de fumée observables à l'échelle macroscopique, on utilise des signaux observables à l'échelle microscopique, à base d'ondes électromagnétiques. À la place des variations de couleurs – blanc ou noir – pour coder l'information binaire, on utilise des variations de longueurs d'onde, de phase ou d'intensité du signal, etc.

Dans le cas de la communication par des signaux de fumée lors de l'élection du pape, le feu est traditionnellement un feu de paille. Celle-ci est mouillée quand il faut produire une fumée blanche. Depuis quelques années, des fumigènes sont utilisés en complément, pour éviter les confusions causées par une fumée trop grise, ou une fumée pas assez visible par mauvais temps.

De la même manière, les protocoles modernes à l'œuvre dans la couche physique utilisent des compléments sophistiqués pour renforcer la clarté du signal, et le rendre plus résistant aux erreurs d'interprétation à la réception. Une des techniques est de transmettre des séquences établies à l'avance, par exemple transmettre la suite 10110111000 au lieu de transmettre simplement 1, et transmettre 01001000111 au lieu de transmettre 0.



Codage imbriqué pour renforcer le signal.
Transmission de séquences de bits prédéfinies, en bas,
interprétées chacune comme un bit unique, en haut, à la réception.

À la réception, on sait que l'on ne devrait recevoir qu'une suite de séquences complètes et correctes, et cela permet donc de deviner l'information d'origine, même quand la transmission d'une partie d'une séquence est brouillée. Cette technique sert dans les communications sans-fil – notamment au sein de la couche physique utilisée avec le protocole Wifi –, qui doivent souvent composer avec des signaux très brouillés par les interférences, les obstacles, etc. Le coût de cette résistance aux erreurs de transmission est donc un certain nombre de transmissions supplémentaires effectuées *a priori*, en amont d'erreurs potentielles.

La couche lien

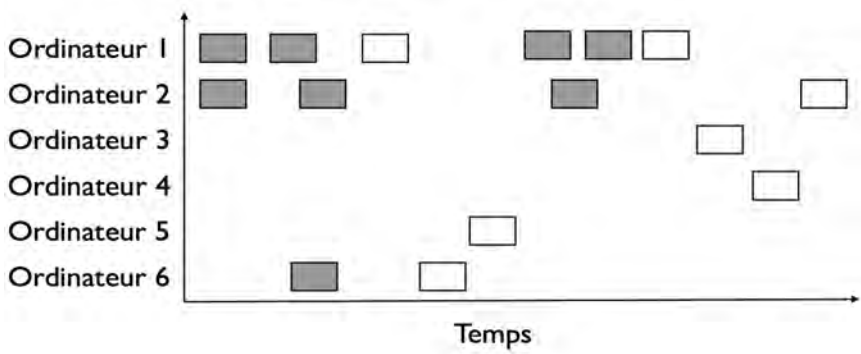
Les protocoles de cette couche coordonnent le transfert de paquets à travers le support physique, identifient les ordinateurs directement connectés à ce support et gèrent le « temps de parole » de chacun sur le support. Selon le support physique utilisé, les protocoles diffèrent, notamment en ce qui concerne la gestion du temps de parole de chacun sur le support.

Pour saisir les concepts de base de ces protocoles, un exemple pionnier, datant des années 1970, est éclairant : ALOHAnet, un réseau d'ordinateurs utilisant des communications sans-fil pour connecter des ordinateurs dispersés sur les îles de l'archipel d'Hawaï à un ordinateur central situé sur l'une d'entre elles. La contrainte principale à cette époque était que tous les ordinateurs du réseau ALOHAnet devaient utiliser l'unique fréquence radio disponible pour communiquer avec l'ordinateur central. Cette contrainte crée des situations où plusieurs ordinateurs pourraient simultanément tenter d'envoyer chacun

un paquet à l'ordinateur central, et, sans le savoir, brouilleraient mutuellement leurs messages qui deviendraient incompréhensibles pour l'ordinateur central. On appelle ce brouillage mutuel une *collision* entre paquets, similaire à la situation où deux personnes parlent en même temps à une troisième qui, de ce fait, ne comprend rien.

Une solution pour gérer les collisions consiste à figer un ordre tournant que les ordinateurs doivent respecter pour que chacun puisse transmettre à son tour pendant un certain temps. Cependant, cette solution centralisée a des inconvénients. D'une part, si l'on ajoute ou enlève des ordinateurs, il faut tout reprogrammer avec un nouvel ordre à respecter. D'autre part, avec cet ordre systématique, on brime un ordinateur qui a soudainement beaucoup à transmettre si, pendant ce temps-là, les autres n'ont rien à dire.

Une autre solution a donc été retenue pour gérer les collisions de manière distribuée et plus flexible : le *protocole ALOHA*. Son mécanisme est simple : chaque ordinateur est identifié par une adresse représentée sous forme d'une suite de k bits, le nombre k étant fixé à l'avance par convention. Dans chaque paquet à transmettre, l'ordinateur ajoute en en-tête son adresse ainsi que celle de la destination. Dès qu'un ordinateur a un paquet à transmettre, il l'envoie immédiatement, et attend un acquittement de la part de la destination lui confirmant que le paquet a bien été reçu. Si l'acquittement n'est pas reçu avant un temps d'attente maximum, fixé à l'avance, on estime que le paquet a été victime d'une collision. Dans ce cas, l'ordinateur attend un certain temps avant de transmettre le paquet de nouveau, la valeur de ce temps d'attente étant choisie aléatoirement pour réduire les chances de nouvelles collisions.



Protocole ALOHA. Paquets victimes de collisions en gris. Paquets correctement transmis en blanc. Le temps d'attente entre un paquet gris et le prochain paquet sur la même ligne est aléatoire.

Si un paquet subit trop d'échecs de transmission, le protocole abandonne.

De nos jours, les protocoles utilisés à la couche lien sont le plus souvent *Ethernet*, par câble, ou *Wifi* et *Bluetooth* par radio. Cette catégorie de protocoles s'appelle les *protocoles de contrôle d'accès* au support physique, ou protocoles MAC (*Medium Access Control*). Ils sont, bien sûr, plus sophistiqués et plus performants qu'ALOHA. Cependant, ils ont le même mécanisme de base qui leur permet de partager efficacement l'accès à un support physique connectant entre eux plusieurs ordinateurs identifiés chacun par une adresse sous forme de suite de bits, et d'envoyer des paquets de données d'un ordinateur à l'autre à travers ce support, câble ou radio. La différence essentielle entre ALOHA et des protocoles comme Ethernet, Wifi ou Bluetooth est dans la manière de réduire les temps d'attente avant de retransmettre après une collision, tout en minimisant le nombre et l'impact des collisions, le but étant d'optimiser l'utilisation du support physique lorsque plusieurs ordinateurs ont beaucoup de données à transmettre en même temps.

Dans la suite de ce chapitre, on appellera lien l'abstraction fournie par les protocoles de la couche lien, permettant donc la transmission de paquets de données sur un support physique donné.

Identifiants utilisés à la couche lien

Les identifiants utilisés par la couche lien doivent être cohérents localement, dans le contexte d'un lien. Cette cohérence est assurée si et seulement si chaque identifiant désigne au plus une *interface* connectant un ordinateur à ce lien.

Du point de vue de la couche lien, la destination d'un paquet est l'une de ces interfaces. Le plus souvent, une interface est identifiée par une *adresse* MAC, par exemple dans les cas où la couche lien utilise Ethernet, Wifi ou Bluetooth. Une adresse MAC consiste en une suite de 48 bits, souvent notés de manière plus humaine sous forme hexadécimale, dans un format regroupant des « mots » de 8 bits, comme par exemple 10 :93 :e9 :0a :42 :ac. L'en-tête des paquets envoyés par la couche lien comporte alors l'adresse MAC de l'ordinateur destinataire, ainsi que l'adresse MAC de l'ordinateur émetteur du paquet.

Il existe cependant une exception utile en pratique pour s'adresser à « tout le monde en même temps » : dans certains cas, en effet, la destination d'un paquet n'est pas une seule interface, mais toutes les interfaces connectées au lien. Dans ce cas, l'adresse de destination indiquée dans le paquet est une adresse MAC spéciale, dite *adresse de diffusion*, désignant par convention « tout le monde connecté à ce lien ». Quand un ordinateur reçoit un paquet avec une

telle adresse comme destination, il traite le paquet comme si celui-ci lui était destiné personnellement.

D'autre part, il existe d'autres formats d'adresses utilisés par des protocoles MAC autres qu'Ethernet, Wifi ou Bluetooth. Le protocole ATM (*Asynchronous Transfer Mode*), protocole MAC utilisé généralement sur fibre optique, met en œuvre des identifiants de 160 bits. Un autre exemple est le protocole *Frame Relay*, fondé sur des adresses de longueur variable. Chaque protocole MAC utilise un format d'adresse adapté à la nature du support physique en jeu, au nombre d'ordinateurs maximal pouvant y être connectés, et à la syntaxe utilisée pour former chaque adresse, qui peut fournir dans certains cas des informations supplémentaires comme l'identité de l'industriel qui a construit l'interface réseau, etc. Chacune de ces adresses est valide localement et utilisée de manière cohérente sur le lien auquel elle est associée.

On notera finalement que la couche lien n'a pas de résolution d'adresse à accomplir, vu qu'elle est la couche la plus basse utilisant le concept d'adresse. La couche lien est en effet immédiatement supérieure à la couche physique, qui n'a elle-même pas de concept d'adresse, se contentant de transmettre un par un les bits représentant les paquets envoyés par la couche lien, sur le support physique requis.

La couche réseau

Les protocoles de cette couche sont responsables de l'aiguillage de chaque paquet vers sa destination, à chaque embranchement entre différents liens rencontrés au cours du périple de ce paquet à travers le réseau. Les protocoles de la couche réseau doivent également identifier les ordinateurs connectés sur le réseau.

Les identifiants utilisés à la couche lien ne sont cohérents que localement, sur le support physique auquel ils sont associés, et leur format varie selon le protocole d'accès utilisé à la couche lien – voir le paragraphe « La couche lien ». Pour identifier les ordinateurs de manière cohérente sur le réseau tout entier, et non plus simplement sur un seul lien, il faut donc utiliser un autre système. Ce système est la couche réseau qui le fournit avec le protocole IP (*Internet Protocol*) qui définit les *adresses IP*, un format d'adresse indépendant des protocoles utilisés à la couche lien, et cohérent à l'échelle du réseau entier. Les adresses IP consistent en 32 bits souvent notés de manière plus humaine sous forme de 4 mots de 8 bits, donc chacun exprimable sous la forme d'un nombre compris entre 0 et $2^8 - 1 = 255$, comme par exemple 216.239.59.104. Ces adresses permettent donc d'identifier un maximum de 2^{32}, soit quatre milliards, ordinateurs de manière unique.

Du point de vue de la couche réseau, la destination d'un paquet est un ordinateur connecté à Internet, identifié donc par une adresse IP, qui figure dans l'en-tête des paquets envoyés par la couche réseau. L'en-tête de ces paquets contient de plus l'adresse IP de l'ordinateur émetteur du paquet. L'en-tête des paquets IP contient de nombreux renseignements en plus des adresses IP, dont notamment deux informations permettant de vérifier la validité du paquet IP. D'une part, un code correcteur sous la forme d'une somme de contrôle : un entier codé sur 16 bits, résultat de l'addition des bits constituant le paquet envoyé, qui doit être recalculée et vérifiée comme étant inchangée à l'arrivée du paquet, sinon IP abandonne le traitement de ce paquet manifestement victime d'une erreur de transmission, qui est donc perdu. D'autre part, une durée de vie pour le paquet, un entier codé sur 8 bits, décrémenté d'une unité à chaque fois que le paquet est aiguillé à un embranchement du réseau. Si ce temps de validité devient nul avant que le paquet n'arrive à destination, IP abandonne le traitement de ce paquet qui, manifestement, ne trouve pas son chemin et est donc perdu.

Pour trouver son chemin à travers le réseau de câbles et de liens radio connectant les ordinateurs entre eux, jusqu'à une destination identifiée par son adresse IP, il faut faire appel à un type de protocole supplémentaire, faisant également partie de la couche réseau : un *protocole de routage*. On appelle un ordinateur utilisant un protocole de routage un *routeur*. Il existe d'ailleurs au cœur d'Internet des ordinateurs qui sont dédiés au routage, dont les programmes ne font partie que des couches 1, 2 et 3. *A contrario*, on appelle les autres ordinateurs des *hôtes*, dont les programmes font partie de toutes les couches, entre autres la couche application utilisée par les internautes. Chaque hôte est connecté à un support physique qui le connecte à au moins un routeur, lequel se charge d'acheminer les paquets émis par cet hôte ou destinés à cet hôte, grâce à la connaissance des aiguillages appropriés à l'état actuel du réseau que lui fournit le protocole de routage utilisé. Un hôte n'a pas besoin d'avoir cette connaissance, vu que les routeurs s'en chargent et font office de « guichet abstrait » pour utiliser cette connaissance.

Un protocole de routage très simple est le suivant : on fixe l'état du réseau, on observe les différents chemins possibles, on en déduit les chemins les meilleurs et on préprogramme chaque routeur avec une table de règles simplifiées s'apparentant à des panneaux routiers :

– pour aller à E, passer par B, distance = 3
– pour aller à A, passer par B, distance = 2
– pour aller à C, passer par D, distance = 2

– pour aller à D, passer par D, distance = 1
– pour aller à B, passer par B, distance = 1
– etc. (une règle par destination possible)

Chaque lettre A, B, C… est en réalité une adresse IP identifiant une destination du point de vue de la couche réseau. Grâce à ce type de table appelée *table de routage*, un routeur peut aiguiller un paquet dans la bonne direction, sachant sa destination finale – une adresse IP –, en le relayant sur le lien qui emmène le paquet le plus près possible de sa destination – l'équivalent de prendre le bon embranchement, la bonne route. Cette solution manuelle a cependant un inconvénient majeur : si l'on ajoute ou enlève des ordinateurs ou si on modifie les liens entre eux, il faut reprogrammer tous les ordinateurs. D'autres solutions ont donc été utilisées pour gérer les tables de routage de manière automatisée et plus flexible : des *protocoles de routage*.

Un protocole de routage très simple est le protocole *vecteur de distance*, basé sur l'algorithme de *Bellman-Ford* fonctionnant de la manière suivante. Chaque routeur diffuse périodiquement, par exemple toutes les 30 secondes, sur tous les liens auxquels il est connecté, un paquet spécial appelé *HELLO* contenant sa table de routage actuelle. Vide au départ, cette dernière se remplit au fur et à mesure que le routeur entend les paquets *HELLO* envoyés par les autres routeurs qu'il entend émettre, et se tient ainsi informé en temps réel de l'ensemble de ses *voisins* : les routeurs avec lesquels il peut communiquer directement via le ou les supports physiques auxquels il est connecté. Par convention, les voisins sont notés dans la table de routage comme étant à distance 1.

De plus, en consultant les tables de routage de ses voisins indiquées dans les *HELLO* qu'il reçoit périodiquement de chaque voisin, un routeur *entend parler* progressivement d'autres routeurs qui ne sont pas ses voisins, mais des voisins de ses voisins – des routeurs notés à distance 2 dans les tables de routage –, puis des voisins des routeurs à distance 2 – donc notés à distance 3 dans les tables de routage – et ainsi de suite, toujours par le truchement de ses voisins directs et de leur *HELLO*. Il peut ainsi répercuter ces nouvelles informations dans sa propre table de routage en renseignant continuellement, via les *HELLO* qu'il envoie à toutes les destinations dont il a connaissance au moment de l'envoi, ainsi que la distance la plus courte pour y arriver, dont il a entendu parler jusqu'à présent, et par le truchement de quel voisin.

Ainsi, la table de routage de chaque routeur se construit correctement et puis se tient à jour automatiquement. Le protocole vecteur de distance n'est cependant pas d'une robustesse à toute épreuve, notamment dans certains cas où le protocole dérègle durablement les tables de routage en ne détectant pas

correctement qu'un ou plusieurs routeurs sont soudainement devenus hors-service. Pour cette raison, des variantes plus sophistiquées du protocole vecteur de distance ont été développées, et d'autres protocoles utilisant des mécanismes de base différents ont été inventés. Tous ces protocoles ont en commun de construire et tenir à jour automatiquement une table de routage similaire à celle ci-avant, et d'organiser l'aiguillage des paquets selon leur destination suivant les indications contenues dans la table de routage de chaque routeur. Cependant, si la table ne contient pas d'indications concernant la destination d'un paquet à transmettre, le protocole de routage abandonne et ne traite pas le paquet, qui est donc perdu. Un paquet peut également, dans certains cas pathologiques, « tourner en rond », si les tables de routage sont déréglées – de manière similaire à des panneaux routiers induisant en erreur. Dans ce cas aussi, la couche IP abandonnera le traitement de ce paquet – au bout d'un certain nombre d'aiguillages, quand la durée de vie du paquet sera devenue nulle – et le paquet sera perdu.

Pour diminuer la mémoire requise pour stocker les tables de routage, certains routeurs stockent une *route par défaut*, sous la forme d'une règle additionnelle s'apparentant à un panneau routier « toutes directions » indiquant un voisin qui, lui, saura aiguiller vers une destination qui n'est pas explicitement listée. Les plus gros routeurs du réseau, quant à eux, doivent avoir réponse à tout, et n'ont pas de route par défaut dans leur table de routage. Ces routeurs, notamment ceux qui sont au cœur d'Internet, doivent stocker une table de routage qui peut atteindre des centaines de milliers d'entrées, une taille qui n'est pas anodine à gérer. Pour diminuer la taille des tables de routage dans ces routeurs centraux, l'agrégation d'adresses et de préfixes IP est utilisée, comme décrit au paragraphe suivant.

Préfixes IP

Comme pour les codes postaux, les adresses IP sont organisées de manière hiérarchique. En effet, en comparant deux codes postaux, on peut en général en déduire leur proximité. Par exemple, deux adresses postales ayant des codes postaux débutant par les mêmes chiffres seront en général plus proches géographiquement que deux adresses ayant deux codes postaux débutant par des chiffres différents. De même, la proximité géographique de deux adresses IP est déterminée par leur similarité : plus précisément, deux adresses IP sont d'autant plus proches que la suite de bits qui les débute en commun est longue. Cette suite de bits initiaux en commun est appelée un *préfixe IP*. Un préfixe IP est utilisé pour indiquer l'ensemble des adresses IP qui commencent par la suite de bits définie par ce préfixe. Si une adresse IP fait partie de cet ensemble,

on dit que l'adresse est issue du préfixe IP. De manière similaire, on peut aussi extraire un préfixe IP d'un préfixe IP de base, le préfixe extrait étant défini par une suite de bits initiaux plus longue que celle définissant le préfixe de base, ce qui correspond à définir un sous-ensemble des adresses appartenant au préfixe de base.

En pratique, un préfixe IP donné est attribué à un lien correspondant, de la manière suivante : tout ordinateur connecté à ce lien est associé à une adresse issue du préfixe IP attribué à ce lien. De manière imbriquée, à l'échelle d'un agrégat de plusieurs liens interconnectés, on associe généralement un préfixe IP à l'agrégat, dont on extrait des préfixes IP plus longs, que l'on attribue chacun à l'un des liens de l'agrégat. On appelle ce procédé l'*agrégation d'adresses IP*.

Cette agrégation permet de réduire la taille des tables de routage. Ces dernières peuvent se contenter de lister un préfixe au lieu de lister chaque adresse correspondant à ce préfixe. En effet, vu que toutes les adresses issues de ce préfixe sont localisées en gros « au même endroit », elles ont donc en commun la même direction à prendre pour les atteindre.

Résolution d'adresse

Du point de vue de la couche réseau, la destination d'un paquet est un ordinateur connecté à Internet, identifié par une adresse IP, qui figure donc dans l'en-tête des paquets envoyés par la couche réseau. L'en-tête de ces paquets contient de plus l'adresse IP de l'ordinateur émetteur du paquet.

Du point de vue de la couche lien, cependant, la destination d'un paquet est un ordinateur connecté directement via un support physique commun, identifié par une adresse MAC utilisant un format différent du format des adresses IP. Vu que l'adresse IP d'un ordinateur peut changer au cours du temps – si l'ordinateur est déménagé, par exemple –, il n'y a pas de lien fixe entre l'adresse MAC correspondant à une adresse IP donnée.

Pour pouvoir résoudre l'adresse MAC correspondant à une adresse IP, il faut donc avoir recours à un mécanisme spécifique. Le plus couramment utilisé dans ce but est ARP (*Address Resolution Protocol*). Son principe est simple : quand un ordinateur demande à découvrir quelle adresse MAC correspond à une certaine adresse IP, il transmet un paquet spécial sur le lien, signalant au possesseur de l'adresse IP indiquée qu'il doit se manifester par « retour de courrier » indiquant son adresse MAC. Les ordinateurs connectés à ce lien entendent cette transmission et, si l'ordinateur utilisant l'adresse IP en question fonctionne normalement, il transmet un paquet réponse indiquant son adresse MAC, à l'intention de l'ordinateur ayant émis la demande. Quand ce dernier reçoit ce paquet réponse, il a résolu l'adresse IP. Il peut noter l'association entre

celle-ci et l'adresse MAC correspondante dans une table valide pour un certain laps de temps, afin d'éviter de devoir redécouvrir systématiquement cette association entre temps – les adresses IP changeant normalement relativement rarement. Au bout de ce laps de temps, cette association est effacée et il faut redemander au possesseur de l'adresse IP en question de se manifester.

Grâce au protocole ARP, la couche réseau peut procéder à la résolution d'adresse, à savoir, dans son cas, fournir un identifiant valide à la couche lien correspondant à l'ordinateur destinataire, à savoir une adresse MAC.

Internet, mais sans garantie

Le réseau de câbles et de liens radio connectant les ordinateurs entre eux peut dès à présent être considéré par l'émetteur d'un paquet comme une sorte de câble virtuel le connectant au destinataire du paquet, qui lui permet de communiquer avec la destination comme si un même câble les connectait directement, même si en réalité la connexion est indirecte, à travers des ordinateurs et des liens intermédiaires. C'est ce concept de câble virtuel que l'on appelle *Internet*. Dans la suite de ce chapitre, on appellera simplement *réseau* l'abstraction fournie par les protocoles de la couche réseau, assurant donc la transmission de paquets de données à travers une suite de liens.

On notera cependant que les transmissions de paquets sur le réseau ne sont pas garanties d'arriver à destination : comme nous l'avons vu, un paquet peut être perdu en chemin par la couche lien ou par la couche réseau.

La couche transport

Cette couche est constituée des protocoles responsables de la paquétisation des données à transmettre, ainsi que de la coordination entre l'envoyeur et le destinataire des paquets, pour assurer la fiabilité de leur transport de bout en bout.

Tout comme il y a un poids maximum autorisé pour une lettre que l'on poste, les paquets de données envoyés sur Internet ont une taille maximale autorisée – par exemple, la taille maximale autorisée pour un paquet Ethernet est de 1 500 octets, en général. Pour cela, lorsqu'une application requiert l'envoi vers une certaine destination d'un fichier qui ne tient pas en entier dans un seul paquet, il faut le fragmenter en petits bouts qui, individuellement, tiennent dans un paquet. On appelle ce procédé la *paquétisation*. Les fragments sont alors envoyés l'un après l'autre vers la destination, chacun dans son paquet individuel, à travers la couche réseau qui traitera alors ces paquets l'un après l'autre de manière indépendante. Plusieurs applications peuvent d'ailleurs utiliser en même temps les services de la couche transport, qui doit donc organiser l'équivalent de la levée et de la remise du courrier dans les boîtes aux lettres

individuelles qu'elle fournit à chaque application. On appelle une telle boîte aux lettres un *port* – noté par un nombre compris entre 0 et 65 535, codé sur 16 bits – et, dans ce contexte, on appelle *multiplexage* la levée du courrier et *démultiplexage* la distribution du courrier.

Avec le service postal de base, deux lettres envoyées en même temps du même endroit peuvent être reçues dans n'importe quel ordre par leur destinataire. Dans certains cas, une lettre peut même se perdre en route et ne pas arriver du tout. De manière similaire, pour un réseau, une des conséquences notables du procédé « par paquet » est que deux paquets contenant l'information constituant un seul et même fichier à l'origine peuvent arriver dans n'importe quel ordre. Certaines fois, un paquet de données peut également se perdre en chemin et ne pas arriver du tout.

Pour pallier ces problèmes potentiels, la couche transport propose des services d'accusé de réception des paquets, et de remise en ordre des paquets reçus conformément à l'ordre dans lequel ils ont été émis. Ces services sont fournis par le protocole TCP (*Transmission Control Protocol*), qui tient un journal par flux de paquets envoyés vers une même destination ; on appelle un tel journal une *connexion TCP*. Le protocole TCP ajoute un en-tête spécial à chaque paquet émis dans un tel flux, contenant notamment un numéro de séquence unique dans le contexte de cette connexion TCP. Les numéros de séquence vont croissant d'une unité pour chaque nouveau paquet envoyé vers la destination, qui, lorsqu'elle le reçoit, envoie un accusé de réception à l'émetteur, mentionnant le numéro de séquence du paquet, et ainsi de suite. Cela permet de réordonner les paquets à la réception en suivant les numéros de séquences croissants de l'en-tête TCP, au cas où ils ne seraient pas arrivés dans l'ordre. Cela permet également à l'émetteur de s'assurer que les paquets envoyés sont arrivés à destination, en vérifiant qu'un acquittement a bien été reçu pour chaque paquet envoyé, mentionnant le numéro de séquence correspondant à ce paquet. Si un acquittement n'a pas été reçu pour un paquet envoyé, l'émetteur le considère comme perdu et l'envoie de nouveau, en espérant recevoir cette fois un acquittement.

Certaines applications n'ont néanmoins pas besoin de tous les services proposés par le protocole TCP. Par exemple, du streaming audio ou vidéo n'a en général pas besoin des services d'accusé de réception : dans ce contexte, il vaut mieux envoyer de l'image/son actualisé que de renvoyer de l'image/son datant d'il y a quelques secondes. En effet, une certaine perte de données est acceptable dans la mesure où la perception humaine est capable de la compenser, tandis qu'attendre la retransmission de paquets perdus cause des arrêts à répétition qui deviennent vite insupportables.

Pour les applications qui n'ont pas besoin de tous les services proposés par TCP, il existe un protocole alternatif *UDP* (*User Datagram Protocol*), qui fournit seulement l'équivalent du service postal de base – à savoir le multiplexage et le démultiplexage – aux applications souhaitant utiliser le réseau, en laissant soin à ces applications d'assurer à leur manière tout service supplémentaire.

Identifiants utilisés à la couche transport et résolution d'adresse

Du point de vue de la couche transport, la destination d'un paquet est identifiée par un numéro de port correspondant à l'application destinataire associé à une adresse IP correspondant à l'ordinateur hébergeant cette application. Ces éléments figurent donc dans l'en-tête des paquets envoyés par la couche transport, que ce soit via TCP ou via UDP. L'en-tête de ces paquets contient de plus le numéro de port de l'application émettrice du paquet ainsi que l'adresse IP de l'ordinateur sur laquelle elle s'exécute.

Du point de vue de la couche réseau, la destination d'un paquet est uniquement identifiée par l'adresse IP correspondant à cet ordinateur. La résolution d'adresse est donc très simple à ce stade : il suffit d'extraire l'adresse IP contenue dans l'identifiant utilisé dans l'en-tête de la couche transport.

Contrôle de congestion

Nous avons réussi : Internet est un réseau fiable ! Des programmes sur un ordinateur peuvent ainsi l'utiliser pour envoyer des données binaires à d'autres programmes sur un autre ordinateur, comme si ce dernier et le premier étaient directement connectés par un même câble, même si en réalité la connexion est indirecte à travers plusieurs liens et plusieurs autres ordinateurs. Cependant, pour gérer les variations du nombre d'utilisateurs accédant à Internet en même temps, un mécanisme équivalant à ceux gérant le « temps de parole » vus au paragraphe « La couche lien » est nécessaire, à l'échelle non plus d'un seul câble, mais d'Internet tout entier, qui peut être considéré à ce stade comme un immense câble virtuel – voir le paragraphe « La couche réseau ».

En plus des services que nous avons vus, le protocole TCP fournit dans ce but un service supplémentaire : l'ajustement du rythme auquel une application envoie un flux de paquets vers une destination donnée. Le but de cet ajustement est de trouver un compromis entre l'intérêt individuel de l'application – envoyer ses paquets le plus rapidement possible pour qu'ils arrivent le plus vite possible – et l'intérêt général de toutes les autres applications utilisant le réseau, à savoir : que ce dernier soit utilisable par tous, et non pas monopolisé par certains. Du point de vue d'une application, Internet est un câble virtuel connectant l'ordinateur sur lequel elle s'exécute avec les ordinateurs hébergeant

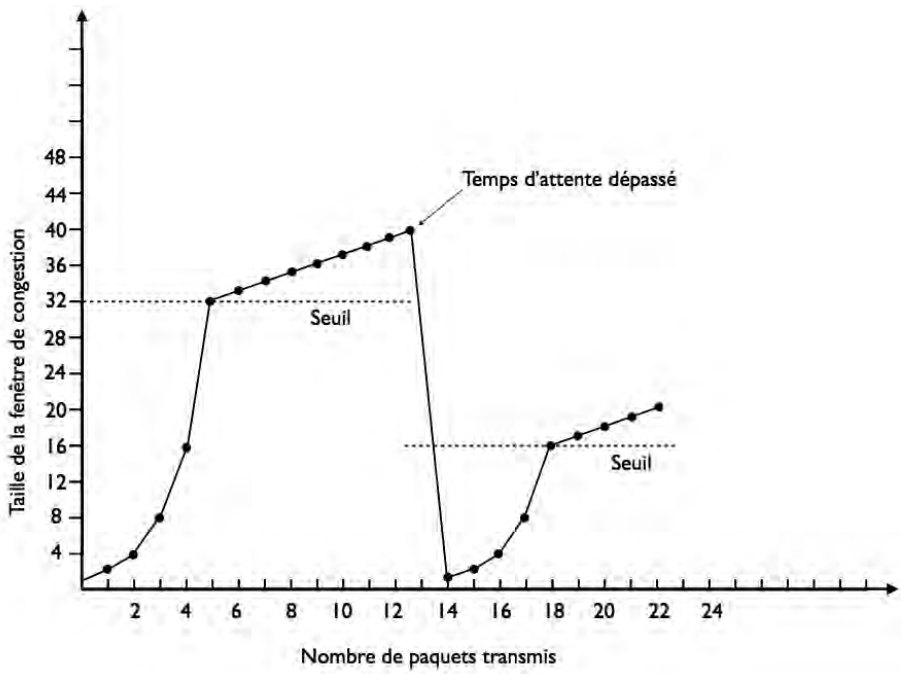
les applications destinataires des données à envoyer. Il lui faut donc découvrir les capacités de transmission vers telle ou telle destination, ces capacités n'étant pas connues à l'avance car elles dépendent de plusieurs paramètres, dont l'état d'engorgement du réseau, qui peut varier de manière extrême d'une seconde à l'autre. La capacité de transmission vers une certaine destination dépend également du chemin utilisé entre l'émetteur et la destination : si, pour aller de *A* à *B*, par exemple, le câble virtuel est composé d'une suite de supports physiques haute capacité, transportant peu de trafic en ce moment, on pourrait potentiellement envoyer tout de suite beaucoup de trafic de *A* à *B*, et ce sans congestion. Si, en revanche, pour aller de *A* à *C* on doit passer par une série de liens, dont un lien radio à très faible capacité, on ne pourra envoyer les paquets de *A* vers *C* qu'au compte-gouttes. Un autre paramètre dont dépend la capacité de transmission vers une certaine destination est la mémoire actuellement disponible sur chaque ordinateur le long du chemin vers cette destination, dédiée à stocker les paquets qui arrivent souvent plus vite que l'on ne peut les traiter. Cette mémoire est appelée mémoire tampon, et si cette dernière est soudainement saturée par trop de trafic sur l'un des ordinateurs intermédiaires ou sur l'ordinateur destinataire, les paquets suivants sont perdus.

Coordonner de manière centralisée l'accès au câble virtuel qu'est Internet souffrirait en principe des défauts de la rigidité évoquée au paragraphe « La couche lien » pour l'équivalent à la couche lien. Mais, en pratique, ce n'est de toute façon pas envisageable : la gestion en temps réel de centaines de millions d'utilisateurs, bientôt de milliards de machines, et déjà de milliers de milliards de paquets acheminés par seconde, ne peut tout simplement pas être concentrée en un seul point.

Une solution distribuée est donc utilisée par le protocole TCP pour ajuster le rythme auquel une application envoie un flux de paquets vers une destination donnée. Le principe de base est simple, il se fonde sur trois paramètres évalués localement par l'émetteur et le récepteur pour une connexion TCP donnée. Le premier paramètre est *W*, le nombre de paquets que l'émetteur tolère avoir envoyés vers la destination sans encore avoir reçu d'acquittement les concernant. On appelle ce paramètre la *fenêtre de congestion*. TCP fait varier *W* dans le temps au gré des détections de pertes de paquets. Le deuxième paramètre est *S*, le seuil au-delà duquel, d'après son expérience, l'émetteur considère que *W* ne devrait s'aventurer que très prudemment. TCP fait aussi varier *S* dans le temps au gré des détections de pertes de paquets. Le dernier paramètre est *Tw*, une estimation du laps de temps au bout duquel on devrait avoir reçu les acquittements correspondant à *W* paquets envoyés. Un échange d'amorçage entre l'émetteur et la destination permet d'estimer la valeur de *Tw*, et initialement $W = 1$ et $S = 64$.

L'émetteur répète la procédure suivante, appelée procédure de fenêtre glissante. Dans une première phase, il envoie W paquets que l'on appelle la fenêtre en cours, et attend qu'ils soient tous acquittés. Si cela arrive avant Tw unités de temps, la valeur de W est doublée. L'émetteur envoie alors les W paquets suivants qui deviennent la nouvelle fenêtre en cours et attend leurs acquittements, ainsi de suite jusqu'à ce que $W = S$. Cette phase est appelée *début lent*.

À partir du seuil $W = S$, une deuxième phase s'enclenche, au cours de laquelle, à chaque itération, l'émetteur augmente W d'une unité – au lieu de doubler sa valeur. Cependant, à tout moment au cours de la procédure, si l'émetteur doit attendre plus de Tw unités de temps avant d'avoir reçu tous les acquittements de la fenêtre en cours, il considère que des paquets ont été perdus à cause d'une congestion du réseau. En conséquence, il ajuste S en divisant sa valeur par deux, réinitialise $W = 1$ et recommence depuis le début : doublement de W à chaque itération si la fenêtre en cours est acquittée à temps, jusqu'au seuil $W = S$ à partir duquel W est augmenté d'une unité par itération si la fenêtre en cours est acquittée à temps. Ce mécanisme peut s'illustrer ainsi.



TCP. Ajustements progressifs de la cadence d'envoi des paquets via les variations de la fenêtre de congestion.

Grâce à ce mécanisme d'ajustements progressifs de la cadence d'envoi des paquets, appelé contrôle de congestion, une application utilisant TCP peut s'adapter en temps réel à ce qu'Internet peut lui fournir comme débit utile instantané pour envoyer un flux de paquets de données. Des versions plus sophistiquées de ce mécanisme sont actuellement utilisées sur Internet, reposant sur le même principe, qui permet à un nombre arbitraire d'ordinateurs et d'applications simultanées d'adapter automatiquement leur flux de données à des débits utiles variant potentiellement de plusieurs ordres de grandeur. Beaucoup pensent que ce mécanisme, joint au format universel IP, sont les deux principes fondamentaux qui ont permis l'essor d'Internet à l'échelle planétaire, et c'est pourquoi on nomme la pile de protocoles décrite dans ce chapitre la « pile TCP/IP ».

La couche application

Le but de la communication entre ordinateurs est de permettre la transmission de données entre applications. Les applications ne sont pas dédiées à faire fonctionner le réseau et ne sont donc pas abordées en détail dans ce chapitre, si ce n'est pour ce qui concerne la résolution d'adresse.

Identifiants utilisés à la couche application et résolution d'adresse

Pour s'adapter à leurs utilisateurs humains, la couche application utilise en guise d'identifiants un système hiérarchique de noms tels que *siteweb.com* ou *service.example.fr*, que les gens peuvent mémoriser facilement et utiliser tous les jours pour se connecter à tel ou tel site depuis leur ordinateur, à travers une application telle qu'un navigateur, par exemple. Le site en question est en réalité une autre application, hébergée sur un autre ordinateur, qui doit donc être contactée via la couche transport, pour pouvoir consulter le site.

Cependant, du point de vue de la couche transport, les applications s'exécutant sur un ordinateur distant sont identifiées par un numéro de port d'une part – voir le paragraphe « La couche transport » – et une adresse IP d'autre part – voir le paragraphe « La couche réseau ». Les ports identifiant l'application émettrice et l'application destinataire sont connus à l'avance par convention et peuvent dont être directement renseignés dans l'en-tête des paquets envoyés via la couche transport – voir le paragraphe « La couche transport ». L'application HTTP (*HyperText Transfer Protocol*), utilisée pour transférer des pages web, a par exemple pour numéro de port attribué le numéro 80. C'est d'ailleurs le système hypertexte constitué des pages reliées par HTTP que l'on appelle le Web, la Toile, ou encore WWW (*World Wide Web*). Cette dernière

dénomination apparaît explicitement dans les adresses web de la forme *http://www.siteweb.com* utilisées par des applications tels les navigateurs.

En revanche, pour permettre plus de flexibilité, l'adresse IP de l'ordinateur qui héberge *siteweb.com* n'est pas fixée par convention et peut changer au cours du temps. Cet ordinateur pourrait par exemple déménager, ou l'application en question migrer vers un autre ordinateur. Ces cas sont traités de la même manière que pour les adresses postales : si une personne déménage, elle peut ne pas changer de nom, mais il lui faut tout de même changer d'adresse.

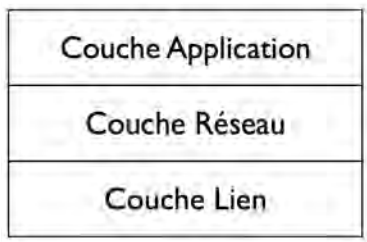
Pour pouvoir associer à une adresse web, comme *siteweb.com*, l'adresse IP de l'ordinateur l'hébergeant, il faut utiliser un mécanisme pour accéder et tenir à jour l'équivalent d'un annuaire, qui associe automatiquement à une adresse web pérenne – par exemple, *exemple.eu* – une adresse IP qui peut donc, elle, changer au cours du temps comme une personne peut changer de numéro de téléphone. Ce mécanisme est fourni par DNS (*Domain Name System*), une application qui maintient des copies d'un tel annuaire à différents endroits du réseau connus sous le nom de serveurs DNS, chacun localisé sur un ordinateur identifié par une adresse IP fixée à l'avance, et qui peuvent donc être consultés via Internet par d'autres applications sur d'autres ordinateurs cherchant à résoudre l'adresse d'un nom – *exemple.eu* dans le cas évoqué ici. L'application DNS a pour numéro de port attribué le numéro 53.

Grâce au DNS et aux numéros de port connus par convention, la couche application peut procéder à la résolution d'adresse, à savoir, dans son cas, fournir un identifiant valide à la couche réseau correspondant à l'application destinataire et à l'ordinateur sur laquelle cette dernière s'exécute, respectivement un numéro de port et une adresse IP.

Exercices corrigés et commentés

Exercice 1

En supposant qu'une pile de protocoles soit conçue de telle façon qu'il n'y ait pas de couche de transport dédiée, mais seulement trois couches



quelle affirmation parmi les suivantes serait correcte ? Une seule réponse possible.

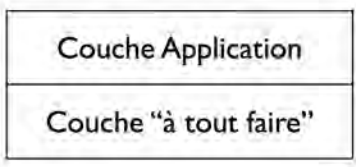
1. La pile de protocoles requerrait que tous les liens soient 100 % fiables : aucune perte et aucune corruption de données.
2. Toutes les machines devraient être connectées par un seul et même support physique, car aucun routage ne serait possible.
3. Seule une application réseau par machine pourrait tourner à la fois.
4. Toutes les affirmations ci-avant.
5. Aucune des affirmations ci-avant.

Correction. La première affirmation est fausse, car il est possible pour chaque application d'inventer son propre mécanisme de transport fiable. La deuxième affirmation est également fausse, car établir la connectivité entre machines à travers le réseau est le rôle de la couche réseau, présente dans la pile.

La troisième affirmation est en revanche correcte. Sur Internet, les adresses IP sont utilisées pour identifier un ordinateur destinataire des paquets de données, tandis que les couches transport y ajoutent la notion de port pour identifier l'application destinataire sur cet ordinateur. Sans cette notion et le multiplexage/démultiplexage qui en découle entre la couche réseau et la couche transport, seule une application à la fois pourrait utiliser le réseau, par ordinateur.

Exercice 2

À chaque fois qu'un paquet passe d'une couche à la couche directement inférieure, des informations supplémentaires sont ajoutées au moyen d'un en-tête. Ainsi, dans une pile de protocoles avec 4 couches, 3 en-têtes sont ajoutés – en-tête transport, en-tête réseau et en-tête lien. Supposons maintenant que l'on nous propose la pile de protocoles alternative illustrée ci-après, en argumentant qu'elle serait plus efficace car possédant moins de couches et donc ajouterait moins de données supplémentaires à transmettre, tout en fournissant exactement les mêmes services.



Quelle affirmation parmi les suivantes serait correcte ? Une seule réponse possible.

1. Cette proposition réduit en effet le nombre de données à transmettre.
2. Cette proposition ne réduit pas le nombre de données à transmettre.

3. Toutes les affirmations ci-avant.
4. Aucune des affirmations ci-avant.

Correction. L'ambition de cette pile de protocoles est de réduire le « coût » supplémentaire introduit par les en-têtes ajoutés à chaque couche. Néanmoins, si cette nouvelle couche « à tout faire » doit fournir des fonctionnalités équivalentes, comme par exemple un transport fiable de bout en bout comme fourni par la couche transport, alors la nouvelle couche devrait ajouter dans son en-tête la même information que celle utilisée par la couche transport. Même argument pour chacune des couches réseau et lien. Au final, le nouvel en-tête ne serait pas différent d'un en-tête regroupant transport + réseau + liaison.

Cette pile alternative aurait d'ailleurs des inconvénients : chaque couche fournit un service particulier, via une interface standard bien définie. De cette manière, avec la pile de protocoles TCP/IP, on peut remplacer les mécanismes d'une couche sans avoir à changer les mécanismes des autres couches. Par exemple, remplacer Wifi par Ethernet à la couche lien est possible et très simple : pas besoin de changer quoi que ce soit aux autres couches. *A contrario*, avec la pile alternative proposée ci-avant, un tel changement s'avérerait complexe.

Exercice 3

Une application utilisant UDP peut-elle bénéficier d'un transfert de données fiable ? Une seule réponse possible.

1. Non.
2. Oui : mais seulement si les liens entre ordinateurs sont garantis fiables.
3. Oui : mais seulement si TCP est modifié pour s'exécuter au-dessus de UDP.
4. Oui : l'application peut fournir son propre mécanisme de transport fiable.
5. Toutes les affirmations ci-avant.
6. Aucune des affirmations ci-avant.

Correction. UDP est un protocole de transport qui ne garantit pas leur transport fiable de bout en bout : ainsi des paquets peuvent être perdus en transit. Néanmoins, il est tout à fait possible pour une application d'utiliser UDP tout en bénéficiant d'un transport fiable : il suffit simplement que cette application fournisse son propre mécanisme de transport fiable. Ce pourrait être un mécanisme de transport similaire à TCP au moyen de fenêtres glissantes, ou au moyen d'un mécanisme simplifié du même genre, comme par exemple : envoyer un paquet, attendre un accusé de réception avant d'envoyer le paquet suivant, si pas d'accusé retransmettre le paquet.

La quatrième réponse est donc la bonne. La troisième est fausse, puisqu'il existe d'autres options de transport fiable que TCP. Enfin, même si tous les

liens sont garantis fiables individuellement, un paquet peut quand même être perdu en route. Par exemple, lorsqu'un paquet arrive à un routeur et que celui-ci cesse de fonctionner juste avant de retransmettre le paquet, ce dernier est perdu. De même, si les tables de routage sont temporairement déréglées, un paquet peut suivre un chemin qui ne mène nulle part. La deuxième réponse est donc également incorrecte.

Exercice 4

En supposant que chaque lien soit 100 % fiable – pas de perte ni de corruption de données –, quelle affirmation parmi les suivantes est correcte ? Une seule réponse possible.

1. TCP serait complètement redondant.
2. Le service de livraison fiable des paquets et le mécanisme de contrôle de congestion de TCP seraient redondants.
3. Le service de livraison fiable des paquets de TCP serait redondant, mais pas le mécanisme de contrôle de congestion.
4. Le mécanisme de contrôle de congestion de TCP serait redondant, mais pas le service de livraison fiable des paquets.
5. Toutes les affirmations ci-avant.
6. Aucune des affirmations ci-avant.

Correction. Rappelons que TCP fournit plusieurs services, dont : la livraison fiable des paquets, livraison des paquets dans leur ordre d'émission, le contrôle de congestion.

Si tous les liens sur Internet étaient fiables, cela n'empêcherait pas deux paquets envoyés coup sur coup de A à B d'arriver dans un ordre différent de celui dans lequel ils ont été envoyés. En effet, rien ne garantit que les deux paquets n'arrivent pas via des chemins différents à travers le réseau, ou que, pour quelque raison que ce soit, un routeur décide de réordonner les paquets avant leur retransmission. Ainsi, la première réponse est fausse, car le service de livraison des paquets dans leur ordre d'émission fourni par TCP n'est pas redondant.

De plus, le fait que les liens soient fiables ne garantit pas qu'un paquet ne soit pas perdu à cause de tables de routage déréglées, ou d'un routeur cessant soudainement de fonctionner. Donc, le service de livraison fiable des paquets fourni par TCP n'est pas redondant. On en déduit que la deuxième et la troisième réponses sont incorrectes.

Finalement, le fait que les liens soient fiables ne garantit pas que les mémoires tampons disponibles sur les ordinateurs le long du parcours d'un paquet ne soient pas saturées, auquel cas ce paquet, n'ayant pas été traité, est perdu. TCP ne garantit pas non plus que ces mémoires ne soient pas saturées, mais

diminue le risque qu'elles le soient grâce au service de contrôle de congestion qu'elle propose. Ce service n'est donc en aucun cas redondant. On en déduit que la quatrième et la cinquième réponses sont fausses et que la dernière est correcte

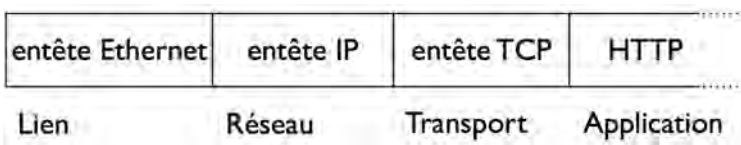
Ainsi, la livraison des paquets dans l'ordre est bien assurée par TCP, qui temporise les paquets reçus jusqu'à pouvoir les délivrer dans le bon ordre à la couche application. Le mécanisme de contrôle de flux de TCP est tout aussi nécessaire, car il permet à l'émetteur des paquets de ne pas saturer le tampon de réception du destinataire. Le contrôle de congestion est quant à lui nécessaire pour permettre à l'émetteur d'adapter sa cadence de transmission des données de sorte à ne pas saturer les tampons de réception des routeurs intermédiaires. Enfin, même si tous les liens sont garantis fiables, ce ne sont pas les seuls endroits où les données peuvent être perdues. Par exemple, lorsqu'un paquet arrive sur un routeur et que celui-ci crashe juste avant de retransmettre le paquet, ce dernier est perdu. Ainsi, un mécanisme de livraison fiable des données est tout de même nécessaire. En conclusion, des liens 100 % fiables sur Internet ne rendent nullement TCP redondant.

Exercice 5

Un ordinateur est connecté à un lien Ethernet – un câble – et utilise TCP/IP, et l'application HTTP. Quel est le premier octet du contenu d'un paquet envoyé par cet ordinateur sur le câble ? Une seule réponse possible.

1. Le premier octet de l'en-tête TCP.
2. Le premier octet des données HTTP.
3. Le premier octet de l'en-tête Ethernet.
4. Toutes les affirmations ci-avant.
5. Aucune des affirmations ci-avant.

Correction. Dans le contexte décrit par la question, HTTP est un protocole de la couche application, utilisant TCP comme transport. Les paquets TCP sont envoyés via IP du point de vue de la couche réseau, et via Ethernet du point de vue de la couche lien – sur le câble. Ainsi le paquet résultant, envoyé par l'ordinateur sur le câble, aura l'allure suivante :



Le premier octet dans le contenu de ce paquet est donc le premier octet du contenu du paquet du point de vue d'Ethernet, c'est-à-dire le premier octet de l'en-tête IP. Donc, toutes les réponses sont incorrectes, sauf la dernière.

Exercice 6

La phase dite de *début lent* de TCP est en fait une période relativement courte, pourquoi ? Une seule réponse possible.

1. Chaque fenêtre de paquets acquittée augmente la taille de fenêtre de congestion, qui croît linéairement et atteint vite le seuil au-delà duquel on passe à la phase suivante du mécanisme de contrôle de congestion de TCP.

2. Chaque fenêtre de paquets acquittée augmente la taille de fenêtre de congestion, qui croît exponentiellement et atteint vite le seuil au-delà duquel on passe à la phase suivante du mécanisme de contrôle de congestion de TCP.

3. Chaque acquittement reçu double la taille de fenêtre, qui croît exponentiellement et atteint vite le seuil au-delà duquel on passe à la phase suivante du mécanisme de contrôle de congestion de TCP.

4. Aucune des affirmations ci-avant.

Correction. La dénomination *début lent* est presque un abus de terminologie pour désigner la phase initiale de TCP. Certes, pendant cette phase, TCP démarre avec une cadence de transmission « lente », mais la cadence augmente exponentiellement, lorsque chaque fenêtre de paquets acquittée double la taille de la fenêtre de congestion, et donc la cadence d'envoi de paquets. Cette croissance exponentielle atteint rapidement le plafond fixé, à partir duquel la croissance est rendue linéaire, plus prudente. La bonne réponse est donc la deuxième. Rappelons qu'à tout moment, si un paquet est perdu, TCP interprète cette perte comme un signe de congestion du réseau et diminue la cadence des envois des paquets : le plafond est divisé par deux et TCP repart à la phase *début lent* avec une fenêtre de congestion minimale, à savoir égale à un. Ce mécanisme permet de s'adapter automatiquement aux capacités intrinsèques des ordinateurs et liens traversés et en temps réel aux conditions de trafic rencontrées.

Exercice 7

Imaginons une couche lien au sein de laquelle le protocole MAC se résume simplement à « s'il y a quelque chose à transmettre, transmettre immédiatement même si le support physique est déjà occupé », sans faire aucun effort pour découvrir et réparer les collisions. Imaginons de plus que TCP est le protocole de transport utilisé. Quel serait l'impact de cette couche lien sur les autres couches ? Une seule réponse possible.

1. Aucun impact sur aucune couche, excepté sur la couche liaison.

2. Les applications n'auraient d'autre choix que de se préparer à des pertes et corruptions massives de données.

3. Aucun impact sur les applications, car TCP serait en mesure de gérer pertes et retransmissions, et ainsi de rendre le tout transparent pour les applications.

4. Les applications rencontreraient une baisse considérable des débits, car TCP serait obligé d'effectuer de nombreuses retransmissions pour chaque paquet.

5. Les applications rencontreraient une baisse considérable des débits car TCP interpréterait les pertes de données comme une congestion et réduirait la cadence d'envoi des paquets.

6. Toutes les affirmations ci-avant.

7. Aucune des affirmations ci-avant.

Correction. TCP est capable de gérer les pertes de données en assurant les retransmissions des paquets perdus, donc logiquement aucune couche ne devrait être touchée de manière fonctionnelle. Mais, en termes de performance, c'est une tout autre histoire. Rappelons que TCP interprète la perte de données comme un signal que le réseau est congestionné et en conséquence diminue la cadence d'envoi des paquets : le plafond est divisé par deux et TCP repart à la phase *début lent* avec une fenêtre de congestion minimale, à savoir égale à un. Dans un réseau où la couche lien ne ferait aucun effort pour découvrir et réparer les collisions, le taux de perte de données serait énorme et par conséquent TCP aurait bien du mal à sortir ne serait-ce que de la phase *début lent*. Les paquets provenant de l'émetteur risquent même d'entrer en collision avec les accusés de réception envoyés par le destinataire. Résultat : suite à des pertes de paquets, TCP serait fréquemment forcé de « redémarrer » et ne serait donc jamais amené à atteindre ni de près ni de loin le débit maximum faisable. Ainsi, c'est bien le mécanisme de contrôle de congestion et non pas de retransmission des paquets de TCP qui est à l'origine de la réduction de débit rencontrée par les applications. La cinquième réponse est donc la bonne.

Exercice 8

Dans un réseau IP – donc sur Internet –, un routeur qui n'a pas dans sa table de routage une destination explicite pour un paquet reçu va :

1. rediriger le paquet vers un routeur par défaut, si une route par défaut est listée dans sa table de routage ;

2. éliminer le paquet, s'il n'y a pas de route par défaut listée dans sa table de routage ;

3. mettre le paquet en attente pour retransmission ultérieure ;

4. renvoyer le paquet vers le voisin qui lui a transmis le paquet ;

5. renvoyer le paquet vers l'émetteur du paquet.

6. Aucune des affirmations ci-avant.

Correction. Les réseaux IP sont dits « sans garanties », dans le sens où, lorsqu'un paquet est prêt à être transmis, il est soit transmis à un ordinateur le rapprochant de sa destination, soit éliminé. La taille des mémoires est un paramètre critique pour les routeurs, et c'est pour cette raison que les paquets ne

pouvant être acheminés tout de suite sont éliminés et non stockés. Donc, si le routeur n'a pas de route explicite vers la destination d'un paquet, ce paquet est éliminé ou envoyé vers un voisin listé comme « toutes directions » si une telle route par défaut est présente dans sa table de routage. Donc, les deux premières réponses sont correctes et les quatre autres sont incorrectes.

Renvoyer un paquet à rebrousse-chemin au voisin qui l'a transmis ou à l'ordinateur qui l'a émis au départ augmenterait la quantité de données échangées à travers le réseau sans bénéfice évident : les ordinateurs en amont du chemin ne disposent probablement pas de chemins alternatifs et ne feront qu'essayer de délivrer le paquet à nouveau par le même chemin, créant ainsi une boucle dans laquelle le même paquet tournerait en rond en passant toujours par les mêmes routeurs sans jamais s'approcher de la destination finale. On en déduit que les trois dernières réponses sont fausses.

Exercices non corrigés

Ces exercices utilisent l'algorithme de routage suivant, basé sur l'algorithme de Bellman-Ford :

Définitions

– Soit u le routeur exécutant l'algorithme, et V l'ensemble des routeurs dans le réseau.

– Soit $distance(w, v)$ la distance la plus courte – en nombre de liens intermédiaires –, connue jusqu'à présent, de u à w via un voisin v de u.

– Soit $distance(w, *)$ la distance la plus courte – en nombre de liens intermédiaires –, connue jusqu'à présent, de u à w, via n'importe quel voisin de u.

– Soit $c(u, v)$ telle que $c(u,v) = 1$ si un lien direct existe entre u et v, et $c(u,v) = \infty$ si aucun lien direct n'existe entre u et v.

Initialisation

1. $\forall (v,w) \in V : distance(w, v) = \infty$

C'est-à-dire, le routeur u ne connaît rien du réseau initialement.

2. $\forall v \in V \,|\, c(u,v) \neq \infty : distance(v, v) = c(u, v)$

Le routeur u enregistre la distance entre lui et ses voisins, c'est-à-dire les routeurs avec lesquels il a un lien direct – via un câble par exemple –, et note intuitivement le fait que, pour atteindre un voisin, il faut utiliser comme prochaine étape le voisin en question.

3. $\forall w \in V \setminus \{u\} \,|\, distance(w, *) \neq \infty$: envoyer $(w, distance(w, *))$ à chaque voisin.

Le routeur u détermine la plus courte distance qu'il connaît entre lui et chaque destination dans le réseau. L'ensemble des paires – destination, distance la plus courte vers cette destination – ainsi calculées par le routeur constitue le

vecteur de distance de ce routeur. Le routeur u envoie son vecteur de distance à chacun de ses voisins. Ces derniers connaissent désormais la distance entre u et toutes les destinations du réseau.

Répéter à l'infini

1. Attendre l'un des événements suivants :

– réception d'un vecteur de distance émis par un des voisins de u.

– détection d'un changement de coût d'un lien avec voisin v.

2. Si le coût d'un lien entre u et v change de $c(u, v)$ à $c'(u, v)$ – *par exemple le coût devient* ∞ *quand un lien disparaît* –, alors :

– $\forall w' \in V$: $distance(w', v) = distance(w', v) + (c'(u, v) - c(u, v))$

Toutes les distances calculées via v sont mises à jour pour refléter le nouveau coût du lien $c'(u, v)$.

3. Si un vecteur de distance est reçu d'un routeur voisin v, déclarant $(w, distance(w,*))$, alors :

– $distance(w, v) = distance(v,*) + distance(w, v)$

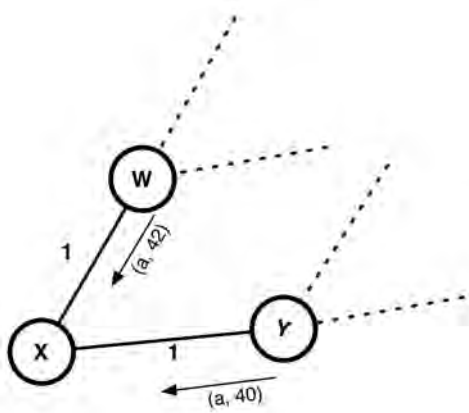
4. Si n'importe quelle distance $distance(w, *)$ a changé :

– $\forall w \in V \setminus \{u\} | distance(w, *) \neq \infty$: envoyer $(w, distance(w, *))$ à tous les voisins de u.

Si le routeur détermine que son vecteur de distance a changé, alors il renvoie son nouveau vecteur de distance à chacun de ses voisins.

Exercice 1

Considérons le réseau



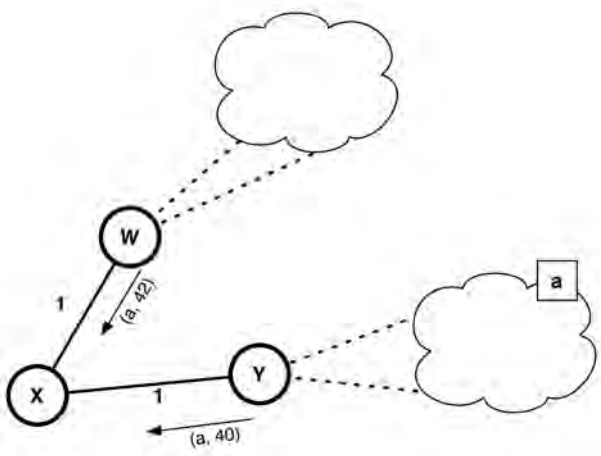
Seuls trois routeurs sont représentés et les pointillés indiquent qu'ils sont reliés à un réseau plus grand. Les liens entre les routeurs X, W et Y sont indiqués par des traits pleins. Le routeur X reçoit un message avec le vecteur de distance de Y indiquant $(a, 40)$, et un message de W indiquant $(a, 42)$. Il n'y a pas d'autres

liens partant de X que ceux indiqués sur la figure. Quelle affirmation parmi les suivantes est correcte pour le réseau ainsi représenté ? Une seule réponse possible.

1. Le routeur X va enregistrer Y comme la prochaine étape sur le chemin le plus court vers a, avec un coût de 40.
2. Le routeur X va enregistrer Y comme la prochaine étape sur le chemin le plus court vers a, avec un coût de 41.
3. Le routeur W a enregistré X comme la prochaine étape sur le chemin le plus court vers a, avec un coût de 42.
4. Le routeur W a enregistré a comme la prochaine étape sur le chemin le plus court vers Y, avec un coût de 43.
5. Toutes les affirmations ci-avant.
6. Aucune des affirmations ci-avant.

Exercice 2

Considérons le réseau



Seuls 3 routeurs sont représentés et les pointillés indiquent que les routeurs W et Y sont chacun connectés à un réseau distinct plus grand. Les liens entre les routeurs X, W et Y sont indiqués par des traits pleins. De plus, le routeur X reçoit un vecteur de distance de Y indiquant $(a, 40)$ et un de W indiquant $(a, 42)$. Notons que la destination a est atteignable uniquement via Y. Supposons maintenant que le lien entre le routeur Y et le routeur X est soudainement rompu, de manière permanente, c'est-à-dire $c(X, Y) = \infty$. Quelle affirmation est correcte parmi les suivantes ? Une seule réponse possible.

1. Le routeur X va découvrir que le coût de tous les chemins via le routeur Y est maintenant ∞, et par conséquent ignorer tous les chemins passant par Y.

2. Le routeur X va immédiatement découvrir que a n'est plus atteignable et le fait savoir à W en lui envoyant un message indiquant que la distance à a via X est ∞.

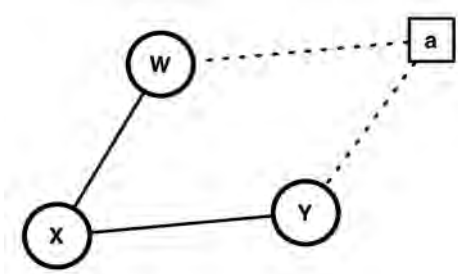
3. Le routeur X va immédiatement découvrir que a n'est plus atteignable et le fait savoir à W en lui envoyant un message indiquant que la distance à a via X est ∞. Le routeur W propage ensuite immédiatement cette information aux autres liens auxquels il est connecté.

4. Toutes les affirmations ci-avant.

5. Aucune des affirmations ci-avant.

Exercice 3

La version simplifiée du protocole vecteur de distance vue jusqu'ici suppose initialement une distance $c(u,v) = 1$ si un lien direct existe entre u et v, et sinon $c(u, v) = \infty$. Une version plus élaborée pourrait prendre en compte des valeurs initiales plus variées, à savoir : $c(u, v) \in [1; \infty[$ si un lien direct existe entre u et v, et sinon $c(u, v) = \infty$. Une version encore plus élaborée pourrait prendre en compte des variations de coût par lien, $c(u, v)$ qui pourrait par exemple augmenter proportionnellement en fonction de la quantité de trafic qu'il véhicule – dans le but d'encourager l'équilibre de la charge de chaque lien dans le réseau. Dans le réseau décrit ci-après, le but d'une telle élaboration serait d'envoyer le trafic de X vers a à la fois via $X - W - a$ et via $X - Y - a$ en parallèle, pour maximiser le débit du flux entre X et a.



Dans ce cas, quelle affirmation est correcte parmi les suivantes ? Une seule réponse possible.

1. Les tables de routage pourraient ne jamais converger, à savoir atteindre un état stable et cohérent entre elles.

2. Le trafic n'utilisera pas les deux chemins en parallèle, mais l'un ou l'autre, en tout cas pas les deux.

3. Le trafic n'utilisera pas véritablement les deux chemins en parallèle, mais alternera indéfiniment entre les deux.

4. Toutes les affirmations ci-avant.
5. Aucune des affirmations ci-avant.

Exercice 4

Un exemple d'application simple est TELNET (*TErminaL NETwork*), qui offre un moyen de communication bidirectionnel généraliste entre deux machines distantes. Ouvrir un terminal et exécuter l'application TELNET via la ligne de commande, en tapant *telnet www.google.com 80*, puis taper sur la touche ENTER. Il s'affiche le texte suivant :

```
Trying 74.125.230.80…
Connected to www.l.google.com.
Escape character is '^]'.
```

Quelle affirmation est correcte parmi les suivantes ? Une seule réponse possible.

1. *TELNET* est une application qui accède au réseau via la couche transport sur l'ordinateur.
2. L'adresse IP74.125.230.80 est l'adresse qui correspond au site *www.google.com*, trouvée grâce au DNS.
3. Dans la commande *telnet www.google.com 80*, le nombre 80 est le numéro de port visé sur l'ordinateur destinataire.
4. Le protocole de transport utilisé par TELNET est TCP.
5. Toutes les affirmations ci-avant.
6. Aucune des affirmations ci-avant.

Questions d'enseignement

L'Internet d'aujourd'hui, basé essentiellement sur IPv4, permet d'identifier et de faire communiquer ensemble 2^{32} machines différentes, soit un peu plus de 4 milliards d'ordinateurs – routeurs et hôtes. Un autre réseau de taille à peu près comparable est le cerveau humain, qui connecte quelque 100 milliards de neurones en moyenne. Ce dernier a évolué depuis des millions d'années avant d'arriver à sa taille et à sa complexité actuelles, une éternité comparée aux 40 ans seulement depuis les prémices d'Internet !

Internet est déjà, sans doute, la plus complexe et la plus vaste construction jamais réalisée par l'homme. Depuis 2011, il n'y a plus d'adresse IPv4 qui ne soit pas allouée : Internet a atteint la taille adulte et change actuellement de garde-robe, grâce à des protocoles et des formats rénovés, IPv6, qui visent à

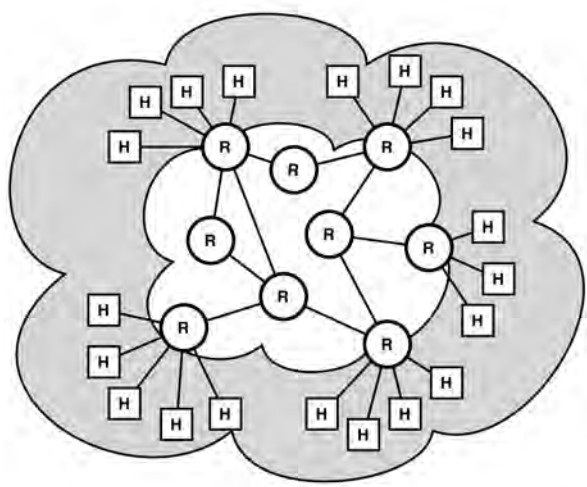
faire communiquer ensemble jusqu'à 2^{128} (c'est-à-dire 3×10^{38}) machines, soit plusieurs ordres de grandeur de plus que le nombre d'Avogadro !

Ce chapitre tente d'introduire les notions essentielles, formant la base du système immensément complexe qu'est Internet. Pour éveiller l'intérêt, comparer la complexité de ce système avec celui du cerveau d'un être humain est peut-être plus profondément motivant qu'interpeller à brûle-pourpoint les élèves sur le dernier sujet à la mode, comme par exemple : « Vous êtes-vous demandé comment marche Facebook réellement ? ».

Gérer la complexité au moyen d'abstractions

Quand on écrit un courriel à `personne@exemple.com`, on ne sait pas où cette personne se trouve physiquement, elle pourrait tout aussi bien être de l'autre côté de la Terre que dans la pièce d'à côté. Même l'ordinateur depuis lequel on écrit ce courriel ne sait pas où se trouve la machine stockant les courriels de `personne@exemple.com`. L'important est que l'ordinateur *n'a pas besoin de le savoir.* Il n'a pas non plus besoin de savoir si la machine stockant les courriels de `personne@exemple.com` est joignable via Lyon ou via Berlin.

Une manière de gérer la complexité du système est d'avoir une séparation claire entre deux catégories de machines sur Internet : les hôtes d'une part et les routeurs de l'autre, la périphérie du réseau d'une part et le cœur du réseau de l'autre.



Cœur et périphérie. Routeurs et hôtes.

Les hôtes sont des machines telles que les ordinateurs utilisés par un internaute de tous les jours, les smartphones, mais aussi les serveurs mail, les serveurs web, etc. Leur rôle est essentiellement *d'utiliser Internet*, en exécutant des applications accédant au réseau via la couche transport. Du point de vue réseau, un programmeur d'application se soucie essentiellement de savoir quel protocole de transport utiliser (TCP ? UDP ?) et quels sont les noms des hôtes ou serveurs avec lesquels communiquer. Le programmeur ne se soucie pas du reste, notamment pas de la façon dont les paquets trouvent leur chemin jusqu'à ces hôtes ou serveurs, et s'en remet au réseau pour cela.

Et le réseau s'en charge ! Quand on configure un ordinateur pour accéder à Internet, on renseigne essentiellement ce dernier de l'adresse du « routeur par défaut » à contacter pour tout accès réseau. Ce routeur est justement à la frontière entre la périphérie et le cœur d'Internet, et la configuration de l'ordinateur concernant l'utilisation de ce routeur par défaut est agnostique quant à l'application ou au protocole de transport utilisés pour communiquer par l'intermédiaire de ce routeur avec un serveur ou un hôte distant, que cet hôte ou serveur soit dans la pièce d'à côté ou à l'autre bout du monde. De la même manière que pour la communication entre humains – voir la transcription d'interview illustrée ci-avant –, l'intermédiaire est agnostique quant aux idées échangées et se contente de retranscrire l'oral en écrit ; dans les réseaux d'ordinateurs, les routeurs se contentent d'aiguiller du mieux possible les paquets que la couche transport lui demande d'envoyer vers leur destination. Pour les applications sur un ordinateur, elles reçoivent et envoient des données, et tout ce qu'il se passe au-delà du routeur par défaut, jusqu'à destination, c'est égal : les routeurs s'en chargent, que ce soit via Ethernet, Wifi, voire des signaux de fumées et/ou des pigeons voyageurs.

Comment enseigner

Il y a essentiellement deux types d'approche didactique pour introduire le domaine des réseaux d'ordinateurs en général et Internet en particulier.

Construire de bas en haut

La première est de construire le système à partir de rien pour arriver à un système fonctionnel. Le déroulé typique de cette approche est de commencer par décrire comment faire le premier pas, à savoir envoyer des informations sur un lien – câble ou radio. Puis on décrit le deuxième pas : le réseau de liens. Comment l'information trouve-t-elle son chemin à travers un dédale de liens et comment est-elle transférée automatiquement le long de ce chemin, même si les liens qui composent ce dernier sont de natures diverses ? On passe ensuite

aux protocoles de transports à l'œuvre au-dessus du réseau et les services de gestion et de garanties qu'ils fournissent. Enfin, on aborde les applications qui utilisent les services de la couche transport. Cette approche didactique se résume essentiellement à la démarche suivante : à partir d'arêtes – les liens – et de sommets – les routeurs –, on construit un graphe, sur lequel des algorithmes sont mis à l'œuvre.

La littérature adopte généralement cette approche, qui a l'avantage d'avancer peu à peu sur des acquis clairs et solides, ce qui n'est pas négligeable. Cependant, cette approche comporte certains risques, dont il faut être conscient en tant qu'enseignant :

Un long chemin avant de pouvoir pratiquer. Pour comprendre comment programmer une application simple utilisant le réseau, il faut avoir absolument tout étudié : les couches lien, réseau, transport. Les détails des couches basses ne sont pas forcément essentiels pour les non-spécialistes, tandis que ceux des couches supérieures le sont. En effet, la plupart des dernières évolutions d'Internet se sont notamment passées dans la couche application, que ce soit le Web, Skype, les réseaux sociaux, les réseaux pair-à-pair ou la vidéo à la demande.

L'envers du décor avant les concepts clés. En un certain sens, enseigner les réseaux d'ordinateurs en construisant de bas en haut est contraire à ce que cette construction même cherche à produire, à savoir : une architecture d'abstractions permettant de s'affranchir de certaines complexités. En construisant de bas en haut, les élèves sont exposés aux complexités d'abord, et seulement ensuite aux abstractions architecturales, alors que ces dernières sont fondamentales.

Conserver l'intérêt des élèves peut être ardu. Ne pas tomber dans un catalogue assommant, pour les élèves comme pour le professeur, de formats et propriétés se révèle parfois difficile. On perd facilement la vue d'ensemble qu'il est nécessaire de garder à l'esprit pour comprendre l'utilité des algorithmes et des protocoles dont on parle.

Déconstruire de haut en bas

La deuxième approche didactique est moins scolaire : il s'agit de déconstruire le système à partir de la couche application. Le déroulé typique de cette approche est de commencer par l'interaction client-serveur pour HTTP ou pour le courriel, et donner l'exemple de l'application TELNET que l'on peut manipuler en travaux pratiques. Ensuite, toujours en travaux pratiques, on peut faire écrire aux élèves un programme dans un langage qu'ils connaissent, similaire du point de vue fonctionnalité à TELNET, envoyant et recevant des

données via TCP, fournissant par exemple de la messagerie instantanée entre deux ordinateurs. On abordera alors les services de la couche transport, par analogie avec les services postaux – service de base, recommandé avec accusé de réception, Chronopost –, et le rôle de la couche réseau, puis le concept même d'organisation en pile de protocoles. Si les élèves se prennent au jeu, une séance interactive viendra assez naturellement, au cours de laquelle on leur demandera de suggérer eux-mêmes des solutions ou des améliorations aux différentes solutions pour les services de la couche transport. La suite se concentre sur les différentes solutions possibles pour fournir les services de la couche réseau, introduire la différence entre hôtes et routeurs, etc., ce qui peut donner lieu à une autre séance interactive sur le sujet. On conclura alors sur le rôle et les mécanismes de la couche lien.

Les avantages principaux de cette approche sont de s'appuyer sur des travaux pratiques motivants dès le début du cours, pourvu que des ordinateurs connectés à Internet soient à disposition, et d'avoir l'occasion de faire comprendre aux élèves l'abstraction d'abord, et de découvrir les détails sous-jacents à l'abstraction ensuite. Cependant, cette approche comporte certains risques, dont il faut être conscient en tant qu'enseignant :

Déconstruire peut être frustrant. Il faut attendre la fin du cours pour avoir des explications sur certains mécanismes de base d'Internet, ce qui peut se révéler inconfortable pour ceux qui cherchent d'emblée le germe de la « réaction en chaîne » qui produit Internet.

Le côté non scolaire peut dérouter. Séances interactives et travaux pratiques sont d'excellents catalyseurs, mais ne sont pas forcément à la portée de tous dans toutes les circonstances, du point de vue des élèves comme du point de vue du professeur.

Les points clés

L'objectif pour les élèves est d'acquérir les notions de base concernant l'architecture matérielle et logicielle d'Internet. Il est important d'avoir compris comment une pile de protocoles fonctionne, et le rôle de chaque couche. Il est également important d'avoir compris le rôle des protocoles de transport sur les hôtes en périphérie d'Internet, et le rôle des routeurs au cœur d'Internet, qui gèrent l'aspect algorithmique sur le graphe sous-jacent du réseau.

Pour finir, il est important que les élèves acquièrent le sentiment d'avoir appris quelque chose en prise avec le réel, par exemple par le biais de la programmation d'une application simple communiquant sur le réseau. De ce sentiment pourrait germer chez eux l'idée qu'ils ne sont pas seulement consommateurs, mais bien des acteurs potentiels d'Internet – ce qui est la réalité.

Compléments

L'informatique en général et les réseaux informatiques en particulier utilisent en pratique des termes anglais et des acronymes provenant d'expressions anglaises. Il est donc impératif d'apprendre les termes anglais en même temps que les termes français, afin de pouvoir directement appliquer les savoirs acquis, ne serait-ce que pour bien comprendre ou modifier la configuration de l'accès réseau d'un ordinateur personnel. Ci-après, un bref lexique, à compléter selon les besoins :

Byte : Octet.
Congestion Window : Fenêtre de congestion (de TCP).
Data : Données.
Header : En-tête.
Host : Hôte.
Layer : Couche.
Link : Lien.
Medium : Support physique.
Network : Réseau.
Packet : Paquet.
Prefix : Préfixe.
Protocol : Protocole.
Router : Routeur.
Slow Start : Début lent (de TCP).
Stack : Pile.

De même, la plupart des spécifications des normes et des protocoles sont disponibles uniquement en anglais. On en conclut donc qu'une certaine maîtrise de cette langue est nécessaire en pratique.

Normes de la couche lien

Les protocoles Ethernet, Wifi ou Bluetooth mentionnés au paragraphe « La couche lien » sont basés sur une évolution des mécanismes de ALOHA, appelée CSMA (*Carrier Sense Multiple Access*), qui permet à un ordinateur d'éviter plus de collisions, en l'obligeant à écouter le support physique avant de transmettre, pour vérifier qu'aucun autre ordinateur n'est déjà en train de transmettre, auquel cas il sursoit et attend un temps aléatoire avant de recommencer la procédure – écouter avant d'essayer de transmettre. Une variante de CSMA appelée CSMA-CD (*Collision Detection*) est à la base d'Ethernet et réduit l'impact des collisions en obligeant un ordinateur à écouter pendant qu'il transmet, pour détecter si une collision est en train d'avoir lieu et dans ce cas arrêter immédiatement de transmettre – au lieu

de finir de transmettre le paquet prévu, qui de toute façon est perdu du fait de la collision. Une autre variante, CSMA-CA (*Collision Avoidance*), est à la base de Wifi et permet d'esquiver davantage de collisions.

Ethernet, Wifi et Bluetooth sont des normes de communications locales entre machines, spécifiées par un organisme appelé l'IEEE (*Institute of Electrical and Electronics Engineers*). Le principe d'une norme est de publier un *modus operandi* garantissant la compatibilité avec certains critères. Une norme de communication entre machines, par exemple, consiste en une spécification consultable publiquement, pouvant ainsi être utilisée par n'importe quel constructeur d'ordinateur pour garantir que ses ordinateurs sont compatibles avec n'importe quel autre ordinateur, même issu d'un autre constructeur, pourvu que la spécification soit aussi respectée par cet autre constructeur. Pour consulter les spécifications d'Ethernet, Wifi ou Bluetooth, il suffit de connaître le nom technique de chacune de ces normes, à savoir IEEE 802.3 pour Ethernet, IEEE 802.11 pour Wifi, et IEEE 802.15.1 pour Bluetooth. D'autres organismes de normalisation existent, tel 3GPP, qui s'occupe des normes de téléphonie cellulaire, qui utilisent des mécanismes différents pour connecter des appareils au réseau téléphonique et à Internet, tel UMTS (nom technique de la 3G).

ARPANET : la naissance de la couche réseau

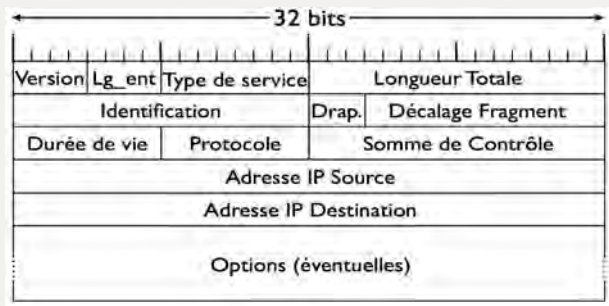
L'aiguillage des paquets de manière indépendante les uns des autres, pouvant prendre alternativement des chemins différents pour aller au même endroit, est appelé la commutation par paquets. Il existe une autre technique pour aiguiller les paquets, basée sur l'établissement d'un chemin physique ou logique fixé entre deux ordinateurs, le temps d'une session de communication entre ces deux ordinateurs, le long duquel sont transférés tous les paquets appartenant à cette session. Cette technique s'appelle la commutation de circuits et est issue des réseaux téléphoniques, où le concept de session – l'occupation d'une ligne téléphonique le temps d'une conversation – a un rôle prépondérant. Le défaut majeur de la commutation de circuit est qu'il n'est pas possible d'utiliser les « silences » sur la ligne d'une conversation pour faire passer d'autres paquets d'autres conversations, ce qui sous-utilise le réseau, contrairement à la commutation de paquets, qui peut mieux utiliser les capacités du réseau. C'est pourquoi la commutation de circuits est peu à peu délaissée.

La commutation de paquets a été utilisée pour la première fois à la fin des années 1960 dans un réseau appelé ARPANET, l'ancêtre d'Internet, développé par les États-Unis pendant la guerre froide afin d'unifier les techniques de connexion permettant à un terminal de communiquer à distance avec des ordi-

nateurs de constructeurs différents. Selon un mythe répandu à propos d'ARPANET, ce dernier aurait été projeté pour doter le pays d'un réseau plus résistant que le réseau téléphonique, le but étant de pouvoir continuer à fonctionner quel que soit l'état de destruction du pays, même après une attaque nucléaire.

L'archétype de la commutation par paquets est Internet, basé sur le protocole IP. Ce protocole est une norme de communication entre machines, spécifié par l'organisme de normalisation appelé IETF (*Internet Engineering Task Force*). Il existe deux versions du protocole utilisé actuellement. L'une d'entre elles est IPv4, la version du protocole IP présentée dans ce chapitre. IPv4 est accompagné d'une suite d'autres protocoles avec lesquels il fonctionne de concert pour former Internet tel que la plupart des internautes d'aujourd'hui le connaissent. Il existe néanmoins une autre version, appelée IPv6, qui est une version rénovée du protocole IP. IPv6 est également accompagné de sa propre suite de protocoles avec lesquels il fonctionne de concert pour former une partie d'Internet moins connue des internautes, mais néanmoins importante car plus à même de gérer la taille gigantesque qu'Internet a acquise, et sa croissance encore à venir. En effet, le nombre de machines connectées à Internet étant devenu énorme, les 2^{32} adresses IPv4 disponibles n'étaient plus suffisantes. IPv6 a donc été développé en utilisant des adresses de 128 bits, permettant d'identifier beaucoup plus de machines, potentiellement 2^{128}. Cependant, la transition entre l'utilisation d'IPv4 et IPv6 n'est pas simple à opérer à grande échelle et suscite de nombreuses questions depuis quelques années.

La suite des protocoles IPv4 ainsi que la suite des protocoles IPv6 sont publiées dans des documents accessibles publiquement sur Internet, dans une série de documents appelés RFC (*Request For Comments*). IPv6, par exemple, est spécifié dans la RFC 246, tandis que IPv4 est spécifié dans la RFC 791. Cette figure



Structure de l'en-tête d'un paquet IPv4, illustrée ici par rangées successives de 32 bits lues de gauche à droite. L'adresse source et l'adresse destination sont les adresses IP de l'émetteur et du destinataire du paquet, respectivement.

illustre la structure de l'en-tête d'un paquet IPv4. On y retrouve les champs correspondant aux adresses IP de l'émetteur du paquet (la source) et de la destination du paquet, ainsi que la somme de contrôle et la durée de vie du paquet IP, que l'on a vu dans ce paragraphe. Ces champs ainsi que tous les autres sont définis dans la RFC 791.

Sécurité réseau

Des myriades d'applications sont utilisées tous les jours par des centaines de millions d'internautes, que ce soit pour recevoir et envoyer des courriels, poster des commentaires ou des photos sur un site web, visionner une vidéo en ligne, acheter un article en vente par correspondance, etc. On l'aura compris : les programmes de la couche application sont extrêmement divers et nombreux et n'ont pour point commun que celui d'utiliser le réseau à travers la couche transport décrite dans ce chapitre.

Il est néanmoins conseillé d'étudier le fonctionnement de quelques applications de base, telles que HTTP, TELNET, ou encore SSH. Dans ce but, on consultera, par exemple, les documents suivants : RFC 2616, RFC 854, et RFC 4251. L'application HTTP est utilisée pour transférer des pages web, tandis que TELNET (*TErminaL NETwork*) offre un moyen de communication bidirectionnel généraliste entre deux machines distantes. L'application SSH (*Secure Shell*) est quant à elle une version sécurisée de TELNET, dans le sens que, en plus des services offerts par TELNET, elle crypte les données envoyées entre deux machines distantes.

On notera à ce propos que, dans ce chapitre, les mécanismes abordés jusqu'à présent ne sont pas sécurisés : les communications entre machines peuvent notamment être victimes d'écoutes intempestives qui révèlent les informations transmises, ce qui pose un problème pour certaines applications, par exemple pour le commerce en ligne. Des mécanismes supplémentaires doivent donc être mis en place pour fournir des services tels que la confidentialité des informations transmises. Il existe par exemple une version sécurisée de HTTP, que l'on appelle HTTPS, utilisée tous les jours par les internautes pour garantir la confidentialité des renseignements bancaires échangés lors d'une transaction dans le commerce en ligne.

La sécurité des réseaux est un vaste domaine qui ne se limite d'ailleurs pas à la problématique de la confidentialité de l'information transmise, mais comprend également des problématiques telles que l'intégrité et l'authenticité de l'information transmise, la garantie de la mise à disposition de l'information, et du principe de non-répudiation lors de transactions sur le réseau, ou encore la protection des protocoles à toutes les couches contre les

dysfonctionnements causés par le sabotage. Ces problématiques ont donné lieu au développement de nombreux mécanismes supplémentaires utilisés de nos jours sur Internet, qui ont des implications complexes quant au fonctionnement actuel du réseau.

Usages et trans-nationalité du réseau

Jusqu'à l'apparition d'Internet et du Web, les documents publiés sous la forme de livres, d'articles de journaux, etc. l'étaient toujours dans un pays particulier, et cette publication était régie par le droit du pays où ce document était publié. La publication sur Internet permet en revanche de mettre des contenus à la disposition de tous les utilisateurs connectés dans le monde entier, potentiellement sans délai et sans intermédiaire. L'émetteur peut même opter pour l'anonymat, changer de point d'émission, ou encore modifier ou supprimer des contenus. Cette nouvelle situation pose de nouvelles questions, auxquelles ne répondent pas les réglementations créées pour une économie culturelle sur supports matériels – livres, journaux, disques, cassettes vidéo, etc. L'administration de la preuve est souvent difficile. La justice peut être lente pour des actes aux effets rapides. Et les lois et coutumes des différents États entrent en conflit avec la vocation universelle de la publication sur les réseaux.

Par exemple, dans beaucoup de pays, la liberté d'expression est protégée et l'apologie du crime est condamnée. Mais, quand ces deux principes entrent en conflit, les lois de différents pays résolvent ce problème de manière parfois différente. Une jurisprudence célèbre a ainsi opposé, en 2000, les conceptions de deux États pourtant de cultures proches. En France, le port ou l'exhibition d'un uniforme, d'un insigne ou d'un emblème d'une personne coupable de crime contre l'humanité est condamnable. Mais cela n'est pas le cas aux États-Unis : ce pays n'a pas ratifié le Statut instituant la Cour pénale internationale qui définit les crimes contre l'humanité ; et le principe de la liberté d'expression est inscrit dans le premier amendement de la constitution. Une association d'anciens déportés a porté plainte, en France, contre un site web de vente aux enchères, situé aux États-Unis, mais accessible dans le monde entier, qui proposait à la vente des objets nazis – brassards à croix gammées, répliques de fioles de Zyklon B, dagues SS, etc. Une question nouvelle posée à ce moment était celle de la possibilité, pour un juge français, de poursuivre un justiciable américain, parce qu'il avait publié, aux États-Unis, un document accessible en France.

Plusieurs réponses à ce type de questions ont été imaginées, sans que personne ne sache encore laquelle s'imposera sur le long terme : l'applica-

tion du droit du pays dans lequel l'information est publiée, l'application du droit du pays dans lequel l'information est accessible – ce qui obligerait, par exemple, un site web à bloquer l'accès, depuis certains pays, à certaines parties de son site ou l'émergence d'un minimum de règles universellement acceptées. Par exemple, plusieurs sites de ventes aux enchères interdisent désormais, de leur propre initiative, la vente d'objets nazis, même si la loi de leur pays ne l'interdit pas.

Cette nécessité de règles universellement acceptées apparaît aussi dans l'organisation d'Internet même. Ainsi, pour que les ordinateurs du monde entier puissent communiquer entre eux, il est nécessaire qu'ils utilisent le même protocole de communication, et pour que les pages web publiées dans un pays soient accessibles dans le monde entier, il faut que le système de nommage de ces pages – les adresses web – soit le même partout. Le réseau mondial n'est donc pas entièrement décentralisé : un petit nombre de décisions doivent être prises en commun pour assurer cette compatibilité. Ici, plusieurs modes d'organisation sont en concurrence, sans, à nouveau, que personne ne sache lequel s'imposera sur le long terme : l'émergence d'organisations internationales régies par des traités entre États, l'émergence d'organisations internationales informelles, dont la légitimité vient uniquement de la confiance qui leur est accordée, ou l'émergence d'organisations propres aux pays les plus connectés, aggravant la fracture numérique entre pays.

Aujourd'hui, la gouvernance du cœur technique du système est concentrée entre l'ICANN (*Internet Corporation for Assigned Names and Numbers*), association de droit californien qui gère le nommage, et l'IETF (*Internet Engineering Task Force*) pour une grande part des protocoles de base. L'IETF, groupe informel, est cependant gérée par l'*Internet Society* (ISOC), une association de droit virginien. La Chine s'est partiellement séparée de ce système en 2006. Plus international dès sa création, le W3C (*World Wide Web Consortium*) définit les standards du Web. L'Union internationale des télécoms (UIT), qui joue un rôle majeur, est une agence spécialisée de l'Organisation des Nations unies (ONU). Depuis le Sommet mondial sur la société de l'information (SMSI), créé en 2003 par l'ONU, des réunions mondiales de discussion ont lieu chaque année : c'est le *Forum pour la gouvernance d'Internet*.

La croissance d'Internet selon des modalités plus internationales est un enjeu essentiel pour son développement et pour la préservation de la diversité culturelle.

Structuration et contrôle de l'information

Nous revenons dans ce chapitre à la notion d'information, abordée à une plus grande échelle. La première idée que nous présentons est celle de la structuration d'informations. La seconde est celle de la communication de l'information sur les réseaux qui demande de nouveaux algorithmes qui permettent la manipulation de grandes quantités : des algorithmes de compression, dont nous avons déjà vu un exemple au chapitre « Algorithmique », des algorithmes de correction d'erreur et des algorithmes de chiffrement. La différence entre les algorithmes de correction d'erreur et de chiffrement fait apparaître deux problématiques transversales à l'informatique : la protection contre les erreurs involontaires et la protection contre les agents malveillants.

Cours

Structures de données de base

Définir des données structurées Moi, c'est « Boulanger, Paul, boulanger à Saint Paul ». Enchanté, « Jean Pierre, de Grande Pierre, tailleur de Pierre ». Comprenne qui pourra. Extraire du discours les informations données peut rapidement devenir hasardeux et finira par être voué à l'échec. Ici, toutes les ambiguïtés seraient levées si nous présentions les choses de manière structurée, par exemple sous la forme :

```
{
prénom = "Paul"
patronyme = "Boulanger"
métier = "boulanger"
lieu = "06570"
}
```

Chaque paramètre est maintenant explicité. On pourra même, pour exprimer le lieu, disposer de la liste des 36 000 villes et villages de France – du point de vue informatique, c'est une liste de taille moyenne – et lever les ambiguïtés dues au fait que dix villages ont le nom « Saint-Paul » ou que ce nom peut s'écrire « St. Paul », « Saint Paul », etc. en utilisant le code postal, au lieu du nom de la commune.

Quelles sont les « bonnes pratiques » pour définir des données structurées ? Voici trois éléments de réponse.

– *Maximiser l'atomicité.* La structure des données doit avoir une décomposition maximale. Par exemple, il ne faut pas spécifier nom = "Jean Pierre" mais bien {prénom = "Jean" patronyme = "Pierre"} afin de lever toute ambiguïté et pouvoir accéder à chaque « atome » d'information. En fait, dès qu'une information, par exemple une adresse postale, nous semble composite, elle doit être décomposée en éléments plus petits.

– *Maximiser la factorisation*. C'est-à-dire éviter la redondance des données : si une donnée se retrouve plusieurs fois à différents endroits, il faut l'extraire, la coder à une place spécifique et créer des références des endroits initiaux vers sa place canonique. Par exemple, le code postal référence de manière unique les communes de France, il est préférable de l'utiliser plutôt que « Saint-Paul-de-Vence, dans les Alpes-Maritimes ». De plus, ces informations ne concernent pas directement notre Paul Boulanger, mais tous les habitants de Saint Paul. Ici, le code postal est une *référence* qui permet de *pointer* de la structure qui décrit Paul Boulanger vers une autre liste, celle des communes de France, où toutes les informations relatives à Saint Paul sont codées une seule fois. Cela permet d'économiser du temps et de la place, mais aussi d'éviter des incohérences de données.

– *Minimiser les sous-structures de données*. C'est-à-dire éviter les arborescences complexes, lorsque les données peuvent être représentées sous forme de liste uniforme des données. Ainsi, nous aurions pu représenter l'information de la manière suivante :

```
{
 nom = {
  prénom = "Paul "
  patronyme = "Boulanger "
 }
…
}
```

renforçant l'idée que prénom et patronyme sont les deux sous-composantes de la composante nom. Cela est plus rigoureux, mais guère indispensable et rend plus complexe l'accès aux données.

L'utilisation de telles *structures logiques* répond à deux objectifs.

– *Maximiser les descriptions génériques*, c'est-à-dire les représentations indépendantes du format de fichier ou de la syntaxe du système qui les exploite. Selon la façon dont les données vont être utilisées, les algorithmes pourront les restructurer à leur façon, mais ici nous les codons de la manière la plus proche du sens que nous leur donnons.

– *Maximiser la description statique des objets*, c'est-à-dire préciser, autant que possible, les propriétés – caractéristiques, paramètres, options – d'un programme comme un ensemble statique de données, plutôt que de glisser ces données dans le code de ce programme.

Nous aurions pu aussi représenter l'information par un algorithme de la forme : `si (prénom == "Paul") alors nom = "Boulanger"`... mais l'information aurait été noyée dans le code du programme. Ce qui est proposé ici est de séparer au maximum données et programme.

Quelques données structurées essentielles Les formulaires que nous trouvons sur le Web nous permettent de définir de tels « n-uplets à champ nommés », ou enregistrements, que nous avons introduits dans le deuxième chapitre et que nous retrouverons dans le septième chapitre.

Données numériques et quantités physiques. Nous avons vu comment coder un nombre. Mais la représentation d'une quantité physique, de toutes les valeurs numériques en fait, ne se limite pas à une valeur. Si Paul Boulanger mesure 1,80 m, cette donnée numérique appartient au type des `tailles_humaines` et se spécifie correctement par rien moins que… six paramètres !

```
{
 taille_humaine = {
  valeur_minimale : float = 0
  valeur_maximale : float = 2.7
  précision : float = 0.01
  valeur_par_défaut : float = 1.7
  unité : identifier = "m"
 }
}
```

à savoir, son identificateur – `taille_humaine` –, ses valeurs minimales et maximales, sa précision, une valeur par défaut et l'unité physique qui est un

identificateur, indispensable pour manipuler des quantités avec des unités éventuellement différentes.

Il est essentiel de toujours définir ces paramètres pour manipuler correctement l'information numérique en question. Les algorithmes qui vont estimer ces données numériques ou les utiliser ont besoin de savoir, par exemple, entre quelles valeurs et à quelle précision travailler. Il faut aussi constater que ces spécifications sont naturelles et accessibles : une règle d'écolier mesure une longueur entre 0 et 20 cm à une précision de 1 mm ; un point sur une image numérique est situé horizontalement entre le bord gauche et droit de l'image à une précision de un pixel et une valeur par défaut égale au milieu de l'image ; toute vitesse se situe à plus ou moins la vitesse de la lumière avec une valeur par défaut égale à zéro, etc. C'est donc un simple exercice de rigueur que d'expliciter ces informations, lors du codage de données numériques.

L'exemple de la date et de l'heure. Juste avant la tragédie du `01-09-11` le « bug de l'an 2000 » a fait couler beaucoup d'encre… De quoi s'agissait-il ? D'une mauvaise standardisation du codage de la date. Que signifie `01/09/11` ? L'Anglo-Saxon lira le 9 janvier 2011 – ou 1911, suivant son siècle. Le Français, le 1er septembre de la même année. Les anciens logiciels, le 11 septembre 2001, ou 1901. Le codage des bases de données sera `2001-09-11 : année/mois/jours`, c'est la norme ISO-8601. C'est le bon ordre pour aller de la plus grande unité à la plus petite, donc pour que l'ordre lexicographique de la chaîne de caractères corresponde à l'ordre des dates. On fera bien attention de ne pas omettre les zéros, à ne pas écrire `2001/9/11`, bref à respecter l'expression régulière `date : [0-9][0-9][0-9][0-9]/[0-9][0-9]/[0-9][0-9]`. On proposera aussi à l'utilisateur un calendrier graphique à cocher plutôt que le laisser s'embrouiller dans les codages chiffrés des mois ; il pourra aussi situer les jours de la semaine.

Dans le système d'exploitation *Unix*, le temps est codé par la différence, mesurée en secondes, entre l'heure actuelle et 0 heure, 1er janvier 1970, heure de Greenwitch, codée sur 64 bits. Dans beaucoup de logiciels, ce codage est fait en millisecondes avec la même origine. Ce qui permet de coder 10^9 années environ, et une précision de 1 ms bien évidemment. S'il avait été codé sur 32 bits, ce nombre aurait eu des valeurs minimales et maximales… assez limitées – amusons-nous à le calculer. Cette précision de la milliseconde est générique : tandis qu'un ordinateur effectue quelques milliards d'opérations élémentaires par seconde, on ne peut lui demander d'avoir une précision absolue d'une nanoseconde.

L'exemple des identificateurs universels de ressources. Lorsque nous tapons par exemple : `http://science-info-lycee.fr`, nous identifions une res-

source disponible sur Internet, ou dans notre ordinateur, de manière universelle. Il se trouve que chaque fragment de cette chaîne de caractères a un sens précis, il correspond à une structure de données qui caractérise exactement la ressource que nous cherchons et la façon dont nous voulons y accéder.

Avec les ingrédients donnés ici, nous verrons en exercice comment comprendre chaque détail de cette spécification.

L'exemple des documents textuels. Quelle est la différence entre une chaîne de caractères brute et un texte ? Dans un texte, il y a une structure – des enluminures, des paragraphes, des références… – et des métadonnées – son titre, son auteur, la langue dans laquelle il a été écrit… – et tous ces éléments peuvent être spécifiés grâce aux constructions que nous avons évoquées ici.

Les métadonnées, le titre du texte, son auteur, ses destinataires, etc., rendent le texte référençable. C'est, par exemple, la structure des courriels : ils ont un sujet – c'est le titre –, un expéditeur – c'est l'auteur, sauf dans quelques cas particuliers –, etc. Un document aura en plus des mots-clés – ou *tags* – qui sont des mots précis – pour le présent chapitre, ce sera sûrement « informatique », « information »… – permettant d'indexer le texte, c'est-à-dire de le retrouver dans une base de données ou une bibliothèque numérique qui contient un grand nombre de documents. Si c'est un document multimédia, les auteurs auront probablement des rôles divers : rédacteur, illustrateur, éditeur, relecteur…

Trois ingrédients vont faire de ce texte plus qu'une chaîne de caractères.

Le texte lui-même est structuré de manière hiérarchique en chapitres, sections, sous-sections et paragraphes, par exemple quelque chose comme

```
{
 titre = "Introduction"
 {
  titre = "Avant-propos"
  …
}
{
 titre = "Présentation"
 …
}
 …
}
```

où l'on voit que l'avant-propos et la présentation sont, ici, des sous-sections de l'introduction. C'est cette structure qui permet de naviguer dans le texte sans être obligé de le lire du premier au dernier caractère à chaque accès !

Des portions de texte sont enluminées, c'est-à-dire mises en italique ou entre guillemets, etc.

Surtout, les textes numériques peuvent avoir des liens hypertexte, c'est-à-dire que nous pouvons cliquer pour accéder à un autre contenu : consulter la définition d'un mot si nous ne le connaissions pas. L'encyclopédie Wikipédia en est l'exemple paradigmatique. Ces liens donnent aussi accès à du matériel supplémentaire au texte – une illustration, une parenthèse vers un élément connexe mais hors du fil du discours, etc. Sur papier, ce sont des notes de bas de page ou des références vers un glossaire ou une bibliographie.

Ces trois ingrédients – structure, enluminure, liens – font du texte, qu'il soit numérique ou sur papier, un document bien plus riche qu'un simple discours linéique. Que le texte soit numérique enrichit évidemment les implémentations possibles, mais il est bon de noter que cet enrichissement n'est pas lié à la numérisation : le livre papier code les mêmes ingrédients.

Des données structurées aux structures de données Ce qui reste à découvrir, c'est que, sous-jacentes à ces données structurées, l'informatique est dotée de structures de données très riches pour représenter la structure de ces données… Elles se rangent dans deux grandes catégories :

Les collections. Les collections sont des suites de données : des listes. Selon la façon dont nous accédons à ces objets, nous parlerons de tableaux où les N objets sont numérotés de 0 à $N-1$ et nous pouvons accéder en lecture ou écriture au i-ème élément directement : si le tableau est de nom `tab`, son i-ème élément se note souvent `tab[i]`. Nous appelons *tableau* une telle structure de longueur fixe, décidée lors de sa construction. Des tableaux de longueur ajustable au fur et à mesure que nous y ajoutons des éléments sont appelés *vecteurs*. Les piles sont des vecteurs auxquels nous pouvons ajouter les éléments à un seul bout et retirer uniquement le dernier élément, comme une pile d'assiettes : se doter de telles restrictions permet de bien cadrer à quelle fonction une telle structure correspond, donc à en spécifier l'usage. Cela permet aussi d'optimiser son implémentation.

Les tables de valeurs. Les tables sont des listes qui associent à une valeur d'entrée une valeur de sortie. Par exemple, un dictionnaire français-anglais qui a des mots français en entrée et leurs correspondants anglais – supposés uniques ici – en sortie. Ou des tables de paramètres avec des noms de paramètres en entrée et des valeurs en sortie. Contrairement au tableau, l'index – appelé « clé » dans ce cas – n'est pas forcément un entier de 0 à $N-1$, mais peut être une chaîne de caractères ou un objet numérique quelconque. Tandis que les valeurs d'une collection sont implicitement ordonnées du premier au dernier, l'ordre de ces paires de valeurs d'entrées/sorties n'a aucune signification.

Ce qui est remarquable ici, c'est que ces quelques éléments décrivent qualitativement la très grande majorité des structures de données informatiques.

Les données du Web sont décrites en utilisant le langage XML, dont le langage HTML est la spécialisation pour les pages web, à quelques nuances syntaxiques près. Cette spécification combine tout simplement collections d'objets et tables de valeurs. La structure logique d'un document XML est une spécification minimale, mais très puissante pour décrire des informations symboliques – l'idée n'est pas nouvelle, mais elle prend en XML une forme assez aboutie et est devenue un standard, ce qui est essentiel pour partager les données. XML, qui date de 1998 pour la 1re édition, est un langage de représentation d'information dit semi-structuré, c'est-à-dire qu'il permet de gérer facilement l'intégration de données à partir de schémas différents, ainsi que les données manquantes – valeurs nulles.

Ce noyau de spécifications XML comprend, entre autres choses :

XML eXtensible Mark-up
le langage lui-même, utilisé pour la représentation de données et documents structurés.
XSLT eXtensible Style Language Transformation
pour définir comment traduire une structure en une autre structure.
XML Schema pour définir le type des données, leur structure et vérifier leur validité.
XHTML eXtensible HyperText Mark-up Language
pour les documents du Web que nous connaissons.
SVG Scalable Vector Graphics
utilisé pour la représentation graphique en 2D.
XUL eXtensible User interface Language utilisé pour spécifier une interface utilisateur graphique (menus, boutons, etc.).
MathML mathématique Mark-up Language, utilisé pour représenter les formules mathématiques.
DocBook utilisé pour spécifier les métadonnées d'un document.

Toutes les informations symboliques des objets numériques sont entièrement spécifiées à travers de tels standards. C'est le reflet numérique des objets du monde réel.

Compression de l'information

La compression de données ou codage de source est l'opération informatique qui consiste à transformer une suite de bits *A* en une suite de bits *B* plus

courte, contenant les mêmes informations, en utilisant un algorithme particulier. Il s'agit d'une opération de codage. La décompression est l'opération inverse de la compression. C'est un couple de mécanismes omniprésent dans les systèmes numériques.

Avec un algorithme de compression sans perte, la suite de bits obtenue après les opérations successives de compression et de décompression est strictement identique à l'originale. Ces algorithmes s'appliquent à tous les objets numériques, sans tenir compte de leur contenu. Avec un algorithme de compression avec perte, la suite de bits obtenue après les opérations de compression et de décompression est différente de l'originale, mais l'information reste sensiblement la même, ces algorithmes sont donc liés au contenu – images, son et vidéo.

Rappelons qu'il n'existe pas de technique de compression de données sans perte universelle, qui pourrait compresser n'importe quel fichier. Par exemple, on vérifie en pratique qu'un fichier qui est déjà le résultat d'une compression se compresse mal, voire grossit par application du compresseur. Mais, dans la pratique, les mots, messages ou fichiers que l'on souhaite compresser ne sont pas quelconques et choisis aléatoirement parmi tous les mots, messages ou fichiers possibles, ils vérifient certaines propriétés que les compresseurs utilisent.

Compresser des données n'est pas si facile : les premières idées, comme celle consistant à utiliser un codage par répétition, où une suite de bits ou de caractères identiques est remplacée par un couple formé du nombre d'occurrences et bit ou caractère répété, mènent à des méthodes qui ne sont pas efficaces en pratique.

Codage par dictionnaire Le codage par dictionnaire est une classe d'algorithmes de compression sans perte qui fonctionne par la recherche d'une correspondance entre le contenu à comprimer et un ensemble de chaînes binaires contenues dans une structure de données appelée le « dictionnaire ». Lorsque le compresseur trouve une telle correspondance, il substitue à la chaîne binaire reconnue la référence dans ce dictionnaire.

Pour décrire en détail un tel algorithme, de type Lempel-Ziv-Welch, faisons les conventions suivantes :

– nous découpons la chaîne binaire en octets ;

– nous considérons un dictionnaire qui associe à chaque chaîne d'octets un numéro ;

– nous initialisons le dictionnaire avec les chaînes à un octet, le numéro de chaque chaîne étant la chaîne elle-même : 00000000 a pour numéro 000000000, etc. ;

– nous codons initialement les numéros sur 8 bits, puis, lorsque cela ne suffira plus, sur 9 bits, puis 10 bits, etc.

À mesure que le compresseur examine le texte, chaque fois qu'une chaîne déjà rencontrée est lue, la chaîne la plus longue déjà rencontrée est déterminée, et le numéro correspondant à cette chaîne avec le caractère concaténé – le caractère suivant du flux entrant – est enregistré dans le dictionnaire. Le numéro pour la partie la plus longue de la chaîne de caractères rencontrée est envoyé en sortie et le dernier caractère est utilisé comme base pour la chaîne suivante. Ce mécanisme permet donc de substituer aux chaînes leur numéro dans le dictionnaire. Le dictionnaire est ainsi construit dynamiquement d'après les motifs rencontrés.

L'algorithme de décompression reconstruit le dictionnaire à partir du texte compressé en entrée, puisqu'il dispose d'un numéro déjà connu et du caractère de base pour la chaîne suivante pour fabriquer le nouveau numéro. Cette opération se fait avec un temps de retard et il faut tenir compte du cas où le numéro de la chaîne suivante correspond au nouveau numéro non encore entré, mais cela se détecte et se gère sans problème. Il va donc aussi automatiquement augmenter le nombre de bits du codage du numéro selon la taille du dictionnaire reconstitué.

Les variantes de ces méthodes permettent de disposer d'un algorithme de compression performant et d'un algorithme de décompression rapide. Ils sont utilisés dans les compressions usuelles – fichiers « zip », images au format « gif », etc.

Codage entropique Nous avons déjà vu, dans le troisième chapitre, un algorithme de codage entropique : l'algorithme de Huffman. De manière plus générale, le codage entropique utilise des statistiques sur le message source, avec un code à longueur variable, qui attribue les mots de code les plus courts aux symboles de source les plus fréquents. La limite de cette possibilité de compression est l'entropie définie dans le premier chapitre.

Un algorithme plus sophistiqué et plus courant est l'algorithme de Huffman adaptatif : au fur et à mesure de l'arrivée des symboles, on compte leurs occurrences, puis on les associe sous forme d'un arbre où chaque feuille correspond à un symbole. Cet algorithme est adaptatif, car l'arbre est construit de manière dynamique au fur et à mesure de la compression du flux. La compression est toujours adaptée aux données, sans calcul statistique avant la compression. Il ne faut donc pas transmettre ou stocker la table des fréquences des symboles, puisque cette dernière se reconstruit à la décompression. L'algorithme est de plus capable de coder sur des flux de données, sans avoir besoin de connaître les symboles à venir pour déterminer le codage.

Codage et mesure en information Ces mécanismes de compression ont un double usage pratique lors de transmission de données et lors d'archivage sur des supports. Ils peuvent être combinés, à ces fins, avec les mécanismes d'encryptage ou de codes correcteurs décrits ci-après.

Les notions définies dans le premier chapitre permettent à la fois de comprendre les limites de ces méthodes et de les utiliser à plus haut niveau.

D'une part, on peut montrer que, pour une source de données d'entropie H, la longueur moyenne L d'un mot de code obtenu par codage de Huffman en utilisant des blocs de n symboles vérifie :

$$H \leq L < H + \frac{1}{n}$$

donc que l'information au sens de Shannon est bien une borne maximale de compression et que le codage de Huffman peut donner une très bonne approximation de cette borne : il est optimal.

Lorsque Huffman a inventé son algorithme, il avait connaissance des travaux les plus théoriques sur l'information de Shannon et il avait la volonté de construire un procédé de compression optimal. Proposer un tel algorithme n'avait donc pas uniquement un but pratique, mais aussi le but, plus théorique, d'atteindre la borne de Shannon.

Par ailleurs, ces algorithmes de compression permettent d'approximer le contenu en information d'un message au sens de Kolmogorov. En appliquant divers mécanismes de compression, on peut évaluer la longueur du programme correspondant qui pourra générer le message – c'est la longueur de l'algorithme de décompression ajoutée à la longueur du message compressé – et des méthodes assez sophistiquées permettent alors d'interpoler une approximation du contenu en information lui-même. Très récemment, il a été montré que la complexité de Kolmogorov est approchée par la taille d'un fichier compressé, et que l'on peut approcher la profondeur de Bennett en considérant le temps de décompression des fichiers compressés.

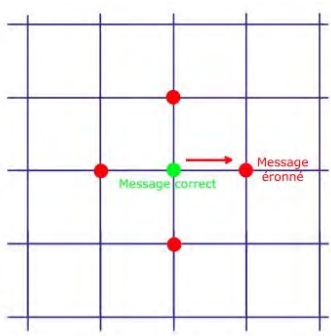
Codes correcteurs d'erreurs

Si la transmission n'est pas parfaite, certains bits seront erronés. Il faut alors ajouter à la suite binaire des bits supplémentaires pour espérer détecter, voire corriger, les possibles erreurs. L'exemple le le le plus plas plus simple sumple simple est de transmettre trois fois chaque élément. Si les trois éléments reçus sont identiques, la transmission a été effectuée sans erreur. Si l'un diffère des deux autres, il y a une erreur que l'on corrige en choisissant la valeur confirmée deux fois. Si deux erreurs se sont produites, cela ne sera ni détecté ni corrigé.

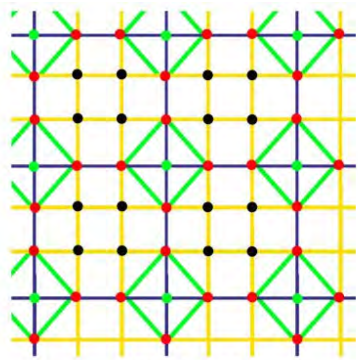
On peut aussi transmettre un bit de contrôle, par exemple 0 ou 1 selon qu'il y a eu un nombre pair ou impair de 1 dans les 8 bits précédents.

Les codes les plus efficaces sont plus compliqués et reposent par exemple sur l'utilisation de propriétés de polynômes sur un corps fini. Dans tous les cas, il faut ajouter de la redondance à l'information pour détecter et/ou corriger les erreurs. La problématique au niveau industriel est double. Dans le cas de la transmission de données, par exemple sur Internet, le rôle du code correcteur peut se limiter à la détection des erreurs et la correction est alors réalisée par une nouvelle demande de transmission du message. Pour d'autres situations, l'objectif est la correction d'erreurs, sans nouvelle demande possible de transmission. Avec deux types de situations : la correction de petites erreurs relativement fréquentes mais isolées, par exemple lors de transmissions, ou moins fréquentes mais beaucoup plus volumineuses, par exemple lors de rayures ou d'impuretés sur un DVD. Dans ce dernier cas, les codes utilisés corrigent jusqu'à 4096 bits consécutifs, soit une rayure de plus d'un millimètre de large. Dans un premier cas, les erreurs induisent une modification des données, où certains bits passent de la valeur 0 à la valeur 1 et vice versa. Dans un second cas, les erreurs provoquent des pertes d'informations aussi appelées effacements.

Le principe qu'une information supplémentaire est nécessaire pour, soit détecter l'erreur, soit la corriger, s'illustre très bien de manière géométrique :



Code sans redondance


Code correcteur

Si une unique erreur se produit, alors le message transmis correspond à un point rouge. Si la redondance a été habilement construite, alors il n'existe qu'un point licite en vert proche du point rouge reçu. Si d'autres erreurs se produisent et que le message transmis correspond à un point noir, l'erreur est détectée sans être corrigée.

Les points noirs consomment de la place inutilement si l'objectif est d'accepter le message même partiellement erroné. Un code est dit *parfait* quand il existe une taille des boules centrées sur les mots de code qui forment une partition de l'espace. On corrige alors certaines erreurs, mais on ne détecte pas d'erreurs non correctibles.

Un code correcteur propose donc une géométrie où les messages licites sont éloignés le plus possible les uns des autres. Les boules centrées sur les bons codes, si elles ne s'intersectent pas, permettent de retrouver le bon message, correspondant à son centre. Une perturbation, tant qu'elle reste suffisamment petite pour ne pas faire sortir le code de sa boule, est correctible.

Un bel exemple est le code de Hamming (7,4) qui, à travers un message de sept bits, transfère quatre bits de données et trois bits de *parité* qui permettent de corriger toute erreur portant sur un bit unique. C'est-à-dire que si, sur les sept bits transmis, l'un est altéré – un 0 devient un 1 ou l'inverse —, alors il existe un calcul algébrique permettant de corriger l'erreur. La génération du code est donnée par la matrice :

	d_0	d_1	d_2	d_3
c_1	1	0	0	0
c_2	0	1	0	0
c_3	0	0	1	0
c_4	0	0	0	1
c_5	1	1	0	1
c_6	1	0	1	1
c_7	0	1	1	1

qui indique comment calculer les sept bits du message codé à partir des quatre bits du message original : $c_1 = d_0$, $c_2 = d_1$, $c_3 = d_2$, $c_4 = d_3$, $c_5 = d_0 + d_1 + d_3$, $c_6 = d_0 + d_2 + d_3$, $c_7 = d_1 + d_2 + d_3$. Ces opérations s'effectuent en arithmétique modulo deux, où l'addition et la multiplication sont définies par les tables suivantes :

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	0

A	B	A . B
0	0	0
0	1	0
1	0	0
1	1	1

L'addition correspond à la fonction logique « ou exclusif » et la multiplication correspond à la fonction logique « et ». La somme d'une suite de valeurs est égale à 0 si la somme est paire dans les entiers et 1 sinon.

À la réception, un calcul matriciel dual défini par la matrice :

	c_1	c_2	c_3	c_4	c_5	c_6	c_7
e_0	1	1	0	1	1	0	0
e_1	1	0	1	1	0	1	0
e_2	0	1	1	1	0	0	1

permet de calculer les bits $e_0e_1e_2$ d'erreur qui valent 000 en cas de transmission correcte et donne *l'indice du bit erroné* en cas d'une erreur de transmission, qui peut alors être corrigé pour retrouver la valeur correcte.

Mathématiquement, ce code est structuré comme un sous-espace vectoriel sur un corps fini, ici le corps à deux éléments. Nous pourrions généraliser à des corps de base d, d étant une puissance de nombre premier. L'espace vectoriel du code est de dimension $n = 7$ dans notre cas et permet de coder des mots de taille $k = 4$ avec une distance minimale de $\delta = 3$ entre les mots du code, permettant de corriger des erreurs sur $\delta/2 = 1$ bit. On peut définir des codes linéaires pour d'autres dimensions $[n, k, \delta]$ et il est possible de vérifier qu'il faut $n - k \geq \delta - 1$. Si la borne est atteinte, nous sommes dans un cas de distance séparable maximale, ce qui n'est pas tout à fait le cas pour le code de Hamming (7,4).

Fonctions de hachage et condensats

Quand les erreurs sont peu probables, on se contente de les détecter, sans essayer de les corriger. Par exemple, lorsqu'on télécharge un gros fichier, la probabilité d'erreur est faible, mais on veut pouvoir détecter qu'il y a eu une erreur, afin de recommencer le téléchargement. Pour cela, on peut utiliser un condensat, c'est-à-dire un résumé du fichier que l'on pourra comparer à un résumé de référence.

En fait, le condensat est le résultat d'une fonction, appelée fonction de hachage, qui, à partir d'une donnée fournie en entrée, calcule une valeur servant à identifier rapidement, bien qu'incomplètement, la donnée initiale. Le

résultat d'une fonction de hachage peut être appelé selon le contexte « somme de contrôle », « empreinte », « hash », « résumé de message », « condensé », « condensat » ou encore « empreinte cryptographique ».

Les fonctions de hachage doivent être rapides : calculer le condensat d'une donnée ne doit coûter qu'un temps négligeable. De plus, elles doivent éviter autant que possible les collisions : ce sont des états dans lesquels des données différentes ont un condensat identique.

En général, on veut un condensat bien plus petit que la donnée à traiter. Ainsi, on veut télécharger le DVD d'une installation Linux d'environ 4 Go et on vérifie que le téléchargement s'est bien déroulé avec un condensat de 256 à 1024 octets. Si deux fichiers ont des condensats différents, ils sont différents, et on demande également que deux données différentes, mais proches, aient des condensats différents. Sinon, l'utilisation comme code détecteur d'erreur serait peu efficace. On demande donc souvent que deux données différant sur un seul bit aient des condensats différents.

Un premier exemple de condensat est le MD5 (*Message Digest 5*) inventé par Ronald Rivest en 1991. C'est un algorithme basé sur les opérations binaires ou exclusif, et, ou, non, l'addition, modulo 32 bits, et la rotation de bits, ainsi qu'un certain nombre de constantes. Depuis, cet algorithme a été étudié et on a découvert des collisions complètes. N'étant plus sûr, MD5 devrait être mis de côté au profit de fonctions plus robustes. L'algorithme SHA-1 (*Secure Hash Algorithm*) produit un condensat de 160 bits. Il utilise les mêmes briques de base que MD5 avec quelques opérations supplémentaires. Il a également été cassé dans les années 2000. Le condensat actuellement conseillé est son successeur SHA-256 dont le condensat est sur 256 bits.

Codage et cryptage

Dans de nombreuses situations, nous ne souhaitons pas voir nos données personnelles – numéro de carte de crédit, photos, lettres, etc. – exposées au grand jour. Cela demande de crypter ces informations. Cette idée est loin d'être nouvelle, puisque Jules César souhaitait déjà envoyer à ses armées des ordres, sous la forme de messages écrits, sans, bien entendu, que ceux-ci soient connus de ses ennemis. L'accès aux messages étant facile – il suffisait de capturer les messagers –, il fallait les crypter.

Codage de César et chiffrement par substitution

La méthode utilisée par César était extrêmement simple : elle consistait à associer à chaque lettre une autre lettre – par exemple la suivante dans l'ordre alphabétique. Pour César, c'était la lettre suivant celle suivant la suivante. Cela

revient à considérer les lettres comme des entiers entre 0 et 25 et à considérer une addition de 3 modulo 26. Ainsi, le message « je veux un bon cryptage » devenait « mh zhxa xq erq fubwdjh », ce qui n'est pas évident à traduire, sauf si l'on connaît le code – il suffit alors de soustraire 3 modulo 26. Cette méthode était efficace, mais son efficacité tenait beaucoup au fait que très peu de gens savaient lire. Elle souffre, en fait, de nombreuses faiblesses : la clé – c'est-à-dire la valeur permettant de coder et décoder les messages – est toujours la même, il suffit donc de la récupérer une seule fois pour pouvoir traduire tous les messages futurs. De plus, en comparant les messages codé et décodé, on en déduit facilement la clé. Enfin, la clé est 3 qui correspond à la lettre C, initiale de César, ce qui aide à confirmer que 3 est la bonne clé. Une version plus récente est le ROT13 qui ajoute 13 modulo 26 et qui permet de crypter légèrement des informations – par exemple, pour ne pas révéler la solution d'un jeu ou la fin d'un film. Il est surtout utilisé dans les forums, news et groupes.

De nos jours, un tel algorithme ne résisterait pas longtemps. Si l'on suppose qu'une lettre est toujours cryptée en une même autre lettre, il suffit d'observer, dans un message crypté, la fréquence des lettres pour retrouver le code. En français, la lettre « e » est la plus fréquente. Il faut donc, en considérant suffisamment de messages cryptés, trouver la lettre la plus fréquente pour lui associer la lettre « e ». On continue ensuite avec les autres lettres en fonction de leur fréquence moyenne en français et dans les messages cryptés. Cette méthode simple fonctionne très bien et permet de casser les codes par substitution assez facilement.

Cryptages symétriques

Dans les grandes catégories d'algorithmes de cryptage, on trouve les cryptages symétriques : c'est la même clé qui sert à chiffrer et à déchiffrer un message. Dans ce cas, il est primordial de garder cette clé secrète et qu'elle soit aléatoire. Une clé qui serait la date de naissance ou le nom du chien de l'expéditeur peut se deviner facilement. Un exemple simple de cryptage symétrique est le suivant. On considère l'opérateur binaire ou exclusif, noté $\oplus$ dont la table de vérité est

A	B	A$\oplus$B
0	0	0
0	1	1
1	0	1
1	1	0

et qui correspond à l'addition modulo 2, vue précédemment.

Revenons au chiffrement d'un message M de taille n. Il nous faut une clé secrète et aléatoire C également de taille n. On fait alors un chiffrement bit à bit : chaque bit de M subit un ou exclusif avec le chiffre de même rang de la clé C. On notera également, par abus de langage, $\oplus$ le ou exclusif sur les suites de bits. Pour déchiffrer le message crypté $M' = M \oplus C$, on procède exactement comme pour le chiffrement et on refait un ou exclusif bit à bit avec la clé. On a alors

$$M'' = M' \oplus C = (M \oplus C) \oplus C = M \oplus (C \oplus C) = M \oplus 0 = M.$$

En effet, $\oplus$ est associatif et, quelle que soit la clé, un ou exclusif avec elle-même donne la suite de zéros de taille n, qui est l'élément neutre de $\oplus$. En le cryptant une seconde fois, on a donc décrypté le message. Bien sûr, cette technique est loin d'être parfaite, car elle demande de fabriquer une clé aléatoire de la même longueur que le message, et surtout que cette clé soit partagée entre l'expéditeur et le destinataire du message.

Partager un secret : la méthode de Diffie-Hellman

Traditionnellement, en cryptographie, les deux personnes qui veulent communiquer de façon secrète s'appellent Alice et Bob. Leur canal de communication – courrier, Internet… – peut être écouté par Ève. Dans ces conditions, comment Alice et Bob peuvent-ils échanger une clé qui pourra servir à crypter une future conversation ?

L'algorithme de Diffie-Hellman permet de résoudre ce problème. Alice et Bob se mettent d'accord publiquement sur un nombre premier p et un générateur g du groupe $\mathbb{Z}/p\mathbb{Z}$. Alice choisit secrètement un nombre a et Bob secrètement un nombre b. Alice calcule g^a modulo p et l'envoie à Bob. De même, Bob calcule g^b et l'envoie à Alice. Une fois ces échanges publics effectués, Alice calcule $(g^b)^a = g^{ab}$ et Bob calcule $(g^a)^b = g^{ab}$. Ils ont donc à leur disposition la même valeur g^{ab} qu'ils pourront ensuite utiliser comme clé.

Quant à Ève, elle a en sa possession p, g, g^a et g^b, valeurs qu'elle a écoutées. Pour que Ève retrouve g^{ab} à partir de ces valeurs, elle doit élever g^a à la puissance b ou g^b à la puissance a. Mais déduire a – resp. b – de g^a – resp.g^b –, problème du logarithme discret, est très difficile : nous ne connaissons pas d'algorithme pour le faire rapidement. Ève est donc dans l'impossibilité pratique de calculer la valeur de g^{ab} sans utiliser un temps démesuré.

Mais ce protocole est vulnérable à l'attaque de l'« homme du milieu », Malo, qui suppose un attaquant capable de lire et de modifier tous les messages échangés entre Alice et Bob. Ainsi, si Malo peut modifier les messages,

il peut remplacer g^a par $g^{a'}$. Dans ce cas, Bob calcule $g^{a'b}$ et utilisera cette clé. De même, Malo remplace g^b par $g^{b'}$ et Alice utilisera $g^{ab'}$. Au lieu de communiquer secrètement, Alice et Bob communiqueront, sans le savoir, via Malo, qui pourra décoder et modifier tous les messages supposés cryptés.

Cryptages asymétriques et RSA

Pour éviter de devoir ainsi partager une clé secrète, on utilise souvent des chiffrements asymétriques – ou à clé publique. L'idée ici est d'avoir deux clés : une clé publique, qui sert à crypter, et une clé privée, qui sert à décrypter. Une métaphore est de mettre à disposition de tout le monde des portes blindées sans poignées et de donner à chacun la poignée de l'une de ces portes. Chacun peut alors facilement mettre un message derrière une porte et la claquer, c'est-à-dire crypter le message avec la clé publique. Désormais, seule la personne qui a la poignée de cette porte peut l'ouvrir et récupérer le message : seule la personne ayant la clé privée correspondante est capable de décrypter le message. Cryptage et décryptage sont ici deux algorithmes distincts.

La cryptographie asymétrique est basée sur l'existence d'une fonction dite à sens unique, pour transformer un message en message codé. Il faut que cette fonction soit simple à appliquer à un message quelconque, mais qu'il soit difficile de retrouver le message original à partir du message codé.

Le principe peut se voir comme un coffre à deux serrures. Lorsque l'une des deux serrures est fermée, la seule façon d'ouvrir la boîte est d'utiliser l'autre serrure. La clé d'une des deux serrures est publique, c'est-à-dire que tout le monde peut l'obtenir, tandis que l'autre est privée, et seule une personne la possède. Par exemple, Bob décide d'envoyer un message secret à Alice qui possède une telle boîte à deux serrures. Dans ce cas, Bob met dans la boîte d'Alice son message et referme la boîte avec la clé publique. Seule Alice pourra ouvrir la boîte, puisqu'elle est seule à posséder la clé privée. Elle seule pourra ainsi lire le message de Bob. Si Ève intercepte le message, elle ne pourra le déchiffrer sans avoir la clé privée d'Alice.

Par ailleurs, la même technique permet de signer un message : Alice le met dans la boîte qu'elle referme à l'aide de sa clé privée. Tous les destinataires pourront alors ouvrir le message et seront certains que celui-ci provient d'Alice, car elle est la seule à posséder la clé privée.

Le but de la cryptologie asymétrique est donc de construire un « coffre à deux serrures » virtuel. Un exemple est la méthode RSA, inventée par Ron Rivest, Adi Shamir et Len Adleman en 1977.

– Avant de recevoir des messages cryptés, Alice doit créer des clés. Pour cela, elle choisit deux nombres premiers p et q, un nombre e premier avec $(p-1)(q-1)$ et deux nombres d et m tels que $e \times d + m \times (p-1)(q-1) = 1$ – ces deux derniers nombres se calculent avec l'algorithme de Bezout. Elle calcule enfin le nombre $n = p \times q$.

– Alice rend publics les nombres n et e, par exemple en les publiant dans un annuaire, sur un site web ou bien en les communiquant directement à l'expéditeur, Bob. Elle conserve secrètement les nombres p, q et d, et peut même détruire p et q, qui ne serviront plus à rien. Ainsi, la clé publique est constituée par les nombres e et n tandis que la clé privée est d.

– Bob, qui veut transmettre une information secrète à Alice, transforme son information en un nombre entier A inférieur à n, ou en plusieurs, s'il est trop long.

– Bob élève ensuite son message A à la puissance e modulo n. Il envoie ce nombre, que nous notons B, de façon non protégée.

– Alice élève alors le nombre B que Bob vient de lui envoyer à la puissance d, modulo n, qui est sa clé secrète, et obtient le message original que Bob lui a envoyé, car, d'après le petit théorème de Fermat, $B^d = A^{ed} = A$, modulo n.

Bob transforme son message en fermant la serrure publique : en élevant le message, éventuellement découpé en plusieurs morceaux, à la puissance e modulo n. Une fois cette opération effectuée, le message n'est plus compréhensible. La seule façon de retourner au message initial est de posséder la clé privée d. Pour obtenir le message original et ouvrir la serrure privée, il s'agit simplement d'élever le message obtenu à la puissance d modulo n.

Signature

Parfois, on ne souhaite pas crypter le message, mais on souhaite s'authentifier, c'est-à-dire garantir que c'est bien Alice qui a envoyé le message et non pas Ève. Par exemple, la date d'un rendez-vous n'a pas forcément besoin d'être secrète, mais Bob veut être sûr d'avoir rendez-vous avec Alice et pas avec Ève. L'algorithme précédent garantit en prime l'identité de l'envoyeur, mais il est assez coûteux, on veut donc signer un message sans le crypter complètement. Une solution simple est de crypter le condensat du message avec l'une des méthodes précédentes.

Ainsi, Alice envoie son message et le condensat de ce message crypté avec une clé symétrique qu'elle partage avec Bob ou cryptée avec sa clé privée. Bob récupère le message et calcule son condensat. Il vérifie alors à l'aide de la clé commune ou de la clé publique d'Alice que le condensat crypté est bien le condensat du message qu'il a reçu. Non seulement cela authentifie le message

comme venant d'Alice, mais cela authentifie également le message, puisque si Ève avait modifié le message, cela aurait également modifié le condensat et Bob se serait rendu compte de la supercherie. Bob peut donc être sûr de l'auteur et du contenu du message. Cette technique est très peu coûteuse : en effet, calculer le condensat est rapide et le crypter également, puisque c'est un message court.

Protection des données et persistance de l'information

Discutons rapidement quelques aspects moins techniques de la structuration et du contrôle de l'information.

Persistance de l'information Lorsque nous stockons une donnée dans un fichier ou dans une base de données, nous comptons sur le fait qu'elle reste accessible sans limite.

Dans le cas d'un fichier, la gestion de la persistance des données est donc confiée au système d'exploitation, mais la préservation de la structure du fichier reste aux risques et périls de l'utilisateur – un logiciel mal conçu peut faire une erreur en ajoutant une donnée à un fichier et détruire l'ensemble des données de ce fichier, devenu illisible.

Dans le cas d'une base de données, on veut éviter ce type de problèmes. Le système de gestion de la base de données doit pouvoir redémarrer dans un état cohérent : celui dans lequel il était avant le début de cette opération. Pour cela, on doit prévoir, avant chaque opération, une sauvegarde des données, à laquelle il est possible de revenir en cas de problème. Pour se protéger, en même temps, des risques de disparition des données due par exemple à un incendie – accidentel ou intentionnel – ou une inondation des locaux dans lesquels se trouve l'ordinateur sur lequel ces données sont stockées, on effectue souvent cette sauvegarde, par exemple sur une machine distante.

Hypermnésie du Net Mais cette persistance de l'information a aussi un aspect moins désirable. Une fois qu'une information est stockée sur un disque dur, sauvegardé régulièrement, ou sur une page web publique – et donc immédiatement archivée par plusieurs moteurs de recherche, comme Google –, il devient impossible de l'effacer, car il restera toujours de nombreuses copies, ici ou là.

Recopier un bit d'information coûte en effet très peu d'énergie. En théorie, il semble que la borne inférieure soit $kT\ln(2)$ soit de l'ordre de 10^{-21} joules. En pratique, l'énergie dissipée par le basculement d'un transistor est de l'ordre de 10^{-14} joules, c'est beaucoup plus, mais cela reste très petit devant, par exemple, les 9.81 joules qu'il faut pour monter un kilogramme d'un mètre. Il ne faut donc pas s'étonner que de telles copies soient fréquentes.

Il faut être conscient du fait que, lorsqu'une photo ou un texte est publié sur la page d'un réseau social, elle peut être consultée non seulement par beaucoup de gens, mais il est presque impossible de l'enlever. Cette photo ou ce texte sera peut-être encore accessible dans cent ans.

Quand un site recopie des informations depuis un autre site, une « règle de bonne conduite » devrait être de supprimer l'information, quand elle est supprimée sur le site original, de manière à limiter la persistance de l'information. Certains imaginent que la loi devrait garantir un « droit à l'oubli », en interdisant de conserver des informations au-delà d'un certain délai, ou, au moins, en encadrant la manière dont elle est conservée.

Protection de la vie privée Bien que les outils juridiques de protection de la vie privée existent depuis plus de trente ans, il n'est pas toujours facile de faire table rase de son passé numérique. L'enjeu est de taille, d'autant que l'exploitation de ces données personnelles représente un potentiel inestimable sur lequel reposent de nombreux modèles économiques sur le Net. Des solutions techniques et juridiques existent, mais aucune n'est parfaite.

Les solutions techniques sont fondées soit sur la destruction des données, soit sur une anonymisation irréversible : mode de navigation privé, possibilité pour l'utilisateur de définir lui-même la durée de conservation de certaines informations…

Du côté des solutions juridiques, la loi Informatique et Liberté nous protège contre l'utilisation abusive de bases de données puisque, selon la loi, les données « sont collectées pour des finalités déterminées, explicites et légitimes, et ne sont pas traitées ultérieurement de manière incompatible avec ces finalités » et « sont conservées sous une forme permettant l'identification des personnes concernées pendant une durée qui n'excède pas la durée nécessaire aux finalités pour lesquelles elles sont collectées et traitées ».

Comme toute loi, celle-ci nous protège, mais elle nous contraint aussi, par exemple, à déclarer à la CNIL nos propres bases de données.

Non-rivalité de l'information L'information a cette propriété que la consommation du bien par un agent n'empêche pas sa consommation par un autre – les agents ne sont pas rivaux pour la consommation du bien. Par exemple, le fait que quelqu'un écoute la radio n'empêche pas les autres de le faire. De même, sur le Web, il y a non-exclusion : tous les agents ont librement accès au bien.

La manière de diffuser des biens non rivaux est différente de la manière de diffuser des biens rivaux. Par exemple, la gratuité pour les biens rivaux entraîne souvent la pénurie, mais ce n'est pas le cas pour des biens non rivaux. Quand des biens rivaux deviennent non rivaux et gratuits, logiciels, œuvres d'art, etc.,

il faut souvent repenser entièrement la manière dont leurs producteurs sont rémunérés.

Exercices corrigés et commentés

Exercice 1. Comprendre ce que l'on tape dans la barre de navigation

En utilisant l'entrée de Wikipédia
`http://fr.wikipedia.org/wiki/Uniform_Resource_Locator`
rendre explicite chaque élément d'un URL (*Uniform Ressource Locator*, Localisateur uniforme de ressource) qui permet d'accéder à une ressource sur Internet. Illustrer avec l'exemple l'URL ci-dessous.

```
http://me:maux2passe@mabanque.fr/compte/retrait
?montant=10000&beneficiaire=toi
```

Correction. *Méthode*. Il se trouve que l'on ne peut pas toujours tout savoir. On ne peut même pas toujours savoir où tout trouver.

En posant un tel type d'exercice, on invite les élèves à aller à la recherche de connaissances qui leur permettent de comprendre ce qu'ils tapent ou cliquent. Dans ce cas, chacun tape quotidiennement de telles chaînes de caractères, dont on voit confusément qu'elles doivent avoir un sens caché – on y reconnaît probablement `.fr` pour France, on devine que `http` doit être un standard, etc. Il suffit de communiquer un URL à une personne qui connaît le codage sous-jacent et à une autre qui ne le connaît pas pour mesurer la différence : la première a juste besoin de mémoriser les éléments particuliers à l'URL, la seconde doit le noter caractère par caractère, avec l'angoisse de se tromper.

En posant un tel type d'exercice, on invite l'élève à aller à la recherche de connaissances qu'il ne connaît certes pas encore, mais qu'il ne sait pas non plus forcément comment chercher. Pour ce qui concerne l'informatique et les mathématiques qui y sont liées, la démarche est le plus souvent simplifiée par Wikipédia. Le premier geste n'est plus d'aller chercher sur Google, mais de demander à Wikipédia. La version française est bien fournie, la version anglaise est parfois plus riche. Les liens internes et externes de bas de page y sont précieux.

Mais ce n'est pas le seul recours. Le *Site du zéro* `http://www.siteduzero.com` – où tout est expliqué… à partir de zéro – est très riche de contenus plus proches de l'informatique technique, tandis que *Interstice* `http://interstices.info` – où l'informatique devient une science… à portée de clic – est un site précieux de ressources culturelles. Un site de ressources documentaires pour les enseignants `http://science-info-lycee.fr` va compléter ces grands silos de contenus.

Comment passer concrètement à l'appropriation de connaissance ? Peut-être en « remalaxant » l'information. À l'ère où l'on zappe et survole souvent les choses, le moyen de ne pas en rester à « j'ai déjà vu ça quelque part » est sûrement de mettre ses propres mots dessus, faire *son* résumé des faits et données collectées. On pense avec des mots. On peut aussi penser avec des éléments structurés, rédigés sous forme de notes écrites. La seule règle à s'imposer : ne pas utiliser le copier-coller, qui empêche de s'approprier le sens du texte.

Contenu. Un URL est une chaîne de caractères structurée combinant les informations nécessaires pour indiquer à un logiciel comment accéder à une ressource Internet et qui se décompose ainsi :

```
URL: $protocole ://$connection(/$chemin($paramètres)?)?
```

– Le *protocole* de communication, `http` pour le Web, spécifie les règles sémantiques et syntaxiques pour initier, paramétrer, effectuer et contrôler un échange automatique de données entre deux systèmes d'information.

– La *connexion* définit quel serveur connecter, la spécification complète étant : `connection :($login(:$mot-de-passe)?@)?($nom-du-serveur|$adresse-ip)(:$port)?` permet de spécifier un identifiant `login` avec éventuellement un mot de passe `mot-de-passe` – qui est transmis en clair, donc à un niveau de sécurité assez bas –, de spécifier le serveur sous son nom de domaine, le nom du serveur ou son adresse Internet, son adresse IP, voire de préciser le numéro de port TCP/IP à utiliser, au cas où le même serveur possède des services n'utilisant pas le port par défaut pour le protocole de communication.

Le nom du serveur est lui-même structuré :

```
nom-du-serveur:($nom(.$nom-de-sous-domaine)*.$nom-du-domaine)
```

où les noms de domaine constituent une énumération standardisée :

```
nom-du-domaine : (info|org|com|fr|…)
```

pour les sites d'information `*.info`, de grandes organisations `*.org`, commerciaux `*.com`, français `*.fr`, etc.

– Le *chemin* est de la forme :

```
chemin : (./|(../)+)?($repertoire/)*($nom-de-base($extension)?)?
```

et permet de localiser la ressource au sein du serveur, exactement comme dans un système de fichiers. C'est un chemin à travers un arbre. L'extension est étroitement liée au type de la ressource, par exemple pour un document bureautique `.doc` ou `.odt`, pour une image `.png` ou `.jpg`, etc.

– Les *paramètres* se spécifient de deux manières :

```
paramètres : (#fragment|[?]requête)
```

soit sous forme de `fragment`, l'identificateur faisant référence à un identificateur à l'intérieur du document, soit sous forme de `requête`

```
requête : nom=valeur([&]nom=valeur)*
```

qui est un n-uplets à champs nommés permettant de paramétrer la requête. On peut par exemple référencer une page précise d'un pdf avec une construction de la forme `http://mon-serveur/le-chemin/le-document.pdf#page=11`, ou interroger une base de données, comme cela sera détaillé au chapitre suivant.

Au-delà du protocole `http`, on peut accéder à un fichier local par le protocole `file`, ou via d'autres protocoles comme le `ftp`.

Plus généralement, un URI (*Uniform Ressource Identifier*, Identificateur uniforme de ressource) permet de spécifier une ressource, en utilisant un protocole de communication autre que `http`. Ainsi

```
mailto:prenom.nom@serveur?subject=mon-sujet&body=Bonjour
```

définit un message à envoyer à une adresse électronique. L'accès à des services Internet peut se faire en définissant des URI.

Exercice 2. Chiffrement par substitution

Chiffrer par le codage de César et en ROT13 la phrase :

Bob veut envoyer un message secret a Alice.

Correction

Bob veut envoyer un message secret a Alice.

César Ere yhxw hqyobhu xq phvvdjh vhfuhw d Dolfh.

ROT13 Obo irhg raiblre ha zrffntr frperg n Nyvpr.

Exercice 3. Chiffrement par ou exclusif

On considère le chiffrement symétrique basé sur l'opérateur ou exclusif. Que penser des clés 0…0 et 1…1 ?

Crypter avec la clé composée de la représentation binaire de π le message « Salut » crypté en ASCII-7 bits.

Correction

Considérons un message M. Le cryptage avec la clé nulle donne le message initial : $M \oplus 0\ldots0 = M$. Le cryptage avec la clé 1…1 donne $M \oplus 1\ldots1 = M$, le complément, bit à bit, de M. Ce sont donc deux clés extrêmement faibles.

Message	S	a	l	u	t
ASCII	1010011	1100001	1101100	1110101	1110100
Clé : π	1100100	1000011	1111101	0001010	1011010
Message crypté	0110111	0100010	0010001	1111111	0101110

Exercices non corrigés

Exercice 1

Définir un format de donnée symbolique correspondant aux données permettant de s'inscrire sur `facebook` – ou toute autre plateforme du Web. Définir celui correspondant à un rendez-vous dans un agenda. Définir celui correspondant aux métadonnées d'un album de photos de vacances.

Exercice 2

En analysant les entrées de Wikipédia `http://fr.wikipedia.org/wiki/OpenDocument` et `http://fr.wikipedia.org/wiki/DocBook`, expliquer en deux ou trois lignes comment ces deux standards s'articulent.

Exercice 3

Prendre un fichier de grande taille et appliquer plusieurs fois un mécanisme de compression, avec des commandes de type :

```
gzip -9 mon-fichier; mv mon-fichier.gz mon-fichier; ls -l mon-fichier
```

et observer que la taille du fichier va, en général, diminuer, puis croître légèrement, avant de se stabiliser. Expérimenter s'il est toujours possible de retrouver exactement le fichier originel en faisant les opérations inverses de décompression.

Exercice 4. Inexistence d'une méthode de compression universelle

Un compresseur sans perte peut être vu comme une injection de l'ensemble des fichiers dans lui-même. Montrer qu'il est impossible qu'une telle fonction compresse strictement tous les fichiers d'une taille donnée.

Indication : considérer les puissances itérées de cette fonction.

Plus difficile : montrer que si une fonction compresse strictement au moins un fichier, alors il existe un fichier qu'elle allonge strictement.

Exercice 5

Écrire un code qui construit l'arbre du codage d'Huffman au fur et à mesure de l'arrivée d'un flux de caractères et l'appliquer à un mécanisme de compression-décompression. Pour simplifier la programmation, on pourra « tricher » en utilisant l'arbre du compresseur dans le décompresseur pour valider le reste de l'algorithme. Puis, on pourra coder deux algorithmes indépendants et vérifier que les arbres de codages restent bien identiques.

Exercice 6

En utilisant l'entrée `http://fr.wikipedia.org/wiki/Indicatif_téléphonique_international` expliquer en quoi nous sommes devant un code préfixe. Si l'on regarde les préfixes des grandes nations émergentes, ce code est-il optimal ?

Exercice 7

Étude du code de Reed-Solomon. Prenons trois nombres u, v, w et transmettons u, v, w avec $s = u + v + w$ et $m = u + 2v + 3w$. À la réception, en supposant que au plus un des nombres est erroné :
– calculer à partir de s la valeur de cette erreur e ;
– déterminer, à partir de m et e, lequel des trois nombres est en erreur.

En déduire l'algorithme de correction.

Questions d'enseignement

À ce stade de notre parcours, nous avons assez de recul pour proposer brièvement quelques clés pédagogiques sur l'enseignement de l'informatique en général.

L'informatique se prête à une pédagogie participative, avec un enseignement par mini-projets, orienté vers le travail en groupe. Apprendre à programmer un petit logiciel, c'est donner à l'élève des clés, mais aussi la liberté de s'approprier ces clés et de les mettre en pratique de manières diverses, car il y a plusieurs manières de mettre en œuvre une solution. Ce n'est pas la seule matière qui offre ce levier, mais c'est un cas exemplaire.

L'informatique favorise l'apprentissage par l'utilisation, ce qui correspond bien à l'esprit humain – par exemple : découvrir un algorithme avant d'en abstraire la notion sous-jacente. L'enseignement scolaire de l'informatique démarre bien sûr par l'apprentissage des usages de logiciels, mais, dès cette étape, il est important de comprendre les concepts sous-jacents à l'utilisation de ces logiciels.

Le levier pédagogique est l'apprentissage de la programmation. Apprendre à programmer permet de faire le lien entre pratique et théorie. Programmer les algorithmes permet non seulement de vérifier que les savoirs sont compris, mais également d'entrevoir toutes les potentialités de l'informatique. Voir François Élie, Bastien Guerry, Dominique Lacroix, Philippe Lucaud, Charlie Nestel, Cécile Picard-Limpens, Thierry Viéville, « Est-il besoin de savoir programmer pour comprendre les fondements de l'informatique ou utiliser les logiciels ? », *Revue de l'EPI*, a1012b.

L'informatique conduit aussi à un apprentissage de la rigueur, par un mécanisme spécifique : celui des essais-erreurs avec une machine neutre qui ne donnera un résultat que si tout est correct, mais qui donnera indéfiniment une chance de corriger, de reprendre, de tester à nouveau. La machine est un outil pour apprendre de manière incrémentale, sans jamais porter de jugement de valeur.

L'informatique permet d'entrevoir l'intérêt des sciences théoriques. On peut toucher – opérer avec, visualiser… – des objets abstraits.

L'informatique est un levier pour les autres sciences, car elle aide à mieux comprendre des notions universelles – par exemple la notion d'information – ou fondamentales – par exemple le calcul mécanique, par opposition à d'autres formes de raisonnement. L'informatique offre aussi la découverte de notions nouvelles, par exemple les suites aléatoires, les définitions récursives, etc.

C'est en apprenant l'informatique le plus tôt possible que l'on tirera le meilleur profit de son rôle transversal à la quasi-totalité des autres disciplines.

Les enjeux de société sont évidents : donner aux futurs citoyens les clés du monde numérique, pour leur permettre de se positionner de manière éclairée sur les choix de société posés par ces nouvelles technologies, par exemple découvrir comment les nouvelles technologies aideront à relever les grands défis qui se posent à nous. Mais ces questions de société sont toujours ancrées dans des questions scientifiques et techniques, dont elles ne doivent pas être séparées.

Les enjeux en termes de formation de l'esprit – en lien avec l'enseignement de philosophie – sont aussi fondamentaux, les éléments de ce chapitre et du premier chapitre le montrent assez.

Ces éléments s'opposent aussi à quelques idées reçues :

L'informatique n'est pas une science, juste de la technique ? Ce qui fait marcher les programmes est bien de la théorie : algorithmique, théorie des langages de programmation, etc. Ce sont des logiciens et des physiciens théoriques du siècle dernier qui sont à l'origine de l'informatique. Les questions de sûreté sont exemplaires : c'est de l'étude scientifique des logiciels qu'émergent aujourd'hui les méthodes permettant de concevoir des logiciels sûrs.

Pas besoin d'apprendre l'informatique, cela s'apprend tout seul ? On dit souvent : ce sont les enfants qui apprennent l'informatique aux enseignants ! Cette double idée reçue est la conséquence d'une confusion entre l'apprentissage des usages et l'apprentissage des fondements. Cette idée reçue néglige aussi le fait que, depuis quarante ans, l'informatique s'est stratifiée et complexifiée : on ne peut plus y bricoler. Le mythe de l'auto-apprentissage se brise devant la nécessité d'apprendre au plus grand nombre des savoirs et des pratiques qui doivent être intégrés à l'échelle d'une société entière. Le risque de mal apprendre et de devoir passer des heures à se corriger, les risques liés aux mauvaises méthodes – perte de données, logiciels non fiables… – deviennent majeurs : l'enseignement de l'informatique en tant que matière rigoureuse est une nécessité.

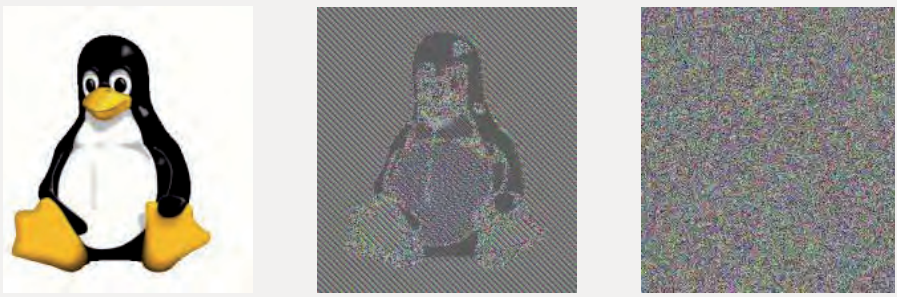
Et nous ne pouvons pas ne pas noter aussi :

L'informatique n'est pas très féminine. C'est bien la mixité des genres qui aide à créer la mixité et l'ouverture des idées… et l'informatique a quelques grands noms féminins : Ada Lovelace – premières notions de programmes et de traitement symboliques –, Emmy Noether – processus algébriques permettant de mécaniser des calculs –, Grace Hopper – langage COBOL, mémoires tampons –, Rose Dieng – ontologies informatiques pour le Web sémantique –, même si la situation reste déséquilibrée en France, contrairement à d'autres pays, comme la Thaïlande, où 65 % des étudiants et des professeurs en informatique sont des étudiantes et des professeures.

Compléments

Une question que nous n'avons pas abordée est celle du codage de messages longs. En général, les techniques sont adaptées au codage de blocs de quelques centaines/milliers de bits. Mais on peut vouloir crypter un fichier plus long, comme une vidéo. Dans ce cas, la méthode naturelle est celle du dictionnaire de codes : ECB (*Electronic codebook*). Elle consiste à découper le message en blocs, à chiffrer chaque bloc avec la méthode et la clé choisies, à recoller les morceaux chiffrés et à envoyer le tout. Le receveur pourra alors redécouper en blocs, les décrypter et recoller les bouts obtenus. Ce cryptage par bloc est en fait facilement attaquable : deux blocs avec le même contenu seront chiffrés de la même manière. On peut donc tirer des informations à partir du texte chiffré en cherchant les séquences identiques. On obtient dès lors un « dictionnaire de codes » avec les correspondances entre le clair et le chiffré.

Voici un exemple frappant en image issu de Wikipédia. L'image de gauche a été cryptée en utilisant la méthode ECB décrite précédemment pour obtenir l'image du milieu. Une autre méthode a été utilisée pour obtenir l'image de droite.



Cryptage d'une image par la méthode du dictionnaire de codes (ECB) ou par une autre méthode.

Pour plus de détails et la description d'autres méthodes, on pourra se référer à Mode d'opération (cryptographie).

`http://fr.wikipedia.org/wiki/Mode_d'opération_(cryptographie)`

Pour aller plus loin

Sur le cryptage : François Cayre, Cryptographie, du chiffre et des lettres
`http://interstices.info/crypto-chiffre-lettres`
)i(nterstices, 2009.

Sur les codes secrets : Mathieu Cunche, À l'attaque des codes secrets
`http://interstices.info/codes-secrets`
)i(nterstices, 2011.

Sur le cryptage RSA : Jonathan Touboul, Nombres premiers et cryptologie l'algorithme RSA
`http://interstices.info/rsa`
)i(nterstices, 2007.

Sur la didactique de l'informatique en général, Werner Hartmann, Michael Näf, Raimond Reichert, *Informatikunterricht planen und durchführen*, Planifier et réaliser un cours d'informatique, Springer Verlag, 2006, dont une traduction française a été publiée en octobre 2011, est une référence.

Bases de données relationnelles et Web

Dernière étape de notre parcours : les bases de données. Parce que la plupart des bases de données sont aujourd'hui accessibles sur le Web, et parce que la plupart des sites web sont, en fait, l'interface de bases de données, c'est dans ce chapitre également que le Web est abordé. Cette thématique des bases de données nous permettra aussi d'approfondir un thème qui apparaissait déjà, mais de manière plus discrète, dans les chapitres « Représentation numérique de l'information » et « Structuration et contrôle de l'information », celui de l'indépendance des données et des programmes : ce sont les programmes qui s'adaptent aux données et non le contraire. Enfin, dans ce chapitre, nous mettrons en œuvre un projet plus ambitieux que dans ceux qui précédent : la réalisation d'un site web complet, qui permettra au lecteur de mettre en perspective son parcours depuis qu'il a ouvert ce livre à la première page.

Cours

Lorsque l'on souhaite faire persister des informations dans des applications complexes, c'est-à-dire que les données durent au-delà de la durée d'exécution du programme, on doit écrire ces informations sur une mémoire de masse, par exemple un disque dur, dans un fichier ou dans une base de données. La différence entre un fichier et une base de données est que les informations sont en vrac dans un fichier alors que les bases de données tirent profit de la structure des informations, que nous avons introduite au chapitre précédent. Les bases de données sont gérées par des logiciels appelés *Systèmes de gestion de bases de données* (SGBD) qui permettent un accès direct à des données précises, par exemple, trouver l'âge de *Jean Pierre,* sans demander une lecture et une analyse de toutes les informations contenues dans un fichier, pour obtenir celle qui nous intéresse. En outre, les bases de données évitent la perte des informations en cas de pannes logicielles, erreurs dans les programmes, ou matérielles, comme la destruction d'un disque.

Les systèmes de gestion de bases de données sont partout autour de nous : la plupart des sites web utilisent des systèmes de gestion de bases de données relationnels pour stocker leurs informations, les systèmes d'informations des banques sont gérés par des systèmes de gestion de bases de données, le Trésor public utilise des systèmes de gestion de bases de données pour sa comptabilité, etc. Le type de logiciels utilisé est le même, malgré la diversité de la structure des données et des données elles-mêmes. Il est impensable aujourd'hui de réaliser un logiciel de gestion d'informations sans avoir recours à une base de données, notamment pour garantir la persistance de l'information.

Rappelons également que les données peuvent être en quantité importante : gigaoctets – 10^9 octets –, téraoctets – 10^{12} octets –, voire pétaoctets – 10^{15} octets. Ces données peuvent également être de formats très différents : nombres, textes, images, vidéos, etc. Gérer ces données implique de pouvoir les stocker, les retrouver et les mettre à jour de manière efficace.

La plupart des données manipulées sont structurées. Les données peuvent être élémentaires comme un nom ou un prénom, composées comme une adresse, voire multivaluées comme les numéros de téléphone d'une personne. Elles décrivent des entités. Le lien entre une entité et une donnée s'appelle une relation.

Une *base de données* (BD) est donc une collection de données structurées reliées entre elles par des relations. Elle permet l'interrogation et la manipulation de ces données par des langages de haut niveau.

Comme nous l'avons vu au chapitre précédent, l'une des applications des bases de données est le Web, puisqu'il s'agit d'une immense source d'informations. Il faut noter que les données présentes sur le Web proviennent la plupart du temps de bases de données relationnelles, même si ces informations sont souvent en format « semi-structuré », c'est-à-dire (X)HTML. Nous montrons donc également dans ce chapitre comment de telles interactions se produisent.

En résumé, ce chapitre est une initiation aux bases de données *relationnelles*, aux systèmes de gestion de bases de données et aux concepts et technologies du Web. Nous expliquerons tout d'abord les concepts fondamentaux des bases de données et de leur conception, puis comment manipuler un système de gestion de bases de données pour y accéder depuis un programme. Enfin, nous réalisons une rapide présentation des concepts et technologies du Web, en montrant comment une page web s'interface avec une base de données.

Le modèle relationnel

Le *modèle relationnel* est un modèle de données s'appuyant sur la *logique des prédicats* ou *logique du premier ordre* et sur la *théorie des ensembles*. Ce modèle

suppose que toute donnée peut être représentée sous la forme d'une relation *n*-aire, c'est-à-dire à *n* arguments. Rappelons qu'une relation à *n* arguments est un sous-ensemble du produit cartésien de *n* ensembles. Dans le cas pratique, une relation est un ensemble fini. De là vient le terme de base de données « relationnelle ».

Relation, attribut et n-uplet

Informellement, dans le modèle relationnel, les données sont stockées dans des tableaux à deux dimensions appelés *tables*. L'ordre entre les colonnes n'a pas d'importance. Par exemple, on ne peut pas parler de la première ou de la deuxième colonne d'une table : chaque colonne est désignée par son nom. On appelle *attribut* le nom donné à une colonne de la table. Une ligne de la table contient des valeurs pour chacun des attributs. Chacune de ces lignes est appelée *n-uplet*, ou *tuple*. L'ensemble des n-uplets d'une table s'appelle une *relation*. Une *base de données relationnelle* (BDR) est formée d'un ensemble de relations.

Équipes

nom	nomCourt	ville
Paris Saint-Germain	PSG	Paris
Olympique de Marseille	OM	Marseille
Olympique Lyonnais	OL	Lyon

Joueurs

equipe	no	nom	prénom	age
OM	1	RIOU	Rudy	31
OM	10	GIGNAC	André-Pierre	25
PSG	1	COUPET	Grégory	37
PSG	4	MAKELELE	Claude	37
OL	1	LLORIS	Hugo	23
OL	9	LOPEZ	Lisandro	26

Rencontres

date	local	visiteur	butl	butv
19/12/10	OM	OL	1	1
28/11/10	OL	PSG	2	2
7/11/10	PSG	OM	2	1

Un exemple de base de données relationnelle.

Nous allons maintenant définir ces notions plus formellement.

À chaque *attribut A* est associé un *domaine* noté $Dom(A)$, qui représente l'ensemble de valeurs possibles de l'attribut. L'ensemble des nombres entiers, l'intervalle de nombres réels [0.0, 10000.0] et l'ensemble {*rouge, vert, bleu*} constituent des exemples de domaines.

Un *schéma de relation* $R(A_1, A_2, \ldots, A_n)$ est constitué d'un nom R et d'une liste d'attributs $A_1, \ldots, A_n$. Chaque attribut A_i représente le rôle joué par le domaine $Dom(A_i)$ dans le schéma de relation R. Un *schéma de base de données* est un ensemble de schémas de relation. Un exemple de schéma de base de données relationnelles, composé de trois schémas de relations, est le suivant

```
Equipes(nom : chaîne, nomCourt : chaîne, ville : chaîne)
Joueurs(equipe : chaîne, no : entier, nom : chaîne, pré-
nom : chaîne, age : entier)
Rencontres(date : Date, local : chaîne, visiteur :
chaîne, butl : entier, butv : entier)
```

Une *relation r* de schéma $R(A_1, A_2, \ldots, A_n)$ est un sous-ensemble du produit cartésien des domaines de R : $r \subseteq Dom(A_1) \times Dom(A_2) \times \ldots \times Dom(A_n)$. On dit aussi que r est une *instance* du schéma de relation R.

Un *n-uplet* de r est noté $t = \langle v_1, v_2, \ldots, v_n \rangle$ où $v_i \in Dom(A_i)$. La valeur du n-uplet t sur les attributs $A_i, \ldots, A_j$ est notée $t[A_i, \ldots, A_j]$.

Contrainte d'intégrité

Les concepts définis jusqu'ici permettent de créer des modèles de données nommés schémas, puis de les instancier pour y conserver des valeurs dans des relations. Cependant, toutes les instances possibles d'un modèle ne sont pas forcément pertinentes. Des *contraintes d'intégrité* fournissent un moyen de restreindre les relations acceptables pour un schéma de base de données. C'est une propriété du schéma de la base de données, que doivent vérifier les instances à tout moment ; par exemple, un salaire doit être compris entre 100 et 100 000 euros. On distingue les contraintes de domaine, les contraintes d'entité qui expriment l'unicité des valeurs d'un attribut et les contraintes référentielles qui expriment les liens entre deux relations.

Une contrainte d'entité, ou *clé*, est une contrainte très courante dans le modèle relationnel. Une clé représente l'ensemble minimal d'attributs qui permet d'identifier de façon unique chaque n-uplet d'une relation. Une *superclé* K d'un schéma de relation R est un ensemble d'attributs $K = \{A_{i1}, \ldots, A_{ik}\} \subseteq R$ tel que pour toute relation r sur R et pour tout n-uplet $t, u \in r$, $t[K] = u[K] \Rightarrow t = u$.

Une *clé candidate*, appelée *superclé minimale* ou tout simplement *clé, K* de *R* est une superclé telle que si $X \subseteq K$ est une superclé de *R,* alors $X = K$. Une relation peut posséder plusieurs clés. En général, tout schéma de relation possède au moins une clé. Par exemple, les attributs `nom` et `nom_court` sont deux clés de la relation `Equipes`. La plupart du temps, on choisit de mettre en avant, pour chaque relation, une clé candidate, appelée alors *clé primaire*.

Une contrainte référentielle, ou *clé étrangère*, explicite les liens entre les tables de la base de données. Elle permet d'assurer que les relations entre tables demeurent cohérentes. Quand une table possède une clé étrangère vers une autre table, il n'est pas possible d'ajouter un n-uplet dans la première si un n-uplet correspondant n'existe pas dans l'autre. L'expression $R[A_{i1}, \ldots, A_{ik}] \subseteq S[B_{j1}, \ldots, B_{jk}]$ est une *clé étrangère* si $\{B_{j1}, \ldots, B_{jk}\}$ est une clé de *S* et si pour tout *r* sur *R* et *s* sur *S*, pour tout $t \in r$, il existe $u \in s$ tel que $t[A_{il}] = u[B_{jl}]$ pour $1 \leq l \leq k$. Une telle contrainte impose que, lors d'une insertion dans la table *r*, la valeur des attributs $A_{i1}, \ldots, A_{ik}$ doit exister dans la table *s*. De même, lors d'une modification dans la table *s*, les n-uplets de *r* doivent être mis à jour de façon adéquate.

Reprenons et complétons l'exemple ci-avant afin d'illustrer l'ensemble de ces notions. La relation `Equipes` possède les deux clés {*nom*} et {*nom_court*}, la relation `Joueurs` {*equipe,no*} et {*nom,prenom*}, la relation `Rencontres` {*date, local, visiteur*}. Le lien entre `Joueurs` et `Equipes` est formalisé par la clé étrangère *Joueurs*[*equipe*] ⊆ *Equipes*[*nom_court*]. Les liens entre `Rencontres` et `Equipes` sont *Rencontres*[*local*] ⊆ *Equipes*[*nom_court*] et *Rencontres*[*visiteur*] ⊆ *Equipes*[*nom_court*].

L'algèbre relationnelle

Le modèle relationnel a permis le développement de plusieurs langages d'interrogation de données. Certains de ces langages sont théoriques, basés sur la logique – *calcul relationnel de n-uplets, calcul relationnel de domaines, datalog* – ou sur la théorie des ensembles – *algèbre relationnelle*. D'autres sont des implémentations plus ou moins standard comme *SQL* qui s'appuie sur l'algèbre relationnelle et le calcul de n-uplets ou *Query by Example* (*QBE*) issu du calcul de domaines. Dans ce paragraphe, nous nous intéressons à l'algèbre relationnelle qui a des applications directes dans les systèmes de gestion de bases de données : elle permet de définir une sémantique simple des requêtes, en facilitant des optimisations pour leur évaluation.

L'algèbre relationnelle est une collection d'opérateurs algébriques, chaque opérateur ayant une ou deux relations en entrée et une relation en sortie, qu'il est possible de composer, cette possibilité étant la *propriété de fermeture* de l'al-

gèbre. Les principaux opérateurs de l'algèbre relationnelle sont la *sélection* (σ) qui extrait un sous-ensemble de n-uplets d'une relation, la *projection* (π) qui extrait un sous-ensemble des attributs d'une relation, le *renommage* ($\rho_{A \to B}$) qui renomme un attribut, le *produit cartésien* ($\times$) qui fait le produit cartésien de deux relations, la *jointure* ($\bowtie$) qui combine deux relations selon un certain critère, la *division* ($\div$) qui exprime le quantificateur universel $\forall$ et l'*agrégation* (γ) qui combine plusieurs n-uplets en un seul. En plus de ces opérateurs, l'algèbre relationnelle fait aussi appel aux opérateurs ensemblistes d'union, d'intersection et de différence. Nous détaillons maintenant ces opérateurs.

Opérateurs de l'algèbre relationnelle

La *sélection* σ, ou *restriction*, permet de sélectionner un sous-ensemble des n-uplets d'une relation vérifiant une condition booléenne. Cette dernière est une *formule de sélection* composée de constantes, de noms d'attributs, d'opérateurs de comparaison, d'application de fonctions et des opérateurs logiques $\vee$ (ou), $\wedge$ (et), et $\neg$ (non). Soit r une relation sur R et F une formule de sélection, la sélection $\sigma_F(r)$ est définie par $\sigma_F(r) = \{t \in r \mid t \text{ satisfait } F\}$. Par exemple, la requête « les joueurs de plus de 30 ans » s'écrit $\sigma_{age>30}(Joueurs)$. Il faut noter que cette condition est évaluée pour chacun des n-uplets. Il s'agit donc d'une condition locale.

La *projection* π permet de sélectionner les n-uplets correspondant à un sous-ensemble des attributs d'une relation. Soit r une relation sur R et $X \subseteq R$, la projection de r sur X est définie par $\pi_X(r) = \{t[X] \mid t \in r\}$. Par exemple, la requête « nom et prénom des joueurs » s'écrit $\pi_{\{nom,prenom\}}(Joueurs)$.

Le renommage ($\rho_{A \to B}$) renomme un attribut. Soit r une relation sur R, $A \in R$ et $B \notin R$, la relation r avec A renommé en B est définie comme $\rho_{A \to B}(r) = \{t \mid \exists u \in r \text{ tel que } t[B]=u[A] \wedge t[C] = u[C] \text{ si } C \neq B\}$.

À partir de deux relations r sur R et s sur S, le *produit cartésien* forme une relation dont les n-uplets sont la concaténation des n-uplets de r avec ceux de s, c'est-à-dire $r \times s = \{t \mid t[R] \in r \wedge t[S] \in s\}$.

La *jointure*, quant à elle, forme une relation dont les n-uplets sont la concaténation des n-uplets de r avec ceux de s qui vérifient la condition de jointure F, c'est-à-dire $r \bowtie_F s = \{t \mid t[R] \in r \wedge t[S] \in s \wedge t \text{ satisfait } F\}$. La jointure peut aussi s'exprimer comme un produit cartésien suivi d'une sélection, c'est-à-dire $r \bowtie_F s = \sigma_F(r \times s)$.Par exemple, la requête « nom des joueurs et nom et ville de leur équipe » s'écrit $\pi_{\{joueur,nom,ville\}}(\rho_{nom \to joueur}(Joueurs) \bowtie_{equipe=nomCourt} Equipes)$ ou de façon équivalente $\pi_{\{joueur,nom,ville\}}(\sigma_{equipe=nomCourt}(\rho_{nom \to joueur}(Joueurs) \times Equipes))$. Si on ne précise pas de condition, la jointure se fait sur les attributs communs aux deux relations. On parle alors de *jointure naturelle*. Dans ce cas, les

colonnes connectées ne sont pas répétées. Si la condition est une égalité, on parle d'*équijointure*. Enfin, si $r = s$, il s'agit d'une *autojointure*.

La *division* ÷ permet d'obtenir une relation dont le produit cartésien avec le diviseur est un sous-ensemble du dividende. Cet opérateur permet de répondre à des requêtes du type « Quels sont les joueurs qui ont joué dans toutes les rencontres de leur équipe ? ». Formellement, si $R(A_1, \ldots, A_n, A_{n+1}, \ldots, A_m)$ et $S(A_{n+1}, \ldots, A_m)$ sont deux schémas, la division de deux relations r sur R et s sur S est définie par $r \div s = \{t \mid \forall t' \in s, (t, t') \in r\}$. La division peut être obtenue à partir de la différence, du produit cartésien et de la projection : $r \div s = u \setminus v$ où $u = \pi_{\{A1, \ldots, An\}}(r)$ et $v = \pi_{\{A1, \ldots, An\}}((u \times s) \setminus r)$.

L'*agrégation* (γ) résume une relation en regroupant certains n-uplets en un seul. Cet opérateur permet de répondre à des requêtes du type « nombre moyen de buts marqués à domicile par chaque équipe ». Pour fusionner un ensemble de n-uplets, on utilise des fonctions spéciales appelées *fonctions d'agrégation* dont les plus courantes sont le comptage (*COUNT*), la somme (*SUM*), la moyenne (*AVG*), le minimum (*MIN*) et le maximum (*MAX*). Soit un schéma $R(A_1, \ldots, A_n)$, l'expression ${}_{B1, \ldots, Bm}\gamma_{F1(C1), \ldots, Fp(Cp)}(r)$ où B_i et C_i sont des attributs de R construit une relation de schéma $B_1, \ldots, B_m, F_1(C_1), \ldots, F_p(C_p)$. Les attributs $B_1, \ldots, B_m$ sont des *attributs de groupement*. L'opérateur γ commence par rassembler les n-uplets de r qui ont même valeur sur $B_1, \ldots, B_m$. Ensuite, pour chaque ensemble de n-uplets, les fonctions d'agrégation $F_1(C_1), \ldots, F_p(C_p)$ sont appliquées. L'exemple de requête donnée ci-avant s'écrirait donc ${}_{local}\gamma_{AVG(butl)}$ (*rencontres*).

À partir de deux relations r et s, l'*union* produit une relation contenant les n-uplets appartenant à r ou à s, $r \cup s = \{t \mid t \in r \vee t \in s\}$. La *différence* produit une relation contenant les n-uplets appartenant à r mais pas à s, $r \setminus s = \{t \mid t \in r \wedge t \notin s\}$. Enfin, l'*intersection* produit une relation contenant les n-uplets appartenant à r et à s, $r \cap s = \{t \mid t \in r \wedge t \in s\}$. Ces opérateurs ne sont définis que des relations r et s de même schéma.

Le langage SQL

L'algèbre relationnelle fait partie des fondements des bases de données relationnelles. C'est un modèle sémantique utilisé en interne par le système de gestion de bases de données pour manipuler les requêtes. Le langage SQL, quant à lui, est destiné aux utilisateurs de systèmes de gestion de bases de données. Il est, aujourd'hui, le langage standard d'interrogation des bases de données. Il est fondé sur l'algèbre relationnelle et le calcul relationnel de n-uplets. Il a été standardisé par l'*ANSI* en 1986, puis par l'ISO en 1989, 1992, 1999, 2003 et 2008. Ces standards successifs ont étendu les capacités du lan-

gage. En plus de ces standards, SQL possède de nombreux dialectes spécifiques à chaque système de gestion de bases de données. En effet, ces systèmes se sont développés parallèlement à la standardisation de la norme SQL, voire avant. En conséquence, seul un sous-ensemble de la norme est supporté par la grande majorité des systèmes, en particulier les systèmes de gestion de bases de données libres. Ce sous-ensemble correspond à la norme SQL-92. C'est cette version que nous présentons ici.

La norme SQL-92 se décompose en plusieurs parties. L'interrogation des données se fait par l'opération de *sélection* (`SELECT`), dont la sémantique est donnée par une expression de l'algèbre relationnelle. Le *langage de manipulation de données* (*Data Manipulation Language ou DML*) permet de modifier les données grâce à l'*insertion* (`INSERT`), la *mise à jour* (`UPDATE`) et la *suppression* (`DELETE`) de n-uplets. Le *langage de définition de données* (*Data Definition Language ou DDL*) se charge de la définition du schéma d'une base de données. Enfin, SQL-92 inclut d'autres opérations comme la gestion des droits ou celle des transactions.

Le langage de définition de données

Le langage de définition de données permet la modification du schéma d'une base de données. Il propose trois opérations : la création (`CREATE`), la suppression (`DROP`) et la modification (`ALTER`). Par exemple, la création d'une table a pour syntaxe : `CREATE TABLE ⟨table⟩ (⟨définitions de colonnes⟩ [⟨contraintes de table⟩])`. Chaque définition de colonne comporte le nom de la colonne, son type et éventuellement une spécification de contraintes d'intégrité. Il est également possible de définir des contraintes d'intégrité sur la table. Les types de données supportés sont les chaînes de caractères de taille fixe, c'est-à-dire complétées à droite par des espaces (`CHAR`(*taille*)), les chaînes de taille variable (`VARCHAR`(*taille*)), les entiers sur 32 bits (`INTEGER`), les nombres en précision fixe (*nbChiffres* chiffres dont *nbDecimales* après la virgule) (`NUMERIC`(*nbChiffres*, *nbDecimales*)), les nombres en virgule flottante simple précision (`REAL`), les nombres en virgule flottante double précision (`DOUBLE PRECISION`) et les types date et/ou heure (`DATE`/`TIME`/`TIMESTAMP`).

Concernant les contraintes d'intégrité, le langage DDL permet de spécifier qu'un ensemble d'attributs ne contient pas de doublons (`UNIQUE`), qu'un ensemble d'attributs ne contient pas de valeur manquante (`NOT NULL`), qu'un ensemble d'attributs est clé primaire (`PRIMARY KEY`), qu'une séquence d'attributs est clé étrangère (`FOREIGN KEY`/`REFERENCES`) ou de contraindre le domaine d'un attribut (`CHECK`).

Par exemple, la requête DDL

```
CREATE TABLE equipes(
nom VARCHAR(30) UNIQUE NOT NULL,
nomCourt CHAR(5) PRIMARY KEY,
ville VARCHAR(50)
);
```

crée la table `equipes`. Les contraintes `UNIQUE` et `NOT NULL` sur nom précisent que cet attribut est une clé candidate. L'attribut `nomCourt` est la clé primaire, donc annoté par `PRIMARY KEY`. Il est donc `UNIQUE` et `NOT NULL`.

Ce deuxième exemple

```
CREATE TABLE joueurs(
equipe CHAR(5) REFERENCES Equipes(nomCourt),
no INTEGER,
nom VARCHAR(30),
prenom VARCHAR(30),
age INTEGER CHECK(age BETWEEN 18 AND 40),
PRIMARY KEY(equipe, no),
CONSTRAINT joueurs_uniq_nom_prenom UNIQUE(nom, prenom)
);
```

crée la table `joueurs`. La contrainte référentielle précise que l'attribut `equipe` est une clé étrangère. La contrainte de domaine sur `age` limite les valeurs entre 18 et 40. La contrainte de table `PRIMARY KEY(equipe, no)` définit la clé primaire. Enfin, la contrainte de table nommée `joueurs_uniq_nom_prenom` précise l'autre clé candidate.

Nous terminons avec la requête de création de la table `rencontres`.

```
CREATE TABLE rencontres(
date_r : DATE,
local : CHAR(5) REFERENCES equipes(nomCourt),
visiteur : CHAR(5) REFERENCES equipes(nomCourt),
butl : INTEGER,
butv : INTEGER,
PRIMARY KEY(date_r, local, visiteur)
);
```

Le langage de manipulation de données

Maintenant que le schéma de la base de données est défini grâce au langage DDL, il reste à ajouter, dans la base de données, les données proprement dites. C'est le rôle du langage DML.

Une insertion de n-uplets a la syntaxe suivante : `INSERT INTO ⟨table⟩ [(⟨liste d'attributs⟩)] VALUES (⟨liste de valeurs⟩) | ⟨requete⟩`. On précise la liste des attributs à renseigner, les autres valeurs sont alors fixées à la valeur par défaut de l'attribut ou à `NULL`. La clause `VALUES` ajoute un n-uplet à la fois, alors que l'utilisation d'une requête à la place de `VALUES` permet d'insérer plusieurs n-uplets en une seule opération.

La requête suivante permet l'ajout du n-uplet ⟨'Paris Saint-Germain', 'PSG', 'Paris'⟩ dans la relation `equipes` :

```
INSERT INTO equipes
VALUES('Paris Saint-Germain', 'PSG', 'Paris');
```

La deuxième requête insère un n-uplet en ne précisant que les attributs `nom` et `nomCourt`. Ce n-uplet n'aura pas de valeur pour l'attribut `ville` (`NULL`).

```
INSERT INTO equipes(nom, nomCourt)
VALUES('Olympique de Marseille', 'OM');
```

Enfin, une dernière requête insère dans la table `joueurs` des n-uplets extraits de la table `juniors`.

```
INSERT INTO joueurs
SELECT no, nom, prenom, age, 'OM'
FROM juniors
WHERE evaluation > 15;
```

L'instruction `UPDATE` modifie les valeurs des n-uplets d'une table : `UPDATE ⟨table⟩ SET ⟨liste d'affectations⟩ [WHERE ⟨condition⟩]`. L'utilisation d'une liste de valeurs permet de modifier la valeur de plusieurs attributs. La clause `WHERE` définit les n-uplets mis à jour.

La requête suivante augmente l'âge des joueurs de l'OM et les affecte au PSG.

```
UPDATE joueurs
SET age = age + 1, equipe = 'PSG'
```

```
WHERE equipe = 'OM';
```

L'instruction DELETE supprime des n-uplets d'une table : DELETE FROM ⟨table⟩ [WHERE ⟨condition⟩]. La clause WHERE précise les n-uplets à supprimer.

Par exemple, la requête suivante supprime toutes les équipes parisiennes.

```
DELETE FROM equipes
WHERE ville = 'Paris';
```

Si la clause WHERE est absente, tous les n-uplets de la table sont supprimés.

```
DELETE FROM joueurs;
```

L'interrogation des données

Le dernier aspect du langage SQL que nous étudierons est l'interrogation des données, grâce à l'instruction SELECT :

```
SELECT ⟨liste d'expressions⟩
FROM ⟨liste de tables⟩
WHERE ⟨conditions⟩
GROUP BY ⟨liste d'attributs⟩
HAVING ⟨conditions⟩
ORDER BY ⟨liste d'attributs⟩
```

La clause SELECT spécifie le schéma de sortie – projection –, la clause FROM précise les tables impliquées et leurs liens – produit cartésien et jointures –, la clause WHERE fixe les conditions que doivent remplir les n-uplets résultats – sélection –, GROUP BY indique les attributs de groupement – agrégation –, HAVING impose une condition sur les groupes, ORDER BY définit les critères de tris des résultats.

La clause SELECT est suivie d'une liste d'expressions. Chaque expression peut être un attribut, un littéral entre guillemets ou une expression calculée. Le caractère « * » est un joker signifiant « tous les attributs de la table ».

Il est à noter que la clause SELECT peut faire apparaître des doublons, c'est-à-dire plusieurs n-uplets identiques. Il est possible d'éliminer ces doublons, avec DISTINCT dans la clause SELECT, mais cette opération est coûteuse. De fait, contrairement aux tables du modèle et de l'algèbre relationnels, qui sont

des ensembles, les tables de SQL sont des multi-ensembles, et les requêtes *SELECT/FROM/WHERE* sont définies sur de tels multi-ensembles.

Les requêtes suivantes illustrent ces aspects.

– Nom et prénom des joueurs

SQL
```
SELECT nom, prenom
FROM joueurs;
```

Algèbre $\pi_{nom,prenom}(joueurs)$

– Tous les attributs des joueurs

SQL
```
SELECT *
FROM joueurs;
```

Algèbre $\pi_{equipe,no,nom,prenom,age}(joueurs)$

– Âge des joueurs (sans doublons)

SQL
```
SELECT DISTINCT age
FROM joueurs;
```

Algèbre $\pi_{age}(joueurs)$

La clause `FROM` accepte en paramètre une liste de tables. La requête porte alors sur le produit cartésien de ces tables. Pour effectuer une jointure entre les tables, il convient d'utiliser les opérateurs de jointure du langage. Le mot-clé `JOIN` placé entre deux tables indique qu'il faut en faire la jointure. La condition de jointure est précisée avec le mot-clé `ON`. Une jointure naturelle s'exprime avec le mot-clé `NATURAL JOIN`.

L'exemple suivant montre deux requêtes de jointure.

– Nom des joueurs et de leur équipe

SQL
```
SELECT j.nom AS nom, e.nom AS equipe
FROM joueurs j JOIN equipes e ON j.equipe =
e.nomCourt;
```

Algèbre $r_1 = \rho_{equipe \to club}(joueurs)$
$r_2 = \rho_{nom \to equipe}(equipes)$
$\pi_{nom,club}(r_1 \bowtie_{club=nomCourt} r_2)$

– Tous les résultats des rencontres

SQL
```
SELECT l.nom AS local,
       butl || '/' || butv AS resultat,
       v.nom AS visiteur?
FROM equipes l
JOIN rencontres r ON l.nomCourt = r.local
JOIN equipes v ON r.visiteur = v.nomCourt;
```

Algèbre $r_1 = \rho_{*\rightarrow r*}(equipes)$
$r_2 = r_1 \bowtie_{l.nomCourt=local} rencontres$
$r_3 = r_2 \bowtie_{visiteur=v.nomCourt} \rho_{*\rightarrow v.*}(equipes)$
$\pi_{l.nom,butl||'/'||butv,v.nom}(r_3)$

Dans cet exemple, des alias d'attributs et de table sont utilisés : la syntaxe `AS nom` dans la clause `SELECT` permet de renommer la colonne en sortie, la syntaxe `joueurs j` définit `j` comme alias de la table `joueurs`. Enfin, la seconde requête fait usage de l'opérateur de concaténation « || » pour mettre en forme les résultats.

La clause `WHERE` permet d'exprimer la sélection. SQL supporte des opérateurs de comparaison, `<`, `<=`, `=`, `>=`, `>`, `<>`, logiques, `AND`, `OR`, de test de valeur manquante, `IS NULL`, `IS NOT NULL`, de recherche textuelle, `LIKE`, de sélection d'intervalle, `BETWEEN`, de liste, `IN`, et d'imbrication de blocs `IN`, `EXIST`, `NOT EXIST`, `ALL`, `SOME`, `ANY`.

Les requêtes suivantes illustrent ces notions.

– Nom et prénom des joueurs de plus de 30 ans

SQL
```
SELECT nom, prenom FROM joueurs
WHERE age > 30;
```
Algèbre $\pi_{nom,prenom}(\sigma_{age>30}(joueurs))$

– Nom et prénom des joueurs qui ont entre 20 et 30 ans et dont le nom commence par 'L'

SQL
```
SELECT nom, prenom FROM joueurs
WHERE age BETWEEN 20 AND 30
AND nom LIKE 'L%';
```
Algèbre $r_1 = \sigma_{age>20 \wedge age<30 \wedge nom\ LIKE\ 'L\%'}(joueurs)$
$\pi_{nom,prenom}(r_1)$

– Nom des joueurs dont l'âge est inconnu

SQL
```
SELECT nom FROM joueurs
WHERE age IS NULL;
```
Algèbre $\pi_{nom}(\sigma_{age\ IS\ NULL}(joueurs))$

Les fonctions d'agrégation, les clauses `GROUP BY` et `HAVING` permettent de « résumer » un ensemble de n-uplets. La clause `GROUP BY` précise les attributs de groupement. Les fonctions d'agrégation sont ensuite appliquées sur chaque groupe. Enfin, la clause `HAVING` permet de spécifier des conditions sur les groupes. Il s'agit donc ici d'une condition *globale* à un groupe.

– Nombre moyen de buts marqués à domicile

SQL
```
SELECT AVG(butl) FROM rencontres;
```
Algèbre $\gamma_{AVG(butl)}(rencontres)$

– Nombre total de buts marqués à domicile par équipe

SQL
```
SELECT nom AS Equipe, SUM(butl) AS Buts
FROM equipes JOIN rencontres ON local = nomCourt
GROUP BY nom
```
Algèbre $r_1 = equipes \bowtie_{local=nomCourt} rencontres$
$_{nom}\gamma_{SUM(butl)}(r_1)$

– Nombre moyen de buts marqués à domicile par équipe ayant disputé plus de 5 rencontres

SQL
```
SELECT nom AS Equipe, AVG(butl) AS Moyenne
FROM equipes JOIN rencontres ON local = nom
      Court
GROUP BY nom
HAVING COUNT(*) > 5;
```
Algèbre $r_1 = equipes \bowtie_{local=nomCourt} rencontres$
$r_2 = {}_{nom}\gamma_{AVG(butl),COUNT(*)}(r_1)$
$\sigma_{COUNT(*)>5}(r_2)$

SQL permet également de regrouper les résultats de plusieurs requêtes à l'aide d'opérations ensemblistes. La requête ⟨*requete*1⟩ `UNION` ⟨*requete*2⟩ retourne l'union des résultats des deux requêtes. La requête ⟨*requete*1⟩ `INTERSECT` ⟨*requete*2⟩ retourne l'intersection des résultats des deux requêtes. La requête ⟨*requete*1⟩ `EXCEPT` ⟨*requete*2⟩ retourne la différence des résultats des deux requêtes. Pour ces trois opérations, les schémas des requêtes doivent être compatibles, c'est-à-dire avoir le même nombre et le même type d'attributs. Ces opérations s'effectuent sur des ensembles, c'est-à-dire que les doublons sont éliminés. Le mot-clé `ALL` ajouté après un opérateur ensembliste évite l'élimination des doublons et permet donc de manipuler des multi-ensembles.

La clause `ORDER BY` spécifie l'ordre de tri des résultats. Elle est suivie d'une liste d'attributs pour lesquels on précise l'ordre de tri – `ASC` pour croissant, `DESC` pour décroissant. Cette clause, qui n'a pas d'équivalent en algèbre relationnelle, ne doit apparaître qu'une seule fois dans une requête et obligatoirement à la fin. En effet, elle transforme un ensemble de n-uplets en liste de n-uplets pour représenter l'ordre. Il n'est donc pas possible d'appliquer un autre opérateur sur le résultat de cette clause.

– Nom et prénom des joueurs triés en ordre croissant selon le nom puis le prénom

```
SELECT nom, prenom FROM joueurs
ORDER BY nom ASC, prenom ASC;
```

Arrivés à ce point, nous avons étudié le modèle de données qui constitue le fondement des bases de données relationnelles, ainsi que deux langages d'interrogation de ces données : l'algèbre relationnelle et SQL. Il reste une question à laquelle nous n'avons pas répondu : comment choisit-on le schéma de base de données à créer ? L'étude de la conception de base de données va nous permettre de répondre.

Conception de bases de données relationnelles

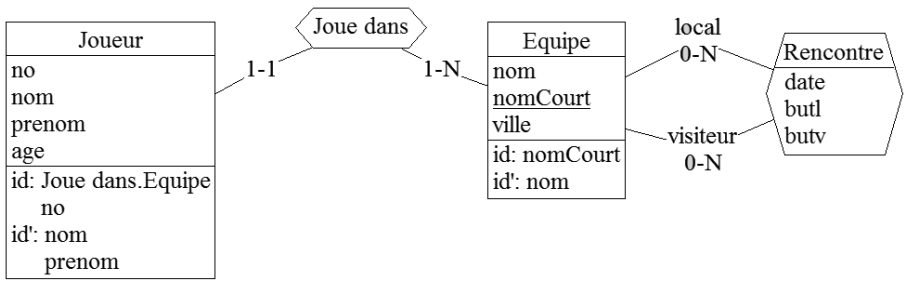
La *conception de bases de données relationnelles* a pour objectif la création d'un modèle relationnel pour une application, à partir de l'*analyse des besoins*. C'est une problématique très importante, car on constate que tous les modèles, pour une même application, ne se valent pas : un schéma de base de données mal conçu peut engendrer des anomalies lors des mises à jour de la base de données, par exemple des incohérences, des problèmes de performance ou un espace occupé trop important. Deux grandes approches sont possibles : la *normalisation* et la *modélisation conceptuelle*.

La *normalisation* consiste à représenter les besoins sous la forme de contraintes d'intégrité. Ces contraintes sont ensuite utilisées comme entrées dans un *algorithme de normalisation* qui génère un schéma de base de données satisfaisant certaines propriétés. On dit que le schéma résultant est en *forme normale*. La difficulté de la normalisation réside dans la nécessité de disposer de l'ensemble des contraintes d'intégrité pour démarrer le processus. L'énumération de ces contraintes étant une tâche ardue, la normalisation s'utilise en pratique rarement de façon indépendante.

La *modélisation conceptuelle* consiste à représenter les besoins sous la forme d'un *modèle conceptuel*. Ce dernier, plus expressif que le modèle relationnel, est ensuite traduit en modèle relationnel, en respectant des *règles de transformation de modèles*. La modélisation conceptuelle a pour objectifs de permettre une meilleure compréhension du problème – *abstraction* – et de permettre une conception progressive – *raffinement*. La démarche consiste à identifier les concepts fondamentaux, à les représenter dans le formalisme choisi, en général un formalisme visuel, puis à en dériver le schéma de la base de données. La difficulté de cette approche est la création du modèle conceptuel. En effet, même si c'est un modèle plus expressif, il demeure difficile à réaliser pour une application de grande taille. Il n'existe pas de recette pour obtenir à coup sûr un bon modèle conceptuel. La réussite d'une telle approche repose sur l'utilisation de bonnes pratiques de modélisation ainsi que sur l'expérience de l'équipe en

charge de cette tâche. Dans la suite de ce paragraphe, nous détaillons cette approche en nous appuyant sur le modèle entité-association.

Le modèle *entité-association* est un ensemble de concepts pour la modélisation des données d'une application. Chaque concept peut être représenté par un symbole graphique associé. Une *entité* est un objet du monde réel qui peut être identifié et que l'on souhaite représenter. Le *type d'entité* correspond à l'ensemble des objets ayant des caractéristiques communes. Une entité possède un ensemble de propriétés nommées *attributs*. Dans le modèle de base, un attribut est *simple*, ou *atomique*, c'est-à-dire qu'il ne possède qu'une seule valeur pour un n-uplet donné. Dans les modèles étendus, un attribut peut être *composé* ou *multivalué*, ce qui permet, par exemple, de gérer en un seul n-uplet les prénoms multiples d'une personne. Chaque entité possède un *identifiant* ou *clé*, c'est-à-dire un ensemble d'attributs qui différencie une instance d'entité parmi les instances de son type d'entité. Sur la figure ci-après, un type d'entité est représenté par un rectangle comportant trois cartouches. Le premier contient le nom du type d'entité, le second les attributs et le troisième les identifiants. L'identifiant primaire est noté `id`, les secondaires `id'`. On remarque que l'identifiant primaire du joueur est un *identifiant hybride* formé du type d'entité voisin et d'un attribut local. En effet, le numéro d'un joueur ne suffit pas pour l'identifier : il peut par exemple exister un numéro 10 – le meneur de jeu – par équipe. L'identification d'un joueur se fait donc grâce à son numéro, mais aussi avec son équipe.



Un diagramme entité-association.

Les entités sont reliées par des *associations*. Le *type d'association A* est un sous-ensemble du produit cartésien des types d'entités E_i participants : $A \subseteq E_1 \times E_2 \times \ldots \times E_n$. Une *instance d'association* a_i est un n-uplet $(e_1, \ldots, e_n)$ où e_i est de type E_i. Une association peut avoir des attributs. Le *degré* d'une association est le nombre d'entités participantes. Les associations les plus courantes sont *binaires*, c'est-à-dire de degré 2. Une entité participant à une association

peut avoir un *rôle* : le rôle qu'elle joue dans l'association. La *cardinalité* d'une association précise le nombre de fois où une instance d'entité peut participer à l'association. On rencontre généralement les cardinalités *un à un* (1 : 1), *un à plusieurs* (1 : *N*) ou *plusieurs à plusieurs* (*N* : *N*). La *cardinalité maximum* précise le nombre maximum de fois où une instance d'entité peut participer à l'association (1 ou *N*). La *cardinalité minimum* précise le nombre minimum de fois où une instance d'entité peut participer à l'association (0 ou 1). Sur la figure ci-avant, les types d'associations sont représentés par des hexagones contenant le nom de l'association. Éventuellement, l'hexagone peut contenir des attributs comme dans le cas de `Rencontre`. Le type d'association est relié aux types d'entité participants. Sur un lien entre un type d'entité *E* et un type d'association *A* apparaissent la cardinalité minimum et maximum. Ces dernières définissent le nombre d'instances de *E* qui peuvent participer à *A*. Sur le schéma, en considérant uniquement les cardinalités maximums, on constate que `Joue dans` est une association 1 : *N* et `Rencontre` une *N* : *N*. Il est également possible de préciser le rôle d'un participant sur le lien : dans le type d'association `Rencontre`, le type d'entité `Equipe` joue le rôle du `local` et du `visiteur`.

Une fois le modèle conceptuel construit, il reste à le transformer dans le modèle logique choisi. La *transformation de modèle* est le processus permettant de traduire un modèle M_1 en un modèle M_2. C'est une technique utilisée à plusieurs niveaux pour la conception de base de données : lors du passage du modèle conceptuel vers le modèle logique ou du modèle logique vers le modèle physique.

La traduction d'un modèle entité-association en un modèle relationnel se fait en respectant quelques règles simples. Chaque type d'entité *E* est transformé en un schéma de relation *R* dont les attributs sont les attributs de *E*. La clé primaire de *R* est choisie parmi les identifiants de *E*. Dans le cas général, un type d'association *A* entre les entités $E_1, \ldots, E_n$ est transformé en une relation *R* dont les attributs sont les identifiants des E_i et les éventuels attributs de *A*. La clé primaire de *R* est l'ensemble des identifiants des E_i participants. Dans le cas d'une association binaire 1 : 1 ou 1 : *N*, la clé de l'entité du côté *N* peut être ajoutée au schéma de relation représentant l'entité du côté 1. Dans le cas d'une association binaire 1 : 1, les deux entités participantes peuvent éventuellement être fusionnées. Ce dernier choix n'est pas forcément souhaitable si les entités participantes ont d'autres associations ou si les deux cardinalités minimales sont différentes de un.

La transformation du diagramme entité-association de notre exemple donne le schéma relationnel suivant. Le type d'entité `Joueur` devient le schéma de relation `joueurs` avec les attributs `no`, `nom`, `prenom` et `age`. Le type

d'entité `Equipe` devient le schéma de relation `equipes` avec les attributs `nom`, `nomCourt` et `ville`. Le type d'association 1 : *N* `Joue dans` est implémenté comme une clé étrangère dans `Joueur`. Le type d'association *N* : *N* `Rencontre` devient le schéma de relation rencontres qui possède comme attribut les identifiants des entités participantes et les attributs `date`, `butl` et `butv`.

Programmation et bases de données

Tout programme manipulant des données, il est assez naturel de se poser la question de l'accès à une base de données depuis un programme. Il existe pour cela différentes approches qui s'appuient sur des technologies variées.

La première possibilité consiste à utiliser un langage de programmation directement intégré au système de gestion de bases de données. On parle alors de *procédures stockées*. Par exemple, le standard SQL/PSM (*SQL/Persistent Stored Modules*) est une extension de la norme SQL. Il ajoute des possibilités de programmation dans un langage impératif, comme celui présenté au deuxième chapitre, aux capacités d'interrogation de SQL. La plupart des systèmes de gestion de bases de données proposent un ou plusieurs langages pour le développement de procédures stockées : PL/SQL pour Oracle, PL/pgSQL pour PostgreSQL, etc.

La deuxième possibilité, nommée *SQL embarqué,* intègre des requêtes SQL à un langage de programmation. Ce principe nécessite alors d'avoir recours à un *préprocesseur*. En effet, le programme incluant les ordres SQL n'est plus un programme valide pour le langage de programmation choisi. Avant la compilation du programme, le préprocesseur transformera les portions de code SQL pour que le programme puisse être compilé. Par exemple, SQLJ est une extension du langage Java qui permet d'utiliser du code SQL dans un programme. Certains systèmes de gestion de bases de données proposent ce type de préprocesseur. Le langage Pro*C/C++ est une extension des langages C/C++ permettant l'utilisation de SQL pour l'accès au système de gestion de bases de données Oracle.

La troisième approche fait appel à des *interfaces de programmation* (*API*) pour permettre l'accès aux systèmes de gestion de bases de données. Le principe est de fournir une bibliothèque de fonctions d'accès à un système de gestion de bases de données, pouvant être appelées à partir d'un langage donné. La norme SQL/CLI (*SQL/Call-Level Interface*) définit une interface standard pour cela. On peut également citer la bibliothèque *ODBC* (*Open DataBase Connectivity*), très communément utilisée, ou *JDBC* (*Java DataBase Connectivity*) qui est le standard de fait pour le langage Java.

Une dernière approche, qui fait appel à des bibliothèques de haut niveau, est rendue nécessaire du fait que la plupart des programmes sont écrits avec un langage objet et que la plupart des données sont conservées dans des bases de données relationnelles, mais ce problème dépasse le cadre de ce chapitre.

Quelle que soit l'approche choisie, l'accès à un système de gestion de bases de données à partir d'un langage de programmation suit un processus que nous présentons maintenant.

Principe général

Une *session* d'interaction avec un système de gestion de bases de données nécessite quatre grandes étapes : la connexion au système de gestion de bases de données et le choix de la base de données, l'exécution d'une requête, la récupération des résultats et la fermeture de la connexion. Les étapes d'exécution de la requête et de récupération des résultats peuvent évidemment être répétées autant que nécessaire lors d'une session.

L'ouverture d'une session avec un système de gestion de bases de données nécessite une connexion préalable. Le rôle de cette étape est d'authentifier la personne qui demande la connexion et de s'assurer que cette dernière possède bien les droits nécessaires pour cela. D'autre part, la plupart des systèmes de gestion de bases de données fonctionnent selon une architecture client/serveur. Lors de la connexion, il faut donc également préciser avec quel serveur on veut interagir. Toutes les informations utilisées pour l'ouverture de session sont en général regroupées au sein d'une *chaîne de connexion*. Cette dernière décrit en particulier le serveur à contacter – une adresse IP et un numéro de port —, le nom de la base de données, le nom de l'utilisateur, son mot de passe et diverses options spécifiques au type de connexion utilisée.

L'exécution d'une requête passe par l'appel d'une fonction spécifique de l'API. Cette dernière exécute donc des requêtes SQL sur le système de gestion de bases de données, à partir du langage de programmation. Cette fonction retourne un *ensemble de résultats*, aussi appelé *curseur*.

La récupération des résultats se fait grâce à ce dernier. Un curseur est une structure de données qui permet d'itérer sur les résultats de la requête.

Ce principe général d'accès à une base de données à partir d'un langage de programmation est naturellement implanté de façon différente en fonction du langage, du système de gestion de bases de données et de l'approche choisie. Le paragraphe suivant, « Publication de données sur le Web » offre un exemple concret de l'accès au système de gestion de bases de données MySQL à partir du langage PHP.

Publication de données sur le Web

Depuis les années 1990, l'information s'est installée sur le Web. Les premiers sites étaient composés de pages HTML (*HyperText Markup Language*) statiques, c'est-à-dire que tout leur contenu était figé. Pour modifier le contenu d'une telle page, il fallait directement éditer le code source de la page, et y apporter les modifications nécessaires. On retrouvera sur `http://www.w3.org/History/19921103-hypertext/hypertext/WWW/TheProject.html` l'aspect et le contenu du premier site web du World Wide Web Consortium (W3C), publié en 1992.

Désormais, la plupart des données des sites web sont stockées dans des bases de données et les sites sont devenus de véritables applications manipulant données et présentation. Dans ce paragraphe, nous partons de la base de données sur les matchs de football, que nous avons construite, pour réaliser un site web consacré au football.

Qu'est-ce qu'un site web ?

Le terme *site web* est passé dans le langage courant. Et si l'on comprend qu'un site web est un ensemble de pages web affichant des informations ou permettant des interactions, on ignore souvent tout de l'architecture interne d'un tel site : comment il a été réalisé et comment il fonctionne. Plus précisément, il faut distinguer, d'une part, les infrastructures et matériels permettant l'hébergement et la diffusion du site et, d'autre part, le code qui correspond au « programme » du site web, exécuté sur cette infrastructure.

Nous présentons rapidement l'infrastructure matérielle nécessaire pour réaliser un site web, et renvoyons le lecteur au cinquième chapitre, pour de plus amples détails, puis nous présenterons les langages permettant l'écriture de pages web statiques, à savoir HTML (*HyperText Markup Language*) et CSS (*Cascading Style Sheets*), ainsi qu'un langage très utilisé, PHP (*Personal Home Page*), permettant l'écriture de pages web dynamiques, c'est-à-dire qui interagissent avec une base de données.

Internet On confond parfois le Web et Internet. Internet, développé depuis la fin des années 1960 aux États-Unis et utilisé commercialement à partir des années 1990, est un réseau informatique planétaire, composé physiquement de câbles de cuivre ou de fibres optiques, qui transmettent une information numérique d'un serveur à un autre. Ces données numériques sont interprétées grâce à des protocoles de communication à plusieurs niveaux : la couche physique, par exemple ADSL ou Wifi, la couche liaison, par exemple Ethernet ou PPPoE, la couche réseau, par exemple IPv4, la couche transport, par exemple TCP et, tout en haut, la couche application avec le standard HTTP

(*HyperText Transfer Protocol*) utilisé par les serveurs web et les navigateurs web comme Firefox, Explorer, Safari, etc. pour communiquer. Le Web est donc *une* application utilisant Internet, mais la plus largement utilisée, d'où la confusion.

Un serveur web ou HTTP Le standard HTTP a été inventé par sir Tim Berners-Lee, qui par là même inventait le Web tel qu'on le connaît. Il est le fondateur et directeur du World Wide Web Consortium (W3C). Le W3C est une organisation de standardisation des technologies du Web qui a en particulier normalisé le langage HTML. Il définit le format des messages qui sont produits par un serveur à destination d'un navigateur. Le navigateur est une application qui reçoit des informations de la part du serveur web, puis interprète leur contenu pour afficher la page web demandée. Il faut donc distinguer HTTP qui est en quelque sorte l'enveloppe contenant, entre autres choses, le code de la page, et le contenu du message, en l'occurrence du HTML. Voici un exemple de communication en HTTP entre un navigateur et un serveur. Le client effectue une demande de transmission de page au serveur `www.exemple-foot.com`, par le biais de la commande GET suivie du nom de la page: `/index.html`, c'est-à-dire le fichier intitulé index.html se trouvant dans le répertoire racine du serveur et du format de transfert – ici HTTP1.1. Notons que le nom du serveur `www.exemple-foot.com` est transformé en adresse IP par un serveur de nom de domaines (DNS) qui connaît la correspondance entre le nom du serveur et son adresse.

```
GET /index.html HTTP/1.1
Host: www.exemple-foot.com
User-Agent: Mozilla 5.0
```

Le serveur répond quelque chose du genre:

```
HTTP/1.1 200 OK
Date: Mon, 1 Jan 2011 14:36:01 GMT
Server: Apache/2.2.9 (Debian)
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitio-
nal//EN">
<html> <head><title>Football</title></head>
```

```
<body> Texte de présentation du site de foot…</body>
</html>
```

Sans entrer dans les détails – qui sont disponibles à `http://www.w3.org/Protocols/rfc1945/rfc1945.txt` pour la version 1.0 et `http://www.ietf.org/rfc/rfc2616.txt` pour la version 1.1 –, nous remarquons que la première ligne indique un code 200 précisant que tout s'est bien passé – une page non trouvée renvoyant le code bien connu *Erreur 404*. Suit un ensemble d'informations décrivant le serveur et la connexion. À la suite de ces informations, il y a un saut de ligne, puis du texte qui correspond à une page web au format HTML, qui commence par `< !DOCTYPE html…` qui définit la structure de la page – ici du HTML 4.0 – puis la page à proprement parler, qui commence avec la balise `<html>`, comme nous l'avons vu au deuxième chapitre.

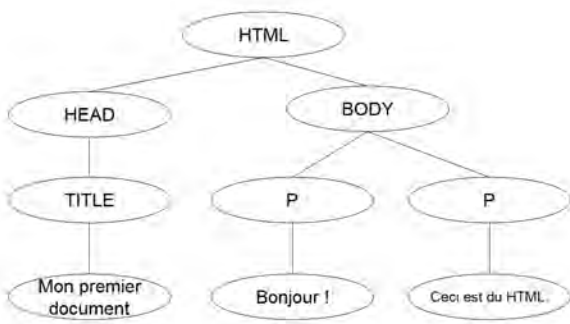
Les standards du Web

Il y a de nombreux standards définissant la manière de représenter les données sur le Web, ainsi que la manière de les présenter. Le standard de base de présentation de données est le format HTML – ou XHTML, qui est moins utilisé, mais compatible. Le format HTML4 est essentiellement un format de présentation, c'est-à-dire qu'il ne décrit pas le contenu sémantique du document, comme on le ferait avec une table d'une base de données sur des joueurs de football, où il y aurait un nom, un prénom, une date de naissance, etc., mais il indique plutôt que l'on veut mettre telle valeur en gras, telle valeur dans un tableau, etc. Voici un petit exemple de page HTML que nous allons commenter. Pour visualiser le rendu de cette page, il suffit de taper ce texte dans un éditeur de texte quelconque, puis, après l'avoir sauvegardé, de l'ouvrir à l'aide d'un navigateur. Dans la plupart des navigateurs, dans le menu *fichier*, une fonction permet d'ouvrir un fichier présent sur le disque de l'ordinateur utilisé.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
      "http://www.w3.org/TR/html4/strict.dtd">
<HTML>
      <HEAD>
      <TITLE>Mon premier document</TITLE>
      </HEAD>
      <BODY>
      <P>Bonjour !
      <P>Ceci est du HTML.
```

```
    </BODY>
</HTML>
```

La première ligne `DOCTYPE` définit la version de HTML utilisée. Une version HTML correspond à une grammaire particulière, composée de mots précis : le vocabulaire HTML. En principe, si la page ne comporte pas d'erreurs de syntaxe, son rendu sera globalement le même quel que soit le navigateur utilisé. Parfois, même si certaines erreurs de syntaxe sont commises, le navigateur essaye de les corriger, mais dans ce cas le rendu varie souvent d'un navigateur à l'autre. Puisque l'objectif final d'une page web est d'être lue par un humain, de petites variations dans la présentation sont, la plupart du temps, sans conséquences. Viennent ensuite plusieurs *balises* – les majuscules étaient préférées dans la version 4.0, depuis les lettres minuscules ont pris le dessus, mais les deux casses sont autorisées – qui peuvent être soit ouvrantes, comme `<HTML>`, soit fermantes, comme `</HTML>`. Le plus simple est de se représenter un document HTML comme un *arbre*, illustré par la figure ci-après, dont la racine est le nœud `<HTML>`. XHTML est justement écrit dans le langage XML (*eXtensible Markup Language*) qui permet de décrire des arbres par l'utilisation de balises : une balise ouvrante indique le début d'un nœud, et toutes les balises contenues entre la balise ouvrante et la balise fermante composent le sous-arbre enraciné sous ce nœud.



Représentation arborescente d'un document HTML.

La liste de toutes les balises possibles est définie dans la spécification HTML écrite par le W3C, et disponible à l'adresse `http://www.w3.org/TR/html401/`. Au lieu de décrire ici toutes les balises une par une, nous allons nous contenter de montrer comment dessiner un tableau, ce qui est assez utile

quand on gère des données relationnelles. Voici un exemple de fichier HTML représentant un tableau de joueurs de football

```
<html>
<body>
<table border=1>
<tr><th>Nom</th><th>Prenom</th><th>Equipe</th></tr>
<tr><td>Benzema</td><td>Karim</td><td>Real  Madrid</td></
tr>
<tr><td>Malouda</td><td>Florent</td><td>Chealsea</td></
tr>
<tr><td>Sagna</td><td>Bacary</td><td>Arsenal</td></tr>
<tr><td>Lloris</td><td>Hugo</td><td>Olympique  Lyonnais</
td></tr>
</table>
</body>
</html>
```

dont le rendu par un navigateur est le suivant

Nom	Prenom	Equipe
Benzema	Karim	Real Madrid
Malouda	Florent	Chealsea
Sagna	Bacary	Arsenal
Lloris	Hugo	Olympique Lyonnais

Rendu d'un tableau en HTML.

Certaines balises ont des attributs, par exemple la balise `<table>` a un attribut `border` qui demande de dessiner un contour aux cellules. Chaque ligne de la table est définie par une balise `<tr>` (*table row*). À l'intérieur d'une ligne, chaque cellule est définie soit par une balise `<td>` (*table data*) soit par une balise `<th>` (*table header*).

Le langage HTML a été étendu en le langage XML (*eXtensible Markup Language*) – voir `http://www.w3.org/XML/` – qui est un format semi-structuré de données où l'utilisateur peut définir lui-même ses propres balises,

et ainsi structurer les données selon son souhait, comme dans une base de données.

La représentation sérialisée d'un document XML ressemble aux exemples que nous avons déjà donnés en HTML, à savoir des balises ouvrantes et fermantes. XML est très utilisé comme format d'échange de données sur le Web, comme format de stockage de fichiers de configuration et comme format d'intégration de données hétérogènes. Il existe même des systèmes de gestion de bases de données permettant de stocker des données XML et de les interroger via un langage de requête – il en existe essentiellement deux : XSLT et XQuery, tous deux développés par le W3C. Les systèmes de gestion de bases de données relationnelles commerciaux, comme Oracle, DB2, SQLServer, etc., supportent tous XQuery et le format XML. Des systèmes de gestion de bases de données purs XML gratuits existent également : eXistDB, Qizx, Zorba, etc. Par exemple, dans le cas précédent, on pourrait définir son propre fichier XML pour stocker des joueurs de football, que l'on représenterait comme suit :

```
<joueurs>
<joueur>
<nom>Benzema</nom><prenom>Karim</prenom><equipe>Real  Ma-
drid</equipe>
</joueur>
 <joueur>
<nom>Malouda</nom><prenom>Florent</prenom><equipe>
Chelsea</equipe> </joueur>
<joueur>
<nom>Sagna</nom><prenom>Bacary</prenom><equipe>Arsenal</
equipe>
</joueur>
<joueur>
<nom>Lloris</nom><prenom>Hugo</prenom><equipe>Olympique
Lyonnais</equipe>
</joueur>
</joueurs>
```

Il faut ensuite définir une feuille de style – par exemple en XSLT – qui est un programme indiquant comment un navigateur doit afficher chaque élément.

Un exemple de programmation web : l'accès à MySQL depuis PHP

Nous donnons maintenant un exemple de programme, écrit dans le langage PHP, qui interagit avec une base de données et produit comme résultat le code source d'une page web, c'est-à-dire un texte représentant un document HTML. Le langage PHP ouvre l'accès à de nombreux systèmes de gestion de bases de données : MySQL, PostgreSQL, Oracle, etc. Nous utilisons, dans cet exemple, le système MySQL qui est l'un des systèmes de gestion de bases de données libres les plus utilisés. Le paragraphe suivant, « Exercices corrigés et commentés », fournit des explications plus approfondies sur l'installation d'un serveur web permettant d'exécuter ces programmes.

Des quatre approches présentées plus haut, au paragraphe « Programmation et bases de données », trois sont envisageables pour l'accès à MySQL depuis PHP. MySQL supporte les procédures stockées et il est possible de les invoquer depuis un programme PHP. En revanche, il n'existe pas d'approche de type SQL embarqué pour PHP. Différentes API donnent accès à un système de gestion de bases de données depuis PHP : certaines sont spécifiques à un produit, d'autres sont génériques et permettent de s'abstraire du type de système de gestion de bases de données utilisé. Enfin, la bibliothèque *Doctrine* interface des objets PHP avec un système de gestion de bases de données relationnel. Parmi ces possibilités, nous avons fait le choix de présenter l'API spécifique de MySQL, solution simple et répandue.

La fonction `mysql_connect` connecte au système de gestion de bases de données MySQL. Elle retourne une *ressource de connexion* – ou `FALSE` en cas d'échec – qui sera passée en paramètre de toutes les fonctions utilisant cette connexion. Elle prend en arguments les paramètres de connexion : `host, port, user, password` – dans cet exemple nous utilisons les paramètres par défaut.

Nous choisissons ensuite la base de données, qui doit avoir été créée préalablement, avec la fonction `mysql_select_db`. La notion de chaîne de connexion évoquée ci-avant est ici décomposée en plusieurs paramètres. La connexion est fermée en fin de session avec la fonction `mysql_close`. Cette dernière prend en paramètre la ressource de connexion à fermer. Les fonctions `mysql_error` et `mysql_errno` permettent de récupérer des informations dans le cas d'une erreur. Ce fichier est à sauvegarder avec l'extension `.php`, à placer dans le répertoire web – souvent nommé `www` – du serveur, et à visualiser dans un navigateur en se connectant sur ce serveur. Attention, ouvrir en double cliquant sur le fichier ne fera rien, si ce n'est ouvrir un éditeur de texte. Il faut lancer le navigateur, puis trouver l'URL, commençant par `http://localhost/…`, pour visualiser le fichier.

```
<?php
// Connexion, sélection de la base de données
$link = mysql_connect('localhost:3306', 'root', '')
or die('Could not connect: ' . mysql_error());
echo 'Connected successfully';
mysql_select_db('foot') or die('Could not select database');

// connexion à une base de données nommée "foot"
// sur l'hôte "localhost" avec un nom d'utilisateur root
// sans mot de passe

// Utilisation de la connexion…
mysql_close($link); ?>
```

La fonction `resource my_query (string $query, resource $connection)` exécute la requête `query` en utilisant la connexion `connection`. Si tout se passe bien, la fonction retourne un curseur nommé *ressource de résultats*. En cas d'erreur, `FALSE` est retourné. On utilise alors les fonctions `mysql_error` et `mysql_errno` pour obtenir des détails sur l'erreur.

```
<?php
// Exécution de la requête SQL
$query = 'SELECT nom, prenom FROM joueurs';
$result = mysql_query($query, $link) or die('Query failed:
' . mysql_error());
?>
```

La fonction `array mysql_fetch_row(resource $result)` permet de récupérer un n-uplet du résultat. Lors de chaque appel, le curseur mémorise le n-uplet à renvoyer lors de l'appel suivant. La fonction retourne un tableau où chaque valeur d'attribut du n-uplet occupe une cellule. Une valeur manquante est fixée à la valeur `PHP NULL`. La fonction retourne `FALSE` quand il n'y a plus de lignes. D'autres fonctions plus évoluées fonctionnent selon le même principe. Par exemple, la fonction `array mysql_fetch_array(resource $result)` renvoie un tableau associatif contenant les noms des attributs et non pas des indices. Enfin, certaines fonctions permettent d'obtenir des informations sur les résultats, appelées *métadonnées*

Fonction	Description
`mysql_num_fields`	Nombre d'attributs
`mysql_num_rows`	Nombre de n-uplets
`mysql_affected_rows`	Nombre de n-uplets affectés par l'ordre DML
`mysql_field_flags`	Détails d'un champ
`mysql_field_name`	Nom du champ
`mysql_field_len`	Taille du champ
`mysql_field_table`	Nom de la table
`mysql_field_type`	Type du champ

Fonctions d'interrogation des métadonnées.

Quand le curseur n'est plus utilisé, on libère la ressource avec `mysql_free_result`.

L'instruction `echo` permet de demander une sortie textuelle. Dans le cas de programmes PHP produits par un serveur web, il faut que le résultat, donc la sortie textuelle du programme, soit un fichier HTML, ce qui lui permettra d'être affiché par le navigateur. L'exemple de code suivant affiche le nom des joueurs de foot.

```
<?php
while ($row = mysql_fetch_row($result)) {
 echo "Joueur : $row[0] $row[1]";
 echo "<br />\n";
}
?>
```

En conclusion, cet exemple complet reprend les fragments du code présenté dans ce chapitre.

```
<?php
// Connexion, sélection de la base de données
$link = mysql_connect('localhost', 'user', 'password')
  or die('Could not connect: ' . mysql_error());
echo 'Connected successfully';
mysql_select_db('foot') or die('Could not select database');

// Exécution de la requête SQL
$query = 'SELECT nom, prenom FROM joueurs';
```

```
$result = mysql_query($query) or die('Query failed: ' .
mysql_error());

// Affichage des résultats en HTML
echo "<table>\n";
while ($line = mysql_fetch_array($result, MYSQL_ASSOC)) {
  echo "\t<tr>\n";
  foreach ($line as $col_value) {
echo "\t\t<td>$col_value</td>\n";
  }
  echo "\t</tr>\n";
}
echo "</table>\n";

// Libère le resultset
mysql_free_result($result);

// Ferme la connexion
mysql_close($link);
?>
```

Exercices corrigés et commentés

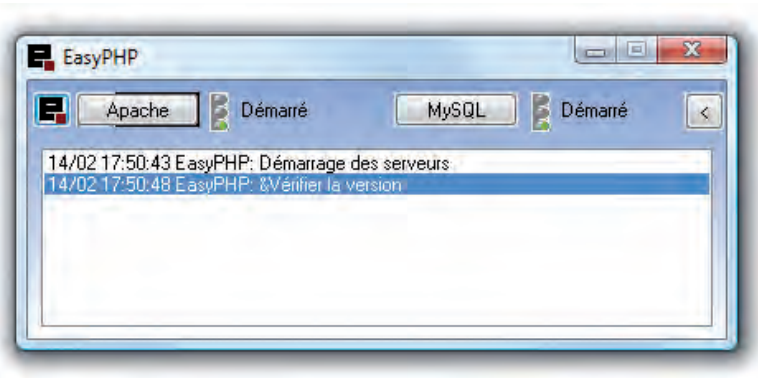
Installation du logiciel easyPHP

Nous présentons dans ce paragraphe l'installation d'un serveur web, d'un système de gestion de bases de données ainsi que sa configuration. Notre choix de logiciels s'est porté sur la distribution *easyPHP* qui peut être téléchargée gratuitement sur le site `http://www.easyphp.org/`, et nous détaillons son lancement sous Windows ; le lecteur transposera en fonction du système d'exploitation qu'il utilise. Nous lui laissons aussi le soin d'installer correctement le logiciel qui se compose, d'une part, du serveur web nommé Apache avec l'interpréteur PHP et, d'autre part, du système de gestion de bases de données MySQL.

Le répertoire d'installation sera nommé dans la suite MYSQLHOMEDIR, nous y mettrons les fichiers `.html` et `.php`. En lançant easyPHP, nous voyons apparaître dans la barre des programmes – en bas à droite – une lettre « e » qui correspond à l'icône représentant le programme

Icône EasyPHP.

En double cliquant sur l'icône, nous faisons apparaître la fenêtre d'état du serveur, symbolisée par deux feux de signalisation. Le feu de gauche correspond au serveur web Apache et celui de droite au système de gestion de bases de données MySQL. En principe, les deux feux doivent être au vert.

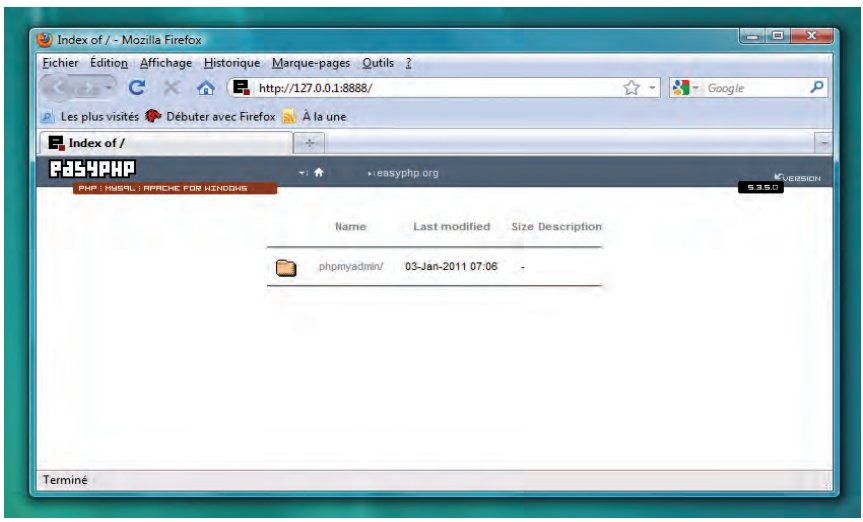


Fenêtre d'état du serveur EasyPHP.

Si ce n'est pas le cas, le problème vient la plupart du temps du fait que easyPHP n'a pas été lancé en mode administrateur. Au lieu de double cliquer sur l'icône pour lancer le programme, il faut alors faire un clic droit et choisir l'option *Exécuter en tant qu'administrateur*. Quand le serveur et le système de gestion de bases de données fonctionnent, nous visualisons le contenu du répertoire des fichiers qui peuvent être affichés par le serveur, en se rendant dans le répertoire www qui se trouve dans le répertoire d'installation de easyPHP. En principe, ce répertoire est vide. Nous y copions le répertoire *phpmyadmin* qui se trouve dans le répertoire d'installation de easyPHP. En faisant un clic droit et en choisissant l'option *web local*, un navigateur, par exemple Internet Explorer, Firefox, Chrome, etc., sera automatiquement lancé.

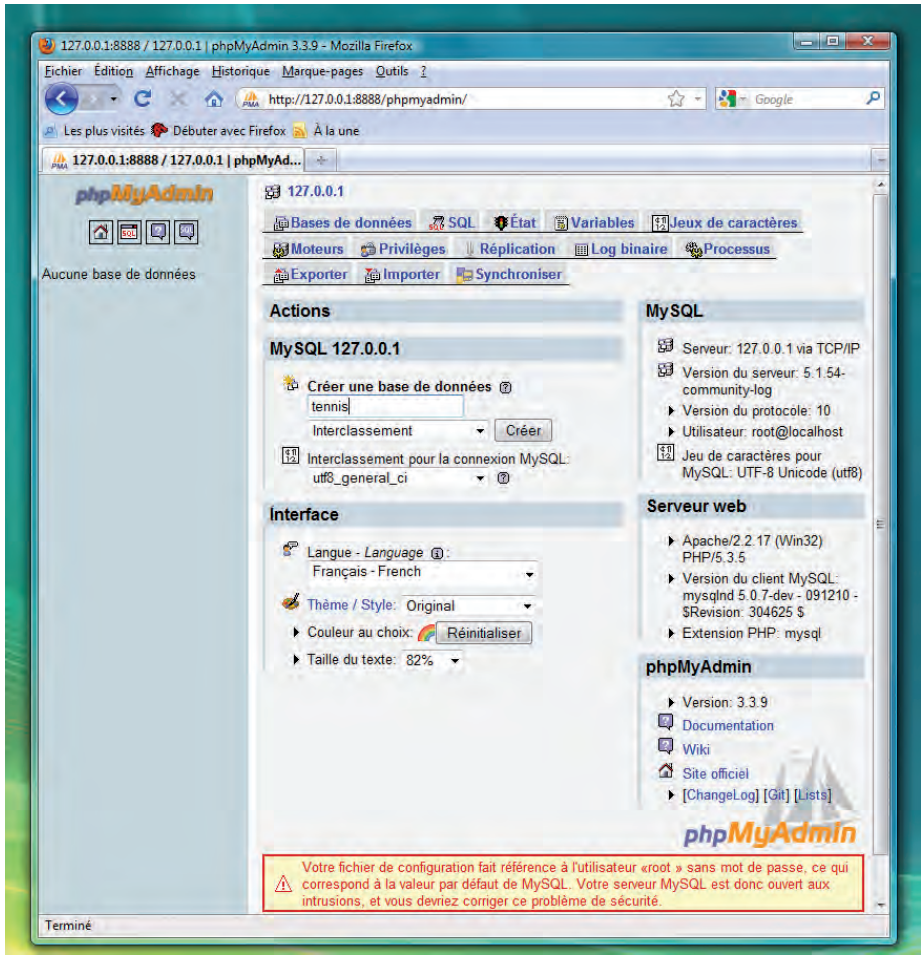
Cela sera le cas, même si la machine utilisée n'est pas connectée à Internet, puisqu'il s'agit d'une adresse locale. Bien entendu, si elle n'est pas connectée, personne d'autre que nous ne pourra visualiser les pages web créées. Si elle est connectée, les pages seront accessibles en tapant dans la barre l'adresse IP de la machine utilisée. Quand on crée un serveur web, il ne faut pas oublier de sécuriser la base de données par mot de passe, et d'ouvrir le port correspondant – par exemple 8888 – sur les divers pare-feu de la machine. On utilisera, dans ce cas, de préférence le port 80 qui peut être reconfiguré en faisant un clic droit sur l'icône d'EasyPHP et en choisissant le menu configuration->Apache et en remplaçant la ligne `Listen 127.0.0.1 :8888` par `Listen 127.0.0.1 :80`.

En lançant le navigateur, l'adresse qui apparaît dans la barre est `http://localhost:8888/` ou `http://127.0.0.1:8888/`. Si aucun numéro de port n'est indiqué, c'est le port 80 qui est utilisé par défaut.



Navigateur web.

La page affichée dans le navigateur présente le contenu du répertoire *www*, elle contient donc le lien nommé *phpmyadmin*. Si nous ajoutons d'autres fichiers dans ce répertoire et que nous rechargeons la page, nous les verrons apparaître dans la liste. En cliquant sur le lien *phpmyadmin* nous lançons l'interface graphique PHPMyAdmin, qui est en fait un ensemble de programmes PHP permettant de gérer la base de données. Nous voyons s'afficher la fenêtre



La page principale EasyPHP.

Nous considérons dans la suite que nous n'avons pas mis de mot de passe – ce qui est la configuration par défaut, bien que ce ne soit pas recommandé.

Schéma de la base

Nous construisons une base de données sur les matchs de tennis. La définition des tables est la suivante.

```
joueur(nom :chaîne, prenom :chaîne, age :entier,
                nationalite :chaîne)
```

```
rencontre(nomgagnant, nomperdant, nomtournoi, annee :entier,
                                                    score :chaîne)
gain(nomjoueur, nomtournoi, annee :entier, rang :chaîne,
                                                    prime :entier)
sponsor(nom, nomtournoi, annee :entier, montant :entier)
                tournoi(nom :chaîne, pays :chaîne)
```

Le schéma de la base.

Les clés primaires sont soulignées, certaines sont multivaluées. Pour des raisons de simplicité, nous avons choisi le `nom` du joueur comme clé primaire. Cela signifie qu'il ne peut pas y avoir deux joueurs possédant le même nom de famille. Pour simplifier, le score sera une chaîne de caractères, par exemple `"6/3-6/4"`, et le rang également, `"gagnant"`, `"finaliste"`, etc.

Voici maintenant la définition des contraintes de clés étrangères :

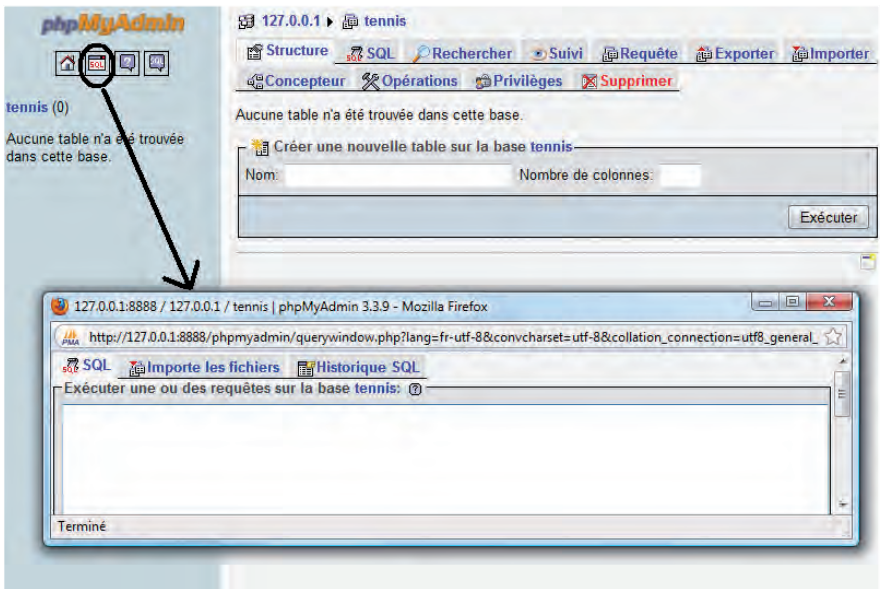
```
rencontre(nomgagnant) REFERENCES joueur(nom)
rencontre(nomperdant) REFERENCES joueur(nom)
rencontre(nomtournoi) REFERENCES tournoi(nom)
gain(nomjoueur) REFERENCES joueur(nom)
gain(nomtournoi) REFERENCES tournoi(nom)
sponsor(nomtournoi) REFERENCES tournoi(nom)
```

Notons que, d'une manière générale, il vaut mieux utiliser des noms sans espaces et n'utiliser que des lettres minuscules, sans accents, pour éviter les confusions.

Nous pouvons créer autant de bases différentes que nous voulons, et également gérer les droits des utilisateurs par rapport à ces bases, ce qui peut être utile dans le cas de la configuration d'un serveur utilisé par plusieurs utilisateurs. Alternativement, et c'est la solution que nous préconisons, il est possible d'installer le logiciel easyPHP sur chacun des ordinateurs utilisés.

Création de la base Nous remplissons le champ *Créer une base de données* avec le nom que nous souhaitons donner à notre base, par exemple `tennis`. Le champ interclassement permet d'indiquer comment sont stockées les chaînes de caractères dans la base, en particulier s'il faut faire une différence entre les majuscules et les minuscules – CS signifie : sensible à la casse et CI : insensible à la casse –, puis le type de langue à utiliser.

En cliquant sur le nom de la base de données `tennis`, nous voyons la liste des tables présentes dans cette base : pour l'instant, aucune. Cette touche permet de revenir en quelque sorte au menu principal de la base de données. Il est possible notamment de créer ou d'effacer des tables. L'interface d'administration se divise en deux parties : la partie gauche qui liste les tables déjà disponibles dans la base de données, et la partie droite d'administration à proprement parler. Dans la partie gauche, il y a une petite icône « SQL » qui sert à ouvrir une fenêtre pour entrer des instructions SQL. Cette fenêtre est à ouvrir lorsque nous aurons sélectionné la base. Nous y reviendrons plus tard.



Base de données Tennis.

Création des tables Il existe deux manières de créer des tables, l'une est graphique, l'autre utilise des instructions SQL. La manière graphique est plus simple, mais elle ne permet pas de gérer beaucoup de contraintes, en particulier de gérer les contraintes de clés étrangères, ce qui est trop limitatif dans notre exemple. Nous devons donc commencer en créant les tables qui n'ont pas de clés étrangères, puisque ces clés référencent des champs de tables déjà existantes.

Question 1

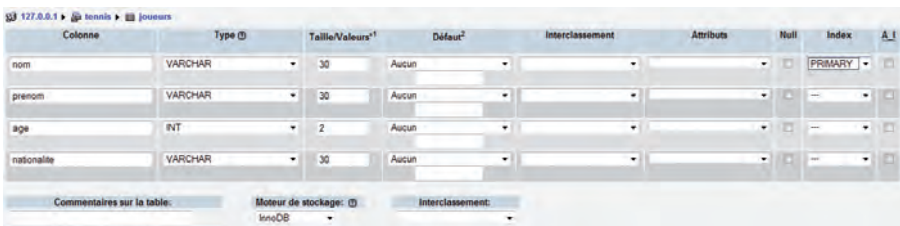
Donner un ordre correct pour la création des tables.

Correction. Nous pouvons créer dans l'ordre que nous souhaitons les tables qui n'ont pas de clé étrangère, c'est-à-dire la table `joueur` et la table `tournoi`. Dès que nous avons créé une table, nous pouvons créer toutes les tables qui ne font référence qu'à des tables existantes. Ainsi, dès que nous avons créé la table `tournoi`, nous pouvons créer la table `sponsor`.

Question 2

Créer les tables, en respectant bien les contraintes d'intégrité référentielles.

Correction. Dans l'interface graphique, nous entrons le nom de la table, par exemple `joueur`, et le nombre de colonnes, par exemple 4. S'ouvre alors une nouvelle page permettant de configurer les colonnes de la table. Il est possible d'ajouter ou de supprimer des colonnes par la suite, même si cela n'est pas recommandé, en particulier s'il y a déjà des données dans la table : car une valeur par défaut sera affectée pour cette nouvelle colonne aux données déjà présentes. Il ne faut remplir que les champs : *Champ*, *Type*, *Taille* et *Index*. Index permet de définir en particulier le fait qu'un attribut fait partie de la clé primaire (`PRIMARY KEY`). Nous illustrons cela dans la figure ci-après. Il faut indiquer les lignes des attributs qui participent à la clé primaire, ici seulement l'attribut `nom`. Les champs de type chaîne de caractères utilisent le type `VARCHAR`. La taille indique le nombre de caractères maximal. Les champs numériques utilisent le type `INT` pour les entiers, la taille indique le nombre de chiffres, ou `DECIMAL` pour les nombres à virgule. Dans ce cas, la taille est composée de deux valeurs X, Y où X représente le nombre de chiffres total et Y le nombre de chiffres après la virgule. Il est également très important de choisir comme type de la table la valeur `InnoDB`, afin de permettre la gestion des clés étrangères. Nous ne nous préoccupons pas des autres valeurs. Enfin, nous cliquons en bas à droite de l'écran sur le bouton `Sauvegarder` pour finaliser la création de la table.



Création de la table joueur.

Lorsque nous créons une table, une requête SQL est générée. Nous conservons cette requête dans un fichier texte pour une utilisation ultérieure :

```
CREATE TABLE ‘joueur‘ (
 ‘nom‘ VARCHAR( 30 ) NOT NULL,
‘prenom‘ VARCHAR( 30 ) NOT NULL,
‘age‘ INT( 3 ) NOT NULL,
‘nationalite‘ VARCHAR( 30 ) NOT NULL,
PRIMARY KEY ( ‘nom‘ )
);
```

Les guillemets sont générés automatiquement et ne servent qu'à l'utilisation de noms de colonnes avec des espaces. D'une manière générale, si nous n'utilisons pas d'espaces, il est inutile d'utiliser ces guillemets, qui ne sont accessibles qu'avec la touche clavier `ALT-GR-7` et non pas la simple touche 4.

Nous créons, de même, la table `tournoi`, qui possède 2 colonnes, en utilisant l'éditeur graphique. Ensuite, nous utilisons la fenêtre SQL pour écrire le code SQL de création de tables nécessitant une ou plusieurs clés étrangères. Pour ce faire, nous écrivons une requête similaire à celle que nous venons de générer, en ajoutant les lignes de clés étrangères :

```
CREATE TABLE rencontre(
nomgagnant varchar(30,
nomperdant varchar(30,
nomtournoi varchar(30,
annee int(4,
score varchar(15),
PRIMARY KEY (nomgagnant, nomperdant, nomtournoi, annee),
FOREIGN KEY (nomgagnant) REFERENCES joueur (nom),
FOREIGN KEY (nomperdant) REFERENCES joueur (nom),
FOREIGN KEY (nomtournoi) REFERENCES tournoi (nom)
) ENGINE=InnoDB;
```

Pour lancer la création de la table, il faut appuyer sur « exécuter ». Nous créons ensuite les tables `sponsor` et `gain` avec les requêtes suivantes, à taper dans la fenêtre SQL. Même s'il est possible d'exécuter plusieurs requêtes à la suite, nous suggérons d'en taper une, puis de l'exécuter, et ensuite de taper la suivante, en vérifiant qu'elle a bien fonctionné.

```
CREATE TABLE sponsor(
nom varchar(30),
nomtournoi varchar(30),
```

```
annee int(4),
montant int(7),
PRIMARY KEY (nom, nomtournoi, annee),
FOREIGN KEY (nomtournoi) REFERENCES tournoi (nom)
) ENGINE=InnoDB;

CREATE TABLE gain(
nom varchar(30),
nomtournoi varchar(30),
annee int(4),
rang varchar(30),
prime int(6),
PRIMARY KEY (nom, nomtournoi, annee),
FOREIGN KEY (nom) REFERENCES joueur (nom),
FOREIGN KEY (nomtournoi) REFERENCES tournoi (nom)
) ENGINE=InnoDB;
```

Une fois ces tables créées, nous visualiserons le schéma de la base en cliquant sur le lien `tennis`, puis sur l'onglet `concepteur`

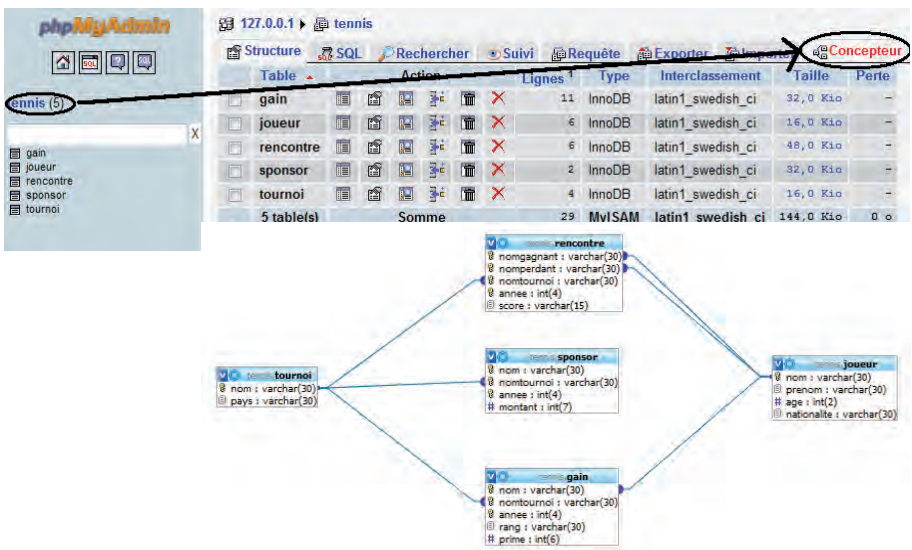
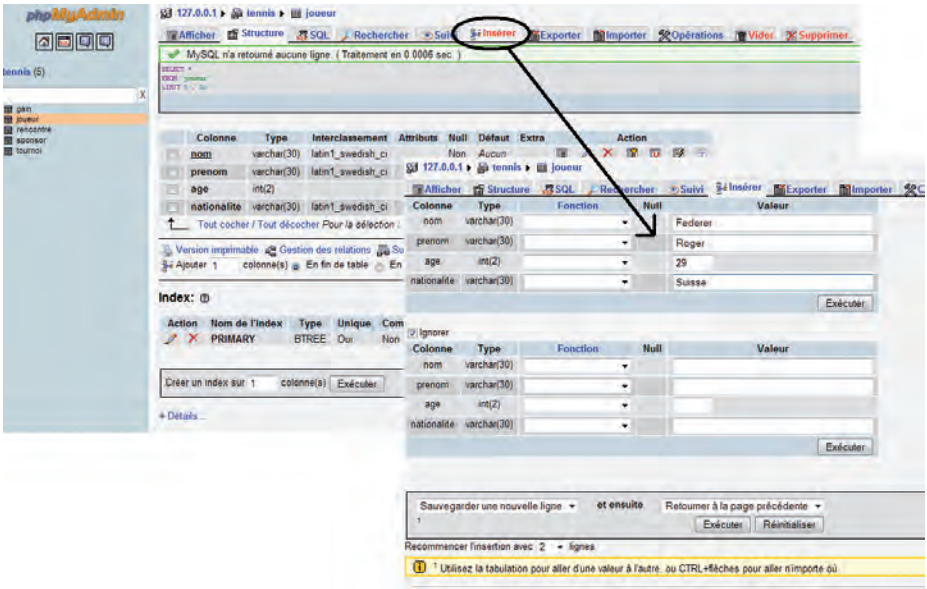


Schéma de la base.

Cela permet de contrôler que les contraintes d'intégrité référentielles sont bien placées correctement.

Insertion de données

En cliquant sur le nom d'une table dans le menu de gauche – par exemple `joueur` –, puis sur insérer, nous arrivons à un formulaire qui permet de remplir une table avec des données.



Formulaire d'insertion.

Une fois de plus, le code SQL est généré automatiquement et peut servir de patron pour écrire d'autres lignes d'insertion.

```
INSERT INTO joueur (nom, prenom, age, nationalite) VALUES
('Federer', 'Roger', '29', 'Suisse');
```

Nous devons nous rappeler les contraintes lorsque nous faisons des insertions : comme `nom` est la clé primaire de la table `joueur`, il sera impossible d'entrer deux fois un joueur dont le nom serait *Federer*. De même, il est impossible de créer une rencontre entre deux joueurs, si nous ne les avons pas préalablement insérés dans la table `joueurs`. De même, pour la table `sponsor`, il faut d'abord avoir entré le tournoi correspondant.

Question 3.

Ajoutez les joueurs Federer, Nadal, Murray, Soderling, Berdych et Djokovic.

Correction

```
INSERT INTO joueur (nom, prenom, age, nationalite) VALUES
('Nadal', 'Rafael', '24', 'Espagne'),
('Murray', 'Andrew', '23', 'Ecosse'),
('Soderling', 'Robin', '26', 'Suede'),
('Berdych', 'Tomas', '26', 'Rep. Tcheque'),
('Djokovic', 'Novak', '23', 'Serbie');
```

Nous vérifions que toutes les valeurs sont bien dans la table en cliquant sur `joueur` dans la barre de menu de gauche.

Question 4

Ajouter le sponsor BNP-Paribas au tournoi de Roland-Garros 2010 avec un montant de 9.000.000 et au tournoi Roland-Garros 2011 avec un montant de 10.000.000.

Correction. Il faut commencer par créer le tournoi de Roland-Garros, puis ajouter le sponsor pour l'année donnée. Une fois le tournoi créé, nous pourrons ajouter tous les sponsors que nous voulons.

```
INSERT INTO tournoi (nom, pays) VALUES
('Roland-Garros, 'France');
```

```
INSERT INTO sponsor (nom, nomtournoi, annee, montant) VA-
LUES
('BNP-Paribas', 'Roland-Garros, '2010', '9000000'),
('BNP-Paribas', 'Roland-Garros, '2011', '10000000');
```

Question 5

Ajouter les matchs suivants :

– Open d'Australie 2010 : Federer bat Murray (6/3-6/4-7/6)
– Roland-Garros 2010 : Nadal bat Soderling (6/4-6/2-6/4)
– Wimbledon 2010 : Nadal bat Berdych (6/3-7/5-6/4)
– US Open 2010 : Nadal bat Djokovic (6/4-5/7-6/4-6/2)
– Open d'Australie 2011 : Djokovic bat Murray (6/4-6/2-6/3)

Remplir également la table `gain`, en sachant que le gagnant remporte 1.000.000 et le finaliste 500.000.

Correction. Il faut d'abord créer les tournois inexistants. Le caractère d'échappement « \' » doit être utilisé pour indiquer l'apostrophe.

```
INSERT INTO tournoi (nom, pays) VALUES
```

```
('Open d\'Australie', 'Australie'),
('Wimbledon', 'Royaume Uni'),
('US Open', 'États-Unis');
```

Les joueurs ayant déjà été créés, nous entrons alors les rencontres

```
INSERT INTO rencontre (nomgagnant, nomperdant, nomtour-
noi, annee, score) VALUES
('Federer', 'Murray', 'Open d\'Australie', '2010', '6/3-6/4-
7/6'),
('Nadal', 'Soderling', 'Roland-Garros', '2010', '6/4-6/2-6/4'),
('Nadal', 'Berdych', 'Wimbledon', '2010', '6/3-7/5-6/4'),
('Nadal', 'Djokovic', 'US Open', '2010', '6/4-5/7-6/4-6/2'),
('Djokovic', 'Murray', 'Open d\'Australie', '2011', '6/4-6/2-
6/3');
```

Nous pouvons ensuite générer automatiquement les gains par les deux requêtes suivantes, que nous pouvons, bien entendu, aussi entrer à la main

```
INSERT INTO gain (nom, nomtournoi, annee, rang, prime)
SELECT nomgagnant, nomtournoi, annee, 'gagnant', 1000000
FROM rencontre;

INSERT INTO gain (nom, nomtournoi, annee, rang, prime)
SELECT nomperdant, nomtournoi, annee, 'finaliste', 500000
FROM rencontre;
```

Requêtes d'interrogation

Si l'on entre davantage de données dans la base, les requêtes donnent des résultats plus amusants. Lors d'une séance d'exercice, cette étape prend souvent beaucoup de temps, et il est préférable de distribuer un fichier contenant des instructions, `INSERT`. Ces fichiers peuvent être importés et exécutés via le menu prévu à cet effet dans PHPMyAdmin.

Voici quelques exemples de questions à poser, avec le code SQL de la requête correspondante. Bien entendu, selon le contenu de la base, l'énoncé de la question peut être modulé.

Requêtes faciles

– Afficher le `nom` des joueurs dont le `prenom` est Roger.

```
SELECT j.nom
FROM joueur j
WHERE j.prenom = 'Roger'
```

– Afficher les annee où s'est déroulé Roland-Garros. Indication : un tournoi s'est déroulé s'il y a des rencontres dans la base.

```
SELECT r.annee
FROM rencontre r
WHERE r.nomtournoi = 'Roland-Garros'
```

Requêtes avec jointure

– Afficher le nom et l'age des joueurs ayant gagné à Roland-Garros, peu importe l'année.

```
SELECT j.nom, j.age
FROM rencontre r, joueur j
WHERE r.nomgagnant = j.nom
AND r.nomtournoi = 'Roland-Garros'
```

– Afficher le nom des sponsors ayant sponsorisé un tournoi ayant lieu en France

```
SELECT s.nom
FROM sponsor s, tournoi t
WHERE t.nom = s.nomtournoi
AND t.pays = 'France'
```

Nous obtenons ici 2 résultats avec la même valeur, comme BNP-Paribas a sponsorisé un tournoi français pendant plusieurs années. On peut ajouter le mot-clé `distinct` pour enlever les doublons, ou bien faire un regroupement dans une clause `GROUP BY`.

– Afficher le nom et prenom des joueurs ayant gagné un tournoi sponsorisé par BNP-Paribas

```
SELECT j.nom, j.prenom
FROM joueur j, rencontre r, sponsor s
WHERE j.nom = r.nomgagnant
AND r.nomtournoi = s.nomtournoi
AND s.nom = 'BNP-Paribas'
```

Requêtes ensemblistes ou imbriquées

– Afficher le nom de tous les joueurs ayant joué au moins un match.

```
SELECT r.nomgagnant as nom
FROM rencontre r
UNION
```

```
SELECT r.nomperdant as nom
FROM rencontre r
```

– Afficher le nom des joueurs ayant joué à Wimbledon quelle que soit l'annee.

```
SELECT r.nomgagnant as nom
FROM rencontre r
WHERE r.nomtournoi = 'Wimbledon'
UNION
SELECT r.nomperdant as nom
FROM rencontre r
WHERE r.nomtournoi = 'Wimbledon'
```

– Afficher le nom des joueurs ayant gagné tous leurs matchs.

```
SELECT r.nomgagnant as nom
FROM rencontre r
EXCEPT
SELECT r.nomperdant as nom
FROM rencontre r
```

La commande EXCEPT n'étant pas implémentée en MySQL, il faut utiliser une requête imbriquée.

```
SELECT distinct (r.nomgagnant) as nom
FROM rencontre r
WHERE r.nomgagnant NOT IN (
SELECT p.nomperdant as nom
FROM rencontre p
)
```

– Afficher le nom des joueurs n'ayant jamais gagné moins de 1.000.000 par tournoi.

```
SELECT distinct(g.nom)
FROM gain g
WHERE g.prime >= 1000000
AND g.nom
NOT IN (
SELECT h.nom
FROM gain h
WHERE h.prime <1000000
)
```

On aurait de même pu utiliser un EXCEPT.

Requêtes agrégats

– Compter le nombre total de matchs joués à l'US Open et afficher le résultat dans une colonne nommée `nbmatchs`.

```
SELECT COUNT(*) as nbmatch
FROM rencontre r
WHERE r.nomtournoi = 'US Open'
```

– Calculer les gains totaux de 'Nadal' dans une colonne 'gaintotal'

```
SELECT SUM(g.prime) as gaintotal
FROM gain g WHERE g.nom = 'Nadal'
```

– Afficher le `nom` et les `gains` (dans une colonne 'gaintotal') des joueurs ayant gagné au moins 1000000 au total.

```
SELECT g.nom, SUM(g.prime) as gaintotal
FROM gain g
GROUP BY g.nom
HAVING SUM(g.prime) >= 1000000
```

– Afficher les `nom` et `annee` des tournois où la somme versée par les sponsors est supérieure aux gains reversés aux participants.

```
SELECT s.nomtournoi, s.annee
FROM sponsor s, gain g
WHERE g.nomtournoi = s.nomtournoi
AND g.annee = s.annee
GROUP BY g.nomtournoi, g.annee
HAVING SUM(montant) > SUM(prime)
```

Notons que cette requête ne retourne pas les tournois pour lesquels il n'y a pas eu de matchs, car la somme sur l'ensemble vide n'est pas définie.

Exercices non corrigés

Dans un exercice consacré aux bases de données, le plus classique est de demander l'écriture de requêtes permettant de répondre à certaines questions. Il faut donc préalablement avoir défini la structure de la base de données, soit par son schéma entité-association, soit en donnant directement les tables relationnelles, ce qui est souvent préférable, car ainsi tous les élèves utiliseront les mêmes noms de tables. On peut également demander aux élèves de modéliser une situation à partir d'un texte et de produire un modèle entité-association et puis des tables. Cette question est en réalité *difficile*. En pratique, elle demande souvent des allers-retours avec la personne ayant écrit le texte, ce qu'il est possible de faire en TD, mais impossible en examen. Nous conseillons donc dans

un premier temps de se restreindre à des requêtes sur un schéma déjà existant, ou bien à un énoncé très précis à modéliser.

Requêtes sur la base de données des joueurs de football

Exercice 1

Il faut s'imaginer que la table contient un grand nombre de données – par exemple, toutes les informations sur la L1 et L2 de football.

Requêtes sur une seule table sans jointure :

1. Nom et NomCourt de toutes les équipes.
2. NomCourt de toutes les équipes domiciliées à Paris.
3. Nom de toutes les équipes dont la ville commence par un « L ».
4. Nom de tous les joueurs dont le prénom est « Stéphane ».

Requêtes avec jointure :

1. Nom des équipes où joue au moins un joueur dont le nom commence par un « A ».
2. Nom des équipes où il n'y a aucun joueur dont le nom commence par « A ».
3. Nom des joueurs ayant participé à un match où au moins un but a été marqué.
4. Nom des équipes n'ayant marqué aucun but.
5. Nom des joueurs ayant joué un match le 19/12/10.
6. Dates des matchs où un joueur de plus de 35 ans a joué.

Requêtes avec agrégats :

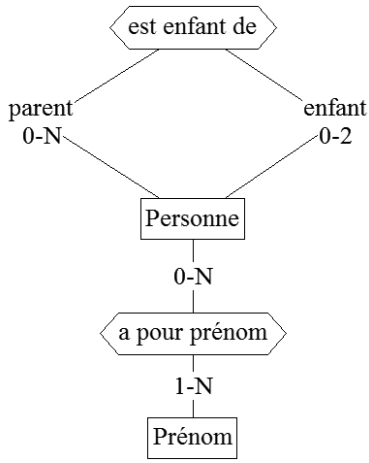
1. Moyenne d'âge de l'ensemble des joueurs.
2. Moyenne d'âge par équipe : indiquer le nom de l'équipe dans une colonne et la moyenne d'âge dans une autre colonne nommée AgeMoyen.
3. Nom et Prénom du joueur le plus vieux.
4. Nom et Prénom du joueur le plus vieux de chaque équipe.
5. Nom et Prénom du joueur le plus vieux de chaque rencontre.
6. Nombre total de buts inscrits par chaque équipe.
7. Nom des équipes ayant marqué au total plus de 40 buts.
8. Nom des équipes ayant marqué au total moins de 10 buts.
9. Nombre total de matchs joués par chaque équipe.

Une base de données généalogique

Exercice 2

Dans cet exercice, on considère une base de données très simple sur des personnes. Une personne doit posséder de 0 à 2 parents et chaque personne

peut avoir de 0 à *N* enfants. Chaque personne possède un nom, un numéro de Sécurité sociale (NSS) qui est un identifiant unique, un sexe (M/F). Un prénom est défini par un attribut valeur qui est également l'identifiant unique de l'entité. Nous avons construit une partie du modèle entité-association :



Modèle entité-association des personnes.

Questions de cours

1. Le modèle entité-association n'est pas complet. Ajouter les attributs nécessaires. Redessiner l'ensemble du modèle sur une feuille pour correspondre au texte explicatif.
2. Pourquoi a-t-on voulu créer une entité Prenom au lieu d'utiliser tout simplement un attribut ?
3. À quoi correspondent sur le schéma les mots parent et enfant ?
4. À quoi correspond la notation 0-2 sur le schéma ?

Dans les questions suivantes, on a transformé le modèle entité-association en modèle relationnel suivant, dans lequel les clés primaires sont soulignées.

```
prenom(valeur :chaîne)
personne(NSS :entier, nom :chaîne, sexe :chaîne)
apourprenom(NSS :entier, valeur :string)
enfantde(NSSParent :entier, NSSEnfant :entier)
```

1. Donner les contraintes de clés étrangères de ce schéma, afin de respecter le modèle entité-association.

2. En utilisant les tables du modèle relationnel, donner la requête SQL permettant d'afficher les NSS de tous les petits-enfants de toutes les personnes ayant comme prénom « Joe ».

3. Avec le modèle relationnel de base, peut-on écrire une requête permettant de trouver tous les descendants d'une personne donnée ? Pourquoi ?

4. Donner l'algorithme d'un programme, incluant des requêtes SQL et des instructions de boucle qui permettraient de réaliser une telle fonctionnalité.

La base de données d'un moteur de recherche

Exercice 3

Nous considérons, dans cet exercice, un moteur de recherche sur le Web, comme Google. Pour simplifier, nous ne nous intéressons pas au *ranking* des pages, c'est-à-dire à leur importance, mais uniquement au problème de retourner des pages contenant un mot-clé précis. Nous utilisons, pour ce faire, la base de données appelée recherche dont le schéma relationnel est donné dans la figure suivante, où les clés primaires sont soulignées.

```
page(URL : chaîne, titre : chaîne)
dico(id : entier, mot : chaîne)
index(id : entier, URL : chaîne)
Avec les contraintes de clé étrangères :
index(URL) REFERENCES page(URL) index(id) REFERENCES
dico(id)
```

Le champ titre de la table page représente le contenu de la balise HTML `<title>` de la page correspondant à l'URL. Nous ne nous posons pas le problème de savoir comment nous avons obtenu les mots de la page à partir du code HTLM de la page, qui est un problème en soi. Nous supposons données la table page et la table index correspondant à un ensemble de pages web.

1. Nous voulons ajouter la page d'URL `http://www.uvsq.fr/in111/cours.html` dans la base. Supposons que cette page contienne les mots « base de données », « cours » et « informatique » et que son titre soit « IN111 ». Sans écrire de code, expliquer ce qu'il faudra faire pour compléter correctement les tables de la base.

2. Donner les URL des pages dont le titre est « base de données ».

3. Donner les couples d'URL ayant au moins un mot en commun. Le résultat de la requête devra contenir deux champs `page1` et `page2`.

4. Donner, pour chaque mot apparaissant dans la base, le nombre de pages différentes dans lesquelles il apparaît et classer ces mots par ordre de fréquence croissante.

5. Donner les URL des pages contenant au moins 100 mots différents.

6. Donner la liste des mots apparaissant dans au plus 10 pages différentes.

7. Donner la liste des mots apparaissant dans toutes les pages de la base de données.

Une base de données géographique

Exercice 4

On se donne le schéma suivant d'une base de données.

```
Pays (NumPays :entier, Nom :chaîne, RefPresident :en-
                          tier,
                RefContinent :entier)
Presidents (NumPresident :entier, Nom :chaîne, Prenom
                :chaîne, Age :entier)
    Continents (NumContinent :entier, Nom :chaîne)
      Superficie (RefPays :entier, Valeur :réel)
     Population (RefPays :entier, Valeur :entier)
```

Les clés primaires sont indiquées en souligné. Les clés étrangères sont données par les contraintes suivantes :

```
Pays(RefPresident) references Presidents(NumPresident)
Pays(RefContinent) references Continents(NumContinent)
     Superficie(RefPays) references Pays(NumPays)
     Population(RefPays) references Pays(RefPays)
```

Cette base de données simpliste permet de modéliser quelques informations sur les pays du monde. La table `Presidents` contient des informations sur les présidents en exercice – on appelle « Président » toute personne qui dirige un pays. La table `Continents` contient les noms des continents, par exemple Europe, Afrique, Amérique du Nord, Asie, etc. Les tables `superficie` et `population` donnent des informations supplémentaires sur les pays.

1. Dans quel ordre doit-on créer les tables dans la base de données ?

2. Avec la structure proposée, un pays peut-il avoir plusieurs présidents ? Un président peut-il être président de plusieurs pays différents ?

3. Donner un schéma dans le format entité-association de cette base qui soit compatible avec les tables relationnelles.

4. Proposer un autre schéma relationnel avec moins de tables. Quel est l'intérêt de la modélisation proposée dans l'énoncé ?

5. Nom du pays dont le NumPays = 1.

6. Donner le nom du président de la France.

7. Donner le nom du pays d'Europe le plus peuplé.

8. Donner la superficie totale de l'Europe.

9. Donner la somme totale du nombre de personnes habitant dans un continent d'une superficie < 15.000.000 km^2.

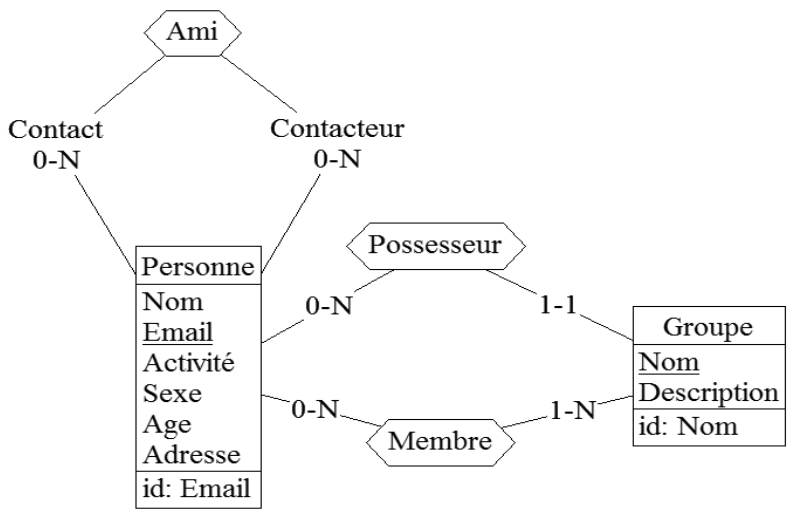
10. Compter le nombre de pays d'Afrique dont la population est > 50.000.000 d'habitants.

11. Donner la liste de tous les pays dont la densité, c'est-à-dire le quotient de la population par la superficie, est inférieure à 20.

La base de données d'un réseau social

Exercice 5

`Facebook.com` est un logiciel de réseau social qui permet à des amis de se constituer en réseaux sociaux. Ce site, qui existe depuis 2004, a atteint dernièrement une popularité record, avec près de 600 millions d'utilisateurs actifs en 2011. Nous allons étudier dans cet exercice une modélisation simpliste du système. Nous considérons dans un premier temps le modèle entité-association suivant.



1. Donner les instructions CREATE TABLE nécessaires pour implémenter ce modèle. Penser à bien définir les clés primaires de chaque table.

2. On souhaite indiquer que le couple (nom, adresse) est une clé candidate. Comment faire ?

3. On souhaite ajouter un attribut photo à l'entité Personne. Donner l'instruction SQL permettant de le faire. Cet attribut sera de type BLOB (*Binary Large OBject)*. De même, on veut ajouter un attribut password. Quels types SQL peut-on utiliser pour cet attribut ? Donner les avantages et inconvénients de chacun.

Requêtes SQL :

1. Nom, Prenom et email de tous les utilisateurs de la base.

2. Email de tous les amis de l'utilisateur `robert@tartempion.fr`.

3. Email de tous les amis des amis de l'utilisateur `robert@tartempion.fr`. Peut-on donner l'email de toutes les personnes liées à un degré de profondeur quelconque à `robert@tartempion.fr` ?

4. Nombre de membres du groupe *Club Tennis Versaillais*.

5. Pour chaque groupe, le nom du groupe et l'âge moyen des membres de tous les groupes appartenant à `robert@tartempion.fr`.

6. Donnez l'email des personnes dont tous les amis font partie de tous les groupes dont fait partie cette personne.

7. Donnez les noms des groupes où certains des membres ne se connaissent pas directement.

Exercice de modélisation : une banque

Exercice 6

La Banque des Petits Suisses (BPS) souhaite réaliser une base de données pour gérer les comptes de ses clients. Les informations à stocker sont les suivantes : chaque client est identifié par un numéro de client unique et possède un nom, un prénom et une adresse. Un compte est composé d'un numéro de compte et d'un montant initial déposé à l'ouverture du compte. Chaque client peut posséder autant de comptes qu'il le souhaite. La banque souhaite ensuite conserver une trace de toutes les opérations effectuées sur les comptes, qui peuvent être soit des dépôts soit des retraits.

Réaliser le modèle entité-association qui permettra de représenter les données de la BPS, puis donner le schéma relationnel correspondant. Écrire une requête permettant de calculer le solde d'un compte dont on connaît le numéro.

Questions d'enseignement

Dans ce paragraphe, nous donnons quelques pistes de réflexion pour mettre en place un cours d'introduction aux bases de données. Il est structuré autour des questions qui se posent quand on organise un tel cours.

Quels objectifs se fixer ?

Dans ce type de cours, il est raisonnable de s'orienter vers l'utilisation des bases de données. En effet, les cours de ce niveau sont suivis par des élèves qui vont poursuivre des études d'informatique, mais aussi par d'autres qui vont suivre des voies très différentes. À l'issue du cours, les élèves devront avoir acquis les connaissances nécessaires pour manipuler une base de données et un système de gestion de bases de données.

Quels thèmes aborder ?

Les sujets que l'on peut traiter dans un cours consacré aux bases de données sont nombreux et vastes. Le choix de l'objectif ci-avant va toutefois nous guider.

Tout d'abord, il faut décider quels modèles logiques présenter. Il se trouve que le modèle relationnel possède deux atouts importants : c'est le modèle le plus répandu, et il repose sur un nombre réduit de concepts à assimiler. C'est donc un choix assez naturel.

Les concepts du modèle relationnel vont être présentés. Pour faciliter leur assimilation, il est intéressant de s'appuyer sur des exemples présentés sous forme de tableau. Le lien entre n-uplet et ligne et entre attribut et colonne permet de mieux comprendre les définitions formelles.

La deuxième décision concerne le choix du langage d'interrogation. Il est difficile de faire l'économie de SQL, qui est le langage standard des bases de données relationnelles. Cependant, il peut être plus simple de ne pas introduire directement la syntaxe du langage. Une approche possible consisterait à utiliser une interface pour les premiers contacts. Par exemple, le logiciel *PhpMyAdmin* fournit une interface d'administration pour le système de gestion de bases de données *MySQL*. L'intérêt de ce type d'outil est qu'en plus de l'interface conviviale, il montre le SQL exécuté pour chaque action, ce qui facilite l'acquisition de la syntaxe. Une autre solution serait de débuter par le langage *Query by Example*. Il est par exemple implémenté dans le système de gestion de bases de données *Microsoft Access* qui fait partie de la suite bureautique *Office*. Ce langage est un langage graphique et intuitif pour l'interrogation d'une base de données. Les requêtes QBE peuvent être traduites facilement en SQL pour aider à acquérir ce dernier. Enfin, concernant SQL, il est important de se foca-

liser sur la norme et non pas sur l'un des nombreux dialectes implantés au sein des systèmes de gestion de bases de données. En effet, tant sur le point de la syntaxe que de la sémantique, certains systèmes de gestion de bases de données prennent d'importantes libertés vis-à-vis de la norme.

Une fois ces deux points traités, l'objectif fixé est atteint : un élève peut manipuler un système de gestion de bases de données. Pour compléter le cours, on suivra plusieurs pistes. Aborder la conception de bases de données donnera à l'élève des éléments pour élaborer ses propres bases de données. Étudier les mécanismes internes des systèmes de gestion de bases de données, comme par exemple l'optimisation de questions, montrera les problèmes que résolvent les systèmes de gestion de bases de données. Enfin, l'ouverture vers des langages de programmation et vers le Web donnera des perspectives intéressantes et attractives.

Comment organiser le cours ?

L'acquisition des connaissances décrites ci-avant nécessite une mise en pratique importante. Le cours doit s'appuyer sur des exemples pour illustrer les concepts nouveaux, voire intégrer des exercices ciblés. Les séances de travaux dirigés et pratiques serviront à asseoir les connaissances présentées en cours. Il est raisonnable de prévoir deux fois plus de travaux dirigés et de travaux pratiques que de cours.

En termes d'enchaînement des thèmes, après une introduction définissant et motivant les bases de données et les systèmes de gestion de bases de données, la présentation du modèle relationnel semble naturelle. Il serait également possible de débuter par la conception de bases de données avec le modèle entité-association par exemple. Toutefois, pour un cours d'introduction, ce choix impose de demeurer à un niveau d'abstraction élevé pendant plusieurs séances, ce qui n'est pas souhaitable. La suite logique consiste à présenter un langage d'interrogation. Une fois ces bases présentées, l'ordre des éventuels autres thèmes n'est pas contraint.

Quels outils choisir pour les travaux pratiques ?

Pour assimiler les concepts vus en cours, il est utile de les mettre en pratique. Pour cela, il faut tout d'abord choisir un système de gestion de bases de données. Il en existe de nombreux, tant commerciaux que gratuits ou libres. Nous allons ici suggérer quelques pistes de choix.

La première possibilité consiste à utiliser *Microsoft Access* qui est intégré à *Office* ou *OpenOffice Base* qui fait partie d'OpenOffice. Ces deux outils proposent une interface conviviale pour la création et la manipulation d'une base

de données relationnelle. Par défaut, l'interrogation se fait à l'aide d'un langage de type *Query by Example*, mais il est aussi possible d'accéder au code SQL. Le support de SQL dans ces deux outils reste cependant un peu limité.

Il existe également des systèmes de gestion de bases de données légers et simples à utiliser. Le système Firebird (`http://www.firebirdsql.org/`) en est un exemple. Ce genre de système propose en général toutes les fonctionnalités nécessaires pour l'apprentissage et est bien adapté à des bases de données de taille moyenne.

Une autre possibilité consiste à se tourner vers les systèmes de gestion de bases de données libres les plus connus : *PostgreSQL* (`http://www.postgresql.org/`) ou *MySQL* (`http://www.mysql.com/`). L'installation et la maintenance de ces outils sont parfois un peu plus complexes que les précédents. En contrepartie, les fonctionnalités disponibles sont d'un excellent niveau. De plus, ils s'intègrent bien dans l'installation plus globale d'un serveur web, comme par exemple avec le package *easyPHP*. Il est à noter que ces logiciels présentent l'avantage de pouvoir être installés par les utilisateurs sur leur ordinateur personnel.

Enfin, les systèmes de gestion de bases de données commerciaux phares comme Oracle, DB2 d'IBM ou SQL Server de Microsoft disposent d'une version gratuite plus légère, utilisable dans un contexte privé. Ces systèmes sont un peu plus délicats à installer et à administrer, mais leur manipulation aide à se faire une idée des outils utilisés dans un contexte professionnel.

En dehors des systèmes de gestion de bases de données, les outils CASE (*Computer-aided software engineering*) fournissent une aide pour la conception de bases de données. Le logiciel gratuit *DB-Main* (`http://www.db-main.be/`) pour la modélisation. Il permet entre autres de concevoir un diagramme entité-association puis de le transformer en relationnel.

Enfin, pour le développement, les environnements de développement intégré que sont *Eclipse* (`http://www.eclipse.org/`) et *Netbeans* (`http://netbeans.org/`) permettent de programmer dans divers langages.

Compléments

Dans ce chapitre, nous avons abordé un certain nombre de sujets à propos des bases de données et du Web. Certains des thèmes présentés mériteraient d'être développés et ne l'ont pas été par manque de place.

Concernant les systèmes de gestion de bases de données, l'étude des mécanismes internes comme la gestion des transactions ou l'organisation physique des données est nécessaire à une bonne compréhension et à un usage

plus poussé. Dans le même ordre d'idée, la compréhension des concepts de l'administration de bases de données permet une manipulation avancée des systèmes de gestion de bases de données. Sur le plan des modèles, d'autres modèles logiques que le modèle relationnel existent : les modèles réseau et hiérarchique sont antérieurs alors que les modèles objet, relationnel-objet et XML sont plus récents.

Système de gestion de bases de données

L'objectif d'un système de gestion de bases de données est de factoriser des modules de contrôle répondant aux besoins communs des applications : interrogation, cohérence, partage, etc. Plus précisément, ces objectifs sont les suivants : rendre les applications indépendantes du modèle physique de stockage comme le disque magnétique ou la mémoire flash – indépendance physique –, donner la possibilité de disposer de vues logiques de la base de données – indépendance logique –, fournir un langage d'interrogation déclaratif – un langage déclaratif est un langage qui permet de décrire le résultat que l'on souhaite obtenir, par exemple, « les joueurs de moins de 21 ans qui ont marqué plus de 10 buts », par opposition à un langage impératif, comme C ou Java, qui décrit une séquence d'opérations permettant d'arriver à la solution –, transformer automatiquement une question en programme – optimisation des questions –, gérer la cohérence des données, être robuste aux pannes – tolérance aux pannes –, gérer les accès concurrents de différents utilisateurs ou programmes, contrôler les accès aux données – gestion de la confidentialité – et enfin s'appuyer sur des standards.

Le groupe ANSI/X3/SPARC a proposé une architecture des systèmes de gestion de bases de données comportant trois niveaux. L'objectif de cette proposition est de rendre les vues des utilisateurs indépendantes de la façon dont les données sont stockées. Parmi ces trois niveaux, le niveau *externe* décrit le point de vue des utilisateurs, le niveau *conceptuel* stocke le schéma des données et le niveau *interne* gère le stockage physique des données. En plus des niveaux de représentation, la norme décrit les modules d'un système de gestion de bases de données. Une partie de ces modules est dédiée à la description de données, la seconde assure la manipulation des données. Ces deux ensembles de modules s'appuient sur le *dictionnaire de données*. Ce dernier est une base de données décrivant l'ensemble des objets manipulés par le système de gestion de bases de données.

Pour conclure sur les systèmes de gestion de bases de données, il nous faut évoquer quelques produits en commençant par un bref historique. Le premier prototype de système de gestion de bases de données relationnel

nommé *System/R* a été développé par IBM dans son centre de San José. Il donnera naissance à *SQL/DS*, puis à *DB2*. En parallèle, à l'université de Berkeley, le système *INGRES* est développé. Il supporte un langage de requête nommé *QUEL*. Le système *POSTGRES* succéda à *INGRES* et est aujourd'hui l'un des systèmes de gestion de bases de données *open source* le plus utilisé sous le nom *PostgreSQL*. *INGRES* a également donné naissance aux systèmes de gestion de bases de données *Informix et Sybase*. Ce dernier évolua ensuite vers *SQL Server* chez Microsoft. Les premières offres commerciales sont apparues avec *Oracle* en 1979, *SQL/DS* (IBM) en 1982, suivi de *base de données2* (IBM) en 1983. La première version du système *open source MySQL* date de 1994. Enfin, pour la gestion de bases de données de moindre taille, on peut citer *Microsoft Access*, *OpenOffice Base* et *Paradox*.

Le modèle relationnel

Le modèle relationnel est un modèle de données proposé en 1969 par Edgar T. Codd à IBM San-José. Dans sa forme initiale, seules des valeurs atomiques sont supportées, c'est-à-dire qu'il n'y a ni valeurs complexes, ni valeurs multivaluées. On parle de *première forme normale* (*1NF*). Une relation est en 1NF si le domaine de chacun de ses attributs ne contient que des valeurs atomiques. Ce type de relation est bien adapté aux applications de gestion. Cependant, cette contrainte peut être pénalisante dans le cas d'applications manipulant des objets complexes. C'est pour cette raison que des extensions au modèle relationnel ont vu le jour. On parle alors de bases de données en *non première forme normale* (NF^2). Parmi les extensions proposées, on peut citer le *modèle relationnel imbriqué* qui autorise qu'une valeur d'un domaine soit lui-même une relation et le *modèle objet-relationnel* qui intègre au relationnel les concepts objet.

Le modèle relationnel et les valeurs manquantes

Ce que nous avons présenté jusqu'alors du modèle relationnel supposait que chaque n-uplet possédait une valeur pour chaque attribut de la relation. Cependant, pour une application réelle, une telle supposition est déraisonnable. Il est donc nécessaire d'introduire un moyen de représenter l'absence de valeur – valeur manquante. Citons trois exemples de situations où cela peut se produire : l'information est pertinente mais n'existe pas pour l'entité correspondant au n-uplet – numéro d'immatriculation d'une personne ne possédant pas de voiture –, l'information n'est pas pertinente pour l'entité – le nom de jeune fille pour un homme – ou l'information existe mais est inconnue au moment de la saisie.

L'absence d'une valeur dans une relation est représentée par un marqueur spécial nommé *NULL*. L'utilisation d'un unique marqueur pour représenter les valeurs non renseignées pose le problème de son interprétation. Il est en effet impossible de déterminer à quelle situation correspond une valeur manquante dans une relation.

De plus, la prise en compte de ce marqueur dans les langages d'interrogation est complexe – logique à trois valeurs, logique à quatre valeurs. Nous verrons ci-après le cas du langage SQL.

En résumé, les marqueurs NULL sont délicats à manipuler et il est préférable de les éviter dans la mesure du possible. L'étude détaillée des implications de ce marqueur sort du cadre de cet ouvrage.

Les langages d'interrogation

Plusieurs langages ont été proposés pour l'interrogation d'une base de données relationnelle. Le *calcul relationnel de n-uplets* proposé par Codd est à l'origine des langages QUEL et SQL. Le *calcul relationnel de domaines* est plus proche de la logique du premier ordre. Le langage *Query by Example* (*QBE*) est une application pratique du calcul de domaines. Le langage *datalog* est un autre langage logique. Les relations entre les bases de données et la logique ont été intensivement étudiées. L'*algèbre relationnelle* est un ensemble d'opérations que l'on peut appliquer aux relations. Enfin, le langage *SQL* tire sa source du calcul de n-uplets et de l'algèbre relationnelle. Hormis l'algèbre relationnelle, ces langages sont tous déclaratifs, c'est-à-dire que l'on décrit ce que l'on veut obtenir mais pas la façon de l'obtenir. L'algèbre relationnelle est le seul langage à proposer des opérations qui doivent être ordonnées. L'étude de ces langages complets a montré qu'ils ont tous le même pouvoir d'expression. Ces langages sont dits *complets*, c'est-à-dire qu'ils ont le même pouvoir d'expression que la logique du premier ordre.

L'algèbre relationnelle

Parmi les opérateurs qui ont été décrits dans la littérature, certains peuvent être définis à partir d'autres. Il existe en fait plusieurs sous-ensembles minimaux d'opérateurs qui permettent d'écrire toute expression d'algèbre relationnelle. Par exemple, l'ensemble $\{\sigma,\pi,\times,\setminus,\cup\}$ est suffisant pour écrire toutes les formules d'algèbre relationnelle.

L'algèbre relationnelle décrit un ensemble d'opérateurs qui possèdent des propriétés algébriques. Par exemple, la propriété $\sigma_F(\sigma_G(r)) = \sigma_G(\sigma_F(r))$ met en évidence la commutativité de la composition de sélections. De telles propriétés sont à la base de l'*optimisation de questions*, c'est-à-dire la refor-

mulation d'une requête en une requête équivalente. Elles permettent de manipuler de manière symbolique des expressions d'algèbre relationnelle et de montrer l'équivalence entre certaines formules.

Chaque expression peut être représentée sous la forme d'un *arbre algébrique*. Chaque nœud de l'arbre représente un opérateur algébrique. Pour que l'expression soit évaluée, le système de gestion de bases de données doit remplacer chaque opérateur par l'algorithme réalisant l'opération. On obtient alors un *plan d'exécution* de la requête. C'est ce plan d'exécution que manipule le système de gestion de bases de données pour faire de l'optimisation de requêtes. Elle consiste à trouver le plan d'exécution le plus efficace pour une requête. La recherche du meilleur plan est un problème combinatoire pour lequel il est difficile de trouver une solution exacte. Le système de gestion de bases de données se contente donc en général d'un bon plan d'exécution. La recherche du plan d'exécution peut se faire à deux niveaux : au niveau algébrique, le système de gestion de bases de données va manipuler les expressions en utilisant les propriétés, au niveau physique, il va choisir les algorithmes les mieux adaptés. Pour choisir les expressions et les algorithmes à utiliser, il s'appuie sur les propriétés (*rule based optimization*) et sur un modèle de coût utilisant des statistiques sur les données (*cost based optimization*).

Prenons l'exemple de la requête suivante, permettant de trouver les noms des villes où il y a une équipe possédant un joueur de plus de 35 ans.

```
SELECT DISTINCT e.ville
FROM equipes e JOIN joueurs j ON e.nomCourt = j.equipe
WHERE j.age > 35
```

Cette requête est divisée en un certain nombre d'opérateurs de l'algèbre relationnelle. On voit qu'il est possible d'écrire plusieurs requêtes équivalentes en termes de résultat. En revanche, certaines pourront être plus rapides à exécuter que d'autres.

$$r_1 = \pi_{ville}(\sigma_{age>35}(equipes \bowtie_{nomCourt=equipe} joueurs))$$
$$r_2 = \pi_{ville}(equipes \bowtie_{nomCourt=equipe} \sigma_{age>35}(joueurs))$$

Le coût d'une jointure dépend de l'algorithme utilisé. Même s'il existe des algorithmes plus efficaces, pour illustrer notre propos simplement, nous considérons ici un algorithme naïf de jointure qui effectue une double boucle, c'est-à-dire parcourt tous les n-uplets de `equipes` et pour cha-

cun parcourt tous les n-uplets de `joueurs` et conserve ceux pour qui la condition de jointure est vraie. Un tel algorithme possède une complexité $O(|equipes| \times |joueurs|)$. Pour comprendre la différence de performance entre ces deux algorithmes, il faut faire des hypothèses sur le contenu des tables. Supposons qu'il y ait 20 équipes, 750 joueurs et 10 joueurs de plus de 35 ans. Dans ce cas r_1 fera $20 \times 750 = 15000$ comparaisons pour la jointure, puis parcourra les 750 lignes résultant de cette étape pour sélectionner les 10 villes des joueurs de plus de 35 ans, enfin la projection parcourra les 10 villes pour ne sélectionner que le résultat, soit un coût total $c_1 = 20 \times 750 + 750 + 10 = 15760$. En revanche r_2 commence par parcourir les 750 joueurs pour trouver les 10 de plus de 35 ans, puis pour ces 10 joueurs, parcourt les 20 villes, et enfin la projection parcourt les 10 n-uplets restant après la jointure pour un coût total $c_2 = 750 + 20 \times 10 + 10 = 960$.

On voit donc qu'il est possible d'optimiser des requêtes SQL, et qu'une connaissance des cardinalités – c'est-à-dire le nombre de n-uplets dans une table, ou répondant à une sélection simple – permet une optimisation avancée. Nous avons illustré une des optimisations les plus simples et qui fonctionne toujours : descendre les sélections en dessous des jointures.

SQL

Le langage *SEQUEL* a été développé au début des années 1970 par Donald D. Chamberlin et Raymond F. Boyce (IBM) pour le système de gestion de bases de données *System/R*. SEQUEL est rapidement devenu le langage SQL. La description de SQL donné dans ce chapitre est très succincte et se concentre sur les concepts fondamentaux. Pour une description détaillée de SQL, le lecteur pourra s'orienter vers l'abondante littérature sur le sujet ou vers le site `http://sqlpro.developpez.com/` qui en fait une très bonne présentation.

Valeur manquante en SQL

La gestion des valeurs manquantes en SQL est réalisée grâce au marqueur `NULL`. Signalons tout d'abord que `NULL` est bien un marqueur et non pas une valeur : il indique l'absence de valeur pour un attribut d'un n-uplet. La conséquence de ce fait est qu'il est inapproprié de comparer une valeur avec `NULL`. Le marqueur `NULL` n'est ni égal, ni différent d'une valeur. Les tests spécifiques `IS NULL` et `IS NOT NULL` servent à vérifier si une valeur est manquante. Un prédicat classique comportant des marqueurs `NULL` sera évalué soit à vrai, soit à faux, soit à *inconnu* en fonction des cas. C'est cette *logique à trois valeurs* qui peut rendre les requêtes complexes en présence

de NULL. Des logiques avec encore plus de valeurs existent, par exemple la logique à quatre valeurs, qui permet justement de préciser le rôle attendu de la valeur nulle : vrai, faux, inconnu et incohérent. Pour plus de détails sur cette question, on pourra se référer au site `http://sqlpro.developpez.com/cours/null/`.

Conception de bases de données relationnelles

Le modèle entité-association a été initialement proposé en 1976 par Peter Chen. Il a ensuite été étendu vers des modèles plus expressifs puis vers des modèles objet. Il existe différentes variantes de la notation, parfois comportant des différences subtiles. Il faut donc être particulièrement attentif et rigoureux par rapport au formalisme utilisé. Un point particulièrement délicat concerne la syntaxe et la sémantique des cardinalités des types d'association. Par exemple, l'emplacement des cardinalités d'une association binaire sur un diagramme entité-association selon le formalisme original est inversé par rapport à un modèle conceptuel de données de la méthode *Merise*. Pour ajouter à la confusion, un diagramme de classe UML utilise, à la place des cardinalités, la notion de multiplicité qui possède une syntaxe proche du formalisme entité-association original mais diffère sur la sémantique. Pour terminer sur les modèles conceptuels, signalons que la réalisation d'un modèle conceptuel représentant un problème donné est une tâche ardue. Cette tâche fait appel à des techniques d'analyse des besoins qui sortent du cadre du présent ouvrage.

Pour aller plus loin

Un bon complément à ce cours est le livre Georges Gardarin, *Bases de données - Objet et relationnel*, Eyrolles, 2003.

Il brosse un panorama très complet des concepts des bases de données avec un niveau de détail qui le rend très accessible. Du fait de l'expérience de son auteur, ce livre présente également les mécanismes internes des systèmes de gestion de bases de données. Enfin, à la fin de chaque chapitre, une liste de références commentées permet au lecteur intéressé d'obtenir plus de détails. D'autres livres abordant les thématiques des bases de données dans leur ensemble sont reconnus comme de très bons supports.

Laurent Audibert, *Bases de données - De la modélisation au SQL*, Ellipses, 2009.

Hector Garcia-Molina, Jeffrey David Ullman et Jennifer Widom, *Database Systems : The Complete Book*, Prentice-Hall, 2008.

Raghu Ramakrishnan et Johannes Gehrke, *Database Management Systems*, McGraw-Hill, 2003.

Avi Silberschatz, Henry F. Korth et S. Sudarshan, *Database System Concepts*, McGraw-Hill, 2010.

Christopher John Date, *Introduction aux bases de données*, Vuibert, 2004.

Ramez Elmasri et Shamkant Navathe, *Conception et architecture des bases de données*, Pearson Education, 2004.

Mokrane Bouzeghoub, Mireille Jouve et Philippe Pucheral, *Le modèle relationnel - Algèbre, langages, applications*, Hermès - Lavoisier, 1998.

Concernant la conception de bases de données, on peut citer

Jean-Luc Hainaut, *Bases de données : concepts, utilisation et développement*, Dunod, 2009.

Heikki Mannila and Kari-Jouko Räihä, *The Design of Relational Databases*, Addison-Wesley, 1994.

Ces deux références, bien qu'étant généralistes, font le choix de se focaliser sur la conception de bases de données.

Enfin, le lecteur intéressé par les aspects plus théoriques des bases de données consultera

Mark Levene and George Loizou, *A Guided Tour of Relational Databases and Beyond*, Springer, 1999.

Serge Abiteboul, Richard Hull et Victor Vianu, *Fondements des bases de données*, Vuibert, 2000.

Jeffrey David Ullman, *Principles of Database and Knowledge-Base Systems*, Computer Science Press, 1988.

David Maier, *The Theory of Relational Databases*, Computer Science Press, 1983.